算法艺术与信息学竞赛

刘汝佳　黄亮　著

清华大学出版社

如果说信息科学与计算机技术为我们开辟了一片新的天地，程序设计是这片天地的灵魂居住的花园，那么程序设计竞赛则是点缀这个花园，使她充满灵气的塔宇。

本书作者刘汝佳，连任三届信息学奥林匹克国家队教练，多次参与NOI和ACM的命题工作。作者黄亮现于宾夕法尼亚大学计算机与信息科学系攻读博士学位。两位作者选用近年来国际国内竞赛题目，以命题者的眼光，对算法与数据结构、数学知识和方法、计算几何几个方面的问题进行了全面、系统的分析与讲解。

无论您是在程序世界观望景仰，还是在IOI（国际信息学奥林匹克）赛场上忘我拼搏，进而在ACM（国际大学生程序设计竞赛）的世界里不息奋斗，或者您本身就是一个计算机专业的学生，本书中来自全球各地的竞赛题目和珍贵的原创试题，加上犀利精当的分析和点评，对您的编程能力的提高一定会有很大的帮助。本书中广阔的知识、充足的信息，新颖而前沿的学术认识以及内容丰富的网络资料等都会对您的综合能力的提高有所帮助。

愿您登上程序设计之塔的高层，登高望远，获得开阔的视野，俯瞰信息技术的新天地，对程序世界得到新的认知。

ISBN 7-302-07800-9
9 787302 078005 >
定价：45.00 元

算法艺术与信息学竞赛

刘汝佳　黄　亮　著

清华大学出版社

北　京

内 容 简 介

本书较为系统和全面地介绍了算法学最基本的知识。这些知识和技巧既是高等院校“算法与数据结构”课程的主要内容，也是国际青少年信息学奥林匹克（IOI）竞赛和 ACM/ICPC 国际大学生程序设计竞赛中所需要的。书中分析了相当数量的问题。

本书共 3 章。第 1 章介绍算法与数据结构；第 2 章介绍数学知识和方法；第 3 章介绍计算几何。全书内容丰富，分析透彻，启发性强，既适合读者自学，也适合于课堂讲授。

本书适用于各个层次的信息学爱好者、参赛选手、辅导老师和高等院校计算机专业的师生。本书既是信息学入门和提高的好帮手，也是一本内容丰富、新颖的资料集。

图书在版编目（CIP）数据

算法艺术与信息学竞赛 / 刘汝佳，黄亮著. —北京：清华大学出版社，2003

ISBN 7-302-07800-9

Ⅰ. 算… Ⅱ. ① 刘… ② 黄… Ⅲ. 算法-自学参考资料 Ⅳ. 0242.23

中国版本图书馆 CIP 数据核字（2003）第 116045 号

出 版 者：清华大学出版社　　**地　　址**：北京清华大学学研大厦

http://www.tup.com.cn　　**邮　　编**：100084

社 总 机：010-62770175　　**客户服务**：010-62776969

组稿编辑：欧振旭

文稿编辑：余　姬　陈韦凯

封面设计：钱　诚

版式设计：郑轶文

印 刷 者：清华大学印刷厂

装 订 者：三河市李旗庄少明装订厂

发 行 者：新华书店总店北京发行所

开　　本：185×260　**印张**：28　**字数**：618 千字

版　　次：2004 年 1 月第 1 版　2004 年 1 月第 1 次印刷

书　　号：ISBN 7-302-07800-9/TP・5685

印　　数：1～4000

定　　价：45.00 元

序　言

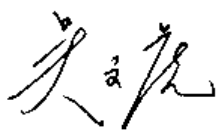
计算机解题的核心是算法设计。算法设计涉及许多先修的基础知识，包括数据结构、高级语言程序设计、离散数学、图论、组合数学、人工智能、计算几何等。当然还包括除数学与信息学之外的其他学科知识，因为没有这些知识，往往连题目都会看不懂，这可能也是要求参加 ACM 大赛的选手应该具备全面科学素养的原因之一。

刘汝佳、黄亮两位作者都曾在高中时参加过信息学奥林匹克竞赛活动，他们在如何用计算机解难题方面投入过很大精力，有着比较丰富的经验。在上大学之后，又投入了大量精力帮助训练中国队的小选手和参加 ACM 世界大学生程序设计大赛。他们深感这些活动对提高学生的能力和全面素质所起的巨大作用。因此，他们利用课余时间，广泛收集各地各类试题，并着力研究、分析与归类，想出比较好的解法，特别是总结出若干的思路和经验写成书和大家共享。从他们开始动笔到成书期间又花了大量的心血，对一些难题的解法也有了独到的见解。这本书应该说是很难得的经验之谈。对一个编程高手来说，借鉴别人的经验是十分重要的。因此，我想将这本书推荐给参加 IOI 的中学生和参加 ACM/ICPC 的大学生阅读。

本书的命名也有独到之处。就我本人的教学经历来看，算法的确是艺术。艺术与科学本来就是孪生姊妹，不科学的东西谈不上艺术。艺术给人以美的感受，而算法属于数学文化范畴，数学的美在算法中得到了充分的体现，特别是当今数学已经进入新的机器时代，利用计算机求解问题，需要人充分开动脑筋，解决一系列难点，解题过程本身就是一个精益求精追求完美的过程。在这样一个过程中，编程者在付出艰辛的努力之后，会有一种获得成功的愉悦。正如一些数学大师所言：数学是理性的艺术，是创造性的艺术。在编程解题的过程中，通过理性的思维和理性的实践，你一定会感受到算法艺术的无穷魅力。一个颇具匠心的好算法会让你拍案叫绝，感受到它的思维艺术之美。我们许多参加过 IOI 和 ACM/ICPC 程序设计大赛的选手，当问起他们当年的拼搏是否艰苦时，他们都说苦中有乐，苦中有甜。这可能就是感受到了这种思维艺术的魅力。

科学思维能力的提高是成就事业最重要的一个因素，希望本书对你思维能力的提高能有很大的帮助。

清华大学计算机系教授，博士生导师
信息学奥林匹克中国队总教练



2003 年 12 月

前　言

信息学（informatics）是一门新兴的学科，主要是指利用计算机及其程序设计来分析问题、解决问题的学问。随着计算机逐步深入生活，网络日趋普及，“信息革命”已经到来。信息学的地位逐步提高，青少年信息学竞赛也方兴未艾，在世界范围内受到了越来越多的重视。国际上，信息学竞赛主要分为两类：中学生的国际信息学奥林匹克和大学生的国际大学生程序设计竞赛。国际信息学奥林匹克（International Olympiad in Informatics, IOI）始于1989年，是联合国教科文组织倡导举行的五项国际学科奥林匹克之一。中国队每届都取得了名列前茅的佳绩，所有选手全部获奖，多次总分名列第一。国际大学生程序设计竞赛（ACM International Collegiate Programming Contest, ACM/ICPC）分为地区赛和国际总决赛，中国从1996年开始参加，现有3个地区赛赛点。清华大学、上海交通大学和中山大学等学校多次进入国际总决赛。上海交通大学队还获得了2002年的世界冠军的好成绩。从大局看，竞赛不是目的，而是推动普及的手段。虽然国内在培养信息学的优秀人才方面做了很大的努力，而且成绩斐然，但仍然存在一些遗憾之处：

第一，书籍资料的缺乏性。由于信息学竞赛的普及不如其他学科竞赛，辅导教师和研究人员相对匮乏，参与写作的人员也较少，致使国内此类书籍缺乏，而好书尤其缺乏。由此造成了不少信息学爱好者求书无门。另外，一些已出版的同类书籍或艰深难懂，或教条灌输，使不少爱好者望而却步，中途放弃，以至于信息学竞成了一种“曲高和寡”的学问。然而，社会信息化和信息学普及的大势已不可逆转，所以，信息学竞赛呼唤通俗易懂又有相当学术含量的好书。

第二，地区的不平衡性。由于信息学发展的时间毕竟不算长，很多地区在中学阶段刚刚开设或者还未开设信息学课程，师资力量相对比较薄弱。而且，信息学竞赛相对于其他学科，毕竟上手比较困难，这使得每年都有相当数目的信息学爱好者因为自学难度过大而中途放弃，从而导致国内信息学教学水平存在着很明显的地区不平衡性。很多地处信息学竞赛起步较晚地区的信息学爱好者们渴望能买到既实用又便于自学的书籍。

第三，国内视野的局限性。国内不少从事信息学竞赛研究的教育工作者把大部分精力放到了国内竞赛题目的研究中，而忽视了其他国家和地区的竞赛题目和研究成果。信息学竞赛题目的风格和内容往往受到本地传统文化等因素的影响，不同地区的题目往往有较大差异，因此熟悉各个国家的出题风格和特点，训练自己各方面的解题能力是很有必要的。所以，对于中高层选手和长期从事信息学竞赛辅导的老师来说，很需要紧跟国际动向、充分介绍国外成果的好书。

总之，从以上3点可以看出，国内信息学的普及度还远远不够。本书的出版，一方面是为了弥补国内信息学便于自学的普及读物方面的不足，拉近国内信息学竞赛起步较晚地

区与发达地区的差距。另一方面是为了向国内读者介绍国外最新研究成果和竞赛试题，填补这方面的空白，拉近国内和国际最新发展的距离。

本书在理论方面参考了国外的最新研究成果的论文报告，在实际运用方面大量选用了在国内研究较少的外国竞赛的优秀题目，对信息学竞赛理论研究和实践都具有一定的参考价值，同时本书也是一本难得的资料集。

本书分为 3 章，第 1 章介绍算法与数据结构。算法与数据结构是信息学中最重要的部分之一，内容多而杂，不容易从整体上把握。本章的前三节介绍复杂度分析基本理论、基础算法和基础数据结构，重在给读者一定的感性认识，然后分三个专题介绍三个重要的问题：数据结构的应用、动态规划和搜索。第 2 章介绍信息学中用到的数学。数学是信息学的基础，因此本书专门用一章的篇幅加以详细论述。本章容量大，理论性比第 1 章强，涉及到基本代数方法、初等数论、组合数学和图论等问题。第 3 章介绍计算几何的基础知识、基本算法以及技巧。3.1 节从最基本的位置和方向问题介绍叉积和点积；3.2 节过渡到多边形、多面体及其容积、重心的求取以及形内形外的判断；3.3 节讨论凸包这个最基本的几何模型及其应用；3.4 节介绍了几个常用的特殊算法，包括分治法、离散化和扫描法，还介绍了增量和随机增量算法。

本书包含的内容非常多，各个层次、各种需求的读者都能在本书中找到适合自己的内容。本书丰富的内容能给读者以很大的选择余地，不同难度的例题和习题中既有引导读者兴趣的入门题目，又有极富挑战性的精彩题目，习题编号前的 * 越多，表示作者越推荐。

本书的第 1 章和第 2 章由刘汝佳编写，第 3 章由黄亮编写，在成书过程中还得到了很多老师和选手的大力支持。

在第 3 章的写作过程中，上海交通大学 ACM 代表队的不少队员和教练给了作者许多帮助。

要特别感谢陆靖；感谢远在美国的陈晓敏，他与作者进行了多次富有启发意义的讨论并提供了不少国内罕见的资料；感谢吕侣、陶云峰、杨寅、严琦琦、林晨曦、龚理、田斌、张羲等同学对本书写作的支持和与作者进行的讨论。

感谢世界著名计算几何学家 Joseph O'Rourke 博士对作者的启发。

在前两章的写作过程中，部分 IOI2002 和 IOI2003 中国国家集训队的成员提出了不少宝贵的意见，也提供了一些资料作为帮助，他们是王知昆、张一飞、李睿、何林、毛子青、骆骥、齐鑫、赵爽、金恺、李益明、符文杰、刘才良、张宁、黄芸。

感谢俞玮和林希德，他们与作者一起进行了大量的试题翻译工作，并讨论和撰写了题目分析。

感谢在讨论中启发作者思维并教会作者不少知识的外国朋友们：瑞典的 Jimmy Mardell、加拿大的 Derek Kisman、波兰的 Tomasz Malesinski、乌克兰的 Alexandar Grushetsky、保加利亚的 Petko Minkov。

感谢中国著名人像摄影家魏德运先生在本书的出版过程中所给予的帮助。

感谢北京师范大学 ACM 代表队的吴莹莹同学为本书的出版所给予的大力支持和帮助。

特别感谢在本书写作过程中对作者大力支持的各位老师！他们是：IOI 中国队总教练吴文虎老师、ACM/ICPC 清华大学代表队总教练王帆老师和邬晓钧老师、ACM/ICPC 上海交通大学代表队总教练俞勇教授、ACM/ICPC 德黑兰赛区总裁判 Ramtin Khosravi 先生、ACM/ICPC 世界总决赛裁判 Shahriar Manzoor 先生和 ACM/ICPC 世界总决赛筹划指导委员会的 Miguel Revilla 先生。

作　者

2003 年 12 月 2 日

如何阅读本书

欢迎大家阅读本书！为了更好地发挥本书的作用，我们建议在阅读本书之前先熟悉本书的内容概况和特点，花一些时间来认真看看“如何阅读本书”部分。

本书的读者对象

本书的读者大致分为四类，我们分别给出不同的建议，供参考。

第一类，中学信息学竞赛选手

中学生的竞赛分为不同的层次。从纯普及的分区联赛，到普及兼选拔的 NOI 全国青少年信息学奥林匹克竞赛，纯选拔性质的 IOI 中国国家队选拔赛 CTSC 和纯竞赛性质的 IOI 国际青少年信息学竞赛，需要掌握的内容和难度都有很大的变化。分区联赛的选手应当学习一门程序设计语言并掌握到相当熟练的程度，然后学习本书中比较重要而易懂的内容；NOI 选手应当学会书中除了部分高难度的例题和带 * 内容之外的其他内容，而国家集训队员应当尽量掌握书中所有内容并完成大部分习题。极少数习题只是提供给有兴趣的读者进行研究，而不用强迫自己掌握。由于中学生竞赛对算法设计的要求很高，所以这一类读者应该特别注意体会书中例题并完成习题。

第二类，ACM/ICPC 大学生程序设计竞赛选手

大学生竞赛的特点和中学生不一样。ACM/ICPC 中编程速度和正确性显得尤其重要，而对算法设计的要求从整体上比中学生竞赛低了很多。针对比赛的这个特点，建议这一类读者重点阅读 1.3 节、2.1 节、2.2 节、2.3 节的全部内容以及 1.4 节、1.5 节、1.6 节、2.5 节中的理论部分和比较简单的例题，而其他内容只需要学习基础内容和 ACM/ICPC，UVA Problem Archive 和 Ural State Problem Archive 中的例题，其中难度过大的题目可略过。国内 ACM 竞赛正处在发展阶段，有一大批选手刚刚接触到竞赛，经验还不够。建议这些选手先浏览全书，然后先阅读理论部分，跳过一些思路巧妙或者难度很大的题目。ACM 竞赛中，速度和正确性是非常重要的，因此强烈建议大家把书中大部分的程序都亲自编写一下，并加入到自己的程序库中。

第三类，竞赛辅导教师和教练

教师和选手不一样，他们往往有比较充足的时间且不用花很多精力来训练自己的编程能力与保持状态，因此可以更从容地阅读本书。建议各位老师能通读全书，不断找出自己最感兴趣的部分深入阅读，用这种方式逐渐了解本书的所有内容。对于教师来说，了解全貌是非常重要的，因为这可以极大地帮助学生沿着正确的路子学习和训练。

第四类，计算机编程爱好者和其他读者

兴趣是最好的老师。这一类读者可以随意选择感兴趣的内容进行学习，如 1.2 节、1.3 节、2.5 节等节的题目都很有趣。另外，由于读者可以忽略大部分的证明和复杂度分析，而重点体会书中蕴涵的思想，这样的学习必将是有趣和吸引人的。

本书的目标和特点

学习信息学主要有两个层次的要求：理解和掌握。本书尽量让读者多理解一些难于理解的内容，若想熟练地掌握，读者必须另外下很大的功夫，配合阅读其他书籍并做大量的练习，进行一些实践。限于篇幅，很多理论和题目的细节本书都无暇顾及。这虽不大影响读者理解内容的精髓，但是当读者在细究一些问题（例如一些术语的准确定义、证明、伪代码和一些实现细节等）时会发现书中并未提到。其实，在多数情况下，这些内容可以在同类书中学到，也可在作者为本书专门设计的主页上找到丰富的学习资料。突出特色，启发思维才是本书的目标。

本书的特点有三个。了解了这些特点有助于读者在学习的过程中有意识地体会作者的“另一个用意”，从而最大限度地掌握内容。

第一个特点是启发性。这也是本书相对比国内已出版的同类书籍最大的特色。与传统此类书籍“填鸭式”灌输算法不同，本书力戒教科书式的死板讲解，力争引导读者进行创造性的思维，点拨关键思路，使读者在探索中自然领悟算法的精髓，并为其优美所吸引，激发浓厚的兴趣。传统书籍讲新算法时，往往作为一个全新的事物引入，而不是转化为已学的知识，也不讲与其他算法的联系，导致学生认为信息学千头万绪，琐碎复杂，而始终不得其精髓，更不用说“融会贯通”。其实算法都是相通的，万变不离其宗。故本书讲解时，大多先由一个有趣的实例引出，引导读者从几个已学的方向自行探索，逐步抛弃错误的方向，改为正确的方向，循循善诱，一步步把读者引向正确的算法，让读者觉得，新算法不过就是从老算法演化而来，并不深奥难懂。因此，建议读者在阅读时特别注意各个知识点是怎样被引入的。一个知识点引用得越曲折，说明这个知识点越是重要，越有可学之处。总之，本书能启发读者进行创造性思维，自然体会算法和知识的正确性、优美性、深刻性、广泛性和统一性，不仅能化为己用，还能有所创新。

第二个特点是信息量大。本书在描述算法时尽可能简洁，突出主要矛盾。书中涉及的源程序在本书网站上都有，而书上的篇幅尽可能多的讨论算法的精髓、与其他算法的异同、比较时空复杂度优劣、适用条件、互相转化等，大部分内容都是同类书籍中很难找到的。读者也许会抱怨某些知识讲得不够深，不够细，但实际上这些有限的篇幅被用来讲解更为有用的知识了。我们用网站来保证读者可以有条件学习到这些知识，而腾出更多的篇幅来介绍本书特有的内容。信息学竞赛是一种讲究创新的比赛，本书在讲解知识之余十分注重鼓励读者思维创新，在介绍大量知识和技巧的同时给读者留下广阔的思维空间。书中某些问题只是轻描淡写地提了一句，但是为读者提供了很大的思考空间，读者可以独立研究，

也可以互相讨论。本书主页上的内容是本书一个很好的补充。只要读者肯钻研，能从本书中获取的信息量是很大的。

第三个特点是内容新。本书大量选用了同类书籍中很少涉及的其他国家的国内比赛和一些区域性比赛中的优秀题目，其中有很多题目是从波兰语，罗马尼亚语等翻译过来的。还有大量例题和习题来自近年来蓬勃发展的网上在线试题库（如西班牙巴拉多里德大学（Universidad de Valladolid）试题库和俄罗斯乌拉尔州立大学（Ural State University）试题库），以及各种在线程序设计比赛的题目。另外，在理论方面参考了国外的最新研究成果的论文报告。这些内容对于国内的绝大多数读者来说，题目耳目一新，本身就反映了国际信息学发展的动态，其解法又代表了世界信息学发展的最新成果。这对于“久经沙场”的老选手和长期从事信息学竞赛的辅导老师来说，也是紧跟国际最新发展的趋势。

使用本书的建议

信息学是一门实践性很强的学科，读者在学习本书时应该多种方法相结合。下面一些建议，这对于大多数希望掌握本书大部分内容的读者来说是适用的。

第一，重视基础和语言的实现能力。本书一般没有完整的源程序，只有比较重要的理论，有 PASCAL/C++或者其伪代码，而且格式也不统一。请大家自己进行编码练习，实现书中介绍的算法，如参考数据结构和算法书籍，本书的网站上也可以找到一些题目的源程序。书里对编码比较困难的部分做了专门提示，请读者留意这些内容。

第二，善用其他参考书结合学习。建议把本书和其他书籍结合起来阅读。例如，数据结构部分可以先阅读本书的 1.3 节，有一个感性认识，然后认真学习相关的教材，搞清楚很多细节，最后再研究一下 1.3 节，这时将会体会到一些新内容。永远记住，本书的性质是导引，目的是让读者明白该学些什么，怎样学，很多具体问题还需要专门学习。如果读者没有这些书籍也不要紧，在本书的网站上可以找到这些熟悉的资料。

第三，感性和理性相结合。本书的内容虽然科学性很强，但这并不意味着读者应该用完全理性的方式来学习。在本书的很多地方，作者都用了相当“感性”的语言。这样做虽然损失了一定的严密性，但是可以让读者了解到其中的思路和动机等内涵很丰富的内容。读者不要轻视这些内容，仔细琢磨一定会受益匪浅。

第四，注意细节。也许一些读者对本书的大部分内容是很熟悉的，但作者想提醒的是，即使某些内容大家已经非常熟悉了，但还是建议仔细阅读本书。不管是为了明确概念，澄清误区还是开阔眼界，很多细节都是值得注意的。

第五，和 Internet 积极配合。在 Internet 飞速发展的今天，本书将采用借助网页来达到更好的学习效果。本书主页的 URL 是：http://oibh.ioiforum.org/book-lrjhl/。如果以后本书的主页转移位置，则可以通过用本书书名作为关键字在 Internet 上检索。在本书的主页中，你可以看到以下内容：

- 进一步阅读的资料；

- 本书中部分题目的测试数据和源程序；
- 本书的勘误表；
- 读者提问和解答；
- 作者和读者写的关于本书的文章；
- 一个专门为读者设立的论坛，供读者交流阅读心得。

目　录

第1章　算法与数据结构

算法与数据结构是信息学竞赛最核心的部分，也是选手必备的基础知识。“数据结构+算法=程序设计”。这些知识不仅很重要，而且是学习其他内容的基础。

本章内容简介

本章从理论分析和实际应用两方面阐述了算法与数据结构的基本知识。由于同类书籍对相关理论作了较为详尽的叙述，本书把重点放在了实例分析上，因为它们可以帮助读者更为深刻地理解算法与数据结构的精髓。

1.1 节概括地叙述了算法、数据结构以及计算理论的一些简单概念，目的是让读者从宏观上对第 1 章的基础做一定的了解。

1.2 节从实例出发，概括地介绍了一些基本算法，包括枚举、贪心、递归、递推法等。本节的重点是让读者领会到常用的算法思想，因此所选的部分例题有相当的难度，读者需反复体会才能明白其中的道理。本节还有一些小专题，供读者选学。

1.3 节介绍基本数据结构，包括线性表、队列、栈、树、二叉树以及图的遍历与拓扑排序。和 1.2 节不同的是，本节的系统性比较强，为了突出趣味性和实用性，本节选用了一些有趣的故事引入这些内容，从而增加读者的感性认识。建议读者阅读专门的数据结构教材来获得系统的理论知识，打好基础。

1.4 节介绍一些实用数据结构，包括哈希表、二叉搜索树、Trie 结构、线段树、堆、并查集等。本节的难度较大，而且实用性比较强，1.5 节和 1.6 节将广泛使用这里介绍的知识。

1.5 节介绍动态规划方法和一些经典动态规划问题，并结合一些例子重点讲述常见的建模技巧。

1.6 节介绍搜索算法。本节内容包含盲目搜索、启发式搜索、博弈算法和搜索的优化问题。本节的部分理论知识和例题比较难懂，建议读者多写程序来实现题解中提到的一些细节。

在大多数小节的后面，作者从近年来国内外竞赛中出现的基本算法与数据结构的题中，精选出一些问题以检验读者的学习效果。部分题目具有相当的难度，读者可以反复思考。书后附有部分练习的提示。

1.1　编程的灵魂——数据结构+算法=程序

本节介绍数据结构、算法的基本概念和复杂度分析的基本方法。本节的理论性比较强

且部分内容和一些有一定基础的读者的“经验”有些违背。这些内容比较抽象，部分内容（如并行计算机、PS 完全问题等）主要只是开阔读者的眼界。本节的重点是渐进时间复杂度的计算和化简，而难点是各个相关概念的正确理解和对计算机解题能力的整体认识。

本节不难学，建议读者同时另外阅读一本介绍计算复杂性的入门书籍，和本书结合起来看。阅读本书之前，读者需要熟练掌握一种程序设计语言，推荐使用 C/C++或者 JAVA，但同时需要了解一下 Pascal，如果不熟悉这些语言，阅读本书的效果则将会大打折扣。

首先需要弄清楚：什么是算法？什么是数据结构？为什么要学它们？简单地说，**算法（algorithm）**就是解决问题的方法或者过程。如果把问题看成是函数，那么算法就能把输入转化成输出；**数据结构（data structure）**是数据的计算机表示和相应的一组操作。这二者是最基本的，但对于初学者来说，最熟悉的名词并不是它们，而是**程序（program）**算法和数据结构的具体实现。这三者的关系在本节的标题中已经指出了：“数据结构+算法=程序”。

本章在介绍算法时强调：

1. 算法思想：深刻地理解这些算法设计思想将有助于开阔读者的思维。

2. 算法分析：从时空性能、适用范围等对介绍的算法进行分析，增强读者对这些内容的理性认识，学会定性、定量地分析算法和进行比较。

需要说明的是，我们强调的是算法设计和分析，而不是对问题本身的分析。对问题本身性质的分析只是粗略提一下，作为算法设计的指导，而不作为重点学习和研究的内容。不过，一些重要的结果和方法，如 NP 完全理论和判定树模型，由于它们容易理解且用处很大，我们仍会花一些篇幅进行讲述。

时空开销增长　要比较两个算法的各方面性能有很多方法。其中之一是各写一个程序，比较它们的运行情况。但是由于程序并不是算法的直接对应结果，无法避免一些和算法无关，只由代码产生的问题。而且需要写程序，这将花费比较大的精力，尤其是在算法复杂，实现容易出错的时候。另一个办法是在算法设计出来以后直接针对算法进行分析，估计它的时间效率和空间开销。由于运行时间等因素和计算机运行速度有关，所以我们只关心在问题规模扩大时时空开销的增长情况。再次强调，我们很少进行程序分析，一般只进行算法分析。由算法描述写出高效程序属于程序设计方法的范畴，这不是我们讨论的重点。

基本操作　为了进行这样的估计，首先给这个估计问题建立合适的模型。用变量 n 表示输入问题的规模，而定义“基本操作”为一个运行时间不依赖于操作数的操作。

考虑两个正整数相加的操作 add。如果是两个不大于 100 的正整数相加，那么运行时间总是常数；如果两个数的大小没有限制，那么操作依赖于两个数的位数。因此可以把情况一中的 add 看作基本操作，情况二中的 add 就不能看成基本操作（这时我们一般把每一位相加看成一个基本操作，它的运行时间是常数）。后面将看到，在不同的时候，把不同的操作看成基本操作。基本操作的选择反映了对问题的主要矛盾的认识。

可以用程序执行的基本操作数来衡量它的时间效率。例如下面程序一[1]：

---

[1] 这是一段 C 程序。作者假定本书的读者熟悉 C 语言程序设计

```
fact = 1;
for(i = 1; i <= n; i++)
  fact *= i;
```

如果把一次加法操作看成一个乘法操作，那么程序共执行了 n 个基本操作。注意，这里忽略了循环时改变变量 i 所花费的时间，而只关心和算法最有关系的基本操作。再看程序二：

```
sum = 0;
for(i = 1; i <= n; i++)
for(j = 1; j <= n; j++)
    sum += a[i][j];
```

它执行了 n^2 个基本操作。

两个程序哪个快呢？不知道。因为不知道加法操作和乘法操作哪个快。假设加法速度是乘法的 10 倍，那么哪个程序快呢？您一定会说：不一定。当 $n\leqslant 10$ 是程序二快，而 $n>10$ 时才是程序一快。这样说有道理。但是算法分析的结论会是：程序一的渐进时间复杂度低，因此更优。理由是：当规模扩大 10 倍时程序一的运行时间只扩大 10 倍，而程序二会扩大 100 倍。在这里，忽略了常数因子，因为我们只对增长情况感兴趣。下面不加说明地给出一些定义，请读者阅读相关书籍来了解这方面的详细内容。

复杂度分析的常用符号　若存在两个正常数 c 和 n_0，对任意 $n>n_0$ 都有$|T(n)|\leqslant c|f(n)|$，则称 T(n)在集合 $O(f(n))$中。记作 $T(n)=O(f(n))$。O 读作“大欧”。它的直观含义是：T(n)的增长速度不会比 $f(n)$差。**大 O 表示法（big-Oh notation）**表示的是运行时间的上限。同理可以定义下限 Ω。如果上下限相等，还可以用 Θ 表示。本书主要只采用大 O 表示法。由于进行了简化，把大 O 表示法衡量的运行时间称为“**渐进时间复杂度（asymptotic time complexity）**”。事实上，还有**小 o 表示法**，表示这个上限是“松”的[①]。空间也可以用它来表示，但是由于在实际问题中，空间需求的一点点差异就可能引起由“存得下”到“存不下”的质的变化，所以空间复杂度常常需要用非渐进形式的精确表示。在这样的情况下，读者应当对一些程序设计细节进行了解，如各种数据类型的空间占用，动态结构的指针附加空间等，这些不是本书的讨论重点，故不展开讨论。

简化法则　利用这个定义可以把运行时间简化。简化的方法是忽略常数和低次项，例如 $3n^4+8n^2+n+2=O(n^4)$。这样做简化以后虽然忽略了一些因素，但是更容易看出算法的时间效率增长情况。如果读者怀疑这种简化法则的合理性，请参考任何一本算法与数据结构的书籍。

复杂度的等级　算法的渐进复杂度有几个等级的。一是多项式算法，例如 $O(n^t)$，t 是常数，它的运行时间随着规模的扩大增长不大，因此叫做**有效算法**；二是指数级算法，如

[①] 有兴趣的读者可以在任何一本微积分课本上学到小 o 表示法

$O(2^n)$等，它的运行时间增长很快，不是有效算法。还有更大的（如 $O(n!)$），只能在 n 很小时才能使用。对于指数级别的算法，一个有趣的结论可以在本书网页上找到。有趣的是，通过时间复杂度的形式，有时候能够猜测出算法的大概思路。例如包含 Ackermann 函数反函数的算法往往代表着算法使用了并查集或者利用了某函数的凹凸性，含 $\log n$ 的往往是用了递归、二分或者各操作时间复杂度含 $\log n$ 的数据结构。而 $n^{1/2}$ 或者更“奇怪”的常数 a 作为指数的 $O(n^a)$类算法的往往是多级算法（如二级检索），分阶段算法（如二分图匹配的 Hopcroft 算法），或者待定参数的算法。a 是最后解出的最优参数。这些算法设计思想应该被本书的读者所重视。

伪多项式　需要注意的是，哪些参数作为“输入规模”并没有统一的标准。特别是在图论问题中，希望时间复杂度不和某一些参数（例如边权上限）发生关系，因此即使时间复杂度是这些参数的多项式函数，也只说这些算法是伪多项式的。需要特别注意的是后面将要提到的 NP 完全理论。在证明一个问题是 NP 完全问题时考虑的“输入规模”。例如在图论中一般考虑的是结点数 n 和边数 m。但是在另一些输入如边权绝对值的最大值、最大结点度或者**树宽（tree-width）**限制在一个比较窄的范围时，有理由把它们也看作输入规模的一部分，从而得到多项式算法。一个典型的例子是**子集和数问题(subset-sum problem)**，当数的范围比较小时可以用动态规划（参见本书 1.5 节）得到一个满意的算法。请读者注意后面提到的 NP 完全理论或者其他关于问题的时间复杂度下界的证明中所默认的输入规模参数集。当输入参数集规模改变时，情况可能发生很大改变。

不同情况下的运行时间　需要说明的是，有的算法在输入规模不变的情况下会根据输入 0 不同而具有不同的运行时间。这时候需要分别分析最好情况、最差情况和平均情况的复杂度，甚至更复杂的概率分析和均摊分析。在后面的章节中，将讨论一些这样的问题。

复杂度和实际运行时间　算法复杂度分析对算法的选择有着指导性作用。假设有台计算机在可以忍受的时间里能进行 10 000 次基本操作。如果计算机的运行速度扩大 10 倍，那么对于不同算法，在这个时间之内可以解决的问题规模是如何变化的呢？如表 1-1 所示。

表 1-1　七种不同的时间复杂度的时间增长速度

$f(n)$	n	n'	变　化
$10n$	1 000	10 000	$n'=10n$
$20n$	500	5 000	$n'=10n$
$5n\log n$	250	1 842	$n'=7.37n$
$2n^2$	70	223	$n'=3.16n$
n^3	21	46	$n'=2.15n$
2^n	13	16	$n'=n+3$
$n!$	7	8	$n'=n+1$

如果不幸选择了最后一种算法，那么计算机速度扩大 10 倍的效果微乎其微。

需要注意的是，渐进时间复杂度低的算法实际运行效果并不一定好，甚至有些实际问题的某些指数级别的算法实际效果是很好的，例如 $O(1.211^n)$的算法当 $n\leqslant 50$ 时都不错（不

要觉得这种算法是不存在的，有兴趣的读者可以翻翻图论方面的论文，尤其是与着色和顶点覆盖有关的算法）。还有些方法的最坏情况下是指数级别的，但是实际情况是几乎不可能遇到的这种最坏情况的，效果比一些多项式算法还要好，其中比较典型的是线性规划问题的单纯性法（它比椭球算法和 Karmarkar 算法效果都好）和最小费用流的松弛算法。所以算法分析不是万能的。我们常常给出一个很坏的分析结果但是实际运行效果会不错。希望读者在分析自己的算法时，在给出时间复杂度下界的同时,充分考虑这个下界达到的可能性。后面提到的概率分析和均摊分析也许会帮你的大忙。

时空辩证关系　在很多情况下，一个好的算法可以同时在空间和时间上达到最优，但在更多情况下二者是矛盾的，我们需要协调它们的关系。用时间换空间的常用方法是重复计算,用空间换时间的常用方法是预处理。在后面的章节中，会多次使用该技巧。

计算模型　本书中讨论的计算模型适合于最常见的微型计算机，它的特点是同一时间只能执行一条指令，并且执行的指令是确定的。事实上，计算理论中会研究很多不同的计算模型，例如可以同时执行多条语句的并行计算机，执行指令不确定的非确定机等。本书不深入讨论计算模型，但是希望读者能明确本书的基本出发点。

随机存取（RAM）模型　未加说明的情况下，本书给出的算法均建立在此模型基础上。它的特点是可以随机访问存储器，而不像磁带机一样只能顺序存取（建议：试着把你学过的算法改成针对磁带机的，看看结果会是什么样子？）。

并行计算机　考虑一台计算机，它同时可以做很多件事情，那么问题解决起来是否容易一些呢？考虑判断两个 L 位二进制整数是否相等的问题。一般的计算机显然要花 L 次比较；随机算法要保证 1/2 的概率正确需要 $L/2$ 次比较；但是并行计算机——如果有 L 个处理器的话，就只需要一个单位时间就可以把 L 个数位全部比较完毕。分析并行算法，我们不仅需要分析时空复杂度，还需要分析它需要多少个处理器。并行计算机是存在的，所以并行计算理论有很强的实用性。但是由于目前个人电脑还很少使用它，本书不对它进行讨论。

非确定机　考虑另外一种“计算机”，它可以“猜”。它的执行方式近乎神奇：它总是按照最好的情况“猜”。例如用它解决 n 皇后问题，只需要让它连续猜 n 次，就猜出了每个皇后的位置。非确定机的特点是：只要有一种猜法能满足要求，它一定能猜到。现在使用的计算机为确定机。目前这还只是一种虚构的机器，没有发现真实的机器可以达到这个要求。

P 类问题和 NP 问题　如果一个问题可以在多项式时间内被确定机解决，则它属于 P 类问题；如果它可以在多项式时间内被非确定机解决，则它属于 NP 类问题。显然 P 是 NP 的子集。那么是否存在在 NP 中却不在 P 中的问题呢？目前还没有人知道——它是计算理论最著名的难题之一，虽然无数迹象表明 P 很可能不等于 NP。

NP 完全（NPC）问题　这类问题目前都没有找到多项式算法，即很可能属于 NP－P。它满足①属于 NP。②NP 中的每个问题都可以使用多项式归约到它。NPC 类问题最显著的特点是：如果其中一个找到多项式算法，那么使用一种问题归约技巧可以让其他所有问题都得到有效解决。

NPC-难度问题　NPC 问题并不是最困难的。甚至有些问题，它虽然满足 NPC 问题的

定义②，却不满足定义①，非确定机并不是万能的。考虑下面的问题：输出 n 个数的所有全排列。由于排列有 $n!$个，不管它怎么猜，输出也至少需要 $O(n!)$的时间，不可能存在多项式算法，即使在非确定机上。

不可解问题　有不可解问题吗？你可能会认为没有。因为所有题目都写搜索程序，运行很长的时间，总是会运行出来的。可惜不可解问题是存在的。考虑这样一个问题：给一个 C 语言程序和一组输入，这个程序最后会终止吗？当然，对于一个给定的程序，或许能判断出来，但是无法设计一个通用的算法，对于任意一个程序和任意一组输入，都能得到正确的结果。这个结论或许很令你吃惊，不过事实上，有一个很简短的证明，放在本书的主页上，有兴趣的读者可以阅读。

***PS 问题和 NPS 问题、Savitch 定理**　和时间一样，令 PS 为所有可以在多项式空间内解决的问题集合。显然 NPC 为 PS 的子集（想一想，为什么？）。P 是否等于 NP 是世界难题，不过 PS 是否等于 NPS 却是很简单的，因为有 Savitch 定理：PS=NPS[①]。

***PS 完全问题**　和 NP 完全问题类似，有 PS 完全问题。一个问题是 PS 完全的，当且仅当：

（1）它在 PS 中。

（2）所有 PS 问题都可以多项式时间归约到它。

PS 完全问题的一个例子是 QBF（Quantified Boolean Formulas）。

当证明了一个算法是 NP 完全问题后，就不应该再去碰它吗？当然不是！在这里要提醒读者，不要把所有的 NP 完全问题打入“冷宫”而不去继续研究。NP 完全问题只是极可能无法设计一个在确定机上总是运行多项式时间，总是能得到正确结果的精确算法。前面说过了，并不一定多项式时间才快，并不需要它总是保持多项式时间，并不需要让它总是得到正确结果，也不需要让它的结果总是精确。一句话，对于 NP 完全问题，仍需要设计实用的算法。

随机算法　如果在算法中使用了随机数，那么称它为随机算法。注意随机算法和非确定机是本质不同的。随机算法是在不确定的时候任选一个，而非确定机是在不确定的时候猜到一个导致最好结果的。

Monte-Carlo 算法、RP 类问题　如果一个问题存在一个随机算法，使得它有 50%以上的概率得到期望的结果[②]，那么这个问题属于 RP 类，该算法称为 Monte-Carlo 算法。如果一个问题是 RP 类问题，可以通过多次运行它的一个 Monte-Carlo 算法而得到“几乎每次都是正确”的算法。一个典型的例子是质数判定问题：它在 RP 中。将在第 2 章中给出它的一个 Monte-Carlo 算法。

***Las Vegas 算法、ZPP 类问题**　如果一个问题存在一个随机算法，使得一定能得到期望的结果，而运行时间的数学期望[③]是多项式的（虽然最坏情况下的运行时间高于任何多项式），那么这个问题属于 ZPP 类，该算法称为 Las Vegas 算法。

[①] 更精确地，有：一个用 p(n)空间的 NTM 可以变换到一个用 $p^2(n)$空间的 DTM

[②] 严格地说，对于该问题的语言来说，该语言中的串至少有 50%的概率被接受，而其他串一定不被接受

[③] 即各种运行时间按概率加权得到的平均数。数学期望是概率论最重要的概念之一

这样，有了对付 NP 完全问题的第一类武器：随机算法。不管是找到了 Monte-Carlo 算法还是 Las Vegas 算法，都可以认为找到了非常实用的算法——从实用的角度讲并不比多项式算法差，而且除了 NP 完全问题，把随机算法应用到 P 问题中，也往往能得到时空复杂度更低的算法。

近似算法　在实际问题中，往往只需要得到不错的解，并不要求是最好的。比如我们花了一分钟时间得到了一个利润为 100 万元的方案，而需要花费一年才能得到一个利润为 100 万元零 1 分的方案，那么有必要非要得到最优解吗？不是的！前面提到了时空的辩证关系，这里要把解的质量也加进来。同样应该考虑时空复杂度和解的质量的关系。对于一些 NP 完全问题，甚至可以得到**完全近似算法（full approximating algorithm）**。

完全近似算法对于一个输入 ε，得到一个 ε-近似解（即得到的结果不超过最优解的 $1+\varepsilon$ 倍），而且运行时间当 ε 一定时是多项式的。一个典型的例子是**子集和数问题（subset-sum problem）**，它有一个完全近似算法。进一步的讨论参见本书的主页。

这样，有了对付 NP 完全问题的第二类武器：近似算法。尤其是完全近似算法，可以得到任意精度的结果。这是时间-解质量平衡的极好例子。

虚拟机器　可以用计算机来模拟并行计算机和非确定机。这种方法就是设计一个“黑匣子”，给用户提供并行计算机和非确定机的界面。当然，这样做需要花费大量的时间（并行计算机一条指令的模拟需要 $O(p)$时间，其中 p 为处理器数量；非确定机一条指令的模拟可能需要指数复杂度的时间），但是这为我们验证算法提供了可能。在程序设计竞赛中，可以用“交互式”题目的方式来实现这两类机器，考查选手更广泛的算法设计知识。

练　习　题

1.1.1　如果 $T(n)=O(n)$，则 $T(n)=O(n^2)$对吗？如果 $T(n)=O(n^2)$，则 $T(n)=O(n)$对吗？

1.1.2　如何从实验上证实一个程序的时间复杂度确为 $O(n^2)$？设计不同的方法，注意细节。

1.1.3　做实验看看在基本操作很简单（如加法）的情况下，时间复杂度为 $O(n)$, $O(n\log n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$, $O(n!)$各能支持到多大的规模？

1.1.4　举出一个问题，它没有一个具有多项式时间复杂的算法，并给出证明。

***1.1.5**　是否存在一个可解问题，不存在一个只用多项式级别的空间解决该问题的确定算法？

***1.1.6**　如果算法带有随机因素，如何重新叙述算法的空间复杂度定义？

本节的内容看起来很简单，其实不然，它是一个非常深的理论。希望对“问题求解”这一问题有进一步了解的读者，可以深入学习自动机、形式语言和计算理论，推荐 John E.Hopcroft,Rajeev Motwani, Jeffrey D.Ullman 的《Introduction to Automata Theory, Languages, and Computation》second Edition，第 8 章~第 11 章。相信它会很大地开阔读者的视野。学

习了这些理论以后，您会发现自己的理论水平有了很大提高。

1.2 基本算法

本节将介绍一些程序设计中最常用的算法，并通过例题来加深读者的理解。这些算法有：枚举、贪心、递归、分治、递推。本节的理论性不强，题目也颇有趣味，大家可以仔细阅读。但是本节中有很多例子具有相当的难度，在第一次阅读时读者不必看懂所有内容。

1.2.1 枚举

如果手工都很容易算出来的东西，有理由相信写成程序以后也能很快得到结果。先来看一个简单的例子。

【例题 1】盒子里的气球[①]

在一个长方体盒子里，有 $N(N\leq 6)$个点。在其中任何一个点上放一个很小的气球，那么这个气球会一直膨胀，直到接触到其他气球或者盒子的边界。按照怎样的顺序在这 N 个点上放置气球，才使放置完毕后所有气球占据的总体积最大？

如图 1-1 所示，图(a)描述了一个长方体的横截面和三个可以放气球的位置。图(b)和图(c)分别描述了一种放气球的顺序。

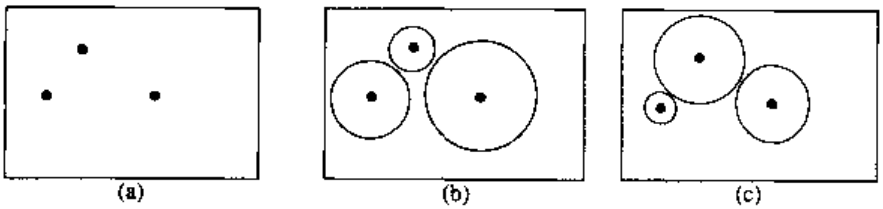


图 1-1　盒子截面与三个气球放置点

【分析】

如果没有计算机，你可以算出答案吗？可以的。这道题目最多只有 6 个气球，因此最多只有 6!=720 种放气球的顺序。对于每种顺序，手算出最后的剩余体积，比较后就知道哪种方案最优。虽然很花时间，但耐心点总是可以算出来的。在没有找到更好的方法之前就用这个方法写个程序，让计算机代替手算，用更快的时间找到解。

用(X_i,Y_i,Z_i)表示气球 i 的球心坐标，用 R_i 表示它的半径（未膨胀的气球认为半径为 0）。对于当前将要膨胀的气球 i，用公式：

$$D_{i,j}=\sqrt{(X_i-X_j)^2+(Y_i-Y_j)^2+(Z_i-Z_j)^2}-Rj$$

计算出它接触气球 j 时半径的大小；同时还可以计算出气球球心到盒子每个面的距离，

① 题目来源：ACM/ICPC World Finals 2002. Ballons in a Box

取所有这些距离的最小值，就是气球的膨胀。知道每只气球的半径，还知道盒子总体积，因此可以很快求出该方案的剩余体积。

枚举算法　在很多时候，无法立刻得出某个问题的可行解或者最优解，但是可以用一种比较“笨”的方法通过列举所有情况然后逐一判断来得到结果，这就是枚举算法的核心思想。枚举算法的特点是比较单纯，往往容易写出程序，也容易证明算法的正确性和分析算法的时间复杂度，可以解决一些规模很小的问题。它的缺点是速度慢，当枚举量很大的时候运行速度无法忍受。

前面已经提到，枚举的思想很简单：一个不漏的考察所有的情况。但是，有时候稍微不注意就会有遗漏，或者枚举了不必要的情况，浪费时间。下面这个有趣的例子可以说明这一点。

【例题 2】图书馆[①]

鲁滨逊决定做一个书架，建立自己的藏书室。他在石头墙上凿了一个矩形壁龛，把一些栓子钉入墙体，然后找来一些木板架在两个水平的栓子上做成书架，任意两块木板不处在同一水平线上，如图 1-2 所示。不巧的是，鲁滨逊忽然发现有一本珍贵的旧书特别大，无法放进他现在架好的书架里。他仔细量了这本大部头的高和厚，想改造一下他的书架，以便于把这本书放进去（别忘了，书架是做在墙里面的，不能超出范围）。

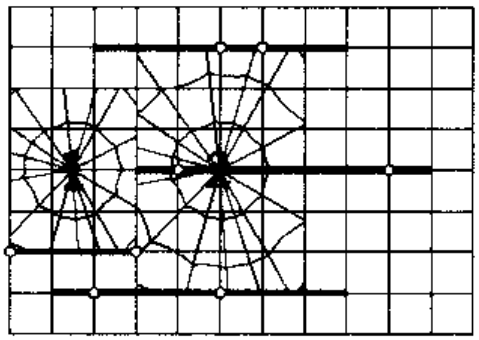


图 1-2　石头墙上的书架

下面是一些改造的操作方法：

（1）把架子留在原地不动；

（2）把木板向左移或者向右移；

（3）把木板锯掉一段后向左或向右移；

（4）把栓子顶到与原来位置处于同一水平线的另一个位置，并把木板左移或者右移；

（5）把木板锯掉一段，移动某一栓子到同一水平线上另一个位置，并把木板左移或者右移；

（6）把木板和两个栓子一起拿掉。

当木板被两个栓子架住，并且木板的中心在两个栓子之间或恰在一个栓子上方时，这个书架是稳定的。鲁滨逊开始设计的藏书室里所有的书架都是稳定的。木板的长度是整数，

[①] 题目来源：ACM/ICPC Regional Contest Northeast Europe 2001. Library

单位是英寸。因为测量工具不够精确，他只能做到以英寸为单位切割木板。在你的改造中，所有的书架必须始终是稳定的。

要找到一个方案来改造鲁滨逊的藏书室，以便于那本古老的大书放进去，而且要使改动尽量的少。要尽量减少移动的栓子数目（操作 4、5 每次移动一个栓子，操作 6 每次移动两个），在这个前提下，找到浪费木板长度最小的方案（操作 3、5 各切割掉一定长度，操作 6 把整个木板的长度都浪费了）。木板的厚度和栓子直径忽略不计。那本大书只能竖直放置，它的全部厚度都要位于木板上，而且只可以碰到其他木板或栓子的边缘。

【分析】

题目中有一个条件只是轻描淡写地提起，但却很重要，那就是任意两块木板不处在同一水平线上。这样，某块木板移动后不会存在架在其他木板或木栓上的情况，因此我们可以孤立地考虑每块木板的移动与锯下，使问题复杂度大大降低。于是，我们很快制定了解题方案：逐个枚举安放大书的位置然后选取其中最优的。

工作分两步：①确立安放大书的位置；②用最优代价为大书腾出空间。由于评价函数要求在移动最少木栓的前提下浪费最短的木板，而两个步骤又是独立的，所以无论步骤①还是②，都：

最先尝试“不移动木栓也不锯木板”的方案，如果失败，

接着尝试“不移动木栓但锯下最少木板”的方案，如果失败，

其次尝试“移动一个木栓但不锯下木板”的方案，如果失败，

然后尝试“移动一个木栓同时锯下最少木板”的方案，如果失败，

最后选择“撤去整个木板”的方案。

1. 确定安放大书的位置

由于大书必然安放在某块木板 i 上，所以只要大书放上去以后不会“顶着天花板”，就应当针对木板 i 枚举大书的安放位置。这一步暂时不考虑其他书架，只需要大书可以稳定放在木板 i 上就可以了。

注意到锯下木板对大书的稳定位置并没有好处（书架本来就是稳定的，如果书的位置太偏，那么锯木板同样无济于事），而且支撑木板 i 是不能撤去的，因此只需要检查“不移动木栓也不锯木板”和“移动一个木栓但不锯下木板”两种方案。

2. 用最优代价为大书腾出空间

现在，放置大书的木板已经准备好了，但是却不一定能把书放上去，因为刚才没有考虑其他书架，而实际上它们是有可能把大书档住的。好在由于每块木栓的移动和木板的锯下可以孤立的考虑，可以分别用最优代价消除每块挡住大书的木板，再累加起来。对于没有档住大书的木板 S，先尝试“不移动木栓但锯下最少木板”（注意：锯后的书架必须保持平衡），然后考虑“移动一个木栓同时锯下最少木板”（这包含了“移动一个木栓但不锯下木板”的情况）。如果还是不行，就只好采用“撤去整个木板”的办法，这是最后的无奈选择。

其实这道题目没有什么难点，关键是认真分析每一个条件，并逐一仔细分类和枚举。这是解决这类问题所需要注意的。这里没有给出考虑各个情况的细节，这个工作留给读者去思考。

枚举法小结　枚举的思想是很简单的，但是可以引发我们很多的思考。

枚举什么？怎样枚举？你能算出枚举方案的总数吗？

你的枚举算法够快吗？如果很明显太慢或者你无法保证，有什么好的优化方法么？

对于第一个问题，往往需要首先对问题进行分析。需要枚举的东西往往没有我们想象的那么多，后面的习题中就有这样的情况。

对于第二个问题，依据有两个方面，一是第一个问题中得出的枚举量的大小，二是判断每个方案所用的时间。所以相应地，优化也应该从两个方面进行，即减少枚举量和减少判断时间。减少枚举量的方法已经在前一个问题中回答了，而判断时间的优化需要具体问题具体分析，本书在后面章节中会适当地介绍一些例题。

在前面的例题中都简要的分析了枚举算法的可行性。但很多情况，不得不怀疑有更好的方法（更快的指数级算法甚至多项式算法），应该进一步分析该问题来找到这个方法。在接下来的几节中，我们将介绍几种方法。

小知识(1)——枚举：理想的辅助算法

枚举（enumerate）法的用处比我们想象的要大。在问题毫无头绪的情况下，枚举为我们打开一个缺口，让其他算法可以成功。在“递推”一部分中，可以看到一个例子。不要怕使用枚举，重要的是要学会细致的权衡枚举的代价和得到的信息量之间的关系。

另一个需要指出的是“枚举”和“多次随机”的有趣关系。很多次随机以后，往往已经取到了所有需要枚举的值，即多次随机的“极限”（这里指的不是数学上的“极限”）就是枚举。但是多次随机有两个特点是枚举法不能比拟的：

① 不确定性：不像枚举通常是按照一定顺序枚举，难以避免在一个大范围之内各个决策都差不多但是必须全都枚举的情况。

② 灵活性：可以很方便地控制运行次数又不用过分担心无法跳出局部。尤其是“被迫”枚举的情况下，多次随机往往效果会比较好。

练　习　题

思考题：

1.2.1　枚举法的时间复杂度一定是指数级别的吗？

1.2.2　分析例题 2 中给出的算法的时空复杂度。有可能达到更低么？

编程题：

1.2.3　奇怪的问题[1]

请回答下面 10 个问题，各题都恰有一个答案是正确的：

（1）第一个答案是 b 的问题是哪一个？

[1] 题目来源：古老的智力题

（a）2　　（b）3　　（c）4　　（d）5　　（e）6

（2）恰好有两个连续问题的答案是一样的，它们是：

（a）2，3　　（b）3，4　　（c）4，5　　（d）5，6　　（e）6，7

（3）本问题答案和哪一个问题的答案相同？

（a）1　　（b）2　　（c）4　　（d）7　　（e）6

（4）答案是 a 的问题的个数是：

（a）0　　（b）1　　（c）2　　（d）3　　（e）4

（5）本问题答案和哪一个问题的答案相同？

（a）10　　（b）9　　（c）8　　（d）7　　（e）6

（6）答案是 a 的问题的个数和答案是什么的问题的个数相同？

（a）b　　（b）c　　（c）d　　（d）e　　（e）以上都不是

（7）按照字母顺序，本问题的答案和下一个问题的答案相差几个字母？

（a）4　　（b）3　　（c）2　　（d）1　　（e）0（注：a 和 b 相差一个字母）

（8）答案是元音字母的问题的个数是：

（a）2　　（b）3　　（c）4　　（d）5　　（e）6（注：a 和 e 是元音字母）

（9）答案是辅音字母的问题的个数是：

（a）一个质数　　（b）一个阶乘数　　（c）一个平方数

（d）一个立方数，　　（e）5 的倍数

（10）本问题的答案是：

（a）a　　（b）b　　（c）c　　（d）d　　（e）e

1.2.4　售货员[1]

叶卡特琳堡有很多公共汽车，因此也有很多市民当上了售票员。如果在所有的市民中，售票员的人数超过 P%而不到 Q%，那么叶卡特琳堡至少有多少市民呢？例如，如果 P=13 而 Q=14.1，那么至少有 15 个市民。

1.2.5　翻硬币[2]

考虑一个翻硬币游戏。有 N（$N\leqslant 10\ 000$）行硬币，每行有 9 个硬币，排成一个 $N\times 9$ 的方阵，有的硬币正面朝上，有的反面朝上。我们每次可以把一整行或者一整列的所有硬币翻过来，请问怎么翻，使得正面朝上的硬币尽量多。

1.2.6　离散函数[3]

有一个离散函数，定义在集合{1,2,3,…, N}，取值在$-2^{32}\cdots 2^{32}$。请找出函数图像上两个点，使得函数在这两点之间的点都在两点连线的下方，且此连线的斜率尽量大。$N\leqslant 10\ 000$。（注：时间复杂度为 $O(n^2)$的算法是容易想到的，那么 $O(n)$算法呢？）

1.2.7　超长数字串[4]

给一个数字串 S：12345678910111213141516171819202122…它是由所有自然数从小到大依次排列起来的。任意给一个数字串 S_1，容易知道它一定在 S 中出现无穷多次。编程求

---

[1] 题目来源：Ural State University Problem Archive 1012 Conductor

[2] 题目来源：Balkan Olympiad in Informatics 1999. Flip Coin

[3] 题目来源：Ural State University Problem Archive 1010 Discrete Function

[4] 题目来源：Ural State University Problem Archive

出它第一次出现的位置。例如对于串“81”，它最先出现在位置27。

1.2.2　贪心法

枚举法的思路很简单，但是时间效率不高。下面要介绍的贪心法不是考虑所有可能的方案，而是每次选择当前的最优策略，因此速度大大提高。

【例题1】钓鱼[①]

在一条水平路边，有 n（$2\leqslant n\leqslant 25$）个钓鱼湖，从左到右编号为1、2、3、…、n。佳佳有 H（$1\leqslant H\leqslant 16$）个小时的空余时间，他希望用这些时间钓到尽量多的鱼。他从湖1出发，向右走，有选择的在一些湖边停留一定的时间钓鱼，最后在某一个湖边结束钓鱼。佳佳测出从第 i 个湖到第 i+1 个湖需要走 $5\times T_i$ 分钟的路，还测出在第 i 个湖边停留，第一个5分钟可以钓到鱼 F_i，以后再每钓5分钟鱼，鱼量减少 D_i。为了简化问题，佳佳假定没有其他人钓鱼，也不会有其他因素影响他钓到期望数量的鱼。请编程求出能钓最多鱼的方案。

【分析】

为了叙述方便，把钓5分钟鱼称为钓一次鱼。首先枚举佳佳需要走过的湖泊数 X，即假设他从湖泊1走到湖泊 X，则路上花去时间 $T=\sum_{i=1}^{X-1}T_i$。在这个前提下，可以认为佳佳能从一个湖“瞬间转移”到另一个湖，即在任意一个时刻都可以从湖泊1到湖泊 X 中任选一个钓一次鱼（想一想，为什么?）。

现在，采取贪心策略，每次选一个鱼最多的湖泊钓一次鱼。对于每个湖泊来说，由于在任何时候鱼的数目只和佳佳在该湖里钓鱼的次数有关，和钓鱼总次数无关，所以这个策略是最优的（请读者仔细想想）。假设一共允许钓 k 次鱼，那么每次在 N 个湖泊中选择鱼最多的一个钓，选择每次钓鱼地点的时间复杂度为 $O(n)$，故总的时间复杂度为 $O(kn^2)$。在学习了1.4节后，读者可以尝试用“堆”来获得一个时间复杂度为 $O(k\text{nlog}n)$的算法，但即使用刚才讲的普通方法，贪心法的时间效率和枚举法相比已经是天壤之别了。

贪心法　每次选择一个局部最优策略进行实施，而不去考虑对今后的影响。一般来说，它的时间复杂度比较低，算法实现也比较容易。但很多题目贪心法并不能得到最优解。即使可以，也比较难证明解的最优性。需要注意的是，贪心法有一种推广，即无回溯的决策过程。有的题目看起来很难，但是如果证明当前决策至少不会比其他决策差，那么就可以一直决策下去。

对于复杂的例子，需要深入分析题目，得到一个简单的数学模型后才能看出是否能用贪心法。

【例题2】照亮的山景[②]

在一片山的上空，高度为 T 处有 N 个处于不同水平位置的灯泡，如图1-3所示。如果

[①] 题目来源：ACM/ICPC Regional Contest East Central North America, Gone Fishing
[②] 题目来源：CEOI 2000 Enlightened Landscape

山的边界上某一点与某灯 i 的连线不经过山上的其他点，我们称灯 i 可以照亮该点。开尽量少的灯，使得整个山景都被照亮。

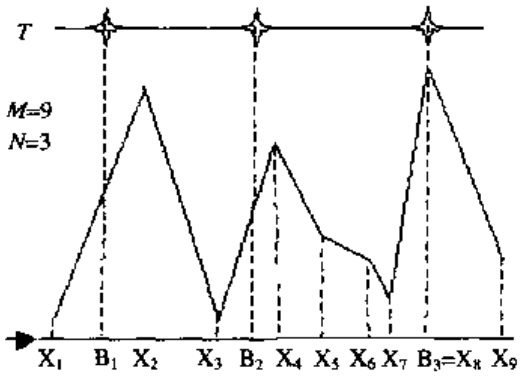


图 1-3　一片山和它们上空的灯

【分析】

本题的几何味道比较浓。山被表示为有 M 个转折点的折线，因此很容易想到把照亮整个山景的问题转化为照亮这 M 个折点（请读者思考：照亮 M 个折点就一定能照到整个山景吗？提示：考虑每条线段中较低的那个点）。

顺着这个思路，接下来可以考察每个灯的照射范围。我们发现一个令人头疼的问题：灯的照射范围是多区间的！如图 1-3 所示，灯 B_1 可以照射折点 $[X_1, X_2] + [X_4] + [X_8]$。不得以之下，我们换一个角度出发，考虑每个顶点能被哪些灯照到。我们高兴地发现，如果两盏灯可以照射到同一个顶点 i，则两盏灯之间的所有灯也能照到顶点 i（请读者画图验证），即，能**照射到一个点的所有灯**形成一个连续的区间 $[l[i], r[i]]$，其中 $l[i]$ 和 $r[i]$ 分别代表能照到灯 i 的最左边的灯和最右边的灯。

现在，我们的问题变成了：给 M 个区间，选出尽量少的点使得每个区间至少有一个点被选出。为了研究方便，首先把包含某其他区间的区间去掉（因为只要满足了小区间，则包含它的大区间也一定满足），然后把所有区间按照起始位置从小到大排序。这样，所有区间的终止位置严格递增（因为区间不相互包含）。

下面的事情就好做了。应该在第一个区间取哪个点呢？直觉告诉我们：应该选区间的最后一个点，因为它可以“让较多的其他区间得到满足”。这个结论是对的。因为对于任何一个最优解（选取点最少的解），如果第一个区间不是选的最后一个点，把那个点换成最后一个点之后，以前满足的区间现在仍然满足。所以，选取最后一个点是最优的。

这样，得到本题的贪心算法：①先计算每个转折点的区间，去掉包含其他区间的区间。②把所有区间按位置排序。③从前往后扫描排序后的每个区间，如果它上面还没有点被选出，则选它的最后一个点。

注意　本题的更一般情形即差分约束系统（differential constraint system），更多的资料请阅读本书主页。

【例题3】镜子盒[①]

数学家 Andris 有一个小盒子，盒子的底部是 $n \leqslant m$ 的格子，每一个格子都可以放一面 45° 朝向的的镜子。在盒子的边界，每行每列的两端，有一些小孔，光线可以从中射入盒子，也可以射出盒子。如图 1-4 所示，从孔 2 射进盒子的光线经过两次反射以后又从孔 7 射出。Andris 想请你设计一个盒子，使得从每个孔射入的光线都会由指定的孔射出。

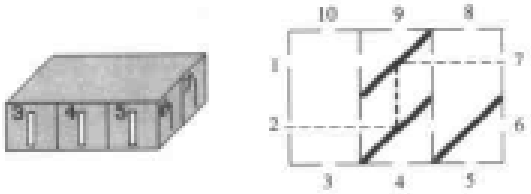


图 1-4　镜子盒及其内部结构

例如，如果它希望从 10 个孔射入的光线分别由孔 9,7,10,8,6,5,2,4,1,3 射出，如图 1-4 所示就是一个满足要求的盒子。

【分析】

题目中的条件非常苛刻，我们不得不朝着贪心法的方向思考。

初始时，镜子盒内不放任何镜子，每条光线射入后直线射出。为方便叙述，称从入口 i 射入的光线在从出口 A_i 射出，从出口 i 射出的光线来自入口 C_i，但我们期望添置若干镜子后最终从入口 i 射入的光线能从出口 B_i 射出。

如果光线 i 和 j 在一个没有镜子的方格内交叉，那么在此交叉方格内放一面镜子可以使得它们的光线出口相互交换，我们称这样的操作为 Change(i,j)。可知，Change(i,j)不会改变其他光线的光路。对于 $A_i \neq B_i$ 的入射光线 i，我们称 Change(i, C_{Bi})操作为 Turn(i)。我们的算法是依次检查每条入射光线 i，如果 $A_i \neq B_i$，那么令 $j = C_{Bi}$，并进行 Turn(i)操作，然后令 $i=j$，再次进行 Turn(i)操作……直到某次 Turn(i)后恰好 $A_j=B_j$。

如果可以保证在添置镜子的过程中，不改变所有 $A_i=B_i$ 光线的光路，同时有待进行 Turn(i)操作的光线 i 和光线 C_{Bi} 一定在某个没有镜子的方格内交叉，那么贪心算法就得以证明。

很明显，如果 $A_i=B_i$，那么任何 Turn 操作都不会涉及光线 i，也就是说我们一定不会在 i 的光路上放置新的镜子，所以 i 的光路不会被改变。

由于从 $n+1 \cdots n+m$ 射入的光线处理起来是一个道理，所以只讨论从 1，…，n 射入的光线。如图 1-5 所示，这是惟一一种光路不交叉的情况：

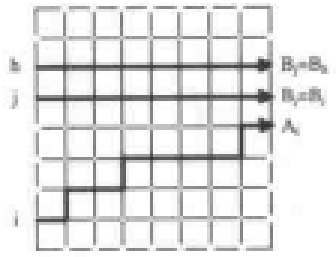


图 1-5　惟一一种光路不交叉的情况示意图

[①] 题目来源：Baltic Olympiad in Informatics 2001, Mirror

光线 i 横穿镜子盒，光线 $j=C_{Bi}$ 也横穿镜子盒，且 j 在 i 的上方。因为在此之前从未改变过 j 的光路，所以 j 是直线传播的。如果 $B_j\in[2n+m+1, 2n+m]$，那么先进行 Turn(j)操作，从而使得光线 i 和操作后的光线 C_{Bi} 相交；如果 $B_j\in[n+m+1, 2n+m]$，那么用对付光线 i 的办法对付光线 j……如此循环直到必然出现某条被考虑的光线 h，满足 $B_h\in[2n+m+1, 2(n+m)]$。我们最先进行 Turn(h)操作，然后依次回溯 Turn(j)、Turn(i)操作，问题便解决了。可见，即使出现光路不交叉的情况，也可以通过先进行 Turn(C_{Bi})操作，使得光路交叉。

贪心法小结　贪心法不应当只被狭义的理解为多阶段决策问题中每次选取局部最优的无回溯与分支的策略，而应该被当作一种解题方法和思维模式。探索贪心法需要清醒的头脑和创造性思维，选择使用贪心法往往需要一定的“勇气”，分析贪心法的性能（包括证明最优性）也常常是一种具有挑战性的工作。通过刚才的两道例题，相信大家对贪心法的威力和灵活性已经有了初步的了解。其实它的应用远不止这些，不过其中思路的产生和正确性的证明都要用到其他知识，我们会在后面的章节陆续给大家介绍。

小知识(2) —— 随机贪心法

即使不能保证得到最优解，贪心法也往往被用来求得不错的解　贪心法的效率高，但是解的质量不太好的情况下，常常用效率来换取质量，即多次贪心去求最优。“多次贪心”既可以是采用不同**贪心策略**，也可以是不同的**贪心程度**（即每次不一定选择局部最优，也可以是局部第二或者第三）。如果需要多次尝试不同的贪心程度，可以考虑用枚举法。但是前面说过，在被迫的情况下，多次随机往往效果更好，这种方法就是所谓的“随机贪心（randomized greedy）”。显然，随机贪心得到的解不比单纯贪心得到的差。

专题——子集系统优化问题、矩阵胚

***子集系统（subset system）**　把一个二元组(E, I)叫做一个子集系统，如果满足：

1．E 是一个非空集合

2．I 是 E 的一个子集族，它在包含运算下封闭，即 I 的每个元素 a 都是 E 的一个子集，并对于 a 的任何子集 a'，a'一定也是 I 的元素。

3．给 E 中每个元素 e 赋予一个正权 $w(e)$。

考虑至少有一条边的带权无向连通图 G[1]。令它的边集为 E，它的所有生成森林的集合为 I，则(E,I)是一个子集系统。这是显然的，因为 E 非空，所以满足条件 1；又因为生成森林可以看成是 E 的一个边集，而且生成森林的生成子图仍是生成森林，所以满足 2；G 是带权的，所以满足 3。

极大独立集（maximal independent set）　把 I 中的元素都称为独立集。对于 I 中的元素 a，如果不存在 I 中的另一个元素 a'使得 a 是 a'的真子集，则称 a 是极大独立集。该极大独立集的基数为它包含的元素个数。在刚才介绍的子集系统中，G 的所有生成树就是所有的极大独立集。所有极大独立集具有相同的基数$|V|-1$。其中$|V|$为 G 的顶点数。

子集系统优化问题　对于子集系统，定义优化问题如下：在子集系统(E, I)中选取一个元素 $S\in I$，使得 $w(S)$最大（定义 $w(S)$为 S 中所有元素的权和）。

一个容易想到的方法是贪心法：先把 E 中元素按照权值从大到小排序为 $e_1,e_2,\cdots$，令集

[1] 这里用到一点图论知识，详见 2.4 节

合 S=空集，然后每次尝试着把 $e_1,e_2,\cdots$，添加到 S 里面。如果添加之后 S 仍是独立集，则添加成功；如果 S 不是独立集，则由定义知以后无论怎样继续添加元素，得到的集合都不可能重新成为独立集，因此撤消此添加操作。当 S 是一个极大独立集时（此时无法继续添加元素）S 即为算法的输出。假设排序的时间复杂度为 $O(sort(|E|))$，检查 S 是否为独立集的时间复杂度为 $O(check(|E|)$，则贪心法的总时间复杂度为 $O(sort|E| + |E|check(|E|))$。

这个算法是贪心的，但并不能确保它能得到最优解。例如有可能出现因为添加了元素 e_1 而导致元素 e_2,e_3,e_4 都无法添加，而 $e_2+e_3+e_4>e_1$ 是可能成立的，这样得不偿失。我们在算法中只可能撤消当前的添加操作，而无法撤消以前的操作，所以贪心法有可能失败。很自然地，我们想知道这样的贪心法何时是正确的。为此，介绍一种特殊的子集系统。

矩阵胚(matroid)　一个子集系统如果满足以下性质，说它是一个**矩阵胚（matroid）**：对于任何两个独立集 S_1, S_2，如果$|S_1|<|S_2|$，那么 S_2-S_1 中一定存在一个元素 e 使得$\{e\}\cup S_1$ 仍是独立集（即：它仍是 I 中的元素）。这个性质称为**交换性质（exchange property）**。

有时候用定义判断一个子集系统是不是矩阵胚是不方便的。另外有一个定理：

定理 1：一个子集系统是矩阵胚当且仅当所有极大独立集具有相同的基数。

由这个定理我们立刻得出，前面提到的子集系统是一个矩阵胚。由线性代数的知识还可以得到：设一个矩阵的所有行集合 E，如果 E 的某个子集 S 中各行线性无关一定推出 $S\in I$，则(E, I)也是一个矩阵胚。

定理 2：子集系统优化问题的贪心法正确，当且仅当该子集系统是一个矩阵胚。

定理的证明留给读者思考，后面将直接使用这两个定理。

WEB　矩阵胚理论的具体应用请阅读本书主页。

练　习　题

思考题：

1.2.8　“如果一个问题有正确的贪心法，它一定是最优算法。”对么？

1.2.9　“枚举”一节的例题 1 可以用贪心法解决么？如果可以，怎么贪心是对的？如果不可以，如何证明？写个程序试验一下。

1.2.10　证明矩阵胚理论的定理 1 和定理 2。

1.2.11　证明以下子集系统是矩阵胚。

（1）给定 $k>0$ 和非空集合 E，I 是 E 的基数不超过 k 的全体集合。证明(E, I)是矩阵胚。

（2）已知(E, I)是矩阵胚，I'为所有满足以下条件的 S 的集合：E-S 包含 I 中的某个极大独立集。证明(E, I')是矩阵胚。（注：我们称(E, I')为(E, I)的补。因为子集系统需要对包含操作封闭，所以这个定义看起来比较别扭）。

（3）给定集合 E 和它的分划（partition）：S_1，S_2，…，S_k，令 I 为所有满足以下条件的 A 的集合：A 和任意 $S_i(1\leqslant i\leqslant k)$最多只有一个公共元素。即 A 可以由 S_1，S_2，…，S_k 中

的若干个集合，每个集合选出一个元素组合而成。证明(E, I)是矩阵胚。

编程题：

1.2.12　喷水装置[①]

有一块草坪，长为 l，宽为 w，在它的中心线上不同位置处装有 $n(n\leqslant 10\,000)$个点状的喷水装置。每个喷水装置 i 喷水的效果会让以它为中心半径为 r_i 的圆都被润湿。请选择尽量少的喷水装置，把整个草坪全部润湿，如图 1-6 所示。

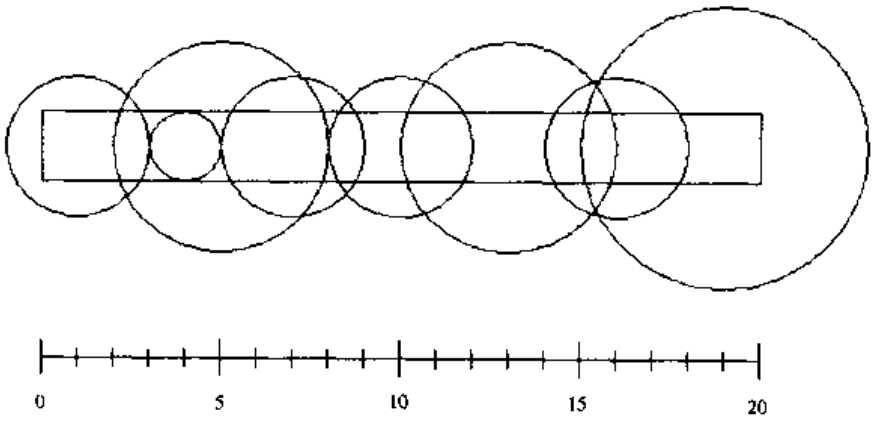


图 1-6　草坪和喷水装置

1.2.13　最优布车方案[②]

“阶梯”型棋盘指的是，每行的格子只出现在一些连续的列上。从第二行开始，每行的起始列数不小于上一行的起始列数，且终止列数也不小于上一行的终止列数，如图 1-7 所示。

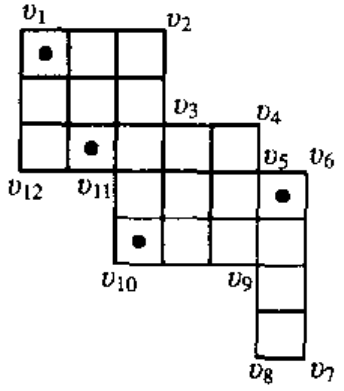


图 1-7　阶梯棋盘和布车方案

给出一个阶梯型棋盘，放上尽量少的“车”，控制所有的格子。一个车可以控制和它在同一行和同一列的所有格子。

图 1-7 描述了一个阶梯型棋盘和相应的最优布棋方案。

[①] 题目来源：University de Valladolid University Online Contest – World Finals Warmup 2002

[②] 题目来源：ACM/ICPC Regional Contest Asia Taipei 2000

***1.2.14 潜水比赛**[①]

在马其顿王国的 Ohide 湖里举行了一次潜水比赛。其中一个项目是从高山上跳下水，再潜水到达终点。这是一个团体项目，一支队伍由 N 人组成。在潜水时必须使用氧气瓶，但是每支队伍只有一个氧气瓶。最多两人同时使用一个氧气瓶，但此时两人必须同步游泳，因此两人到达终点的时间等于较慢的一个人单独游到终点所需要的时间。好在大家都很友好，因此任何两个人都愿意一起潜水。安排一种潜水的策略，使得最后一名选手尽量早的到达终点。

1.2.3 递归与分治法

和前两种思想比起来，递归是比较难于理解的，但又是一种重要的思想。如果一个问题可以转化成一个结构相同，规模更小的问题，可以通过递归来解决。递归也是一种分析方法，可以帮助我们看清楚事物的本质。

WEB 介绍递归时，几乎每本书都举了 Hanoi Tower 的例子。本书略过这个例子，而把它的一些相关资料（有些性质是优美而令人惊奇的！）放在本书主页上，欢迎访问。

【例题 1】折纸痕[②]

你喜欢折纸吗？给你一张很大的纸，对折以后再对折，再对折……每次对折都是从右往左折，因此在折了很多次以后，原先的大纸会变成一个窄窄的纸条。现在把这个纸条沿着折纸的痕迹打开，每次都只打开“一半”，即把每个痕迹做成一个直角，那么从纸的一端沿着和纸面平行的方向看过去，会看到一个美妙的曲线。

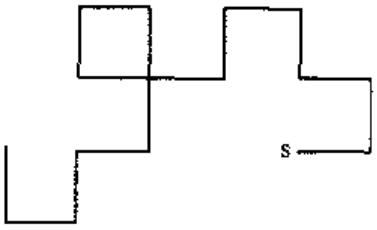


图 1-8 对折 4 次后形成的曲线

例如，如果你对折了 4 次，那么打开以后你将看到如图 1-8 所示的曲线。注意，该曲线是不自交的，虽然有两个转折点重合。给出对折的次数，请编程绘出打开后生成的曲线。

【分析】

刚拿到这个题目，初学者往往会觉得无从下手。折纸的过程是简单的，但生成的痕迹却是复杂的。不妨试着从简单的过程入手，寻找复杂的痕迹是如何生成的。

每次折叠的展开都可以看做是先把两个叠在一起的部分展成直角，然后分别展开两个部分。细心的读者一定已经注意到，两个部分是完全相同的。只需要求出一个部分展开的

[①] 题目来源：Balkan Olympiad in Informatics 2000 Diving, modified

[②] 题目来源：UVA Problem Archive

情况，也能知道另外一部分的情况，从而得出整张纸的情况。也就是说，需要求折纸次数为 n 的痕迹，只需要求出折纸次数为 $n-1$ 的痕迹，再把两个痕迹以直角连接起来。大家不妨验证一下 n=1,2,3 的情况是否符合。

递归法　递归法把问题转化为规模更小的子问题解决。递归法思路清晰，编程简单，但有时候难以想到。如果确定了用递归法解题，思考的重点应该放到建立原问题和子问题之间的联系。有的问题有很明显的递归结构，但是需要仔细思考，才能正确的转化为结构相同的子问题。

【例题 2】三色多边形[①]

有一个 $N(4\leqslant N\leqslant 1\ 000)$边形，所有的顶点都是红色、绿色和蓝色三种颜色之一，三种颜色都出现在该多边形的顶点上，且任意两个相邻顶点不同色。请用不在非顶点处相交的对角线把多边形切成 $N-2$ 个三角形，使得每个三角形的三个顶点都不同色。

【分析】

对于这样的题目，我们总是有自己设计一个切割方法的欲望。不妨试一试，首先让我们做一个大胆的猜想，即：任意一个符合上述条件的凸多边形都可以被成功划分，这样，如果能在不违反条件的前提下（注意，这就转换成了结构相同的子问题），将任意一个合法凸多边形的规模缩小，并找到合理的结束状态，那么试题也找到了递归的解决方案。

考虑特殊情况：某种颜色的顶点有且只有一个。于是很容易想到划分办法：以那个有独特颜色的顶点 i 为三角形的一个顶点，以任意相邻两顶点 $k,k+1(k,k+1\neq i)$为三角形另外两个顶点进行划分，如图 1-9 所示。

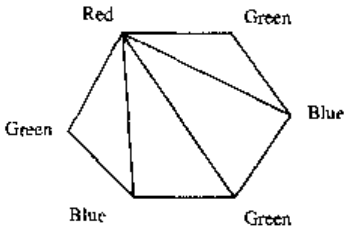


图 1-9　某颜色只有一个的情况

考虑特殊情况以外的其他情况：由于三种颜色都出现在该多边形的顶点上至少 2 次，且相邻顶点不同色，所以必存在连续的三个顶点 $k,k+1,k+2$ 两两颜色不同（想一想，为什么？）。如果将三角形 $k,k+1,k+2$ 从凸多边形上割下，那么剩余凸多边形仍然合法。至此，我们找到了将任意一个合法凸多边形规模缩小的办法，问题圆满解决。

递归是一种思维方法。熟悉了这种思维方式，可以解决一些初看起来无从入手的问题。

【例题 3】聪明的学生[②]

一位教授逻辑学的教授有三名非常善于推理且精于心算的学生 A、B 和 C。有一天，教授给他们三人出了一道题：教授在每个人脑门上贴了一张纸条并告诉他们，每个人的纸条上都写了一个正整数，且某两个数的和等于第三个。于是，每个学生都能看见贴在另外两个同学头上的整数，但却看不见自己的数。

这时，教授先对学生 A 发问了：“你能猜出自己的数吗？”A 回答：“不能。”教授又转身问学生 B：“你能猜出自己的数吗？”B 想了想，也回答：“不能。”教授再问学

① 题目来源：Ural State University Problem Archive
② 题目来源：CTSC 2001 Clever. 命题人：张力

生C同样的问题，C思考了片刻后，摇了摇头："不能。"接着，教授又重新问A同样的问题，再问B和C，……经过若干轮的提问之后，当教授再次询问某人时，此人突然露出了得意的笑容，把贴在自己头上的那个数准确无误的报了出来。

现在，如果告诉你：教授在第N次提问时，轮到回答问题的那个人猜出了贴在自己头上的数是M，你能推断出另外两个学生的头上贴的是什么数吗？

提示：总是头上贴着最大的那个数的人最先猜出自己头上的数。

【分析】

由题意可知，每个人推断的依据仅仅是另外两个人的头上数，以及大家对教授的提问所做出的否定回答，因此，找出他们推理的过程就成为解决本题的关键。

由于三个正整数中，一定有某个数是另两个数的和，因此，当一个学生看到另两个学生头上的数时，他就知道自己的数只有两种可能，另两数的和或差；而要猜出自己头上的数，只需否定两种可能中的一种，那另一个就是答案了。

因为三个数都是正整数，所以，若某个学生看到另两个学生的数相等，那他立刻就可以判断自己的数是另两个数的和（因为不可能是0），相反，若他猜不出来，则给了另两个人一个信息——自己的数和第三个人的数不同，而这也可以成为作出推理的一个重要条件。

为了把问题说得更加清楚，不妨举个例子来说：当N=5，M=8时，有一个可能的情况是A，B，C三人头上贴的数分别为2，8，6。B为什么能在第5次提问时——即第2次问到他时——猜出自己的数呢？他是这样想的：

"我看到的是A的2和C的6，因此，我头上的数只可能是4（=6−2）或8（=2+6）。但假如我的数是4，那在教授的第3次提问时，C就能猜出自己的数来！——因为那时他看到的是A的2和我的4，C就面临着2（=4−2）和6（=2+4）的选择，但假如他是2，那在教授第2次提问时，我就能猜出来，因为我看到的是两个2！因此他能肯定自己不是2，而是6。——而实际上C并未猜出，由此，我头上的数必定是8！"

B的推理实际上是一个"假设——排除"的过程，即假设自己的数为两个可能数中一个，如果另两人中有人可以在自己之前猜出，那么被假设的那种情况就可以排除，从而也就猜出了自己的数。而原题中又给我们提示："总是头上贴着最大的那个数的人最先猜出自己头上的数"，因此只需要排除另两个数的差的那种情况就可以了。

现在，实际上已经可以解决这样一个问题：

已知A，B，C三人头上贴的数为X_1，X_2，X_3，求教授至少需提问多少次，轮到回答问题的那个人才能猜出自己头上的数。

再来分析一下这个问题。由原题提示中的结论"总是头上贴着最大的那个数的人最先猜出自己头上的数"，于是当X_1，X_2，X_3给出之后，谁最先猜出就已经确定了。不妨设最终猜出的人是B，那么即有$X_1+X_3=X_2$，而B作出判断的依据即是排除了$X_2=|X_1-X_3|$的可能，而他能够排除这种可能的依据则是$X_1=X_3$，或假设$X_2=|X_1-X_3|$的前提下，在前两次提问时，A或C必定能够猜出自己的数，而实际上他们没有猜出——即当前教授提问的次数，已经大于当三人头上的数分别为X_1，$|X_1-X_3|$，X_3时，直到某人猜出自己的数为止教授需要提问的次数，这样B就能确定自己的数。而对于X_1，$|X_1-X_3|$，X_3时求提问次数，就又成了

一个子问题，在这种情况下，最先猜出自己的数的人即为 A 或 C（这取决于 X_1 与 X_3 的大小，若 $X_1>X_3$，则 A 先猜出，否则就是 C 先猜出），这样就又安装上面的思路求解。

这显然就是一个递归求解的过程。

设函数 $Times(i, j, t_1, t_2, t_3)$表示编号为 t_1[1]的人头上的数为 i，编号为 t_2 的人的数为 j，编号为 t_3 的人的数为 $i+j$（即由 t_3 最先猜出自己的数）时，教授需要提问的次数；$P(t_1,t_2)$表示教授按照 1-2-3 的顺序，从 t_1 问到 t_2，最少需要提问的次数[2]，则根据上面的分析，有如下关系：

$$times(i,j,t_1,t_2,t_3)=\begin{cases}t_3, & i=j\\ times(j,i-j,t_2,t_3,t_1)+P(t_1,t_3), & i>j\\ times(i,j-i,t_1,t_3,t_2)+P(t_2,t_3), & i<j\end{cases}$$

于是，根据这一递归函数即可解决上面提出的问题。

当然，前面解决的问题和原题还有不同，原题已知 N，M 要求三人头上的数，而刚解决的问题是已知三人头上的数，要求 N，M——恰好将问题与条件互换了。那么如何利用已有的结论来解决原来的问题呢？很简单——用枚举法（还记得前面强调过的么？枚举可以为其他算法创造条件）！

由于 M 是三个数中最大的，则另两数只能在区间[1, M-1]内，又因为有这两数之和等于 M，因此，只需枚举其中一个 i，另一个就可以由 M-i 求出。对于枚举的每一种情况，判断其相应的 *Times* 函数值是否等于 N，若等于，则为问题的一个解。至此，本题已获圆满解决。

小知识(3) —— 分形

如果你已经习惯了递归的思维方式，笔者建议你学一点简单的**分形（fractal）**。简单地说，分形的意思是“自相似”，即自己的某部分和整体相似——典型的“递归感觉”。

什么是“自相似”呢？这里举一个例子。

把所有自然数从小到大写出来：1, 2, 3, 4, 5, 6, 7, 8, 9,10, 11, 1, 2,…

写成二进制：1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100, …

数一数每个数的“1”的个数，得到一个序列：1, **1**, 2, **1**, 2, **2**, 3, **1**, 2, **2**, 3, **2**, …

有没有发现这个数列很神奇？把它的奇数项都去掉，得到的序列和自己完全一样！还有一个方法也能得到这个的数列。从 0 开始。每次把刚才的序列的每个数后面都填一个它的后继，即 $n\to n\ (n+1)$。像这样（为了观察方便，加入了空格）：

序列 1：　0

序列 2：　**0**　　1

序列 3：　**0**　1　**1**　2

序列 4：　**0**　1　**1**　2　**1**　2　**2**　3

序列 5：　**0 1 1 2 1 2 2 3 1 2 2 3 2 3 3** 4

第三种构造方法是每次把前一个序列的所有元素复制一份，再把复制的元素都增加一，

[1] 为了叙述方便，特将 A，B，C 三人编号为 1，2，3，下同

[2] 实际上，当 $t_2>t_1$ 时，$P(t_1,t_2)=t_2-t_1$，否则，$P(t_1,t_2)=t_2+3-t_1$

像这样：

序列 1：0

序列 2：0 1

序列 3：0 1 1 2

序列 4：0 1 1 2 1 2 2 3

序列 5：0 1 1 2 1 2 2 3 1 2 2 3 2 3 3 4

这是一种比较简单的“自相似”。在本书的第 2 章，我们将看到一个更直观的自相似例子，希望读者一看就能看出那是分形。

分形理论博大精深，在数学、物理、化学、生物、艺术、经济等各方面都有着非常广泛的应用，但是本书限于篇幅无法详细加以叙述。

WEB 关于分形的讨论和一些 internet 资源请访问本书主页。

递归法的一个典型应用是分治。这里举一个简单的例子。

【例题 4】丢失的数[①]

给出 n-1 个数 a_1，a_2，…，a_{n-1}，每个数都是 1~n 中的自然数，而且两两不相同。显然，有且仅有一个 1~n 中的数未在 a_1，a_2，…，a_{n-1} 中出现。是哪个呢？假设每次可以询问第 i 个数的右数第 j 个二进制位 d_{ij}，请用尽量少的询问次数确定这个丢失的数。

【分析】

一个最笨的方法是询问所有二进制的位，算出每个 a_i，把它们放在一个长度为 n 的数组里。最后检查哪个位置没有元素放进去，那个位置所代表的数就是所求。由于每个数有 $\log_2 n$ 个二进制的位，因此询问总次数为 $n\log_2 n$。这个方法把所有的信息都获取了，有没有更好的方法呢？我们用递归法来解决这一问题。为了方便读者理解，举个例子。

假如 n=16，16 个数是 0111，0010，0101，1010，0001，1111，0110，1000，0100，0011，1110，1011，1100，0000，1101，丢失的数是 1001。先询问所有数的最后一位（即除以 2 的余数），如表 1-2 所示。

表 1-2　第一次询问结果

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
d_{i0}	1	0	1	0	1	1	0	0	0	1	0	1	0	0	1

经计算（请读者自己验证），1~16 的数中末位为 0 和 1 的应各占 8 个。现在末位为 1 的只有 7 个，因此丢失的数末尾为 1。这样，我们排除了 8 种可能性，把范围缩小了一半。

现在询问倒数第二位（注意只需要询问 7 个元素），如表 1-3 所示。

表 1-3　第二次询问结果

i	1	3	5	6	10	12	15
d_{i1}	1	0	0	1	1	1	0

① 题目来源：经典问题

和刚才类似，末尾为 1 的数中，倒数第二位为 0 的应该有 4 个，现在只有 3 个，因此丢失的数倒数第二位为 0。现在我们的范围再次缩小。这一次是问倒数第三位，如表 1-4 所示。

表 1-4　第三次询问结果

i	3	5	6
$d_{i,2}$	1	0	1

显然，丢失的数倒数第三位应为 5。由于现在只剩一个数了，a_5 的第四位为 0，因此丢失的数第四位为 1，答案为 1001。一共询问了 15+7+3+1=26 次，比刚才的 $16*\log_2 16=64$ 次好得多。

回顾刚才的解法，每次把范围缩小了一半，第一轮询问后得知丢失的数末尾为 1，因此把询问范围缩小为末尾为 1 的数中，试图找出它们中是哪个数丢失了。这样，让 T(n)代表原问题的询问次数，有 T(n) = T(n/2) + n（后面将看到，除以 2 是向上取整还是向下取整无关紧要），而 T(1) = 1 解出这个方程，就得到了该方法的询问次数。容易用数学归纳法证明，T(n)≈2n。

专题二——解递归方程、主定理

刚才我们很容易得出了算法，但是在得出最后的结论之前先得到了递归方程 T(n)=T(n/2)+n，通过解这个递归方程才得到我们的结论。递归方程的形式有很多，解法也有很多，限于篇幅，这里只介绍递归树方法和主定理及其简单推广。

考虑这样的递归式：

$$T(n) = aT(n/b) + f(n)$$

其中 a 和 b 是常数，而 $f(n)$是 n 的某个函数。如果把递归过程写成一棵树，如图 1-10 所示。

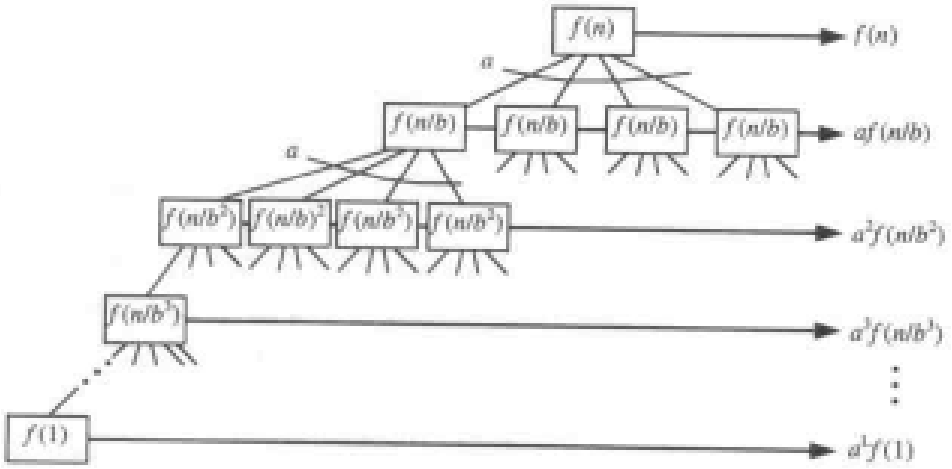


图 1-10　递归树

因此所求结果应为：$T(n) = f(n) + af(n/b) + a^2 f(n/b^2) + \cdots + a^L f(n/b^L)$。其中，$n/b^L=1$，因此 $L=\log_b n$。假定总是有 $f(1)=\Theta(1)$，因此连加式的最后一项应为 $\Theta(a^L)= \Theta(a^{\log_b n})=\Theta(n^{\log_b a})$

下面介绍简化的主定理（对于它的一般表示请读者阅读《算法导论》）的证明概要：

简化的主定理（master theorem）：对于常数 $a \geq 1, b > 1$，$f(n)$是一个函数，$T(n)$递归定义如下：

$$T(n) = aT(n/b) + f(n)$$

其中 n/b 理解为 n/b 取向上取整或者向下取整都可以。则 $T(n)$可以通过比较 $a*f(n/b)$和 $f(n)$来计算。

情形 1：如果存在常数 $K>1$ 使得 $a*f(n/b) \leqslant f(n)/K$，则 $T(n) = \Theta(f(n))$

情形 2：如果存在常数 $K>1$ 使得 $a*f(n/b) \geqslant Kf(n)$，则 $T(n) = \Theta(n^{\log_b a})$

情形 3：如果 $a*f(n/b) = f(n)$，则 $T(n) = \Theta(f(n)\log_b n)$

利用刚才的递归树，很容易证明此定理：情形 3 时，$T(n)$的每个加数都相等，而项数为 $L=\log_b n$；情形 1 和情形 3 的连加和是一个几何级数，和为最大项的常数倍（请读者证明）。在情形 1 中最大项是第一项 $f(n)$，情形 2 中最大项是最后一项。下面举例如下：

（1）$T(n) = T(n/2) + n$

这是刚才的例子。$a = 1, b = 2, f(n) = n, a*f(n/b) = n/2$。取 $K=2$，满足情形 1，因此为 $\Theta(n)$。

（2）$T(n) = 2T(n/2) + 1$

$a = b = 2, f(n) = 1, a*f(n/b) = 2$，取 $K=2$ 满足情形 2，因为 $\log_b a=1$，因此仍为 $\Theta(n)$。

（3）$T(n) = 2T(n/2) + n$

$a = b = 2, f(n) = n$，$a*f(n/b) = n = f(n)$，满足情形 3，为 $\Theta(n\log_2 n)$。

（4）$T(n) = T(n/2) + 1$

$a = 1, b = 2, f(n) = 1, a*f(n/b) = 1 = f(n)$，满足情形 3，因此为 $\Theta(\log n)$。

下面的例子复杂一些，主定理不再有用，需要用递归树，变量代换等其他途径来解决问题。

（5）$T(n) = 2T(n^{1/2}) + \log_2 n$

这个方程不是主定理给出的形式，不过可以用简单的变量代换来解决。令 $m=\log_2 n$，则 $T(2^m) = 2T(2^{m/2}) + m$，然后令 $S(m) = T(2^m)=T(n)$，则有 $S(m) = 2S(m/2) + m$。然后仍然有 $S(1)= \Theta(1)$，因此根据前面的解答，$S(m) = \Theta(m\log m)$，因此 $T(n) = \Theta(\log n \log\log n)$。

（6）$T(n) = 4T(n/2) + n\log_2 n$

尝试着应用主定理，$a = 4, b = 2, f(n) = n\log_2 n$，则 $a*f(n/b) = 4*n/2*\log_2(n/2) = 2n\log_2 n - 2n$。这比 $2f(n)$要小一点。怎么办呢？取 $K=1$ 即可。在渐进意义上，只要 n 稍微大一点，马上就有 $a*f(n/b) > f(n)$了。因此符合情形 2，$T(n) = \Theta(n^2)$。

（7）$T(n) = 2T(n/2) + n/\log_2 n$

这里的 $f(n) = n/\log_2 n$ 似乎比较奇怪，应用主定理时遇到了问题，因此需要从更为基本的内容——递归树来考虑。

第 0 层：$f(n) = n/\log_2 n$

第 1 层：$af(n/b) = 2*n/2/\log_2(n/2) = n/(\log_2 n-1)$

第 2 层：$a^2 f(n/b^2) = 4*n/4/\log_2(n/4) = n/(\log_2 n-2)$

…

第 i 层：$a^i f(n/b^i) = 2^i*n/2^i/\log_2(n/2^i) = n/(\log_2 n-i)$

这样，可以直接把和式计算出来。最后一层即是第 $\log_2 n-1$ 层，因此

$$T(n)=\sum_{i=0}^{\log_2 n-1}\frac{n}{\log_2 n-i}=\sum_{j=\log_2 n}^{1}\frac{n}{j}=nH_{\log_2 n}=n\ln\log_2 n=\Theta(n\log\log n)$$

注意，刚才使用了变量代换，并利用到了结论（$\{H_n\}$称为调和级数）：

$$H_n=1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}\approx\ln n$$

（8）$T(n)=T(3n/4)+T(n/4)+n$

这里的 $T(n)$不符合主定理中的形式，因此仍然需要从递归树考虑。这里的递归树中不同的叶子具有不同的深度，因此 $T(n)$不能写成比较简单的和式。不过容易证明，所有叶子之上的任意一层（不妨称它们为“完全层”）的结点函数值总和为 n，因此 $T(n)$应该在通过分别忽略深度大的叶子和深度小的叶子，有：$n\log_4 n\leqslant T(n)\leqslant n\log_{4/3}n$。由于上下界只相差一个常数，所以 $T(n)=\Theta(n\log n)$。后面将看到，这个递归式是随机快速排序的情形。

（9）$T(n)=T(n/5)+T(7n/10)+n$

和刚才不一样，完全层的结点函数值总和不再是 n 了。第 0 层是 n，第 1 层是 $9n/10$，第 2 层是 $81n/100\cdots$。因此如果仍只考虑完全层，则完全层的结点总数为：$T(n)=n+9n/10+81n/100+\cdots$。考虑上界：简单地把树伸展到无穷，则 $T_{ub}(n)=\Theta(n)$（请读者证明）。下界：只考虑完全层，则仍有 $T_{lb}(n)=\Theta(n)$。上下界相等，故 $T(n)=\Theta(n)$。

（10）$T(n)=n^{1/2}T(n^{1/2})+n$

这一次仍然从递归树入手。每个结点的度不相同，但是每一层的结点函数值之和仍然是 n（请读者证明），因此仍可以用求和的方法来计算。设树的层数为 L，则递归到第 L 层后所剩结点 $n^{2^{-L}}=2$（想一想，为什么不是等于 1？），因此 $2^{-L}=\log_n 2$，因此 $L=\log_2\log_2 n$。这样，$T(n)$为 L 个 n 之和，因此 $T(n)=\Theta(n\log\log n)$。

（11）$T(n)=4n^{1/2}T(n^{1/2})+n$

和刚才类似，第 i 层的结点函数值之和为 4^i，总层数为 $\log_2\log_2 n$，则和式为一个几何级数，其和与最大项（最后一项）同阶，即 $T(n)=\Theta(4^{\log\log n}n)=\Theta(n\log^2 n)$。

（12）$T(n)=T(n/2)+T(n/4)+1$

和（9）类似，第 i 个完全层的结点函数值和为 2^i，因此可以得到上界（这次不能把树伸展到无穷，只需要让每个叶子都成为第 $\log_2 n$ 层即可），为 $\Theta(n)$，下界为完全层的结点函数值和，则下界为 $\Theta(n^{1/2})$，无法夹逼出 $T(n)$。正确的做法是令 $t(k)=T(2^k)$，则

$$t(k)=t(k-1)+t(k-2)+1$$

用母函数的方法可以求出（见本书 2.3 节）$t(k)=\Theta(\Phi^k)$，其中 Φ 为黄金比率 $\frac{1+\sqrt{5}}{2}\approx 0.618$，因此 $T(n)=t(\log_2 n)=\Theta(\Phi^{\log k})=\Theta(n^{\log\Phi})\approx\Theta(n^{0.69424})$，比线性时间复杂度还要低。一个例子是计算几何中 ham-sandwich 树。

WEB 关于带有 min/max 函数的情形以及主定理的推广，请阅读本书主页。

练　习　题

思考题：

1.2.15　用递归的方式思考问题有什么好处？是否应该尽量用递归的方式思考问题？

1.2.16　本节的三道例题中，会不会出现解决 A 问题需要先解决 B 问题，但解决 B 问题又需要先解决 A 问题的情况？请分别说明。

***1.2.17**　如果 1.2.6 所说的情况出现了，显然是不能用递归方法的。请问：在这样的情况下，你能想到哪些可能会有效的方法呢？

编程题：

1.2.18　蚂蚁的递归访问[①]

在一个规模为 $2^n \times 2^n$ 的棋盘上，其中的一些格子被标记为不可经过的区域。一只蚂蚁计划访问棋盘上所有可以经过的格子，并且这些格子被且只被访问一次。蚂蚁是从棋盘的左上角开始行进的，访问路线必须结束在棋盘的边上——这样蚂蚁才可以离开棋盘。当然，我们假设初始时蚂蚁所在的格子是可以经过的。

每一步，蚂蚁可以从一个格子移动到与其相邻的格子中（也是就向上、下、左、右四个方向移动）。

蚂蚁的访问路线是递归定义的。也就是说，蚂蚁如果要访问一个 $2^k \times 2^k$ 的棋盘，那么将这个棋盘划分成 4 份 $2^{k-1} \times 2^{k-1}$ 的规模，在每一部分中，蚂蚁的访问也必须满足先前的条件：必须将这部分的所有可经过格子都访问一遍，然后才能访问另一部分。当然，对这个 $2^{k-1} \times 2^{k-1}$ 规模的部分棋盘，我们又可以将其划分成 4 部分继续递归的定义蚂蚁访问的路线……

如图 1-11 所示中给出了两条蚂蚁的访问路线，棋盘的规模是 $2^3 \times 2^3$。这两条路线都是从左上角（0，0）点开始的。第一条路线结束点是在棋盘顶部的边上，第二条路线的结束点是在棋盘左侧的边上。

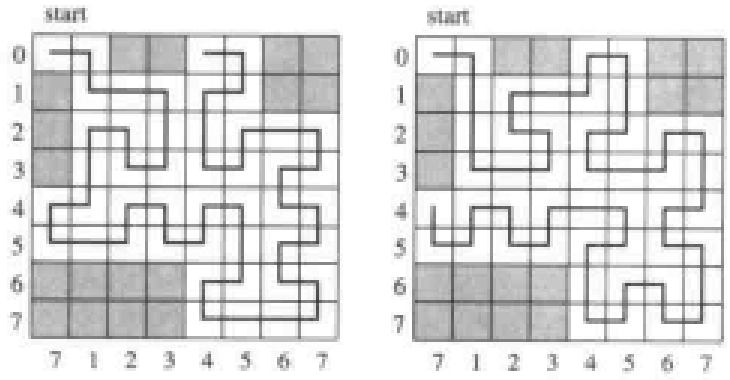


图 1-11　蚂蚁的两种访问路线

[①] 题目来源：Polish Olympiad in Informatics

编写一个程序，对给定的棋盘和已经标记过不可经过的格子，告诉蚂蚁，能否确定一条访问路线使得蚂蚁最后的出口在棋盘上的某条边上（顶部边、底部边、左侧边或右侧边）。

1.2.4　递推

递推和递归有着很多的相似之处，甚至可以看做是递归的反向。例如在上一节的“折纸痕”问题中，也可以从 n=1 的痕迹一步步递推出 n=2，3，…，的图形。前面已经看到了，递归的目的性很强，只解需要解的问题，而递推有点“步步为营”的味道，我们不断的利用已有的信息推导出新的东西，而递归是构造出了一个通过简化问题来解决问题的途径。递推在组合数学中有着典型应用，我们会在相关章节进行讨论，这里希望通过一些有趣的题目介绍递推算法中包含的思想。利用现有信息得到新信息，是递推的精髓。

【例题 1】月亮之眼[①]

吉儿是一家古董店的老板娘，由于她经营有道，小店开得红红火火。昨天，吉儿无意之中得到了散落民间几百年的珍宝——月亮之眼。吉儿深知“月亮之眼”价值连城，它是由许多珍珠相连而成的，工匠们用金线连接珍珠，每根金线连接两个珍珠；同时又对每根金线染上两种颜色，一半染成银白色，一半染成黛黑色。由于吉儿自小熟读古籍，所以还晓得“月亮之眼”的神秘传说：“月亮之眼”原是古代一个寺庙的宝物，原本是挂在佛堂一根顶梁柱上的，整个宝物垂直悬挂，所有珍珠排成一线，且都镶嵌在柱子里，而每一根金线又都是绷紧的，并且金线的银白色一端始终在黛黑色一端的上方，如图 1-12 所示；然而，在一个月圆之夜，“月亮之眼”突然从柱里飞出，掉落下来，宝物本身完好无损，只是僧侣们再也无法以原样把“月亮之眼”嵌入柱子中了。吉儿望着这个神秘的宝物，回忆着童年读到的传说，顿时萌发出恢复“月亮之眼”的冲动，但是摆弄了几天依旧没有成功。

现在，要麻烦您来帮助吉儿完成这项使命。

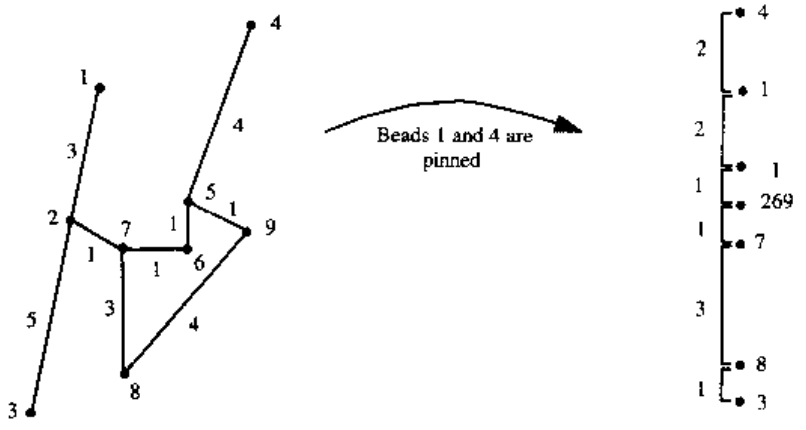


图 1-12　“月亮之眼”和它的镶嵌方案

要设计一个程序，对于给定的“月亮之眼”进行周密分析，然后给出这串宝物几百年

① 题目来源：Balkan Olympiad In Informatics 1998. Evil Eyes

前嵌在佛堂顶梁柱上的排列模样。给定的“月亮之眼”有 n（$n \leqslant 255$）个珍珠和 P 根金线，所有珍珠按一定顺序有了一个序号：1, 2, 3，…，n。每根金线由三个整数 R_1,R_2,L 表示。R_1 表示此金线银白色一端连接的珍珠序号；R_2 表示此金线黛黑色一端连接的珍珠序号；L 表示金线长度。

【分析】

设珍珠 x 离柱子底部的高度为 $h[x]$，那么由题意可以知道：如果有一条长度为 i 的金线，银白色一端为珍珠 a，而黛黑色一端为珍珠 b，那么 $h[a]-h[b]=i$。如果知道了 $h[a]$，那么根据这个方程可以知道 $h[b]$。如果可以列出关于 $h[b]$和另外一个柱子 c 的方程（如 $h[c]-h[b]=i'$），那么可以推出 $h[c]$。这样，不断地寻找恰好有一个珠子位置已知的方程，推出另外一个珠子的位置，最终可以推出整个“月亮之眼”的安放情况。每次需要检查 p 个方程，共检查 n 次（每次推出一个柱子），所以算法的时间复杂度为 $O(np)$（注：其实并不需要每次都检查所有的方程看看有没有可以新的可以计算出来的珠子，读者在学习 1.3 节的“拓扑排序”之后很容易得到一个 $O(n+p)$的算法）。

问题在于，怎样找到第一个已知的 $h[x]$呢？很简单，随便指定一个就可以了。由于给出的信息只能确定各个珠子的相对位置，所以可以先假定其中一个的绝对位置，最后再作调整。例如，图 1-12 中的例子有 9 条金线，如表 1-5 所示。

表 1-5　月亮之眼中的金线和珍珠

编　　号	1	2	3	4	5	6	7	8	9
银白色珍珠	1	2	2	4	5	5	6	7	9
黛黑色珍珠	2	3	7	5	6	9	7	8	8
长　　度	3	5	1	4	1	1	1	3	4

按照我们的方法，首先假设 $h[1]=0$，由金线 1 知 $h[2]=-3$，由金线 2 知 $h[3]=-8$，由金线 3 知 $h[7]=-4$。用类似的方法求出 $h=(0,-3,-8,2,-2,-3,-4,-7,-3)$。由此可见珍珠 3 才是处于最低的位置的，令 $h[3]=0$，把月亮之眼往上平移 8 个单位后得到 $h=(8,5,0,10,6,5,4,1,5)$，如图 1-12 所示。

递推法　根据已知信息不断计算出未知信息，直到得到结果。它和递归比较起来，优点比较灵活，计算不需要按照特定的顺序，但思路往往不容易想到。

递推的时候不要着急，即使一次只能得到一个数，我们也有希望从这个数开始慢慢的推出其他数。在这里，极端原则往往起着很重要的作用。

【例题 2】Yanghee 的数表[①]

Yanghee 是一个小学生。他的数学老师给全班同学布置了一道家庭作业，即根据一张由 n（$n<50$）不超过 5 000 个的正整数组成的数表，两两相加得到 $n(n-1)/2$ 个和，然后把它们排序。例如，如果数表含有四个数 1,3,4,9，那么正确答案应该是 4,5,7,10,12,13。Yanghee 做完作业以后和小伙伴们出去玩了一下午，回家以后发现老师给的数表不见了，可是他算

① 题目来源：ACM/ICPC Regional Contest Asia Taejon 2000. Lost Lists

出的答案还在。你能帮助 Yanghee 根据他的答案计算出原来的数表吗？

【分析】

为了研究方便，设这 n 个整数从小到大依次为 A_1，A_2，A_3，…，也将 $n(n-1)/2$ 个和数从小到大依次设为 K_1，K_2，K_3，…。

从边缘数据入手，根据简单的大小比较，很容易发现 $A_1+A_2=K_1$, $A_1+A_3=K_2$。因为无法直接确定 A_2+A_3 的大小，所以只能假设 $A_2+A_3=K_x$，然后通过解方程求出 A_1、A_2、A_3 的值。知道 A_1、A_2、A_3 以后，接下来的工作就是依次递推出每个数。假设已经求出前 W 个数的值，并将相应的 $W(W-1)/2$ 个和数从数列$\{K_n\}$中去除，那么考虑剩下的和数中最小的那一个 K_i。为使得 K_i 最小，K_i 必然是 A_1~A_w 中的最小数与 A_{w+1}~A_n 中的最小数相加而得，即 $K_i=A_1+A_{w+1}$。由于 K_i、A_1 已知，因此 A_{w+1} 也确定了。

或许读者对刚才介绍的方法还有些糊涂，下面举个例子。

K={4, 5, 7, 10, 11, 12, 13, 13, 14, 19}，因为 A_1+A_2 总是最小的，A_1+A_3 第二小（想一想，为什么？），故：

$$\begin{cases} A_1 + A_2 = 4 \\ A_1 + A_3 = 5 \end{cases}$$

我们不知道 A_2+A_3 的值，但是由于只有 $A_1+A_x(1\leqslant x\leqslant n)$可能比它小，因此它在 K 中的位置应该为 3~（n+1）。我们枚举每种情况。

假设 $A_2+A_3=K_3=7$，得到方程组：

$$\begin{cases} A_1 + A_2 = 4 \\ A_1 + A_3 = 5 \\ A_2 + A_3 = 7 \end{cases}$$

它有整数解 $A_1=1$，$A_2=3$，$A_3=4$。把这三个数两两之和 K_1,K_2,K_3 去掉，则现在的 K={10,11,12,13,13,14,19}。这三个数中最小的是 10，显然它等于 A_1+A_4，因此 $A_4=10-A_1=9$。把 A_4 产生的和 A_1+A_4, A_2+A_4，A_3+A_4 去掉，得：K={11,13,14,19}。其中最小数 11 应该等于 A_1+A_5，因此 $A_5=11-A_1=10$。

假定 $A_2+A_3=K_4=10$，方程组没有整数解；

假定 $A_2+A_3=K_5=11$，方程组没有正整数解。

因此本题的惟一解是 A={1,3,4,9,10}。枚举 x 的值($A_2+A_3=K_x$)需要 $O(n)$次，每次枚举需要递推 $O(n)$个数，每递推一个数需要用 $O(n)$的时间来去掉已经得到的和，故总体复杂度 $O(n^3)$，问题顺利解决。（注意，这里枚举又充当了辅助算法。没有它，递推将无法进行）

WEB 本书的主页里有更多的例子。其中有利用二进制展开来做的递推，还有如 Joseph 问题这样的特殊递推。总之，递推不一定需要是每次增 1 或者按照拓扑顺序。

小知识——递推的几种用法

最简单的递推方式是迭代（iteration），它可以看成是按部就班的填充表。它被用来实现我们后面将介绍的动态规划。这个“按部就班”可以是简单的线性序（把“折纸痕”的递归法改成递推，就是从 1 推到 2，2 推到 3，…，的简单顺序），也可以是偏序关系进行

拓扑排序后得到的，如“月亮之眼”。

递推的顺序可以是不确定的，思考题中的（时区）就是一个很好的例子。它是后面将会介绍的约束传播方法的一种特殊情形，在递推前并不知道元素会以怎样的顺序被确定，而是采取“步步为营”的方法，每次把可以推出来的结果推出。这样的递推方法比较灵活，它的思想也广泛地用到状态空间搜索中，即当目前已经无法递推出任何元素的时候开始枚举，递推得以继续进行。

如果视野再开阔些，我们可以采取另一种方法——不是枚举而是假定。这种方法当题目的数学关系比较强时才有用——递推不行的时候随便指定一个，继续递推。当然，由于值是假定的，我们可能会推出矛盾。所以我们把“随便指定”改成设未知数 x，则“矛盾”变成了方程。递推的结果是一个方程组，在很多情况下是有不错的经典解法的。

专题三——询问式交互问题

计算机除了能根据输入数据经过计算后得到输出数据，还有一个重要的应用是和**用户**（**user**）进行**交互**（**interact**）。前面的“丢失的数”问题就是一个简单的交互式问题。该小节的练习题部分也出现了交互问题。这类问题一般分为两类，一是询问式，二是博弈式。一般说来前者的用户一般是被动的，程序一般需要用尽量少的询问次数来获取想要的信息。后者的用户一般是主动的，程序需要和用户对抗或者合作达到一个特定的目的。询问式交互问题类型很多，解法也非常灵活，在本节中，我们考虑用递推的思想解决一个看起来不容易的询问式交互问题。

【例题 3】原子链[1]

有 N 个原子，编号为 1，2，3，…，N。把它们排成一行，相邻两个原子的距离为 1，如图 1-13 所示。

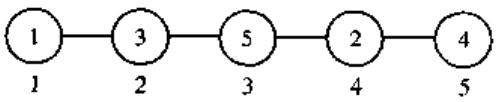


图 1-13　原子链示意图

可是，在一般情况下，很难一下子测定整个原子链的详细情况，而只能一次测定两个指定编号的原子的距离。请编程用尽量少的测定次数确定整个原子链的情况。如果每个原子最多被测量过四次，你可以得到一半的分数；如果每个原子最多只被测量过三次，你的程序可以得到满分。

【分析】

这是一道交互式题目，需要收集和整理信息，所以关键在于询问方式的设计。

为方便叙述，用 D(x,y)表示原子 x 和 y 之间的距离。

如果已知 D(1,2) = a，那么想要确定原子 3 与它们的位置关系必须询问 D(1,3)和 D(2,3)。也就是说除了原子 1、原子 2，每确定一个原子需要询问 2 次，为确定整条原子链，共需询问 1+(n−2) ×2 次。平均每个原子被询问 4−6/n 次，显然不符合要求。

所以，我们打算将原子不是一个一个，而是两两插进原子链内。假设已知 D(1,2) = a，

[1] 题目来源：CEOI 2001, Chain

经过询问又知 D(1,3) = b，D(2,4) = c，D(3,4) = d。列表 1-6 进行分析。

表 1-6　原子位置关系和需要满足的条件

	原子位置关系	满 足 条 件
情况 1：3 在 1 的左边，4 在 1 的左边	4−3−1−2	$a+b-c+d=0$
情况 2：3 在 1 的左边，4 在 1 的右边	3−4−1−2	$a+b-c-d=0$
	3−1−4−2	
	3−1−2−4	$a+b+c-d=0$
情况 3：3 在 1 的右边，4 在 1 的左边	4−1−3−2	$a-b-c+d=0$
	1−4−3−2	
	4−1−2−3	
	1−4−2−3	
	1−2−4−3	$a-b+c+d=0$
情况 4：3 在 1 的右边，4 在 1 的右边	1−3−4−2	$a-b-c-d=0$
	1−3−2−4	$a-b+c-d=0$
	1−2−3−4	

单从“满足条件”上看，没有任何一个条件同时属于两种情况，似乎可以推算出原子 3、4 在原子链中的位置了。但还有一个问题，就是有没有可能（a,b,c,d）同时满足属于不同情况的两个条件呢？再次列表分析发现，真正有可能同时成立的只有：

情况 2 的中的 $a+b-c-d=0$ 和情况 4 的中的 $a-b+c-d=0$。

要想同时成立，必然满足 $b=c$ 和 $a=d$。如果可以保证原子链中始终存在 3 个被询问次数小于 3 的原子 x,y,z，那么，可以先询问 D(3,4)，然后从 x,y,z 中选出两个原子（假设是 x,y），使得 a = D(x,y) $\neq$ d，从而避免上述情况的出现。

怎样保证原子链中一定存在 3 个被询问次数小于 3 的原子呢？先开始，询问 D(1,2)、D(1,3)和 D(2,3)，确定它们的位置关系。此时，原子 1、2、3 都只被询问过 2 次。以后，每次往原子链中增添 2 个原子，原子 x,y 都将被询问第 3 次，剩余原子 z 仍只被询问 2 次，加上新增原子也是只被询问 2 次的，因而使得原子链中仍然存在 3 个被询问次数小于 3 次的原子。

还有个小问题，就是如果 n mod 2 = 0，那么询问到最后将剩余单独的原子 n，此时我们询问 D(x,n)和 D(y,n)确定 n 的位置。

练　习　题

思考题：

1.2.19　递归和递推是两种互逆的方法吗？请结合此两小节的例题说明。

1.2.20　递归法的核心思想是什么？从算法上讲（不考虑程序实现），它的优势是什么？

1.2.21　递推法的核心思想是什么？从算法上讲(不考虑程序实现)，它的优势是什么？

***1.2.22**　考虑二者都无法单独解决问题时，把枚举法作为辅助算法分别和它们结合使用。递归法中可以枚举什么？递推法中又可以枚举什么？试对二者进行分析和比较。

编程题：

1.2.23　整数对[1]

考虑一个数 A（首位不为 0），把它去掉一个数字以后得到另外一个数 B。给出 A 与 B 的和 $N(1\leqslant N\leqslant 10^9)$，编程计算出所有可能的数 A。例如，若 $A+B=34$，则 A 可能是 31 或者 27。

1.2.24　寻找魔法豌豆[2]

小仙女 Cinderella 有 n（$n\leqslant 10\,000$）个袋子，每个袋子里有无限的豌豆。$n-1$ 个袋子里的豌豆是正常的，每粒重一克；有一个特殊袋子里的豌豆是魔法豌豆，每个重两克。每次 Cinderella 可以用每个袋子里取出若干豌豆，然后放在一起称称有多重。由于她的秤坏了，不能正确的称出大于 1 717 克的东西，所以即使 Cinderella 选出的豌豆总重大于 1 717 克，称量的结果将仍然是 1 717 克（不过这并不会损坏秤）。请帮助她用 10 次称量找出装有魔法豌豆的袋子。

1.2.25　锁链[3]

Byteland 并不总是一个民主的国家，它的历史上也有过黑暗的年代，但美好的一天总会到来：Byteland 的军事统治者终于决定结束长期的战争，并释放关押着的反对派激进分子。但他并没有打算让反对派领导人 Bytesar 自由，而是决定用 Bytish 锁链把他锁在墙上。这种锁链由许多固定在墙上的铁环和铁棒组成，由于环不都是套在棒上，要想把整副锁链取下来是十分困难的。

“长官，为什么你要把我锁在监狱里，不让我出去庆祝自己自由的到来？”Bytesar 喊着。

“可是 Bytesar，你根本就没有被锁起来，我肯定你能自己把这些铁环取下来。”Byteland 长官狡猾地回答到，“不过必须在 1 点钟之前完成，并且不能发出一点噪音。这可是在晚上，否则我就不得不找警察来了。”

你的任务是帮助 Bytesar，铁环由 1，…，n（$n\leqslant 1\,000$）依次编号，取下或套上都必须按以下规则：

- 一次只能把一个环取下或套上；
- 编号为 1 的环无论何时都能取下或套上；
- 如果编号为 1，…，$k-1$（$1\leqslant k\leqslant n$）的环已经从棒上取下，并且 k 环套在棒上，那么我们就可以取下或套上编号为 $k+1$ 的环。

你需要写一个程序，读入锁链描述并计算出从棒上取下所有环所需的最少步数。

---

[1] 题目来源：ACM/ICPC Regional Contest NEERC 2001

[2] 题目来源：Internet Problem Solving Contest 2003, modified

[3] 题目来源：*Polish Olympiad in Informatics, 2001*

1.2.26　银行抢劫[1]

就在昨天晚上，一所银行遭到了抢劫，而劫匪至今还未被抓到。Robstop 检查官为此很生气，因为这已经是今年发生的第三次银行抢劫案了，虽然每次 Robstop 都下令封锁出城的所有通道，竭尽全力的阻止劫匪逃跑，但是总是检查官通知市民们注意劫匪的行踪，当他询问大家的时候，所有人的回答都一样："对不起，我什么也没看见。"

这次 Robstop 可受够了！他决定坐下来静静的分析一下劫匪是怎么逃走的。Robstop 所在的城市是一个 $W \times H$ 的长方形，所有出城的通道将在时间 t 以后被全部封锁。在这之前，检查官的高性能侦察装置记录下了一些这样的信息："在时间 t_i 的时候，抢劫犯不在矩形 R_i 中！"抢劫犯每个时间单位最多只能朝东、南、西、北中的某个方向移动一步。写一个程序，根据监视器提供的信息计算出抢劫犯在每个时刻的位置。

1.2.27　狂风刮进办公室[2]

安迪和他的助手们刚刚完成了一项伟大的软件工程。他们把资料整理好堆放在桌上，但风把资料吹乱了。安迪不得不重新整理。他给所有的纸张随机地编上不同的号码，称为安迪编号。资料由若干个程序组成，每个程序有惟一的原始号码。一个程序开始于某张纸，跨越随后的若干连续纸张，同一纸张上可能有多个程序。如果安迪编号为 x 的纸张上有原始号码为 y 的程序，那么安迪的助手会在黑板上写下数对(x,y)。

现在给你所有的数对，请你根据它们给出一种可能的狂风刮进办公室前的纸张顺序（用安迪编号来表示）。

***1.2.28**　时区[3]

你是一个常年往来于世界各地做生意的忙碌的商人。每天你会收到各个失去的客户发来的消息，恰好每个时区一个。糟糕的是，这些消息的末尾只注明了该消息发送时客户所在时区的时间，而不是你这里的时间。请根据这些消息收到的先后顺序，判断哪个消息属于哪个时区。重要提示：本题的输入数据保证消息和时区的对应关系可以惟一确定。假定一天有 n 小时，共有恰好 n 个时区。例如：当 n=5，且 5 个消息按照收到的先后顺序排序，发送时间分别为 00:17, 02:50, 04:00, 02:01, 00:02，则可以确定它们分别来自失去 3,1,0,2,4，即它们收到时你所在时区的时间为 03:17, 03:50, 04:00, 04:01, 04:02。

至此，我们对基本算法思想的介绍告一个段落。有兴趣的读者可以访问本书的网站来获得最新的题目和资料。本节的理论知识不多，但是对思维的锻炼却是极大的，建议读者反复推敲其中的内容（尤其是贪心和递推两部分），认真思考书中给出的练习题目。

1.3　数据结构（1）——入门

本节主要讲述一些基本数据结构，这是本书第一个理论和实际相结合的小节。初学者

[1] 题目来源：UVA Problem Archive
[2] 题目来源：Internet Problem Solving Contest, Windy Office
[3] 题目来源：Baltic Olympiad in Informatics, 2000, Time Zones

往往感到学起来很吃力，这不要紧，因为习惯和接受数据结构中蕴涵的思想本来就需要一个循序渐进的过程，建议初学者先阅读相关书籍再学习本节，而不要试图第一次就懂得本节的全部内容，这样不仅可以事半功倍，而且可以把注意力集中到本书中介绍的思维方法中，体会数据结构的核心思想和方法。

本节的内容都是用实际问题或者生动的故事引出的，这当然可以让读者更有兴趣，但更重要的是这些问题和故事的背后藏着很多思路和动机，值得读者细细品味。

最后，值得注意的是，由于很多理论内容许多书有很详细的介绍，这里只是稍微提了一下，并未加以深入而细致的分析。其中的一些内容，如 Huffman 编码、循环队列、链栈等，由于很多书中已有介绍，限于篇幅，这里也不再赘述。读者在遇到困难时不妨多参考其他书籍。

1.3.1　栈和队列

数据结构　前面说过，数据结构是由某一数据对象及该对象中所有数据成员之间的关系组成。但是对本书而言，这个定义并不重要。读者可以通过阅读参考书籍得到严格的定义，而本书则着重通过一系列的例子来让读者慢慢体会数据结构的含义和精髓。

线性结构　数据结构中，有一类比较简单，称为线性结构。它们的特点是简单元素有序的排在一起，虽然比较简单，但是有时候用处很大。

线性表（list）　是由叫元素(element)的数据项组成的有限的有序序列。有序即每个元素都有自己的位置，都有确定的前驱（除了第一个元素）和后继（除了最后一个元素）。既然组成一个有序序列，未加说明的情况，我们认为每个元素的类型是相同的。

线性表的操作　线性表的操作主要有三种：插入、删除、查找。为了实现这三种操作，有两种方法：**顺序表**（array-based list）和**链表**（linked list）。这不是本节的重点，大家可以阅读[5]的 2.1、2.2 两节。下面只讲四点。

- 顺序表在物理上是一个连续的内存区域，数组的位置与元素相对应，这说明顺序表是适合**随机访问**的。在顺序表的最后加入或删除一个元素的复杂度为 $O(1)$，但是在中间插入一个元素可能会涉及到很多元素的移动，在最坏情况下会移动 n 次（n 为元素个数）。顺序表需要预先分配空间，所以适合于元素个数不作较大变动的场合。
- 在链表中，每个元素的位置是随意的，但是为了能**遍历**整个表，每个元素除了包含它本身数据，还应该包含它后继元素的位置。如果元素还包含它前驱元素的位置，那么我们还可以进行反向遍历。链表的随机访问是比较麻烦的，我们不得不从第一个元素开始沿着链表一个个找到所需要的元素。链表的另一个缺点是有附加的空间占用。
- 顺序表一般用数组来实现，链表一般用指针动态分配内存来实现。它们的应用很广泛，这里就不举例子了，大家可以参考相关书籍。

小知识——顺序表和链表的简单改进

顺序表和链表的缺点是可以改进的，例如，在顺序表的删除操作中，如果连续进行很多次删除，可以先建立删除标志，等删除元素比较多以后再一次性去掉所有删除过的数据。在链表的随机访问中，也可以通过记录中间一些元素的位置，每次从最近的一个元素开始遍历，不过在插入和删除的时候，中间元素的位置需要额外的时间进行维护。

我们还可以把顺序表和链表结合起来，例如线性表的每一个元素是一个链表，这样插入、删除、随机访问的时间可以比较平衡。在后面的HASH表中我们可以看到这样的例子。

两种特殊的线性表　在很多实际问题中，虽然各个元素形成了一个线性表，但是操作却比较特殊、仅在表的两端进行。我们并不关心任意时刻表的内容，而关心插入和删除的元素。队列和栈是线性表限制操作位置后的产物，应用十分广泛。队列的特点是**先入队的元素先出队（First In First Out, FIFO）**，在实际中体现出公平原则，可以利用它来暂存没有来得及处理而但需要按一定顺序依次处理的元素；栈的特点是**先入队后出队（Last In First Out, LIFO）**，体现出每个元素底部的元素对它来说是透明的，即这些元素的个数对它出栈的时机没有任何影响，可以利用这一点来进行具有递归结构的许多操作。

1．栈及其应用

铁　　轨[①]

某城市有一个火车站，其中的铁路如图1-14所示：

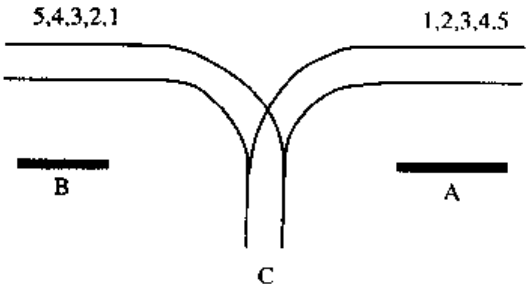


图1-14　火车站铁轨

每辆火车都从A方向驶入车站，再从B方向驶出车站，同时它的车厢可以进行某种形式的重新组合。假设从A方向驶来的火车有n节车厢（$n \leqslant 1\,000$），分别按顺序编号为1，2，…，n。假定在进入车站之前每节车厢之间都是不连着的，并且它们可以自行移动，直到处在B方向的铁轨上。另外假定车站C里可以停放任意多节的车厢。但是一旦当一节车厢进入车站C，它就不能再回到A方向的铁轨上了，并且一旦当它进入B方向的铁轨后，它就不能再回到车站C。

负责车厢调度的工作人员需要知道能否使它以a_1，a_2，…，a_n的顺序从B方向驶出。请写一个程序，用来判断能否得到指定的车厢顺序。

① 题目来源：UVA Problem Archive

栈　在解决这个问题之前，先考虑下面的情形：依次把 n 个大小不等的盘子放在桌子上，先放最大的，最后放最小的，然后再依次把表面的盘子拿走，那么不难看出，最先拿走的是最小的盘子，最后才能拿到最大的。在这里，插入和删除都只能在表的一边（盘子的顶部）进行，我们把这样的线性表叫做**栈（stack）**。可操作端叫**栈**顶。

栈的实现　一般用顺序表 stack 和栈顶指针 top 来实现栈。

栈的基本操作　栈有两个基本操作：**入栈（push）**，`stack[top++] = item;` **出栈（pop）**，`item = stack[--top];` 需要注意的是，插入的时候需要判断栈是否已满，而删除的时候需要判断栈是否为空。

WEB　虽然栈并不是一种复杂的"高级"数据结构，但是有时候巧妙地应用它，能设计出易于实现、时间复杂度低的算法。一个优美的例子是计算带符号排列（signed permutation）的覆盖图（overlap graph）的连通分量的线性算法（该算法是计算两个带符号排列的反向距离（inversion distance）的瓶颈）。相关资料可以在本书网页上找到。

回到题目中来，分析一下铁轨的特点，很容易发现火车的动作是符合"后进先出"特点的，所以本题的模型是一个栈。可以通过简单模拟来判断火车是否能够达到目标状态。

在火车行驶中，用 x 表示 A 站最前方车厢的编号（如果 A 站已无火车，那么 $x=0$），用 y 表示依照出站序列 a_1，a_2，…，a_n，下一列需要进入 B 站的车厢的序号（如果火车已全部驶入 B 站，那么 $y=0$），用数组 d 模拟栈记录当前车站 C 中的车厢。初始时，$x=1$，$y=a_1$，$d=[]$。在行驶过程中：

如果 $y=0$，那么火车调度成功，退出模拟过程；否则
　如果 $x=y$，那么让车厢 x 直接从 A 开进 B；否则
　　如果 $d\neq[]$且 $y=$（最后进入 C 的火车编号），那么让 y 从 C 开出进入 B；否则
　　　如果 $x\neq0$，那么让车厢 x 驶入 C；否则
　　　　输出无解。

小知识——上下文无关文法、下推自动机

考虑这样一个问题：由左右括号"（"、"）"组成一个表达式，编程判断它是否合法。例如()()和(())是合法的，())(就不合法　熟悉栈的读者可以很容易写出一个基于栈扫描的线性时间算法，但是为了把研究深化，先把这里的"合法表达式"进行形式化描述。即：

（1）空串是合法的。

（2）如果 A 是合法表达式，则(A)是合法的。

（3）如果 A 和 B 都是合法表达式，则 AB 是合法的。

这就是我们"上下文无关文法"（Content-Free Grammar, CFG）描述。当然，这不是正规描述，但是相信读者已经体会到了其中的思想：递归生成规则。上下文无关文法所定义的语言是上下文无关语言（CFL），判断一个串是否为某一个给定的上下文无关语言中的串有 $O(n^3)$ 的算法。有一些特殊的 CFL 可以通过构造一种称为确定下推自动机（Deterministic Pushdown Automata, DPDA）的结构来判定，从而得到线性时间复杂度的判

定算法。这就是栈扫描算法的理论基础。

2. 队列及其应用

小球钟——时间与运动[①]

时间是运动的一种方式，所以常常用运动来度量时间。如图 1-15 所示的小球钟就是一个通过不断在轨道上移动小球来度量时间的简单设备。

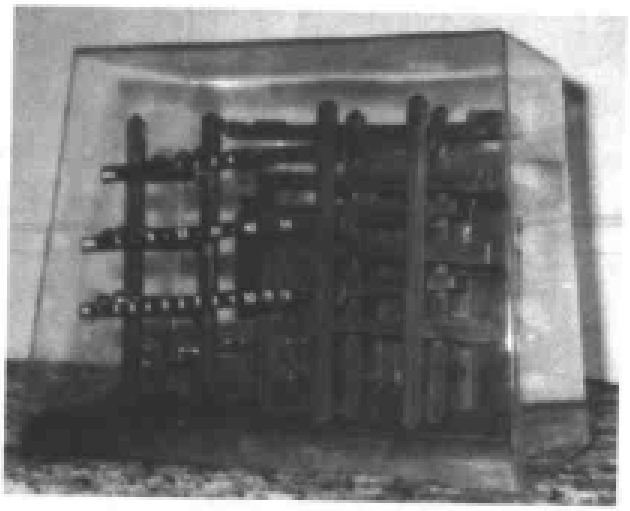


图 1-15　小球钟

每分钟，一个转动臂将一个小球从小球队列的底部挤走，并将它上升到钟的顶部并将它安置在一个表示分钟，5 分钟和小时的轨道上。这里可以显示从 1:00 到 12:59 范围内的时间，但无法表示“a.m.”和“p.m.”。若有 2 个球在分钟轨道，6 个球在 5 分钟轨道及 5 个球在小时轨道上，就显示时间 5:32。

不幸的是，大多数市场上提供的小球钟无法显示日期，尽管只需要简单地加上一些轨道就可以了。当小球通过钟的机械装置被移动后，它们就会改变其初始次序。仔细研究它们随着时间的流逝发生的次序的改变，可以发现相同的次序会不断出现。由于小球的初始次序最后迟早会被重复，所以这段时间的长短是可以被度量的，这完全取决于所提供小球的总数。

每分钟，最近最少使用的那个小球从位于球钟底部的小球队列被移走，并将上升安置于显示分钟的轨道上，这里可以放置 4 个小球。当第 5 个小球滚入该轨道，它们的重量使得轨道倾斜，原先在轨道上的 4 个小球按照与它们原先滚入轨道相反加入到钟底部的小球队列，引起倾斜的第 5 个小球滚入显示 5 分钟的轨道，该轨道可以放置 11 个球。当第 12 个小球滚入该轨道，它们的重量使得轨道倾斜，原先 11 个小球同样以相反的次序加入钟底部的小球队列，而这第 12 个小球滚入了显示小时的轨道。该轨道同样可以放置 11 个球，但这里有一个外加的固定不能被移动的小球，这样小时的值域就变为 1 到 12。从 5 分钟轨道滚入的第 12 个小球将使小时轨道倾斜，这 11 个球同样以相反的次序加入钟底部的小球队列，然后第 12 个小球同样加入钟底部的小球队列。

① 题目来源：ACM/ICPC World Finals

输入小球的个数，输出该时钟在经过多少天的运行可以回到它的初始小球序列。例如有 45 个小球的钟经过 378 天会回到初始状态。

队列　和栈一样，先考虑下面的情形：在排队的时候，所有人构成了一个线性表。插入只在队尾进行，而删除只在队头进行，我们把这样的线性表叫做**队列（queue）**。插入端叫队尾，删除端叫队头。

队列的实现　一般用顺序表 queue、队首指针 front 和队尾指针 rear 来实现队列。它有两个基本操作。

队列的基本操作　基本操作有两个：**入队（enqueue）**：`queue[++rear]=item;`　**出队（dequeue）**：`queue[++front] = item;` 也就是说，**没有移动元素而是移动指针**。这样做的好处是减少指令执行时间，但大家可能已经注意到了，在出列元素很多的时候，表前面的位置已经浪费了。

环行队列　采用环行队列可以减少空间浪费。首先需要规定一个队列长度 size，这样，入队代码变为了：`rear = (rear+1) % size; queue[rear] = item;` 而出队代码是 `front = (front+1) % size; queue[front] = item;` 需要注意的是，表前面的元素并非所有时候都无用。在 1.6 节中，将看到它们的作用。

链队　虽然不常用，但有时也用链表来实现队列。由于二者的插入和删除肯定在表的两端，所以仅仅在长度不定的时候，才考虑使用链表。但由于链表实现相对比较麻烦，而指针的大量使用会降低速度和浪费空间，一般是估计栈和队列的最大长度，仍用顺序表实现。

回到刚才的问题。为了得到问题的解，需要模拟小球钟的运作。学习了队列以后，我们不难发现分钟、5 分钟和小时轨道符合“后进先出”的特点，是栈；而球钟底部的小球符合“先进先出”特点，它们构成了一个队列。栈和队列都操作都比较简单，题目似乎已经解决了，只需要逐分钟地模拟小球钟的运作，直至钟底部的小球队列重又回到初始状态为止，这期间流逝的天数即为小球钟的运作周期。但需要注意的是，这样全部模拟耗时太大，必须改进。

刚才是所有小球一起模拟的，能不能把它们分开，先求出每个小球回到原来位置上的周期，然后求所有小球周期的最小公倍数呢？可以的。先将小球编号 1，2，…，n，然后模拟小球钟最先 24 小时的运行情况，得到一天后钟底部的新小球队列 P_1，P_2，…，P_n。这样，实际上是在两次的钟底部小球队列间建立起一种置换 $F(1, 2, \cdots, n)=(P_1, P_2, \cdots, P_n)$。有了这个条件，对于当前处在位置 x 的小球，无论小球编号多少，都知道它再过 24 小时应当处于位置 P_x。这样，在求得以上置换的基础上，可以求出每一个小球回到原位置的周期，然后求所有小球周期的最小公倍数即可。

小知识——栈和队列的应用

栈是先进先出，队列是后进先出。这一点区别让它们的应用场合截然不同。

栈可以用来保护现场。把栈 S 看成是由底部元素和它上面所有元素构成的子栈 S_0 所组成的，则我们总是先处理 S_0。处理 S_0 时，在 S_0 之前进入 S 的元素并不会受到 S_0 的影响，

也不会影响 S_0。这样的特性和前面讲到的“递归”是很相似的。事实上，程序设计语言就是用的“栈”来实现的递归。

队可以体现元素的先后顺序。用这一点可以用来进行模拟。在后面介绍的 BFS 中，用队列来保证先访问所有 n 层结点，再访问 $n+1$ 层结点。

另外，从本意上讲，队列应该是先进先出的，但是我们借用这个名字来给很多其他数据结构命名。最常见的是**优先队列（priority queue）**。在这种队列中，入队的顺序是无关的，每次都选一个优先级最小的元素出队。“优先队列”这个概念本身不涉及到数据结构的具体实现，其中最常见的实现是后面即将介绍的“堆”。

3. 种子填充法

笑　脸[①]

Ace 女王最近颁布了一条新法律，禁止在 Wondernet 上发送笑脸。但是女王担心人们还是会冒着被砍头的危险对这条法律置之不理，尤其是 Alice。女王派警察秘密截获了 Alice 的所有 E-mail 后，叫他们帮忙数数这些 E-mail 里有多少个笑脸，即 Alice 应该被砍头多少次。

一张脸包含一些元素（Face Element, FE）：两只眼睛，一个鼻子，一张嘴巴和一些诸如眉毛等的其他东西。一个 FE 是一片八连的“1”。每张脸至少包含一个封闭的脸边界，两只眼睛和一张嘴，其他 FE 都是可以省略的。

脸边界把其他 FE 包围在里面，而两只眼睛各包围一个由 0 组成的四连区域（这个区域可以包含其他 FE）。嘴可以包围一个由 0 组成的四连区域，也可以不包围，但是其他 FE 都不能包围这样的区域。嘴总在眼睛的下面（即：嘴最下方的 1 在眼睛最下方的 1 的下面），而在所有 FE 中，嘴是除了边界之外左-右跨度最大的一个。

如果一张嘴最上方的 1 同时也是嘴的最左边或者最右边的 1，我们就说这是张笑脸。给出 Alice 那封包含很多脸的 E- mail，请数一数里面包含了多少笑脸。如图 1-16 所示，左边是笑脸，右边是非笑脸。

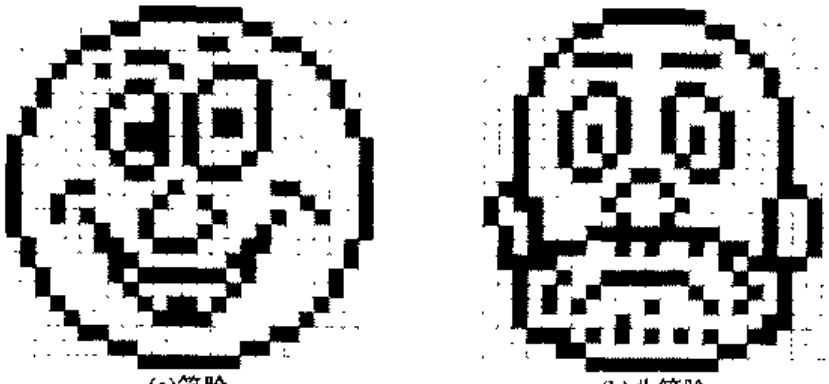


图 1-16　笑脸和非笑脸示例

要找出笑脸，首先要找出脸，再加以判断。为此，先介绍**种子填充法（floodfill）**。填充过程就像是一滴墨水滴在了纸上。开始只是一个小点，然后慢慢的扩散开来，最后填满

[①] 题目来源：Internet Problem Solving Contest 2001

一个封闭区域。借助队列的知识，很容易得到下面的 floodfill 算法：

floodfill 算法　建立一个队列 Q。初始时，Q 仅包含最开始的那个“点”，对于每个出队的元素，把它周围需要扩散到的元素加入队列中。为了避免重复访问，需要记录每个元素是否已经访问过，即访问标志。这个算法的时间复杂度为 $O(m)$，其中 m 是需要扩散到的元素总数；附加空间为 $O(m)$，这是它访问标志所用空间。如果访问标志可以和原始数据共用同一块内存区域，则附加空间为 $O(1)$。

floodfill 的应用　Floodfill 是一种很常用的预处理方法。当“连在一起很大一块东西可以一并处理”的时候，可以做一次 floodfill，分离出一个个“连在一起”的块，然后再做处理。在本题中“连在一起”的点就是一个 FE。

本题中关于“笑”的定义太复杂，所以检查官决定只用程序找出所有不同的脸，再用人眼判断哪些在笑。

为了方便起见，这里假设左上角不属于脸的一部分。首先，从 E-mail 的左上角开始做四连（想一想，为什么不是八连？）的 floodfill，则填充到的格子是脸的间隙，没有填充到的才是脸本身。然后，逐行逐个扫描各个像素，一旦碰见某个 1 像素，一张脸就找到了。用宽度优先队列找出所有与此像素 8 连通的 1 像素，标记它们为已检查状态，这些 1 像素就构成了脸的外轮廓。把外轮廓放入恰好包含它的矩阵中，逐行扫描，将处在外轮廓内的像素从 E-mail 中如实翻版过来，同样标记它们为已检查状态，并把外轮廓外的所有像素设为 0 像素。

怎么判断此脸是否为新类型的脸呢？将它逐一和所有已找到的旧类型的脸比较，判断存储此脸的矩阵是否和存储旧脸的矩阵大小、元素完全相同。如果相同，在旧脸的计数器上加 1；如果不同，存储此矩阵为新类型的脸，并初始化计数器为 1。

最后，用人眼观察，把所有笑脸的计数器内的数累加，结果即是所求答案。当然，有兴趣的读者也可以试着写一个完整的程序，自动判断每个脸是否为笑脸，这里不再叙述。

WEB floodfill 是计算机图形学，机器人视觉等领域的基本方法，很多算法都是在它基础之上完成的。相关资料可以在本书网页上找到。

练　习　题

思考题：

1.3.1　“铁轨”一题中，为什么给出的算法能正确地判断出一个出栈顺序是否可能？

1.3.2　栈空通常意味着什么？请举例说明。

1.3.3　队空通常意味着什么？请举例说明。

1.3.3　如果需要把所有已经出队元素保存起来，应该如何做？

1.3.4　floodfill 可以用栈来做吗？如果不行，说明理由；如果可以，试和基于队列的 floodfill 比较。哪种方法的时间耗费大？空间需求呢？

1.3.5　用两个队列实现一个栈；用两个栈实现一个队列。分析你的算法的时间复杂度。

编程题：

1.3.6　位图的块[①]

给一个 $n(n \leqslant 1\,000)$行 $m(m \leqslant 10\,000)$列位图，计算黑色连通块的个数和其中不包含“洞”的连通块的个数。“洞”是指包含在一个连通块中的一条闭折线中的白色像素。例如图 1-17 的左图的 x 格不是一个洞，因为连通块内找不到一条必折线可以包含它。图 1-17 的右图有 5 个块，其中 4 和 5 是不含洞的。图 1-17 描述了块的含义并列举了一个位图例子。

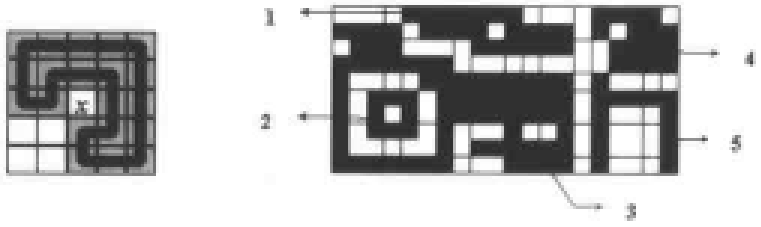


图 1-17　块的含义和位图举例

1.3.7　矩形分划[②]

矩形分划是指把一个矩形分成若干不重叠的小矩形。图 1-18 是三个合法的矩形分化。

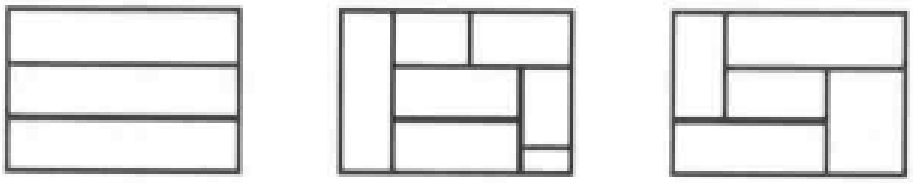


图 1-18　合法分划

在图 1-19 中，B 是由 A 继续分划形成的，我们称 B 比 A 精细，A 比 B 粗糙。精细和粗糙都偏序关系，例如 C 不比 A 或 B 中的任一个精细或者粗糙。

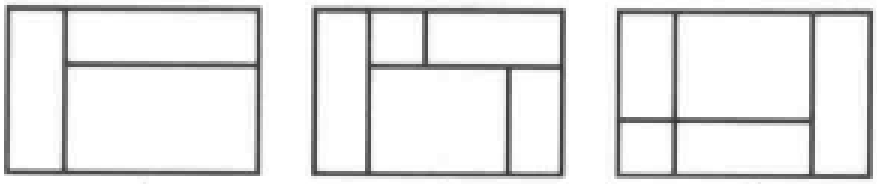


图 1-19　三个分划

给定同一矩形的两个分划 D 和 E，有无限多种分划比 D 和 E 都精细。例如下图，F 和 G 都同时比 D 和 E 精细。在这无限多个分划中，有惟一一个分划比其他分划都粗糙，我们称它为 D 和 E 的下确界。图 1-20 所示中，F 是 D 和 E 的下确界。

---

① 题目来源：CEOI2001, Bitmap

② 题目来源：ACM/ICPC World Finals 2002, Partition

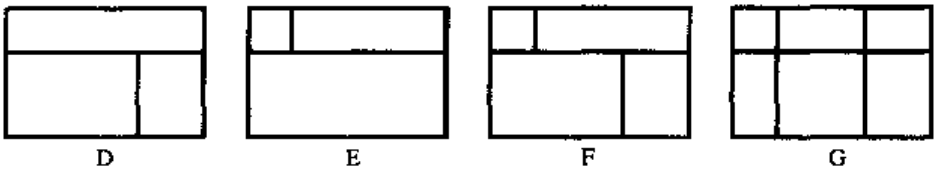


图 1-20　下确界的概念

类似的，在图 1-21 中，H 和 J 都比 D 和 E 粗糙，但是 J 比 D 和 E 都粗糙的分划中是最精细的，所以 J 为 D 和 E 的上确界。

图 1-21　上确界的概念

给同一矩形的两个分划，求它们的下确界和上确界。

1.3.8　程序复杂度[①]

TC（Too Complex）软件公司的主要业务是为智能机器人编写程序。该公司拥有一些很有才华的程序员，但是，他们在编写程序的时候，却常常出现一些低级的失误，而导致程序的效率下降。比如下面的一个 pascal 程序：

```
for i:=1 to n do
  for j:=1 to n do begin
    a[j]:=a[j div 2];
    b[i]:=b[i]+a[i];
  end;
```

上述程序段的渐进时间复杂度为 $O(n^2)$，但实际上，内层循环与外层循环之间并无嵌套的必要，完全可以将其改写成如下形式：

```
for j:=1 to n do a[j]:=a[j div 2];
for i:=1 to n do b[i]:=b[i]+a[i];
```

这时，很容易看出，程序最佳的书写形式的复杂度应当是 $O(n)$。

由于不佳的程序书写，运行 TC 公司程序的机器人反应速度出奇地慢，这当中的很多机器人已经决定购买别的公司的程序了。当然，这很令 TC 地老板伤脑筋，所以，他雇佣了你，要求你编写一个程序检验某个 TC 程序员的程序是否以最佳的形式写出的。

具体地说，你的程序应当读入另一个源程序，求出该程序现在的书写形式所对应的时间复杂度多项式（称为原始时间复杂度多项式），以及最优形式的时间复杂度（最优时间

[1] 题目来源：IOI99 中国国家集训队原创试题. 命题人：齐鑫

复杂度多项式）。

TC 程序员的源程序代码可以形式化的定义如下：

程序　　::=“begin”{语句}“end”
语句　　::= 循环语句 | 简单语句 | 控制语句
循环语句::=“loop”变量 循环次数 {语句}“end”
简单语句::=“op”{变量}
控制语句::= break | continue
循环次数::= n | 正整数
正整数　::= 1 | 2 | … | 32767
变量　　::= *a* | *b* | … | *z*

说明：

- 各标识符之间用不少于一个的空格隔开，不区分大小写；
- 嵌套的两个循环不能共用同一个循环变量；
- 你的程序中所有的计算数字范围不必超过 2×10^9；
- 只计算一般语句的执行时间，每个一般语句的执行时间都视为 1；
- 一般语句中出现的循环变量，是该语句的操作过程中所有需要用到的变量；
- 循环变量不会在循环体外被引用；
- 控制语句中 break 的作用是跳出当前循环体，continue 的作用是跳过当前循环体中的剩余语句，忽略不在任何循环体中的控制语句。

程序的最佳书写形式，是指在不改变程序语义的基础上，将不必置于内层循环体内部的语句（循环语句或一般语句），调整到较外层循环体内或最外层循环体之外，所得到的程序。

1.3.2 串

有一种特殊的线性表称为**串（string）**，它被看作是一个符号序列。例如“algorithm”就是一个串，它有多个字符串在一起构成。当然，只含一个字符甚至含零个字符的也看成串。例如，任意说的一句话就是一个串。串中所有字符的集合构成了**字母表（alphabet）**，通常用符号 Σ 来表示。和串有关的一些算法时间复杂度和 Σ 的大小有关。

猜猜我想说什么[①]

- “佳佳，我想对你说句话。”
- “说呀。”
- “不行，我要你猜。”
- “怎么猜？”

[①] 题目来源：IOI2002 Practice, String

- “这是一句英语。我把我这句话里的所有标点符号都去掉，大小换成小写，这样就得到了一个由26个小写字母组成的串。”

- “然后呢？”

- “你每次可以猜一个串，我会告诉你我的串里是否包含你的串。”

- “好。‘e’?” “有！” “‘b’呢？” “有！” “‘a’呢？” “没有！” “‘by’？” “有！” “‘ye’?” “有！”

- “我知道了，是‘bye’!”

- “猜对了！拜拜！”

佳佳愣了一下。其实，佳佳并没有真正猜出来，因为这个串也可能是“byebye”或者“bydefye”。还有好多种情况。如果不考虑这些单词是不是有意义的，那么还需要很多次询问才能保证猜到。怎样做，才能用尽量少的次数保证猜到一个串呢？

这道题目的意思很简单，但并不是那么容易回答的。题目信息太少了，我们甚至不知道这个串S的长度是多少！幸好这个串只由26种符号组成的（一般的，假设有$a=|\Sigma|$种字符），应该把这个作为突破口。

首先，应该把所有单个字符作为子串问一遍，找到S里包含的一个字符c。显然最多只需要问26遍就可以找到一个c了（因为串S不为空）。

现在，希望把c向右延伸，即看看有什么样的字符x_1，使得$S'=cx_1$在S里出现。如果有这样的x_1，那么我们实际上得到了S的一个长度为2的子串。只要一直能找到这样的x，总可以把c向右延伸得到串$S'=cx_1x_2\cdots x_k$，直到不能再延伸，每次得到的串S'都是S的子串。读者也许已经发现了，S'不能向右延伸意味着S'是串S的后缀！这时只需要用类似的方法再向左延伸S'，直到不能向左延伸，这时候的S'已经扩展为了S本身了。由于每次最多试验a个字符，而第一个和最后一个字符试验了两次，其他字符各只试验了一次，因此总的试验次数不超过$a(n+2)$。

WEB 串的问题有很多，包括求串的最长重复子串，寻找无重复子串的无限长序列，本题的更好方法……，而涉及到的数据结构包括后缀树等，内容博大精深。本书的主页有一些进一步的阅读资料，欢迎有兴趣的读者访问。

模式匹配问题　从刚才的问题中看到，其实单从数据组织的形态来看，串并没有什么很特殊的地方，但是和其他线性结构不一样，一般把一个串看成一个整体，在实际应用中，串有一种特殊的操作叫做“模式匹配”，这是其他线性结构一般不进行的操作。也就是：对于两个串S_1和S_2（S_1叫做**主串**（**text**），S_2叫做**模板串**（**pattern**）），问S_1中是否包含串S_2？若是，给出它的位置。例如，S_2=“computer”，S_1=“I'm studying computer science.”，那么S_2包含S_1，它在S_1的第14个字符处首次出现。一个很自然的问题是：如何高效地做模式匹配？

朴素的模式匹配算法　把对于主串中的每个位置i，检测从它开始的串是否以模板串为前缀。假设主串的长度为n，模板串的长度为m（未加说明的情况下，以后的n和m都是

此含义），则这个方法需要试验 $O(n)$次，每次最多进行 $O(m)$次字符比较，因此方法的总时间复杂度为 $O(nm)$。

需要特别注意的是：进一步的分析表明，如果主串和模板串都是随机选择的，字母表的大小为 d，则这个朴素匹配算法的比较次数为：$(n-m+1)\frac{1-d^{-m}}{1-d^{-1}} \leqslant 2$（$n-m+1$），是线性的。

Knuth-Morris-Pratt(KMP)算法　可以看出，朴素的模式匹配算法很简单，但是做了很多无用功。假如 S_1=“tomatogoodtomatobad”，S_2=“tomatoisbad”，那么在前 7 个字符时第一次匹配失败。按照方法一，会立刻去试验 S_1 的第二个字符。其实，根据 S_2 的特点，既然前 6 个字符都匹配成功了，说明前 6 个字符恰好为“tomato”，由于以“mato”,“atoi”，开始的串不可能以 S_2 为前缀，所以没有必要从它们开始检测。而以“to”开始的串有可能以 S_2 为前缀，因此下一次应该直接检测 S_1 的第 5 个字符。

前缀函数　也就是说，如果在匹配到 S_2 的第 i 个元素的时候失败，那么通过刚才的方法计算出一个前缀函数值 pre[i]，然后把 S_1 的当前指针往后移动 pre[i]个位置，S_2 的当前指针不变。这个方法就叫 KMP（Knuth-Morris-Pratt）算法，它在几乎所有数据结构书籍中均有介绍，这里略去。需要强调的是，pre 函数的计算实际上是自己匹配自己。在匹配的过程中利用了已经算出的 pre 值来计算新的 pre 值。

Rabin-Karp 算法　这个算法没有 KMP 算法出名，应用也不太受到重视（因为最坏情况下的时间是 $O(nm)$的），但是这里还是需要稍费笔墨把它介绍一下。假设字母表为{0,1,2,3,4,5,6,7,8,9}（读者很容易把下面的结论推广到其他形式的字母表），那么一个长度为 m 的串（例如模板串）就可以看作是一个 m 位的十进制数。用 $P[i]$表示模板串的第 i 个字符，$T[i]$表示主串的第 i 个字符，p 表示模板串，t_s 表示串中从第 $s+1$ 个字符开始，$s+m$ 个字符结束的长度为 m 的串，则模式匹配问题转换为：是否存在一个 $i(0 \leqslant s \leqslant n-m)$，使得 $p = t_i$。在不考虑 p 代表的数可能很大的情况下，可以把 t_0，t_1，…，t_{n-m} 都计算出来，然后用 p 和它们一一比较，只需要进行 $n-m+1$ 次。

现在遗留下两个问题。

问题一：如何计算 p 和 t_i？对于 p，可以用 Horner 规则：

$$p = P[m] + 10\,(P[m-1] + 10(P[m-2] + \cdots + 10(P[2] + 10P[1]))).$$

t_0可以用同样的方法计算，时间复杂度为$\Theta(m)$。接下来，可以用递推式计算剩下的：

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]$$

只要注意到 t_s 和 t_{s+1} 的表达式有很多公共部分，读者不难自己证明此式。其中 $h = 10^{m-1}$ 需要在预处理中计算出来（可以直接计算 $O(m)$，也可以用二进制展开 m，时间复杂度为 $O(\log m)$），这样，计算所有 t_i 的总时间复杂度为 $O(n+m)$。

问题二：p 和 t_i 可能很大，不仅储存不方便，而且比较时间也不能看作常数。怎么办呢？我们可以用结果对一个很大的质数 q 取模。则预处理时 $h = 10^{m-1} \bmod q$，而 p 和 t_i 的表达式都应对 q 取模。比较可以在常数时间内完成了，但是如果发现 $p \equiv t_i \pmod q$，并不能保证 $p = t_i$，需要进一步验证。如何进一步验证呢？有如下几种方法：

方法一：直接比较 p 和 t_i 是否相等。这样每次验证需要 $O(p)$的时间。

方法二：对多个 q 取模。如果所有的 q 都是质数而且乘积大于 10^m-1，则由中国剩余定理，如果所有余数相等，那么 p 和 t_i 一定相等。

方法三：预先把 $p \equiv t_i(\text{mod } q)$的所有解算出来，设计出一个能更快区别出它们的算法。

显然，方法三最困难，而且当解很多时时间很慢，且很可能最好的“区别”算法仍然需要比较完整个 p 和 t_i。方法二看起来不错，但是由于取模和比较操作是在任何情况下都需要完成的，所以它延长了平均处理时间（读者不妨仔细算一算实际的工作量）。推荐方法一，它不仅简单，而且遇到很多次验证的情况毕竟很少。事实上，如果验证次数很少（近似为常数），而 q 又比 m 大，那么 RK 算法是线性的。

下面考虑有多个模板的情形。如果所有模板 $p_1,\cdots, p_k$ 的长度相同，均为 m。假设所有的 $p_i \bmod q$ 各不相同（一般可以通过试验不同的 q 来达到这个要求），则仍然只需要 $O(m)$的时间就计算出所有 $t_i \bmod q$。对于每个 t_i，可以用二分查找的办法计算出它可能和哪个模板匹配，如果 $t_i \equiv p_j(\text{mod } q)$，则验证 t_i 和 p_j 是否相同。平均的时间复杂度为 $O(m+n\log k)$。这里有两个问题：

问题一：如果各个模板长度不相同会怎样呢？

问题二：把 RK 算法推广的二维情形。模板是 $m*m$ 的矩阵，主串是 $n*n$ 的矩阵。模板的左上角可能在主串的任何位置出现，然后不能被翻转或者旋转。（提示：为了快速地递推出模板对应的“数”，应该先考虑一维情形，做一些预处理）

专题四——有限状态自动机

为了更进一步理解串匹配算法，我们介绍有限状态自动机（finite automata）。

有限状态自动机（DFA）：一个五元组$(Q, q_0, A, \Sigma, \delta)$，其中：$Q$ 为状态集；q_0 为起始状态；A 为终态集；Σ 为字母表；δ 为转移函数。用图 1-22 来描述一个自动机。

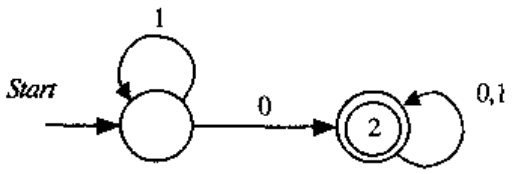


图 1-22　自动机举例

圆圈代表状态，弧代表状态转移。例如从状态 1 有一条标记为 0 的弧指向 2，说明从状态 1 输入字母 0 将会转移到状态 2。状态 1 前面有一个 start 标记，称它为初态；状态 2 上加了圆圈，我们说它是一个终态。终态可以有多个，但是初态只能有一个。

这里介绍的自动机是**确定性（deterministic）**的，因此从同一个状态出发，不会有两条标记有相同字母的弧指向不同的状态。

自动机把一个字符串作为输入，从初态开始，一个一个读字符串里的字母，根据每个字母来进行状态转移。记 $\varphi(s)$为把 s 中的字母一个一个读入并进行状态转移后最后到达的状态，如果 $\varphi(s)$属于终态集合 A，则自动机**接受（accept）**一个字符串 s，否则自动机**拒绝（reject）**该字符串。在刚才的自动机中，$\varphi(1101)$为状态 2(状态 1→状态 1→状态 2→状态

2)，因此 1101 将被接受。

字符串自动机（string automata）：对于一个模板串 P，可以构造一个有限状态自动机，使它接受的字符串集合为以 P 为后缀的字符串集合。

为了构造字符串自动机，先定义一个**后缀函数（suffix function）** $\sigma(x)$，其中 x 是一个字符串。它是最大的 k，满足条件(*)：P 的前 k 个字符是 x 的后缀（特殊情况：$k=0$ 时，空串 ε 是 x 的后缀）。这样，有：

（1）$\sigma(\varepsilon)=0$；

（2）$\sigma(x)=m$ 当且仅当 P 是 m 的后缀；

（3）如果 x 是 y 的后缀，那么 $\sigma(x)\leqslant\sigma(y)$（由定义和后缀关系的传递性容易证明）

显然，应该构造一个自动机，接受所有满足 $\sigma(x)=m$ 的串，思路是以 $\sigma(x)$的大小来划分状态，即状态集合为{0，1，2，3，…，m}，让 $\varphi(s)$等于状态 $\sigma(s)$。显然，惟一的终态为 m，因为这些串以模板 P 为后缀。

状态有了，下一步应该设计状态转移函数。假设主串已经输入了 i 个字符，到达了状态 q，即 $\sigma(T_i)=q$。再新输入一个字符 $i+1$ 以后，字符串 T 如图 1-23 所示（t_i表示主串的第 i 个字符）：

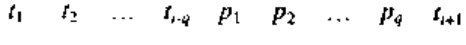


图 1-23　字符串分割示意图

由于 $\sigma(s)$是满足条件(*)的最大的 k，因此 T_{i+1} 满足条件的最长后缀的起始位置一定在 p_1 或者以后。这样，证明了 $\delta(T_i,a)=\sigma(P_q,a)$，其中 P_q 代表模板的前 q 个组成的前缀。

这样，余下的问题是：如何计算所有的状态转移函数，即 $\sigma(P_q,a)$。最笨的方法是，枚举所有的 q 和字母 a（最多有 $\min\{|\Sigma|,m\}$种可能），每次计算一个 σ 函数值需要枚举最多 m 个 k 值，每次判断是不是前缀需要 $O(k)$的时间，因此计算一个 σ 函数值的时间复杂度为 $O(m^2)$。由于有 $m*\min\{|\Sigma|,m\}$个函数值需要计算，因此总的时间复杂度为 $O(m^3*\min\{|\Sigma|,m\})$。借用 KMP 算法思想（留给读者思考），可以把计算所有函数值的时间降到 $O(m|\Sigma|)$，这就是总的预处理时间。

构造出了自动机以后，匹配就是很简单的了：如果在某时刻到达了终态，那么这个位置一定是模板一次出现的结束位置，正好进行 n 次操作（状态转移）。

以下三个问题留给读者思考。

问题一：考虑两个模板串的情形，如何构造两个模板的自动机？你的自动机所包含的状态数目应当尽量少。

问题二：假设模式串中存在一个主串中没有的特殊字符*。而且规定*可以匹配任意长度的字符串，包括空串（就像 UNIX 中的*通配符一样）。设计一个自动机，要求匹配过程的时间复杂度为 $O(n)$。

问题三：借用 KMP 算法的前缀函数,在 $O(m|\Sigma|)$时间计算出所有转移函数值。（提示：先证明当 $q=m$ 或者 $P[q+1]\neq a$ 时，有 $\delta(q,a)=\delta(pre[q],a)$

WEB 一种更简单易懂的方法是先构造非确定自动机 NFA，然后用子集构造法构造等价的 DFA，最后用填表法进行 DFA 的最小化。关于自动机进一步的讨论和相关学习资料，请访问本书主页。

小知识——串匹配算法面面观

串匹配有很多方法。

值得一提的是，KMP 算法有一个推广，用它可以求出主串的每一个位置 i 出发所能匹配到的最大长度。另外，甚至还有只使用 $O(1)$附加空间的模式匹配算法，它的时间复杂度也是线性的。

串匹配的另一个研究方向是压缩文本的串匹配。现在已经有线性时间复杂度和无附加空间的算法来匹配经过 LZW 压缩的文本。

串的其他问题，例如**最小表示问题**（**m.s.p**）和最长重复子串问题也是很具有研究价值的。有兴趣的读者可以阅读相关资料。

另外需要提到的是，字符串算法一般除了需要分析时间渐进复杂度之外，往往需要给出具体的比较次数，例如比较 $2n$ 次就不如 $n+2$ 次好。

练　习　题

思考题：

1.3.9　给出 $n(n\leqslant 5)$个长度不超过 $L(L\leqslant 2\,000)$的字符串，求它们最长公共子串的长度。请给出一个 $O(nL^2)$的算法。

***1.3.10**　“猜猜我想说什么”一题中，每次猜测都按照随机顺序进行，求猜测次数的期望。

编程题：

****1.3.11**　**项链**[①]

Byteland 是由著名的珠宝匠 Byteman 制作的精美项链的产地。这些项链是通过用线把宝石穿在一起制成的，这些宝石被分成 26 种，用小写拉丁字母 a ~ z 表示（同种宝石没有区别）。Byteman 从不制造两串相同的项链，这是他们的荣誉和骄傲，因此他们保留有他们曾经制造的所有项链的描述。某些项链非常的长，这就是他们把这些描述用缩写形式表示的原因。所有的描述都是一些“片断”组成的序列，每个“片断”含有一些字母（表示项链的花纹），后面跟着一个表示这个花纹在这个项链中重复的次数的整数。例如，描述“abc 2 xyz 1 axc 3”表示项链 abcabcxyzaxcaxcaxc（这个项链是由“abc”重复 2 次，“xyz”重复 1 次，“axc”重复 3 次组成的）。由于无法确定项链的起始位置（项链可以被 arbitrarily 旋转），某些项链可能有多于一种的描述形式。比如上面描述的那个项链也可以被描述为 cabcxyzaxcaxcaxcab 或 xcaxcaxcabcabcxyza。

写一个程序，读入两个描述，判断它们是不是在描述同一个项链。片段最多有 1 000 个，每个片段最多重复 100 000 次，所有片段长度和不超过 10 000。

[①] 题目来源：Polish Olympiad in Informatics, 2001

**1.3.12　病毒[1]

二进制病毒审查委员会最近发现了如下规律：某些确定的二进制串是病毒的代码。如果某段代码中不存在任何一段病毒代码，那么我们就称这段代码是安全的。现在委员会已经找出了所有的病毒代码段，试问，是否存在一个无限长的安全的二进制代码。

例如，如果{011, 11, 00000}为病毒代码段，那么一个可能的无限长安全代码就是010101…。如果{01, 11, 000000}为病毒代码段，那么就不存在一个无限长的安全代码。

请写一个程序，读入病毒代码，判断是否存在一个无限长的安全代码。

1.3.3　树和二叉树

勇士 Ilya 的故事[2]

有这样一个传说。邪恶的 Idol 掳走了年轻的勇士 Alyosha Popovich，用铁链把他锁在一块魔法石上，让他不能动弹。由于沙皇正在四处征战，看来只能由 Alyosha 的好朋友 Ilya Murumetz 来援救他了。“我要独自从 Idol 手中救出我亲爱的朋友 Alyosha。如果他不幸遇难，我也不想独活。”说完之后，Ilya 骑上骏马，踏上了营救 Alyosha 的征途。

不久，Ilya 来到了一个岔路口的一块石头旁，不过不是那块魔法石。就像故事里经常出现的那样，石头上写着：“往左走，你的时间将会不够；往右走，你将一去不返。”Ilya 沮丧地坐了下来，眼前突然出现一只小鸟。小鸟静静地落在了他的身旁，轻声问他：“你怎么啦，我的好朋友？”Ilya 告诉了小鸟他的烦恼，小鸟对他说：“和童话故事中常常出现的一样，如果你往左走，每一小时后你会都再次见到一块同样的石头，直到你来到海边，Idol 想让你迷路，故意这样设置的。回家吧 Ilya，不要螳臂当车了。”小鸟说完后消失了，就像从来就没有出现过一样。Ilya 陷入了沉思。几分钟后，他再次坚定了自己的决心，骑上骏马继续前进。

一小时以后，Ilya 果然来到了一块同样的石头旁。可正当他再次一筹莫展的时候，他突然发现那块魔法石就在不远的地方！但是 Ilya 不能直接过去，因为那样的话，他将会葬身于沼泽之中。他同时看到大海离他也很近，他甚至可以数出来，从这里到海边只有 E 小时的路程，而从魔法石到海边有 D 小时的路程。他还回忆起小时候父亲对他讲过的话：离海边最近的任意一块石头到海边都只有 F 小时的路程。

Ilya 开始仔细观察离海最近的那些石头。他发现，这些石头可以用自然数 1，2，3，…，编号！如果他每次都往右走，他最终会到达石头 1；如果只有最后一次向左走，他会到达石头 2；如果他仅在倒数第二次向左走，其余路口向右走，他会到达石头 3！他还看到一个离自己最近的海边石头编号为 E_p，而其中一个离魔法石最近的海边石头编号为 D_p，如图 1-24 所示。

[1] 题目来源：Polish Olympiad in Informatics
[2] 题目来源：Ural State University Problem Archive

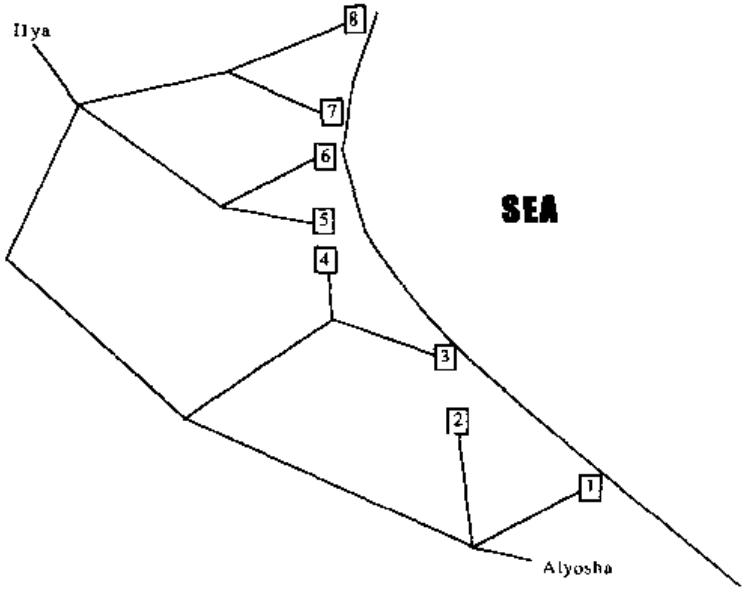


图 1-24　海边

当他计算出正确的路线，准备骑马飞奔的时候，骏马突然转过头来对他说："Ilyusah! 我现在已经非常疲惫，最多只能跑 H 小时的路了！"如果是这样的话，Ilya 究竟是能成功救出 Alyosha，还是要永远被困在这个迷宫式的荒岛上？

为了解决这个问题，需要了解二叉树的基本知识。虽然情况危急，但是磨刀不误砍柴工，还是先介绍一下后面将会使用到的知识吧，其他略去，读者可以参考任何一本数据结构书籍。

二叉树的定义　二叉树（binary tree）可以采用如下的递归定义：二叉树要么为空，要么由根结点（root），左子树（left subtree）和右子树（right subtre）组成。左子树和右子树分别是一棵二叉树。可以形象地把二叉树用图形表示出来，如图 1-25 所示（注意：和现实生活不同，计算机里的树是"倒置"的，根在上，叶子在下）。

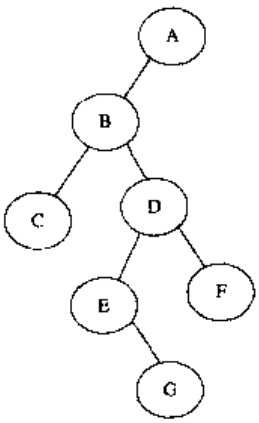


图 1-25　二叉树示例

二叉树的实现　在计算机中，二叉树通常是用指针实现的，每个结点包含数据域，左子树指针和右子树指针，需要源程序的读者请查阅数据结构相关教材，这里不再赘述。树还有其他表示方法，例如数组表示，或者增设一个父亲指针，在有些情况下这些方法会很方便。

完全二叉树　若设二叉树的高度为 h，则共有 $h+1$ 层。除第 h 层外，其他各层（0~$h-1$）的结点数都达到最大个数，第 h 层从右向左连续缺若干结点，这就是**完全二叉树**（**complete binary tree**）。

完全二叉树的数组实现　如果将一棵有 n 个结点的完全二叉树自顶向下，同一层自左向右连续给结点编号 0，1，2，…，$n-1$，然后按此结点编号将树中各结点顺序地存放于一个一维数组中，并简称编号为 i 的结点为结点 i $(0\leqslant i\leqslant n-1)$。则有以下关系：

（1）若 $i=0$，则 i 无双亲；若 $i>0$，则 i 的双亲为 $\lfloor (i-1)/2 \rfloor$。

（2）若 $2\times i+1<n$，则 i 的左子女为 $2\times i+1$；若 $2\times i+2<n$，则 i 的右子女为 $2\times i+2$。

（3）若 i 为偶数，且 $i\neq 0$，则其左兄弟为 $i-1$；若 i 为奇数，且 $i\neq n-1$，则其右兄弟为 $i+1$。

（4）i 所在层次为 $\lfloor \log_2(i+1) \rfloor$。

这样，可以用线性表来储存二叉树。当然，如果二叉树不是完全的，那么将有一些标号没有对应的结点。后面将学到的“堆”就是采用这种表示方法。

树和森林　树（tree）要么为空，要么是由**根结点**（**root**）和 n $(n\geqslant 0)$**棵子树**（**subtree**）组成。森林由 $m(m\geqslant 0)$棵树组成。

树的表示法　需要特别注意的是，二叉树并不是树的一种，因为二叉树的子树中有严格的左右之分，而树是没有的。这样一来，可以用每个结点的父结点来表示一棵树，即**父亲表示法**。后面要介绍的并查集就可以用这个方法（需要注意的是，这种方法可能表示的是森林，而不是树）。但是这样一来，很难进行遍历操作，所以通常使用“**左儿子右兄弟**”表示法，这实际上是把树转化成了二叉树：某结点在树中的左儿子作为二叉树中该结点的左儿子，它在树中右兄弟作为二叉树中它的右儿子。此外，树还可以用广义表表示。

【例题 1】蚂蚁和瓢虫[1]

蚂蚁和蚜虫是共生的。蚜虫分泌出蜜汁给蚂蚁饮用，蚂蚁帮助蚜虫赶走它的天敌——瓢虫。在蚂蚁山附近有一个树，这里是蚜虫生活的地方，蚜虫吸取树的汁液。有 n 个蚂蚁兵，用 1 到 n 编号。一个瓢虫威胁着这个文明，它经常出现在蚜虫活动的地方。当瓢虫坐在树上时，蚂蚁兵会出动把它赶走。它们按照如下的规则：

（1）树上任意两点之间都只有一条路径，所有的蚂蚁都沿着它所在点到瓢虫的路径前进，每移动一个位置，花的时间是单位 1。

（2）如果蚂蚁和瓢虫在同一个位置，那么蚂蚁立即把它赶走。

（3）如果某个蚂蚁的路径上有另外一只蚂蚁，那么距离目标较远的蚂蚁待在原地不动，较近的那个蚂蚁继续前进。

（4）如果有多个蚂蚁要进入同一个位置，那么选择编号最小的蚂蚁，其余的蚂蚁留在

[1] 题目来源：Polish Olympiad in Informatics

原位置不动。

（5）当蚂蚁到达了瓢虫的位置以后，把它赶走，然后停留在该位置。

瓢虫是非常顽固的动物，它被赶走了以后还会再停留到别的位置。然后蚂蚁继续行动。为了使问题简单化，假定从一个位置到相邻位置花1个单位的时间。

编程读入树的描述，蚂蚁的开始位置，以及瓢虫停留地点，计算出每个蚂蚁的最后位置，以及该蚂蚁赶走瓢虫的次数。

【分析】

这是一道模拟题。“任意两点之间仅有一条通路”告诉我们，这是一棵树。只要弄清每一次瓢虫出现后蚂蚁的移动情况，问题便迎刃而解。

由于所有的蚂蚁都朝着瓢虫所在点 Root 前进，所以以点 Root 为树根会利于问题的简化。用左儿子右兄弟法建立一棵树，树建好后，所有的蚂蚁兵都由下往上、由叶往根移动。遍历整棵树，对树上的任意一个结点记录该结点所有儿子被蚂蚁最先占领的时间和占领该儿子结点的蚂蚁编号，记为（T_1,S_1），(T_2,S_2)，…，(T_k,S_k)，依照时间最短基础上编号最小的原则，取$(T,S) = \mathrm{Min}\{(T_i+1, S_i) \mid 1 \leqslant i \leqslant k\}$即是该结点被蚂蚁最先占领的时间和占领该结点的蚂蚁编号。

对于任意一个蚂蚁，已经求得它前往瓢虫路上每一个结点被最先占领的时间和占领该结点的蚂蚁编号，从中取最小的，也就知道了该蚂蚁兵的移动步数。让每只蚂蚁移动它该移动的步数，同时考虑移动后处在根结点 Root 的蚂蚁，记录它赶走瓢虫的次数加一。

遍历　在线性结构中，很容易对表按顺序进行遍历，但是在非线性结构中，遍历要复杂一些，方法也不惟一。对于二叉树来说，通常有三种方式：先序遍历（preorder traversal）、中序遍历（inorder traversal）和后序遍历（postorder traversal）。三种遍历方式都是递归进行的，而且是按照根结点访问的位置决定的，左子树总是比右子树先访问。先序遍历的访问顺序为根、左子树、右子树；中序遍历的顺序为左子树、根、右子树，而后序遍历的顺序为左子树、右子树、根。树也可以进行遍历，其中两种方式是层次遍历（BFS）和深度优先遍历（DFS），它们分别用队列和栈来保存所有访问但未扩展的结点。

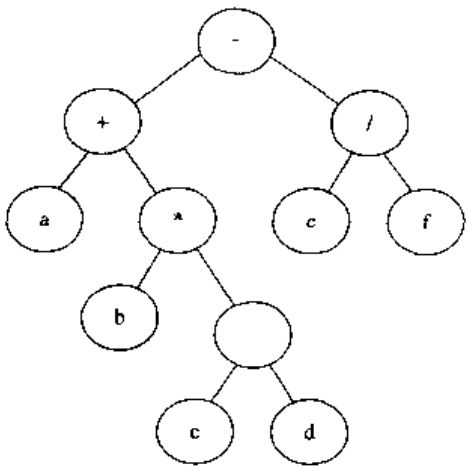


图 1-26　表达式树

例如图 1-26 中的二叉树，三种遍历的结果依次为-+a*b-cd/ef，a+b*c-d-e/f，abcd-*+ef/-。

重建二叉树　遍历的一个重要应用是重建二叉树，即根据前序遍历序列和中序遍历序列得到一棵惟一的二叉树。遍历还可以帮助统计树的有关信息，如高度、结点总数等。

【例题 2】隔三遍历[①]

树有一种“隔三”遍历法，即遍历得到的序列(a_1，a_2，…，a_n)满足：序列中任意两个相邻点在树上的距离小于或等于 3。例如图 1-27(a)的一个合法遍历为（1,3,4,6,5,2,7）。编程对任意一棵 N 个结点（N≤5 000）的树进行隔三遍历。

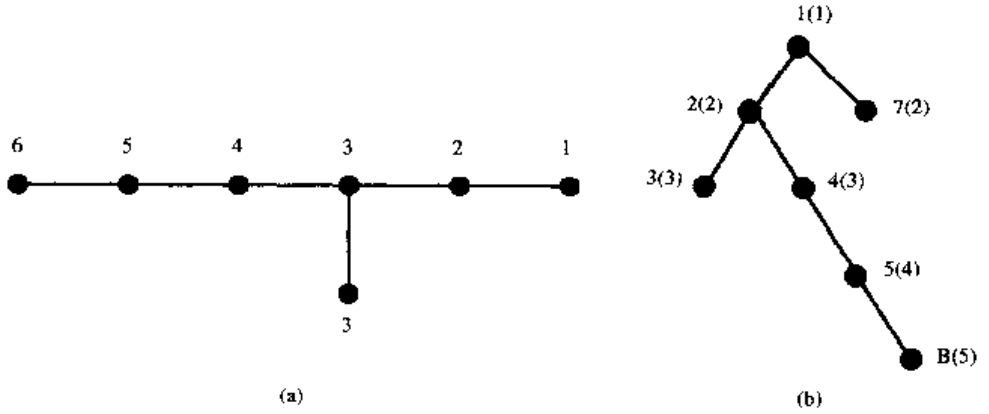


图 1-27　树的隔三遍历

【分析】

这道题目的规模很大，只能考虑进行构造而不是搜索。由于是一般的树，可以模仿二叉树遍历的方法。让我们看看样例题的答案是如何得到的。首先深度优先遍历无向图成为一棵多叉树，并标上深度，如图 1-27(b)所示。然后对每个结点 x，考虑它的深度 depth：如果 depth 为奇数，那么对结点 x 进行先序遍历；如果 depth 为偶数，那么对结点 x 进行后序遍历。

这样做为什么是对的呢？请大家先直观的想一想，再用数学归纳法加以证明。依照这样的法则，样例中，遍历序列为{1,3,4,6,5,2,7}。

介绍了这么多，相信读者已经可以解决 Ilya 的难题了吧。这个问题留给读者思考。

WEB　关于无穷性引理、树计数、同构判定、标号等问题的讨论可以在本书主页上找到。

小知识——树的特点和应用

除了这里提到的一些性质外，树还有以下性质：

- 树连通不含圈，因此可以通过收缩图中的圈来得到一棵树。
- 树是递归结构，因此可以在上面实施动态规划（见 1.5 节）。
- 树有很多变种，广泛地应用到排序和检索中（见 1.6 节）。
- 树作为一种特殊的图，在图论中也应用广泛，例如作为一些算法的辅助结构（如最短路树），而树的另外一些特性，如色多项式、最大独立集以及同构判定也有比一

① 题目来源：Polish Olympiad in Informatics

般图好得多的结果。更进一步地，有一个猜想是，所有树都是优美（graceful）的。

专题五——最近公共祖先问题（LCA）

在这一部分中，介绍涉及到树的最基本的问题之一，求树的最近公共祖先（Least Common Ancestors, LCA）。该问题表述如下：

LCA 问题：给出一棵有根树 T，对于任意两个结点 u, v，求出 LCA(T, u, v)，即离根最远的结点 x，使得 x 同时是 u 和 v 的祖先。

把 LCA 问题看成询问式的：给出一系列询问，程序应当对每一个询问尽快做出反应。对于这类问题有两种解决方法：一是用比较长的时间做预处理，但是等信息充足以后每次回答询问只需要用比较少的时间。这样的算法叫做**在线算法（online-algorithm）**。另外有一类算法是先把所有的询问读入，然后一起把所有询问回答完成，这样的算法叫做**离线算法（offline-algorithm）**。它们解决的问题都是询问式的，但是方法和特点不同，而且适用范围也不同（如果询问给出是有间隔的，往往只能用在线算法）。希望读者通过 LCA 问题能够对两类算法的基本设计方法有一个粗略的了解。

最简单的在线算法是先对所有可能的 $O(n^2)$种询问计算出结果，然后每次询问都可以在 $O(1)$的时间内直接得到结果。可以把它转化为 $O(n^2)$次单个的 LCA 计算（实际上它已经是和离线算法一样了）。每次可以用如下方法：

单个 LCA 问题的朴素算法： 从 u 的父亲开始顺着树往上枚举 u 的祖先并保存在一个列表 L 中，然后再用类似的方法枚举 v，当第一次发现某个祖先 x 在 L 中，则输出 x。

由于 L 可以达到 $O(n)$的，所以朴素算法的时间复杂度下限为 $\Omega(n)$。用此法的在线算法的时间复杂度下限为 $\Omega(n^3)$。算法可以通过递推来改进。

在线 LCA 问题的 $O(n^2)$-$O(1)$算法： 令 $L(u)$为 u 的深度（离根的距离）。不妨设 $L(u) \leqslant L(v)$，则如果 u 是 v 的父亲，LCA(u,v) = u，否则 LCA(u, v) = LCA(u, *father*(v))。这样递推的总时间复杂度为 $O(n^2)$，即在 $O(n^2)$的预处理，$O(1)$的询问时间解决了 LCA 问题。如果一个在线算法的预处理时间复杂度为 $O(f(n))$，询问时间为 $O(g(n))$，则用 $O(f(n))$-$O(g(n))$来表示它。

刚才的递推方法给了我们一个启发。当 $L(u) \leqslant L(v)$时，可以根据 LCA(u,v)的答案把所有结点分成若干个等价类：u 的子树上结点 v 都满足 LCA(u, v) = u；u 的父亲 *father*(u)的任何不以 u 为根的子树上结点 v 都满足 LCA(u, v) = *father*(u)；*father*(*father*(u))的任何不以 *father*(u)为根的子树上结点 v 都满足 LCA(u, v) = *father*(*father*(u))…这个思路给我们提供了一个不错的离线算法。

LCA 问题的 Tarjan 算法： 用 LCA(*root*[T])调用下面的过程，算法将会回答所有询问。最开始处理所有询问，如果存在询问 LCA(u, v)，就把 v 保存在集合 $Q(u)$里。初始时所有结点颜色均为 WHITE。

```
void LCA(u){
  MAKE-SET(u);
  ancestor[FIND-SET(u)] = u;
  for(u 的每个儿子 v){
```

```
    LCA(v);
    UNION(u, v);
    ancestor[FIND-SET(u)] = u;
  }
  color[u] = BLACK;
  for(Q(u)中的所有元素 v){
    if (color[v] == BLACK)
      printf("LCA(%d , %d) = %d\n", u, v, ancestor[FIND-SET(v)]);
  }
}
```

其中 MAKE-SET(u)表示建立一个集合，只包含元素 u，且 u 为集合的代表元。UNION(u, v)表示把 u 和 v 所在的集合合并，u 为集合的代表元。FIND-SET(u)表示找出集合的代表元。可以看出，在任何时候一个集合里面的元素都形成了一棵树，每处理完一棵子树就把它并到父亲所在的集合中。该算法执行的是深度优先遍历，因此对于任何处理过（即黑色）的结点 v，v 当前所在的集合的代表元就是 v 和当前处理结点 u 的 LCA。

算法的时间复杂度取决于 MAKE-SET, UNION 和 FIND-SET 的实现细节。把这个问题留在 1.4 中继续讨论。这是一个相当经典的离线算法，算法的巧妙之处在于利用了并查集，使得和当前处理元素 u 有关的所有询问都可以立即得出，但是又不需要一一计算（刚才的递推算法必须一一计算），因此平均下来每次询问时间几乎为常数。

为了更好地解决 LCA 问题，先把它转化为看起来更容易的 RMQ(Range Minimal Query)问题：

RMQ 问题：给一个长度为 n 的数组 A，回答询问 RMQ(A, i, j)，即 A[i…j]之间的最小数的下标。在不引起误会的情况下，直接用 RMQ(i, j)来表示这个询问。

给树 T 做深度优先遍历，并记录下每次到达的结点。第一个记录的结点是根 root(T)，每经过一条边都记录它的端点。由于每条边恰好经过了两次，因此一共将记录 $2n-1$ 个结点。我们用 $E[1, \cdots, 2n-1]$来表示这个数组，并用 $R[i]$来表示 E 数组中第一个值为 i 的元素下标，那么对于任何 $R[u] < R[v]$的结点 u, v 来说，DFS 中从第一次访问 u 到第一次访问 v 所经过的路径应该是 $E[R[u], \cdots, R[v]]$。虽然这些结点会包含 u 的后代，但是其中深度最小的结点一定是 u 和 v 的 LCA。即：令数组 $L[i]$表示结点 $E[i]$的深度，那么当 $R[u] \leqslant R[v]$时，LCA(T, u, v) = RMQ(L, $R[u]$, $R[v]$)；类似地，如果 $R[u]>R[v]$，LCA(T, u, v) = RMQ(L, $R[v]$, $R[u]$)。

这样，在 $O(n)$时间内把 LCA 问题转化为了 RMQ 问题。而且这个 RMQ 问题还比较特殊：相邻两个元素的值相差为 1 或者-1。这样的问题称为±1-RMQ。

一般 RMQ 问题的 Sparse Table(ST)算法：这个算法记录了所有长度形如 2^i 的所有询问的结果。用 $d[i, j]$表示从 i 开始的，长度为 j 的区间内的 RMQ，则有递推式：$d[i, j] = \min\{d[i, j-1], d[i+2^{j-1}, j-1]\}$，即用两个相邻的长度为 2^{j-1} 的块的取更新长度为 2^j 的块。这样，预处理的时间复杂度为 $O(n\log n)$。询问时只要取 $k=[\log_2(j-i+1)]$，那么令 A 为从 i 开始的长度为 2^k 的块，B 为到 j 结束的长度为 2^k 的块，那么 A 和 B 都是$[i,j]$的子区间，但是 A 和 B 一起将覆盖$[i,j]$。这样，A 的最小值与 B 的最小值的较小者就是 RMQ(i,j)，即：

$$RMQ(i, j) = \min\{d[i, k], d[j-2^k+1, k]\}$$

对于±1-RMQ，希望利用到加/减一性质得到更好的在线算法。如何利用这个性质呢？有如下显然的定理：如果长度均为 n 的数组 A 和数组 B 满足：对于任何 $1 \leqslant i \leqslant n$，$A[i]-B[i]=$常数，则 A 和 B 的最小值在同一个位置。这个定理的意义在于：长度为 n 的数组本质不同的只有 2^n 个！这样，一个新的算法产生了：

±1-RMQ 问题的 $O(n)$-$O(1)$算法：把数组 A 划分成每部分为 $L=\log_2 n/2$ 的小块，则一共有 $2n/\log_2 n$ 块。用 $O(n)$的时间求出所有小块的最小值，令 A'[i]表示第 i 个小块的最小值，则可以用 $O(2n/\log_2 n \times \log(2n/\log_2 n)) = O(n)$的时间内做好 ST 算法的预处理，以后可以在 $O(1)$的时间内回答 A'上的 RMQ 询问。

对于一般的询问 RMQ(i, j)，可以先求出 i 所在的块编号 x 和它在块中的下标 a，以及 j 所在的块编号 y 和它在块中的下标 b。

（1）如果 $x=y$，则执行块内 RMQ：In-RMQ(x, a, b)，表示第 x 块中下标 a 到 b 的最小值。

（2）否则区间[i, j]分成三部分，即在块 x 中，从 i 到块末的最小值 In-RMQ(x, a, L)，在块 y 中从块首到 j 的最小值 In-RMQ(y，1，b)以及第 x+1 块到 y−1 块的最小值 RMQ(A'，x+1，y−1)。区间划分如图 1-28 所示。

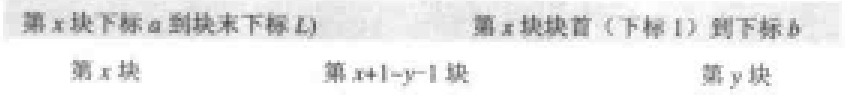


图 1-28　区间划分

那么结果为：min{In-RMQ(x, a, L), In-RMQ(y, 1, b), RMQ(A', x+1, y−1)

由于使用了 ST 算法，已经可以在 $O(1)$的时间内计算出 RMQ(A'，x+1，y−1)了，因此我们的重点是计算第 x 块内从下标 a 到下标 b 的最小值 In-RMQ(x，a，b)。还记得刚才的定理吗？由于所有的小块长度均为 $L=\log_2 n/2$，所以它们最多只有 $2^L=n^{1/2}$种本质不同的数组，每个数组最多有 $L \times L=O(\log^2 n)$种询问。因此可以用前面介绍的递推法用 $O(n^{1/2}\log^2 n)$的时间事先求出所有可能数组的所有询问，再用 $O(n)$的时间计算出每个小块属于哪个数组，就可以用查表的方法在 $O(1)$的时间内算出所有 In-RMQ 的值，问题条件得以解决。

练　习　题

思考题：

1.3.13　可以用处理树的方法处理森林吗？

1.3.14　求解Ilya的难题。

1.3.15　给一棵二叉树，证明一定可以把它的结点分成三个部分A，B，C，使得每一个结点和儿子以及兄弟（如果有的话）处于不同的集合，使得$\max\{|A|, |B|, |C|\} - \min\{|A|, |B|, |C|\} \leqslant 1$。

1.3.16　给一个有n个结点的树，每个结点i上有一个正权$w(i)$，每条j边也有一个正权$c(j)$。设计一个线性时间复杂度的算法，找出一个结点u，使得对于所有其他点i，$w(i) \times L(i)$的总和尽量小。其中$L(i)$表示i到u的路程（即路径上所有边的权和）。

编程题：

1.3.17　树重建[①]

给出一棵标号树的 BFS 序列和 DFS 序列（定义见下小节），设计一个算法重新建立这棵树（结点数 $n \leqslant 1\ 000$）。当某结点被扩展时，它的所有孩子应该按照编号从小到大的顺序访问。例如一棵树的 BFS 序列为 4 3 5 1 2 8 7 6，DFS 序列为 4 3 1 7 2 6 5 8，则一棵满足条件的树如图 1-29 所示。

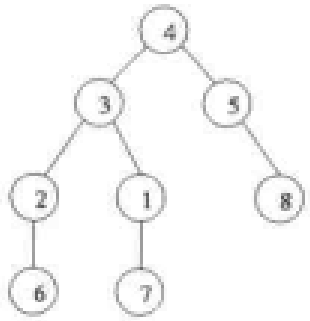


图 1-29　一棵满足条件的树

***1.3.18　树分割**[②]

给出一棵有 n 个结点的树，每个顶点上有一个正整数权 $w(i)$。把树分成连通的 K 个部分($1 \leqslant K \leqslant n$)，使得权和最小的那部分权和尽量大。①假设最大权值为 m，设计一个关于 m,n 的多项式算法；②设计一个和 m 无关的，n 的多项式算法。

****1.3.19　毛毛虫**[③]

毛毛虫是含 $N(n \leqslant 10\ 000)$个结点的一棵树，它包含一条主链，使得所有点要么在主链上，要么和主链上某结点相邻。如图 1-30 所示有两只合法的毛毛虫，灰色的结点代表主链。

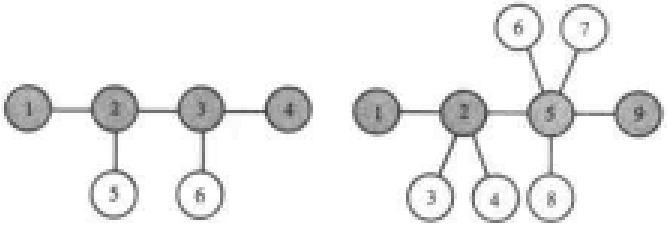


图 1-30　两只毛毛虫

我们希望给毛毛虫的每个顶点标号 1，2，3，…，N，使得所有边的两端结点标号差的绝对值恰好包含了 1，2，3，…，N-1，每个数正好一次。如图 1-31 所示是一个合法的标

① 题目来源：UVA Problem Archive Online Contest
② 题目来源：经典问题
③ 题目来源：CEOI 2000, Caterpillar，经典问题

号。边上的数字为标号差。

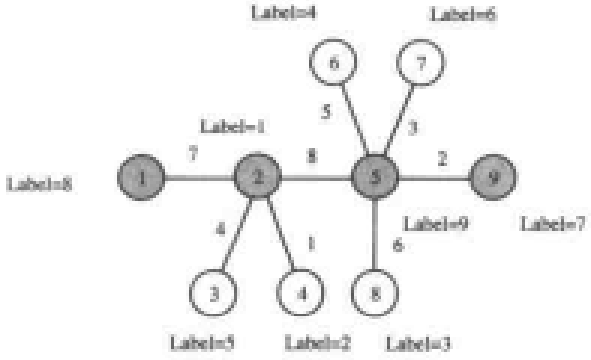


图 1-31　一个优美标号

提示：本题的算法非常简单，但是证明需要一些技巧。

1.3.4　图及其基本算法

树结构和线性结构相比已经复杂很多了，但是它只是一种层次性的数据结构，数据间的关系比较特殊。这一节所要介绍的“图”是一个比树更加复杂的数据结构。这里只介绍基本概念，而一些重要内容将在图论部分继续加以讨论。

1．图的定义

这里的图是指图论里的图，而不是几何上、地理上或者是美术上的“图”。它的定义是：

图的定义　图（graph）可用 G = (V, E)来表示，即顶点(node / vertex)集合 V 和边(edge)集合组成的二元组。E 中的每条边是 V 中一对顶点(u,v)。如果(u,v)是无序对，那么称该图为无向图（undirected graph），否则为有向图（directed graph）。任意两个顶点最多只有一条边（多条边成为重边），且每个点都没有连接到它自身的边的图叫简单图（simple graph）。如果未加说明，以后只讨论简单图。若有 n 个顶点的无向图 $n(n-1)/2$ 条边，则此图为完全图（complete graph）K_n。

顶点和边　和树一样，图也是可以用图形表示的。用一个小圆圈表示顶点，而线段表示边。如果是有向图，在线段上加箭头。如果(u,v)是 $E(G)$中的一条边，则称 u 与 v 互为邻接顶点（adjacent vertex）。一个顶点 v 的度（degree）是与它相关联的边的条数。在有向图中，顶点的度等于该顶点的入度与出度之和。顶点 v 的入度（in-degree）是以 v 为终点的有向边的条数；顶点 v 的出度（out-degree）是以 v 为始点的有向边的条数。

权　某些图的边具有与它相关的数，称之为权（weight）。这种带权图叫做网络（network）或带权图（weighted graph）。

路径　在图 G=(V, E)中，若从顶点 v_i 出发，沿一些边经过一些顶点 $v_{p1}, v_{p2}, \cdots, v_{pm}$，到达顶点 v_j。则称顶点序列$(v_i\, v_{p1}\, v_{p2} \cdots v_{pm}\, v_j)$ 为从顶点 v_i 到顶点 v_j 的路径（path）。它经过的边(v_i, v_{p1})、(v_{p1}, v_{p2})、…、(v_{pm}, v_j)应是属于 E 的边。非带权图的路径长度（length）是指此

路径上边的条数。带权图的路径长度是指路径上各边的权之和。若路径上各顶点 $v_1,v_2,\cdots,v_m$ 均不互相重复，则称这样的路径为简单路径。若路径上第一个顶点 v_1 与最后一个顶点 v_m 重合，则称这样的路径为回路或环。

连通性　在无向图中，若从顶点 v_1 到顶点 v_2 有路径，则称顶点 v_1 与 v_2 是连通（connected）的。如果图中任意一对顶点都是连通的，则称此图是连通图（connected graph）。非连通图的极大连通子图叫做连通分量（connected component）。在有向图中，若对于每一对顶点 v_i 和 v_j，都存在一条从 v_i 到 v_j 和从 v_j 到 v_i 的路径，则称此图是强连通图（strongly connected graph）。非强连通图的极大强连通子图叫做强连通分量（strongly connected component）。一个连通图的生成树（spanning tree）是它的极小连通子图，在 n 个顶点的情形下，有 $n-1$ 条边。

2. 图的实现

图在计算机中的存储主要有两种方式：邻接矩阵和邻接表。

邻接矩阵　在图的邻接矩阵（adjacent matrix）表示中，有一个记录各个顶点信息的顶点表，还有一个表示各个顶点之间关系的邻接矩阵。设图 $A=(V,E)$ 是一个有 n 个顶点的图，则图的邻接矩阵是一个二维数组 gr，当 (i,j) 为图中的边时 gr[i,j]=1，否则 gr[i,j]=0。容易发现，无向图的邻接矩阵是对称的，有向图的邻接矩阵可能是不对称的。如果需要储存带权图，那么只需要把 gr[i,j] 的值改为权 w(i,j) 的大小。带权的邻接矩阵无法保存重边。

邻接表　图的邻接表（adjacent list）把同一个顶点发出的边链接在同一个边链表中，链表的每一个结点代表一条边，叫做边结点，结点中保存有与该边相关联的另一顶点的顶点下标 dest 和指向同一链表中下一个边结点的指针 link。在有向图的邻接表中，第 i 个边链表链接的边都是顶点 i 发出的边。也叫**出边表**。在有向图的逆邻接表中，第 i 个边链表链接的边都是进入顶点 i 的边。也叫**入边表**。带权图的边结点中保存该边上的权值 cost。

前向星　星形表示法把所有的边集中在一起形成一个边列表。按照边的组织方式，星形表示法分为前向星和后向星。其中，前向星（forward star）表示按照第一结点从小到大排序，即先记录以结点 1 为起点的所有边，再记录以结点 2 为起点的所有边，……。它的好处是比链表节省空间和时间。只要不增加或删除边，它能很快地找到以一个点出发的所有边，只要记录每个点 i 出发的第一条边的位置 $f[i]$，则从它出发的边在前向星边列表中的第 $f[i]$~$f[i+1]-1$ 号元素。

3. 图的遍历

有的时候，需要按照某种顺序访问图中的所有顶点（例如，做一些数据统计或者给特定顶点做标记），这样的操作叫做图的“**遍历**”或者“周游”。

一般来说，不提倡按照随意的顺序进行周游。常用的遍历方法有两种：广度优先遍历和深度优先遍历。两种方法的思路和实现方法有很多类似之处，下面分别进行讨论。在遍历的时候，把图看成是无权的。在理论中，需要研究随机图。随机图的周游和普通图有着很大的区别，不过这里不作考虑。

图的广度优先遍历　广度优先遍历（BFS）的思想是，从一个顶点出发，按照到它的最短路径长度从小到大的顺序遍历。和 floodfill 类似，也是用一个队列来实现。广度优先遍

历的好处是：如果是遍历一个很大的图，需要找到某两地的最短距离，可以在遍历到这个顶点的时候立刻停止遍历。算法的时间复杂度为 $O(n+m)$。

拯救大兵瑞恩的故事[1]

1944 年，特种兵麦克接到国防部的命令，要求立即赶赴太平洋上的一个孤岛，营救被敌军俘虏的大兵瑞恩。瑞恩被关押在一个迷宫里，迷宫地形复杂，但是幸好麦克得到了迷宫的地形图，如图 1-32 所示。

迷宫的外形是一个长方形，其在南北方向被划分为 $N(N\leqslant 15)$行，在东西方向被划分为 $M(M\leqslant 15)$列，于是整个迷宫被划分为 $N\times M$ 个单元。我们用一个有序数对（单元的行号，单元的列号）来表示单元位置。南北或东西方向相邻的两个单元之间可以互通，或者存在一扇锁着的门，又或者存在一堵不可逾越的墙。迷宫中有一些单元存放着钥匙，并且所有的门被分为 $P(P\leqslant 10)$类，打开同一类门的钥匙相同，打开不同类门的钥匙不同。

大兵瑞恩被关押在迷宫的东南角，即（N,M）单元里，并已经昏迷。迷宫只有一个入口，在西北角，也就是说，麦克可以直接进入（1,1）单元。另外，麦克从一个单元移动到另一个相邻单元的时间为 1，拿取所在单元的钥匙的时间以及用钥匙开门的时间忽略不计。

你的任务是帮助麦克以最快的方式抵达瑞恩所在单元，营救大兵瑞恩。

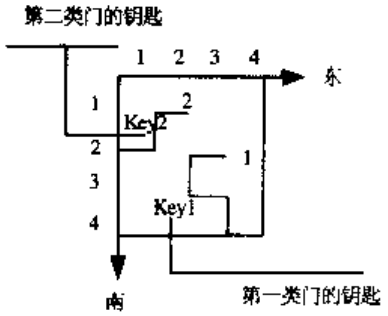


图 1-32　迷宫示例

例如，对于上图所示的迷宫，最少需要 14 个时间单位。

麦克的当前状态由当前位置和“获得了哪些钥匙”来决定，即[x,y,S]，其中(x,y)是坐标，S 是钥匙集合。由于钥匙只有 10 种，所以 S 可以用一个二进制串来表示，它的取值范围是[0…1023]。这样，把每个坐标-钥匙集合的**不同组合**都作为单独的顶点，实际上是得到了一个图 G，它有 $m\times n\times 2^p$ 个结点，每个结点最多连四条边，因此边数不超过 $mn\times 2^{p-1}$。在图 G 上从[1,1,s]（注意：（1,1）格可能有钥匙，这里的 s 指的是获取所有钥匙以后的钥匙集合）进行宽度优先遍历，就可以在找到目标结点[n,m,*]的时候（*指任意要是集合，因为只要救出大兵瑞恩，我们不用管麦克已经拿到了哪些钥匙）立刻停止。时间复杂度为 $O(2^p\times m\times n)$。

如果不能很好地理解这个例子，建议先解决这个问题的简化版：假设没有钥匙，也没有门……

[1] 题目来源：CTSC 1999

图的深度优先遍历　深度优先遍历（DFS）的思想是，从一个顶点出发，沿着边尽量走到没有访问过的顶点。如果没有未访问过的顶点，就沿着与边相反的方向退一步。注意，我们强调“没有访问过”，是因为对于有环图，一般的深度优先遍历会产生循环，因此需要记录每个顶点是否已经访问过。

DFS 的过程可以看成是递归的：从一个顶点 u 出发，记录 u 已被访问，然后对于它邻接的每个顶点 v，从 v 深度优先遍历该图。既然是递归过程，也可以用栈来实现它。实现方法和 BFS 很类似，遇到未访问过的顶点，只需要把“入队”操作改成“入栈”。广度优先遍历的好处是，平均情况下占用的附加空间（栈）比较少（想一想，为什么？）。如果是遍历一个很大的图，只需要找到从一个顶点到另一个顶点的任意一条通路，可以用 DFS，因为实现较简单，用的空间也较少。

英雄和公主的故事

很久很久以前，有一个国王。国王有一个美丽善良的女儿，深得父亲的喜爱。突然有一天，公主神秘地消失了，没有留下一点蛛丝马迹，让国王感到焦急万分却又无可奈何。国王派出了最好的骑士四处寻找公主，而他自己只能日夜祈祷，希望心爱的女儿早日回到自己的身边。

日子一天天过去了，派去寻找公主的骑士回来了。他们给国王带来了一个好消息，也带来了一个坏消息。好消息是，他们打听出公主的下落了；坏消息是，公主是被世界上最可怕的恶魔 Diablo 掳走了。

就在国王几乎绝望的时候，一位不愿意透露姓名的少年出现在了国王面前。少年说，他愿只身前往营救公主，不达目的绝不回来。说完，便消失在了茫茫夜色中。

魔王宫殿建在黑暗的山洞中。在洞口和宫殿之间，有一个迷宫。迷宫里藏着各种妖魔鬼怪，无论是谁进去了，随时都有丧生的可能。少年知道，贸然前往迷宫凶多吉少，于是决定先去拜访住在附近的“小灵通”。

“小灵通”告诉少年：Diablo 喜爱黑暗，惧怕光明。若要制服它，少年必须想办法把阳光从洞口引到魔王宫殿。幸运的是，在迷宫里竟然有很多像镜子一样可以反光的东西，如图 1-33 所示。少年可以用魔法随意转动镜子，让阳光按照他预先设计好的路线射入魔王宫殿……

(a) 原始迷宫　　(b) 转动镜子后

图 1-33　魔王设置的迷宫

可是镜子太多了，他究竟应该施用法术转动哪些镜子，才能把阳光引入魔王宫殿，进而潜入宫殿打败魔王、救出美丽的公主呢？

由于镜子是可以翻转的，所以光线在且仅在镜子处改变方向。我们的一般思路是："构造图 G：每面镜子是一个结点 V。两个结点 U，V 相邻当且仅当两面镜子在同一行或者同一列，且中间没有其他镜子"，如图 1-34(a)所示。然后在图上作深度优先遍历，找到阳光入口到公主牢房的一条通路，然后根据路径的情况沿途设置镜子，如图 1-34(b)所示。

图 1-34　两种模型

图 1-35　纯粹性和完备性

这样的思路对不对呢？不妨考虑两个问题，即 G 的通路集合是否满足两条性质：纯粹性和完备性。只有同时满足两条性质，G 的通路和实际光路才是一一对应，否则这样的模型是没有反映真实问题的。

完备性：如图 1-35(a)，这样的光路是错误的，但是我们的方法有可能求出这样的"解"。

纯粹性：如图 1-35(b)，这样的光路是可能的，但是我们的方法得不到，因为同一个结点经过了两次。

所以需要修改模型，即把一个点拆成上下左右四个点，作为"入射方向"，如图 1-34(b)所示。这样，光路和 G 的通路一一对应。用 DFS 求出一条通路，由于每个点只连 $O(1)$条边，所以算法的时间复杂度为 $O(n)$。求出通路以后，接下来的工作就容易得多了，即根据光的转向布置镜子，例如图 1-36 所示的四种情况：

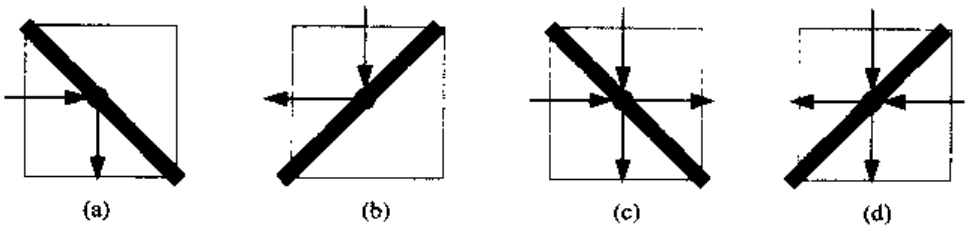


图 1-36　根据光线转折确定镜子摆放

需要注意的是，这样设置镜子不一定使镜子转动数目最少，即使我们用 BFS 找到最短路，这样转动的镜子数也不一定最少（想一想，为什么？），但这样的方案是可行的。

刚才先用宽度优先遍历拯救了一个英雄，又用深度优先遍历营救一位美丽的公主，图

的遍历对于迷宫问题来讲确实是很有用的。BFS 的好处是把结点按照离某一点从近到远的顺序遍历，就像是一个视力不好的人把眼镜掉在了地上。他先会在离自己最近的地面上用手摸，如果找不到才慢慢把目标移到离自己比较远的地方。DFS 的好处在于内存耗费比较少。在第 2 章我们还将看到，利用 DFS 很容易将边分类，从而得到一些图的有用信息。

4. 拓扑排序

电气工程师

电气工程师们刚刚为学校的计算机大楼安步好了网线，可由于工作疏忽，从机房到办公室的网线有交叉。在机房，所有网线从左到右编号为 1,2,3,…,*N*，而在办公室，网线从左右不再是编号为 1,2,3,…,*N* 的线。工程师们无法确定办公室内线的顺序，但他们发现两条线最多只相交一次，而且检测出了任意两条线是否相交。如图 1-37(b)，实线表示两条线交叉，虚线表示不交叉。给出每两条线是否交叉的信息，请计算办公室内从左到右各条线的编号。例如，从图 1-37(b)的信息可以确定实际布线情况如图 1-37(a)所示，编号依次为 3,2,4,1,5。

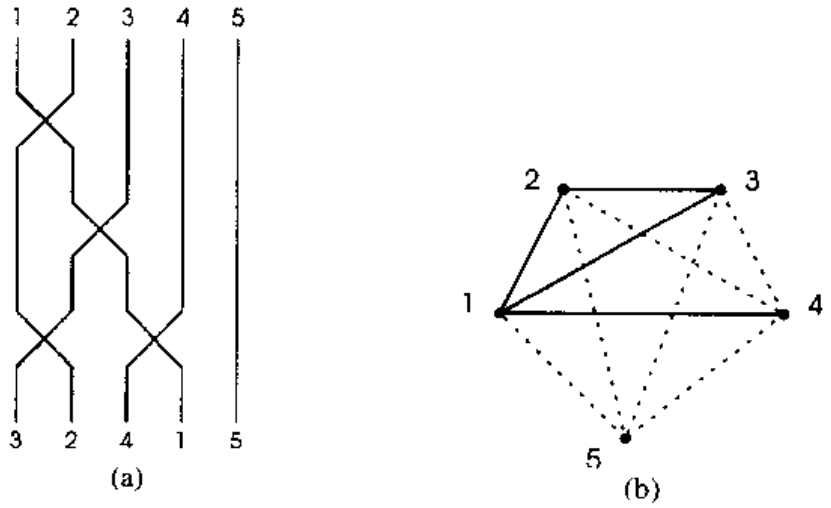


图 1-37　实际布线和相交情况

拓扑排序　在解决这个问题之前，先介绍拓扑排序。在“基本算法”一节中，曾提到过这个问题。“有一个有向图 G。如果有向边(*u*,*v*)的含义是 *u*>*v*，那么所有顶点的大小关系是怎样的？”这就是拓扑排序问题。拓扑排序是从局部大小关系退出全局大小关系。对于无法确定大小的，可以任意指定一种大小关系。容易看出，拓扑排序可以成功当且仅当该图没有有向回路。证明方法是构造性的：每次取出入度为 0 的点，它们是“最大的”，并在图中删除它们。如果所有点的入度都不为 0，那么必然存在一条回路(想一想，为什么？)，这时拓扑排序失败。

拓扑排序算法　每次找出未排序好的顶点中度数为 0 的一些顶点。注意到这个事实：第二次度数为 0 的顶点一定与本次度数为 0 的顶点相连。这样，只要在本次删除与最大顶点关联的边时记录一下删除后入度为 0 的点，这些点就是下次的“最大顶点”。由于每条

边恰好考虑一次，因此算法的时间复杂度为 $O(m)$。用一个入度为 0 的点的队列来实现，每取出一个“最大”点，删除与它关联的边后把入度变为 0 的点加入队列。

回到这道题目中。两条线最多只相交一次，如果导线 x 与导线 $y(x<y)$相交，则在办公室 x 排在 y 的后面；如果导线 x 与导线 $y(x<y)$不相交，则在办公室 x 排在 y 的前面。这样，根据导线两两是否相交的信息，可以知道导线两两之间的先后顺序。有了这些顺序，剩下的工作惟有拓扑排序。

练　习　题

思考题：

1.3.20　前向星算法的优势和弱点分别是什么？

1.3.21　什么样的图可以进行拓扑排序？存在多项式算法列举出所有拓扑顺序吗？

1.3.22　把一个图的边看成顶点，原图中相邻边对应的新图顶点间连一条边，能得到一个新的图吗？

***1.3.23**　对于任意一个图 G，都能找到 1.3.22 的逆变换吗？如果能，有多项式的变化算法吗？

编程题：

***1.3.24**　**与非门电路**[①]

可以用与非门（NAND）来设计逻辑电路。每个 NAND 门有两个输入端，输出为两个输入端与非运算的结果。即输出 0 当且仅当两个输入都是 1。给出一个由 N 个 NAND 组成的无环电路，电路的输入全部连接到一个相同的输入 x，如图 1-38 所示。

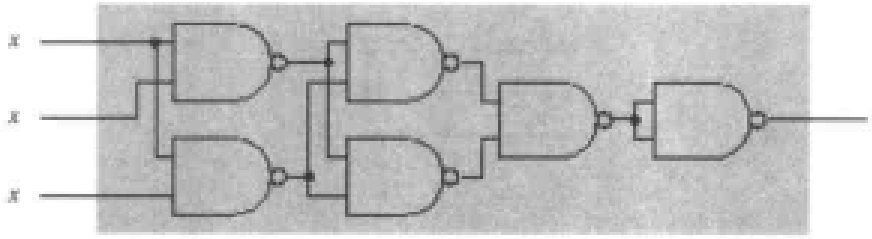


图 1-38　一个电路

请把其中一些输入设置为常数，用最少的 x 完成相同功能。图 1-39 是一个只用一个 x 输入但是可以得到同样结果的电路（读者应当习惯用有向无环图来解释这类无反馈的电路）：

[①] 题目来源：ACM/ICPC Regional Contest CERC 2001

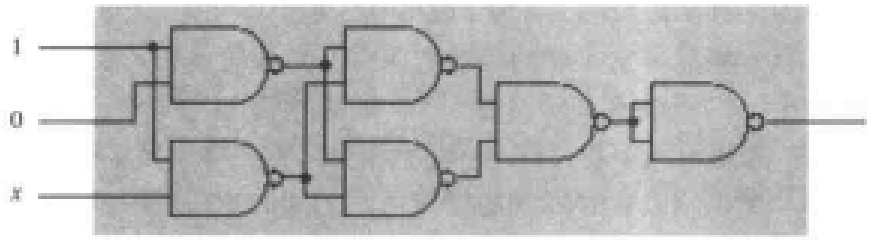


图 1-39　简化后的网络

1.3.25　比较网络[①]

比较网络仅由线路和比较器构成。如图 1-40(c)所示，比较器是具有两个输入 x 和 y 以及两个输出 x'和 y'的一个装置，且执行下列函数：

$x'=\min(x,y)$

$y'=\max(x,y)$

因为图 1-40(a)中给出的比较器的图形表示太大而不方便，所以将按常规把比较器画为一条垂直线，如图 1-40(b)所示。输入在左面，输出在右面，较小的输入值在输出端的上部，较大的输入值在输出端的下部。因此可以认为比较器对两个输入进行了排序。

线路把一个值从一处传输到另一处，它可以把一个比较器的输出端与另一个比较器的输入端相连，在其他情况下它要么是网络的输入线，要么是网络的输出线。假定比较网络含 n 条输入线和 n 条输出线，需要排序的值通过输入线进入网络，由网络计算出的结果通过输出线输出。我们所说的输入序列<a_1,a_2，a_3，…，a_n>和输出序列<b_1，b_2，b_3，…，b_n>分别指输入线和输出线中的值。

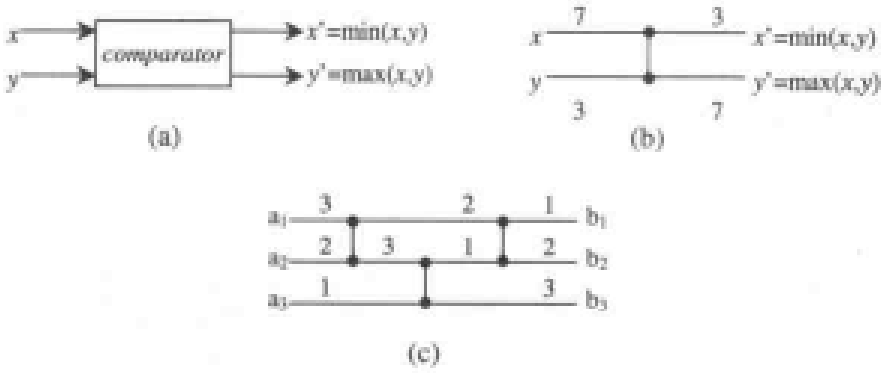


图 1-40　比较器、简化画法和比较网络举例

图 1-40(c)说明了一个比较网络，它是一个由线路互相连接着的比较器的集合，把具有 n 个输入和 n 个输出的比较网络画成一个由 n 条水平线组成的图，比较器则垂直地与两条水平线相连接。需要注意的是图中的一条水平线并不是仅代表一条线路，而是连接到各个比较器上的不同线路的一个序列。每个比较器的输入端要么与网络的 n 条输入线路 a_1，a_2，a_3，…，a_n 中的一条相连接，要么与另一个比较器的输出端相连接。类似地，每个比较器

[①] 题目来源：IOI2000 中国国家集训队原创题目. 命题人：高寒蕊

的输出端要么与网络的 n 条输出线路 b_1，b_2，b_3，…，b_n 中的一条相连接，要么与另一个比较器的输入端相连接。互相连接的比较器主要应满足如下要求：互相连接所成的图中必须没有回路——我们沿图中一个比较器的输出端到达另一个比较器的输入端再到输出端、输入端……，如此下去，通过的路径必须不能回到第一个比较器，也不能两次经过同一个比较器。因此，如图 1-40(c)所示，画一个比较网络时可以把网络输入端画在左边，而把网络的输出端画在右边，数据从网络左边向右边移动从而通过网络。只有当同时有两个输入时，比较器才能产生输出值。

对于一个固定的比较网络，一种输出输入只会产生一种输出序列，但是一种输出序列却可能对应着多种输入序列。现在已知一个固定的比较网络，和一种输出序列，要求出一种输入序列产生这种输出序列。

*****1.3.26　推门游戏**[①]

有一个古老的任天堂游戏是这样的：

如图 1-41 所示，你从 S 处出发，每次可以往东、南、西、北四个方向之一前进。如果前方有墙壁，游戏者可以把墙壁往前推一格。如果有两堵或者多堵连续的墙，游戏者不能将它们推动。另外，游戏者也不能把游戏区域边界上的墙推动。

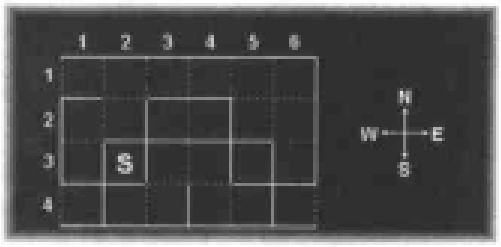


图 1-41　一个游戏迷宫

用最少的步数走出迷宫（边界处没有墙的地方就是出口）。

WEB　这是一道有趣又吸引人的题目，它的进一步讨论文章可以在本书网页上找到。

1.3.5　排序与检索基本算法

排序和检索是数据结构中很重要的问题。这里仅以线性表为例进行说明。

1．二分检索

爱丽丝和精灵的故事[②]

“森林里有精灵吗？”爱丽丝问。

“有啊，而且有很多呢！”小仙女回答说。

“可是我一只也看不见啊。”

---

① 题目来源：UVA Problem Archive Online Contest – World Finals Warmup. 命题人：Jimmy Mårdell

② 题目来源：Internet Problem Solving Contest 2001

“那是因为你并不知道它们现在是什么样子啊，爱丽丝。它们很怕生，所以改变了它们模样，让你认不出来了。”

“那它们到底长什么样子呢？”爱丽丝又问。

“这片魔法森林可以用一串字母来表示：‘rabbittreebugabbitreetreerabbinrabbit’. 精灵们的模样是它的一个子串。现在就你来猜猜看，精灵们究竟长什么样子吧！”

“是不是‘bug’？”爱丽丝想了想说。

“很遗憾，不是的。不过我可以给你一个提示：精灵的数目比‘bug’要多。”

“那，是不是‘tree’？”

“不是，精灵比‘tree’要少。”

“那么，精灵们一定是‘rabbit’啦！”爱丽丝露出了笑容。

“答对了，就是‘rabbit’！”

这次虽然轻松地猜到了精灵们的模样，但是以后如果遇到更大的森林，爱丽丝担心自己会猜不出来。请帮她设计一个猜测精灵模样的程序，用尽量少的猜测次数得到正确的答案。为了不让爱丽丝太难猜，精灵们的数目保证不和任何其他子串的数目一样多。

在这个故事里，如果求出了所有长度的字符串（如果对于某长度有超过一种字符串，那么它肯定不是精灵，因此可以忽略），再按照长度从小到大进行排序，那么可以得到一个有序的线性表。现在的任务是通过一次次询问（比较操作）来找到精灵所代表的字符串。这里介绍一种方法：二分检索（binary search）。

📖 **二分检索**　二分检索的思想是每次把范围缩小一半。例如，你需要在一个从小到大有序的有 15 个元素的线性表中查找数“11”，而已知第 8 个数是“6”，“11”如果在表中，那么它一定在第 9 个数~第 15 个数这个范围之内。

二分检索算法　算法的思想是根据数据的有序性，每次把查找范围缩小一半，时间复杂度为 $O(\log n)$。解释如下：由于查找范围每次都缩小一半，设在有 n 个元素的线性结构中的比较次数为 d[n]，则 d[n]=d[(n-1) div 2+1]+1。而 d[1]=0，解这个递推式，得 d[n]=$O(\log n)$。算法实现比较简单，采用迭代的方式，记录当前元素可能存在的位置的下界 left 和上界 right。每次考查第 mid=(left+right) div 2 个元素，根据和目标的比较结果来求出新的查找范围并更新 left 或 right 的值。

排序　前面说过，二分检索只能用在有序表中，因此把无序表变为有序的方法，即**排序**（或者译为“分类”，英文是“sort”），就显得十分重要。当然，即使不进行二分检索，排序也能作为很多算法必不可少的预处理步骤。

【例题 1】电缆

Wonderland 居民决定举行一届地区性程序设计大赛。仲裁委员会志愿负责这次赛事并且保证会组织一次有史以来最公正的比赛。为此，所有参赛者的电脑和网络中心会以星状网络连接，也就是说，对每个参赛者，组委会会用一根长度一定的网线将他的计算机与中心连接，使得他们到网络中心的距离相等。

为了买网线，组委会与当地的网络公司联系，要向他们购买一定数目的等长网线，这

些缆线要尽可能长一点，使得组织者可以让选手们彼此远离。

于是公司指派管理网线事务的负责人解决此事。负责人清楚地知道仓库里每根网线的长度（精确到厘米），他也可以将他们以厘米的精度切割——前提是他得知道切成多长。但是现在，这个长度他算不出来，于是他彻底迷茫了。

你要做的，就是帮助困惑的负责人。编一个程序求出为了得到一定数目的等长网线，每根网线最大的可能长度。

【分析】

如果事先知道网线的长度，那么通过整除，可以知道等长网线的数量并判断是否足量。如果足量，称此长度为合法长度。**网线越短，网线就越多，就越可能是合法长度**。这是种单调递增的线性关系。基于这种关系，可以二分求解网线的长度，即是当确定最大合法网线长度在区间$[p_1, p_2]$内时（初始时，p_1=0，p_2=所有网线的长度和），我们尝试判断长度$p_3=[(p_1+p_2)/2]$是否合法。如果合法，那么可知最大合法网线长度在区间$[p_3, p_2]$内；如果不合法，那么可知最大合法网线长度在区间$[p_1, p_3-1]$内。如此循环，直到$p_1=p_2$即是问题的解。

优化问题和判定问题　从这个例子中看到：除了检索有序线性表，二分检索还可以把一类最优化问题转化成判定问题。这类问题的共同特点是，给一个条件 C，求满足该条件的最大数 k，其中 $1 \leqslant k \leqslant u$（$l$,$u$ 已知）。有时候，这个问题还满足这样的条件，“对于两个数 $a<b$，如果 b 满足该条件，则 a 一定也满足”。把最优化问题转化为判定问题是很有用的，在后面的动态规划和图论部分会继续介绍这一技巧。另外，在连续型问题中，常常使用二分的方法逐步逼近最优解。例如几何上的计算，我们将在几何部分中介绍。

2. 基本排序算法

黑白按钮[①]

Sam 有一个 N×M 的方格，每个格子里有一个红色或者蓝色的鹅卵石。每一行的左边有一个黑色按钮，如果你按下它，该行里所有的红色鹅卵石会变成蓝色，而所有蓝色鹅卵石会变成红色。每一列的上面都有一个白色的按钮。如果你同时按下两个白色按钮，对应的两列的鹅卵石会进行交换。

Sam 突然想到一个问题：给出两个游戏状态，他是否能够从一个状态出发，经过一系列操作（每次按一个黑色按钮或者同时按两个白色按钮），变成另一种状态？例如，以下两种状态就可以相互转换。在图 1-42(a)中按第 2 行的黑色按钮，再同时按第 1、第 3 列的白色按钮就能转化到图 1-42(b)。

Blue	Red	Blue	Red
Red	Blue	Blue	Red
Blue	Blue	Blue	Blue

(a)

Blue	Red	Blue	Red
Red	Red	Blue	Blue
Blue	Blue	Blue	Blue

(b)

图 1-42　两种状态的转化

[①] 题目来源：Internet Problem Solving Contest

稍微分析就会发现，指令执行的顺序是无关紧要的，所以不妨认为先执行行操作，再执行列操作。

先假设 B 的第 1 列是 A 的第 x 列变换而来的，然后判断这种假设的正确性。依次取 x 的值为 1…n，如果每个 x 值都不能使假设成立，那么问题的答案是否定的。

如果知道 B 的第 1 列是 A 的第 x 列变换而来的，那么根据两列每行鹅卵石颜色一一对应的关系，就可以知道每个黑色按钮的操作情况。根据这个操作情况，对 A 执行所有的行操作，然后分别将 A 和 B 的每一列按照同样的函数排序，最后比较排序后的 A、B，如果 A=B，那么假设成立。

📖 *排序是一种很常见的数据处理方法，在前面的各节中，我们不止一次地使用了排序。在刚才的例子中，排序也起到了关键作用。但是究竟应该怎样排呢？暂时抛开时间效率，介绍两种很“显然”的排序算法。*

插入排序算法　算法思想是依次处理原始表中各个元素，每次都把它插入到有序表的合适位置，保持表有序。比较次数：最佳 $O(n)$、平均和最差 $O(n^2)$；交换次数：最佳 0 次、平均和最差 $O(n^2)$

选择排序算法　算法思想是每次都从原始表中选择最小（最大）的元素插入到有序表中，并从原始表中删除。比较次数为 $O(n^2)$，交换次数为 $O(n)$。

冒泡排序算法　很多书中第一个介绍的排序算法是冒泡排序（bubble sort）。其实，冒泡排序的实质就是选择排序，而且它常常比选择排序多做一些不必要的交换。这里不介绍该算法，有兴趣的读者可以在任何一本数据结构书籍找到非常详尽的介绍，

两种排序方法是简单的，但是其中的思想应用十分广泛。

【例题 2】煎饼

有一叠煎饼正在锅里。煎饼共 N 张，每张都有一个数字，代表它的大小。如下图所示。厨师每次可以选择一个数 k，把从锅底开始数第 k 张上面的煎饼全部翻过来，即原来在上面的饼现在到了下面。例如图 1-43(a)，依次执行操作 3，1 后得到图 1-43(c)的情况。

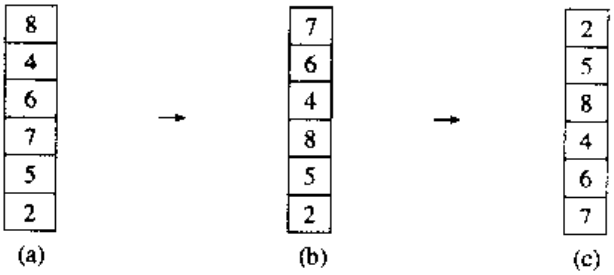


图 1-43　煎饼及其操作

设计一种方法使得所有煎饼按照从小到大排序。

【分析】

这道题目是叫我们排序，但是基本操作却是“颠倒一个连续子序列”。不过这没有关系，我们还是可以按照选择排序的思想，依次把第一小，第二小，……，的元素排到正确的位置上。假设需要把第 j 小的煎饼 i 排到位置 j 去（显然有 $i>j$，因为采用选择排序，排

到第 j 个位置时前面 j-1 个数已经是最小的 j-1 个数了），只需要执行操作 i，就把煎饼 i 翻到锅顶了，只需要再执行操作 j，就把在锅顶的煎饼 i 翻到位置 j 了，而且**两次操作均不影响已经排序好的煎饼** 1…j-1。这样，最多每个煎饼翻两次，一共不超过 $2n$ 次就可以把所有煎饼排序好了。

小知识——在线排序和部分排序

比较这两种方法元素插入有序表的顺序，你会发现：插入排序是按照原始数据的顺序，而选择排序是按照最后排序好的顺序插入。这意味着什么呢？

问题一：相邻两个输入之间有一定时间间隔。

这样，插入排序可以在下一个输入到来之前利用时间间隙做运算，因为它处理数据的顺序就是输入的顺序；如果时间间隔很大，那么该算法可以充分地利用时间而不是总在等待。而选择排序却不一样，因为它每次选择表中的最大元素，而这个“最大元素”与所有数据有关，它必须等待所有数据输入完毕。如果输入间隔太大，那么算法就会浪费大量的时间来等待输入。

问题二：你拿到了期末考试的成绩总分表，但是还没有经过排序。你没有计算机，但是老师偏偏把排名次的任务交给了你，而且还补充了一点，同学们希望早点知道前 10 名的名单。

如果你只学习了插入排序和选择排序，你会使用哪一种呢？对于排序任务来说，二者差不多，但是对于同学们的要求呢？选择排序进行了前 10 次以后，就可以得到前 10 名的名单，但是插入排序却不行，因为在处理完所有元素之前，我们都不能确定最后的结果，因为有可能第一名是最后一个元素。

提出这两个问题的目的只是为了告诉大家，排序算法在解决实际问题中可能遇到的问题。事实上，我们有更好的方法来解决这两个问题，大家在学习了堆排序和顺序选择算法之后就会明白。作为比较排序的改进，我们有复杂度处于中间的 ShellSort。我们还有基于比较的并行算法，或者通过排序网络进行排序。

3. 快速排序和归并排序

有序表合并问题　归并排序是我们介绍的第一个平均情况时间复杂度为 $O(n\log n)$的算法。事实上，它的最坏情况也是 $O(n\log n)$的。为了解释归并排序，先看一个问题：给出两个有序表（假设都为升序），把它们合并得到一个新的有序表。

合并算法　仿照选择排序的方法，每次选剩下元素中最小的一个放到排序好的表中。由于两个原始表都是有序的，所以每次只需要比较两个表头元素哪一个比较大就可以了。若一个表为空，另外一个表按顺序复制到结果中即可。表 1-7 给出了一个合并的例子。每次比较的较小元素用黑体表示。

表 1-7　有序表合并举例

步　骤	表　A	表　B	结　果
1	1 2 4 7 9	3 5 6 10 11	空
2	2 4 7 9	3 5 6 10 11	1

续表

步　骤	表　A	表　B	结　果
3	4 7 9	3 5 6 10 11	1 2
4	4 7 9	5 6 10 11	1 2 3
5	7 9	5 6 10 11	1 2 3 4
6	7 9	6 10 11	1 2 3 4 5
7	7 9	10 11	1 2 3 4 5 6
8	9	10 11	1 2 3 4 5 6 7
9	空	10 11	1 2 3 4 5 6 7 9
10	空	空	1 2 3 4 5 6 7 9 10 11

这个算法把每个元素恰好处理一遍，因此时间复杂度为 $O(n+m)$。利用递归的思想，可以用这个方法来设计一种新的排序算法：归并排序。

归并排序算法　算法思想是从中间把数组分成元素个数尽量相等的两半，分别对它们进行排序，然后合并两个有序表。算法的平均，最坏时间复杂度均为 $O(n\log n)$（还记得 1.2 节中介绍的主定理吗？可以用它直接得到这个结论）。归并排序的时间复杂度和刚才介绍的插入排序以及选择排序相比，时间复杂度已经降低了一个档次。但是大家也许听说过一种算法叫做“快速排序（Quick Sort）”。既然叫这个名字，这个算法必然有它的独到之处。下面，我们通过一个小故事来引入这个算法。

傻瓜 Ivanushka 的故事[①]

很久很久以前，有一个城堡里住着沙皇和他的女儿 Vasilisa。Vasilisa 美丽、聪明、善良，因此前去求婚的年轻小伙子络绎不绝。遗憾的是 Vasilisa 看不上他们之中的任何一人，因此所有求婚者都失望而归。沙皇终于有一天忍受不了女儿的挑剔，于是向全城人宣布：“谁最先解决了我的难题，我就把女儿嫁给他！”。

和所有其他人一样，傻瓜 Ivanushka 很想试一试。他来到沙皇那里，沙皇给了他一个张印满奇怪符号的纸：“这里有一个神奇的程序，它会告诉你你未来的新娘。只要你输入 N 个数，答案自然会显示出来。我给你一天的时间，明天的这个时候你再来见我，告诉我你选择的数。”

望着这堆奇怪的符号和数字，Ivanushka 傻了眼。他根本不知道这个程序是什么意思，可是一天的时间眼看就要过去了，你能帮帮他吗？

这就是沙皇写的 C 程序：

```
#include <stdio.h>
#include <fstream.h>
long c,A[N];
long P(long l, long r){
```

① 题目来源：Ural State University Problem Archive

```
    long x=A[l],i=l-1,j=r+1,t;
    while(1){
       do{--j; ++c;
       }while(A[j]>x);
       do{++i; ++c;
       }while(A[i]<x);
       if(i<j){
          t=A[i]; A[i]=A[j]; A[j]=t;
       }else
          return j;
    }
}

void Q(long l, long r){
    long n;
    if(l<r){
       n=P(l,r);
       Q(l,n);
       Q(n+1,r);
    };
}

int main(void){
    c=0;
    for(long i=0; i<N; ++i) fin>>A[i];
    Q(0,N-1);
    if(c==(N*N+3*N-4)/2) fou<<"Beutiful Vasilisa";
    else                 fou<<"Immortal Koshcei";
    return 0;
}
```

快速排序的思想　不知道 Ivanushka 有没有看懂，C 程序实质是个快速排序算法。快速排序与刚才介绍的合并排序一样都是基于分治模式之上，总是先将数列分成非空的两组从而将问题规模缩小，然后对两个数组分别递归调用排序过程；不同的是快速排序要求第一个数组中的元素不大于第二个数组中的元素，而合并排序不要求；所以分治解决子问题后，合并排序需要一个合并两组元素的过程，而快速排序是原地有序的，因而不需要。

快速排序的主过程　快排主程序伪代码如下，这实际就是上述 C 程序中的 `void Q(long l, long r)`过程：

过程 Qsort（实参：数列起始位置 $p1$，终止位置 $p2$）{

　如果 $p1<p2$，那么{

　　1. 把数列分成 $A_{p1\sim p3}$ 和 $A_{p3+1\sim p2}$；

2．递归调用过程 Qsort($p1$，$p3$)；

3．递归调用过程 Qsort($p3$+1，$p2$)。

};

}

快速划分法　可见，快排的核心算法是将数列分成第一组所有元素不大于第二组所有元素的两列数。下面就给出核心算法的伪代码，这实际也是上述 C 程序中的 long P(long l, long r)函数：

函数 Partition（实参：数列起始位置 $p1$，终止位置 $p2$）{

1．找一个数列划分的参照标准 x=A_{p1}，使得第一组所有元素不大于 x，第二组所有元素不小于 x。

2．划分采用两头往中间逼近的算法，即从数列尾开始往前找出第一个小于 x 的元素 A_J，再从数列首开始往后找出第一个大于 x 的元素 A_I，交换元素 I 和元素 J，然后重复刚才的过程，使得 I 前方的元素总不大于 x，J 后方的元素总不小于 x，直到位置 $I \geq J$ 划分过程就结束了。此时返回值 J，J 即是主程序中的 $p3$。初始时，I=$p1$−1，J=$p2$+1。

While（true）{

Repeat{J=J−1}直到 $A_J \leq x$；

Repeat{I=I+1}直到 $A_I \geq x$；

If I<J then 交换元素 A_I 和 A_J；

else 返回值 J，退出循环，结束划分；

}

}

为什么在函数 Partition 中不返回值 I 呢？因为在主过程中，是对 $A_{p1\to p3}$ 和 $A_{p3+1\to p2}$ 进行递归调用的，而不是 $A_{p1\to p3+1}$ 和 $A_{p3\to p2}$。返回 J 值可以保证数列的规模总在严格地缩小。

快速排序的时间效率　快速排序的最坏运行情况是 $O(n^2)$，比如说顺序数列的快排。但它的平摊期望时间是 $O(n\log n)$，且 $O(n\log n)$记号中隐含的常数因子很小，比复杂度稳定等于 $O(n\log n)$的合并排序要小很多。所以，对绝大多数顺序性较弱的随机数列而言，快速排序总是优于合并排序。

其实，可以通过加入随机函数来阻止最坏运行情况的出现。为什么每次调用函数 Partition 都要严格取 x=A_{p1} 作为划分标准呢？完全可以加入随机函数，随机的让 A_{p1} 和 $A_{p1\to p2}$ 中的某个元素交换，这样被划分出来的两个子数列大小容易接近，划分容易平衡。一旦划分平衡了，快排平摊时间就远离 $O(n^2)$而靠近 $O(n\log n)$。（事实上，加入随机以后的递归式就是 1.2.3 节中的式子 h）

还有一点点关于快速排序的优化，就是当数列规模较小时，划分的平衡性容易被打破，而且关键是频繁的过程调用耗费的时间已超过了 $O(n\log n)$为 $O(n^2)$省出的时间。所以规定，当数列规模<N 时，改用冒泡排序。实践证明，N 取 20 效果较好。

归纳起来，得到：

快速排序算法　算法思想是找一个数 x，调整数组里的元素使得 x 左边的数都比 x 小，右边的数都比 x 大，然后递归给 x 左边的所有数和右边的所有数分别进行排序。

讲了这么多，现在回过头来看看可怜的傻瓜 Ivanushka。我们主要想通过 Ivanushka 的问题向大家阐述一下快速排序，而问题本身并没有多少价值。简单尝试一下发现，递增的顺序数组可以满足 c=($N\times N+N\times 3-4$)/2 的要求，这可以用数学归纳法证明：

定义函数 F(N)= ($N\times N+N\times 3-4$)/2，

可知 F(1)= ($1\times 1+1\times 3-4$)/2，

如果 F(k)= ($k\times k+k\times 3-4$)/2，

那么 F(k+1)=指针 I 的移动次数+指针 J 的移动次数+F($p3$)+F($k-p3$)

$$=1+k+1+F(1)+F(k)$$

$$=((k+1)\times(k+1)+(k+1)\times 3-4)/2,$$

所以递增顺序数组的 C 值=($N\times N+N\times 3-4$)/2。

给傻瓜 Ivanushka 一个 1，2，3，…，N 的数列，他就可以迎娶美丽的公主 Vasilisa，可见快速排序是很有必要掌握的。

【例题 3】士兵排队[①]

有 N($N\leqslant 10\ 000$)个士兵分散在一个网格形的土地上。网格土地上每一个格子位置由一个整数坐标给出，士兵可以移动，每一次移动可以向上、向下、向左或向右移动一个单元（即他可以使 X、Y 坐标增加 1 或减少 1）。

这些士兵最后要进入某一行且排在一起（即他们的最终位置为(x,y)，(x+1,y)，…，(x+N−1,y)，点(x,y)是任意的），最少需要多少步？假设格子很大，同时可以站无限多名士兵。

【分析】

由于格子无限大，只需要确定每个士兵最终位置，答案与他们具体行走的方式无关。由于每个士兵每次只有四个方向可以走，因此行和列位置可以分开考虑。

首先考虑行。因为所有士兵站在同一行，因此这一行应该是所有士兵的行坐标的**中位数**。再考虑列，由于都站在一起，所以只要确定了中间一个士兵的位置就可以了。事实上，最好的方案是，中间那个士兵不动。如果有偶数个士兵，中间两个士兵任选一个不动都可以（请读者证明这一点）。这样，行和列分别求出**中**位数，这个问题就可以得到解决了。

可是，中位数应该怎样求呢？

快速选择算法　借用快速排序的思想，可以解决一个更容易的问题，选择无序表中第 k 小的元素。更特别的，在刚才那道题目中需要知道“位置处于中间的数”，即中位数（如果元素有偶数个，那么中间的两个元素都是）。方法很简单：在快速排序的第一部结束后，不要把两个子序列分别排序，而是根据两个序列的元素个数判断第 k 大的数在哪个序列中，然后递归找那个序列的第 k' 大元素。

① 题目来源：CEOI 1998

WEB 在随机划分的情形下，该算法的期望时间复杂度是线性的。其实还有更好的确定性算法，它保证时间复杂度一定是线性的，进一步讨论放在本书主页上。

排序网络、0-1 原则　在 1.1 节中我们曾提到过：如果可以同时比较多组数，排序时间将大幅度降低。在这里，介绍排序网络，并引出 0-1 原则。回忆习题 1.3.25 介绍的比较网络。它可以对一组输入得到一组输出。如果对于任何一组输入得到的输出均是从小到大排序好的，那么称这样的比较网络是一个**排序网络**。图 1-44 所示的比较器就是一个排序网络。

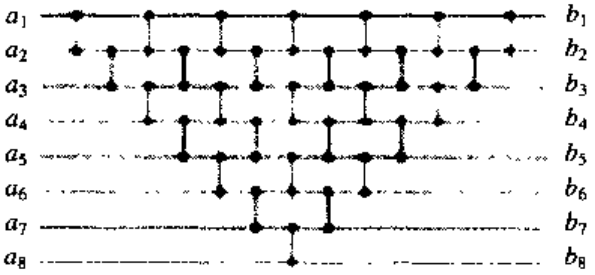


图 1-44　排序网络举例

排序网络的研究主要是设计和验证两个方面。这里不介绍它的设计问题，而只考虑如何验证一个比较网络是否为排序网络。一个最简单的方法是枚举输入数据的 $n!$种大小关系，枚举次数为 $O(n!)$。下面的定理可以把枚举次数减少到 $O(2^n)$。

0-1 原则：一个比较网络是排序网络当且仅当它可以正确的把任意只由 0 和 1 组成的输入序列排成升序。

定理的证明并不困难，有兴趣的读者可以自己试一试。

最后，用一个例题来结束本节，它和排序网络有一种特殊的联系。它的解题过程也是很有启发性的。

【例题 4】最小可靠交换[①]

给定 n 个整数寄存器 $r1$，$r2$，…，rn，定义一个比较-交换指令 CE(a,b)，a、b 是寄存器序号（$1 \leqslant a < b \leqslant n$）：

CE(a,b):: if 内容(ra)>内容(rb) then 交换寄存器 ra 和 rb 的内容。

一个比较-交换程序（简称 CE 程序）是任意有限的比较-交换指令序列。如果运行一个 CE 程序后寄存器 $r1$ 的值总是所有寄存器的值中最小的，这个 CE 程序叫作最小值查找程序。如果从一个最小值查找程序里删除任何一个比较-交换指令后这个程序仍然是最小值查找程序，那么它是可靠的。

给定一个 CE 程序 P，最少应该在 P 的末尾添加多少条指令才能得到一个可靠的最小值查找程序？例如，考虑这个为 3 个寄存器设计的 CE 程序：CE(1,2)；CE(2,3)；CE(1,2)。为了使它变成一个可靠的最小值查找程序，加上指令 CE(1,3)和 CE(1,2)就足够了。

【分析】

如果执行完毕缺少任意一条指令的 CE 程序后，最小值不可能存在于寄存器 i，那么称

[①] 题目来源：ACM/ICPC Regional Contest CERC 2001

i 是可靠的；否则，如果执行完毕完整的 CE 程序后，最小值不可能存在于寄存器 i,那么称 i 是连通的；否则，执行完毕完整的 CE 程序后，最小值仍有可能存在于寄存器 i，那么称 i 是孤立的。

对于可靠寄存器 i，无需对它进行任何额外的操作；对于连通寄存器 i，因为需要增加至少一条指令使其变成可靠的，所以不如认定该增设指令为（1，i）；对于孤立寄存器 i，因为需要增加至少两条指令使其变成可靠的，所以不如认定该增设指令为（1，i）和(1，i)。

明确了这些，我们的任务就很明了了，就是确定每个寄存器的类型。

初始时，寄存器 1 是可靠的，其余寄存器是孤立的。随后逐条执行 CE 指令，在每条指令（a,b）后研究该指令对寄存器类型造成的变化。

- 对寄存器 a 的影响

（1）如果 a = 1，那么无影响，因为寄存器 1 永远是可靠的。否则——

（2）如果 a 是孤立的，那么无影响。因为如果最小值在 a，那么增加指令(a,b)后，最小值仍然在 a。否则——

（3）如果 b 是可靠的，那么无影响。因为最小值一定不在 b,所以（a,b）是一条做无用功的废指令。否则——

（4）如果 b 是孤立的，那么 a 会变成孤立寄存器。因为如果最小值在 b,那么现在最小值就一定在 a。否则——

（5）如果 b 是连通的，那么 a 会变成连通寄存器。考虑当前指令(a,b)前的所有指令，如果完整地执行这些指令，那么最小值不在 a；如果删除某条指令使得最小值在 b，那么执行当前指令（a,b）后，最小值就一定在 a。根据寄存器类型的定义，现在的 a 是连通寄存器。

- 寄存器 b 的影响

对寄存器 b 的影响简单得多：如果 b 是孤立的，那么现在它是连通的；如果器 b 是连通的，那么现在它是可靠的；如果器 b 是可靠的，那么现在它仍然可靠。

知道了寄存器 i 的类型，也就知道需要增设几条(1,i)指令。将所有寄存器增设指令的条数相加，即是问题最终的答案。算法的空间复杂度和时间复杂度均是最优的。

小知识——排序问题的复杂程度

利用判定树模型，可以证明“在基于比较的排序中，时间复杂度的下界是 $O(n\log n)$”，可以简要证明如下，在没有比较的时候，排序的结果有 $n!$种情况。第一次比较，由于有两种结果，所以最坏情况下最终结果还有 $n!/2$ 种可能性。K 次比较以后最终结果还有 $n!/2^K$ 种可能性。当 $n!/2^K<2$ 时结果才被确定下来，这时比较了 $\log_2(N!) = O(n\log n)$次。在这个意义下，归并排序和快速排序是最优的。

不过，如果排序不是基于比较的，而且关键字的分布比较集中，那么可以得到线性时间复杂度的算法，在后面会加以介绍。

如果是对实数进行比较有 $O(n\log\log n)$的算法，有兴趣的读者可以阅读相关论文。

练　习　题

思考题：

1.3.27　给出一个简单图各个顶点的度数，设计一个时空复杂度均为 $O(n)$的算法给这些度数排序。

1.3.28　关于比较网络：

（a）证明一个比较网络可以把 $n, n-1, n-2, \cdots, 1$ 排成升序当且仅当它可以把 1,0,0,…,; 1,1,0,0,…,0;1,1,1,0,0,…,1,1,1,1,1,0,…,0 等 n 个序列排成升序。

（b）给一个排序网络，证明对于每个 $1 \leqslant i < n$，该网络至少应该包含一个比较第 i 和第 i+1 个元素的比较器。

（c）如果一个序列先递增，再递减，或先递减，再递增，称该序列是 bitonic 的（特殊的，单调序列也是 bitonic 的）。证明一个比较网络能给任意 bitonic 序列排序当且仅当它能给任意 bitonic 的 0-1 序列排序。把这样的网络称为 bitonic 排序网络。

（d）设计一个 $O(n^2)$的算法，判断一个比较网络是否为 bitonic 排序网络。

（e）如果一个比较网络可以把两个长度为 n 的已排序序列合并成一个已排序序列，称它是一个合并网络。给出和 0-1 原则的类似结论判定一个比较网络是否为排序网络。用这个结论设计的判定算法时间复杂度为多少？

编程题：

1.3.29　千年庆典—— 赤道大篝火[1]

有一些想法虽然绝妙绝伦，却没办法实施。“千年庆典 —— 赤道大篝火”就是这样一个例子。或许这个想法会在下一个千年被重新发现，但是我希望在它没有被彻底遗忘之前让更多的人知道它：把涂了焦油的圆木撒上火药粉后铺满整个赤道，然后在不同地方于不同的时刻点燃篝火，看着火焰沿着赤道向两个方向慢慢地蔓延，直到熊熊火焰布满整个赤道。

这项伟大计划的设计者最想知道的，就是赤道上哪个地方会在什么时候最后燃起烈火。你能告诉他吗？

1.3.30　鸡蛋[2]

人所共知，鸡蛋从高处摔下会摔得粉碎。对通常的鸡蛋来说一楼就足够了。但是某些母鸡下的蛋可能在 100 000 000 层摔下仍能完好无损。因此，对鸡蛋强度的测试打算在摩天大楼上进行。对鸡蛋的强度定义是：如果某种鸡蛋在第 k 层摔不碎，但是在第 $k+1$ 层会摔碎，那么该种鸡蛋的强度就是 k。如果摩天大楼有 n 层高，我们认为任何鸡蛋在第 0 层都摔不碎，第 $n+1$ 层都会摔碎。

---

[1] 题目来源：Internet Problem Solving Contest 2001

[2] 题目来源：Polish Olympiad in Informatics

实验室的管理员打算削减试验经费，他限制了为了确定所给出的这种鸡蛋的强度所能摔破的鸡蛋的最大个数，而且必须用最少的试验次数得出鸡蛋的强度。也就是说给了一定数量的同种鸡蛋和一座摩天大楼，必须在最少的试验次数后确定这种鸡蛋的强度。

不要认为鸡蛋的强度在试验开始之前就已经确定了。测试程序会修改鸡蛋的强度使你的程序所用的试验次数最多而且它不会违反你的程序已经提出的任何问题。所以，请优化你的程序使其在最坏的情况下试验的次数最少。

1.4　数据结构（2）——拓宽和应用举例

前面说过，数据结构的基本内容是表示和操作。前面介绍了栈、队列、树、二叉树、图等表示方法，本节围绕这些常用的操作来介绍一些新的数据结构。

本节中的东西虽然很基本，但是在讲解的过程中作了不少补充和扩展。这些新的内容在同类书籍中出现得不多，希望读者用心体会。

和同类书籍不同，我们在顺序安排上不按照任何分类标准，不追求系统化，而是从实用的角度出发。每个小节介绍一种有特殊用途的数据结构，先介绍理论和实现，再举例子。本节的所有内容都是重点，实用性很强而且很快就要被 1.5 节和 1.6 节用到。本节的难点是二分检索树和线段树的应用。堆、并查集和哈希表的应用范围及其广泛，希望读者给予充分的重视。Trie 结构和后缀树介绍得不多，但实际上它们都是很有用的。

在介绍具体的数据结构之前，希望读者明白几个基本概念：

常见的操作　插入（insert）是指在数据结构中增加一个数据。有的数据结构并不需要插入结点；删除（delete）是指在数据结构中删除一个数据。对不少数据结构来说删除是比较麻烦的，好在很多时候需要插入而不需要删除，而只需要根据联机输入插入结点。不需要插入和删除的数据结构称为静态数据结构。这样的数据结构我们往往牺牲一些预处理时间和附加空间来换取更好的性能而不必担心维护的复杂度。查找（search）是指在数据结构中查找指定的数据。大部分数据结构都需要检索，而专门为检索方便设计的数据结构也有很多，将在后面介绍。

时间效率和空间开销的关系　这个关系是很有趣的，它们有时候是一对矛盾体的两个方面，我们常常利用时间去换空间，或者用空间换时间。后面会看到一些平衡两者复杂度的例子。有时候，时间效率和空间开销有时候又是统一的。我们在精简了数据表示的方式，缩减了空间的同时，往往可以更有效地设计更加快速的基本操作。

不同操作的对立统一　除了时空这一对矛盾之外，不同操作之间也常常是对立统一的。例如，常常需要维护一种动态数据结构，不断地改变里面的内容，还不时地查询里面是否有我们需要的元素。这样的数据结构是利于检索的，如果附加记录一些统计信息，还可以用来做数据统计。如果用有序数组，插入的平均时间复杂度为 $O(n)$，而查找的时间复杂度为 $O(\log n)$；如果用链表，插入的时间复杂度变为 $O(1)$，但是查找的时间复杂度升为 $O(n)$。能不能让两者的时间复杂度都比较低呢？后面将解决这一问题。

可扩充性和实现复杂度　需要考虑的另外一个问题是数据结构的可扩充性与实现复杂度。可扩充性在“并查集”、“二叉搜索树”中都有体现，大家可以仔细体会。实现的复杂度在本节中强调得并不多，但是它确实是一个很实际的问题，在学习一些比较复杂的数据结构的，读者可以注意一下它的实现细节。

1.4.1　并查集

设想需要对**不相交集合（disjoint set）**进行两种操作：（1）检索某元素属于哪个集合；（2）合并两个集合。我们最常用的数据结构是并查集的森林实现（其他实现如链表实现在编程和效率上都不如森林实现，这里不再叙述）。也就是说，在森林中，每棵树代表一个集合。用树根来标识一个集合。

合并操作　为了把两个集合 S_1 和 S_2 并起来，只需要把 S_1 的根的父亲设置为 S_2 的根就可以了。这里有一个优化：让深度较小的树成为深度较大的树的子树，这样查找的次数会少些。这个优化称为启发式合并。可以证明，这样做以后树的深度为 $O(\log n)$（请读者思考）。

查找操作　查找一个元素 u 也很简单，只需要顺着叶子到根结点的路径找到 u 所在树的根结点，也就确定了 u 所在的集合。这里又有一个优化：找到 u 所在树的根 v 以后，把 u 的父亲设置为 v，这样也会减少查找次数。**这个优化称作路径压缩。**在使用路径压缩以后，由于深度经常发生变化，因此我们不再使用深度作为合并操作的启发函数值，而用一个称为 rank 的数。刚建立的新集合的 rank 为 0，以后当两个 rank 相同的树合并时，随便选一个棵树拥有新根，并把它的 rank 加 1；否则 rank 大的树拥有新根，两棵树的 rank 均不变。

时间效率　Tarjan 在 1975 年证明了，这样的并查集算法的时间复杂度为：$O(m\alpha(m,n))$（执行 $n-1$ 次合并和 $m \geq n$ 次查找）。$\alpha(n)$是 Ackermann 函数的某个反函数，它可以近似的看成是小于 5 的。这样，查找也几乎是线性的。还记得 1.3 节中介绍的 LCA 问题的 Tarjan 算法吗？用并查集的森林表示法来实现那些操作，则该算法的总时间复杂度近似为 $O(n+Q)$，其中 Q 为询问次数。

【例题 1】代码等式[1]

每一个由元素 0 和 1 组成的非空的序列称为一个二进制代码。一个代码等式就是形如 $x_1x_2\cdots x_i=y_1y_2\cdots y_j$，这里 x_i 和 y_j 是二进制的数字（0 或 1）或者是一个变量（如英语中的小写字母）。每一个变量都是一个有固定长度的二进制代码，它可以在代码等式中取代变量的位置，称这个长度为变量的长度。为了解一个代码等式，需要给其中的变量赋予适当的二进制代码，使得我们用它们替代代码等式中的相应的变量后，这个等式成立。

对于每一个给出的等式，计算一共有多少组解。

例如，*a,b,c,d,e* 是变量且它们的长度分别是 4,2,4,4,2。考虑等式：1*bad*1 = *acbe*，这个等式共有 16 组解。变量最多有 26 个，且等式每一端的数字和变量的长度和不超过 10 000。

【分析】

首先，把每个长度为 k 的变量分拆为 k 个长度为 1 的变量，则结果不变。这样，就得

[1] 题目来源：Polish Olympiad in Informatics

到了 L 个等式（L 为等号两边的长度）。每个等式都告诉我们某两个变量相等，因此得到的是一系列等价关系。用并查集很容易在近似 $O(L)$的复杂度下找出所有的等价类。只要不包含 0 或者 1 的等价类取值均有两种情况，因此设等价类的个数为 p，则 2^p 为所求。当然，甚至可以不用找出所有等价类再计数，只需要在合并不同集合时把等价类个数减 1 即可。另外，需要注意的是矛盾情况以及两边位数不等需要用 0 补齐的情况。但不管怎么说，本题已经成功解决了。

【例题 2】团伙[1]

在某城市里住着 n 个人，任何两个认识的人不是朋友就是敌人，而且满足：

1．我朋友的朋友是我的朋友

2．我敌人的敌人是我的朋友

所有是朋友的人组成一个团伙。告诉你关于这 n 个人的 m 条信息（即某两个人是朋友，或某两个人是敌人），请你计算出这个城市最多可能有多少个团伙。

【分析】

本题看起来很简单，但是很容易就会设计出错误的算法。由第一条性质可以知道朋友具有传递性，因此可以把一个团伙看成一个等价类。但是第二条“我敌人的敌人是朋友”看起来有点令人疑惑，这样看来，似乎只能有两个团伙了。即某人 i 的朋友组成一个团伙，他的敌人们组成另一个团伙。但是不要忘了，i 还有不认识的人，所以我们的任务是把这些不认识的人分成尽量多的团伙。

对于只有朋友信息的数据，可以对于每个朋友信息，做一次集合合并。但是对于敌人信息，又该如何处理呢？刚才说过了，一个人的所有敌人一定在同一个团伙中，因此只需要给每个集合 i 记录它的“敌人集合”e[i]，当收到 i 和 j 是敌人的信息后，把 i 和 e[j]合并，j 和 e[i]合并，注意正确的处理和空集合并、自己跟自己合并这两种特殊情况。最后，等价类的个数就是所求答案。

刚才两道例题有个共同的特点：并没有充分利用到并查集的“查询”操作，而只利用了“并”。这样做很大程度上是为了编程方便和思路清晰。后面几道题目充分地利用到了并查集的查询能力，希望读者细心体会。

【例题 3】银河英雄传说[2]

宇宙历七九九年，银河系的两大军事集团在巴米利恩星域爆发战争。泰山压顶集团派宇宙舰队司令莱因哈特率领十万余艘战舰出征，气吞山河集团点名将杨威利组织麾下三万艘战舰迎敌。

杨威利擅长排兵布阵，巧妙运用各种战术屡次以少胜多，难免恣生骄气。在这次决战中，他将巴米利恩星域战场划分成 N（$N \leqslant 30\,000$）列，每列依次编号为 1，2，…，N。之后，他把自己的战舰也依次编号为 1，2，…，N，让第 i 号战舰处于第 i 列(i = 1，2，…，N)，形成“一字长蛇阵”，诱敌深入，这是初始阵形。当进犯之敌到达时，杨威利会多次发布合并指令，将大部分战舰集中在某几列上，实施密集攻击。合并指令为 $M\ i\ j$，含义为

[1] 题目来源：Baltic Olympiad in Informatics, 2003

[2] 题目来源：NOI2002, Galaxy. 命题人：骆骏

让第 i 号战舰所在的整个战舰队列，作为一个整体（头在前尾在后）接至第 j 号战舰所在的战舰队列的尾部。显然战舰队列是由处于同一列的一个或多个战舰组成的。合并指令的执行结果会使队列增大。

然而，老谋深算的莱因哈特早已在战略上取得了主动。在交战中，他可以通过庞大的情报网络随时监听杨威利的舰队调动指令。

在杨威利发布指令调动舰队的同时，莱因哈特为了及时了解当前杨威利的战舰分布情况，也会发出一些询问指令：$C\ i\ j$。该指令意思是，询问电脑，杨威利的第 i 号战舰与第 j 号战舰当前是否在同一列中，如果在同一列中，那么它们之间布置有多少只战舰？

作为一个资深的高级程序设计员，你被要求编写程序分析杨威利的指令，以及回答莱因哈特的询问，所有指令总数 K 不超过 500 000。

【分析】

初始时，每条战舰作为一个独立的作战队列，随着 M 命令的发布，分离的队列不断合并。考虑一下问题的规模 N 条战舰 K 条指令，这就要求使用一种高效的数据结构，对于每条指令可以在较短的平摊时间内将两个子集合并成一个集合，建立它们之间的关系，同时保持原各子集合内部的关系不变。分析至此得知，这是道典型的关于不相交可并集合的试题，应当使用并查集数据结构。

不过本题需要得到一些额外的信息，不得不对并查集进行扩充：开设一个长度为 n 的数组，对每条战舰 i 分配 3 个变量 $a[i]$、$b[i]$、$c[i]$，含义如下：

$a[i]$表示战舰 i 属于以战舰 $a[i]$为首的队列，称战舰 $a[i]$为战舰 i 的根；

$b[i]$表示战舰 i 到战舰 $a[i]$(包括战舰 $a[i]$)之间的战舰数量，称 $b[i]$为战舰 i 到战舰 $a[i]$的深度；

$c[i]$表示战舰 i 所属队列后方（包括战舰 i 本身）的战舰数量（注意：此变量只有当 $a[i]=i$ 时才有意义），称 $c[i]$为队列长度。

初始时，$a[i]=i$，$b[i]=0$，$c[i]=1$。对于命令“$M\ i\ j$”，从 i 出发向前寻找 i 的根，直到找到真根 Root，即 a[Root]=Root；在寻根的同时，累加计算战舰 i 到真根的深度。找到真根后，又一次遍历从 i 到真根 Root 的路径，进行路径压缩。对于在遍历中经过的每一个战舰 i（假设战舰 i 在修改前根为 a_1，深度为 b_1），根据已经修改过的战舰 i 的 a 值和 b 值对战舰 a_1 的 a 值和 b 值进行修改，令 $a[a_1]$=Root，$b[a_1]=b[i]-b_1$。用同样方法处理战舰 j。最后进行战舰 i、战舰 j 所在队列的合并，假设 $a[i]=R1$，$a[j]=R2$，那么需要做的修改只是令 $a[R1]=R2$，$b[R1]=c[R2]$，$c[R2]=c[R2]+c[R1]$。

对于命令“C i j”，分别对战舰 i、战舰 j 进行路径压缩，然后判断是否 $a[i]=a[j]$，如果是那么输出$|b[i]-b[j]|-1$。根据前面关于并查集复杂度的结论，此题算法复杂度大约为 $O(K)$。

【例题 4】可爱的猴子[1]

树上挂着 n 只可爱的猴子，编号为 1，…，n（$2\leqslant n\leqslant 200\,000$）。猴子 1 的尾巴挂在树上，每只猴子有两只手，每只手可以抓住最多一只猴子的尾巴，也可以不抓。所有猴子都是悬空的，因此如果一旦脱离了树，猴子会立刻掉到地上。第 0，1，…，m（$1\leqslant m\leqslant 400\,000$），

[1] 题目来源：Polish Olympiad in Informatics, 2003

秒中每一秒都有某个猴子把他的某只手松开，因此常有猴子掉在地上。请计算出每个猴子掉到地上的时间。

【分析】

不管是谁抓住谁的尾巴，只需要记录猴子 i 和猴子 j 是否直接相连，那么本题就是要求每只猴子第一次脱离猴子 1 的时间。这是并查集似乎是相反的，需要把并查集改造成“分查集”吗？不需要，其实只需要把输入反过来。首先算出所有放手行为执行以后猴子连接的情况，然后从第 m 秒开始进行“时光倒流”，每次把一只猴子的手接上，那么问题转变为了每只猴子第一次是什么时候并入猴子 1 所在的集合的。具体来说，如果在第 t 秒时，猴子 i 把它的某只手 j 重新抓住猴子 k 时，把 i 所在的集合与 k 所在的集合合并。如果 i 与 1 同在一个集合，那么 k 当前所在集合的所有猴子的掉落时间均为 t。需要枚举 k 所在集合的所有元素，把它们的答案均设置为 t。但问题是，并查集并没有提供这个操作。解决方法很简单，同时给每个集合做一个链表，查找操作在森林中进行，合并操作同时在森林和链表中进行。时间复杂度保持不变，而枚举操作的时间复杂度变为了 O（元素个数）。由于每个元素最多只被枚举一次，故总的时间复杂度仍近似为 $O(n+m)$。

提醒读者注意，被认为比较“差”的数据结构有可能提供了被人忽视的特殊操作。

【例题 5】蜗牛[①]

有 n 只蜗牛，爬行的速度分别为 1，2，3，…，$n(1\leqslant n\leqslant 1\,000\,000)$。每次把井盖打开，要么丢一只蜗牛进去并立刻盖上井盖，要么等待一只蜗牛爬到井口，把它拿出并后立刻盖上井盖。蜗牛爬出井后不会再次被放入。当井盖盖上后，由于井里一片漆黑，蜗牛感到很失望并迅速滑落回井底。执行了 $m(m\leqslant n)$次操作（即放蜗牛或者等蜗牛爬出），请编程求出蜗牛爬出井的顺序。

【分析】

显然，需要维护一个数据结构，它支持两个操作：插入（蜗牛）和删除最大值（爬得最快的蜗牛将爬出）操作，并回答每次对最大值的询问。既然又是询问问题，有在线算法和离线算法两种。$O(m\log n)$时间的在线算法将在下小节给出，本节只考虑一个近似为 $O(n+m)$的离线算法。我们把注意力放在所有的取最大值操作 MAX 上。先在最后加入 $n-m$ 次虚拟的取最大值操作，然后记第一次 MAX 操作之前的插入操作序列为 S_1，第 $i-1(1<i\leqslant m)$次 MAX 操作和第 i 次 MAX 操作之间的插入操作序列为 S_i。

记第 i 个 MAX 操作为 M_i，则如果蜗牛 n 在 S_j 中，那么 M_j 的输出一定是 n（想一想，为什么？）。既然 M_j 的输出已经确定，那么应该删除 M_j，即把 S_j 合并到 S_{j+1} 中，然后再查找蜗牛 $n-1$ 所在的序列 S_k，则 M_k 的输出为 $n-1$…如此下去，从 n 到 1 依次查找每个蜗牛所在序列，就可以得到它后面的 MAX 操作的结果并把它和紧随其后的序列合并。用并查集来实现序列 $S_1\cdots S_n$，则可以在几乎 $O(n+m)$的时间内得到结果。

专题六——均摊分析

在介绍并查集的时间复杂度时，直接给出了结论：执行 $n-1$ 次合并和 $m\geqslant n$ 次查找，时间复杂度为 $O(m\alpha(m,n))$。为什么要这么说呢？为什么不说每次合并和查找需要用的时间

[①] 题目来源：经典问题

呢？这个结论是**均摊分析**（**amortized analysis**）得到的。它的基本思想可以通过一个简单的例子来说明，读者应仔细区别均摊时间复杂度和**平均情况**（**average-case**）时间复杂度差异。

二进制计数器问题：有一个 k 位二进制计数器只支持加一操作，该操作的时间复杂度如何？我们把改变一个二进制位看作常数时间。

显然最坏情况下（k 个 1 再加 1），所有位都需要改变，因此最坏情况时间复杂度为 $O(k)$。但是实际上由于计数器在不断地加一，根本不可能遇到连续若干次最坏情况。因此应该分析一个操作序列的总时间，而不是单个操作的时间。这样的分析就是均摊分析。平均情况时间复杂度是针对不同输入来计算平均值的，均摊分析是一个序列的所有操作取平均值，二者是从不同方面上来说的。均摊分析的结果可以是最坏情况，也可以是平均情况。

第一种方法为**累积分析**（**aggregate analysis**），即先计算 n 个操作的总时间 $T(n)$，然后每个操作的均摊时间复杂度就是 $T(n)/n$ 了。刚才的结论是一个长度为 n 的序列的总时间为 $O(kn)$，但是事实上最坏情况是 $O(n)$。因为第 0 位每次变化，因此一共变化 n 次；第 1 位每两个操作变一次，一共变 $n/2$ 次……第 k 位变化 $n/2^k$ 次，故总变化数小于 $2n$ 次。每次操作的均摊复杂度为 $O(n)/n = O(1)$。

第二种方法为**会计分析法**（**accounting method**）。有的数据结构支持多种操作，那么累计分析就无法计算出一个确定的值，但是会计分析法仍然有效，它把每个操作的实际时间消耗看做资金消耗（时间比做金钱！），而另外定义这种操作的“投资额”，供以后操作的使用。如果可以保证始终有可用资金，那么实际消费的资金不会超过投资总量，因此算法的时间耗费（即资金消耗）的上限为总投资额。

在刚才的例子中，假设把任何一个 0 变成 1 的操作时投资 2 美元，而把 1 变成 0 时不进行投资，那么实际情况是这 2 美元中的一个当时就会用掉，而另一个可以提供给这个 1 变回到 0 时使用。这样，在任意时刻，可用资金的数目等于 1 的数目，因此它总是非负的。由于每次最多只有一个 0 变成 1（但可能有多个 1 变成 0），因此投资总量不超过 $2n$ 美元。虽然每改变一位都要用掉一个美元，但是由于刚才证明了始终有可用资金，因此资金消耗不超过 $2n$ 美元，即：n 个操作的总时间不超过 $O(n)$。

请读者仔细体会这个方法。把实际的时间消耗看作资金消耗，通过定义合理的投资过程，把资金消耗限制在一个小范围内。因此，这个方法的关键是给各种操作设计合理的投资额，满足任何时间都有可用资金，而且该投资额应该尽量小（这样得到的上界才会更准确）。

另一种会计分析法在每次操作时给第 i 个位投资 2^i 美元，则它从上一次翻转起一共累计得到了 1 美元投资，正好够它这一次翻转，因此资金不会短缺。由于每次操作投资 2 美元，所以一共投资了 2 美元，因此总时间不超过 $O(1)$。

两种会计分析法是不同的：第一种是给操作划分成一些微操作，不同的微操作对应于不同的投资额；第二种是给操作的对象分成不同的类别，给不同的对象进行不同程度的投资。但是二者的思想都是一致的，用好计算的投资额去估计不好计算的资金消耗。

第三种方法叫做**势能分析法**（**potential method**）。它比前两种方法更普遍，但是也更

难学。它借用了物理学“势能”的概念，把“势能”看作整个数据结构的一个状态函数。用 D_i 表示经过 i 次操作以后的数据结构，Φ_i 表示 D_i 的势能，c_i 表示第 i 次操作（即把 D_{i-1} 变成 D_i 的操作）的实际时间耗费，定义第 i 次操作的均摊时间耗费 $a_i = c_i + \Phi_i - \Phi_{i-1} = c_i + \Delta\Phi$（此式为势能分析法的核心）。

这样，前 i 个操作的总均摊时间耗费为：$c_1+c_2+\cdots+c_n+\Phi_n-\Phi_0$。应当设势能函数 Φ 使得 $\Phi_0=0$，$\Phi_i\geqslant 0(i>0)$。这样，总实际时间耗费 sum$\{c_i\}$将小于总均摊时间耗费 sum$\{a_i\}$（请读者证明）。于是，均摊时间是实际时间的上限。

在刚才的计数器问题中，定义当前计数器的势能为它包含的 1 的个数，则 $\Phi_0=0$，$\Phi_i\geqslant 0(i>0)$均成立，势函数合法。进一步，令第 i 个操作把 a 个 0 变成 1，把 b 个 1 变成 0，则 $c_i = a+b$，势能增量为 $a-b$（请读者推导），因此 $a_i = 2a = 2$。因此每个操作的均摊复杂度为 2，请读者仔细体会这个方法。该方法的精妙之处在于把操作时间和它引起的势能变化结合起来考虑，得到容易计算的均摊时间复杂度。但是没有通用的方法设计势函数，导致此法相当灵活而不易掌握。简单地说，一个好的势函数需要在一个很快的操作式稍微增加一点，但是遇到一个耗时操作时能迅速下降。刚才的问题中，正是设计了这样一个函数。这个设计原则一定程度上允许我们通过对问题的感性认识和简单分析设计出一个合理的势函数。

【例题 6】基于 rank 的启发式合并和带路径压缩的并查集的均摊分析

【分析】

这里我们不给出最好的结论，而只证明 m 次操作，其中有 n 个是 MAKESET 操作时，运行总时间为 $O(m\log^* n)$。其中 $\log^* n$ 定义为最小的 i 使得 $\log^{(i)} n\leqslant 1$，$\log^{(i)} n$ 即 n 连续取 i 次以 2 为底的对数，例如 $\log^{(0)} n=n$，$\log^{(2)} n = \log\log n$。因此 $\log^*(2^{65\,535}) = 5$。由于目前几乎不可能接触到 $2^{65\,535}$ 这样大的数，因此假设总有 $\log^* n\leqslant 4$。无疑，它也是一个增长相当慢的函数！虽然理论上它比 α 函数增长快一些，但是从实用上来讲没有太大的区别。

首先，来分析 rank 的性质。对于不是根结点的元素，它的 rank 一定严格小于父亲的 rank。这一点用数学归纳法很容易证明。还可以证明如果根结点的 rank 为 r，那么整棵子树的大小 size$\geqslant 2^r$。因此，有：

结论一：对于大小为 n 的树，它的根的 rank 最多为$\lfloor\log_2 n\rfloor$。

对于任意整数 r，在操作执行过程中一旦有一棵树的根的 rank 从 $r-1$ 变到 r，标记该树的所有结点（由刚才的结论，我们至少标记了 2^r 个结点）。由于这棵树一旦有新根以后该根的 rank 至少为 $r+1$，因此每个元素最多被标记一次。由于每一个 rank 为 r 的元素都标记了 2^r 个结点，而结点一共有 n 个，因此有

结论二：rank 为 r 的元素最多有 $n/2^r$ 个。

为了后面分析方便，把所有元素分成若干个块，如果某个元素 i 的 rank 为 rank(i)，那么它被分到第 $\log^*$(rank(i))块。引入记号

$$2\uparrow\uparrow b = 2^{2^{\cdot^{\cdot^{2}}}}\Big\}b \Big\} b = \begin{cases} 1, & b=1 \\ 2^{2\uparrow\uparrow(b-1)}, & b>1 \end{cases}$$

则有元素 x 在第 b 块当且仅当

$$2\uparrow\uparrow(b-1) < \text{rank}(x)\leqslant 2\uparrow\uparrow b$$

这样，每一个块 b 里面元素最多有

$$\sum_{r=2\uparrow\uparrow(b-1)+1}^{2\uparrow\uparrow b}\frac{n}{2^r}<\sum_{r=2\uparrow\uparrow(b-1)+1}^{\infty}\frac{n}{2^r}=\frac{n}{2^{2\uparrow\uparrow(b-1)}}=\frac{n}{2\uparrow\uparrow b}$$

由于 rank 的最大值为$[\log_2 n]$，因此最大块的编号为 $\log^*(\log_2 n)=\log^* n-1$。

结论三：一共有 $\log^* n$ 个块，每个块 b 的元素最多有 $n/(2\uparrow\uparrow b)$个。

现在，可以用会计分析法来分析该并查集的均摊时间复杂度。由于 MAKESET 和 UNION 显然每次都只用常数时间，只需要证明 m 个 FIND 所花费的总时间是 $O(m\log^* n)$。在 FIND(x_0)的时候，假设从 x_0 开始经过序列 x_0，x_1，…，x_L 最后到达根结点 x_L。让这个序列中的每个元素投资 1 美元，但是不同的元素可能对不同的项目进行投资，如图 1-45 所示。

1．根 x_L 对“leader 项目”投资。

2．根的儿子 x_{L-1} 对“child 项目”投资。

3．x_i 不是根或者根的儿子，并且它和父亲结点 x_{i+1} 不属于同一块的对“block 项目”投资。

4．x_i 不是根或者根的儿子，并且它和父亲结点 x_{i+1} 属于同一块的对“path 项目”投资。

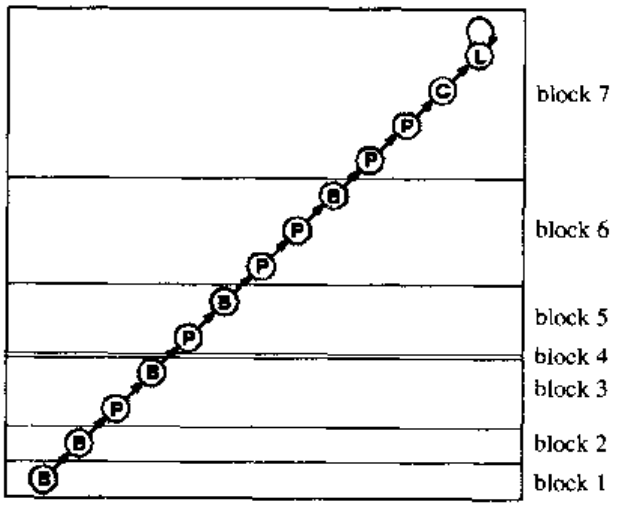


图 1-45　投资种类：B 为 block 项目，P 为 Path 项目，C 为 child 项目，L 为 leader 项目

显然，每次 FIND 操作对 leader 项目投资 1 美元，child 项目投资不超过 1 美元，block 项目投资不超过 $\log^* n$ 美元（每个块最多对应 1 美元），因此 m 次 FIND 操作以后对 leader、child 和 block 三个项目的投资额不超过 $2m+m\log^* n$ 元。惟一不好计算的是 path 项目。

回忆：如果一个结点现在不是根，则它以后永远也无法变成根，而只有根的 rank 才可能改变。因此对于任何一个给 path 项目投资的结点 x_i，它的 rank 始终不变。每经过一次 FIND，x_i 的父亲结点被设为了根，而根的 rank 是严格比 x_i 原来父亲的 rank 大的，因此 rank(*father*(x_i))每次严格增大。由于 rank(x_i)不会改变，因此最终 rank(*father*(x_i))会和 rank(x_i)位于不同的块内，从此以后 x_i 再也不用给 path 项目投资了。不管怎样，由于块 b 的 rank 最大值为 $2\uparrow\uparrow b$，因此 x_i 最多投资 $2\uparrow\uparrow b$ 次。而块 b 一共不超过 $n/(2\uparrow\uparrow b)$个元素，因此块 b 内所有元素对 path 项目的投资额不超过 n 美元。对于所有 $\log^* n$ 个块来说，对 path 项目的总

投资额为 $n\log^* n$。

四种投资额之和为 $2m + m\log^* n + n\log^* n = O(m\log^* n)$，这就是我们要证明的结论。

练　习　题

思考题：

***1.4.1**　如果允许在并查集中删除某个元素，应该怎样做呢？

1.4.2　例题 1 和例题 2 用宽度优先遍历可以做吗？为什么？使用并查集有什么好处？

1.4.3　在二进制计数器问题中，如果开始 counter 不是 0，那么用势能分析法时 Φ_0 不等于 0。试用定义计算出实际的时间耗费并说明此时总的均摊时间复杂度仍是 $O(n)$。

***1.4.4**　考虑另一种形式的并查集：每个元素保存一个指向该集合代表元素的指针（并查集的穿线森林（threaded forest）表示。该表示不需要用路径压缩），则 FIND 是 $O(1)$的。应该怎样实现合并操作，才能使 m 次 MAKESET 和 n 次 UNION 的最坏情况均摊时间为 $O(m+n\log n)$？如果 UNION 特别少，这个方法和标准的并查集实现哪个更好？为什么？

1.4.5　并查集的算法中如果不使用启发式合并和路径压缩，时间复杂度如何？这两个“顺便”的改进对性能影响之大，给你什么样的启示？

编程题：

1.4.6　**窗户**[①]

在笛卡儿坐标系中有一个多边形，多边形的边平行于坐标轴，每两条相邻的边是垂直正交的并且每一个顶点的坐标都是整数。还给出一个边也平行于坐标轴的矩形窗户。多边形的内部被涂成灰色，那么有几个分开的灰色部分将在窗户中被看到，如图 1-46 所示。

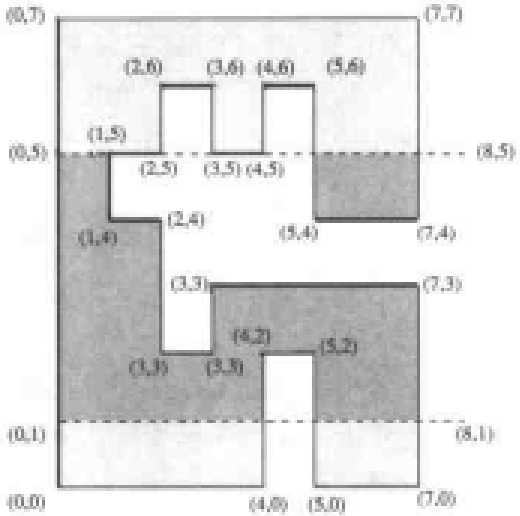


图 1-46　窗户

① 题目来源：Polish Olympiad in Informatics

有两个灰色的部分将被通过窗子看到。

***1.4.7　奇数偶数**[1]

你和你的朋友玩这样一个游戏：

你的朋友写下一个由 0 和 1 组成的字符串。你从中选择连续的子串（例如从第 3 到第 5 连续的 3 个数字），然后问他这个子串中 1 的个数是奇数还是偶数。你的朋友回答了，然后你继续问类似的问题。

你估计你的朋友的回答中有一些错误，你想证明他的错误。所以，你决定编写一个程序来解决这个问题。这个程序会根据你的问题以及他的回答来判断他是否错了。你程序的目标是找到第 1 个可能的错误的回答，也就是说存在一个数字串满足前面所有的问题，但对当前的问题不满足，而且没有一个数字串满足。

例如，序列长度为 10，信息条数为 5。5 条信息分别为 1 2 even，3 4 odd，5 6 even，1 6 even，7 10 odd，那么正确答案是 3，因为存在序列(0,0,1,0,1,1)满足前 3 条信息，但是不存在满足前 4 条的序列。

1.4.2　堆及其变种

在刚才的例题中，需要维护一个可以插入和取最小值的数据结构。刚才的离线算法固然简单高效，但是它是一个离线算法，适用范围比较窄。本节将介绍一种数据结构，用它可以很容易设计出刚才“蜗牛”问题的在线算法。插入和取最小值的时间复杂度均为 $O(n\log n)$。

堆是一棵完全二叉树，每个结点有一个权。它的特点是根的权最小，且根的两个子树也各是一个堆。前面讲过，可以用数组来模拟完全二叉树。堆的操作主要有两个：

堆的插入　由于需要维持完全二叉树的形态，需要先将结点 x 插到新的位置，然后把 x 上升调整（做元素交换）到合适的位置，即当 x 比它父亲小时，把 x 和它的父亲交换。

堆的删除　由于需要维持完全二叉树的形态，需要先用“最后一个结点”x 把要删除的结点覆盖掉，再把 x 下降调整到合适的位置，即当 x 比它的某一个儿子大时，把 x 和它的较小儿子交换。

堆操作的时间复杂度　容易证明，插入和删除的最坏情况时间复杂度为 $O(\log n)$，取最小值的时间复杂度为 $O(1)$（因为堆的根就是最小值）。另外，改变一个值并调整堆的时间复杂度也是 $O(\log n)$（请读者自己实现这一过程）。

堆排序算法　显然，每次取出最小值可以实现排序，而且它可以是部分排序。如果是部分排序的话，显然应该高效地建立堆。是一个一个插入元素吗？不是的，那样的时间复杂度将达到 $O(n\log n)$，还不如直接排序后构建一个堆。正确的做法是先将自底向上一层一层地调整每个结点到正确的位置，则交换操作执行次数不超过 $4n$（请读者证明）。

---

[1] 题目来源：CEOI 1999

【例题 1】积水[1]

有这样一块土地，它可以被划分成 $N\times M$ 个正方形小块，每块面积是一平方英寸，第 i 行第 j 列的小块可以表示成 $P(i,j)$。这块土地高低不平，每一小块地 $P(i,j)$都有自己的高度 $H(i,j)$（单位是英寸），如图 1-47 所示。

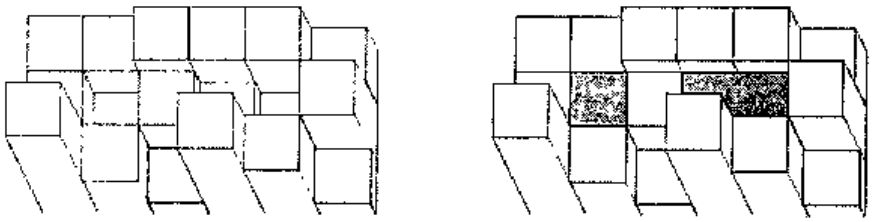


图 1-47　积水示意图

一场倾盆大雨后，由于这块地地势高低不同，许多低洼地方都积存了不少降水。假如你已经知道这块土地的详细信息，你能求出它最多能积存多少立方英寸的降水么？

【分析】

本题初看上去似乎无从下手。不妨先求出一些最容易求出来的格子，例如边界。边界上积水一定为 0！能否通过它们进一步求出其他格子的水量呢？经验告诉我们，应该从边界中最低的格子 x 入手！就像是一个有洞的水桶，桶的容量取决于最低的那个“洞”！假设这个格子的高度为 h，那么它相邻格子 y 的水位不会超过 h（否则将从 x 流出整个土地！）。如果 y 的高度小于 h，那么 y 的水位将是 h，我们可以继续对 y 周围的格子进行“灌水”，直到这块水全部被高度大于 h 的地势包围起来。换句话说，实际上是把高度大于 h 的地方和已经确定水位的地方看作是“障碍物”，然后从格子 x 开始做了一次 floodfill（在 1.3 节中介绍过），则填充好的格子水位全是 h，并且这些格子周围的土地全部应该标记为禁止积水（否则积水将转移到格子 x 并流出土地）。这样，我们每次取禁止积水的格子中最低的一块，做一次 floodfill 并添加一些新的格子到“禁止积水”集合中。如果用堆来实现“禁止积水”集合，那么由于每个格子最多只能加入和删除一次该集合，而且每个格子最多被一次 floodfill 覆盖到，因此总的时间复杂度为 $O(mn\log mn)$。

【例题 2】赛车[2]

有 n 辆赛车($1\leqslant n\leqslant 250\ 000$)，从各不相同的地方(位置 $0\leqslant x_i\leqslant 1\ 000\ 000$)，以各种的速度(速度 $0<v_i<100$)开始往右行驶，不断有超车现象发生，如图 1-48 所示。

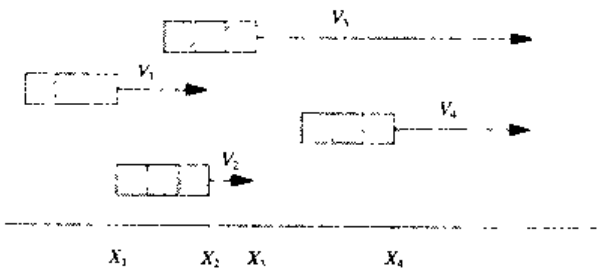


图 1-48　赛车比赛

[1] 题目来源：Polish Olympiad in Informatics
[2] 题目来源：CEOI 2003. The Race.

给出 n 辆赛车的描述（位置 x_i，速度 v_i），赛车已按照位置排序（$x_1<x_2<\cdots<x_n$）。输出超车的总数，以及按照时间排序的前 m（$m\leqslant 10\,000$）个超车事件（如果事件同时发生，先输出超车位置早的，输入数据保证没有超车事件会同时同地发生）。

【分析】

第一问实际上是逆序对的数目。因为车 i 将会超过车 $j(i<j)$当且仅当 $v_i>v_j$。这样的数对数目就是逆序数。

一种方法是利用归并排序的思路，把数组分成两半，则逆序数等于两半数组的逆序数，加上 i 位于左半，j 位于右半但是 $v_i>v_j$ 的数对数目。显然在归并时，当左半的某元素 i 移出序列时，右半序列还剩的数的个数就是形如(i,j)的逆序对数目。这样，第一问可以在 $O(n\log n)$ 的时间内解决。

注意到 $v\leqslant 100$，所以可以用另外一个方法解决。用 num[v]记录当前车之前速度为 v 的车共有多少辆，从左到右考虑每一辆车 i，形如 (j, i) 的逆序对数目应该是当前的 num[1]+num[2]+…+num[v_i-1]，然后更新 num，即让 num[v_i]加一，准备考虑下一辆车。这样，每辆车计算的时间复杂度为 $O(v)$，总时间复杂度为 $O(n\times v)$。后面将看到，如果用线段树来实现 num，则总的时间复杂度可以降到（$(n+v)\log v$）。

第二问实际上是要做事件模拟。事件的发生有先后顺序，所以应该用堆来维持可能发生的事件列表。对于任何时刻，下一超车事件（如果有的话）必定是某车追上了该时刻离它最近的前面车（注意：我们并不是每个车第一个追上的是现在离它最近的，而是从全局看的）。因此应该把每个车 i 连同它前面第一辆车的编号（记为 $e[i]$）保存在一个堆里，并计算出该事件发生的时刻（注意，是绝对时刻而不是相对上一事件的经过时间，这样可以避免累计误差。发生时刻应用分数表示，而不是实数，以避免实数比较产生误差），每次取出最小的时刻，输出该事件（注意，要考虑到同时发生的多个事件）。

容易忽略的是，如果车 i 超了车 j，那么满足 $e[x]=i$ 的车 x 必须更新为 $e[x]=j$；$e[j]$需要更新为 i。由于 $e[x]=i$ 的车只有一个，所以只要同时再记录一个 $f[i]=x$（即紧跟其后的车），那么更新问题就解决了。同时这里也看到了记录“距离最近的车”而不是“下一个可以超的车”的好处了。后者在更新的时候可能会需要更新多个车的信息，不仅麻烦，而且效率也不高。这样，一共执行了 m 次取最小值（用来输出）和堆调整（更新车的相邻信息）操作，时间复杂度 $O(m\log n)$，而建堆时间为 $O(n)$，故总的时间复杂度为 $O(n+m\log n)$。

二项堆和 fibonacci 堆虽然有比较高的理论价值，但是由于程序比较麻烦，而且实际效率并不令人满意，这里就不介绍了。

【例题 3】可怜的奶牛[①]

农夫 John 有 n（$n\leqslant 100\ 000$）头奶牛，可是由于它们产的奶太少，农夫对它们很不满意，决定每天把产奶最少的一头做成牛肉干吃掉。但还是有一点舍不得，John 打算如果不止有一头奶牛产奶最少，当天就大发慈悲，放过所有的牛。

由于 John 的奶牛产奶是周期性的，John 在一开始就能可以了解所有牛的最终命运，不过他的数学很差，所以请你帮帮忙，算算最后有多少头奶牛可以幸免于难。每头奶牛的产

① 题目来源：OIBH Reminiscene Programming Contest　命题人：刘汝佳

奶周期 T_i 可能不同，但不会超过 10。在每个周期中，奶牛每天产奶量不超过 200。

【分析】

这道题目的核心是每次取一个最小元素，但令人头疼的是，元素是会周期性变化地，所以不能把它们直接组织成一个堆。如果采用最笨的方法，每次先求出每头牛的产奶量，再求最小值，则每天的复杂度为 $O(n)$，总复杂度为 $O(T\times n)$，其中 T 是模拟的总天数。由于周期不超过 10，如果有的牛永远也不会被吃掉，那么我们需要多模拟 2 520 天（它是 1，2，3，…，10 的最小公倍数）才能确定，这样的复杂度太高了。

不妨换一种思路。周期同为 t 的奶牛在没有都被吃掉之前，每天的最小产奶量也是以 t 为周期的。因此如果把周期相同的奶牛合并起来，每天只需要比较 10 类奶牛中每类牛的最小产奶量就可以了，每天的复杂度为 $O(k)$，其中 k 为最长周期。这段话可能并不太好理解，这里有一个例子。假设周期为 6 的牛有 4 头，它们的其他参数与“合并”结果如表 1-8 所示。

表 1-8　周期为 6 的奶牛参数表

项　目	第 6n+1 天	第 6n+2 天	第 6n+3 天	第 6n+4 天	第 6n+5 天	第 6n+6 天
牛 1	2	5	3	5	7	4
牛 2	3	1	6	7	5	4
牛 3	5	3	3	5	3	9
牛 4	4	4	3	8	8	2
合并结果	2（牛 1）	1（牛 2）	3（多牛）	5（多牛）	3（牛 3）	2（牛 4）

这样，在和其他组进行比较时，如果是第 6n+1 天，就把牛 1 拿去比较，因为其他牛都比它大。如果是 6n+3 天就把产奶量 3 拿去比较，如果它是最小的，那么就有多头奶牛最小，因而都不会被吃掉。这样，每次只需要比较 k 组牛的“代表”就可以了，每天模拟的时间复杂度为 $O(k)$。下面考虑维护这个表的开销。

只要周期为 6 的牛都不被吃掉，**这个表一直是有效的**。但是在吃掉一头奶牛后，我们需要修改这个表，使它仍然记录着每天的最小产奶量。一个办法是重新计算，复杂度为 $O(h)$，其中 h 是该组的牛数，另一个方法是把**一个周期中每天**的最小产奶量组织成堆，每次删除操作的复杂度是 $O(k\log h)$。由于每头奶牛最多被吃掉一次，因此用在维护“最小产奶量结构”的总复杂度不超过 $O(nk\log n)$。每天复杂度为 $O(k)$，总复杂度为 $O(T\times k+nk\log n)$，大大低于前面所讲的复杂度。

【例题 4】最轻巧的语言[①]

字母表 A_k 由 k 个英文字母组成。字母表中每一个字母都有一个权，一个单词的权等于单词中每个字母的权和。一种定义在字母表 A_k 上的语言是一个由字母表中字母组成的单词的集合。语言的权是语言中所有单词的权和。如果语言中任何一个单词都不是其他单词的前缀，则说这种语言是 prefixless。

希望找到一种由 n 个单词组成的，prefixless 语言，并使它的权最小。

① 题目来源：Polish Olympiad in Informatics

假设 k=2，字母 a 的权 $w(a)$=2，字母 b 的权 $w(b)$=5。则单词 ab 的权 $w(ab)$=2+5=7，单词 aba 的权 $w(aba)$=2+5+2=9。语言 J={ab,aba,a}的权 $w(J)$=21。但语言 J 不是 prefixless，因为单词 ab 是单词 aba 的前缀。以上字母表中权最小的 3 个元素的 prefixless 语言是{b,aa,ab}，它的权是 16。

写一个程序，读入整数 n（≤10 000），k（≤26）和字母表 A_k 中 k 个字母的权（≤10 000），计算由 n 个单词组成 prefixless 语言的最小权值。

【分析】

考虑一棵树。设一个权值为 0 的虚结点作为根，然后得到一棵 k 叉树，每个结点代表一个字符串。我们语言中的单词是叶子，而仅最后一个字母不相同的长度为 p 的两个单词结点可以合并成它们的前 p-1 个字母组成的字符串。显然，串 x 为串 y 的前缀当且仅当 x 是 y 的祖先，所以只要取叶子组成该语言，则任何一个单词都不是其他单词的前缀。比如样例中的最优编码对应于图 1-49 中的一棵二叉编码树，三个叶子结点对应的编码分别为{b，aa，ab}。现在的问题是怎样寻找一种建树的方法。

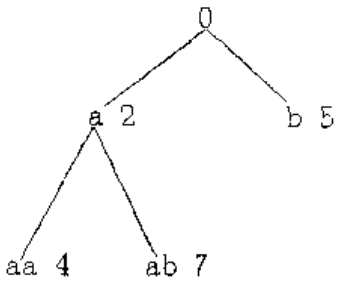


图 1-49　建树的方法

用贪心算法来解决这道题。开始时，令树根有 k 个儿子，分别是{a，b，c，…}，每一次，找出一个权值最小的叶子，将它作为当前要扩展的结点。比如取出单词 a，扩展出的 k 个新的叶结点就是 aa，ab，ac，…。这样每次扩展一个结点，知道不可能产生更优秀的编码方案位置。

具体实现时，可以不实际建树，而是采用模拟建造 k 叉树的方法。开始时，令单词集合中有{a，b，c，…}，每一次，从集合中取出一个权值最小的单词作为当前要扩展的单词，将它删去，加上由它扩展出的新单词。如果当前单词集合中最小的 n 个单词权和大于已知最优解时，则可以停止。

对于这样的操作，可以用堆来实现，这样可以在 $O(1)$的时间内找出权值最小的单词，而插入单词的时间复杂度是 $O(\log N)$。这样做时间上没有问题了，但空间却不能承受。可以知道，堆中的元素最多可能有 N^2 个，由于 N 最大可能达到 10 000，因此不可能实现。可以发现，如果一个单词的权值已经大于当前第 n 小的单词，那么它扩展出来的新单词也不可能进入前 n 个单词，因此这个单词是没有意义的，可以删除。这样，在任何时候，都只需要考虑当前的前 n 个单词，堆的大小只需 N 个，完全可以满足要求。然而这样做又产生了新的问题。由于只有一个堆，只能知道最小的元素，而当堆中元素多于 n 个时，不知道要哪几个是权值最大的。为了解决这个问题，还需要再使用一个堆，这样可以在 $O(1)$的时间内找出当前权值最大的单词。

现在产生了一个问题：一个元素在一个堆中删除后，还需要在另一个堆中删除，这样要为两个堆中的每一个元素建立一个指针，指向它在另一个堆中的位置。这样做既并不影响算法的时间复杂度，又保证了可以同时在两个堆中插入和删除元素。这样的技巧在很多题目中还有应用，即最大堆-最小堆。回顾我们的思路：贪心法→最小堆实现→删除最大元素来减少空间需求→增加最大堆→两个堆构成最大堆-最小堆。

专题七——两种特殊的堆：杨式图表和笛卡儿树

本专题介绍两种特殊的堆。杨式图表（Young Tableau）是一个矩阵，它满足条件：

- 如果格子(i, j)没有元素，则它右边和上边的相邻格子也一定没有元素。
- 如果格子(i, j)有元素 a[i,j]，则它右边和上边的相邻格子要么没有元素，要么比 a[i,j]大。

例如，含有 1,2,3 这三个元素的杨式图表为四个，如图 1-50 所示。

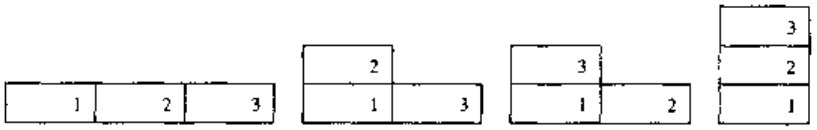


图 1-50　三个杨式图表的例子

首先，不加证明地给出一些结论（证明留给读者）：

定理一：1~n 所组成的 Young Tableau 的个数由下式给出：

$$a(1)=1,\ a(2)=2$$

$$a(n) = a(n-1) + (n-1)a(n-2)$$

可以把公式写成显式的，但是这个递推公式通常更有用。这个数很重要，它还等于自可逆的排列数目。

定理二（钩子公式）：对于给定形状，不同的 Young Tableau 的个数为 $n!$除以每个格子的钩子长度（即该格子右边的格子数和它上面的格子数之和）+1 的积，其中 n 为总格子数。例如图 1-51 左边的形状中，每个格子中间填的数代表该格子的钩子长度，则不同 Young Tableau 的数目为 $7!/((5+1)(2+1)(0+1)(3+1)(0+1)(1+1)(0+1)) = 35$，右边是两个具有左边形式的合法 Young Tableau。

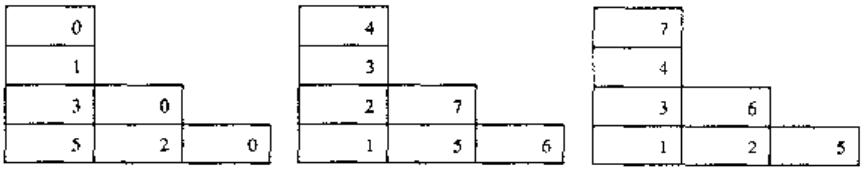


图 1-51　钩子公式使用示意图

如果把没有元素的格子看成有正无穷的格子，则两个条件可以变成一个。显然，它的每行从左到右递增，每列从上到下递增。如果沿着矩阵的左对角线看，杨式图表是一个二叉堆。虽然二叉堆有专门的插入和删除算法，但是由于杨式图表的特殊性，它还有一种特殊的插入算法，即 bumping 算法。

Young Tableau 的插入算法（bumping 算法）：从最底行开始，从左到右找第一个比 x 大的数。如果找不到，插到行尾；如果找到了符合条件的 y，把 x 和 y 交换，并从第二行开始用类似的方法插入 y。由于列数是递减的，所以算法必然终止，且每个元素最多比较和交换一次。

对于一个 $n\times m$ 的 Young Tableau，还可以设计一种高效的查找算法：

Young Tableau 查找算法：从底行最右列开始查找。如果当前元素比 x 大，向左走（因

此当前列一定不存在该元素），一旦碰到比 x 大的元素就向上（因此当前行的左边不可能有该元素了）。这样，任意时刻 x 的可能存在范围都是以当前位置为右下角的矩形区域。因为不会向下或者向右走，因此总的比较次数不超过 $m+n$。

笛卡儿树（Cartesian Tree）：也是一种特殊的堆，它根据一个长度为 n 的数组 A 建立。它的根是 A 的最小元素位置 i，而左子树和右子树分别为 A[1…i-1]和 A[i+1…n]的笛卡儿树。

给定一个数组，如何建立它的 Cartesian Tree 呢？最简单的方法就是直接按照定义，用顺序选择法来找最小值，最坏情况需要 $O(n^2)$，下面给出一个 $O(n)$的算法。

Cartesian 树的建立算法：从 A[1…1]开始建立，每次加入一个数，修改 Cartesian 树。不难发现，每次加入的数 A[i] 一定在旧树 C[i-1]的最右路径上，而且一定没有右儿子。只要沿着最右路径自底向上把各个结点 p 和 A[i]做比较，如果 p 小，则 A[i]插入，作为 p 的右儿子，否则把 p 作为 A[i]的左儿子。由于每个结点最多进入和退出最右路径各一次，因此均摊时间复杂度为 $O(n)$。

有了 Cartesian 树，现在可以解决 1.3.3 节专题五遗留的问题：一般 RMQ 问题。在专题五中，把 LCA 问题转化为了±1-RMQ 问题，得到了一个精巧而且易于实现的 O(n)-O(1)算法。现在，用 Cartesian 树把一般 RMQ 问题转化为 LCA 问题，从而得到一般 RMQ 问题的 O(n)-O(1)算法，转化方法非常简单。

定理：数组 A 的 Cartesian 树记为 C(A)，则 RMQ(A, i, j) = LCA(C(A), i, j)。

定理非常直观，请读者自己证明。

练　习　题

思考题：

1.4.8　本节介绍的“堆”可以方便地进行合并吗？为什么？

1.4.9　如果需要给“堆”加入查找操作，应该用什么样的数据结构与它联合使用？

****1.4.10**　试设计 bumping 算法的逆操作（删除），把一个元素删除后再用 bumping 算法插入，得到的 Young Tableau 不变。

****1.4.11**　m×n 的 Young Tableau 里面装满了数，试给出一个算法把所有数排序。你的算法时间复杂度应该低于直接快速排序的复杂度 $O(mn\log mn)$。

编程题：

1.4.12　方格[①]

有一个 N×N（1≤N≤2 003）的点阵，相邻点之间会有一条带整数权 w 的有向弧（1≤w≤500 000）。并且，从左上角的点（$v_{1,1}$）到任何一点的所有路径的长度（途经的所有弧的权之和）都相等，如图 1-52 所示。

① 题目来源：CEOI 2003. Square.

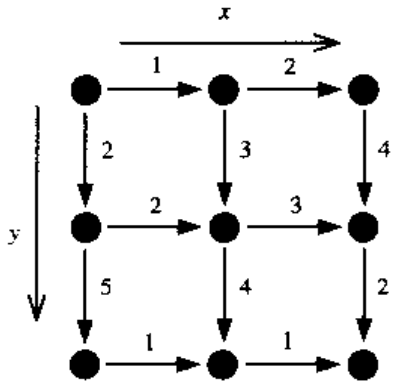


图 1-52　方格举例

给定一个整数 L（$1 \leqslant L \leqslant 2\,000\,000\,000$），你要找到一个点，使从 $v_{1,1}$ 到它到道路长度恰为 L。本题是一道交互式题目，你可以询问某条边的长度。询问次数不得超过 6 666 次。

**1.4.13　仓库[1]

芬兰的一个高科技公司有一间很大的长方形仓库。仓库有一位工人及一位经理。仓库的四边（根据其环绕的次序）分别称为左、上、右及下边。仓库内由很多大小相同的正方形摆放位置组成行和列。横行由上而下编号为 1，2，…，而竖列则由左至右编号 1，2，…。

仓库内有些货柜，用来存放一些贵重的高科技元件。每个货柜都有自己惟一的编号。每个货柜都占用一个正方形摆放位置。假设仓库非常之大，使得到达仓库的货柜永远不会填满任何一行或任何一列。货柜不会从仓库移走，不过有时会有新的货柜运到这个仓库。仓库的入口是在仓库的左上角。工人将货柜通过一个特别的方式存放在仓库的左上角附近，使得在需要时可以由货柜的编号找到货柜所在的位置。具体来说，他所用的方法如下：

假设下一个要摆放的货柜 k（我们把它称为货柜 k），工人就会由第一行开始，由左至右尝试在该行内找出第一个编号大于 k 的货柜。如果找不到，他就会取出货柜 k 摆放在该行最右边的摆放位置内。相反，如果他找到一个编号大于 k 的货柜 l，他就会取出货柜 l 而用货柜 k 来放在 l 的位置上，之后，他就会将货柜 l 以相同的方法尝试摆放在下一行上。若工人到达一个没有任何货柜的行时，他就会把这个货柜摆放在该行的最左端。

假设货柜到达仓库的次序是 3，4，9，2，5，1。则它们被摆放在仓库内的情况如下：

1 4 5

2 9

3

某天，经理来到工人面前，两人有以下的一段对话。

经理：货柜 5 是不是在货柜 4 之前到达的？

工人：不，这是不可能的。

经理：这么说来，你是可以由货柜摆放位置知道它们到达的先后次序？

工人：一般来说是不可以的。例如对于目前货柜摆放的位置而言，这些货柜到达的次

[1] 题目来源：IOI2002. Depot

序可能是3，2，1，4，9，5或是3，2，1，9，4，5，或者是其他14种不同可能之一。

经理为了不想显得这工人比他更精明，所以他就走开了。现在，你就要帮这位经理编写一个程序用以从货柜摆放的位置来确定货柜到达仓库的可能次序。货柜的总数≤13。

1.4.3 字典的两种实现方式：哈希表、二叉搜索树

本节我们讨论字典。字典的基本操作是插入和查找。对于有的字典，还需要支持删除操作。对于不需要删除的字典，哈希表是一种理想的实现方式，它效果好，实现简单，但是分析复杂，需要涉及到概率分析等知识；二叉搜索树的可扩展性强，支持删除操作，但是实现起来比哈希表麻烦，对于随机性强的数据效果也不如哈希表好。

哈希表　哈希表（也叫散列表 Hash Table）的思路是根据关键码值直接访问。也就是说，这是一个把关键码值映射到表中的一个位置来访问记录的过程。这个映射函数叫做哈希函数，用 h 来表示，存放记录的数组叫做哈希表。表里的每个位置叫做一个槽（slot），槽的数目用 M 表示。哈希表适合集合检索，不适合多个元素关键码相同的情况，也不适合范围检索，即查找关键字处于一个范围之内的所有元素。

哈希表的插入和查找算法　哈希表的插入和查找几乎是一样的，即：

（1）计算函数值 h(K)。

（2）从槽 h(K)开始，使用（如果需要）冲突解决策略定位包含关键码 K 的记录。

（3）如果需要插入，在槽里把数据插入即可。

如果冲突可以忽略不计，那么两种操作的复杂度都是 $O(1)$。

哈希函数的选取　怎样选取好的 HASH 函数才使得计算不过于复杂，冲突又比较小呢？对于这个问题，我们只有一些经验上的解决方式。

对于数值来说，可以：

方法一：直接取余数（一般选取 M 作为除数，而 M 最好是个质数，这样冲突少些）。

方法二：平方取中法，即计算关键值平方，再取中间 r 位形成一个大小为 2^r 的表。

方法一容易产生分布不均匀的情况，而方法二要好得多，因为几乎所有位都对结果产生了影响。但是它的计算量大，一般也较少使用。

对于字符串，常用的方法有：

方法一：折叠法，即把所有字符的 ASCII 码加起来。

方法二：采用 ELFhash 函数，即（它用于 UNIX 的“可执行链接格式，ELF”中，因此这里把它写成 C 函数）：

```
int ELFhash(char *key){
  unsigned long h=0;
  while(*key){
     h=(h << 4) + *key++;
     unsigned long g=h & 0Xf0000000L;
     if (g) h^= g >> 24;
```

```
        h &= ~g;
    }
    return h % M;
}
```

这是一个很有用的 HASH 函数，它对长短字符串都很有效，推荐大家把它作为字符串的 HASH 函数。

冲突解决方法　在插入数据 a 时，如果槽里已经有了元素 b（即元素 a 的位置等于被元素 b 占据了），该怎么办呢？有两种处理方法，分别称为**开散列**和**闭散列**。

开散列法　开散列也叫拉链法，通俗地说，就是“既然元素 a 和 b 都该放里面，只好挤一挤了”。即在每个槽里存放所有该放在里面的元素。怎么把很多元素放在一个槽里呢？只要在槽里放一个链表表头就可以了，该链表中包含所有该放在槽里的元素(在实际应用中，往往不把链表做成传统的使用动态内存的结构，而是自己维护一个大数组，给链表元素分配数组下标，这样又方便又节省时间和空间)。不过有一个问题：链表中的元素按照什么顺序排列呢？考虑两个问题，一是查找成功时的效率，显然可以按照访问的频率排序；二是查找失败时的效率，如果按照关键值排序，那么即使查找失败也不需要遍历整个链表。我们再一次看到了数据结构中相互矛盾的两个问题，读者在应用时候要根据实际情况协调这两个方面。

闭散列法　闭散列也叫开放地址法。通俗地说，就是“既然 a 可以霸占 b 的位置，b 也可以用同样的手段，霸占另一个元素 c 的位置。”也就是说，不严格按照关键值的 HASH 函数值来选择槽，而是在位置被占用时按照某种方法另外选一个空槽。当然，这样做是危险的，因为有可能产生元素的聚集而使得查找时间大量增加。有关这方面的内容，如线性探查、二次探查及相应的聚集问题以及效率分析，有兴趣的读者可以参考这方面的书籍。它的分析比较复杂，而且效果不好，所以本书所讲到的散列一律用开散列解决冲突。

线性时间排序　利用 HASH 表，可以实现前面所说的“线性时间排序”，但前提是数据的取值范围比较窄，这样不仅空间节省，而且能保证时间复杂度比较低。例如已知 10 000 个人的年龄（范围为 1~200），要给他们按照年龄排序（对于年龄相同的人，谁排在前面无所谓），那么最好的方法就是 HASH 排序。如果有冲突，可以建立链表或者其他 HASH 表的冲突解决方式，但是如果冲突太多，用这种排序方法就不合算了。

在哈希表中删除元素　另外，虽然往往不需要，但有时候还是需要在 HASH 表中删除元素。一般只是增加一个删除标记，在删除标记很多以后再一次性永久删除。

【例题 1】马尔可夫链[①]

现在需要一些随机的可以读的英语文本，但是计算机运用语言的能力跟一个婴儿差不了多少，由它随机生成的文本是毫无意义的。当然，也可以给它一本字典，让它从中随机挑选单词，但这也无济于事，因为计算机不懂语法。实际上，可以用一种相当简单的方法，让计算机写出一段比较像样的文章来。这种技术被称为“马尔可夫链算法”，它利用一个

① 题目来源：CTSC 2001. 命题人：石润婷

样本来产生一段结构与样本相似的文章。

可以把一段英语文本看成是一系列相互重叠的短语组合，该算法把每个短语分成两部分：由一个以上的词构成的前趋和只有一个词的后继。这个算法用一句话就能说明问题：根据特定的前趋在样本中随机地选取后继。实践证明，采用 3 个词的短语效果最好。算法具体过程如下：

1. 随机地在样本中选取连续的两个词 w1、w2 作为文本的开头。
2. 输出 w1、w2。
3. 在样本中随机选取一个 w1、w2 的后继 w3，如果不存在 w3，则结束算法。
4. 输出 w3。
5. 令 w1=w2，w2=w3。
6. 转 3。

这个算法也可以根据需要随时停止。为了帮助理解算法，这里有一个例子：

Show your flowcharts and conceal your tables and I will be mystified. Show your tables and your flowcharts will be obvious.

下面是一些输入的前趋和它们的后继，如表 1-9 所示。

表 1-9　输入前趋和后继举例

前　趋	后　继	
Show your	Flowcharts	tables
your flowcharts	and	will
Flowcharts and	Conceal	
Flowcharts will	Be	
your tables	and	and
Will be	Mystified	obvious
be mystified	Show	

如果选择了 Show your 作为文本的开头，那么它的后继有两个 flowcharts 和 tables，从中随机选取一个。假设选取了 flowcharts 作为后继，那么前趋就变为 your flowcharts，这样不断重复，直到没有任何后继或是产生了足够多的词。

你的任务是完成这个算法的最核心部分，即对于一系列给定的前趋，分别输出它们所有可能的后继。样本文件最多含有 10 000 个单词，大小不超过 200k，前趋个数不超过 2 000 000 个，单词不超过 255 个字母。文本完全是随机生成。

【分析】

本题求对于每一个有序的单词对(w1,w2)，求在给定文章中(w1，w2)的所有后继。如果该单词对在文章中不存在或找不到任何后继，则输出一个空行。如果某单词对的一个后继在文章中出现了不止一次，则输出的次数等于出现的次数。（这样在随机选取的时候可以使该词被选中的频率较高）由于题目规模较大，线性的查找肯定不能在规定时间内出解，于是采用高效的算法和数据结构成了解题的关键。

方法一：二分查找

对于本题，由于样本文件最多有 10 000 个单词，因此表中的单词对数目最多为 10 000，二分查找每次最多找 14 次。二分查找的过程花费了大部分的时间，如果能尽快找到单词的位置，就可以极大地提高效率。

方法二：采用哈希表

对于本题，理想的情况是不经过比较或尽量少地比较而找到给定单词对的位置，这就必须令单词对和文件中的位置尽量一一对应。由于题目给定的文章完全是随机生成的（这一点很重要），所以一般总能使冲突数目平均分布。因此问题的关键是如何选定哈希函数？考虑两种极端地选取方法。

（1）令 Hash 函数为 $h(w1,w2)=(w1,w2)$，则对于给定文章中的$(w1,w2)$，$h(w1,w2)$对应的位置个数约为 1。这样虽然达到理想情况，然而映射过程需要进行最多 510 次比较，是一种低效的 HASH 函数，而且根本就没有这么多的内存。

（2）HASH 函数为 $h(w1, w2)$为 $w1[1]$，这样和 $w2$ 是完全无关的，因此产生大量堆积。

需要寻找一种使映射时比较的时间和冲突数折衷的 $h(w1, w2)$，使映射方法简单而又不需太多得比较次数。将（2）扩展得到另一种方法即：$h(w1, w2)=(w1[1], w2[1])$

一共有 52^2 个位置和 10 000 个表项，因此每个表项对应约 $10\ 000/52^2=3.7$ 项。如果令 $h(w1,w2)=(w1[1]+w1[2], w2[1]+w2[2])$，那么一共有 $104^2=10\ 816$ 个位置，几乎能保证一一对应。

建议读者自己试验这几种方法，完全随机地生成样本文章和前趋，比较一下各种方法的效果如何。

【例题 2】促销[1]

促销活动遵循以下原则："参与本活动的顾客，应将自己的个人信息写在所付账单背面，并将账单投入指定的箱子。在每天的销售结束后，箱子中消费金额最大和最小的两张账单将被选出。消费最多的顾客将得到一笔奖金，奖金的数目等于两张账单金额之差。为了避免因为一次消费而得到多笔奖金，依照以上原则选出的两张账单将不被放回到箱子中，但箱子里剩下的账单可以继续参加第二天的活动。"

你的任务是根据每天投入箱子的所有账单，计算出整个促销活动中超市要付出的奖金总额。本题中约定：

1. 整个促销活动持续了 n 天，$n\leqslant 5\ 000$。
2. 第 i 天放入的钞票有 $a[i]$张，$a[i]\leqslant 10^5$，且 $s=\sum a[i]\leqslant 10^6$。
3. 第 i 天放入的钞票面值分别是 $b_i[1], b_i[2], \cdots, b_i[a[i]]$，$b_i[j]\leqslant 10^6$。

【分析】

本题实际上是在操作一个线性表 D，它可以支持的操作和执行的次数如表 1-10 所示（请大家仔细阅读题目，并进行验证）。

[1] 题目来源：Polish Olympiad in Informatics

表 1-10　操作的含义和次数

操　作	含　义	次　数
Init	将表 D 清空	1
Add	在表 D 中加入一个数字 x	$s=\Sigma a[i]\leqslant 10^6$
Remove	从表 D 中删除一个数字 x	$2n\leqslant 10^4$
Getmin	求表 D 中的最小数	$n\leqslant 5\,000$
Getmax	求表 D 中的最大数	$n\leqslant 5\,000$

本题的算法并不复杂，关键就在于这些操作的效率。下面，着重讨论几种不同的数据结构和它们的效率。

方法一：采用普通的线性表

采用普通的线性表，则线性表的长度不超过 s，各操作的时间复杂度如表 1-11 所示。

表 1-11　线性表的操作复杂度

操　作	Init	Add	Remove	Getmin	Getmax
时间复杂度	$O(s)$	$O(1)$	$O(1)$	$O(s)$	$O(s)$

由此可以看出，采用普通的线性表，时间复杂度为 $O(ns)$，太大了。

方法二：采用二叉堆

由于表 D 既要求最大值，又要最小值。因此，需要一个小根堆和一个大根堆。同样，堆的长度不超过 s，各操作的时间复杂度如表 1-12 所示。

表 1-12　堆的操作复杂度

操　作	Init	Add	Remove	Getmin	Getmax
时间复杂度	$O(s)$	$O(\log s)$	$O(\log s)$	$O(1)$	$O(1)$

总时间复杂度为 $O((n+s)\log s)$，基本上可以承受，空间复杂度也是 $O(s)$，不太理想。

我们注意到，因为整个活动不超过 5 000 天，如果有一天 $a[i]=10^5$，那么把那天的钞票按面值排序，只有面值最小的 5 000 张和最大的 5 000 张可能有用，而其他的显然都可以抛弃。一般地，只需要保存面值最小的 n 张和最大的 n 张钞票就可以了。这样，算法的空间复杂度就降到了 $O(n)$，而时间复杂度也就进一步降到了 $O((n+s)\log n)$。

有没有更快更简单的实现方法呢？答案是肯定的，我们注意到下面两点：

（1）第一种数据结构耗时主要在操作 getmax 和 getmin 上。

（2）$1\leqslant b_i[j]\leqslant 10^6$，最多是 6 位数。

第一点说明关键在于优化 getmax 和 getmin；第二点提示我们采用哈希表，得到：

方法三：采用哈希表

定义 $d[x]$，表示数 x 在表 D 中出现的次数。因为 x 在[1，10^6]区间内，令 $m=10^6$，则标志数组各项操作的时间复杂度如表 1-13 所示。

表 1-13　哈希表的操作复杂度

操　作	Init	Add	Remove	Getmin	Getmax
时间复杂度	$O(m)$	$O(1)$	$O(1)$	$O(m)$	$O(m)$
实　现	d 清零	Inc($d[x]$)	Dec($d[x]$)	扫遍整个表	扫描整个表

很遗憾，这样做算法的时间复杂度还是高达 $O(s+nm)$，但如果把数组 d 分段，比如每段长为 L，再令一个数组 c，$c[i]$表示在区间$[(i-1)L+1, iL]$中的数在表 D 中出现的次数，则在找最小、最大值的时候，可以分两步走：先在 c 数组中找到最小（或最大）的 i，使得 $c[i]>0$，然后再在区间$[(i-1)L+1, iL]$中，从 d 数组中找到具体的那个数。如果我们令 $L=m^{1/2}$，则找最小、最大值的复杂度可以降到 $O(m^{1/2})$。于是得到：

方法四：采用分段哈希表

定义 $d[x]$，表示数 x 在表 D 中出现的次数，其中 x 属于区间$[1, m]$。令 $L=m^{1/2}$，再定义 $c[i]$表示在区间$[(i-1)L+1, iL]$中的数在表 D 中出现的次数。这个数据结构各项操作的时间复杂度如图 1-14 所示。

表 1-14　分段统计哈希表的操作复杂度

操　作	Init	Add	Remove	Getmin	Getmax
时间复杂度	$O(m)$	$O(1)$	$O(1)$	$O(m^{1/2})$	$O(m^{1/2})$
实　现	D 清零	Inc($d[x]$) Inc($c[x \text{ div } L]$)	Dec($d[x]$) Dec($c[x \text{ div } L]$)	先查 c 后查 d	先查 c 后查 d

这个算法的时间复杂度是 $O(s+nm^{1/2})$，对于题目给出的数据范围，该算法的时间消耗比二叉堆还要低，程序也很容易实现，惟一的不足就是空间复杂度是 $O(m)$。这是个典型的空间换时间的算法。

看到这里，读者可能会说："如果再继续分段，复杂度不是更低了吗？"请注意，复杂度由两部分决定的，一个是 add，一个是 getmin/getmax。继续分段固然可以减少 getmin/getmax 的时间，但是 add 的时间增加了，成为算法的瓶颈。当然，并不是说一次分段效果最好，还可以尝试让每个段长度递减，就可以避免两次检索时间都很长（因此很快查到段序号的说明段序号小，那么第二次查找时间会长一点；如果查段序号的时间太长，到了最后一定处于一个比较小的段，第二次查找时间就比较短了）。我们只是想再次提醒读者，数据结构的选择存在很多的矛盾需要我们去协调。

二叉搜索树的定义　在 1.3 节中，介绍了二叉树。在本节中，介绍一种特殊的二叉树：二叉搜索树（又称二分检索树、排序二叉树、二叉检索树，Binary Search Tree、BST）。它要么是一棵空树，要么它的左子树的所有结点比根结点小，右子树的所有结点比根结点大，而它的左子树和右子树分别是一棵二叉搜索树。

查找和插入算法　二叉搜索树是递归结构，查找和插入可以用递归的方法，根据关键字和根的大小关系递归在子树中递归（类似于二分检索，只是二分检索每次是直接确定范

围，而二叉搜索树每次是找到一棵子树的根）。

删除算法　删除 x 需要分情况讨论。

情况（1）：无子女，则直接删除。

情况（2）：仅有一个子女 y，则如果 x 是它父亲 f 的左儿子，则令 f 的左儿子为 y；如果 x 是它父亲 f 的右儿子，则令 f 的右儿子为 y。

情况（3）：有两个子女 y 和 z。我们的想法是从 y 或者 z 的后代中找出一个结点覆盖 x 的位置。不难分析出，这个结点可以是大于 x 的最小结点 a（想一想，为什么），或者小于 x 的最大结点 b。找 a 的方法是从 x 的右儿子出发一直往左走，直到没有左儿子为止（想一想，这时的结点为什么是大于 x 的最小结点？）；找 b 的方法类似。这样，用 a（或 b）代替 x 即可，BST 不需要进行任何其他改变。

操作的时间复杂度　虽然平均情况插入，删除和查找的时间复杂度都是 $O(\log n)$，但是存在退化情形，因为二叉树可能变得不平衡。例如依次往一棵空树中插入 1，2，3，…，N，得到的将是一条长长的链。删除元素 N 的时间复杂度显然为 $O(N)$。

平衡二叉树　有没有保持平衡的二叉搜索树呢？有！例如红黑树，AVL 树等。AVL 树并没有使用新的数据结构而只是用一组规则来让二叉树平衡起来，即平衡化旋转规则。有兴趣的读者可以阅读相关资料。

Treap　在实践中，有一种很容易实现而且效果不比 AVL 等严格的平衡二叉树差的概率数据结构叫做 Treap，是 BST 和 Heap 的结合(Tree + Heap = Treap)。需要注意的是，这里的“Heap”只是指根的权值最小且子树也为“heap”，它并不一定是完全二叉树。每次插入/删除结点的时候，给结点随机分配一个优先级，让 Treap 关于优先级形成一个堆的同时，关于关键码形成一个 BST。为了满足这一点，需要利用平衡化旋转，有兴趣的读者可以参考相关书籍。

【例题 3】采矿[①]

金矿的老师傅年底要退休了。经理为了奖赏他的尽职尽责的工作，决定在一块包含 n（$n \leqslant 15\ 000$）个采金点的长方形土地中划出一块长度为 S，宽度为 W 的区域奖励给他（$1 \leqslant s, w \leqslant 10\ 000$）。老师傅可以自己选择这块地的位置，显然其中包含的采金点越多越好。你的任务就是计算最多能得到多少个采金点。如果一个采金点的位置在长方形的边上，它也应当被计算在内。

【分析】

从左到右用两根竖线来进行扫描整个金矿，使得两根线之间的区域在 X 坐标上相差不超过 S，然后再统计这一个带状区域中的每一个宽度为 W 的矩形，如图 1-53 所示。

图 1-53 是两条扫描线 L1 和 L2。L2 在 L1 前面动，通过调整，使得 L1 到 L2 的距离不大于 S。这时中间带状区域的点成为进一步研究的对象，每个点进出要处理的带状区域各一次。

[①] 题目来源：Polish Olympiad in Informatics

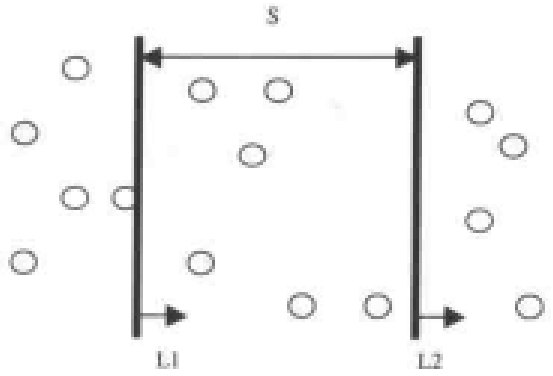


图 1-53　横向扫描

对于带状区域中的所有点，由于他们的横坐标差不会大于 S，所以可以忽略所有的横坐标，仅考虑他们的纵坐标。例如在一个带状区域内有 5 个点的纵坐标分别是{5，3，9，1，9}，w=2，很自然地考虑将这几个坐标排序成{1，3，5，9，9}，然后可以通过类似横坐标的扫描方法来求得宽度为 w 的矩形。但是在本题中，这种方法不是特别好。进一步考虑对纵坐标的特殊处理：对于每一种坐标 y，建立成两个点事件(y,+1),(y+w+1, −1)。例如将前面的点标成(1,+1),(4, −1),(3,+1),(6, −1),(5,+1),(8, −1),(9,+1),(12, −1), (9,+1),(12, −1),一共是 10 个点事件，再将他们按照 y 的坐标排序，得(1,+1),(3,+1),(4, −1), (5,+1),(6, −1), (8, −1), (9,+1), (9,+1),(12, −1), (12, −1)，把后面的标号反映在一个 y 坐标的映射上，然后从低到高求和，如表 1-15 所示。

表 1-15　事件到 y 坐标的映射

Y 坐标	1	3	4	5	6	8	9	12
数	+1	+1	−1	+1	−1	−1	+2	−2
连加和	1	2	1	2	1	0	2	0

注意坐标下的求和，这些和中最大的一个对应于该带状区域中包含最多点数的矩形。每个位置上的和记录的是此位置之前所有标号的和。

通过这步巧妙的转换，可以用前面的二叉排序树来实现 y 坐标的点事件处理。同时前面已经说过，每一个点进出带状区域仅各一次，因此要利用树的统计实现：在插入或者删除一个点事件之后，能够维持坐标下∑的值；能够在很短时间内得到∑中最大的一个值。得到大致算法如下：

将所有的点事件映射到 y 坐标中，最多有 n=15 000 个点，所以可能有 30 000 个不同的坐标，将这些值建立一棵可用以统计的二叉排序树，即 BUILD。

在树上的每个结点，要设立两个值。一个是 SUM，记录以该点为根的子树上所有点上值的和，开始时 SUM=0。另一个是 MAXSUM，记录以该点为根的子树上最大的∑，注意这里的定义是以该结点为根的，也就是在子树上的值。可以通过下面这个函数来完成一个点事件的插入，并且维护 SUM 和 MAXSUM 的特性。其中 k 是一个标号，其值为+1，或者−1。

仔细分析这个过程，它有两个部分：

（1）向下找到 y 所在的结点。

（2）向上对路径上的每个点的相关量进行修改。

在步骤（2）中，SUM 的计算不需要解释。MAXSUM 的计算实际上是一个递推的过程，因为是要取得连续和的最大值，所以有 3 种情况：

1．最大值在左树上；

2．最大值正好包含根结点；

3．最大值在右树上。

在完成了 INSERT 操作后，MAXSUM[1]就是当前的一个最优解。INSERT 算法的执行复杂度为树高，即 $\log n$。

有了 INSERT 过程，很容易完成整个算法，只要先将所有的点按照横坐标先排序，然后利用两条扫描线 L1 和 L2 进行扫描。根据前面的分析，n 个点进出扫描的带状区域各一次，每一次对应两个点事件，因此总共是 $4n$ 次 INSERT 操作。整个算法的时间复杂为 $O(n\log n)$。

练　习　题

思考题：

***1.4.14**　设计一种数据结构，可以在短时间内执行以下三种操作：插入单个元素（整数），删除单个元素，返回值最接近的一对元素（如果多对元素，任意返回一对）。

1.4.15　hash 函数的函数值分布应该尽量满足什么条件？hash 表的大小是否受到 hash 函数本身的限制？

编程题：

1.4.16　天上的星星[①]

天文学家们喜欢观察星空图。它们把每颗星星看成一个点，并把每颗星星左下方（即横坐标和纵坐标都不比它大）的星星颗数作为它的“等级”值。

如图 1-54，星星 5 的等级为 3，因为星星 1，2，4 都在它的左下方；星星 2 和 4 的等级都是 1。请帮助天文学家求出每个星星的等级。

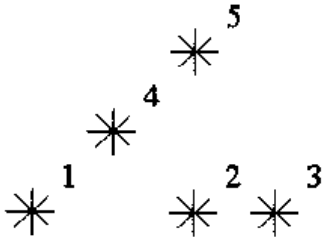


图 1-54　星空图

① 题目来源：Ural State University Problem Archive

1.4.17　围棋[①]

围棋是一项起源于古中国的智力游戏。据说，它是由于在桌面上模拟战争而产生的游戏。围棋的棋盘上有 19 条横线和 19 条纵线，然后两个玩家轮流在这 19 条横线和纵线的 361 个交点上放置黑色和白色的棋子。最后，玩家中控制较多交点数的一个将赢得游戏。

如下是围棋的几项规则：

当游戏开始的时候，棋盘上是没有任何棋子的。然后黑方在棋盘上选定一个交点放上他的黑色棋子，然后是白方，并以此类推。棋子是不能放在已经放置了棋子的交点上的。如果某个玩家试图放在这样的地方，那么他的行为将被视为弃权。当两个玩家都放完一次棋子，就称其为一个回合。当然，在一个回合中任何一个玩家都可以选择弃权。

我们称从横向或纵向连在一起的棋子为一块。如果一块棋子中的任一个棋子都不与空白的交点相连，那么就称这块棋子为死棋。当第一次形成死棋的时候，就应该把这块棋子从棋盘上拿走。例如图 1-55 中的所有黑色棋子都是死棋。

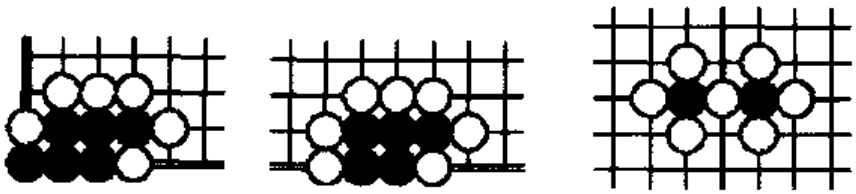


图 1-55　死棋举例

如果在一个玩家在某位置放上棋子之后，并没有把对方的任一个棋子变成死棋而把自己的棋子变成了死棋，或者放上这个棋子并提掉所有的死棋之后，棋盘布局和另一个玩家上次放上棋子之前的状态是一样的，那么这个位置就被称作禁手，并且这个玩家这次行动将被作为弃权看待。

如果说在一个有效的布子之后，棋盘的状态重现，并且在这些布子的过程中没有人选择弃权，那么这次比赛的结果就是平局，并且玩家需要重新开始一个新的比赛。例如图 1-56 就是一个三劫循环。

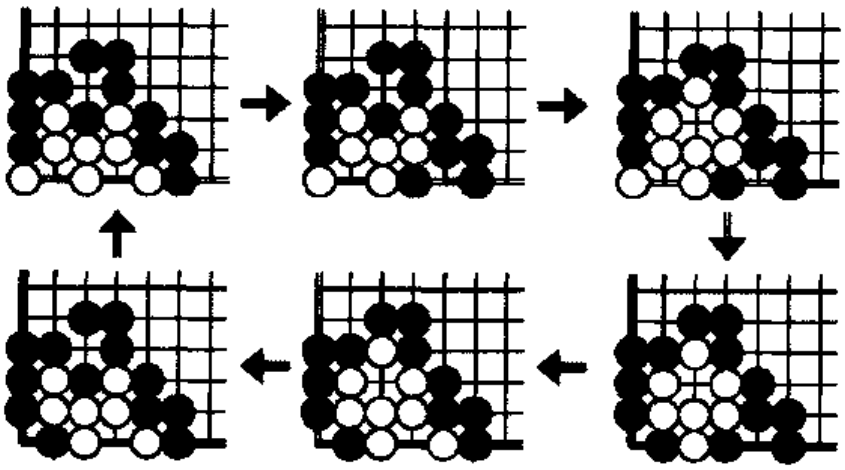


图 1-56　三劫循环

[①] 题目来源：ACM/ICPC Regional Contest Shanghai 2000, Weiqi

现在的问题就是要你写一个程序，读入两个玩家的布子方案，并模拟他们的比赛。

1.4.18　10-20-30 游戏[①]

有一种纸牌游戏叫做 10-20-30。游戏使用除大王和小王之外的 52 张牌，J、Q、K 的面值是 10，A 的面值是 1，其他牌的面值等于它的点数。

我们把 52 张牌叠放在一起放在手里，然后从最上面开始依次拿出 7 张牌从左到右摆成一条直线放在桌子上，每一张牌代表一个牌堆。每次取出手中最上面的一张牌，从左至右依次放在各个牌堆的最下面。当往最右边的牌堆放了一张牌以后，重新往最左边的牌堆上放牌。

如果当某张牌放在某个牌堆上后，牌堆的最上面两张和最下面一张牌的和等于 10、20 或者 30，这三张牌将会从牌堆中拿走，然后按顺序放回手中并压在最下面。如果没有出现这种情况，我们会检查最上面一张和最下面两张牌的和时候为 10、20 或者 30，解决方法类似。如果仍然没有出现这种情况，最后检查最下面的三张牌的和，并用类似的方法处理。例如：如果某一牌堆中的牌从上到下依次是“5，9，7，3”，那么放上“6”以后的布局如图 1-57 所示。

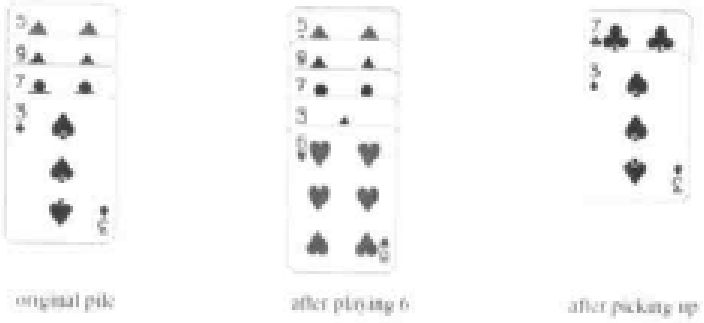


图 1-57　一个排堆的初始、中间和最终布局

如果放的不是 6，而是 Q，对应的情况如图 1-58 所示。

图 1-58　一个排堆的初始、中间情况和最终布局

如果某次操作后某牌堆中没有剩下一张牌，那么把该牌堆便永远地清除掉，并把它右边的所有牌堆顺次往左移。如果所有牌堆都清楚了，游戏胜利结束；如果手里没有牌了，

[①] 题目来源：ACM/ICPC World Finals

游戏以失败告终；有时候游戏永远无法结束，这时我们说游戏出现循环。给出 52 张牌最开始在手中的顺序，请模拟这个游戏并计算出游戏结果。

1.4.19　方块堆砌的小镇[①]

给定一个平面投影大小为 W×L 的城堡的前视图和右视图如图 1-59 所示，要求编一个程序，判断能否搭成与给定的前视图和右视图相吻合的城堡，若能则求出搭建这个城堡所需的最少和最多的积木的数目，其中城堡的最大高度不超过 5 000，W 和 L 均不超过 100 000。

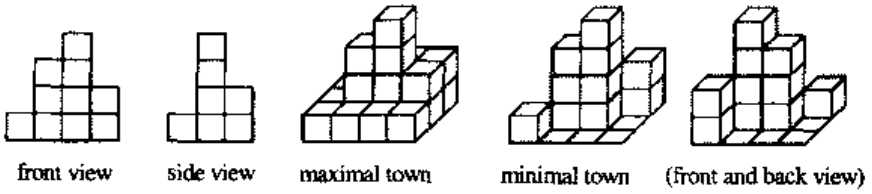


图 1-59　城堡的各种视图

1.4.4　两个特殊树结构：线段树和Trie

树结构的基本思想是分割。普通二叉搜索树是按照对象来进行划分，因此效果往往和数据结构内对象有关；本节的两个数据结构是根据关键码的可能范围来分的，这种技术叫做关键空间分解。

线段树的处理对象是线段（一般意义上的区间也可以看成是线段），它把线段组织成利于检索和统计的形式，它的本质是线段的二叉搜索树。但是线段可以分解和合并，线段树又有一些一般二叉搜索树所没有的特殊的操作。另外，线段树操作的是整个区间，它的渐进时间效率不依赖于数据结构中的对象。

线段树的定义　线段树是一棵二叉搜索树，最终将一个区间[1, *n*]划分为一些[*i*, *i*+1]的单元区间，每个单元区间对应线段树中的一个叶结点。每个结点用变量 count 来记录覆盖该结点的线段条数。图 1-60 举了一个线段树的例子。

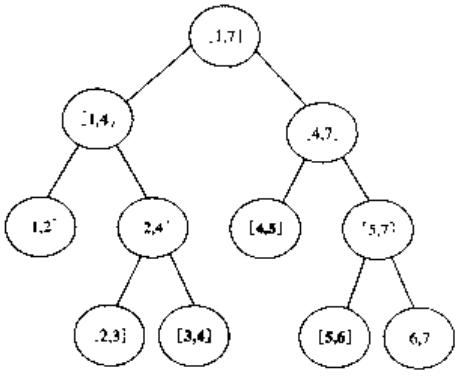


图 1-60　区间[1，7]所对应的线段树，区间上有一条线段[3，6]

[①] 题目来源：CEOI 1999

线段树处理对象　线段树的处理对象是一段较狭窄的区间，区间上的格点对应有限个固定的变量，好像是线性的数组用完全二叉排序树的形式表达。处理问题的时候，抽象出区间上的格点，也即是明确每个格点对应变量的含义。

线段树的实现和空间耗费　如果结点 i 的处理对象是区间$[a,b]$，那么区间$[a,(a+b)/2]$和$[(a+b)/2,b]$分别是 i 左儿子和右儿子的处理对象。当求解区间$[a,b]$内某个参数的统计值时，可以直接调用 i 上的统计信息，而无需将区间$[a,a]$→$[b,b]$的信息一个个累加。这是线段树在时间效率上主要的优点。和堆及二叉树相同，线段树也是动态维护的数据结构。如果结点 i 的处理对象是区间$[A_i,B_i]$，那么 C_i 作为计数器，记录覆盖到区间$[B_i,E_i]$的线段个数，L_v，R_v 分别表示了 I 左儿子和右儿子的编号。考虑空间复杂度。最坏情况是满二叉树，结点总数为 $L+[L/2]+[L/4]+\cdots\approx 2L\ (L=b-a)$。

线段树的插入和删除　显然，算法是基于二分的。和一般的 BST 不一样的是，只要考虑的总区间不变，线段树的结点本身是不变的（这一点不难理解），改变的只是计数器 C_i 和其他信息。虽然也是二分，但是可能需要同时递归到两个分支中去(例如往[1,7]中插入[3,5]，则同时需要往[1,4]和[4,7]中插入[3,5])。好在任意时刻最多只有两个未被完全覆盖的区间需要递归处理（请读者证明），因此插入和删除的复杂度为 $O(h)$，其中 h 为树的高度。显然，$[a,b]$ 上的线段树满足 $h=O(\log(b-a))$，这也提示我们可以用非递归的迭代形式来写插入删除过程，时间效率更高。

***线段树的扩充**　除了完全覆盖区间的线段条数 C_i 之外，往往还需要记录两个量来帮助我们获得更多的信息：

- 测度 m：结点所表示区间中线段覆盖过的长度。
- 独立线段数 line：指的是区间中互不相交的线段条数。
- 权和 sum：区间所有元线段的权和。

请读者仔细写出三者的递推式（提示：为了记录 line，还需要记录区间的两个端点是否被覆盖）。

【例题 1】火星地图[1]

2051 年，科学家们探索出了火星上 N（$N\leqslant 10\,000$）个不同的矩形（坐标为不超过 10^9 的正整数）区域并绘制了这些局部的地图，如图 1-61 所示。巴罗的海太空研究所希望绘制出火星的完整地图。

科学家们首先需要知道这些矩形共占了多大的面积，你能帮助他们写一个程序计算出结果吗？

【分析】

本题最直接的方法是用容斥原理来做，时间复杂度很高。另一种方法是统计所有 1×1 的小方格子中有多少被覆盖掉，如图 1-62 所示。

[1] 题目来源：Baltic Olympiad in Informatics 2001，经典问题

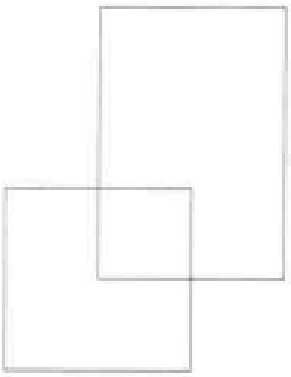


图 1-61　火星地图举例

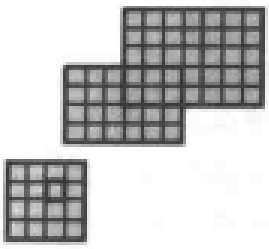


图 1-62　单位方格统计法

显然，这种方法和坐标范围是很有关系的，而本题的坐标范围太大，怎么办呢？虽然坐标范围不知道，但是矩形只有 n 个，涉及到的坐标只有 $4n$ 个，一种常见的处理方法称为“离散化”，即用出现过的所有横坐标值和所有纵坐标值把平面划分成网格形状，每个格子是一些大小不全相等的元矩形，这些元矩形一个有 $O(n^2)$个，如图 1-63 所示。这样，每个元矩形要么全被覆盖，要么全不被覆盖，因此本题转化为统计所有被覆盖的元矩形的面积和。每添加一个矩形，最坏情况下要把所有元矩形都标记为“覆盖”，因此算法的总时间复杂度为 $O(n^3)$。

由于矩形表示的区域是一维区间的笛卡儿积，因此可以考虑需把线段树扩充一下，让它能处理矩形。具体做法是把一个平面切割成四份，分别考虑插入删除和统计。我们在二维线段树上连续插入 N 个矩形区域，再进行统计即可。由于已经经过了离散化，因此平面区域范围是(1,1)~(2n,2n)，算法的总时间复杂度为 $O(n\log^2 n)$。这道题看起来很圆满地解决了吗？不是！考虑空间复杂度，你会发现它是 $O(n^2)$，无法承受。

回想本题的解决过程，只需要处理 n 个矩形，因此不需要直接应用二维线段树，而可以采用和“采矿”一样的“自左向右扫描（left-right scan）”的技术来做，如图 1-64 所示。

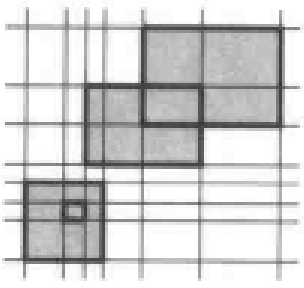


图 1-63　离散化统计法

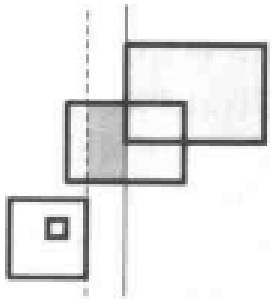


图 1-64　扫描法示意图

如上图，在相邻两条竖线之间的区域是矩形。先把所有竖线按照从左到右排序，然后从左到右进行扫描，左右竖线（在程序实现中分别用+1 和−1 来表示）分别标志着矩形进入和退出阴影区域，也就会涉及到线段的插入和删除操作。有了刚才的经验，知道可以用离散化后的线段树来做，时间复杂度降为了 $O(\log n)$，由于一共处理最多 $2n$ 条竖线，因此总的时间复杂度为 $O(n\log n)$，空间复杂度为离散化后的一维线段树空间复杂度，即 $O(n)$。

Trie 结构　当关键字是串的时候，往往用 Trie。我们的动机是：利用串的公共前缀来节约内存，加快检索速度。例如，需要保存“computer”和“command”，由于它们的前三个字母是相同的，所以希望它们共享前三个字母，而只有剩下部分才进行分开存储。显然，这样的结果是一个树结构，把它叫做字母表的 Trie 结构。一般地，只要是由关键字集 S 中若干元素串在一起的结构都叫做 Trie。

Trie 的特点　它的特点如下：

- 根结点不包含字母，除根结点外每一个结点都仅包含一个大写英文字母；
- 从根结点到某一结点，路径上经过的字母依次连起来所构成的字母序列，称为该结点对应的单词。单词列表中的每个词，都是该单词查找树某个结点所对应的单词；
- 每个结点的所有儿子包含的字母都不相同。

Trie 的插入删除和查找　Trie 的插入、删除和查找都非常简单，用一个一重循环就可以了，即第 i 次循环找到前 i 个字母所对应的子树，细节这里就不再赘述了。只是在存储的时候最好采用左儿子右兄弟方法，而不是在每个结点开一个有 26 个元素的指针数组。另一个方法是不用指针而是一个大的数组来实现 Trie，这时每个结点有一个 26 个元素的下标数组，表示相应的儿子在大数组中的下标。这样速度和空间消费都会有所改善。

***Trie 的变体**　需要注意的是，Trie 和 BST 一样，也有结构不理想的情况，即有大量单个儿子的结点。针对这个问题，D.Morrison 发明了一种 Trie 的变体称为 PATRICIA，它把数据记录在叶结点上，而使用内部结点存储关键码模式的位置，本书不详细讨论。

【例题 2】最长回文子串[①]

给出一个串 X，求出它的最长子串 S，使得 S 和 S^R 相等（S^R 为 S 反转后得到的）。

【分析】

在分析题目之前，先介绍一种特殊的 Trie，称为**后缀树（suffix tree）**。它的定义如下：一个字符串 S 的后缀树为字符串 S\$的所有后缀（一共有 n+1 个）所组成的 Trie，其中\$不是字母表中的字母。当字母表大小固定时，后缀树可以用它的发明者 Weiner 创立的 $O(n)$ 算法构造出来，但是该方法非常复杂（而且它的无数简化版还是很复杂），这里不介绍这些算法。后缀树的意义在于：由于任何子串都是某后缀的前缀，因此串 A 为串 B 的子串当且仅当 A 能在 B 的后缀树中“找到”（可能在到达叶子之前终止）。加上\$的意义在于强制所有的后缀都对应叶结点。

需要说明的是后缀树的大小（结点数）。由于后缀的长度分别为 1,2,3,…,n+1，所以结点数可以高达 $O(n^2)$。聪明的读者大概已经想到解决方法了：把只有一个儿子的结点合并！由于在没有单儿子结点时，内结点的数目不超过叶子的数目，而叶子只有 n+1 个（一个后缀对应一个叶子），因此树的总结点数不超过 2n+1。由于进行了结点合并，一个结点可能包含不止一个字母，把后缀树的定义稍微修改一下，即每个结点不保存字符串，而在连接父亲和儿子的边上标记两个整数 i 和 j，表示此边代表子串 $S[i\cdots j]$，这样处理以后，从根到每个叶子的路径和 S 的所有后缀仍然是一一对应的。把压缩后的后缀树定义为压缩后缀树，

① 题目来源：经典问题

压缩前的称为原后缀树（未加说明的情况下，后缀树是指压缩后的）。

对于后缀树中任何一个点 i，定义 path(i)为从根到 i 路径上所有边上的串连接起来的，而定义一个串 s 的位置（locus）为满足 path(i)=s 的结点 i。一个串 s 的扩展为以 s 为前缀的任意串，而扩展位置（extended locus）为 s 的“位置存在”的最小扩展的位置。显然，在未进行结点合并的后缀树中，任何串 s 的位置和扩展位置是一样的（或者都不存在），但是由于有结点合并，因此有些串虽然在压缩后缀树中没有位置，但是它在原后缀树中是有位置的，因此才引入“扩展位置”来避免不正确地处理它。显然，如果 s 的位置存在，则 s 的扩展位置和位置一样。

回到本题，做一个新串 $w=X?X^R$，其中?在 X 中不出现。那么，对于 X 的每一个位置 j，求出以 j 为中心的最长回文串，即求出最大的 k，使得 $x[j\cdots j+k-1]=(x[j-k\cdots j-1])^R$（代表偶数长度的回文串），或者 $x[j+1\cdots j+k]=(x[j-k\cdots j-1])^R$（代表奇数长度的回文串）。只需求 w 的两个后缀 $w[j\cdots 2n+1]$和 $w[2n+3-j\cdots 2n+1]$的最长公共前缀（偶数长度的情形）和 $w[j+1\cdots 2n+1]$和 $w[2n+3-j\cdots 2n+1]$的最长公共前缀（想一想，为什么?）。图 1-65 表示了偶数长度时的情形。

字母	a	b	c	d	…	d	c	b	a	…	
位置			j					$2n+3-j$		…	$2n+1$

图 1-65 偶数长度时的字符串示意图

显然，由于 X 被翻转以后放在了后面，因此检查 cd 是否等于$(ab)^R$转变为了检查 cd 是否等于后面从位置 $2n+3-j$ 开始的 ba。这步检查可以通过从求 j 开始的后缀和从 $2n+3-j$ 开始的后缀的最长公共前缀。由于在后缀树中，一个串的前缀是它的祖先，因此最长公共前缀就是两个后缀的最近公共祖先（LCA）。前面已经介绍了求 LCA 的 $O(n)$-$O(1)$算法，因此本问题也可以在 $O(n)$时间内完美解决。

WEB 有一个比较简单的 $O(n\log n)$Monte-Carlo 算法也可以构造后缀树。它利用了 Rabin-Karp 算法中用到的 Hash 函数，用分治的方法先把奇数起始位置的后缀构造一棵树（用 Hash 函数递归处理，再修正），然后由它得到偶数起始位置的后缀树，最后把两树合并。算法的正确与否取决于 Hash 函数。我们可以把算法运行多次，也可以直接用一个简单判定算法检查结果是否正确。关于此算法，以及二维后缀树、后缀数组（suffix array）、后缀二叉树（suffix binary tree）、后缀仙人掌（suffix cactus）等内容在本书主页上都有简单介绍。

练 习 题

思考题：

1.4.20 把线段树扩充成 k 维的，则它的时空复杂度为多少？

1.4.21　如果要进行字符串检索，BST 和 Trie 哪个更合适？整数呢？

编程题：

1.4.22　最长弱重复子串[1]

如果一个串 P 在某个文本 S 里出现了至少两次（这两次可以重叠，也可以有间隔，但是不能完全重合），那么说 P 是 S 的弱重复子串。用 $O(|S|)$时间求出 S 的最长弱重复子串。

****1.4.23　暗室里的油画**[2]

在暗室里有一幅油画。你手里拿着一个幅 $w \times h$ 的小图，它是油画的一部分，你希望知道它出现在油画中的什么位置？虽然你记住了这一个小图的每一个细节，但是暗室里的光线太暗，你什么也看不见。你所能做的只有比较油画的某一位置(x_1,y_1)和小图的某一位置(x_2,y_2)的颜色是否相同（假定油画和小块都是有一个个有颜色的方格组成）。你应该怎样做，才能尽快地知道小图出现在油画的什么位置呢？小图在油画的位置(x,y)出现意味着对于任意的(i,j) $(0 \leqslant i<w, 0 \leqslant j<h)$，小图的$(i,j)$位置和油画的$(x+i, y+j)$颜色一样。

提示：由于很难对模板进行一次性预处理，我们需要用一种特殊的顺序来进行匹配，并在合适的时候一点一点地把模板处理成容易使用的形式。

*****1.4.24　太空物质吸收器**[3]

你正开着飞船穿梭于 N 个星球之间。飞船装备着 M 个吸收光束发生器（SBG），每个 SBG 都可以收集一种不同的物质，一共有 M 种。你的同伴已经考察出了每个星球上有哪些物质，而你需要开飞船去收集这些星球上的所有物质。不幸的是，你的 SBG 并不是很先进。所有 SBG 共用一个激光源，当激光源打开时，所有开着的 SBG 开始吸收物质。你不能尝试着去吸收一个星球上不存在的物质，否则你的飞船会爆炸！而且，所有 SBG 都只能开关一次，因此一旦打开了一个 SBG，你只能访问含有该物质的星球，直到你关闭该 SBG。

你必须事先安排好一个飞行计划，让飞船安全地完成任务。也就是说，让包含任何一个物质的所有星球在你的飞行计划中连续。例如，有三种物质，五颗行星。第一种物质在行星 1,2,4 上有；第二种物质在行星 3,4 上有；第三种物质在行星 1,3,4,5 上有，那么一种合法的访问顺序是：5 3 4 1 2。

WEB　本题的解决需要一种特殊的数据结构：PQ 树，它可以很好地处理排列集合(permutation set)，相关信息可以在本书主页上找到。

*****1.4.25　最长重复子串**[4]

给出小写字母组成的字符串 S，找出最长的子串 x，使得 x 和自身的连接 $w=xx$ 在 S 中出现。本题有 $O(n^3)$, $O(n^2)$, $O(n\log n)$和 $O(n)$算法，你能设计出哪些？

WEB　本题的 $O(n\log n)$算法是利用的分治的思想解决，可以不用后缀树。本题有一个利用后缀树的十分巧妙的线性算法解决，有兴趣的读者可以在本书主页上找到

---

[1] 题目来源：经典问题
[2] 题目来源：IOI2002 Backup Task
[3] 题目来源：Internet Problem Solving Contest, 2002. 经典问题
[4] 题目来源：经典问题

1.5　动　态　规　划

动态规划是解决多阶段决策最优化问题的一种思想方法，我们可以从很多方面来体会它的思想。和一般书籍不一样，这里淡化“阶段”的概念，而只强调状态和状态转移，不从理论开始讲而直接通过应用动态规划的两种动机来讲，然后介绍若干常用的模型。本节无意把读者的思维限定在给出的几个模型中，但是从教学实践中看，这种方法对于打开读者思路仍然是利大于弊。

本节的所有内容都很重要，尤其是例题，决心掌握动态规划的读者应当花时间全部掌握。本节的难点是模型的理解，包括一些难度偏大的题目和动态规划理论中的一些问题。本节的例题很多，希望读者认真学习。

学习本节之后，应该对于每道题目，都能在很短的时间内把思路完整地整理一遍。对于大部分题目，应当能很快地写出程序。

1.5.1　动态规划的两种动机

本节介绍动态规划的两种动机。先举例子，再引出两种动机，最后进行总结。

动态规划的第一种动机是用利用递归的重叠子问题，进行记忆化求解，即先用递归法解决问题（就像1.2节中的一样），再利用重叠子问题转化成动态规划。先来看一道例题。

【例题1】括号序列[1]

定义如下规则序列（字符串）：

1．空序列是规则序列；

2．如果 S 是规则序列，那么（S）和[S]也是规则序列；

3．如果 A 和 B 都是规则序列，那么 AB 也是规则序列。

例如，下面的字符串都是规则序列：

()，　[]，　(())，　([])，　()[]，　()[()]

这几个则不是规则序列：

(，　[，　]，　)(，　([()

现在，给出一些由‘(’，‘)’，‘[’，‘]’构成的序列，请添加尽量少的括号，得到一个规则序列。

【分析】

可以用递归法来解决此问题。为了叙述方便，设序列 $S_iS_{i+1}\cdots S_j$ 最少需要添加 $d[i,j]$ 个括号，根据不同情况，我们可以用不同的方式转化成子问题。

- S 形如(S')或者[S']：

[1] 题目来源：ACM/ICPC Regional Contest, NEERC 2001

只需要把 S' 变成规则的，则 S 就是规则的了。

- S 形如(S'：

先把 S' 变成规则的，再在最后加一个“）”，则 S 就是规则的了。

- S 形如[S' 或者 S']或者 S' }：和上一种情况类似。
- 只要序列长度大于 1，都可以把 S 分成两部分 $S_i \cdots S_k$, $S_{k+1} \cdots S_j$，分别变成规则序列，则连在一起的 S 也是规则序列。

最优化原理和最优子结构　说到这里，不自然地用了这个问题的一个性质，在把原问题转换成规模更小的子问题时，原问题最优当且仅当子问题最优，这就是最优化原理。它不是显而易见的，也不是总成立的，因为在刚才的题目中，如果需要让添加括号数目的个位数字尽量小，则这个方法就不适用了。请读者仔细思考：为什么在本题中最优化原理成立？在今后的讨论中，如果这个原理成立，则说问题有最优子结构。

这样，得到一个递归过程：

```
function Bracket(i,j:integer);
begin
  if i>j then return 0
  else if i=j then return 1
  else begin
    Answer := ∞;
    if s[i]s[j]='()' or s[i]s[j]='[]' then
       Answer := min(Answer, Bracket(i+1,j-1));
    if s[i]='(' or s[i]='['then
       Answer := min(Answer, Bracket(i+1,j)+1);
    if s[j]=')' or s[j]=']'then
       Answer := min(Answer, Bracket(i,j-1)+1);
    For k:=i to j-1 do
       Answer := min(Answer, Bracket(i,k)+Bracket(k+1,j));
  End;
end;
```

这个递归算法是对的，但是速度却很慢，因为它的时间复杂度是指数级的（想一想，为什么？）。有没有办法把速度变快呢？有的。如果记 Bracket(i,j)的值为 $d[i,j]$，由于 $1 \leqslant i \leqslant j \leqslant n$，所以一共只有 $O(n^2)$个 d 值是需要计算的，在递归的时候做了大量的重复工作。

这个算法有两种改进法。

方法一：每次在调用 Bracket 函数之前检查是否已经计算过这个值了，如果是，则直接从表中读出。即

```
function Bracket(i,j:integer);
begin
```

```
  if Calculated[i,j] then return d[i,j];
  // 此处为原来的 Bracket 函数代码
  d[i,j]:=Answer;
  Calculated[i,j]:=true;
end;
```

方法二：按照一定的顺序计算所有的 d 值。由于计算 $d[i,j]$需要知道 $d[i+1,j]$,$d[i,j-1]$和 $d[i+1,j-1]$，所以按照 $j-i$ 递增的顺序计算 $d[i,j]$，即

```
for i:=1 to n do d[i,i-1]:=0;
for i:=1 to n do d[i,i]:=1;
for p:=1 to n-1 do
  for i:=1 to n-p do
  begin
    j := i+p;
    d[i,j]:=∞;
    if s[i]s[j]='()' or s[i]s[j]='[]' then
       d[i,j] := min(d[i,j], d[i+1,j-1]);
    if s[i]='(' or s[i]='[' then
       d[i,j] := min(d[i,j], d[i+1,j]+1;
    if s[j]=')' or s[j]=']' then
       d[i,j] := min(d[i,j], d[i,j-1]+1;
    For k:=i to j-1 do
       d[i,j] := min(d[i,j], d[i,k]+d[k+1,j]);
End;
```

如果说方法一的算法复杂度还不是很明显的话，方法二的复杂度已经很明显了，它是 $O(n^3)$。请读者仔细阅读前面介绍的思路和两种改进方法，这两种改进的方法就是本节要介绍的动态规划。动态规划的动机是递归时产生了大量的重叠子问题，这两种方法分别为记忆化搜索和自底向上的递推。

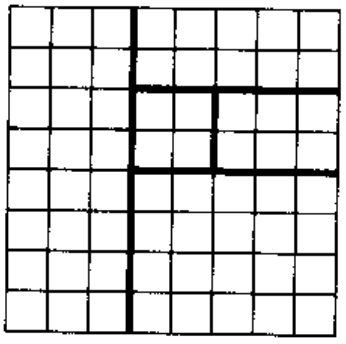
状态和状态转移方程　以后的题目中，不会再像这样把程序写出来，而只把 d 值的递推式和边界条件写出来。只要有了递推式和边界条件，就很容易把程序写出来，不论是记忆化搜索还是递推。把每个 d 值称为一个状态，d 值的递推式称为状态转移方程。很容易写出一个三重循环，第一层按一定的顺序计算每个状态，第二层在计算每个状态时考虑不同的递推路径（称为决策数目），最里层进行每个单独状态转移。算法复杂度 = 状态数 × 决策数目 × 转移费用。

状态是一个极其重要的概念。如果对动态规划很熟悉且具有相当的算法基础，那么在成功地设计出状态之后，动态规划算法就完成了一大半。因此强烈建议读者反复思考本书的每道题目，直到几乎每道题目都能很快设计出状态。设计状态的时候需要考察问题的无后效性，然后思考状态转移的时候需要考察问题的最优子结构。这两点

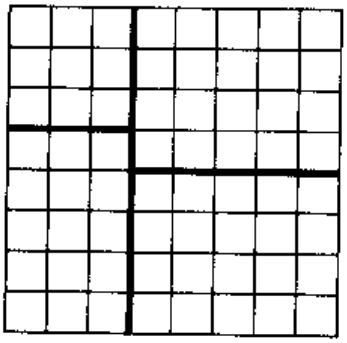
将在后面着重说明.

【例题 2】棋盘分割[1]

将一个 8×8 的棋盘进行如图 1-66 所示的分割：将原棋盘割下一块矩形棋盘并使剩下部分也是矩形，再将剩下的部分继续如此分割，这样割了(*n*-1)次后，连同最后剩下的矩形棋盘共有 *n*（*n*<15）块矩形棋盘（每次切割都只能沿着棋盘格子的边进行）。



(a) 允许的分割方案



(b) 不允许的分割方案

图 1-66　棋盘

原棋盘上每一格有一个分值，一块矩形棋盘的总分为其所含各格分值之和。现在需要把棋盘按上述规则分割成 *n* 块矩形棋盘，并使各矩形棋盘总分的均方差最小。

均方差 $\sigma=\sqrt{\frac{\sum_{i=1}^{n}(x_i-\bar{x})^2}{n}}$，其中平均值 $\bar{x}=\frac{\sum_{i=1}^{n}x_i}{n}$，$x_i$ 为第 i 块矩形棋盘的总分。

请编程对给出的棋盘及 *n*，求出 σ 的最小值。

【分析】

均方差的公式比较复杂，先将其变形为

$$\sigma^2=\frac{1}{n}(n(\bar{x})^2+\sum_{i=1}^{n}x_i^2-2\bar{x}\sum_{i=1}^{n}x_i)=\frac{1}{n}\sum_{i=1}^{n}x_i^2-(\bar{x})^2$$

由于平均值是一定的（它等于所有方格里的数的和除以 *n*），所以只需要让每个矩形的总分的平方和尽量小。考虑左上角坐标为(x_1,y_1)，右下角坐标为(x_2,y_2)的棋盘，设它把切割 *k* 次以后得到的 *k*+1 块矩形的总分平方和最小值为 $d[k,x_1,y_1,x_2,y_2]$，则它可以沿着横线切，也可以沿着竖线切，然后选一块继续切（这里用到了递归！）。故状态转移方程为：

$d[k,x_1,y_1,x_2,y_2]=\min\{$

$\min\{d[k-1,x_1,y_1,a,y_2]+s[a+1,y_1,x_2,y_2],\ d[k-1,a+1,y_1,x_2,y_2]+s[x_1,y_1,a,y_2]\}\ (x_1\leqslant a<x_2)$,

$\min\{d[k-1,x_1,b,x_2,y_2]+s[x_1,b+1,x_2,y_2],\ d[k-1,x_1,b+1,x_2,y_2]+d[x_1,b,x_2,y_2]\}\ (y_1\leqslant b<y_2)$

$\}$

设 *m* 为棋盘边长，则状态数目为 m^4n，决策数目为 $O(m)$。预处理先用 $O(m^2)$时间算出左上角为(1,1)的所有矩阵元素和，这样状态转移时间就是 $O(1)$，故总的时间复杂度为 $O(m^5n)$。由于 *m*=8，*n*≤15，这个方法还是够快的。

[1] 题目来源：NOI99

【例题3】决斗[1]

当 Louis XIII 和他的大臣 Richelieu 红衣主教进入 Full Barrel 旅馆时，有 n 个吃完了饭的步兵正在饮酒。由于喝了很多酒，步兵们吵起了架，因此爆发了一场决斗。但是按怎样的规则决斗呢？他们决定所有人呆在一个圈里，且按一定的顺序拉人。被拉出的步兵就与紧靠其右的人决斗。失败者将“退出”游戏，更确切地说是其尸体被仆人抬走，然后死者右边的步兵就与赢者直接相邻了。

多年后，史学家通过阅读赢者的回忆录，认为最终的结果在相当程度上取决于决斗的顺序。可能出现 A 比 B 打得好，B 比 C 打得好，C 比 A 打得好。那么，当然在这三人的决斗中，第一场决斗影响着最后的结果。假如 A、B 先打则 C 赢，B、C 先打则 A 赢。因而史学家决定论证哪些步兵有可能生还。

说得更具体一些，编号从 1~n 的 n 个人按逆时针方向排成一圈，他们要决斗 n-1 场。其中第 i 个人与第 i+1 个人决斗（如果 i=n 则与第一个人决斗），死者退出圈子，紧靠死者右边的人成为与赢者直接相邻的人。任意两人之间决斗的胜负都将在一矩阵中给出（如果 A[i,j]=1 则步兵 i 与步兵 j 决斗时，i 总是赢，如果 A[i,j]=0 则步兵 i 与步兵 j 决斗时，i 总是输），请求出所有可能赢得整场决斗的人的序号。

【分析】

假设需要判断 x 是否能赢得整场战斗，把环看成链，x 点拆成两个，那编号为 x 的人能从所有人中胜出的充分必要条件是他能与自己“相遇”。这样，在连续几个人的链中，只须考虑头尾两个人能否胜利会师，中间的则不予考虑。设 meet[i,j]记录 i 和 j 能否相遇，能则为 true，否则为 false，则问题转化为了是否能找到一个 k，使得 i 和 k，k 和 j 均能相遇（这里实际上是递归求解！），而 i 或者 j 能打败 k。即，状态转移方程为

$$\text{meet}[i,j]=\begin{cases}\text{true} & \text{存在}i<k<j\text{使得meet[i,k]且meet[k,j]且 (e[i,k]或者 e[j,k])}\\ \text{false}\end{cases}$$

时间复杂度为 $O(n^3)$，空间复杂度为 $O(n^2)$。

把问题看作是多阶段决策过程，是动态规划的第二个产生动机。

【例题4】“舞蹈家”怀特先生[2]

怀特先生是一个大胖子。他很喜欢玩跳舞机（Dance Dance Revolution, DDR），甚至希望有一天人家会叫他“舞蹈家怀特先生”。可惜现在他的动作根本不能称做是在跳舞，尽管每次他都十分投入的表演。这也难怪，有他这样的体型，玩跳舞机是相当费劲的。因此，他希望写一个程序来安排舞步，让他跳起来轻松一些，至少不要每次都汗流浃背。

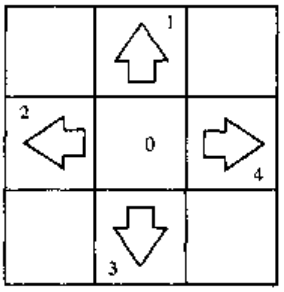


图 1-67　跳舞机踏板

DDR 的主要内容是用脚来踩踏板。踏板有 4 个方向的箭头，用 1，2，3，4 来代表，如图 1-67 所示。每首歌曲有一个箭头序列，游戏者必须按照这个序列依次用某一只脚踩相应的踏板。在任何时候，两只脚都不

[1] 题目来源：Polish Olympiad in Informatics

[2] 题目来源：ACM/ICPC Regional Contest, Shanghai 2000, Dance Dance Revolution

能在同一个踏板上，但可以同时待在中心位置 0。

每一个时刻，它必须移动而且只能移动他的一只脚去踩相应的箭头，而另一只脚不许移动。这样，它跳 DDR 的方式可以用一串数字 $l_1l_2\cdots l_n$ 来表示，其中 l_i=0 表示他用左脚踩第 i 个箭头，l_i=1 表示他用右脚踩第 i 个箭头。

跳完一首曲子之后，怀特先生会计算他所消耗的体力。从中心移动到任何一个箭头耗费 2 单位体力，从任何一个箭头移动到相邻箭头耗费 3 单位体力，移动到相对的箭头（1 和 3 相对，2 和 4 相对）耗费 4 单位体力，而留在原地再踩一下只需要 1 单位。怀特先生应该怎样移动他的双脚（即，对于每个箭头，选一只脚去踩它），才能用最少的体力完成一首给定的舞曲呢？

例如，对于箭头序列 Left, Left, Up, Right，它应该分别用左、左、右、右脚去踩，总的体力耗费为 2 + 1 + 2 + 3 = 8 单位。

【分析】

怀特先生需要作出一系列决策：对于每个舞步 i，选择它移动的脚 foot[i]，并把它移动到目标位置 A_i(这是已知的)。一个决策序列对应一个可行解，而他需要从中选出体力耗费最少的一种。

无后效性　对于每个舞步 i，怀特先生应该根据什么来进行决策呢？根据他以前作出的决策吗？显然不是。根据他已经耗费的体力吗？也不是。稍微分析一下就可以看出：只需要根据他现在双脚的位置。只有这个决定了他今后的决策。我们把这样的性质叫做**无后效性**。也就是说，决策只取决于当前状态的特征因素，而和到达此状态的方式无关。无后效性是应用动态规划的重要条件。如果不满足无后效性，那么状态表示是不合理的，因为同一个状态可能对应很多本质不同的实例。例如在本题中，如果仅选取当前舞步序号 i 作为状态值，那么显然它不满足无后效性，因为可以进行哪些决策取决于当前双脚的位置，而这两个位置并不在状态表示中

根据这个性质，又可以用递归的思路解题：枚举第 i 次所作出的决策，计算出达到的状态 s，递归从状态 s 出发进行动态决策。如果跳到第 k 个舞步时左脚在位置 x，右脚在位置 y 时所需要耗费的最少体力记为 $d[x,y,k]$，那么，合法的决策有：

决策一：把一只脚从中心移动到目标位置；

决策二：一只脚移动到相邻位置；

决策三：一只脚移动到相对位置；

决策四：一只脚在原地再踩一下。

则，状态转移方程可以写成：

$d[x,y,k] = \min\{d[a[k], y, k+1] + \text{effort}(x, a[k]), d[x, a[k], k+1] + \text{effort}(y, a[k])\}$

其中 effort(a, b)表示从位置 a 移动到位置 b 需要花费的体力，$A[k]$表示舞曲中第 k 个箭头的编号。

决策和多阶段决策问题　在本题中，状态转移方式是进行“决策”。对于有多个阶段的决策问题，如果具备最优子结构（最优子结构重新被描述为“最优决策序列的子序

列也是最优的”）和无后效性，就可以用动态规划来解决它。

【例题5】积木游戏[1]

SERCOI 最近设计了一种积木游戏。每个游戏者有 N 块编号依次为 1，2，…，N 的长方体积木。对于每块积木,其三条不同边分别称为“a 边”、“b 边”和“c 边”，如图 1-68 所示。

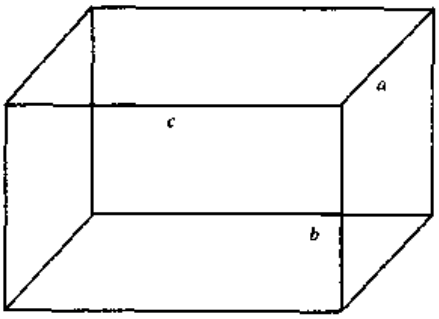


图 1-68　长方体积木

游戏规则如下：

1．从 N（N≤100）块积木中选出若干块，并将它们分成 M（1≤M≤N）堆，称为第 1 堆，第 2 堆，…，第 M 堆。每堆至少有 1 块积木，并且第 K 堆中任意一块积木的编号要大于第 K+1 堆中任意一块积木的编号（2≤K≤M）。

2．对于每一堆积木，游戏者要将它们垂直摞成一根柱子,并要求满足下面两个条件：

- 除最顶上的一块积木外，任意一块积木的上表面同且仅同另一块积木的下表面接触，并且要求下面积木的上表面能包含上面的积木的下表面，也就是说，要求下面积木的上表面两对边的长度分别大于等于上面积木两对边的长度。
- 对于任意两块上下表面相接触的积木，下面积木的编号要小于上面积木的编号。

3．根据每人所摞成的 M 根柱子的高度之和来决出胜负。

请编一程序，寻找一种摞积木的方案，使得所摞成的 M 根柱子的高度之和最大。

【分析】

因为题目要求编号小的堆积木编号较大，这不太自然，在不改变结果的前提下，把题目改作编号小堆的积木编号较小，这显然不会影响到最终的高度和，而且，此时每一种合理的堆放方法可看作按编号递增的顺序选择若干积木，按堆编号递增的顺序逐堆放置，每堆中积木依次在前一个上面堆放，而最终形成一种堆放方案。

这道题目的“决策”方式有三种：①当前块另起一堆；②当前块加在前一堆上(如果可能的话)；③当前块不使用。但这里就有一个问题了，如何判断当前块是否能加在前一块上。显然，如果只记录已经堆出的柱子根数 i，已经考虑了的积木个数 a，和顶上积木块编号 b，这个问题是无法判断的。需要增加一维状态参量，记录顶面的积木是哪一面朝上？修改后的状态为（i,a,b,k），其中 k 为顶面的编号（积木 b 的三个面分别编号为 1,2,3）。

问题表示与状态定义的区别　有了问题表示以后，状态的含义并不一定确定下来了。

[1] 题目来源：NOI97

这个(*i*,*a*,*b*,*k*)是什么意思呢？可以理解为从决策开始到状态（*i*,*a*,*b*,*k*）已经获得的最大高度，也可以理解为从（*i*,*a*,*b*,*k*）到决策结束还能获得的最大高度。

- 如果按照第一种理解，状态转移需要做的工作是对一个给定状态 *S*，列举出可以转移到 *S* 的所有状态，然后从这些前趋状态中取最优。
- 如果按照第二种理解，状态转移需要做的工作是对一个给定状态 *S*，列举出可以从 *S* 转移到的所有状态，然后从这些后继状态中取最优。

向前递推法和向后递推法　回忆 1.2 中介绍的“递推”，可以看出这两种理解不过是递推方向不同。第一种是向后递推，它的最后结果是结束状态的指标函数值；第二种是向前递推，它的最后结果是起始状态的指标函数值。在前面的例题中，决策往往是“对称”的，两种状态定义的递推难度差不多，但是本题不是这样！如果读者试着自己写出状态转移方程，会发现第二个方法的方程写起来自然很多，因为一个决策对应一个后继！因此采用第二种状态定义。

定义 *d*[*i*,*a*,*b*,*k*]为已经用前 *a* 个积木得到了 *i* 根柱子，目前顶面为积木 *b* 的面 *k*，以后还能获得的最大高度，则状态 *d*[*i*,*a*,*b*,*k*]可以使用的决策为：

决策一：新起一堆→*d*[*i*+1,*a*+1,*a*+1,*k*']，这个决策当 *i*<*m* 时合法。

决策二：加在当前堆→*d*[*i*,*a*+1,*a*+1,*k*']，如果积木 *a*+1 的 *k*，面可以被积木 *b* 的 *k* 面包含。

决策三：不使用当前块→*d*[*i*,*a*+1,*b*,*k*]，这个决策随时合法。

状态转移是那么自然，状态总数为 $O(n^2m)$，决策数目为 $O(1)$，状态转移时间为 $O(1)$，因此算法的时间复杂度为 $O(n^2m)$。

动态规划思路产生的两种方法　我们花了两节的内容分别用递归思想和多阶段决策思想阐明了动态规划思路的两种产生方法。这两种思路是对称的：用递归思路建立模型，容易算出状态的前趋，适合顺推；用多阶段决策建立模型，容易算出状态后继，适合逆推。

无后效性和子结构　它们的本质一样，即利用无后效性定义状态，进一步根据最优子结构定义状态的指标函数并进行状态转移。重叠子问题越多，动态规划的优越性也就越明显。状态和状态转移是动态规划的精髓，下面的一节将进行专门讨论。

状态表示　动态规划方法的核心是状态表示。状态的可定义性来源于问题的无后效性。由于状态是人为规定的，可以通过增加维数的方法来消除后效性，也可以在保持无后效性的前提下改变状态定义，以得到更多的重叠子问题。

状态转移　有了状态定义之后，思考的重点在于状态转移。状态转移的可定义性来源于问题的最优子结构。而当问题不具备最优子结构的时候，通常通过增加维数的办法来解决。但值得注意的是，这样的结果有可能使得子问题不重叠，从而使动态规划变得没有意义（回忆：动态规划的基本动机就是充分利用重叠子问题）。

动态规划的两种实现方式　即递推和记忆化搜索。递归很灵活（它可以避免无用的状态，而且书写简单，可以自顶向下“剪枝”），但无法用后面用到的很多优化如滚动数组减少空间消耗，状态合理组织降低时间复杂度等。具体用哪种方式读者可根据情况而定。

练　习　题

思考题：

1.5.1　用自己的话解释“最优子结构”。举出几个具有该结构的例子和不具有该结构的例子。

1.5.2　是否所有的递归过程都能转化为“动态规划”。

1.5.3　动态规划方法为什么比普通的递归快？

1.5.4　回忆 1.2 节中对递归和递推的比较。本节中提到动态规划的两种方法——记忆化搜索和递推的关系是否一样？这对你有什么启发？

1.5.5　“决斗”一题如果给出的是每两人决斗时各自的胜利概率，求一个让某人胜利概率最高的方案，动态规划还能奏效吗？这说明动态规划模型具有什么性质？

1.5.6　总结动态规划思路产生的两种方法。这两种思路本质上是一样的吗？

1.5.7　状态表示、状态转移、最优子结构和无后效性有什么联系？

编程题：

1.5.8　**艺术馆的火灾**[1]

这幢古老的建筑是一个艺术馆，它珍藏着上百件绘画、雕塑以及其他艺术品，就连建筑本身也是一件艺术。但是，岁月并不在乎它的精致与美丽，时光在渐渐剥夺着这幢木屋的生命。终于，在一个月色昏暗的夜晚，它着火了。

艺术馆是一幢两层的小楼，每一层有 N 个房间，每个房间中收藏的艺术品的价值都可以用一个正整数来表示。在消防队员火速赶到的时候，火势已经蔓延了整个建筑，消防队员们观察每个房间中的火势，将它们分别用一个正整数来表示。下面是一个 N=4 的例子，每个房间的第一数为艺术品价值，第二个数为该房间的火势。在这个例子中，二层楼的第四个房间中艺术品的价值最大，为 60，而一层楼的第四个房间中艺术品的价值仅为 20；二层楼的第四个房间中火势最强，为 70。而一层楼的第三个房间中火势较弱，为 20。如 1-69 所示：

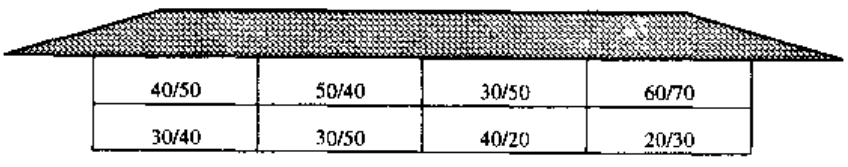


图 1-69　艺术馆火势和艺术品价值

由于火情紧急，消防队员们准备使用一种新型的灭火器。这种灭火器只能发射 K 次，每次发射将覆盖一个矩形的区域（矩形的高度可以是 1 也可以是 2）。它的威力巨大，所

[1] 题目来源：IOI2000 中国国家集训队原创试题. 命题人：张辰

到之处不但火焰会被扑灭，就连同在一处的艺术品也难以幸免。因此，如何善用这种灭火器成了最大的问题。

给出艺术馆每层的房间数 N 和灭火器的发射次数 K，以及每个房间中艺术品的价值和火势，你需要决定灭火器每次应该怎样发射（也可以不发射），才能将这次火灾的损失降到最低限度。这个损失用你所摧毁的艺术品的总价值，加上剩余的火势总值（这些火焰将需要消防队员们亲身去扑灭）来衡量。

****1.5.9　机器人的名字**[①]

公元 2020 年，数量极为巨大的智能化机器人充斥了整个星球，每个机器人都有一个仅由英文字母和空格组成的名字，并且不能出现两个机器人重名的情况(正如你所知道的那样，在这个全球联网的时代，重名就意味着将充电账单记在另一个机器人名下等严重的问题)，所以，机器人的名字已经出现了不断增长的趋势。本来，使用数字命名法是解决问题的很好方法，但是，新通过的 ARR(Act of Robot Rights)，已经禁止了数字命名等歧视机器人的行为，而且赋予了机器人在不导致重名的前提下，选择个性化名字的权利，这无疑进一步助长了盲目加长名字的倾向。

很快，这一趋势就带来了许多麻烦，如存储器缺货等。所以，CRR(Commission on Robot Rights)需要你编写一个能够将机器人的名字尽可能压缩的程序来解决这个问题，并输出名字的最短压缩长度，以便 CRR 能够为每个机器人分配一个大小适当的 MON(Memory of Name)。

考虑到各种机器人的知识水平(有相当数量的机器人不能理解哈夫曼编码)，以及他们起名字的习惯(很多机器人喜欢使用有大量重复段的名字)，所以，你被要求使用一种比较易于理解的压缩方法：对名字中的重复子串进行压缩，即用 $[S_i]k$ 表示 k 个相同的子串 S_i(其中，S_i 称为重复子串，k 是一个单字节整数，只占一个字符位置)，如果这 k 个子串并没有连在一起，则可以在 $[S_i]k$ 的后面加上 $\{S_1\}\ t_1\{S_2\}\ t_2 \ldots \{S_r\}\ t_r$（其中，$1<t_i<k$，$t_i < t_{i+1}$，$i=1,\cdots,r$），表示在第 t_i 个 S_i 的后面放置 S_i，S_i 称为插入子串。而且允许嵌套压缩，即 S_i 和 S_i 也都可以是压缩后的字符串。

比如某机器人的名字是 I_am_WhatWhat_is_WhatWhat，则压缩后的结果为 I_am_[What]4{_is_}2，长度为 19(例子中的空格用下划线"_"表示，数字 2 和 4 实际上是用单字节二进制表示的)。

注意：名字不会以空格开始或结尾，大小写字母也不视为相同。

1.5.2　常见模型的分析

在本节中，分析几个最常见的模型。建议读者在掌握这些题目的同时思考状态表示和状态转移通常是怎样的？为什么会这样？

线性模型　线性模型是最容易导致动态规划算法的。只要把线性结构分成两部分，往

[①] 题目来源：IOI1999 中国国家集训队原创试题，命题人：齐鑫

往可以用递归来解决。另一个想法是每次增加一个元素，逐步扩大考虑范围，这里举几个例子。

【例题 1】方块消除[①]

Jimmy 最近迷上了一款叫做方块消除的游戏。游戏规则如下：n 个带颜色方格排成一列，相同颜色的方块连成一个区域（如果两个相邻方块颜色相同，则这两个方块属于同一区域），如图 1-70 所示。游戏时，你可以任选一个区域消去。设这个区域包含的方块数为 x，则将得到 x^2 个分值。方块消去之后，其余的方块就会竖直落到底部或其他方块上。而且当有一列方块被完全消去时，其右边的所有方块就会向左移一格。

图 1-70　方块消除游戏举例

如图 1-70，依次消去灰、白、黑三个区域后，你将得到 $4^2+3^2+2^2=29$ 分。虽然游戏很简单，但是要拿高分也很不容易。Jimmy 希望你能找出得最高分的最佳方案，你能帮助他吗？

【分析】

先看一个颜色序列

1 1 1 3 3 2 4 4 4 5 5

根据消方块的规则，连在一起的相同颜色方块可以合并，比如上面颜色序列表示成：(1,3),(3,2),(2,1),(4,3),(5,2)，其中(a,b)表示有 b 个颜色为 a 的方块连在一起。

于是题目的方块可以表示成 color[i]，len[i]，$1\leqslant i\leqslant l$，这里 l 表示有多少“段”不同的颜色方块。color[i]表示第 i 段的颜色，len[i]表示第 i 段的方块长度。比如上面的例子：l=5, color[1…5]=(1,3,2,4,5), len[1…5]=(3,2,1,3,2)

让 f [i,j,k]表示把(color[i], len[i]), (color[i+1],len[i+1]), …, (color[j−1],len[j−1]), (color[j], len[j] + k)合并的最大得分。

考虑(color[j], len[j]+k)这一段，要不马上消掉，要不和前面的若干段一起消。

- 如果马上消掉，就是 f [i,j−1,0]+sqr(len[j]+k);
- 如果和前面的若干段一起消，可以假设这“若干段”中最后面的那一段是 p，则此时的得分是 f [i,p,k+len[j]]+f [p+1,j−1,0]。

于是 f [i,j,k]=max{f [i,j−1,0]+(len[j]+k)^2, f [i,p,k+len[j]]+f [p+1,j−1,0]}

其中，color[p]=color[j], $i\leqslant p<j$，边界条件是 f [i,i−1,0]=0

请读者仔细体会这个算法。从不好处理的“若干段”只提取了最后一段 p，成功地设计出了动态规划算法。但是这个算法是 $O(n^4)$，时间效率很低。我们发现更新解的必要条件是 color[p]=color[j]，能不能有效地利用这个条件来优化算法呢？这个问题留给读者思考。

【例题 2】公路巡逻[②]

在一条没有分岔的公路上有 n（$n\leqslant 50$）个关口，相邻两关口之间的距离是 10km。所

[①] 题目来源：IOI2003 中国国家集训队讨论题目

[②] 题目来源：CTSC2000. 命题人：李申杰

有车辆在公路上的最低速度为 60km/h，最高速度为 120km/h，且只能在关口处改变速度。

有 $m(m\leqslant 300)$辆巡逻车分别在时刻 T_i 从第 n_i 个关口出发，匀速行驶到达第 n_i+1 个关口，路上耗费时间为 t_i（s）。两辆车相遇指他们之间发生超车现象或同时到达某个关口。

求一辆于 6 点整从第 1 个关口出发去第 n 个关口的车最少会与多少辆巡逻车相遇，以及在此情况下到达第 n 个关口的最早时刻。所有车辆到达关口的时刻必须是整秒。

【分析】

用 $d[i,T]$表示在时刻 T 到达第 i 个关口的途中与巡逻车最少相遇次数，状态转移方程为：

$$d[i,T]=\min_{300\leqslant t\leqslant 600}\{d[i-1,T-t]+w[i-1,T-t,T]\}$$

如果把 6 点整记作时刻 0，则边界条件为 $d[1,0]=0$，问题的解为 $\min\{d[n,T]\}$。

函数 $w[i-1,T-t,T]$是目标车于时刻 $T-t$ 从第 $i-1$ 个关口出发，于时刻 T 到达第 i 个关口途中与巡逻车相遇的次数。状态总数为 $O(kn^2)$，决策数目为 $O(k)$，时间复杂度还依赖于 w 的计算。

方法一：在每个决策中都进行一次计算，对所有从第 i 个关口出发的巡逻车进行判断，由于每辆巡逻车恰好被判断一次，故这样每个状态的计算 w 的平均时间复杂度为 $O(m/n)$，算法总时间复杂度为 $O(kn^2)\times O(k)\times O(m/n)=O(k^2nm)$。

方法二：仔细观察状态转移方程可以发现，在对状态 $d[i,T]$进行转移时，所计算的函数 w 都是从第 i 个关口出发的，而且出发时刻都是 T，只是相应的到达时刻不同。能否考虑这些车之间的联系，从而利用已经得出的结果，减少重复运算呢？

我们来考虑 $w[i,T,k]$与 $w[i,T,k+1]$之间的联系。这两种情况只“差一点儿时间”，结果也应该很相似，因此希望能很快从前者推出后一个。为了叙述方便，设前者对应的目标车为车 A，后者对应的车为车 B。

对于每辆从第 i 个关口出发的巡逻车，设其出发和到达时刻分别为 S_{time} 和 T_{time}，则

- 若 $T_{time}<k$ 或 $T_{time}>k+1$，则目标车 A、目标车 B 与该巡逻车的相遇情况相同；
- 若 $T_{time}=k$，则目标车 A 与该巡逻车相遇，对目标车 B 的分析又分为，若 $S_{time}\leqslant T$，则目标车 B 不与该巡逻车相遇，否则目标车 B 也与该巡逻车相遇；
- 若 $T_{time}=k+1$,则目标车 B 与该巡逻车相遇，对目标车 A 的分析又分为，若 $S_{time}\leqslant T$，则目标车 A 不与该巡逻车相遇，否则目标车 A 也与该巡逻车相遇；

令 $g[k]=w[i,T,k+1]-w[i,T,k]$，由上述讨论得：

$$g[k]=G((T_{time}=k+1)\ \text{且}\ (S_{time}\leqslant T))-G((T_{time}=k)\ \text{且}\ (S_{time}\leqslant T))。$$

其中函数 $G(P)$表示所有从 i 个关口出发，且满足条件 P 的巡逻车的数目。

这样就找到了函数 w 之间的联系。于是，在对状态 $d[i,T]$进行转移时，先对所有从第 i 个关口出发的巡逻车进行一次扫描，在求出 $w[i,T,T+300]$的同时求出 $g[T+301..T+600]$，这一步的时间复杂度为 $O(m/n)$。在以后的状态转移中，由 $w[i,T,k+1]=w[i,T,k]+g[k]$，仅需 $O(1)$ 的时间就可以求出函数值 w，状态转移时间仅为 $O(1)$。算法总时间复杂度为 $O(kn^2)\times(O(m/n)+k\times O(1))=O(knm+k^2n^2)$。由于 $n\leqslant 50$, $m\leqslant 300$，所以方法二比方法一快得多。

【例题 3】并行期望值[1]

考虑一个可以并行执行两个高级语言程序的机器。每个高级语言程序由若干条指令构成均为赋值指令，它的形式是：

<变量> := <运算数 1> op <运算数 2>

其中，变量名由不超过 20 个字母组成。运算数是一个变量名或者小于 100 的正整数。op 是运算符，它可以是加号“+”或者减号“-”。

执行之前，高级语言程序首先被翻译成机器语言。一条语句 X := Y + Z 被翻译成：

```
Mov R1, Y
Mov R2, Z
Add R1, R2
Mov X, R1
```

Mov 指令把第二个操作数送到第一个中去，Add(Sub)操作进行加法（减法），并把结果保存在第一个操作数中。注意 Y 和 Z 代表变量或者整数常量。X := Y - Z 的机器语言代码类似，只是 Add 指令被 Sub 指令代替了。

处理器接受了两个机器语言程序后，每次随机选一个程序，执行它的下一条指令。如果某个程序执行完毕，处理器会连续把另一个程序执行完。两个程序共享所有变量（一开始的时候各个变量的值均为 0），但是拥有两个独立的寄存器 R1 和 R2。

由于执行顺序不确定，最后各个变量的值也是不确定的。不过，可以计算出每个变量在所有情况下最终值的平均数（即并行期望值）。现在给你两个高级语言程序，请编程算出所有变量的并行期望值。每个高级语言程序最多有 25 条指令，两个程序最多共使用 10 个变量。

【分析】

首先，读入数据并将高级语言程序转换成机器语言程序并保存起来，每次将一条高级语言的赋值指令：

<变量> := <运算数 1> op <运算数 2>

转换成形式一致的四条机器语言的赋值指令：

<Rx1> := <运算数 1> + <0>

<Rx2> := <运算数 2> + <0>

<Rx1> := <Rx1> op <Rx2>

<变量> := <Rx1> + <0>

如果是转换程序 1 中的高级指令，那么 $x = a$；如果是转换程序 2 中的高级指令，那么 $x = b$。这使得两程序分别拥有了独立的寄存器 Ra1、Ra2 和 Rb1、Rb2，同时又将寄存器一般化为普通变量，方便了以后的处理。接下来的讨论均是针对机器语言的。

指令 (w,i) 表示程序 w 的第 I 条指令。变量 $d[x,y,k]$ 表示程序 1 执行完毕 x 条指令，程序 2 执行完毕 y 条指令后（即状态<x,y>）变量 k 的并行期望值。序列 $\{p_1, p_2, p_3, \cdots, p_{x+y}\}$ 是一条达到状态<x,y>的路径的具体描述，其中 $p_i = 0$ 表示第 i 次机器选择程序 1 中的指令

[1] 题目来源：ACM/ICPC Regional Contest Tehran 2001

执行，否则相应的，$p_i=1$ 表示第 i 次机器选择程序 2 中的指令执行。按照序列$\{p_1, p_2, p_3, \cdots, p_{x+y}\}$并行执行两个程序所得变量 k 的值称为状态$<x,y>$下序列 p 的 k 值。很明显，

$$d[x, y, k]=\frac{\text{状态}<x, y>\text{下所有不同序列的}k\text{值和}}{\text{状态}<x, y>\text{下不同序列的总数}}$$

根据排列组合初步知识，状态$<x,y>$下不同序列的总数 $=C_{x+y}^{x}$。

考虑 $x,y>0$ 的一般情况，状态$<x,y>$的 k 值和 = 状态$<x-1,y>$执行（1,x）后的 k 值和（记为 k_{sum1}）+ 状态$<x,y-1>$执行(2,y)后的 k 值和（记为 k_{sum2}）。怎么求状态$<x-1,y>$执行（1,x）后的 k 值和呢？分情况讨论：

（1）指令(1,x)中被赋值的变量不是 k，那么 k_{sum1} = 状态$<x-1,y>$的 k 值和。

（2）指令(1,x)形如 $k:=<1\_a>\ op\ <1\_b>$

为方便叙述，这里把常数看作拥有固定值的变量。假设状态$<x-1,y>$下序列 P_i 的 k 值为 K_i，1_a 值为 A_i，1_b 值为 B_i，易知，序列$\{P_i,0\}$的 k 值为 A_i op B_i。现在考虑状态$<x-1,y>$下的所有序列 P_1，P_2，P_3，…，P_{sum}，它们执行指令(1,x)后的 k 值分别为 A_1 op B_1、A_2 op B_2、A_3 op B_3、…、A_{sum} op B_{sum}。

Sum 是个巨大的数，大到时间和空间上都不允许记录每个序列 P_i 相应的 A_i 和 B_i 值。但幸运的是，我们无需知道 A_i,B_i 的具体值。为什么呢？由于加法交换律及合并律的存在，

$$K_{sum1}=\sum_{i=1}^{sum}K_i=\sum_{i=1}^{sum}(A_i\ \text{op}\ B_i)=(\sum_{i=1}^{sum}A_i)\ \text{op}\ (\sum_{i=1}^{sum}B_i)$$

也就是说，只要知道状态$<x-1,y>$的 A 值和、B 值和，K_{sum1} 就通过简单的 op 运算得出了。发现了问题的最优子结构，剩下的工作就很简单了。

$$\text{令 } K_1=\begin{cases}d[x-1, y, k] & \text{(情况(1))}\\ d[x-1, y, 1\_a]\ \text{op}\ d[x-1, y, 1\_b] & \text{(情况(2))}\end{cases}$$

$$K_2=\begin{cases}d[x, y-1, k] & \text{(情况(1))}\\ d[x, y-1, 2\_a]\ \text{op}\ d[x, y-1, 2\_b] & \text{(情况(2))}\end{cases}$$

然后整理一下得：状态$<x,y>$的 k 值和$=K_{sum1}+K_{sum2}=K_1\times C_{x+y-1}^{x-1}+K_2\times C_{x+y-1}^{y-1}$。状态转移方程为：

$$\text{d}[x,y,k]=\frac{K_1\times C_{x+y-1}^{x-1}+K_2\times C_{x+y-1}^{y-1}}{C_{x+y}^{x}}=\frac{K_1\times\dfrac{(x+y-1)!}{(x-1)!\,y!}+K_2\times\dfrac{(x+y-1)!}{x!(y-1)!}}{\dfrac{(x+y)!}{x!\,y!}}=\frac{K_1\times x+K_2\times y}{x+y}$$

时间复杂度仅为 $O(l_1l_2n)$，其中 l_1, l_2 分别为两个程序的长度，n 为变量个数。本题的巧妙之处是发现了最优子结构，这也是在线性模型中设计动态规划的主要障碍。

【例题 4】高性能计算机[1]

现在有一项时间紧迫的工程计算任务要交给你——国家高性能并行计算机的主管工程师——来完成。为了尽可能充分发挥并行计算机的优势，我们的计算任务应当划分成若干

[1] 题目来源：IOI2001 中国国家集训队冬令营试题，命题人：齐鑫

个小的子任务。

这项大型计算任务包括 A 和 B 两个互不相关的较小的计算任务。为了充分发挥并行计算机的运算能力，这些任务需要进行分解。研究发现，A 和 B 都可以各自划分成很多较小的子任务，所有的 A 类子任务的工作量都是一样的，所有的 B 类子任务也是如此（A 和 B 类的子任务的工作量不一定相同）。A 和 B 两个计算任务之间，以及各子任务之间都没有执行顺序上的要求。

这台超级计算机拥有 p 个计算结点，每个结点都包括一个串行处理器、本地主存和高速 cache。然而由于常年使用和不连贯地升级，各个计算结点的计算能力并不对称。一个结点的计算能力包括如下几个方面：

（1）就本任务来说，每个结点都有三种工作状态：待机、A 类和 B 类。其中，A 类状态下执行 A 类任务；B 类状态下执行 B 类任务；待机状态下不执行计算。所有的处理器在开始工作之前都处于待机状态，而从其他的状态转入 A 或 B 的工作状态（包括 A 和 B 之间相互转换），都要花费一定的启动时间。对于不同的处理结点，这个时间不一定相同。用两个正整数 t_i^A 和 t_i^B(i = 1，2，…，p)分别表示结点 i 转入工作状态 A 和工作状态 B 的启动时间（单位：ns）。

（2）一个结点在连续处理同一类任务的时候，执行时间——不含状态转换的时间——随任务量（这一类子任务的数目）的平方增长，即，若结点 i 连续处理 x 个 A 类子任务，则对应的执行时间为 $t = k_i^A x^2$；类似地，若结点 i 连续处理 x 个 B 类子任务，对应的执行时间为：$t = k_i^B x^2$。其中，k_i^A 和 k_i^B 是系数，单位是 ns，$i = 1，2，\cdots，p$。

任务分配必须在所有计算开始之前完成，所谓任务分配，即给每个计算结点设置一个任务队列，队列由一串 A 类和 B 类子任务组成。两类子任务可以交错排列。

计算开始后，各计算结点分别从各自的子任务队列中顺序读取计算任务并执行，队列中连续的同类子任务将由该计算结点一次性读出，队列中一串连续的同类子任务不能被分成两部分执行。

现在需要编写程序，给这 $p(\leqslant 20)$个结点安排计算任务，使得这个工程计算任务能够尽早完成。假定任务安排好后不再变动，而且所有的结点都同时开始运行，任务安排的目标是使最后结束计算的结点的完成时间尽可能早。两类子任务的数目都不超过 60 个。

【分析】

很明显，该问题可以分成两个子问题：

（1）计算第 i 个结点完成 a_i 个 A 任务和 b_i 个 B 任务需要的最短时间。

（2）给第 i 个结点分配 a_i 个 A 任务和 b_i 个 B 任务，取所有结点运行时间的最大值作为实际运行的总时间。

对于子问题（1），很容易验证该问题的最优子结构和无后效性，因此很容易想到一个动态规划算法：让 $t[\text{Task},a_i,b_i]$表示结点 p_i，还有 a_i 种 A 任务和 b_i 种 B 任务，并且当前需要执行 Task 类任务（Task=A 或 B）所需要的最短时间，那么状态转移方程是：

$$t[A,a_i,b_i] = \min_{1\leqslant i\leqslant a_i}\{t[B,a_i-i,b_i]+t_a+k_a i^2\}$$

$$t[B,a_i,b_i] = \min_{1\leqslant i\leqslant b_i}\{t[B,a_i,b_i-i]+t_b+k_b i^2\}$$

边界是 $d[A,0,0]=d[B,0,0]=0$；子问题 1 的解是 $time[p_i,a_i,b_i]=\min\{d[A,a_i,b_i],d[B,a_i,b_i]\}$。

如果我们一个一个结点地分配任务，实际上就是把子问题 2 看成了一个多阶段决策问题，同样很容易证明它的无后效性并且具备最优子结构。因此我们可以把结点作为阶段进行动态规划，用 $d[p_i,a_i,b_i]$代表前 p_i 个结点完成 a_i 个 A 任务和 b_i 个 B 任务的最短时间，则状态转移方程为：

$$d[p_i,a_i,b_i]=\min_{0\le a\le a_i,0\le b\le b_i}\{\max\{d[p_i-1,a_i-a,b_i-b],\mathrm{Time}[p_i,a,b]\}\}$$

边界是 $d[1,a,b]=\mathrm{Time}[p_i,a,b]$。

这样，算法时间复杂度为 $O(pn_a^2n_b^2)$，由于递推的时候只需要保存相邻两层的状态，空间是完全可以承受的。另外，虽然时间复杂度比较高，但是使用一些小技巧进行优化之后，所有数据均能在 5 秒之内解出。

问题（1）的解仅是由 t_a,t_b,k_a,k_b,a_i,b_i 这 6 个数据决定的，能不能不递推而直接计算呢？

我们知道，两种任务是交替进行的，因此任务的实际运行方案也可以用两个序列来表示，也就是两种任务被分成的各个块的长度。$La_1,La_2,\cdots,La_m$；$Lb_1,Lb_2,\cdots,Lb_n$；

其中 $m=n\pm1$，则总时间可以表示为 TotalTime=Time$A$+Time$B$，其中

$$\mathrm{Time}A=t_a*m+k_a*\sum_{i=1}^{m}La_i^2,\quad \mathrm{Time}B=t_b*m+k_b*\sum_{i=1}^{n}Lb_i^2$$

显然，A 和 B 的分块是独立的，而且是对称的，因此下面我们只讨论任务 A。注意到任务 A 每块的执行方式对于结果没有任何影响，因此我们不妨设 $La_1\le La_2\le\cdots\le La_m$。直观地看，如果块数一定，则每块的长度越平均越好，于是有下面的猜想。

猜想：对于最优方案所有的 $i,j(1\le i,j\le m)$,有 $La_j-La_i\le1$，即所有块的长度最多只有两种。

证明如下：假设在最优方案中存在 i,j，使 $La_j-La_i>1$，那么记 $La_i'=La_i+1$，$La_j'=La_j-1$。显然，两个新的块长度方案仍然是可行的，下面比较这两个方案。

$$\mathrm{Time}\,A'-\mathrm{Time}\,A=k_a(La_i'^2+La_j'^2-La_i^2-La_j^2)=2k_a\times(La_i-La_j1)<0$$

故新方案更优，与方案的最优性矛盾。命题得证。

这样，只要有了任务 A 的块数 m,其执行时间时间很容易算出来，而且前面已经提到过，对于任务 B 的块数 m，只有三种情况：即分成 $m-1,m,m+1$ 块。这样，枚举 m（至少为 1，最多为 A 任务的个数）和 n（必须是 $m-1m$,或 $m+1$,而且要大于零，小于或等于 B 任务的个数），最多枚举 60×3=180 次就得到结果了。但是这个改进是否十分有效呢？不见得。第一，动态规划方法可以一次递推到很多数据，本方法一次只能得到一个。第二，本算法的时间复杂度不算很高，实际运行时间是不长的。不过，也应该看到，这一理论上的结果可以对后面的分析提供一个有用的结论，并且为上界的直接计算创造了可能性。

回到子问题 2。如果记录以下递推得到的中间数据，会发现在决策枚举中，常常遇到这样的情况：开始分配给结点的任务少，结点本身的运行时间 t_1 倒是少了，但是由于剩下的任务多，后面结点的运行时间 t_2 很大。由于最后是取较大值，当前结点的短时间不能给我们带来任何好处！注意到随着 t_1 增大,t_2 一般会减小，在 t_1 和 t_2 接近的时候将到达一个峰值，一个自然的想法产生了：二分法,在枚举分配任务 B 数目的时候进行。但是仔细一想就

会发现这其中的问题：t_2随着t_1不一定是单调递减的[1]，二分法不能保证得到正确的解。

不能二分，就没有别的办法了吗？当然不是。可以利用搜索中剪枝的思想（参见1.6节）做决策枚举的优化，首先用贪心法得到一个时间上限T，然后把原算法的五重循环改为：

```
For pi:=1 to p do
Begin
  递推time数组(子问题1)
  设置d[ai,bi]的初始值
  for a:=0 to na do
  for b:=0 to nb do
   for time[a,b]<=T then
     for ai:=a to na do
      for bi:=b to nb do
        {递推}
end;
```

也就是先枚举决策，再枚举状态。这样虽然有点怪，但是如果time[a,b]>T,就可以直接跳到下个b的值，从而避免了里面的a_i, b_i两重循环。因此这样做是十分值得的。那么上界T怎样求呢？刚刚推导的time[a,b]计算方法就是一个很好的工具。

由于直接计算方法太粗糙，为了推导方便，把它改成有近似公式。把A和B都分成m块，每块近似看成相等的（允许分数），则有

$$\text{Totaltime=Time } A+\text{Time } B \approx (t_a+t_b)m+k_a m(a_i/m)^2+k_b m(b_i/m)^2=m(t_a+t_b)+(k_a a_i^2+k_b b_i^2)/m$$

由于取使时间最小的m,由基本不等式，有

$$\text{TotalTime} \approx 2\sqrt{(t^A+t^B)*(k^A*a_i^2+k^B*b_i^2)}$$

显然，TotalTime的近似值与a_i,b_i的值有关，必须把与a_i,b_i无关的部分提取出来作为结点的“能力值”，然后根据“能力值”按比例进行任务分配，也就是说，不同结点分得的任务数正比于其“能力值”。最后利用前面的方法直接计算出这种分配方法对应的时间作为上界T。由于任务有两种，对应于不同的任务，结点表现出来的“能力值”应当不同。于是有：

上界1：根据完成A和B的能力对任务A和B分别进行分配。

其中，结点完成任务A和B的“能力值”分别由下面的公式计算[2]：

$$\begin{cases} \text{Ability}A=\sqrt{\dfrac{1}{k^A\times(t^A+t^B)}} \\ \text{Ability}B=\sqrt{\dfrac{1}{k^B\times(t^A+t^B)}} \end{cases}$$

[1] 例如存在这样的情况：给当前结点增加一个B任务后，A任务能更充分的“分块”,导致t_1减少，同时，由于剩下的任务少了，t^2也减少

[2] 在实现过程中，还需要调整各个结点的任务数以保证任务总和分别是na和nb

结点所分得的任务 A 数正比于它的 Ability$A$，而任务 B 数正比于它的 Ability$B$。

考虑到有的机器的 Ability$A$ 和 Ability$B$ 差别较大，两种任务分别分配可能造成结点的 A，B 任务数目相差大，从而使较多的任务不能充分地分成小块，增加了实际运行时间。为了尽可能地避免这样的事情发生，于是又有了：

上界 2：按照结点总体能力把 A 和 B 统一分配得到上界。

也就是 $\text{Ability}=\sqrt{\frac{1}{(k^A+k^B)\times(t^A+t^B)}}$。结点分得的任务 A 数和任务 B 数都正比于这个“综合能力值” Ability。程序运行情况如表 1-16 所示。

表 1-16　程序运行情况

数　据	1	2	3	4	5	6
答　案	93	203	27181	1941	1182	833
上界 1	179	275	27181	4262	2717	2456
上界 2	179	259	27181	2402	2198	1508
时　间	<0.01	<0.01	<0.01	0.22	0.55	0.60

上界做了两个，还需要继续改进吗？不必了，因为即使程序直接把结果作为上界，运行结果只能快不到 0.1 秒，比现在的程序快不了多少，我们的上界已经是比较令人满意的了。

串模型　串模型也是一种线性模型，但是由于串有特殊的“匹配”、“取前缀后缀”操作，因此很多情况下他们的模型具有一些一般线性结构所没有的特点。

【例题 5】模板匹配[①]

考虑只使用*的通配符，这里*可以代表零个或多个字符。通配符 Q 称为两个通配符 P_1 和 P_2 的公共模板，如果被 Q 匹配的字符串一定既能被 P_1 匹配也能被 P_2 匹配。例如 *ba*ab* 就是**ab**和 *ba*b* 的公共模板。P_1、P_2 的一个模板集合 Q_1，Q_2，…，Q_n 成为完备的，如果每个 Q_i 都是 P_1、P_2 的公共模板，且任何一个既能被 P_1 匹配又能被 P_2 匹配的字符串至少能被一个 Q_i 匹配。给两个通配符 P_1，P_2(不超过 20 个字符，最多包含 6 个*)，求出：

1．P_1，P_2 的一个公共模地板。

2．一个不超过 6 666 个模板的完备模板集合。

3．包含模板数目最少的完备模板集合。

例如，对于 *ba*ab* 和**ab**，一个包含最少数目模板的完备集合为：{*ba*ab*b*, *bab*b*, *ba*ab*, *bab*}

【分析】

在解决问题之前，先把问题用一个更简单的形式叙述一下。如果把一个模板看作一个集合（即它能匹配到所有字符串集合），则模板包含关系等同于集合包含关系。本题的核心是：给出两个集合 P_1 和 P_2，求 P_1 和 P_2 的一个交集。任务 1 是求交集的任何一个子集；任务 2 是求交集的一个覆盖。其思想是枚举每一个种同时被两个模板匹配的串 s，则只要每

① 题目来源：CEOI 2001. Pattern

种情况都被一个模板覆盖就可以了。为了说明这个问题，来看四个例子：

例 1：a*和 b*没有交集，因为无论公共串 s 的第一位是什么都不行。

例 2：a*b 和 ab*有交集，且公共串 s 的第一位只有一种情况：a，剩下的任务是要让 s 从第 2 位起同时匹配*b 和 b*。注意：这里用到了递归的概念，如果令子问题(i,j)表示需要匹配 P_1 从第 i 位起的剩下串和 P_2 从第 j 位起的剩下串，则在此种情况下，(i,j)转化为了$(i+1,j+1)$。

例 3：a*b 和*ab 有交集，且 s 的第一位必须是 a，和刚才一样，用递归的概念。显然对于 P_1 来说还需要匹配*b，但是对于 P_2 来说，可以匹配 ab，也可以匹配*ab，甚至可以匹配 b。则(i,j)转化为了$(i+1,j)$、$(i+1,j+1)$和$(i+1,j+2)$。显然，为了解决任务 1，我们只需要转换到任一种状态，而为了得到“集合覆盖”（任务 2），这三种情况都要考虑。

例 4：*ab 和*b 有交集，且 s 的第一位随便是什么都可以，用*表示（想一想，为什么）。现在状态转移到了什么地方呢？是$(i+1,j)$和$(i,j+1)$（想一想，为什么？）。

把四个例子一般化，就得到了本题的递归解法。注意，下面的状态转移和刚才提到的有所不同，请读者仔细思考为什么。设 $s(i,j)$为匹配 P_1 第 i 位起的字符串和 P_2 第 j 位起的字符串所得到完备模板集合，记 $c+s(i,j)$为字符 c 连接 $s(i,j)$中每个字符串后所得到的新集合，则：

$$s[i,j]=\begin{cases}\varnothing & p_1[i]\neq p_2[j], p_1[i]\neq'*', p_2[j]\neq'*'\\ \{c+s[i+1,j+1]\}, & p_1[i]=p_2[j]=c\neq'*'\\ s[i,j+1]\cup\{p_1[i]+s[i+1,j]\}\cup\{p_1[i]+s[i+1,j+1]\} & p_1[i]\neq'*', p_2[j]='*'\\ s[i+1,j]\cup\{p_2[j]+s[i+1,j]\}\cup\{p_2[j]+s[i+1,j+1]\} & p_1[i]='*', p_2[j]\neq'*'\\ \{'*'+s[i+1,j]\}\cup\{'*'+s[i,j+1]\} & p_1[i]=p_2[j]='*'\end{cases}$$

和普通的动态规划不一样，本题的状态值不是一个数，而是一个字符串集合。但同样可以借用它的思想，递推出所需要的答案。需要注意的是，在进行集合操作的时候需要合并相同的字符串。模板为什么不超过 6 666 个呢？题目中说了，每个字符串中‘*’中最多出现 6 次。而只有在 $P_1[i]$和 $P_2[j]$至少有一个是‘*’时集合元素才会增加。请读者计算一下，在不合并重复模板的情况下，最坏情况会得到多少个字符串？

本题也可以不考虑后缀而考虑任意子串，即 $s[i_1, j_1, i_2, j_2]$表示匹配 $P_1[i_1, j_1]$和 $P_2[i_2, j_2]$得到的完备模板集合，则虽然状态总数增加，但是可以根据情况避免生成不必要的模板，该算法的分析留给读者思考。另外，本题可以从两边一起递推的，这样的状态会少得多（像 1.6 节将介绍的双向广度优先搜索一样，我们扩展状态的一边），有兴趣的读者可以试试。

【例题 6】不可分解的编码[6]

给定 $n(n\leqslant 20)$个 01 编码串 $S_1,S_2,\cdots,S_n$，你的任务是寻找一个编码串 T，使得它至少可以被分解为两种不同的 S_i 的排列。例如：

给定 5 个 01 编码串：S_1=0110，S_2=00，S_3=111，S_4=001100，S_5=110。那么一个符合要求的编码串 T 是：001100110，它有以下两种分解方法：

00+110+0110 (S_2+S_5+S_1) 或 001100+110 (S_4+S_5)

[6] 题目来源：ACM/ICPC World Finals 2002

而 0110110 就不符合要求，它只有一种分解方法 0110+110 (S_1+S_5)。

你要寻找长度最短的符合要求的编码串 T，若有多个符合要求的最短编码串 T，则输出字典顺序最小的。

【分析】

考虑两种策略。

算法 1：搜索策略

在题目给定的规模下，搜索策略是有效的。搜索的关键是要抓住编码串 T 的性质，也就是至少存在两种不同的分解方法。从搜索分解方法出发，同时搜索两种分解方法，可以大大减少搜索量。还是来看上面的例子，如表 1-17 所示。

表 1-17　搜索步骤

Step	T	分解方法 1	分解方法 2
1	00	S_2	空
2	001100	S_2+1100	S_4
3	001100	S_2+S_5+0	S_6
4	001100110	S_2+S_5+S_1	S_4+S_5

搜索完成，T=001100110。

从上面的例子可以看出，搜索实际上是在两种分解方案的交替延伸中进行的。在搜索过程中，总是存在一种可行的完整的分解方案，而需要枚举延长另一种分解方案，这可能会导致 T 串的延长(如 Step1 到 Step2)，这样知道两种分解方案都可行且完整，也就找到了一个可行解。从所有可行解中寻找长度最短的就是所求的。

算法 2：动态规划

这道题还可以用动态规划求解。它仍然是采用与搜索策略同样的思路，所不同的是引入了记忆化的方法。仔细分析可以发现，在搜索的过程中，不完整的分解方案所无法匹配的剩余部分，如 Step2 中的 1100 和 Step3 中的 0，一定是某个 S 编码串的后缀。于是，只要将所有的 S 编码串的后缀定义为状态，在搜索的过程中采用记忆化的方法记录已经计算过的所有状态的最优解，就可以避免重复运算，消除冗余。这正是动态规划的思路。下面阐述具体思路。

显然，初始时 S_1，S_2 均为空串。产生解的过程可分为若干步，每步选择一个串插在 S_1 或 S_2 的末尾，那么每个中间状态可以描述成图 1-71 所示。

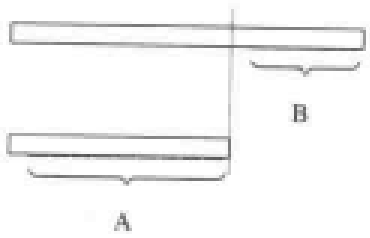


图 1-71　中间状态

其中 A 为已匹配的部分，B 为未匹配的部分，且 B 是某个数字串的后缀。这样可以定义状态 $d[i,j]$为：

当 B 部分为第 i 个数字串的后缀 word[i，j+1，…，length[i]]时，A 部分的长度最短。其中，word[i]表示第 i 个串，length[i]为第 I 个串的长度。

我们发现，其实后面的状态转移与 A 部分无关，仅与 B 部分有关。因此，可以用动态规划的方法来解决这道题，问题转化为求最短路。

初始时

$$d[i,j]=\begin{cases} j, & \text{如果存在word}[k]=\text{word}[i,1\cdots j] \\ \infty, & \text{其他} \end{cases}$$

从某个状态 $d[i,j]$出发进行转移，可以分为两种情况：

（1）存在某个串 word[k]是 word[i,j+1,…,length(s)]的前缀，则

$$d[i,j+\text{length}[k]]=\min\{d[i,j+\text{length}[k]],\ d[i,j]+\text{length}[k]\}$$

（2）存在某个串 word[k]，使得 word[i，j+1，…，length(s)]是 word[k]的前缀，则

$$d[k,\text{length}[i]-j]=\min\{d[k,\text{length}[i]-j],\ d[i,j]+\text{length}[i]-j\}$$

最后的解是所有 d[k,length[k]](k=1,…,n)中最小的一个。

需要特别指出的是上面 min 函数的定义。因为题目要求找出的串是长度最短且在同长度的串中字典序最小的一个，因此，若长度不等时，可以直接取小的那个；若长度相等，则要追溯前面的状态，取字典序较小的那个。这与一般的动态规划是不太相同的，需要特别注意。

虽然本题可以采用经典的求最短路算法来实现，但因为有可能要追溯前面的状态，为了编程方便，可以采用记忆化搜索的方式。另外，由于程序中需要大量用到查找某个字符串是否存在的操作，为了提高程序效率，可以用 Trie 的结构来存储。

区间模型　这一类题目的关键是用区间来表示状态，并建立合理的状态转移。这些题目往往比较巧妙，下面举几例说明。

【例题 7】青蛙的烦恼[1]

池塘里有 n 片荷叶（$1\leqslant n\leqslant 1\,000$），它们正好形成一个凸多边形。按照顺时针方向将这 n 片荷叶顺次编号为 1，2，…，n。

有一只小青蛙站在 1 号荷叶上，它想跳过每片荷叶一次且仅一次（它可以从所站的荷叶跳到另外任意一片荷叶上）。同时，它又希望跳过的总距离最短。

请你编程帮助小青蛙设计一条路线。

【分析】从直观上不难看出最短的路线中不存在相交的边。

如图 1-72 所示的路线（实线表示一步到达，虚线表示多步）可以做如图 1-73 所示的路线变换。

[1] 题目来源：经典问题

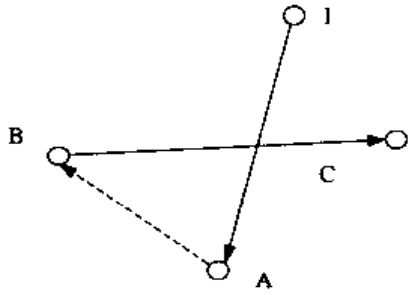


图 1-72　一种合法路线

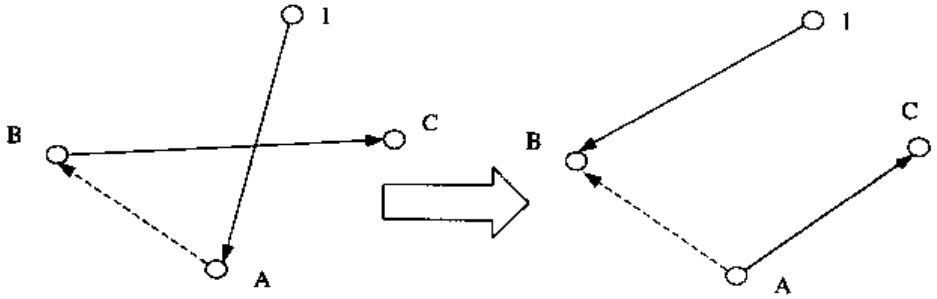


图 1-73　路线变换

变换后从 1 到 C 的距离显然比变换前的距离要短，故变换前的方案必然不是最优方案。根据这个结论可以知道：从 1 出发第一步只能到 2 或者 n。

同理，如果第一步到了 2，则第二步只能到 n 或者 3；如果第一步到了 n，则第二步只能到 n-1 或者 2…因为边不能相交！

接下来的任务就好办了。根据证明容易知道，从 1 出发，第一步只能到 2 或者 n。如果第一步到 2，则问题转化为"以 2 为起点，遍历{2…n}中的顶点一次且仅一次"；如果第一步到 n，则问题转化为"以 n 为起点，遍历{2…n}中的顶点一次且仅一次"。注意，又用到了递归！

设 $d[s,L,0]$表示从 s 出发，遍历{s…s+L-1}中的顶点一次且仅一次的最短距离；$d[s,L,1]$表示从 s+L-1 出发，遍历{s…s+L-1}中的顶点一次且仅一次的最短距离。则状态转移方程为：

$$\begin{cases} d[s,L,0]=\min\{\text{dis}(s,s+1)+d[s+1,L-1,0],\text{dis}(s,s+L-1)+d[s+1,L-1,1]\} \\ d[s,L,1]=\min\{\text{dis}(s+L-1,s+L-2)+d[s,L-1,1],\text{dis}(s+L-1,s)+d[s,L-1,0]\} \end{cases}$$

$d[s,1,0]=0$, $d[s,1,1]=0$

其中 dis(a，b)表示第 i 个顶点和第 j 个顶点之间的欧几里得距离。

状态有 $O(n^2)$个，决策数目和状态转移时间都是 $O(1)$，因此算法的时间复杂度是 $O(n^2)$。

WEB　本题有 $O(n\log n)$的算法，但是该算法需要用到比较多的结论，放在本书主页上。

【例题 8】排列问题[①]

在整数 1，2，…，N 的排列中，有些排列满足性质 A：该排列中除了最后一个整数以

[①] 题目来源：经典问题

外的每一个整数后面都跟有（不必直接紧跟）一个同它相差为 1 的整数。例如：$N=4$，排列 1432 是具有性质 A 的，而 2431 则不满足。

设有一个 N 个数的排列，已知其中 $P(P\leqslant N)$个位置上的数，求共有多少个这样的排列——在 P 个位置上具有已知的数，且具有上述性质 A。例如：$N=4$，且已知第 1 位、第 2 位分别是 1 和 4，则 1432，1423 就是这样的两个排列。

【分析】

通过枚举 N 比较小的时候满足题目的排列，发现一个规律：任何一个排列的后 k 位（$1\leqslant k\leqslant n$）是若干连续整数组成的集合。可以用数学归纳法证明这个结论（留给读者思考），进一步地，还可以证明只要满足任意后 k 位（$1\leqslant k\leqslant n$）是若干连续整数组成的集合，则这个排列一定符合题目要求。

有了这两个结论，很容易用递归的思想把原问题转化为子问题。设 $d[s, r]$表示满足下面条件的序列 C 的总数：C 由集合$[s\cdots s+r-1]$中的数组成，且后 k 位（$1\leqslant k\leqslant r$）是若干连续整数组成的集合。如果原问题中倒数第 i 个位置上的数已经确定为 x（$1\leqslant i\leqslant r$），那么 C 的倒数第 i 个位置上的数也要是 x。

可以用加法原理得到状态转移方程：

$$d[s,r]=\begin{cases} d[s+1,r-1], & \text{倒数第 } r \text{ 位必须为 } s \\ d[s,r-1], & \text{倒数第 } r \text{ 位必须为 } s+r-1 \\ d[s,r-1]+d[s+1,r-1], & \text{倒数第 } r \text{ 位不确定} \\ 0 & \text{其他情况，不能保证后 } r-1 \text{ 位为连续整数，故无解} \end{cases}$$

$d[s,1]=1$，目标是求 $d[1,n]$。状态有 $O(n^2)$个，决策数和状态转移数均为 $O(1)$，故算法的时间复杂度为 $O(n^2)$。

【例题 9】最优排序二叉树[①]

一个边长为 n 的正三角形可以划分成若干个小的边长为 1 的正三角形，称为单位三角形。如图 1-74，边长为 3 的正三角形分成三层共 9 个小的正三角形，把它们从顶到底，从左到右以 1~9 编号，如图 1-74(a)。同理，边长为 n 的正三角形可以划分成 n^2 个单位三角形。

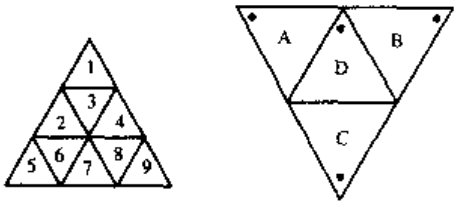


图 1-74　正三角形

四个这样的边长为 n 的正三角形可以组成一个三棱锥。将正三棱锥的三个侧面依顺时针次序(从顶向底视角)编号为 A, B, C，底面编号为 D。侧面的 A, B, C 号三角形以三棱锥的顶点为顶，底面的 D 号三角形以它与 A, B 三角形的交点为顶。图 1-74(b)为三棱锥展开后的平面图，每个面上标有圆点的是该面的顶，该图中侧面 A,B,C 分别向纸内方向折叠即可

① 题目来源：CTSC 2001. 命题人：杨帆

还原成三棱锥。我们把这 A、B、C、D 四个面各自划分成 n^2 个单位三角形。

对于任意两个单位三角形，如有一条边相邻，则称它们为相邻的单位三角形，显然，每个单位三角形有三个相邻的单位三角形。现在，把 $1\sim4n^2$ 分别随机填入四个面总共 $4n^2$ 个单位三角形中。

现在要求你编程求由单位三角形组成的最大排序二叉树。所谓最大排序二叉树，是指在所有由单位三角形组成的排序二叉树中结点最多的一棵树。对于任一单位三角形，可选它三个相邻的单位三角形中任意一个作为父结点，其余两个分别作为左孩子和右孩子。当然，做根结点的单位三角形不需要父结点，而左孩子和右孩子对于二叉树中的任意结点来说并不是都必须的。

例如，如果四面对应如图 1-75，则相应的最优二分检索树如图 1-76 所示。

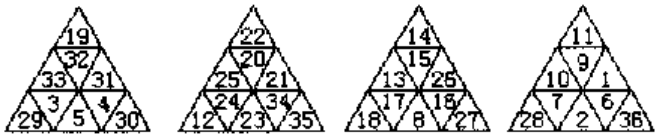


图 1-75　四面情况

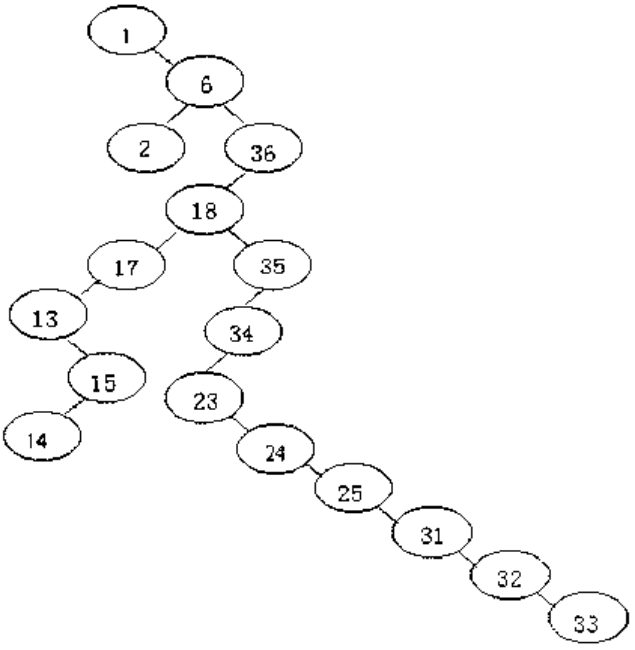


图 1-76　最优二叉排序树

【分析】

在讨论问题的解法之前，先来看看二叉排序树的性质。二叉排序树是一棵满足下列性质的二叉树：

（1）它或是一棵空树，或是一棵二叉树，满足左子树的所有结点的值都小于根结点的值，右子树的所有结点的值都大于根结点的值；

（2）它若有子树，则它的子树也是二叉排序树。

根据性质（1），可以知道，二叉排序树的左右子树是互不交叉的。也就是说，如果确定了根结点，那么就可以将余下的结点分成两个集合，其中一个集合的元素可能在左子树上，另一集合的元素可能在右子树上，而没有一个结点同时可以属于两个集合。这一条性质，满足了无后效性的要求，正因为二叉排序树的左右子树是互不交叉的，所以如果确定根结点后，求得的左子树对求右子树是毫无影响的。因此，如果要使排序树尽可能大，就必须满足左右子树各自都是最大的，即局部最优满足全局最优。

根据性质（2），二叉排序树的左右子树也是二叉排序树。而前面已经分析得到，左右子树也必须是极大的。所以，求子树的过程也是一个求极大二叉排序树的过程，是原问题的一个子问题。那么，求二叉排序树的过程就可以分成若干个阶段来执行，每个阶段就是求一棵极大的二叉排序子树。

由此，本题中二叉排序树满足阶段性（性质（2））和无后效性（性质（1）），可以用动态规划解决。

下面分析具体解决问题的方法。

首先，对给出的正三棱锥建图，建成一张普通的无向图。根据正三棱锥中结点的性质，每个结点均与三个结点相连。而根据二叉排序树的性质，当一个结点成为另一个结点的子结点后，它属于左子树还是右子树也就确定下来了。所以，可以对每个结点进行状态分离，分离出三种状态——该结点作为与它相连的三个结点的子结点时，所得的子树的状态。但是，一个子结点可以根据它的父结点知道的仅仅是该子树的一个界（左子树为上界，右子树为下界），还有一个界不确定，所以还需对分离出来的状态再进行状态分离，每个状态代表以一个值为界（上界或下界）时的子树状态。整个分离过程如图 1-77 所示。

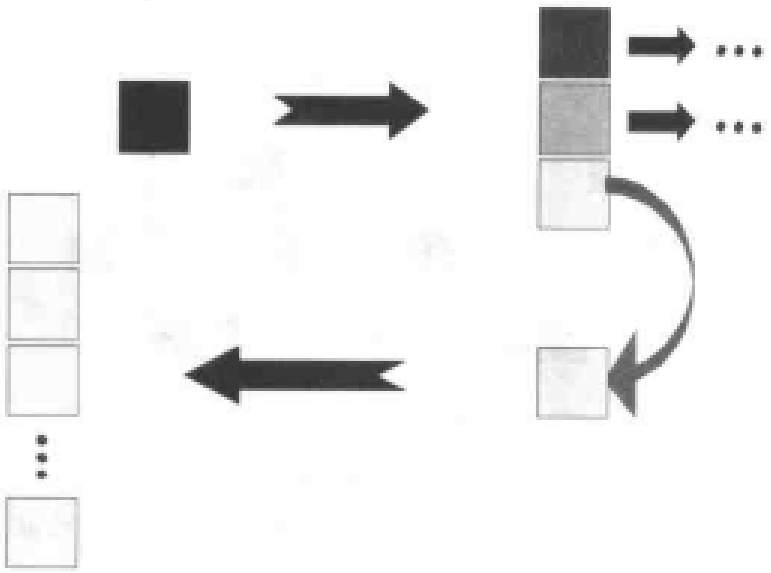


图 1-77　递推过程示意图

确定了状态后，要做的事就是推出状态转移方程。

前面已经提到，一个极大的二叉排序树，它的左右子树也必须是极大的。因此，如果确定以结点 n 为根结点，设所有可以作为它左子结点的集合为 N_1，所有可以作为它右子结点的集合为 N_2，则以 n 为根结点，结点大小在区间$[l, r]$上的最大二叉排序树的结点个数。

$$A(n,l,r)=\begin{cases}0 & l>r \text{或} n\notin[l,r]\\ 1 & l=r\\ \underset{n_1\in N_1}{\mathrm{Max}}\{A(n_1,l,n-1)\}+\underset{n_2\in N_2}{\mathrm{Max}}\{A(n_2,n+1,r)\} & n\in[l,r]\end{cases}$$

所要求的最大二叉排序树的结点个数为

$$\mathrm{MaxNodes}=\underset{1\le i\le n}{\mathrm{Max}}\{A(i,1,n)\}\quad \text{其中}n\text{为结点的总个数}$$

从转移方程来看，定义的状态是三维的，那么，时间复杂度理应为 $O(n^3)$。其实并非如此。每个结点的状态虽然包含下界和上界，但是不论是左子结点还是右子结点，它的一个界取决于它的父结点，也即是一个界可用它所属的父结点来表示，真正需要分离的只有一维状态，需要计算的也只有一维。因此，本题时间复杂度是 $O(n^2)$（更准确地说应该是 $O(3n^2)$）。

此外，由于本题呈现一个无向图结构，如果用递推形式来实现动态规划，不免带来很大麻烦。因为无向图的阶段性是很不明显的，尽管从树结构中分出了阶段。不过，实现动态规划的方式不仅仅是递推，还可以使用搜索形式——记忆化搜索。用记忆化搜索来实现本题的动态规划可以大大降低编程复杂度。

状态压缩模型　有时候，虽然无法得到多项式算法，但是由于充分利用了重叠子问题，仍然可以得到满意的算法。

【例题 10】Bugs 公司[①]

Bugs 公司生产一种 2×3 单位尺寸的高科技芯片。芯片被嵌入 $N\times M$（$N\le 150, M\le 10$）单位尺寸的模板内。模板接受过严格检查，损坏的单位小方格已被标上黑色记号，如图 1-78 所示。

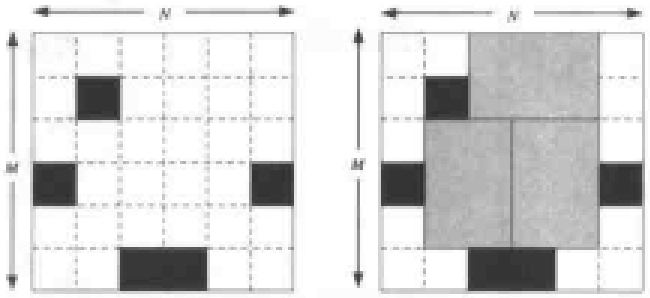


图 1-78　Bugs 公司的芯片

嵌入芯片的要求是，放置芯片的区域内不能有黑色记号，同时芯片与芯片不能重叠。公司希望将尽量多的芯片嵌入模板，所以请你求出可能的最大芯片数量。

【分析】

观察后发现，N 和 M 的范围相去甚远，同时总数据规模较大。这就给与提示，此题很可能是需要信息加密的动态规划，即把一行看成一个整体。

怎样看成一个整体呢？把底边贴着第 y 行底边放置的芯片称为第 y 行芯片。由于芯片

① 题目来源：CEOI 2002

长度只有3，所以第 y 行芯片的放置只受行 $y-1$ 和 $y-2$ 放置情况的影响。同时由于一旦方格（$x,y-1$）被黑色记号或其他芯片占据，方格（$x,y-2$）即便空闲对第 y 行芯片的放置也没有任何意义。所以，需要记录的信息只有：

（1）$a[x]=0$ 表示方格（$x,y-1$）空闲，方格（$x,y-2$）空闲；

（2）$a[x]=1$ 表示方格（$x,y-2$）被占据，方格（$x,y-1$）空闲；

（3）$a[x]=2$ 表示方格（$x,y-1$）被占据。

由于每一列有3种可能的状态，而每一行有 M 列，所以行 $y-1$ 和 $y-2$ 的放置情况可以用 M 位的3进制 $P_1P_2\cdots P_M$ 表示，称为放置序列。将该3进制数换算成10进制数 $P(10)$，并称 $P(10)$ 为 $P_1P_2\cdots P_M$ 的放置编号。

下面设计状态和状态转移方程。用 $d[i,j]$ 表示前 $1\cdots i$ 行放置编号为 j 时的最大芯片数量，先进行信息解压，即将10进制数 j 转换成 M 位3进制数 $P_1P_2\cdots P_M$，目的是方便状态转移。我们打算逐行递推，即是从前 $1\cdots i$ 行的放置推出前 $1\cdots i+1$ 行的放置情况。令

$$Q_i=\begin{cases}P_i-1(P_i>0)\\0(P_i=0)\end{cases}$$

将放置序列 Q 加密得放置编号 k。由于后移了一行，所以在不放置任何新芯片的情况下，$d[i+1,k]=d[i,j]$。

假设 x 满足 $Q_x\rightarrow Q_M=0$，那么考虑方格$(x,i+1)$的放置情况：

情况（1）方格（$x,i+1$）处不放置芯片，那么转而考虑方格$(x+1,i+1)$；

情况（2）如果 $P_x=P_{x+1}=0$ 并且区域（$x\rightarrow x+1$,y$-2\rightarrow$y）内均无黑色标记，那么在方格（$x,i+1$）处放置一张芯片，该芯片高度为3宽度为2且方格（$x,i+1$）是它的左下角。然后赋值 $k=k+2\times3^{x-1}+2\times3^{x}$，转而考虑方格$(x+2,i+1)$的放置情况；

情况（3）如果 $P_x,P_{x+1},P_{x+2}\leqslant1$ 并且区域（$x\rightarrow x+2$,$y-1\rightarrow y$）均无黑色标记，那么在方格（$x,i+1$）处放置一张芯片，该芯片高度为2宽度为3且方格（$x,i+1$）是它的左下角。然后赋值 $k=k+1\times3^{x-1}+1\times3^{x}+1\times3^{x+1}$，转而考虑方格$(x+3,i+1)$的放置情况。

该程序的时间复杂度为 $O(nm3^m)$。需要注意的是，由于 $d[i,j]$只受到 $d[i-1][]$和 $d[i-2][]$的影响，因此在任意时刻只需要记录三行，这种方法称为“滚动数组”。需要注意的是，递归加记忆化的方法无法使用滚动数组，因此状态的计算方式改变了。

读者可能已经看出了，本题还有优化余地。当 m 取上限10时，该算法中两列有 $3^{10}=59\,049$ 种情况，但是实际上没有这么多。假设原始数据中两列均为空，那么 y-1 列的黑格子不能是单独的，即如果$(x,y-1)$是黑色，则$(x-1,y-1)$和$(x+1,y-1)$必有一个是黑色的。读者可以顺着这个思路进行进一步的讨论。

【例题11】迷宫统计[①]

Jimmy 自己办了一个游园活动，其中一个项目是让游客去走一个随机生成的 m 行 $n(1\leqslant m\leqslant5,1\leqslant n\leqslant6)$列的迷宫，迷宫里有空地，也有障碍物（每个障碍物恰好占一个方格）。游客总是从左上角走到右下角，每次可以往东南西北四个方向之一行走。迷宫的生成方式很简单：每一个格子都有一个独立的概率 p，表示该格子为障碍物的概率。如果程序生成

① 题目来源：Elite Problemsetters' First Contest, Problem A. 命题人：Derek Kisman

了一个无解的迷宫，它会重新生成一个。你的任务是计算每个格子成为一个有解迷宫中的障碍物的概率。

【分析】

本题是一道难题，比赛时 380 只队伍中仅有两人做出此题。此题难就难在即使在 $m\leqslant 5$, $n\leqslant 6$ 的时候，有解迷宫也是很多的。$m=n=5$ 时有解迷宫有 1 225 194 个；$m=5$, $n=6$ 时高达 30 754 544 个。如果把所有有解迷宫都列举出来再计算概率，需要花费约 10 分钟的时间。我们的思路是：不列举所有有解迷宫，而是把这些迷宫分成若干个不相交的集合，在每个集合中分别计算概率。这样，得到了算法一。

算法一：根据特征通路的存在性分类

让 maze[x][y] = 0 表示(x,y)必须有障碍，maze[x][y]>1 表示(x,y)必须是空地，maze[x][y]=1 表示(x,y)可以是任意形式，则一个 maze 数组对应了一个迷宫（不一定是有解的）集合，把 maze 叫做该迷宫集合的模板。初始的时候，所有合法迷宫组成的集合的模板满足：起点和终点元素为 2（必须是空地），而所有其他元素均为 1（任意形式）。

对于任意一个模板 M，随便在这个模板中找一条左上角到右下角的通路 P（通路中的元素满足 maze[x][y]>0），则可以把这个模板所对应的集合划分成两个不相交的集合：用 A 来表示所有存在通路 P 的迷宫，B 来表示所有不存在通路 P 的迷宫，则把 P 中元素的 maze 值全部加 1（表示这些格子必须是空地），得到的模板即为 A 的模板。B 的模板比较复杂，还需要进一步把它分类。

既然不存在通路 P，显然 P 上至少应该有一个位置必须有障碍。用 $B(i)$代表通路 P 上的最后一个障碍位置为 i 的迷宫所对应的模板，则 $B(i)$应该是把位置 i 处设置为 0（障碍），位置 i 以后的设置为大于 1（空地，因为 i 是最后一个障碍），而位置 i 以前的设置为 1（无论前面的格子是什么，通路 P 一定不存在）。设 P 的长度为 k，则把模板 M 分解为了 k+1 个不相交的模板：A，$B(1)$，$B(2)$，…，$B(k)$。这样递归下去就可以了。模板的递归边界是所有一定含某通路 P 的 A 类集合，只需要处理每个这样的模板所对应的迷宫就可以了。

给出一个模板 M，设满足 maze[x][y]=0 的格子集合为 S_1,满足 maze[x][y]>1 的格子集合为 S_2，设满足 maze[x][y]=1 的格子集合为 S_3,那么模板 M 所对应的所有迷宫（这些迷宫都有解，因为包含一条通路）的概率和为：

Prob（M）=（S_1 每个元素为障碍的概率乘积）×（S_2 每个元素为空地的概率乘积）

如图 1-79 所示，左边的矩阵表示各格子为障碍的概率，右边表示模板。

0.0	0.1	0.2
0.3	0.4	0.5
0.6	0.7	0.8

空地(2)	空地(2)	空地(2)
未知(1)	未知(1)	空地(2)
障碍(0)	障碍(0)	空地(2)

图 1-79　模板和它的概率矩阵

则 S_1 每个元素为障碍的概率乘积为 0.6×0.7=0.42，S_2 每个元素为空地的概率乘积为 1.0×0.9×0.8×0.5×0.2=0.072，因此这个模板所对应的有解迷宫总概率为 0.072。

对于每个格子(x, y)，还需要统计在模板 M 中，它为障碍的迷宫总概率。显然当 maze[x,y]

=0 时，这个概率就是 Prob(M)；如果 maze[x,y]>1，这个概率为 0；如果 maze[x,y]=1，则这个概率为 Prob(M) ×（(x,y)为障碍物的概率）

这样，把所有 A 类模板对应的迷宫总概率加起来，得到所有有解迷宫的总概率 ProbAll，另外还统计了所有 A 类模板中，每个格子(x,y)为障碍时的概率总和 ProbBlock(x,y)，由条件概率公式，则所有答案 ans(x,y) = ProbBlock(x, y) / ProbAll。

程序的运行时间已经缩短到了 3 秒，但是仍然认为集合划分得太细致，导致计算量仍很大。有没有其他方法呢？尝试上题的思路一列一列地进行状态压缩。

算法二：基于状态压缩的动态规划

既然是一列一列递推，需要知道当前列（或者多列）需要记录哪些信息，即哪些信息对最后的答案有用。上题中，需要保存最后两行，但是本题保存一列、两列或者三列都可能不够。如果需要保存完，那么状态就太多，怎么办呢？一种想法是只记录当前列的每个格子是否能和起点连通。但是这样做是不对的，因此即使当前某个格子和起点不连通，以后也是可能连通的。这样做在状态转移的时候遇到了困难。正确的做法是另外记录当前列每两个格子是否连通，即记录在当前列 y 中，每个格子(x,y)的 last[x]值，其中 last[x]=0 表示它和起点连通，否则它表示在(x,y)正上方与(x,y)连通的格子中的最小行编号（如果没有这样的格子，则 last[x] = x）。这样，用(i, last[])表示一个状态，即前 i 列，第 i 列的 last 数组为 last[]的所有迷宫集合，并用 d[i, last]表示这些迷宫的总概率。为了统计和各个格子相关的概率，还需要增加一维 b，用 d[i, last, b]表示，前 i 列，第 i 列的 last 数组为 last[]，其中第 b 个格子为障碍的迷宫总概率（0≤b≤mn, j=0 表示总概率）。

这样，每次进行状态转移时，枚举当前列的所有(m+1)!(mn+1)种状态和 2^m 种决策（下一列的障碍情况），状态转移的时候需要做 BFS，但是由于只需要用上一列的 last 值和当前列，因此状态转移的复杂度为 2m+1，而且当 i 和 last 一定时，对于不同的 b 计算 d[i, last, b]只需要 BFS 一次，因此算法框架为：

```
for i := 1 to n do
  for k := 1 to (m+1)! do
  begin
    计算出第 k 个可能的的 last[]数组;
    for p:=1 to 2^m do
    begin
      计算出第 p 个可能的当前列
      花 2m+1 的时间做一次 BFS
      for b :=0 to m*n do
        递推 d[i, last, b]
    end;
  end;
```

总的计算量约为 $n(m+1)!2^m(mn+1+2m+1) = O(mn^2 2^m m!)$。显然和 m 有关的数比 n 大得多，因此总认为 $m \leq n$，否则把 m,n 交换，并把矩阵翻转。和上题一样，可以用滚动数组，这样

空间需求是 $O((m+1)!mn)$，这样的程序只需要不到 0.2 秒就得到了正确的结果。

树状模型　树本来就是一种递归结构，因此很多情况下都可以用动态规划来解决。下面举一个例子，它具有一定的通用性。

【例题 12】贪吃的九头龙[1]

传说中的九头龙是一种特别贪吃的动物。虽然名字叫“九头龙”，但这只是说它出生的时候有九个头，而在成长的过程中，它有时会长出很多的新头，头的总数会远大于九，当然也会有旧头因衰老而自己脱落。

有一天，有 M 个脑袋的九头龙看到一棵长有 N 个果子的果树，喜出望外，恨不得一口把它全部吃掉。可是必须照顾到每个头，因此它需要把 N 个果子分成 M 组，每组至少有一个果子，让每个头吃一组。

这 M 个脑袋中有一个最大，称为“大头”，是众头之首，它要吃掉恰好 K 个果子，而且 K 个果子中理所当然地应该包括惟一的一个最大的果子。果子由 N-1 根树枝连接起来，由于果树是一个整体，因此可以从任意一个果子出发沿着树枝“走到”任何一个其他的果子处。

对于每段树枝，如果它所连接的两个果子需要由不同的头来吃掉，那么两个头会共同把树枝弄断而把果子分开；如果这两个果子是由同一个头来吃掉，那么这个头会懒得把它弄断而直接把果子连同树枝一起吃掉。当然，吃树枝并不是很舒服的，因此每段树枝都有一个吃下去的“难受值”，而九头龙的难受值就是所有头吃掉的树枝的“难受值”之和。

九头龙希望它的“难受值”尽量小，你能帮它算算吗？

如图 1-80 所示的例子中，果树包含 8 个果子，7 段树枝，各段树枝的“难受值”标记在树枝的旁边。九头龙有两个脑袋，大头需要吃掉 4 个果子，其中必须包含最大的果子，即 N=8，M=2，K=4。

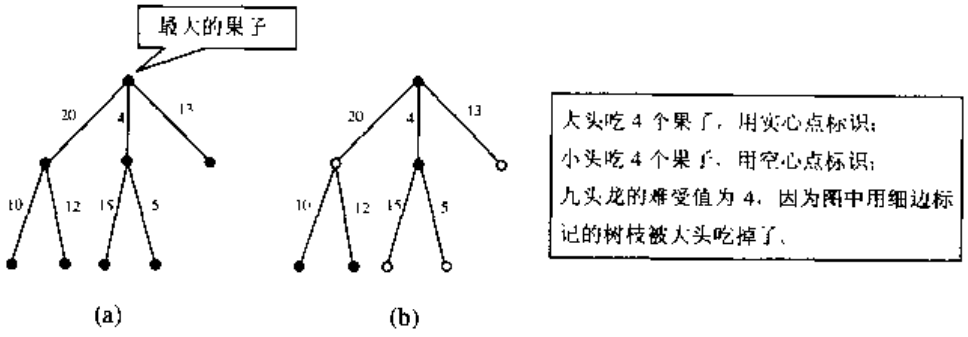


图 1-80　一棵果树和对应的最优方案

【分析】

首先，无解的情况很容易判断：如果果子不够吃（即 $N<K+M-1$），那么肯定无解；否则，至少存在一组解。为了方便叙述，以后只讨论有解的情况。

当 $M=2$ 时，小头只有一个。对于每一段树枝，如果两边的果子都是大头吃或都不是

[1] 题目来源：NOI2003. 命题人：毛子青. Internet Problem Solving Contest 2001. B 加强版

大头吃，那么这段树枝就要被吃掉；当 $M\geqslant 3$ 时，小头至少有两个。在确定大头吃的果子以后，把剩下的果子按照在整棵树上高度的奇偶性分成两类，让第一个小头吃高度为奇数的果子，第二个小头吃高度为偶数的果子，则连接这些果子的树枝都不需要吃掉。显然，这样的分配情况是最优的。

因此，可以得出结论：

$$\text{难受值}=\begin{cases}\text{两端的果子都被大头或小头吃的树枝难受值之和} & M=2\\ \text{两端的果子都被大头吃的树枝难受值之和} & M>2\end{cases}$$

这样，以后都不考虑小头，只考虑大头。用 $f(i,j,k)$表示以点 i 为根的子树中有 j 个果子分给大头的最小难受值，其中，$k=0$ 表示点 i 给小头吃，$k=1$ 表示点 i 给大头吃，则状态转移的时候，需要枚举 i 的每个子结点所在的子树分配几个果子给大头吃，以及这些子结点是否由大头吃，情况很复杂，应当怎样处理呢？

问题的复杂之处在于 i 可能有多个儿子需要考虑，但是由于所有儿子形成一个线性结构，可以在这个线性结构中再次使用动态规划（第二层动态规划）。或者，把这个二次动态规划的过程理解成把多叉树转化为二叉树也可以。

对于每一个树中的点，记录两个指针：兄弟 brother 和孩子 son。一个点的孩子指针指向它的第一个子结点，兄弟指针指向它的父结点的下一个子结点，这样将一棵多叉树转换成了二叉树。通过将多叉树转换为二叉树，可以用类似于二叉树的处理方法来解决多叉树的问题。

状态表示需要作相应的变化，$f(i,j,k)$表示以点 i 为根的子树和以它兄弟为根的子树与以它兄弟的兄弟为根的子树……中有 j 个果子分给大头的最小难受值，其中，$k=0$ 表示点 i 的父结点给小头吃，$k=1$ 表示点 i 的父结点给大头吃。这样，就把原来复杂的多叉树问题转为较为简单的二叉树问题。

这样，可以设计出动态规划的算法。为了叙述方便，不妨将原多叉树称为 T，转换后的二叉树称为 T'。设 $fa[i]$表示在 T 中 i 的父结点。

定义状态：$f(i,j,k)$表示在 T'中以点 i 为根的子树有 j 个果子分给大头吃的难受值，其中 $k=0$ 表示 $fa[i]$被大头吃，$k=1$ 表示 $fa[i]$被小头吃。

设 i 在 T' 中的两个子结点为 X_1,X_2,则状态转移方程是

$$f(i,j,k)=\min\begin{cases}f(X_1,j_1,1)+f(X_2,j-j_1-1,k)+d[k,1]\times\cos t[\text{i, fa[i]}] & \text{第}i\text{个点被大头吃}\\ f(X_1,j_1,0)+f(X_2,j-j_1,k)+d[k,0]\times\cos t[\text{i, fa[i]}] & \text{第}i\text{个点被大头吃}\end{cases}$$

d 定义为 $d[i,j]=\begin{cases}1 & (i=1)\text{且}(j=1)\\ 1 & (i=0)\text{且}(j=0)\\ 0 & \text{其他情况}\end{cases}$　且（$M=2$）边界：$f(0,0,k)=0;f(0,j,k)=+\infty\ j>0$

算法的理论时间复杂度上限为 $O(NK^2)$，空间复杂度为 $O(NK)$，状态量为 $2NK$，但是这个分析是很不精确的。记 $C(i)$为 T' 中以 i 为根的子树上的结点总数，则当 $j>C(i)$时，状态 $f(i,j,k)$就不存在，因此实际的合法状态和决策量少得多。定义 $g(i,j,k)$为状态 $f(i,j,k)$的决策量，对于非法状态令 $g(i,j,k)=0$，则当 i 只有一个儿子时 $g(i,j,k)=2(j\leqslant C(i))$，否则 $g(i,j,k)$不超过 j。因此对于给定的 i，所有 $g(i,j,k)$之和 sum(i)为：

（1）如果 i 是叶子，则 sum(i)=2

（2）如果 i 只有一个儿子，则 sum(i) = 2C(i)

（3）如果 i 有两个儿子，则 sum(i)≤1+2+…+C(i) = C(i)(C(i)+1)/2

这里不进行详细的推导，只举两个极端情况。

（1）链的情况：总决策量为 $2\times(1+2+\cdots+n) = n(n+1) = O(n^2)$

（1）完全二叉树的情况：总决策量为，$1\times n/2 + 3\times n/4 + 5\times n/8 + \cdots + (2n-1)/2^n = O(n)$

事实上，链还可以进一步优化，它可以直接算出来，即使是局部有链的情况，例如 i 只有一个儿子，也可以改为枚举它的第一个分叉后代的两个儿子的分配情况，而中间的链可以通过贪心得到。这样的时间复杂度又如何？读者不妨思考一下。

练　习　题

编程题：

1.5.10　快乐的蜜月[1]

位于某个旅游胜地的一家宾馆里，有一个房间是总统套房。由于总统套房价格昂贵，因此常常无人光临。宾馆的经理为了创收，决定将总统套房改建为专门为新婚夫妇服务的蜜月房。宾馆经理不仅大幅度降低了蜜月房的价位，而且还对不同身份的顾客制定了不同的价位，以吸引不同身份、不同消费水平的游客。比如对于来订蜜月房的国内来宾、海外旅客、港澳台同胞等，区别收取费用。

宾馆经理的举措获得了不同凡响的效果。由于蜜月房环境幽雅，服务周到，因此生意红火。宾馆经理在每年年底都会收到第二年的所有蜜月房预订单。第 i 中预订单包括以下几个必要的信息：到达日期 S_i、离去日期 T_i 和顾客身份 P_i，此种顾客收费 C_i。

由于宾馆只有一间蜜月房，同一时间只能接待一对新婚夫妇。因此并不是所有的预订要求都能得到满足。当一些预订要求在时间上发生了重叠的时候，就称这些预订要求发生了冲突。

对于那些不与任何其他预订要求发生冲突的预订单，必然会被接受，因为这对宾馆和顾客双方面来说都是件好事。而对于发生冲突的预订要求，宾馆经理则必须拒绝其中的一部分，以保证宾馆有秩序地运转。显然，对于同一时间内发生冲突的预定要求，宾馆经理最多只能接受其中的一个。经理也有可能拒绝同一时间段内的所有预定要求，因为这样可以避免顾客间发生争执。经理在做出决策后，需要将整个计划公布于众，以示公平。这是一个必须慎重的决定，因为它牵涉到诸多方面的因素。经理首先考虑的当然是利润问题他必然希望获得尽可能多的收入。可是宾馆在获得经济效益的同时，同时也应该兼顾到社会效益，不能太惟利是图，还必须照顾到顾客们的感情。如果宾馆经理单从最大获利角度出

[1] 题目来源：CTSC2000. 命题人：石润婷

发来决定接受或拒绝顾客的预订要求的话，就会引起人们的不满。经理有一个学过市场营销学的顾问，顾问告诉经理，可以采取一种折中的做法，放弃牟利最大的方案，而采纳获利第 k 大的方案。他还通过精确的市场分析，找到了 k 的最佳取值点($k \leqslant 100$)，告诉了宾馆经理。

现在请你帮助宾馆经理，从一大堆预订要求中，在上述原则下寻找到获利第 k 大的方案。宾馆经理将根据此方案来决定接受和拒绝哪些预订要求。

当然，可能有若干种方案的获利是一样大的。这时候，它们同属于获利第 i 大的方案而不区分看待。例如，假如有 3 种方案的收入同时为 3，有 2 种方案的收入为 2，则收入为 3 的方案都属于获利最大，收入为 2 的方案都属于获利第二大。依次类推。

假设所有的住、离店登记都在中午 12 点进行。顾客的身份共划分为 t 类，共有 r 个预订要求($r \leqslant 20\,000$)。

1.5.11　移动机器人[1]

在二维网格平面上有许多机器人在移动。每个机器人的状态由它所在的位置和面向的方位确定。每个机器人按照各自固定的指令执行移动，位置由一对整数(x, y)表示。机器人的方向有 4 个，用角度表示，分别是 0，90，180，270。命令有两种，转身和移动。转身命令有一个参数 D，是 90，180 或 270，机器人当前的方向改变 D 个度数，C 度将变为$(C + D) \bmod 360$。移动指令没有参数，机器人将按它的方向前进一个单位。0 方向的移动，位置改变（1，0），方向 90 改变（0，1），方向 180 改变（-1，0），方向 270 改变（0，-1）。

一个机器人依次完成它自己的指令序列，序列执行完后机器人将停在最终的位置上。

两个机器人之间的行动不互相影响，同一个位置可以有多个机器人。

在机器人开始移动前，可以去掉一些指令，所以控制中心可以改变机器人的行动路线和最终位置。控制中心希望使所有的机器人最后到达同一个位置以进行检查。同时希望能够在去掉最少的指令情况下完成这个目标。

共有 $R(2 \leqslant R \leqslant 10)$个机器人。每个机器人有它的初始位置，初始方向和命令序列。命令序列长度不超过 50。计算需要去掉的最少的命令数以及此时所有机器人的最终位置。如果有多个地点，输出任意一个即可。

1.5.12　佳佳的筷子[2]

中国人吃饭喜欢用筷子。佳佳与常人不同，他的一双筷子有三只，一双短筷子加上一根比较长的（一般用来穿香肠之类的食物）。两只较短的筷子的长度应该尽可能接近，但是最长的那根长度是无所谓的。如果一双筷子的长度分别是 A，B，$C(A \leqslant B \leqslant C)$，则用$(A-B)^2$的值表示这双筷子的质量，这个值越小，这双筷子的质量越高。

佳佳请朋友吃饭，并准备为每人准备一双这种特殊的筷子。佳佳有 N（$N \leqslant 5\,000$）只筷子，他希望找到一种办法搭配好 K 双筷子，使得这些筷子的质量值和最小。

---

[1] 题目来源：Baltic Olympiad in Informatics, 2002

[2] 题目来源：OIBH Reminiscence Programming Contest. 命题人：刘汝佳

*1.5.13　偷懒的工人[①]

人们都说 A 公司的工人很勤劳，因为只要一有可以做的工作，他们马上就会去做，而不会出现有任务可以完成，但他却闲着没事干的情况。虽然话说得没错（因为公司有这个规定），但是工人们实际上是很懒的，他们希望在符合公司规定的前提下让自己的工作时间尽量短。

假设某工人有 n 个任务要做，第 i 个任务恰好需要 t_i 单位的时间才能完成，而且必须在时间区间$[a_i,d_i]$被执行（即任务的开始时刻不小于 a_i，结束时刻不大于 d_i，$t \leqslant d_i-a_i<2t_i$）。假设该工作在同一时间只能进行同一个任务，而同一个任务要么不做，要么在规定的期限内不间断的做。他应该怎样选择任务，才能让自己的工作时间尽量少呢？

1.5.14　铁路调度[②]

火车是人类生产生活的重要交通工具。火车在铁路上行驶，铁路上的某些地方设有火车站。火车站内往往设有一些从主干线分叉出去的铁路支路，供火车停靠，以便上下客或卸载货物。铁路支路有一定的长度，火车也有一定的长度，且每列火车的长度相等。

假设某东西向的铁路上，有一小站。该站只有一条铁路支路可供火车停靠，并且该铁路支路最多能够容纳 M 辆火车（$M \leqslant 3$，是正整数）。为了火车行驶的通畅，该站只允许火车自东方进站，自西方出站，且先进站的火车必须先出站，否则，站内火车将发生堵塞。

该火车站工作任务繁忙。每天都有 N 辆（$N \leqslant 250$，是正整数）自东方驶向西方的火车要求在预定时刻进站，并在站内作一定时间的停靠。为了满足每辆进站火车的要求，小站的调度工作必须井井有条地开展。在小站每天的工作开始前，小站工作人员必须阅读所有火车的进站申请，并决定究竟接受那些火车的申请。而对不能满足要求的火车，小站必须提前通知它们，请它们改变行车路线，以免影响正常的铁路运输工作。由于火车进站、出站的用时可以忽略不计，小站允许几辆火车同时进站或出站，且小站工作人员可以任意安排这些火车进站的先后排列次序。小站的工作原则是尽量多地满足申请火车的要求。

你的任务是编程帮助工作人员考察某天所有火车的进站申请，并决定最多能满足多少火车的要求，以及这些火车的编号。

1.5.15　平板涂色[③]

CE 数码公司生产了一种名为自动涂色机（APM）的产品。它能用预定的颜色给一块布满不同尺寸互不覆盖的矩形平板涂色。

为了给平板涂色，APM 需要使用一组刷子。每个刷子涂一种不同的颜色 C。APM 拿起一把有颜色 C 的刷子，并给所有预定颜色为 C 且符合下面限制的矩形涂色。

为了避免颜料渗漏是颜色混合，一个矩形只能在所有紧靠在它上方的矩形涂色后，才能涂色。例如图 1-81 中，矩形 F 必须在矩形 C 和 D 涂色后才能涂色。注意，每一个矩形必须立刻涂满，不得只涂一部分。

你得写一个程序求一个使 APM 拿起刷子的次数最少的涂色方案。注意，如果一把刷子被拿起超过一次，则每一次都必须记入总数中。如上图 1-81，至少要把刷子拿起来 3 次，

---

① 题目来源：ACM/ICPC Regional Contest. Tacjon 2002
② 题目来源：IOI1999 中国国家集训队原创题目. 命题人：石润婷
③ 题目来源：ACM/ICPC Regional Contest Tehran 1999

平板的左上角坐标总是(0,0)，坐标范围是0…99，矩形不超过15个。

1.5.16　道路重建[①]

农夫John的农场有$n(n\leqslant 150)$个牛舍，有些牛舍之间有双向道路连接，而每两个牛舍之间有且仅有一条通路（因此可以表示成一棵树）。由于任意一场地震都很可能让John的农场变得不连通，因此John希望估计一下，至少需要毁坏多少条道路才会让一个恰好有$p(p\leqslant n)$个牛舍的子树脱离其他牛舍。例如图1-82中。

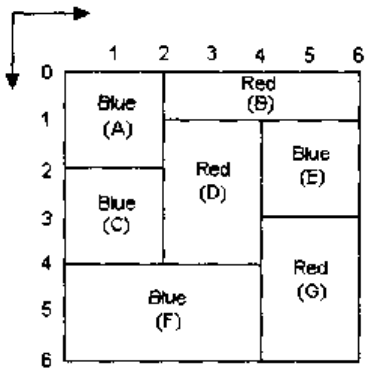


图1-81　平板举例

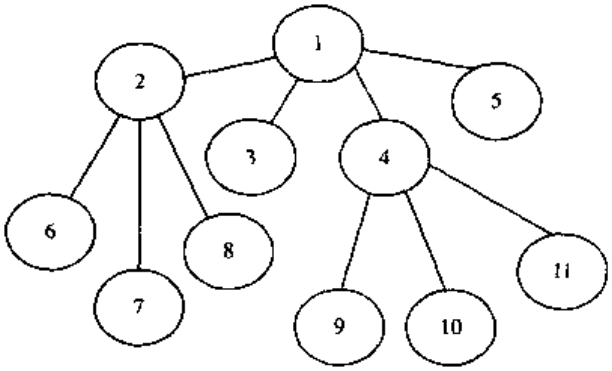


图1-82　John的农场示意图

要想让6个结点的子树孤立出来，只需要把1~4边和1~5边删掉就可以了。这样，1,2,3,6,7,8构成的子树和其他结点分离。

1.5.17　圆和多边形[②]

给你一个圆和圆周上的N个点。请选择其中的M个，按照在圆周上的顺序连成一个M边形，使得它的面积最大。

例如在图1-83的例子中，右上方的多边形最大。

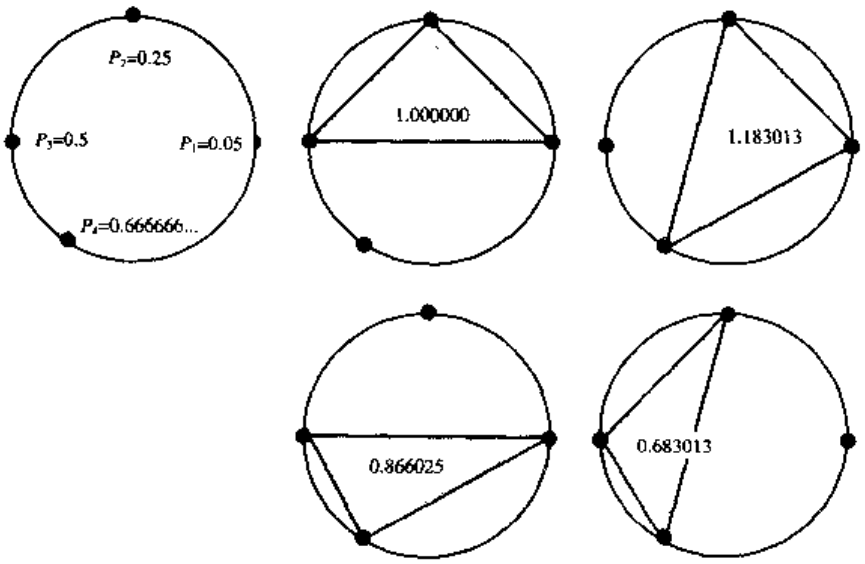


图1-83　圆和多边形

① 题目来源：USACO
② 题目来源：ACM/ICPC Regional Contest Tsukuba 2000

1.5.18　铁球落地[1]

有这样一个游戏，$n(n \leqslant 100\,000)$个平台的上空有一个铁球，如图 1-84 所示。球每次落到某平台上以后，游戏者可以选择它左滚还是右滚，球滚动和下落的速度都是 1。由于球的质量不大好，它每次的下落高度不能超过一个给定值 MAX。设计一种策略，使得球尽快落到地面而不被摔碎。假设地面高度为 0，且无限大。

图 1-84　铁球和平台

1.5.19　免费糖果[2]

桌上有 4 堆糖果，每堆有 N（$N \leqslant 4$）颗。佳佳有一个最多可以装 5 颗糖的小篮子。他每次选择一堆糖果，把最顶上的一颗拿到篮子里。如果篮子里有两颗颜色相同的糖果，佳佳就把它们从篮子里拿出来放到自己的口袋里。如果篮子满了而里面又没有相同颜色的糖果，游戏结束，口袋里的糖果就归他了。当然，如果佳佳足够聪明，他有可能把堆里的所有糖果都拿走。为了拿到尽量多的糖果，佳佳该怎么做呢？

1.5.20　丢三落四的老鼠[3]

Tom 猫送了小 Jerry 一份精彩的生日礼物，不过 Jerry 把礼物遗忘在了可能是小狗也可能是大狗的家中。现在 Jerry 打算从自己家出发将它取回。

卡通城由 N 个居住点和若干条连接居住点的双向街道组成，经过街道 x 需花费 T_x 分钟。可以保证，任两个居住点间恰有一条通路。Jerry 住在点 C，小狗、大狗分别住在点 A 和点 B。Tom 和 Jerry 都有卡通城的地图，但 Jerry 知道点 A、B、C 的具体位置，而 Tom 不知。

为了尽早拿回生日礼物，Jerry 会遵守以下两条规则：

（1）如果 A 距离 C 比 B 距离 C 近，那么 Jerry 先去小狗家寻找礼物，如果找不到，Jerry 再去大狗家；反之亦然。

（2）Jerry 总沿着两点间惟一的通路行走。

现在，Tom 希望你告诉它，最坏情况下 Jerry 需要耗费多长时间才能拿到生日礼物。

例如图 1-85，Jerry 住在点 2，小狗住在点 3，大狗住在点 4。从点 2 到点 3 需要 2 分钟时间，从点 2 到点 4 需要 3 分钟时间，所以 Jerry 会先去点 3 寻找生日礼物，一旦找不到，Jerry 再去点 4。这样，如果生日礼物很不幸地遗留在点 4，Jerry 需花费总共 1 + 1 + 1 + 2 = 5 分钟的时间。城市一共不超过 200 000 个居住点。

---

[1] 题目来源：CEOI 2000, Ball

[2] 题目来源：OIBH Online Programming Contest #1. 命题人：刘汝佳

[3] 题目来源：NOI2003. 命题人：林希德

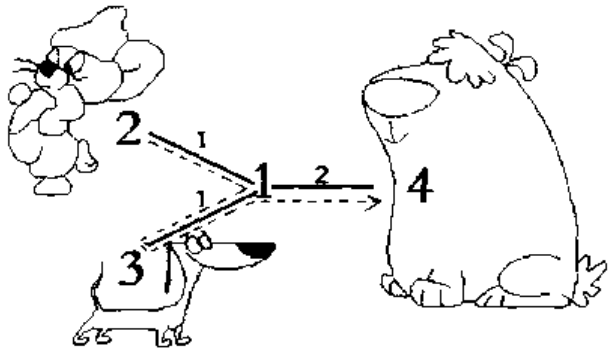


图 1-85　Jerry 所在的城市

1.5.3　若干经典问题和常见优化方法

本节通过对一些经典问题进行较为深入的分析，读者可以对动态规划有更深刻的认识。

方程的形式　多阶段决策问题利用递归的思想，把规模为 n 的问题转化为规模为 $n-1$ 的问题，直到转化为可以直接求解的原子问题。在一般情况下，这样的递归方法的时间复杂度是指数级别的，但是如果所有不同的子问题的数目是多项式级别，那么只需要多项式时间就可以解决这个问题,这就是动态规划的本质。动态规划算法有三个要素：①所有不同的子问题所组成的表（它包含的问题数目称为问题的大小（size））；②问题解决的**依赖关系（dependency）**可以看成是一个图；③填充子问题表的顺序（它实际上是②所得到了图的一个拓扑排序）。如果子问题数目为 $O(n^t)$，且每个子问题需要依赖于 $O(n^e)$个其他子问题，称这个问题为 tD/eD 的。这样，可以得到四类典型的动态规划方程。

方程一（1D/1D）：定义一个实函数 $w(i,j)(1\leqslant i<j\leqslant n)$，已知 $D[0]$，状态转移方程为

$$E[j]=\min_{0\leqslant i<j}\{D[i]+w(i,j)\},1\leqslant j\leqslant n$$

其中 $D[i]$可以根据 $E[i]$在常数时间里计算出来。

方程二（2D/0D）：已知 $D[i,0]$和 $D\{0,j](0\leqslant i,j\leqslant n)$，状态转移方程为：

$$E[i,j]=\min\{D[i-1,j]+x_i,D[i,j-1]+y_j,D[i-1,j-1]+z_{i,j}\}$$

其中 $x_i,y_j,z_{i,j}$ 都可以在常数时间里算出来。

方程三（2D/1D）：定义实函数 $w(i,j)(1\leqslant i<j\leqslant n)$，已知 $d[i,i]=0(1\leqslant i\leqslant n)$，状态转移方程为：

$$C[i,j]=w(i,j)+\min_{i<k\leqslant j}\{C[i,k-1]+C[k,j]\},1\leqslant i<j\leqslant n$$

方程四（2D/2D）：定义实函数 $w(i,j)(0\leqslant i<j\leqslant 2n)$，已知 $D[i,0]$和 $D[0,j](0\leqslant i,j\leqslant n)$，状态转移方程为

$$E[i,j]=\min_{\substack{0\leqslant i'<i\\0\leqslant j'<j}}\{D[i',j']+w(i'+j',i+j)\},1\leqslant i,j\leqslant n$$

其中 $D[i,j]$可以根据 $E[i,j]$在常数时间里算出来。

对于这四种典型的方程，如果方程是 *t*D/*e*D 的，可以立刻得到一个最简单的时间复杂度为 $O(n^{t+e})$，空间复杂度为 $O(n^t)$的算法。当然，在很多情况下可以把空间复杂度降低（只保存可以用在今后计算中的子问题结果），但是我们往往更关心的是时间复杂度。

下面，介绍几个常见的例子和对应的优化方法。

问题 1：最长公共子序列(LCS)

给两个长度分别为 n_1 和 n_2 的字符串 s_1,s_2，求二者的最长公共子串。这里，串 a 是串 b 的子串被定义为串 a 可以由串 b 删除零个或多个字符后得到（不允许改变字符顺序）。例如，“bde”是“abcde”的子串，但不是“beda”的子串。

设 $d[i,j]$为串 $s_1[i\cdots n_1]$和串 $s_2[j\cdots n_2]$的最长公共子串，如果 $s_1[i]=s_2[j]$，则状态 $d[i,j]$可以转移为 $d[i+1,j+1]$，在任何时候都可以转移为 $d[i,j+1]$，i 和 $d[i+1,j]$中较大的一个。请读者仔细思考：这几个状态转移各代表什么含义？还有其他状态转移方式没有考虑到吗？

这是个 2D/0D 问题（如果状态是 e 维的，决策是 t 维的，说问题是 eD/tD 的），因此算法的时间复杂度为 $O(n_1n_2)$。这个算法并不是最好的，没有利用到匹配点的特殊性（只要在匹配点才能够将指标函数加一）。这个特殊性叫做状态的稀疏性。

问题的稀疏性　这个方法主要应用在问题二、四中。有时候，问题是稀疏的（即，只需要计算一小部分子问题的解）。我们希望复杂度接近必须计算的问题集的规模（对于非常稀疏的问题，这个改进是很可观的）。对于这个方法，一般是首先通过预处理计算出必须计算出的子问题集 S，根据 S 的特点，已经有一些相应的优化算法。

处理稀疏问题的经典方法比较复杂，需要用到一些特殊的区域定义，并分成矩形和三角形两种情况加以讨论。本节只以稀疏的 LCS 问题为例，讨论这类问题的一般处理方法。在充分理解了这个方法以后，对于其他问题读者不难做出类比。

【例题 1】排列 LCS 问题[1]

给出 1…n 的两个排列 P_1 和 P_2，把它们看成字符串后求它们的 LCS。

【分析】

由于在刚才的分析中，如果 $P_1[i]\neq P_2[j]$，则 $d[i,j]$不能转移到 $d[i+1,j+1]$，因为目标函数不能增加。这样，我们应该只关心满足 $P_1[i]=P_2[j]$的状态 $d[i,j]$，也就是说，可以把状态转移方程写成：

$$d[i,j] = \max\{d[i',j']\}+1, \quad P_1[i]=P_2[j], \quad i'>i, j'>j$$

显然对于任意的 i，满足 $P_1[i]=P_2[j]$的 j 有且仅有一个，令 pair[i]为这个惟一的值，则一共只有 n 个需要考虑的状态，即 $d[i, \text{pair}(i)]$，那么边界条件是 $d[n, \text{pair}(n)]=1$。按照 $i=n-1,n-2,\cdots,1$ 的顺序来计算，则只要把目前计算出来的所有状态 $d[i',\text{pair}(i')]$（它们自然满足 i,>i）用合理的方式组织起来，可以很快地从其中找到满足 pair(i')>pair(i)的最大状态即可。

在 1.4 节介绍的二叉搜索树就可以满足要求：对于每个状态 $d[i, \text{pair}(i)]$，pair(i)是关键码，$d[i, \text{pair}(i)]$是附加信息，而另外还有一个信息 max，即以它为根的子树中状态的最大值。

[1] 题目来源：经典问题

请读者自己写出插入状态和取最大值的操作，算法的时间复杂度为 $O(n\log n)$。

问题 2：最长上升子序列（LIS）给一个序列，求它的一个递增子序列，使它的元素个数尽量多。

例如序列 1,6,2,5,4,7 的最长上升子序列是 1,2,5,7（还有其他的，这里略去）。

定义状态 $d[i]$是从第 1 个元素到第 i 个元素为止的最长子序列长度，则状态转移方程为：

$$d[i]=\min_{k<i \text{且} a[k]<a[i]}\{d[k]+1\}$$

这是个 1D/1D 问题，是有优化余地的（对于 eD/tD 问题，希望得到 $O(n^e)$算法而不仅仅是最直接的 $O(n^{e+t})$算法）。

对于两个数 x,y（假设 $a[x]\leqslant a[y]$），考虑状态 $d[x]$和 $d[y]$。显然，对于所有的 $a[i]>a[y]$都有 $a[i]>a[x]$，即能选 y 作为决策的状态 i 也能选 x 作为决策。如果此时还有 $d[x]\geqslant d[y]$，显然在任何时候 y 都不会是最优决策（x 始终比它优），故状态 $d[y]$是不需要保存的。这样，把所有需要保存的状态排成一列 $d[i_1]<d[i_2]<\cdots<d[i_k]$，则一定有 $a[i_1]<a[i_2]<\cdots<a[i_k]$。这样，满足 $a[i_j]<a[i]$的最大的 i_j 就是状态 i 的最优决策。注意，动机是只保存可能导致最优解的决策！

这样，需要维持一个有序数据结构保存所有需要的 d 值，需要能够支持：

- 查找操作，以便尽快在表中找到比 $a[i]$小的最大元素序号；
- 插入操作，以便插入新的需要保存的 d 和 a 值；
- 删除操作，以便删除不再有用的 d 和 a 值。

具体来说，先执行查找操作，找到最优决策 k，计算出 $d[i]=d[k]+1$，然后把 $d[i]$插入到合适的位置，维持表关于 a 有序。但此时 d 值不一定有序，因此有可能进行删除，具体方法如下。假设 $d[i]$被插到了第 m 个位置，即 $i_m=i$ 则有三种情况：

- $d[i_{m-1}]<d[i_m]<d[i_{m+1}]$，则 d 也有序，不需要调整；
- $d[i_{m+1}]<d[i_m]$，则 $d[i_m]$，以及在表中第 m 个位置右边的所有 d 值比 $d[i_m]$小的都删除；
- $d[i_m]<d[i_{m-1}]$，则 $d[i_m]$自己不需要保存，故把 $d[i_m]$本身删除。

和 2D/0D 问题的稀疏情形一样，二叉搜索树适合于这三种操作，三种操作的平均时间复杂度均为 $O(\log n)$。这样，LIS 问题在 $O(n\log n)$的时间复杂度内得到了解决。

问题 3：最优二分检索树

给出 N 个数据 $a_1\leqslant a_2\leqslant\cdots\leqslant a_n$和它们的权值 f_1，f_2，…，f_n，构造一棵二分检索树 T，使得每个数据 a_i的权值和深度 d_i的乘积之和（称为代价）最小。

二分检索是一棵二叉树，它要么为空，要么满足它的左子树的结点全部小于树根，右子树的结点（如果有的话）全部大于树根。关于二分检索树的更详细的介绍参见 1.6 节。根据定义，如果整棵树的根是 a_i，那么左子树的结点集合就是$\{a_1,a_2,\cdots,a_{i-1}\}$，而右子树的结点集合就是$\{a_{i+1}，a_{i+2}，\cdots，a_n\}$。

根据这个思想，定义状态 $d[i,j]$为把数据$\{a_i\cdots a_j\}$的元素组织成的二分检索树的最小代

价，则需要进行一个决策，即选择树根 $k(i\leqslant k\leqslant j)$，然后把数据$\{a_i\cdots a_{k-1}\}$和$\{a_{k+1}\cdots a_j\}$组织成最优二分检索树，作为 k 的左子树和右子树。

左子树和右子树的最优代价分别为 $d[i,k-1]$和 $d[k+1,j]$，设设集合 $\text{sum}[i,j]=f_i+f_{i+1}+\cdots+f_j$，则新的二分检索树的代价为 $d[i,k-1]+d[k+1,j]+\text{sum}[i,j]$。这样，我们的状态转移方程为：

$$d[i,j]=\min_{i\leqslant k<j}\{d[i,k-1]+d[k+1,j]\}+w[i,j]$$

其中 $w[i,j]=\text{sum}[i,j]$，时间复杂度为 $O(n^3)$，这是个 2D/1D 问题，是有优化余地的。

四边形不等式　决策单调性。

下面介绍四边形不等式和决策单调性，把时间复杂度降到 $O(n^2)$。如果一个函数 w 满足：

$$w[a,c]+w[b,d]\leqslant w[b,c]+w[a,d]\quad (a<b<c<d)$$

则说 w 满足凸四边形不等式（简称 w 为凸），如果函数 w 满足：

$$w[i,j]\leqslant w[i',j'] \quad ([i,j])\quad [i',j']$$

则说 w 关于区间包含关系单调。基于这两个定义，可以证明：

定理 1　如果 w 同时满足四边形不等式和区间单调关系，则 d 也满足四边形不等式。

定理 2　让 $d[i,j]$取最小值的 k 为 $K[i,j]$，则 $K[i,j-1]\leqslant K[i,j]\leqslant K[i+1,j]$。

定理 3　w 为凸当且仅当 $w[i,j]+w[i+1,j+1]\leqslant w[i+1,j]+w[i,j+1]$。

请读者自己证明这三个定理（提示：在定理一中，需要用数学归纳法，按照定义分情况讨论；在定理 2 中，只需要证明比 $K[i,j-1]$小的决策在 $d[i,j]$中仍然没有决策 $K[i,j-1]$优，这需要在 d 满足的四边形不等式的两边同时加上几项）。

这样，每次的决策范围变成了 $K[i+1,j]$到 $K[i,j-1]$。按照 $j-i$ 递增的顺序递推各个状态值，则对于每个确定的 $j-i$ 来说，决策总量为 $O(n)$，故总的时间复杂度为 $O(n^2)$。

定理 3 说明了验证 w 是否为凸的方法如下：固定 i 算出 $w(i,j+1)-w(i,j)$关于 i 的表达式，看它是关于 i 递增还是递减。如果是递减，则 w 为凸。同样地，也可以固定 j 算出 $w(i+1,j)-w(i,j)$关于 j 的表达式，如果它关于 j 递减，则 w 为凸。两种情况都需要是减函数，不习惯推导公式的读者可以直接记住这个结论。

问题 4：任务调度问题

有一个包含 $N(N\leqslant 10\,000)$个作业的任务序列需要在一个单处理机上执行。每个作业有它自己的运行时间 T_i 和代价因子 F_i。将序列划分成若干个块来执行，每个块包含若干个序号连续的作业。处理机按照作业序号从小到大依次执行各个块，在同一个块里，所有作业的完成时间 O_i 都一样，它们定义为该块开始执行的时刻与块里所有作业的运行时间之和。一个块开始执行之前，需要用长度为 S 的时间来进行调整工作。每个作业的代价 Cost_i 为它的完成时间与代价因子的乘积 O_iF_i。第一块任务的开始时间为 0。例如，$S=1$，有 5 个作业被分成三个块{1,2},{3},{4,5}，它的各个参数如表 1-18 所示(其中 T 和 F 为已知，O 和 Cost 是根据分块方法计算出来的)。

表 1-18　任务的属性

作业序号	1	2	3	4	5
T	1	3	4	2	1

续表

作业序号	1	2	3	4	5
F	3	2	3	3	4
O	5	5	10	14	14
Cost	15	10	30	42	56

整个任务的总代价为各个作业的代价之和。在这个例子中，总代价为 153。显然，不同的块划分方法得到的代价一般是不同的。请编程求出所有块划分方法所能达到的最小代价。

如果令 $d[i]$为前 i 个任务的最小代价，那么在状态转移的时候，还需要记录前 i 个任务的完成时间（或分成的块数），状态必须增加一维。可以把状态的含义改变一下。假设第 i 个任务的完成时间为 T，显然后面的所有任务的时间都在 T 基础之上增加的。不妨把这部分代价，即 $T(F_{i+1}+\cdots+F_n)$也算在状态 $d[i]$里，这样在状态转移时就可以每次都看作从时间 0 开始工作了。这样，状态转移方程为：

$$d[j] = \min\{d[i] + w(i,j)\},\quad i=1,2,3,\cdots,j-1$$

其中 $w(i,j) = (S + \text{sum}T(i+1,j))\text{sum}F(i+1,n)$，这是 1D/1D 问题的标准形式，其中 $E[i]=D[i]$。和刚才的情况一样，下面我们证明 $w(i,j)$满足凸四边形不等式。

令 $ST(i)=T_1+\cdots+T_i$，$SF(i)=F_1+\cdots+F_i$，令 $x=SF(n)$，则 $w(i,j)$可以写成

$$w(i,j) = (S + ST(j) - ST(i))(SF(n) - SF(i))$$

计算得 $w(i, j+1) - w(i, j) = b(i)T_j$，其中 $b(i) = SF(n) - SF(i)$是递减的，由定理 3 可知，$w(i, j)$是凸的。证明了 $w(i,j)$是凸的又怎么样呢？有什么好的方法可以处理它吗？

凸完全单调性　和四边形不等式密切相关的，定义矩阵凸完全单调性。如果 A 是一个 $n\times m$ 矩阵，则它是凸完全单调的(convex totally monotone)当且仅当对于所有的 $a<b, c<d$，

$$A[a,c]\geqslant A[b,c]\rightarrow A[a,d]\geqslant A[b,d]$$

也就是说，如果元素 x 在 y 的正上方且 $x>y$，则与 x 同行并在 x 右边的所有元素 z 都比与 z 同列、与 y 同行的元素大。类似地，A 是凹完全单调的（concave totally monotone）当且仅当对于所有的 $a<b, c<d$，

$$A[a,c]\leqslant A[b,c]\rightarrow A[a,d]\leqslant A[b,d]$$

需要说明的是，四边形不等式可以推出完全单调性，但反之不真。下面的算法只用完全单调性，虽然很多时候 w 满足四边形不等式。

最小值行编号的性质　如果令 r_j 为矩阵 A 第 j 列的最小值所在的行（为了简单起见，假设所有元素都不相同），则凸完全单调性可以推出 $r_1\leqslant r_2\leqslant\cdots\leqslant r_m$（即列的最小元素行编号非降）。类似地，凹完全单调可以推出 $r_1\geqslant r_2\geqslant\cdots\geqslant r_m$。

动态规划的目的和优化思路　令 $B[i,j] = d[i] + w(i,j)$，则可以证明：如果 $w(i, j)$是凸的，则 $B[i, j]$也是凸的。动态规划中的决策就是行编号，而目的可以表示为求出 B 每列的最小值。一般的动态规划实际上是从左到右填充 B 的每一列，填充完一列后再一一比较，求出列最小值。下面的方法实际上是在说不需要一一比较，甚至不需要计算完整的列，而只需要计算需要计算的行。事实上，后面将看到每列的最小值只可能出现在两行中！以后将不

用“状态”和“决策”两个概念，而使用矩阵中的行列等概念，请读者注意。

w 为凸的情形　设 best[j]为第 j 列的最小值行编号，则根据最小行编号性质，best[1] ≤ best[2] ≤…≤best[n]。把 best 相同的值合并，则得到一些区间[h_{i-1}+1, h_i]，表示第 i 行是列区间[h_{i-1}+1,h_i]的共同最小值行，因此 h_i 为满足 best[j]=i 的最大列编号 j。注意到方程中规定 $i<j$，所以 B 矩阵只有主对角线以上的元素才是有效的。则，显然有 best[1]=1。为了计算 best[2]，把 B[1,2]和 B[2,2]进行比较（第 2 列只有这两个元素有效）：

- 如果 B[1,2]大，那么第一行的区间就可以关闭了，因为根据最小行编号性质，第一行以后再也不可能成为最小值行了。
- 如果 B[2,2]大，那么第一行的区间应当继续。下次还需要比较 B[1,3]和 B[2,3]、B[1,4]和 B[2,4]吗？当然不是。根据凸完全单调性，一旦某个发现 $B[1,j_0]>B[2,j_0]$（前者在后者的正下方），对于任何 $j>j_0$，都有 $B[1,j] > B[2,j]$，因此从列 j_0 开始，第一行也永远不可能成为最小值行了。如果能一开始就找到这个 j_0，就不需要每次进行比较了。

由于 $B[i,j] = d[i] + w(i,j)$，因此 $f(i_1,i_2,j)=B[i_1,j] - B[i_2,j] = d[i_1]-d[i_2]+w(i_1,j)-w(i_2,j)$。如果算出了 $f(i_1,i_2,j)$的零点 zero(i_1,i_2)，就能够找到第一个让 $B[i_1,j_0]>B[i_2,j_0]$的 j_0。对于一些特殊函数，零点可以在 O(1)时间内求出，但是对于一般的函数，需要 $O(\log n)$的时间进行二分查找。

这样，可以把第一行的区间终点记录下来，只在这个范围之内不考虑第二行。那么第三行呢？如果在该区间之外，那么第一行不用考虑；在区间之内，第二行不用考虑，无论哪种情况都只需要把第三行和其中一行做比较。为了及时地把第一行“封住”并给第二行“解封”，需要把所有区间结束位置算出来地行都保存起来，并按照列递增的顺序访问。由于凸完全单调性保证了最小行编号递增，因此区间结束位置递增，故应该用队列保存这些行和它们的结束位置。

设队列为 $i_1<i_2<\cdots<i_k$(i_1 为队首)，则开始位置 $h_1\leqslant h_2\leqslant\cdots\leqslant h_k$。那么，如果新的行更优，应该清空整个队列，否则当求出了第一个使得当前行比 i_1 行更优的列 h 时，从尾到首扫描队列，把开始位置 h 查入到合适的位置，保证仍然有 $h_1\leqslant h_2\leqslant\cdots\leqslant h_k$，并把它后面的行都删除（想一想，为什么）。虽然一次这样的操作有可能需要扫描整个队列，最坏情况下是 $O(n)$的，但是由于一次比较伴随一次删除（或者插入结束），而删除一共进行了 $O(n)$次，因此比较也最多 $O(n)$次，维护队列的总时间复杂度为 $O(n)$。但是由于每次用二分找零点，因此总时间复杂度为 $O(n\log n)$。当然也可以用二分查找来进行插入，但是这样做就不容易分析复杂度了，最坏情况不一定好。

WEB 对于 w 函数比较复杂的情况，需要用二分查找求零点，则时间复杂度为 $O(n\log n)$
实际上有对任意 w 函数适用的 $O(n)$算法，请阅读本书主页

下面，换一个角度来处理。把 $d[i]$解释成只考虑从任务 i 开始的任务时，最少总代价，则状态转移方程写为：

$$d[j] = \min\{d[i] + w(i,j)\},\quad i=j+1,\ \cdots,\ n$$

其中 $w(i,j) = (S + ST(i) - ST(j))(SF(n) - SF(j))$

对于 $i<k<l$，有：在计算列 i 时，k 比 l 优，$B[k,i]\leqslant B[l,i] \Leftrightarrow d[k]+w(k,i)\leqslant d[l]+w(l,i) \Leftrightarrow d[l]-d[k]+$

$w(l,i)-w(k,i) \geqslant 0 \Leftrightarrow d[l]-d[k]+(T_k+T_{k+1}+\cdots+T_{l-1})\times(F_i+F_{i+1}+\cdots+F_n) \geqslant 0$

$\Leftrightarrow (d[k]-d[1]) / (T_k+T_{k+1}+\cdots+T_{l-1}) \leqslant (F_i+F_{i+1}+\cdots+F_n)$

如果定义函数 $g[k,l]= (d[k]-d[1]) / (T_k+T_{k+1}+\cdots+T_{l-1})$，$f[i]= (F_i+F_{i+1}+\cdots+F_n)$，则

结论 1　$B[k,i]\leqslant B[l,i]$ 当且仅当 $g[k,l] \leqslant f[i]1\leqslant i<k<l$

进一步地，假设有 $g[j,k] \leqslant g[k,l]$，那么 $g[j,k]$和 $f[i]$的大小关系只有两种情况：

（1）$g[j,k]\leqslant f[i]$，这时，由结论 1 立即得到 $B[j,i]\leqslant B[k,i]$，即在第 i 列时，行 j 比行 k 好；

（2）$g[i,k]>f[i]$，由不等式传递性得 $g[k,l]>f[i]$，由结论 1 得行 1 比行 k 好。

也就是说，无论哪种情况，k 都不是最优决策！

结论 2　满足 $g[j,k]\leqslant g[k,l]$ $(1\leqslant j<k<l\leqslant n)$的 k 不会是第 i 列$(i<j)$的最优决策。

根据这两条结论，保留从 $d[n]$开始计算到 $d[1]$的顺序，维持一个“对以后 d 值计算有帮助（可能成为以后某个 $d[k]$的最优决策）的决策集合”，期待得到更好的时间复杂度。为此，先把这些决策集合排序成

$$i_r<i_{r-1}<\cdots<i_2<i_1, \qquad ①$$

并且有

$$g[i_r, i_{r-1}] > g[i_{r-1}, i_{r-2}] > g[i_{r-2}, i_{r-3}] >\cdots> g[i_3, i_2] > g[i_2, i_1] \qquad ②$$

读者可能会问，为什么它们需要满足式②呢？原因很简单。不妨假设有一个 i_p 满足 $g[i_{p+1}, i_p] \leqslant g[i_p, i_{p-1}]$，由结论 2 可以知道，$i_p$ 对以后的 d 值计算没有影响，根本就不需要保存它！

这样，可以设计如下算法。

首先，决策集合初始化为空（$d[n+1]=0$，任何数都不可能是最优决策），并计算出所有 $f[i]$。然后，从 n 到 1 递减的计算每个 $d[i]$的值。计算的时候分两步。

（1）由于保证了式②，只要把 $f[i]$插入到不等式②的合适位置，即

$$g[i_r, i_{r-1}] > g[i_{r-1}, i_{r-2}] > \cdots > g[i_{t+1}, i_t]>f[i]>\cdots>g[i_3, i_2] > g[i_2, i_1]$$

那么根据结论 1 可以得到：最优策略就是 i_t，不仅如此，它右边的决策 i_{t-1}，…，i_1 在以后也不会用到（注意结论 1 中 i 的范围），应当被删除。

（2）求出 $d[i]$以后，i 有可能被以后的计算用到，因此需要把 i 插入到决策集合中。为了保持式②，从左向右扫描决策集合，找到第一个满足 $g[i, i_v] > g[i_v, i_{v-1}]$的位置插入，而把该位置左边的元素全部删除（根据结论 2，它们不可能被用到）。

这样，维持决策表的复杂度为 $O(n)$（元素共 n 个，每个元素被插入和删除最多一次），而每次状态转移的复杂度为 $O(1)$，因此总的时间复杂度为 $O(n)$。

把刚才的方法总结一下，关键在于把第 i 列的第 k 行比第 l 行优的条件写成了 $g[k,l]\leqslant f(i)$。只要写成了这种形式，剩下的过程就和刚才一样了。

w 为凹的情形　有两个地方不一样，一是需要用一个栈而不是队列来保存候选行（想一想，为什么），二是在把新行插入栈时，不能把后面的所有元素删除。其他情况和凸情形很类似，这里不再叙述。对于一般 w 函数最好的方法是 $O(n\alpha(n))$的。

【例题 2】 *序列分割*[①]

有一个长度为 $n(n\leqslant 1\ 000\ 000)$的整数序列 A（不一定为正整数），可以把它分成若干

[①] 题目来源：Balkan Olympiad in Informatics, 2003. Euro. Modified

个片段，假设第 i 个片段是从第 x_i 个元素到第 y_i 个元素，这些元素之和为 s_i，则该片段的收益为 $s_i y_i - T$，其中 T 是正常数。设计一种序列划分方法使得总收益尽量大。

【分析】

用最简单的分析方法可以得到一个 1D/1D 方程，存在一个 $O(n^2)$的算法，但是由于每个数可以是负数，因此很难像上题一样优化。考虑到当 s_i 为正时应该不忙着做分割（不分割的话不但少消耗 T 的代价，而且 s_i 的系数增大），因此有如下猜想：

猜想①：令 $S_i=A_1+A_2+\cdots+A_i$，j 是满足 $S_j<0$ 的最小正整数（如果不存在，令 $j=n$），则对于任何 $i<j$，从 $A[i]$后分割一定不能导致最优解。

只要用反证法，就会发现分割撤消以后一定能得到更优解（请读者详细推导），因此猜想成立。顺着这个思路，做一个更大胆地猜想。

猜想②：把刚才得到的 j 称作第一关键点，把 j 删除后剩下的序列同样求出它的一个点 j'（这时实际上有 $S_i=A_{j+1}+A_{j+2}+\cdots+A_j$），把它称作第二关键点。则从第一关键点、第二关键点之间的任何一点进行分割后一定不能导致最优解。

这个证明困难一些，还是用反证法，把两个关键点之间的第一个分割点移动到第后一个关键点，发现得到的解更优。

这样，可以不断重复这一过程，把第一关键点前的数压缩成一个数 B_1，然后求出 B_2，B_3，…，最后得到一个压缩序列$\{B_k\}$。表 1-19 说明了这个过程。

表 1-19　压缩过程

原数列	4	2	−3	1	−7	1	2	−4	6
连加和	4	6	3	4	−3	1	2	−1	6
备　注					$B_1=-3$			$B_2=-1$	$B_3=6$

这样，每次发现连加和小于 0，就把得到的和加入序列$\{B_i\}$，并把计数器清零，便可在 $O(n)$的时间内进行序列压缩。这样，得到了一个好得多的序列，最多只有最后一个元素非负。设这个压缩序列有 m 项，则直接作用在它上面的动态规划时间复杂度为 $O(nm)$。设 $E[i]$ 为 B_i 在原数列中所对应的最后一个下标，$d[i]$为前 i 个数进行分割所获得的最大收益，则状态转移方程为：

$$d[j] = \max\{d[i] + \text{sum}(i+1,j) \times E[j]\} - T$$

为了叙述方便，把压缩序列中所有元素以及 T 取负，则答案仅该变符号，状态转移方程为：

$$d[j] = \min\{d[i] + \text{sum}(i+1,j) \times E[j]\} + T$$

这个方程跟刚才的式子已经很像了，还是把它写成：

$$d[j] = \min\{d[i] + w(i,j)\}，\text{其中 } w(i,j) = \text{sum}(i+1,j) \times E[j] + T$$

可以用完全类似刚才那题的方法来做，这里就不叙述了。

WEB　本题还有 $O(mT^{1/2})$和 $O(n\log^2 n)$两种做法，都很具有借鉴意义。有兴趣的读者可以访问本书主页。

练 习 题

**1.5.21 多排列的LCS[1]

给出1，…，n的m个排列，求所有排列的LCS。

1.5.22 回文词[2]

给出一个长度不超过5 000的串S。给S添加尽量少的字母，使它变成一个回文词，即首尾对称的词。例如cbabd可以添加两个字符变成dcbabcd，它是首尾对称的。

1.5.23 友好城市[3]

Palmia国有一条横贯东西的大河，河有笔直的南北两岸，岸上各有位置各不相同的$N(N\leqslant 5\,000)$个城市。北岸的每个城市恰好有一个友好城市在南岸，而且不同城市的友好城市不相同。

每对友好城市都向政府申请在河上开辟一条直线航道连接两个城市，但是由于河上雾太大，政府决定避免任意两条航道交叉（如果航道交叉，船只很可能在雾中相撞）。编程帮助政府做出一些批准和拒绝申请的决定，使得在保证任意两条航线不相交的情况下，被批准的申请尽量多。

1.5.24 邮局[4]

按照递增顺序给出一条直线上坐标互不相同的$n(n\leqslant 10\,000)$个村庄，要求从中选择$p(p\leqslant 1\,000)$个村庄建立邮局，每个村庄使用离它最近的那个邮局，使得所有村庄到各自所使用的邮局的距离总和最小。试编程计算最小距离和，以及邮局建立方案。

提示：本题可以用四边形不等式优化。

1.5.25 基因串[5]

基因串是由一串有限长度的基因所组成的，其中每一个基因都可以用26个英文大写字母中的一个来表示，不同的字母表示不同的基因类型。一个单独的基因可以生长成为一对新的基因，而可能成长的规则是通过一个有限的成长规则集所决定的。每一个成长的规则可以用三个大写英文字母$A_1A_2A_3$来描述，这个规则的意思是基因A_1可以成长为一对基因A_2A_3。

用大写字母S来表示一类被称作超级基因的基因，因为每一个基因串都是由一串超级基因根据给出的规则所成长出来的。

请写一个程序，读入有限条成长的规则和一些我们想要得到的基因串，然后对于每个基因串，判断它是否可以由一个有限长度的超级基因串成长得出。如果可以，给出可成长为该基因串的最短超级基因串的长度。

---

[1] 题目来源：经典问题
[2] 题目来源：IOI2000. Palindrome
[3] 题目来源：CEOI 1996
[4] 题目来源：IOI2000
[5] 题目来源：Polish Olympiad in Informatics

***1.5.26**　奶牛转圈[1]

奶牛蹬车队由 N（$1 \leqslant N \leqslant 20$）名队员组成。它们想确定一种比赛策略使得一名队员最早穿过终点。为了抵挡疾风，奶牛们成群地骑车。当以每分钟 x（x 是整数）圈的速度骑车时，领头的奶牛以每分钟 $x \times x$ 能的速度消耗体力，同时其他奶牛以每分钟 x 能的速度消耗体力。当处于某整数分钟的时刻时，领头奶牛退场。

奶牛一共需要跑 D（$1 \leqslant D \leqslant 100$）圈。每头牛都有相同的初始整数体力值 E（$1 \leqslant E \leqslant 100$）。要求给出最早到达终点的时间，它是个整数，因为在某分钟缺一点的时刻到达和刚好到达这里认为是一样的。

*****1.5.27**　元件折叠[2]

线性排列的元件 C_1，C_2，…，C_n。C_i 宽 w_i，高 h_i。要折叠成宽度为 W 的若干行（即每行元件总宽度$\leqslant W$），每行高度为该行中最高元件高度。行 i 与 j 行之间为布线通道，若 C_i 与 C_{i+1} 之折叠，则它们所在行之间布线高度为 l_i。规定 $l_n=0$，如图 1-86 所示。求最小总高度。

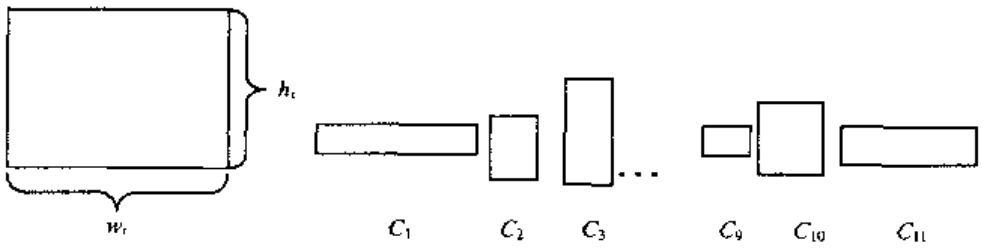


图 1-86　元件和排列

提示：合理组织有用状态，用堆把时间复杂度从 $O(n^2)$降到 $O(n\log n)$。

1.5.28　DNA 序列[3]

一些生物的复杂结构可以用其 DNA 序列来表示，而 DNA 序列又可表示为带有标记的核苷酸字符串。最近，福州大学信息学院生物信息学研究小组的一项工作表明，大多数蛋白质序列可以从核苷酸数据库记录中推导出来。研究小组的科学家们用密码学方法将 DNA 序列变换为 2 维 Cray 码后发现，任一 DNA 的 Cray 码序列 S 具有如下性质。

（1）列 S: $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$是一个有限序列；

（2）列 S 中各项（x_i，y_i）均落在一个网格边长为 1 的 300×3000 平面网格 N 的网格点上；

（3）序列 S 中各项最多分布在网格 N 的 3 行中。

科学家们发现，若将序列 S 的每一项都看作平面上一个点，从序列的任意一点出发，连接序列中每个点恰好一次，又回到出发点的最短平面回路 T 的长度与该序列表示的 DNA 密码密切相关。为了有效地破译 DNA 密码，科学家们希望设计出一个高效的计算程序，能对给定 DNA 的 Cray 码序列 S，快速计算出其最短回路 T 的长度。

提示：先得到 $O(n^3)$算法，再用四边形不等式优化到 $O(n^2)$。

---

[1] 题目来源：USACO
[2] 题目来源：UVA Problem Archive Online Contest
[3] 题目来源：福建省选拔赛 2000

1.6　状态空间搜索

搜索被称为“通用解题法”，在算法中占有重要的地位。但是由于它巨大的局限性和自身灵活性，也被认为是最难学难用的算法之一。这一节是本章最难懂的内容，因为搜索法是大家熟知的，但是有很多搜索思想并不是显而易见的。

为了不给大家造成思维局限，本节不给出各个搜索算法的实现细节和框架，因此强烈建议大家先阅读有关的人工智能搜索的书籍如或者是其他信息学竞赛理论书籍，这样在阅读本节的时候，会对这些算法有更深的体会。

在本节中，首先介绍状态空间的概念，然后盲目搜索算法。这些方法包括：深度优先搜索，广度优先搜索和双向广度优先搜索。本节接着介绍了启发式搜索，重点在 A*算法和 IDA*算法。再接着，将学习敌对搜索中的极大极小过程和 Alpha-Beta 剪枝。本节最后考虑了搜索中的剪枝优化的常见手段和一点理论分析。

学习本节之后，希望读者对于任意一个问题，能很快建立状态空间，提出一个合理的搜索算法并简单地估计它的时空性能。能比较熟练地实现本节中的大部分题目的程序（这步比较困难，但是要想把基本功打扎实，这是必须的）。状态空间搜索是一门实践性非常强的学问，读者应当尽量多地去尝试用它来解决问题（一个好的方法是去做习题），通过对不同问题的单独分析，逐步积累经验。笔者希望读者领会的东西远非这几个题目所能覆盖到的。

1.6.1　状态空间

读者不妨考虑这样一个问题，在枚举算法中，是按照生成方案→验证方案这样的两步曲来进行的。除了事前分析之外，两步之间没有任何联系，是孤立的。一个很自然的想法产生了：能不能在生成方案的中途也做一些验证，从而避免一些肯定不会是解的方案生成呢？答案是肯定的。不过需要把解的生成分成若干步骤，这样才给每个步骤的验证提供了可能。

状态　状态转移　智能体　在介绍状态空间之前，先明确几个概念。通俗地说，状态（state）是对问题在某一时刻的进展情况的数学描述，状态转移（state-transition）就是问题从一种状态转移到另一种（或几种）状态的操作。如果只有一个智能体（Agent）可以实施这种状态转移，则目的是单一的，也就是从确定的起始状态（start state）经过一系列状态转移而到达一个（或多个）目标状态（goal state）。

简单的多智能体系统　如果不止一个智能体可以操纵状态转移，例如两人下国际象棋，那么它们可能会朝不同的，甚至是对立（例如所有的对战游戏）的目标进行状态转移。这样的问题只简单介绍一下双人对弈的情形。

状态空间　搜索的过程实际是在遍历一个隐式图（请回忆 1.3 节介绍的“图”的概念），

它的结点是所有的状态，有向边对应于状态转移，而一个可行解就是一条从起始结点出发到目标状态集中任意一个结点的路径。这个图称为状态空间（state space），这样的搜索称为状态空间搜索（Single-Agent Search），得到的遍历树称为解答树。

刚才的解释过于形式化了，读者可能不容易理解。这没有关系，后面的例子会让你明白这段话的意思。

1.6.2 盲目搜索算法

盲目搜索算法的种类比较多，介绍几种比较简单的：

纯随机搜索（Random Generation and Random Walk） 听起来比较“傻”，但是当深度很大，可行解比较多，解的深度又不重要的时候还是有用的，而且改进后的随机搜索可以对付解分布比较有规律（相对密集或平均，或按黄金分割比例分布等）的题目。一个典型的例子是：你在慌乱中找东西的时候，往往都是进行随机搜索。

广度优先搜索（BFS）和深度优先搜索（DFS） 大家都很熟悉它们的时间效率，空间效率和适用的题目类型了吧。广度优先搜索的例子是你的眼镜掉在地上以后，你趴在地板上找。因为你总是先摸最接近你的地方，如果没有，再摸远一点的地方……深度优先搜索的典型例子是走迷宫——因为你没有办法用分身术来站在每个走过的位置。需要特别指出的是。这两者的双向或者反向搜索方式有时也采用。

重复式搜索 这些搜索通过对搜索树扩展式做一些限制，用逐步放宽条件的方式进行重复搜索。这些方法包括：

迭代加深搜索（Iterative Deepening） 限制搜索树的最大深度 Dmax，然后进行搜索。如果没有解就加大 Dmax 再搜索。虽然这样进行了很多重复工作，但是因为搜索的工作量与深度成指数关系，因此上一次（重复的）工作量比起当前的搜索量来是比较小的。这种方法适合搜索树总的来说又宽又深，但是可行解却不是很深的题目（一般的深度优先可能陷入没有解的又很深的地方，广度优先空间又不够）。

迭代加宽搜索（Iterative Broadening） 它限制的是从一个结点扩展出来的子结点的最大值 Bmax,但是因为优点不是很明显，应用并不多，研究得也比较少。

柱型搜索（Beam Search） 它限制的是每层搜索树结点总数的最大值 Wmax。显然这样搜索树大小与深度成正比，但是可能错过很接近起点的解，而增加 Wmax 的时候保留哪些结点，Wmax 增加多少是当前正在研究的问题。

下面，结合一些实例子来介绍几个比较重要的搜索方法。

1. 深度优先搜索

加密网格[1]

为了保密，海军舰队司令官和舰长们使用一种特殊的栅格对联络信息进行加密。这种栅格是一张边长为 $2N$ 的正方形网格，挖去了 N^2 个格子。加密方法如图 1-87 所示。

[1] 题目来源：CEOI 96

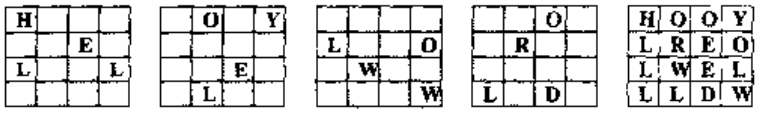


图 1-87　加密网格和一个加密过程

把栅格铺在一张方格纸上，从上到下从左到右在各个小孔处按顺序在纸上填入原文的前 N 个字母。然后把栅格顺时针旋转 90°，再次从上到下从左到右在各个小孔处填入原文的下 N 个字母……直到栅格转回原来的位置，原文的 $4N$ 个字母都被填到了方格纸上。加密网格的构造必须保证每张纸上的每个格子恰好被填了一个字母。

给一个加密后的结果以及原文，求一个加密网格。

网格中有 $4N^2$ 个格子，挖掉了 N^2，似乎需要枚举有 $C(4n^2,n^2)$种方法。其实并没有这么多，网格是要旋转的，而它的四个旋转恰好要覆盖所有的格子，因此可以把旋转重合的格子作为一类，每类四个格子，共 n^2 个。这样，实际上是在每一类中选一个格子，共有 $4^{n\times n}$ 种方案。也就是说，只要在搜索的时候避免选中同一类中的格子，就可以减少一些不必要的枚举。

虽然方案数减少了，但还是有很多。不应该采用 1.2 节中介绍的完全枚举的方法，而是一点一点地试验，及时排除不必要的枚举。在本题中，需要分别确定第 1，2，3，4 个格子的位置。由题目，它们应该填着字母‘H’，‘E’，‘L’，‘L’。这样，不用枚举就直接确定了第一个格子在左上角（因为只有这一个格子填着字母“H”），然后枚举第二个字母的两种情况，在每种情况下，再枚举第三个字母‘L’的每种可能位置。

回到这个问题中来。虽然算法总会正确地找到解，但是大家可能会觉得这样太慢了。的确，搜索算法的时间复杂度通常是指数级的（多项式复杂度的“搜索”的本质已经不再是这里所讲的状态空间搜索了）。不过就本题而言，实际情况并不是太糟。如果某个字母特别少，那么枚举量会比较少；如果字母很多，那么解会比较多。只要及时排除不可能的情况，通常会比较容易找到解的。

深度优先搜索算法

按照深度优先的顺序遍历状态空间，通常用递归或者栈来实现。用递归往往会使程序更加简短和易于调试，所以把用递归实现的程序框架写在这里，供读者参考。

```
procedure DepthFirstSearch(State:Statetype; depth:integer);
begin
  for Operand:=1 to OperandCount(State) do
  begin
    NewState:=DoOperand(State,Operand);
    If Answer then
       PrintAnswer
    Else if depth<maxdepth then
       Search(NewState, depth);
```

```
    End;
  end;
```

框架的解释　对于第一次学习搜索的读者来说，上面的代码需要解释一下。首先，depth 指的是搜索的层树，也是搜索树的层数。Operand 指的是算符，它的含义很广，只要是从一个状态转移到另外一个状态的方法，都可以看成是算符。例如在刚才的例子中，算符就是“当前空格的位置”，它是解答树的边（请读者再次仔细观察那棵树）。Answer 过程用来判断当前状态是否为目标状态。DoOperand 是在当前状态 State 上使用算符 Operand 后得到新状态 NewState 的一个过程。只要层数没有达到上限（即：没有搜索完所有应该搜索的东西），就从新状态 NewState 开始递归搜索下一层。这个框架只是最简单的，它有很多地方可以变形。另外，还有一些地方需要注意。

剪枝优化　有时候在到达叶子结点之前就可以断定以某结点 u 为根的子树不可能包含可行解或者最优解，因此不需要扩展这棵树，就像是拿一把剪刀把这棵子树剪去，因此称为剪枝。在后面会专门探讨这个问题。

搜索顺序　在搜索之前先要给需要枚举的对象排序，然后一个一个地搜索。如果没有顺序（例如在上题中，第 i 层并不是一定要枚举第 i 个格子的位置，而是随便选一个格子确定它的位置），会重复得到很多本质相同的解（想一想，为什么？）。好的顺序比一般顺序扩展的结点更少。

结点扩展顺序　不同的算符有主观上的“好坏”之分。先尝试“看起来比较好”的算符往往可以提前得到可行解或者是比较优的解。

对于刚才的题目来说，状态应是已经确定的孔的位置。枚举下一个孔可能的位置，然后进行递归搜索就可以了。

2. 广度优先搜索

广度优先搜索的想法就要简单得多。如果代价和搜索树深度成正比，那么可以通过广度优先搜索得到解。由于空间占用大，BFS 用处不是很广，一般只用在路径寻找问题中，但是一旦使用，它比深度优先搜索要快得多。

最优程序[①]

写程序不是一件简单的事情，尤其是要把程序写得尽量短。假设有这样一台计算机，它只有一个数据栈，并支持整数的五种运算：ADD，SUB，MUL，DIV，DUP。假设栈顶的三个元素分别为 a,b,c，如图 1-88 所示，五种运算的效果依次为：

ADD：a 和 b 依次出栈，计算 $a+b$，把结果入栈。

SUB：a 和 b 依次出栈，计算 $b-a$，把结果入栈。

MUL：a 和 b 依次出栈，计算 $a\times b$，把结果入栈。

DIV：a 和 b 依次出栈，计算 b/a 并向下取整，把结果入栈。

DUP：a 出栈，再连续让两个 a 入栈。

① 题目来源：ACM/ICPC Regional Contest SWERC 1997

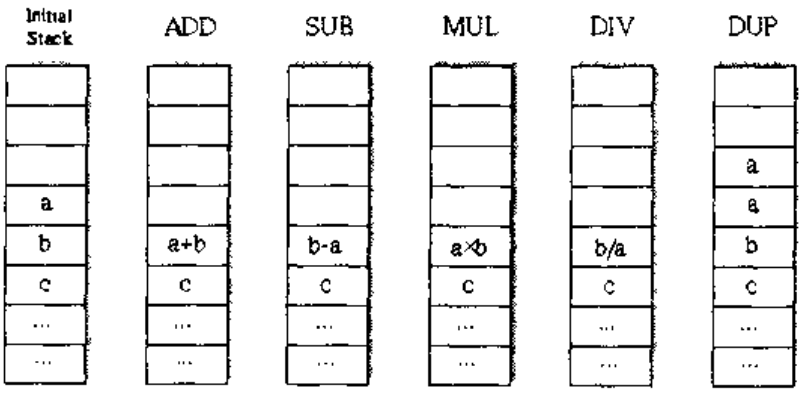


图 1-88　该计算机支持的五种运算

遇到以下情况之一，机器会产生错误：

- 执行 DIV 操作时，栈顶元素为 0。
- 执行 ADD，SUB，MUL，DIV 操作时栈里只有一个元素。
- 运算结果的绝对值大于 30 000。

给出 $n(n\leqslant 5)$个整数对(a_i,b_i)，$(a_i$互不相同)设计一个最短的程序，让它对于任何 $1\leqslant i\leqslant n$，如果栈中只有一个元素 a_i，则程序执行后栈定元素都是 b_i。

如果没有除法操作，那么一个程序实际上是一个多项式。但由于除法操作（而且还是向下取整的整数除法）的存在，程序的等价表达式不一定是多项式了。不过还是可以先求 Lagrange 插值多项式得到一个不错的解，同时，这也保证了：如果忽略了计算过程中结果不能大于 30 000 的限制，本题是一定有解的！

当然，这只是一点点理论分析，要想完整地解决这个问题，只能搜索。怎样搜呢？深度优先是不行的，因为本题没有深度限制，深度优先可能陷入一条没有解的子树永远也走不出来了。正确的方法是广度优先搜索，即完全地依次考察深度为 1，2，3，…的结点。这样，一旦求出了一个可行的程序，它一定是最短的。

广度优先搜索　按照广度优先的顺序遍历状态空间，一般用 OPEN_CLOSE 表来实现。不要用循环队列，因为需要保存已出列的结点（还记得 1.3 节中的思考题吗？）。参考算法框架如下：

```
Procedure BreadthFirstSearch(InitialState: StateType);
Begin
  Enqueue(InitialState);
  While Not EmptyQueue do
  Begin
    Dequeue(State);
    For Operand:=1 to OperandCount(State) do
    Begin
      NewState:=DoOperand(State, Operand);
      If Answer(NewState) then PrintAnswer;
```

```
        Else Enqueue(NewState);
      End;
      End;
End;
```

上面的框架中，有一些地方可以根据需要改变：

路径的保存　有时候除了需要找到一个目标状态，还需要保存从初始结点到它的路径。路径可以随状态保存，也可以用链表穿起来。如果保存所有结点或者路径长度不定，推荐用链表。

判重　除非隐式图是无环的（如本题），或者有很强的结点扩展方式来避免生成重复，广度优先搜索是需要判重的。判重工作在这个框架中省略掉了，大家可以自己补充。后面在介绍 A*的时候会重新提到这个问题。

在本题中，状态本来应该是当前程序的指令序列，但是由于序列长度不定，可以只记录当前指令来记录它的前趋结点，这样只要沿着前趋返回到初始结点，把沿途结点的当前指令串起来就得到了完整的指令序列了。为了结点扩展和判断的方便，还可以记录下栈的情况。

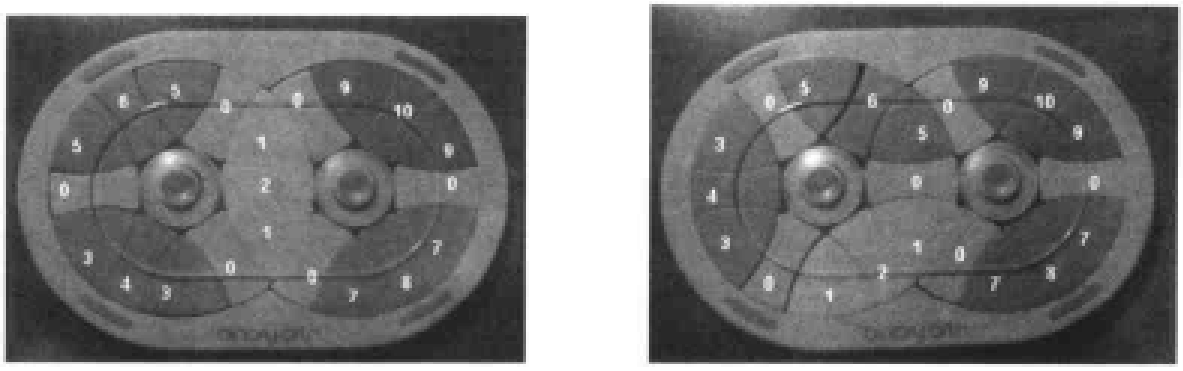
3. 双向广度优先搜索

双向搜索看起来是广度搜索的“加强版”，但应该看到，它的适用范围要窄得多。

一般说来，方法是从初始结点和目标结点开始分别做广度优先搜索，每次检测两边是否重合。每次扩展结点后总是选择结点比较少的一边进行下次搜索，而不是机械地两边交替。不过这也不是惟一的方法，下面就有一个例子，它是搜索完一面后再搜索另一面：

旋转的玩具[1]

下图是一种新式玩具。该玩具由两个圆盘组成，两个都可以顺时针或逆时针旋转。例如，图 1-89(a)的左边圆盘顺时针旋转一个单位以后就成了图 1-89(b)。在本题里，需要写一个程序，旋转最少的单位数，把任意一个初始状态转化为图 1-89(b)(最多只允许旋转 16 次)。



(a)　　(b)

图 1-89　新式玩具

[1] 题目来源：ACM/ICPC Regional Contest

本题也是道十分典型的路径寻找问题。但是本题和 8 数码问题的差别也是明显的。首先，本题很难找到一个合适的启发函数，该玩具一次就改变了 12 个数字的位置，而且数字还有重复。对于这样的情况，没有必要费尽心思去设计启发函数，而考虑盲目搜索。

周界搜索　由于目标状态和初始状态都确定，显然可以用效率比较高的双向广度优先搜索。但是由于在每个测试数据中，目标结点都是一样的，反向搜索进行一次就够了。这样，先反向搜索一次，把结点都保存起来再正向搜地方法叫**周界搜索**（perimeter search）。

周界的表示　第一次搜索的结果保存在一个称为“周界”的结构中，那么它应该如何表示呢？显然，周界中的结点不再变更，只需要对它进行检索，而不进行插入和删除，在后面会学到，在这种情况下，HASH 表或是字母树都可以胜任。

正向搜索策略的选择　既然采用的是周界搜索，正向搜索的方法就比较灵活了。可以用广度优先搜索，深度优先搜索，迭代加深搜索，考虑到目标是路径长度最短，采用最容易实现的迭代加深搜索，当然广度优先也可以。

反向搜索深度　在本题中，反向搜索的深度对于程序的效率影响比较大。深度太浅，正向搜索的时间效率会很低，而深度太大，不仅反向搜索本身时间效率不高，而且要求更大的 HASH 表或者字母树来处理更多的元素。实验指出，当反向搜索深度为 7（比最大深度的一半略小）的时候时间效率最好。

作为比较，表 1-20 列出了一些程序的运行时间（程序 1 使用深度优先搜索，程序 2 为反向搜索 7 层，结点保存在无序数组里。程序 3 把程序 2 中的结点改为保存在有序数组里，用二分查找检索。程序 4 把程序 2 中的结点保存在 HASH 表里，程序 5 把反向搜索改为 8 层）。

表 1-20　程序运行时间

程　序	1	2	3	4	5
时　间	Too long	8.01	0.97	0.08	0.24

练　习　题

编程题：

1.6.1　超级天平[①]

如图 1-90 所示是一个处于平衡状态的超级天平。天平上有 5 个秤砣，重量分别是 1kg，2kg，…，5kg，相临两个刻度之间的距离为 1 米。

可以用这两个式子来检验天平的平衡性。

第一层：$-3\times3+(-1)\cdot 5+2\times(1+2+4)=0$　　第二层：$-2\times1+(-1)\times2+1\times4=0$

用字符串来描述超级天平的结构，各个数表示秤砣的位置。例如上面的天平结构就可

[①] 题目来源：Baltic Olympiad in Informatics 1999

以被描述成：(-3, -1,2(-2, -1,1))。一个平衡的超级天平也可以用一个字符串来表示，其中表示秤砣的位置的数被替换成了秤砣的重量。例如图 1-90 的天平就可以被描述成：(3,5,(1,2,4))。

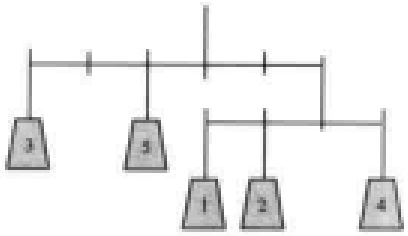


图 1-90　一个平衡的天平

写一个程序，对于一个含 $n(n\leqslant 17)$个空位的天平结构，使用 1kg…nkg 的秤砣恰好各一次，构造出一个平衡的天平。

1.6.2　奶牛的加密术[①]

农夫 Brown 的奶牛和农夫 John 的奶牛密谋一起从自家主人的农场中同时逃跑出来，而且发明了一种独特的加密术，以便对他们之间传递的消息进行加密。

例如，其中一只奶牛想说“International Olympiad in Informatics”，它会随机地在信息中插入三个字母 C、O、W（保证 C 在 O 前，O 在 W 前），然后把从 C 到 O 之间的部分和 O 到 W 之间的部分交换，从而完成一次加密处理。例如：

International Olympiad in Informatics -> CnOIWternational Olympiad in Informatics

或者（对于同一条信息，不同的加密方法会得到不同的结果）：

International Olympiad in Informatics -> International Cin InformaticsOOlympiad W

奶牛们总是相当谨慎，它们觉得必须对一条信息反复加密（即后一次加密的原文是前一次加密的结果）以后才能放心，哪怕解密的时候麻烦一点。

一天晚上，农夫 John 的奶牛收到了一条经过多重加密的信息。写一个程序帮助它判断一下这条信息的原文是否可能为：“Begin the Escape execution at the Break of Dawn”（明天一早开始逃跑）。

1.6.3　智力玩具[②]

Balltazar 教授是一个足球迷，而他的生日恰好是他动身去法国观看 1998 世界杯足球赛的前几天。因此，他的朋友们送了他一个十二面体的智力玩具，以便他在观看比赛觉得乏味的时候可以娱乐一下。

这个玩具有 12 个全等的 5 边形面，编号为 1…12。图 1-91 就是这个文具的前后两半。该图同时包含了 12 个边的一种编号方法，以后将统一采用这种编号法，对于所有数据都一样。首先要把玩具的两半用胶水按这样的方法粘在一起：面 7 和面 8,12,11,2,6 相临（两个面相临当且仅当它们共边），而且左图中两条边 a 和 b 分别和右图中的边 a 和 b 重合（粘在一起）。

---

① 题目来源：USACO
② 题目来源：CEOI 98

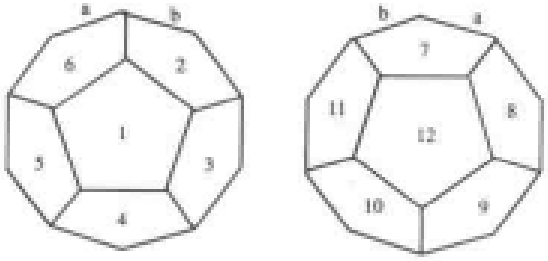


图 1-91　玩具的两半

把玩具拼好以后，就可以往玩具上贴贴片了。有 12 张 5 边形的贴片，也编号为 1 到 12。贴片上的每边都有{0,1,2}中的一个数字。每张贴片恰好和玩具的各个面一样大，因此可以把任意一张图贴到玩具的每个侧面。

Balltazar 教授希望找到一种帖法，使得任意两个相临的贴片的公共边上写着同一个数字，他应该怎样做呢？

1.6.4　平分资源[①]

某国家划出一块矿产资源丰富的土地，并给予许多优惠的政策，以招募大型矿产资源开发公司来进行开发。

这块土地可以被划分成 6×6 的网格（如图 1-92），在这些方格中共有 4 个煤矿，每个矿藏占一格。现在要将这些资源平均分配给四个应招的公司 A，B，C，D 进行开发（每个公司开发一个煤矿和一个铁矿），同时要将这土地平分成四块给这四个公司，以便管理。

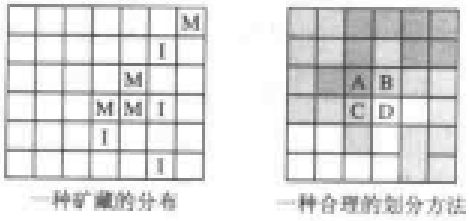


图 1-92　矿藏分布和划分方法

为了减少争议，应按以下的要求进行划分：

1. 土地只能沿网格线进行划分；
2. 为了集中管理，中心的四个网格已经分给四个公司建立总控站，如图所示；
3. 每个公司分得的土地必须都与公司的总控站连通；
4. 这四块土地的大小、形状必须相同；
5. 每块土地恰好包含一个煤矿和一个铁矿；
6. 每个公司的土地必须连成一片。

现在这个国家需要一种合理的划分方案，以及所有可能的划分方案的总数。

请编程帮助解决这个问题。

***1.6.5**　百慕大魔鬼三角的蛋糕[②]

百慕大魔鬼三角的人类做任何东西都是三角形的，直到某一天，一位糕点师傅想做一

[①] 题目来源：CTSC 97

[②] 题目来源：ACM/ICPC Regional Contest Tehran 2001. Burmuda

块正六边形的蛋糕，但他所用的材料只能是一堆大大小小的小正三角形。例如图 1-93 就是用边长 2 和 3 的小正三角形拼成的边长为 9 的正六边形。

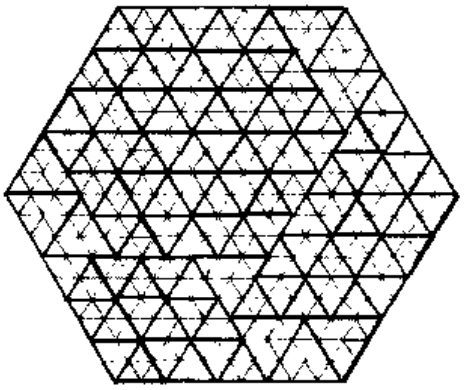


图 1-93　蛋糕举例

现在给要做的蛋糕的边长 s 和 n 种能作为原材料的小正三角形的边长，问糕点师傅的蛋糕能做成功吗？

1.6.6　新别墅[1]

Black 先生买了一座漂亮的新别墅，每个房间都有自己的一盏灯。这本来应该是好事，然而 Blank 先生却烦恼不已。原来，虽然大多数房间的都有这些灯的开关，但是这些开关和它控制的灯却常常不在同一个房间里。虽然经销商说这正是这别墅与众不同的地方，但是 Black 先生总觉得是工程师们造房子的时候心不在焉所致。

有一天，Black 先生回到家的时候，天色已晚。他站在走廊上，发现没有一个房间的灯是亮着的。他惧怕黑暗，因此从不进入一个漆黑的房间，也从不把自己所在房间的灯关掉。但是他实在是太累了，希望以最快的速度到达他的卧室，而让其他灯（包括走廊里的）都关闭着。从一个房间走到相临的房间，打开一个灯，关闭一个灯所花费的时间都是 1。

1.6.3　启发式搜索算法

启发式搜索算法　我们觉得一些问题很有“想头”，主要是因为启发信息比较多，思考起来容易如手，但是却不容易找到解。我们不愿意手工一个一个盲目地试验，同样也不愿意程序机械地搜索。也就是说，希望尽可能地挖掘题目自身的特点，让搜索智能化。我们将会看到，下面介绍的启发式搜索就是这样的一种智能化搜索方法。

估价函数　刚才的算法中，没有利用状态本身的信息，只是利用了状态转移来进行搜索。事实上，在解决问题的时候常常会估计状态离目标到底有多接近，进而对多种方案进行选择。把这种方法用到搜索中来，可以用一个状态的估价函数来估计它到目标状态的距离。这个估价函数是和问题息息相关，体现了一定的智能。为了以后叙述方便，先介绍一

[1] 题目来源：ACM/ICPC Regional Contest

些记号，如表 1-21 所示。

表 1-21　启发式搜索用到的符号

S	问题的任何一种状态
$H^*(s)$	S 到目标的实际（最短）距离，可惜事先不知道
$H(s)$	s 的估价函数，s 到目标距离的下界，也就是 $h(s) \leqslant h^*(s)$。如果 h 函数对任意状态 s_1 和 s_2，还满足 $h(s_1) \leqslant h(s_2)+c(s_1, s_2)$（其中 $c(s_1, s_2)$代表状态 s_1 转移到 s_2 的代价），也就是状态转移时，下界 h 的减少值最多等于状态转移的实际代价，我们说 h 函数是**相容**（**consistent**）的（其实就是要求 h 不能减少得太快）
$G(s)$	到达 s 状态之前的代价，一般就采用 s 在搜索树中的深度
$F(s)$	s 的启发函数，也就是到达目标的总代价的估计。直观上，应该有 $f(s)=g(s)+h(s)$，即已经付出的和将要付出的代价之和。如果 f 是相容的，对于 s_1 和它的后辈结点，有 $h(s_1) \leqslant h(s_2)+c(s_1, s_2)$两边同时加上 $g(s_1)$，有 $h(s_1)+g(s_1) \leqslant h(s_2)+g(s_1)+c(s_1, s_2)$，也就是 $f(s_1) \leqslant f(s_2)$。因此 f 函数单调递增

下面举一个例子，介绍估价函数的设计方法。

编辑书稿[①]

现在的作家们不再使用笔和纸来写书了。他们直接把文字输入到家里的计算机里，然后必要的时候做一些修改。

现在有一位知名的小说家想给他的小说做一些重大的修改，重新安排一些段落的顺序。他把所有段落按照改动后的编好了号（P_1代表改动后的第一段，依次类推），然后准备用“剪切（cut）”和“粘贴（paste）”操作完成这项修改。

他一次可以剪切一段，也可以剪切多段，但是这些段在当前的编辑状态下必须是连续的。而且，由于剪贴板只有一个，他不能连续两次使用“剪切”操作，而必须“剪切”和“粘贴”交替进行。例如，有6段文字：$P_2, P_4, P_1, P_5, P_3, P_6$，他需要两次操作，即：

（1）剪切P_1，粘贴到P_2前，形成$P_1, P_2, P_3, P_4, P_5, P_6$

（2）剪切P_3，粘贴在P_3后，形成$P_1, P_2, P_3, P_4, P_5, P_6$

他希望使用“剪切”的次数尽量少，他应该怎么做呢？

本题和传统的 8 数码问题有类似之处，但是它的启发函数并不是很容易找。如果也是把所有段落的“离家距离和”作为 h 值的话，则 h 函数是不相容的，因为一次剪切可能使很多段落的离家距离同时减少很多。即使是选“不在正确位置上的段落数目”也没有满足相容条件，因为可能一次让很多段落同时“回家”。

剪切是以一些连续段落为一个整体，这启发我们应该设计一个和“相对位置”有关系的 h 函数，例如 $h(s)$=“后继段落正确的段落数目”。显然假设把连续段落 S 从 P_1 的后面

[①] 题目来源：ACM/ICPC Regional Contest Kanpur 2001

移动到 P_2 的后面，则只有①S 的最后一个段落；②P_1；③P_2 这三个段落的后继有变化。因此每次 h 最多减少 3。虽然这时候仍然不相容，但是它已经是常数级别了。可以把 $h'=h/3$ 作为估价函数值，则每次 h 最多减少 1，但为了方便，推荐把 f 扩大 3 倍，即让 $f(s)=h(s)+3g(s)$，这时候 f 仍然单调增。设计好了启发函数后，应该采取什么样的搜索算法呢？

贪心搜索（Best-First Search）　像广度优先搜索一样用一个队列储存待扩展，但是按照 h 函数值从小到大排序(其实就是优先队列)。显然由于 h 估计的不精确性，贪心搜索不能保证得到最优解。要想得到最优解，需要使用下面介绍的几种方法。

A*算法　和贪心搜索很类似，不过是按照 f 函数值进行排序。但是这样会多出一个问题：新生成的状态可能已经遇到过了。为什么会这样呢？由于贪心搜索是按照 h 函数值排序，而 h 只与状态有关，因此不会出现重复，而 f 值不仅与状态有关，还与状态转移到 s 的方式有关，因此可能出现同一个状态有不同的 f 值。解决方式也很简单，如果新状态 s_1 与已经遇到的状态 s_2 相同，保留 f 值比较小的一个就可以了（如果 s_2 是待扩展结点，是有可能出现 $f(s_2)>f(s_1)$的情况的，只有已扩展结点才保证 f 值递增）。可以证明，A^*算法扩展到的第一个目标结点一定是离初始状态最近的一个。需要注意的是 $h=0$ 的时候就是广度优先搜索，它没有启发因素（代价估计），但是仍然能求出最优解，因为 f 仍然单调递增。

A*算法的实现　算法需要按照 F 函数值递增的顺序扩展结点。由于每次取 F 函数最小的结点，需要设置一个保存待扩展结点的结点表一，又因为需要判重，所以还需要保存所有结点（包括已扩展的和待扩展的）的表二。在下面的框架中，没有给出两个表的具体实现，但是可以看出，表一有“取最小值”。“插入”和“替换”操作；表二有“查找并插入”操作。如果读者还没有忘记 1.4 节的话，可以立刻知道，比较好的方法是，用堆来实现表一，排序二叉树实现表二。另外判重也可以用无序数组（虽然要一个一个判断，但是插入方便，而且可以在数组中加入一个该结点的特征值来避免一些不必要的判断，还记得 1.3 节中提到的字符串匹配的 *RK* 算法吗？），它比排序二叉树实现起来简单，效果也不错。

程序框架

```
Procedure Expand(State:StateType);
Begin
  for I:=1 to OperandCount(State) do
  begin
    NewState := DoOperand(State, i);
    NewState.h := CalculateH(NewState);
    NewState.g := State.g + 1;
    NewState.f := NewState.g + NewState.h;
    NewState.father := State;
    NewState.LastOp := OperandCount;
    i:=FindInsertTableTwo(NewState);      // 在表二中找，找不到就插入
    If i=0 then
      InsertToTableOne(NewState);              // 新状态，因此插入到表一中
    Else if NewState.f < State[i].f then
```

```
      ReplaceInTableOne(NewState, i);          // 在表一中用 NewState 替换 i
    end;
  end;

  procedure Astar(depth:integer);
  begin
    CalculateInitialState;
    TableOne := [InitialState];                // 表一：待扩展结点集
    TableTwo := [InitialState];                // 表二：待扩展和已扩展结点集
    ok:=false;
    while not ok and not TableOneEmpty do      // 没找到解且待扩展结点非空
    begin
      State := ExtractMinInTableOne;           // 找 f 值最小的带扩展结点
      If IsAnswer(State) then
      Begin
        PrintState(State);
        Ok := true;
          End
      Else Expand(State);                      // 扩展结点 State
    End;
    If not Ok then NoAnswer;
  End;
```

A*算法的弱点　最明显的弱点是空间需求太大，是指数级别的。如果不考虑空间问题，那么 A*在理论上是时间最优的。

既然 A*算法存在空间问题，那么能不能借用深度优先搜索的空间优势，用重复搜索的方式来缓解危机呢？经过研究，Korf 于 1985 年提出了一个 Iternative Deepening A*(IDA*)算法，比较好地解决了这一问题。在介绍这个算法之前来看一个例子。

埃及分数[①]

在古埃及，人们使用单位分数的和（形如 1/*a* 的，*a* 是自然数）表示一切有理数。如：2/3=1/2+1/6，但不允许 2/3=1/3+1/3，因为加数中不允许有相同的。

对于一个分数 *a*/*b*，表示方法有很多种，其中加数少的比加数多的好，如果加数个数相同，则最小的分数越大越好。例如：

19/45=1/3+1/15+1/45
　　=1/3+1/18+1/30
　　=1/4+1/6+1/180
　　=1/5+1/6+1/18.

① 题目来源：经典问题

最好的表示方法是最后一种，因为 1/18 比 1/180,1/45,1/30,1/180 都大。给出 $a,b(0<a<b<1\,000)$，编程计算最好的表达式。

本题的特别之处在于它的状态空间是无限大的！首先，深度没有明显的上界；其次，每个加数的选择也是无限的，即：搜索树的每个结点的儿子是无限的！如果用深度优先搜索很可能陷入死循环。而对于题目的数据规模我们也无法实现预料空间是否足够。这时，考虑采取迭代加深搜索的方法。迭代搜索本身解决了深度上界的问题，而且可以利用一些简便的条件去掉明显不能导致最优解的儿子。按照分母递增的顺序来进行搜索，如果搜索到 i 层的时候前 i 层分数的和为 c/d,而第 i 个分数为 $1/e$，则接下来至少还需要 $(a/b-c/d)/(1/e)$ 个分数，加起来的总和才能达到 a/b。例如当前搜索到 19/45=1/5+1/100+…则后面的分数每个最大为 1/101，至少需要(19/45−1/5) / (1/101) =23 项总和才能达到 19/45，因此前 22 次迭代是根本不会考虑这棵子树的。此方法对 8 个测试数据的运行时间如表 1-22。

表 1-22　8 个测试数据的运行时间

Case	1	2	3	4	5	6	7	8
A/B	2/15	36/612	27/441	4/109	59/211	101/103	907/911	523/547
时间	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	0.11	0.60

而采用通常的广度优先搜索或者深度优先+上下界剪枝的程序对于最后几个数据的运行时间为好几分钟，甚至更长。

IDA*算法　由前所述，IDA*算法是基于迭代加深的 A*算法，忽略所有 f 值大于深度限制的结点。为了进一步加快速度，这里给出的是基于栈的算法框架，请读者仔细体会。

```
Procedure IDA_STAR(StartState)
Begin
  PathLimit := H( StartState ) - 1;
  Success := False;
Repeat
  inc(PathLimit);
  StartState.g:= 0;
  Push(OpenStack,StartState);
  Repeat
    CurrentState:=Pop(OpenStack);
    If Solution(CurrentState) then
      Success = True
    Else if PathLimit ≥ CurrentState.g + H(CurrentState) then
      Foreach Child(CurrentState) do
        Push(OpenStack, Child(CurrentState));
  until Success or empty(OpenStack);
until Success or ResourceLimitsReached;
end;
```

框架的解释　解释一下这个框架。一开始，把深度最大值 Dmax 设为起始结点的 h 值，开始进行深度优先搜索，忽略所有 f 值大于 Dmax 的结点，减少了很多搜索量。如果没有解，再加大 Dmax 的值，直到找到一个解。容易证明这个解一定是最优的。

IDA*的优势　由于改成了深度优先的方式，与 A*比较起来，IDA*更加实用：①不需要判重，不需要排序（因此堆和排序二叉树都不用了！），只用到栈，操作简单。②空间需求大大减少，与搜索树大小成对数关系。

DFSBnB 算法　还有一种启发式搜索算法，其实前面已经提到过了，这就是深度优先搜索+最优性剪枝（DFSBnB）。关于 IDA*和 DFSBnB，有一个很有趣的现象，就是擅长解决问题的互补性。另外，他们的框架很简单，几乎可以在 DFID 和 DFS 的基础上直接改，不过剪枝上有差别。

小知识——启发式双向搜索

如果把双向搜索看作两个不同方向搜索的综合，那么单是这两种搜索算法本身就有很多种选择，除了 BFS，还可以选择 DFS、A*等。而事实上双向搜索并不是两边搜索的简单加合，还需要有一个双向搜索控制策略来调节，这也是双向搜索具有多样性的原因之一。

双向广度优先搜索实际上属于交替搜索。在实际运用中总是先扩展结点数少的一边。在交替搜索中，人们试图将广度优先搜索替换成其他的单向搜索方法，但是效果都不理想。例如，双向的深度优先搜索一般只被用在起始结点和目标结点都确定且需要求所有可行解的算法中，在最优化题目中，两边的第一次“交汇”处的解不一定是最优的，而在求可行解的题目中，两边的搜索常常因为“擦肩而过”而迟迟找不到解。

在启发式搜索中，双向搜索一直被认为是价值不大的。因为受到双向广度优先搜索的影响，传统的双向启发式搜索，也是正反向交替的搜索。根据启发函数中估价函数的定义，它们可分为两种。

第一种用的是 Front-to-End 的估价函数。例如第一个双向启发式搜索算法 BHPA。在这个算法中，两边的估价函数是当前结点到 t(正向)或者 s(反向)的估计代价，整个算法就好象是两个 A*一样。理论研究表明，两边搜索的第一次交汇并不一定是最优解，而实际情况也说明这种算法并不比 A*好。

第二种用的是 Front-to-Front 的估价函数，也就是计算一边搜索到的某个结点和另一边搜索到的每个结点的估计代价。由于计算估价函数的计算量过大，这种方式也并不实用，有时候必须牺牲解的最优性来换取一些时间效率。

练　习　题

编程题：

1.6.7　赶时间[1]

大家都在赶时间的时候，十字路口上通常会出现塞车的情况。如图 1-94，十字路口是

[1] 题目来源：ACM/ICPC Regional Contest

一个 6x6 的网格。交通工具有两种，一种是汽车（Car），长为两格，另一种是卡车（Truck），长为三格。两种车的宽度都是一格。你驾驶着 0 号汽车，希望在尽量短的时间接触十字路口的最右边，这样可以认为你已经脱离了令人头疼的十字路口。

在每个时间单位，最多只有一个交通工具可以移动，且只能往前或往后移动一个单位，不许转弯，也不许开到 6x6 网格外面（外面更加塞车！）。例如在下图中，Truck2 只能往南开一步，而 Car0 无法移动。

在图 1-94 的例子中，最少需要 16 步：Car1 往东开一步，Truck3 往北开一步，Car5 往北开一步，Car6 往西开三步，Truck7 往西开两步，Truck4 往南开两步，Truck2 往南开三步，Car0 往东开三步。这样，你的 Car0 到达网格的最右边，成功地走出了十字路口。

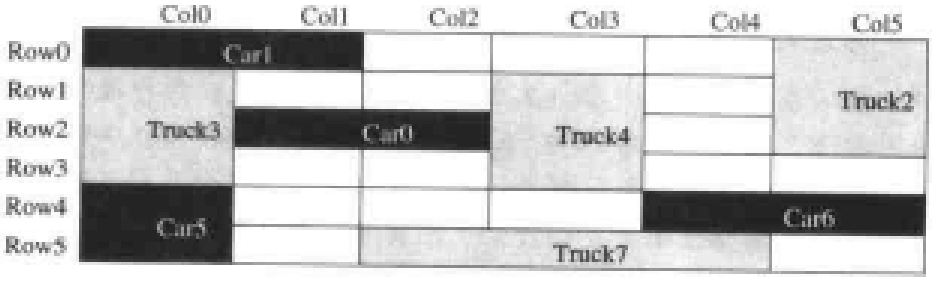


图 1-94　十字路口举例

1.6.8　永别了，朋友![1]

Liz 和 Lilly 以前是很要好的朋友，可是最近因为一点小事弄得很不愉快，最后决定相互说拜拜。不仅如此，她们都准备在她们的房子附近放一些石头挡在路中央，使得她们无论怎么走都看不到对方。

她们居住的村庄被分成了 N×N 个网格，Liz 和 Lilly 的房子分别在(1,1)和(*n*,*n*)。每个格子有三种类型：土地，岩石和湖泊。她们无法穿越岩石和湖泊，但是在土地上行动自如；岩石可以挡住她们的视线，但是土地和湖泊显然不行。她们每次只能朝东、南、西、北移动一格，而可以看到前方 k 格远的地方(她们从不斜着眼看东西)。例如：从(1,1)格最远可以看到(1,*k*+1)和(*k*+1,1)。她们两个都很懒，因此希望新放置好的岩石越少越好，而且不能离她们的房子太近——距离至少为 *m* 才行（两个格子(x_1,y_1)和(x_2,y_2)的距离是$|x_1-x_2|+|y_1-y_2|$）。当然，如果存在一些两人都无法走到的格子，她们是无法在那些地方放上石头的。

给出 *n*,*k*,*m* (5≤*n*≤20, 1≤*k*,*m*≤n)和村庄的地图，编程求出需要放置的石头的最小值(-1 代表无解）。

1.6.9　智破连环阵[2]

B 国在耗资百亿元之后终于研究出了新式武器——连环阵（Zenith Protected Linked Hybrid Zone）。传说中，连环阵是一种永不停滞的自发性智能武器。但经过 A 国间谍的侦察发现，连环阵其实是由 *M* 个编号为 1，2，…，*M* 的独立武器组成的。最初，1 号武器发挥着攻击作用，其他武器都处在**无敌**自卫状态。以后，一旦第 *i*（1≤*i*<*M*≤100）号武器被

[1] 题目来源：OIBH Online Programming Contest #1，命题人：刘汝佳
[2] 题目来源：NOI2003，命题人：刘汝佳

消灭，1 秒种以后第 i+1 号武器就自动从无敌自卫状态变成攻击状态。当第 M 号武器被消灭以后，这个造价昂贵的连环阵就被摧毁了。

为了彻底打击 B 国科学家，A 国军事部长打算用最廉价的武器——炸弹来消灭连环阵。经过长时间的精密探测，A 国科学家们掌握了连环阵中 M 个武器的平面坐标，然后确定了 $n(n\leqslant 100)$个炸弹的平面坐标并且安放了炸弹。每个炸弹持续爆炸时间为 5 分钟。在引爆时间内，每枚炸弹都可以在瞬间消灭离它平面距离不超过 k 的、处在攻击状态的 B 国武器。和连环阵类似，最初 a_1 号炸弹持续引爆 5 分钟时间，然后 a_2 号炸弹持续引爆 5 分钟时间，接着 a_3 号炸弹引爆……以此类推，直到连环阵被摧毁。

显然，不同的序列 a_1，a_2，a_3，…，消灭连环阵的效果也不同。好的序列可以在仅使用较少炸弹的情况下就将连环阵摧毁；坏的序列可能在使用完所有炸弹后仍无法将连环阵摧毁。现在，请你决定一个最优序列 a_1，a_2，a_3，…，使得在第 a_x 号炸弹引爆的时间内连环阵被摧毁，这里的 x 应当尽量小。

1.6.4　博弈问题算法

三角形大战[①]

在图 1-95 的棋盘上进行着一个游戏。游戏有 A、B 两人参加。A 先走，每人每次任选一条虚线填成实线。而如果某人填完一条线段后，该线段与另两条实线组成了一个单位三角形，该三角形被标记为该游戏者所有，且该游戏得到一次奖励机会，其中一步游戏如图 1-96 所示。当 18 条线段被填充完毕后，拥有三角形多的玩家获胜。编程求出当两个人都采用最好的对策时最后取胜的玩家。

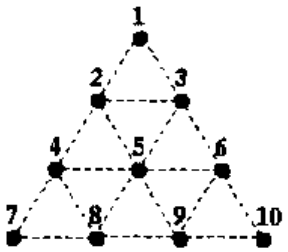


图 1-95　三角形大战的棋盘

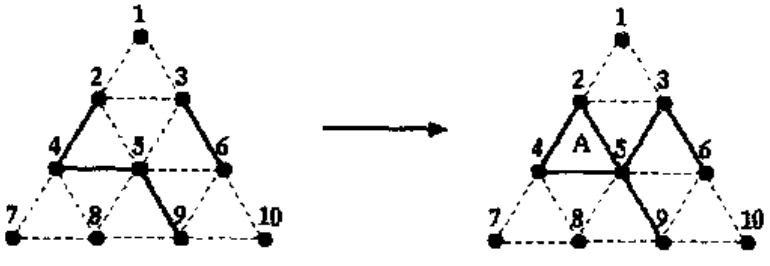


图 1-96　A 在图(a)涂线段 2-5 后获得三角形 2-4-5 和一次奖励机会并用它涂线段 3-5

[①] 题目来源：ACM/ICPC Regional Contest

本题是一道典型的博弈搜索题目，和前面一样，还是先介绍一些相关理论。

局面估价函数　我们给每个局面（state）规定一个估价函数值 f，评价它对于己方的有利程度。胜利的局面的估价函数值为∞，而失败的局面的估价函数值为-∞，假设没有和局。对于任意一个局面，应该怎样计算它的估价函数值 f 呢？

Max 局面　假设这个局面轮到己方走，有多种决策可以选择，其中每种决策都导致一种子局面（sub-state）。由于决策权在我们手中，当然是选择估价函数值 f 最大的子局面。因此，该局面的决策函数值等于子局面 f 值的最大值，把这样的局面称为 max 局面。

Min 局面　假设这个局面轮到对方走。它也有多种决策可以选择，其中每种决策都也会导致一种子局面。但由于决策权在对方手中，在最坏的情况下[1]，对手当然是选择估价函数值 f 最小（对对方最有利）的子局面。因此，该局面的决策函数值等于子局面 f 值的最小值，把这样的局面称为 min 局面。

终结局面　如果双方都不能走，显然胜负已分（请再次注意，我们假设没有和局），f 值根据规定取值。综上所述，得到了最单纯的极大极小算法。

完全极大极小过程　对于一个局面，递归计算它所有子局面的估价函数值。如果是 max 层，转移到其中函数值最大的子局面，否则转移到函数值最小的子局面。可以把已经算过估价函数值的局面都记录下来，以免重复工作。

```
function minmax(State, Player:integer):integer;
begin
  if WeWon then minmax:= +∞;
  else if WeLost then minmax:= -∞;
  else begin
    min:=+∞;
    max:=-∞;
    for Move:=1 to MoveCount do
    begin
      NewState := DoMove(State, Move);
      Value := minmax(NewState, Next(Player));
      If Value<min then min:=Value;
      If Value>max then max:=Value;
    End;
    if Player=MyProgram then minmax:=max;
    if Player=Opponent then minmax:=min;
  end;
end;
```

主观估价值　看到这里，读者也许会说，这样的话，所有局面的 f 值非∞即-∞！，的确如此。如果双方都采用最优策略，任意局面都是必胜的或者是必败的。但是在实际比赛

[1] 对于比较弱的对手，我们可以忽略一些“对手不大容易想到的策略”所导致的子局面，但这样做有风险

中，完整地计算出一个局面的 f 值计算量太大（想一想，为什么？），通常规定计算的最大递归深度 maxdepth，如果达到了该深度，f 值就是按照某种主观方法得出的估价值。只需要在过程参数里加上一个“depth”，当 depth 比较大的时候直接把局面估价函数值作为 minmax 的返回值。

Alpha-Beta 剪枝　聪明的读者可能已经注意到了，这样的算法做了很多无用功。举一个简单的例子。对于一个 max 局面 S，已经计算出了它的第一个子局面 S_1（为 min 局面）的 f 值为 5，而现在正在计算 S 的第二个子局面 S_2（也是 min 局面）的值，没有计算完。假设我们已经得知了 S_2 的某个子局面的 f 值为 2，那么 S_2 的 f 值至多才是 2，比 S_1 的 f 值小。因此，选择 S_1 肯定比 S_2 好，应马上停止计算 S_2。对于 min 局面，有类似的例子，请读者试着写出来。这个简单的例子，就是著名的 Alpha-Beta 剪枝的依据，请大家自己写出剪枝的条件，这里不再赘述。

回到刚才的问题。题目要求当“两个人都采用最好策略”时的获胜者，意味着搜索必须完全，但这样做很慢。关于该算法的优化，首先可以根据自己的主观感觉想到以下两种方法。

方法一：如果一方已经获得了 5 个三角形，实际上他已经肯定胜利了，因为这样对方即使拿到了所有剩下的三角形也不会超过他。

方法二：如果存在一些条填充以后可以获得三角形的线，就只考虑填充它们的情况（这样不但得到了一个三角形，而且还是该轮到你填充，相当于白送一个三角形）。这样做似乎有道理，但是反例也容易找到（大家想想看，为什么？）。此法不可行。

第二种方法的失败提醒我们，在敌对搜索中，对于带有强烈主观色彩的剪枝方法要特别小心，不过有趣的是，虽然第二种方法不能保证最优，但是在很多时候的确是很不错的策略，如果以“下一步能得到的三角形数”作为权值，用前面提到的结点排序方法可以大幅度地提高 alpha-beta 剪枝效果（参见表）。

最后，由于状态空间比较小，而一个输入文件中测试点的数目又有很多，因此利用 HASH 表加速的方法会很有用。状态总数为 $2\times4\times2^{18}$ 个(当前游戏者有两种，A 方的三角形个数 4 种情况，线段填充情况 2^{18} 个)，如果采用位压缩方法，需要占用的空间仅为：$2^{16}\times4=256K$，完全可以承受。

下表给出了程序 1（极大极小过程），程序 2（加入 Alpha-Beta 剪枝），程序 3（带结点排序的 Alpha-Beta 剪枝），程序 4（程序 3 加入剪枝），程序 5（程序 4 加入 Hash 表）表 1-23 记录了对于所有测试数据的总运行时间。

表 1-23　三角形大战的运行时间表

程　序	1	2	3	4	5
运行时间	>60	10.33	0.93	0.55	0.16

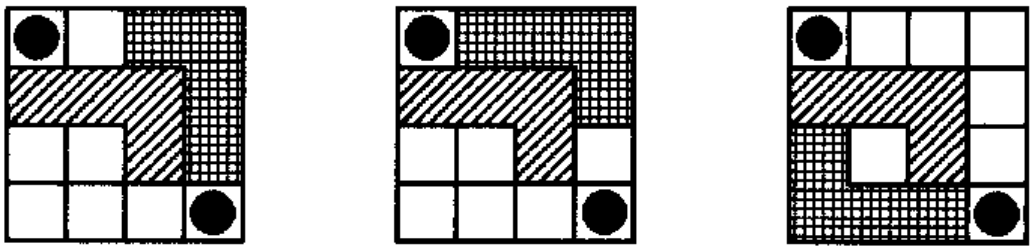
下面，来考虑有和局的情况。如果是终局和局，可以简单地把和局的 f 值设为 0，那么可以继续用刚才的算法，但是如果是因为“无法结束”（循环）而被迫成为和局呢？我们来看一个例子。

【例题 1】L 游戏[1]

L 游戏是 Edward de Bono 设计的一个游戏。游戏规则玩起来，却很考验游戏者的智力。

这是一个在 4×4 棋盘上进行的双人游戏，每人有一张“L 牌”，可以覆盖棋盘上一个 L 型区域。棋盘上还有两个圆形棋子，两个游戏者都可以移动它们。游戏的目的是让对手不能移动他的 L 牌。

从初始状态开始，两个游戏者轮流移动棋子。每进行一次移动，游戏者可以拿起他的 L 牌，旋转或者翻转以后放到棋盘上某个空位，但不能放回移动前的位置。移动 L 牌以后，游戏者可以移动最多一个圆形棋子到棋盘上任何一个空位，也可以不移动。



(a)　　(b)　　(c)

图 1-97　游戏的三个布局

如图 1-97，任一个布局的带格子图案的 L 牌都可以移动到另外两个布局中的位置。图 1-98 描述了一个游戏终止的布局。如果此时该带格子图案的 L 牌移动，该游戏者就输了，因为他无法把他的 L 牌放到一个新的，没有被占据的位置。

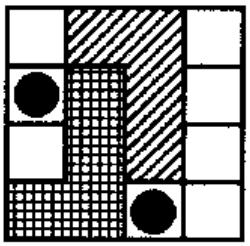


图 1-98　一个游戏结束的状态

给出一个初始状态，判断先手是否有必胜策略，或者是否无法避免败北，或者只能和对手和棋。

循环产生的平局　首先应当注意的是，平局的产生不是因为平局的终止局面，而是因为循环的产生。循环的判断并不是简单地找出局面空间里是否有环，因为有可能可以通过强制措施避免进入一个环内而让某人取胜。例如在上面的图中，只要游戏者愿意，他们可以一直在第一、二幅图之间反复循环。但问题是他们不总是都愿意循环的。如果某人有必胜策略，他一定会按照那个策略走下去，直到胜利。也就是说局面是和局当且仅当两人都没有必胜策略。我们需要对极大极小过程进行修改，避免圈的产生。在 max 局面先考虑是否能取胜，然后判断是否能进入圈，如果都不行就是负局面。需要注意的是，较好的预处

[1] 题目来源：Baltic Olympiad in Informatics, 2002. 命题人：Jimmy Mårdell

理能通过快速计算子局面来大幅度改善程序效率，强烈建议读者用程序来实现这个算法，并尽可能地加快程序地运行速度。

小知识——敌对搜索的 MemSSS*算法

敌对搜索是状态空间搜索中一个很重要的方面，它和前面介绍的问题的显著区别在于它既不是希望得到某两个状态之间的路径，又不对某个满足要求的状态的具体内容感兴趣，而是希望为当前局面找到一个不错的策略。

由于敌对搜索是基于子结点比较的，在解决空间问题的时候，状态空间搜索中常用的迭代加深方法已经不那么实用，相反，在前面很少提到的内存限制方法反而成了一种行之有效的方法。这也提示我们在进行算法类比的时候要注意分析题目自身的特点。

最典型的敌对搜索算法就是刚才介绍的极大极小过程+alpha-beta 剪枝，不难看出它是基于深度优先搜索的。既然有基于深度优先搜索的敌对搜索算法，是否可以把单向搜索中的广度优先搜索也引入到敌对搜索中来呢？

早在 1979 的时候，Stockman 就提出了一种基于最优搜索的算法 SSS*，它总是扩展最“理想”的结点。虽然理论指出 SSS*生成的结点不总比 alhpa-beta 多[1]，但是前面已经提到，最优搜索的最大缺点是它的空间开销（带结点排序的极大极小过程是 $O(bd)$，而最坏情况下 SSS*是 $O(b^d)$的，其中 b 为分支因子，d 为树的深度）。SSS*如果不进行空间上的改进是根本不实用的。需要特别指出的是，基于最优搜索的敌对搜索算法容易判断环的出现，即“平局”。回忆基础篇中介绍的“L 游戏”，由于空间不成问题，因此用 SSS*解决该问题比用极大极小过程+Alpha-beta 剪枝要方便得多，而且还可以成功地解决和局的问题（想一想，应该怎样解决？）。

SSS*的空间问题的根源在于它使用的是最优搜索，很自然地想到借助在最优搜索中常用的空间问题解决方法来改进 SSS*。著名的 A*在实际中被改成 IDA*，那么在 SSS*中可以用迭代加深搜索吗？答案是肯定的，不过 IDSSS*算法的时间效率不高。敌对搜索毕竟和启发式的路径寻找问题有着很大的不同：一个优秀的棋手能够预测几步甚至 10 步以后的情况，虽然他所列入考虑范围之内的结点数和 SSS*相比是微不足道的，但他们总能作出相当不错的决策，即：“无关”结点的减少对最后决策的作出起到的作用是微不足道的！

有的书中介绍过一种“内存限制 A*(SMA*)”算法来缓解 A*的空间危机，它的思想是当空间不足时把 f 值比较大的结点丢掉。但是在很多状态空间巨大，可行解分布稀少而启发式信息不够好的求可行解题目中（例如搬运工问题），SMA*几乎总是以失败告终，因为它总会在某个时候丢弃了可行解。但是通过刚才的分析，“内存限制”的搜索方式对于敌对搜索来说是相当合适的 – 它不可能丢失解（策略总是存在的），而且如果找到了好的估价函数，在很多情况下仍能得到最好的或者稍差一点的解。

基于这个思想，MemSSS*在 1996 年被正式提出，它除了沿用 SMA*的思想，限制生成的结点总数，而且在“忘记结点”之后总是记录下每个结点已经被忘记的最好结点的估

[1] 更精确地，alpha-beta 不可能跳过 SSS*所扩展的任何结点，但是 SSS*算法可能会跳过一些由 alpha-beta 扩展的结点。

价函数值（请大家想想为什么？）。实践证明，和刚才的分析一样，这个方法是非常有效的，尽管它不是总能找到最好的策略。

1.6.5　剪枝

在算法 1.6.1 小节中提到了剪枝。剪枝是一个非常有趣的课题，往往需要发挥自己的创造性和想象力，同时需要有敏锐的观察力和一定的经验。在这里，通过对几个例子的分析来说明一些基本的剪枝方法，总结出一些实用的思考和设计的方法。

一般来说，在剪枝的时候主要思想有三种：**极端法、调整法、数学方法。**

极端法　极端法广泛地应用在各种搜索算法的剪枝中。它的基本思想是通过对当前结点进行理想式扩展，通过否定这样的“理想情况”来避免对当前结点的扩展。

【例题 1】带宽[①]

如图 1-99 所示，给出一个 V 个结点的图 G 和一个结点的排列，用“结点带宽（bandwidth）”来表示某结点和其相临结点排列中的最远距离。

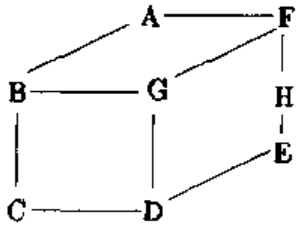
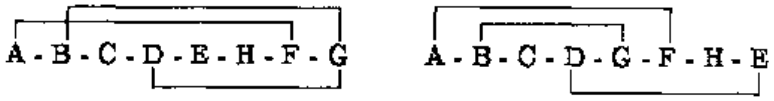


图 1-99　一个网络的例子

例如，在图 1-100(a)中 A 的结点带宽为 6。用整个图的带宽来表示图中所有结点的结点带宽的最大值。例如在图 1-100(a)中，带宽为 6（各个结点的带宽分别为 6,6,1,4,1,1,6,6）；而图 1-100(b)中，带宽为 5（各个结点的带宽为 5,3,1,4,3,5,1,4）。



(a)　　(b)

图 1-100　两种结点排序方案的比较

编程求出任一个带宽最小的结点排列方案。

【分析】

本题最简单的做法是年历回溯法，即从左至右依次确定排列中每个位置的结点。由于 N 个结点排列的个数是 $N!$，枚举量太大，应当进行剪枝。记录下当前已经搜索到的最小带宽 K。首先，如果已经发现某两个结点的距离大于等于 K，该方法不可能比当前解更优，应当剪枝；然后，如果在搜索到结点 U 的时候，U 结点还有 M 个相临点没有确定位置，那

[①] 题目来源：UVA Problem Archive

么对于结点 U 来说，最理想的情况就是这 M 个结点紧跟在 U 后面，这样的结点带宽为 M，而其他任何“非理想情况“的带宽至少为 $M+1$。如果 $M \geq K$，即连理想情况都不能得到当前最优解的答案，显然应当剪枝。

调整法　调整法的基本思路是通过对子树的比较剪掉重复子树和明显不是最有“前途”的子树。

【例题 2】小木棍[1]

乔治有一些同样长的小木棍，他把这些木棍随意地砍成几段，直到每段的长都不超过50。现在，他想把小木棍拼接成原来的样子，但是却忘记了自己最开始有多少根木棍和它们的长度。给出每段小木棍的长度，编程帮他找出原始木棍的最小可能长度。

【分析】

本题的数学模型是：给出 n 个数 a_1，a_2，…，a_n 求出最小的数 k，使得这 n 个数分成若干组，每组的和都为 k。显然，k 必须为 n 个数之和 m 的约数，只需要从小到大枚举 m 的每个约数 t（当然，t 不能小于 n 个数中的任何一个），判断 t 是否可能为初始木棍长度就可以了。

使用深度优先搜索的办法（程序 1）来完成这一步工作。依次搜索每根原始木棍的组成，在每根木棍中依次搜索每一段的长度，为了避免重复搜索，规定必须按照木棍的第一段长度递增的顺序搜索每根木棍，在每根木棍中按照长度递减的顺序考虑每个可能的长度。经过这样的处理（程序 2），对于多数测试数据来说，程序的时间效率已经有了较大的提高，但是对于一些特殊的数据仍然迟迟不能出结果，必须进一步优化。

来考虑这样的情况，如果某长度刚好能够填满一根原始木棍，是否应该考虑其他长度的小棍呢？如果使用更长的原始木棍，长度将会超过 t，显然不行；而如果使用更短的小木棍，即使存在某个也能填满该原始木棍的方法，该方法也一定不会比用大木棍更有希望获得可行解。这样，一旦发现某木棍能够填满该原始木棍，就没有必要考虑其他木棍了，时间效率大大提高，所有数据加起来不到 0.01 秒解出。

数学方法　除了刚才介绍的两种一般方法之外，对于一些具体问题，也可以利用一些专门知识进行剪枝。例如，在图论中借助连通分量[2]，数论中借助模方程的分析等[3]。在下面的个例子中，通过代数中的放缩法进行更精确的下界估计，从而达到较好的剪枝目的。

【例题 3】生日蛋糕[4]

7 月 17 日是 Mr.W 的生日，ACM-THU 为此要制作一个体积为 $N\pi$ 的 M 层生日蛋糕，每层都是一个圆柱体。设从下往上数第 $i(1 \leq i \leq M)$层蛋糕是半径为 Ri，高度为 Hi 的圆柱。当 $i<M$ 时，要求 $Ri>Ri+1$ 且 $Hi>Hi+1$。由于要在蛋糕上抹奶油，为尽可能节约经费，我们希望蛋糕外表面（最下一层的下底面除外）的面积 Q 最小。令 $Q=S\pi$。

请编程对给出的 N 和 M，找出蛋糕的制作方案（适当的 Ri 和 Hi 的值），使 S 最小。

---

[1] 题目来源：ACM/ICPC Regional Contest CERC 1995
[2] 例如求独立集和着色问题
[3] 特别适用于开关切换一类的问题
[4] 题目来源：NOI 99

除 Q 外，以上所有数据皆为正整数，$N \leqslant 10\,000$，$M \leqslant 20$）

【分析】

本题是一道静态规划问题。由于数据规模比较大，使用动态规划的话空间会不足，而且重叠子问题也比较少，因此考虑使用搜索的方法。由于深度已定（M），考虑使用深度优先算法，只在当前体积大于 N，或者当前面积已大于目前最优解面积这一最显然情况下剪枝，才是我们的程序 1（时间效率见后）。注意到本题的关键是两个等式：

$$\text{限制：}\quad N=\sum_{k=1}^{m} R_k^{\,2} h_k \tag{1}$$

$$\text{目标：最小化}\quad S=R_1^{\,2}+\sum_{k=1}^{m} 2R_k h_k \tag{2}$$

考虑用刚刚介绍的极端法剪枝。假设前 i 层的体积为 $T\pi$，如果剩下的几层都取最小可能值，总体积仍然比 N 大，说明前 i 层的方案不可行。同理，如果剩下几层都取最大可能值，总体积仍然比 N 小，也应该剪枝。

对于目标，由于是希望 S 最小化，只能根据估算下界进行剪枝。设前 i 层表面积为 $W\pi$，如果每层面积都去最小值得到的总面积还是不小于 S，就应当剪枝。

程序 2 对于前 5 个测试数据（官方数据）来说已经是比较快的了，但能不能进一步提高算法的时间效率呢？观察能力比较强的读者可能已经看出了，如果将(1)的两边同时除以 $R[i]$，得到的式子和 S 的表达式很接近。把这一点作为使用放缩法的动机，下面就应该考虑如何进行具体实施了。

假设已经确定了前 i 层的体积为 $T\pi$，d 表面积为 $W\pi$，那么剩余 m-i 层的体积满足：

$$N-T=\sum_{k=i+1}^{m} R_k^{\,2} h_k$$

而剩余部分的表面积满足：

$$LeftS=\sum_{k=i+1}^{m} 2R_k h_k=2\sum_{k=i+1}^{m}\frac{R_k^2 h_k}{R_k}\geqslant 2\sum_{k=i+1}^{m}\frac{R_k^2 h_k}{R_i}=\frac{2}{R_i}(N-T)=P$$

显然，如果 $P \geqslant S-W$，根据不等式的传递性，有 $LeftS \geqslant S-W$，即 $W+LeftS \geqslant S$，显然应该剪枝。再以程序 2 的基础放加入本方法得到的程序 3 时间效率进一步提高，不仅数据 1~5 全部在 0.01 秒以内出解，对于规模更大的数据 6~10[1]的运行时间也缩短了不少，如表 1-24。

表 1-24　《生日蛋糕》的运行时间

数　据	1	2	3	4	5	6	7	8	9	10
程序 1	<0.01	145.49	<0.01	0.82	4.73	long	long	long	long	long
程序 2	<0.01	0.44	<0.01	<0.01	0.05	0.71	1.43	4.51	9.51	24.45
程序 3	<0.01	<0.01	<0.01	<0.01	<0.01	0.11	0.27	0.71	1.37	7.14

此法的难点在于对问题的细致分析，实现起来往往比较容易。例如本题，在分析透彻以后，只需要在程序 2 的基础上加入 1 行便可得到程序 3，但是要想到这种方法却不是那

[1] 这些数据是为了更好的区分各种方法而自己设计的数据，它们的规模已经超过了题目的限制。

么容易的。

上面的几种思路太抽象，在很多情况无法直接根据这些思路来直接设计剪枝方法。在平时训练中，应当宏观和微观相结合的思考方法，培养对题目中各个对象的特点进行分析综合的能力，不仅要培养灵感和“题目嗅觉”，找准剪枝的突破口和动机，更要养成细致分析的好习惯。

剪枝控制策略　在设计出一些剪枝算法以后，还需要用实验来检验它是否可行和有效。这涉及到剪枝实验程序的设计和实验结果的分析。经常对各种剪枝法进行实验分析和比较可以极大地提高我们的分析思考能力和灵感。搜索方式要配合剪枝手段才能发挥潜力，下面举一个例子。

【例题 4】汽车问题[①]

有一个人在某个公共汽车站上，从 12:00 到 12:59 观察公共汽车到达本站的情况，该站被多条公共汽车线路所公用，他依次记下公共汽车到达本站的时刻。

- 在 12:00－12:59 期间，同一条线路上的公共汽车以相同的时间间隔到站。
- 时间单位用“分”表示，从 0 到 59 。
- 每条公共汽车线路至少有两辆车到达本站。
- 公共汽车线路数 K 一定≤17，汽车数目 N 一定小于 300。
- 来自不同线路的公共汽车可能在同一时刻到达本站。
- 不同公共汽车线路的车首次到站时间和到站的时间间隔都有可能相同。

请为公共汽车线路编一个调度表，目标是公共汽车线路数目最少的情况下，使公共汽车到达本站的时刻满足输入数据的要求。

两种搜索对象　本题是一道典型的约束满足问题。题目的两个关键对象是“车辆“和“路线”。因此，分别采用基于“车辆”和“路线”的年历回溯（深度优先算法）是有很大差异的。车的特征是时间，而路线的特征是第一辆车和第二辆车。

两种搜索树的形态　如果依次确定每一辆车，按到达时间顺序，依次确定那些没有定线路的车进行，该车属于某新线路的第一辆或者某已有线路的第二辆，可以预见，因为已有线路越来越多，该搜索树越深，枝条就越多；如果依次确定每个路线，需要依次枚举它们的第一辆车和第二辆车。可以预见，因为可以用的车越来越少，该搜索树越深，枝条越少。

对于剪枝来说，对它减去的子树的结点数比较感兴趣。在第一种方法中剪枝往往可以减去大片枝叶，而第二种方法却只能减去小部分（因为剪枝条件往往在深度比较大的情况才能满足）两种方法的时间效率对比如表 1-25 所示。

表 1-25　两种搜索顺序的对比

数　据	1	2	3	4	5	6
方法一	0.01	0.01	0.05	0.11	1.48	1.76
方法二	0.01	0.01	9.07	>100	>100	>100

① 题目来源：IOI 94

剪枝的极限　在求可行解的题目中，如果能准确无误地把所有无解的树枝剪掉，程序的效率将会如何呢？在下面举一个例子中，算法仅扩展了 110%的有用结点，几乎达到了极限。

【例题 5】Betsy 的旅行[1]

一个正方形的小镇被分成 $N \times N$（$2 \leqslant N \leqslant 9$）个小方格，Betsy 要从左上角的方格到达左下角的方格，并且经过每个方格恰好一次。编程对于给定的 N，计算出 Betsy 能采用的所有的旅行路线的数目。

【分析】

算法一：路径枚举

如果直接用 DFS 来解决这个问题，效率极低，连 N=6 的情况都无法很好地解决，所以，剪枝优化势在必行。本题是解的计数问题，但是由于我们关心的解是路径而不是状态，所以很难利用对称性来减小状态空间。

首先从“必要条件”，即合法的解所应当具备的特征的角度分析剪枝的方法，主要有两个方向：

（1）一条合法的路径，除出发点和目标格子外，每一个中间格子都必然有“一进一出”的过程。所以在搜索过程中，必须保证每个尚未经过的格子都与至少两个尚未经过的格子相邻（除非当时 Betsy 就在它旁边）。

（2）一个条件的基础上，还可以从宏观的角度分析，进一步提高剪枝判断的准确性。显然，在一个合法的移动方案的任何时刻，都不可能有孤立的区域存在。虽然孤立区域中的每一个格子也可能都有至少两个相邻的空的格子，但它们作为一个整体，Betsy 已经不能达到。

在具体操作上，对于第一个剪枝条件，可以设一个整型标志数组，分别保存与每个格子相邻的没被经过的格子的数目，Betsy 每次移动到一个新位置，都只会使与之相邻的至多 4 个格子的标志值发生变化，只要检查它们的标志值即可。

对于第二个剪枝条件，处理就稍稍麻烦一些。但仍然可以使用局部分析的方法，即只通过对 Betsy 附近的格子判断，就确定是否应当剪枝，图 1-101 简要说明了剪枝的原理。

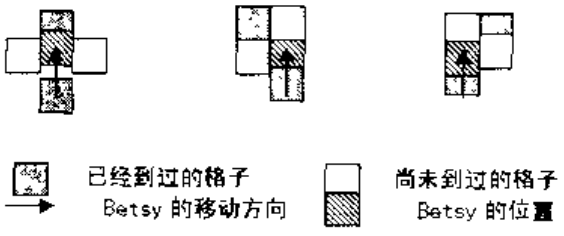


图 1-101　剪枝原理

事实上采用的是枚举法来实现该剪枝方法，程序简洁而高效。

如果大家的观察能力比较强，还可以找到另外一个比较简单不过效果也不大明显的剪枝：如果 Betsy 到达倒数第二行，第二列，而它左边和下边的格子都没有经过，就应该剪

[1] 题目来源：USACO

枝。经过三种优化的程序运行时间见表 1-26。

表 1-26　三种剪枝的比较

N	5	6	7	8	9
答　案	86	1 770	88 418	8 934 966	2 087 813 834
纯 DFS 时间	0.05	14.95	太长	太长	太长
剪枝 1	<0.01	0.11	太长	太长	太长
剪枝 1,2	<0.01	0.05	4.01	496.87	34 小时
剪枝 1,2,3	<0.01	0.05	3.85	471.10	33 小时

算法在 N=9 的时候慢得让人难以忍受！是否应当设计一些更好的剪枝方法来有效的改善程序效率呢？借助实验程序来帮我们的忙。

记录每棵子树得到的可行解，分别记录后继结点中有可行解的结点数 node1 和实际生成的结点数 node2，对于使用了剪枝 1,2,3 的程序 *N*=5,6,7,8 得到的结果如表 1-27 所示。

表 1-27　剪枝效果

N	5	6	7	8
有用结点数	1 335	29 896	1 642 701	168 722 318
扩展结点数	1 363	33 222	1 962 143	213 510 067
无用扩展	2%	11%	19%	27%

扩展的结点比实际必须生成的根本多不了多少。即使把所有不可能产生解的枝条都减去，程序效率也不会有太大改善。要想让程序更快，要么换一种算法，要么在代码优化上下功夫，而剪枝在本题中已经接近极限。

当然，这只是针对搜索而言。聪明的读者已经猜到了本题可以用 1.5 节中介绍的基于状态压缩的动态规划来解决。

算法二：基于状态压缩的动态规划

把问题换一个提法，我们有如图 1-102 所示 6 种格子。

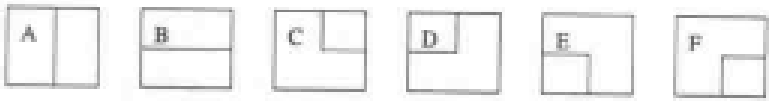


图 1-102　6 种拼块和它们的编号

则问题转化为有多少种方法用它们组成一个条通路？要是通路，必须满足：

- A，E，F 的下面必须是 A，C 或者 D；
- B，D，E 的左边必须是 B，C 或者 F；
- B，C，F 的右边必须是 B，D 或者 E；
- 起点和终点都只能是 A 或 B；
- A，C，D 不能出现在第一行的非起点位置；

- A，E，F 不能出现在最后一行的非终点位置；
- B，D，E 不能出现在第一列的非起点非终点；
- B，C，F 不能出现在最后一列；
- 起点和终点必须连通。

和 1.5 节的《迷宫统计》一样，还是需要记录各个格子的连通性情况，一旦发现有不可能与起点连通的格子要立刻排除掉。虽然有 6 种方格，看起来每行有 6^n 种情况，但实际上由于上面的限制，每一行的状态平均只有约 3^n 种，而且在枚举当前列的情况时，还可以进一步减少枚举量，实际的程序是很快的。

小知识——如何使搜索算法更符合题目特点

对于不同类型的问题，通常采取不同的思考方式，针对同一个问题的不同特点，还可以考虑综合使用多种算法，或者根据这些特点自己设计出新的算法。当在做这一创造性劳动之前，需要首先对一些常见问题的特点加以分析。

首先，应该考虑的是**问题的目标**。问题是路径寻找问题还是约束满足问题或者其他，是求任意可行解，全部可行解（或者可行解计数）还是最优解？如果是求最优解，解的优劣程度和哪些因素有关？例如，代价和和路径长度成正比例的题目可以考虑用广度优先搜索。

然后，是问题的**启发信息**。如果希望使用启发式搜索，必须设计并分析一些可能的启发函数。一般的启发式搜索都要求启发函数具有可接纳性，因此一个常用的设计方法是"放宽对象之间的约束"。或者说，尽量单独地考虑各个对象。例如，数码问题中设计的启发函数实际上就是忽略了数码间的阻碍关系，而博弈树叶结点估值函数则常常忽略很多局面因素而简单地将各个局部估价值加和起来。需要指出的是，并不是所有问题都有比较像样的启发函数的。在尝试多次以后如果仍未找到理想的启发函数，考虑放弃启发式搜索而采用其他方法。除了标准的启发函数，也可以利用题目中不太"规范"的启发信息来改进程序，例如改进结点扩展的顺序。

接着，是**搜索树的形态**。例如解的深度限制，平均分枝因子，各种长度的环的数目，一般来说可以根据这一点来估计剪枝效果。

其次，是**解的分布**，解的分布包括稠密程度和位置两个方面。如果解很少，通常会用一些启发性手段的搜索算法如 IDA*或者深度优先+上下界剪枝，反之，可以用迭代加宽搜索或者启发式修补甚至加入相当随机成分的搜索等比较冒险的方法，利用解的大量分布来获得不错的实际效果。而如果解比较稀少，常常对解的分布很感兴趣。特别是能明显地主观感觉到可能解的特点的题目，往往尝试着利用这些主观感受解题。限制差异搜索和启发式修补等方法就在候选之列。一个重要的理论是"早期错误（Early Mistakes）"理论[1]。

最后，看**解的质量**。如果一个问题对于解的质量要求不是很高，那么可以考虑的算法就要多得多。在很多算法中，可以考虑牺牲解的质量来满足系统的其他限制（如时间，空间限制）。例如，对于代价与路径直接相关的题目，可以牺牲 A*中 f 函数的可接纳性，或

[1] 在这种理论的指导下，人们发明了四种实用性比较强的算法

者在 HPA 算法中增大 $h(n)$的成分，在状态空间的建立过程中，可以通过删除不大可能有用的状态或者算符，用解的质量换取极大的时间效率改善。

练　习　题

1.6.10　算符破译[①]

考古学发现，几千年前古梅文明时期的数学非常地发达，他们懂得多位数的加法和乘法，其表达式和运算规则等都与现在通常所用的方式完全相同（如整数是十进制，左边是高位，最高位不能为零；表达式为中缀运算，先乘后加等），惟一的区别是其符号的写法与现在不同。有充分的证据表明，古梅文明的数学文字一共有 13 个符号，与 0,1,2,3,4,5,6,7,8,9,+, ×,= 这 13 个数字和符号（称为现代算符）一一对应。为了便于标记，我们用 13 个小写英文字母 a,b,…,m 代替这些符号（称为古梅算符）。但是，还没有人知道这些古梅算符和现代算符之间的具体对应关系。

在一个石壁上，考古学家发现了一组用古梅算符表示的等式，根据推断，每行有且仅有一个等号，等号左右两边为运算表达式（只含有数字和符号），并且等号两边的计算结果相等。假设这组等式是成立的，请编程序破译古梅算符和现代算符之间的对应关系。

1.6.11　破坏正方形[②]

有一个火柴棍组成的正方形网格，每条边有 N 根火柴，共 $2n(n+1)$根。从上到下从左到右给各个火柴编号，如图 1-103(a)。现在，拿走一些火柴，问在剩下的火柴中，至少还要拿走多少火柴才能破坏所有正方形？例如在图 1-103(b)中，拿掉 3 根火柴就可以破坏掉仅有的 5 个正方形。

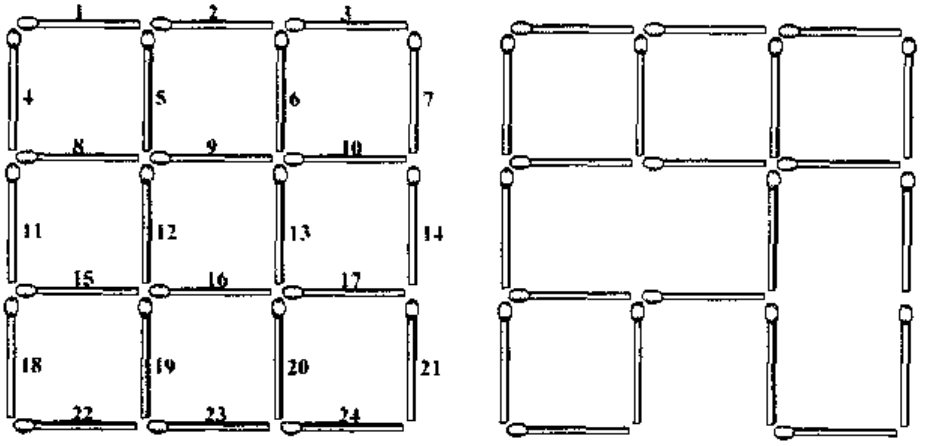


图 1-103　火柴编号和剩余火柴

[①] 题目来源：NOI2000 命题人：李申杰

[②] 题目来源：ACM/ICPC Regional Contest Taejon 2001

1.6.12　列车调度[①]

佳佳是 X 火车站的列车调度员。X 车站很小，其结构如图 1-104 所示。

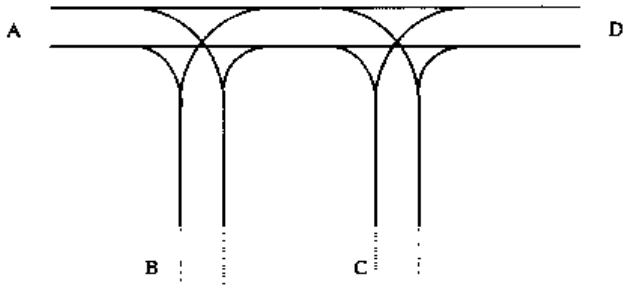


图 1-104　铁轨举例

其中，A 是车站的入口轨道，D 是出口，B 和 C 是两个长度足够的中转轨道（最底端没有出口）。所有的轨道都是单线，即任一时刻，至多只有一辆列车通过，所以，A 端列车只能顺次进站，中转轨道 B 和 C 中，也只有最顶端的列车（如果有的话）可以移动。

调度员只可以执行下述 6 种调度操作（当然，他一次只能移动一辆可以移动的列车）：

1. 让 A 端的进站列车进入 B，记作 A→B；
2. 让 A 端的进站列车进入 C，记作 A→C；
3. 让 A 端的进站列车直接移动到 D 出站，记作 A→D；
4. 让中转轨道 B 中的顶端列车移动到 C 的顶端，记作 B→C；
5. 让中转轨道 B 中的顶端列车移动到 D 出站，记作 B→D；
6. 让中转轨道 C 中的顶端列车移动到 D 出站，记作 C→D。

作为列车调度员，佳佳的工作是这样的：每天早晨，有 N（N≤50）辆列车从 A 端进站（不妨按进站顺序编号为 1…N），然后，他将要按照指令，通过调度操作，使列车按照某个给定的顺序出站。

佳佳怀疑有些命令是否真的能够实现，所以他决定请你编写一个程序，根据给定的出站序列，判断是否存在可行的调度方案，更进一步，如果这样的调度方案存在，请找出其中操作步数最少的一种（如有多种这样的方案，只需找到其中的一种即可）。

*1.6.6　专题：路径寻找问题

很多题目都可以归纳为路径寻找问题。路径寻找问题的状态空间模型在基础篇中已经做了详细叙述。前面所介绍的盲目搜索，启发式搜索算法都可以用来解决这一类问题。另外图论中的最短路 Dijkstra 算法也可以被用到这里，称为**相同代价搜索**（Uniform-cost search）。

人们很早就开始研究路径寻找问题。经典的 8 数码问题、魔方问题和搬运工问题都属于此类型。该类型题目的共同特点是：状态和算符比较直观，然而状态空间往往出奇的大。

[①] 题目来源：IOI1999 中国国家集训队原创试题. 命题人：齐鑫

针对这一特点，在解决路径寻找问题时需要考虑的首要问题就是状态空间的建立与修改。本节针对这一问题提出了建立状态空间的基本方法，并介绍了三种不同的修改方法，分别适应不同的题目类型和要求。需要指出的是，虽然本节内容是由路径寻找问题引出的，但是主要思想同样适用于其他类型的状态空间搜索题目。

在比较典型的路径寻找问题中，我们常常最关心的是状态的表示。状态的表示直接响到算法的空间开销（尤其是哈希表）和很多基本操作的时间效率（例如结点扩展）。因此，在设计状态的表示时要兼顾空间开销和时间效率。

在很多情况下，对于搜索算法的考虑都是源于对建立的状态空间形态等特点的分析。因此建立一个合适的状态空间十分重要。建立状态空间并不困难，但问题在于很多情况下，得到的状态空间是相当大的（有时候是无限大的！），这时需要采取一些策略。下面介绍的三种方法分别采用逐步扩大搜索范围，局部修改状态空间和多次搜索的方法，在一定程度上缓解了状态空间的巨大引起的“危机”。

方法1：逐步扩大搜索范围　其实，有时候可以不通过修改状态空间就能使原来状态空间巨大的问题得到较好地解决。只需要采用迭代搜索，虽然实质上不影响状态空间，但是由于它只是慢慢地扩大搜索的范围，实际上状态空间的有效部分往往会大大减小，从而很大地提高搜索效率。

迭代加宽搜索　在前面，介绍了两种最常用的迭代式搜索：迭代加深搜索（DFID）和IDA*。它们的共同的好处是占用空间少，并且一旦找到解可以保证它最优从而立即终止搜索过程。和迭代加深搜索类似，有一种迭代加宽搜索（Iterative Broadening, IB），第 i 次迭代只允许扩展每个结点前 i 个儿子。不过由于某种原因[1]，这种算法一般只用在可行解相当多，深度限制明显且有一定启发性信息的求任意可行解的题目。但从下面的题目中，大家可以体会到 IB 和 DFID 在思想上的区别和联系，以及迭代搜索算法的普遍“动机”。

【例题1】[2]**外公的难题**

Happy 老爷爷的外孙 Jacky 能写会算，聪明绝顶，老爷爷非常喜欢他。Jacky 最喜欢算 24 点了，可是因为他的知识还不够，他只会做加法，减法和乘法，另外也会使用括号。这次外公给他出了一道难题：从 N 张牌中任选若干张分成 M 堆，使得每堆的各张牌都可以算出 24 点。这本来并不困难，但是外公又要求 M 的值最大，这可难坏了小 Jacky。你愿意帮帮他么？

【分析】

很容易得到一个朴素的回溯搜索算法：按照堆中牌的张数递增的顺序搜索，但是效率很不理想。由于每堆张数的上限没有给出，预处理比较困难，同时导致了状态空间的巨大。显然，对于很多数据来说，基本上不会用到张数很多的堆（注意：这一结论实际上是关于状态空间形态的估计），能不能避免因为过早的时候张数多的堆而推迟了最优解的发现呢（最优解发现得早对剪枝相当有利）？不妨采用迭代加宽搜索。第一次迭代，只允许每堆

[1] 参见 Ginsberg, *Iterative Broadening*, 1990

[2] 题目来源：第四次曙光杯竞赛，命题人：刘汝佳，有修改

最多 2 张，第二次迭代，每堆可以是 2 张或者 3 张……对于大部分数据来说，第三次迭代之前就可以找到最优解了。

迭代搜索的动机　前面已经提到，迭代搜索的基本思想就是逐步扩大状态空间。在状态空间不太大的情况下得到问题的解，这是迭代搜索算法的根本动机。在客观上，迭代搜索往往是把最优化问题转化成了一系列判定性问题。如果各个规模的判定性问题时间效率差别不大，可以使用二分法（例如最大最小匹配），而在搜索中，迭代参数往往对时间效率有很重要的影响，故往往只能进行迭代。需要特别注意的是由于题目类型在很大程度上决定了解的分布情况，完全可以根据题目的特点设计出一些新的迭代搜索算法。后面将要介绍的 LDS 就是一个很好的例子。

15 数码问题　大概最经典的路径寻找问题之一就是 15 数码问题。目前已知最成熟（又容易实现，效率又高）的方法就是前面提到的 IDA*算法。考虑每个数码 $i(1\leqslant i\leqslant 15)$，设它的初始位置是$(x, y)$，目标位置是$(x', y')$，则定义它的离家距离 $dist(i) = |x-x'| + |y-y'|$。所有数码的离家距离和为 TotalDist。显然，每走一步 TtotalDist 最多减 1，因此下界是 TotalDist。事实证明，TotalDist 是一个很好的下界，对于绝大部分有解的问题都能很快找到解。那么无解的数据呢？它们是可以直接判断出来的。把空格看成数码 16，对于任意的布局 S，定义 perm(S)为从上到下从左到右排出 S 的各个元素得到的 1~16 的排列，则 perm(终态)= {1, 2, 3, 4, 5,…,16}。这样，对于任意的 S，设 16 的位置为 i，考虑四种移动方法：

- 空格上移：相当于是元素 i 和元素 $i-4$ 交换。
- 空格下移：相当于是元素 i 和元素 $i+4$ 交换。
- 空格左移：相当于是元素 i 和元素 $i-1$ 交换。
- 空格右移：相当于是元素 i 和元素 $i+1$ 交换。

不管哪种情况，排列 perm(S)的逆序数+dist(16)的奇偶性不变（请读者验证）。进一步有定理：给初态 S，15 数码问题可解当且仅当 perm(S)的逆序数 + dist(16) 为偶数。

方法 2：局部修改法　在很多情况下，使用迭代搜索无法取得满意的效果，尤其是当解特别稀少而分布情况又难以把握的时候（想一想，这种情况下为什么不适合用迭代搜索）。这时候，就应当考虑对它进行局部修改，以达到缩小状态空间的目的。和迭代搜索比较起来，状态空间的局部修改要灵活得多，但也要冒险地多，困难得多。过分谨慎地修改起不到明显的效果，而大刀阔斧的修剪状态空间很可能导致解的质量的大幅度降低，甚至找不到可行解。针对这一情况，尝试着分别分析两种情况下的局部修改方法，一种更加注意保证解的质量，而另一种更加追求修改的成果。

状态修改　在严格要求解的质量的题目，往往在状态上做文章，可以改进状态表示来达到优化状态存储（例如 BFS）和对状态进行操作的时间。最常用的方法之一是利用对称性合并等价状态。例如在 8 数码问题中，可以用整数编码的方法构造哈希表判重。另外忽略掉一些可以用状态中的其他因素确定下来的东西。另外，不储存或生成某些不大可能得到解的状态，也是常用方法之一。在后面的例题中，大家会不止一次地体会到这种优化方法。

状态转移的修改　如果可以适当地牺牲解的质量，本方法往往可以改在状态转移上下

工夫，从而得到有效得多的状态空间修改方法，这个时候，启发信息显得尤其重要。一种常见的方法是“宏”的使用（后面会再做讨论），另一种方法是利用“相关性理论”来控制结点扩展，它的基本思想是：连续两次算符需要“相关”，而不能“没头没脑”的作出决策。当然，需要再强调一次，这两个扩展方法都是基于很强的启发信息之下才有可能实施的，而且具有一定的冒险性：它很可能会牺牲解的质量，甚至丢失解。

方法 3：搜索任务分割法　在更多的实际问题中，迭代搜索和状态空间的局部修改都无法产生不错的效果，而且如果状态空间中包含着概率因素（例如交互式题目中反馈信息就往往有不确定的因素），前面两种方法会受到严重限制。这时，需要对状态空间在整体上进行较大的修改，其中最常用的方法是分割搜索任务，也就是多次搜索

显然，在三种方法中，此法对状态空间的影响最大，但是危险性也最大：它几乎总是不能保证一定能找到最优解甚至可行解。和上一个内容类似，针对两种不同的问题，提出两种不同的多次搜索方法。

极小极小过程　一种方法是“投石问路”，也就是一步一步地做出决策，然后一步一步执行。典型的算法是极小极小过程（Minimin Search）和敌对搜索中的极大极小过程类似，不过每个结点都是 MIN 结点，而叶结点的启发函数也不是敌对搜索中的静态估价函数，而是 A*中的 F 函数。容易看出，只要不断地执行这个过程，就可以像下棋一样一步步走向目标。

Realtime-A*　需要注意的是，有的时候由于深度限制，该过程返回的决策并不太好，导致作出决策并进行下一轮搜索的时候才发现当前的最好选择是“悔棋”。这时候，需要在“悔棋”以后对当前结点的启发函数进行修正（因为正是这个不准确的启发函数导致上一次的错误决定），以便今后再次搜索到该结点的时候能够得到一个更为准确的估价函数。RealTime-A*就是这样一个算法。

容易看出，该算法的好处在于它可以对不完整的状态空间进行一次次的试探性搜索，但在本质上并没有缩小整个状态空间，倒象是迭代搜索一样的逐步扩大状态空间。

分阶段搜索　另一种就是名副其实的分阶段搜索了。需要事先根据经验预定一些子目标，再进行多个小规模的搜索。有的时候，这些小目标具有层次性或者顺序性（例如一些规划问题），有的时候却几乎互不关联（称为孤岛），因此确定合适小目标往往是非常困难的。常常需要对问题本身进行深入的分析才行。

分阶段搜索的危险性　在有的时候，这个方法可以很安全地正确分割状态空间，从而极大地缩小状态空间，但是，在很多情况下，这样的搜索方法有相当的冒险性，但同时也具有不小的智能性和灵活性。事实上，这种方法是把解答树分割成了多棵子树，而一个规模较小或者解和明显的子问题常常可以不用搜索就可以直接求解。

搜两次策略　在实际解题中，用得相对比较多的是“搜两次”策略，特别是在限制满足问题中。先通过粗搜索得到一个比较小的状态空间，再在上面进行第二次搜索。此方法虽然看起来做了一些重复性工作，但是如果第二次搜索中要用到一些比较耗时的基本操作（例如判断解的可行性等），第一次搜索时可以忽略这些操作，而把它们单独组成一次新的搜索，效果还是比较明显的。

通过以上的讲解，应当对状态空间有个更加清醒的认识，而不要局限于传统的思路和方法，培养自己的创造能力，为不同的题目建造合适的状态空间。前面提到的三种方法不是并列的，而是递进的，这一点请读者用心体会。

小知识——如何让算法充分利用资源

在很多情况下，只能够选择一些比较实际的方法，即满足时间空间等系统限制的方法。如果某些矛盾特别突出，往往需要一些特殊的手段来缓解这其中的矛盾。

首先是**时间上的限制**。竞赛题目往往有严格的时间限制，虽然首先应该考虑高效率的算法，但是这样的算法并不总是容易找到的，甚至根本不存在。这时，有时必须牺牲解的最优性来采用一些近似方法，有时候可以根据时间限制输出当前的最优解（俗称“截时”）。

有趣的是，在启发式搜索和敌对搜索中，迭代加深的引入在几乎起着相反的效果。一方面，在IDA*算法中，找到的第一个解就是最优解，因此得到最优解之前截时是得不到任何结果的。而深度优先+上下界剪枝经常能较快地找到一个较优解，并逐步改进解的质量。另一方面，在敌对搜索中，普通的极大极小过程+alpha-beta 剪枝不敢在搜索完毕之前贸然返回当前结果（因为还未搜索到的子树有可能极大地影响到当前的解），而在加入迭代加深策略以后，可以在截时后返回上次迭代的结果。

其次是算法的**空间开销**。由于在竞赛的环境中，计算机的存储量是一定的，因此在设计算法的时候并不是希望占用的空间尽量少，而是主张“物尽其用”，考虑一些空间占用较小的算法，例如深度优先搜索，IDA*等。这时，怎样适当的利用一些额外空间来改善算法的时间效率是一个相当实际的问题。一个典型的应用是结点判重，可以构造哈希表，在DFS 中还可以利用有限状态机来做。另一个应用是存放中间结果，例如前面在双向搜索中介绍的周界搜索。

*1.6.7　约束满足问题

约束满足问题也是一种很常见的典型问题。问题可以抽象为已知一些变量和它们之间在取值上的约束，求一种满足所有约束的取值方法。例如经典的N 皇后问题就属于此类问题。

这类问题虽然也可以用第一类问题的解决方法来做，但是由于有其自身的特点，还有更多的算法。和路径寻找明显不同的是，路径寻找问题往往（显式或隐式地）给出了目标状态，因此往往只对起始结点到目标结点的路径感兴趣；而约束满足问题往往只对目标状态本身的内容感兴趣，而不在乎到达它的具体方法（事实上，在很多约束满足问题中，目标状态本身就反映了搜索路径，例如N 皇后问题）。

由于约束满足问题的目标是满足所有约束，显然“约束”本身的类型和特点很大程度上影响着问题的性质和解法，因此用来解决约束满足问题的算法的基本设计和优化思想都是在“满足条件”上做文章。

年历回溯算法(chronological backtracking)　它的主要思想是依次确定每一个变量的

取值，像深度优先算法一样，如果某变量的所有取值都会违反某约束，就回溯到上一个变量重新取值。需要注意的是，我们对目标状态比较感兴趣，而不是到达它的路径，因此对于一个问题常常有多种建立状态空间的方式。这在后面有更详细的描述。

状态空间的建立　该算法在竞赛中应用相当广泛，但用它求解题目的时候，状态空间的建立往往是一个重要而棘手的问题。和路径寻找问题不一样的是，在建立状态空间的时候，状态的表示并不是最重要的（请读者回忆状态表示在路径寻找问题中的地位），而搜索顺序和结点扩展方式等常常更明显地影响着算法的效率。

约束图和约束传播　一个不错的方法是利用约束图（constraint）进行约束传播（constraint propagation）”。约束图是一个有向图，每个结点代表一个变量，并且记录着该变量当前的可能取值。每次在选取当前要确定的变量之前先通过“约束传播”来去掉不可能的取值，直到无法继续缩小取值范围为止。一般说来，下一步需要选择一个取值范围最小的变量进行试赋值。可以实现在初始化的时候确定变量的搜索顺序，也可以每次动态地选择一个合适的变量进行枚举。

另外，也使用相反的方法，不是动态地删除不符合约束的取值，而是静态地预先计算出一些满足约束的变量组并放在利于检索的数据结构中，用查找代替生成的方法来提高效率。在约束比较强的题目中，这是一个行之有效的方法。

【例题1】篮球冠军赛[①]

这里是BSCB（波罗的海冠军赛）的结果。立陶宛赢得了全部4场比赛，而瑞典却一次也没取胜。详细的统计结果如表1-28。

表1-28　比赛统计结果

名　称	取胜场数	失败场数	得分总数	失分总数
LIT	4	0	344	267
LAT	3	1	368	287
EST	2	2	396	397
FIN	1	3	414	473
SWE	0	4	267	365

（在本题中，球队的名字和它们的顺序不会改变，对于所有测试数据都一样）

给出像上面那样的比赛结果表和所有10场比赛的20个得分Pi（$1 \leqslant Pi \leqslant 200$，所有得分都不相同），编程求出像表1-29这样的各场比赛得分表。

表1-29　各场比赛得分表

	LIT	LAT	EST	FIN	SWE
LIT	—	48	106	120	70
LAT	41	—	111	137	79
EST	102	89	—	123	82

① 题目来源：Baltic Olympiad in Informatics, 1998

续表

	LIT	LAT	EST	FIN	SWE
FIN	66	97	117	—	134
SWE	58	53	63	93	—

(各场比赛情况表，例如，LAT - SWE 的比分是 79 – 53)

注意：20 个比赛得分按照任意次序给出，输入数据保证有惟一解。

【分析】

本题是一道非常典型的限制满足问题。在进行深入讨论之前，先写出一个纯粹的年历回溯算法，按从上到下，从左到右的顺序搜索，每行搜索了 4 个数据以后根据统计表计算出第 5 个数据，每列搜索完 4 行以后计算出第 5 行，每搜索一个元素以后检查当前胜负场数是否超过限制。这样写出来的程序十分简单，但是它的时间效率如何呢？对于比赛中提供的 5 个测试数据，该程序（程序 1）的运行时间如表 1-30。

表 1-30　运行时间表一

数　据	1	2	3	4	5
顺序搜索	3.68	1.32	162.31	40.71	90.22
逆序搜索	25.05	10.00	72.03	38.02	74.45

运行时间很不令人满意。显然，本题给出的限制应当说是比较多的。限制多了，剪枝条件相应的也很强，但同时搜索的顺序等方面也就比较难以把握。怎么安排搜索顺序，使得强的限制条件能较早地剪掉大片无解的枝叶呢？很自然的想到了刚才提到的方法，通过预处理，记录所有满足约束的变量组，把判断改为了检索。对于每个 p，保存相加得到 p 的所有 4 元组[1]（稍微计算可知，这样的 4 元组只有 6 195 个），每次都根据当前选择在表中临时去掉一些不可能使用的 4 元组（例如，其中包含了已经使用过的分数），避免今后检索到实际不可能满足约束的 4 元组。实际上这就是约束传播的方法，改进后的程序时间效率如表 1-31 所示。

表 1-31　运行时间表二

数　据	1	2	3	4	5
程序 2	<0.01	<0.01	0.22	0.16	0.49

当然，这些方法不总是最有效的，常常需要把将要使用的其他优化方法协同考虑，让它们最好的发挥各自的功效。

限制差异搜索（Limited Discrepancy Search, LDS）　它是由 Harvey 和 Ginsberg 于 1995 年提出来的。算法事实上是假设了“算法的决策是基本上正确的，最多犯 K 次错误”。K=0 就是贪心算法。该算法比贪心要好得多，因为它知道按照 0，1，2，…，的顺序迭代差异值 k。如果决策果真“基本正确”，算法会在不大的 k 处终止在一个可行解处。

[1] 在实现的时候，我们实际上是做成不用指针的链表形式

该算法给搜索树的每个结点赋予一个启发函数值（该值不同于A*算法使用的F值。它不一定是可接纳的，甚至不一定是具体的数值，该值只用于比较），然后把每个结点的子结点按照函数值从小到大排序（即：从左到右的第一子结点的启发函数值最大，第二子结点的启发函数值次之……）。由于启发函数的大小会和实际情况有一定偏差，如果偏差不太大，就可以采用迭代差异值的方法按这样搜索：

第一次迭代，差异值为0，即每次都走第一子结点。实际上该算法已经退化为了贪心算法。

第二次迭代，差异值为1，即走一次第二子结点，其余的走第一子结点。

第三次迭代，差异值为2，即走两次第二子结点，其他走第一子结点，或者走一次第三子结点，其他走第一子结点。

一般地，第$i+1$次迭代，差异值为i，设走k步后走的子结点序号之和为m，则$m-k=i$。

这个算法有一些其他的变种，例如，在每次迭代的时候，原始的LDS需要定一个深度限制maxdepth,而许多情况下无法很好地确定深度上界。而LDS的一个变种Depth-Bounded Discrepancy Search, DDS（由Toby Walsh于1997年提出）可以不用深度上界。有兴趣的读者可以参考有关文献。

启发式修补（Heuristic Repair）　它其实并不能保证在有限的时间内得到解（它可能进入死循环！），但是对于有些问题，它可以很快得出解。算法的思想是：从一个非可行解（即违反了一些约束的方案）开始，每次改变一个变量的值，使得改变以后不满足的约束尽量小，也就是贪心的修补当前方案。用该方法可以很出色地解决“N皇后问题”，即：

N皇后问题　在$n \times n$的棋盘上放置N个皇后，使她们互不攻击。两个皇后互不攻击当且仅当她们不在同一行、同一列或同一对角线（不一定是两条主对角线）。

相信读者在学习回溯法的时候曾经解决过8皇后问题。现在N的规模已经大了很多，回溯法已经不能解决这么大的问题（其实N皇后问题有构造方法，这里略去）。它不但时间效率太低，而且效率对于规模相近的数据来说也很不稳定（这和第一个解在解答树中的位置有关）。不过，值得注意的是：n皇后问题的解是有很多的。利用回溯法求得解的总数，n=15的时候有2 279 184个解，n=16时已经有14 772 512个解了。既然解那么多，考虑通过启发式修补的方法（因为有很大的机会通过修补得到解）。

时间效率的影响因素　值得注意的是，影响算法效率的最显著因数是每次修补的时间效率和修补的次数，有两种具体实现，一个尽量减少每次修补的时间，另一个尽量增大每次修补去掉的冲突数。显然，由于当n增大时，可以追求每次修补的效果的实际修补次数不一定能提高多少，但是每次修补的效率会显著降低，当然应当使用第一种算法[①]。算法的运行时间和修补次数见表1-32。

表1-32　n皇后问题运行时间表

N	200	500	1 000	2 000	3 000
修补次数	341	818	1 629	3 206	4 932
参考运行时间	<0.01	0.22	0.99	3.96	9.18

① 在具体实现的时候，我们使用了随机函数，不过由于每次不初始化随机数种子，程序的效率是稳定的

值得说明的是，启发式修补在很多情况下不是作为完整的算法来解决问题的，例如求最大独立集的分支定界法中，用它可以得到一个不错的上界。

小知识——搜索算法和其他方法的结合

搜索算法和其他算法结合的方式主要有两种。一种是**局部搜索**，一种是**搜索修正**。

小范围搜索是将搜索用来解决其他算法的某个关键子问题，这种方法可以借助搜索算法的“通用性”解决小规模问题，保证局部最优。例如当比较复杂的贪心不太好实现的时候，可以把局部搜索的结果作为贪心的决策。

搜索修正方法是借用搜索的框架，把其中一些有“潜力”的结点用其他方法加以改进。

练　习　题

1.6.13　机场跑道[①]

一个矩形机场的平面图可以分为 $M \times N$ 个同样大小的方块（M 表示行，N 表示列，$M \leqslant 10$，$N \leqslant 10$），每个方块可能是以下四种设施之一：

- 起跑线，用**表示；
- 跑道，用&&表示；
- 建筑物，用##表示；
- 停机场，依次编号为 01，02，…，99。

例如，图 1-105 是某个机场(4×5)的平面图：

01	&&	**	&&	02
03	04	&&	05	06
07	08	&&	##	09
10	11	&&	&&	12

图 1-105　机场平面图

飞**机**在机场起落、移动和停泊的规则有：

- 飞**机**只能在起跑线上起落。若机场有多个起跑线，则飞机可在任一起跑线上起落。
- 飞机在起跑线上降落后，要寻找一个空的停机场，并移动到该处停泊，直至其离开机场为止，期间飞机不得离开该停机场。飞机起飞时，要能从该停**机**场移动到某个起跑线。
- 飞机只能在机场内的起跑线、跑道和空的停机场上移动，不能经过建筑物或已有飞机停泊的停机场。
- 由于飞**机**在**机**场移动所花的时间很短，可忽略不计，因此可假设在某个飞机移动

[①] 题目来源：NOI 96

时，其他所有飞机都不动,也无其他飞机起飞或降落。飞机在机场的任一处最多有四种可能的移动方向，即上、下、左、右。

- 每个停机场在同一时间只能停泊一架飞机。
- 每天早晨 5:00 机场开放，晚 22:00 机场关闭，机场开放前和关闭后均无飞机停泊。

已知每天出入机场的飞机共 P 架，有一张这 P 架飞机的日航班表，如表 1-33 所示。

表 1-33　航班表

飞机编号	到达时间	起飞时间
01	05:00	13:00
05	08:00	16:00
13	07:00	21:00
04	09:00	09:30
39	12:50	13:50

为了使这 P 架飞机能顺利出入机场，需要寻找一种满足要求的飞机停泊调度方案（即：每架飞机降落后均能移动到某个空的停机场，起飞均能从停机场移动到某个起跑线）。

注意：航班表中任意两个时间均不相同。

对于任意一个给定的飞机场平面图，根据给定的航班表（P≤30），寻找一种满足要求的调度方案。

1.6.14　采矿[1]

一架人类的航天飞机将人类基地建在了一个荒芜的星球上。面对紧缺的能源，他们只能在最短的时间内用 SCV（一种智能机器人）采集必须的矿藏。对于这个艰巨的任务，他们希望得到编程高手们的帮助。

在这个星球上，有着两种不同的矿。一种被称为“冰矿”，是一种类似 H_2O 的凝固物的蓝色高能矿藏。另一种被称为“气矿”，是四氯化碳的一种异态形式。人类通过这两种矿的提炼，获得可供生存的能源。

SCV 是一种惟一可以采集这两种矿的智能机器人。他们每采集一次冰矿需要花费 t_1 的时间，每采集一次气矿需要花费 t_2 的时间。采集结束后，将得到 8 个冰矿或者 8 个气矿单位。每一次 SCV 只能采集冰矿或者是气矿中的一种。

SCV 可以通过主基地制造。每制造一个 SCV，主基地将花费 50 单位的冰矿。而主基地由于制造能力有限，在同一时间只能制造一个 SCV。制造一个 SCV 需要 t_3 的时间。

在开始时，人类拥有 50 个单位的冰矿和 4 个 SCV。在最短的时间之内，他们需要至少采集到 p_1 单位的冰矿和 p_2 单位的气矿。请计算出他们需要的最短时间。

1.6.15　灌溉[2]

农田是一个 $n \times m$（$n,m \leq 30$）的矩阵，而你将负责为它设计一套灌溉系统。农田的 k（$k \leq 200$）个小方格内是喷水龙头，可以朝东西南北四个方向直线喷洒非负整数距离的水

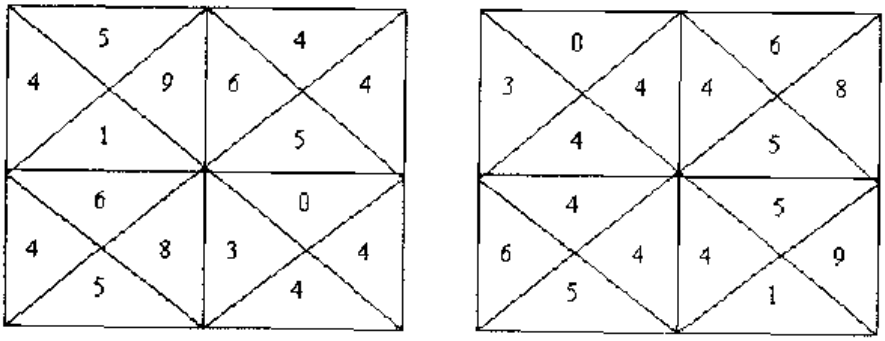
[1] 题目来源：CTSC 2000. 命题人：钱文杰

[2] 题目来源：南斯拉夫冬令营试题 2001

即灌溉那里的小方格（四个方向的喷洒距离可以不同）。为避免渴死或淹坏庄稼，同一小方格要么是喷水龙头安放处，要么恰好属于一个喷水龙头的灌溉范围。灌溉一个小方格需要一个单位的水量，而每个喷水龙头有各自不同的水量。为避免浪费，每个喷水龙头的水必须恰好用完。现在给你农田的大小和每个喷水龙头的位置及水量，请你设计一整套灌溉系统。

1.6.16　Gnome tetravex[①]

哈特近来一直在玩有趣的 Gnome Tetravex 游戏。在游戏开始时，玩家会得到 $n \times n$（$n \leqslant 5$）个正方形。每个正方形都被分为 4 个标有数字的三角形（数字的范围是 0 到 9）。这四个三角形分别被叫做“左三角形”、“右三角形”、“上三角形”和“下三角形”。例如，图 1-106(a)是 2×2 的正方形的一个初始状态。



(a)　　(b)

图 1-106　初始状态和结束状态举例

玩家需要把正方形移动到结束状态。在结束状态中，任何两个相邻的三角形上的数字都应相同。图 1-106(b)是一个结束状态的例子。

看起来这个游戏并不很难。但是说实话，哈特并不擅长这种游戏。他能成功地完成最简单的游戏，但是当他面对一个更复杂的游戏时，他根本无法找到解法。

某一天，当哈特在玩一个非常复杂的游戏的时候，他大喊道：“电脑在耍我！不可能解出这个游戏。”对于这样可怜的玩家，帮助他的最好方法告诉他游戏是否有解。如果他知道游戏是无解的，他就不必再把如此多的时间浪费在它上面了。

1.6.17　智慧方块[②]

Vexed 游戏（智慧方块）是 James McCombe 发明的一种类似俄罗斯方块的游戏。在游戏中，一面木头墙上放置了一些标有字母的白色石块。

如果某个石块的左边（或者右边）是空的，那么该石块可以向左（或者向右）移动一步，木头墙永远不能移动，悬空的石块会自动下掉。每个石块都有一个标记，两个或更多具有相同标记的石块相碰时会形成石块群，石块群会自动消失。如果同时形成了多个石块群，那么它们会一起同时消失。石块群消失后，悬空的石块同样会自动下掉。下掉后石块群同样会消失……如此循环，直到石块不再变化为止。游戏的目标就是让所有的石块消失。

题目来源：ACM/ICPC Regional Contest Shanghai, 2001

题目来源：ACM/ICPC World Finals 2001

图 1-107(A)→(H)就是个从初始状态到石块全部消失的游戏过程：首先上面的“Y”石块左移，这样两个“Y”石块形成石块群自动消失；然后上面的“X”石块右移，右移后下掉，这样两个“X”石块也形成石块群自动消失。

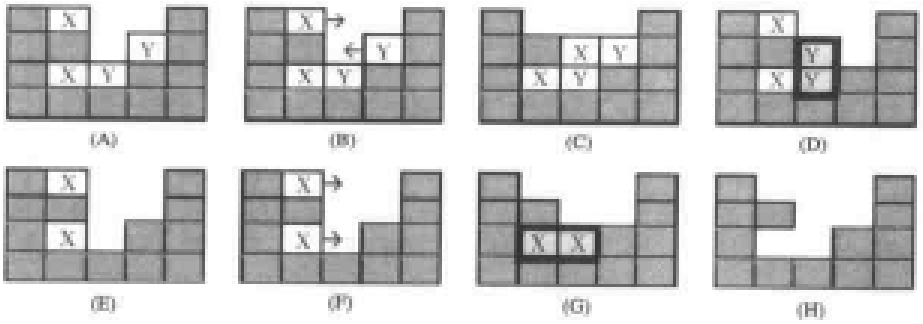


图 1-107　一个完整的游戏过程

图 1-108 是另一个游戏过程：首先最左边的“Z”右移（图 A），形成“X”、“Z”石块群（图 B）；石块群消失后，又形成“Y”、“Z”、“X”石块群；石块群消失后，最后形成“X”石块群，石块群消失后游戏结束。

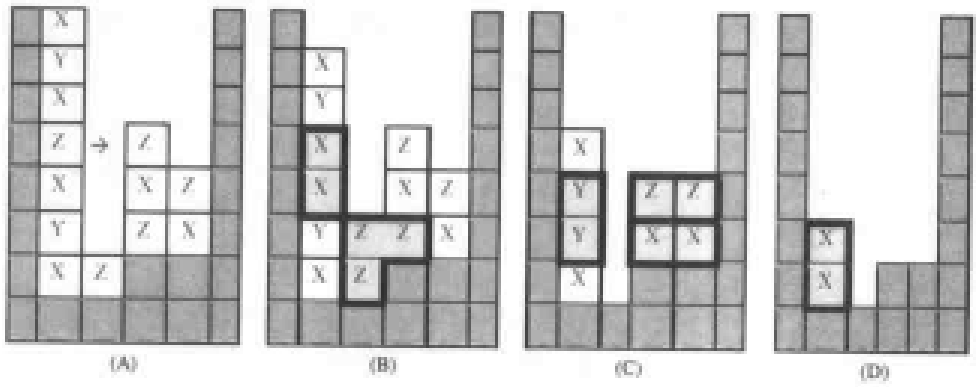


图 1-108　另一个游戏过程

写一个程序，任意给出一个游戏，给出一个解。

1.6.18　最优逻辑电路设计[1]

W 教授在 T 大学计算机系里开了一门“数字逻辑”课，主要讲授如何设计逻辑电路。这一天，W 教授布置了一个实验：设计并实现一个 4 端输入、4 端输出的逻辑译码电路。设计这样的电路原本并不困难，但是，教授给出了如下的要求：

（1）只允许使用 2 端输入、1 端输出的门电路作为实现电路的组件，而且可用门电路的种类和数目都已给定；

（2）使用最少数目的门电路。

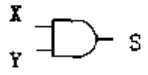
这两个要求难倒了全系的同学，于是，Q 同学找到了正在参加 CTSC（中国队选拔赛）的你，希望你能帮忙编写一个程序，自动找出符合要求的连接方式。

在数字逻辑中，所有信号都可以看作只有两个值“高电平”和“低电平”，分别用“1”

[1] 题目来源：CTSC 2001. 命题人：齐鑫

和“0”来表示。

一个门电路元件的特性由其输入/输出功能表惟一给出，所谓功能表，就是输入信号电平与输出信号电平之间的关系表。比如，“与门”的符号和功能表如图 1-109 所示。



X	Y	S
0	0	0
1	0	0
0	1	0
1	1	1

图 1-109　与门符号和功能表

图 1-109 中，如果“与门”的两个输入端 X 和 Y 都是高电平“1”，则输出端 S 也是高电平“1”，否则，输出端 S 是低电平“0”。

假定，本次实验提供的门电路都具有输入对称性，即交换两个输入端的信号，输出不变。但是，如果门电路的输入端悬空（即没有加输入信号），则输出无意义。

在连接电路的过程中，一个门电路的输出端可以将信号送到其他多个元件的输入端；而门电路的一个输入端则只能接收来自一个输出端的信号。如图 1-110 所示。

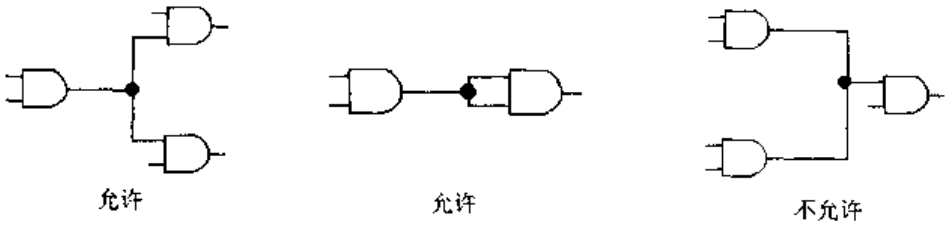
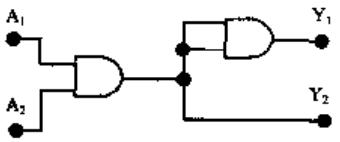


图 1-110　允许和不允许的情况

另外，规定信号必须单向传输，即一个门电路的输出不能直接或间接通过其他门电路回到同一门电路的输入端。

要求你设计的译码电路是一个有四个输入端和四个输出端的逻辑电路，该译码电路的输入和输出关系通过功能表给出，即给出每种输入组合下的四个输出端的情况。显然，一共有 2^4=16 种输入组合。比如，一个由前述“与门”构成的 2 输入、2 输出的简单译码电路如图 1-111 所示（其中，A_1，A_2 是输入端，Y_1，Y_2 是输出端）。



A_1	A_2	Y_1	Y_2
0	0	0	0
1	0	0	0
0	1	0	0
1	1	1	1

图 1-111　一个成功设计的例子

1.6.19　加密[①]

有一个加密方法是把每个英文字母映变换成不同的英文字母，给出 26 个字母的一个排列 C_1,C_2,C_3，则把加密的时候应该把字母表中第 I 个字母变换成字母 C_i，空格不变。例如对于排列 ZYXWVUTSRQPONMLKJIHGFEDCBA，文本 THIS IS THE FIRST SAMPLE 将变成 GSRH RH GSV URIHG HZNKOV。该加密方法十分不保险，因为在得到了密文以后，即使没有对应关系表，也能根据英语中可能有的词汇猜出原文。编程完成这项解密工作。

输入文件 dict.txt 第一行为一个整数 D（$1 \leqslant d \leqslant 50\ 000$），为可能出现的单词数目。以下 D 行每行一个单词，每个单词最多 20 个字符，所有单词长度和不大于 350 000。

文件 puzzle.in 第一行为密文的单词数目 T，以下 T 行，每行一个单词。

输出如果对应表不惟一（忽略原文没有出现的字母）或者无解，输出-1，否则为一个长度为 26 的字符串，即对应表。原文中没有出现的字母处打“*”。

1.6.20　马戏团入场券[②]

Farmer John 要带 16 只母牛去看杂技，母牛们将坐在观众前附近的一个 4×4 座的牛区。所有座位、行、列如图 1-112 标上数字。

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

图 1-112　一个 4×4 网格

母牛们随机坐在位置上，等它们坐下之后才进行检票。很明显，它们没有必要坐在合适的位置上，位置已被设计成可以左右转一行或上下转一列。图 1-113 的例子说明了由图 1-112 的状态经过 4 种转动操作所到达的目标状态。

4	1	2	3
5	6	7	8
9	10	11	12
13	14	15	16
右转第一行			

2	3	4	1
5	6	7	8
9	10	11	12
13	14	15	16
左转第一行			

5	2	3	4
9	6	7	8
13	10	11	12
1	14	15	16
上转第一列			

13	2	3	4
1	6	7	8
5	10	11	12
9	14	15	16
下转第一列			

图 1-113　四种操作的结果

给定牛的原始位置，用较少的旋转次数将牛重排，使得他们的入场券与座位号相符。

无论原始位置如何，总有至少一种可行方案。

你的每个测试点的成绩取决于你的方案与所有选手的最优方案的接近程度。你的输出序列必须少于 500 个操作。

---

① 题目来源：ACM/ICPC Regional Contest Tehran 2001.

② 题目来源：USACO US Open 2002

提示：序列 1l 1l 1l 4u 1r 4d 1l 1l 4u 1r 4u 4u 4u 交换了左上角的牛与它右边的牛。

1.6.21　清洁公司[①]

有一个大厅最近要召开一个很重要的会议，因此会议的组织者请了一个清洁公司来给大厅做清洗。大厅是一个 $N\times M$ 的网格区域（N，$M\leqslant 500$），其中有些格子是障碍物。

清洁公司使用一种自动清洗机来完成清洁工作。机器是一个边长为 L 的正方形（$2\leqslant L\leqslant 100$），总覆盖着边长为 L 个格子的正方形网格区域。机器每步可以沿着东、南、西、北四个方向之一移动一个单位长度，凡是被机器覆盖过的区域均被认为是清理过了。

图 1-114 就是一个大厅的例子，白色的格子代表地板，黑色的格子代表障碍物，灰色的格子代表清洗机。

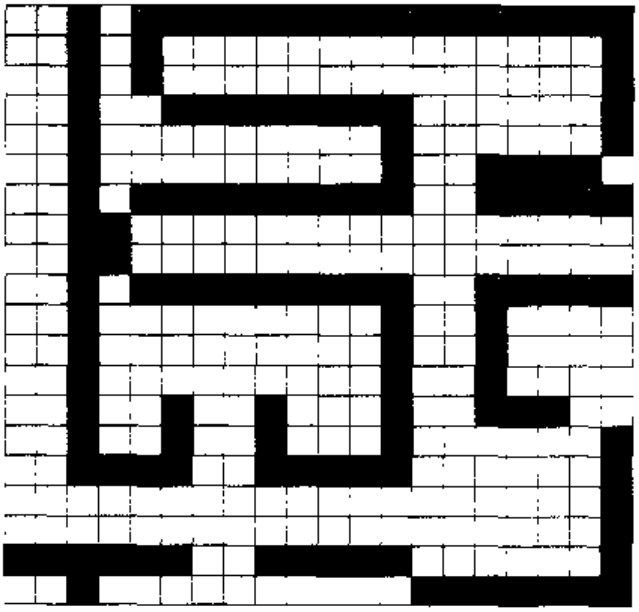


图 1-114　大厅举例

可惜机器的速度太慢，只能用它移动 H 步（$50\leqslant H\leqslant 10\,000$）。怎样才能在规定的步数里清理尽可能大的面积呢？

[①] 题目来源：保加利亚 PC Magazine Contest

第2章　数学方法与常见模型

数学方法与常见模型是信息学竞赛不可忽视的基础，选手必须掌握。本章的理论性比较强，只有学好了它们，才能站在一个更高的角度去看待更多的信息学问题。

本章内容简介

数学非常重要，但是它们的理论部分比较难懂。为了让读者容易接受又能学到有用的东西，本章去掉了一些几乎每本相关教材都要介绍的“必讲内容”，而突出重点，通过例题让读者不仅知道“怎么做”，还知道“为什么要这样做”。

2.1 节概括地介绍了一些代数方法。本节内容比较简单，理论性也不强，主要目的是拓宽读者的思路。

2.2 节介绍初等数论。本节理论和实际结合得比较紧密，理论部分不是很难，但例题比较有启发性，建议读者认真学习。

2.3 节介绍了组合数学的基础知识。本节重点在计数方法，包括基本排列组合、容斥原理、Pólya 定理、递推关系与母函数、离散变换与反演等内容。

2.4 节和 2.5 节都是图论的内容，但是侧重点不同。2.4 节主要是一些概念和基本问题，重点介绍了图论的基础知识，理论性要强一些，也有一些例子帮助读者理解。这些知识包括连通性、着色问题、独立集与团、可行遍性问题、平面图等。

2.5 节介绍了图论的基本问题和算法。这些内容本身是非常重要的，而且这些算法所反映的算法设计思想也是很有借鉴意义的。在本节中，我们把重点放在了算法设计与分析上，这些内容包括最小生成树、最短路径问题、二分图最大匹配和网络流问题。

由于数学是为算法设计服务的，因此这里只介绍了竞赛中常用的东西，而略去了很多理论问题，有兴趣的读者可以参考相关教科书。和第 1 章一样，强烈建议读者在学习本节之前先简单地阅读一下组合数学和图论的其他书籍，这样，在学习本章的时候才能把注意力集中在一些关键概念和方法的深入理解上，体会到其中的精髓。

本章的理论性比第 1 章强很多，但同样又是逻辑性很强的，因此我们试图用形式化的语言把这些东西说得严谨些，同时采用通俗的语言来阐明这些逻辑关系。

2.1　代数方法和模型

本节的主要内容是代数方法和模型。本节的理论知识很少，需要读者掌握初等代数和

简单的线性代数知识。本节的内容比较简单，但是有启发性，突出了代数式、分段函数、向量、复数、矩阵等基本工具的应用，希望读者仔细阅读思考。

字母代替数的好处　在下一个例题中，将用字母代替数进行处理。它的好处是一般化，而且式子的性质有时候更容易启发我们往正确的方向思考。

【例题 1】“麻烦”子[1]

随着物理学的发展，科学家们发现了越来越多的粒子。或许有一天，物理学家们会发现一种“麻烦”子，因为它们必须成对地产生或者消失。

需要设计一个实验消灭所有的“麻烦”子。实验装置是一个有 8 个照相机的立方体，每个照相机占据一个顶点，编号为 $A, B, C, \cdots, H$，如图 2-1 所示。每个照相机一开始都有一定数量的“麻烦”子，不同的照相机里初始的“麻烦”子数量可能不同。

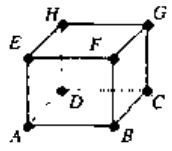


图 2-1　立方体实验装置

每次可以选择两个相邻顶点，让它们中同时产生或者消失一个“麻烦”子。例如，如果 8 个照相机初始有 1,0,1,0,3,1,0,0，那么一个合法的操作序列是“EF-, E-, AD+, AE-, DC-”。

【分析】

这是一个数学味很浓的题目。首先，可以把这个立方体的 8 个顶点染成红色和蓝色，要求任意两个相邻顶点颜色不同。不难发现，每次操作必然是同时改变红色顶点和蓝色顶点中的粒子数量。因此，假如一开始红色顶点中的粒子总数跟蓝色顶点中的不等，问题一定无解。

为了方便说明，定义一种“输送”的操作。假设立方体某一个面的 4 个顶点顺次为 A,B,C,D，其中包含粒子数分别为 a,b,c,d。可以先让 A,B 同时增加 c 个粒子，此时 4 个点中的粒子总数为 $a+c,b+c,c,d$。然后将 B,C 同时减少 c 个粒子，此时粒子分布成为：$a+c,b,0,d$。这个操作等效于把 C 中的粒子移动到 A 中（注意 C 和 A 是同色的）。

利用上面的操作，可以把所有红色顶点中的粒子输送到一个红色顶点 P 中，把所有蓝色顶点中的粒子输送到一个和 P 相邻的蓝色顶点 Q 中。由于红、蓝顶点中粒子总数相等，因此此时 P,Q 中粒子总数相等。最后抵消 P,Q 中所有粒子，就完成了题目要求的操作。

分类讨论　我们经常需要讨论一些复杂的问题，直接建立代数模型比较困难，而分类讨论就要容易得多。下面就是一个不错的例子。

【例题 2】沙漠[2]

佳佳被派遣去做一次沙漠旅行，他将要从司令总部出发，直线走 L 米。此时，佳佳正在研究军事地图：地图上司令总部位于坐标(0,0)，地图还标有 N（$N \leqslant 500$）个四角坐标为正整数且边平行于 x, y 轴的矩形区域，矩形区域相互不覆盖，且不同的矩形有不同的行进速度，也就是说佳佳穿过某矩形的时间等于他在该矩形内的路程长度除以该矩阵的行进速度。沙漠其他地区也有一个固定的行进速度。现在给 L 的值、地图描述和所有矩阵及沙漠其他地区的行进速度，要求出佳佳最短的旅行时间。注意，L 值很大，足以使路程的终止

[1] 题目来源：Ural State University Problem Archive
[2] 题目来源：Ural State Univeristy Problem Archive

地点总不在任何矩阵内。

【分析】

因为“路程的终止地点总不在任何矩阵内”，所以直线穿越沙漠无非有如图 2-2 所示的 4 种所示情况。

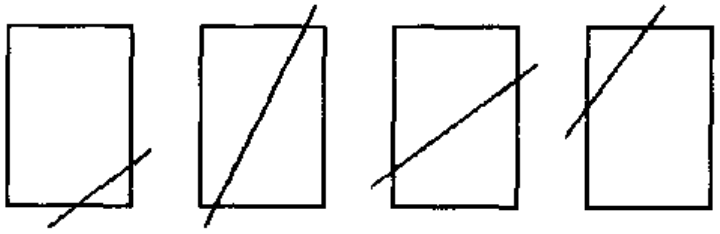


图 2-2　穿越单个矩形区域的 4 种情况

假设直线斜率为 k，矩形左下角坐标（x_1,y_1），右上角坐标（x_2,y_2），则 4 种情况中直线穿越矩阵的长度分别为：

（1）$(x_2+(y_1-2x_2y_1)/k)\sqrt{1+k^2}$

（2）$((y_2-y_1)/k)\sqrt{1+k^2}$

（3）$(x_2-x_1)\sqrt{1+k^2}$

（4）$(x_1+(y_2-2x_1y_2)/k)\sqrt{1+k^2}$

它们都可以表示成$(a+b/k)\sqrt{1+k^2}$的形式。易知，只要 k 在不跨越任何一个矩阵顶点的范围内（称为一个单位区间）变化，则总时间表达式的系数不会改变。求该表达式的极值，就可以求出该范围内时间的最小值。最后，枚举计算直线通过所有单位区间旅行时间，其中最优的为所求。

复数计算　在处理旋转和伸缩问题的时候，复数往往可以给计算带来方便，下面就举一个例子。

【例题 3】浪人苏比[①]

苏比“居住”在 X 镇。X 镇的两条主街道 M 和 N 在镇中心 O 处相交成 a 角，每条街道的一侧有连续的石椅，供行人歇息。因为该地靠近赤道，终年高温多雨，考虑到行人避雨的需要，政府在石椅上方加建了连续的雨篷，这样一来，石椅却成了 X 镇一些流浪汉的最佳去处。苏比就是其中之一。

泰德先生在对苏比进行了持续的跟踪观察后，发现苏比的流浪过程极有规律，如图 2-3 所示。

- 苏比不在同一主街道上连续休息两次。
- 苏比在同一街道上的连续两次休息的石椅必不相同。
- 苏比在连续的两次的歇脚处之间的行程是一定的，为 10。

当泰德先生开始观察时，苏比位于 M 街道上的 $A(c,0)$处，他的第 1 个落脚点位于他所能到达的 N 街道的最北处 B，其后各落脚点分别是 $C,D,E,\cdots$

① 题目来源：IOI1999 中国国家集训队原创试题. 命题人：夏方方

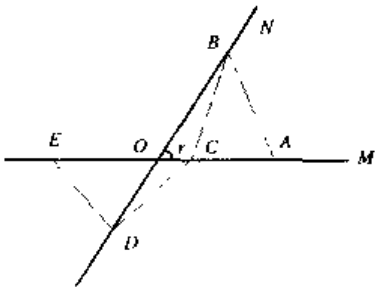


图 2-3　X 镇的两条主街道和苏比的流浪路线

你的任务是帮助泰德先生计算出苏比第 $2n$ 个落脚点的横坐标值（向右为正）。

【分析】

观察苏比移动的规律后可以发现：

- 当 k 为奇数时，第 $k+2$ 次移动的方向可由第 k 次移动方向逆时针旋转 $2r$ 角得到。
- 当 k 为偶数时，第 $k+2$ 次移动的方向可由第 k 次移动方向顺时针旋转 $2r$ 角得到。

如果用向量（复数）来表示各次移动，将使计算变得简单。

设 $w_1=(\cos 2r)+(\sin 2r)\mathrm{i}$; $w_2=(\cos(-2r))+(\sin(-2r))\mathrm{i}$

Z_k=第 k 次移动所对应的向量($k\leqslant 2n$), Z_0= 初始位置对应的向量 c, Z =最终位置对应的向量，则

$$Z=Z_0+Z_1+Z_2+\cdots+Z_{2n}=Z_0+(Z_1+Z_3+\cdots+Z_{2n-1})+(Z_2+Z_4+\cdots+Z_{2n})$$
$$=Z_0+Z_1(1+w_1+w_1^2+\cdots+w_1^{n-1})+Z_2(1+w_2+w_2^2+\cdots+w_2^{n-1})$$
$$=Z_0+Z_1\times(1-w_1^n)/(1-w_1)+Z_2\times(1-w_2^n)/(1-w_2)$$

经过以上推导，只需进行简单的复数运算即可得到最终位置 z。

考虑到 n 值可能很大，单纯按照上述公式计算可能造成误差的大量积累，应将 n 缩小至等效的最小情况。事实上，因为 w_1,w_2 是单位根，苏比的流浪过程是以 $2a$ 次为周期循环的。所以等效 $n=(n \bmod 2a)$。

向量　和复数类似，向量在处理旋转伸缩时有着重要的作用，下面也举一个例子说明，它是有启发性的。

【例题 4】好动的佳佳[1]

佳佳是个静不下来的孩子，他总喜欢到处乱跑。奶奶知道佳佳会依次沿着向量 a_1，a_2，…，a_n 跑，不过每次奶奶可以选择是让佳佳“正着跑”还是“反着跑”，借此来控制佳佳的路线。例如，如果佳佳在点(1,2)，向量为(5,-4)，那么“正着跑”会到达(6,-2)，而“反着跑”会到达(-4,6)。

佳佳不愿意老沿着同一个方向一直跑，因此每个向量的长度都不超过 L。奶奶希望佳佳最后离起点的距离不要超过 $\sqrt{2}L$，她应该让佳佳哪几次“反着跑”呢？

【分析】

可以证明这个题是一定有解的，因为对于任意两个模不超过 L 的向量，它们和的模是不可能超过 $\sqrt{2}L$ 的。下面讨论 3 个或 3 个以上向量的情况。从所有 n 个向量中任意选取 3

[1] 题目来源：Ural State University Problem Archive

个，这3个向量有两种可能情况。

情况一：存在两个向量p,q，它们的夹角大于等于120度。在这种情况下，直接把p,q求和，得到的结果的模必然小于等于L；

情况二：存在两个向量p,q，它们的夹角小于60度。在这种情况下，把p和$-q$求和，得到的结果的模必然小于等于L。

每执行一次上面的操作，就可以把向量的总数减少一个，只需要把向量减少到两个，然后直接将这两个向量求和即可。

矩阵　矩阵是一个强有力的工具，两个$n\times n$矩阵相等相当于$n\times n$个标量等式。用矩阵描述一个复杂物体是非常紧凑的，从下面一个例题就可以看出。

【例题5】细菌[1]

N（$N\leqslant 100$）个培养皿排成一个圈，每个里面都有一些细菌。培养皿逆时针编号为$0,1,2,\cdots,n-1$，第i个培养皿里有b_i个细菌。细菌不停地活动着，它们可能会进行6种动作：

- die i 0，表示第i个培养皿的所有细菌都死亡，即$b_i \leftarrow 0$；
- reproduce i k，表示第i个培养皿每个细菌分裂成k个，即$b_i \leftarrow b_i \times k$，$k\leqslant 100$；
- copy i j，表示把第j个培养皿的所有细菌复制到第i个培养皿，即$b_i \leftarrow b_i + b_j$；
- teleport i j，表示把第j个培养皿的所有细菌转移到第i个培养皿，即$b_i \leftarrow b_i + b_j$，然后$b_j \leftarrow 0$；
- swap i j，表示交换第i和第j两个培养皿的细菌；
- merry-go-round 0 0，表示每个培养皿的细菌都同时转移到它逆时针的下一个培养皿。

一旦某个培养皿里有超过k个细菌，每k个细菌会合在一起进化成一个高级组织而脱离培养皿，即如果$b_i>k$，则$b_i \leftarrow b_i \bmod k$。这些细菌重复的执行着$M$（$M\leqslant 20$）条指令（编号为$0,1,\cdots,M-1$），即第$S$时刻执行第（$S-1$）mod M条指令。问在第T（$T\leqslant 10^9$）时刻执行指令后，每个培养皿各有多少个细菌？已知初始的时候每个培养皿恰好有一个细菌。

这里是一个例子：$n=8, m=6, t=11$，6条指令如下：

reproduce 2 5

copy 4 2

die 1 0

merry-go-round 0 0

teleport 5 3

swap 0 2

则每个时刻结束后各培养皿的细菌数目如表2-1所示。

【分析】

可以用模拟的方法来解决，每个时间单位可以用最多$O(N)$的时间来计算，所以总的时间复杂度为$O(NT)$。有没有快一些的方法呢？当然有。令当前各个培养皿内细菌的数目为向量b，则可以把任何一种操作看成一个矩阵A，使得操作以后的培养皿内数目为bA。

[1] 题目来源：Internet Problem Solving Contest 2003

表 2-1　各时刻结束时的培养皿情况表

Second	Change	Colonies							
		0th	1st	2nd	3rd	4th	5th	6th	7th
beginning		1	1	1	1	1	1	1	1
1st	reproduce 2 5	1	1	5	1	1	1	1	1
2nd	copy 4 2	1	1	5	1	6	1	1	1
3rd	die 1 0	1	0	5	1	6	1	1	1
4th	merry-go-round 0 0	1	1	0	5	1	6	1	1
5th	teleport 5 3	1	1	0	0	1	4	1	1
6th	swap 0 2	0	1	1	0	1	4	1	1
7th	reproduce 2 5	0	1	5	0	1	4	1	1
8th	copy 4 2	0	1	5	0	6	4	1	1
9th	die 1 0	0	0	5	0	6	4	1	1
10th	merry-go-round 0 0	1	0	0	5	0	6	4	1
11th	teleport 5 3	1	0	0	0	0	4	4	1

- die i 0 操作：把单位矩阵 i 的(i,i)置为 0 即可。
- reproduce i k 操作：把单位矩阵的(i,i)置为 k 即可。
- copy i j 操作：把单位矩阵的(j,i)置为 1 即可。
- teleport i j 操作：把单位矩阵的(j,i)置为 1，再把(j,j)置为 0 即可。
- swap i j 操作：把单位矩阵的第 i 行和 j 行交换。
- merry-go-round 0 0 操作：把单位矩阵的第一行移动到最后一行的后面。

这样，设第 i 条指令对应的矩阵为 X_i，则本题是要求 $A=(1,1,1,\cdots,1)\times X_1\times X_2\times\cdots\times X_T$

由于 $X_1=X_{m+1}=X_{2m+1}\cdots$而矩阵乘法是满足结合率的，因此如果令 $X=X_1\times X_2\times\cdots\times X_m$，$T \bmod m=r$，$[T/m]=t$ 则

$$A=(1,1,1,\cdots,1)\times X^t\times X_1\times\cdots\times X_r$$

其中 X^t 可以按照 2.2 节中介绍的快速幂取模算法类似的方法在 $\log t$ 次矩阵乘法内计算出来。计算 X 和 $X_1\times\cdots\times X_r$ 需要 $m-1$ 次矩阵乘法，因此一共需要 $\log t+m=\log(T/m)+m$ 次矩阵乘法。矩阵乘法的时间复杂度为 $O(n^3)$，因此总的时间复杂度为 $O(n^3(m+\log(T/m)))$，比模拟法快　多了。

WEB　限于篇幅，线性代数的基本知识如矩阵的操作等这里都不介绍，请需要学习的读者可访问本书主页。

方程组模型　在一些问题比较复杂的情况下，方程组往往可以帮我们理清问题，下面举一个例子。本节后面的习题也有几个题目涉及到此知识，希望读者细心体会。

【例题 6】X 行星[①]

X 行星上的居民总是把自己的房子建造成三角形。他们建造房子的方法很特殊：最开

① 题目来源：CEOI 2000. X-Planet

始只有一堵墙，然后添加两面墙形成一个房子。以后每次都选择一面已经有的墙，然后建造两面新的墙构成一个新房子。例如图 2-4 就是一个合法的建造过程。

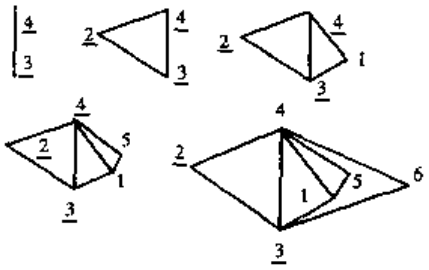


图 2-4　X 行星的住宅区建造过程

居民们最后建造了有 N 个顶点（$3 \leqslant N \leqslant 5\,000$）个住宅区，每个顶点都一个灯，还有一个开关。按下开关可以把该顶点处的灯以及所有与该顶点相邻的顶点处的灯都改变状态（开→关，关→开）。给出灯的初始状态，求一个按开关序列使得所有灯都打开。

【分析】

首先，我们可以发现开关灯的顺序是无关的，而且对于任何一盏灯，开关按下奇数次相当于按一次，按下偶数次相当于不按，因此可以用一个布尔变量 L_i 来描述第 i 盏灯，称该灯的“开关变量”。这样，一个开关序列就转化成了所有开关变量的取值，我们很容易想到列方程求解。

对于每个顶点 i，假设该处的灯初始状态为 s_i，那么只要能控制 i 的所有灯的开关变量值的异或值等于(not s_i)，那么该灯被打开。例如在最后一个图中。如果点 1 处的灯开始是关着的，因此 s_1=0，而能控制灯 1 的有灯 1，3，4，5，因此有方程:

$$L_1 \text{ xor } L_3 \text{ xor } L_4 \text{ xor } L_3 = (\text{not } s_1) = 1$$

对于每个灯都可以写出一个方程，因此最后得到一个方程组，含有 N 个未知数。虽然这不是代数方程组，但是高斯消元法仍然适用，因为对于任意元素 x，都有 x xor x=0，所以在消元的时候，只要每次选取合适的两个方程做 xor 操作就可以了，而且还不用考虑误差，因为只进行逻辑运算。

WEB 高斯消元法是一类很常用的线性方程组解法，限于篇幅这里不作介绍，需要学习的读者可以在本书主页上找到详细的资料和练习题。

高斯消元法解普通方程的时间复杂度为 $O(n^3)$，可是目前还有一个条件没有利用：房子是按照一种特殊的方法建立的。希望利用构造房子方法的特殊性得到方程组的特殊性，用特殊的方法消元而得到更低的时间复杂度。

每次构造房子都是“选择一面已经有的墙，然后建造两面新的墙构成一个新房子”。假设选择的是墙(i, j)，然后建造墙(i, x),(j, x)而得到新房子(i, j, x)，那么我们实际上是在方程 i 和 j 中各加入了一项 L_x，并建立的关于 x 的方程。假设这是最后一个建造的房子，那么只有这三个方程包含未知数 L_x。如果把新方程 L_i xor L_j xor L_x = not s_x 变形，可以得到

$$L_x = (\text{not } s_x) \text{ xor } L_i \text{ xor } L_j$$

把这个式子代入方程 i 和方程 j，就消去了未知数 L_x，得到了 $n-1$ 个未知数，$n-1$ 个方程的方程组。这样下去，可以用造房子的相反过程“拆房子”最终求出最初的那堵墙两边的点的开关变量（即消元过程），然后重新用造房子过程一步步求出其他点的开关变量（即回代过程）。由于每次消元只需要代入两个方程，因此消一个元的复杂度为 $O(n)$，消元的总时间复杂度为 $O(n^2)$，回代过程也是 $O(n^2)$的。

由于题目只给出了住宅区的最后形态而不是建立过程，需要自己找出建造过程，即每次出图中的两度点。这个过程留给读者分析，注意合理地运用数据结构来节省时间。

WEB 本题的背景是 2.4 节中将要提到的弦图．弦图可以加速高斯消元法．进一步的讨论参见本书主页．

练　习　题

2.1.1　序列[①]

把一个序列称做 1 序列，如果这个序列中相邻两个数字都恰好相差 1 并且序列的第一个数字为 0。具体来说：$\{a_1,a_2,\cdots,a_n\}$是一个 1 序列，当：

- 对任何 k ($1 \leqslant k < n$)：$|a_k - a_{k+1}| = 1$
- $a_1 = 0$

写一个程序，从文件中读入两个数据 n 和 s，分别表示序列的长及它的元素之和。找出一个符合条件的 1 序列，或者表明这样的序列不存在。

2.1.2　反等差数列[②]

给一个整数 n，把 1~n 这 n 个整数重新排列成$\{a_i\}$，使得对于排列中的任意三个元素 $a_p,a_q,a_r(1 \leqslant p<q<r \leqslant n)$，子序列 a_p,a_q,a_r不是等差数列，即 $a_q - a_p \neq a_r - a_q$。

2.1.3　幂的和[③]

佳佳想计算这个式子的值：

$$s_k(n) = \sum_{i=1}^{k} i^k$$

一项一项加的话太慢了，所以佳佳准备把这个和写成这样的形式：

$$s_k(n) = \frac{1}{M}(a_{k+1}n^{k+1} + a_k n^k + \cdots + a_1 n + a_0)$$

这是一个 $k+1$ 次多项式，其中 a_{k+1}，a_k，a_{k-1}，…，a_0，M 都是整数。给定一个 k，就可以得到一个关于 n 的多项式。如果规定 M 是正数而且取最小可能值，那么对于给定的 k，多项式 s_k 是惟一的。佳佳只要把 n 的值代进去就可以计算出想要的结果了。例如：

① 题目来源：Polish Olympiad in Informatics
② 题目来源：Polish Olympiad in Informatics
③ 题目来源：ACM/ICPC Regional Contest NEERC 1998

$$s_1(n)=\frac{1}{2}(n^2+n+0),\quad s_2(n)=\frac{1}{6}(2n^3+3n^2+n+0),\quad s_3(n)=\frac{1}{4}(n^4+2n^3+n^2)$$

给定一个 $k(1\leqslant k\leqslant 20)$，请你帮佳佳计算出多项式 S_k。

2.1.4　奇怪的数列[①]

有一列奇怪的数是这样定义的：$H=(A1\times X_{n-1}\times X_n+A2\times X_{n-1}+A3\times X_n+A4)$，如果 $H>B1$ 那么将 H 减去 C，直到 $H\leqslant B2$ 赋值 $X_n=H$。此处 $n>1$，$A1$、$A2$、$A3$、$A4$、$B1$、$B2$ 和 C 都是非负常数。

很容易发现在适当的 p、q 处对于所有的 $n\geqslant 0$ 存在关系 $X_{n+p}=X_{n+p+q}$。你的任务就是找出最小的 p 和 q。

2.1.5　三臂起重机[②]

三臂起重机用来为货车装载货物。假设车厢已经按照 1，2，3，…，编号，每一节车厢上仅可以放入一集装箱的货物。

三臂起重机的工作方法很简单。它每次可以从仓库中吊出三箱货物，把它们放到编号为（i，$i+p$，$i+p+q$），或者编号为（i，$i+q$，$i+p+q$）的三节车厢中，p,q 已知，且 p,$q\geqslant 1$。现在目的是把编号为 1，2，…，n 的车厢全部装满货物，且已知货车恰有 $n+p+q$ 节车厢。为了完成这个任务，必须为起重机安排一个装运程序，程序中包含若干行，每一行含有三个数字 x、y、z，满足 $1\leqslant x<y<z\leqslant n+p+q$，表示此次吊起的三件货物分别放到了编号为 x、y、z 的车厢中。

输入为三个数 p，q，n，分别是三臂起重机运货的两个参数以及需要装货的车厢数。$1\leqslant n\leqslant 300\,000$，$2\leqslant p+q\leqslant 60\,000$。你的程序输出正确当且仅当按照输出安排的吊运方法可以把编号为 1,2,…,n 的 n 节车厢全部装满，且每一节车厢至多装载了一箱货物。

2.1.6　中央暖气[③]

冬天降临了，乌拉尔的中央暖气供应系统却没有打开，因为它出了一点小小的问题：只有当所有的 n 个（$1\leqslant n\leqslant 250$）电子管都打开时，中央暖气系统才开始运行。有 n 个技术员，每人负责控管一或更多电子管，同一条电子管可能被多个技术员管理。当某位技术员接到开关命令时，他会将所有由他控管的电子管开启的关闭，关闭的开启。

你现在的任务是决定发布开关命令给哪些技术员以便运行中央暖气。

2.1.7　奶牛和多边形[④]

佳佳有 N 头奶牛编号为 1 到 $N(N\leqslant 10\,000)$。所有奶牛现在都在一个 $P(3\leqslant P\leqslant 20)$个顶点的凸多边形围栏里。佳佳教奶牛们玩一种游戏：$P$ 头奶牛跑到多边形的 P 个顶点上，其余奶牛挤在多边形的某条边的中间，使得每条边上的奶牛数恰好相等。佳佳说，如果每条边上所有奶牛（包括该边中间的奶牛和边的两个端点上的两头奶牛）编号之和一样，所有奶牛放假一天。奶牛们很高兴，可马上又开始发愁了：究竟怎样才能满足佳佳的要求呢？

输入两个整数 P 和 N，输出一种可行的方案。如果无解，输出-1。例如，$P=4$（四边

[①] 题目来源：Ural State University Problem Archive
[②] 题目来源：Ural State University Problem Archive
[③] 题目来源：Ural State University Problem Archive
[④] 题目来源：Romanian Olympiad in Informatics

形），N=8（八头奶牛），一种可行方案如图 2-5 所示。

因为每边上的和都是 12。

1	5	6
8		4
3	7	2

图 2-5　一种可行方案

2.1.8　矩阵法[①]

设 $A=\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix}$，则对于任何向量$(x,y)$，$(x,y)A=(y,x+y)$

利用这个性质和矩阵幂的快速求法，

（1）给出求第 i 个 fibonacci 数的 $O(\log n)$算法（忽略高精度计算）。

（2）给出任意常系数线性递推式的求第 i 项的快速方法（忽略高精度计算）。

2.1.9　迸裂的豌豆[②]

有 N 个碗排列成一行，其中一些碗里有些豌豆。这些豌豆很奇怪，它们自己会突然蹦起来，在空中迸裂成两半，然后分别落到两旁的碗里。

如图 2-6 所示是一个初始局面，它经过一次迸裂以后形成如图 2-7 所示的局面。

图 2-6　初始局面

图 2-7　一次迸裂后的局面

第一个碗左边有一块档板，所以该碗内的豌豆迸裂以后，左边那半会落回第一个碗内。如图 2-8 所示是再经过两次迸裂形成的局面。第 N 个碗的豌豆迸裂以后，右边那半会落在地上并消失。

图 2-8　三次迸裂后的局面

有些状态不可能是另一个状态的后继，我们把这样的状态称为“根状态”。给出一个状态，求它是根状态经过几次崩裂以后形成的。

① 题目来源：经典问题

② 题目来源：ACM/ICPC Regional Contest NWERC

2.1.10　公平外交[①]

N 个国家恰好处在多边形的 N 个顶点上，且只有处在同一条边上的两个国家才定期进行外交会晤。为公平起见，任两个国家的会晤地点到这两个国家的距离相等，同时为了节省外交花费，会晤地点到国家的距离应该尽可能的短。如图 2-9 所示，实线代表 N 个国家构成的多边形，虚线代表 N 个会晤地点构成的多边形。

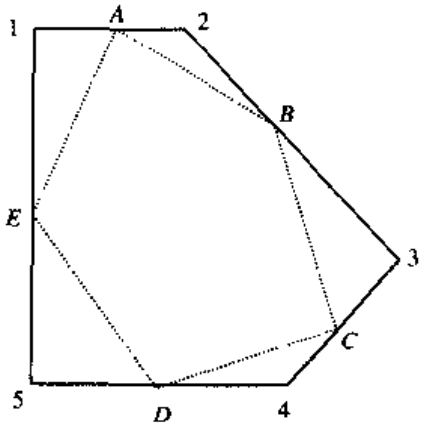


图 2-9　国家构成的多边形

任务 1：给你 N 个国家的平面位置，要你求出 N 个会晤地点的位置。

任务 2：给你 N 个会晤地点的平面位置，要你求出 N 个国家的位置。

2.1.11　GPA 排名系统[②]

目前，高等院校往往采用 GPA（Grade Point Average）来评价学生的学术表现。传统的排名方式是求每一个学生的平均成绩，以平均成绩作为依据进行排名。

但是这样的排名方法已经引起了教育界以及社会各界人士的争议，因为它存在着许多弊端。对于不同的课程，选课学生的平均成绩会不同程度地受到课程的难易程度和老师的严厉程度的制约。因而这样的排名系统无形中就鼓励了学生选择一些比较容易的课程，因为这样可以事半功倍地获得较高的平均分。

为了克服这些弊端，需要对排名系统做一定的改进。

一种改进的方案是对选第 i 门课的每一个学生的成绩加上一个特定的修正值 d_i，例如编号为 j 的学生该课的成绩 G_{ij} 修改为 $G'_{ij}=G_{ij}+d_i$。最终使得经过调整后，该课的平均分等于选该课的所有学生的所有课的平均分。对每一门课都做这样的调整，使得上述条件对所有课程都满足。这种调整方案一定程度地避免了传统排名系统的不公正。而你的任务正是根据一个大学某一个年级学生某学年的成绩，给出他们的排名。假设每一个学生都至少选一门课。

输入学生人数 m（$1\leqslant m\leqslant 500$）和课程数目 n（$1\leqslant n\leqslant 100$）以及各个学生各门课的成绩，输出采用改进方案后这些学生的排名。如果在上述调整后，有若干学生平均分相等（精确到小数点后的三位），则他们的名次相同，按照任意顺序输出。

[①] 题目来源：Ulm Univeristy Local Contest 2002

[②] 题目来源：CTSC 2001，命题人：石润婷

当然许多时候，上述调整无法顺利进行，即调整的目标无法达到（因此，在实际问题中，往往在最小二乘意义下获得一种最接近目标的调整方案）。也有可能或者因调整不惟一而不能确定学生的名次。若以上两种情况发生，则输出“fail”。

***2.1.12　超级幻方[①]

一个 5×5 的正方形里放了 1~25 恰好一次。它被称作是超级幻方，如果满足：每一行每一列每个对角线（包括 2 条主对角线和 8 个其他对角线）之和全相等。

01	07	13	19	25
14	20	21	02	08
22	03	09	15	16
10	11	17	23	04
18	24	05	06	12

图 2-10　一个超级幻方

这就是一个超级幻方。例如对角线 7+21+15+4+18=65，列 13+21+9+17+5=65 等。所有 20 个和都相等。佳佳已经在正方形里填了一些数了，他想把正方形填满得到一个超级幻方。他能办到吗？如果可以，能得到多少个不同的幻方？

WEB　本题有 25 个未知数，可以列出 20 个方程，但实际上只有 17 个是独立的。把解空间表示成正交拉丁方的和，然后推导所有数恰好取 1~25 各一次的条件。本题的详细解法可以在本书主页上找到。

***2.1.13　移位寄存器[②]

可以用移位寄存器来得到二进制伪随机数。移位寄存器可以储存 n（$n \leqslant 5\,000$）个 bit，即 $a_1,a_2,\cdots,a_n$。每次往右移动一位，即 $a_0,a_1,a_2,\cdots,a_{n-1}$，而 a_n 被输出，新加入的位 a_0 由外电路生成。外电路是一个 XOR 门，有 n 个开关，s_i=0 或 1，如图 2-11 所示。

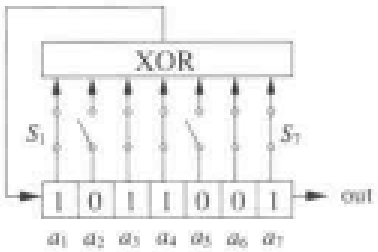


图 2-11　移位寄存器的外电路

从电路图中可以看出，输出 a_0 为 a_1 XOR a_3 XOR a_4 XOR a_6 XOR a_7，对于当前状态

[①] 题目来源：Elite Problemsetters' First Contest. 命题人：刘汝佳. 经典问题

[②] 题目来源：CEOI 2003. Register

$a_1...a_n$=1011001，反馈信号 a_0=1 XOR 1 XOR 1 XOR 0 XOR 1 = 0，因此下个时钟周期 $a_1...a_n$=0101100，整个电路输出为 1。从初始状态 1011001 开始 14 个时钟周期的情况如表 2-2 所示。

表 2-2　前 2*N* 个时钟周期的输出

tick	a1	a2	a3	a4	a5	a6	a7	out
0	1	0	1	1	0	0	1	-
1	0	1	0	1	1	0	0	1
2	1	0	1	0	1	1	0	0
3	1	1	0	1	0	1	1	0
4	0	1	1	0	1	0	1	1
5	0	0	1	1	0	1	0	1
6	1	0	0	1	1	0	1	0
7	1	1	0	0	1	1	0	1
8	0	1	1	0	0	1	1	0
9	1	0	1	1	0	0	1	1
10	0	1	0	1	1	0	0	1
11	1	0	1	0	1	1	0	0
12	1	1	0	1	0	1	1	0
13	0	1	1	0	1	0	1	1
14	0	0	1	1	0	1	0	1

给出前 2*N* 个时钟周期的输出，求出 XOR 电路。如果无解，输出相应信息；如果有多组解，仅输出一组即可。

WEB 本题很容易用高斯消元法在 $O(n^3)$时间内解决，但事实本题还有 $O(n^2)$的算法。有兴趣学习该算法和更多特殊方程组解法的读者可以访问本书主页。

2.1.14　礼物?![①]

一条美丽的小河上整齐地放置着 *N* 块岩石，形成了一条笔直的线。*N*（$N \leqslant 10^6$）块石头从左至右依次编号为 1…*N*，如下所示：

[Left Bank] - [Rock1] - [Rock2] - [Rock3] - [Rock4] … [Rock N] - [Right Bank]

相邻石头的距离是 1 米，左岸和岩石 1 的距离以及右岸和石头 *N* 之间的距离也是 1 米。

一天，小青蛙正要过河，一只大青蛙突然跳到他面前笑眯眯的对他说：

“你好啊，小青蛙！儿童节快乐！我送你一个礼物，放在了第 M 块岩石上。想不想看看？”

“好啊，我这就去！”

“等一等！这个礼物是专门给聪明的小青蛙准备的，你不可以直接跳去拿哦。”

“啊？那，我该怎么做呢？”

① 题目来源：OIBH Online Programming Contest #1，命题人：刘汝佳

“多跳几次嘛。第一次从左岸跳到石头1上，以后第i跳必须恰好跳$2i-1$米——不管你往前跳还是往后。不过不要跳到岸上去，这样你就没机会再回去拿了。”

“这样啊……好像不大容易呀。让我想想吧！”小青蛙自言自语到，“到底该不该试试呢？”

2.2 数论基础

本节的主要内容是和信息学关系比较密切的数论知识和方法，包括素数、整除、进位制、最大公约数、同余等问题。本节内容广而不深，例题也不难，主要是为激发大家的兴趣和开阔读者的眼界。

2.2.1 素数和整除问题

信息学中应用的关于素数和整除的知识并不多，关键在于要灵活运用。

整除和约数　如果a和b是整数，$a\neq 0$，若有整数c使$b=ac$，就说a整除c。在a整除b时，记a是b的一个因子（约数，divisor），b是a的倍数（multiple）。符号用$a|b$表示a整除b，a不能整除b记为$a\nmid b$。整数可整除性的基本性质有：

（1）若$a|b$，$a|c$，则$a|(b+c)$；

（2）若$a|b$，那么对所有整数c，$a|bc$；

（3）若$a|b$，$b|c$，则$a|c$。

最后一条性质提醒我们：整除关系具有传递性。由于它显然也具有自反性和反对称性，所以它是一个**偏序关系**。进一步，$<\mathbb{I},\mathbb{Z}>$是一个格（在组合数学部分，将在这个格上使用Möbius反演）。

素数和合数　大于1的正整数p称为素数（prime），如果p仅有的正因子是1和p。大于1又不是素数的正整数称为合数（compound）。如果n是合数，那么n必有一个小于或等于$\sqrt{n}$的素因子。

算术基本定理　每个正整数都可以惟一地表示成素数的乘积，其中素数因子从小到大依次出现（这里的“乘积”可以有0个、1个或多个素因子）。

注意，这个定理也叫做惟一分解定理。它是一个定理而不是公理！虽然在大多人看来，它是“显然成立”的，但它的确是需要证明的定理。

除法的定义和同余　令a为整数，d为正整数，那么有惟一的整数q和r，其中$0\leqslant r<d$，使得$a=dq+r$。可以用这个定理来定义除法：d叫除数，a叫被除数，q叫商，r叫余数。如果两个数a,b除以一个数c的余数相等，说a和b关于模c同余，记作$a\equiv b(\mathrm{mod}\ c)$。

最大公约数和最小公倍数　令a和b是不全为0的两个整数，能使$d|a$和$d|b$的最大整数称为a和b的最大公约数，用$\gcd(a,b)$表示，或者记为(a,b)。令a和b是不全为0的两个

整数，能使 $a|d$ 和 $b|d$ 的最小整数称为 a 和 b 的最小公倍数，用 lcm(a,b)表示，或者记为[a,b]。

令 a 和 b 为正整数，则

$ab = \gcd(a,b) \times \mathrm{lcm}(a,b)$

为了证明这个定理，考虑 a 和 b 的素因子分解式，设：

$$a = p_1^{a_1} p_2^{a_2} \cdots p_n^{a_n},\quad b = p_1^{b_1} p_2^{b_2} \cdots p_n^{b_n}。$$

注意，允许一些 a_i 或者 b_i 为 0（但不同时为 0），这样可以保证两边出现的素数相等。那么可以得到：

$$\gcd(a,b) = p_1^{\min(a_1,b_1)} p_2^{\min(a_2,b_2)} \cdots p_n^{\min(a_n,b_n)}$$

$$\mathrm{lcm}(a,b) = p_1^{\max(a_1,b_1)} p_2^{\max(a_2,b_2)} \cdots p_n^{\max(a_n,b_n)}$$

这两个公式可以用来计算 gcd 和 lcm。

关于最大公约数，通常还有如下定义：

互素和两两互素　如果 gcd(a,b)=1，则称 a 和 b 互素；如果整数 $a_1,a_2,\cdots,a_n$ 任意两个元素的最大公约数均为 1，则称这些数两两互素。

素数判定　关于素数判定，考虑两个问题：

问题一（筛）：如何求出 1~n 中的所有质数？

问题二（素数测试）：给出一个数 n，如何判断它是不是素数？

对于问题一，最常使用的是筛法。这里只介绍最古老也是最经典的一种。

Eraosthenes 氏筛法　每次求出一个新的素数，就把 n 以内的它的所有倍数都筛去。在实现的时候，对于一个素数 p，只需要筛去 $p\times p$，$p\times(p+1),\cdots$等就可以了，因为 $p\times q(q<p)$已经在 q 的第一个素因子被找到的时候被筛去了。

小知识——素数的筛法

素数的筛法有很多种，其中一个很重要的讨论问题就是时空复杂度的平衡（time-space tradeoff）。前几年比较好的算法的时间复杂度为 $O(n)$，空间复杂度为 $O(\mathrm{sqrt}(n)/\log n)$。另外还有时间复杂度为 $O(n/\log n)$，但空间复杂度为 $O(n/(\log n\log\log n))$的算法。最近又研究出了一些空间花费很少新的筛法，例如可以通过对 1~n 的所有素数执行 Jacobbi Sums 测试，只需要做 $n(\log n)^{O(\log\log\log n)}$次算术操作，空间复杂度为 $O(\log n^{o(\log\log\log n)})$。

对于问题二，有好几种方法。

方法一：朴素的判别法。从 2 开始试除小于 n 的所有自然数，时间复杂度为 $O(n)$。

方法二：注意到如果 a 是 n 的因子，那么 n/a 也是 n 的因子，所以如果 n 有一个大于 1 的真因子，则它必有一个不大于 $n^{1/2}$的因子，时间复杂度为 $O(n^{1/2})$。

方法三：更进一步的，如果 n 是合数，它必有一个**素因子**不大于 $n^{1/2}$。如果要检测一个 M 以内的数是否为素数，需要事先建立一个 $M^{1/2}$ 以内素数表。假设 $n^{1/2}$ 以内的素数有 k 个，方法三的时间复杂度为 $O(k)$。

这些方法的共同特点是：太慢。有没有快一点的算法呢？有！下面介绍的 Miller-Rabbin 算法就是其中之一。

伪素数　如果 n 是一个正整数，如果 $a^{n-1}\equiv 1(\text{mod } n)$，我们说 n 是基于 a 的伪素数。如果一个数是伪素数，它几乎肯定是素数。另一方面，如果一个数不是伪素数，它一定不是素数。由于我们可以自由的选取基，所以如果在选取了若干个基后发现 n 都是伪素数，就几乎可以肯定 n 是素数了。

Miller-Rabbin 测试　不断选取基 b(s 次)，计算是否每次都有 $b^{n-1}\equiv 1(\text{mod } n)$，若每次都成立，则 n 是素数，否则为合数。下面给出算法框架。需要说明的是，这里调用了一个函数，即计算 $a^b \bmod c$ 的值的函数 modular_exp(a,b,c)。它最简单的实现方法是一个循环，时间复杂度为 $O(b)$，但实际上可以用二进制扫描的方法得到一个 $O(\log^3 b)$的算法，将把这个问题放到“进位制”一节中介绍。算法的时间复杂度为 $O(s\times\log^3 n)$，true 表示素数。

```
Function Miller-Rabbin(n:longint):boolean;
begin
  for i:=1 to s do
  begin
    a:=random(n-2)+2;
    if modular_exp(a, n-1, n) <> 1 then return false;
  end;
  return true;
end;
```

请注意：这个算法属于 1.1 节中介绍的 Monte-Carlo 算法系列，它是概率型的，不是确定型的。不过由于多次运行后出错概率非常小，在实际中是可以信赖的。

小知识——素数测试

素数的测试是一个很重要的问题。一般来说，一个好的素数测试需要满足三个条件：正确性、普遍性和高效性。简单的试除算法太慢，概率算法又不能保证一定正确，而一些其他算法只能针对某一类整数才能做出判定。

概率算法依赖于 ERH（extended Riemann Hypothesis），在这个基础上介绍了 Miller-Rabin 算法，它的时间复杂度为 $O((\log n)^3)$，而成功概率为 3/4。值得注意的是：如果假定 ERH 是对的，那么根据 Ankeny 的理论，这些概率算法是可以改成确定性的，因为如果 N 是合数，那么只需要查找一个大小为 $2(\log n)^2$ 的证据集合就可以了，总的算法时间复杂度为 $O((\log n)^5)$。

有的判定算法要么是基于一些附加信息的，要么是只对一类特殊整数有效。例如，给出了 $n-1$ 的分解式，可以在 $O((\log n)^6/\log\log n)$的时间内判定 n 是否为质数。梅森数 $M_p=2^p-1$，费马数 $F_k=2^{2^k}+1$ 和形如 $f\times 2^s+1$ 的数分别有 Lucas-Lehmer 测试、Pepin 测试和 Proth 测试。

无条件的素数测试（Unconditional Primality Test）是基于代数数论（Algebraic Number Theory），近 15 年发展起来的方法，其中椭圆曲线占有比较重要的地位。这些算法有：APR 算法（时间复杂度为$(\log n)^{O(\log\log n)}$）、Jacobi sum 测试、Random curve 测试（在一定的假

定下渐进时间复杂度为 $O((\log n)^{12})$，Abelian variety 测试（时间复杂度为 $O((\log n)^6)$）等。这些理论涉及到相关理论发展得很快，有兴趣的读者可以参阅算法数论的书籍如《算法数论》。

最大公约数的求法　最大公约数一般不是用前面介绍的定理即因子分解来做的，而是用迭代法做。

欧几里德辗转相除法　利用 gcd(*a*,*b*)=gcd(*b*, *a* mod *b*)(*a*>*b*)进行迭代，算法框架如下。$O(\log b)$次整数运算，或者说 $O(\log^3 b)$次位运算。需要注意的是，在实际操作的时候，尽量应该把它写成迭代形式而不是递归，在运算量很大的情况下二者效率差别比较明显。

```
function gcd(a,b:longint):longint;
begin
  if b=0 then gcd:=a
  else gcd:=gcd(b, a mod b);
end;
```

提示：这里 $O(\log b)$是一个粗略的界，事实上当 n 固定后 gcd(*m*,*n*)的平均运行时间($m \leqslant n$)近似为 $12\ln 2 / \pi^2 \ln n$。另一种不用除法的 gcd 算法基于以下事实($a \geqslant b$)：

（1）若 $a=b$，则 gcd(*a*,*b*) = *a*

（2）如果 *a*,*b* 均为偶数，则 gcd(*a*,*b*) = 2×gcd(*a*/2, *b*/2)

（3）如果 *a* 为偶数，*b* 为奇数，则 gcd(*a*,*b*)=gcd(*a*/2,*b*)

（4）如果 *a* 和 *b* 均为奇数，则 gcd(*a*,*b*) = gcd(*a*−*b*, *b*)

扩展的欧几里德算法　如果(*a*,*b*)=*d*，那么一定存在整数 *x*,*y* 满足 $ax+by=d$。可以仿照 gcd 的求法用迭代法求出 *x* 和 *y*。具体的迭代方法不难得出来（注意从 *b*=0 的情形倒推），请读者自己推导或者参考下面的框架。

```
function extended_gcd(a,b:longint; x,y:longint):longint;
begin
  if b=0 then begin
     extended_gcd:=a;
     x:=1;
     y:=0;
  end
  else begin
     extended_gcd:=extended_gcd(b, a mod b);
     t:=x;
     x:=y;
     y:=t-(a div b)*y
  end;
end;
```

需要注意的是，满足 $ax+by=d$ 的数对（*x*,*y*）不是惟一的。因为当 *x* 增加 *b*，*y* 减少 *a*，和不变。

小知识——因数分解

因数分解也是一个很重要的问题。一般把这个问题转化成更简单的问题：给一个合数N，给出它的一个大于1的真因子，然后递归解决这个问题来得到完整的素因数分解式。我们只考虑这个简化后的问题。

大家知道，朴素的试除法最坏情况的时间复杂度为$O(n^{1/2})$，然而可以证明，在[1,n]中，至少有一半的元素可以在$O(n^{0.35})$的时间内解决。值得一提的是，传统的分解算法中还有一个基于费马小定理和gcd测试的$(p-1)$算法思路很特别，而关于它的平均情况时间复杂度尚没有很准确的结论。

现代的分解算法可以分为两类，即“combination of congruence”和“groups of smooth order”。它们的算法思想和相关的具体算法都可以在本书的网页中找到。

【例题1】佳佳的困惑[1]

给出一个数N，含数字1、2、3、4，把N的所有数字重新排列一下组成一个新数，使它是7的倍数。

【分析】

把数字1、2、3、4从中抽出，然后把其他数字按照原顺序排列（事实上，怎么排列都无所谓）组成自然数w，$w\times10\,000$整除7取余有7种可能，即是为0、1、2、3、4、5、6。这时如果能用数字1、2、3、4排列出7个数，使它们整除7取余的值分别为0、1、2、3、4、5、6，把这个4位数接在w后面即为问题的解。幸运的是，我们可以做到这一点。7个数如表2-3所示。

表2-3　7种余数对应的排列

余　数	0	1	2	3	4	5	6
排　列	3241	1324	1234	2341	1243	1342	2143

【例题2】除法表达式[2]

除法表达式有如下的形式：

$$X_1 / X_2 / X_3 / \cdots / X_k$$

其中X_i是正整数且$X_i\leqslant1000\,000\,000(1\leqslant i\leqslant k，k\leqslant10\,000)$。除法表达式应当按照从左到右的顺序求和，例如表达式1/2/1/2的值为1/4。可以在表达式中嵌入括号以改变计算顺序，例如表达式(1/2)/(1/2)的值为1。现在给一个除法表达式E要求告诉是否可以通过增加括号使表达式值为E'，E'是整数。

【分析】

易知X_1必为分子，X_2必为分母，其他数可以为分子也可以为分母。但是任何一个数让它作分子永远比做分母更有使得利于E'为整数。所以，令$E'=X_1 /（X_2 / X_3 / \cdots / X_k）$，然后判断它是否是整数。

[1] 题目来源：Ural State University Problem Archive
[2] 题目来源：Baltic Olympiad in Informatics, 2000

方法一：将分母 X_2 分解质因数，由于 $X_2 \leqslant 1\ 000\ 000\ 000$，所以质因数个数不超过 29 个。逐一扫描 $X_1, X_3, X_4, \cdots, X_k$，看能否将 X_2 约掉。

方法二：还是逐一扫描 $X_1, X_3, X_4, \cdots, X_k$，看能否将 X_2 约掉，但不进行因数分解，而是每次约掉它和 X_2 的最大公约数。

两种方法哪个更好？相信读者已经明白了。

【例题 3】数字游戏[1]

给你 $2N-1$ 个整数（其中 $N=2^k$，k 是不超过 10 的正整数），每个都是不超过 1000 的正整数。选出其中的 N 个整数，使得它们的和 S 是 N 的倍数。例如，$N=4$，有 7 个整数 1,2,3,4,5,6,7，那么可以选出 1,3,5,7，因为 $S=1+3+5+7=16$ 是 4 的倍数。

【分析】

本题一个很显眼的条件是 $N=2^k$，这使我们联想到了递归。

如果 $k=1$，那么需要选出两个数，它们的和是 2 的倍数。这是可以实现的，因为根据第 2.3 节介绍的鸽笼原理，3 个数必有两个数 a,b 同奇偶，把它们两个加在一起就可以了。

下面考虑 $k=2$ 的情况。这时有 7 个数，一定可以找到三对同奇偶的数 $(a_1,a_2),(b_1,b_2)$ 和 (c_1,c_2)（注意，再次利用了鸽笼原理），把它们分别加在一起以后得到 3 个数。因为这 3 个数都是 2 的倍数，因此“选出两个数，使得它们的和是 4 的倍数”等价于把这 3 个数分别除以 2 以后，“选出两个数，使得它们的和是 2 的倍数”。这样，问题转化为了 $k=1$ 的情况，而 $k=1$ 是能够处理的。

类似地，可以用递推的方法，每次都把得到的数分成两类，然后进行合并。

【例题 4】fibonacci 质数[2]

斐波那契数列是这样的：1、1、2、3、5、8、13、…，其中除了第一、第二个数，其余的数都是它的前两个数的和。如果某个斐波那契数与任何比它小的斐波那契数互质，那么称它为斐波那契质数。最小的斐波那契质数是 2，第二小的是 3，第三小的是 5，第四小的是 13，…。给出正整数 k，要求输出第 k 小的斐波那契质数是第几个斐波那契数。

【分析】

和“互质”有关的问题，可以考虑余数，考查所有数除以 M 的余数所组成的序列 $\{S_i\}$。$S_1=S_2=1$。设第一个零元素为 $S_k=0$，而它前面一个元素 $S_{k-1}=a$，那么 $S_{k+1}=S_{k+2}=a$。由于序列中每个数只和前两个数有关，因此从 S_{k+1} 开始的子序列“相当于”前面的序列乘以 a。由此可见，当且仅当 p 为 k 的倍数时，$S_p=0$！这样，对于两个数 $a>b\geqslant 1$，如果 a 是 b 的倍数，那么 f_a 是 f_b 的倍数。

因此，有：

- 第 1，2 项为 1，没有“比它小的数”，所以它不是斐波那契质数；
- 第 4 项也是斐波那契质数，因为虽然它是 F_2 的倍数，但是由于 $F_2=1$，所以它还是质数；
- 从第 5 项开始，某项为斐波那契质数当且仅当它的项数为质数。即，第 3,4,5,7,11,13,17,19,23,29，…项为斐波那契质数。

---

[1] 题目来源：UVA Problem Archive Online Contest，命题人：龙翀

[2] 题目来源：UVA Problem Archive Online Contest

【例题 5】神秘数[1]

有两个数 a 和 b（$1 < a < b$）。M 先生知道 $a \times b$ 的值，S 先生知道 $a+b$ 的值，两人有如下对话：

M 先生：我不知道 a 和 b 的值。

S 先生：我也不知道，而且之前我还知道你不知道。

M 先生：我现在知道 a 和 b 的值了。

S 先生：我现在也知道 a 和 b 的值了。

给定 x 和 y（$2 \leqslant x, y \leqslant 550$），求所有满足对话场景的$(a,b)$且 $x \leqslant a < b \leqslant y$。假设二位先生都足够聪明且没有撒谎。

【分析】

显然，题目的关键在于如何判断(a, b)是否满足上述 4 句对话。

由于这 4 句话有着很强的逻辑性，不妨用数学语言来描述，从而能更直接和客观对(a, b)进行分析。为了方便分析，根据题意首先设两个集合如下：

$$M(a, b) = \{ (x, y) \mid x \cdot y = a \cdot b \text{ 且 } 1 < x < y \}$$

$$S(a, b) = \{ (x, y) \mid x + y = a + b \text{ 且 } 1 < x < y \}$$

第一句话：M 先生说，他不知道 a 和 b 的值。

直接理解这句话，此时 M 先生虽然知道 $a \times b$ 的值，却无法确定 a 和 b。那么，换个角度，如果集合 $M(a, b)$只有一个元素，M 先生显然可以确定 a 和 b 的值！

设 $f_1(a, b)$ 表示(a, b)是否满足第一句话。

$$f_1(a, b) = \begin{cases} \text{true} & |M(a,b)| = 1 \\ \text{false} & \text{else} \end{cases}$$

第二句话：S 先生说，他不知道 a 和 b 的值，同时在之前知道 M 先生不会知道。

这句话有两个部分：

前一部分，S 先生不知道 a 和 b 的值，则集合 $S(a,b)$显然不会只有 1 个元素；

后一部分，S 先生知道 M 不知道。S 先生凭什么做出这样的判断呢？假设 S 先生知道 $a + b = 10$，则 S 先生推测 a 与 b 有可能是 3 和 7，如果 $f_1(3, 7) = \text{true}$，则 M 先生一开始便会猜测到。

而 S 先生断定 M 先生一开始绝对不会猜到，因此对于每种存在的可能 $a' + b' = a + b$ 都会有 $f_1(a', b') = \text{false}$，这样 S 先生才能做出如此大胆的猜测。

将这句话对应到数学表达上，设 $f_2(a, b)$表示(a, b)是否满足前两句对话：

$$f_2(a,b) = \begin{cases} \text{true} & f_1(a,b) = \text{true} \quad // \text{首先第一句话要满足} \\ & |S(a,b)| = 1 \quad // \text{第二句话前一部分} \\ & \forall (p,q) \in S(a,b)\ f_1(p,q) = \text{true} \quad // \text{第二句话后一部分} \\ \text{false} & \text{else} \end{cases}$$

第三句话：M 先生说，他现在知道 a 和 b 的值了。

M 先生也在推测，他是在前两句的基础上进行推测的。由于他知道 $a \times b$ 的值，因此如

[1] 题目来源：ACM/ICPC Regional Contest SEERC 2001

果只存在一对$(p,q)\in M(a,b)$满足前两句对话，那么M先生就可以断定a与b的值就是p和q！

用$f_3(a,b)$来表示(a,b)是否满足前三句对话：

$$f_3(a,b)=\begin{cases}\text{true} & f_2(a,b)=\text{true}\quad \text{//首先前两句话要满足只存在一对}(p,q)\in M(a,b)\text{满足}\\ & f_2(p,q)=\text{true}\quad \text{//第三句话}\\ \text{false} & \text{else}\end{cases}$$

第四句话：S先生说，他现在也知道a和b的值了。

S先生是在前三句话基础上进行推测的。类似第三句话的推理，如果此时只存在一对$(p,q)\in S(a,b)$满足前三句对话，那么S先生可以断言a与b的值就是这对p和q！

用$f_4(a,b)$来表示(a,b)是否满足前四句话：

$$f_4(a,b)=\begin{cases}\text{true} & f_3(a,b)=\text{true}\quad \text{//首先前三句话要满足只存在一对}(p,q)\in S(a,b)\text{满足}\\ & f_3(p,q)=\text{true}\quad \text{//第四句话}\\ \text{false} & \text{else}\end{cases}$$

至此，已经把4句话分别转化成了4个**逻辑判别函数**，要求在给定的范围内，满足$f_4(a,b)$的所有数对。

这4个判别函数是层层嵌套，逐步构造出的，显然$f_1(a,b)$与$f_2(a,b)$是判别的基础。

层次一：$f_1(a,b)$的实现。

判断条件只有一个$|M(a,b)|=1$。举个例子：M(3,4)={(2,6),(3,4)}有两个元素，而M(3,7)={(3,7)}只有一个元素，M(3,9)={(3,9)}也只有一个元素。

通过分析，不难发现$|M(a,b)|=1$的判断可以转化成对a,b性质的判断：

a是质数且b是质数或者a是质数且$b=a\times a$ <==> $|M(a,b)|=1$

于是，可以事先预处理，求出一个素数表，这样就可以在$O(1)$的时间内判断出f_1函数的结果。

层次二：$f_2(a,b)$的实现。

f_2函数共有3个判断条件：

（1）$f_1(a,b)=\text{true}$ 直接调用f_1函数即可；

（2）$|S(a,b)|=1$ 此判断条件等价于判断 $a+b\leqslant 6$；（想一想，为什么？）

（3）对所有$(p,q)\in S(a,b)$ 有$f_1(p,q)=\text{true}$。

如果列举所有的(p,q)来一一判断，效率显然不高。那么对于什么样的$p+q=a+b$，会产生$f_1(p,q)=1$呢？回顾f_1函数的性质，不难发现$f_1=\text{false}$的情况可以分以下两类：

① p为质数，$q=p^2$

如此则有$p+p^2=a+b$，解出p。若p是质数，则存在$(p,p^2)\in S(a,b)$且$f_1(p,p^2)=\text{false}$！条件三不满足。

② p和q均为质数

（i）$a+b$为偶数

我们知道歌德巴赫猜想在小范围内是正确的，不妨在这里应用之，如果$a+b$是偶数，那么必然存在质数p和q：$p+q=a+b$且$f_1(p,q)=\text{false}$！

（ii）$a+b$ 为奇数

假设 $p=2$，若 $a+b-2$ 也是质数，则存在$(2,a+b-2)\in S(a,b)$且 $f_1(2,a+b-2)=$ false！条件二不满足。

若 p 为非 2 的质数，q 也为非 2 的质数，则 p 和 q 必定是奇数，$p+q$ 必定为偶数，不可能组成奇数 $a+b$！

如此一来，对函数 f_2 的判断也可以在 $O(1)$的时间内解决了。

层次三：$f_3(a,b)$的实现。

（1）f_3 函数有两个条件：

（2）$f_2(a,b)=$ true 直接调用 f_2 函数即可；

只存在一对$(p,q)\in M(a,b)$满足 $f_2(p,q)=$ true 列举所有的(p,q)逐一判断。

如此一来，对函数 f_3 的判断可以在 $O(\sqrt{ab})$内解决。

层次四：$f_4(a,b)$的实现。

f_4 函数有两个条件：

（1）$f_3(a,b)=true$；

（2）只存在一对$(p,q)\in S(a,b)$满足 $f_3(p,q)=$ true。

f_4 函数反复调用到了 f_3 函数，由于 f_3 函数时间复杂度不是 $O(1)$的，所以效率有可能降低。因此，不妨先算出所有可能的 f_3 函数的值，存在一张表里，在 f_4 函数中就可以**直接调用**了。

所以，本题的大致算法如下：

（1）构造素数表；

（2）预先算出所有可能的 f_3；

（3）枚举范围内的所有数对，推算其 f_4 函数值是否为 true。

这样的算法时间复杂度大约为 $O(Y^3)$，其中 Y 为输入的上界，是可以接受的。

2.2.2　进位制

进位制　给一个非负整数 N，如果 $N=a_t\times b^t+a_t\times b^{t-1}+\cdots+a_1\times b+a_0$,$(0\leqslant a_i<N$, a_i 为整数)那么我们把序列$(a_ta_{t-1}\cdots a_1a_0)_{(b)}$称为 N 的 b 进制表示。如果 N 在 b 进制下是 t 位数的，那么有不等式 $b^t\leqslant N<b^{t+1}$，即 $t\leqslant\log_bN<t+1$。由于 t 是自然数，因此 $t=[\log_bN]$。

读者不难根据定义自己写出两个 N 进制数互相转化的程序[1]。

快速幂取模　前面提到过，可以用二进制扫描的方法快速地计算 $a^b \bmod c$ 的值，下面就来看看这个过程。根据模算术的基本知识（参见 2.2.3 节），$(a\times b) \bmod c=((a \bmod c)\times b) \bmod c$，那么可以把 $a^b \bmod c$ 变成一系列比较小的数的乘积。把 b 写成二进制$(a_ta_{t-1}\cdots a_1a_0)$，那么有：

$$b=a_t2^t+a_{t-1}2^{t-1}+a_{t-2}2^{t-2}+\cdots+a_12^1+a_02^0$$，其中 a_i 非 0 即 1。

[1] 还有其他形式的进制，如 $a_i=-1,0,1$ 的平衡三进制表示

这样，有：$a^b \bmod c=(a^{a_t 2^t+\cdots+a_0 2^0}) \bmod c=((a^{a_0 2^0} \bmod c)\times a^{a_1 2^1} \bmod c)\cdots$

也就是说，只经过 $t=\log_2 b$ 次计算就可以了。关键在于：能否在常数时间内完成每次计算？答案是肯定的。首先，可以忽略 a_i，因为它们**非 0 即 1**，所以可以计算出每项，但是只使用 a_i 为 1 的那些项。注意到 $a^{2^{i+1}} \bmod c=(a^{2^i} \bmod c)^2 \bmod c$（因为$(2^i)^2=2^{i+1}$），这样，就可以在常数项的时间里由 2^i 项推出 2^{i+1} 项，总的时间复杂度为 $O(\log b)$。如果考虑整数相乘的时间复杂度，则总时间复杂度为 $O(\log^3 b)$。

康托展开　在考虑排列的时候经常需要用到康托展开。把整数写成 $P=a_n n!+a_{n-1}(n-1)!+\cdots+a_2 2!+a_1 1!$ $(0\leqslant a_i<i, i=1,2,\cdots,n)$的形式称为把 P 进行康托展开。请读者仿照进位制的处理方法写出求整数的康托展开的算法以及给出两个整数的康托展开，求它们的和的算法。

【例题 1】自动取款机[①]

ByteLand 上的每位居民都有一个自己的银行账号，账号上记录了居民的存款金额，以硬币为单位。他们可以用自己的账号在 100 台自动取款机上进行取款或付款的操作。

这 100 台自动取款机的编号从 0 到 99，每台取款机都有自己的运行特点。当你在编号为 i 的取款机上进行操作时，若 i 是偶数，你将从取款机中获得 2^i 个硬币；若 i 是奇数，你将付给该取款机 2^i 个硬币。

假设你要从银行取 14 枚硬币，那么你可以在第 4 号取款机上操作，获得 16 枚硬币，接着在第 1 号取款机上操作，付出 2 枚硬币，最终就得到了 14 枚硬币。

假设你要付给银行 7 枚硬币，那么你可以在第 3 号取款机上操作，先付出 8 枚硬币，接着在第 0 号取款机上操作，取回 1 枚硬币，最终你付给银行 7 枚硬币。

注意，由于银行系统的保险措施，每台取款机最多只能被操作一次。

现在，L 先生将要向银行支取或交纳一定数额的硬币。你能设计出一个方案，对某些取款机进行操作，从而恰好完成 L 先生的要求。

【分析】

这道题描述了一个自动取款机的工作过程，需要你来编一个程序决定如何付款、取款。自动取款机的工作过程是这样的，如果 i 为偶数，则第 i 个取款机将收取 2^i 元的钱；如果 i 为奇数，则第 i 个收款机将支付 2^i 元的钱。所以顾客一共支付的费用是

$\sum_{i=2k} a_i 2^i-\sum_{i=2k+1} a_i 2^i$（其中 a_i 表示第 i 个取款机是否被使用），稍加分析，可以发现这个式子也可以写成如下形式：$\sum a_i(-2)^i$。

这个式子与整数的二进制表示法非常相似，所不同的只是把 2 换成了-2。已经知道，求一个十进制整数的二进制表示只需要 $\log n$ 的时间（不考虑高精度计算时间），那么这种计算二进制的方法在这里还能适用吗？先举几个例子看一看。

如图 2-12 所示的例子表示-2 进制可以像二进制那样进行转换，因此这道题就变得非常容易，要只要求一个高精度数除以-2 的高精度运算即可。但是由于涉及到带符号位的高精度运算，仍然需要做一些额外的处理。

① 题目来源：Polish Olympiad in Informatics

图 2-12　进制转换的例子

具体而言，根据当前数的正负、奇偶分别进行处理。用 key 来表示当前数的正负，用一个数组来记录当前数的绝对值。每次除法都改变 key 的值，再按如下方法处理：

- 正偶数：直接除以 2。
- 正奇数：减 1 后除以 2。
- 负偶数：其绝对值直接除以 2。
- 负奇数：将其绝对值加 1 再除以 2。

以上过程一直进行到该数为 0。

进位制实际上是用一串数字来表示一个数。有时候，并不需要求一个数的完整表示，而只对某个数字片段或者数字的个数感兴趣。

【例题 2】人类学家的烦恼[1]

考古学家正对着断壁残垣上一串古老的数字发愣。数字串左边是完好无损的，右边超过总长度一半的数字串却被千万年的雨水洗去了容颜。考古学家惟一知道的就是这串数字是 2 的正整数幂。作为知名数学家，你有责任告诉困惑的考古学家最小的指数 E（如果存在的话）使得 2^E 前面的数字与残垣上数字串左边完好的部分相符合。

【分析】

假设最左边的数字串为 X，而右边有 n 位已经无法辨认，则这个数为 $X\times10^n$ 到 $(X+1)\times10^n-1$，因此有 $X\times10^n\leqslant2^E<(X+1)\times10^n$。两边取以 2 为底的对数，则有不等式：

$$A=\log_2X+n\log_210\leqslant E<\log_2(X+1)+n\log_210=B$$

如果 A 和 B 中夹有一个整数 E，那么这个 E 就是符合条件的数。由于 E 会随着 n 的增大而增大，因此从小到大枚举 n，就可以找到最小的 E。

WEB　本题最好的做法实际上是用连分数逼近，更多的内容请访问本书主页。

【例题 3】征服者的军营[2]

征服者常和他的司令官玩这样一种游戏：一座 $N(N\leqslant2\,000)$层的台阶上，每层台阶上站有一定数量的士兵。每一次征服者将士兵分成两组，然后由司令官决定让某一组的士兵留下而另一组的士兵回家，留下的所有士兵都向上爬一层台阶。如果某个士兵能爬上最高层的台阶，那么征服者胜利，否则司令官胜利。如果司令官总采取最佳策略，那么征服者要

[1] 题目来源：UVA Problem Archive
[2] 题目来源：CEOI 2001. Conquer

怎样才能尽可能赢司令官呢？

【分析】

如果只在 S 层有两名士兵，那么征服者必然将两名士兵分到不同的两组，而司令官让其中一名士兵回家休息，另一名就爬上了 S-1 层。可见，S 层的一名士兵等价于 S+1 层的两名士兵。从这个角度说来，一名 S 层的士兵具有价值 2^{N-S}，整个军队具有的总价值 Sum 等于所有士兵的价值和，和前面介绍的二进值十分相似。

证明 1：如果 $\text{Sum}<2^{N-1}$，那么征服者必输无疑。

每一轮游戏，无论征服者怎样分组，必然存在一组士兵人数小于 2^{N-2}，司令官让这组士兵留下并向上爬一层台阶，另一组回家，这样 Sum 仍然也将永远保持小于 2^{N-1} 的状态。如果某个士兵能爬上最高层的台阶，那么 $\text{Sum}\geq 2^{N-1}$，因为 $\text{Sum}<2^{N-1}$ 和 $\text{Sum}\geq 2^{N-1}$ 是矛盾的，所以只要司令官愿意，征服者必输无疑。

证明 2：如果 $\text{Sum}\geq 2^{N-1}$，那么征服者一定有办法胜利。

引理：只要 $\text{Sum}\geq 2^{N-1}$，征服者一定可以把士兵分成总价值均$\geq 2^{N-2}$的两组。

引理证明很简单，由下往上把第 S 层的 $2K$ 个士兵替换成 S-1 层的 K 个士兵，使得替换后每层士兵人数均≤ 1。因为 $\text{Sum}\geq 2^{N-1}$，所以第 1 层一定站着至少一个士兵。又因当前征服者还未胜利，所以这个第 1 层的士兵一定是第 2 层的两个士兵替换而得。将第 2 层的这两个士兵分到不同的两组，就可以保证两组士兵总价值均$\geq 2^{N-2}$。当然这些第 2 层的士兵也有可能是第 3 层的士兵替换而得，不过无所谓，先前记录替换的过程，现在根据记录将士兵还原。

每一轮游戏，征服者都有办法将士兵分成总价值均$\geq 2^{N-2}$的两组。无论司令官让哪一组回家，剩下一组士兵向上爬一层台阶后 Sum 仍然保持$\geq 2^{N-1}$的状态。最终，当士兵人数减少到 1 的时候，由于 $\text{Sum}\geq 2^{N-1}$，必有一个士兵站在最高层的台阶上，所以只要征服者愿意，司令官必输无疑。

对于 $\text{Sum}<2^{N-1}$ 的情况，征服者第一次就将士兵分成一组满员、一组空集，以求必输状态的尽早结束。对于 $\text{Sum}\geq 2^{N-1}$ 的情况，来看看怎样完成士兵的分组。由下往上将尽量多的士兵从第 S 层替换到 S-1 层，同时记录 $G[S-1]$表示当前 $S-1$ 层的所有士兵中有多少是 S 层替换上来的。还原士兵的时候，首先将一个第 2 层的士兵分到第一组，如果所有第 2 层的士兵都是替换而得的，那么又尝试将两个第 3 层的士兵分到第一组……由此类推，每一次都将尽量多的 S 层原有的士兵分到第一组，如果数量不足需求，就用 S+1 层的双倍士兵来填补不足的数量。

如果不考虑计算 Sum 的时间，那么每一轮游戏复杂度大约是 $O(N)$，总体复杂度为 $O(N^2)$。

WEB 本题是一个动态游戏，“游戏论”将会有更多的例子。本书主页将会对此理论进行一些介绍并放上相关资料。

练　习　题

思考题：

2.2.1　证明 $\gcd(F_n, F_m) = F_{\gcd(n, m)}$。

2.2.2　设计一个在线算法，用尽量少的预处理时间和 $O(1)$的时间回答询问：有多少个素数 p 满足 $t_1 \leqslant p \leqslant t_2$? 其中 $1 \leqslant t_1, t_2 \leqslant 10^6$。

2.2.3　设计一个从 a 进制转化到 b 进制的算法。你的算法应当用尽量少的附加空间，时间效率尽量高。

编程题：

2.2.4　反素数[①]

若一个正整数 N 满足：则在 $1 \cdots N$ 的整数中，N 具有最多的约数，那么称 N 是一个反素数。如 1、2、4、6、12、24 都是反素数。

写一个程序，给出一个正整数 $N(N \leqslant 2 \times 10^9)$，找出不超过 N 的最大的反素数。

2.2.5　神奇的数对[②]

可以证明：对于任何的整数 X 和 Y，如果 $5X+4Y$ 能被 23 整除，那么 $3X+7Y$ 一定也能被 23 整除。

对于给定的 N_0，和整数对(A_0, B_0) $(N_0, A_0, B_0 \leqslant 10\ 000)$，找出所有的整数对$(A, B)$使得对于任何的整数 X 和 Y，如果 A_0X+B_0Y 能被 N_0 整除，那么 $AX+BY$ 都能被 N_0 整除$(0 \leqslant A, B < N_0)$。

2.2.6　灯泡[③]

有 n 个灯泡排列成环形（灯泡 1 左边是灯泡 n，灯泡 $k(k>1)$左边是灯泡 k-1），在时间 0 时有的亮，有的不亮。灯泡 i 在时间 $t+1(t \geqslant 0)$改变状态（亮→不亮，不亮→亮）当且仅当灯泡 i 左边的灯泡在时间 t 是亮的。求在时间 M 时，每个灯泡的状态$(n \leqslant 10^6, m \leqslant 10^9)$。

****2.2.7　Farey 序列**[④]

把所有分子和分母都小于 N 的最简分数从小到大排成一行，形成的序列称为 Farey 序列。例如 N=4 时的序列是 1/4, 1/3, 2/3, 3/4。

给出 N 和 k，求出 N 所对应的 Farey 序列中第 k 大的数$(N \leqslant 10^6)$。

提示：用二分法转化为计数问题，用筛法的思想来加速。

2.2.3　同余模算术

同余问题应用十分广泛。在本节中，先介绍一些基本概念和定理，然后着重讲解一些

① 题目来源：Polish Olympiad in Informatics
② 题目来源：PhTL #1 Training Contests，Fall 2001
③ 题目来源：Baltic Olympiad in Informatics, 2003
④ 题目来源：Balkan Olympiad in Informatics, 2003

模方程的解法。

在 2.2.1 中已经介绍了同余的概念。下面先重复这个定义。

同余　如果 a 和 b 除以 c 的余数相同，说 a 和 b 关于模 c 同余，记作 $a\equiv b(\mathrm{mod}\ c)$。下面先介绍一些同余的基本性质。

性质 1. 同余关系是自反，对称、传递的，因此是一个**等价关系**。

性质 2. 若 $a\equiv a_1, b\equiv b_1(\mathrm{mod}\ m)$，则 $a+b\equiv a_1+b_1, a-b\equiv a_1-b_1, ab\equiv a_1b_1\ (\mathrm{mod}\ m)$.

性质 3. 若 $ac\equiv bd, c\equiv d(\mathrm{mod}\ m)$，且$(c,m)=1$，则 $a\equiv b(\mathrm{mod}\ m)$.

缩系　以上三条性质把整数分成了 m 个等价类，每一类由相互同余的数组成，称为模 m 的剩余类。在一个模 m 的剩余类中，如果有一个数和 m 互素，则剩余类中所有数和 m 互素，称该类和 m 互素。和 m 互素的剩余类的个数记为 $\varphi(m)$。在和 m 互素的剩余类中各取一个代表元：$a_1,a_2,a_3,\cdots,a_{\varphi(m)}$，它们组成 m 的一个缩剩余系，简称缩系。

$\varphi(m)$的性质　关于 $\varphi(m)$，有以下性质：

- 对于质数 p, $\varphi(p)=p-1$；$\varphi(p^n)=p^n(p-1)$。
- 若$(m,m')=1$，$\varphi(mm')=\varphi(m)\,\varphi(m')$，即 φ 函数是积性函数。

这两条性质可以由 2.3 节推导出的 φ 的计算公式直接得到。

Euler 定理　若$(k,m)=1$，则 $k^{\varphi(m)}\equiv 1\ (\mathrm{mod}\ m)$。由 Euler 定理立即求得。

费马小定理　对于素数 p 和任意整数 a，有 $a^p\equiv a(\mathrm{mod}\ p)$。反过来，满足 $a^p\equiv a(\mathrm{mod}\ p)$，$p$ 也“几乎一定”是素数（回忆刚刚介绍的 Miller-Rabin 素数测试的原理）。

线性同余方程 $ax\equiv b(\mathrm{mod}\ n)$　显然，它等价于存在整数 y，使得 $ax-ny=b$①记 $d=(a,n)$，$a=d\times a'$, $n=d\times n'$，那么必须有 $d\mid b$，否则方程变为 $a'\times x - n'\times y = b/d$；②左边是整数，右边却不是，因此无解。

方程的解　这样，把方程 1 变成了方程 2，且$(a',n')=1$（因为 d 是 a 和 n 的**最大**公约数）。回忆扩展的辗转相除法，可以立刻得出方程②的解。x 不是**惟一**的，但是 x 的所有解属于同一个模 n'剩余系，即所有 x 相差 n'的整数倍。由于 $n=n'\times d$，所以**所有 x 相差 n/d 的整数倍**。如果把解看成是一些模 n 的等价类，那么答案就只有 d 个了。它们分别以

$$x,\ x+n/d,\ x+2\times n/\mathrm{d}\ ,\cdots,\ x+(d-1)\times n/d$$

为代表元。

求 $ax\equiv b(\mathrm{mod}\ n)$所有解的算法　利用欧几里德辗转相除法，模方程等价于存在整数 y，使得 $ax-ny=b$。算法框架如下，时间复杂度为 $O(n+\log b)$。

```
procedure modular_linear_equation(a,b,n:longint);
begin
  d:=extended_gcd(a,n,x,y);
  if b mod d<>0 then no_answer;
  e:=x*(b div d) mod n;
  for i := 0 to n-1 do
    ans[i+1] := (e + i*n div d) mod n;
end;
```

方程组的情形　由于 $ax \equiv b(\bmod n)$已经变成了若干形如 $x \equiv b(\bmod n)$，所以只考虑这样的情况，即方程组 $x_i \equiv b_i(\bmod n_i)$。

先看一个简单的情况，即只有两个方程的情况：

令 m 为 m_1,m_2 的最小公倍数，则同余方程组

$$x \equiv a_1(\bmod m_1) \quad ①$$

$$x \equiv a_2(\bmod m_2) \quad ②$$

有解的充分必要条件是$(m_1,m_2) \mid a_1 - a_2$。如果这个条件成立，则方程组仅有一个小于 m 的非负整数解。利用扩展的辗转相除法，很容易把它解出来。

①等价于 $x=a_1+m_1y$；②等价于 $x=a_2+m_2z$，联立得 $a_1+m_1y=a_2+m_2z$。

令 $c=(m_1,m_2)$，则：

$$(a_1-a_2)/c = m_2/c \times z - m_1/c \times y$$

和前面的方法类似，由辗转相除法求出 p,q 使得：

$$(m_2/c) \times p + (m_1/c) \times q = (m_2/c, m_1/c) = 1$$

那么只需要取 $z = p \times (a_1-a_2)/c$，则得到了一个解 $x=a_2+m_2z$。

中国剩余定理　设 $m_1,m_2,\cdots,m_k$ 两两互素，则下面同余方程组：

$$\begin{aligned} x &\equiv a_1 (\bmod m_1) \\ x &\equiv a_2 (\bmod m_2) \\ &\vdots \\ x &\equiv a_k (\bmod m_k) \end{aligned}$$

在 $0 \leqslant x < M = m_1 m_2 \cdots m_k$ 内有惟一解。

记 $M_i = M/m_i (1 \leqslant i \leqslant k)$，因为$(M_i,m_i)=1$，故有一整数 p_i,q_i 满足 $M_ip_i+m_iq_i=1$，如果记 $e_i=M_ip_i$，那么有：

$$e_i \equiv \begin{cases} 0 (\bmod m_j), & j \neq i \\ 1 (\bmod m_j), & j = i \end{cases}$$

很明显，$e_1a_1+e_2a_2+\cdots+e_ka_k$ 就是方程组的一个解（请读者自己验证），这个解加减 M 的整数倍后就可以得到最小非负整数解。

求 $x \equiv a_i(\bmod m_i)$的算法（m_i 两两互素）　最开始求 M，对于 1~k 的每个 i，先求 M_i，再求 e_i，把所有 $e_i \times a_i$ 累加起来。算法框架如下，时间复杂度为 $O(k\log m)$，其中 m 为 m_i 的最大值。总空间 $O(k)$，附加空间仅为 $O(1)$。

```
function linear_modular_equation_system(k:integer);
begin
  M:=1;
  For i:=1 to k do
    M:=M*m[i];
  Ans:=0;
  For i:=1 to k do
  Begin
```

```
    Mi=M div m[i];
    Extended_gcd(Mi,mi,pi,qi);
    Ans := (Ans + Mi * pi * a[i]) mod M;
  End;
  If Ans<0 then Ans:=Ans + M;
  Linear_modular_equation_system:=Ans;
End;
```

大整数的一种表示方法　假设有一台计算机，处理 100 内的整数比 100 以上的要快得多，那么可以把整数表示为模两两互素的 100 以内的整数的余数多元组，就可以将差不多所有整数计算限制在 100 以内的整数上。例如可以用 99,98,97,95 作为模数(它们两两互素)，根据中国剩余定理，每个小于 99×98×97×95 = 89 403 930 的非负整数均可惟一地用该整数被这 4 个 4 个因数除的余数表示。例如 123 684 = (33,8,9,89)，类似地，413 456=(32,92,42,16)，这样，123 684+413 456 = (33,8,9,89) + (32,92,42,16) = (65 mod 99, 100 mod 98, 51 mod 97, 105 mod 95) = (65, 2, 51, 10)，为了得到最后的表达式，可以解同余方程组。在加减乘法运算很频繁但不超出范围的情况下，这种表示方法是很合适的——它只在初始化和输出结果时涉及到大整数运算。

考虑把一组 2^k-1 的整数作为模。可以证明 $\gcd(2^a-1, 2^b-1) = 2^{\gcd(a,b)}-1$，而且很容易完成模形为 2^k-1 的取余数运算，因此这样的一组模是有用的。

一般情形　对于 m_i 不是两两互素的情况，可以利用刚才每次解两个方程的方法，每次合并两个方程，算法的时间复杂度仍然是 $O(k\log m)$。这个方法也可以用一种形象的方法“循环步长法”来实现。

对于其他类型的同余方程，简单作一下介绍。

多变元的线性同余方程　方程的形式为：$a_1x_1+a_2x_2+\cdots+a_nx_n+b\equiv 0(\mathrm{mod}\ m)$，它有解$(x_1,x_2,\cdots,x_n)$的充分必要条件为：$(a_1,a_2,\cdots,a_n,m)\mid b$。若此条件成立，解的个数为：$m^{n-1}(a_1,a_2,\mathrm{m},\cdots,a_n,m)$。

证明用到了数学归纳法，且证明过程本身可以得到该方程的解，这里稍微用点篇幅把它叙述一下：

n=1 显然成立。

令$(a_1,\cdots,a_n,m)=d$，且$(a_1,\cdots,a_{n-1},m)=d_1$，则$(d_1,a_n)=d$，由方程得：$a_nx_n+b\equiv 0(\mathrm{mod}\ d_1)$。

这样，把方程分成了两部分。一部分是

$$a_1x_1+\cdots+a_{n-1}x_{n-1}+b_1d_1\equiv 0(\mathrm{mod}\ m)$$

一部分是

$$a_nx_n+b\equiv 0(\mathrm{mod}\ d_1)$$

方程成立当且仅当两部分分别满足，故可以分别求解两个方程。

由归纳假设，第一部分的解数为 $m^{n-2}d_1$（还可以求出它们），第二部分解数为 md/d_1，因此根据乘法原理，总解数为 $md/d_1\times m^{n-2}d_1 = m^{n-1}d$。

高次方程　高次方程 $f(x)=a_nx^n+\cdots+a_0\equiv 0(\mathrm{mod}\ m)$就要麻烦得多。它的解的个数很不规

则（只能证明当重数计算在内它的解数不大于 n），但有如下结论：

若$(m_1,m_2)=1$，则同余方程$f(x)\equiv 0(\bmod m_1m_2)$的解数等于二方程：

$f(x)\equiv 0(\bmod m_1)$，$f(x)\equiv 0(\bmod m_2)$　　(这里的$f(x)$并不一定是多项式)

的解数之积。

这个结论很容易用孙子定理证明（回忆前面介绍的两个简单线性同余方程的情况，显然$f(x)\equiv(\bmod m_1m_2)$是方程组的解，而根据推导的结论，它是惟一的）。

因此，只考虑模是素数的整数幂 p^l。进一步地，在一个特殊条件下，还可以把 p^l 降低为 p。因为有：

定理：若$f(x)\equiv f'(x)\equiv 0(\bmod p)$无解，则$f(x)\equiv 0\ (\bmod p^l)$的解数等于$f(x)\equiv 0\ (\bmod p)$的解数。

其中$f'(x)$定义为当$f(x)$的定义域变为实数集时的$f'(x)$，即$f'(x)=na_nx^{n-1}+\cdots+2a_2x^2+a_1$。

在高次方程中，有一类比较特殊，它就是：

方程 $x^k\equiv n(\bmod p),\ p\nmid n$。直接给出结论，$x^k\equiv n(\bmod p),\ p\nmid n$ 或无解，或有$(k, p-1)$个解。如果有解，称 n 是模 p 的 k 次**剩余**，否则称 n 是模 p 的 k 次**非剩余**。特别地，如果 m 为大于 1 的正整数，n 与 m 互素，若 $x^2\equiv n(\bmod m)$可解，则 n 称为模 m 的**二次剩余**，否则为**二次非剩余**。

设 p 为奇素数，$p\nmid n$，定义 Legendre 符号：

$$\left(\frac{n}{p}\right)=\begin{cases}1, & n\text{为模}p\text{的二次剩余}\\ -1, & n\text{为模}p\text{的二次非剩余}\end{cases}$$

且由定义立刻得：当 $n\equiv n'(\bmod\ p)$时$\left(\frac{n}{p}\right)=\left(\frac{n'}{p}\right)$，故只需要考虑 $n<p$ 的情形。关于二次剩余，有如下基本定理，设 p 为奇素数，模 p 的缩系中有$(p-1)/2$ 个二次剩余，$(p-1)/2$ 个二次非剩余，且 i^2 $(i=1,2,\cdots,(p-1)/2)$就是所有的模 p 二次剩余。

可以证明：Legendre 符号也是积性函数，即若 $p\nmid n$，$p>2$，则

$$\left(\frac{mn}{p}\right)=\left(\frac{m}{p}\right)\left(\frac{n}{p}\right)$$

这样，只需要计算$\left(\frac{-1}{p}\right)\left(\frac{2}{p}\right)\left(\frac{q}{p}\right)$（$q$ 为奇素数），根据 Legendre 符号的积性就能计算所有数 n 的 Legendre 符号了。为此，介绍 3 个定理：

定理一：若 p 为奇素数，则$\left(\frac{-1}{p}\right)=(-1)^{\frac{p-1}{2}}$

即　当且仅当 $p\equiv 1(\bmod 4)$时，2 为模 p 二次剩余。

定理二：若 p 为奇素数，则$\left(\frac{2}{p}\right)=(-1)^{\frac{1}{8}(p^2-1)}$

即　当且仅当 $p\equiv\pm 1(\bmod 8)$时，2 为模 p 二次剩余。

定理三（二次互反律）：设 p,q 为奇素数，且 $p\neq q$，则

$$\left(\frac{p}{q}\right)\left(\frac{q}{p}\right)=(-1)^{\frac{p-1}{2}\times\frac{q-1}{2}}$$

利用这个定理，当 $p>q$ 时利用 $\left(\frac{p}{q}\right)=\left(\frac{p \bmod q}{q}\right)$；当 $p<q$ 时利用互反律转化为 $\left(\frac{q}{p}\right)$，类似于辗转相除法一样不断地缩小 p 和 q，直到可以直接计算出 Legendre 符号为止。

【例题 1】仓库问题[1]

仓库中有 n 种货物，货物标号为 1…n，仓库老板制订了 k 套不同的货物。当某套货物删除或更换某样货物后与另一套货物相同时，认为这两套货物是“类似”的。比如说货物套“1 2 3 4”就货物套“3 2 1”、“1 2 5 3 4”、“1 2 3 4 2”和“1 5 4 3”类似，而和货物套“1 2”、“1 1 2 2 3 4”和“4 5 3 6”不类似。

仓库为 $m(101>m>n>0)$个商店服务，成套运输货物。任意两套“类似”的货物不能送往同一个商店，当然可以不送任何货物去某一家或多家商店。请编程求出如何运送所有的这 k 套货物给 m 家商店。

【分析】

首先，注意到 $m>n>0$ 这个条件。假设有两个相似的集合$\{x_1,x_2,\cdots,x_k,p\}$，$\{x_1,x_2,\cdots,x_k,q\}$。由定义可知 $p\neq q$，但可以为 0。令

$$s_1=x_1+x_2+\cdots+x_k+p,\qquad s_2=x_1+x_2+\cdots+x_k+q$$

由 $0<x_1$，x_2，…，$x_k\leqslant n$，$0\leqslant p$，$q\leqslant n$ 可知，$0<|s_1-s_2|\leqslant n$。也就是说，s_1，s_2 模 $n+1$ 的结果不等。

又因为 $m\geqslant n+1>n$，所以对于给定的集合 S，只需把它分配到 sum(S) mod (n+1)+1 号仓库即可。

【例题 2】二进制 Stirling 数[2]

第二类 Stirling 数 $S(n,m)$表示将一个含有 n 个元素的集合划分成 m 个非空子集的方案个数。例如，将一个含有 4 个元素的集合划分成 2 个非空子集共有 7 种方案：

{1,2,3}∪{4}，{1,2,4}∪{3}，{1,3,4}∪{2}，{2,3,4}∪{1}，{1,2}∪{3,4}，{1,3}∪{2,4}，{1,4}∪{2,3}

以下递归公式可以根据任意给定的 m 和 n 算出 $S(n,m)$。

$S(0,0)=1$；$S(n,0)=0$（当 $n>0$）；$S(0,m)=0$（当 $m>0$）；

$S(n,m)=mS(n-1,m)+S(n-1,m-1)$（当 $n,m>0$）。

你的任务比这更“简单”：给定整数 n 和 m，且 $1\leqslant m\leqslant n\leqslant 10^9$，判断 $S(n,m)$的奇偶性，即计算 $S(n,m)$ mod 2 的值。例如 S (4,2) mod 2=1。

【分析】

在无法推出显式的情况下应当考虑列表观察法，因为它很有可能成为有效和巧妙的解题途径。将 $S(n,m)$（$n\in[1\cdots64]$，$m\in[1\cdots64]$）的所有值用动态规划的办法计算并罗列出来，得到图 2-13，其中■代表 1，□代表 0。

---

[1] 题目来源：Ural State University Problem Archive

[2] 题目来源：ACM/ICPC Regional Contest CERC 2001

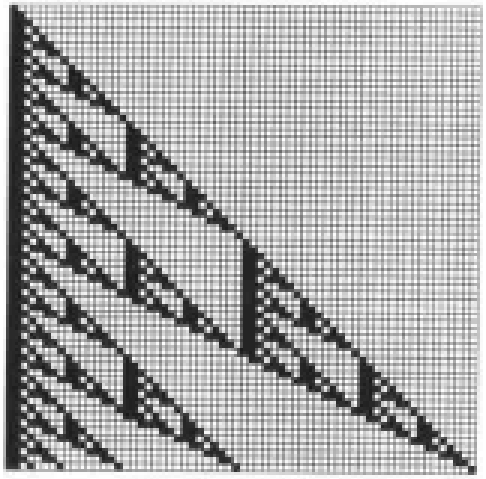


图 2-13　有规律的图形

不难发现，图形是极富规律的：在 \ 对角线的斜下方，每一个顶点坐标为（1,1），（2^k, 2^k），（2^{k-1},1）（$k>1$）的三角形都被三边中点划分成 4 个面积相等的小相似三角形，4 个小三角形中，最中间的那个全部由□组成，其余 3 个图案一模一样，且最上方的那 1 个顶点坐标为（1,1），（2^{k-1}, 2^{k-1}），（2^{k-2},1）。

很明显，如此循环的三角形划分可以递归定义 $S(n,m)$的值，求解方法如下：

（1）如果 $n=m$，那么输出 $S(n,m)=1$ 结束递归；如果 $m=0$ 或者 $n<m$，那么输出 $S(n,m)=0$ 结束递归；

（2）找出最小的正整数 k，使得 $2^k \geqslant m$ 并且 $2^k-(2^k-m+1)\ \mathrm{div}\ 2+1 \geqslant n$；

（3）如果 $m>2^{k-1}$，那么令 $n'=n-2^{k-1}$，$m'=m-2^{k-1}$ 并且转至步骤 5；

（4）如果 $n-m \geqslant 2^{k-2}$，那么令 $n'=n-2^{k-2}$ 并且转至步骤 5；

（5）输出 $S(n,m)=0$ 结束递归；

（6）递归求解并输出 $S(n',m')$的值。

程序编起来很简单，时间复杂度为 $O(\log n)$。事实上，这个公式也不用观察，可以直接证明，读者可以试试。

提示：这道题目和 1.2 节中介绍过的一个重要理论有关，记得是什么吗？

【例题 3】荒岛野人[①]

克里特岛以野人群居而著称。岛上有排列成环行的 M 个山洞。这些山洞顺时针编号为 1,2,…, M。岛上住着 N 个野人，一开始依次住在山洞 $C_1,C_2,\cdots,C_N$ 中，以后每年，第 i 个野人会沿顺时针向前走 P_i 个洞住下来。每个野人 i 有一个寿命值 L_i，即生存的年数。如图 2-14 所示的 4 幅图描述了一个有 6 个山洞，住有 3 个野人的岛上前 4 年的情况。3 个野人初始的洞穴编号依次为 1，2，3；每年要走过的洞穴数依次为 3，7，2；寿命值依次为 4，3，1。

奇怪的是，虽然野人有很多，但没有任何两个野人在有生之年处在同一个山洞中，使得小岛一直保持和平与宁静，这让科学家们很是惊奇。他们想知道，至少有多少个山洞，才能维持岛上的和平呢？

[①] 题目来源：NOI2002，命题人：刘汝佳

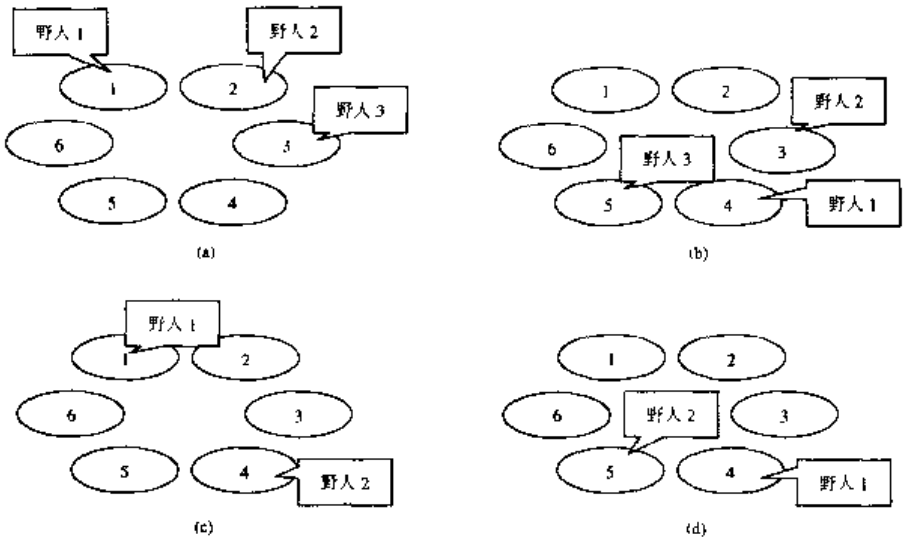


图 2-14　荒岛上连续 4 年的情况

【分析】

为方便叙述，假设野人 i 的初始位置为 P_i，每年跨越 S_i 个山洞，寿命为 T_i。很明显，$M \geqslant \text{Max}\{P_i;\ i \in [1 \cdots n]\}$。

此题的核心问题是对于一个确定的 M，如何求出任一对野人 i 和 j（这里规定 $S_i \geqslant S_j$）的最早相遇时间 Age。

如果相遇时间（不要求是最早相遇时间）为 x，那么一定满足等式：

$$P_i + S_i \times x \equiv P_j + S_j \times x \pmod{M}$$

也就是说 $(S_i - S_j) \times x \equiv (P_j - P_i + M) \pmod{M}$

这是一个线性同余方程，它是刚才已经介绍过的。在本题中 $a = S_i - S_j$，$b = (P_j - P_i + M) \bmod M$，$n = M$，$x,y$ 是待求变量。

如果方程无解，野人 i 和 j 永远不会相遇；否则，可以求出所有解中的最小非负整数 Age，那么，如果 $\text{Age} \leqslant T_i$ 并且 $\text{Age} \leqslant T_j$ 那么野人 i 和 j 会在有生之年相遇，从而打破荒岛的平静。这里的山洞个数 M 不符合试题要求。

这样，只需要从 $\text{Max}\{P_i\}$ 到 10^6 枚举山洞的个数 M，对每一个 M 考虑是否任何两个野人在有生之年不会处在同一山洞中，直到找到符合要求的 M。

练　习　题

编程题：

2.2.8　玻璃弹珠[①]

我有 n 个玻璃弹珠，想要买一些盒子把它们装起来。盒子有两种。

[①] 题目来源：UVA Problem Archive Online Contest

类型 1： 每个盒子花费 c_1，可以装 n_1 颗玻璃弹珠；

类型 2： 每个盒子花费 c_2，可以装 n_2 颗玻璃弹珠。

希望每个盒子都装满，在这个前提下使费用最低。应该如何购买盒子呢？

2.2.9　庆典的日期[1]

古斯迪尔文明曾在约 10 亿年前在地球上辉煌一时，尤其在历法、数学、天文等方面的发展水平已经超过现代。在古城的众多庙宇中，考古人员都发现了一种奇特的建筑，该建筑包含一排独立的房间，如图 2-15 所示。

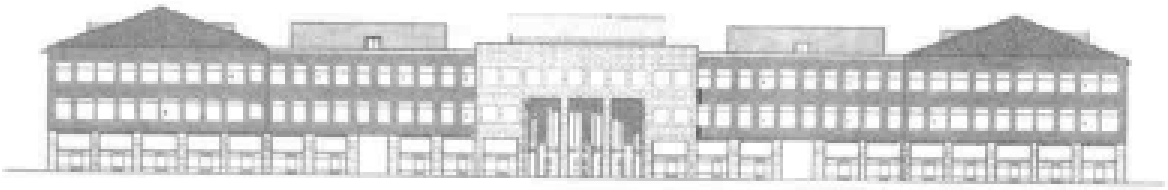


图 2-15　一个奇特的建筑

图 2-16 是一个规模较小的建筑的内部结构，包括 9 个房间。

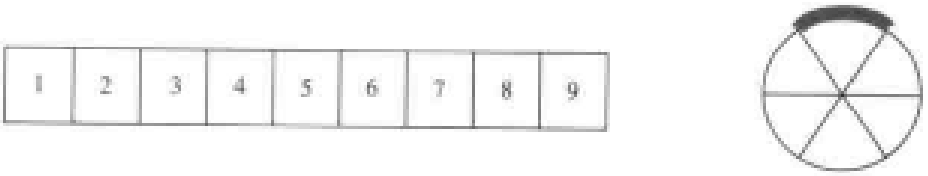


图 2-16　房间和转盘

在每个房间的中央，挂有一个转盘，每个转盘分为 6 个格子，每个格子写着一个 1 到 9 的数字。转盘可以逆时针转动。转盘的红色标记始终指向上方的格子。每个房间的转盘都不相同。

CC 考古工作室近日成功地破译了当时的文字，对进一步研究古斯迪尔文明做出了重要贡献。首先，研究人员翻译了当时的宗教书籍，得知了建筑的用途。原来每个寺院都要在建成以后每隔若干年举行一次大型的庆典。由于“天机不可泄漏”，寺院方面并不直接说明庆典的日期，而是采用“暗示”的方法。奇特的建筑就是为了确定庆典的日期而专门建造的。

房间从左到右编号为 1，2，3，…，n，同时寺院有 n 个祭司也从 1 到 n 编号，这些祭司每年到房间中祈祷一次，建寺那年祭司和自己编号相同的房间祈祷。同时，转盘上红色标记指示的格子的数字就是该祭司第二年祈祷的房间编号。在祭司祈祷完毕以后，将转盘逆时针旋转一格。转盘的设计使得在每年祈祷时，每个房间只有一个祭司。

从建寺以后，当某一年祈祷时，每个祭司的编号都和祈祷房间的编号相同时，就是举行庆典的日期。实际上，每隔若干年，就会有一次庆典。

---

[1] 题目来源：IOI2000 中国国家集训队原创题目，命题人：陈宏

2.2.10　超级马[①]

在一个无限的棋盘上有一个超级马，它可以完成各种动作。每一种动作都是通过两个整数来确定——第一个数说明列的增量（正数向右，负数向左），第二个数说明行的增量（正数向右，负数向左），移动马来完成这个动作。动作种类数不超过 100。

对任意一个超级马进行确认，是否通过自己的行动可以到达盘面上的每一个格子。

2.2.11　火柴问题[②]

有 n 根火柴，甲乙两人轮流从中拿取，一次至少拿一根，至多拿先前对方一次所取火柴数目的两倍。甲先拿，开始甲可以拿任意数目的火柴（但不得拿完）。最先没有火柴可拿的一方为败方。请问，甲能否获胜？（$1<n\leqslant 10^9$）

2.2.12　石子问题[③]

甲乙两人面对若干堆石子，其中每一堆石子的数目可以任意确定。例如共 n=3 堆，其中第一堆的石子数 a_1=3，第二堆石子数 a_2=3，第三堆石子数 a_3=1。两人轮流按下列规则取走一些石子，游戏的规则如下：

- 每一步应取走至少一枚石子；
- 每一步只能从某一堆中取走部分或全部石子；
- 甲乙双方事先约定一个数 m，并且每次取石子的数目不能超过 m 个；
- 如果谁无法按规则取子，谁就是输家。

我们关心的是，对于一个初始局面，究竟是先行者（甲）有必胜策略，还是后行者（乙）有必胜策略。

2.2.13　质数游戏[④]

A 和 B 玩一个质数游戏。初始时计数器 c=0，然后 A 和 B 轮流选择一个 1~n 的正整数并加到 c 里。如果某一方选数以后计数器 c 不是质数，则该游戏者输。例如，A 选 3，B 选 2，A 选 5，则 A 输，因为此时计数器 c=3+2+5=10 不是质数。给出 $n\leqslant 1\,000$，A 是否有必胜策略？如果有，A 第一次应该选什么数？

2.2.14　旋转盘[⑤]

这个难题由一个随意放着一圈 m 个黑色盘子和 n 个白色盘子的带有能翻转（即把顺序反过来）3 个连续的盘子的旋转式栅门的环形轨道构成。

图 1 中，轨道上有 8 个黑色盘子和 10 个白色盘子。

你可以通过转动旋转式栅门来翻转其中的 3 个盘子，或者将轨道上的每个盘子顺时针轮换一个位置，如图 2-17 所示。

这个难题的目标是通过翻转和轮换将同一颜色的盘子集中到相邻的位置上，如图 2-18 所示。

---

① 题目来源：Polish Olympiad in Informatics
② 题目来源：经典问题
③ 题目来源：经典问题
④ 题目来源：经典问题
⑤ 题目来源：ACM/ICPC Regional Contest Taejon 2001

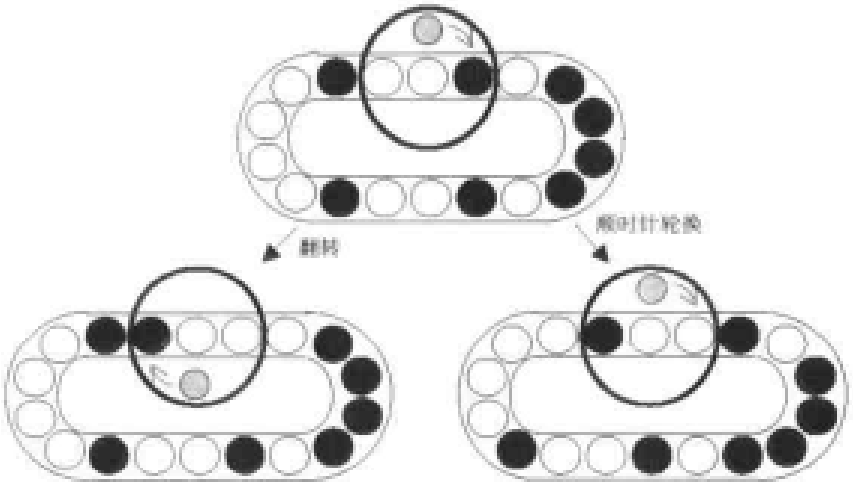


图 2-17　翻转和轮换

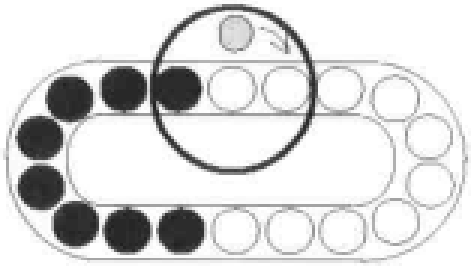


图 2-18　一个目标序列

你需要写一个判定给定的序列是否能达到目标的程序。如果目标可以达到，输出“YES”，否则输出“NO”。

2.2.15　牛场围栏[①]

佳佳计划为他的牛场建一个围栏，以限制奶牛们的活动。他有 $N(N\leqslant 5\,000)$种可以建造围栏的木料，长度分别是 $l_1, l_2, \cdots, l_N$，每种长度的木料无限。修建时，他将把所有选中的木料拼接在一起，因此围栏的长度就是他使用的木料长度之和。但是，聪明的佳佳很快发现很多长度都是不能由这些木料长度相加得到的，于是决定在必要的时候把这些木料砍掉一部分以后再使用。不过由于佳佳比较节约，他给自己规定：任何一根木料最多只能削短 M 米。当然，每根木料削去的木料长度不需要都一样。不过由于测量工具太原始，佳佳只能准确的削去整数米的木料，因此如果他有两种长度分别是 7 米和 11 米的木料，每根最多只能砍掉 1 米，那么实际上就有 4 种可以使用的木料长度，分别是 6、7、10、11 米。

牛牛是佳佳的牛场中的最聪明的奶牛，佳佳请她来设计围栏。牛牛不愿意自己和同伴在游戏时受到围栏的限制，于是想刁难一下佳佳，希望佳佳的木料无论经过怎样的加工，长度之和都不可能得到他设计的围栏总长度。

不过牛牛知道，如果围栏的长度太小，佳佳很快就能发现他是不能修建好的。因此他希望得到你的帮助，找出无法修建的最大围栏长度。

[①] 题目来源：IOI2002 中国国家集训队冬令营试题. 命题人：李劼明

2.3　组合数学初步

1666年莱布尼兹所著《组合学论文》一书问世，这是组合数学的第一部专著，书中首次使用了组合论（Combinatorics）一词。组合数学的蓬勃发展则是在计算机问世和普遍应用之后。由于组合数学涉及面广，内容庞杂，并且仍在迅速发展中，因而还没有一个统一而有效的理论体系。

组合数学问题随处可见，它的历史渊源扎根于古老的数学娱乐和游戏之中，而在当今社会中同样发挥着重要的作用。简单地说，组合数学研究一个集合的物体进行满足一些规则的排列。具体地来说，组合数学往往研究的是这些排列的存在性、计数和分类。有时候还需要构造出满足条件的一个或最优的排列。归纳起来，可以一般描述为：组合数学是研究离散结构的存在、计数、分析和优化等问题的一门学科。其中一些典型的问题，读者可以通过后面的例子中看到。

本节介绍组合数学的基本知识。由于介绍组合的好书很多，本书没有必要，也不可能介绍和这些书籍相同的内容。本节希望有自己的特点，因此在写作的时候尽量加强以下两个方面：一是整体轮廓。很多读者（特别是时间不多的）在初学组合数学的时候往往找不到方向，不知道如此多的内容各是为了解决什么问题，采取的是什么方法。本节略去所有证明而只保留组合数学的基本框架，让读者尽可能快地对整体轮廓有一个了解，然后选择一本好书进行深入学习。二是实例分析。本节的例子不多，但是和很多书籍中的例子不同。读者如果结合起来阅读会更容易理解这些内容。

前三节的内容都很基本，应熟练掌握；递推和生成函数应用比较广泛，而且比较灵活，读者可以仔细体会本节中的几个例子。Pólya 原理应用也很广泛，但是不易理解，本节尝试用“三步法”让大家了解定理的内容和程序设计方法，而具体的算法证明请读者阅读相关书籍。大部分读者对离散变换比较陌生，本节力求用尽量少的篇幅讲清反演的动机和基本定理，具体的定理证明请阅读相关书籍。Möbius 反演是该小节的重点。

2.3.1　鸽笼原理和Ramsey定理

在这一节中，来看一下一些鸽笼原理（pigeon hole principle）。

基本原理　如果 $n+1$ 个物体放进 n 个盒子，那么至少有一个盒子包含两个或者更多的物体。考虑一个问题：给出 n 个数，从中选出若干个数使得它们的和为 n 的倍数。有一个简单的方法如下，虽然题目没有限制怎样选数，但是我们可以证明：存在若干个连续数，它们的和是 n 的倍数。为此，我们考察和 $S_k=a_1+a_2+\cdots+a_k$。如果存在一个 S_k 是 n 的倍数，那么把前 k 个数选出来就可以了。否则所有 n 个 S_k（n 个物体）除以 n 的余数只有 $1,2,3,\cdots,n-1$ 这 $n-1$ 种可能（$n-1$ 个盒子），由基本原理知，必然有两个不同的和 S_i 和 S_j（$i<j$）除

以 n 的余数相同，故部分和 $S_j-S_i=a_{i+1}+\cdots+a_j$ 是 n 的倍数。

加强形式　令 q_1，q_2，…，q_n 为正整数。如果将 $q_1+q_2+\cdots+q_n-n+1$ 个物体放入 n 个盒子中，那么或者第一个盒子至少有 q_1 个物体，或者第二个盒子至少有 q_2 个物体，或者第 n 个盒子至少有 q_n 个物体。

平均定理　如果 n 个非负整数 m_1，m_2，…，m_n 的平均数至少等于 r，那么这 n 个整数 m_1，m_2，…，m_n 至少有一个满足 $m_i\geq r$。

Ramsey 定理　如果 $m\geq 2$，$n\geq 2$ 是两个整数，则存在一个正整数 p 使得如果给完全图 K_p 的顶点着红色或者蓝色，则一定存在红色的 K_m 子图或者蓝色的 K_n 子图。对于 $m\geq 2$，$n\geq 2$，这个定理断言了 p 的存在性。把满足条件的最小的 p 记做 $r(m, n)$，称为 ramsey 数。

提示：Ramsey 定理可以从两个数推广到多个数的情况，也可以从给边（点对）着色推广到给所有含 t 种元素的子集着色的情况。值得注意的是 $t=1$ 就是鸽笼原理的加强形式。所以把 Ramsey 定理看成是鸽笼原理的推广。几乎所有的组合数学书中都可以查到比较小的 Ramsay 数，这里就不重复了。

需要特别提到的是这些定理在计算机解题中的作用。可能有读者认为这几个定理没有提供任何实际问题的算法，只是一个存在性的证明，因而对解决实际问题没有什么意义。这样的想法是错误的。从刚才的例子可以看到，存在性可以成为算法设计的依据。后面还将看到，存在性也可以成为复杂度估计的依据。对于复杂的问题，利用 Ramsay 定理得到一个存在性往往是解决问题的突破口。

2.3.2　排列组合和容斥原理

由于本书并不是一本数学书，对于相关知识点，本书只给出结论，且不证明，有兴趣的读者可以阅读相关书籍。

1. 加法原理和乘法原理

首先复习两条最基本的原理：

加法原理（addition principle）　把事情分成 N 类，每种有 C_i 种做法，则该事情共有 $C_1+C_2+\cdots+C_N$ 种方法。

乘法原理（multiplication principle）　把事情分成 N 步，每步有 C_i 种做法，则该事情共有 $C_1C_2\cdots C_N$ 种方法。

可以利用加法原理得到一个“减法”原理，即用总数减去不符合要求的。

【例题 1】单色三角形[1]

给定空间里的 n 个点，其中没有三点共线。每两个点之间都用红色或黑色线段连接。如果一个三角形的三条边同色，则称这个三角形是单色三角形。对于给定的红色线段的列表，希望能找出单色三角形的个数。

[1] 题目来源：Polish Olympiad in Informatics

【分析】

所有三角形的个数为C_n^3，由于三角形只有单色和非单色两种，所以根据“减法原理”，如果求出非单色三角形的个数，就求出了单色三角形的个数。注意到，如果$\triangle ABC$是非单色三角形，则A，B，C三个点中恰好两个点连接的两条线段不同色。而如果$\triangle ABC$是单色三角形，则A，B，C三个点中，每个点连接的两条线段同色。这样，非单色三角形的个数等于由一个点连接两条非同色线段的情况数的一半。

设第i个点连接的红色线段数为r_i，由于每个点只引出红色或黑色的线段，故黑色线段数目为$n-1-r_i$。由乘法原理，第i个点连接两条非同色线段的情况数为$r_i(n-1-r_i)$。所以，非单色三角形的个数$\text{count}=\frac{1}{2}\sum_{i=1}^{n} r_i(n-1-r_i)$，而单色三角形数$=C_n^3-\text{count}$。

需要注意的是，两个原理的地位并不是相等的，有理由把乘法原理看成是加法原理的简单推论。而加法原理的一个深化是后面要介绍的容斥原理。

回忆一些经典问题的做法。首先，讨论最简单的排列组合问题，得到：

问题一：排列数　N个不同物体不重复地取M个作排列的方法数$P(N,M)=N!/(N-M)!$。

问题二：组合数　N个不同物体不重复地取M个作组合的方法数$C(N,M)=N!/(N-M)!M!$。

Stirling公式　为了计算$n!$，当n很大的时候，有Stirling近似公式：

$$n!\sim\sqrt{2\pi n}\left(\frac{\pi}{e}\right)^n$$

多重集的情形　如果只借助基本排列组合公式和两个基本原理，只能得出两个看起来并不太完美的结论：

问题三：（1）k种元素，重数分别为n_1，n_2，…，n_k，所有$n=n_1+n_2+\cdots+n_k$个元素的**全排列数**为$n!/(n_1!n_2!\cdots n_k!)$。

问题四：（2）k种元素，**重数均为无限大**，选r个的组合数为$C(r+k-1,r)$。

这两个结论只是两个特殊情况，没有对一般情况给出很好的结果。虽然可以利用容斥原理求出重数均为给定数n_i的r组合数，但是无法把它写成简洁的数学表达式。

幸运的是，借助生成函数，可以解决更为复杂的问题。

问题：一般排列组合问题

有k种元素，重数均为无限大。规定第i种元素选取的个数c_i必须属于一个给定的集合S_i，求r排列数和r组合数，并列举出方案。

显然，刚才介绍的四种问题都是它的特殊情况。其中，问题一、二的$S_i=\{0,1\}$，而问题三的$S_i=\{n_i\}$，问题四的$S_i=\{0,1,2,\cdots\}$。具体的方法将在“生成函数”一小节中介绍。

需要特别提醒读者的是：这些问题看起来是很相似的，在套用公式之前必须看清楚问题到底是怎样的。集合是可重的吗？求排列数还是组合数？当问题是一般排列组合问题的特殊情形1，2，3，4时，可以用简单方法计算，而如果是一般情形，就必须用到生成函数了。

2. 容斥原理

作为加法原理的推广，有两个容斥原理（inclusion and exclusion principle）和两个推广的公式。这四个公式的证明略去，读者可以在大部分组合数学书籍中找到。

公式 1　容斥原理（简单形式）

设 S 是一个集合，A_1，A_2，…，A_q 是 S 的一些有限子集，则有

$$\left|\bigcup_{i=1}^{q} A_i\right| = \sum_{i=1}^{q}|A_i| - \sum_{1\le i<j\le q}\left|A_i \cap A_j\right| + \cdots + (-1)^{q+1}\left|\bigcap_{i=1}^{q} A_i\right|$$

$$\left|\bigcap_{i=1}^{q} A_i\right| = \sum_{i=1}^{q}|A_i| - \sum_{1\le i<j\le q}\left|A_i \cup A_j\right| + \cdots + (-1)^{q+1}\left|\bigcup_{i=1}^{q} A_i\right|$$

公式 2　容斥原理（加权形式）

设 S 是一个集合，$\forall x\in S$ 定义权函数（取正值的函数）$m(x)$，A_1，A_2，…，A_q 为 S 的有限子集。子集 A 的权定义为属于 A 的所有元素的权值之和，令 $Q=\{1,2,3,\cdots,q\}$，则有

$$m(\bigcup_{i\in Q} A_i) = \sum_{k=1}^{q}(-1)^{k+1}\sum_{\substack{K\subset Q\\ |K|=k}} m(\bigcap_{i\in K} A_i) \qquad m(\bigcap_{i\in Q} A_i) = \sum_{k=1}^{q}(-1)^{k+1}\sum_{\substack{K\subseteq Q\\ |K|=k}} m(\bigcup_{i\in K} A_i)$$

公式 3　Jordan 筛法公式

设 A_1，A_2，A_3，…，A_q 为有限集合 X 的子集，在 X 中**所有恰属于 P 个集合的**元素的权和为：

$$M_q^p = \sum_{k=p}^{q}(-1)^{k-p}C_k^p \sum_{\substack{K\subset Q\\ |K|=k}} m(\bigcap_{i\in K} A_i)$$

特别地，当 p=0 时取 $m(\bigcap_{i\in\varnothing} A_i) = m(X)$，则有

公式 4：Sylvester 公式

设 A_1，A_2，A_3，…，A_q 为有限集合 X 的子集，在 X 中**不属于任何集合**的元素的权和为

$$M_q^0 = m(X) + \sum_{K\subseteq Q}(-1)^{|K|} m(\bigcap_{i\in K} A_i)$$

问题：在数论中，“小于 n 且与 n 互素的数的个数”叫做 n 的欧拉 φ 函数。设 n 的素因子标准分解式为

$$n = p_1^{t_1} p_2^{t_2} \cdots p_q^{t_q}$$

给出欧拉函数的表达式。

【分析】令 A_i 为不大于 n 且是 p_i 倍数的正整数集合，则

$$|A_i| = \frac{n}{p_i},\quad \left|A_i \cap A_j\right| = \frac{n}{p_i p_j},\cdots$$

由 Sylvester 公式可得

$$\varphi(n) = n(1-\frac{1}{p_1})(1-\frac{1}{p_2})\cdots(1-\frac{1}{p_q})$$

3. 棋盘多项式

互不攻击的象[①]

图 2-19 是一个国际象棋棋盘，两个“象”互不攻击当且仅当它们不处于同一条斜线上。例如图中 B_1 和 B_2 互相攻击，但是 B_1 和 B_3，B_2 和 B_3 都不互相攻击。

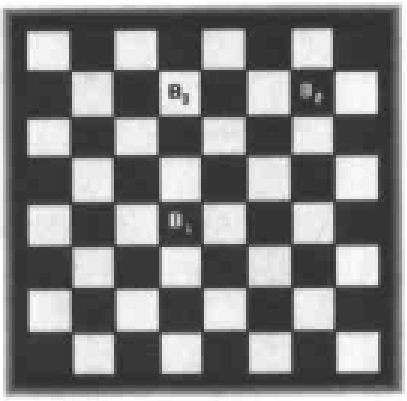


图 2-19　国际象棋棋盘和“象”

在一个 $n\times n(n\leqslant 30)$的棋盘上放 k 个互不攻击的象有多少种方法？例如，当 $n=8$，$k=6$ 时有 5 599 888 种方法。

在解决这个问题之前，来看几个概念。

布棋方案数 R_k（C）　设 C 是一个棋盘，$R_k(C)$表示把 k 个相同的棋子布到 C 中且任意两个棋子不能处于同一行或者同一列的方法数。并规定对任意的棋盘 C 有 $R_0(C)=1$。

棋盘多项式　设 C 是棋盘，则 $R(C)=\sum_{k=0}^{\infty}R_k(C)x^k$ 叫做它的**棋盘多项式**。

可以证明布棋方案数 R_k（C）具有下面的性质：

（1）对于任意的棋盘 C 和正整数 k，如果 k 大于 C 中的方格总数，则 R（C）$=0$。

（2）R_1（C）等于 C 中的方格数。

（3）设 C_1 和 C_2 是两个棋盘，如果 C_1 经过旋转或者翻转变成了 C_2，则 $R_k(C_1)=R_k(C_2)$。

（4）设 C_1 是从棋盘 C 中去掉指定的方格所在的行和列以后剩余的棋盘，C_1 是从棋盘 C 中去掉指定的方格以后剩余的棋盘，则有 $R_k(C)=R_{k-1}(C_1)+R_k(C_1)$ $(k\geqslant 1)$。

（5）设棋盘 C 由两个子棋盘 C_1 和 C_2 组成，如果 C_1 和 C_2 的布棋方案是互相独立的，则有 $R_k(C)=\sum_{i=0}^{k}R_i(C_1)R_{k-i}(C_2)$。

显然，在上述定义中当 k 大于棋盘的格子数时 R_k（C）$=0$，所以 R（C）一般只有有限项。例如：R（  ）$=R_0$（）$+R_1$（）$x+R_2$（）$x^2=1+2x+x^2$

根据 R_k（C）的性质不难得到 R（C）的性质。

[①] 题目来源：UVA Problem Archive Online Contest

（1）$R(C)=xR(C_i)+R(C_l)$，其中 C_i 和 C_l 的定义如前所述。

（2）$R(C)=R(C_i)\times R(C_l)$，其中 C_i 和 C_l 的定义如前所述。

利用这两条性质可以计算棋盘多项式。

例如：$R(\square\!\square)=xR(\square)+R(\square^{\square})=x(1+x)+(1+2x)=1+3x+x^2$

有禁区的排列问题　用棋盘多项式可以解决有禁区的排列问题，因为 $X=\{1, 2, 3, \cdots, n\}$的一个排列恰好对应了 n 个棋子在 $n\times n$ 棋盘上的一种排布（想一想，为什么）。如果在排列中限制元素 i 不能放在第 j 个位置，则相应的布棋方案中的棋盘第 i 行第 j 列就不能放置棋子。把所有这些不许放置棋的方格称作**禁区**。

用容斥原理可以证明以下定理。

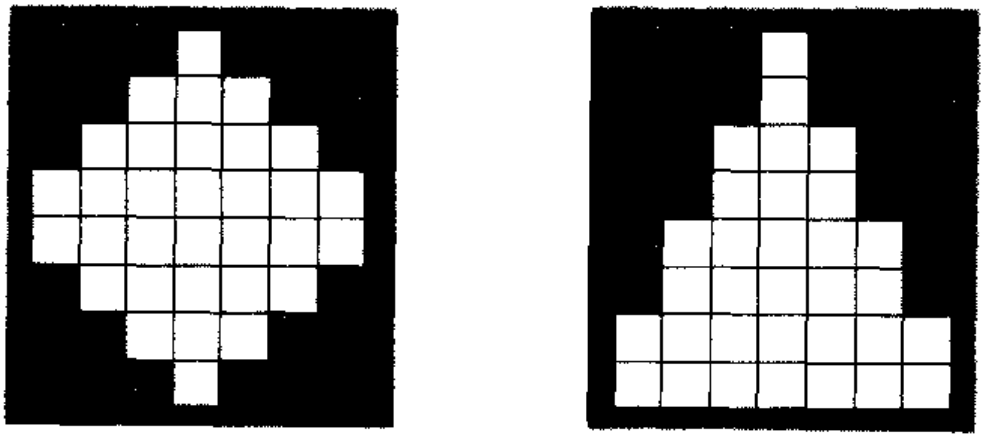
定理：设 C 是 $n\times n$ 的具有给定禁区的棋盘，这个禁区对应集合$\{1,2, \cdots, n\}$中的元素在排列中不允许出现的位置。则这种有禁区的排列数是 $n!-r_1(n-1)!+r_2(n-2)!+\cdots+(-1)^n r_n$，其中 r_i 是 i 个棋子放置到禁区的方案数。

需要说明一点，这个定理适用于 $n\times n$ 棋盘的小禁区的布棋问题。如果是 $m\times n$ 的棋盘或者是禁区很大的布棋问题，那么只能直接用 $R(C)$ 来求解。

【分析】

回到刚才的问题，注意到白格象和黑格象不会互相攻击，所以问题可以化解为白格放 $i(i\leqslant k)$个象和黑格放 $k-i$ 个象分别处理，并且处理方法是一样的。

为了直观，把白格从棋盘中抽出，小方格和棋盘都顺时针旋转 45 度，再压缩，或者把黑格从棋盘中抽出，小方格和棋盘都逆时针旋转 45 度，再压缩，均成为图 2-20(a)。希望知道在这样一个棋盘中放置 i（或者 $k-i$）个两两不同行不同列的象有多少种放法。由于行列关系只是相对的，行列的绝对顺序无关紧要，所以把宽度小的行置于宽度大的行之上，将图 2-20(a)转换成图 2-20(b)。



(a)　　(b)

图 2-20　转化后的棋盘

剩下的工作就不用多说了。

练　习　题

编程题：

2.3.1　电子锁[①]

某机要部门安装了电子锁。$M(M\leqslant 7)$工作人员每人发一张磁卡，卡上有开锁的密码特征。为了确保安全，规定至少要有 $N(N\leqslant 4)$个人同时使用各自的磁卡才能将锁打开。

现在需要计算一下，电子锁上至少要有多少种特征，每个人的磁卡上至少有几个特征。如果特征的编号以小写字符表示，将每个人的磁卡的特征编号打印出来，要求输出的电子锁的总特征数量最少。例如 $M=3$，$N=2$，则电子锁上要有三种特征，个人的磁卡上要有两种特征。

2.3.2　彩票[②]

大街上到处在卖彩票，一元钱一张。购买撕开它上面的锡箔，你会看到一个漂亮的图案。图案有 n 种，如果你收集到所有 n 种彩票，就可以得大奖。请问，在平均情况下，需要买多少张彩票才能得到大奖呢？

2.3.3　群论与Pólya定理

本节介绍 Pólya 定理。为此，先回顾置换群的概念，然后介绍定理的内容和应用。

1．置换群

【例题 1】传球游戏[③]

佳佳最喜欢与他老师和小伙伴们一起玩一种传球游戏。游戏开始时所有小朋友们分成两组，每组 n 人，围成一个圈。每一个小朋友都有一个编号($1,\cdots,n$ 之间)，这个编号在其所在组中是惟一的。游戏开始之前，每个小朋友都有一个球，球上有编号($1,\cdots,n$ 之间)，并且一个组中的球不会有两个相同编号。然后，所有小朋友必须闭上眼睛，游戏开始。随着老师口中的哨子发出的节奏，每个小朋友都用一只手把球传到右边，而用另一只手接左边的来球。

突然，老师的哨子停了，关键的时刻到了。小朋友马上睁开眼睛，开始与同组的小朋友之间进行传球，争取以最短的时间把球传到位。传到位是指一个组中的每一个小朋友手上的球的编号与他自己的编号相同。最后获胜的就是那个最先把球传到位的组。如果一旦哪方出现传球失误(球没被接到而落地)，或犯规（一个人手上拿两个或两个以上的球）这

① 题目来源：NOI 92
② 题目来源：UVA Problem Archive
③ 题目来源：IOI2000 中国国家集训队原创题目，命题人：郭一

一组就被判输。

这个游戏非常有趣，小朋友们玩了许多次。聪明的佳佳总结出一条经验：**总是两个人之间对传**。也就是说，不会出现 a 把球传给 b，而 b 没有把球传给 a 的这种情况。这样可以避免小朋友之间的失误与犯规。不过还有个关键问题就是怎么传，究竟应该把手上的球传给谁？

现在需要你编一个程序来帮助小朋友们确定传球方法。你的程序首先需要计算出从一种初始状态开始。

子问题 1：至少需要几次对传才能将球传到位。

子问题 2：至少需要多少时间才能将球传到位。每一个时间单位一个小朋友可以不做任何动作，也可以与另外一个小朋友之间进行对传。

注意有些对传可以同时进行。比如小朋友 1 与小朋友 2 之间的对传和小朋友 3 与小朋友 4 之间的对传就可以在一个时间单位之内完成，但是被计作两次对传。

【分析】

在解决这个问题之前，需要先学习一些群论，特别是置换群知识。和上一节一样，我们不准备对理论知识做太多的说明，而只给出一些相关结论。

群　给定一个集合 $G=\{a,b,c,\cdots\}$和集合 G 上的二元运算，并满足：

①封闭性：$\forall a,b\in G, \exists c\in G, a\cdot b=c$；

②结合律：$\forall a,b,c\in G, (a\cdot b)\cdot c=a\cdot(b\cdot c)$；

③单位元：$\exists e\in G, \forall a\in G, a\cdot e=e\cdot a=a$；

④逆元：$\forall a\in G, \exists b\in G, ab=ba=e$，记 $b=a^{-1}$。

则称集合 G 在运算“·”之下是一个群，简称 G 是群。一般 $a\cdot b$ 简写为 ab。

置换　n 个元素 1，2，…，n 之间的一个置换为

$$\begin{pmatrix}1 & 2 & \cdots & n\\ a_1 & a_2 & \cdots & a_n\end{pmatrix}$$

表示 1 被 1 到 n 中的某个数 a_1 取代，2 被 1 到 n 中的某个数 a_2 取代，直到 n 被 1 到 n 中的某个数 a_n 取代，且 a_1，a_2，…，a_n 互不相同。

置换群　置换群的**元素是置换，运算是置换的连接**。例如：

$$\begin{pmatrix}1 & 2 & 3 & 4\\ 3 & 1 & 2 & 4\end{pmatrix}\begin{pmatrix}1 & 2 & 3 & 4\\ 4 & 3 & 2 & 1\end{pmatrix}$$

$$=\begin{pmatrix}1 & 2 & 3 & 4\\ 3 & 1 & 2 & 4\end{pmatrix}\begin{pmatrix}3 & 1 & 2 & 4\\ 2 & 4 & 3 & 1\end{pmatrix}=\begin{pmatrix}1 & 2 & 3 & 4\\ 2 & 4 & 3 & 1\end{pmatrix}$$

可以验证置换群满足群的四个条件。

循环　记

$$(a_1a_2\cdots a_n)=\begin{pmatrix}a_1 & a_2 & \cdots & a_{n-1} & a_n\\ a_2 & a_3 & \cdots & a_n & a_1\end{pmatrix}$$

称为 n 阶循环。**每个置换都可以写若干互不相交的循环的乘积**，两个循环$(a_1\ a_2\cdots a_n)$和$(b_1$

$b_2\cdots b_n$)互不相交是指 $a_i\neq b_j$, $i,j=1,2,\cdots,n$。例如：

$$\begin{pmatrix}1&2&3&4&5&6\\3&5&6&4&2&1\end{pmatrix}=(136)(25)(4)$$

这样的表示是惟一的，置换的循环节数是上述表示中循环的**个数**，例如(136)(25)(4)的循环节数为 3。循环也称轮换。

回到刚才的问题，把初始状态表示成如下的置换形式：

$$\begin{pmatrix}1&2&3&\cdots&n\\a_1&a_2&a_3&\cdots&a_n\end{pmatrix}$$

表示 1 号小朋友手上拿着 a_1 号的球，2 号小朋友手上拿着 a_2 号的球…，n 号小朋友手上拿着 n 号的球。根据群论知识，可以把它惟一地表示成若干轮换的积。而每两个小朋友之间的对传球就相当于在原置换的基础上乘上一个对换。现在考虑在**轮换上乘上一个对换**的结果。

（1）对换的两个元素分属于不同的轮换中

设两个轮**换**为（a_1，a_2，a_3，…，a_n），（b_1，b_2，b_3，…，b_m），乘上的对换不妨规定为(a_1, b_1)（轮换中任何一个数字都可以写在最前面），容易验证结果为（b_1，b_2，…，b_n，a_1，a_2，…，a_n）。

结论：两个**轮换合并**成了一个轮**换**。

（2）对换的两个元素分属于同一个轮换中

不妨设在轮**换**（a_1，a_2，…，a_i，a_{i+1}，…，a_n）上乘一个（a_1，a_i），结果为（a_1，a_2，…，a_{i-1}）（a_i，a_{i+1}，…，a_n）

结论：一个轮换**分拆**成了两个轮换。

我们的目的状态是（1）（2）（3）…（n），所以显然每次对换一定是在同一个轮换中的两个元素之间进行。由此可见，无论如何传球，只要每次传球的两个人都在同一个轮换内，所需传球次数总是一样的，即 n 轮换个数。这就是子问题 1 的解。

余下的问题就是怎么把一个轮换以最少的步数拆开。即

$$(a_1,a_2,a_3,\cdots,a_n)\Longrightarrow(a_1)(a_2)(a_3)\cdots(a_n)$$

每步可以乘上若干个互个相交的对换。事实上，这个问题的结论出人意料地简单：

若原轮换的长度为 1，次数为 0；若原轮换的长度为 2，次数为 1，否则次数为 2。

前两个结论显然成立，对于长度大于 2 的轮换，有一个一般的方法：

发现（a_1，a_2，…，a_n）（a_2，a_n）=（a_1，a_n）（a_2，a_3，…，a_{n-1}）=（a_1，a_n）（a_2，a_{n-1}）（a_3，a_4，…，a_{n-2}）=…

所以只需一步就可把一个长度大于 2 的轮换拆成一系列对换之积（最后可能有一个长度为 1 的轮换）。接下来也只需一步即可变成目的状态，至此子问题 2 得以解决。

【例题 2】无聊的排序[①]

你弟弟有一项家庭作业需要你帮助完成。老师给了他一列数，需要他把这些数按升序排列。你可以每次交换两个数的位置，而一次交换的代价被定义成交换的两个数的和。写

[①] 题目来源：ACM/ICPC World Finals 2002

一个程序，用最小的交换代价和来帮助弟弟完成这项无聊的排序工作。

【分析】

本题可以抽象为把一列数从初始状态变成目标状态，即完成一个置换。根据群论知识，置换可以分解为 s 个不相交的循环的乘积。例如，初始状态为 8 4 5 3 2 7，而目标状态为 2 3 4 5 7 8，则可以分解为两个循环的乘积，即（8 2 7）（4 3 5）。

显然，由于每次只有被交换的两个数的位置改变，所以要想改变一个数的位置，只能通过交换完成，而不能像插入排序一样，可以借助其他数来完成，即各个循环是相互独立的，所以应该依次完成每个循环。

对于任意一个循环 i，设它的长度为 k_i，容易证明至少需要交换 k_i-1 次，即每次让一个元素到达目标位置，而当第 k_i-1 个元素到达目标以后显然第 k_i 个也已经到达目标。既然交换次数一定，应该让每次交换的代价尽量少。显然每个元素至少要交换一次，因此一个较好的方法是让循环中最小元素 t_i 参加所有的交换，其他元素只各参加一次，总花费为 $sum_i+(k_i-2)t_i$，其中 sum_i 为循环 i 中所有数的和。

回到刚才的例子。在循环(8 2 7)中，2 是其中最小的数。由于 2 占据了 7 的位置，所以 2 和 7 交换，得到序列 A = 8，4，5，3，7，2，这时 2 占据了 8 的位置，故 2 和 8 交换，得到序列 A = 2，4，5，3，7，8。此时循环里的各个数都到达了目标位置，称该循环已完成。

不过细心的读者可能已经发现，这样的方法不一定是最优的，因为我们可以借助循环外的数和它们交换。具体的说，我们可以让 t_i 先和所有 n 个数中的最小值 m 交换，让 m 进入循环，并和剩下的 k_i-1 个元素各交换一次把它们送入目标位置，最后再让 m 和 t_i 交换，退出该循环。显然第二种方法的花费为 $sum_i+t_i+(k_i+1)m$，它有可能比第一种方法优。例如初始状态为 1，8，9，7，6，目标状态为 1，6，7，8，9，可分解为（1）（8 6 9 7）。第一个循环只有一个元素，忽略；第二个循环如果按照第一种方法，花费为 6+7+8+9+(4-2)×6=42，而第二种花费仅为 6+7+8+9+6+（4+1）×1=41。如表 2-4 所示。

表 2-4　最优方案

操　作　数	结 果 序 列	说　　明
1,6	6,8,9,7,1	全局最小值进入循环
1,9	6,8,1,7,9	1 占据了 9 的目标位置
1,7	6,8,7,1,9	1 占据了 7 的目标位置
1,8	6,1,7,8,9	1 占据了 8 的目标位置
1,6	1,6,7,8,9	1 退出循环，6 重新进入

综合两种方法，总花费为：$\cos t = \text{sum} + \sum_{i=1}^{s} \min\{(k_i-2)t_i, t_i+(k_i+1)m\}$

从这个表达式中我们可以看到，在找循环节的时候只需要记录每个循环节长度 k_i 和它的最小元素 t_i，不需要模拟交换操作。由于元素都是不大于 1000 的正整数，可以用线性时间复杂度的计数排序。由于找循环最多每个元素访问一次，所以总的时间复杂度也是线性

的，空间占用很小。

2. Pólya定理

下面介绍 Pólya 定理。虽然这并不是个特别复杂的定理，但是为了帮助大家理解，把这个定理的介绍分为三个步骤。其中有些概念并不是很容易理解，建议大家不要一开始就抓住一些细节不放，而是反复阅读，并认真验证书中给出的例子。

步骤1　提出问题

对 2×2 的方阵用黑白两种颜色涂色，问能得到多少种不同的图像？经过顺时针旋转使之吻合的两种方案，算是同一种方案（为了叙述方便，四个格子编号如图 2-21 所汇款单）。

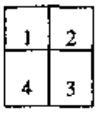


图 2-21　格子编号

下面来解释这道题目的意思。先不考虑旋转，列举出 16 种方案，如图 2-22 所示。

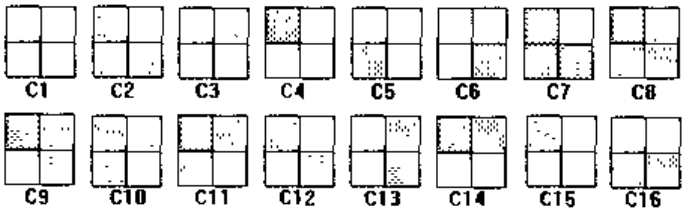


图 2-22　16 种方案

根据方阵的旋转方法一共有 4 种：旋转 0 度、旋转 90 度、旋转 180 度和旋转 270 度，经过尝试发现其中互异的一共只有 6 种：C3、C4、C5、C6 是可以通过旋转相互变化而得，算作同一种；C7、C8、C9、C10 是同一种；C11、C12 是同一种；C13、C14、C15、C16 也是同一种；C1 和 C2 是各自独立的两种。于是，得到了下列 6 种不同的方案，如图 2-23 所示。

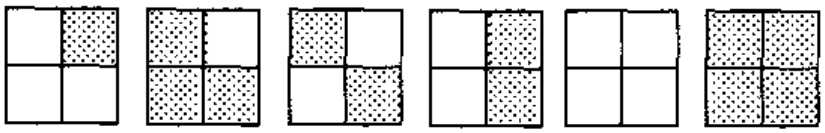


图 2-23　6 种本质不同的方案

当然，这个枚举法只是为了帮助大家理解题意。当问题由 2×2 的方阵变成 20×20 甚至 200×200 的方阵，就不能再一一枚举了，需要找到更好方法。把这个问题一般化得到：

问题：着色方案计数问题

给出一个有 p 个元素的集合 S，用 k 种颜色给这些元素着色，求本质不同的方案个数。为了定义“本质不同”的含义，需要给出一个关于这 p 个元素的置换群 G，对于 G 中的任意置换 f，每个方案 a 和它在 f 作用下的结果被看成是本质相同的。

注意，根据群论知识可以证明，“本质相同”关系是一个等价关系，因此该关系把着色方案分成了若干个等价类，目的就是要数出等价类的数目。

对于刚才的问题中，S 包含四个格子 1，2，3，4，k=2，有 4 个置换。

$$\text{转 } 0^\circ：\ f_1=\begin{pmatrix}1&2&3&4\\1&2&3&4\end{pmatrix}\qquad \text{转 } 90^\circ：\ f_2=\begin{pmatrix}1&2&3&4\\2&3&4&1\end{pmatrix}$$

$$\text{转 } 180^\circ：\ f_3=\begin{pmatrix}1&2&3&4\\3&4&1&2\end{pmatrix}\qquad \text{转 } 270^\circ：\ f_4=\begin{pmatrix}1&2&3&4\\4&1&2&3\end{pmatrix}$$

因此，在本题中置换群 G={转 0°，转 90°，转 180°，转 270°}

特别提示：如果你对“置换”和“置换群”的概念理解得不够深刻，这个例子可以让你对“置换”有一个感性认识。例如转 90° 的置换 f_2 表示的是“格子 1 变成格子 2，格子 2 变成格子 3，格子 3 变成格子 4，格子 4 变成格子 1……”。简单地说，置换可以看成是一种重新排列。我们只是把它写成了一个简单的形式，在任何一列中，如果上面的元素是 a，下面的元素是 b，那么该置换把方案 a 变成了方案 b。至于“置换群”，它仅仅是若干置换的集合。在这个集合上定义了运算“连接”（即，连续做两次置换），成为一个群。

步骤 2　Burnside 引理

一般来说，置换会使一个着色方案变成另外一个（例如，方案 3 在 f_2 作用下变成了方案 6），但是有些方案比较特殊，它在某些置换下变换到它自身（例如方案 12 在 f_3 作用下不变），或者说它在置换 f 下不变。如果记 $C(f)$ 为在置换 f 下保持不变的着色方案个数，那么可以证明：“本质不同的着色方案数为所有置换 f 的 $C(f)$ 值的平均数”。这个结论写得正式一些就是以下引理。

定理 2.3.1　Burnside 引理

设置换群 G 作用于有限集合 χ 上，则 χ 在 G 作用下的等价类的数目为：

$$N=\frac{1}{|G|}\sum_{g\in G}\chi(g)$$

其中 $\chi(g)$ 为 g 在 χ 上的不动点个数，即满足 $g(a)=a$ 的个数。

需要注意的是，在这个定理中“有限集合 χ”指的是着色方案集，而不是元素集 S。回到刚才的例子，在每个置换下，不变的着色方案如表 2-5 所示。

表 2-5　不变的着色方案

置　换	含　义	该置换下不变的着色方案	C(f) (该置换下不变的方案数)
f_1	转 0°	1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16	16
f_2	转 90°	1,2	2
f_3	转 180°	1,2,11,12	4
f_4	转 270°	1,2	2

故方案数为所有 $C(f)$ 的平均数，即（16+2+4+2）/4=6，和枚举的结果一致。

下面来考虑程序实现。如果要用它来计数，问题的关键是要求出所有的 $C(f)$。如果求给定置换的 $C(f)$ 的方法是“依次判断每个元素（着色方案）是否在该置换下不变”，由于每个着色方案包含了所有 p 个“格子”的颜色信息，考察每个着色方案的时间复杂度为 $O(p)$；考察所有 n 种着色方案（即计算一个 $C(f)$）的时间复杂度为 $O(np)$。由于置换有 s 个，因此有以下结论。

结论：直接用 Burnside 引理求解着色方案计数问题的时间复杂度为为 $O(nsp)$

其中 n 为着色方案个数，s 表示置换个数，p 表示格子数。请注意，这里 n 的规模是很大的，因此要在程序中直接使用，Burnside 并不理想。然而，我们不想把这个定理扔掉，而是寻求一个好的方法计算 $C(f)$，然后再用 Burnside 引理求解。

步骤 3　Pólya 定理

为了寻找一个好的方法计算 $C(f)$。为此，请复习一下前面介绍的“循环”概念。通俗地说，循环$(a_1a_2\cdots a_k)$是一种特殊的置换，它代表“元素 a_1 变成 a_2，a_2 变成 a_3，…，a_{k-1} 变 a_k，a_k 变成 a_1”。在后面的讨论中，把 G 中的置换写成循环的形式，这样能发现置换的一些内在属性。

前面说过，每个置换都可以写若干互不相交的循环的乘积，而且如果忽略各个循环在表达式中位置，这样的表示是惟一的。置换 f 的循环节数 $m(f)$是上述表示中循环的个数。例如：$\begin{pmatrix}1&2&3&4&5&6\\3&5&6&4&2&1\end{pmatrix}=(136)(25)(4)$，它的循环节为 3。

有了循环节的概念，可以立刻理解并喜欢上如下定理。

定理：如果用 k 种颜色给有限集 S 着色，那么对于一个置换 f，在该置换下不变的置换方案数 $C(f)=k^{m(f)}$。

有了这个定理，把 $C(f)$的计算转化为了置换 f 的循环节 $m(f)$的计算。在刚才的例子中，各个循环的循环节如表 2-6 所示。

表 2-6　循环节

置　换	含　义	循环分解式	$M(f)$（f 的循环节）
f_1	转 0°	(1)(2)(3)(4)	4
f_2	转 90°	(1234)	1
f_3	转 180°	(13)(24)	2
f_4	转 270°	(1432)	1

因此不同方案数为$(2^4+2^1+2^2+2^1)/4=6$。

直接把这个定理代入 Burnside 引理，可以得到 Pólya 定理。

定理 2.3.2　Pólya 定理

设 G 是 p 个对象的一个置换群，用 k 种颜色涂染这 p 个对象，若一种染色方案在群 G 的作用下变为另一种方案，则这两个方案当作是同一种方案，这样不同染色方案数为

$$l=\frac{1}{|G|}\sum_{f\in G}k^{c(f)}$$

其中 $c(f)$为置换 f 的循环节数。

注意到 $m(f)$的计算只和 f 有关，和其他因素都无关！设格子数为 p，很容易设计出复杂度为 $O(p)$的算法来把置换分解成循环的乘积。这样，得到以下结论。

结论　用 Pólya 定理可以在 $O(ps)$的解决着色方案计数问题。

有兴趣的读者可以阅读相关的组合数学书籍而得到更多的知识。例如 Burside 引理的加

权形式，置换群的轮换指标 $Z(G; t_1, t_2, \cdots, t_n)$，母函数形式的 Pólya 定理。

【例题 3】多边形[①]

给出 $n,m(n \geq m)$，从单位正 n 边形的顶点中选出 m 个，能组成多少种不同的 m 多边形（若一个 m 边形可由另一个 m 边形通过旋转、翻转、平移得到，则认为两个 m 边形同种）？

【分析】

首先，由于 m 个点都在同一个圆周上，所以 m 个点构成的多边形一定为凸多边形。

其次，当 $m \geq 3$ 时，平移后不可能发生重合（证明略，请读者自己完成），因此只要考虑旋转和翻转，问题可以转化为用 0、1 两种颜色给 n 个均匀分布在圆周上的点进行染色，使 m 个染成 1，$n-m$ 染成 0，绕着圆周的中心旋转或者按圆周的某条直径翻转重叠的方案视作同一种着色方案，求本质不同的方案数。

把每一种旋转或者翻转看作一个置换，如果不要求染色的个数，则只要计算每个置换循环的节数，而加上个数的限制，只要进行适当的修改便可以了。对于每个置换，把它看成若干个不相交的循环，它的 V 值设为：将置换中 m 个元素染成黑色，$n-m$ 个元素染成白色，并且使每个循环中的所有元素同色的方案数。那么最终答案就为所有置换的 V 值的平均数。下面使用 Pólya 定理计算 V 值，请读者亲自验证以下结果。

（1）旋转

顺时针旋转 i 格的置换中，循环的个数为 $\gcd(n,i)$，每个循环的长度为 $n/\gcd(n,i)$。

证明并不困难，请读者自己完成。

（2）翻转

这时分 n、m 的奇偶性考虑。

当 n 为奇数时，只有一种情况，即：$[n/2]$个循环长度为 2，还有一个长度为 1，所以 $V = C_{[n/2]}^{[m/2]}$；

当 n 为偶数时，有两种情况：

- 有 $n/2$ 个置换满足：有 $n/2$ 个循环长度为 2，此时
  若 m 为偶数，则 $V = C_{n/2}^{m/2}$；若 m 为奇数，则 $V=0$。
- 有 $n/2$ 个置换满足：$n/2-1$ 个循环长度为 2，两个循环的长度为 1，此时
  若 m 为偶数，则 $V = C_{n/2-1}^{m/2} + C_{n/2-1}^{m/2-1} = C_{n/2}^{m/2}$；若 m 为奇数 $V = 2C_{n/2-1}^{(m-1)/2}$。

翻转的总结如下：

n 为偶 m 为奇	n 为奇或 n 为偶 m 为偶
$\sum_{i\text{为翻转}} V = nC_{[n/2]-1}^{[m/2]}$	$\sum_{i\text{为翻转}} V = nC_{[n/2]}^{[m/2]}$

（3）优化

计算翻转时的公式很简单，但旋转还很复杂。如果枚举旋转的格数，复杂度显然较高。有没有好方法呢？可以不枚举 i，反过来枚举 L。

---

① 题目来源：IOI2003 中国国家集训队讨论题目　经典问题

由于 $L|M$，$L|N$，所以 $L|(M, N)$，枚举了 L，再计算有多少个 i 使得 $0\leqslant i\leqslant N-1$ 并且 $L=n/\gcd(n, i)$。即 $\gcd(n,i)=n/L$。

不妨设 $a=n/L=\gcd(n,i)$，

不妨设 $i=a\times t$ 则当且仅当 $\gcd(L, t)=1$ 时

$\gcd(n,i)=\gcd(a\times L,a\times t)=a$

因为 $0\leqslant i<n$　所以 $0\leqslant t<n/a=L$

所以满足这个条件的 t 的个数为 $\phi(L)$｛$\phi(x)$表示不超过 x 的与 x 互质的数的正整数个数｝

所以旋转的所有置换的 V 值之和为 $\sum_{L|\gcd(M,N)} C_{N/L}^{M/L}\phi(L)$

复杂度分析：而如果不用这个式子，需要计算的次数显然为 $\sum_{L|\gcd(M,N)}\phi(L)=\gcd(M,N)$。因此，将计算组合数的次数从 $\gcd(M,N)$ 降为了 $\gcd(M, N)$ 的约数个数，这样一来计算的次数将大大降低。当 $\gcd(M, N)$ 不超过 10 000 时，最多只有 64 次，当 $\gcd(M, N)$ 不超过 32767 时，也不过只有 96 次（由于高精度的运算占了主要的计算时间，所以做的时候不必要用上面的式子，而可以用一个 Tot 数组，Tot_i 记录下循环节个数为 i 的方案数，这样可以降低编程的难度）。

练　习　题

编程题：

2.3.3　三维数码难题[①]

考虑三维数码难题，即给出 $n\times n\times n$ 的立方体，恰好有一个方块为空，其他每个方块上写着 $1\sim n^3-1$ 的不同数字。每次可以把一个与空位相邻的方块移动到空位中，给出初始状态和终止状态，问是否可以实现。

2.3.4　同构计数[②]

一个竞赛图是这样的有向图：

- 任两个不同的点 u、v 之间有且只有一条边（如 $u\rightarrow v$ 或 $v\rightarrow u$）；
- 不存在环（即对点 u，不存在边 $u\rightarrow u$）。

用 P 表示对竞赛图顶点的一个置换（有限群的一个置换，是从 X 到 X 的一个映射函数）。当任两个不同顶点 u、v 间直接相连的边的方向与顶点 $P(u)$、$P(v)$间的一样时（即若竞赛图中有边 $P(u)\rightarrow P(v)$，才有 $u\rightarrow v$，置换 P 可称为一种同构。对给定的置换 P，我们想知道对多少种竞赛图置换 P 可称同构。

例子：有顶点 1、2、3、4 和置换 P：$P(1)=2$，$P(2)=4$，$P(3)=3$，$P(4)=1$，对于图 2-24

① 题目来源：UVA Problem Archive.
② 题目来源：Polish Olympiad in Informatics

所示的四种竞赛图，置换 P 可称同构。

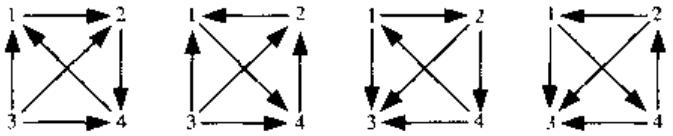


图 2-24　四种同构的竞赛图

编写一个程序，读入一个 n 元置换的描述，算出 t，即置换 P 能称同构的不同的 n 元竞赛图的数目。

2.3.5　手镯[①]

镶嵌着灿烂宝石的手镯是珠宝厂生产的特殊产品。随着珠宝生产的商业化，相同的手镯越来越便宜，但个性化的手镯越来越值钱。因此，厂商尽量根据客户需求生产镶嵌不同宝石的手镯。

一天，珠宝厂的经理问自己："用相同的方式能生产多少不同类的手镯呢？"也就是说，如果手镯有 n 个位置可用来嵌入宝石，有 m 种宝石可用来嵌入，能生产出多少不同类的手镯？

正如你所知，手镯是一种环形珠宝，嵌入宝石的位置在环上等距的地方，手镯可被翻转或旋转。假设有两个手镯，如果将其中一个手镯做某种翻转和旋转使得两个手镯完全一样，我们认为这两个手镯是同类的。

写一个程序，输入可嵌宝石的位置数 n（$2<n\leqslant 2^{7-m}+19$）和可用来嵌入的宝石种数 m（$1<m\leqslant 7$），输出可生产出的手镯种类数 s（$s<10^{17}$）。

例如 n=4，m=3 时，s=21；而 n=20，m=7 时，s=1 994 807 299 453 766。

2.3.4　递推关系与生成函数

下面介绍递推关系和生成函数，它们在计数问题中占有重要的地位。

1．递推关系和常见组合数

在一些计数问题中，很难直接得到计数序列$\{a_i\}$的通项公式，但是可以建立相邻几项的关系，可以利用这些关系一步步计算出需要的值，或者求解递推关系得到通项公式。常见的递推关系在任何一本组合数学书里都有详尽的介绍，这里只补充一些常见的组合数，这些结论并不需要立刻记住，不过它们是有用的。

❑　Fibonacci 数、Lucas 数和 Catalan 数

Fibonacci 数　集合{1，2，3，…，n}的不包含相邻整数的子集的个数，包括空集，总数记为 F_{n+1}。

Lucas 数　集合{1，2，3，…，n}的不包含相邻整数，也不同时包含 1 和 n 的子集的个数，包括空集，总数记为 F^*_n。

① 题目来源：经典问题

Catalan数　定义 C_n 为 Catalan 数，它等于**将正 n 边形用对角线剖分成三角形**的方法数。它们满足的递推关系和生成函数如表 2-7 所示。

表 2-7　Fibonacci 数、Lucas 数和 Catalan 数的递推关系和生成函数表

项　目	通 项 公 式	递 推 关 系	生 成 函 数
F_n	$F_n=\dfrac{\alpha^{n+1}-\beta^{n+1}}{\sqrt{5}}=\left\lfloor \alpha^{n+1}/\sqrt{5} \right\rceil$	$F_{n+1}=F_n+F_{n-1}$	$\dfrac{1}{1-x-x^2}=\sum\limits_{k=0}^{\infty}F_k x^k$
F_n*	$F_n^*=\alpha^n+\beta^n$	F_{n+1}*$=F_n$*$+F_{n-1}$*	$\dfrac{t+2t^2}{1-t-t^2}=\sum\limits_{n\geq 1}F_n^* t^n$
C_n	$C_n=\dfrac{1}{n-1}C_{2n-4}^{n-2}$	$C_{n+1}=\sum\limits_{k=2}^{n}C_k C_{n+2-k}$ $(n-3)C_n=\dfrac{n}{2}\sum\limits_{k=3}^{n-1}C_k C_{n-k+2}$	$\dfrac{1-\sqrt{1-4x}}{2x}=\sum\limits_{k=0}^{\infty}C_{k+2}x^k$

表中，$\alpha=\dfrac{1+\sqrt{5}}{2}$，$\beta=\dfrac{1-\sqrt{5}}{2}$，$\lfloor x \rceil$ 表示与 $x(x<>m+1/2, m\in \mathbf{Z})$ 最接近的整数。

它们的前几项如表 2-8 所示。

表 2-8　Fibonacci 数、Lucas 数和 Catalan 数的前几项

N	0	1	2	3	4	5	6	7	8
F_n	1	1	2	3	5	8	13	21	34
F_n*		1	3	4	7	11	18	29	47
C_n			1	1	2	5	14	42	132

它们有一些有趣的性质，还和一些博弈游戏有着密切的联系。

❑　第一类 Stirling 数和第二类 Stirling 数

第一类 Stirling 数　记 $[x]_n=x(x-1)(x-2)(x-3)\cdots(x-k+1)$。如果把它展开，记

$$[x]_n=\sum_{k=0}^{n}s(n,k)x^k$$

其中系数 $s(n,k)$ 称为第一类 Stirling 数。它的表达式为：

$$s(n,k)=(-1)^k\sum_{1\leq i_1<i_2<\cdots<i_k\leq n-1}i_1 i_2\cdots i_k$$

第二类 Stirling 数　用 $[x]_n=x(x-1)(x-2)(x-3)\cdots(x-k+1)$ 来“展开” x^n，得到

$$x^k=\sum_{k=0}^{n}S(n,k)[x]_k$$

其中系数 $S(n,k)$ 称为第二类 Stirling 数，它的表达式为：

$$S(n,m)=\frac{1}{m!}\sum_{k=0}^{m-1}(-1)^k C_m^k(m-k)^n$$

它们的递推公式为

$S(n+1,k)=S(n,k-1)-nS(n,k)$　　$S(n,0)=0, S(n,n)=1, S(n,k)=0(n<k)$

$S(n+1,k)=S(n,k-1)-kS(n,k)$　　$S(n,1)=S(n,n)=1$

它们的组合意义是

在对称群 S_n 中恰含有 k 个轮换的置换的个数等于$|s(n,k)|$。

n 个元素划分成 k 类的方法数是 $S(n,k)$。

与 Stirling 数密切相关的两个数是：

Bell 数　N 元集合的所有划分数称为 Bell 数，记做 B_n。根据 $S(n,k)$的意义很容易得到：

$$B_n=\sum_{k=1}^{n}S(n,k)$$

Lah 数　设 $[-x]_n=\sum_{k\geq 0}L_{n,k}[x]_k$，则称系数 $L_{n,k}$ 为 Lah 数。它与 Stirling 数的关系是：

$$L_{n,k}=\sum_{j\geq 0}(-1)^j s(n,j)S(j,k)$$

它们有表 2-9 所示性质。

表 2-9　Bell 数和 Lah 数的性质

项　目	通　项	递 推 公 式	生 成 函 数
B_n	$B_{n+1}=\frac{1}{e}(1^n+\frac{2^n}{1!}+\frac{3^n}{2!}+\cdots)$	$B_{n+1}=\sum_{k=0}^{n}C_n^k B_k$，　$B_0=1$	$e^{e^t-1}=\sum_{n=0}^{\infty}\frac{B_n}{n!}t^n$
$L_{n,k}$	$L_{n,k}=(-1)^n\frac{n!}{k!}C_{n-1}^{k-1}$，　$n,k\geq 0$	$L_{n+1,k}=-(n+k)L_{n,k}-L_{n,k-1}$ $L_{0,0}=1, L_{n,k}=0\ (n<k)$	$\frac{1}{k!}\left(\frac{-x}{1+x}\right)^k=\sum_{n=0}L_{n,k}\frac{x^n}{n!}$

2. 生成函数

定义一个序列$\{a_i\}$的**生成函数**（或称“母函数”）为 $a_0+a_1x+a_2x^2+a_3x^3+\cdots$。为什么要定义这样一个函数呢？因为生成函数把$\{a_i\}$的所有信息浓缩到了一个多项式里，可以把它作为整体进行运算，而不需要单独考虑每一个 a_i。

为了尽快地体现出它的优势，回忆前面介绍过的有关内容。

问题：一般排列组合问题

有 k 种元素，重数均为无限大。规定第 i 种元素选取的个数 c_i 必须属于一个**给定的集合** S_i，求 r 排列数和 r 组合数，并列举出方案。

对于这个问题，不加证明地直接给出以下结论：

定理：在一般组合问题中，对于每个元素 k，构造序列$\{a(k)_i\}$，其中 $a(k)_i$ 等于 1 当且仅到 $I\in S_k$。设该问题的 r 组合数 b_r，则序列$\{b_i\}$的生成函数等于所有$\{a(k)_i\}$的生成函数的积。

这个结论听起来挺别扭，不过举个例子就很明了了。

假设有苹果、香蕉和桃子三种水果。如果苹果只能选不超过 3 个，选香蕉的个数必须是 5 的倍数，而桃子只能选 2，3 或者 5 个，问要选 k 个水果有几种方法？

对于苹果，$S_1=\{0,1,2,3\}$，　它的生成函数是 $1+x+x^2+x^3$；

对于香蕉，$S_2=\{5,10,15,\cdots\}$，　它的生成函数是 $x^5+x^{10}+x^{15}+\cdots$；

对于桃子，S_3={2,3,5}，　　它的生成函数是 $x^2+x^3+x^5$。

因此，r 组合数序列的生成函数是$(1+x+x^2+x^3)(x^5+x^{10}+x^{15}+\cdots)(x^2+x^3+x^5)$，把多项式展开后，$x^k$ 项的系数就是所求。

为了解决排列问题，需要定义另外一种**指数生成函数**。

定义 一个序列$\{a_i\}$的指数生成函数为 $a_0+\frac{a_1}{1!}x+\frac{a_2}{2!}x^2+\frac{a_3}{3!}x^3+\cdots$，则有以下定理。

定理　在一般排列问题中，对于每个元素 k，构造序列$\{a(k)_i\}$，其中 $a(k)_i$ 等于 1 当且仅到 $I\in S_k$。设该问题的 r 排列数为 b_r，则序列$\{b_i\}$的指数生成函数等于所有$\{a(k)_i\}$的指数生成函数的积。

【例题 1】装饰栅栏[1]

Richard 先生刚刚修建完新房子，现在惟一缺少的是一排颇具装饰性的木头栅栏。栅栏由 N（$N\leqslant 20$）块高度分别为 1，2，3，…，n 的木头组成，且高低起伏，即，木头 i（$1<i<n$）必须比木头 i-1 和木头 i+1 都高或者都矮。

一排装饰栅栏可以描述为 1，2，3，…，n 的某个排列 A_1，A_2，…，A_n。对于栅栏 A_1，A_2，…，A_n 和栅栏 B_1，B_2，…，B_n，如果存在 i，满足 $A_k=B_k$（$1\leqslant k<i$）并且 $A_i<B_i$，那么我们称栅栏 A 字典顺序先于栅栏 B。Richard 先生给所有的装饰栅栏字典排序并标上从 1 开始的连续编号，例如图 2-25 就是 $n=4$ 时的情况。

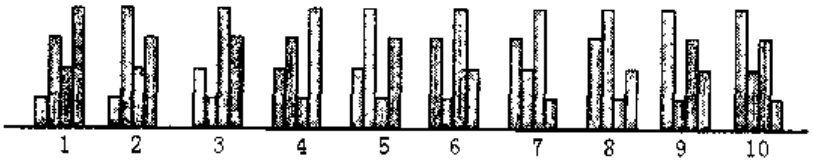


图 2-25　n=4 时的栅栏情况

根据所给编号 num，Richard 想要你告诉他具体描述栅栏的排列 A_1，A_2，…，A_n，例如 $G(n=4,\text{num}=6)=(3,1,4,2)$，$G(n=4,\text{num}=10)=(4,2,3,1)$。

【分析】

解决这类题的常见思路是：为求以 A_1，A_2，…，A_k 为前 k 项的所有栅栏中字典排序第 num 小的序列，从 1 至 n 逐一尝试 A_{k+1}，直到满足 $\text{num}\geqslant\sum_{i=1}^{A_{k+1}}$（以 A_1，A_2，…，$A_{k,l}$ 为前 k+1 项的所有栅栏的数目）。所以，问题的关键在于如何求解以 $A_{1\sim k+1}$ 为前 k+1 项的栅栏数目。

假设从 1…N 中除去数 $A_{1\sim k}$ 后剩余 $B_{1\sim n-k}$，那么求以 $A_{1\sim k+1}$ 为前 k+1 项的栅栏数目就等于求以 $B_{1\sim n-k}$ 为后 n-k 项、头两块木块高度上升（如果 $A_{k-1}>A_k$）或者下降（如果 $A_{k-1}<A_k$）的栅栏数目。又因为所谓的“高低起伏”只是木头与木头高度之间的相对大小关系，而与木头的绝对高度无关，所以倘若 A_{k+1} 是 $B_{1\sim n-k}$ 中第 T 小的数，那么求以 $B_{1\sim n-k}$ 为后 n-k 项，头两块木块高度上升（或者下降）的栅栏数量就等同于求以 1→n-k 为后 n-k 项、头一木块高度为 T，头两木块高度上升（或者下降）的栅栏数量 $g[n-k][T]$.up（或者 $g[n-k][T]$.down）。

[1] 题目来源：CEOI 2002

很明显

$$g[n][T].\text{up}=\sum_{i=T}^{n-1} g[n-1][i].\text{down}\ ,\quad g[n][T].\text{down}=\sum_{i=1}^{T-1} g[n-1][i].\text{up}$$

知道了这个公式，此题便迎刃而解。

不过，需要注意：A_1 是最小的使得 $\text{num}\geqslant\sum_{i=1}^{A_1-1}(g[n][T].\text{down}+g[n][i].\text{up})+g[n][A_1].\text{down}$ 或者 $\text{num}\geqslant\sum_{i=1}^{A_1} g[n][i].\text{down}+g[n][i].\text{down}$ 的数，而不是最小的使 $\text{num}\geqslant\sum_{i=1}^{A_1} g[n][i].\text{down}$ 或者 $\text{num}\geqslant\sum_{i=1}^{A_1} g[n][i].\text{down}$ 的数，另外要注意单独处理 $n=1$ 的情况。

下面的例子不能直接按照通常的方法递推，因为它是关于实数的函数。

【例题 2】Pibonacci 数[1]

应该听说过 fibonacci 数和常数 pi(pi = 3.1415926535…)吧。如果把二者结合起来，就可以创造一个奇怪的东西：Pibonacci 数。可以对任何实数 $x\geqslant 0$ 定义一个函数 $P(x)$：

$$P(x)=\begin{cases}1 & 0\leqslant x<4\\ P(x-1)+P(x-pi) & 4\leqslant x\end{cases}$$

给定整数 $x(x\leqslant 30\,000)$，计算 $P(x)$。例如，$P(4)=2$，$P(25)=4024$，$P(47)=14\,355\,865$。

【分析】

最简单的方法就是直接递归，但是（细心的读者一定看出来了）本题在递归的过程中有很多重复的式子需要计算，因此考虑将 $P(x)$的表达式写出来再计算。由于每次递归不是减 1，就是减 pi，因此任意时刻 P 函数的参数都是 x 减去整数个 1 和整数个 pi。由于递推边界都是不大于 4 的非负实数，令让 $r(i)$表示满足 $x-i-j\times pi\leqslant 4$ 的 j 的最小值，则有 $r(i)=[(x-i-4)/pi]$，且任意一个递推边界都可以表示成 $P(x-i-r(i)\times pi)$。因此，用待定系数法设：

$$P(x)=a_0P(x-0-r(0)\times pi)+a_1P(x-1-j_1)+\cdots=\sum_{i=0}^{x-3}a_iP(x-i-r(i)\times pi)=\sum_{i=0}^{x-3}a_i$$

只需要求出 a_i，问题就得到了解决。

在递归的时候，$P(x)$转化成了 $P(x-1)$和 $P(x-pi)$，因此 $P(x-i-r(i)\times pi)$是由 $P(x)$经过了 $i+r(i)$ 层递归得到的，且在这 $i+r(I)$层中必须恰好有 i 层减 1，$r(i)$层减 pi，最后才能得到 $x-I-r(i)\times pi$。

- 若 $x-I-r(i)\times pi\geqslant 3$，则最后一层可以是由 $x-1$ 或者 $x-pi$ 得到，即 $a_i=C^i_{i+r(i)}$；
- 如果 $x-i-r(i)\times pi<3$，则最后一层必须是由 $x-pi$ 得到，则 $a_i=C^i_{i+r(i)-1}$。

把满足第一个条件的数 i 称做 A 类数，满足第二个条件的称为 B 类数，那么最简单的方法就是每次判断 i 的类型，然后直接计算组合数。由于每次计算 a_i 需耗时 $O(i^2)$，因此总耗时间 $O(x^3)$，无法在短时间内解出，但是注意到数列$\{a_i\}$的相邻 2 项的差别不大，所以可以用组合数的展开式计算 $d_i=a_i/a_{i-1}$ 的值，则 $a_i=a_{i-1}\times d_i$。

可以分四种情况分别讨论 i 和 $i-1$ 的类型，则无论哪种情况下，设

$a_{i-1}=C^{i-t}_m$，$a_i=C^i_{m+a}$，由于 $r(i)-r(i-1)$非 0 即-1（想一想，为什么？），故$-1\leqslant a\leqslant 2$，则有

[1] 题目来源：Internet Problem Solving Contest, 2001

$$\frac{a_i}{a_{i-1}}=\frac{(m+a)!}{i!(m+a-i)!}\Big/\frac{m!}{(i-1)!(m+1-i)!}=\frac{(m+a)!(m+1-i)!}{im!(m+a-i)}=\begin{cases}\dfrac{(m+1-i)(m-i)}{im}, & a=-1\\ \dfrac{m+1-i}{i}, & a=0\\ \dfrac{m+1}{i}, & a=1\\ \dfrac{(m+2)(m+1)}{i(m+2-i)}, & a=2\end{cases}$$

则不管在哪种情况下，d_i的计算量是常数级的。

这样，a_1=1，初始时 m=$r(0)$，而 a_{i+1}可以在 $O(1)$的时间内由 a_i得到，故按照上式计算的时间复杂度为 $O(x)$。最后需要注意的是 x 很大时结果已经超过了长整数范围之内，需要用高精度加法和高精度与单精度的乘除运算。

【例题 3】巧克力[①]

有袋子里均匀地装着 c 种颜色的巧克力，每种巧克力均有无限多。佳佳每次从袋子里拿一块放在桌子上，如果桌子上已经有一块颜色相同的巧克力，佳佳就把两块巧克力都吃掉。佳佳一共取出 n 块巧克力，请问最后桌上有 m 块的概率有多大？例如 c=5，n=100，m=2 时，概率为 0.625。

【分析】

由于 $m>c$ 或者 $m>n$ 或者 m 和 n 不同奇偶性时概率为 0，以下解法中，均认为 $m\leqslant\min\{c, n\}$，且 m 和 n 同奇偶。用递推法解决。设 $d[i,j]$为取出 i 块巧克力，恰好有 j 块的概率，则

$$d[i,j]=d[i-1,j-1]\times(c-j+1)/c+d[i-1,j+1]\times(j+1)/c$$

算法的复杂度为 $O(nc)$。

这道题目是 2002 年国际大学生程序设计竞赛北京赛区比赛的题目。笔者当时负责评测此题，发现几乎所有选手都是采用此法解题，其中大部分选手超时，而勉强通过测试的程序运行时间也很长。其实本题更好的做法是用生成函数。假设佳佳不在中途吃掉任何巧克力，那么本题实际上是要让 m 种颜色取奇数个，$c-m$ 种颜色取偶数个，求排列有多少种。

由刚才的结论，奇数项指数生成函数为$(e^x-e^{-x})/2$，偶数项生成函数为$(e^x+e^{-x})/2$。因此选 m 个奇数，$c-m$ 个偶数的指数生成函数为$[(e^x-e^{-x})^m(e^x+e^{-x})^{c-m}]/2^c$。当然，由于是哪 m 种颜色取奇数个并不确定，因此方案数还要乘以 $C(c,m)$，再除以总方案数为 c^n 就得到了概率。

由于 $m\leqslant c$，因此可以 $O(c^2)$的时间里把生成函数展开成 e^x 的有理式。展开以后可以依次考虑每一项中 x^n 的系数，然后加起来就可以了。下面来说明样例的求解过程。

根据刚才的分析，本题的答案为多项式 $p(x)=[(e^x-e^{-x})^m(e^x+e^{-x})^{c-m}]/2^c\times c(c, m)/c^n$ 的 x^n 项系数乘以 $n!$。在样例中 c=5, n=100, m=2，因此 $p(x)=[(e^x-e^{-x})^2(e^x+e^{-x})^3]/2^5\times10/5^{100}$。

暂时不管系数，把多项式 $p'(y)=(y-y^{-1})^2(y+y^{-1})^3$ 展开，其中 $y=e^x$。由多项式乘法，$p'(y)=(y^2-2+y^{-2})(y^3+3y+3y^{-1}+y^{-3})=(y^5-2y^3+y)+(3y^3-6y+y^{-1})+(3y-6y^{-1}+3y^{-3})+(y^{-1}-2y^{-3}+y^{-5})=y^5+y^3-2y-4y^{-1}+y^3+y^{-5}$。

如何求出 $p'(y)$中 x^n 项的系数呢？$y^5=e^{5x}$，它的展开式是 e^x 的展开式用 $5x$ 替换 x 得到

[①] 题目来源：ACM/ICPC Regional Contest. Beijing 2002

的，因此 x^{100} 的系数为原 e^x 的 x^{100} 的系数乘以 5^{100}，即 $1/100! \times 5^{100}$。请注意，最后还要乘以 5^{100} 和 100!，因此全部都约去了！y^{-5} 中 x^{100} 的系数是 $1/100! \times (-5)^{100}$，也可以约去！但 y^3 中 x^{100} 的系数为 $1/100! \times 3^{100}$，只能约去 100!，而保留 3^{100}，因此有：

设 $y=e^x$ 的多项式 $p'(y)=(y-y^{-1})^m(y+y^{-1})^{c-m}$ 的 y^k 系数为 a_k，则本题的最后答案为：$\text{sum}\{(a_k + a_{k-1})(k/c)^n\} / 2^c \times c(c, m)$ $(k>0)$。在样例中，由于 $n=100$ 很大，因此 $(3/5)^{100}$ 可以忽略不计，最后结果为 $2 / 2^5 \times c(5,2) = 0.625$。由于 $c \leqslant 100$，所以 2^c 可能会比较大，所以最好在做多项式乘法的时候把 2^c 拿进去，即

结论： 令 $g(y) = (0.5y-0.5y^{-1})^m(0.5y+0.5y^{-1})^{c-m}$，$y^k$ 的系数为 a_k，则本题的答案为

$$\text{sum}\{(a_k + a_{k-1}) \times (k/c)^n\} / 2^c \times c(c, m) \quad (k>0)$$

计算 $(k/c)^n$ 可以用对数，时间复杂度为 $O(1)$，计算整个式子的时间复杂度为 $O(c)$，加上多项式乘法的复杂度 $O(c^2)$（能否不做多项式乘法而直接得到各个所需要的 a_k，进而把时间复杂度降低到 $O(c)$呢？这个问题留给读者思考），总的时间复杂度为 $O(c^2)$。当 n 比较大的时候，$(k/c)^n$ 还可以忽略，甚至不需要做完整的多项式乘法，计算量几乎为 $O(1)$（可以证明，$n \geqslant 5000$ 时误差不超过 10^{-9}）。

WEB 我们可以把 Pólya 定理推广成生成函数的形式，而生成函数的另一个重要作用就是求解递推关系，这些内容在本书主页上可以找到。

练　习　题

编程题

2.3.6　掷色子[①]

色子有很多种。虽然传统的色子都是 6 个面，但是在一些特殊的游戏中会使用一些特殊的色子，它们可能有 2，3，4，…最多 50 个面，如图 2-26 所示。每次投掷它们的时候，每个面朝上的机会都是平等的。投掷具有 $F(F \leqslant 50)$ 个面的色子 $n(n \leqslant 50)$ 次，得到的所有点数和加起来为 $S(S \leqslant 4\,000)$ 的概率有多大？

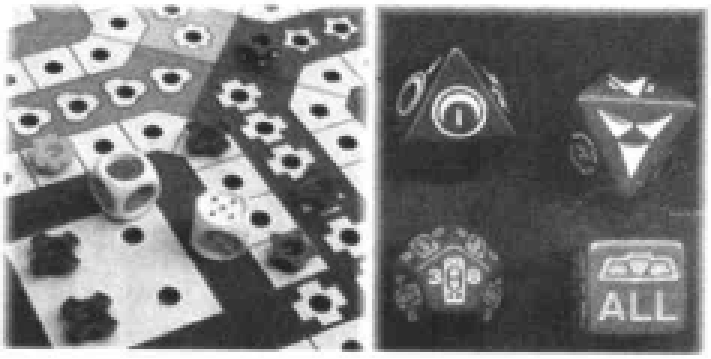


图 2-26　形形色色的色子

[①] 题目来源：UVA Problem Archive

2.3.7　和施瓦辛格共进晚餐!!![1]

你的朋友告诉你附近的电影院近期将上映施瓦辛格的大片《终结者 II》。和往常一样，电影票正在热卖中，而不同的是，这次购票者的生日被记录下来了。这是为什么呢？原来，电影院准备搞一次有奖活动来促销，即如果某位购票者的生日和在他前面购票的某位观众的生日一样，他将有幸被邀请与男主角的扮演者施瓦辛格共进晚餐。这样的机会只给一个人，即第一个满足条件的观众。当然，为了给队首的观众一点机会，如果他的生日和卖票者的一样，他将成为幸运者（不过大家事先并不知道卖票者的生日）。你很希望能和自己的偶像共进晚餐，那么，你应当排在购票队伍中的第几位才最有希望获得这一机会呢？由于一年有 365 天，那么最好排在第 19 位，这样有 18.61%的机会。不过如果一年的天数有 $T(20 \leqslant T < 100\,001)$天，结果又是怎样的呢？

（附注：你可以把施瓦辛格换成你偶像的名字来增加你解题的兴趣）

*2.3.8　串并联网络[2]

考虑一个二端网络。一端叫做“源”，一端叫做“汇”。如果一个二端网络可以由如下方法迭代得到，则说它是一个串并联网络：

（1）单独一条边为一个串并联网络。

（2）若 G1 和 G2 都是串并联网络，分别合并二者的源和汇可得到一个新的串并联网络（并）。

（3）若 G1 和 G2 都是串并联网络，合并 G1 的汇和 G2 的源可得到一个新的串并联网络（串）。

交换串联中的两个元素不会引起网络的本质变化，因此图 2-27 的三个网络本质相同。

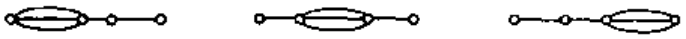


图 2-27　串联本质相同的网络

交换并联中的两个元素也不会引起网络的本质变化，因此图 2-28 三个网络本质相同。

图 2-28　并联本质相同的网络

这样，4 条边的不同网络只有 10 个，如图 2-29 所示。

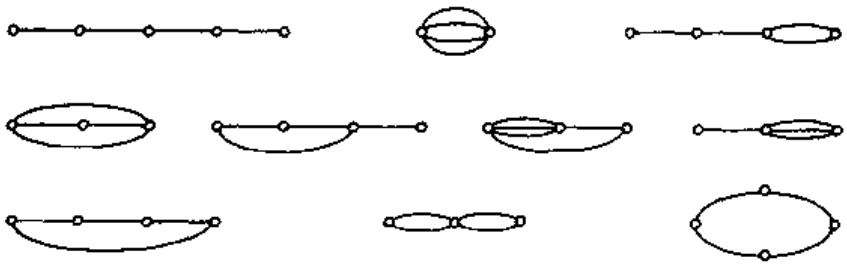


图 2-29　四条边的所有网络

[1] 题目来源：UVA Problem Archive

[2] 题目来源：UVA Problem Archive

求 $n(n\leqslant 30)$条边的不同网络的个数。例如，15 条边的网络有 1 399 068 个。

***2.3.9　奶牛计数**[①]

Farmer John 的农场里居住着 $N(n\leqslant 2000)$头奶牛，为了方便管理，Farmer John 希望用 1 到 N 数字对这 N 头奶牛进行编号。

然而，奶牛们早已经厌倦了从 1 到 N 按顺序简单的编号。奶牛 Bessy 提出这样一种编号规则：按顺序第 i 个奶牛的编号应当小于第 i+2 和第 i+3 个奶牛的编号（如果存在的话）。

当然，这种编号方式可能使不同的奶牛有相同的编号。不过 Farmer John 还是很乐意尝试这种新颖的编号方式。同时，他对一共有多少编号方案产生了兴趣。

不过这个方案数可能太大了，而 Farmer John 对大数字的运算显然力不从心，于是，他就来求助身为高级程序设计员的你。

你能帮助 Farmer John 计算出一共有多少编号方案吗？注意了，Farmer John 一看到大数字就会头昏，所以你只要提供答案的最后 5 位。

****2.3.10　数字不干胶**[②]

Charles 在商店里买了很多很多盒装的不干胶，所有不干胶上都印着 0~9 中的某个数字。每个盒里装的各种不干胶数目都一样：有 i_0 个数字 0，i_1 个数字 1，…，i_9 个数字 9，且每盒中各种数字的不干胶数目都不超过 9。最开始，所有的盒子都是关着的，Charles 每次打开一个新的盒子，然后从已经打开的盒子中取出需要的不干胶拼成一个数，第一次拼成 1，第二次拼成 2……第 N 次拼成 N。为了拼成数 N，Charles 需要为 N 的每一个数字使用一张不干胶。例如，打开第 2070 的盒子以后，为了拼成数 2070，它需要从已经打开的盒子（无论是以前打开的还是这次打开的）中取出一个“2”、两个“0”和一个“7”。取出的不干胶不能在以后再次使用。如果某次打开了一个盒子以后无法拼成相应的数，Charles 就停止工作。给出 i_0，i_1，i_2，…，i_9 的值，编程计算 Charles 一共能拼出多少个数。例如，如果每盒中有各种数字的不干胶恰一张，则 Charles 一共可以拼出 199 990 个数。

2.3.5　离散变换与反演

本节讨论一种常见的计数手段：反演，它是一种间接的方法。看了下面的一个例子你就会明白。

1．引例和基本定理

问题：错排问题

求在 n 个元素的所有排列中，第 $i(1\leqslant i\leqslant n)$个位置不是 i 的排列数目 D_n。

【分析】

利用容斥原理可以得到这个问题的解，但是有的读者可能还见过如下解法。

“恰好有 r 个位置的元素是处于正确位置的排列数目为 $C_n^r D_{n-r}$（先选 r 个元素放到它本来的位置，其余元素不许再排到自己的位置），把每种情况加起来，就有 $\sum_{r=0}^{n} C_n^r D_{n-r} = n!$，

[①] 题目来源：Romanian Olympiad in Informatics

[②] 题目来源：Baltic Olympiad in Informatics, 2000

解出 D_n 便可以得到 $D_n=\sum_{r=0}^{n}(-1)^{n+r}C_n^r r!=n!\sum_{r=0}^{n}(-1)^r\frac{1}{r!}$。”

这个解法似乎“与众不同”它不是用已知的东西去表示未知的东西，而是用未知的东西去表示已知的东西，然后把未知东西解出来。什么样的情况可以把未知东西解出来，怎样解呢？我们有如下定理。

定理　设有两个多项式簇 $p_k(x)$，$q_k(x)$，$k=0$，1，2，…，n。每个多项式 $p_k(x)$和 $q_k(x)$都是 k 次的。

若它们满足：$\begin{cases}p_k(x)=\sum_{i=0}^{k}\alpha_{i,k}q_i(x)\\ q_k(x)=\sum\beta_{i,k}p_i(x)\end{cases}$ $(k=0,\ 1,\ 2,\ \cdots,\ n)$，

而系数满足以下正交性：$\sum\alpha_{i,k}\beta_{j,i}=\delta_{j,k}=\begin{cases}1, & j=k\\ 0, & j\neq k\end{cases}$

则下列两式可以**相互推导**(或者说两式**可逆**)：$\begin{cases}f(k)=\sum_{i=0}^{k}\alpha_{i,k}g(i)\\ g(k)=\sum_{i=0}^{k}\beta_{i,k}f(x)\end{cases}$ $(k=0,\ 1,\ 2,\ \cdots,\ n)$。

其中，$f(k)$和 $g(k)$是两个整标函数。而这个相互推导式就是所谓的**离散变换**（discrete transform）或者**反演**（inversion）。

这个定理并不那么易懂，好在多数读者不需要深究其细节。可以把它忘掉，因为从后面内容中，系数的正交性是已经保证了的；而对于相互推导过程，也将会从书中的例子学会它。只需要知道，这样一个定理是为了保证刚才介绍的“反演法”的可行性。请读者仔细体会刚才的例子，直到可以轻而易举地把这个解题过程独立写出来。这是很重要的，因为后面要介绍几种不同的离散变换，它们并不都是很容易理解的。

2．二项式变换和Stirling变换

二项式变换是一种很常见的变换。略去一般的变换式，只给出最简单的变换如表 2-10 所示。

表 2-10　简单的二项式变换表

序　号	A_n	B_n
1	$a_n=\sum_{k=0}^{n}C_n^k b_k$	$b_n=\sum_{k=0}^{n}(-1)^{k+n}C_n^k a_k$
2	$a_n=\sum_{k=n}^{\infty}C_k^n b_k$	$b_n=\sum_{k=n}^{\infty}(-1)^{k+n}C_k^n a_k$
3	$a_n=\sum_{k=0}^{n}C_{n+p}^{k+p}b_k$	$b_n=\sum_{k=n}^{\infty}(-1)^{k+n}C_{n+p}^{k+p}a_k$
4	$a_n=\sum_{k=n}^{\infty}C_{k+p}^{n+p}b_k$	$b_n=\sum_{k=n}^{\infty}(-1)^{k+n}C_{k+p}^{n+p}a_k$

刚才的解法就是用了变换式（1）。二项式变换可以保证系数正交性，有兴趣的读者可以自己试着证明一下。

前面介绍了第一类 Stirling 数 $s(k,i)$和第二类 Stirling 数 $S(k,i)$，大家可以试着证明一下，变换

$$f(k)=\sum_{i=1}^{k}s(k,i)g(i)$$
$$g(k)=\sum_{i=1}^{k}S(k,i)f(i) \qquad (k=1,\ 2,\ \cdots,\ n)$$

是可逆的，我们称它为 Stirling 变换。

3．Möbius反演

有些时候，等式右边的求和式并不是简单地从 1 加到 k。例如需要对“n 的所有约数”的函数值求和，或者对“S 的所有子集”的函数值求和。在这种情况下，需要用 Möbius 反演来求。为此，先来看看刚才所提到的“整除”关系和“包含”关系有什么特点。

熟悉集合论的读者一定知道，它们都是**偏序关系**，满足**自反、反对称、传递**的二元关系。对于没有学过“关系”的读者，可以这样解释：偏序关系可以理解成一个“关系运算符”，例如“≤”（以后若不加说明，讨论的偏序集都记做$<X, \leqslant>$，但这里的运算符“≤”不一定指的整数集上的“小于等于”）。如果

（1）对于任何一个元素 a 都有 $a\leqslant a$；

（2）如果对于两个不同元素 a，b，如果满足 $a\leqslant b$，就一定不满足 $b\leqslant a$（注意，有可能两个都不满足！）；

（3）如果存在三个元素 a，b，c，使得 $a\leqslant b$ 且 $b\leqslant c$，则一定满足 $a\leqslant c$。

则说这是一个偏序关系运算符。

特别地，如果该偏序集对于任何两个元素 a，b，要么有 $a\leqslant b$，要么有 $b\leqslant a$，则把它称为**全序集**（也叫**线性有序集**）。例如自然数集上的“≤”是全序关系；但复数集上的“≤”就不是全序关系。

偏序集有一种图形表示。和关系的图表示不一样，当且仅当 a≤b 且 a 和 b 中没有其他元素能“夹在中间”（即不存在不同于 a，b 的元素 c 使得 $a\leqslant c\leqslant b$）时，才给 a 和 b 连一条线，并把 a 放在 b 的下方，而不是通过给边加上箭头来表示 a 和 b 究竟哪个“大”。

偏序集是经常可以遇到的一类集合。通俗地来说，它们的“有些元素能比较大小，有的不能”。由于有的不能比较，无法给它们排序从而获得一些好处；但是又希望利用一些零碎的大小信息来帮助解决问题，为此，考虑两个方面的问题。

- **扩展（变大）**　可以把偏序集扩展得到全序集，方法就是前面说过的“拓扑排序”。
- **分拆（变小）**　希望找到具有某类比较好的序关系的子集。为此，介绍两个概念。
- **链**：偏序集的一个子集 C，它的任何两个元素都可以比较大小（因此可以线性排序）。
- **反链**：偏序集的一个子集 A，它的任何两个元素都不能比较大小。

二者有如下关系：若 C 是一个链而 A 是一个反链，则 A 和 C 最多只有一个公共元素。

有了这两个概念，很自然地提出两个问题：

问题 1：怎样把偏序集划分成最少数目的链？

问题 2：怎样把偏序集划分成最少数目的反链？

在回答这两个问题之前，首先注意到一个事实：假如存在一个大小为 r 的链 C，则 C 里不存在两个元素属于同一个反链。这样，至少需要 r 条反链才能覆盖链 C，进而覆盖整个偏序集。如果设所有链中最长的一条的大小为 r^*，那么问题 1 的答案至少为 r^*。幸运的是，r^*条已经足够，因为有两个相互对偶的偏序集分解定理（其中定理 2 通常叫做 Dilworth 定理）。

定理 1　$<X,\leqslant>$是偏序集，r 是链的最大的大小，则 X 可被划分成 r 个但不能再少的反链。

定理 2　$<X,\leqslant>$是偏序集，m 是反链的最大的大小，则 X 可被划分成 m 个但不能再少的链。

定理的证明略去，有兴趣的读者可以参考专门书籍。

有了这些知识，来看看 Möbius 反演：

设$<X,\leqslant>$是一个局部有限偏序集（即对于其中任何元素 x，y，满足 $x\leqslant a\leqslant y$ 的元素 a 只有有限多个）或局部有限格（格的定义这里略去，本书中不会用到）。在 X 上定义函数

$$\mu(x,y)=\begin{cases}1, & x=y\\ -\sum\limits_{x\leqslant u<y}\mu(x,u), & x<y\\ 0, & \text{其他}\end{cases}$$

则称 $\mu(x,y)$ 为$<X,\leqslant>$上的 **Möbius 函数**，在这个函数的基础上，有以下不定理。

Möbius 反演定理　设 $f(x)$，$g(x)$是$<X,\leqslant>$上的两个整数函数，则以下两式可逆，即

$$f(x)=\sum_{0\leqslant u\leqslant x}g(u),$$

$$g(x)=\sum_{0\leqslant u\leqslant x}\mu(u,x)f(u),$$

其中 0 元为$<X,\leqslant>$中的最小元。

该定理告诉我们，满足定义的 Möbius 函数一定可以满足系数正交性，不过我们对它的证明并不感兴趣。跳过这个证明，把注意力转到三种特殊偏序集上的反演函数和它们的应用上。

整数因子格

首先，考虑这样一个格（不用知道格是什么意思，只需要知道整数因子格是什么），它的元素是**正整数**，运算符是**整除符号**“|”。

整数因子格$<Z^+,|>$上的 Möbius 反演：

固定 x=1，定义 Möbius 函数如下（请读者仔细验证它是否符合定义）：

$$\mu(n)=\begin{cases}1, & n=1\\ (-1)^k, & n=p_1,p_2,\cdots,p_k,\text{此处}p_1,p_2,\cdots,p_k\text{是互不相同的素数}\\ 0, & \text{其他}\end{cases}$$

则以下两式互逆：

$$f(n)=\sum_{d|n} g(d)$$

$$g(n)=\sum_{d|n} \mu(d) f(\frac{n}{d})=\sum_{d|n} \mu(\frac{n}{d}) f(d)$$

由于讨论的是整数，所以需要一点数论知识来进行深入讨论。不加证明地给出下面的结论。

（1）这里的 Möbius 函数是积性函数。

（2）f 是积性的当且仅当它的 Möbius 变换是积性的。

有限集的幂集格

接下来，再来考虑这样一个格，它的元素是**有限集**，运算符是集合的**包含关系**。

有限集的幂集格 $<P(S),\subseteq>$ 上的 Möbius 反演：

定义 Möbius 函数

$$\mu(B,A)=\begin{cases}(-1)^{|A|-|B|}, & B\subseteq A\\ 0, & \text{其他}\end{cases}$$

则以下两式互逆：

$$f(A)=\sum_{H\subseteq A} g(H)$$

$$g(B)=\sum_{H\subseteq B}(-1)^{|B|-|H|} f(H)$$

有限集的划分格

最后，来考虑一个比较复杂的**格**，其元素是给定集合 X 的**划分**（**所有划分的集合记做** P），偏序关系为“**加细**”（即，$P_1\leqslant P_2$ 当且仅当 P_1 是 P_2 的加细）。如果对集合的“划分”和“**加细**”概念不太熟悉，这里再简单的描述一下这两个概念。

通俗地说，集合 S 的划分就是说把 S 中的元素分成若干组，使得每个元素在且仅在一组中。正式地说，S 的划分 P_s 是若干个两两不相交**的**集合的集合（这些不相交的集合中，每一个都是一“组”，所有组的集合就是划分 P_s）。例如集合{1,2,3,4,5,6}，把 1，3 **分**成一组 A（即集合{1,3}），2，4，5 **分**成一组 B(即集合{2,4,5})，6 单独一组 C（即集合{6}），那么划分 P_s 就是{A,B,C}={{1,3},{2,4,5},{6}}。

划分 P 的加细也是一个划**分**。它是把 P 中的每个“组”进一步分割成若干个不相交的集合。例如 P={{1,3},{2,4,5},{6}}，如果把集合{2,4,5}分成集合{2},{4},{5}，那么加细后的划分 P' ={{1,3},{2},{4},{5}}。简单地说，如果有两个划分 P_1，P_2，在 P_2 里不同组的元素在 P_1 里也不同组，那么 P_1 就是 P_2 的**加细**。

举一个有限集的划分格的例子。

X={a,b,c},

P_1={X}={{a,b,c}}, P_2={{a,b},{c}}, P_3={{a},{b,c}}, P_4={{a,c},{b}}, P_5={{a},{b},{c}}, P={P_1,P_2,P_3,P_4,P_5}。

那么，$P_5\leqslant P_2\leqslant P_1$，$P_5\leqslant P_3\leqslant P_1$，$P_5\leqslant P_4\leqslant P_1$。

弄清了划分和加细的概念以及划分格的含义，可以得到以下定义。

有限集的划分格<P,≤ >上的 Möbius 反演：

定义 Möbius 函数为

$$\mu(P_1,P_2)=\begin{cases}(-1)^{m+n_1+n_2+\cdots+n_m}(n_1-1)!\cdots(n_m-1)!, & \text{当}P_1\leqslant P_2\text{且满足条件}D\\ 0, & \text{否则}\end{cases}$$

（其中条件 D 为：设 $P_2=\{A_1,A_2,\ldots A_m\}$，每一个 A_m 对应 P_1 中 n_i 个子集的并）

则以下两式可逆：

$$f(P_k)=\sum_{Q\leqslant P_k}g(Q)$$

$$g(P_k)=\sum_{Q\leqslant P_k}\mu(Q,P_k)f(Q)$$

这个 Möbius 函数不大好理解，通过几个例子来说明，这个函数到底应该怎样求。

令 $P_1=\{A_1,A_2,A_3,A_4\}=\{\{a,b,c,d\},\{e,f\},\{g,h,i\},\{j\}\}$，$m=4$

它的加细 $P_2=\{\{a,c\},\{b\},\{d\},\{e,f\},\{g\},\{h,i\},\{j\}\}$，则

$A_1=\{a,b,c,d\}$“分裂”成了 P_2 中的三个集合$\{a,c\},\{b\},\{d\}$，因此 $n_1=3$；

$A_2=\{e,f\}$在 P_2 中不变，因此 $n_2=1$；

$A_3=\{g,h,i\}$“分裂”成了 P_3 中的两个集合$\{g\},\{h,i\}$，因此 $n_3=2$；

$A_4=\{j\}$不变，因此 $n_4=1$。

代入公式得 $\mu(P_2,P_1)=(-1)^{4+3+1+2+1}(3-1)!(1-1)!(2-1)!(1-1)!=-2$。（注意，0!=1）

本节内容比较难懂，但其思想比较简单，即列出方程，反解出未知量。从前面的定理可以看出，列出来的方程应该是某个容易求出来的“整体量”等于所有部分量之和。其中整数因子格为所有约数对应的量；有限集的幂集格为所有子集对应的量；有限集的划分格为所有加细对应的量。这些方程往往都是“显然的”，但是又不容易想到去用它们。它反映的是一种逆向思维，而这种思维在组合数学中是非常有用的。

练　习　题

编程题：

2.3.11　带标号连通图计数[①]

n 个结点的带标号连通图有多少个？（带标号的意思是指同构的图算多个）

提示：任意图 G 的每一个连通分支对应 V 的一个子集，则它的“连通情况”对应于 V 的一个划分 p，用 p=pattern(G)表示。例如如果 v_1,v_3 组成一个连通分支，v_2,v_4,v_5 组成另外一个连通分支，则对应的划分为$\{\{v_1,v_3\},\{v_4,v_5,v_6\}\}$。对于任意一个划分 p，让 $C(p)$表示满足 pattern(G)=p 的图的个数，$D(p)$表示满足 pattern(G)≤p 的图的个数，则 $D(p)$可以由乘法原

[①] 题目来源：经典问题

理直接算出，由有限集的划分格上的 Möbius 反演即可求得 $C(p_0)$，其中 $p_0=\{V\}$。

2.3.12　数字接力[①]

$k(k>1)$个小朋友一起玩一种数字接力游戏。游戏开始时，主持人公布一个自然数 $n(n>1)$，然后从小孩 1 开始报数。每个小孩可以随便选择一个不小于 1，不大于 n 的自然数进行报数。如果不存在一个大于 1 的整数既是 n 的约数也是所有小孩报的数的约数，则游戏成功结束。问：有多少种报数方案使得游戏可以成功结束？

2.4　图论基本知识和算法

在第 1 章中，我们简单地学习了图，还学会了遍历图。在本节中，将看到与图论有关的很多其他知识。这里不打算系统地讲解图论，而是希望让读者对图论中一些比较重要的理论和方法有一个比较全面的了解，从而促进今后的进一步学习。建议在阅读本节的同时参考一些同类书籍，这样才能在较短时间内感性地学习到一些重要内容的同时对一些细节问题也有所了解。

本节的重点是本节涉及到的算法和例子，理论部分不用太在意。本书有意没有给出系统的理论知识，这是由本书的特点决定的。对理论有兴趣的读者可以参考相关书籍。

本节学习难度比较大，但是内容比较吸引人。建议先通读本节内容，但不需弄清细节，只需要有个整体的印象，重点应放在本节中对算法思想的描述、例子的分析和一些细节的提示。学习本节之后，希望读者能掌握本节介绍的大部分算法并体会它们的思想（不要只能背出算法框架），且能从本节的例子中找到一些建模的灵感和技巧，并认真思考本节中的思考题（部分题目有相当的难度）。

2.4.1　基本概念和定理

连通性问题在图论中占有很重要的地位。这里不打算深入展开讨论，而只是介绍一些结论和相关的算法。

1．图的连通性

点连通性　设无向图 G 是连通的，若有结点集 $V_1\subseteq V$，使得图 G 删除了 V_1 所有结点后所得的子图是不连通的，而删除了 V_1 的任意真子集后，所得的子图仍然是连通图，则称集合 V_1 为图 G 的点割集。若某一结点就构成点割集，则称该结点为割点。包含点数最少的割集所包含的点数称为 G 的点连通度 $k(G)$。

边连通性　设无向图 G 为连通的，若有边集 $E_1\subseteq E$，使得图 G 删除了 E_1 所有边后所得的子图是不连通的，而删除了 E_1 的任意真子集后，所得的子图仍然是连通图，则称集合 E_1 为图 G 的边割集。若某一边构成边割集，则称该边为桥（或割边）。包含边数最少的边

[①] 题目来源：经典问题

割集所包含的边数称为 G 的边连通度 $k'(G)$。

连通性　在无向图中，若从顶点 v_1 到顶点 v_2 有路径，则称顶点 v_1 与 v_2 是连通的。如果图中任意一对顶点都是连通的，则称此图是连通图。非连通图的极大连通子图叫做连通分量；在有向图中，若对于每一对顶点 v_i 和 v_j，都存在一条从 v_i 到 v_j 和从 v_j 到 v_i 的路径，则称此图是强连通图。非强连通图的极大强连通子图叫做强连通分量。对于这几个概念，还有以下两个定理：

- **Menger 定理**：设 G 是简单图，则 G 是 k-连通图的充分必要条件是 G 中任何两点之间有 k 条独立路（即无公共点的路）。
- **度序列的 k 连通定理**：设图 G 是度序列为 $d(v_1)\leqslant d(v_2)\leqslant\cdots\leqslant d(v_n)$，若存在 $k(0\leqslant k\leqslant n)$ 使 $d(v_n)\leqslant j+k-1$，$j=1$，2，3，…，$n-1-d(v_{n-k+1})$，则 G 是 k-连通的。

关于连通性，图论中还有很多内容，例如 2-连通图的性质和 3-连通图的性质和构造。Menger 定理也可以推广为 Mader 定理。我们还可以利用它来研究边不相交的生成树。如果和平面图等内容结合起来，还会有一些令人欣喜的结论。

2. 图的着色

在本节中，我们来考虑一个有趣的问题：图的着色。

点着色问题　给图 G 的每个顶点着一种颜色，任意两个相邻顶点必须着不同的颜色。最少需要几种颜色？

边着色问题　给图 G 的每条边着一种颜色，任意关联于同一个顶点的两条边必须着不同的颜色。最少需要几种颜色？

点着色记数问题　给图 G 的每个顶点着一种颜色，任意两个相邻顶点必须着不同的颜色。如果颜色一共有 k 种，那么不同的着色方案（不一定每种颜色都要用到）有多少种？

对于问题一，为了叙述方便，引入记号 $\chi(G)$ 表示问题一的解，称为图 G 的“色数”。还可以用另外一种形式来叙述问题一：“把图 G 的顶点集划成数目最少的子集 V_1,V_2，…，V_k，使得每个子集的导出子图都是零图”。我们不加证明地给出色数的如下性质：

性质 1. 对于有 n 个顶点的图 G，$1\leqslant\chi(G)\leqslant n$。

左边的等号当且仅当 G 为零图成立，右边的等号当且仅当 G 为完全图时成立。

性质 2. 如果图 G 最大阶数的导出零图包含 q 个结点，则 $\chi(G)\geqslant\left\lceil\dfrac{n}{q}\right\rceil$。

性质 3. 设图 G 至少有一条边，则 $\chi(G)=2$ 当且仅当 G 为二分图时成立。

性质 4. 设图 G 的顶点中最大的度数为 Δ，则 $\chi(G)\leqslant\Delta+1$。

性质 4 是可以改进的。奇数阶循环图中 $\chi(G)=3$，$\Delta=2$；完全图中 $\chi(G)=\Delta+1=n$。

性质 5. 设简单图 G 不是完全图也不是奇数阶循环图，则 $\chi(G)\leqslant\Delta$。

性质 6. 若 $G(n,m)$ 是个简单图，则 $\chi(G)\geqslant\dfrac{n^2}{n^2-2m}$

从上面的性质中可以看出，求色数是很困难的（否则会直接给出一个好的结论，而不用给出这么多并不好的上界和下界）。关于它的一些讨论也是困难的。不过正因为这一点，有关色数的研究是很多的。在本小节下面引入了“团”的概念以后，还将继续讨论色数，

并定义一种类型的图，叫做“完美图”。

对于问题二，为了叙述方便，引入记号 $\chi'(G)$ 表示问题二的解，称为图 G 的“边色数”。如果不讨论多重图（有 Vizing 定理：对于无环的 p-重图，则 $\Delta \leqslant \chi'(G) \leqslant \Delta+p$；$\chi'(G) \leqslant [3\Delta/2]$），那么边色数有如下的结论。

结论：若 G 为简单图，则 $\chi'(G)=\Delta$ 或者 $\chi'(G)=\Delta+1$。

这个结果比色数要好得多，因为它只有两种取值！这样，我们把满足 $\chi'(G)=\Delta$ 的图叫做第一类边色图（如 K_{2n}，二分图），满足 $\chi'(G)=\Delta+1$ 的图叫做第二类边色图（如 K_{2n+1}）。剩下的讨论一般就是判断一些图类是第一类边色图还是第二类，这里不展开讨论。

对于问题三，一个显然的结论是：如果 $k<\chi(G)$，方案数肯定为 0！但是 k 比较大的时候呢？我们有一个振奋人心的结论如下。

结论：用 k 种颜色给图 G 着色，方案数为关于 k 的**多项式**，记为 $P_G(k)$ 或者 $P_k(G)$。

既然是多项式，那么讨论就方便很多了。我们先不加证明地给出该多项式的一些性质：

性质 1. 它的次数为图 G 的顶点数 n。

性质 2. 首项系数为 1，常数项为 0，k^{n-1} 的系数为 $-|E|$。

性质 3. 系数为正负交替的整数，且绝对值随 k 的次数呈单峰型。

性质 4. 不为 0 的系数的最小次数为 G 的连通分支数。

用这些性质可以帮助读者验证求出的色多项式是否有明显错误。

如何求色多项式呢？可以用递推的方法做，即把大图化成小图。例如，对于 G 中的一条边 $(u,v)\in E$，把它删除掉不就变成小图 G_1 了吗？不过为了建立递推关系式，需要考虑这样做会有什么影响。原来的图中，u，v 必须着不同的颜色，但新得到的图 G_1 就没有这个限制了。如果用“减法原理”的话，那么实际的着色方案 $P(G)$ 应该是：$P(G_1)$ 减去在 G_1 中强制规定 u，v 必须着相同的颜色后的着色方案数（注意体会这句话的含义！）。怎样才能强制规定 u，v 必须着相同颜色呢？合并 u，v 结点就可以了！如果把合并 u，v 后的图叫做 G_2（注意：合并结点也就是把大图变成了小图），那么我们得到如下所示的一个有用的关系式。

结论：色多项式递推关系：$P(G)=P(G_1)-P(G_2)$

由于 G_1 的边数减少，G_2 的顶点数减少，所以递推关系一定会到达边界。边界是什么呢？当然是“没有边”的图（零图），因为找不到边 e 可以删除了。显然，对于零图 N_n，色多项式为 k^n。由于递推关系中只有减法，所以我们实际上已经证明了刚才的结论：任意图的色数都是关于 k 的**多项式**。我们还可以用这个方法证明刚才的几个性质。

这里有一些特殊图的色多项式，可以被直接计算出来。如果能把它们作为递归边界，将给计算带来方便：

① 空图的色多项式为 k^n；

② 完全图的色多项式为 $[k]_n=k(k-1)(k-2)\cdots(k-n+1)$；

③ T 为树的**充分必要条件**为 T 的色多项式为 $k(k-1)^{n-1}$；

④ 圈的色多项式为 $(k-1)^n+(-1)^n(k-1)$

⑤ 轮的色多项式为 $k(k-2)^n+(-1)^nk(k-2)$。

另外，连通图 G 的色多项式不大于 $k(k-1)^{n-1}$。

在计算中，我们尽量删除图中的桥，因为删除以后可以得到两个连通块，而我们有如下结论。

结论：如果图 G 是不连通的，那么它的色多项式等于每个连通分量的色多项式的乘积。

可惜，稍微分析一下就可以发现，计算色多项式的复杂度是指数级的。所以在计算色多项式的时候，应该首先分析问题的特点。

3. 独立集，团，完美图

独立集　对于一个图 G，选其中的一些顶点构成一个集合 U。如果集合中任意两个顶点在图中不相邻，我们说 U 是图 G 的一个独立集(或者说，团就是一个顶点集，由它导出的子图是零图)。由于独立集的子集也是独立集，所以我们关心的是这样一些独立集，它们不被任何更大的独立集所包含。这样的独立集叫做极大独立集。极大独立集中包含顶点最多的一个很自然地被称做最大独立集。它包含的顶点数目称为该图的独立数，记为 $\alpha(G)$。对于一般图来说，$\alpha(G)$是很难计算的，好在对于一些特殊的图，我们可以直接得出它们的独立数：零图 N_n 的独立数为 n；完全图 K_n 的独立数为 n；二分完全图 $K_{m,n}$ 的独立数为 $\max\{m,n\}$。这三个结论都是显然成立的。

团　把独立集的概念“反过来”，可以得到：对于一个图 G，选其中的一些顶点构成一个集合 U。如果集合中任意两个顶点在图中相邻，我们说 U 是图 G 的一个团。(或者说，团就是一个顶点集，由它导出的子图是完全图)。和独立集类似，定义包含点数最大的团的阶数为图 G 的团数。记为 $\omega(G)$。

团数和独立数的关系　由于团和独立集是互补的概念，我们有：

$$\alpha(G)=\omega(\overline{G}),\quad \omega(G)=\alpha(\overline{G})\quad \cdots(*)$$

团划分　前面说过，顶点着色实际上是把顶点集划分成独立集。既然独立集和团又是互补的概念，则可建立顶点着色的互补概念：团划分，即把顶点集划分成团。G 中一个团划分的最小团数定义为 G 的团划分数，用 $\theta(G)$表示。这样，有类似(*)的式子：

$$\chi(G)=\theta(\overline{G}),\quad \theta(G)=\chi(\overline{G})\quad \cdots(**)$$

进一步，还有两个互补的式子：

$$\chi(G)\geq\omega(G)\quad (1)$$

$$\theta(G)\geq\alpha(G)\quad (2)$$

第一个式子说的是“色数不小于团数”。它显然成立，因为最大的团里面的所有点必须着不同的颜色；第二个式子说的是“团划分数不小于独立数”中的某一个。它显然也成立，因为最大独立集中的每个元素必须属于不同的团。

等号成立的条件　我们很自然的提出这样的问题：对于什么样的图（1）式的等号成立，对于什么样的图（2）式的等号成立呢？我们渴望让两个等式成立的图有什么惊人的性质。可惜，这个问题的答案并不令人满意。随便取一个图 H，设它的色数为 p，那么它的团数不大于 p。不过只要再选一个 K_p，把它和 H 看成一个大图 G 的两个连通分量，那么由于 K_p 的存在，G 的团数为 p，而色数等于 H 的色数 p。这样，一个由任意图 H 和一个阶数等于 H 的色数的图组合起来的图都让（1）式的等号成立，而并不需要具有什么特别的性质。

因此，我们把条件加强一下，规定 G 的任意导出子图都必须让某个等号成立，期望满足它的图能够特殊一些。也可用以下概念来解释。

完美图　如果图 G 的任意导出子图都满足 $\chi(G)=\omega(G)$，我们说 G 是 χ-完美的；如果图 G 的任意导出子图都满足 $\theta(G)=\alpha(G)$，我们说 G 是 θ-完美的。从两个角度定义出两个完美图，总是有些遗憾的。好在有如下**定理**：图 G 是 χ-完美的当且仅当它是 θ-完美的。等价地，G 是 χ-完美的当且仅当 G 的补图是 χ-完美的。这样，把两种完美图合并起来，统称完美图（perfect graph）。它们让(1)、(2)的等号同时成立。

两类完美图　完美图是存在的。进一步地，可以证明有一大类图都是完美图。这类图就是弦图（chordal graph）。所谓弦图，就是指任何长度大于 3 的圈都有弦（连接圈内不相邻的两个点的边）的图。它的一种特殊情况是区间图（interval graph），它也是非常重要的一类特殊图，稍后我们会继续讨论这两类图。

WEB　图论中一个很重要的概念叫做树宽度（tree-width），和它相关还有很多概念如树分解（tree decomposition）、k-树（k-tree）等。有兴趣的读者可以在本书主页上找到它们的介绍。

2.4.2　可行遍性问题简介

可行遍性问题是一个很实际的问题：按一定顺序走遍图的所有顶点或者所有边。具体来说我们考虑如下 4 个问题：

- **Hamilton 回路问题**　是否可以从某点出发顺着边走，每个点恰好走一次以后回到出发点？
- **欧拉回路问题**　是否可以从某点出发顺着边走，每条边恰好走一次以后回到出发点？
- **旅行商问题（TSP）**　如果每个边上的权不同，那么在问题一的解中求出权和最小的那一条。
- **中国邮递员问题**　如果问题 2 的答案是“否”，怎样才能在每条边至少走过一次后回到出发点，并使回路上权和尽量小？

1. 哈密顿回路

问题 2 和问题 4 已经被很好的解决了，而问题 1 和问题 3 已经被证明是 NP 完全问题（回忆我们在 1.1 节中给出的 NP 完全问题的说明：它们目前都没有发现有多项式算法）。欧拉回路问题我们将在后面专门讲解，这里只介绍哈密顿（Hamilton）回路。判断一个图是不是 Hamilton 是一个 NP 完全问题，所以在这里只能分别给出一些必要条件和充分条件：

必要条件　若 $G=(V, E)$ 是 Hamilton 图，则对 V 中任何非空子集 S 有 $\omega(G-S)\leqslant|S|$。这里 $\omega(H)$ 表示 H 的连通分支数。

充分条件比较多，这里列举几个：

最小次条件（Dirac 条件）　$n(n\geqslant 3)$ 阶简单图 G 若满足 $\delta\geqslant n/2$，则 G 是 Hamilton 图。

次数条件　设 G 是简单图（$n\geq 3$），若对任何 k（$1\leq k\leq (n-1)/2$），次数不大于 k 的点数少于 k；当 n 是奇数的时候还需满足次数不大于$(n-1)/2$ 的点数不超过$(n-1)/2$，则 G 是 Hamilton 图。

次数和条件（Ore 条件）　设 G 是 n 阶简单图，若任意两个不相邻的点 u,v 有 $d(u)+d(v)\geq n$，则 G 是 Hamilton 图。

次序列条件（Chvatal 条件）　设 G 是 n 阶简单图，次序列为 $d_1\leq d_2\leq\cdots\leq d_n$，若对每一个 $k(d_k\leq k<n/2)$有 $d_{n-k}\geq n-k$，则 G 是 Hamilton 图。

边数条件　设 G 是 n 阶简单图（$n\geq 3$），边数 $m>C(n-1,2)+1$，则 G 是 Hamilton 图。且当 $m=C(n-1,2)+1$ 时，非 Hamilton 图只有 $C_{1,n}$ 和 $C_{2,5}$。这里的 $C_{m,n}$ 的意义为 $C_{m,n}=K_m\vee(K^c_m+K_{n-2m})$。

范更华条件　设 G 是简单的 2-连通图，$n\geq 3$，如果 $d(v,u)=2\rightarrow\max(d(v),d(u))\geq n/2$，则 G 是 Hamilton 图。

注：Ore 条件还可以修改为：设 G 是 n 阶简单图，若存在不相邻的两个点 u,v 满足 $d(u)+d(v)\geq n$，G 是 Hamilton 图的充分必要条件是 $G+uv$ 为 Hamilton 图。

泛圈图　如果一个图包含长度为 3，4，5，…，n 的图，则把它称为泛圈图（Pancycle graph）。有两个关于它的定理：

Hakimi 和 Schmeichel 定理　n 阶简单图 G 的非降次序列 $d_1\leq d_2\leq\cdots\leq d$ 满足条件：

$$d_k\leq k<n/2\rightarrow d_{n-k}\geq n-k$$

则 G 为泛圈图或二分图。

Bondy 定理　n 阶（$n\geq 3$）简单图 G 满足：对所有不相邻的点 x，y 都有 $d(x)+d(y)\geq n$，则 G 为泛圈图或 $K_{n/2,n/2}$。

骑士周游　作为一个特殊情况，我们来看看骑士周游。对于一个 $n\times n$ 棋盘，如果骑士可以从某格 A 跳到格 B，就把 A 和 B 连上一条边，这样就得到了一个图。图有 Hamilton 圈吗？不一定。如果 n 是奇数，一定不存在 Hamilton 圈；（提示：先染色，再用奇偶分析法）如果 n 是偶数，一定存在。事实上，当 n 为偶数时我们有构造算法。

WEB 有趣的是，德国人 Martin 和 Ingo 还研究过这种 Hamilton 圈的个数。他们利用 ZBDD 计算出 8×8 棋盘上的骑士周游方案有 33，439，123，484，294 种，有兴趣的读者可以在本书主页上找到这篇论文。

2. 欧拉回路及其应用

无向图的欧拉回路　如果一个无向图所有顶点的度为偶数，那么该图可以用起始点与终止点相同的一笔画出，这一笔经历的路线叫做无向图的欧拉回路。

有向图的欧拉回路　如果一个有向图所有顶点的入度等于出度，那么该图可以用起始点与终止点相同的一笔画出，这一笔经历的路线叫做有向图的欧拉回路。

欧拉回路的套圈算法　首先检查图 G 是否符合一笔画的条件。如果符合，那么标记顶点 1 为待查找状态。过程 Euler(i)寻找开始于顶点 i 并且结束于顶点 i 的欧拉回路，具体步骤如下：

（1）寻找从 i 出发的环 $P_1P_2\cdots P_x$（$P_1=P_x=i$）；

（2）标记顶点 $P_{1\to x}$ 为待查找状态；

（3）对所有处在待查找状态的结点 P_j 递归调用过程 Euler(P_j)。

将 Euler(P_j)找到的环 $Q_1Q_2\cdots Q_y$ 插入到环 $P_1P_2\cdots P_x$ 中，得到回路 $P_1P_2\cdots P_jQ_2\cdots Q_yP_{j+1}\cdots P_x$。时间复杂度为 $O(m)$。

欧拉路　如果一个无向图恰有两个顶点 x，y 的度为奇数，那么该图可以用起始点于 x 与终止点于 y 的一笔画出，这一笔经历的路线叫做无向图的欧拉路。如果一个有向图中，顶点 x 出度比入度大 1，顶点 y 入度比出度大 1，其余所有顶点入度等于出度，那么该图可以用起始点于 x 与终止点于 y 的一笔画出，这一笔经历的路线叫做有向图的欧拉路。

欧拉路算法　寻找欧拉路的算法和寻找欧拉回路的算法只有一点不同。以无向图为例，欧拉路预处理的时候需要找出度为奇数的两个顶点 x 和 y，以及一条从 x 到 y 的路经 $P_1P_2\cdots P_k$（$x=P_1$，$y=P_k$），并初始化 $p[1]=k$，$p[i]=P_i$（$1\leqslant i\leqslant k$）。

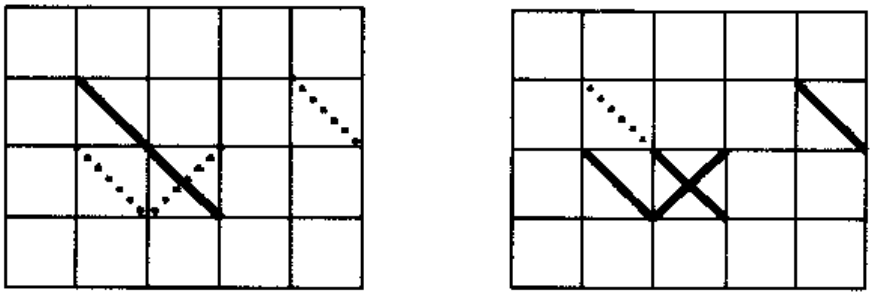
欧拉路的计数　欧拉回路和树型图有着有趣的联系。例如，我们可以利用树型图来数欧拉回路。有如下定理：如果 G 的每个点的入度等于出度，$t_i(G)$代表 i 为根的树型图数目，那么对于任意结点 i，欧拉回路的数目为 $s(G)=t_i(G)\prod_{j=1}^{n}(d^+(v_j)-1)!$。

可以用矩阵法求出有根树型图的数目，这里不再叙述。

WEB　我们可以对欧拉图的定义作扩展，即得随机欧拉图(randomly eulerian)。通俗地说，就是“不管怎么走，只要不刻意的去重复走同一条边，一定能走出一条欧拉回路而不会在途中无法继续。”这些内容可以在本书主页上找到。

【例题 1】绣花[1]

考古学家发现了一块布，布上做有针线活，叫做“十字绣”，即交替地在布的两面穿线。布是一个 $n\times m$ 的网格，线只能在网格的顶点处才能从布的一面穿到另一面。每一段线都覆盖一个单位网格的两条对角线之一，而在绣的过程中，一针中连续的两段线必须分处布的两面。如图 2-30 所示，给出布两面的图案（实线代表该处有线，虚线代表背面有线），最少需要几针才能绣出来？一针是指针不离开布的一次绣花过程。例如图 2-30 的图案最少需要 4 针。



布的正面　　布的背面

图 2-30　布的两面

[1] 题目来源：Ural State University Problem Archive

【分析】

把网格的交叉点抽象成图的顶点，把布正面的线抽象成图的正边，布背面的线抽象成图的负边。有边相连的顶点形成一个连通块，分别求出每个连通块所需的针数，最后相加即为问题的解。

考虑某个连通块，对于其中的某个顶点，正边和负边的刺绣一定是交替进行的。所以如果|正边数-负边数|=K>0，那么以该顶点作为开始端点或者结束端点的针数至少为 K。事实上，如果某一针的开始端点和另一针的结束端点是同一个顶点，那么可以将两针连成一针，所以应该说以该顶点作为开始端点或者结束端点的针数恰好为 K。于是，把所有顶点正负边数的差相加再除以 2（除以 2 是因为每一针都有开始和结束共 2 个端点）即是该连通块的针数。值得注意的是，如果差的和为 0，那么该连通块只用了一针且该针开始和结束端点相同，所以针数为 1 而不是 0。

【例题 2】漆门[1]

幼儿园里有很多房屋，房屋与房屋之间连以走廊，走廊与房屋之间有一扇门。幼儿园长想把门漆成绿色或者黄色，使得：任意一条走廊两头门的颜色不同；任意一间房屋上的门，绿色门的数量与黄色门的数量相差不超过 1。

【分析】

“绿色门的数量与黄色门的数量相差不超过 1”。这很难不让人从房屋联想到欧拉路中的顶点。事实上，如果每个房屋门的数量为偶数，那么幼儿园本身就是个欧拉回路。用求欧拉回路的办法遍历所有走廊恰好一次，并且规定：如果从房屋踏上走廊，那么之间的门漆上绿色；如果从走廊踏上房屋，那么之间的门漆上黄色。走廊两头连着房屋，每条走廊被踏进一次，随后被踏出一次，根据上述漆门规则，任意一条走廊两头门的颜色不同。同时，由于每个房屋被踏进和踏出的次数一样，所以任意一间房间上的门，绿色门的数量与黄色门的数量相差为 0。

当然，存在欧拉回路是最理想的状态。对于不是欧拉回路的幼儿园，也可以将它改造成欧拉回路。怎么改造呢？很简单。根据图的性质，门数量为奇数的房屋的个数一定为偶数个，所以将这样的房屋两两任意配对，在每一对间增加一条虚设的走廊。这样，幼儿园就成了欧拉回路，我们在给幼儿园漆完门后，撤去所有的虚设的走廊和相应的门。由于每个房屋被撤去的门最多只有一扇，所以同样保证任意一间房屋上的门，绿色门的数量与黄色门的数量相差不超过 1。

【例题 3】原始基因[2]

一个抽象化原始基因的编码是一串连续的整数 $K=(a_1, a_2, \cdots, a_n)$，一个原始基因的特征是一个有序的整数对$(l, r)$，它在基因编码中连续出现，即在原始基因中存在 i，使 $a_i=l$，$a_{i+1}=r$。在基因序列中不会出现形如（p,p）的特征。

编程计算包含这些特征的最短原始基因。

【分析】

对于本题，我们可以建立一个有向图的模型，把每个整数看作图中的顶点，每一个有

[1] 题目来源：Ural State University Problem Archive
[2] 题目来源：Polish Olympiad in Informatics

序整数对看作一条有向边。问题转化为求在一个有向图中加最少的边，使图中存在欧拉道路（或回路）。可以知道：若一个图中已经有欧拉回路，则不需要加边；否则可以先加边使图中存在欧拉回路，然后再任意去掉一条边，求可以得到一条欧拉道路。以下只考虑求欧拉回路。

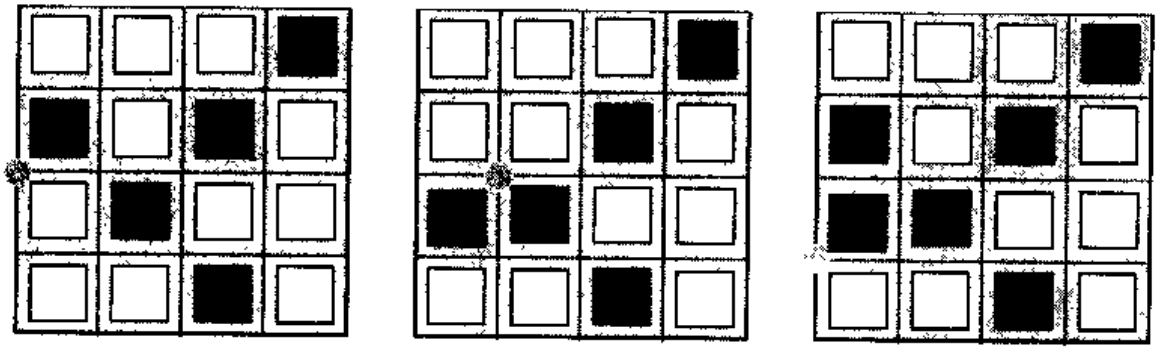
先考虑一个不存在欧拉回路的只有一个连通分支的有向图。刚才已经讲过，一个有向图存在欧拉回路的充要条件是图中只有一个连通分支且每一点的出度等于入度，因此需要加上边，使每个点的出度等于入度，这样图中就会有一条欧拉道路。具体的做法是求出每一点入度出度差的绝对值的和 m，$m/2$ 就是形成欧拉回路所需要加的边数。

然而事实上图中可能存在不止一个连通分支，这就需要将所有连通分支连起来。如果某一个连通分支中有一个点曾经加过一条边，则可以令这条边的另一个顶点在其他连通分支中，这样可以将这个连通分支于其他的连通分支连接起来。如此，只要这个连通分支中原先不存在欧拉回路，就可以通过加边的方法使它存在连通分支，同时将它和其他连通分支连接起来；反之，如果某个连通分支中没有一个点曾增加过边，则需要另加一条边以使这个连通分支与其他连通分支连接起来。

【例题 4】超级翻转[1]

有一个 $N\times N$ 的网格，每一“格”上有一个可以翻转的方块，方块的两面分别涂成黑、白两种颜色。另外，有一个沿着网格线活动的东西——不妨称之为“动子”。初始时，每个方块随机地被翻成黑或白色，“动子”停在网格线的某个顶点上。例如图 2-31(a)就是一个 4×4 的网格的一种可能的开局情况。

动子在网格线上运动时，从一个顶点 A 到相邻的另一个顶点 B 之后，以网格线 AB 为边的两个或一个网格上的方块就会翻转——白变黑，黑变白。例如图 2-31(a)的“动子”向右移动一步之后变成图 2-31(b)，向下移动一步之后变成图 2-31(c)。



(a)　　(b)　　(c)

图 2-31　三个游戏布局

问题是：给定一个初始状态和一个目标状态，求“动子”的一种运动轨迹，可以把初始状态翻转成目标状态，最后“动子”停在哪里是无所谓的。

【分析】

问题中所关系到的对象有三个：格、边、顶点（动点所能停留的位置）。题目要求解

[1] 题目来源：IOI2003 中国国家集训队原创题目，经典问题

的目标是动点的一个运动轨迹——若干相邻的边，由于经过某条边的时间先后并没有任何的关系，我们可以暂时忽略边的顺序，只考虑每条边经过的次数。

① 一个格的四周有 4 条边，这 4 条边每经过一次，此格的状态就改变一次。那么如果这 4 条边经过了奇数次，则这个格子将会改变状态（即最初是黑（白）则会变为白（黑）），否则状态不变。

反过来说：若一个格子的初末状态不同，则它周围的边必经历了奇数次，否则为偶数次。

② 设动子起点和终点（动子最后停留的位置，通过枚举得到）为 S 和 T，则对于某个顶点 $V(V\neq S,T)$，V 相邻的几条边必定总共经过了偶数次，因为动子每次到达这个点后必定会离开。当 $S=T$ 时，S（T）相邻的几条边也必定总共经过了偶数次；当 $S\neq T$ 时，S 或 T 相邻的几条边必定总共经过了奇数次。

上面两个边的次数的关系，是两个满足条件的路径必要的条件。但是否充分呢？

答案是肯定的。

为了减少一些不必要的移动，我们可以将次数大于 1 的边不断的减少 2，直至变为 0 或 1，这样仍然能够满足上面两个条件。然后再把路径中的顺序考虑进来：若每条边的经过的次数已知了，怎么安排它们的顺序，得到一个可行的路线呢？如图 2-32 所示。

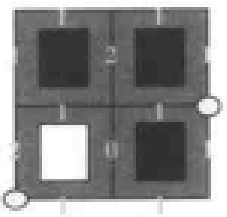


图 2-32　行走顺序的安排

这很容易让人联想到欧拉路：如果所有要走的边都是连通的，那么存在一条 S 到 T 的欧拉路，这条路显然就是满足题意的。问题是，如果有多个连通块呢？其实这也很容易处理，可以通过添边，使得在仍然满足上述两个条件下，连通分量的数目减少。

具体地说：若有多个连通块，任选两个不同连通分块中的点 a,b，找一条 a 到 b 的路径，将路径上的边的次数都加 2。最终就能使得所有次数大于 0 的边连通，然后就转化为一笔画问题——也就是求欧拉路，如图 2-33 所示。

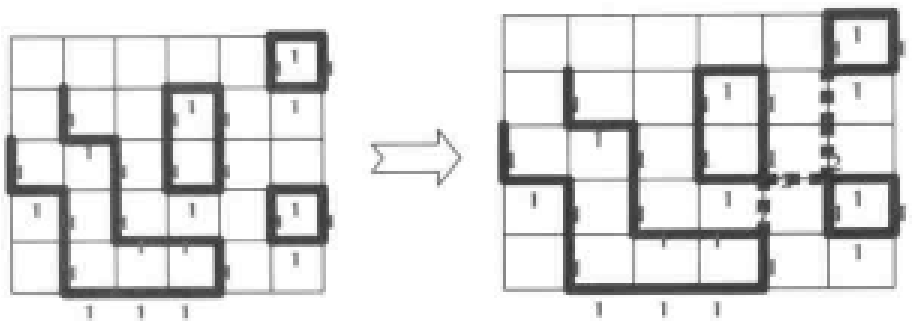


图 2-33　多个连通块的处理

其实，为了编程更简单，使所有边的访问次数都增加 2 就行了。不过这时走的步数就太多了。

有了前面的简单分析，问题的解决就十分容易了：

步骤 1　设 Xi=0 或 1，表示某条边访问了的次数，若 Xi=0 则为偶数次，否则为奇数次。

对于某个格子 i，若它的初末状态不同，则它相邻的所有边的和 mod 2 = 1，否则和 mod 2=0。对于某个顶点 i,如果$(i\neq S)$and$(i\neq T)$or$(S=T)$那么所有它相邻的边的和 mod 2=0,否则和 mod 2 = 1；

步骤 2　解出满足上面条件（边数有 $2n(n+1)$，关系式 $n\times n+(n+1)\times(n+1)=2n(n+1)+1$ 个，实际上就是一个 $2n(n+1)$个未知数 $2n(n+1)+1$ 个方程的模线性方程组）；

步骤 3　合并连通块；

步骤 4　求一条从 S 到 T 的欧拉路。

练　习　题

编程题：

2.4.1　广场上的地板砖[①]

Richard 是一个数学爱好者，他总是把所有东西都和数学问题联系起来。有一天黄昏，他和往常一样散步，广场上的地板砖让他产生了灵感，想起了一道组合问题。

地板转是一块块边长为 1 米的小三角形。如果把四片砖放在一起，可得到一个边长为 2 米的大三角形，如图 2-34 所示。Richard 把四块砖编号为 1，2，3，4，然后用红，绿，蓝三种颜色给每块砖涂上颜色。由于相同的颜色连在一起会比较难看，所以他觉得任意两块有公共边的三角砖都不能涂上相同的颜色。他很快就算出了所有 24 种着色方案。由于砖是有编号的，所以旋转或者对称后看起来一样的方案也被看作是不同的。

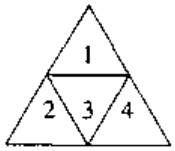


图 2-34　由四块三角砖组成的边长为 2 米的大三角形

如果用更多的三角砖组成更大的大三角形（例如用 9 块砖组成边长为 3 米的；用 16 块砖组成边长为 4 米的……），用更多的颜色去着色，方案数是多少呢？Richard 把这个问题留给你，让你写个程序算一算。假设大三角形的边长不超过 6 米，颜色不超过 4 种。

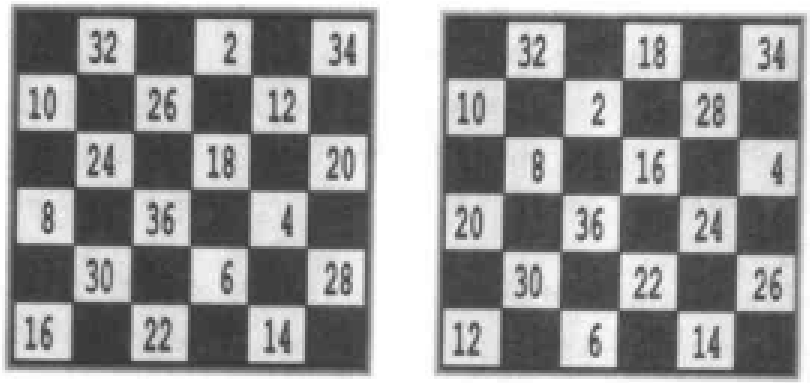
2.4.2　骑士周游[②]

给一个 $N\times N$（$N\leqslant 100$）的棋盘，求一个骑士周游。图 2-35(a)给出了一个回路，而图 2-35(b)是一条路径，因为它的起点和终点不重合。棋盘上的数字表示骑士到达该格子时跳过的步数。

2.4.3　Gridland[③]

长期以来，计算机科学家们着力寻找一种有效的方法来处理困难的计算问题。由于一些问题有效的算法已经找到，像排序、计算多边形面积、最短路等问题都成为了“简单”的问题。而困难的问题，我们只得到了些非多项式算法，货郎担问题就是其中之一：给出 N 座城市和它们之间的连接关系，找出一条对每个城市访问一次且仅一次的最短路。

[①] 题目来源：ACM/ICPC Regional Contest Shanghai 2000
[②] 题目来源：经典问题
[③] 题目来源：ACM/ICPC Regional Contest NWERC 2001



(a)骑士周游回路　　(b)骑士周游路径

图 2-35　骑士周游

Gridland 的总统雇佣你来编一个程序，计算在 Gridland 中货郎担环游问题。在 Gridland 中，每个城市都在一个矩形网格的点上。城市中的道路只有东、西、南、北、东南、东北、西北、西南 8 个方向。东西、南北方向的道路长度都为 1。城市中的道路长度就是图上的欧几里德距离。图 2-36 中，Gridland 在一个 2×3 网格上，最短的环游路线长度为 6。

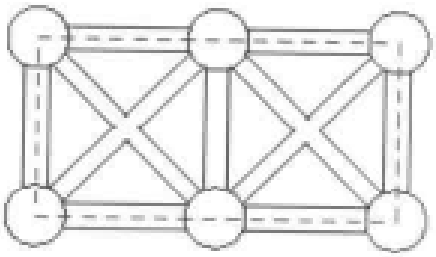


图 2-36　Gridland 地图

2.4.4　邮递员[①]

邮递员每天都辛辛苦苦的工作，沿着每条路给村庄里（总共有 n 个村子）的人送信。村子之间有道路相连，形成一个网状结构。每个村子可能处于两条路的十子路口上（如村庄 1），4 条路的米子路口上，或者一条路的中央（如村庄 3，5），如图 2-37 所示。

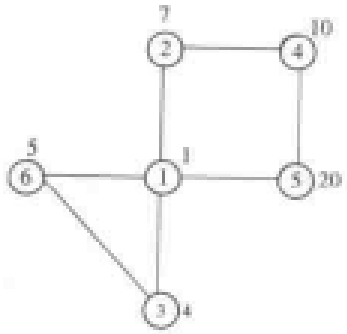


图 2-37　村庄地图

[①] 题目来源：Baltic Olympiad in Informatics, 2001

村子里的人都希望自己村子能尽量早地收到信件，因此和邮递员达成了一个协议：每个村庄都有一个期望值 W_i，如果该村子是邮递员经过的第 K 个不同的村子，那么如果 $K \leqslant W_i$，则村子付给邮递员 W_i-k 元钱，否则邮递员付给村子 $K-W_i$ 元。另外，每经过一条不同的路，邮局会付给邮递员 1 元钱，而邮局规定每条路（总共 m 条）都至少经过一次。邮递员应该怎样走，才能挣最多的钱呢？

2.4.5 城市观光[①]

城市旅游代理 TAB 要组织一个巴士观光的计划。公司的总部，即每次观光的出发和结束的位置需要确定。观光的线路要包括城市所有的街道，否则顾客会怀疑有一些有趣味的景点被错过了。街道不都是直的，每条街道连接了两个十字路口，而且街道都是双向的。每个十字路口连接了 4 个街道，两个十字路口可能有多条街道连接。在某一条街道的中间，汽车不能调头。从任何一个路口都可以到达另外一个路口。每条街道的中心，都有景物，例如是雕塑或者是纪念碑等。每一条街道的吸引力用一个非负整数来描述。TAB 的总部应当建立在某一条街道的中央，靠近某一个景物。制订一条路线的时候，我们需要考虑旅游的有趣程度。有趣程度在观光的过程中是不断改变的。汽车每前进 1 英里，有趣值就减 1。当第一次看见某一个景物的时候，有趣程度值就加上这个景物的吸引力值。旅行开始时，有趣程度值等于 TAB 总部旁的这个景物的吸引力值。我们说一个旅行计划是有吸引力的，是指在旅行的过程中，有趣程度值从不为负。

依据城市的描述，找到满足条件的线路，或者指出不存在这样的线路。

2.4.6 赌博机[②]

一台赌博机有 n（$n \leqslant 1\,000$）个发生器 $G_1,G_2,\cdots,G_n$。G_i 可以产生的自然数集合是 S_i（S_i 中只包括 1 到 n 的自然数，S_i 可以是空集）。令 k_i 表示 S_i 中数的个数，则所有 k_i 的和不超过 12 000。

第一次活动时 G_i 从 S_i 中选择一个自然数显示。以后 G_i 每次从 S_i 中选择一个以前没有选择过的数显示。如果 S_i 中所有的数都已选过，则它显示 0。

开始时 G_1 是活动的，然后赌博机这样运行：如果当前发生器显示了一个数 r，则下一个活动的发生器是 G_r，如果显示的数是 0，则机器停止运转。

如果 G_n 产生了 0，并且所有的 S_i 为空，那么机器失败。如果机器在停止时没有失败，那么它获胜。

2.4.3 平面图

一个无向图 $G=<V, E>$，如果能把它画在平面上，且除 V 中的结点外，任意两条边均不相交，则称该图 G 为平面图。例如：图 2-38（a）经变动后成为 2-38（b），故图 2-38（a）为平面图。而图 2-38（c）无论如何变动，总出现边相交，故图 2-38（c）为非平面图。

---

① 题目来源：Polish Olympiad in Informatics, 2001

② 题目来源：Polish Olympiad in Informatics

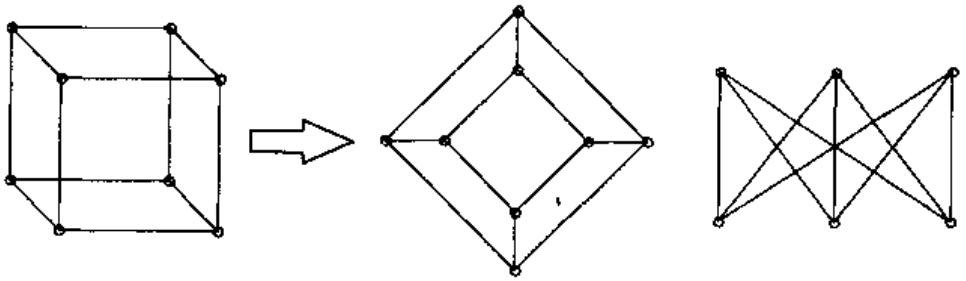


图 2-38　平面图

基本概念　设 G 是一个连通平面图，由图中的边所包围的区域，在区域内既不包含图的结点，也不包含图的边，这样的区域称为图 G 的一个面，包围该面的诸边所构成的回路称为这个面的边界。面的边界的回路长度称作该面的次数。可以证明，对于一个有限平面图，面的次数之和等于其边数的两倍。有些面在图形之外，不受边界约束，称做无限面。如果我们把图形看作包含在比整个平面还要大的一个矩形之内，那么在计算图形面的数目时，就不会遗漏无限面了。

对偶图　给定平面图 G，它有面 F_1，F_2，…，F_n，若有图 G^*满足下述条件：

① 对于图 G 的任一个面 F_i，内部有且仅有一个结点 $v_i^* \in V^*$。

② 对于图 G 的面 F_i，F_j 的公共边 e_k，存在且仅存在一条边 $e_k^* \in E^*$，使 $e_k^*=(v_i^*, v_j^*)$，且 e_k^*和 e_k 相交。

③ 当且仅当 e_k 只是一个面 F_i 的边界时，v_i^*存在一个环 e_k^*和 e_k 相交，则图 G^*是图 G 的对偶图。根据一个图得到它的对偶图实际上是将该图的结点改变成为对偶图的边，该图的边改变为对偶图的结点。

关于平面图的几个重要的参数，我们有如下定理：

欧拉定理　设有一个连通平面图 G，共有 v 个结点 e 条边 r 块面，则 $v-e+r=2$。

推论 1：给定连通简单平面图 $G=<V, E, F>$，若$|V| \geqslant 3$，则$|E| \leqslant 3|V|-6$ 且$|F| \leqslant 2|V|-4$。

推论 2：设 $G=<V, E, F>$是连通简单平面图，若$|V| \geqslant 3$，则存在 $v \in V$，使得 $d(v) \leqslant 5$。

这个定理告诉我们，在平面图的算法中，可以取 $m=O(n)$. 我们可以用邻接表而不是邻接矩阵，也可以用前向星或者哈希表来存。

【例题 1】地图的五着色[1]

给定一地图，要求用不超过 5 种颜色涂每一个区域，使得相邻区域的颜色不同。（区域数≤500）

【分析】

把每个区域看成点，相邻区域之间连一条边，则问题转化为对每个点着色并使得相邻点颜色不同。根据地图的平面性可知：转化后的图是平面图。问对任意平面图 G，是否都能用不超过 5 种颜色着色？

[1] 题目来源：经典问题

定理：对于任意平面图 G，都能用不超过 5 种颜色着色。

证明：只需考虑 G 是连通简单平面图的情形。

- 若$|V|\leqslant 5$，则命题显然成立。
- 假设对所有的平面图 $G=<V, E>$，当$|V|\leqslant k$ 时命题成立。现在考虑图 $G_1=<V_1, E>$，$|V_1|=k+1$ 的情形。由推论 2 可知：存在 $v_0\in V_1$，使得 $d(v_0)\leqslant 5$。在图 G_1 中删去 v_0，得图 G_1-v_0。由归纳，图 G_1-v_0 可用 5 种颜色着色。下面分两种情况。

情形一：v_0 的邻接结点使用色数不超过 4，则可对 v_0 着色，得到一个最多是五色的图 G_1。

情形二：$d(v_0)=5$ 且各邻接点分别着不同的颜色，设与之相邻的点按顺时针排列为 v_1,v_2,v_3,v_4,v_5。它们分别着不同的颜色 c_1,c_2,c_3,c_4,c_5。我们的想法是调整 $v_1\sim v_5$ 的颜色，让 G_1-v_0 的着色方案仍然可行，且 $v_1\sim v_5$ 一共只用不超过 4 种颜色。

考虑在 G_1-v_0 中，颜色为 c_1 或者 c_3 的点（记为点集 V'）所诱导的 G_1 v_0 的子图 H，显然 v_1 和 v_3 都在 H 中（因为 v_1 的颜色为 c_1，v_3 的颜色为 c_3）。下面又分两种情况。

情形一：v_1,v_3 属于 H 的不同连通分支，则在 v_1 所在的连通分支中调换颜色 c_1 与 c_3 后，v_1 所在连通分量的着色方案仍然可行，且 v_1 的颜色改为 c_3，但 v_3 的颜色不变。这样，v_1,v_2,v_3,v_4,v_5 五个点是四着色的，再令 v_0 着 c_1 色，得到 G_1 的一种五着色.

情形二：若 v_1,v_3 属于 H 的同一连通分量，则点集 $V'\cup\{v_0\}$所诱导的 G_1 的子图中含有一个圈 C（因为 H 中 v_1 和 v_3 本来就连通了，而加入 v_0 后新增一条路径 $v_1\rightarrow v_0\rightarrow v_3$）。由于 $v_1\sim v_3$ 是顺时针排列的，因此 v_2,v_4**不能同时在圈的内部或外部**，这样，考虑在 G_1-v_0 中，颜色为 c_2 或者 c_4 的点所诱导的 G_1-v_0 的子图 H 中，v_2 和 v_4 必属于不同的连通分支。做与上面类似的调整，又可得到 G_1 的一种五着色。

故对任何连通简单平面图 G，G 是五着色的。根据刚才的证明，不难得到一个算法。

算法如下：

```
procedure Paint(G:Graph);
begin
   找出度最小的点 v0
   Paint(G-v0)
   if 在图 G 中无法对 v0 着色 then
      对 v0 的相邻点枚举所有点对，直到找到属于不同分图的点对，对其进行调整。
   任选剩下的一种颜色，对 v 着色
end;
```

算法的时间复杂度：$O(N^2)$，空间复杂度：$O(N)$。

【例题 2】滑雪[1]

一个滑雪队在 Byte 山上组织了一次训练。山的北坡有一个滑雪场，所有的滑雪者都要从山上的起点站滑到山下的终点站。此次训练中各队员同时出发到终点站会合，除了始末两处外，队员们的滑雪路径不能相交，且山上的滑雪道只能从上往下滑。

[1] 题目来源：Polish Olympiad in Informatics

滑雪道的分布地图由多块被林地连接的空地组成，每块空地都处于不同的高度。两块空地间至多由一块林地连接。滑雪过程中，滑雪者可以选择路径访问任一空地（但不必全部经过）。各滑雪道只在空地相会，既不穿隧道，也不临空飞越。编写一个程序算出能参加训练的最大队员数。

【分析】

求互不相交的路径的最大数目，很容易把路看作流，进而想到用类似最大流参见 2.5 节的做法来解题。这种做法的核心如图 2-39 所示。

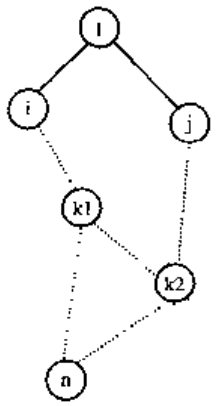


图 2-39　滑雪问题的最大流模型

先能将确定的一条路径 $i \to k1 \to k2 \to n$ 修改扩展为两条路径：$i \to k1 \to n$ 和 $j \to k2 \to n$。

但本题描述的图十分特殊，边之间不交错且各点的后继点都按从左到右给出，于是按顺序是不可能先找到路径 $i \to k1 \to k2 \to n$。因为在 $k2$ 的左边存在 $k1$ 的后继点，一定先找到路径：$i \to k1 \to n$。这样问题就很简单了，只需用贪心策略，深度遍历此图得到的路径数，就是最大的。

【例题 3】水平可见线段的三角形[1]

平面上有 N（$N \leqslant 8\ 000$）条互不相连的竖直线段。如果两条线段可以被一条不经过第三条竖直线段的水平线段连接，则这两条竖直线段被称为“水平可见”的。三条两两“水平可见”的线段构成一个“三元组”。求给定输入中“三元组”的数目（坐标值为 0 到 8 000 的整数）。

【分析】

我们的工作分两步进行，一是求出两两线段是否可见的关系图，二是根据关系图数出垂直可见三角形的数量。

步骤 1　我们使用线段树帮助判断。

初始时线段树为空，以后按照 x 坐标递增顺序逐一用垂直线段覆盖树中结点，并记录覆盖关系，也即是可见关系。

为了叙述方便，我们用数组的形式来描述线段树，定义如下：

- $g[1]$表示 y 坐标 0→16 000 范围内被编号为 $g[1]$的线段覆盖。
- $g[k]$表示 y 坐标 $p1 \to p2$ 范围内被编号为 $g[k]$的线段覆盖。($p1$、$p2$ 见如下递归定义)

编号为 0 的线段是空白线段，也即是说此范围内未被任何线段覆盖；$g[k]=-1$ 表示 y 坐标 $p1 \to p2$ 范围内被不只一条线段覆盖，此时循环定义 $g[2k]$表示 y 坐标 $p1 \to (p1+p2)$ div 2 范围内的覆盖情况，$g[2k+1]$表示 y 坐标 $(p1+p2)$ div 2+1 $\to p2$ 范围内的覆盖情况。

覆盖：假设当前将用线段 i 覆盖树中结点，它跨越 y 坐标范围 $i1 \to i2$；结点 k 记录 y 坐标范围 $p1 \to p2$ 内的覆盖情况；$a[i][j]$=true 表示线段 i、j 水平可见，那么

① 如果 $g[k] \geqslant 0$ 并且 $i1 \leqslant p1$，$p2 \leqslant i2$，记录 $a[g[k]][i]=a[i][g[k]]$=true，同时赋值 $g[k]=i$。

[1] 题目来源：ACM/ICPC Regional Contest CERC 2001

② 如果 $g[k]=-1$ 或者 $i1>p1$，$p2>i2$，那么二分处理结点 $g[2k]$和 $g[2k+1]$。

预处理：需要补充说明一下刚才遗漏的问题，一是需要使用 Qsort 将所有线段按照 x 坐标递增顺序排列，二是试题中指明 y 坐标范围 0→8 000，为什么却要将根结点 g[1]的范围设成 0→16 000 呢？请看图 2-40。

线段 i 和 j 是水平可见的，但由于它们没有共同覆盖一段长度≥1 的线段，所以在覆盖过程中，会误认为 $a[i][j]$=false。怎么办呢？很简单，用结点 $k\times2$ 表示端点 k 的覆盖情况，用结点 $k\times2-1$ 表示开区间线段$(k-1,k)$的覆盖情况。这样，线段 i 和 j 共同覆盖结点 $3\times2=6$，在覆盖过程中我们就可以判定 $a[i][j]$=true 了。

复杂度：单就某条垂直线段 i 来讲，它可能覆盖多达 $i-1$ 条线段，需要处理 $O(i)$数量级的结点，但总体上说，结点被修改的总次数是达不到 $O(n^2)$的。为什么呢？请看图 2-41。

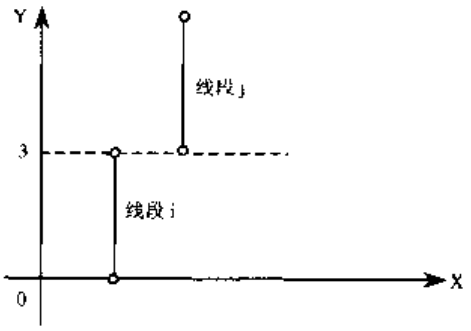


图 2-40　预处理复杂度

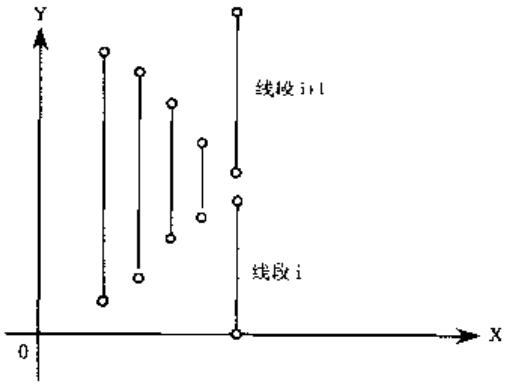


图 2-41　复杂度估计示意图

图中一旦线段 i 和 $i+1$ 覆盖了前面的所有 $i-1$ 条线段，线段 $1\rightarrow i-1$ 就永远不会再被覆盖了。线段 i 覆盖的线段越多，线段 $i+1\rightarrow n$ 可能覆盖的线段就越少。注意到水平可见关系构成一个平面图（想一想，为什么），根据前面的结论，此图中边数不超过顶点数的 3 倍，故可见线段对的数量不会太多。事实上，结点修改总平摊复杂度是耗费在线段树上的开销，即 $O(n\log n)$。

步骤 2　根据欧拉定理的推论 2，存在 $v\in V$，使得 $d(v)\leqslant5$。

我们找出这个度≤5 的线段，判断与它相邻的任意两线段是否可见。如果可见，便构成了一个垂直可见三角形。判断完毕后，将这条线段及相应的边从关系图中删除，随后寻找下一个度≤5 的线段，直到图为空。当然，我们需要合理的数据结构帮助完成。

步骤 2 的复杂度只有 $O(n)$，虽然它的常数因子大了点。

WEB　一个很自然的问题是：如何判断一个图是否为平面图？如是，找出一个平面嵌入方案　关于平面性判别，有一个简单的 Kuratowski 定理，但是它并不实用。好在我们还可以用代数方法来研究，并且利用它得到这个问题的线性时间复杂度的算法。在平面图上讨论连通性、着色问题等都有一些有趣的结论，有兴趣的读者可以阅读本书主页

2.4.4　图的基本算法与应用举例

本节学习图最基本的算法和它们的应用。本节实践性强且难度比较大，希望读者用心体会其中的思想。

下面学习 DFS（Depth First Search，深度优先搜索）。它可以帮助我们获取图的信息，方法是在遍历的时候记录一些东西，然后把它们加以整理。具体来说，需要记录两种东西：时间戳和结点颜色。本节给出的所有算法的时间复杂度均为 $O(m)$。

时间戳和结点颜色　通过盖印时间戳和标记结点颜色来记录信息。时间戳 A_i 表示结点 i 是第 A_i 个被遍历并检查完毕的结点，结点颜色 C_i 有三种状态：白色表示结点 i 尚未被遍历；灰色表示结点 i 已经被遍历但尚未检查完毕；黑色表示结点 i 不但被遍历而且被检查完毕。初始时，标志一切结点为白色，时间戳为 0，然后对根 Root 进行 DFS(Root)的遍历。

图的深度优先搜索　过程 DFS(i)是这样定义的：

（1）标记 C_i 为灰色；

（2）依次检查和 i 相连的每个白色结点 S_j，一旦找到，立即对 S_j 施行 DFS(S_j)遍历；

（3）遍历结束后查找下一个和 i 相连的白色结点，对该结点进行同样的 DFS 遍历……

直到检查完所有和 i 相连的白色结点，标记 C_i 为黑色。时间戳+1，A_i= 时间戳。下面给出算法框架。加注释的部分将被用来求割顶和桥。

```
Procedure DFS(结点编号 k，k 的父亲结点编号 father，deep: integer;)
var i,tot:integer;
begin
  染色 Ck = 灰色;
  Dk = deep 记录顶点 k 在树中的深度。
  //①Ancestork = deep，Tot = 0

  for i = 1→n
  begin
   //①if（结点 i 和 k 相连）and（i ≠ father）and（Ci = 灰色）
   //①    then Ancestork = Min{Ancestork，Di};

   if（结点 i 和 k 相连）and（Ci = 白色）then begin
      DFS(i,k);

      //①tot = tot+1，Ancestork = Min{Ancestork，Ancestori}
      //①if（（k = Root）and（tot>1））or
      //①  （（k ≠Root）and（Ancestori ≥ Dk））
      //①  then Cutk = true
      //②if（Ancestori>Dk）then Brigek,i = true
```

```
    end;
  end;
  染色 C_k = 黑色
end;
```

DFS 遍历本身除了查找连通块外并没有太多用处，关键是在遍历的同时，可以记下很多有用的信息。下面来看看哪些是与 DFS 相关的常用信息：

无向连通图的割顶　我们先考虑割顶的性质

（1）考虑根顶点 Root。如果顶点 x 和 y 同是 Root 的儿子，那么由此证明 x 无法通过非 Root 的顶点与 y 相连，所以当根 Root 有数量>1 的儿子时，根是图的割顶。

（2）考虑非根顶点 i，再考虑 i 的某个儿子结点 j。易知：

① 和 j 相连的白色结点都将成为 j 的子孙。

② 和 j 相连的灰色结点都是 j 的祖先，由 j 指向 i 祖先的边称为后向边。

③ 黑色结点不可能与 j 相连。

如果 j 和 j 的子孙都不存在指向 i 祖先的后向边，那么删除顶点 i 后，顶点 j 和 i 的祖先或者兄弟将无法连通。因此，当且仅当 i 的某个儿子及儿子的子孙均没有指向 i 祖先的后向边时，i 是图的割顶。

割顶的求法　在 DFS 框架的基础上增加 $Ancestor_k$ 和 Tot 值的计算。

$Ancestor_k$ 记录和 k 及 k 的子孙相连的辈份最高的祖先，当 $Ancestor_j<D_i$（j 是 i 的儿子）时 j 和 j 的子孙存在指向 i 祖先的后向边。Tot 表示顶点 k 的儿子的数量。注意，根与非根顶点要区别对待。

将刚才 DFS 算法框架中的“//①”的注释符号删除，就得到了求割顶的算法。Cut$k$ = true 表示记录顶点 k 为图的割顶。

无向连通图的桥　如果 y 是 x 的儿子并且 $Ancestor_y>D_x$（注意不是 $Ancestor_y \geq D_x$），那么删除边（x,y）后，顶点 y 将与 x 不连通，所以（x,y）是图的桥。

桥的求法　它也是基于 DFS 框架的算法，将刚才 DFS 算法框架中的“//①”和“//②”的注释符号删除后，就得到了求桥的算法。$Brige_{k,i}$ = true 表示记录边（i,k）为图的桥。

有向图的极大强连通分支　我们直接给出算法。

（1）对图进行 DFS 遍历，遍历中记下所有的时间戳 A_i。遍历的结果是构建了一座森林 $W1$，我们对 $W1$ 中的每棵树 G 进行步骤（2）到步骤（3）的操作；

（2）改变图 G 中每一条边的方向，构造出新的有向图 Gr；

（3）按照 A_i 由大到小的顺序对 Gr 进行 DFS 遍历。遍历的结果是构建了新的森林 $W2$，$W2$ 中每棵树上的顶点构成了有向图的极大强连通分支。

下面给出证明：

① 考虑图 G 中属于同一强连通分支的两个顶点 x 和 y。

因为 G 中 x 和 y 可以互达，所以 Gr 中 x 和 y 也可以互达。x 和 y 一定在 $W2$ 中的同一棵树上。

② 考虑 $W2$ 中以顶点 Root 为根的同一棵树上的两个顶点 x 和 y。图 Gr 中从 Root 到 x

有一条路，因而图 G 中从 x 到 Root 有一条路，所以 Root 要么是 x 的祖先，要么是先于 x 遍历完毕的旁系亲戚。又由于 $A_{root}>A_x$，所以 Root 不可能是先于 x 遍历完毕的旁系亲戚，而只可能是 x 的祖先，所以图 G 中存在从 Root 到 x 的一条路，所以 x、Root 是互达的顶点对。x 和 Root 互达，同理 y 和 Root 也互达，所以 x、y 亦互达。

综上所述，当且仅当 x、y 是 $W2$ 中同一棵树上的顶点时，x、y 互达。$W2$ 中的每棵树构成了一个有向图的极大连通分支。

图的收缩　没有割顶的无向图又称作 2-连通分支，亦称作块。把每个块收缩成一个点，就得到一棵树，它的边就是桥。根据树的性质，从任意一个块 A 经过任意一条简单路径走到另一个块 B 所经过的桥集合是相同的。利用这个结论，可以解决下面的问题。

【例题 1】往返路[1]

在城市交通图中有 N（$N\leqslant 1\,000$）个城市和 M（$M\leqslant 10\,000$）条连接它们的双向路。你所在的旅游团要从城市 A 经过一些城市后到达城市 B，玩耍一番以后再返回城市 A。为了使旅途更加愉快，你得为旅游团设计两条路线（A 到 B，B 到 A），使得每条路线都不重复地经过任何一条路，而同时要求包含在两条线路中的路的数目尽量少。

【分析】

利用刚才介绍的找“桥”算法，我们首先找到图中的桥，然后任意找 A 到 B 的一条简单路。路上经过的“桥”都是 A 到 B 的必经路。只要我们在交通图中去掉这条路上非桥的边后再找一条 B 到 A 的路（由于我们并未去掉桥，所以图仍然连通），则两条路中公共的路恰好是它们的必经路，这个数目是最少的。

【例题 2】连通图编号问题[2]

任给一个 M 条边的连通图，给它的每条边不重复遗漏地用 $1\cdots M$ 的整数编号，使得任意与顶点 u 关联的所有边的最大公约数为 1。

【分析】

利用 DFS 得到的方法异常简单：对图作一次 DFS，按照访问的顺序依次给每条边编号就可以了。第一个顶点出发的边有一个编号为 1；而其他所有顶点出发的边都有两个的编号相邻（想一想，为什么？），因此对于每个顶点 u，u 连出的所有边的编号的最大公约数为 1。用 BFS 无法到达这个要求，因为它无法保证其他顶点出发有两个相邻编号。

【例题 3】跳舞蝇[3]

Byteland 一直以奇妙的跳舞蝇而闻名于世。驯养的苍蝇能和着音乐的节奏精确地做每一次飞跃。通常，训练者会在桌上放一排硬币，这些硬币的排列并不按照特定的顺序。每枚硬币上都有一行题字：$i\to j$，i 是这枚硬币的编号，j 是站在硬币 i 上的苍蝇下一步应该飞往的硬币编号。训练者在每个硬币上放一只苍蝇，然后开始放音乐。那些苍蝇就跟着音乐的节拍开始跳舞，在每一拍中苍蝇都会直接跳到编号为 j 的硬币上。在舞蹈中，可能会出现多只苍蝇在同一硬币上的情况。这样，跳舞蝇就会一起继续表演。假定有 n 只苍蝇，n 枚硬币。则一旦确定了 n 枚硬币上的题字，那么这场表演也就确定了。然而，对硬币不同

[1] 题目来源：CEOI 2001
[2] 题目来源：经典问题
[3] 题目来源：Polish Olympiad in Informatics

的设置也可能导致相同的表演，只要我们把硬币按适当的顺序排列。

先来看 3 枚硬币上的表演。如果表演是这样进行：从第一个硬币到第二个，第二个到第三个，第三个又回到第一个硬币（表示为 1→2，2→3，3→1）。具体表演是 3 只苍蝇绕圈跳跃，从不相遇。那么表演 1→2，2→3，3→3，则是与其不同的。因为该表演为：两拍后，所有的苍蝇在第 3 枚硬币上相遇，然后一直在原地跳跃。

但是，设置 1→2，2→3，3→2，4→4 和 1→1、2→3、3→2、4→3 就是同样的类型。如果把前者的硬币从左到右排列，而把后者从右到左排列，就会看到相同的表演。

如果跳舞蝇的两次表演是相同的，就会使观众不满，请编写一个程序，读入两组硬币设置，验证是否能把硬币按一定顺序排列，使跳舞蝇给出相同的表演。

【分析】

判断两个图是否同构并不容易，但本题涉及的图却十分特殊：每个顶点有且只有一条指出去的边。于是从其中任一点出发，沿有向边前进，最终形成一条带圈的有向路。从圈上的各点出发，逆着有向边向外扩展，可构成一个子图，这个子图包含了这个圈及所有能到达这个圈的点，为了描述方便，不妨称这为“弱连通子图”，如图 2-42 所示。

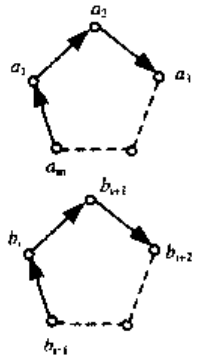


图 2-42　弱连通子图

这样整个图就可以看成是由一个或多个“弱连通子图”构成的，于是能否判断两个“弱连通子图”同构就成了本题的关键，不难归纳出它们同构的条件如下：

① 若子图 A 与子图 B 同构，则首先应当满足顶点总数相同和圈上的点数相同两个必要条件。

② 子图均以圈为中心向外扩展，当然要先确定它们两者在圈上的对应点，如图 2-42：$a_1 \to b_i$, $a_2 \to b_{i+1}$, $\cdots$, $a_{m-i+1} \to b_1$, $\cdots$, $a_m \to b_{i-1}$

③ 子图 A 要与子图 B 同构，必须要求圈上的任一对应点 a_p 和 b_q 在圈外的分支图中也要同构，而这一分支图一定是一棵倒向的多叉树。

综合以上三点，其根本在于判断两棵多叉树同构与否。

多叉树体现到一种层次的关系，为此我们可以用“(”和“)”即正反括号表示这种层次关系。

定义 $S[i]$表示与以 i 为根结点的多叉树的相对应的括号序列，有：

$$S[i]=\begin{cases} \text{“()”} & \text{（若}i\text{为叶子结点）} \\ \text{“(”}+S[k_1]+S[k_2]+\cdots+S[k_W]+\text{“)”} & \text{（}k_2,k_2,\cdots,k_W\text{为的子结点）} \end{cases}$$

为了能通过比较 $S[a_p]$和 $S[b_q]$直接判断两棵多叉树是否同构，需把任一结点的子树按从小到大的顺序排列。假定子树 Tree1 与子树 Tree2 分别是以 t_1、t_2 为根结点的多叉树，有：

若 Length($S[t_1]$)<Length(S[t_2])，则 Tree1>Tree2；

若 Length($S[t_1]$)>Length(S[t_2])，则 Tree1<Tree2；

若 Length($S[t_1]$)=Length(S[t_2])，则有三种情况：

① 若 $S[t_1]$=$S[t_2]$，则 Tree1=Tree2；

② 若 $S[t_1][1\cdots k-1]=S[t_2][1\cdots k-1]$，且 $S[t_1][k]=$“)”，$S[t_2][k]=$“(”，则 Tree1<Tree2；

③ 若 $S[t_1][1\cdots k-1]=S[t_2][1\cdots k-1]$，且 $S[t_1][k]=$“(”，$S[t_2][k]=$“)”，则 Tree1>Tree2；

这样处理后，若 $S[a_p]=S[b_q]$，则以 a_p 和 b_q 为根结点的两棵多叉树同构。

在具体实现中，由于 $n\leqslant 2\,000$，括号序列的长度最大能达到 4 000，字符串类型无法承受，编程时可用 0 和 1 代替。

【例题 4】参观洞穴[①]

Byteland 有很多洞穴，其中一个如图 2-43 所示。

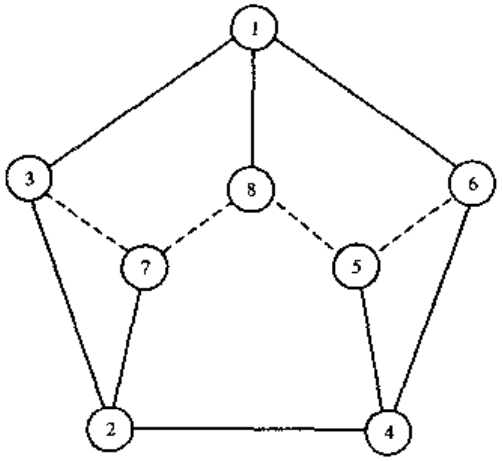


图 2-43　洞穴示例

该洞穴有一些有趣的特点：

- 所有的房间和通道在同一层且任意两条通道都不交叉。
- 其中一些房间组成一个圆圈的形状，这些房间称做外房间，其余房间都在外房间组成的圆圈里面，所有这些房间称做内房间。洞穴的惟一入口（同时也是惟一出口）直接通往其中一个外房间。
- 从每个房间出发，恰好有三条通道通往另外三个不同的房间。显然，从每个外房间出发的三条通道分别通往它在圆圈上两个相临的外房间和某个内房间。
- 外房间组成的圆圈上的通道称做外通道，所有其他通道称为内通道。
- 对于每个房间来说，我们都能找到一条通往任意一个其他房间的只经过内通道的路线，但是如果我们规定每个内通道只能经过一次的话，这样的路线是惟一的。
- 不是所有通道都好走。我们把通道分成两种：好走的通道和难走的通道。

Byteland 政府决定把这个有趣的洞穴改造成一个风景区供游客参观。为了方便管理，政府决定事先给游客规定一条参观线路，每个房间参观一次（除了路线起始和终止房间——它被参观了两次）为了让游客更加满意，预定的这条路线中，难走的通道数目要尽量少。

【分析】

如果忽略外通道，那么任意一对房间之间的简单通路是惟一的，因而以房间 1 的根内通道为边遍历整张图，结果将得到一棵树。又因为任意房间的度为 3，所以该树是棵二叉

[①] 题目来源：CEOI 1997

树，并且内房间均有 2 个儿子，外房间均为叶结点或树根。请看图 2-44，它是某个洞穴以 A 为根遍历得到的二叉树，树上每个非叶结点的左右儿子是严格有序的，这种严格有序使得 $A \rightarrow E \rightarrow I \rightarrow J \rightarrow K \rightarrow M \rightarrow N \rightarrow A$ 恰是外房间和外房间相邻通道组成的圆圈。

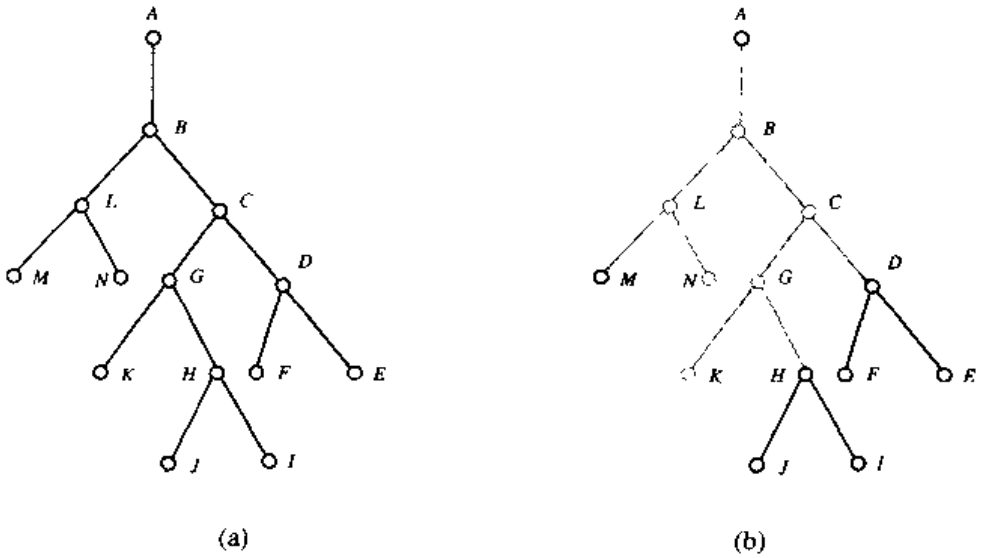


图 2-44　遍历洞穴得到的二叉树

如果参观洞穴的时候，通过外通道（A,E）由 $A \rightarrow E$，同时通过外通道（M,A）由 $M \rightarrow A$，那么辈分最高的内房间 B 应该如何遍历呢？很明显，如果选择 $E \rightarrow D \rightarrow C \rightarrow B \rightarrow L \rightarrow M$ 的道路去参观房间 B，那么 E、M 之间的外房间 I、J、K、N 都将被孤立而永远参观不到，因此该道路两头的外房间必须是相邻的，也即是一条开始于 K 结束于 N 不含外通道的道路。根据题中条件，这样的道路惟一，它必然是 $R = K \rightarrow G \rightarrow C \rightarrow B \rightarrow L \rightarrow N$。

将道路 R 与 R 相关联的边抹去，剩余房间就被分离成为不连通的多棵小二叉树，如图 2-44 所示，分别考虑这些小二叉树中辈分最高的内房间，确定参观这些内房间的路径的方法和 B 是一样的。于是得出算法——

大体思想：不断给边涂红色，当所涂边的数量为 N 的时候，一条参观路线就得到了。计算所有参观路线的困难值，从中取最优。

算法步骤：

（1）因为 E、M 是入口房间 A 的相邻房间，所以如果规定 E 先于 N 被参观的话，那么 E 一定是第 2 个被参观的外房间，而 M 一定是最后一个被参观的外房间。确定 $M \rightarrow A \rightarrow E$ 的参观路线，并将参观路线上的边涂成红色。

（2）选取尚未被红色边覆盖的辈分最高的内房间 I，如果 I 存在，那么转至步骤（3）；如果 I 不存在，那么转至步骤（4）。

（3）确定 I 被参观的惟一的路线 R，其中，R 的两头是一对相邻的外房间。将 R 上的边涂成红色，转至步骤（2）。

（4）如果相邻外房间 x，y 无法通过红色边连通，那么将外通道（x,y）涂成红色。

（5）输出参观路线。

复杂度分析：根据刚才的结论，一旦 $M\rightarrow A\rightarrow E$ 的参观路线确定了，那么剩余房间被参观的顺序也就被惟一确定了。如果所有房间已经按照辈分从高到底排序，那么确定剩余房间参观顺序的复杂度是线性的，它等于 O(边数)。因此，算法复杂度几乎完全取决于 $M\rightarrow A\rightarrow E$ 路径的条数。根据题中条件，这样的路径只有 3 条：

① 通过外通道由 $M\rightarrow A$，通过外通道由 $A\rightarrow E$；

② 通过外通道由 $M\rightarrow A$，通过内通道由 $A\rightarrow E$；

③ 通过内通道由 $M\rightarrow A$，通过外通道由 $A\rightarrow E$。

枚举这 3 条 $M\rightarrow A\rightarrow E$ 的路径，根据它们确定相应的剩余房间的参观路径和参观路径的困难值，从中取最优。总体算法复杂度仍然是 O(边数)，常数因子≈3×4=12。

但是不要忘了，预处理的时候，除需遍历构造整棵二叉树外，还要将所有房间按照辈分从高到底排序，所以总体算法复杂度应当为 $O(n\log n)$，常数因子大约为 30。

【例题 5】公主和英雄[①]

公主和英雄在玩一个游戏。公主画了一个有 n（$n\leqslant 10\ 000$）个顶点的凸多边形并按随机顺序为顶点编号 1、2、……，n，然后她告诉英雄所有边和一些不交叉对角线（多边形的顶点认为是不交叉点）的信息，但并不说明它们的顺序（边和对角线的信息就是它的两个端点编号），她要求英雄猜出多边形顶点的顺序。公主很善良，所以顶点顺序的答案存在且惟一。

例如，如果 n=4 且（1,3），(4,2)，(1,2)，(4,1)，(2,3）是四条边和一条对角线的顶点，那么这个多边形的顶点的顺序是 1，3，2，4（视旋转和翻转后重合的解为同一个解）。

【分析】

构造一个图 G，G 的顶点是多边形的顶点，G 的边是公主所给的边或者对角线。我们发现，去除任何一条边的两个顶点，都不会造成图连通性的破坏；去除任何一条对角线的两个顶点，都会造成图连通性的破坏，所以这个图只有一条 Hamilton 回路，而找到它也就找到了多边形顶点顺序。为方便叙述，我们称多边形第 I 个顶点编号为 A_i。

算法一

如果图中所有顶点的度均为 2，那么它们都是多边形的边，可以直接根据边确定顶点顺序。否则，察看某个度>2 的顶点的 3 条边，找到其中必然存在的一条对角线（x,y）。

去除顶点 x，y，图会被分成不连通的两个子图，通过遍历求出两子图分别包含的顶点集合 $S1$ 和 $S2$；根据|$S1$|或|$S2$|，就可以知道 x，y 在多边形上的距离。令 $A_1=x$，则相应的 $A_{1+|S1|}=y$。

当知道 $A_{1\sim k}$的值后，考察和 k 相连的图的边（k,u），求出 u 到 k 在只跨越一个集合时的距离 p（即是说如果 u、k 之间间隔了 p 个顶点，那么这 p 个顶点要么都属于集合 $S1$，要么都属于集合 $S2$），由于位置 1→k 已被其他编号的顶点占据，所以 u 的多边形位置只能是 $k+p$，即是令 $A_{k+p}=u$。

有一点优化：

[①] 题目来源：ACM/ICPC Regional Contest CERC 2001

就是当 u 惟一时，毫无疑问,(k,u)是边而不是对角线。

如果（k,u）是多边形的边，那么我们在令 $A_{k+1}=u$ 后就可以直接考察和 k+1 相关连的边，而忽略所有和 k 相关连的未被考察的边。

算法一复杂度为 $O(|V|\times|E|)=O(n^2)$，但由于常数因子较大，因而效率低下。

算法二

由于对角线不交叉，所以必有某个顶点 i 度为 2，易知，和 i 相连的两条边为多边形的边。如图 2-45(a)，假设哈密顿回路中顶点被遍历的顺序为序列… …A_{i-2}、A_{i-1}、A_i、A_{i+1}、A_{i+2}…，那么当我们将多边形的角 i 砍去后，剩余 n−1 个顶点、n−1 条边和 m−1（或者 m）条对角线构成的图一定可以惟一确定一条哈密顿回路… …A_{i-2}、A_{i-1}、A_{i+1}、A_{i+2}…。介于此，我们需要完成的工作就是不断将图 2-45(a)中度为 2 的顶点 i 以及相应边（i−1,i）、（i,i+1）移除，然后给残留图添置边（i−1,i+1）——如果边（i−1,i+1）不存在的话——使之成为新的多边形，最后将顶点 i 和（i−1,i）、（i,i+1）中属于原始多边形的边（例如图 2-45(a)中，实线表示原始多边形的边，虚线是后来添置的边）放入图 2-45(b)。

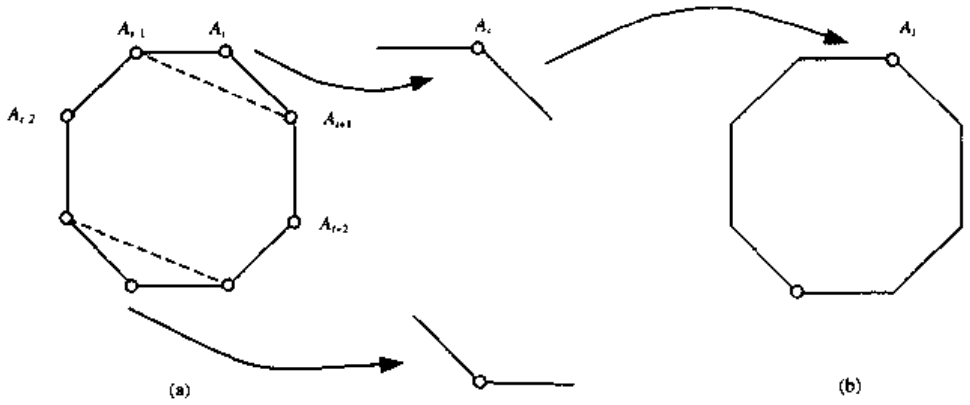


图 2-45　算法二示意图

找出当前度为 2 的顶点很容易，可是怎么判断移除的边是否是原始多边形的边呢？很简单。如果原始图的哈密顿回路路径如…A_x、A_{x+1}…A_{y-1}、A_y…，那么当且仅当 A_{x+1}…A_{y-1} 的顶点都被删除后，（A_x,A_y）才会成为属于新多边形而不属于原始多边形的边。判断顶点 A_x 和 A_y 在图 2-45(b)中是否连通，也就确定了（A_x,A_y）是否是原始多边形的边。

我们需要指针保存原始图的结构，需要最小堆找出度为 2 的顶点，还需要并查集帮助判断图 2-45(b)中任意两顶点是否连通。每次从图 2-45(a)中移除一个顶点到图 2-45(b)都要调用一次最小堆、两次并查集并遍历所有相关联的原始的或后来添置的边。所有原始或后来添置的边最多为 2n−3 条，所以算法二总体复杂度为 $O(n\log n)$。

算法三

直接用 DFS 遍历整张图，用寻找割顶的方法求出每个顶点 i 的 deep[i]值和 low[i]值。如果树上的某条边(u,v)（u 是父亲）是多边形的边，那么（u,v）的存在一定不会破坏 v 所有子孙的连通性，所以 v 不会含有数量>1 的儿子。反之，（u,v）一定是对角线。

考虑顶点 u 和它的儿子 v:

情况 1：v 没有子女。

见图 2-46，如果（u,v）是对角线，那么 x 不可能先于 u 被遍历到，所以 v 不可能无子女，因而（u,v）是多边形的边。

情况 2：v 有惟一的子女 w，且 u 是树根。易知，（u,v）是多边形的边。

情况 3：v 有惟一的子女 w，且 u 不是树根。如果（u,v）是对角线，那么（u,v）将 v 的所有子孙与 u 的所有祖先阻隔开来，所以 low[v] = deep[u]。反之，如果 low[v]<deep[u]，那么（u,v）是多边形的边。

由此，可鉴别出树上 n-1 连线分别是多边形的边还是对角线。这 n-1 条连线中最多有 $n/2$ 条是边，其余≥$n/2$ 条是对角线。把这些对角线从图中删除，然后重复刚才的过程，前后最多 2 次 DFS 就可以找出并删除所有的对角线。

算法三复杂度为 $O(n)$。

【例题 6】通讯员[①]

“Byteland”王国的国王最近很是烦恼：秘密情报机构通知他，“Hacker”帝国正准备进攻他的国度。甚至，“Hacker”帝国的秘密特工已经潜入了“Byteland”的某个城市中试图阻止“Byteland”的防卫准备。“Byteland”国王决定通知他的国民们准备同即将到来的侵略者做斗争。

“Byteland”的国民生活在城市中。城市的编号从 1 到 n，3≤n≤500。首都的编号是 1。“Byteland”王国有非常发达的交通网络，城市间的路都是双向的。当然每两个城市之间最多只有一条公路直接连接，但是对任意两个不同的城市 A 和 B，总存在两条 A 到 B 的路，使得除了起点和终点外两条路不经过任何相同的城市。

“Byteland”国王决定派两个信使通知他所有的国民。每个信使沿公路访问城市，警告其中的居民即将到来的威胁。信使当然可能访问某些城市很多次。但是，他们第一次访问某些城市的顺序必须由国王决定。

访问的顺序是一个数列 x_1，x_2，…，x_n。当然，x_1=1，表示信使是从首都出发的。在从城市 x_i 前进到城市 x_{i+1} 时，信使可以取道任何他先前经过的城市。不幸的是，“Hacker”帝国的特工在他们占领的那个城市中设置了陷阱，计划将前来的信使抓获和囚禁。因此，两个信使在访问城市的过程中可能会被抓住。国王希望，信使们在被抓住之前能够将所有的城市至少通知一遍。

请你编写一个程序帮助国王确定一个城市访问的顺序 x_1，x_2，…，x_n。信使们必须按照这个顺序来通知沿途上的居民。并且保证，按照这个顺序前行，在信使被抓住之前，每个城市都会被通知到。

一个例子：

如图 2-47 所示，第一个信使的移动路线可以是：1 2 3 2 5 4，第二个信使的路线是：1 4 5 4 3 2。

---

①题目来源：Polish Olympiad in Informatics

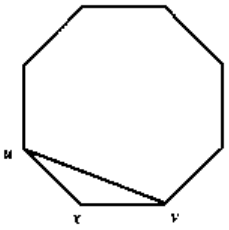


图 2-46　算法示意图

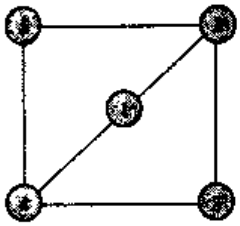


图 2-47　Byteland 的地图

这样，不管特工在什么位置，通讯员被抓住之前每个城市至少被一个通讯员访问过。

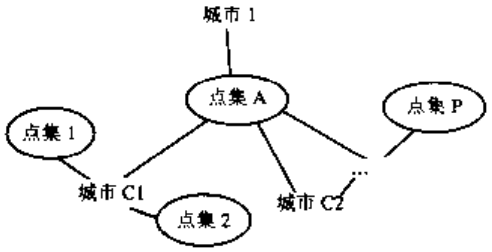
【分析】

两人都是从 1 出发的，因此只需考虑 2 到 n 这 $n-1$ 个城市的访问顺序。我们可以发现两个人访问城市的序列一定是反序的，否则一定可以找到一个情况使得两人都无法达到某一个城市，关于这一点是很容易证明的：若两个访问序列 A_1，A_2，…，A_{n-1}，B_1，B_2，…，B_{n-1}，反序，我们可以得到 $A_i=B_{n-i}$，反之我们一定能够找到一个最小的 i 使得 $A_i\neq B_{n-i}$，当敌人处于第 A_i 个城市时，两人都无法访问到第 B_{n-i} 个城市。这样就证明了这个问题。

那么我们的任务就是找到两个相反的访问序列使得存在合法的路径。只需找到第一个人的访问序列即可推出第二个人的访问序列。那么第一个人的访问序列又要满足什么条件呢？

进一步分析可知，如果每次将第一个人访问过的城市从图中删去（城市 1 不删），且每次都能够保证剩下的点仍然是连通图，则我们就找到了这样一个访问序列。并且可以证明我们扩展访问序列时一定可以找到一个满足要求的点。

不妨设已删去的点集为 A（不含城市 1），惟一会造成找不到满足要求的点的可能情况如图 2-48 所示。



(图中省略了与城市 1 相连的一些边)

图 2-48　点集划分与连通情况示意图

否则如果存在点集 A 能够到达一个非割点则就已找到符合条件的点。显然城市 1 与点集 1、点集 P 不会同时有边相连，这样城市 C_1、城市 C_2、…都不会是割点，并且由于原图中任两个城市间都有不相交的两条路径，因此点集 1 与点集 P 中至少有一个为空集，那么上图只可能为图 2-49 所示的两种形式。

那我们只需删去城市 C_1 即可。因此每次扩展都能找到满足要求的点。这样问题就得到解决。

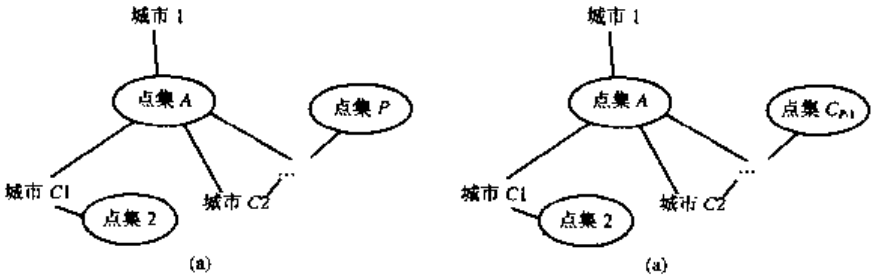


图 2-49　两种情况

【例题 7】幼儿园小朋友分组[①]

幼儿园里有 n 个小孩。不幸的是小孩常常打架。每个小孩有不超过 3 个仇敌。是否有可能将所有小孩分成两组使得每个小孩最多和他的一个仇敌同组？

【分析】

这道题目可以看作是给所有的点黑白二染色。

现给 1 染黑色，放入队列。每次从队列中取出一个点，把与其相邻的还未染色的点燃成相反的颜色（因为不同的连通分量之间互不相干，可以假设所有点构成一个连通图），这样一次广搜就能给所有的点一个初始色。

然后如果发现某个点 x 有两个以上的仇敌与它染同样的颜色，就把 x 的颜色变一下。如此反复就能求得答案。

现在有两个问题：

① 这样变是不是总能得到答案？

② 这样做的时间复杂度是多少？

有一个重要的结论：**一个点颜色至多变一下**。也就是说不会出现某个点的颜色反复变的情况。假如这个结论成立，则上面提出的两个问题都能解决：① 由于不会反复变，在有限步内（至多 n 步）就不能继续变了，这时的状态就是答案。② 每个点至多变一次，所以时间复杂度是 $O(n)$。

下面，我们证明这个结论。

证明：我们用广搜给每个点初始色，于是形成了一颗搜索树，深度为奇数的点染黑色、深度为偶数的点染白色。如果一个点 x 存在两个以上的仇敌和它染同样的颜色，我们称 x 是一个“坏点”。

任意一个非叶子结点显然不是一个坏点。首先是根结点 R，由于我们采用的是广搜，所有与 R 有仇的点都被染成了白色（R 是黑色），所以 R 必然不是坏点。对于任意一个非根、非叶子结点 p，都存在一个父亲和至少一个孩子，所以 p 就至少有两个仇敌与它染不同颜色；而题目中说每个人至多 3 个仇敌，故而与 p 染同色的 p 的仇敌至多一个，因此 p 也不是坏点。

所以坏点必然是叶子结点！

---

① 题目来源：Ural State University Problem Archive

设某个坏点 x，与它染同色的两个仇敌是 a，b（显然 a,b 都不是根结点）。我们把 x 反色，就相当于令 x 成为 a 的孩子。我们要证明的就是 x 的颜色不可能再次改变。

x 改色的必要条件是 a，b 中至少有一个改色。由于 x 成为了 a 的孩子，a 不是叶子结点，因此 a 不可能在 x 之前改色，所以必然是 b 改色。设 b 的父亲是 t，那么 t 与 x 这两个 b 的仇敌都和 b 染不同的颜色，要想使得 b 改色，必须先把 t，x 两者至少一个改色。而 t 不是叶子结点，所以必须把 x 改色才有可能让 b 改色。

归纳一下就是：x 改色前 b 要改色；b 改色前 x 要改色。这显然是一个不可能的任务。于是 x 的颜色不会被第二次改变（具体地说 a，b 的颜色也不会改变）。

具体实现的时候把所有的坏点放在一个栈里面，每次取一个点如果还是坏点就反色，同时考察所有与之相邻的点，看看是不是变成了坏点，如果是就放入栈中。总的时间复杂度是 $O(n)$。

WEB 本书的第一版介绍了弦图、区间图等内容，由于篇幅比较长而且题目缺乏，后来删除了。有兴趣的读者可以在本书主页中找到这一部分。

练　习　题

编程题：

2.4.7　n 维城市[①]

在 n 维的空间中最多有 n+1 个顶点彼此两两等距。n 维生物在 n 维国家特里斯曼中建造了两两相距 dis 的 n+1 座城市。城市道路修建规则如下：

① 任两座城市都通过道路相连。

② 每条道路仅连接两座不同的城市。

③ 任两座城市间不会有超过一条以上的直达道路。

④ 当且仅当城市 A 和 B 间存在直达道路时，称它们是相邻的。如果城市 A 与城市 B，C 相邻，那么城市 B，C 一定不相邻。

⑤ 间的道路不是直线型的就是弧线型的。弧线型的道路是半径为 d 的一段圆弧，这里 $d \geq 0.5\times$ 两城市的直线距离。最多有 50 %的道路是弧线型的。

⑥ 可以假设任两条道路不相交。

特里斯曼的交通部部长打算修建尽可能多的道路，在此基础上使道路总长度最大。现给出 n、dis、d 的值，请你告诉部长道路的最大数量和最大总长度。

2.4.8　远程通信[②]

巴罗的海上有两个小岛：Bornholm 和 Gotland。在每个小岛上都有一些神奇的远程通信端口，每个端口可以运行在两种模式下，即发送模式和接收模式。Bornholm 和 Gotland

[①] 题目来源：UVA Problem Archive

[②] 题目来源：Baltic Olympiad in Informatics, 2001

分别有 n 个和 m 个这样的端口，每个端口都连接着另一个小岛上的一个端口，称为“目标端口”，如图 2-50 所示。

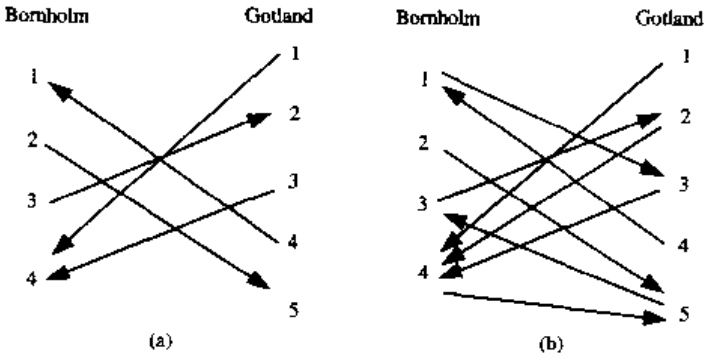


图 2-50　端口设置示意图

请设置这 $n+m$ 个端口的模式，使得所有端口都处在工作状态，即对于接受模式的端口 A，另一个小岛上至少有一个以 A 为目标端口的端口被设置成发送模式；对于发送模式的端口 B，它的目标端口一定处在接收模式中。

2.4.9　旅行路线[①]

有 N（$1 \leqslant N \leqslant 200$）座编号从 1 到 N 的城市以及 M 条双向行驶的道路。任两座城市间最多只有 1 条道路相连。暑假里 DSAP 的成员打算制定若干条旅行路线。每条路线通过 K（$K>2$）座不同的城市 T_1、T_2、…、T_K 后返回 T_1。你的任务就是帮助制定 T 条这样的旅行路线，使得任一条路线至少访问了一座其他路线都没有访问到的城市。这里的 T 要尽可能大。

2.4.10　龙穴迷宫[②]

神话中的 Byte 山上有一座迷宫。迷宫的入口坐落在山顶。迷宫由红色、绿色和蓝色的房间组成。两间相同颜色的房间看上去完全一样无法区别。每个房间内有编号为 1、2、3 的三堵墙，要从一房间到另一房间，只有穿过上面房间的一堵墙到下面的房间（不一定垂直地穿过）。从入口处的房间可以到达其他任一房间，所有的通道最终都通向最底层的龙穴，如图 2-51 所示。通过迷宫的一次旅行可用一系列在房间内选择要穿过的墙的编号来表示。这样的一个序列，称为一个行程计划。

Byte 龙就住在龙穴里。传说，如果谁能画出这个迷宫的地图，并献给龙，谁就将得到一大笔财富；但如果失败了，就会被 Byte 龙一脚踢下山。

一个叫 Bytezar 的英雄历尽艰辛终于画出了一份地图，然而 Byte 龙却说，虽然所有的房间都标在地图上，但其中不少房间出现了不止一次。

我也画了一张类似的地图。但我发现：尽管我画出的房间数较少，但旅行者按任一行程计划穿过迷宫，途中依次见到的房间和原来一样。于是我决定尽可能减少图上的房间。

请编写一个程序，读入 Bytezar 的地图，计算出 Byte 迷宫的真实房间数。房间总数不超过 6 000。

[①] 题目来源：Ural State University Problem Archive

[②] 题目来源：Polish Olympiad in Informatics

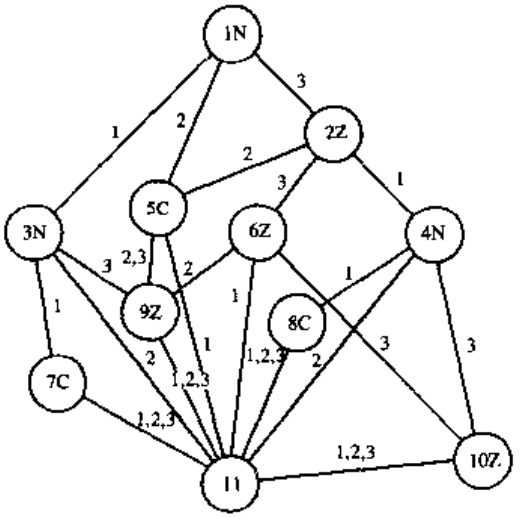


图 2-51　龙穴迷宫

2.4.11　追捕游戏[①]

追击是一个两人棋盘游戏。棋盘是由编号为 1~n（$n \leqslant 3\,000$）的方格组成的。对于每一对方格，我们知道它们是否相邻。每一个玩家拥有一个受他支配的棋子。游戏开始时，棋子都是放在固定的不同的方格中。在一个回合中，玩家可以把他的棋子放在原地不动或者把它移到相邻的格子中。

一个游戏棋盘具有如下的性质：

- 其中没有三角形。（不存在不同的三个格子，满足两两相邻）
- 任何一个格子都可以被每个玩家移动到。

一场游戏中有许多回合。在每一个回合中，每个玩家做一次简单的移动。每一轮都由玩家 *A* 开始。如果两个棋子到了同一格，我们就称玩家 *A* 被玩家 *B* 抓住了。对于一个给定的棋子的初始位置，判断是否无论 *A* 怎么走，*B* 都一定能抓住 *A*。如果是，那么在两个人都采取最佳策略的情况下，*B* 需要多少步才能抓住 *A* 呢？

图 2-52 描述了一个可能的棋盘。

考虑图 2-52 中的棋盘，相邻的方格都被连上了边。如果，开始的时候玩家 *A* 和 *B* 的棋子分别放在 9 和 4 上，那么玩家 *B* 可以在第三回合抓住玩家 *A*（他们都以最佳策略来玩）；如果，开始的时候玩家 *A* 和 *B* 的棋子分别放在 8 和 4 上，那么玩家 *B* 无法抓住玩家 *A*（如果玩家 *A* 操作正确）。

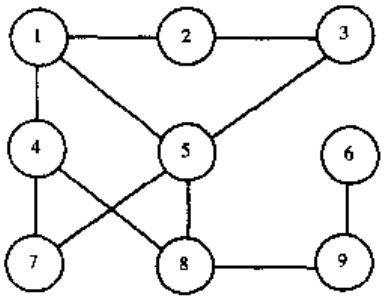


图 2-52　一个可能的游戏棋盘

写一个程序，读入对于棋盘和棋子初始位置的描述，判断玩家 *B* 是否能抓住玩家 *A*。如果是的话，计算他需要多少回合（假设他们都用最佳策略来玩）。

[①] 题目来源：Polish Olympiad in Informatics

2.5　图论基本算法

这是本章难度最大、最灵活的一小节，数据结构、算法设计、图论知识和数学模型方法融为一体，构成了本节的主要内容。本节的例题比较多，启发性很强，读者不妨多读几遍，慢慢体会其中的内容。

2.5.1　生成树问题

生成树问题是图论最基本的问题之一。这里，我们讨论几种与生成树有关的问题。

1．岛国——最小生成树

岛　　国[1]

航海家们在太平洋上发现了几座新岛屿，其中最大的一个岛（称为主岛）已经连接到Internet，但是其他岛和主岛之间没有电缆连接，所以无法上网。我们的目的是让所有岛上的居民都能上网，即每个岛和主岛之间都有直接或间接的电缆连接。

图 2-53 就是这样一个岛屿，每个岛的居民数目标注在了旁边。每条实线表示一根电缆，它的长度等于两个岛屿中心位置的几何距离。为了节省成本，希望所有电缆的总长度尽量小。应该怎样连接呢？

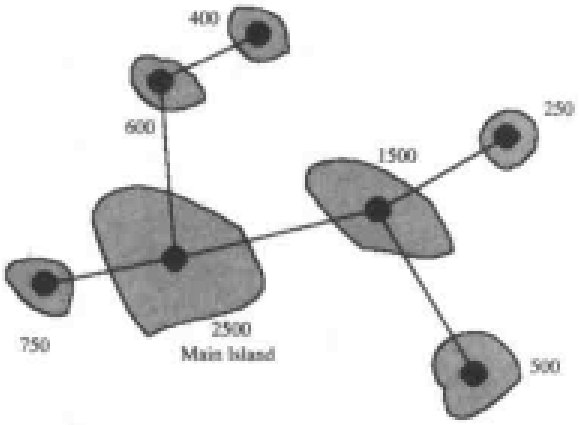


图 2-53　岛国

【分析】

要使总长度尽量小，显然得到的图是不含圈的。否则在圈上任意去掉一条边，所有点仍然连通。即本题是要求一棵权值最小生成树（minimal spanning tree）。由于树不含圈，且边数为 $n-1$，所以我们实际上是不停地往答案中添加边，只要避免圈的生成，那么在添

[1] 题目来源：ACM/ICPC World Finals 2002

加 n 条边以后就得到了一棵树。问题是，怎样添加边呢？有两个不同的算法。一个是逐步扩展的，另一个是逐步合并的。

Prim 算法思想　任意时刻的中间结果都是一棵树。从一个点开始，每次都花最少的代价，用一条边把一个不在树里的点加进来。为了方便选则离树最近的点，需要借助堆。每次选边的复杂度为 $O(\log m)$，所以总复杂度为 $O(m+n\log m)$。

Kruskal 算法思想　任意时刻的中间结果是一个森林。初始是 n 个点的集合，每次选权最小且不产生圈的边加进来，合并两个森林。为了方便检测环和加边，需要借助并查集。每次加边的复杂度为 $O(\alpha(n,m))$，所以总复杂度为 $O(m\log m+n\alpha(n,m))$

以上两个算法都是贪心的，也是正确的，这里不作证明。有兴趣的同学可以自己试一试，它们并不困难。聪明的读者一定已经看出：Kruskal 算法实际上就是矩阵胚系统的贪心算法。从这两个算法的流程可以看出，如果所有的权都不相等，则最小生成树是惟一的。

***K 小生成树**　这里我们简单介绍 k 小生成树的求法。首先我们给出可行交换、树的邻树和集合的邻树三个定义：

定义：生成树 T 删除一条旧边 f 并加入一条新边 e 的操作叫做交换。如果交换后得到的图仍是一棵树，则此交换为**可行交换**，记为 $(+e,-f)$。生成树 T 和 T'，如果它们是同一个图 G 的两个不同的生成树，且其中一棵是由另一棵进行一次可行交换后得到的，则它们互为邻树。对于生成树 T 和生成树集合 S，如果 T 不是 S 中的树，且 T 是 S 中某棵树 T' 的邻树，则称 T 是生成树集合 S 的邻树。

我们不加证明的给出以下定理。

定理：设 T_1，T_2，…，T_k 是 G 的前 k 个最小生成树，则生成树集合 $\{T_1,T_2,\cdots,T_k\}$ 的邻树中边权和最小的一个树 T 可作为第 $k+1$ 小生成树。

请注意，我们说 T "可作为" 第 $k+1$ 小生成树，即在我们的定理中，第 k 小生成树的选择并不是惟一的，即使有两棵树的权和一样大，它们也被赋予了不同的序号。

从这个定理出发，无法直接得到一个高效的算法。事实上，需要利用边的收缩方法并借助堆才能得到满意的时间复杂度，这里略去。

2. 野餐计划——最小度限制生成树

野 餐 计 划[①]

矮人虽小却喜欢乘坐巨大的轿车，轿车大到可以装下任意多个矮人。某天，N 个矮人打算到野外聚餐。为了集中到聚餐地点，矮人 A 要么开车到矮人 B 家中，留下自己的轿车在矮人 B 家，然后乘坐 B 的轿车同行；要么直接开车到聚餐地点，并将车停放在聚餐地。

虽然矮人的家很大，可以停放无数辆轿车，但是聚餐地点却最多只能停放 K 辆轿车。现在给你一张加权无向图，它描述了 N 个矮人的家和聚餐地点，要你求出所有矮人开车的最短总路程。

[①] 题目来源：ACM/ICPC Regional Contest ECNA 2000

【分析】

为了方便叙述，把顶点 V_0 的度数≤K 称做**度限制条件**，把满足这一条件的生成树称为**度限制生成树**，把权值和最小的度限制生成树称为**最小度限制生成树**。

如果撇开度限制条件，那么这道题就是要求我们熟知的最小生成树。但是加入了这个条件，问题就变得不那么简单，如何解决这一限定条件将成为解决本题的关键。

首先，还是让我们从熟悉的模型入手，避开度数的限制。假如把最小度限制生成树中 V_0 所连接的边都删掉，剩下的边构成顶点 $V_1\cdots V_n$ 的一个森林，而这个森林中任意点的度数都没有限制。那么不妨从这里开始入手：森林中最简单的情况是只有一棵树（假如不存在一棵树处理的方法将是类似的，因此可以假定去掉 V_0 后图仍是连通的），很容易证明此时这棵树肯定是顶点 $V_1\cdots V_n$ 的最小生成树。我们不妨先求出 $V_1\cdots V_n$ 的最小生成树看看能否对**最小度限制生成树**的求解带来某些有用的信息。

通过分析论证，可以得出如下定理。

定理 1：若 T 为 $V_1\cdots V_n$ 的某棵最小生成树，边（i，j）$\{i,\ j>0\}$不属于 T，$E-\{(i,\ j)\}$的最小度限制生成树也是 E 的最小度限制生成树。

证明：

设 E 的最小度限制生成树为 D，如果边（i，j）不属于 T，则在 T 中，顶点 i，j 之间肯定通过一系列的边（x_1，x_2）…（x_k，x_{k+1}）…（x_{c-1}，x_c），$x_1=i$，$x_c=j$ 连接，则 $\text{cost}(x_k,\ x_{k+1})\leqslant\text{cost}(i,\ j)$[1]，不然与 T 是最小生成树相矛盾。若（i，j）又在 D 中，那么（i，j）是 D 的一个桥，将 D 分为两个连通分量 A，B，$i\in A$，$j\in B$，分两种情况：

① $x_2\cdots x_{c-1}$ 同属一个连通分量，不妨设为 A，将（i，j）从 D 中删掉、将（x_{c-1}，j）补上得到 D'，显然 D' 仍然为 $V_0\cdots V_n$ 度限制生成树，且 $\text{cost}(D')-\text{cost}(D)=\text{cost}(x_{c-1},\ x_c)-\text{cost}(i,\ j)\leqslant 0$；

② $x_2\cdots x_{c-1}$ 分属两个连通分量 A、B，则一定存在 x_a 属于 A，x_b 属于 B，$b=a+1$[2]，将（i，j）从 D 中删掉、将（x_a，x_b）补上得到 D'，同样得到 $\text{cost}(D')-\text{cost}(D)\leqslant 0$。

总能够得到某棵 $V_0\cdots V_n$ 度限制生成树，不包含（i，j），并且总长不比 D 多。同时，由于 D 为 E 的最小度限制生成树，所以 $\text{cost}(D)\leqslant\text{cost}(D')$，故 $\text{cost}(D)=\text{cost}(D')$，结论成立。

这一结论显然能带来巨大的价值，如果我们求出某个 T，然后就可把所有不在 T 上的边（i，j）$\{i,\ j>0\}$删掉（不断地利用上述定理，将这些边一条一条删掉）得到 E'。问题就转化成了求 E'的最小生成树。这样一来光从数目上看$|E'|\leqslant 2n$，降低了一个级别，何况这些边都是很有规律的。

现在回到原问题上来，求 V_0 度数不超过 K 的最小生成树，不妨令 H_i 为 V_0 的度数为 i 的最小生成树。那么问题就是求 $\min\{H_i,\ 1\leqslant i\leqslant K\}$，显然 $H_1=T+\{(0,\ x)\}$，（0，x）是所有与 0 关联的边中权值最小的。H_i 是否可根据 H_{i-1} 得到呢？

一种贪心的思路是：在 H_{i-1} 上加入一条边（0，x），得到一个环，然后再删掉环中一条不与 0 关联的边，设为（x_a，x_b），这样就能得到一棵 V_0 的度数为 i 的生成树，这样得到的生成树的权值和为 $\text{cost}(H_{i-1})+\text{cost}(0,\ x)-\text{cost}(x_a,\ x_b)$，为了使权值最小，我们可以枚举

---

[1] cost(X)，为边集 X 的权值之和。

[2] 不妨设 $x_2\in A$，总有一个 $x_i\in B$，设 i 最小为 b 时，$x_i\in B$，此时 $x_{i-1}\in A$。

x 的值，删除时选取一条权值最大的。然而，这样得到的是否是一棵 V_0 的度数为 i 的最小生成树呢？

答案是肯定的，下面将给出证明，不过之前将给出几个简单的定义以使证明更加简洁。

- 添删操作：在树上添加某条与 V_0 关联的边，在构成的环中删掉某条不与 V_0 关联的边；
- 最大添删操作：删掉的边是可删除边中权值最大的“删除操作”；
- 差额最小添删操作：在所有可以进行的“添删操作”中能使添加的边的权值减去利用“最大添删操作”删掉的边的权值最小的一个“最大添删操作”。[1]

要证明的结论就是：

定理 2：在 H_1 的基础上，进行 i-1 次“差额最小添删操作” b_1，b_2，…，b_{i-1} 得到一棵 V_0 的度数为 i 的生成树，称为 H_i'，满足 $\text{cost}(H_i')=\text{cost}(H_i)$。

若 S 为一棵树，$a=(P,Q)$[2] 为 S 的一个“添删操作”，则定义 $S(a)$为 S 通过操作 A 得到的新树，一系列操作 $S(a_1)(a_2)\cdots(a_m)$简写成 $S(a_1a_2\cdots a_m)$。给每棵树 D 定义一个“下步操作” $\text{next}(D)$：若某棵树 $D=H_1(b_1b_2\cdots b_{k-1})$则设 $\text{next}(D)=b_k$，否则 $\text{next}(D)$为 D 的任意一个“差额最小删除”。

下面的 3 个结论是显然的，所以证明过程略去。

- 由于添加的边肯定不在 T 中，而删除的边只能在 T 中，所以每次添删操作都会处理不同的边，并且每条边只会被操作一次。
- H_i 肯定可以由 H_1 通过若干次“添删操作”得到，不妨设 $H_i=H_1(c_1c_2\cdots c_{i-1})$。
- $\{c_h\}=\{b_h\}$ ⇔ 所有 h 都有 $c_h=\text{next}(H_1(c_1c_2\cdots c_{h-1}))$。

限于篇幅，定理 2 的证明略去。

定理 1 和 2 可以很容易地推广到$\{V_1\cdots V_n\}$不存在最小生成树的情况。

其中，**定理 2 的推广形式**为：设$\{V_1\cdots V_n\}$有 t 个连通块 $R_1\cdots R_t$，每个块的最小生成树为 $T_1\cdots T_t$，给每棵树选一条与 V_0 相连的费用最小的边，构成一棵 V_0 的度数为 t 的最小生成树 H_m，在进行 i-t 次“差额最小添删操作” b_1，b_2，…，b_{i-t}后得到的一棵 V_0 的度数为 i 的生成树，称为 H_i'，满足 $\text{cost}(H_i')=\text{cost}(H_i)$。

利用定理 2 的推广形式，不难写出下面的主算法流程。

（1）找$\{V_1\cdots V_n\}$所有的连通块，求出每个连通块的最小生成树 T_i。

时间复杂度为 $O(n\log n+m)$（Prim+堆）。

（2）在每个块中，选择 V_0 相邻的最小边，得到 H_t。Min←H_t，V←$\text{cost}(H_t)$。

时间复杂度为 $O(n)$。

（3）循环 i←t+1 to k do

在 H_{i-1} 上选择“差额最小添删操作”，添加并删除一条边得到 H_i，令 V<V+cost(添边)-cost(删边)，若 V<Min，则令 Min←V；如果找不到“差额最小添删操作”则 Break[3]。

---

[1] 注意：可能有多个“最大添删操作”的差额（即添加边的权值减去删掉边的权值）相同，所以待会将证明的是：每次任意选择一个差额最小的都将达到最优解

[2] P 表示添加的边，Q 表示删除的边

[3] 实际上，可以证明当 V≥Min 时就可以 Break 了

时间复杂度为 $O(k*$查找“差额最小添删操作”的时间复杂度$)$。

（4）输出 Min。

易知：算法的瓶颈在第（3）步，若用枚举的方法查找“差额最小添删操作”的复杂度为 $O(n^2)$，显然高了，但怎么样才能降低呢？

设 Max_i 为 V_i 到 V_0 的路径上不与 V_0 相连的权值最大的边的费用。那么可以在查找时将复杂度降为 $O(n)$，分为两步：

（1）枚举一条未选边(V_0,V_i)，找到 $cost((0,i))-Max_i$ 的最小值，这样就找到了应该添加的边；

（2）枚举 V_iV_0 原有路径上的一条不与 V_0 相连的边，找到一条权值最大的。

Max_i 的初始值可在 $O(n)$的时间内求得，而在进行了一次添删操作后，也只需花上 $O(n)$的时间复杂度来进行维护——若添加的那条边为（0,i），则从 i 开始遍历$\{V_1\cdots V_n\}$即可完成维护。

这样一来，第(3)步的复杂度就将降为 $O(kn)$，故总的复杂度就可降为 $O(n(k+\log n)+m)$，空间复杂度为 $n+m$。

3. 地震——最优比率生成树

地　震[①]

可怜的农夫 John 被地震毁坏了他的整个农场，于是 John 决定重建它。修建好了所有的 $N(1\leqslant N\leqslant 400)$块田地之后，John 意识到应该修一些连接农场的路，以便他沿着路从任意一块田走到任意的另一块。熟悉了地形以后，John 发现只有 $M(1\leqslant M\leqslant 10\ 000)$条双向道路可以被修建。由于手头紧，他希望不要用太多的钱完成这项任务。

于是，John 想到了专门从事修路工作的专业奶牛工程队。John 同意为工程队支付 F 美元，而工程队希望能尽量快的获取利润。

工程队拿到了一份表格，上面记录着所有可以修的的路和它们修复的时间 t 及修建的成本 c，而且可能有多条路连接相同的两块田地。

编程求出工程队的最快赚钱速度，即让总利润/总时间最大。F，t，c 都不超过 10^9。对于每个测试数据，保证存在可行的修路方案，虽然这个方案可能使工程队赚不到钱。

【分析】

这道题目是要求一棵生成树，但不是希望边权之和最大，所以不能直接用最小生成树算法。但是我们希望能把问题转化为这个熟知的问题，因此需要把问题作一点转化。

设 $x_1,x_2,\cdots,x_m$ 在集合$\{0,1\}$中取值，$x_i=1$ 当且仅当边 i 在生成树中出现，则我们希望

$$r=\frac{F-c_1x_1-c_2x_2-\cdots-c_mx_m}{t_1x_1+t_2x_2+\cdots+t_mx_m}\text{ 最大（其中 }t\text{ 为正数）}$$

为了让 r 最大，先设计一个子问题 $Q(l)$，即让

$$z=F-c_1x_1-\cdots-c_mx_m-l(t_1x_1+\cdots+t_mx_m)=F+d_1x_1+\cdots+d_mx_m\text{ 最大（}d_i=l\times t_i-c_i\text{）。}$$

把这个最大值记作 $z(l)$。对于一个确定的 l，只要求出以 $d_i=l\times t_i-c_i$ 为边权的最大生成树

[①] 题目来源：USACO US Open 2001

就可以算出 $z(l)$。

显然原问题的最优解 r^*满足 $z(r^*)=0$。因为 t_i是正数，所以 1 减小到 l'时，把 $z(l)$的最优解向量代入 $z(l')$的表达式中可以得到一个更大的 z 值，即 $z(l)$严格单调递减（它还是分段线性的和凹的，这里我们没有用到）。因此我们有：

- $z(r)>0$ 当且仅当 $r<r^*$
- $z(r)=0$ 当且仅当 $r=r^*$
- $z(r)<0$ 当且仅当 $r>r^*$

容易知道：当所有边都被修建的时候 r 最小，设它为 A；当分母最小（以 t 为边权求的最小生成树权和）且分子为 F 时 r 最大，设它为 B。只需要在区间$[A,B]$中使用二分法，就可以在 $O(\mathrm{MST}(n,m)\times\log(B-A))$时间内解决这个问题。其中 $\mathrm{MST}(n,m)$是求解 n 个结点 m 条边的最小生成树问题的时间复杂度。

虽然并不是严格的多项式算法(和边权的范围有关)，但对于题目给出的数据范围，这个算法已经可以在很短的时间求出答案了。

这道题目的模型是 0-1 分数规划（0-1 fractional programming）。这里我们将刚才用的方法称为“参数搜索”（parametric search）。参数搜索的形式除了这里介绍的二分法之外还有其他方法，例如 Dinkelbach 算法：每次直接把子问题 $Q(l)$的最优解代入 r 的表达式算出一个新的 l'，进行下一次迭代。可以证明（证明很复杂，这里略去），在本题中该算法最多迭代 $\min\{n^2,\log(nM)\}$次，其中 M 是所有系数的最大值。对于平面图，由于最小生成树可以在 $O(n)$内求出，所以本题采用的二分法可以在 $O(n\log n)$时间内求出解。事实上还有线性时间复杂度的算法，有兴趣的读者可以参考相关论文。

2.5.2　最短路问题

最短路问题是图论中的核心问题之一，下面讨论几个基本算法。

1．罗密欧与朱丽叶的故事——dijkstra算法

罗密欧与朱丽叶[①]

罗密欧和朱丽叶是一对恋人，但是他们的父母却极力想破坏他们，甚至不允许他们见面。幸好，朱丽叶每周日的下午要到教堂去参加唱诗班，所以她每次都盼望在路上能碰到罗密欧，因为他每周日下午都要去一个附近的体育馆踢足球。朱丽叶每次都是下午 4 点准时离开家，沿着某一条去教堂最近的路；而罗密欧也是下午 4 点准时离开家，沿着一条去体育馆最近的路。假设他们俩的行走的速度一样且恒定。

日子一天天过去了，朱丽叶从来都没有碰到过罗密欧，她开始怀疑她的盼望是否根本不可能实现。你能告诉她吗？

① 题目来源：Internet Problem Solving Contest 1999

对于刚开始学习图论的读者来说，处境比朱丽叶的还要糟糕，因为现在还不会找出最短路，所以需要先学习单源最短路问题（Single Source Shortest Path problem, SSSP）。单源最短路问题的形式化描述是：

给出有向带权图 $G=(V, E)$，求某顶点 u 到其他所有点的最短路（即权和最小的路）。

首先我们考虑一种简单的情况：所有权非负。我们来看图 2-54。

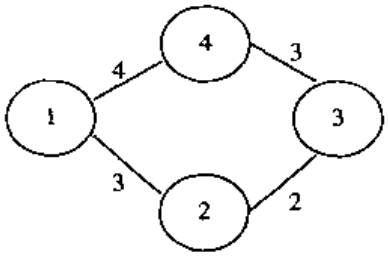


图 2-54　非负权无向网络

假设点 1 到点 u 的距离是 d_u。显然有 $d_1=0$，我们只需要求出其他点的 d 值就可以了。下面哪个点比较好求呢？是点 2。$d_2=w(1,2)=3$。为什么这么说呢？因为从 1 直接到 2 最近，如果先到其他地方再回到点 2 的话，第一步的距离就超过 d_2 了。由于权非负，所以路的总长一定超过 $w(1,2)$。

不知读者注意到没有，最短路也具有 1.5 节提到的“最优子结构性质”。如果 u 到 v 的最短路是 $u\to s_1\to s_2\to\cdots\to s_k\to v$，那么子路径 $u\to s_1\to s_2\to\cdots\to s_k$ 一定是一条最短路。否则用一条更短的路替换它，u 到 v 的路也不是最短路。所以，在以后的计算中我们每次都是“选一个已经求出最短路的点，加上一条边求出到另一个点的最短路”。

假设已经求出了点集 S 中每个点的最短路长度，那么从任意一个点 u 出发加一条边 (u,v) 都可以得到一个新的“最短距离” $d'v=d_u+w(u,v)$。但只有其中最小的那个距离 $\min\{d'v\}$ 才保证是真正的最短距离 d_v，因为从其他点“绕”回该点的路径一定更长（回忆我们是怎么求出 d_2 的）。读者可能已经注意到了，这个算法和 Prim 算法相似，它也是不断扩展一棵树，只是 Prim 每次选一个权最小的边来加点，本算法是选一个 d' 值最小的点 u 加进来并让 $d_u=d'_u$。和 Prim 算法类似，也需要借助堆来求最小的 d' 值。

用这个方法，可以进一步求出 $d_3=d_2+w(2,3)=5$，$d_4=w(1,4)=4$。在这里，我们充分的利用了“权非负”这一有利条件，按照 d' 值递增的顺序求出了需要的答案。这个算法就是著名的 dijkstra 算法。

Dijkstra 算法： 令 $d'(v)=\min\{d[u]+w(u,v) \mid u$ 的最短距离已求出且边 (u,v) 存在$\}$，设 d' 最小的元素为 i，则每次令 $d_i=d'_i$（即求出了 i 的最短路长度）。每次求出一个 d 值，n 次以后就可以得到所有点的距离。算法复杂度为 $O((m+n)\log m)$。

现在，我们回过头再看看刚才的问题。把路抽象成图的无向边，把路与路的交点抽象成图的顶点，用 D_{ij} 表示顶点 i 与顶点 j 之间直接相连的路的长度（如果有多条，去其中最短的）。假设朱丽叶的家在顶点 S_1，教堂在顶点 T_1，罗密欧的家在顶点 S_2，体育馆在顶点 T_2。现在让我们来看看朱丽叶可能到达哪些顶点。

由于朱丽叶每次都沿着某一条去教堂最近的路，所以我们首先用 dijkstra 算法求出 S_1 到任一个顶点 i 的最短时间 $Time_i$，然后从 T_1 开始从后往前倒推。我们用 A 表示已知的朱丽叶可达的点的集合（初始时 $A=[T_1]$），如果存在顶点 i，j，使得 $Time_j+D_{ij}=Time_i$，$i\in A$，$j\notin A$，那么可知 j 是可达点，令 $A=A+[j]$。如此反复修改集合 A 的值，直到无新元素可添加，我们就知道了所有朱丽叶可能到达的顶点和相应的到达时间。同样地，也用集合 B 记录所有罗密欧可能到达的顶点和到达时间。如果集合 A 和 B 有顶点相同且时间相同的元素，那么朱丽叶的愿望便可能实现。

2. 出纳员的雇佣——bellman-ford算法

出纳员的雇佣[①]

Tehran 的一家每天 24 小时营业的超市，需要一批出纳员来满足它的需求。超市经理雇佣你来帮他解决一个问题——超市在每天的不同时段需要不同数目的出纳员（例如：午夜时只需一小批，而下午则需要很多）来为顾客提供优质服务，他希望雇佣最少数目的出纳员。

经理已经提供你一天里每一小时需要出纳员的最少数量——$R(0), R(1), \cdots, R(23)$。$R(0)$ 表示从午夜到上午 1:00 需要出纳员的最少数目，$R(1)$表示上午 1:00 到 2:00 之间需要的，等等。每一天，这些数据都是相同的。有 N 人申请这项工作，每个申请者 i 在每天 24 小时中，从一个特定的时刻开始连续工作恰好 8 小时，定义 t_i（$0\leqslant t_i\leqslant 23$）为上面提到的开始时刻。也就是说，如果第 i 个申请者被录用，他（她）将从 t_i 刻开始连续工作 8 小时。

你将编写一个程序，输入 $R(i)$（$i=0\cdots23$）和 t_i（$i=1\cdots N$），它们都是非负整数，计算为满足上述限制需要雇佣的最少出纳员数目。在每一时刻可以有比对应的 $R(i)$更多的出纳员在工作。

【分析】

这道题目看起来和图论没什么关系，但事实上它可以用最短路问题的另一个解法 bellman-ford 算法来解决。下面介绍这个算法。

虽然实际问题中一般以正权的形式出现，但是很多情况下负权也是存在的。这个时候，**最短路不一定存在**。容易证明在无向连通图中，两点间的最短路存在当且仅当图中不含**负权圈**。由于问题变复杂了，我们无法得到像 dijkstra 那样时间复杂度这么低的方法，但是仍然有一个不错的算法，它就是 Bellman-ford 迭代算法。

既然是迭代算法，就不能像 dijkstra 那样，按照一定的顺序直接算出每个点的距离值，而是不断地修改每个点的当前最小距离值，直到求出真正的最短距离为止。

算法的思想是基于以下事实：“两点间如果有最短路，那么每个顶点最多经过一次。也就是说，这条路不超过 $n-1$ 条边。”（如果一个顶点经过了两次，那么我们走了一个圈。如果这个圈的权为正，显然这是不合算的；如果是负圈，那么最短路不存在）。

根据最短路的最优子结构性质，路径边数上限为 k 时的最短路可以由边数上限为 $k-1$

① 题目来源：ACM/ICPC Regional Contest Tehran 2000

时的最短路“加一条边”来求出，而根据刚才的结论，最多只需要迭代 $n-1$ 次就可以求出最短路。这就是著名的 Bellman-ford 算法

Bellman-ford 算法：初始 d[起点]=0，其他点 d 值为∞。从 1 开始迭代路径长度上限为 k，每次考虑图中的每条边（u,v）并检查是否有 $d_v>d_u+w(u,v)$。若有，则更新 d_v 的值。写成伪代码就是：

```
for k:=1 to n-1 do
  for 每条边(u,v) do
    if (d[u]<∞) and (d[v]>d[u]+w(u,v)) then d[v]:=d[u]+w(u,v)
```

每次比较 d_v 和 $d_u+w(u,v)$及更新 d_v 叫做 一次**松弛操作**。算法的时间复杂度为 $O(nm)$。注意，这里有一个优化：如果某次迭代时没有任何一个 d 值改变，就可以立刻退出迭代而不需要把所有 $n-1$ 次迭代都做完。在很多时候这个优化是很明显的，特别是初始情况已经接近最短路的时候。为了获得好的初始解，可以利用 dijkstra。

下面回到刚才的问题。首先将输入数据整理成：

r[0…23]——每个时刻需要的出纳员数目。

t[0…23]——每个时刻应征的申请者数目。

求 s[0…23]——$s[i]$表示 0~i 时刻雇用的出纳员总数。$s[i]$-$s[i-1]$就是在 i 时刻录用的出纳员数目。设 $s[-1]=0$，sum 为雇用的所有出纳员总数。于是一个可行方案满足：

$$\begin{cases} s[i]-s[i-1]\geqslant 0 \\ s[i-1]-s[i]\geqslant -t[i] \\ s[23]-s[-1]\geqslant sum \\ s[i]-s[j]\geqslant \begin{cases} r[i] & i>j \\ r[i]-sum & i<j \end{cases} \qquad i=(j+8)\bmod 24 \end{cases}$$

当 sum 已知的时候，根据这些不等式构造图 G：以-1，0，…，23 为顶点，如果存在不等式 $s[a]$-$s[b]\geqslant c$，就从 b 向 a 连一条权为 c 的有向边，以-1 为起点，用迭代法求单源最长路，如果途中不存在圈且 $s[23]=sum$，就找到一个可行解了。

由于 sum 实际上是不可知的，所以只要在主程序中枚举 sum，就可以得到需要雇佣的出纳员数目的最小值。当然，我们还可以用二分法。

3．每对结点间的最短路的floyd-warshall算法

本节中，我们来学习一个新的问题：求每对结点之间的距离。显然我们可以对每个点求解一次 SSSP，但是下面这个方法更方便且容易记忆。

再次考虑最短路的最优子结构性质。设 $d[i,j,k]$是在只允许经过结点[1…k]的情况下，i 到 j 的最短路长度，则它有两种情况（想一想，为什么）：

- 如果最短路经过点 k，那么 $d[i,j,k]$应该等于 $d[i,k,k-1]+d[k,j,k-1]$；
- 如果最短路不经过点 k，那么 $d[i,j,k]$应该等于 $d[i,j,k-1]$。

综合起来：$d[i,j,k]=\min\{d[i,k,k-1]+d[k,j,k-1],d[i,j,k-1]\}$，边界条件是 $d[i,j,0]=w(i,j)$（不存在的边权为∞）。

相信读者看到这里已经明白，我们实际上是做了一次动态规划。由于在递推过程中 k 是递增的，所以只需要一个二维数组就可以了。具体来说就是：

Floyd-Warshall 算法：初始 $d[i, j]=w(i, j)$。从小到大枚举 k，对每对结点 (u, v)，检查更新它们的最短路值。伪代码为：

```
for k:=1 to n do
for i:=1 to n do
  for j:=1 to n do
    if (d[i,k]<∞)and(d[k,j]<∞)and(d[i,k]+d[k,j]<d[i,j])then
      d[i,j]:=d[i,k]+d[k,j]
```

时间复杂度为 $O(n^3)$，空间复杂度为 $O(n^2)$。

WEB 最短路问题是最基本的图论问题之一。APSP(All Pairs Shortest Paths)有复杂度为 $O(mn+n^2\log n)$的算法，当 m 较小时非常有用。进一步地，在无向图上还有 $O(mn\alpha(m,n))$的算法。如果是无向图，权非负且为整数，那么 SSSP 可以在 $O(m)$之内解决，从而 APSP 可以在 $O(mn)$内解决。如果图比较特殊，例如是平面图，或者线段相交图，SSSP 有快得多的算法。另外一个重要的问题是动态的 APSP 问题，边权是可以改变的。研究这些问题的相关论文都可以在本书主页上找到。

【例题 1】瘦陀陀和胖陀陀[1]

一场可怕的战争后，瘦陀陀和他的好朋友胖陀陀（如图 2-55 所示）将要凯旋。瘦陀陀处在城市 A，胖陀陀处在另外一个未知的城市。他们打算选一个城市 X（这个由瘦陀陀来决定），胖陀陀会赶在瘦陀陀之前到达城市 X，然后等待瘦陀陀也赶到城市 X 与他汇合，并举办一次庆祝宴会（由瘦陀陀请客），接着一起回到他们的家乡城市 B。由于胖陀陀嘴馋，他要求举办宴会的城市必须是瘦陀陀回家的路线中举办宴会最贵的一个城市。

图 2-55　瘦陀陀和胖陀陀

[1] 题目来源：UVA Problem Archive Online Contest

瘦陀陀正专注地看回家的地图，地图上标有 n（$n\leqslant 200$）个城市和某些城市间直达的道路，以及每条道路的过路费。瘦陀陀还知道在每一座城市举办一次宴会的花费。现在给出地图描述和城市 A、B 的位置，请你告诉瘦陀陀回家的最小费用。你的程序会接收到多次询问，即对于每对城市(c_1,c_2)，你的程序应该立刻给出瘦陀陀从 c_1 到 c_2 的最小花费。

【分析】

由于胖陀陀规定必须在最贵的城市举办宴会，因此不能简单地选择一条几何上的最短路走——如果路上有一个花费特别贵的城市，总的开销会剧增。如果不考虑聚会，那么问题就简单了，因此我们考虑把聚会地点确定下来，即：对于每个点，先把它作为聚会地点 X，然后删去比它贵的宴会地点（因为路径不能经过那些点，否则需要在那些更贵的点聚会）以后求出 AX 和 XB 的最短路，连在一起就是所求的路线。这样，每次询问的时间算法复杂度为 $O(nm+n^2\log n)$（假设 dijkstra 用标准堆实现）。

可是询问可能有很多，能不能事先把所有可能的询问都计算出来，然后询问的时候只查表就可以了呢？回答是肯定的。不过不能简单的用刚才的算法对每对结点执行一次，这样时间复杂度就为 $O(n^3m+n^4\log n)$，太高了。可以这样改进：对于每个聚会地点 X，预先算出删去比它贵的点以后 X 到其他所有点的最便宜路径，预处理复杂度为 $O(nm+n^2\log n)$，然后每次枚举 X 以后可以直接读出 AX 和 XB 的最小花费，故询问时间复杂度为 $O(n)$，计算所有询问的时间复杂度为 $O(n^3)$。

【例题 2】新桥[1]

一座城市有 n（$n\leqslant 5\,000$）个小区，小区间有 m（$m\leqslant 100\,000$）个双向行驶的道路，如图 2-56 所示。小区 A 的居民每天要到小区 B 或者小区 C 工作。为帮助大家节约路上的时间，市长打算修建一座长度不超过 Lmax 的新桥，使得修建新桥后：

① 从小区 A 到小区 B 的最短路径比现有最短路径要短；

② 从小区 A 到小区 C 的最短路径比现有最短路径要短；

③ 在满足条件①、②的基础上使得两条最短路的长度和最小。

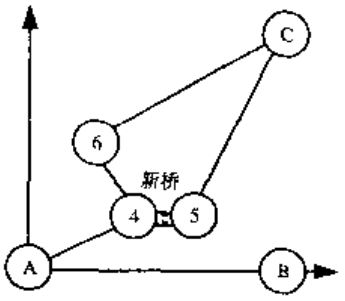


图 2-56　新桥

【分析】

一个最容易想到的方法是：枚举新桥连接的两个小区 u,v，然后在加入边(u,v)后重新计算 A-B 和 A-C 的最短路长度。这样做的最坏情况时间复杂度为 $O(n^2m+n^3\log n)$，不令人满意，

[1] 题目来源：Balkan Olympiad in Informatics

我们需要把算法加以改进。

可以注意到，如果加入了新桥(u,v)，那么新的A-B，A-C必须要经过桥(u,v)才能满足条件。也就是说，我们只需计算A-u，A-v，B-u，B-v的长度，A-B的最短路只可能是A-u-v-B或者A-v-u-B。这样做以后，预处理的复杂度为$O(m+n\log n)$，主过程的复杂度为$O(n^2)$，总的复杂度为$O(n^2+m)$。

还可以顺着这个思路进一步改进该算法。假设新桥是(u,v)，那么是否有可能A-B的最短路为A-u-v-B而A-C的最短路为A-v-u-C呢？不可能！因为新的两条路径长为$L(A,u)+L(u,v)+L(v,B)+L(A,v)+L(v,u)+L(u,C)>L(A,u)+L(u,C)+L(A,v)+L(v,B)$。而等式的右边对应于在没有桥之前就有的合法通路$A$-$v$-$B$和$A$-$u$-$C$。因此，两条新路长的和大于两条老路的长度和，故这不可能是合法解。

这样，任何一个合法解都可以表示为桥（u,v），而最优路径为A-u-v-B和A-u-v-C。即的枚举方式变为：先枚举v，求出v-B和v-C，然后枚举u，使得$L(u,v)+L(A,u)$尽量小。为了加速，可以按照$L(A,u)$递增的顺序或者$L(u,v)$递减的顺序枚举。虽然最坏情况时间复杂度还是$O(n^2+m)$，但是运行时间会缩短。

下面再来考虑一下$L\max$。如果可以建造桥的位置数目k不大（例如$k=O(m)$），那么如果不重复遗漏的枚举这k个位置，复杂度将变为$O(k+m+n\log n)$，速度也将有很大提高。可以用分治法来解决这个问题，这在本书的几何部分已有较详细的讨论。

4. 最短路算法总结和应用举例

刚才我们介绍了最短路的 dijkstra 算法和 bellman-ford 算法，但是并未把最短问题的算法上升到理论高度。在下面内容中，我们来深入研究这一问题。

在问题满足最优性原理的时候，可以使用类似于 dijkstra 算法的标号法（labelling algorithm）来完成。标号算法的共同特点是在每个结点上赋予一个或多个标号，它可以分为两种类型。

类型一：标号修正算法。算法是迭代式的，标号都是临时的（因为标号只代表距离上界）。算法思想是不断地逼近最优解，只在最后一步才确定想要的结果。Bellman-ford 算法属于此类型。

类型二：标号设定算法。算法的执行过程是不断把临时标号变成永久标号的过程，大家熟悉的 dijkstra 即属于此类型。

需要说明的是，标号修正算法由于需要研究终止条件和迭代次数，往往不如标号设定算法容易掌握，效率也不如后者高，但求解范围更广（例如 Bellman-ford 可以求解一般费用网络，而 dijkstra 只适合非负权网络）。标号设定算法的核心过程就是不断地把临时标号永久化，在永久化的同时产生（如果可能的话）新的临时标号。我们在这里举三个例子，加深读者对标号设定算法的理解。

【例题 3】穿越沙漠[3]

没有水或食物，你将无法前行。为了穿越沙漠，应该购买多少食物呢？

在出发地，你可以采购食物和获得无限多的免费水，但由于背包容量有限，在任意时

[3] 题目来源：ACM/ICPC World Finals 2002

候拥有的水和食物总量不得超过一个给定值 *limit*。沙漠中有若干绿洲，在绿洲你可以得到无限多的水，可以存储食物。当然也可以在出发地存储食物。

你将被告知出发地、目的地和所有绿洲的平面位置。每走 x（x 是实数）单位长的路程你就要消耗 x 个单位的水和 x 个单位的食物。请确定在出发地最少购买的食物量。注意：出发地的食物按整数单位出售且总量为 10^6。

【分析】

由于水是免费的，所以我们需要让带的食物尽量少。假设已经知道从绿洲 j 出发时必须在绿洲 j 保存的食物总量 $d[j]$，考虑从绿洲 i 出发，第一次到达绿洲 j 的情况：需要在绿洲 i 保存多少食物，才能保证在绿洲 j 保存 $d[j]$单位的食物？

为了在绿洲 j 储存 t 单位的食物，我们需要在 i 和 j 之间进行往返。假设 i 和 j 的直线距离为 *dist*，则每次应该携带 *dist* 单位的水和 *limit-dist* 单位的食物到 j，然后从 j 带 *dist* 单位的水和 *dist* 单位的食物返回 i。每次往返让 j 多储存了 $limit-3\times dist$ 单位的食物，而最后一次到 j 后并不需要返回 i，因此最后一次可以在绿洲 j 储存 $limit-2\times dist$ 单位的食物。因此，需要往返的次数 m 为不小于$(d[j]-limit+2\times dist)/(limit-3\times dist)$的最小整数（注意需要判断无法进行往返的情况）。在往返时消耗的食物量为$(2m+1)\times dist$，因此从绿洲 i 出发第一次到达绿洲 j 时，至少需要储存 $d[j]+(2m+1)\times dist$ 单位的食物。

剩下的问题就简单了。虽然我们从绿洲 i 出发第一次并不一定是到达绿洲 j，但可以利用最短路的标号法思想，每次选择需要储存食物最少的绿洲，把它的标号永久化。每次确定一个永久标号的时间花费为 $O(n)$，故总的时间复杂度为 $O(n^2)$。

【例题 4】隐形石头[①]

在一个 $N\times M$ 的矩形网格上，有些格子是空地，有些格子是石头。你希望从其中一个空地格（称为起点 E）花尽量少的时间走到另外一个空地格（称为终点 X）。每次可以往上、下、左、右之一的方向前进一格。

你几乎记得全部地图，只是忘记了一个石头的位置，即：你所记得的空地格中，恰好有一个其实是石头，而你所记得的石头确实都是石头。更糟的是，由于你的视力比较差，因此必须走到那个“隐形”（其实是你没记住）石头的相邻格子中才能看到它。请设计一种策略，使得最坏情况下花的时间最短。例如图 2-57 的情况。

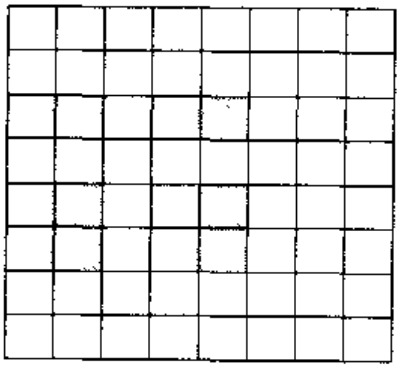


图 2-57　迷宫举例

[①] 题目来源：UVA Problem Archive Online Contest. 命题人：Jimmy Mårdell

如果一开始往上走，走到十字路口面前才发现原来十字路口那里是块石头，那么需要17个单位时间才能走到X。如果一开始就往右走，那么无论发现石头在哪里（甚至根本没有看到隐形石头），总能保证在15个单位时间走到X。

【分析】

这道题目的第一个难点在于“策略”应该怎样表示。似乎它并不是一条固定的路线，而是一个动态的东西，可以根据隐形石头的位置进行改变。好在石头只有一个，可以用一条路径来表示。我们采取的策略是：如果路径上没有隐形石头，就沿着路径走到终点；只要一遇到石头，由于石头只有一个，我们立刻可以计算出新的最短路并沿着它到达X。

有了这个简单表示，就可以把dijkstra算法加以变形用到此题中。假设在格子A处采取最优策略情况下的最坏最短时间为d[A]，则d[A]可以这样计算：假设采取朝上走的策略，分如下两种情况。

情况一：上面的格子是隐形石头，那么花费的时间为distance[up[A]，A，X]。其中distance[A,B,C]的含义是在地图上把格子A设置为石头后格子B到格子C的最短路，它可以通过预处理计算出来。

情况二：上面的格子是空地，那么花费的时间（最坏情况）为d[up[A]]+1。

类似地，我们可以讨论采取往左、右、下走的情况。

当然，情况二可以计算的前提是d[up[A]]已知。有的读者想到这里可能会陷入一种困境：情况二说明我们必须按照一个拓扑顺序进行计算d值，但是上下左右是对称的，我们在希望d[up[A]]先被求出的同时也可以用同样的理由考虑d[up[A]]，而希望d[A]先被求出。事实上，这里的拓扑顺序已经不再来源于图中的边了（请读者对比有向无环图的拓扑顺序以及在动态规划中的应用），而是d值的大小。d是非负的，而且在计算过程中不减，这才是本题的算法乃至于更一般的最短路的标号算法可行的根本原因。

【例题5】双调路径[①]

如今的道路密度越来越大，收费也越来越多，因此选择最佳路径是很现实的问题。城市的道路是双向的，每条道路有固定的旅行时间以及需要支付的费用。路径由连续的道路组成。总时间是各条道路旅行时间的和，总费用是各条道路所支付费用的总和。同样的出发地和目的地，如果路径A比路径B所需时间少且费用低，那么我们说路径A比路径B好。对于某条路径，如果没有其他路径比它好，那么该路径被称为最优双调路径。这样的路径有可能不止一条，或者说根本不存在。

给出城市交通网的描述信息、起始点和终点城市，求最优双调路径的条数。城市不超过100个，边数不超过300个，每条边上的费用和时间都不超过100。

【分析】

这道题目棘手的地方在于标号已经不再是一维，而是二维，因此不再具有全序关系。我们让d[i,c]表示从s到i费用为c时的最短时间，可以通过预处理计算出每个结点s到任意结点i的最短时间基础上的最小花费c_i，则我们只需要保留标号$d[i,0], d[i,1], \cdots, d[i,c_i]$。

其实，我们并不需要严格的拓扑顺序，而只需要一个让标号永久化的理由。拓扑顺序

① 题目来源：Baltic Olympiad in Informatics, 2002

能保证标号的永久化，但还有其他方式。在本题中，标号 $d[i,c]=t$ 永久化的条件是：从其他永久标号得不到费用不大于 c 且时间不大于 t 的临时标号（注意：我们利用了费用和时间的非负性），即：所有的“极小临时标号”都可以永久化。这样，一个附加的好处是一次把多个临时标号同时变成永久的。

假设时间上限为 t，费用上限为 c，城市数为 n，边数为 m，则每个点上的标号不超过 $O(nc)$个，标号总数为 $O(n^2c)$个，每条边考虑了 $O(nc)$次。如果不同顶点在同一费用的临时标号用堆来维护，不同费用的堆又组成一个堆的话，那么建立（或更新）临时标号的时间为 $O(mnc\log n\log c))$，总的时间复杂度为 $O(n^2c+mnc\log n\log c)$，本题的规模是完全可以承受的。实际上由于标号总数往往远小于 n^2c，程序效率是相当理想的。

WEB 最短路还有一种完全不同的解法称为偏离算法（deviation algorithms），由于篇幅比长，已经在本书中删除了。有兴趣的读者可以在本书主页上找到这些内容的介绍。

练　习　题

思考题：

2.5.1　设计一个 $O(m^2)$的算法，求出一个正权无向图中从 s 出发到 t 的路径中最大边和最小边相差最小的一条。

***2.5.2**　在上题中，如果要把时间复杂度降低到 $O(m\log m)$，需要解决一个什么动态问题？

2.5.3　设计一个 $O(n^3)$的算法，求出正权无向图的最小圈。

***2.5.4**　修改 dijkstras 算法，使它能求出前 k 短路。这些路可以重复经过一些结点或者边。

***2.5.5**　修改 floyd-warshall 算法，使它能算出每两点间的前 k 短路。你的算法复杂度应该是 $O(n^3k)$。

编程题：

2.5.6　路的最小共倍数[①]

给出一个带权无向图（边权为 1…1 000 的整数）G。对于 v_0 到 v_1 的任意一条简单路 p，定义 $s(p)$为 p 上所有边权的最大公约数。考虑 v_0 到 v_1 的所有路 $p_1,p_2,\cdots$，求所有 $s(p_1),s(p_2),\cdots$ 的最小公倍数。

2.5.7　货币兑换[②]

若干个货币兑换点在我们的城市中工作着。每个兑换点只能进行两种指定货币的兑换。不同兑换点兑换的货币有可能相同。每个兑换点有它自己的兑换汇率，货币 A 到货币 B 的

[①] 题目来源：USACO 2003
[②] 题目来源：经典问题

汇率表示要多少单位的货币 B 才能兑换到一个单位的货币 A。当然货币兑换是要支付一定量的中转费。例如，如果你想将 100 美元兑换成俄元，汇率是 29.75，中转费率为 0.39，那么你会兑换到$(100-0.39)\times 29.75=2963.3975$ 俄元。

城市中流通着 N 类货币，用数字 1 至 N 标号表示。每个货币兑换点用 6 个数字来描述：整数 A 和 B 是兑换货币的编号，实数 RAB、CAB、RBA、CBA 分别是 A 兑换成 B 和 B 兑换成 A 的汇率和中转费率。

尼克有一些第 S 类货币，他想在若干次交换后增加它的资金，当然这些资金最终仍是第 S 类货币。请你告诉他应该怎么做。

2.5.8　速度限制[①]

在这个繁忙的社会中，我们往往不再去选择最短的道路，而是选择最快的路线。开车时每条道路的限速成为最关键的问题。不幸的是，有一些限速的标志丢失了，因此你无法得知应该开多快。一种可以辩解的解决方案是，按照原来的速度行驶。你的任务是计算两地间的最快路线。

你将获得现代化城市的道路交通信息。为了使问题简化，地图只包括 N 个（$2\leqslant N\leqslant 150$）路口（标号为 $0\cdots N-1$）和 M 条道路。每条道路是有向的，只连接了两条道路，并且最多只有一块限速标志，位于路的起点。两地 A 和 B，最多只有一条道路从 A 连接到 B。你可以假设加速能够在瞬间完成并且不会有交通堵塞等情况会影响你。当然，你的车速不能超过当前的速度限制。注意开始时你位于 0 点，并且速度为 70。

2.5.9　会议呼叫[②]

有一个电信公司最近推出了一种三方的会议呼叫系统，允许三个客户同时进行通话。下图描述的是一个电信网络，其中圆圈代表客户端，每条线段两端的客户都可以直接相互。为了让三个客户同时通话，公司只需要激活一些直接连接，使得三个客户端连通。由于激活不同的直接连接的费用不同，公司希望你为它找一个最省钱的激活方式。例如图 2-58，为了连接客户 1，4，6，最好的方式是激活加粗的直接连接，总费用是 27。

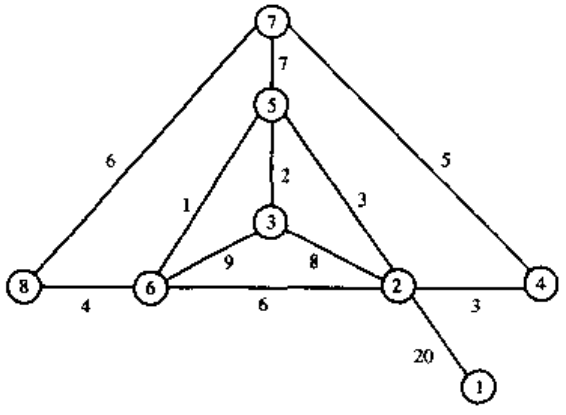


图 2-58　会议呼叫系统

[①] 题目来源：Baltic Olympiad in Informatics, 2002
[②] 题目来源：Balkan Olympiad in Informatics, 1998

***2.5.10　公平会面[①]

两位外交官处在不同的城市 *A* 和 *B*，他们打算请你选择一座城市 *X* 作为会面的地点，并分别为他们制定详细的到达路线。为公平起见，两位外交官的路线总长要一样，并且任何一位外交官不会经过同一座城市两次，还有就是除了会面地点两位外交官不会到达过同一座城市。在这个基础上，你应该使得路线长度最短。假设这样的会面地点和到达路线一定存在。

如图 2-59 所示，市编号从 1 到 p（$1 \leqslant p \leqslant 200$），如果城市 i 和城市 j 之间有一条路，那么用整数 Di, j（$1 \leqslant Di, j \leqslant 10$）表示这条路的长度。

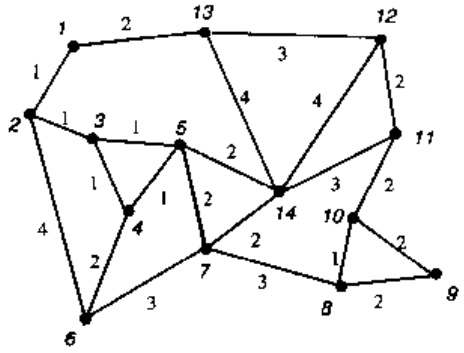


图 2-59　城市地图

输入文件的第一行有三个整数用空格隔开，第一个整数为 L（$L \leqslant 500$），表示以下有 L 行。第二个和第三个整数表示两个外交官目前所在的城市。下面有 L 行，每行有三个数 i，j 和 Di, j，表示城市 i 和城市 j 之间有一条长度为 Di, j 的路。注意，在城市 i 和城市 j 之间可能存在多条路。

输出包含两行，每行为一个外交官的行进路线。每条路线由一系列整数构成，每个整数表示路线上经过的城市。第一个点是外交官初始时所在的城市，最后一个点表示两个外交官的会晤城市。两条路线的的顺序并不重要。

2.5.3　网络流问题

本小节介绍最大流问题。此问题的应用非常广泛，且难度比较大，读者需认真体会。

1．奶牛的新年晚会——最大流问题

奶牛的新年晚会[②]

奶牛们将要举办一次别开生面的新年晚会。每头奶牛会一些不同样式的食品（单位是“盘”），到时候她们会把自己最拿手的不超过 k 样食品各做一盘（例如一个馅饼、一根香肠和一片面包）带到晚会上，和其他奶牛同胞一起分享。但考虑到同种食品带得太多了

[①] 题目来源：Balkan Olympiad in Informatics, 1998
[②] 题目来源：USACO

会因为没有牛愿意吃而浪费，奶牛们给每种食品的总盘数都分别规定了一个不同的上限值。但这样让她们很伤脑筋：究竟应该怎么做，才能让晚会上的食品总盘数尽量多呢？

例如，有 4 头奶牛，每头奶牛最多可以带 3 盘食品。一共有 5 种食品，它们的数量上限是 2，2，2，2，3。奶牛 1 会做食品 1~4；奶牛 2 会做食品 2~5；奶牛 3 会做食品 1,2,4；奶牛 4 会做食品 1,2,3。那么，她们最多可以带 9 盘食品到晚会上。即：奶牛 1 做食品 2~4；奶牛 2 做食品 3~5；奶牛 3 做食品 1 和 2；奶牛 4 做食品 1。这样，四种食品各有 2，2，2，2，1 盘，没有超过它们各自的数量限制。

【分析】

本题可以用一个图来表示。如果奶牛 a 会做食品 b，那么我们连一条边（Cow[a], Food[b]）。这样建模完毕以后，我们的目的是选一些边，从每头牛出发最多有 k 条边被选中；从每种食品 i 出发最多有 u_i 条边（u_i 为食品 i 的盘数上限）被选中。但是，这样的模型是无法操作的，下面把这个图改造一下，用它来表示一个可行方案（不一定是最优的）。注意，不是用这个图来表示问题本身，而只是表示一个解。

首先，根据方案把原始图上的边加权，被选中的边权为 1，没被选中的边权为 0。

然后，加一个点 s，把它和每个奶牛结点 u 连成一条边（s,u），它的权等于从 u 出发的所有边的权和（即：这些边被选中的条数）。因此 $w(s,u)\leqslant k$。

最后，加一个点 T，把它和每个食品结点 v 连成一条边（v,t），它的权等于连接到 v 的所有边的权和。因此 $w(v,T)\leqslant u_v$。

这样的图就表示了一个方案，而且和题目中给出的所有要素联系起来了。但是，怎样让 $w(s,u)$等于从 u 出发的所有弧的权和呢？这里需要一点创造性思维。设想有一股水流从 s 流到 t。水从 s 流入后，分叉经过每个 X 结点和 Y 结点，最终汇集到 t。如果水在过程中没有消耗（如蒸发、渗透、泄漏等），那么流在弧（s,u）中的水量一定等于 u 出发的所有弧中的水量和。想到什么了吗？对！如果我们把刚才定义的权看成是流量，那么关系式 $w(S,u)=\sum_{(u,v)\in E} w(u,v)$ 就是情理之中的了。下面把这个定义形式化，即：

定义：给有向无权图 $G=(V, E)$中的每条弧（u,v）赋予一个值 $f(u,v)$，并规定两个点 s 和 t。如果对于除 s，t 外的任意一个结点 i，都有：$\sum_{(u,i)\in E} f(u,i)=\sum_{(i,v)\in E} f(i,v)$ ①

那么我们说 f 是图 G 的一个流，并把 s 称为源（source），t 称为汇（sink）。

我们把①叫做**流量平衡条件**。它表示除了源和汇之外的中间结点既不消耗流量，也不提供流量，它们只起中转作用。源是流的提供者，而汇是流的消耗者。

在刚才的问题中，并不是任意满足流量平衡条件的流都合法，它还必须满足：

$f(u,v)\leqslant 1$（每头奶牛不能做两盘相同食品）；$f(s,u)\leqslant k$ 和 $f(v,t)\leqslant u_v$②

总盘数最大实际上是要求：汇收集到的流量最大（它等于源提供的流量）。

如果给每条弧规定一个流量上限（称为**容量**），那么②可以统一为**容量限制条件**："所有弧的流量不超过它的容量"。把这些因素综合起来，最大流问题就自然地呈现在我们面前了。

最大流问题：求有向带权图 $G=(V, E, C)$的一个流，它满足容量限制条件 $f(u,v)\leqslant C(u,v)$，且源点提供的流尽量大。

2. 航天计划问题——最小切割最大流定理

航天计划问题[①]

NASA 正在为航天飞行计划一系列的太空飞行，在每次飞行中必须决定进行何种商业性实验和配载何种仪器设备。对每次飞行，NASA 考察一个实验集合 $E=\{E_1,\cdots,E_n\}$，并且实验 E_j 的商业赞助人已同意为该实验的结果向 NASA 支付 P_j 美元。实验使用的全部仪器的集合为 $I=\{I_1,\cdots,I_n\}$；每个实验所需要用到的仪器为 I 的子集 R_j，运送仪器 I_k 的费用为 C_k 美元。请确定要进行哪些实验并运送哪些仪器才能使净收益最大。其中净收益指进行实验所获得的全部收入与运送仪器的全部费用的差额。

【分析】

显然，我们只需要选择一个进行实验的集合 A，则运送的仪器的集合为 A 的邻集 B。净收益为 A 中所有实验的收益和减去 B 中所有仪器的费用和。

我们建立一个二分图 G，它的 X 结点为所有的仪器，每个仪器 i 用 X_i 来表示，连接一条容量为 I_i 的弧 $s\rightarrow X_i$；它的 Y 结点为所有的实验，每个实验 i 用 Y_i 来表示，连接一条容量为 E_i 的弧 $Y_i\rightarrow T$。如果实验 i 要用到仪器 j，则连接一条容量为无穷大的弧 $X_j\rightarrow Y_i$。这个网络的最大流是可以求出来的，可是这个最大流有什么意义吗？为了说清这个问题，我们先来看看网络的割。

定义：如果去掉某弧集 s 中的所有弧以后源点 s 和汇点 t 不再连通，则把弧集 s 称为网络的 s-t 割。

割的容量为其中所有弧的容量之和。去掉割以后 s 所在的连通分量叫做 S 集，t 所在的连通分量叫做 T 集。

显然，所有弧的集合就是一个割，它是容量最大的割。我们很自然希望得到容量最小的割，为此可以利用如下定理。

最小切割最大流定理：s-t 最大流的流值等于 s-t 的最小切割容量。

证明略去，读者可以在几乎任何一本介绍网络流的书籍中找到。回到刚才的问题中，我们求出了最大流的同时求出了最小割。假设处在 T 集中的 Y 结点集合为 A，则选择进行 A 中的实验是最好的方案。

分两步来证明这个结论：

步骤 1：最小切割不可能包含无限容量的边。这是显然的，因此如果在 T 集中的任一个实验 E_i 需要仪器 I_j，那么 I_j 一定也在 T 集中，它保证了切割一定对应一个可行方案。

步骤 2：对于任何一个仪器 i，如果仪器在 T 集合中，说明该仪器应被运送，这时容量 I_i 被计算在了切割中；对于任何一个实验 j，如果实验在 T 集合中，说明实验可以完成，这时容量 E_j 并没有计算在切割中。这样，该方案所对应的切割数值上等于 $sum\{I_i\}+sum\{E_j\}$，

[①] 题目来源：经典问题

其中 i 取遍所有运送的仪器，j 取遍所有不进行的实验。如果设所有实验的总收益为 E，则上式可以写成 E-($sum\{E_j\}$-$sum\{I_i\}$)，其中 j 取遍所有进行的实验，i 取遍所有运送的仪器，这恰好就是题目中定义的净收益。这样，最小切割就对应了最大收益。

WEB 虽然有最小切割最大流定理，但是求最小割的方法中不都需要依赖于流。事实上，脱离了最大流，我们可以有很多优秀的算法，有兴趣的读者可以参考相关资料。还有一个有趣的问题：哪些弧是不必要的，即删去以后最大流不变？这个问题已经有了一些结果，虽然有弧必要的充分必要条件，但是判断条件是否成立却是个 NP 完全问题。这些内容都可以在本书主页上找到简要的介绍

3. 终极情报网——最小费用流问题

终极情报网[①]

在最后的诺曼底登陆战开始之前，盟军与德军的情报部门围绕着最终的登陆地点展开了一场规模空前的情报战。

这场情报战中，盟军的战术是利用那些潜伏在敌军内部的双重间谍，将假的登陆消息发布给敌军的情报机关的负责人。那些已经潜入敌后的间谍们都是盟军情报部的精英，忠实可靠；但是如何选择合适的人选，以及最佳的消息传递方法，才能保证假消息能够尽快而且安全准确地传递到德军指挥官们的耳朵里，成了困扰盟军情报部长的最大问题。他需要你的帮助。

具体来说，在敌后一共潜伏着我方最优秀的 N（N<300）名间谍，分别用数字 1，2，…，N 编号。在给定的作战时间内，任意两人之间至多只进行一次点对点的双人联系。我将给你一份表格，表格中将提供任意两位间谍 i 和 j 之间进行联系的安全程度，用一个[0，1]的实数 $S_{i,j}$ 表示；以及他们这次联系时，能够互相传递的消息的最大数目，用一个正整数 $M_{i,j}$ 表示（如果在表格中没有被提及，那么间谍 i 和 j 之间不进行直接联系）。

假消息从盟军总部传递到每个间谍手里的渠道也不是绝对安全，我们用 [0，1] 的实数 AS_j 表示总部与间谍 j 之间进行联系的安全程度，AM_j 则表示总部和间谍 j 之间进行联系时传递的消息的最大数目。对于不和总部直接联系的间谍，他的 AM_j=0（而表格中给出的他的 AS_j 是没有意义的）。当然，假消息从间谍手中交到敌军的情报部官员的办公桌上的过程是绝对安全的，也即是说，间谍与敌军情报部门之间要么不进行直接联系，要么其联系的安全程度是 1（即完全可靠）。

现在情报部打算把 K（K<300）条假消息"透露"到德军那里。消息先由总部一次性发给 N 名间谍中的一些人，再通过他们之间的情报网传播，最终由这 N 名间谍中的某些将情报送到德军手中。

对于一条消息，只有安全的通过了所有的中转过程到达敌军情报部，这个传递消息的过程才算是安全的；因此根据乘法原理，它的安全程度 P 就是从总部出发，经多次传递直到到达德军那里，每一次传递该消息的安全程度的乘积。而对于整个计划而言，只有当 N

[①] 题目来源：CTSC 2001. 命题人：张力

条消息都安全的通过情报网到达德军手中，没有一条引起怀疑时，才算是成功的。所以计划的可靠程度是所有消息的安全程度的乘积。

显然，计划的可靠性取决于这些消息在情报网中的传递方法。我需要一个方案，确定消息应该从哪些人手中传递到哪些人手中，使得最终计划的可靠性最大。

【分析】

题目的描述比较长，但比较直观，容易看出这是个网络流的模型。仔细分析题目的条件，我们可以找到网络中的点，即：一个源点 s，即盟军总部；一个汇点 t，即敌军的情报部；中间经过的点，即间谍。但是和最大流问题不同的是，边权上升到了两个，即相互之间可传递的最大消息数 C 和相互之间传递的安全系数 P。

前五个要素都是我们所熟悉的，但是安全系数是什么呢？首先，由于最终的安全系数等于每个消息经过的路径的安全系数的乘积，而如果给安全系数取自然对数再反号，安全系数 P 就变成了非负实数 W（我们把它称为“费用”），那么求乘积的关系就变成了求和的关系。这样，我们的目标不再是流最大，而是要每条边上流量与费用的乘积之和最小。

最小费用最大流问题：给定网络 $G=(V, E, C, W)$，求网络上的一个流 f，使得 f 是网络的最大流，且每条弧的流量与费用的乘积加起来的总和 $\cos t = \sum_{(u,v)\in E} f(u,v)\times w(u,v)$ 最小。

最小费用流问题比最大流要困难得多，理论和算法步骤也要复杂些，很多内容都超出本书的范围。这里只介绍最容易理解的最小费用增广路算法和效率比较高的松弛算法，其他的算法在此略去。

4．最大流问题：算法和应用举例

对于任意一条弧（u,v），如果 $f(u,v)>0$，我们称它为**非零弧**；如果 $f(u,v)<C(u,v)$，则该弧为**不饱和弧**。对于网络中任意一个弧（u,v），我们把边（u,v）称为**前向弧**并定义其可改进量为 $C(u,v)-f(u,v)$，把边（v,u）称为**后向弧**并定义它的可改进量为 $f(u,v)$。

把有向图的边改成无向边以后，对于 s 到 t 的一条简单路，如果路上的每条边 (u,v) 的可改进量均大于 0，则称这条路为一条增广路。所有边的可改进量的最小值为增广路的**可改进量**。

引入增广路的原因是我们有如下定理。

增广路定理：网络达到最大流当且仅当不存在增广路。

更进一步地，如果有一条可改进量为 k 的增广路，则沿着它增广（所有前向弧流量增加 k，所有后向弧的流量减少 k）以后流仍然是可行的（想一想，为什么？），且网络流量增加 k。这样，我们得到如下最大流的增广路算法。

增广路算法：从一个可行流开始不断地寻找可增广路，然后沿着它增广，直到它不存在。

这样的算法是增广路算法的直接应用，也是最常用的最大流算法之一。Ford 和 Fulkerson 于 1956 年提出了一种标号算法，它类似于 dijkstra 算法进行标号过程，并求出一条可增广路。由于算法比较简单，这里不作介绍。

为了方便实现，对于一个流 x，构造一个附加网络（也叫残量网络）$N(x)$，使得对原网

络的任意一条弧（u,v），在 $N(x)$中有：

- 加入边（u,v），权为 $u(u,v)=c(u,v)-f(u,v)$；
- 加入边（v,u），权为 $u(v,u)=f(u,v)$。

需要注意的是，其中有的边权为 0 边。它们不应该属于网络的一部分，但是保留它们可以保证弧的数量不随流而改变，便于我们分析。只要未加说明，我们所指的“边”并不包括这些权为 0 的边。这样，原网络中的增广路等价与 $N(x)$中一条 s-t 路。寻找增广路可以通过找 $N(x)$中的 s-t 来实现。Ford-Fulkerson 算法实际上就是随便找一条增广路。增广过程的复杂度为 $O(m)$，因此总的复杂度为 $O(mv)$，v 为最大流流量。

增广路算法有很多，我们只介绍最短增广路算法，即每次寻找包含弧的个数最少的增广路进行增广。可以证明，如果每次采用最短增广路，则最多只需要进行 $mn/2$ 次增广，因此算法的总时间复杂度取决于每次寻找最短增广路的平均时间。应该如何寻找最短增广路呢？如果用宽度优先的方法，则寻找时间为 $O(m)$，算法总时间复杂度为 $O(nm^2)$。下面，我们引入距离标号的概念，可以让平均寻找时间降低为 $O(n)$，则算法的总时间复杂度为 $O(n^2m)$。

给每个顶点 i 赋予一个非负整数值 $d(i)$来描述 i 到 t 的“距离”远近，称它为距离标号，如果它满足以下两个条件：

- $d(t)=0$，因为 t 离自己的距离为 0。
- 对残量网络 $N(x)$的任意一条弧(i,j)，$d(i)\leqslant d(j)+1$，即距离标号不能太大。

如果 $N(x)$中的弧(i,j)满足 $d(i)=d(j)+1$，我们称(i,j)是允许弧（admissible arc）。只由允许弧组成的路是允许路（admissible path）。显然，允许路是 $N(x)$中的最短增广路，但是最短增广路并不一定对应一条允许路，因此有时候需要修改距离标号，通过增加一些弧来得到允许路。我们先描述算法：

求最大流的使用距离标号的最短增广路算法。

算法思想：利用距离标号，寻找允许路来得到最短增广路。在找不到允许弧时让距离标号增加。

算法步骤：

```
procedure shortest_augument;
begin
  预处理，置初始流为 0
  计算距离函数 d(i)=i 到 t 最少走过的弧数目。
  当前结点 i:=s

  While d(s)<n do
  Begin
    if i 出发有允许弧(i,j) then              //沿着允许弧(i,j)前进一步走到 j
    begin
    记录 j 在允许路上的前驱结点 pre(j)=i; 当前结点 i=j;
    if 当前结点 i=t then                     // 找到增广路
```

```
      沿着允许路增广，修改残量网络，当前结点 i:=s          // 用 pre 函数退回到 s
      end
      else begin        // 没有允许弧，则通过修改距离标号创造一个新的允许弧
        if i 出发有弧(i,j) then
          d(i)=min{d(j)+1 | (i,j)在残量网络中}          // 这样从 i 出发就有允许弧了
        else            //  没有办法创造允许弧
          d(i)=n; 并且当 i≠s 时回退一步，即 i=pre(i);  // 禁止以后再考虑点 i
      end;
    End;
  end;
```

时间复杂度：$O(n^2m)$

需要说明的是，由于距离标号在算法中始终合法，所以算法可以求出最大流。注意到这样一个事实：如果某次迭代中 i 出发的某弧（i,j）不是允许弧，则在结点 i 修改标号之前的其余迭代中它也不可能是允许弧（因为 $d(i)$不变，而 $d(j)$不减，所以等号仍然不成立）。

这样，在查找允许弧的时候只需要从上一次找到的允许弧开始找。如果设立一个“当前弧”数据结构，则只有在修改距离标号时把当前弧重置为第一条弧，而平时只需要“接着上次”找。可以证明，该算法的时间复杂度为 $O(n^2m)$。

预流推进算法　我们来换一个思路：事先不保证收支平衡，模拟大量流从源流到汇，从输入大于输出的**预流**（preflow）得到满足平衡条件的可**行流**。在这个概念中，每个结点的输入大于等于输出。我们把结点 i 的输入与输出的差定义为结点 i 的**赢余** $e(i)$。$e(i)>0$ 且 $i\neq s,t$ 的顶点称为“活跃”顶点。

预流不满足流量平衡条件，但满足容量限制条件，所以只需要使用一种操作让它的流量趋于平衡，这种操作就是推进（push）。沿着弧（i，j）的一次推进把结点 i 的赢余尽可能多地推进到结点上。由于需要满足容量限制，推进量不能大于 $u(i,j)$；又因为推进的来源是结点 i 的赢余，所以推进量不能大于 $e(i)$，因此推进量 $p=\min\{e(i), u(i,j)\}$。一次推进的效果是：$e(i)$和 $u(i,j)$减少 p，$e(j)$增加 p。为了讨论方便，我们定义：如果 $p=u(i,j)$，由于推进后弧(i,j)饱和，则称这次推进为**饱和推进**，否则为**不饱和推进**。

初始时 s 出发的所有弧饱和，不和 s 邻接的点赢余为 0。当没有活跃结点时算法结束，可以证明此时的流是最大流。

最简单的预流推进算法是每次随便找一个活跃结点，随便找一条允许弧进行推进，在找不到允许弧的时候按照和前面完全一样的方法修改距离标号来创造一条允许弧。和前面一样，仍然需要使用“当前弧”结构，并在修改距离标号时重置当前弧为第一条弧。可以证明，这种简单的预流推进算法最多进行 nm 次饱和推进和 $O(n^2m)$次非饱和推进，总的时间复杂度为 $O(n^2m)$。

最高标号预流推进算法　事实上，可以定一些规则，让预流推进算法有更好的时间效率。下面介绍一种时间复杂度为 $O(n^2m^{1/2})$的算法：最高标号预流推进算法（highest-label preflow-push algorithm）。

首先，注意到这样的事实：在做完一次饱和推进后，结点 i 的赢余可能仍然不是 0。刚才的算法可能马上接着去找其他的活跃结点，而这里我们规定继续对它沿着其他允许弧推进，直到 $e(i)=0$ 或者找不到 i 出发的允许弧，需要对 i 重新标号为止。把对结点 i 的连续推进过程称作对结点 i 的一次检查。

刚才提到过，非饱和推进是预流推进算法的瓶颈。如何减少非饱和推进的次数呢？直观的想法是“让少数结点聚集大量的赢余，然后通过对这些结点的检查把非饱和推进变成一串连续的饱和推进”。因此，我们从距离标号大的结点开始推进，把流量慢慢的累积到距离标号小的结点，减少非饱和推进。

求最大流的最高标号预流推进算法。

算法思想：每次找距离标号最大的活跃结点进行检查，直到活跃结点不存在。

算法步骤：

```
procedure highest_label_preflow_push;
begin
  预处理，置初始流为 0
  让 s 出发的所有弧饱和。
  计算距离函数 d(i)=i 到 t 最少走过的弧数目。令 d(s)=n
  if 存在活跃结点 then
  begin
     i := 距离标号最大的活跃结点;
     repeat      //对结点 I 进行检查
        找一条 i 出发的允许弧进行推进;
     until e(I)=0 or 需要重新标号;
  end;
end;
```

算法复杂度：时间 $O(n^2m^{1/2})$

在具体实现的时候，用一组链表 LIST(k)来记录距离标号为 k 的所有结点并记录当前距离标号的上界 *level*。我们按 LIST(*level*), LIST(*level*−1)的顺序查找各个结点集合，直到找到一个结点 i，它具有最大距离标号。把 i 从 LIST 中删除并进行检查。在 i 被重新标号为 p 后，只需要把 i 插入到 LIST(p)中并让 *level*=p（因为 p 一定是最大距离标号），继续执行主循环。可以证明，查找 i 的步骤不会成为算法的瓶颈，该算法的时间复杂度为 $O(n^2m^{1/2})$。

最后，我们来考虑两种特殊网络中的最大流。如果每条弧的容量都为 1，则称它为**单位容量网络（unit capacity network）**。如果除 s,t 之外的每个结点要么只有一条入弧，要么只有一条出弧，则称它为**单位容量简单网络（unit capacity simple network）**。

首先考虑单位容量网络。

显然最大流不超过 $n-1$，因此用 ford-fulkerson 算法的时间复杂度为 $O(mn)$。用最短增广路算法的瓶颈操作是增广，它的时间复杂度也为 $O(mn)$。下面我们综合两种方法，分两个阶段求最大流。

阶段一：用最短增广路算法，当 $d(s)>d^*=\min\{[2n^{2/3}], [m^{1/2}]\}$时停止。

阶段二：用 Ford-Fulkerson 算法(或者用 BFS 找增广路)，求出最大流。

在阶段一中，由于 $d(s)\leqslant d^*$，故每个结点标号最多增加 d^*次。由于对任意一条弧来说，每次增广必让它饱和，故沿弧（i,j）的相邻两次增广必让 i 和 j 的标号都至少增加 2。因此，每条弧增广最多 $d^*/2$ 次。因此阶段一的时间复杂度为 $O(d^*m)$。

可以证明，阶段一结束后的流量 v^*和最大流 v 的差 $v^*-v\leqslant d^*$。因此阶段二的时间复杂度为 $O(d^*m)$。总的时间复杂度为 $O(d^*m)=O(\min\{n^{2/3}m, m^{3/2}\})$。

对于单位容量简单网络，和刚才的方法类似，取 $d^*=[n^{1/2}]$，则算法的时间复杂度为 $O(d^*m)=O(n^{1/2}m)$。

WEB 和流有关的问题是十分热门的，人们研究了它的很多变种，如反向流问题、对偶流问题和动态流问题就是其中三个比较重要的问题。这些问题需要线性规划的知识才能较好地理解，这里略去　还有一些有趣的问题，如 EqualFlow 和 MinimaxFlow，本书主页上都有介绍。

【例题 1】圆桌吃饭问题[1]

$N(N\leqslant 70)$个幼儿园的小朋友坐在一起吃饭。一共有 M（$M\leqslant 50$）张桌子可以使用，而每个小朋友都希望自己所在桌子上的小朋友都不是自己幼儿园的。给出每个幼儿园 i（$1\leqslant i\leqslant N$）的小朋友数目 m_i（不大于 100）和每张桌子 i（$1\leqslant i\leqslant M$）所能容纳的人数 c_i（不大于 100），请给出一种可行的方案。

【分析】

用 $d[i,j]$表示在桌子 j 上的幼儿园 i 的小朋友数目，显然 $d[i,j]$只能是 0 或者 1。而对于给定的 i，所有 $d[i,j]$之和应该等于 m_i，而对于给定的 j，所有 $d[i,j]$之和应该等于 c_i。和《奶牛的新年晚会》一样，我们设置一些幼儿园结点和一些桌子结点。每个幼儿园结点 K_i和桌子结点 T_j都连一条容量为 1 的边，设置新的源点 S 和汇点 T，对于所有 $1\leqslant i\leqslant N$，连一条 $S\rightarrow K_i$的边，容量为 m_i；对于所有 $1\leqslant i\leqslant M$，连一条 $T_i\rightarrow T$ 的边，容量为 c_i。则存在可行方案当且仅当最大流量为所有 m_i之和。点数$|V|=N+M+2$，边数$|E|=N+M+NM$。不妨取 $M=O(N)$，则$|V|=O(N)$，$|E|=O(N^2)$，以下比较几种常见的方法：

方案一：最短增广路算法。一般的最短增广路算法是 $O(|V||E|^2)=O(N^5)$，使用距离标号的方法是 $O(|V|^2|E|)=O(N^4)$。

方案二：预流推进算法。一般的预流推进算法和使用距离标号的最短增广路法一样，而最高标号预流推进算法是 $O(|V|^2|E|^{1/2})=O(N^3)$。

WEB 本题的模型中的容量虽然不是单位的，但是图确是非常特殊的——忽略 S 和 T 以后是一个完全二分图。能否利用这一性质来得到更好的算法呢？读者不妨考虑贪心算法，把每个小孩一个个放到合适的桌子边上去，看看你的方法是否正确，并给出证明或者相应的修改。本书主页上将继续讨论本题。

[1] 题目来源：UVA Problem Archive Online Contest

【例题 2】数字游戏[1]

有一列互不相同的数 A_1，A_2，…，A_n，你需要从前往后选出若干数，使得选出的数递增。例如，对于序列 1, 4, 2, 5, 3，选出 1, 2, 3 是合法的，但是选出 4, 2, 3 就是不合法的。

（1）一次最多能选出几个数？

（2）设第一问的答案为 M，并规定每次必须恰好取 M 个数，问最多能取几次？每个数至多能取一次。

【分析】

本题的第一问是求 LIS，在 1.5 节中已经讨论过了。我们用动态规划在 $O(n)$时间内求出了所有 $d[i]$，即以 i 开头的最长上升序列长度。由于要取多次，可把取数的过程看成一个流，那么每个数最多取一次就变成了点容量为 1 的情形。

按照 $d[i]$的大小把所有数分成层，其中第 k 层的任何数 p 的 $d[p]=k$，而要让这次恰好取 M 个数，下一个数 q 必须满足 $p<q$ 且 $d[q]=k-1$，也就是说 p 需要和第 $k-1$ 层的所有比它大的数连上一条边，由于本题中的容量在点上而不在边上，所以需要把一个点 i 拆成两个 i 和 i'，把边 $i\to j$ 变成 $i'\to j$。增加一个源 S 和汇 T，我们得到了一个单位容量网络，点有 $N+2$ 个，边的条数呢？粗略估计是 2 层时最大，为 $n^2/4$ 个。由于单位容量网络的最大流可以在 $O(\min\{n^{2/3}m, m^{3/2}\}=O(n^{8/3})$内求出，因此本题可以在 $O(n^{8/3})$内解决。

其实本题可以用贪心法来做，读者不妨试试。

【例题 3】混合图的欧拉回路[2]

给出一个 N 个点和 M 条边的混合图（即有的边是无向边，有的边是有向边）试求出它的一条欧拉回路，如果没有，输出无解信息。

【分析】

无向边只能经过一次，所以本题中不能把它拆成两条方向相反的有向边，而只能给每条无向边定向，经典的圈套圈算法不再有效（读者可以试着举出反例）。回忆有向图存在欧拉回路的充要条件：基图（把所有有向边变成无向边以后得到的图）连通，且每个点的出度等于入度。因此本题的关键就是：判断能否存在一个定向，使得每个结点的入度等于出度。“入度等于出度”让我们想到了网络流的平衡条件，因此本题可以用网络流来做。

本题和流有关，但是看起来不是去求什么最大流，而且连源和汇都没有，显然不能直接套用最大流的框架。考虑把每个点 i 拆成两个点 i 和 i'，增加弧 $i\to i'$，容量为正无穷。每条无向边(i,j)变成 $i'\to j$ 和 $j'\to i$，容量均为 1；每条有向边 $i\to j$ 变成 $i'\to j$，容量为 1。显然，问题转化为了求一个让所有形如 $i\to i'$ 的弧的流量都是结点 i 的度数的一半的流。如果能给这些弧的流量定一个下界 $b(i)=\text{degree}(i)/2$ 就好了，下面的方法满足了我们的要求。

我们增加一个源点 S 和汇点 T，并给每个结点 i 增加弧 $i\to T$ 和 $S\to i'$，容量均为 i 的度数的一半。这样，对于没有 S 和 T 的网络的任何一个大于下界的可行流，都可以把 $i\to i'$ 中 $b(i)$个单位的流量分给 $i\to T$ 和 $S\to i'$，剩下的流量仍存在于 $i\to i'$ 中，而且流量平衡条件仍满足（读者不难验证）。

---

[1] 题目来源：UVA Problem Archive Online Contest. 命题人：Monirul Hasan

[2] 题目来源：经典问题

这样就建立了一个有 $2N+2$ 个点和 $M+3N$ 条边的网络，由于流量恰好为 M（请读者证明），所以最短增广路算法的时间复杂度为 $O(MN)$（因为只增广 M 次）。

注意：本题中处理流量下界的方法可以推广到任意网络。本题还可以用原无向边任意定向的方法来做，请读者思考。

【例题 4】家园[①]

由于人类对自然的疯狂破坏，人们意识到在大约 2300 年之后，地球将不能再居住，于是在月球上建立了新的绿地，以便在需要时移民。令人意想不到的是，2177 年冬由于未知的原因，地球环境发生了连锁崩溃，人类必须在最短的时间内迁往月球。

现有 n 个太空站处于地球与月球之间（编号 1…n），m 艘公共交通太空船在其中来回穿梭，每个太空站 S_i 可容纳无限多的人，每艘太空船 p_i 只可容纳 H_{pi} 人。对于每一艘太空船 p_i，将周期性地停靠一系列的太空站（S_{i1}，S_{i2}，…，S_{ir}），如：（1，3，4）表示停靠太空站 1 3 4 1 3 4 1 3 4…。任一艘太空船从任一个太空站驶往另一个任意的太空站耗时为 1。人只能在太空船停靠太空站（或地球、月球）时上船或下船。初始时人全在地球上，太空船全在初始站（太空船 p_i 处于 S_{i1}），目标是让所有的人尽快地转移到月球上。

【分析】

仔细分析题意，不难看出本题具有一个流网络应有的各个要素：源——地球；汇——月球；中间点——太空站；边——往返于各个太空站之间的太空船；边的容量——太空船最多可容纳的人数。所不同的是，本题还多出一个时间因素，并且是给定流量，要求最短时间。我们面临着难题：一般的流网络无法表示两点间不同时刻的不同容量和流量。因此，就需要对流网络进行改造：拆点。

如果最短时间为 T，就把每个点 V_i（包括源和汇）拆成 $V_{i,0}$、$V_{i,1}$、…、$V_{i,T}$ 这 $T+1$ 个点。这样，太空船在 $t-1$ 时刻从 V_i 出发（$t\leqslant T$），t 时刻到达 V_j 的动作就可以用边 $V_{i,t-1}\rightarrow V_{j,t}$ 来表示，如果 $C(V_{i,t-1}, V_{j,t})=a$，$f(V_{i,t-1}, V_{j,t})=b$，那么就表示太空船可以容纳 a 个人，而此时正载有 b 个人；又因为每个太空站可以容纳无限的人停留，所以对于每个 $V_{i,t-1}$，都应有一条容量不限的边指向 $V_{i,t}$。

这样改造之后，源就是 $V_{0,0}$ 点，汇就是 $V_{-1,t}$ 点，从源到汇的流量就表示到时刻 T 为止从地球到月球最多可以输送的人数。

从小到大地尝试每个 T 值，第一个能够把 k 个人全部送达月球的时刻就是最优解。

在算法的实现上，采用宽度搜索找增广路径的算法（Edmonds-Karp 算法）来求最大流，并且，每当 T 值加 1 时，只需在原有流量的基础上进行增广，而无须重新求解。算法的空间复杂度为 $O(T\times n^2)$，算法的时间复杂度为 $O((n\times T)^3)$。求解的全过程中找不到增广路径的宽度搜索次数为 $O(T)$（因为一旦找不到增广路径，T 值就会加 1），找到增广路径的为 $O(|V|\times|E|)$，一次宽度搜索的复杂度为 $O(|E|)$，把它们综合起来，并代入 $|E|=(m+n)\times T$——m 表示 m 艘太空船某一时刻必然正从一个太空站驶往另一个，n 表示 $V_{i,t-1}$ 指向 $V_{i,t}$ 的边——和 $|V|=n\times T$，即得 $O(T^3\times n\times(m^2+n^2))$。

---

① 题目来源：CTSC 1999

WEB 本题根据时间把顶点拆成了多个。类似地，我们可以考虑“动态流”问题，它的两个基本问题是“给定时间限制，如何把尽量多的流从源点送到汇点？”和“给定流量，如何花费尽量少的时间从源点送到汇点？”。如果弧容量不随时间改变或者随时间为分段常数函数，则我们有强多项式算法，而不需要像本题这样对时间拆点。

5. 最小费用流：算法与应用举例

重新考虑残量网络。由于我们引入了“费用”因素，还需要定义残量网络中弧的费用。对于原网络中的弧（u,v），则残量网络中的 $w'(u,v)=w(u,v)$, $w'(v,u)=-w(u,v)$。我们有：

- **最小费用路算法**：在残量网络中求 s-t 的最小费用路，并沿着它增广，直到不存在 s-t 路。
- **消圈算法**：在最大流残量网络中不断找负权圈并沿着它进行增广，直到不存在这样的负权圈。

可以每次用 bellman_ford 求最小费用路（最小费用路算法）或者求负权圈（消圈算法），最小费用路算法最多迭代 v 次（v 为最大流量），故复杂度为 $O(mnv)$；消圈算法的时间复杂度上限是 $O(nm^2cw)$（其中 c 是容量最大值，w 是费用最大值，因为每次消圈后费用至少减少 1，而初始费用不超过 mcw），但实际上一般还是很快的。而且有一个按照特定次序消圈的算法时间复杂度为 $O(nm^2\log n)$，因此最小费用最大流问题是可以在多项式时间内解决的。

WEB 两个算法各有所长，事实上，它们有两个重要的改进算法：原始-对偶算法和网络单纯形法　原始-对偶算法的思想是通过定义结点势函数来利用最大流一次求多条最小费用路；网络单纯形法（network simplex algorithm）的思想是通过“可满足树（feasible tree）”数据结构和快速权更新来加快负圈寻找。本书主页上详细的介绍了网络单纯形法，它是实践中最有效的最小费用流算法之一

【例题 5】道路扩容[1]

C 城市的交通质量已经达到了令人难以忍受的地步。但是政府不打算增加过多的资金来建设高架公路或者立交桥。他们计划以有限的资金，对现在的交通网络进行扩容。

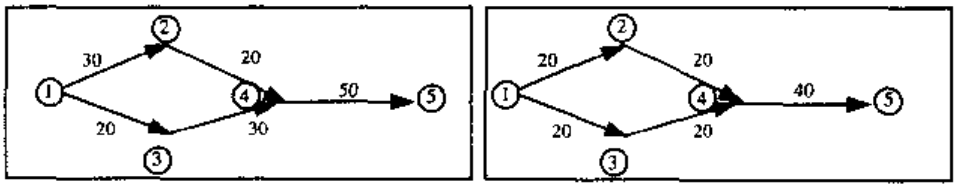
作为整项计划的一个子任务，政府的技术部门正在研究对以下给出的这种交通网络进行扩容的方案。

交通网络以邻接矩阵 $A_{n\times n}$ 形式给出：若 $A_{i,j}$ 不为 0 则表示路口 i 到路口 j 之间的公路（道路均为单向）所能承载的**最大车流量**，这条公路上的**实际车流量**不能超过 $A_{i,j}$；若 $A_{i,j}$ 为 0 则表示路口 i 到路口 j 之间没有公路。路口 1 是整个网络的**发点**。即所有的车辆从这里进入网络；路口 n 是整个网络的**收点**。也就是说，所有车辆从这里离开网络。如果不出现交通堵塞，则在除 1 和 n 外的任意一个路口 V，进入 V 的所有公路上的**实际车流量**总和应该等于离开 V 的所有公路上的**实际车流量**总和。

在保证无交通堵塞时，进入网络的总车流量存在一个最大值。例如图 2-60(a)的交通网络（边上的数字表示能承受的最大车流量），进入网络的总车流量最大为 40（此时每一条

[1] 题目来源：IOI2000 中国国家集训队原创题目，命题人：张力

边上的实际流量如图 2-60(b)所示）。如果要增加这个最大值，就要对现有的某些车流量已经饱和的公路进行扩容。



(a)　　(b)

图 2-60　道路扩容

工程部门已经分析了对每条公路扩容的代价，并给出了每条公路可承载的**最大车流量**增加一个单位时，所需消耗的资金，现在，请编程求出一个方案，表明对哪些公路进行扩容，分别进行多大的扩容，将使进入交通网络的总车流量的最大值增加 k 个单位（当然，不能出现交通堵塞），同时消耗的资金最少。

【分析】

与一般的网络流问题不同，本题是对网络最大流的一种逆向的探讨。通常的网络流问题是在确定网络的情况下求流的方案；而本题则是在流量确定的情况下，求网络扩充的方案。我们可以构造一个网络，每个弧的容量为∞，规定当容量不超过公路的实际容量时，费用为 0，一旦超过，单位流量的费用为该道路的扩容代价值。

如果可以构造出这样的网络，显然求一次最小费用最大流就可以了，但是这样的网络并不是常规的网络，因为它的费用是“分段”给出的。不过这不要紧，我们只需给一条公路建立两条弧，一条的容量为 $A_{i,j}$ 费用为 0；另一条容量为∞，费用为 $B_{i,j}$。由于我们求的是最小费用流，自然当流量不大于 $A_{i,j}$ 时流会走费用为 0 的弧，而当流量超过限制时，才会迫不得已而经过容量无限但有费用的弧。

练　习　题

2.5.11　开发计划[①]

J 公司准备制定一份未来一段时期内的科研与产品开发计划。公司的各部门都提出各自的计划项目，汇总成一张计划表。该计划表包含了许多项目，每个项目是一个研究课题、一个技术试验、一项市场调查或者是一个推销活动等。对于每个项目，计划表中都给出了它的预算，开支或者盈利。由于某些项目的进行必须依赖其他项目的成果，所以如果要进行这个项目的话，它所依赖的项目也是必不可少的。

例如，要推出一个新产品，先要分析其消费群体的需求，对产品的性能做出定位，然后设计产品的各个细节，包括形象包装等。

现在，假设你是 J 公司的总裁，你的目标是从这份计划项目表中挑选出一些项目，使

[①] 题目来源：IOI2000 中国国家集训队原创题目，命题人：江鹏

得你的公司能获得最大的利润。

2.5.12　因特网宽带[①]

在因特网上，计算机是相互连通的，两台计算机之间可能有多条信息连通路径。流通容量是指两台计算机之间单位时间内信息的最大流量。不同路径上的信息流通是可以同时进行的。例如，图 2-61 中有 4 台计算机，总共 5 条路径，每条路径都标有流通容量。从计算机 1 到计算机 4 的流通总容量是 25，因为路径 1-2-4 的容量为 10，路径 1-3-4 的容量为 10，路径 1-2-3-4 的容量为 5。

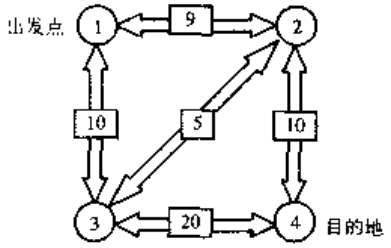


图 2-61　计算机网络

请编写一个程序，在给出所有计算机之间的路径和路径容量后求出两个给定结点之间的流通总容量（假设路径是双向的，且两方向流动的容量相同）。

2.5.13　出题者的烦恼[②]

我将要出一张总共有 100 道涉及 20 个类别试题的试卷，每个类别要求安排多少道试题是主考官决定的。我被提供了 1000 道备选题，我将从中选出 100 道不同的试题，将每道试题分在某一个类别下出成试卷。每道备选题都表明了它可以属于的类别，同一道题可能属于多个类别。我很烦恼，不知道应该怎样安排试题才符合主考官的要求，你能告诉我吗？

2.5.14　马戏团[③]

在一个城市里所有街道都是单行道，每条街道连接着两个不同的马戏团，且这个城市里的所有道路都是无圈的。你的任务是编一个程序，求出最少的人，使得他们遍历所有的马戏团且任何一个马戏团仅被一个人访问。每个人的出发点、目的地、路线都是任意的。

2.5.15　锦标赛[④]

在一个锦标赛中，任何两个人需要恰好比试一场，胜者得 2 分，败者得 0 分，比赛不可能出现平局。现在已经进行了一些比赛，并告诉你这些比赛的结果，问哪些人可能得到最后的冠军？这些人同第二名的分数差距最大可能是多少？

***2.5.16　机器人规划[⑤]

给出一棵树，有一个机器人要从某个点 s 走到另外一个点 t。树上的某些点上有一些可以移动的障碍物，在任何时刻每个点上不能同时有两个物体，即不能同时有机器人和障碍物，也不能同时有两个或更多的障碍物。每一步都可以把机器人或者一个障碍物移动到一

[①] 题目来源：ACM/ICPC World Finals 2000. 经典问题
[②] 题目来源：UVA Problem Archive Online Contest
[③] 题目来源：ACM/ICPC Regional Contest, SEERC
[④] 题目来源：经典问题
[⑤] 题目来源：经典问题

个相邻的空点上，如图 2-62 所示。

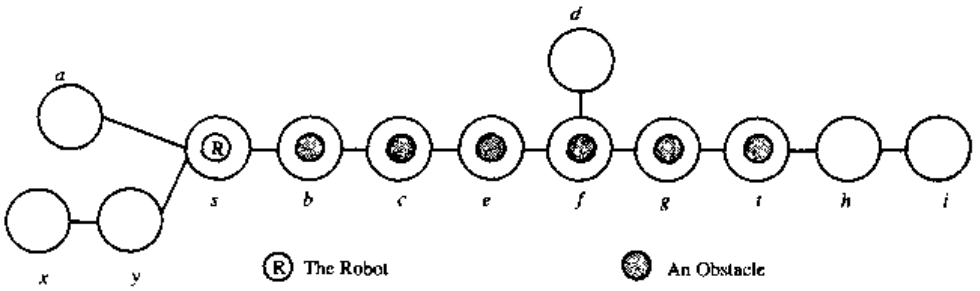


图 2-62　机器人问题举例

要求用最少的步数把机器人从点 s 移动到点 t。试给出该问题的一个多项式算法。

WEB 本题非常困难，是著名的 GMPIR 问题的简化版本。有兴趣的读者可以阅读本书主页。

2.5.4　二分图相关问题和模型

二分图是一类很重要的图，它的顶点可以分成两个集合 X 和 Y，图的所有边一定是有一个顶点属于集合 X，另一个顶点属于集合 Y。和二分图有关的问题中，要数匹配最为重要。

1．神奇的魔术师——二分图最大匹配

神奇的魔术师[①]

英俊潇洒的魔术师来到了舞台上，后面跟着他的助手——一位漂亮的小姐。魔术师首先表演了些例如从帽子里变出兔子，从助手的围巾中拿出一束束鲜花等简单的魔术，然后把助手锁到了一个不透明的密封箱子里。魔术师拿 N 张牌（N 是不大于 15 的奇数，所有牌都不一样），其中一位观众从中选择$(N+1)/2$ 张牌，剩下的牌被魔术师放到帽子里去并永远地消失了。魔术师选出一张牌交给那位观众，叫他给其他观众展示那张牌，然后藏到自己的口袋里。魔术师这才把助手放出来。她仔细的看了看剩下的$(N-1)/2$ 张牌，马上准确的说出了那位观众口袋里的那张牌，台下响起一片掌声。魔术师和他的助手是怎样做的呢？

附：常见问题解答[②]：

助手会心灵感应吗？答：不会。她是个虽然长得漂亮，但是没有任何超能力的凡人。

助手可以看穿箱子吗？答：不可以。参见问题 1 的解答。

看了这个问题，你觉得这个魔术神奇吗？如果不相信助手有超能力，你就需要做一番推理，看看助手是怎么知道的。

[①] 题目来源：Internet Problem Solving Contest

[②] 此为原题中设置的小幽默，这里为了忠实原文，一起翻译了过来。

首先，把魔术的过程写得简单一些。即：N 张牌（观众）→$(N+1)/2$ 张牌（魔术师）→$(N-1)/2$ 张牌。关于助手，除了介绍之外题目只说："她仔细的看了看剩下的$(N-1)/2$ 张牌，马上准确的说出了那位观众口袋里的那张牌……"显然，助手推理的依据只有那$(N-1)/2$ 张牌，即那$(N-1)/2$ 张牌和观众口袋里的那张牌一定存在着某种联系。准确地说，$(N-1)/2$ 张牌决定了观众口袋里的牌。观众口袋里的牌是魔术师选出来的，因此魔术师实际上是通过选牌，让助手可以通过剩下的$(N-1)/2$ 张牌确定去掉的那张，从而惟一确定开始的$(N+1)/2$ 张牌。即：剩下的牌（它们的集合称为 Left）和开始的（它们的集合称为 Orig）应该存在一个一一对应关系。由于两个集合的元素个数 $C(n, (n-1)/2) = C(n, (n+1)/2)$，所以这个一一对应关系是有希望找到的。

为了找到这个一一对应关系，先介绍二分图。二分图（bi-partite graph）是这样一个图，它的顶点可以分为两个集合 X 和 Y。所有的边关联的两个顶点中，恰好一个属于集合 X，一个属于集合 Y。也就是说：同类结点不邻接。图的一个匹配（matching）是一些边的集合，任意两条边没有公共端点。图中包含边数最多的匹配称为图的**最大匹配**。如果所有点都在匹配边上，称这个最大匹配是完美匹配（perfect matching）。

把 Left 中的每个元素（即一种可能的剩余牌集合）看成一个 X 结点，Orig 中的每个元素看成一个 Y 结点；如果某 X 结点 u 可以通过某 Y 结点 v 中去掉一张牌得到（这样魔术师才可以把 v 变成 u），则连上边（u,v），就得到了一个二分图 G，且我们所求的"一一对应关系"就是图 G 的完美匹配。

我们举 n=5 的例子。因为每张牌都不相同，我们把所有牌编号为 1,2,3,4,5，则：

所有$(n+1)/2$ 张牌的情况有 $C(5,3)$=10 种，它们是：

Orig={ (1,2,3), (1,2,4), (1,2,5), (1,3,4), (1,3,5), (1,4,5), (2,3,4), (2,3,5), (2,4,5), (3,4,5)}

所有$(n-1)/2$ 张牌的情况有 $C(5,2)$=10 种，它们是：

Left={(1,2), (1,3), (1,4), (1,5), (2,3), (2,4), (2,5), (3,4), (3,5), (4,5)}

这 10 个 X 结点和 10 个 Y 结点组成的图 G 如图 2-63(a)所示，完美匹配如图 2-63(b)所示。

只要记住这张图，不管魔术师面对的是哪三张牌，他总可以去掉一张，剩下那三张牌所匹配到的两张牌（连边的方式使得这总是可以做到的）。而只要助手也记住了同一个图，不管她面对的是哪两张牌，她都可以知道这是哪三张牌变来的，从而"猜"出观众口袋里的牌是哪张。

注意：虽然在这里用二分图匹配解决了此题，但是此题实际上是有构造方法的。这个方法留给大家去寻找。提示：假设取出的 $m=(n+1)/2$ 个数是 a_1，a_2，…，a_m，那么如果把第$(a_1+a_2+\cdots+a_m) \bmod m + 1$ 个数去掉的话会如何？

求二分图的最大匹配有一种匈牙利算法，它的时间复杂度为 $O(nm)$，其思想是用宽度优先搜索来找可增广路（和 floodfill 算法类似）。事实上我们可以做得更好，使二分图匹配问题转化为单位容量简单网络的最大流问题：只需要在二分图的基础上，加入源点 s 和汇点 t，让 s 与每个 X 结点连一条边，每个 Y 结点和 t 连一条边，所有弧的容量为 1。则最大匹配就是新网络的最大流，饱和弧对应着匹配边。这样，用前面介绍的 $O(mn^{1/2})$算法即

可解决问题，该算法也叫做 Hopcroft 算法。

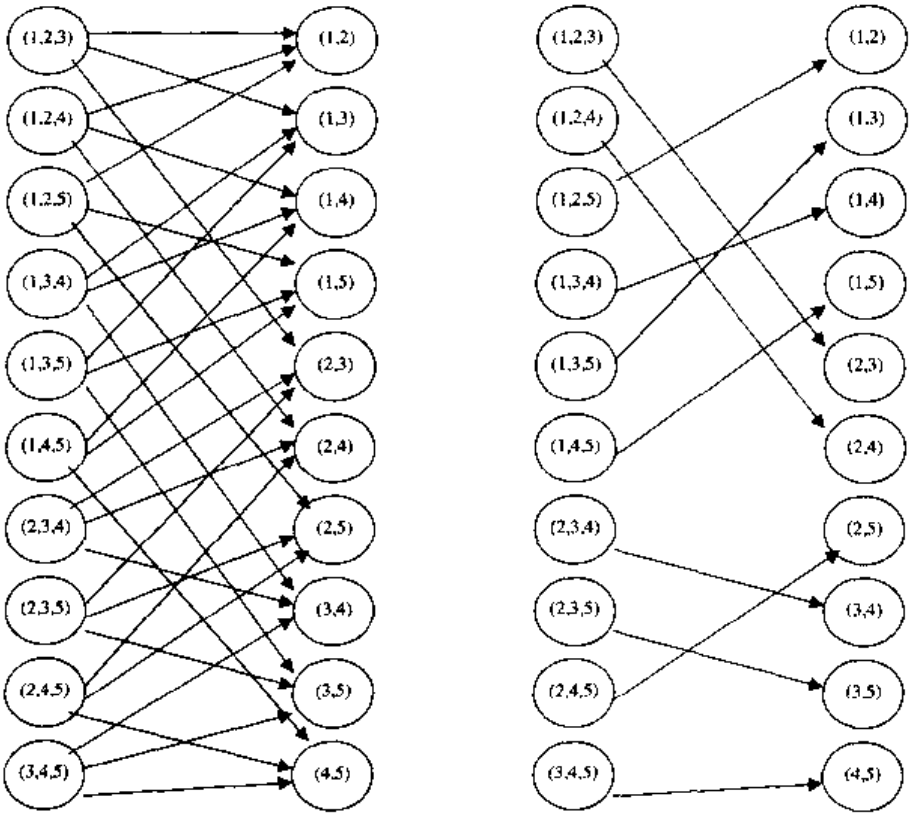


图 2-63　n=5 的图 G 和魔术方案

如果 X 结点和 Y 结点一样多，而且所有点的度数都相等，则一定存在一个完美匹配，即所有点都被匹配上。这个结论有时会导致一些构造算法。

2．任务安排——二分图的最小覆盖数

任 务 安 排[①]

有两台机器 A 和 B 及 N 个需要运行的任务。每台机器有 M 种不同的模式，而每个任务 i 都恰好在一台机器上运行。如果它在机器 A 上运行，则机器 A 需要设置为模式 a_i，如果它在机器 B 上运行，则机器 B 需要设置为模式 b_i。每台机器上的任务可以按照任意顺序执行，但是每台机器每转换一次模式需要重新启动一次。请合理为每个任务安排一台机器并合理安排顺序，使得机器重启次数尽量少。

【分析】

本题的建模需要一点技巧。显然，机器重启次数是两台机器需要使用的不同的模式个数。但是如果把每个任务看成一个 X 结点，把每台机器的每个模式看成一个 Y 结点，则此模型没有任何意义。应该把每个任务看成一条边，即 A 机器的每个模式看成一个 X 结点，

[①] 题目来源：ACM/ICPC Regional Conest Beijing 2002

B 机器的每个模式看成一个 Y 结点，任务 i 为边 (a_i,b_i)。本题即为求最少的点让每条边都至少和其中的一个点关联。有以下结论：这个最少点数（称为覆盖数）就是最大匹配数 M，证明如下：

（1）M 个是足够的。只需要让它们覆盖最大匹配的 M 条边，则其它边一定被覆盖（如果有边 e 不被覆盖，把 e 加入后得到一个更大的匹配）

（2）M 个是必需的。仅考虑形成最大匹配的这 M 条边，由于它们两两个无公共点，因此至少需要 M 个点才能把它们覆盖。

3．棋盘上的骑士——二分图的独立数

棋盘上的骑士[①]

一个 $n\times n(n\leqslant 100)$的棋盘上，有一些单位小方格不能放置骑士。棋盘上有若干骑士，任一个骑士不在其他骑士的攻击范围内。请告诉我棋盘上最多能有几个骑士。骑士攻击范围如图 2-64 所示（S 是骑士的位置，X 表示攻击范围）。

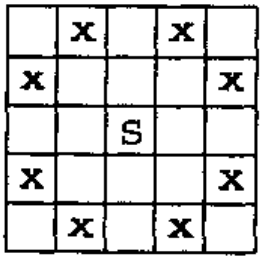


图 2-64　棋盘和骑士攻击范围

【分析】

如果把剩余单位小方格抽象成图的顶点，并在相互属于对方攻击范围的顶点对间连线，那么很明显，由于跳马特殊的“日”字形路线，使得只有黑色与白色小方格间才有连线，所以此题的模型是一个二分图 G。而我们的目标就是找出 G 的最大独立集。为方便叙述，称 G 的顶点集为 V，边集为 E，最大独立点集为 U，最大匹配边集为 M，M 覆盖的顶点集为 EM，下面给出$|U|=|V|-|M|$的证明：

（1）由于 U 中任两顶点之间无边，所以 M 中的每条边都至少含有一个集和 $V-U$ 内的顶点，且这些顶点各不相同，因此$|M|\leqslant|V|-|U|$。

（2）假设对于 M 中的边（x,y）存在边（a,x）和（y,b），使得 $a,b\in E-EM$，那么分如下两种情况

① a，b 有边，那么 $M'=M+(a,b)$是图 G 的合法匹配且$|M'|>|M|$，与 M 是最大匹配矛盾；

② a，b 无边，那么 $a\to x\to y\to b$ 构成了增广路径，亦与 M 是最大匹配矛盾。

所以，M 中的边至少有一个顶点独立于 E-EM。假设 M 中的有两条边 (x_1,y_1) 和 (x_2,y_2)，存在边(a,x_1), (y_2,b), (y_1,x_2) $(a,b\in E\text{-}EM)$，那么 $a\to x_1\to y_1\to x_2\to y_2\to b$ 也构成了增广路径，这与 M 是最大匹配矛盾。所以假设不成立，一定可以从 M 中每条边上各取一个顶点，将它们加入 E-EM 所得集合仍然是独立集。由此知，$|U|\geqslant|V|-|EM|+|M|=|V|-|M|$。

[①] 题目来源：Baltic Olympiad in Informatics, 2001

由（1）和（2）得出结论：$|U|=|V|-|M|$。

4. 丘比特的烦恼——带权二分图的最佳匹配

如果二分图是加权的，那么我们可以求出完美匹配中权和最大的，这个匹配称为二分图的最佳匹配或者最大权匹配。最佳匹配的应用也很广泛，请看下面的例子：

丘比特的烦恼[①]

随着社会的不断发展，人与人之间的感情越来越功利化。最近，爱神丘比特发现，爱情也已不再是完全纯洁的了。这使得丘比特很是苦恼，他越来越难找到合适的男女，并向他们射去丘比特之箭。于是丘比特千里迢迢远赴中国，找到了掌管东方人爱情的神——月下老人，向他求教。

月下老人告诉丘比特，纯洁的爱情并不是不存在，而是他没有找到。在东方，人们讲究的是缘分。月下老人只要做一男一女两个泥人，在他们之间连上一条红线，那么它们所代表的人就会相爱——无论他们身处何地。而丘比特的爱情之箭只能射中两个距离相当近的人，选择的范围自然就小了很多，不能找到真正的有缘人。

丘比特听了月下老人的解释，茅塞顿开，回去之后用了人间的最新科技改造了自己的弓箭，使得丘比特之箭的射程大大增加。这样，射中有缘人的机会也增加了不少。

情人节的午夜零时，丘比特开始了自己的工作。他选择了一组数目相等的男女，感应到他们互相之间的缘分大小，并依此射出了神箭，使他们产生爱意。他希望能选择最好的方法，使被他选择的每一个人被射中一次，且每一对被射中的人之间的缘分的和最大。

当然，无论丘比特怎么改造自己的弓箭，总还是存在缺陷的。首先，弓箭的射程尽管增大了，但毕竟还是有限的，不能像月下老人那样，做到“千里姻缘一线牵”。其次，无论怎么改造，箭的轨迹终归只能是一条直线，也就是说，如果两个人之间的连线段上有别人，那么不可向他们射出丘比特之箭，否则，按月下老人的话，就是“乱点鸳鸯谱”了。

作为一个凡人，你的任务是运用先进的计算机为丘比特找到最佳的方案。

【分析】

人只有男人和女人两类，而任何一类人都只能和他（她）的异性相配，这就为我们提供了现成的二分图。很明显，二分图的左边是 n 个男人，右边是 n 个女人。首先用几何方法判断是否某个男人和某个女人之间距离不超过弓箭射程且相连线端上（注意是线段，不是直线）没有第三者，如果是，那么他们之间有边，边的容量为 1，费用为互相之间的缘分大小。边建立完毕后，剩余的事情就是最优二分图匹配了。

5. 魔术球问题——最小路径覆盖

魔术球问题[②]

有 N 根柱子，现在有任意正整数编号的球各一个，请你把尽量多的球放入这 N 根柱子中，满足：

---

① 题目来源：CTSC 2000. 命题人：杨帆

② 题目来源：OIBH Reminiscene Programming Contest. 命题人：刘汝佳

① 放入球的顺序必须是 1,2,3,…，且每次只能在某根柱子的最上面放球；

② 同一根柱子中，相邻两个球的编号和为完全平方数。

请问，最多能放多少个球在这 N 根柱子上？例如，4 根柱子可以放 11 个球，如图 2-65 所示。

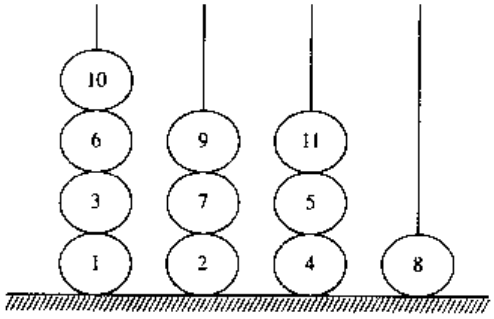


图 2-65　4 根柱子的最优解

【分析】

把每个球看做一个顶点，如果两个球 i 和 j（$i<j$）的编号和 $i+j$ 为平方数，那么把两个顶点连成一条有向边 $i\to j$。我们首先解决这样一个判定问题：给定球总数 n 和柱子数 m，能否把所有球放到柱子上？即需要解决如下所示的一个问题。

最小路径覆盖问题：用尽量少的不相交简单路径覆盖有向无环图 G 的所有顶点。

我们给这个问题建立一个二分图模型。把所有顶点 i 拆为两个：X 结点 i 和 Y 结点 i'，如果图 G 中存在有向边 $i\to j$，则在二分图中引入边 $i\to j'$，设二分图的最大匹配数为 m，则结果就是 $n-m$。这个结果不难理解，因为匹配和路径覆盖是一一对应的。路径覆盖中的每条简单路径除了最后一个结点之外都有惟一的后继和它对应（即匹配结点），因此匹配边数就是非路径结尾的结点数。因此匹配数达到最大时，非路径结尾的结点数达到最大，故路径结尾结点数目最少，即路径数最少。

值得一提的是，利用二分图匹配求解后可以发现：n 取任何可以验证的所有值时，最大值均为 $f(n)=\left\lfloor \frac{n^2}{2}+n-\frac{1}{2} \right\rfloor$，而且对于任意 n 来说，可以用贪心法构造出一个结果为 $f(n)$ 的解，故 $f(n)$ 是答案是下界（当然，如果把“完全平方数”改为质数，匹配模型仍然有效，但是贪心法就不再适用了）。我们猜想 $f(n)$ 就是正确答案，可以证明这个猜想是正确的，证明也不复杂：只需根据 n 的奇偶性分类讨论，考虑最大的 $n+1$ 个球中最小的两球体积 min 和最大两球的体积和 max。由于 min 和 max 夹在两个相邻完全平方数中间（请读者计算一下以验证这个结论），因此这 $n+1$ 个球的任意两个都不能在同一根柱子上，矛盾。

6. 综合应用举例

【例题 1】皇家卫士[1]

从前有位国王，有座由 $N\times M$（N，$M\leqslant 200$）个单位小方格组成的矩形城堡。城堡中有些格子是墙，有些是陷阱，其余的是空地，如图 2-66 所示。为加强保卫，国王打算在城堡

[1] 题目来源：CEOI 2002

中安排尽可能多的卫士，每个卫士占领一个单位小方格，当然此单位小方格必须是空地。因为安全原因，卫士与卫士之间是相互不联系的，所以如果某个卫士朝东或者朝西或者朝南或者朝北观望时，看见了另一个卫士，也就是说两个卫士处在同一水平或垂直线上，且他们之间没有墙阻碍视线（陷阱并不阻碍视线），那么他们会把对方当成入侵的敌人而举枪消灭。这种情况是国王不愿意发生的。现在给你城堡的地图，请你安排尽可能多的卫士。

(a)　　(b)

图 2-66　城堡举例

图2-66（a）是城堡地图，其中■是墙，◎是陷阱，□是空地。图2-66(b)是解答，其中◎是卫士。

【分析】

为方便判断，我们在皇宫的外围增设一圈围墙。称一段无法再向左右扩展的连续非墙格子为行线段，一段无法再向上下扩展的连续非墙格子为列线段。

构建一个二分图，图的左边顶点是所有的行线段，右边顶点是所有的列线段。如果某条行线段与某条列线段的交点存在且是空地，那么在该两顶点间连一条边。

如果在某个小方格 k 内设置皇家卫士，那么 k 必然是某条行线段 i 和某条列线段 j 的交点，称 i 与 j 相交于 k。易知，任意一个行或列线段可以设置最多一名卫士，所以任意一条线段上最多存在一个交点，任意一条行（列）线段最多和一条列（行）线段相交。这就构成了一一对应的匹配关系。而我们的任务只是求二分图的最大匹配。

用前面学习的二分图匹配算法。它的复杂度是 $O(|V|^{1/2}|E|)$，似乎会超时。其实不用担心。我们可以找出图的所有连通块，然后分块处理。皇宫的墙设置得越多，连通块的数量就越多，连通块的规模就越小；反之，皇宫的墙设置得越少，行或列线段的数量就越少，连通块的规模也大不起来。初步估计，皇宫如图 2-67 构造时，连通块的规模会达到或逼近最大。

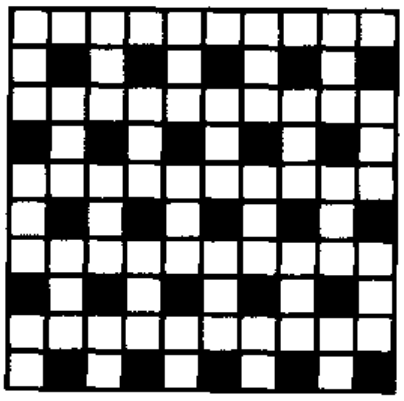


图 2-67　最坏情况的城堡

这时 $|V|=O(|E|)=O(nm)$，则复杂度 $O(|V|^{1/2}|E|)=O((nm)^{3/2})$。但这种最大规模出现的概率极

小，同时陷阱的存在会减少边的数量，所以此算法是可行的。读者可以试着更精确地估计一下运行时间的上限。

【例题 2】固定分区的内存管理[1]

早期的多程序操作系统常使用一种内存分区的技术，即把所有的可用内存划分成一些大小固定的区域，不同的区域一般大小不同，而所有区域的大小之和为可用内存的大小。给定一些程序，操作系统需要给每个程序分配一个区域，使得它们可以同时执行。可是每个程序的运行时间可能和它所占有的内存区域大小有关，因此这个任务并不容易。

你需要写一个程序计算最优的内存分配策略，即给定m（$m\leqslant 10$）个区域的大小和n（$n\leqslant 50$）个程序在各种内存环境下的运行时间，找出一个调度方案，使得平均回转时间尽量小。具体来说，你需要给每个程序分配一个区域，使得没有两个程序在同一时间运行在同一个内存区域中，而所有程序分配的区域大小都不小于该程序的最低内存需求。

一个程序的回转时间为它结束时刻和它请求执行的时刻（对于所有程序来说，这个时刻都是 0）的差。平均回转时间是所有程序的回转时间的平均值。如果有多个最优调度方案，任意输出一组解即可。

【分析】

本题是 2001 年国际大学生程序设计竞赛总决赛的一道题目，比赛时无人做出此题。因为虽然“给每个程序分配一个内存区域”有点像是匹配，可是同一个内存区域可以有多个程序，而且在同一区域中，程序的执行顺序也影响平均回转时间，情况似乎很复杂。

为了解决本题，先来看一个内存区域的情况。假设在这个内存区域按顺序执行的 k 个程序的运行时间分别为 t_1，t_2，t_3，…，t_k，那么第 i 个程序的回转时间为 $r_i=t_1+t_2+\cdots+t_i$。所有程序的回转时间为 $t_1\times k+t_2\times(k-1)+t_3\times(k-2)+\cdots+t_{k-1}\times 2+t_k$。这样，如果程序 i 是内存区域 j 的倒数第 p 个执行的程序，那么它对于总的回转时间的增加量为 $t[i,j]\times p$，其中 $t[i,j]$为程序 i 在内存区域 j 中的运行时间。这样，解法立刻产生了：构造一个二分图 G，X 结点为 n 个程序，Y 结点为 $n\times m$ 个“位置”，其中位置（j,p）表示第 j 个内存区域的倒数第 p 个执行的程序。每个 X 结点 i 和 Y 结点（j,p）连有一条权为 $t[i,j]\times p$ 的边，然后求最小权匹配。

为了求最小权匹配，我们用最小费用最大流模型来解决。增加一个源点 s 和一个汇点 t，对于每个 X 结点 i 增加一条容量为 1，费用为 0 的弧 $s\rightarrow i$；对于每个 Y 结点 j 增加一条容量为 1，费用为 0 的弧 $j\rightarrow t$，把原图中的边 $i\rightarrow j$ 改为容量为 1 费用为 $w(i,j)$的弧 $i\rightarrow j$，则得到的费用网络的最小费用最大流对应原二分图的最小权匹配。

如果用最小费用路算法求解最小费用流问题，则算法复杂度为 $O(|V||E|\times v)$，而最大流量 $v=O(|V|)$，故算法复杂度为 $O(|E||V|^2)$。在本题中$|V|=O(nm)$，$|E|=O(n^2m)$，因此总的时间复杂度为 $O(n^4m^3)$。但实际情况远达不到 $O(n^4m^3)$，还是可以接受的。求二分图最小权匹配的有时间复杂度为 $O(|V|^3)$的算法，限于篇幅这里不再叙述，有兴趣的读者可以阅读专门书籍。

【例题 3】玩具兵[2]

小明的爸爸给他买了一盒玩具兵，其中有 K 个步兵，K 个骑兵和一个天兵，个个高大

---

[1] 题目来源：ACM/ICPC World Finals 2001

[2] 题目来源：CTSC2002. 命题人：刘汝佳

威猛，形象逼真。盒子里还有一个 $M\times N$ 棋盘，每个格子(i,j)都有一个高度 H_{ij}，并且大得足以容纳所有的玩具兵。小明把所有的玩具兵都放到棋盘上去，突然想到了一种很有趣的玩法：任意挑选 T 个不同的格子，并给每个格子 i 规定一个重要值 R_i，游戏的目标就是每次沿东南西北之一的方向把一个玩具兵移动到其相邻的格子中（但不能移动到棋盘外面去），最终使得每个挑选出的格子 i 上恰好有 R_i 个玩具兵。小明希望所有的玩具兵都在某个选定的格子中，因此他总是使选出的 T 个格子的重要值之和等于玩具兵的个数。为了增加难度，小明给玩具兵们的移动方式做了一些规定：

- 步兵只会往高处爬，因此如果两个格子 A 和 B 相邻，当且仅当格子 A 的高度小于或等于 B，步兵才可以从 A 移动到 B。
- 骑兵只会往低处跳，因此如果两个格子 A 和 B 相邻，当且仅当格子 A 的高度大于或等于 B，骑兵才可以从 A 移动到 B。
- 天兵技术全面，移动不受任何限制。

可是没玩几次，小明就发现这个游戏太难了，他常常玩了好半天也达不到目的。于是，他设计了一种“超能力”，每使用一次超能力的时候，虽然不能移动任何一个玩具兵，但可对它们进行任意多次交换操作，每次交换两个玩具兵。等这次超能力使用完后又可和平常一样继续移动这些玩具兵。借助强大的超能力，这个游戏是容易玩通的，但是怎样才能让使用超能力的次数最少呢？

【分析】

本题是笔者为 2002 年中国国家队选拔赛精心设计的一道难题。我们很容易想到匹配，不过模型的建立是需要认真思考的。首先，应当看出，天兵可以随意移动，因此只需要让其他 $2K$ 个兵（把它们叫做常规兵）到达目的地就可以了。又由于同类的兵没有区别，每次重新排列位置可以等价的看成兵在变化种类。为了解题方便，我们大胆地把题目改一下魔法的效果：所有常规兵的种类变化，并让其中一个常规兵直接到达一个目的地。

这样改是否合理呢？答案是肯定的。首先，由于两类常规兵一样多，因此只要一对一地交换，是一定可以让所有常规兵同时进行种类变化的。其次，如果某常规兵不需要变化种类，那么让它按照原定路线接着走，直到需要变化种类为止；如果已经到达目的地，那么强制它变化种类也没有什么影响。由于天兵随意移动，只要把它移动到一个目的地，再和某常规兵交换就可以把它送到目的地了。

这样，问题就单纯多了。我们可以构造一个二分图 G，每个 X 结点是一个常规兵，每个 Y 结点是一个目标格子，重要性为 r 的格子分为 r 个 Y 结点。对于每一对 XY 结点(u,v)，如果常规兵 u 最少需要经过 w 次种类转化而“到达”格子 v，那么连一条(u,v)边，权为 w。所有权可以通过宽度优先搜索来求出。得到这个图以后，图的一个匹配就完全地代表了一个方案。我们每次都让所有兵走到需要变种类的位置上，然后所有常规兵一起变种类。这样，如果常规兵 u 匹配到格子 v，那么它可以在 w_{uv} 次交换后到达目的地。

我们从小到大枚举魔法使用次数 P，然后依次判断能否达成目标。由于限制了魔法使用次数，因此权大于 P 的边全部得去掉。由于使用了 P 次魔法，可以让 P 个常规兵直接到达目标，因此只需要 $2K-P$ 个常规兵匹配上格子就可以了——这些 $2K-P$ 个兵匹配边的长度都不超过 P，因此可以在不超过 P 次交换后到达目的地；而剩下的 P 个兵每次使用魔法都

将有一个被天兵送到达目的地。由于 P 不会超过 $2K$（每次送一个常规兵去目的地，也只需要 $2K$ 次魔法），二分图的结点有 $2k$ 个，边不超过 k^2 个，因此核心部分的复杂度为 $O(k)\times O(k^2\times(2k)^{1/2})=O(k^{3.5})$。综上所述，本题的算法步骤为：

（1）预处理，用 BFS 求出每个常规兵到每个目标格子需要进行的最少种类变化次数；

（2）按刚才介绍的方法建立二分图 G；

（3）从 0 开始枚举 P，在 G 的基础上去掉权大于 P 的边得到图 G'，如果 G' 的最大匹配数不小于 $2K-P$，那么 P 就是所求答案。

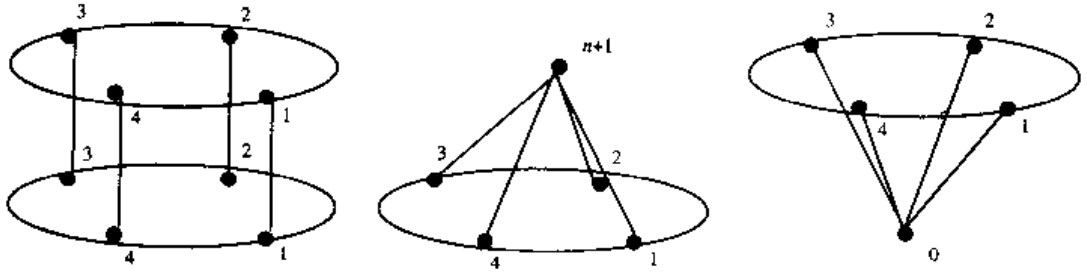
【例题 4】千年盛典[1]

新千年到来之际，世界各地的人们都在准备着自己独特的庆祝方式，思尔彼加岛的居民也不例外。千年盛典在一个巨大的椭球型宫殿里进行，宫殿从底到顶分为 n 层。每层都是一个圆，圆周上等距的放置个 m（m 为偶数）个大理石柱，按照逆时针编号为 1，2，3，…，m。显然，每个石柱都有两个相邻石柱。

最底层（可以看作是第 0 层）没有石柱，但它设有一个祭坛，那是盛典仪式的起点和终点。最顶层（可以看作是第 n+1 层）也没有石柱，而是悬挂着一个巨大的圣火炬。

第 2~n-1 层的每根石柱（$1\leqslant j\leqslant m$）出发有若干绳索与其他石柱连接：其中两根连到同层的两个相邻石柱，称为"同层绳索"（不难想象，每层的同层绳索形成一个环）；k 根连到第 i-1 层的 k 个不同石柱，另 k 根连到第 i+1 层的 k 个不同的石柱，称为"层间绳索"，如图 2-68(a)所示。出于美的考虑，层间绳索两端的石柱编号同奇偶。k 为奇数。

从祭坛出发有 m 根绳索分别连到第 1 层的每个不同的石柱，称为"底层绳索"，而从圣火炬出发也有 m 根绳索连到第 n 层的每个不同的石柱，称为"顶层绳索"，如图 2-68(b)和图 2-68(c)所示。注意绳索是没有方向的，因此一根绳索同时算作是从两根石柱出发的。这样，层间绳索有$(n-1)\times m\times k$ 条，同层绳索有 nm 条，底层和顶层绳索各 m 条，一共有 $m(nk-k+n+2)$条。



(a) 同层和层间绳索　　(b) 底层绳索　　(c) 顶层绳索

（以上三图中，n=2, m=4, k=1）

图 2-68　三种绳索示例

盛典的第一项是点火仪式。仪式需要岛上最身强力壮的勇士克里斯和克里特相互配合，从祭坛出发沿着绳索走到火炬，祈祷并点燃圣火，再沿着绳索回到祭坛。绳索是经过精心

[1] 题目来源：CTSC2003. 命题人：刘汝佳

安排的，数目和位置到恰到好处，没有多余的，也不会不够用，因此，二人必须分别经过每条绳索一次且仅一次。两人的线路不会相互影响，因此我们假定克里斯先走，等他返回祭坛后才轮到克里特。

绳索是特制的。仪式开始时，它们如水晶般透明，但当某人第一次到达某一根石柱或者第一次到达圣火炬时（由于祭坛是整个仪式的起点，所以第一次回到祭坛时不会引起任何绳索的颜色变化），他刚刚经过的绳子会自动变色。变色的结果取决于完成这次动作的人和变色的绳索种类。如下表所示：

动作完成者	同层绳索	其他绳索
克里斯	金黄	深灰
克里特	银白	浅灰

需要特别注意的是，绳索不能两次变色。换句话说，当一个勇士站在任何一条已经变过色（不管是被谁变的）的绳索面前，如果绳索的另一端没有被他走过，他是不能沿着这条绳索走的。

如果在勇士回到祭坛时，每一层的石柱之间都能够呈现出 $m/2$ 根金线和 $m/2$ 根银线交叉相间（金线→银线→金线→银线→…→金线→银线）的美丽图案，那该是多么壮观和激动人心啊！勇士们该怎么做，才能让它成为现实呢？

【分析】

本题是笔者为 2003 年中国国家队选拔赛精心设计的一道难题。本题的条件很多很复杂，让我们一点一点来分析。把每个石柱，祭坛和火炬都看成点，每两个间有边相连当且仅当二者有绳索连接，则得到了一个有 $v=nm+2$ 个结点的图。两个勇士的路线都是欧拉回路，那么变色的绳索有什么特殊性质呢？

首先，对于每个勇士来说，他每次第一次到达某个石柱或火炬都会引起一次变色，因此一共恰好有 $v-1$ 条绳索被他变色。而容易证明任意时刻所有被他自己变色的绳索是连通的，而且除了祭坛的入度为 0 外，其他点的入度均为 1，因此这些绳索恰好形成了一棵以祭坛为根的外向树。

反过来，对于任意一棵以祭坛为根的外向树 T，我们都可以找到一条有向欧拉回路，使得沿着它经过以后变色的索恰好形成树 T。方法很简单，只需要把所有边反向，外向树 T 就成了内向树 T'。对于每个结点来说，规定变色边一定要是它的出弧中最后一个经过的（因为边反向了，因此第一次到达点时变色改成了最后一次离开点时变色），则最后一定可以走出一条欧拉回路，因此只需把回路再过来即可。

有没有可能最后走不出欧拉回路呢？不可能。由于路是“一进一出”的，因此到达某个点时，如果它除变色边外的所有其他出弧都走完了，那么这个点的所有入弧一定也走完了。因此，我们是从叶到根逐步构造变色树，始终保持整个图的连通性，故构造过程一定可以顺利结束。这个思想利用了欧拉回路和外向树的映射关系，用它可以直接得到 2.4 节中介绍的欧拉回路计数公式。

现在，我们的问题变成了：构造两棵没有公共边的外向树 T_1 和 T_2（没有公共边才能让绳索最多被一人变色），使得每层的同层绳索恰好是 T_1 和 T_2 中的边交替。

下一步需要点创造性思维了。我们对于每两个相邻层求一次完美匹配（由每个点的度都是 k 可知，存在完美匹配，而且匹配上的石柱编号同奇偶），然后从祭坛出发设置 $m/2$ 条边分别指向第 1 层的第 1，3，5，…，$m-1$ 根石柱，则这些石柱的入度都为 1，再从这些石柱出发都设置一条边指向它们的匹配对象，则第 2 层的石柱 1，3，5，…，$m-1$ 入度为 1，…，直到第 n 层的石柱 1，3，4，…，$m-1$ 入度为 1。再从第 n 层的第一根石柱出发设置一条指向圣火炬的边，则圣火炬的入度为 1。然后从每层的石柱 $2i-1$（$m=1$，2，3，…，$m/2$）出发引一条指向石柱 $2i$ 的边，则整个图除了祭坛的入度为 0 外，所有点的入度均为 1，外向树设置完成，而且同层绳索中变色的和不变色的恰好交替。只要用它构造欧拉回路，问题就解决了——对于另外一个勇士，只需要把路线绕整个宫殿的中心轴“旋转”一下，使每个石柱和原先在它右边的相邻石柱重合，就可以使金银交替，且层间绳索也不会两次变色。

可是新的困难出现了。我们给一些边定了向，但是却无法保证能够给剩下的无向边也进行定向，有向保证欧拉回路的存在。用例题 2 的方法吗？可以，不过太麻烦，而且难以证明一定有解。下面的思路仍然是构造性的，它利用了图的特殊性。

对于祭坛，只需要对称从第一层的石柱 2，4，6，…，m 出发引一条边到祭坛，则祭坛的出入度平衡。对于圣火炬，和祭坛类似设计：从第 n 层的石柱 1，3，5，…，$m-1$ 各引一条边到圣火炬，而从圣火炬向第 n 层的石柱 2，4，6，…，m 各引一条边。

同层绳索给每个石柱的出度和入度都加一，所以不用考虑，而给匹配边和顶层/底层绳索的定向保证了这些边同时给所有石柱的出度和入度加一，所以也不用考虑。还剩下一些未匹配的层间绳索需要定向。我们考虑第 i 和第 $i+1$ 层的未匹配层间绳索。由于这时从每个石柱出发还有 $k-1$ 条绳索，为偶数，所以可以连续求 $k-1$ 次完美匹配，第 1，3，5，…，$k-2$ 次匹配后把匹配边定为 $i+1$ 指向 i 层；第 2，4，6，…，$k-1$ 次匹配后把匹配边定为 i 指向 $i+1$ 层，则每两次匹配后所有点的出入度平衡，最终得到一个欧拉图。

借助前面设计的外向树，我们可以得到勇士 1 的欧拉回路。在考虑勇士 2 时，把所有边反向并进行前面提到的“旋转”变换，这样可以得到勇士 2 的欧拉回路，问题终于解决了。

本题是一道难度较大的综合题，涉及到欧拉回路、外向树、匹配等图论知识，在程序实现时，不能简单用邻接矩阵或者邻接表，而需要根据图的特殊性分别储存三类边，这给编程和调试都带来了比较大的困难。通过解决此题，我们的图论综合素质得到了锻炼，希望读者在看完这篇解答后能自己继续独立思考，写出本题的代码。

练　习　题

2.5.17　方格取数[1]

在一个 $n\times n$ 的方格里，每个格子里都有一个正整数。从中取出若干数，使得任意两个

[1] 题目来源：经典问题

取出的数所在的格子没有公共边，且取出数的总和尽量大。

2.5.18　团队分组[①]

你的任务是把一些人分成两组，使得：

- 每个人都被分到其中一组；
- 每个组都至少有一个人；
- 一组中的每个人都认识其他同组成员；
- 两组的成员人数尽量接近。

这个问题可能有多个解决方案，你只要输出任意一个即可，或者输出这样的分组法不存在。

2.5.19　调皮的导盲犬[②]

有一个盲人牵着一只导盲犬在路边散步。可是，这只狗很贪玩，常常悄悄的溜到一些好玩的地方去，只在主人转弯的时候回到主人身边帮助他，如图 2-69 所示。

这条狗怕走得太远而找不到主人，而且它的速度最多只有主人的两倍，所以决定每次离开主人以后最多去一个好玩的地方，然后回到主人下一次转弯的地方等他。它该怎样做，才能去到尽量多的有趣地方呢？注意：对于狗来说，同一个地方第二次去就不觉得新鲜了。

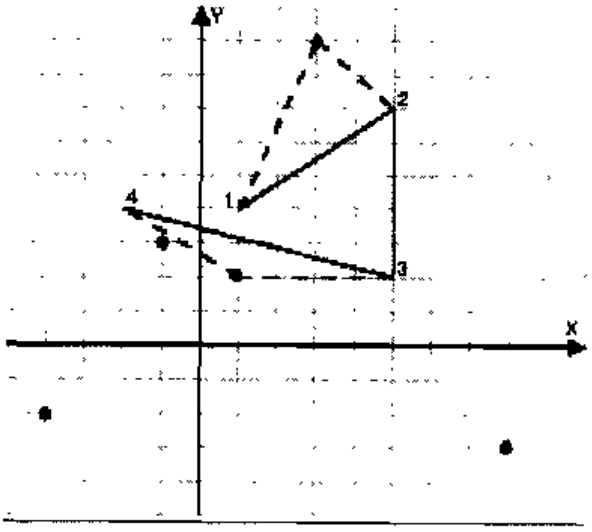


图 2-69　主人和狗的路线

（如图 2-69 所示，实线为主人的路线，虚线为狗的路线，点表示好玩的地方）

2.5.20　会议[③]

A 国的 *M* 位代表和 *B* 国的 *N* 位代表将要召开一次重要的会议($M, N \leqslant 1000$)。大会由 *K* 个小会议厅组成，每个小会议厅邀请两位指定的代表进行会谈，其中一位代表来自 *A* 国另一位来自 *B* 国。同一位代表可能被多个小会议厅邀请，但他最多只能参加一个小会议厅的会谈。作为大会的总负责人，请你告诉我最多能安排多少个小会议厅的会谈。

---

① 题目来源：ACM/ICPC Regional Contest NEERC 2001
② 题目来源：UVA Problem Archive
③ 题目来源：Ural State University Problem Archive

2.5.21　神秘之山[①]

M 个人在追一只奇怪的小动物。眼看就要追到了，那小东西却一溜烟蹿上一座神秘的山。众人抬头望去，那山看起来就是图 2-70 这个样子。

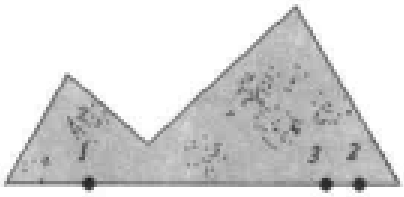


图 2-70　神秘之山和三个人

山由 *N*+1 条线段组成，各个端点从左到右编号为 0…*N*+1，即 $x_i<x_{i+1}(0\leqslant i\leqslant n)$，且 $y_0=y_{n+1}=0$（$1\leqslant y\leqslant n$）。

根据他们的经验，小动物极有可能藏在 1…*N* 中的某个端点。有趣的是大家很快发现了原来 *M* 恰好等于 *N*，这样他们决定每人选一个点看看它是否躲在那里。

一开始，他们都在山脚下。第 *i* 个人的水平位置是 s_i，他们每人选择一个中间点 x_i，先以速度 w_i 水平走到那里，再一口气沿直线以速度 c_i 爬到他的目的地。由于他们的数学不好，他们只知道如何选择一个最好的整数来作为中间点的横坐标 x_i。很明显，路线的任何一个部分都不能在山的上方，因为他们不会飞。

他们不希望这次再失败了，因此队长决定要寻找一个方案，使得最后一个到达目的地的人尽量早点到。他们该怎么做呢？

2.5.22　棋盘游戏[②]

在 24×24 的棋盘上放置相同数量的白棋子和黑棋子，每个小格放一个。不同的放置方式称之为不同的状态。若两个棋子所在小格有公共边，则称这两个棋子相连。每个棋子至多有 4 个棋子与它相连。在游戏中只允许两个相连的棋子交换。你的任务是找到最少的移动次数，使初始状态转化到目标状态。

**2.5.23　代表团

在某个城市里有 *n*（$n\leqslant 1000$）个俱乐部，该城市里的每个人都是 0 个或多个（不超过 50 个）俱乐部的成员，而恰好是一个政党的党员。每个俱乐部需要选举一个人组成一个代表团访问另一个城市，规定每个人只能代表一个俱乐部（即：俱乐部和代表团成员是一一对应的），而任何政党的党员人数不得等于或超过代表团总人数的一半。试设计一个可行的方案。

**2.5.24　加号和减号[③]

有一个(2*n*+1)×(2*n*+1)个网格，其中有的格子是黑色的。我们每次可以选择 2*n*+1 个格子组成使得任意两个格子既不同行也不同列，然后同时改变这些格子的颜色，即黑变白，白变黑。给出一个初始状态，求一种改变颜色的方案，使得最后只有不超过 2*n* 个格子是黑

[①] 题目来源：OIBH Online Programming Contest #1. 命题人：刘汝佳
[②] 题目来源：CEOI 1999
[③] 题目来源：Ural State University Problem Archive

色的。

例如，从图 2-71 可以通过改变格子{(1,1),(2,2),(3,3)},{(1,2),(2,3),(3,1)}, {(1,1),(2,3),(3,2)}, {(1,3),(2,1),(3,2)}的颜色得到。

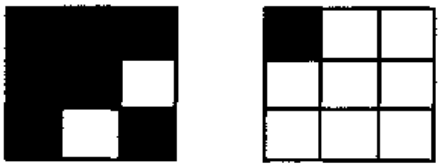


图 2-71　初始状态和终止状态举例

提示：本题可以用匹配和构造两种方法解决，读者不妨试一试。

2.5.25　动物园[①]

位于郊区的动物园很早就采用了当时先进的自动化管理设施对动物进行管理。但是由于当时的系统没有考虑 2000 年问题，使得管理人员十分担心。虽然采取了很多防范措施，系统还是在世纪之交出现了一些 bug，部分动物的笼子门自动打开了，关在里面的动物都跑出来了。

动物园的地形描述为一个 $n\times n$ 的网格，一个格子可以是建筑物或者平地。笼子的位置只可能在平地，动物也只在平地运动。每种动物的奔跑速度不一样，例如老虎一分钟可以跑 5 个格子，猫一分钟只可以跑 2 个格子等。如图 2-72 所示是一个例子（其中阴影部分是建筑物）。

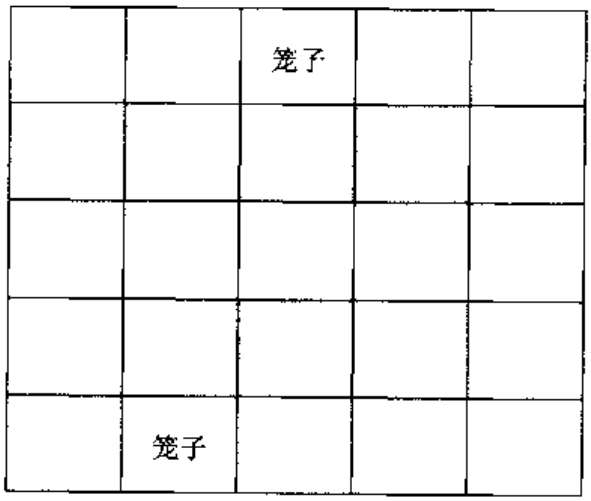


图 2-72　动物园示例

每个笼子只关一只动物，不同的笼子关不同的动物。不同的笼子可能在同一个格子里。

幸好动物园已经关闭，动物不会跑出动物园。警长 Still 接到报警后率领一支干警奔赴现场。这时动物已经跑出了笼子，所以干警们花了很多时间才控制住了局势，所有的动物都已经送到动物园的广场。但是此时有一个棘手的问题，由于系统完全崩溃，无法得知

[①] 题目来源：IOI2000 中国国家集训队原创试题.命题人：陈宏

动物是从哪个笼子里面跑出来的。此时，干警们记得动物的一些行动，都是如下的形式："第 t 分钟看到某某动物在位置（x, y）"

Still 希望通过这些零碎的信息得到动物是从哪个笼子跑出来的。

2.5.26　Unix 插头[①]

国际记者招待会将要举行，你负责安排会场。会场装备了各类插座以适应不同类型的插头。不幸的是，插座的数量可能不够。因此，需要让尽可能多的插头有插座可用。在会议开始前，你拿到了所有记者将要使用的设备清单。你注意到，对于一些插座，有许多设备都需使用，而对于另一些插座，则没有任何设备需要用它们。为了解决这一问题，你来到附近一家零件店。这家店出售转换插座类型的插座板。插座板可以连续使用，也即一个插座板可以从另一个插座板引出。这家商店并不出售多用插座板，但其所出售的插座板都货源充足。插座，设备，插座板数目都不超过 100。

***2.5.27　整数因子团问题**[②]

整数因子团是一个有趣的智力游戏。游戏规则如下：

① 给定由 n 个连续自然数组成的序列 1，…，n。

② 从当前序列中选择一整数，它在当前序列中至少有 2 个因子（该整数本身可以算作一个因子，例如整数 60 的所有因子是：60，30，20，15，12，10，6，5，4，3，2，1。它们构成整数 60 在当前序列的因子团。

③ 从当前序列中删去所选数的所有因子。

④ 重复步骤②和③，直到空序列或当前序列中所有数字在序列中仅有一个因子。

你在游戏中的得分是步骤②中所选择的数字的总和。

例如，当 n=6 时，给定的序列为 1，2，3，4，5，6。一个游戏过程是：

步骤②：选择整数 5；

步骤③：从当前序列中删去整数 5 的所有因子 5，1。当前序列改变为：2，3，4，6；

步骤②：选择整数 6；

步骤③：从当前序列中删去整数 6 的所有因子 6，2，3。当前序列改变为：4。游戏结束。

该游戏的得分是 11。

另一个游戏过程如下：

步骤②：选择整数 5；

步骤③：从当前序列中删去整数 5 的所有因子 5，1。当前序列改变为：2，3，4，6；

步骤②：选择整数 4；

步骤③：从当前序列中删去整数 4 的所有因子 4，2。当前序列改变为：3，6；

步骤②：选择整数 6；

步骤③：从当前序列中删去整数 6 的所有因子 6，3。当前序列改变为空序列。游戏结束。

[①] 题目来源：UVA Problem Archive

[②] 题目来源：经典问题. taxman

该游戏的得分是 15。

从上面的例子不难看出，游戏的步骤②所选数字的次序对游戏的得分有很大影响。应该如何选择才能使游戏获得最大得分？$2\leqslant n\leqslant 120$。如 $n=6$ 的解为 15，$n=15$ 的解为 81，$n=30$ 的解为 301，$n=50$ 的解为 808，$n=65$ 的解为 1328。

WEB 对于每个数 p 来说，能删除它当且仅当它至少还剩一个因子 q，使得 p/q 为质数。我们把满足 p/q 为质数的数对连一条边 $p\to q$，然后……本题还有其他做法，有兴趣的读者可以在本书主页上找到关于本题更多的讨论。

第3章　计算几何初步

本章内容简介

本章进入一个相对独立的领域——计算几何的基本知识和基本算法。

3.1 节从最基本的线段相交问题出发，从解析几何进入计算几何，介绍叉积和点积这两个最基本的计算几何工具，把读者引入计算几何这个位置和方向的世界。

3.2 节过渡到最基本的几何体——多边形、多面体，并详细讨论了两个相关问题——容积、重心的求取以及形内形外的判断。

3.3 节讨论最常用的几何模型——凸包。这里我们一方面着力从本质上挖掘凸的优美性质，并展示了凸包和排序的深刻的内在联系；另一方面对凸包的求解做了非常细致的讨论，着重分析了各种特殊情况。最后，从理论上分析凸包和凸多边形的优势，并把这些优势应用于具体的几何问题。

3.4 节介绍几种特殊算法。首先是矩形几何有关的离散化及其扫除法，然后结合凸包介绍分治法和增量法，最后是一个随机增量算法的专题——线性规划的几何解法。

阅读本章，最好的参考文献是两本英文书：[86]和[94]。比较通俗易懂的中文入门书有[7]的“计算几何”一章，比较专业的中文教材是[92]。算法方面可以参考两本英文名著：[93]和[84]，或者本书第 1 章相应内容。

3.1　位置和方向的世界——计算几何的基本问题

在本节中，我们将从一个十分简单的问题——判线段相交及求其交点出发，讨论计算几何最基本的问题——位置和方向，介绍计算几何最基本的工具——叉积和点积。最后给出一些应用实例。

我们首先约定，除非特别申明，本节中的所涉及的坐标系都是笛卡尔直角坐标系，N 维空间中的点表示为一个 N 维向量。例如，二维平面上的点表示为 $\vec{P}=(x,y)$，当然，不牵涉到运算时，也可以简写成 $P=(x,y)$。

基本问题：判断二维平面上的两条线段是否相交。并拓广到三维情形。

输入：四个点，分别表示第一条线段的两个端点和第二条线段的两个端点。

输出：YES/NO（表示相交与否）。

当然，这里需要特别申明的是，此处“相交”是指**两条线段恰有惟一一个不是端点的公共点**，我们称为“**规范相交**”（proper intersection[86]）。即如果一条线段的一个端点恰在另一线段上，则不视为相交；如果两条线段部分重合，也不视为相交，如图 3-1 所示。这些比较复杂的情况（“**非规范相交**”），我们后面会加以讨论（见 3.1.2 小节之“特殊情况与点积”）。现在首要的问题是，不考虑这些特殊情况，如何解决这个基本问题？进一步地，如何简单地解决这个问题？

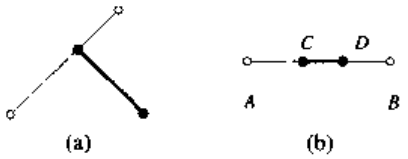


图 3-1　不视为相交的特殊情形（非规范相交）举例

解析几何的解法及其弊端　我们很容易想到解析几何中的基本方法——写出直线方程，联立求解得交点。这不失为一种做法，但（计算机）处理起来，有诸多不利之处。首先，不能用斜截式或点斜式写方程。因为碰到平行于 y 轴的线段斜率将是∞，无法表示。于是可以增加一个冗余参数，用两点式写出一般式方程，如 $Ax+By+C=0$。

但是，首先必须判断解的情况：

- 若无解则为平行（或者三维的异面直线）；
- 若有无穷多解则说明两条线段共线，判为不相交（按我们定义的规范相交）；
- 若有惟一解，再判断这个交点是否分别在两条线段的内部（非端点），若是则为相交，否则为不相交。

浮点误差　继续深入下去，将发现，即使是惟一解的情况，求交点过程中，使用了浮点数的除法，这将带来巨大的浮点误差，给相交的判断带来极大的不确定性。尤其是当两条线段近乎平行时，该算法对浮点除法的精确度相当敏感。最具讽刺意义的是，我们通过大量牺牲换来的“交点”，对于本问题的结果还没有直接的意义，得出交点后对每条线段每个端点每一维分量都要进行比较，总共需 2×2×2=8 次，若是三维则需 2×2×3=12 次比较运算。当然，如果不用一般式方程，改用参数方程，二维、三维的比较次数都可以减少为 4 次（每条线段 2 次，相当于一维），但是建立参数方程的运算代价比较昂贵，用到了除法和开根运算，误差较大，如用三角函数，则误差更大。而其实我们只问是否相交，对交点并不感兴趣。实际上，大量计算几何问题和竞赛中的几何题也往往仅对“相交与否”感兴趣（后文我们将讨论大量这样的例子）。于是，我们不得不转向另外的算法——计算几何，该算法非常简洁，且仅限于加、减、乘和比较运算，还很容易推广到三维甚至高维情形。

当然，这里必须指出的是，交点并非完全没用，也有不少几何问题不但问是否相交，若相交还要求交点。是否仍需用方程来求交点呢？不必，计算几何的方法也可以更方便地求得交点。不过，计算几何方法是否强大到可以避免一切浮点误差呢？回答也是否定的——任何在特定计算机上运行的程序都不可能完全避免浮点误差。只不过计算几何相对于解析几何把浮点误差降低到了最低限度。

下面就先把这个判相交的问题转化为计算几何问题，再用计算几何的知识解决它。

3.1.1　从相交到左右——基本问题的转化

1. 从判线段相交到判两点在直线异侧

不求交点判相交　既然上文我们说只关心相交与否，那么能否绕过求交点问题呢？再考虑一下“相交”的定义——两条线段恰有一个不是端点的公共点，画一张相交线段的草图，如图 3-2 所示，观察每条线段两个端点与另一条线段的关系，你是否发现这样的规律：**两条线段相交时，每条线段两个端点都在另一条线段的异侧**。其实从定义上看，这是显然的。

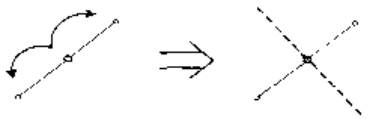


图 3-2　线段两个端点在另一条线段所在直线的异侧

对于每条线段，公共点（交点）不是端点

→ 那自然在线段的内部 → 即两个端点之间

→ 这样两个端点自然在公共点的异侧 → 于是在另一条线段所在直线的异侧。

再来考察问题的另一个方向：当每条线段两个端点都在另一条线段所在直线异侧时，两条线段是否一定相交呢？其实我们只要把上面最后一段话全部倒过来讲，发现这也是显然的：

首先，一条线段两个端点都在另一条线段所在直线异侧

→ 这两条线段所在直线必然相交（有且仅有一个交点）

→ 每条线段两个端点都在另一条线段所在直线异侧，所以必然跨在交点两侧

→ 即交点在每条线段两个端点之间 → 即在每条线段内部

→ 便成了两个线段的公共点 → 于是这两条线段恰有一个不是端点的公共点。

现在，我们终于得到并且证明了如下的重要关系：

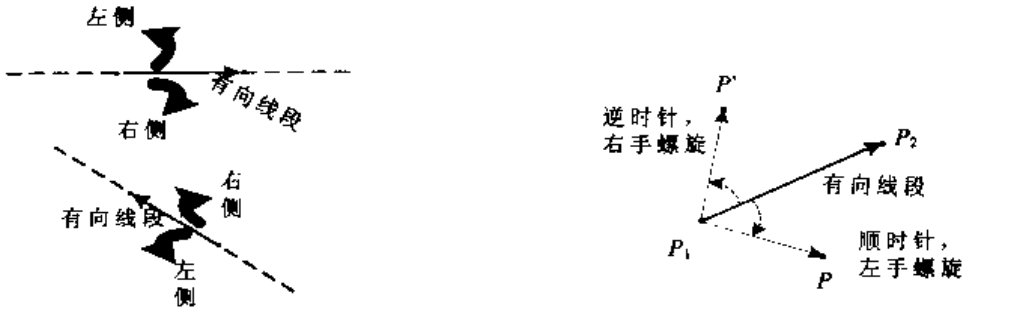
线段相交 ⇔ 两条线段恰有一个不是端点的公共点（定义）⇔ 每条线段两个端点都在另一条线段所在直线异侧（充要条件）。

2. 从判异侧到判两个向量左右手螺旋

从异侧到左右　于是我们对一个判线段相交的问题，跳出了求交点的思维定势，从而转化成一个判两点是否在直线异侧的几何问题。那么，怎么判异侧、同侧的问题呢？这里，得首先想想“侧”到底是个什么概念。画张草图，如图 3-3 所示，我们知道一条直线（除非与 x 轴平行）必然把平面分成左右两个部分。如果另一条线段的两个端点落在同一部分（同左或同右），那就是同侧；否则（一左一右）就是异侧。现在把这条分平面的直线改成题中的线段，然后，规定第一个端点为起点，第二个端点为终点，于是就成了一条有向线段即向量[1]。于是，如果你的眼睛从起点向终点看去，那么平面自然分为左右两部，即使

[1] 本章不对有向线段和向量作严格区分，即认为所有方向相同、长度相等的向量，不管起点如何，都视作相同的向量

线段与 x 轴平行也没关系，因为现在有向线段和原来的直线不同，它本身就有方向性，**左右都是相对于它本身的方向的，而不是相对于绝对的坐标系的**。这也是计算几何相对于解析几何的一大本质优势。

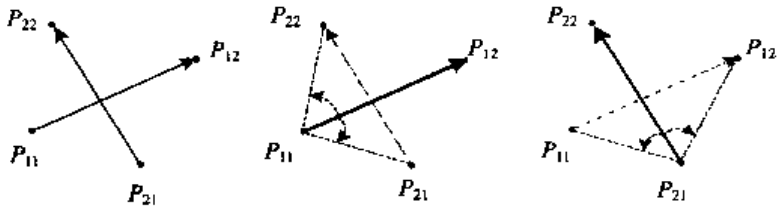


(a) 有向线段的左右侧　　(b) 判断点在有向线段的左右侧——向量左右手螺旋

图 3-3　从异侧到左右手螺旋

向量的左右手螺旋　那么，只要能够判断一个点在有向线段的左边还是右边即可以了。令有向线段的两个端点为 P_1，P_2，要判断的点为 P，我们来判断点 P 在向量 $\overrightarrow{P_1P_2}$ 的哪一边。也很简单，只要添一条辅助向量 $\overrightarrow{P_1P}$ 即可。我们现在只要判断对于公共起点 P_1，向量 $\overrightarrow{P_1P}$ **相对向量** $\overrightarrow{P_1P_2}$ **是顺时针还是逆时针方向**，或者说，从向量 $\overrightarrow{P_1P_2}$ 到向量 $\overrightarrow{P_1P}$ **是左手螺旋方向还是右手螺旋方向**。

问题的解决　有了上述工具，就可以方便地解决线段相交问题。如图 3-4 所示，设两条线段分别为 $P_{11}P_{12}$ 和 $P_{21}P_{22}$。首先对 P_{21} 运用上述测试判断其在 $\overrightarrow{P_{11}P_{12}}$ 左侧还是右侧。同样处理 P_{22}，就判得 P_{21} 和 P_{22} **是**否在向量 $\overrightarrow{P_{11}P_{12}}$ 的异侧或同侧。同样地，交换一下，我们也可以判断 P_{11} 和 P_{12} 分别在向量的哪一边，是否在向量 $\overrightarrow{P_{21}P_{22}}$ 的异侧。如果都是异侧，则认为两线段**相**交。上述方法被形象地称为“跨立实验”（参见[7]）。



(a)　(b)　(c)

图 3-4　基本问题的解决——跨立实验

于是只要能判断**左**右手**螺**旋就可以了。到这里为止，我们终于把判线段相交的问题，转化成判两点**是**否在直线异侧的问题，又把判异侧的问题，转化为判两个向量左右手螺旋的问题。

那么，如何判断两个向量是左手螺旋还是**右**手螺旋呢？这就是 3.1.2 小节所要讨论**的**内容：计算几何**最基本**、最核心的算法——叉积，以及它的姐妹算法——点积。

以上，我们用了很大的篇幅，详细讨论了基本问题转化为左右手方向问题的全过程，

其用意之一是考虑到不少读者初次接触计算几何这个全新的领域，但更主要的是想通过这个转化过程，让读者领略到几何本身，即“空间位置关系”中内在的美。解析几何固然能解决很多几何问题，但它毕竟是用代数方法来解几何问题，很大程度上掩盖了几何的魅力。而我们通过这个转化过程，力图使读者重新发现已经生疏了的几何本身的美，并就此踏上美妙的计算几何之旅。

3.1.2　左右和前后——叉积和点积

为了解决3.1.1小节提出的问题，本小节中将介绍两种向量的基本运算——叉积和点积。

1. 从左右手螺旋到叉积

需要一种运算　为了解决左右手螺旋问题，即对于公共端点 P_0，有向线段 $\overrightarrow{P_0P_1}$ 到有向线段 $\overrightarrow{P_0P_2}$ 是否构成右手螺旋方向（即逆时针方向），我们先把原点平移到 P_0 点，记向量 $\vec{a}=\overrightarrow{P_0P_1}$，$\vec{b}=\overrightarrow{P_0P_2}$。

现在的目标是：找到一种关于 $\vec{a}$，$\vec{b}$ 的**运算**，使得它能判断顺时针方向还是逆时针方向，换言之，当 $\vec{a}$，$\vec{b}$ 成右手螺旋和成左手螺旋时，这种运算的结果是迥然不同而且容易辨别的。最好的情形是，它们的结果正好是相反的，而这种运算又必须比较简单。当然，在第一象限里可以通过斜率比较来实现：

$$\frac{y_b}{x_b}-\frac{y_a}{x_a}>0$$

但是这个式子无法推广，而且除法代价高、误差大，分母又不能为 0，我们最好只用加减乘。于是可以尝试一下，把上式通分，发现这种运算（暂记为 $\vec{a}\circ\vec{b}$）恰好可以满足要求。

$$\vec{a}\circ\vec{b}=x_ay_b-x_by_a=-(x_by_a-x_ay_b)=-\vec{b}\circ\vec{a}$$

当 $\vec{a}$，$\vec{b}$ 成右手系时，这种运算的结果总是正的，成左手系时总是负的。

特别注意当 $\vec{a}$，$\vec{b}$ 共线时，不论同向还是反向，这个运算结果总是零。

这样一来就解决了判断左右手方向的问题。

读者可能已经发现，这样定义的运算就是二维平面上的叉积的数值[1]。

$$\vec{a}\times\vec{b}=x_ay_b-x_by_a=\begin{vmatrix}x_a & y_a\\ x_b & y_b\end{vmatrix}=-(x_by_a-x_ay_b)=-\vec{b}\times\vec{a}$$

叉积与有向面积　回忆力学中力的合成的平行四边形法则，如图 3-5 所示，设 $\vec{c}=\vec{a}+\vec{b}$，很容易证明叉积运算结果的绝对值等于平行四边形 *oacb* 的面积，也是△*oab* 面积的两倍。如果保留正负号，就成了**有向面积**，即 oab 成右手系时为正面积，左手系时为负面积。我们在后文将会多次看到，“有向面积”的概念是一个很本质的概念，并且会在多边形边界方向和多边形面积中发挥巨大的作用。

[1] 严格地说这不是叉积的大小，只是二维情形的认识

再换一个角度看问题，旋转坐标系得$\vec{a}$与x轴重合，$\vec{b}$如果在x轴上方，就在$\vec{a}$的逆时针方向，$\vec{a}\,\vec{b}$成右手系，$\vec{b}$如果在x轴下方，就在$\vec{a}$的顺时针方向，$\vec{a}\,\vec{b}$成左手系。

叉积的几何形式　那么这与三角函数有什么联系呢？

如图 3-6 所示，如果考虑从$\vec{a}$到$\vec{b}$的有向夹角θ，则根据 sin 函数x轴上方为正，下方为负，x轴上为 0 的特征，恰好对应了$\vec{a}\,\vec{b}$成右手系、左手系和共线的三种情况。

图 3-5　叉积与面积

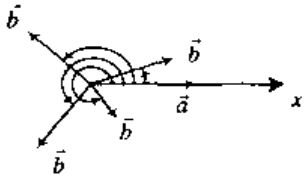


图 3-6　叉积与三角函数

其实，考虑图 3-5 所示的平行四边形的面积：$S=|\vec{a}|h-|\vec{a}||\vec{b}|\sin\theta$，如果把$\theta$作为有向夹角，则面积也成了有向面积。由此得到了叉积的几何形式：$\vec{a}\times\vec{b}=|\vec{a}||\vec{b}|\sin\theta$[①]。

这就与刚才的关于图 3-6 的讨论一致了。

2. 三维叉积的几何意义

到目前为止，我们已经可以完整地解决前文提出的基本问题了。但是，刚才所有这些还不是真正意义上的叉积，只是叉积在二维的表现形式，或者说只是叉积在解决左、右手螺旋判定时的一个应用。现在有必要把视野从二维拓广到三维，以便看清叉积的实质。

三维叉积　我们在二维平面里看叉积，只能把它看成一个数量，其实，在三维中，叉积还是一个向量，其大小就是前述的平行四边形的面积（绝对值），那么方向呢？很自然地，既然二维平面中，叉积的符号表示左右手螺旋，即右手螺旋时大拇指的方向，那么，在三维空间中，起方向也是右手螺旋，从$\vec{a}$到$\vec{b}$时大拇指的方向。即$\vec{a}\times\vec{b}$的方向垂直于两向量$\vec{a}$，$\vec{b}$，（也就是垂直于$\vec{a}\,\vec{b}$平面），且$\vec{a}$，$\vec{b}$、$\vec{a}\times\vec{b}$构成右手系，如图 3-7 所示。

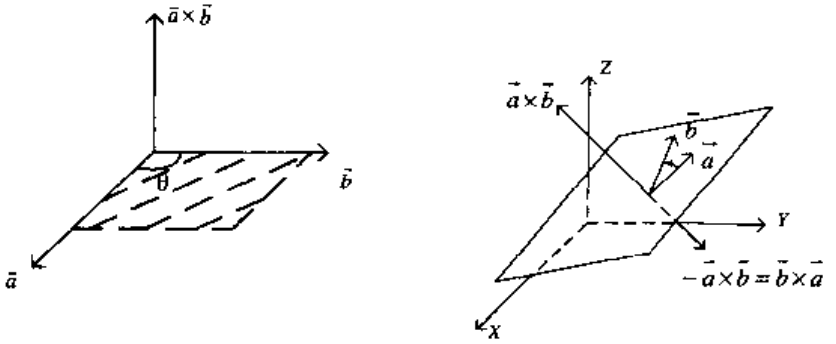


图 3-7　叉积与右手系

$$|\vec{a}\times\vec{b}|=|\vec{a}||\vec{b}|\sin\theta\quad(0\leqslant\theta<\pi)$$

① 这不是真正的叉积

这样就成了平面的法向量，我们今后在讨论平面时将经常用它来表示平面的方向，当然，对于计算几何，重要的还是给出可操作的形式。

$$\vec{a}\times\vec{b}=\begin{vmatrix} \boldsymbol{i} & \boldsymbol{j} & \boldsymbol{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix}=(a_yb_z-a_zb_y)\,\boldsymbol{i}+(a_zb_x-a_xb_z)\,\boldsymbol{j}+(a_xb_y-a_yb_x)\,\boldsymbol{k}$$

下面，我们尝试对它作一个简单的解释。从形式上看，它的每一维分量又是一个二维的叉积（这其实是代数上的 Laplace 展开）。你也许很快会发现这三个二维叉积的参与向量，恰是 $\vec{a}$、$\vec{b}$ 向三个平面 *oxy*，*oyz*，*ozx* 的投影，如果把 $\vec{a}\times\vec{b}$ 也沿 $\boldsymbol{i}$、$\boldsymbol{j}$、$\boldsymbol{k}$ 三个方向分解，例如 $\boldsymbol{i}$ 方向就是 $(a_yb_z-a_zb_y)\boldsymbol{i}$，而它恰好就是 $\vec{a}$、$\vec{b}$ 向 *oyz* 平面投影 (a_y, a_z) 和 (b_y, b_z) 的叉积。这就很好地解释了上式，今后，我们还会发现，投影和分解的思想将广泛地应用于向量运算。

然而，至此我们仍没有认识到叉积的实质（高维情形），留待后文（多边形一节）讨论。

3. 判平面线段相交的实现与应用举例

有了上述理论准备，现在考虑如何简洁、高效地实现基本问题的解决。这里只讨论上文定义的那种“规范相交”的判定，至于一般意义的相交，以及求交点，留待本小节后文讨论。

把上面的理论讨论总结一下，就是：

两线段 *AB*、*CD* “规范相交”

$\Leftrightarrow (\overrightarrow{AB}\times\overrightarrow{AC})_z\cdot(\overrightarrow{AB}\times\overrightarrow{AD})_z<0$ 且 $(\overrightarrow{CD}\times\overrightarrow{CA})_z\cdot(\overrightarrow{CD}\times\overrightarrow{CB})_z<0$

其中 $\vec{V}_z$ 表示 $\vec{V}$ 的 *z* 轴分量。

或者，更简单地，我们用有向面积的概念，*Area* (*A*, *B*, *C*) 表示△*ABC* 的有向面积，即

$$Area(A,B,C)=\frac{1}{2}(\overrightarrow{AB}\times\overrightarrow{AC})_z$$

为了方便，我们用

$$Area2(A,B,C)=2\cdot Area(A,B,C)=(\overrightarrow{AB}\times\overrightarrow{AC})_z=\begin{vmatrix} x_b-x_a & y_b-y_a \\ x_c-x_a & y_c-y_a \end{vmatrix}$$

浮点误差及其处理　但是，计算机实现毕竟不像数学那么完美，因为实数是用浮点数表示和运算的，其精度自然受到限制，特别是经过浮点减法和乘法以后，误差已经累积得很大了，这时候，这个关键的“<0”判断就很容易出错。

例如，我们可以做个实验，令 a =1234567890123456，b =9876543210987654，$A(a, b)$，$B(a+1, b+1)$，$C(a-2, b-2)$，显然 *A*、*B*、*C* 在一直线上，即 *Area2*(*C*, *A*, *B*)=0，但用计算机做叉积，却发现不等于零[1]。

所以，应该改写“<0”这个判断，允许存在一定误差，例如，可以认为 0 附近的某小邻域 $U(0, \delta)$ 内的点都看作 0，由此，可以写一个类似符号函数 sgn(*x*) 和 C 语言字符串比较函数 strcmp 的三出口浮点数符号函数：

[1] 关于实数的计算机实现，超出了本书的范围，读者可以参考有关计算机组成的专业书

```
int dblcmp(double d)
{
if(fabs(d)<precision)
    return 0;
return(d>0)?1:-1;
}
```

这里 precision 可取一个小常量，如 10^{-6}，当然也可以用“相对误差”，即与参加叉积的向量长度有关。

这个函数相当重要，它把浮点数映射为 -1, 0, 1 的符号，另外也提供了实数比较的安全的方法，是计算几何中解决浮点误差的关键，将在今后发挥很大的作用。

代码实现　余下的问题就很简单了，代码段如下：

平面上点的表示：

```
typedef struct { double x,y; } Point;
```

叉积运算的代码：

```
double det(double x1,double y1,double x2,double y2)
{
return x1×y2-x2×y1;
}
double cross(Point a,Point b,Point c)
{
return det(b.x-a.x,b.y-a.y,c.x-a.x,c.y-a.y);
}
```

判断的关键代码段如下：

```
int segcrossSimple(Point a,Point b,Point c,Point d)
{
return(dblcmp(cross(a,c,d))^ dblcmp(cross(b,c,d)))==-2 &&
    (dblcmp(cross(c,a,b))^ dblcmp(cross(d,a,b)))==-2;
}
```

其中 ^ 是按位异或（bitwise XOR），考虑到 dblcmp 的返回值都在-1, 0, 1 当中，此时 $a \wedge b==-2$ 等价于 $a \times b<0$ 或 $a \times b==-1$，且速度最快（$a, b = -1, 0, 1$）。

【例题 1】房间最短路问题[1]

给定一个内含阻碍墙的房间，如图 3-8 所示，你要编程找到一条从起点到终点的最短路。房间的边界固定在 $x=0$，$x=10$，$y=0$ 和 $y=10$。起点和终点固定在（0,5）和（10,5）。

[1] 题目来源：UVA Problem Archive 393, The Doors

房间里还有 0 到 18 个竖直的墙，每个墙有两个门。输入给定墙的个数，每个墙的 x 位置和两个门的 y 坐标区间，所有输入都是整数。输出最短路长度。下图描绘了这样的一个房间并给出了最短路。

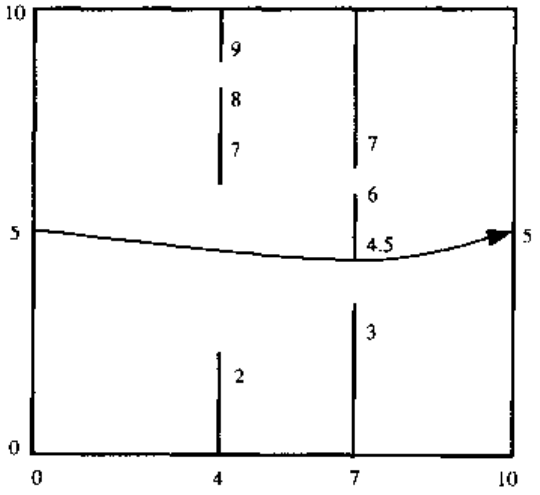


图 3-8　房间举例

【分析】

这是一道简单而且有趣的几何题。如果将每扇门的上下两点，加上起点和终点作为无向图的顶点，两个相互之间可以直达（即连线不经过任何障碍）的顶点之间连一条边，长度为两点之间的距离，这样就转化为一个图论的最短路问题，求起点到终点的最短路径，可以用最简单的 Dijkstra 算法来实现，也可考虑到图的极端特殊性，采用动态规划法。当然，这里主要关心的问题是“直达”的判断。即对于任两个顶点，是否存在一个障碍线段与两点连线相交，这里的相交，显然是前文定义的那种线段“规范相交”。于是，我们只要套用刚才的判线段相交的叉积算法就可以了。

当然，由于此题的特殊性，障碍线段都是竖直线段，所以如果用解析几何的办法也较简单。例如可以用类似定比分点的方法确定交点 x 坐标，然后只要比较 y 坐标是否在障碍线段范围内即可。不过，鉴于除法的代价和误差，而本题又是整点，若用叉积则没有任何误差，所以还是选择了计算几何算法。

4．特殊情况与点积

上文中我们用叉积完整地解决了“规范相交”的判定，但是还遗留了两个问题：

① 对于一般意义的相交（两条线段至少有一个公共点）的判定，或者说，我们要能识别规范相交与非规范相交

② 对于规范相交，要能同时方便地得到其交点。

非规范相交　首先，注意到非规范相交情况是非常多的，同时，要能区别出一些不相交的情况，例如图 3-9 所示。

回忆规范相交的判定式：

$$Area2(A, B, C)\times Area2(A, B, D)<0 \text{ 且 } Area2(C, D, A)\times Area2(C, D, B)<0$$

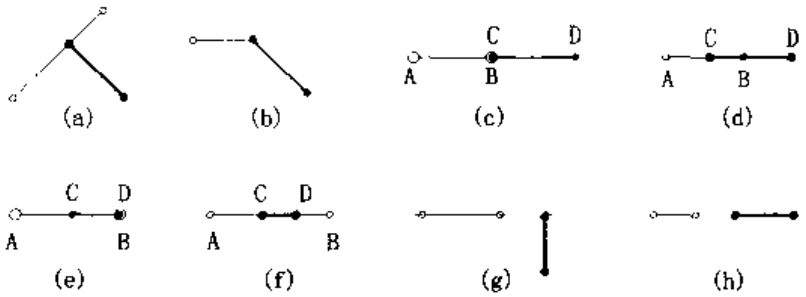


图 3-9　非规范相交与不相交的情况

（其中（a）～（f）是非规范相交，（g）、（h）是不相交，（a）～（c）有惟一的交点，（d）～（f）有无数个交点）

我们发现，规范相交，上述 4 个叉积均不为 0（dblcmp 返回值不为 0），而非规范相交至少有一个为 0，但是图 3-9(g)、(h)两种不相交的情况也有叉积为 0——有一点在另一条线段延长线上。

由此可以把非规范相交的判定简写为：至少有一端点在另一条线段上（内部或与端点重合）。

点在线段上的判定　那么如何实现上述判定呢？所谓点 A 在线段 BC 上，无非以下两点：

① A 在 BC 所在直线上（即叉积 $\overrightarrow{AB}\times\overrightarrow{AC}=0$）；

② A 在 BC 的范围内。

因为叉积已经算好了（为了判断规范相交），所以只要发现某个叉积为 0 时，就做②的判断，若成立，则判为非规范相交。

至于②的判断，比较直观的想法是通过坐标的比较，但是碰到线段接近水平、竖直时坐标范围极小，容易引起误差，而且，我们希望这个比较函数最好能设计成 dblcmp 一样的三出口符号函数，即

$$between(C,A,B)=\begin{cases}-1 & (C\in AB)\wedge(C\neq A)\wedge(C\neq B)\\ 0 & (C=A)\vee(C=B)\\ 1 & C\notin AB\end{cases}$$

所以，这里先介绍另一种方法——点积，然后再回来讨论坐标比较。

点积与叉积是对称的，同样在计算几何中有着广泛的应用。叉积判断的是左右，点积判断的是前后，更形象地说，点积是考察两个向量在方向上的一致程度（相似性），如图 3-10 所示。

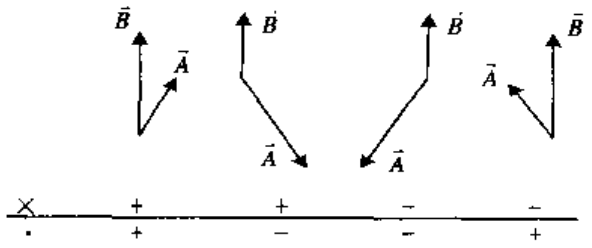


图 3-10　叉积和点积；左右与前后；sin 与 cos；四个象限

点积的定义：向量 $\vec{A}(x_1, x_2, \cdots, x_n)$，$\vec{B}(x_1', x_2', \cdots, x_n')$

点积 $\vec{A} \cdot \vec{B} = \sum_{i=1}^{n} x_i x_i'$　（注意点积结果是标量）

对于两维向量 $\vec{A}(x_1, y_1)$，$\vec{B}(x_2, y_2)$，$\vec{A} \cdot \vec{B} = x_1 x_2 + y_1 y_2$

容易看出，二维平面上，叉积、点积分别与 sin、cos 对应，自然，相对于叉积的 $\vec{A} \times \vec{B} = |\vec{A}||\vec{B}|\sin\theta$，点积 $\vec{A} \cdot \vec{B} = |\vec{A}||\vec{B}|\cos\theta$。因为向量夹角总是[0,π]，cos 在这个范围内有反函数，而 sin 没有，所以点积在今后的应用中，常被用于求夹角 θ 的大小：（可以参考习题 3.1.7）。

$$\theta = \cos^{-1} \frac{\vec{A} \cdot \vec{B}}{|\vec{A}| \cdot |\vec{B}|}$$

需要指出的是，这种方法对于三维或高维空间也成立。

点积用于前后判定　而这里要用的却不是夹角，只是前后方向，而且是非常特殊的情况下——同一直线上三点的方向，方法非常简单：如果已知 C 在 AB 所在直线上，那么只要看 $\overrightarrow{CA} \cdot \overrightarrow{CB}$ 的符号即可。若等于 0 则 C 与 A 或 B 重合；若小于 0 则 C 在 AB 内部；若大于 0 则 C 在 AB 外部。这时再用 dblcmp 函数“滤波”一下即可得到上文 *between(C,A,B)*交所要求的效果。

这个办法的优点是，用一个定义简洁的运算实现了三出口判断，缺点是需要两个浮点乘法，所以又回到坐标比较上来。

坐标比较　首先，已知 C 在 AB 所在直线上的情况下，只要考察任意一维分量即可。为了尽量避免误差，可以选择跨度大的那一维。

例如，假设$|a.y-b.y| > |a.x-b.x|$，那么就判断是否 $c.y \in [\min(a.y, b.y), \max(a.y, b.y)]$。

这时，为了实现与点积一样的效果（三出口），可以用一个巧妙的乘法：

$$\text{dblcmp}(c.y-\min(a.y,b.y)) \times \text{dblcmp}(c.y-\max(a.y,b.y))$$

来实现。大家可以发现，不论 C 在什么位置，上述结果均符合要求，如下表所示。

C	<min	=min	min~max	max	>max
dblcmp1	-1	0	1	1	1
dblcmp2	-1	-1	-1	0	1
*	1	0	-1	0	1

5. 求线段交点及应用举例

叉积求交点　前面我们说过，叉积不仅可以用来判线段相交，还可以用来求线段交点，现在就来讨论这个方法并举一个例题。

我们知道，若已确定两线段相交，可以用解析几何中的直线方程求解来求交点，这时其运算是可靠的，误差也降到了最低限度。不过，既然已经计算了叉积，如果再写两点式方程，就太浪费了。如图 3-11 所示，线段 AB、CD 交于 P，有

$$\frac{|DP|}{|CP|}=\frac{S_{\Delta ADB}}{S_{\Delta ACB}}=\frac{|\overrightarrow{AD}\times\overrightarrow{AB}|}{|\overrightarrow{AC}\times\overrightarrow{AB}|}$$

$$x_P=\frac{S_{\Delta ABD}\cdot x_C+S_{\Delta ABC}\cdot x_D}{S_{\Delta ABD}+S_{\Delta ABC}}=\frac{Area2(A,B,D)\cdot x_C-Area2(A,B,C)\cdot x_D}{Area2(A,B,D)-Area2(A,B,C)}$$

（注意，规范相交时 $Area2(A,B,D)\times Area2(A,B,C)<0$）

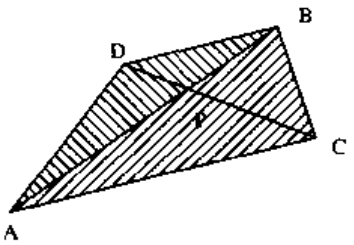


图 3-11　利用叉积和定比分点求交点

这就是利用叉积的面积意义，通过定比分点求出 P，其中涉及的两个叉积值已经在判相交时计算过，只要定比分点即可，非常简单。

我们仅对规范相交的情况求交点。当然，非规范相交中的(a)、(b)、(c)三种情况也是有惟一交点的，其中(a)、(b)可以适用上述定比分点公式，而(c)不适用；另外，把(c)和(d)、(e)、(f)分开也比较麻烦，留给读者思考。

完整的代码　下面给出包含非规范相交和叉积求交点的完整代码：

```
// 0 no intersection, 1 proper intersection, 2 improper intersection
// p - point of intersection
int segcross(Point a,Point b,Point c,Point d,Point & p)
{
double s1,s2,s3,s4;
int d1,d2,d3,d4;

d1=dblcmp(s1=cross(a,b,c));
d2=dblcmp(s2=cross(a,b,d));
d3=dblcmp(s3=cross(c,d,a));
d4=dblcmp(s4=cross(c,d,b));

// 若规范相交则求交点的代码
if ((d1^d2)==-2 && (d3^d4)==-2)
{
    p.x=(c.x*s2-d.x*s1)/(s2-s1);
    p.y=(c.y*s2-d.y*s1)/(s2-s1);
    return 1;
}
```

```
// 判定非规范相交
if (d1==0 && betweenCmp(c,a,b)<=0 ||          // 若a,b,c共线，且c在ab范围内
    d2==0 && betweenCmp(d,a,b)<=0 ||
    d3==0 && betweenCmp(a,c,d)<=0 ||
    d4==0 && betweenCmp(b,c,d)<=0)
    return 2;
return 0;
}
```

其中 dblcmp 和 cross 已经介绍过了。这里 betweenCmp(a,b,c)有两种实现方法：

① 坐标比较

```
// 判断a是不是在bc范围内
int betweenCmp(Point a,Point b,Point c)
{
if (fabs(b.x-c.x)>fabs(b.y-c.y))
    return xyCmp(a.x, min(b.x,c.x), max(b.x,c.x));
else
    return xyCmp(a.y, min(b.y,c.y), max(b.y,c.y));
}
int xyCmp(double p, double mini, double maxi)
{
return dblcmp(p-mini)*dblcmp(p-maxi);
}
```

② 点积

```
int betweenCmp(Point a,Point b,Point c)
{
return dblcmp(dot(a,b,c));
}
double dot(Point a,Point b,Point c)
{
return dotdet(b.x-a.x,b.y-a.y,c.x-a.x,c.y-a.y);
}
double dotdet(double x1,double y1,double x2,double y2)
{
return x1*x2+y1*y2;
}
```

计算几何中不少问题都是需要先判相交，再求交点的，下面看一个有趣的题目。

【例题 2】管道问题[①]

有一宽度为 1 的折线管道，如图 3-12 所示，上面各顶点为(x_0, y_0), (x_1, y_1),…, (x_n, y_n)，下面各顶点为(x_0, y_0-1), (x_1, y_1-1),…, (x_n, y_n-1)，假设管壁都是不透明的、不反射的，光线从左边入口处的(x_0, y_0), (x_0, y_0-1)之间射入，向四面八方直线传播，问光线最远能射到哪里（x坐标）或者能穿透整个管道。

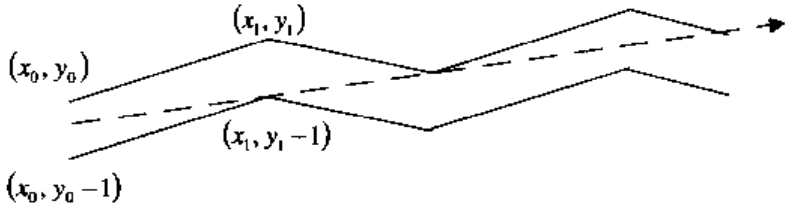


图 3-12　管道问题

【分析】

此题初看可能十分困难，但是上下顶点对于限制光线非常关键。首先，我们想到如果一根光线自始至终未曾擦到任何顶点，肯定不是最优的（可以通过平移使之优化）。然后，如果只碰到一个顶点，那也不是最优的，可以通过旋转，使它碰到另一个顶点，并且更优，最后要说明，最优光线必然是擦到一个上顶点和一个下顶点。以上三步的证明用反证法并不困难，所以留给读者。

不碰任何顶点 —上下平移→ 碰一个顶点 —绕此点旋转→ 碰两个顶点 —绕前点或后点旋转→ 碰一上一下两个顶点

（其中每一步都是根据目前阻挡光线的管壁的方向，选择使光线更优的操作）

于是有了一个简单的算法，任取一个上顶点和下顶点，形成直线 l。若 l 能射穿左入口，即当 $x = x_0$ 时，直线 l 在（x_0, y_0）和（$x_0, y_{0\,1}$）之间，则是一条可行光线。再从左到右依次判断每条上、下管壁是否与 l 相交，相交则求交点，并把交点 x 值与当前最佳值比较，若所有管壁都不与 l 相交，说明 l 射穿了整个管道。当然，这个算法有较大的改进余地，留给读者考虑。

练　习　题

思考题：

3.1.1　我们的算法仅对规范相交的情况求交点，现在请你设计一个尽可能简单的算法，能对非规范相交中惟一交点的情况（图 3-9 的(a)~(c)）也求出交点。

3.1.2　严格证明例 2 中的结论（最优光线必然是擦到一个上顶点和一个下顶点）。

3.1.3　本节讲述两个例题时，由于专注于几何分析，对算法雕琢较少，所以都留下了较大的优化空间。现在请你尽可能地优化两题的算法，并到检测结果。

3.1.4　我们讲到，需要一种运算，使得它能简单地判断两个向量是否顺时针方向。我

[①] 题目来源：UVA Problem Archive 303

们说只要这个运算满足一些要求即可，那么除了叉积，你还能找到其他这样的运算吗？

编程题：

3.1.5　迷宫寻宝[①]

平面上[0，100]×[0，100] 的区域被若干隔板划分为许多小房间，每个房间的每面墙壁的中点是一扇门。现有一宝藏放在某个房间某个位置，问人从这个区域外至少经过几扇门才能找到宝藏。例如图 3-13 为至少经过 2 扇门。注意浮点误差。

3.1.6　自行车路线[②]

平面上有一些矩形建筑物，一个人骑车从起点到终点，不能穿越任何建筑物，但可以沿着建筑物的边骑车，如图 3-14 所示。这里，我们先假设任意两个建筑物没有公共点。

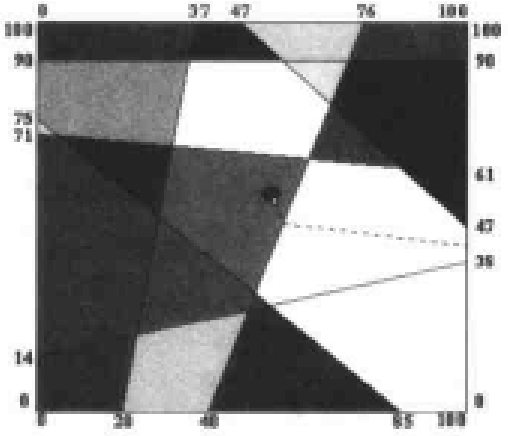


图 3-13　Treasure Hunt

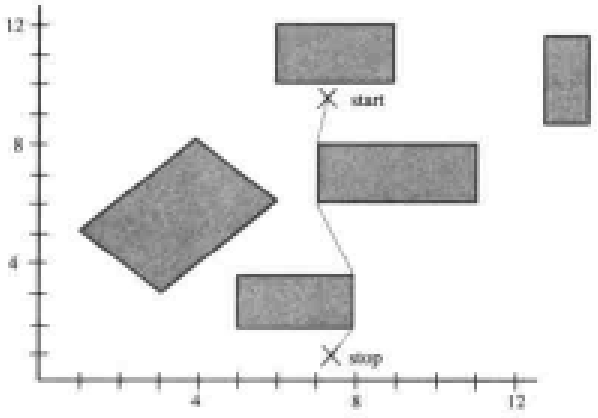


图 3-14　自行车最短路线

本题看似简单，但请你把算法设计得尽可能严密一些，例如图 3-15 的情况你考虑了吗？

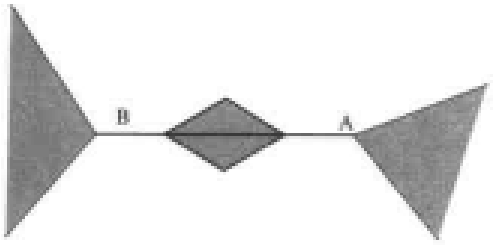


图 3-15　特殊情况：*AB* 恰好穿越一条对角线

变化 1：可以把矩形扩展为凸多边形（参见 3.2 节），你的算法要做哪些改动？

变化 2：如果允许建筑物相交，你的算法还成立吗？如果不成立，要加一个什么判断？你现在能实现吗？如果不能请参见 3.2.3 节。

3.1.7　篱笆视角问题[③]

有一个封闭的篱笆（平面上简单多边形，如图 3-16 所示），现有一光源在原点(0,0)，问光源照亮了多少角度的篱笆？例如：

---

① 题目来源：UVA Problem Archive 754
② 题目来源：ACM/ICPC World Finals 1996, Cutting Corners
③ 题目来源：UVA Problem Archive 667

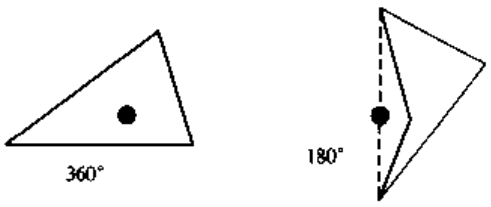


图 3-16　篱笆问题的例子

原题引入时，是先通过照度定义和积分的，但最后可以化为求照亮的角度范围。

（提示：本题方法比较巧妙，而且可用后文判断一个点是否在点集的凸包内，参见习题 3.3.7）

3.2　多边形和多面体的相关问题

以上我们所讨论的几何对象还都是最简单的点和线段（直线、射线等），而日常生活和计算几何中常见的还有另一类简单的几何体如多边形，及其在高维的拓广——多面体甚至 n 维形。当然，本书主要涉及前二者。虽然多边形和多面体非常简单，但是，抽象而严密地定义它们并非易事，特别是它们还有许多变化和特殊情况。以下从最简单的多边形出发，讨论面积（体积）、重心以及点在形（体）内、形（体）外的判别。这些知识和算法也是计算几何中最基本、最常用的，而且是下一节凸包算法的基础。本节主要讲述理论和方法，例题不多，因为这些知识本身很容易被广泛应用，所以我们在本节习题里给出了不少有趣的题目，留给读者揣摩。

3.2.1　卫兵问题——多边形和多面体的概念

1．多边形、简单多边形

多边形的定义　直观地讲，一个多边形就是二维平面上被一系列首尾相接、闭合的折线段围成的区域。抽象一些，可以给出以下定义[86]：令 P_0，P_1，…，P_{n-1}，为平面上的 n 个点。$E_0=P_0P_1$，$E_1=P_1P_2$，…，$E_i=P_iP_{i+1}$，…，$E_{n-1}=P_{n-1}P_0$ 为 n 条线段[①]。

这 n 条线段围成一个多边形，当且仅当：

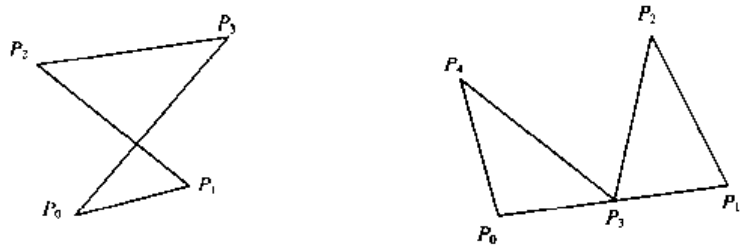
任何两条相邻线段 E_iE_{i+1} (i=0~n−1)有且仅有一个公共点，即端点 P_{i+1}。　①

多边形分类　考虑图 3-17 中图形，它们是多边形吗？

这就涉及多边形的一个分类问题。图 3-17 这种自相交（即不相邻线段有公共点）的多边形称为“**复杂多边形**”。它们不满足：

$$\forall j \neq i+1, e_i \cap e_j = \phi \qquad ②$$

[①] 注意我们此处已默认 $p_i=p_{i \bmod n}$，即循环模，p_n 就是 p_0，当然也可以写成从 1~n 的形式，不过从数学的严格性来说，以 0~n-1 这种写法为好



(a) 复杂多边形　　(b) 邻界多边形

图 3-17　复杂多边形和邻界多边形的例子

而一般意义上的自身不相交的多边形称为“**简单多边形**”。它同时满足①、②两个条件。

由于今后见到的大部分是简单多边形，所以后文若不特别说明，“多边形”一词指简单多边形。请注意一种特别情况：**临界多边形**（见 3-17(b)），根据我们的定义，它也属于复杂多边形，不过从几何性质上讲，它更多地接近简单多边形。

2. 内外的定义和排列方向

为了输入一个多边形，必须规定一个正方向，例如可以简单地规定逆时针方向为正，但是对于很复杂的图形，这种规定比较难以操作，也不够严格。

右手螺旋方向　一种比较严密的定义是数学分析中常用的右手螺旋方向，直观地讲，就是假设一个人沿着多边形走，如果每时每刻，多边形的内部都在其左手方向，那么这个走向就是正方向，这和叉积的正方向是一致的——当用右手的四个手指沿多边形正方向时，拇指就垂直纸面向上，这就是整个多边形这个面的外法线方向，如图 3-18 所示，这种定义方法将在下文多面体的方向定义中起重要作用。

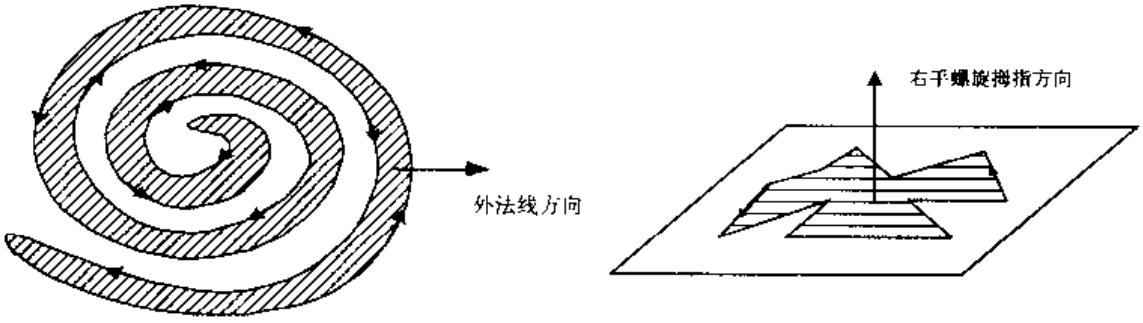


图 3-18　右手螺旋方向和外法线方向

内部的定义　这种定义固然很优美，但却是以“内部”的定义为前提的。对于很复杂的图形，有时是很难一下子识别内外部的。一个数学上比较优美的定义方法是：根据 Jordan 曲线定理，一个简单的闭合的平面曲线必分平面为两个区域，其中一个是有限区域——内部，另一个无限区域就是外部。当然，也定义无穷远处 $P(\infty, \infty)$为外部，根据连续性，与 P 连通的点都是外部，剩下的除了边界就内部。

以上两个定义都不太适合计算机处理，因为计算机没有形象思维能力，没有“围成区域”的概念，当然也不能从正方向来定义内部（循环定义）。

其实，先定义内部还是先定义正方向，也是可以商榷的。至于方向和内部的代数化（可操作的）定义我们留在下文展开。先定义内部的可用射线法或转角法，参见后文 3.2.3 小节“判点在形内、形外”；先定义方向的可用面积法，参见后文 3.2.2 小节“求面积、体积、重心”。

注意对复杂多边形，上述方法均不适用，因为复杂多边形不是一个简单闭合曲线，其内部的定义本来就不清楚，十分复杂，限于篇幅，留给读者思考。

3．凸多边形、星形多边形

凸多边形　平面几何中定义过一种常见的特殊多边形——凸多边形。直观的定义如下：对任何一条边 e，整个多边形在 e 的一侧（如果是正方向，则必是左侧）。严格的代数化定义留在后文 3.3.1 小节“凸包定义”展开。凸多边形有很多优美的性质，为很多问题带来了极大的方便。

星形多边形　还有一种比凸多边形更一般，但比一般多边形特殊一点的多边形——星形多边形。直观的定义是：存在多边形内部一点，它能“看到”多边形内所有点。所谓“**看到**”，直观的比方，就是把多边形想象成一个房间，墙（多边形的边）是不透明的，光线直线传播，于是“看到”指视线不被阻挡。所有满足定义要求的点组成的区域称为星形多边形的“**核**”。一个星形多边形一般是由核和围绕着“核”的尖角构成，如一个五角星形状的多边形。抽象的讲法是一个星形多边形总与一个圆同态。很明显，凸多边形内任一点都可“看到”所有点，所以，凸多边形是星形多边形，其内部区域就是“核”，如图 3-19 所示。

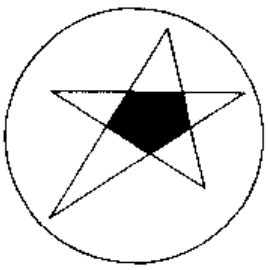


图 3-19　星形多边形的核：星形多边形总与一个圆同态

这里需要把上述“看到”的直观定义严密化一下，我们说在多边形内的两点 x、y 能互相看到，当且仅当线段 xy 全在 P 内，即 $xy \subseteq P$。但是请注意一种特殊情况：xy 恰经过“墙上”一点。这是一种临界情况，为了去掉这种情况，定义一种新的“看到”，称为“严格看到”或“清晰看到”（clear visibility）。

xy 能互相严格看到 $\Leftrightarrow xy \subseteq p \wedge (xy \bigcap \partial p) \subseteq \{x, y\}$

卫兵问题　关于星形多边形和“看到”的概念，有一个有趣的问题：卫兵问题。主要是讲在一个博物馆（可以想象为一个多边形）的墙上，挂满了名画，问至少放几个卫兵才能“看到”所有的墙壁（名画）。这个问题有很多变种，下面我们讨论几个代表性的，其他的留作习题。

（1）是否一个卫兵就够了，如果是，那么应放在什么地方？

为了简单起见，首先把题目限制为各边均为水平或竖直，如图 3-20 所示。一个重要的

结论是：对于任意一条边（墙），为了看到其内侧（即面向多边形内部的一侧——墙上的名画），卫兵至少要站在这条边的**内侧半平面**内。基于这个分析，需要把每条边对应的内侧半平面求交，即能得到所求区域。当然，任何一步若发现交集为空，则不必再做下去了。对于水平、竖直这种特殊情况，求交集变得非常简单，因为任何区域都是平行于轴的长方形，于是只要知道上、下、左、右四个边界位置 *minx*，*maxx*，*miny*，*maxy* 即可。对于任何一条边，注意我们是按正方向输入的，于是仅凭这条边的走向即可知道内部在什么方向。例如，对于向右走的边 $(x_1,y)\rightarrow(x_2,y)$ $(x_2>x_1)$，内部在上方，于是 *maxy*=min(*maxy*, *y*)。当然，一开始可设 *minx*=*miny*=$-\infty$，*maxx*=*maxy*=$+\infty$，∞可设为一个足够大的数。

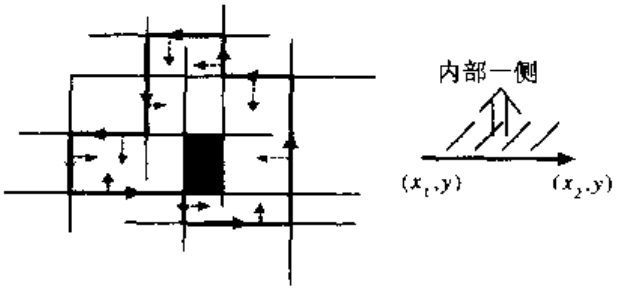


图 3-20　卫兵问题的水平竖直情形

于是，对于四种边：

			内侧	操作
向右	$(x_1,y)\rightarrow(x_2,y)$	$(x_1<x_2)$,	上	minit (*maxy*,*y*)
向上	$(x,y_1)\rightarrow(x,y_2)$	$(y_1<y_2)$,	左	minit (*maxx*,*x*)
向左	$(x_1,y)\rightarrow(x_2,y)$	$(x_1>x_2)$,	下	maxit (*miny*,*y*)
向下	$(x,y_1)\rightarrow(x,y_2)$	$(y_1>y_2)$,	右	maxit (*minx*,*x*)

定义 minit(*a*,*b*)为 *a*=min(*a*,*b*), maxit (*a*,*b*)为 *a*=max(*a*,*b*)

以上我们完整地解决了水平、竖直情况下的问题。

对于一般情况的多边形，方法还是一样的：每次把一个凸多边形与一个半平面求交（因为半**平面**也是凸的，凸图形的交必为凸），这是一个很基本很重要的问题，但是现在工具储备不够，留待后文 3.4.5 小节详细讨论。

（2）**卫兵与核**　肯定有读者会问，这个问题（是否存在一个点能看到所有边上点）似乎与判断一个多边形是否是星形多边形（是否存在一个点能看到所有内部点）等价，这个卫兵的区域就是多边形的核。但是，我们必须证明其“等价性”。

首先，如果卫兵能看到多边形内所有点，显然也能看到边上所有点。

反向的问题稍微困难一点，如果一个卫兵能看到所有的边（内侧），则以这个卫兵为中心，剖分多边形，则卫兵能看到每条边与他自己形成的三角形区域内所有点。这种剖分显然能覆盖整个多边形，因此卫兵能看到所有点，即是一个星形多边形，如图 3-21 所示。

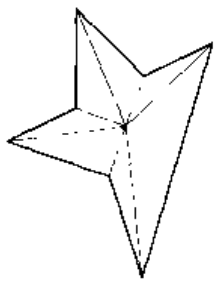


图 3-21　卫兵能看到所有点→星形多边形

（3）多个卫兵　一个更加有趣的问题是，如果有多个卫兵，这两个问题是否还等价呢？具体地说，如果一组卫兵能看到所有边，是否多边形内任一点都被至少一个卫兵看到呢？

直觉好像总认为这两者是等价的，但是有反例。两个比较简单的例子是“四角飞镖”和 H 的形状。如图 3-22 所示，四个卫兵分别站在标为黑点·的角上，可以认为卫兵的位置无限靠近该角，这四个卫兵确实能看到所有的边，但是中间阴影区域却不被任何卫兵看到。

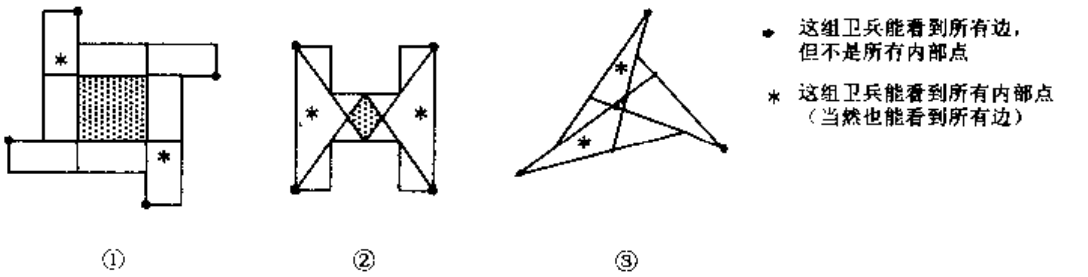


图 3-22　卫兵问题：看到所有边不一定能看到所有点（飞镖形例子）

当然，可以进一步简化为三角形的形状，只要 6 个顶点，这可能已经是最简单的例子了。

（4）至少几个卫兵？　卫兵问题的另外变种是问一般情况下，给定一个多边形，至少放几个卫兵就能看到①所有边上的点，②所有内部点。

另外，在最优情况下，①、②是否等价？注意图 3-22 中的三例都不是最优情况，其实这些图形都只要两个卫兵就能看到所有内点（*位置）。

问题①的解称为 G(边)，问题②的解称为 G(内)，显然 G(边)≤G(内)，我们要问是否有 G(边)=G(内)？留给读者思考。

关于卫兵问题，还有一些著名的变种，参见习题，或者文献[86]。

4. 多面体

我们很容易简单地认为，多面体是多边形在三维空间的简单推广，比如可以仿照多边形的定义，定义多面体为：一系列“面”构成，每个面都是一个简单多边形，面与面之间满足以下三种情况之一：

（1）它们分离；

（2）它们仅有一个公共点；

（3）它们有两个公共点以及连接这两个顶点的公共边。

多面体的复杂性　事实上有些很简单的图形却不能融入上述定义，例如一个三棱柱上面出个角（小三棱锥），如图 3-23 所示，注意三棱柱上底面成了一个有洞多边形，它甚至连前面“多边形”小节中定义的复杂多边形都不是。

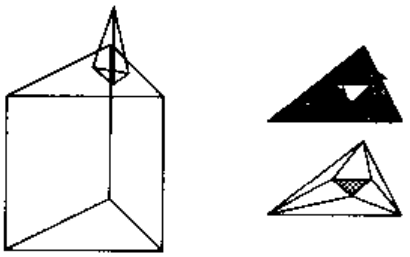


图 3-23　多面体定义的问题及两种解决方案：有洞多边形和剖分

解决这个问题的方法有两个：

（1）定义有洞多边形，即洞的边界方向按数学分析中的定义（就是仍然按“沿边界”行走时内部都在左侧）。有洞多边形还是能保持很多很好的性质（如面积、内外）。有洞多边形因为围成的仍是一个闭合区域，所以它比复杂多边形性质更好一些。

（2）第二种方法是权宜之计，但可以不定义有洞多边形，其实就是规定多面体的每个面都是简单多边形，甚至凸多边形、三角形，我们可以把每个面上的多边形（包括有洞的）分解为凸多边形（或简单多边形、三角形），当然，这需要允许相邻面共面。

我们采用了（2），主要因为尽管理论上说不如（1）漂亮，但是输入起来比较方便。

这还不够，还要排除几种不正常的情况，于是，我们还要加上两点限制：

- 局部拓扑：简而言之，就是每一条边都被恰好两个面共享。
- 全局拓扑：简单的定义是多面体的表面是二维连续区域，即多面体表面等价于一个平面图。

以上是直观的刻画，欲了解严格的定义，请读者参考一些拓扑学著作。

事实上，重要的不是定义，而是多面体的表示。各种不同的表示方案，复杂程度不同，所蕴涵的几何和拓扑信息也不同。这个内容留待稍后讨论，此处先讲两类特殊多面体——凸多面体和星形多面体。

5. 凸多面体和星形多面体

和二维多边形一样，这两种多面体也有很好的性质。下面用类似的方法来定义和讨论它们。

- **凸多面体**：多面体在任意一个面的一侧（内侧），其中内侧即面的内法线方向。至于其严格代数化的定义，也在“凸包”一节中与二维情形同时给出。
- **星形多面体**：多面体内存在一个点，能看到多面体内所有点（包括边界），这种点所形成的区域称为“核”，求核的方法也与平面时一致，不过推广为半空间交集。当然三维卫兵问题会有更多变种，这里就不讨论了。

6. 多面体的表示

最关键的问题是如何在计算机中简便地表示一个多面体，又要尽可能地体现几何和拓扑信息。

点-面模型　最简单的表示法只考虑局部的几何信息，而忽略拓扑信息。例如最原始的方法是表示为若干个“面”，每个面再表示为一个简单多边形。注意，顶点的顺序按照前文“多边形定义”规定的方向：多边形的内部在左手方向。这其实就是外法线方向，即当右手四指沿边的方向时，拇指指向多面体的外部。

由于多面体的顶点重复较多，一个顶点可能同时属于好几个面，所以可加一个改进：只保存顶点的索引，而不保存其坐标，这样就得到一个最简单的**“点-面模型”**，如图 3-24 所示。

这个模型可以处理大多数简单的问题，如体积、判断体内体外等，但有两个明显的缺陷：

- 一条边既然属于两个面，自然就被重复表达了两次（尽管方向相反）。
- 没有反映拓扑信息，如面的相邻、一条边与哪两个面相邻等。

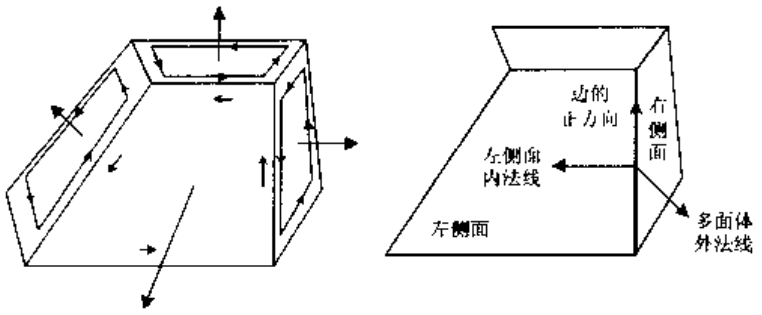


图 3-24　多面体的表示：点-面模型和点-边-面模型

点-边-面模型　一个改进的模型为**点-边-面模型**。对于每条边，首先任意规定一个方向为正方向，然后每个面在引用边时，要指明正方向还是反方向，同时为了表达局部拓扑信息，每条边要指出两个相邻面，其中第一个面在正方向的左侧，另一个在右侧（反方向的左侧）。

注意：所谓左侧，直观地讲，就是人站在多面体外，身体与这条边平行，头顶朝正方向，面向多面体，则左手所指即左侧。严格地讲，就是在这条边上，多面体外法线、边的正方向和左侧面的内法线成右手螺旋。

如果要进一步表达全局拓扑信息，就要利用多面体表面与平面图的对应性（欧拉定理），这些模型比较复杂，应用也较高深，在此就不介绍了。请读者参考有关图形学书籍。

3.2.2　求多边形、多面体的容积和重心；高维情形

本小节中，将看到叉积思想的又一应用。我们已经知道，叉积的结果是一个向量，包含两个方面：一是方向，二是数值大小——代表面积。前面，已经讨论了方向的应用。这里将看到两者结合，用于多边形面积的计算，并把它简单地拓广到多面体的体积。同时，将应用同样的思想来解决和面积紧密相关的问题——重心。

基本问题：给定一个简单多边形，求其面积和重心。并拓广到三维情形。

输入：多边形（顶点按逆时针顺序排列）。

输出：面积[1] S 和重心点 C。

这个问题乍看起来很简单，其实不然。当然，我们总可以用 Montc-Carlo 法，结合后文判形内形外（3.2.3 小节）的方法来模拟逼近，但这办法太麻烦，又不精确，关键是还不能体现几何本身的美，所以还是来探究计算几何的简单做法。

我们暂时无法求 N 个点的多边形的面积，那就先来看最简单（点数最少）的一种多边形——三角形。下面，首先讨论三角形的面积和重心，然后把一个多边形划分为若干个三角形，从而解决 N 个点的多边形的面积和重心。

1. 三角形面积和重心

对于面积，我们仍从计算几何和解析几何两个方向分别讨论。

[1] 这里约定：S 代表面积（绝对值），A 代表有向面积，即 $S=|A|$

计算几何方面：我们知道，$\triangle ABC$ 的面积就是 $\overrightarrow{AB}$ 和 $\overrightarrow{AC}$ 两个向量叉积的绝对值的一半。其**正负**表示三角形顶点是在右手系还是左手系，或者说是逆时针走向，还是顺时针走向，如图 3-25 所示。

(a) ABC 成左手系，负面积　　(b) ABC 成右手系，正面积

图 3-25　有向面积

$$\text{Area}(A,B,C)=\frac{1}{2}\overrightarrow{AB}\times\overrightarrow{AC}=\frac{1}{2}\begin{vmatrix}x_A & y_A & 1\\ x_B & y_B & 1\\ x_C & y_C & 1\end{vmatrix}=\begin{vmatrix}x_B-x_A & y_B-y_A\\ x_C-x_A & y_C-y_A\end{vmatrix}\qquad(*)$$

在下文将会看到，**有向面积**的概念和上述公式将会产生奇效，当然，为了得到这个公式，解析几何的办法也是可行的：用 $S=\frac{1}{2}|AB|\cdot h$ 的面积公式，写出直线方程，求 C 到 AB 的距离即可。只是这个方法推导较繁，关键是还不能得到有向面积，在此就不赘述了。

面积解决之后，再来讨论三角形的重心，这里采用解析几何的办法。我们知道，重心就在 AB 边上的中线 CM 的 2/3 分点 P 处，如图 3-26 所示，因此，取 AB 的中点 $\overrightarrow{M}=\frac{\vec{A}+\vec{B}}{2}=\left(\frac{x_A+x_B}{2},\frac{y_A+y_B}{2}\right)$，在 $\overrightarrow{CM}$ 上应用定比分点公式

$$\vec{P}=\frac{2\overrightarrow{M}+\vec{C}}{3}=\left(\frac{x_A+x_B+x_C}{3},\frac{y_A+y_B+y_C}{3}\right)=\frac{\vec{A}+\vec{B}+\vec{C}}{3}$$

也就是说，三角形的重心在三顶点坐标的代数平均数处，聪明的读者也许会想，这不就是质心系重心公式吗？为什么此处仍适用？那么它是否还适用于 N 个点的多边形呢？我们将在下文详细讨论。

2．凸多边形的三角形剖分

有了三角形的基础，现在考虑 N 个顶点的多边形。

先考虑凸多边形，如图 3-27 所示。

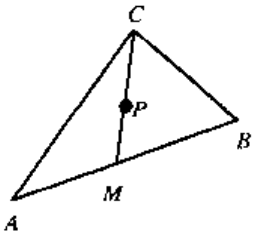


图 3-26　三角形的重心

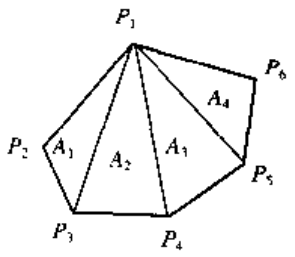


图 3-27　凸多边形的三角形剖分

很自然地，我们会想到以 P_1 为扇面中心（本节顶点编号为 1~n），连接 $P_1P_i(i=3\sim n-1)$，就得到 $N-2$ 个三角形，由于凸性，保证这些三角形全在多边形内，那么这个凸多边形的（有向）面积 $A=\sum_{i=1}^{N-2}A_i$。

于是，$A=\sum_{i=1}^{N-2}A(p_1p_{i+1}p_{i+2})=\sum_{i=1}^{N-2}\frac{1}{2}\begin{vmatrix}x_1 & y_1 & 1\\ x_{i+1} & y_{i+1} & 1\\ x_{i+2} & y_{i+2} & 1\end{vmatrix}$，将公式展开，消去重复项，得

$$A=\frac{1}{2}\left[\begin{vmatrix}x_1 & y_1\\ x_2 & y_2\end{vmatrix}+\begin{vmatrix}x_2 & y_2\\ x_3 & y_3\end{vmatrix}+\cdots+\begin{vmatrix}x_n & y_n\\ x_1 & y_1\end{vmatrix}\right]$$

如果我们令 $P_{N+1}=P_1$（在 mod 意义下是显然的），则上式进一步简化为

$$A=\frac{1}{2}\sum_{i=1}^{N}\begin{vmatrix}x_i & y_i\\ x_{i+1} & y_{i+1}\end{vmatrix}\qquad(***)$$

这个公式非常简洁，那么能否推广到一般的简单多边形呢？特别是对于这个优美的式子，你还能想到什么解释？还是留到后文讨论吧。

3. 凹多边形

下面我们考察凹多边形是否也能适应这个剖分方法。为简单起见，先考察最简单的凹多边形——凹四边形的例子。如图 3-28 所示，考虑同样的剖分方案，把多边形分为$\triangle P_1P_2P_3$和$\triangle P_1P_3P_4$，这时我们发现这就没有凸多边形那么好的性质了，$\triangle P_1P_2P_3$ 部分在多边形外，而$\triangle P_1P_3P_4$ 整个在外，其实 $S=S(\triangle P_1P_2P_3)-S(\triangle P_1P_3P_4)$，那么我们就用割补法，但是哪些三角形面积该取正，哪些该取负呢？

聪明的读者肯定会想到用叉积判断，右手系取正，左手系取负，这时，我们联想到“负面积”的概念，自然，这里 $\triangle P_1P_3P_4$ 就构成左手系，本就应该取负号，于是原来的方法仍是对的，虽然 $S=\sum_{i=1}^{N-2}S_i$ 不再成立，$A=\sum_{i=1}^{N-2}A_i$ 却依然成立。这里，通过“负面积”再一次看到了叉积的好处，并且认识到“有向面积”A 比“面积”S 其实更本质。

显然一般简单凹多边形也适用。如图 3-29 所示。

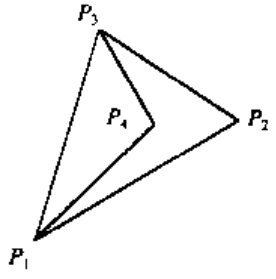


图 3-28　凹四边形的三角形剖分

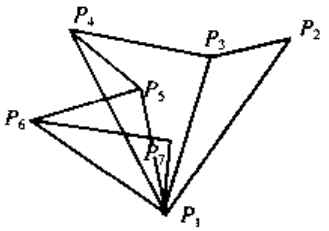


图 3-29　一般凹多边形的三角形剖分

$S=S_1+S_2-S_3+S_4-S_5$，$A=A_1+A_2+A_3+A_4+A_5$，所以，总的 $A=\frac{1}{2}\sum_{i=1}^{N}\begin{vmatrix}x_i & y_i\\ x_{i+1} & y_{i+1}\end{vmatrix}$ 对简单多边形都适用。需要特别指出的是，与三角形一样，这里最后结果 A 值的正负号也对应了多边形

顶点的排列方向：若结果为正则说明输入是正方向（逆时针），结果为负说明反方向（顺时针）。

其实，这就可以用来定义多边形的正方向。前文是从 Jordan 曲线定理和“有限区域”定义了内部，然后再定义正方向。现在，我们通过行列式和的正负号来定义正方向，然后再由“内部在左手侧”来定义内部。与前文的方法相比，这个方法更适合计算机，因为它不需要“围成区域”的形象思维，而只要一个简单的计算即可。

4．梯形剖分

是否还有其他办法呢？聪明的读者很快会想到，三角形很特殊，四边形也很简单啊，那么能不能用四边形来剖分多边形呢？关键是用哪一种四边形，在四边形中，平行四边形太特殊，一般四边形又太一般，这时不难想到折衷形——梯形，如果把梯形的平行边竖直放，是否就把多边形切成竖直条状片了呢？更简单地，我们只要直角梯形，直角边对齐 x 轴，放成如图 3-30 形状，绕多边形各顶点兜一圈，就求出面积了[85]。

这个方法就是非常重要的竖条分析（Vertical Stripes）法，这里不详细讨论，放在后文特殊算法（见 3.4.2 小节）中详细分析。

这样作图，明显也是考虑“负面积”的概念，绕形兜一圈，正负抵消，恰好把形外的部分去掉，得到多边形面积。具体地，对每一单个梯形，如图 3-31 所示。

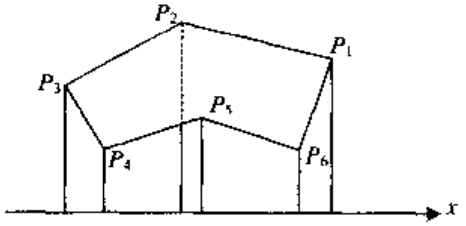


图 3-30　梯形剖分

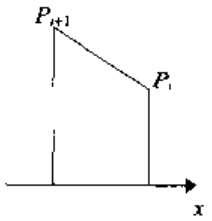


图 3-31　梯形剖分的单个梯形

梯形面积 $A_i=\frac{1}{2}(y_i+y_{i+1})(x_i-x_{i+1})$，而 $A=\sum_{i=1}^{N}A_i=\frac{1}{2}\sum_{i=1}^{N}(y_i+y_{i+1})(x_i-x_{i+1})$ 展开，消去重复项，得 $A=\frac{1}{2}\sum_{i=1}^{N}\begin{vmatrix}x_i & y_i\\ x_{i+1} & y_{i+1}\end{vmatrix}$。

这个方法似乎要简单一些，而且，与前面的三角形剖分不同，这是外剖分，即剖分的基线（点）不在形上，例如这里是 x 轴，其实任意平行于 x 轴的直线都一样，可作剖分基线。如图 3-32 所示。

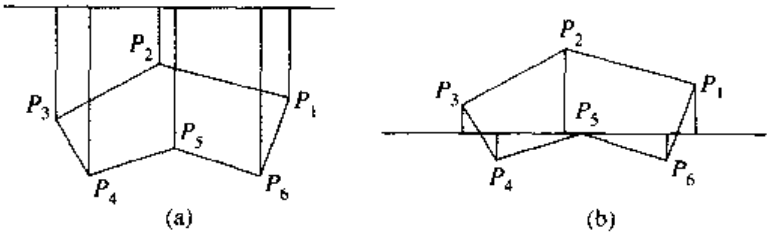


图 3-32　梯形剖分的另外两种情况

请读者特别考虑图 3-32(b)，因为其中有图 3-33 所示的复杂多边形情形，这有利于加深对“负面积”的理解。

5. 任意点为扇心的三角形剖分

还有没有更简单的方法呢？我们能把多边形分成 N-2 个三角形，为什么不能分成 N 个三角形呢？例如对于凸多边形，严格地说，是对于星形多边形，以其核内一点为扇心，可剖分成 N 个三角形，如图 3-34 所示。

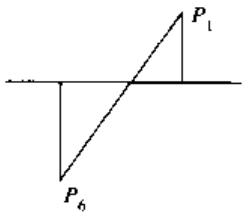


图 3-33　梯形剖分的复杂多边形

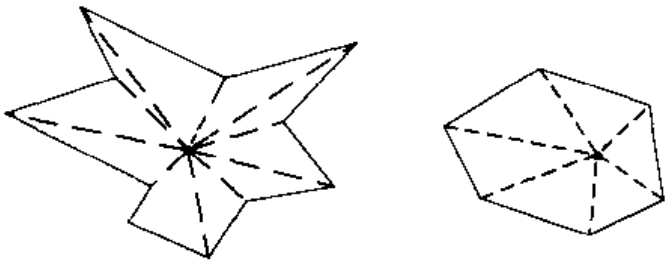


图 3-34　星形多边形和凸多边形：核内一点剖分

其实，这对于一般简单多边形内部一点也成立（即扇心不一定在核内），如图 3-35 所示。

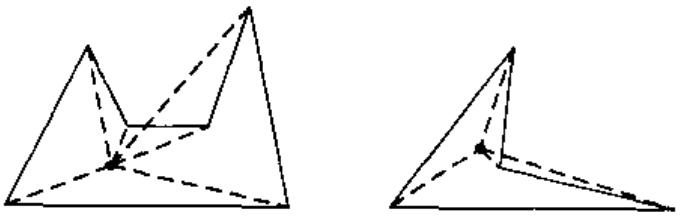


图 3-35　一般简单多边形内部一点剖分

这时我们发现，以 P_1 为扇心划分三角形办法仅是一个特例，即$\triangle P_1P_1P_2$ 和$\triangle P_1P_NP_1$ 退化为线段了。

再进一步，能不能仿照梯形剖分，把扇心移到任一点（即可在形外）呢？画一个图，如图 3-36 所示，稍作思考，就会发现这是显然的，当然，为了严密，也可从三角形开始，作一个归纳法证明，不过鉴于这个证明非常显然，又可加深对“负面积”的理解，故留给读者。

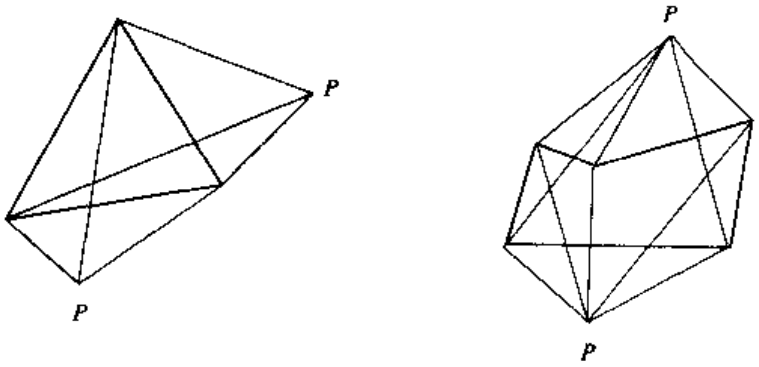


图 3-36　扇心在形外的剖分

既然可以把扇心放在任一点，为什么不放在原点呢（这与梯形剖分把准线放在 x 轴一样）？这样可以简化运算，现在

$$A=\sum_{i=1}^{N}A(\Delta OP_iP_{i+1})=\frac{1}{2}\sum_{i=1}^{N}\overrightarrow{OP_i}\times\overrightarrow{OP_{i+1}}=\frac{1}{2}\sum_{i=1}^{N}\begin{vmatrix}x_i & y_i\\ x_{i+1} & y_{i+1}\end{vmatrix}$$

果然非常简单，现在我们一步得到（***）式。

聪明的读者可能会发现，这个过程太麻烦了，其实只要从（***）式出发，直接考虑其几何意义即可。$\begin{vmatrix}x_i & y_i\\ x_{i+1} & y_{i+1}\end{vmatrix}=\begin{vmatrix}x_i-0 & y_i-0\\ x_{i+1}-0 & y_{i+1}-0\end{vmatrix}$ 本身就是起点在原点的两个向量的叉积，其几何意义是△OP_iP_{i+1} 的有向面积的 2 倍。

当然，这既是对（***）式一个最好的解释，也是计算面积最简便的一个方法。这个算法非常容易实现，复杂度为 $O(n)$。

现在，我们通过这个漫长的探索过程，经过了三角形面积→以 P_1 为扇心剖分凸多边形→凹多边形→梯形剖分→核内与核外剖分→任意点的三角形剖分→以原点为扇心的三角形剖分的历程。从中可看到，探索的过程往往是曲线发展的，走了不少弯路，以至有种“众里寻他千百度”的感觉。

6. 多边形的重心：猜想——反例——方法——公式

猜想公式

我们解决了面积，自然也可以用同样的方法解决重心。但是请想想，还有没有更简单的、不通过剖分多边形的办法呢？很容易想到，既然三角形的重心是 $(\frac{x_1+x_2+x_3}{3},\frac{y_1+y_2+y_3}{3})$ 也就是三顶点的代数平均数，那么这个公式能否推广呢？也就是说对一个 N 个点的简单多边形，它的重心是否也在其顶点的代数平均数，即 $(\frac{\sum_{i=1}^{N}x_i}{N},\frac{\sum_{i=1}^{N}y_i}{N})$，或者抽象地，$\vec{C}=\frac{\sum_{i=1}^{N}\vec{P_i}}{N}$ 呢？照例，如果你认为是对的，怎么证明呢？如果你认为是错的，反例呢？

反例

相信读者很快会找到反例，反例其实很简单也很普遍。例如最简单的反例是——梯形，如图 3-37 所示，对于平行边平行于 x 轴的梯形来说，如果上边小于下边，那么其重心应该在中位线以下（考虑极端情况如退化为三角形）。而根据上述猜想公式计算，却恰在中位线上，显然是不对的。

梯形的例子　　退化为三角形的情形

图 3-37　反例 1：梯形

（图中黑点为实际重心大致位置）

再举一反例，考虑一个圆内接多边形，如果上部点取得很密集，下部很稀疏（如图 3-38 所示），则根据上述猜想公式计算，重心应很偏上。而我们直观的估计，则显然重心应在中部附近，考虑极限情况，若上部一个小区间内点无限多（即进化为圆弧），则根据猜想公式，重心应在此圆弧附近，这显然是荒谬的。

上部密集，下部稀疏　　上部进化为圆弧

图 3-38　反例 2：圆内接多边形

（图中黑点为猜想公式重心大致位置）

两种均匀分布的不同　通过这个反例，我们进一步认识到，这个公式之所以错，是因为它其实是质点系重心公式，即如果认为多边形的质量仅分布在其顶点上，且均匀分布，则这个公式是对的。但现在多边形的质量是均匀分布在其内部区域上的，也就是说，是与面积有关的，而上述公式仅考虑顶点质量分布，显然会得到荒谬结果。下面，回到基本思路——面积与多边形的剖分来继续讨论，并且在末尾还会得到一个与面积公式类似的重心公式。

有了关于面积曲折探索的过程，在这里就不用再从头开始了，只要从最方便的路径——原点的三角形剖分出发，便可容易求出重心公式。

加权平均　同样剖分成 N 个三角形，分别求出其重心和面积，这时可以想象，原来质量均匀分布在内部区域上，而现在质量仅分布在这 N 个重心点上（等价变换），这时就可以利用刚才的质点系重心公式了。只不过要稍微改一改，改成加权平均数，因为质量不是均匀分布的，每个质点代表其所在三角形，其质量就是该三角形的面积，注意是有向面积——这就是权。

于是

$$C=\frac{1}{A}\sum_{i=1}^{N}A_i\overrightarrow{C_i}$$

$$\overrightarrow{C_i}=\text{Centroid}(\triangle\frac{\vec{O}+\overrightarrow{P_i}+\overrightarrow{P_{i+1}}}{3}=\frac{\overrightarrow{P_i}+\overrightarrow{P_{i+1}}}{3})$$

所以

$$\vec{C}=\frac{1}{A}\sum_{i=1}^{N}(\overrightarrow{P_i}+\overrightarrow{P_{i+1}})(\overrightarrow{P_i}\times\overrightarrow{P_{i+1}})$$

写成分量式：

$$\begin{cases}C_x=\dfrac{1}{6A}\displaystyle\sum_{i=1}^{N}(x_i+x_{i+1})\begin{vmatrix}x_i & y_i\\ x_{i+1} & y_{i+1}\end{vmatrix}\\ C_y=\dfrac{1}{6A}\displaystyle\sum_{i=1}^{N}(y_i+y_{i+1})\begin{vmatrix}x_i & y_i\\ x_{i+1} & y_{i+1}\end{vmatrix}\end{cases}$$

聪明的读者一定发现，重心还有比面积更好的性质——与输入顶点的逆时针和顺时针无关，因为面积的正负号恰好抵消了。

还需要提醒读者的是，从以前讲过的其他剖分方法也可以求得这个公式，作为练习，请读者试从梯形剖分出发推导这个公式。

7. 从三角形到四面体

现在，回到前面提到的一个问题：为什么三角形的重心公式恰与质点系重心公式吻合呢？好像三角形内部的质量全分布在三个顶点上一样。我们知道，三角形的特殊之处在于，三个点是平面上构成多边形所需的最少点数。而且与 N 边形($N\geq 4$)不同，它不需要有逆时针序和顺时针序，即能惟一确定一个三角形，而 $N\geq 4$ 时，要 N 个有序的点才行。另外，三角形是稳定的，即它只有凸图形，而 $N\geq 4$ 的多边形则不稳定（即可凹陷、凸出），如图 3-39 所示。

四面体的体积

现在我们放眼三维情形。三维空间中，四个点是构成多面体最少的数目，因此四面体（即三棱锥）也与三角形一样，有稳定性和唯一确定性等性质。下面我们从容积（长度、面积与体积的一般化）角度再来进一步分析。考虑图 3-40 中的四面体 $ABCD$。

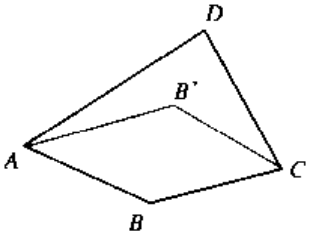


图 3-39　四边形的不稳定性：可凹陷、凸出

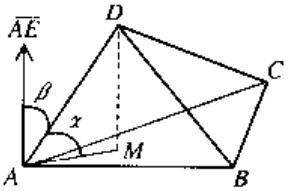


图 3-40　四面体的体积

由立体几何知识得 $V=\frac{1}{3}S_{\triangle ABC}h$

其中 h 是 D 到 ABC 面的距离，设 M 是 D 在 ABC 面上的投影，则 $h=|MD|$，

$S_{\triangle ABC}=\frac{1}{2}\left|\overrightarrow{AB}\times\overrightarrow{AC}\right|$，$h=\left|\overrightarrow{AD}\right|\cdot\sin\alpha$，其中$\alpha$为$\overrightarrow{AD}$与$\overrightarrow{AM}$的夹角。

考虑叉积，记 $\overrightarrow{AE}=\overrightarrow{AB}\times\overrightarrow{AC}$，因为 $\overrightarrow{AE}$ 上面 ABC，所以 $\overrightarrow{AE}$ 与 $\overrightarrow{AD}$ 的夹角 $\beta=\frac{\pi}{2}-\alpha$

这就为求 $\sin\alpha$ 提供了绝好的途径，用点积 $\cos\beta=\frac{\overrightarrow{AD}\cdot\overrightarrow{AE}}{\left|\overrightarrow{AD}\right|\cdot\left|\overrightarrow{AE}\right|}$ 代入，得

$$V=\frac{1}{3}S_{\triangle ABC}\cdot h=\frac{1}{3}\cdot\frac{1}{2}\left|\overrightarrow{AB}\times\overrightarrow{AC}\right|\cdot\left|\overrightarrow{AD}\right|\cdot\left|\cos\beta\right|=\frac{1}{6}\left|\overrightarrow{AB}\times\overrightarrow{AC}\right|\cdot\frac{\left|\overrightarrow{AD}\right|\cdot\left|\overrightarrow{AD}\cdot\overrightarrow{AE}\right|}{\left|\overrightarrow{AD}\right|\left|\overrightarrow{AE}\right|}$$

把 $\overrightarrow{AE}=\overrightarrow{AB}\times\overrightarrow{AC}$ 代入，最终得 $V=\frac{1}{6}\left|\overrightarrow{AD}\cdot\overrightarrow{AE}\right|=\frac{1}{6}\left|\overrightarrow{AD}\cdot(\overrightarrow{AB}\times\overrightarrow{AC})\right|$

当然，与面积相仿，不加绝对值就是**有向体积**，$\overrightarrow{AB},\overrightarrow{AC},\overrightarrow{AD}$ 成右手系时为正，左手系时为负，共面时为 0。

混合积及其几何意义

这里，通过几何运算，得到了一个与三角形面积公式相仿的体积公式。同时还得到了

一种与叉积相仿的向量运算——$\overrightarrow{AD}\cdot\left(\overrightarrow{AB}\times\overrightarrow{AC}\right)$，这个形式叫**向量的混合积**。

根据叉积和点积的定义，容易得

$$\overrightarrow{AD}\cdot\left(\overrightarrow{AB}\times\overrightarrow{AC}\right)=\begin{vmatrix}A_x & A_y & A_z & 1\\ B_x & B_y & B_z & 1\\ C_x & C_y & C_z & 1\\ D_x & D_y & D_z & 1\end{vmatrix} \qquad (*)$$

如果平移使 A 到原点，记 $\vec{d}=\overrightarrow{AD},\vec{b}=\overrightarrow{AB},\vec{c}=\overrightarrow{AC}$

则 $\vec{d}\cdot\left(\vec{b}\times\vec{c}\right)=\begin{vmatrix}d_x & d_y & d_z\\ b_x & b_y & b_z\\ c_x & c_y & c_z\end{vmatrix}=\begin{vmatrix}b_x & b_y & b_z\\ c_x & c_y & c_z\\ d_x & d_y & d_z\end{vmatrix}$

对比二维意义下的叉积：$\overrightarrow{AB}\times\overrightarrow{AC}=\begin{vmatrix}A_x & A_y & 1\\ B_x & B_y & 1\\ C_x & C_y & 1\end{vmatrix}=\begin{vmatrix}b_x & b_y\\ c_x & c_y\end{vmatrix}$ $(\text{令}\vec{b}=\overrightarrow{AB},\vec{c}=\overrightarrow{AC})$

是何其相似啊！

$\vec{a}\cdot\left(\vec{b}\times\vec{c}\right)$的几何意义是一个数，等于以为 $\vec{a},\vec{b},\vec{c}$ 边的平行六面体（有向）体积，也恰等于6倍的以 $\vec{a},\vec{b},\vec{c},\vec{b}\ \ \vec{c},\vec{a}\ \ \vec{c},\vec{a}\ \ \vec{b}$ 为边的四面体（有向）体积，如图 3-41 所示，这就与二维情况下叉积的意义类似。其实，关键问题在于我们的视野。在二维空间中，只能认识到叉积是一个（有方向的）数，而在三维空间中则可以认识到它也是一个向量；在三维空间中只能认识到混合积是一个数，而在四维空间中则可以像叉积的三维形式一样写出一个四阶行列式，其中一行是四个（单位）方向向量 $\vec{i},\vec{j},\vec{k},\vec{l}$，就成了一个四维向量。更进一步地，发现叉积和混合积的向量形式都不是本质的，也不能推广到高维情况，而只有行列式是本质的，也能很方便地推广到高维情况。同样，前文用行列式求和的正负号定义多边形正方向的办法，也是本质而容易推广的。

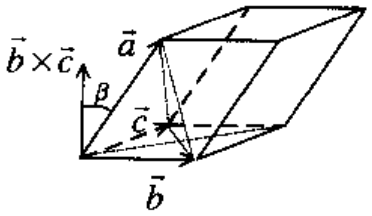


图 3-41　混合积的几何意义

8. “单纯形”的n维积和重心

单纯形和 n 维积

出于研究性和前瞻性的考虑，下面进一步深入高维情形的讨论。

首先把三角形和四面体这些最特殊、最简单的几何体推广到高维空间中去。

N 维空间中的 $n+1$ 个点，这些顶点的坐标为

$$(x_1,y_1,\cdots,t_1),\ (x_2,y_2,\cdots,t_2),\ \cdots,\ (x_{n+1},y_{n+1},\cdots,t_{n+1})^{①}$$

包含这些点的最小凸体（严格定义参见后文 3.3.1 小节）称为由这 $n+1$ 个点张成的“单纯形”。如 $n=1$ 时为线段，$n=2$ 时为三角形，$n=3$ 时为四面体。读者能否猜测一下单纯形的广义容积的形式？

① 严格地讲最好写成二维下标的形式，写成这种形式仅仅是为了书写和阅读上的方便

与三角形的面积公式和四面体的积公式（以及线段长度公式）类似，这个 n 维单纯形的 n 维积可以由下列公式求出：

$$\frac{1}{n!}\begin{Vmatrix} x_1 & \cdots & t_1 & 1 \\ x_2 & \cdots & t_2 & 1 \\ \vdots & \vdots & \vdots & \vdots \\ x_{n+1} & \cdots & t_{n+1} & 1 \end{Vmatrix}$$

当然，有向 n 维积（如有向长度、有向面积、有向体积）只要去掉绝对值即可。证明的关键在于同时进行行列式变换和几何变换，难度不大，留给读者。

这个行列式就把叉积和混合积统一起来，推广到了高维形式。而前面这个系数 $\frac{1}{n!}$ 是三角形面积等于 1/2 平行四边形的面积、四面体体积等于 1/6 平行六面体体积的推广，对于下文重心的讨论具有决定意义。

单纯形的重心

我们刚才从面积、体积拓展到了单纯形 n 维积，那么重心能不能同样拓展呢？

三角形非常特殊，其重心就是质点系重心公式——3 个顶点坐标的代数平均数，这种特殊的性质是否适用于所有单纯形呢？不妨探索一下。

首要的问题是，三角形重心为何在平均数处呢？重心（或质心）是使物体（几何图形）保持平衡的点，这样，显然与力矩有关，也就与容积有关了。其实，严格地说，面积、体积乃至高维的容积都是定义在积分意义下的，于是，重心也应由积分来计算。简单的二维情形，如图 3-42 所示。

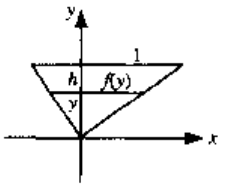


图 3-42　重心与积分

三角形面积 $S=\int_0^h f(y)dy$，

重心 y 坐标 $C_y=\dfrac{\int_0^h yf(y)dy}{S}=\dfrac{\int_0^h yf(y)dy}{\int_0^h f(y)dy}$，$y$ 是力臂，$f(y)dy$ 是重力。

于是，根据相似关系（比例关系）可得 $f(y)=\dfrac{y}{h}l$，我们不关心常数，因为上下可消去，于是变成

$$\frac{\int_0^h y^2dy}{\int_0^h ydy}=\frac{\left.\frac{y^3}{3}\right|_0^h}{\left.\frac{y^2}{2}\right|_0^h}=\frac{2}{3}y\Big|_0^h=\frac{2}{3}h$$

说明三角形重心恰在下 1/3 分点处，其实，对于四面体，甚至高维形，关键在于比例关系上 $f(y)\sim y^n$（n 维）。

于是，总变成 $\dfrac{\int y^{n+1}dy}{\int y^n dy}=\dfrac{n}{n+1}h$ 的形式，即重心总在（n+1）分点处。

但是这离“n 维单纯形重心在其顶点坐标的平均数处”尚有一步之遥，留给读者完成（提示：归纳法、相似、等价替换）。

至此，我们已经完全有能力解决三维及高维容积和重心问题了。

9. 多面体的四面体剖分和体积、重心；高维情形

有了上文提出的“单纯形”容积和重心公式，很容易求出 N 维空间中“n 维形”的容积和重心。为简单起见，先从三维空间中的多面体出发，进而推广到高维情形。

与平面多面体的三角形剖分类似，我们总是试图把一个多面体分割成许多四面体来解决。受平面任意点三角形剖分的启发，可以从空间任一点为扇心，即三棱锥的顶点，方便起见，当然也可选择原点，这样可以把原来一个四阶行列式变成三阶的，然后在每个面上，再进行三角形剖分，例如以第一个顶点 P_1 为扇心，像求多边形面积一样剖分成 N 个三角形 $\triangle P_1P_iP_{i+1}(i=1\sim N)$。关键是要保证每个三角形的方向是外法线方向，也就是说 $\overrightarrow{P_1P_i}\times\overrightarrow{P_1P_{i+1}}$ 一定要指向多面体外部。

当然，肯定有读者会问，既然可以用平面中任一点剖分多边形，那为什么不在剖分每个面多边形时加以利用，而采用以 P_1 为扇心的较差办法呢？原因很简单，我们所说的任一点是多边形所在平面的任一点，三维中多面体的每个面所在平面不是特殊位置，无法像二维一样选择一个特殊点。

于是，得到如下的体积和重心公式。

设多面体共 N 个面，每个面为一个多边形，顶点数 $N_i(i=1\sim N)$ 个，分别为 $P_{ij}(i=1\sim m,j=1\sim N_i)$，（有向）体积是 $V=\sum_{i=1}^{M}V_i$，

$$V_i=\frac{1}{6}\sum_{j=1}^{N_i}\begin{vmatrix}x_{i,1} & y_{i,1} & z_{i,1}\\ x_{i,j} & y_{i,j} & z_{i,j}\\ x_{i,j+1} & y_{i,j+1} & z_{i,j+1}\end{vmatrix}\quad(p_{i,N+1}=p_{i,1})$$

总体积为

$$V=\frac{1}{6}\sum_{i=1}^{m}\sum_{j=1}^{N_i}\begin{vmatrix}x_{i,1} & y_{i,1} & z_{i,1}\\ x_{i,j} & y_{i,j} & z_{i,j}\\ x_{i,j+1} & y_{i,j+1} & z_{i,j+1}\end{vmatrix}$$

重心仿照多边形的加权平均数，同样可得

$$\vec{C}=\frac{1}{V}\sum_{i=1}^{m}V_i\vec{C_i},\quad \vec{C_i}=\frac{1}{V_i}\sum_{j=1}^{N_i}V_{ij}\vec{C_{ij}},\quad \vec{C_{ij}}=\frac{\vec{O}+\vec{P_{i,1}}+\vec{P_{i,j}}+\vec{P_{i,j+1}}}{4}$$

总的，

$$\vec{C}=\frac{1}{24V}\sum_{i=1}^{m}\sum_{j=1}^{N_i}(\vec{P_{i,1}}+\vec{P_{i,j}}+\vec{P_{i,j+1}})\begin{vmatrix}x_{i,1} & y_{i,1} & z_{i,1}\\ x_{i,j} & y_{i,j} & z_{i,j}\\ x_{i,j+1} & y_{i,j+1} & z_{i,j+1}\end{vmatrix}$$

其中，$\vec{C_i}$ 是第个面与原点所成多棱锥的重心，再分解下去，$\vec{C_{ij}}$ 是把第 i 个面剖分成 N_i 个三角形后，第 j 个三角形与原点构成的四面体重心。

现在，推广到 n 维空间中去，我们的方法自然还是把“n 维形”剖分成许多 n 维空间“单纯形”。

首先看一下“n 维形”的结构。一个 n 维空间中的“n 维形”，由若干个“面”组成，当然，此地的“面”是相对概念，指（n-1）维空间；每个（n-1）维“面”又由若干（n-2）维“面”构成，直到两维面由若干点组成。所以，描述这个图形的数据结构最好是递归的树结构，以点为叶子结点，深度为 n-1。算法的实现就是递归往下分解，每得到一条树枝（到了递归边界），找到树枝上每个结点的最左儿子（叶子）（这些可在建树时完成），加上本叶子结点的右邻居，共 n 个顶点，构成一个 n 阶行列式，意义是这 n 个顶点和原点组成的 n 维单纯形的容积，然后累加起来就得总容积，求加权平均就得重心。请读者想象一下这样做的几何意义，对比二维和三维，就能理解了。需要指出的是，求一个行列式的值，如果按定义做，是 $O(n!)$的，当然，可以通过行列式的初等变换（高斯消元法）把行列式变为上三角形式，这样做的阶为 $O(n^3)$，总的复杂度是 $O(vn^3)$，v 是图形所含顶点数目，n 是维数。

3.2.3　判点在形内形外形上；多面体的情形

本小节我们转向另一个常见的主题：判断点与多边形（或多面体）的**位置关系**。设点为 P，几何体（多边形或多面体）为 D，则 P 与 D 的关系有三种情况：

（1）点在内部：$P \in D - \partial D$

（2）点在外部：$P \in \overline{D}$

（3）点在边界上：$P \in \partial D$

一般的著作不区分(1)(3)或(2)(3)，即一般不考虑(3)，但是我们始终认为，特殊情况乃是几何问题中始终存在不可忽略的一部分，如何简洁有效地处理特殊情况恰是一门很大的学问。

本小节中，主要介绍两种基本方法，即射线法与环顾法；主要讨论两个对象，即多边形和多面体。

1．射线法

对于判断平面上一点在多边形内外，**射线法**可能是最自然的想法。这种算法顾名思义就是从这个“点”出发，任意作一条射线，例如为了方便通常取 x 轴正方向，根据射线与多边形相交的次数的奇偶性来判断内外，奇数次在形内，偶数次则在形外，如图 3-43 所示。

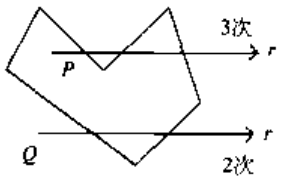


图 3-43　射线法的基本情况

特殊情况　这个算法虽说很直观，但读者很快就会发现，有一些特殊情况比较难处理，如图 3-44 所示。

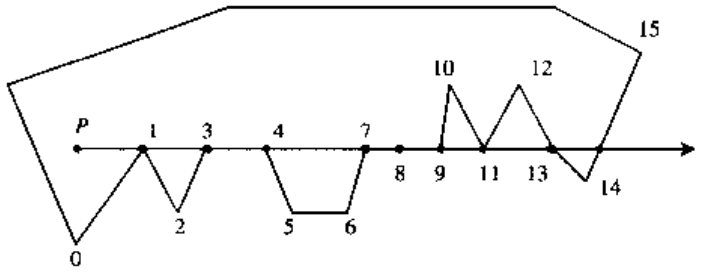


图 3-44　射线法的特殊情况

缩点法　点 P 在多边形内部，但是射线到底与之相交了几次呢？其实我们尚未严格定义何为“**射线与多边形相交**”。一种自然的想法是对于恰在射线上的点，与其相邻的端点（或线段）在射线异侧就视为相交，否则就不视为相交。例如这样可辨别出 0-1-2 与 10-11-12 并非真正相交，而 12-13-14 真正相交（从直观上看也应是这样）。但是对于整条边都在射线上的情况呢(3-4, 7-8-9)？我们可以这样做：遇到一个在射线上的点，就向后连续跳过所有也在射线上的点，直到第一个不在射线上的点为止，然后再用上述判断条件。这相当于把射线上的线段都“缩”掉了，如 3-4,7-8-9 就会被缩掉，结果如图 3-45 所示。

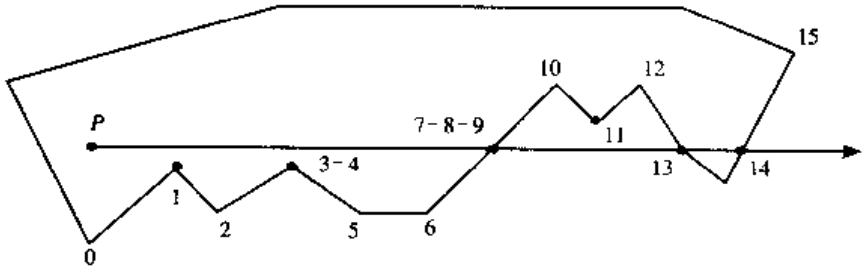


图 3-45　射线法特殊情况的第一种解决方案——缩点法

平移法　这样的定义或解决方案似乎不够优美和简洁，于是我们换个角度想，比如对点 1，它略微上升一些或下降一些都不影响结果（2 个交点和 0 个交点），例如可以把射线稍微上移一个很小的量 ε，这时原来所有的恰好在线上的点（即 0 中 1，3，4，7，8，9，11，13）都在射线下方了，如图 3-46 所示，这样做虽说改变了相交的情况，某些不严格相交的点（如 11）被视作相交两次（10-11，11-12），但确实不可能影响判断点在内外的结果。

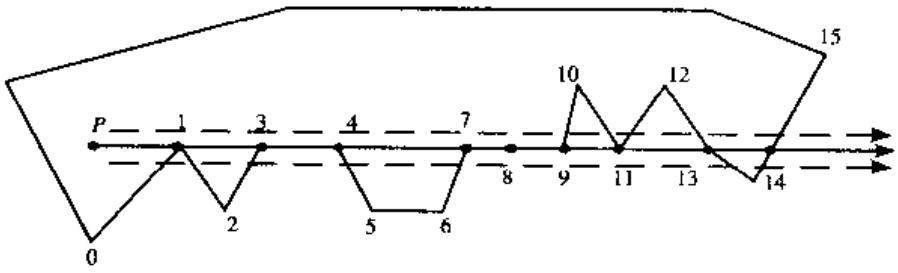


图 3-46　射线法特殊情况的第二种解决方案——平移法

其实在实现时，甚至不必真的把这条射线平移——从另外一个角度想，只要修改一下判断条件即可：每条线段，只要一个端点（高点）高于射线，另一端点（低点）恰在射线上或低于射线，且交点在 P 点右侧（x 轴正方向）即可。

对于判断交点在 P 的右侧，若求交点则要涉及浮点除法，速度慢，误差大，这里还是建议采用叉积法：对于一个相交线段，区别高点 A 和低点 B，$\overrightarrow{PB}$ 到 $\overrightarrow{PA}$ 为右手螺旋，则交点在 P 的右侧，否则在左侧，如图 3-47 所示。

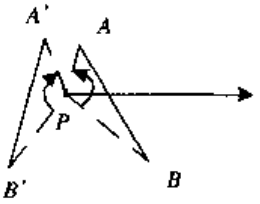


图 3-47　叉积用于判断（射线法）交点左右

至此，这个算法已经完整了，但是否一定正确呢？

边界情况还是考虑边界情况，很快发现当 P 恰在多边形边上（包括端点上）时会出问题。这是因为这时叉积恰为 0。这时上述对叉积结果的判断可能有两种写法：>0 或者≥0。我们发现，如果 P 在 1，3，4，7，8，无论哪种情况都会被认为是内部；而在“>0”的情况下 9，11，14 处被认为是“外部”，13 被认为“内部”，而“≥0”时这四个点恰相反。这些都是不能接受的。

而且从根本上说，希望能辨出恰在边界上的情形，甚至要求辨别出恰在端点的情形，即要求输出为：

0----------P 在多边形外部：$P\in \overline{D}$

1----------P 在多边形（严格）内部：$P\in \partial D$

2----------P 在多边形一边上，但不在其端点：上 $p\in \partial D-\partial^2 D$

3----------P 在多边形一端点上：$P\in \partial^2 D$

对于这个要求，简单的做法只要附加单独判断即可（因为不影响时间复杂度）。可以在循环中对每条边先做叉积，如果=0 就进入单独判断（用点积或坐标比较），>0 则判断上下顶点和射线的位置关系。

2．环顾法

射线法虽然简单，但特殊情形的处理倒也是颇费了一些周折的。虽然平移一个小位移，成功地解决了问题，但可能有读者就认为它不那么“优美”了，所以这里给出一个更加优美的算法，但是速度却远慢于射线法。

环顾法也称**绕圈法**或**转角法**，是通过沿多边形走一圈，累计绕了点 P 多少角度来判断 P 的内外，其思想类似于前文习题 3.1.7 的篱笆张角问题，只不过前文的张角变成了这里的（有向）转角，如图 3-48 所示。如果考虑转角的正负性（即右手方向为正，左手为负），则转角的代数和对于内部总是 2π，对于外部总是 0，这个思想在前文计算多边形面积的地方（3.2.2 小节）已经提过。

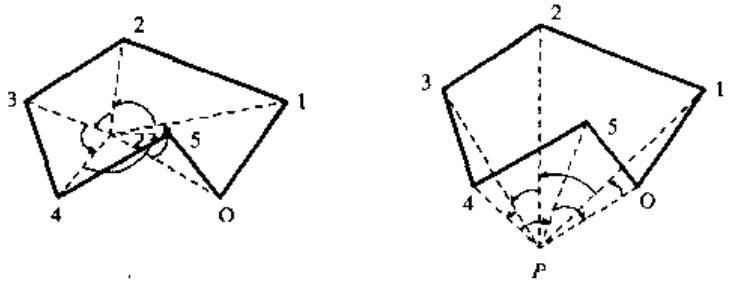


图 3-48　环顾法（转角法）的基本原理

具体实现时可以对于每一条边，先计算对 P 的叉积方向，再通过点积和反三角函数 arccos 来计算角度（此边对点 P 的张角）的绝对值——此时角度必在[0, π]。叉积大于 0 则加上，小于 0 则减去，最后看累计角度是多少，由于 0 和 2π 是离散的，因此这个算法误差很小。

特殊情况

但是读者一定发现，上述实现中忽略了一个重要细节，叉积=0 怎么办？

分析一下，发现产生叉积=0 的无非以下三种情况，如图 3-49 所示。

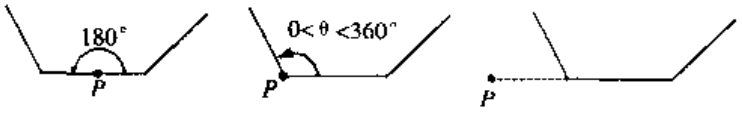


图 3-49　环顾法的三种特殊情况

（1）P 位于一条边上（但不在顶点上），$P \subseteq \partial D - \partial^2 D$，点积= -1（180°）；

（2）P 位于一顶点上，点积=0（90°）；

（3）P 位于一边的延长线上（可能在形内，也可能在形外）点积=1（0°）。

一种比较巧妙的处理方法是：如果遇到叉积=0 就忽略。首先（3）不会产生问题（如图 3-50 所示）；（1）会导致最后转角累计为 180°；（2）会导致转角累计为一个 0°~360° 的角度。而这恰恰可以作为一种判定标准，刚好把三种情况分开，而且也与直观认识吻合，可以作为转角法定义的一部分。（不过，三线公点的中点是个例外——无法区分(1)和(2)）。

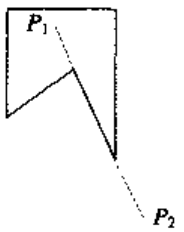


图 3-50　环顾法：在一边的延长线上不会出错

另一种方法是简单地采取射线法中的特别判定策略。可以在发现叉积=0 时再做范围判断。若是（3）则继续，（1）（2）则退出。

优美的代价　最后需要指出的是，虽然环顾法看似比射线法优美一些，且复杂度相同（都是线性），但常数上，由于反三角函数、开根运算、浮点除法的存在，前者比后者慢

得多，有实验（参见文献[86]）指出，前者比后者慢 20 倍以上。

定义内部和方向　前文讲到，缺少一种适合计算机的内部定义方法。而这里讲的射线法和转角法都是充要的，因此可以作为内部的定义：

- 对于射线法，根据射线和多边形交点数目奇偶性来定义内外（一般情况下）。
- 对于转角法，特别注意其实与输入的顺序无关：不论输入是什么绕向，最后得转角累计值为±360° 者为内部；0° 者为外部；其他情况为边界（其中±180° 者为边上非端点）。

这样定义以后可以通过“内部总在左手”来定义正方向。

3. 多面体的射线法

在二维情形的基础上，考察三维空间中点在多面体内的判断。很自然地，会想到把上述两种算法推广到三维中去，事实上它们也确实可以拓广，但是随着维数的上升，难度和繁度（特殊情况）也大大增加，所以两种算法在延拓时，要做很多改进。

首先还是考虑射线法，同样地构造一条三维射线，看它与几个面相交，根据奇偶性来判断内外。但是当时就比较困难的特殊情况（射线恰好经过一个顶点或重合一条边），现在将会扩展为更加复杂的情形，如图 3-51 所示。

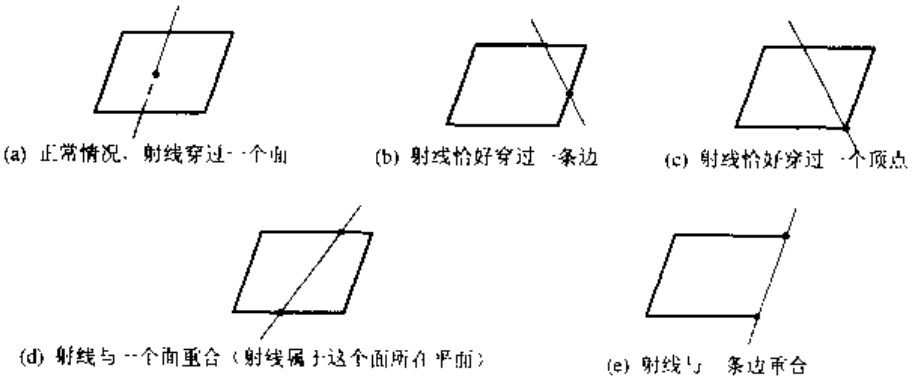


图 3-51　三维射线法的特殊情况

如此多的特殊情况可能不是一个简单的算法所能解决的，所以我们换一条思路：虽然特殊情况很复杂，但是发生的概率却很小，所以可以随机地产生一条射线，若没有发生特殊情况则成功，否则再随机产生一条，直至不发生特殊情况为止。这个算法是可行的，而且是高效的，限于篇幅，就不再深入讨论下去，当然，读者还可以考虑一下如何简洁地处理所有这些特殊情况。

4. 立体角和多面体的环顾法

前文已经指出，环顾法没有那么多特殊情况，因而更优美一些。这里，当维数增加，情况大大复杂以后，其优势则进一步凸现出来。当然，为了适应三维，必须把“平面张角”的概念拓广为“**立体角**”（Solid Angle），同样也是“**有向**”立体角，这个“方向”（正负）取决于一个面与点 P 组成的棱锥的“有向体积”的正负（见前文 3.2.2 小节的“从三角形到正面体”），很明显这和二维转角的“有向面积”概念一脉相承，那么与二维中张角大小对应的立体角怎么定义呢？

问题的转化　对于一个 n 棱锥的顶角，可以通过三角形剖分化为 n 个三棱锥（有向）

顶角的的代数和。所以，问题就转化为求三棱锥的顶角（立体角）。

这个三棱锥的顶角定义为：以顶点为球心，作单位球，则单位球所截到的“球面三角形”面积就是三棱锥的顶角大小。

设球面三角形为$\triangle ABC$，则$S_{\triangle ABC}=R^2\ \delta=\delta$，而$\delta=A+B+C-\pi$称为“球面角超”，即立体角。于是问题就转求球面夹角A，B，C了。

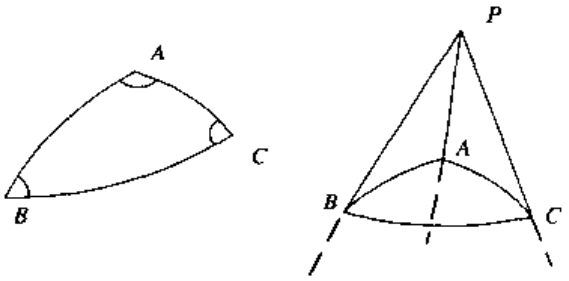
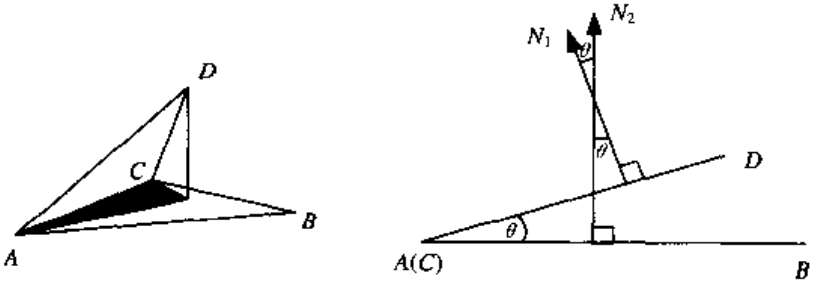


图 3-52　球面三角

$\because\triangle ABC$在球面上，$\therefore PA\perp AC, PA\perp AB$（径向⊥切向）。$\therefore$角$A$即为$PAB$与$PAC$的两面角。于是问题又转化为求两面角。

求两面角的方法很多，如图形 3-53 所示，立体几何中常用求点D在面上的投影来做，但显然在计算几何中不是最简单的办法。



(a) 投影，化为面积之比　　(b) 叉积，化为法向量夹角

图 3-53　两面角的两种求法

叉积求两面角　如果读者想到前文强调的叉积的几何意义，就会想到一种非常简单的办法：叉积$\vec{a}\times\vec{b}$代表$\vec{a}$、$\vec{b}$决定平面的（有向）法矢量，那么两面角其实只要比较两个面的法矢量夹角即可。图 3-53(b)中为求ADC面与ABC面的夹角，可先求$\overrightarrow{N_1}=\overrightarrow{AD}\times\overrightarrow{AC}$，$\overrightarrow{N_2}=\overrightarrow{AB}\times\overrightarrow{AC}$，再算$\overrightarrow{N_1}\cdot\overrightarrow{N_2}$，即可求得两面角。

以上，我们通过几步变换，把一个立体角的问题最终转化为两面角的计算，并用叉积很简单地求解。

更深入的解释　需要指出的是，转角法（包括平面与立体角）是可以推广到高维的，但是首先要对平面转角法与立体角的共性深入探讨一下。

平面转角法的另一种解释：以Q点为圆心，作单位圆，累加每条边所映射到的小扇形有向面积，其和$=\pi R^2=\pi$为内部，$=0$为外部，如图 3-54 所示。

立体角的另一种解释：以Q点为圆心作单位球，累加每个面所映射到的小球面棱锥的

有向体积，其和=$\frac{3}{4}\pi R^3=\frac{3}{4}\pi$ 为内部，=0 为外部。

甚至求两面角的办法在平面中也可用于求两个向量夹角，相当于求与它们垂直的向量的夹角。所以整个过程极其相似，只不过立体角要多一步三角形剖分，而这在低维→高维拓广过程中也是必不可少的。

于是，如果要拓广更高维，也是以 n 维空间中的点 Q 的为中心，作 n 维球，再累加每个“面”（n-1 维空间）对应的有向 n 维棱锥的 n 维积，若等于整个 n 维球的有向 n 维积则在形内，=0 在形外。

5．凸多边形与星形多边形

上文提出的对于平面多边形的判断形内形外算法，都是线性的，如果把条件加强一些，例如考虑凸多边形，能否得到更好的时间复杂度呢？

当然，读者很容易想到，对于凸多边形，若用转角法，则不必累计转角值，只要依次对每条边判断方向即可，有一条边反向则立即退出，但是这并不改变最坏情况的时间复杂度。

特别性质　如果先不考虑特殊情况，用射线法的思路，则若点在形外，有 0 或 2 个交点，若在形内只可能有 1 个交点。但是如果任选形内一点 x，此点出发作经过 Q（欲判点）的射线 xQ，则不论内外都有且仅有一个交点。这个性质极好，只要知道了是哪一条边与这根新射线相交，然后再判断 x 和 Q 在该边异侧还是同侧即可。

那么，现在的问题就是：如何更快（快于线性）地找到这根与射线相交的边？

二分法　我们很容易想到，由于凸多边形其极角是连续的（严格递增），所以只要用二分法查找即可定位该射线的极角究竟在哪一条边的范围内，这样的算法是 $O(\log n)$的。

具体实现时应该定义 xP_0 线段为极角参考位置，即令 P_0 处极角为 0，再增加 $P_n=P_0$，其极角为 2π，这样可以保证 xQ 的极角必落在此范围内。然后用二分法查找，如图 3-55 所示，可能有两种情况：

（1）xQ 极角恰与某 xP_i 极角相同；

（2）xQ 极角处于某 xP_i 与 xP_{i+1} 之间。

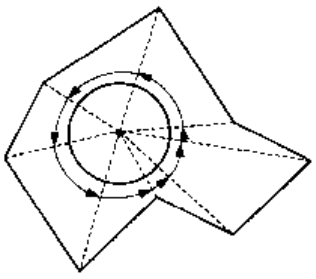


图 3-54　平面转角法的深入解释

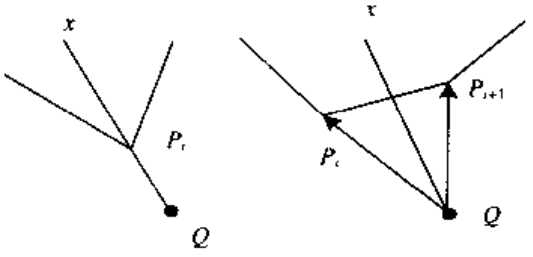


图 3-55　二分法实现时的两种情况

对于（1），说明 xP_iQ 三点共线，调用前文坐标比较或点积的方法（参见 3.2.1 小节的“特殊情况与点积”），判断是形外、点上还是形内；对于（2）判断叉积 $\overrightarrow{QP_i}\times\overrightarrow{QP_{i+1}}$，>0 形内，<0 形外，=0 边上。

需要指出的是，有些读者认为，虽然这样判形内、形外是 $O(\log n)$的，但前期读入这 n

个点已经是线性的了，所以总时间复杂度不可能小于线性，这种讲法不错，在实际中（尤其竞赛中）整个从初始化到判断的时间复杂度确实线性的，但从理论上讲，判形内、形外这个过程确实是 $O(\log n)$，甚至读者可以想象一种“在线询问”（Online Inquiry）机制，即算法要求哪个点的信息时才输入该点坐标，这样就绝对 $O(\log n)$。

刚才我们的凸多边形算法基于其内部一点，这个 x 的求取很简单，任取三点求其重心即可，但若遇到三点共线，则删去其中一点，再取三点。（当然，我们承认在有三点共线时，这一步本身最坏就是线性的了）

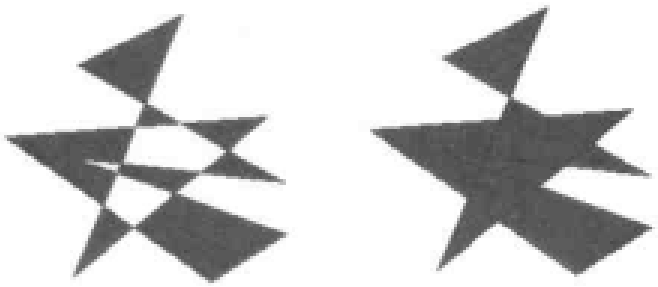
其实还可以再拓广一下，考虑星形多边形，如果把凸多边形内部任一点换成星形多边形核内一点，也可以用二分法，其实，凸多边形只是星形多边形的一种特例——其内部就是核。所以，我们总结成以下定理：**星形多边形若已知其核内一点，就可在 $O(\log n)$时间内判断点在形内形外问题。**

当然，前文讲“卫兵问题”（3.2.1 小节）时已经提到，确定星形多边形的核相对困难一些，将留待本章最后的专题（3.4.5 小节）中详细讨论。我们将看到，求星形多边形的核是 $O(n \log n)$的。

练　习　题

思考题：

3.2.1　复杂多边形的内部定义方案十分复杂，可以有好多种，例如奇偶法和累加法，如图 3-56 所示。



(a) 奇偶法　　(b) 累加法

图 3-56　复杂多边形内部定义的两种方法

分别对这两种定义，设计相应的面积、重心求法和内外判别法。你还能提出其他定义方法吗？

3.2.2　卫兵问题的另外一些变种：

（1）给定一个多边形，至少放 G(边)个卫兵就能看到所有边上的点，至少放 G(内)个卫兵就能看到所有内部点。显然 G(边)≤G(内)，我们要问是否有 G(边)=G(内)？

（2）对于 n 个顶点的多边形，最坏情况下至少放几个卫兵才能看到所有内部点？

3.2.3　考虑高维转角法的具体实现方案。

3.2.4　设计一种算法，简洁地处理三维射线法遇到的所有特殊情况。

3.2.5　从梯形剖分出发推导重心公式。

3.2.6　证明 n 维单纯形的 n 维积公式。（提示：同时进行行列式变换和几何变换）

编程题：

3.2.7　（参见前文习题 3.1.6）自行车问题的变种。现在我们把题目的要求变为：

（1）允许建筑物相交，但建筑物都是凸多边形；

（2）允许建筑物相交，建筑物可以是凹多边形。

注意（2）有一定难度，但是用本节的知识完全可以解决。

3.2.8　凸多边形的本影和半影[①]

平面上有两个凸多边形 A 和 B，相互分离且都不退化，如图 3-57 所示。假设 B 是光源，求 B 被 A 遮挡住的视野，也可以认为是 A 形成的影子范围。

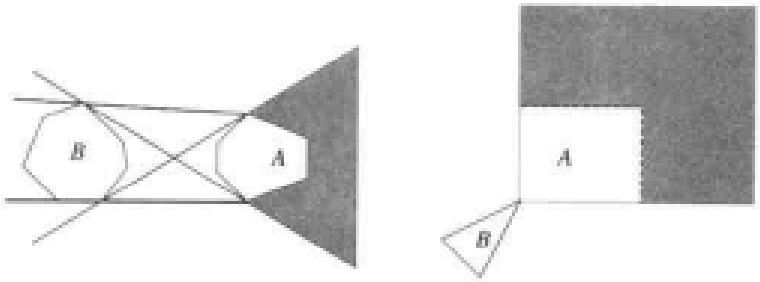


图 3-57　影子的范围

（提示：本题及其对偶问题（本影范围）本质上就是求 4 条公切线。$O(n^2)$的算法是很显然的，但你能否有线性的做法呢？进一步地，联想本节讲过的凸多边形判断形内形外的特殊性，能否尝试更快的算法？）

3.2.9　极品牛奶[②]

汤米最近买到了一种极品牛奶，据说把它涂到面包上，吃掉以后人会变得更聪明。汤米把牛奶倒在一个大杯子里面，把面包放在里面蘸牛奶（面包的一边接触到杯子的底部），如图 3-58 所示。

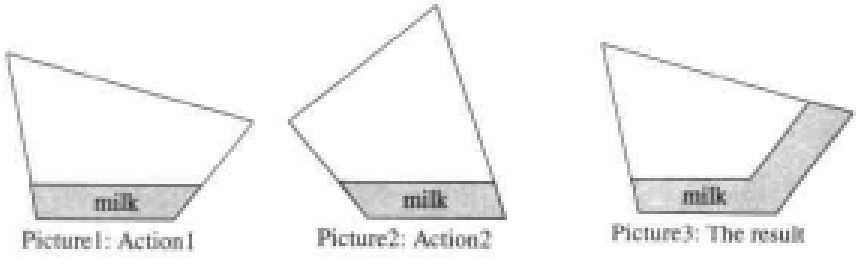


图 3-58　面包蘸牛奶的过程

由于这种牛奶比较贵，杯子里就只有深度为 h 的那么一点，因此每次只有牛奶水面和

① 题目来源：UVA Problem Archive 746

② 题目来源：OIBH Online Programming Contest #1，命题人：刘汝佳

面包底部之间的区域覆盖着牛奶。汤米希望覆盖着牛奶的面包区域尽量大，但是他最多只愿意为此反复把面包放在牛奶里蘸 k 次，那么究竟如何做才是最好的呢？（注意：杯子的底部比面包的任何一条边都宽，因此可以任意的选一条边蘸）

3.2.10　载油问题[①]

一个简单多边形的油桶，竖直放置，如图 3-59 所示。但是年久失修，油桶的边上出现了若干漏油点，现问实际能装多少油？

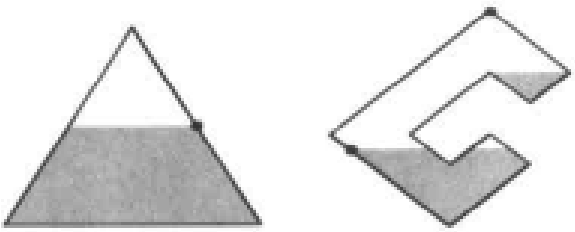


图 3-59　油桶（黑点表示这一点会漏油）

3.3　打包裹与制造合金——凸包及其应用

凸包是计算几何发展史上第一个被深入研究的几何模型。其特殊的优美性质为其带来了深刻而广泛的应用。这里我们首先从凸性入手刻画凸包；然后讨论如何求凸包，其中特别仔细分析了特殊情况；再对凸包算法进行理论上的正确性和效率分析，挖掘了它和排序算法的本质联系；最后根据凸包和凸多边形的优美性质讨论其应用。

3.3.1　凸包的普遍性和广泛应用性：凸的定义与优美性质

在本小节中，我们将展示凸性和凸包在实际问题中的广泛性及其抽象定义的优美性。首先将通过两个例子介绍凸包。

1．凸包的引入

打包裹的例子　第一个例子是很简单也很常见的——打包裹。

【例题 1】篱笆问题[②]

假设你种了很多树，想用一个篱笆把所有的树都包在里面。出于经济考虑，显然，这个篱笆应该是越小越好。

于是你可以想象用一根很长的绳子，用打包裹的办法绕着这些树走一圈，扎紧绳子，就得到了所要求的最小篱笆，这就是一个凸包。这个例子同时提醒我们，打包裹的方法不失为求凸包的一种可行的办法，另一种直观想法是一个橡皮圈的收缩，恰好套住时形成凸包，如图 3-60 所示。

[①] 题目来源：ACM/ICPC World Finals 2002, Toil for Oil

[②] 题目来源：经典问题

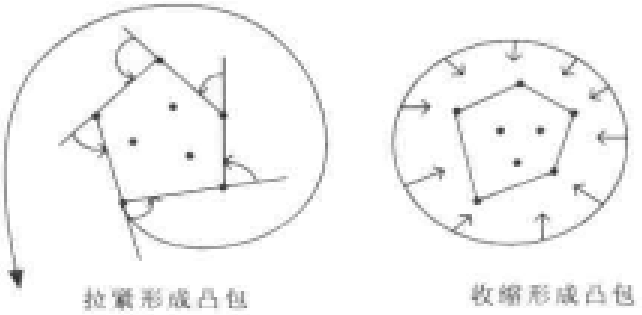


图 3-60　凸包的两种直观解释：打包裹和收紧橡皮圈

制造合金的例子　第二个例子可能初看与凸包毫无关系——制造合金[12]。

【例题 2】合金制造问题[1]

假设你有一些金-银合金，它们的含金量和含银量各不相同，现在要求你通过按某种比例混合，制造出新的合金，那么哪些含金是可能被合成的？更精确地，通过这些现有的合金制造出的新合金，它们的含金量、含银量在什么范围内？

这个问题初看有些难，那么就先看其简化版：

① 单纯考虑含金量。显然，新造出的合金的含金量只能在原来的合金含金量范围之内，即在原来的最大和最小之间（如图 3-61（a）所示）。

② 进一步，如果加上含银量，但假设原来只有 2 种合金，也很简单，新合金的金/银比例必在原来的合金之间的范围内，精确地讲，原来合金为 A、B，其含金/银比例分别为 (x_A,y_A)，(x_B,y_B)，则新合金 C：$C=\lambda A+(1-\lambda)B$，即

$$\begin{cases} x_c=\lambda x_A+(1-\lambda)x_B \\ y_c=\lambda y_A+(1-\lambda)y_B \end{cases} \qquad \lambda\in[0,1]$$

从几何上讲，C 必然在线段内（是 AB 的内分点，含端点）（如图 3-61（b）所示）。

③ 那么如果原来 3 个合金 A，B，C，则直观得新合金必在△ABC 内（如图 3-61（c）所示）。

④ 推广到一般情况，n 个原金 $A_1A_2\cdots A_n$，则新合金必在这些点的凸包内（如图 3-31（d）所示）。

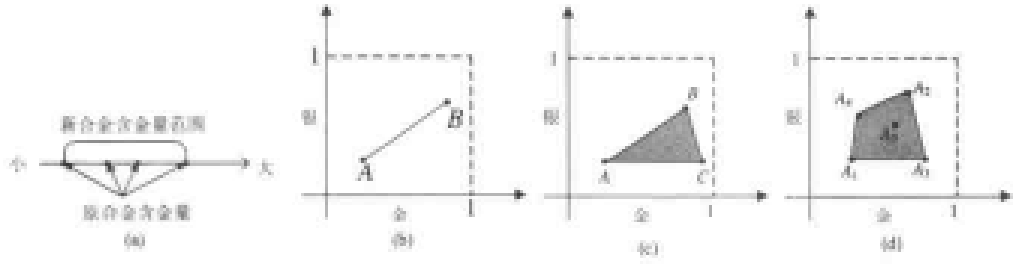


图 3-61　凸包与合金问题

到这一步，我们已经把一个原本不相关的问题成功地转换为凸包问题。

⑤ 进一步推广，我们不仅局限于两种金属的纯度，而是考虑 m 种金属的纯度，即：

[1] 题目来源：经典问题

原有 n 种合金，每种合金 i 中金属 $j(j=1\sim m)$的含量百分比为 x_{ij}，则相当于把二维平面上的问题推广到 m 维空间中的凸包问题。

上述两个例子不仅很简单，而且还很深刻，它提示了凸包和凸性的两种定义方法：直观定义和抽象定义。

2．凸性和凸包的定义

例 1 的打包裹提醒我们平面几何中关于凸多边形的定义。

定义 1：凸多边形 $\Leftrightarrow$整个图形在任一条边的一侧

这不仅是一个很优美的性质，同时还提供了一种求凸包的最直观的办法——沿边界打包裹——将在 3.3.2 小节详细讨论。

例 2 则展示了凸性更深刻的一面。

定义 2：D 是凸图形 $\Leftrightarrow \forall x_1, x_2 \in D, \dfrac{x_1 + x_2}{2} \in D$

即对于一个凸图形，任意两个内点的中点，也在此图形内。更一般化地，我们不仅考虑中点，还考虑所有内分点，于是有如下意义。

定义 3：D 是凸图形 $\Leftrightarrow \forall x_1, x_2 \in D, \forall \lambda \in [0,1],\ \lambda x_1 + (1-\lambda)x_2 \in D$。即任意两点的内分点都在此图形内。

显然，因为 x_1x_2 的任意性，定义 2 等价于定义 3。注意到 $\lambda x_1 + (1-\lambda)x_2 \in D, \lambda \in [0,1]$ 的形式很优美，这是线性组合 ax_1+bx_2 的一种特殊形式，加上条件 $a+b=1$ 则是定比分点，几何意义是 x_1x_2 直线本身；再加上条件 $a\geqslant 0$ 且 $b\geqslant 0$，则是内分点，即线段 x_1x_2。由于其在凸性中的特殊地位，又名凸组合，如图 3-62 所示。

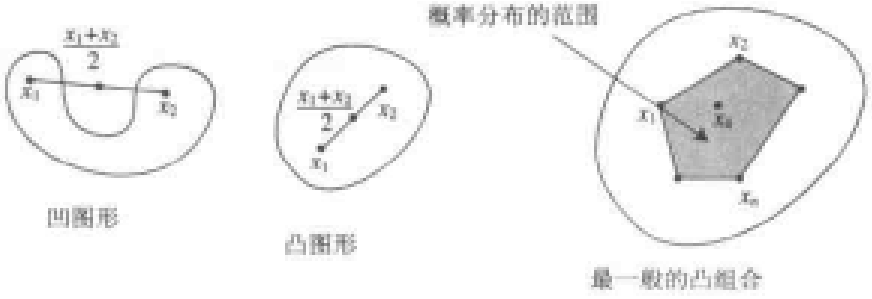


图 3-62　凸性的几种定义：凸组合的代数定义，概率分布

现在把凸组合再推广一下，给出凸性的又一种定义。

定义 4：D 是凸图形 $\Leftrightarrow \forall_i \in D, (i=1,2,\cdots,n)$，$\forall \lambda_i \geqslant 0$ 且 $\sum_i \lambda_i = 1$，有 $\sum_i \lambda_i x_i \in D$

即 n 个内点的加权平均（凸组合）也是内点。这里把 λ 和(1-λ)推广到 $\lambda_i \geqslant 0$，$\sum_i \lambda_i = 1$，这称为一种概率分布，和概率论中的定义吻合。

同样，我们看到，由于 x_1x_2 的任取性，定义 3 与定义 4 等价。这 3 个等价定义中，定义 2 最为简洁，定义 4 最一般化。这些定义适合 n 维空间。

凸包、极点、凸表出　直观地讲，对于一个平面点集或者一个多边形，它的**凸包**指的是包含它的最小凸图形或最小凸区域（例如定义 4 的概率分布本身就定义了一个凸区域）。

代数的语言是：凸包是点集中所有点的凸组合。凸包上的顶点（共线中点除外）称为**极点**（extremal points）。

那么，极点有什么特殊性呢？它们不能被凸包内部的点“**凸表出**”。“凸表出”即线性表出的一种特殊形式，即定义 3 中的内分点或定义 4 中的概率分布点，其实，用例题 2 中的话来说，就是通俗的“按比例混合”。即：凸包的极点，是不能用凸包内部的点按比例混合而成的，而对于原有合金，它们的混合而成的新合金，只能在凸包内部（或凸包边上的内部）。这就是定义 3 和定义 4 对例题 2 的最好解释。对于极点，它们既然不能被凸包内部的点混合而成，则显然只能是原有合金，于是得一基本定理：（定理 1）**凸包上的顶点是原点集中的点**。

以上从两个风格迥异的实例出发，经过多次的推广，讨论了凸包的凸性的各种定义，尤其是其中几个概念，如“凸组合”、“概率分布”、“极点”的定义，还有打包裹的思想，都是深刻而优美的，在以后的讨论中有着广泛的应用。

3．凸包应用初探

我们知道，凸包是基于原有点集或原多边形的。凸包相对于它们的优势主要有以下两点，这也决定了凸包的两个主要应用：

① 点的数量大大减少。定量研究表明（参考文献[93]）：k 维球体中均匀独立分布 n 个点，其凸包顶点数 $m(k)=O(n^{\frac{k-1}{k+1}})$。$k$=2（平面）时，$m(2)=O(n^{\frac{1}{3}})$；$k$=3（空间）时，$m(3)=O(n^{\frac{1}{2}})$。可见凸包可以大大降低**平均情况时空复杂度**。当然，简单的替换无法降低最坏情况下的时空复杂度，即若所有原点集中的点都恰好位于凸包上时。那么有没有用凸包降低最坏情况时空复杂度的呢？

② 凸包相对于原点集，增加了一个“序”（多边形）；而对于原多边形，凸多边形则拥有许多特殊的优美性质。这些序和性质有时候可以从本质上带来新的算法，降低**最坏情况时空复杂度**。

凸包的应用初探如图 3-63 所示。

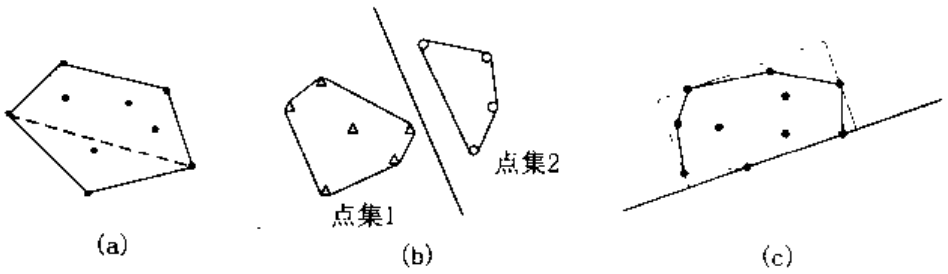


图 3-63　凸包应用初探三例：点集直径、点集剖分和最小外接矩形

【例题 3】点集的直径：任意两点间距离的最大值。

能否化为凸包上的点对之间的距离最大值？这样就忽略了凸包内部所有点——降低平均情况时空复杂度。

【例题 4】判断平面上的红、蓝两个点集，是否可以被某条直线分割开来。

最原始的做法也是有的：任意两个红点连线，称为红线集，对任意两个蓝线也连线，

称为蓝线集，只要红线、蓝线不存在相交，原点集就是可分离的，但这个算法时间复杂度极高，为 $O(n^4)$。然而，能否把它转化为判断两个点集凸包的位置关系呢？而判断两个凸包相交与否，最傻的做法就是对两个凸包，任取一对边，判相交——这就降低了最坏情况时空复杂度。

【例题 5】一个稍微复杂的问题是最小矩形（smallest box）：求能够包住一个点集的最小（面积）矩形。可不可以先求凸包，再尝试凸包每一条边作为矩形的一边（即把原问题转化为包住凸包的最小矩形）？

以上仅仅是初步讨论了凸包的应用，只是为了让读者在进一步学习凸包实现的算法以前，首先意识到凸包是非常有用的。这三个例题，虽然这里都给出了大致思路，但是还有很多细节，例如证明使用凸包对原题是等价的，以及凸包以外的做法和使用凸包的相对优势，这些都会在下文 0 小节详细展开。

3.3.2　凸包的实现

本节主要讲凸包实现的两种常用也是最简单的算法——卷包裹法（Gift-Wrapping）和 Graham-Scan 法。特别详细讨论了它们在特殊情况下的处理。

1．卷包裹法（Gift-Wrapping）

凸包问题的输入，一般为平面点集或平面多边形，我们暂时不把这两类问题区别对待。凸包问题的输出一般是一个凸多边形(有序)，然而，也有一些应用只要求凸包上的点（而不要求顺序）。对于这两种情况的区别，我们也留待 3.3.3 小节讨论，而此处只考虑前一种情况。

我们回忆一下 3.3.1 小节例 1 中的思想——卷包裹法，就是凸包问题的最直观的一种解法.

形象思维假想一根无限长的绳子，一开始，左端粘在点集最低点上。若有多个最低点，则取最左的最低点。然后拉动绳子的另一端逆时针绕行，每次贴紧一条边（这就是凸包上的一条边），这样绕行一周，绳子的形状就是凸包，这就是所谓卷包裹（Gift-wrapping）法，如图 3-64 所示。

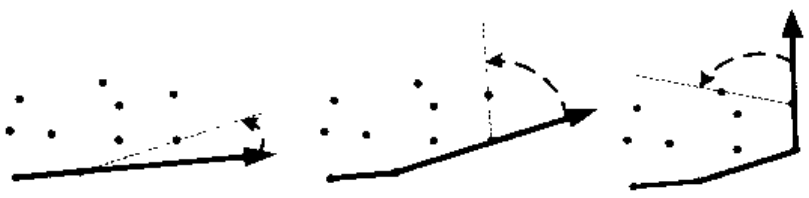


图 3-64　卷包裹法的基本思想

刚才是人的形象思维，那么计算机的逻辑思维如果实现呢？关键是上段中“拉动……绕行，每次贴紧一条边”的描述过于模糊。对于计算机，需要解决的是怎样寻找下一条“贴紧的边”。

为简单起见，先假设没有三点共线的情况。

假设上一条“贴紧的边”是 AB，只要考虑 AB 的延长线 BD 以 B 为轴逆时针旋转时，先碰到 BC_1，BC_2，…，BC_5 中哪条边。从计算几何的角度讲，就是判断哪个在最“右手”方向。最原始的办法就是寻找一条边，其他所有边皆在前左手方向，如图 3-65 中 BC_2，…，BC_5 都在 BC_1 左手方向。当然这样做每一步都要 $O(n^2)$，效率太低。

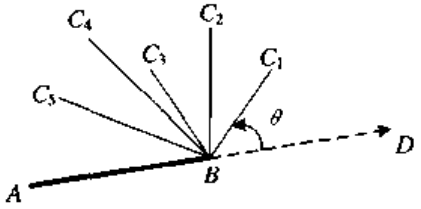


图 3-65　卷包裹法中寻找下一条边的方法：极角与叉积

极角与叉积　一个很显然的改进是：比较极角大小，即 BC_i 与 BD 所成角大小。这样只要简单地“擂台赛”一遍，就可以找到极角最小的边。这样的算法每步 $O(n)$，总时间 $O(n^2)$。然而，直接比较需要计算极角的值，时间效率与精度都很低。联想前文 3.2.2 节所提到的叉积与极角的关系，很容易发现，可以把“极角大小”比较用“左右手”关系比较（叉积）来实现。即：BC_i 极角比 BC_j 极角小⇔BC_i 在 BC_j 的右手方向。

序　我们进一步推广，“左右手”关系比极角更本质，其实比较的只是空间**相对位置关系**，而不像**角**度比较、长度比较那样基于一个绝对的量。这就好像只知道一堆数字之间的**相**对大小关系，而不知道它们具体大小，还是能够进行排序。抽象地说，“右手”关系是一种“偏序（partial ordering）”（非自反，反对称[1]，传递）。只要有“偏序”（推出律），就仍可以用一般的比较办法来求**极值**或者（拓扑）排序。进一步，“右手”关系是一种“全**序**（total ordering）”[2]，因此可以求最值，并且有惟一排序。

值得注意的是这里“偏序”、“全序”的思想，即这种思想不仅在计算几何中常用，而且适合于所有的算法设计。

以上还可以看出“序”和“排序”的概念对于凸包的重要性，这一点将是下节深入分析的主题。

特殊情况　下面讨论一下特殊情况——三点共线情况。

这取决于问题的要求，在多点共线时，输出的凸包是包括所有可能的共线点还是不包括任何共线点（即只有极点），如图 3-66 所示。

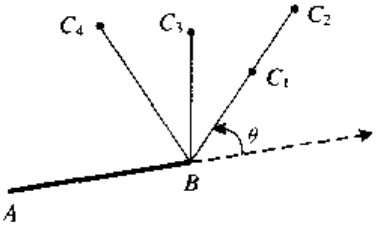


图 3-66　卷包裹法的共线情况

[1] 其实反对称可以从非自反和传递两者推出来
[2] 全序是指任何两个元素之间都有关系的偏序

对于这个“卷包裹法”，这两种要求都是容易实现的，只要把上面的“比较”即“序”的定义改为：先判叉积，若不共线则“序”为左右手方向，否则用判断其前后（点积或坐标比较），如果是第一种要求，则取前点，否则取后点。

卷包裹法非常简单而易实现，时间效率为 $O(n^2)$。

2. Graham-Scan算法

刚才讲的卷包裹法固然直观易懂，但时间效率较低，那么有没有更高效的算法呢？

卷包裹法的每一步都确定性地得到一条最终凸包上的边，能否换个思路，不追求每步必朝最终凸包前进一步，而是考虑“临时凸包”或“局部凸包”。这种凸包是试探性增长的，目前凸包上的边并不一定都是将来最终凸包上的边，而是通过逐步纠正错误来逼近“最终凸包”。

试探性凸包

例如一个简单的凹多边形，我们尝试从 p_1 开始，沿着多边形的顺序，试探性地增长凸包，如图 3-67 所示。

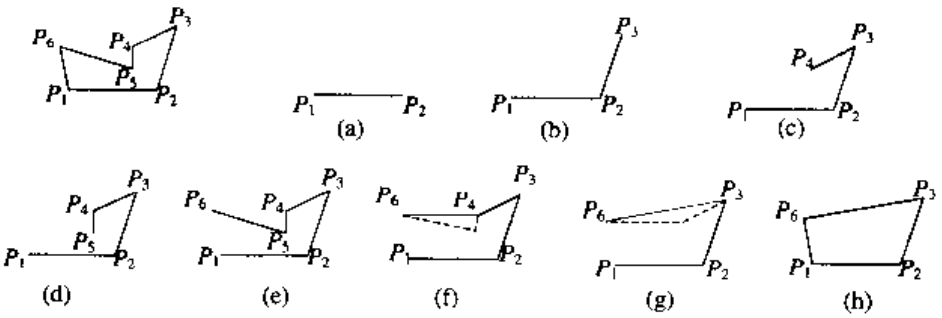


图 3-67　Graham scan:试探性增长凸包

我们沿用卷包裹法的初始工作：选择最低点 p_1 作为起点，同时沿用卷包裹法的“价值观”，只要向左转就是“好样的”。与卷包裹不同的是，并不是一次确定一条最右的边，而是只要当前边在上一条边的左手方向，就把当前边作为“临时凸包”中的一条边，继续扩展；否则回溯，直到向左转为止，如：p_1p_2 从开始(a),(b),(c),(d)，这三步每一步的当前边都在上一条边的左转方向，故顺利扩展。但是到了(e)就不行了，p_5p_6 在 p_4p_5 的右手方向，说明 p_4p_5 的选择有问题，于是回溯（退栈）至(f)，考虑 p_4p_6，发现它又在 p_3p_4 的右手方向，于是再回溯至(g)，p_3p_6 在 p_2p_3 的左手方向，最终得(h)中的凸包。

上述算法带有简单的回溯，因此宜用栈实现，栈中存储的是到目前为止的“局部凸包”，如果发现问题，即当前边相对于栈顶边右转，就退栈，重新考虑以前决定，直至当前“局部凸包”正确为止。这个算法描述如下：

```
push(p1); push(p2);
i=3;
while i<=n do
  if pi 在栈顶边 pt-1pt 左手方向
       then push(pi) 并且 i++
       else pop();
```

那么这个算法是否比卷包裹法更高效呢？

均摊分析　答案是肯定的，只要看上述 while 语句执行了几次即可，每个点很显然都恰入栈一次，而最多出栈一次，因为每次出栈就永久性地放弃了这个点，它再也不会入栈。所以总共执行了 $2n$ 次，故时间复杂度是 $O(n)$。这种分析方法就是 1.4 节中提到的“**均摊分析**”（amortized analysis）（参见文献[86]）。

需要一个序　上述算法的输入是一个简单多边形，那么它能否运用于一般的平面点集呢？平面点集是一组无序的点，它没有多边形的顶点序列那样的“序”，所以，首先要对平面点集按某种“序”排序。另外，上述算法是否真的运用于任何简单多边形呢（我们举的例子只是非常简单的情况）？这个问题留待后文讨论。

最简单的直观的“序“应该是内部一点的极角序，如图 3-68 所示。

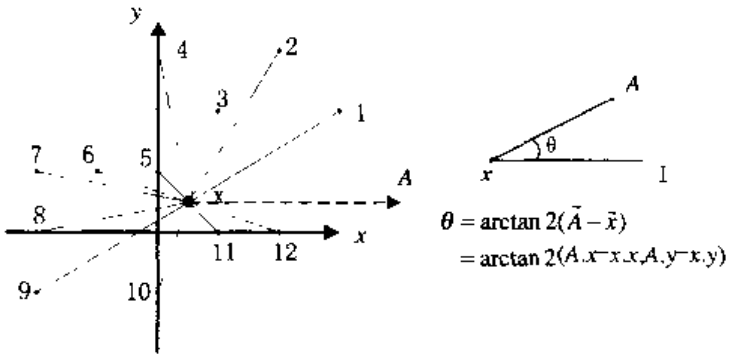


图 3-68　内部一点极角排序及 arctan2 函数

极角序的问题　先假设最简单的情况：没有三点共线。则任意三点组成的三角形内部的点必是凸包内的点，以某个这样内部点 x 为中心，点集中的所有点按关于的极角逆时针排序，如图。至于排序的起始位置，其实可任意三点定一条射线，为了简单起见，就以 x 点出发向右延伸，平行于 x 轴的一条射线（xA）为起始位置，于是得如图 3-68 所示的顺序。具体实现这个排序时，有几种可选的策略。

（1）由于这个序是循环的（绕了 360°），卷包裹法中的那个简单的叉积用于极角比较法已不适用于此种排序。

（2）一种简单的策略是回到原始的极角值。C 语言有个 arctan2 函数，它可以方便地求出每个点相对于 xi 射线的极角（在 0° ~360°），但是这会带来浮点误差，对于那种输入全是整数，不允许任何浮点误差的情况不适用。（当然我们可以设计分数比较的方法，但显然不漂亮）

（3）比较可行的还是用叉积，不过要注意循环的判断。例如可以得到两个向量分别在 xA 所在直线的上半平面还是下半平面；如果是同一半，再算两个向量的叉积。

另外，这个算法还有不少问题：起始点与终点的确定、x 点的确定以及三点共线。

（1）起始点与终点的确定

我们举的例子只是一种比较幸运的情况——起点恰在凸包上，事实上很容易发现第一个入栈的点（即起始点）永不退栈，也就是肯定在输出的凸包中。若起点事实上不在凸包上时，这个算法就会出错；另外，终点也有这个问题，它也必在输出中，所以我们要能保

证起点和终点都肯定在凸包上，参见文献[86]。

（2）排序参考点 x 的确定

前面假设了不存在三点共线，而这种假设是不现实的，当存在三点共线时，就要找到一组不共线的三点，再求其重心（当然假设不是所有点都共线，否则，凸包为一直线段——这种假设也将在以后的讨论中去掉），一种简单的线性做法[5]是每次取三个点，是共线则删去中间那个点，再选。由于每个点至多被删去一次，所以是线性的，不过好像还不够简单。

第一种算法　针对上述两个问题，提出一种非常简单有效的改进方法——把排序参考点移至一个肯定在凸包上的点。例如，就可取卷包裹法的起点（即取最低点，若有多个，则取其中最左者），如图 3-69 所示。首先这样很自然地在线性时间内解决了问题（2）。而且对于极角的比较也更简单了，因为这样一来就不存在循环的问题，可以直接用叉积比较。这同时也就解决问题(1)，因为现在极角最小和最大的两点（起、终点）都肯定在凸包上了。至此，已经得到了一种简单且高效的平面点集凸包算法，其时间复杂度为排序 $O(n\log n)$，扫描 $O(n)$，所以总的是 $O(n\log n)$，明显优于卷包裹法的 $O(n^2)$。

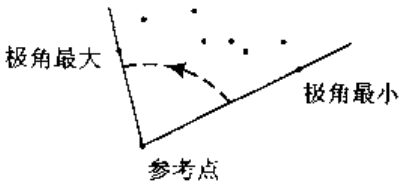


图 3-69　改进后的极角排序

共线点难题　但这种算法是不是十全十美了呢？让我们考虑第三个问题——三点共线和重点。重点比较容易解决，只要删掉即可。而三点共线则始终是困扰凸包的一个问题，大致有两种情况：凸包边上的共线和排序时的共线。凸包上的共线在卷包裹法中也讨论过。对于严格要求极点（extreme points）的问题，这个算法解决和卷包裹法一样，只要对于比较（叉积）采用严格比较。但是对于要求所有点的问题，这种算法就出问题了：考虑第一条边和最后一条边（即上述极角最小、最大情况），如图 3-70 所示，*ABC* 是第一条边，*EDA* 是最后一条边。我们要求 *ABCED* 都出现在结果中，这就牵涉到排序共线点（极角相同的点）的问题。

(a) 两种共线的例子：
排序时的共线 *ABCDE*；凸包边上的共线 *FGEH*

(b) 两种共线重合：
始边 *ABC* 和终边 *EDA*

图 3-70　共线问题

一种简单的策略是当极角相同时，取近者为先（相对于参考点），这样 B 能输出，D 却被弹出栈了。其实，不管采取何种策略，始边和终边总是矛盾的，而且排序时共线的问题也仅对这两边有影响（其他情况都在凸包内部，不论顺序如何都不影响，见图 3-70(a)）。当然，为了解决这个问题，可以采用近者为先，然后对终边特殊处理一下（如对终边上的

点另外排序）的办法。但是这样的做法总是权宜之计，因此有必要找一种更加简单、更加优美、更加通用的方法。

水平序和第二种算法　我们对“序”的问题做一改革，突破“极角”排序，改用水平序，即按 y 坐标排序，坐标相同的再按 x 坐标排序。

如图 3-71 所示，上文按极角排序的可重排。

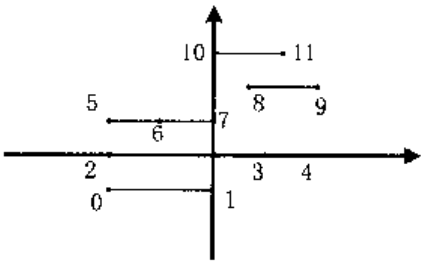


图 3-71　对 68 例子的水平序重排结果

这样做有两个明显的优势：

（1）排序比较更简单了，只是简单的比较，没有运算。

（2）起始点更好找了，就是排序后的 0 点。

这样做同样保持 Graham -Scan 的性质，只不过要把扫描过程分成两步，右链和左链。先做右链，以 0 到排序最后点（即最高点 11），再反向做左链（已经生成在右链上的点不考虑），从最高点 11 到 0。

最关键的，这样做在几个特殊之处也不错。起边（0-1）、终边（5-2-0）、最高边（11-10）都没问题，对于凸包上共线点的取舍，可以方便地采用与卷包裹法相同的严格、不严格比较来控制。进一步观察发现，这种新的 Graham-Scan，甚至连所有点都共线时（最特殊的一种情况）也成立，可以说是最简单、高效、优美而通用的算法。

3.3.3　凸包算法正确性与时间效率

我们在 3.3.2 小节已经得到了一个相对简单、高效且通用的 Graham-Scan 算法，现在还有几个更深入一些的理论问题需要解决：

（1）这个凸包算法是很直观的，但是其正确性尚待严格的证明。

（2）既然这个算法是 $O(n \log n)$的，那么有没有更优的算法？

（3）这个算法是否通用于任意简单多边形？（就是指以多边形顶点的顺序为序）

1．正确性

既然这个算法是“试探性”增长的，每一步的结果都是一个“局部”凸包，那么我们很自然地就想到利用归纳法来证明其正确性。

归纳法证明（参考文献[94]）　假设排序后的点集为$\{p_0, p_1, \cdots, p_{n-1}\}$，令 $CH(i)$代表$\{p_0, p_1, \cdots, p_n\}$的局部凸包，注意其显然包括 p_0 和 p_i。不失一般性，我们只考虑右链，左链同样可得。为了证明是凸包，要证：

（1）结果链是凸的；

（2）是包含当前所有点的最小凸图形。

对于（1），我们每次保证左转已经隐含了这一点。对于（2），根据前文 3.3.1 小节的定理 1，凸包上的点都是原点集的点（因此不能再小了），所以只要证：凸包已包含了点集内所有点。对于右链，只要证所有点都在右链左侧。

归纳边界：一开始栈内为 p_0 与 p_1，它们显然在右链上，考虑到 p_{i-1} 其右链为 $CH(i\text{-}1)$。

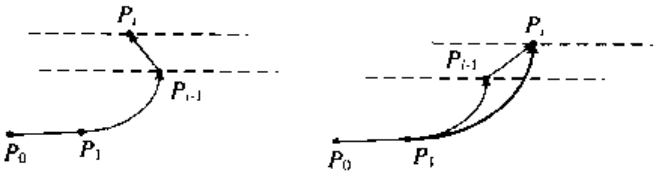
归纳假设：从 p_0 到 p_{i-1} 的点都在 $CH(i\text{-}1)$左侧。

现在准备加上 p_i 点，只要让新链 $CH(i)$在$\{p_0, p_1, \cdots, p_i\}$中所有点右侧。

有两种情况，向左转还是向右转，向左转时命题正确是显然的，因为 $CH(i)=CH(i\text{-}1)+\overrightarrow{p_{i-1}p_i}$，只是在老链上加了一条边而已，仍然保持凸。

向右转时复杂一些，连续退栈后形成图 3-72（b）中形状。此时其实新链比老链更右，根据归纳假设，$\{p_0, p_1, \cdots, p_{i-1}\}$即排序中低于 p_{i-1} 的点都在老链 $CH(i\text{-}1)$左侧，那么显然更加应该在新链 $CH(i)$左侧，加上 p_i 本身，我们就证明了$\{p_0, p_1, \cdots, p_i\}$中的点都在 $CH(i)$左侧。当然，当 p_{i-1} 与 p_i 的 y 值相同时也正确，于是原命题得证。

这种证明方法在对凸包相关问题的证明中尤为有效，后文将继续推广。



(a) 向左转的情形　　(b) 向右转的情形

图 3-72　Graham-Scan 凸包算法正确性的证明

2．时间效率下界与排序

在 3.3.2 小节中，我们已经对这算法的时间复杂度作了分析，排序为 $O(n\log n)$，扫描用均摊分析（Amortized Analysis）法，为 $O(n)$，所以总时间复杂度为 $O(n\log n)$。还有没有更快的方法（当然是指最坏情况下）？

转化思想　下面将从凸包与排序的关系角度来分析这个问题，因为排序问题的时间复杂度下界是早已确立的，为 $O(n\log n)$。

考虑排序问题 A：一组来排序的数（x_1，x_2，…，x_n），$x_i \geqslant 0$，其时间为 $S(n)$；

考虑凸包问题 B：在时间 $T(n)$内构造一个含 n 个点的点集的凸包。

我们想 B 用来解 A，当然时间为 $T(n)+O(n)$，$O(n)$是把 B 的结果转化为 A 所用的线性时间。

为了用 B 解 A，我们构造一组点（x_1^2，x_2^2，…，x_n^2），可以用凸包算法来求这个点集的凸包。显然，这些点又都在凸包上（相当于 $y=x^2$ 曲线是取 n 个点），如图 3-73 所示。（事实上任何凸函数皆可）凸包算法的输出是有序的（逆时针绕向），即假设（x_1，x_2，…，x_n）排序后为：

（x_{i1}，x_{i2}，…，x_{in}），则凸包的输出为：

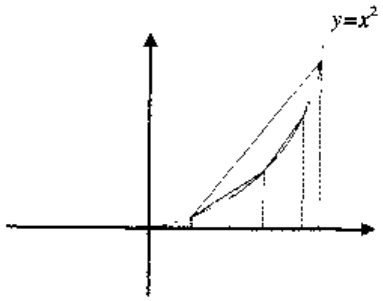


图 3-73　凸包与排序的关系

$$(x_{i1},\ x_{i1}^2) \to (x_{i2},\ x_{i2}^2) \to \cdots \to (x_{in},\ x_{in}^2) \to (x_{i1},\ x_{i1}^2)$$

所以凸包的输出可用于排序。

既然这样，如果凸包算法足够快，假设比最快的排序还快，那么完全可以用凸包取代排序，但这样就突破了排序算法已经确立的下界了。

所以凸包算法不可能比排序的下界还快。同时 3.3.2 小节中我们确实看到了 $O(n\log n)$ 的 Graham-Scan 的扫描法。

∴$O(n\log n)$是凸包算法的（可取）下界。

如果把要求放松，只要输出凸包上的点，而不要次序的话，直观的想法是时间复杂度可能会小一些，而事实上数学家已证明其下界同样是 $O(n\log n)$，参见文献[86]。

注意：刚才我们用转化方法求下界，其思想在下文许多讨论中会有广泛的应用。

3．多边形的凸包

前文引入 Graham-Scan 的时候，确实是举了一个多边形的例子。这是因为多边形相对于点集，多了一个序，那么 Graham-Scan 应用这个序是否肯定正确呢？

首先很容易想到复杂多边形显然是不行的，那么对简单多边形呢？当然，根据 3.3.2 小节的讨论，起点 P_1 本身就不一定在凸包上。所以显然要找一个肯定在凸包上的点开始 Graham-Scan，例如最低点中的最左点，以这个点为 P_1，其他点序号顺延。那么，现在这个算法是否正确了呢？

反例　其实我们也容易找到反例——绕圈情形。当做到 P_8 时，栈内局部是 $P_1P_2P_3P_4P_8$（到这一步还是完全正确的）。而接下来的几条边 P_8P_9，P_9P_{10} 都是左转，所以都自然入栈，于是最终形成 $P_1P_2P_3P_4P_8P_9P_{10}P_{11}P_1$ 的“凸包”。其中 P_4P_8 与 P_9P_{10} 相交，结果非常荒谬，如图 3-74 所示。

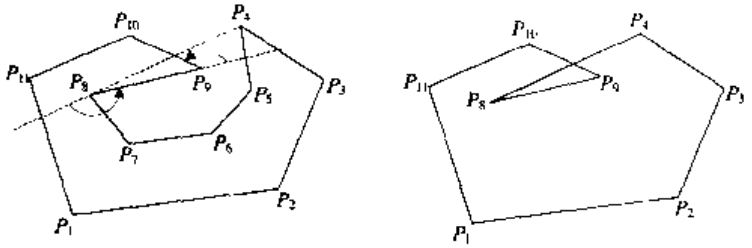
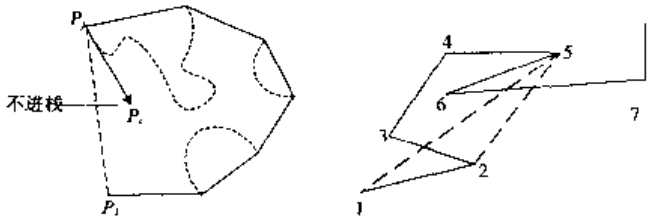


图 3-74　多边形凸包的反例

大家可能已经发现，这时 Graham-Scan 沿着多边形行进时，丧失了“全局方向感”。如本例中，P_8P_9 这一步走到了当前凸包 $P_1P_2P_3P_4P_8P_1$ 内部去了，于是陷入泥潭。那么能做一个小改进，如果发现当前尝试的边试图进入目前凸包内部的话，则“不进栈”，如图 3-75 所示。所谓进入内部，严格地说，设栈顶为 P_j，目前尝试 P_i，若 $\overrightarrow{P_jP_1} \times \overrightarrow{P_jP_i} > 0$ 则为进入内部。

能否改进

现在加入这个补丁，是不是就对了呢？刚才那个例子，显然是对了，但是请看图 3-75（b），做到 5 时凸包为 1—2—5，这时尝试 5—6，发现没有进入目前凸包，左转（相对于栈顶边 2—5），于是入栈，结果又出问题了。



(a) 当前边 P_jP_i 试图进入凸包内部　　(b) 反例

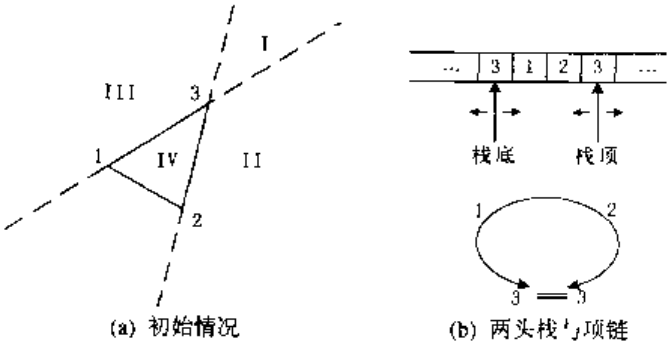
图 3-75　防止进入凸包内部及其反例

当然，可以针对这个例子再加补丁，规定 P_j 在 P_jP_i 右转的话也不进栈之类，但是，补丁加得越多，算法的正确性反而越不明朗。真正好的算法总是简单、优美、统一的，而不是靠分情况讨论、加补丁的。

Melkman 算法　所以，这里介绍一种非常巧妙的算法参见文献[88]。它的基本思想仍然继承 Graham-Scan，但用一个两头栈（deque，又译双端队列），也就是两端都可以进行增、删。它与经典 Graham-Scan 的最大区别在于，它不仅在栈底一端维护“凸性”，而且在栈底一端也进行维护。于是，它得到每时每刻都是一个真正的包含目前所有点的凸包。

初始时，最初三个点的三角形就是其凸包。如果 1—2—3 成右手系，则栈里是（底）3—1—2—3（顶），否则交换 1—2，反正保持栈里成右手系。

然后考虑 4，可能落在 Ⅰ，Ⅱ，Ⅲ，Ⅳ四个区域，如图 3-76 所示。



(a) 初始情况　　(b) 两头栈与顶链

图 3-76　Melkman 多边形凸包算法

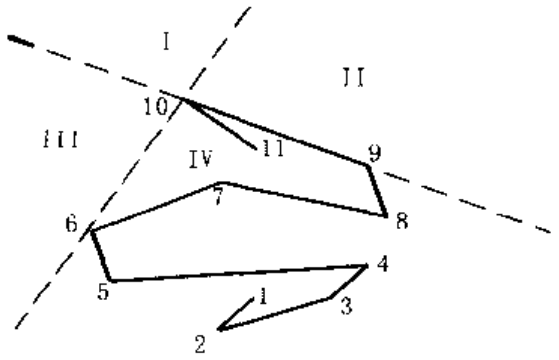
若是Ⅳ，走入了凸包内部，“误入歧途”，应该忽略，考虑下一个点，直到走出这个区域。

若是Ⅱ，则对栈顶一端进行 Graham-Scan 式维护（退栈直至左转），对栈底加上 4（因为对栈底来说 1—3—4 是凸的）。

若是Ⅲ，则与Ⅱ对称，对栈底进行维护（注意这里是退栈直至右转——或者统一地说，退栈直至“凸转”），同时对栈顶加上 4（凸）。

若是Ⅰ，则当作同时是Ⅱ，Ⅲ处理——即两头都是进行维护，这也是直观的。

于是如此循环，考虑所有点。图 3-77 是一个详细的例子。



(a) 到 11 为止的图

加入	deque
1	1 2 3
4	4 1 2 3 4
5	5 2 3 4 5
6	6 5 2 3 4 6
7	7 6 5 2 3 4 7
8	8 7 6 5 2 3 4 8
9	9 7 6 5 2 3 4 8 9
10	10 6 5 2 3 4 8 9 10
11	10 6 5 2 3 4 8 9 10

(b) 每访问一个顶点 deque 的变化过程

图 3-77　Melkan 算法的实例

其实这个算法说白了，就是两头 Graham-Scan 维护，栈顶一端始终保持左转，栈底一端右转（保持两边都是凸转）。另外一种角度看来，就是栈里面的序列始终保持“右手系”，栈顶和栈底永远是同一个顶点，就好像项链串起来一样（如图 3-76(b)）。

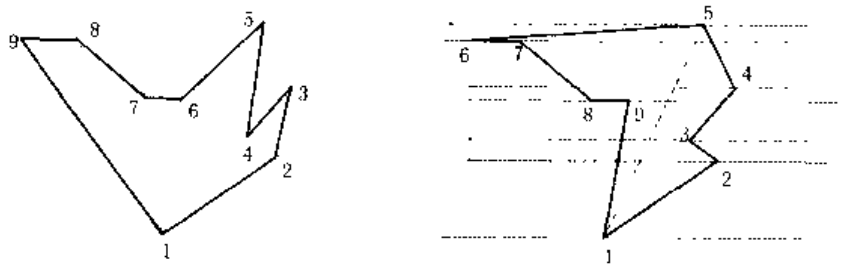
这个算法除了特别简洁、优美以外，还有一个优势：它是一种“在线算法”(online algorithm)，就是说它可以随时接收输入，而不用一次先全部输入完，也就是不用任何预处理，例如我们甚至不用先找一个肯定在凸包上的点（而事实上第一个点完全可能不在凸包上）。在线算法因此可以用于需要与用户对话的交互系统，而我们其他大部分算法只能用于批处理系统。

关于多边形凸包的进一步讨论，请读者参考习题 3.3.2 和 3.3.3。

4．对凸包和排序的进一步思考

点集凸包的新思路　现在有了线性时间的多边形凸包算法，不妨从另一个角度思考——能不能用它来改进点集凸包？如果能在较快时间内把一个点集变成一个多边形，就能在较快时间内求点集凸包了。当然，任意序会带来复杂多边形，显然不适用，所以一定要把点集变成简单多边形。例如我们很容易想到 3.3.2 小节讲的两种排序——极角序和水平序，就可以用来完成这个任务。

例如极角序和水平序的情形如图 3-78 所示。



(a) 极角序　　(b) 水平序

图 3-78　对于同一个点集，两种排序后的结果多边形

注意：水平序要像上 3.3.2 小节那样，先找上下端点，分为左右链处理（即端点连线左右分开）。

考虑到刚才已经证明的点集凸包的下界$\Omega(n\log n)$，如果“把点集变简单多边形”这步更快的话就要突破点集凸包的下界了。由此就可以得到一个有趣的结论：

定理　把一个点集变成简单多边形的时间复杂度的（可取）下界是$\Omega(n\log n)$。

而且，一般而言，$O(n\log n)$的算法都直接或间接与排序有关。

于是我们可以重新理解一种新的广义的 Graham-scan，共两步：

（1）把点集（通过排序）变成简单多边形，复杂度为$O(n\log n)$。

（2）用 Melkman 算法求简单多边形的凸包，复杂度为$O(n)$。

总的复杂度还是$O(n\log n)$。

由此可见，凸包和排序的关系的确是非常紧密的，这可能是由于凸包本身就代表了一种性质非常好的“序”，对点集求凸包的过程，无非就是排出这种序。这种序的优美性质，就是下面 3.3.4 小节中凸包应用的源泉。

3.3.4　应用举例

在前文 3.3.1 小节中，我们简单地预览了一下凸包的应用范围，并提出了三个例题。在这一小节中，将围绕“凸包改进时间复杂度”这一中心，具体讨论凸包这些应用实例。先从比较简单的“改进平均时间复杂度”入手，然后再研究“改进最坏情况时间复杂度”。

1．凸包改进平均时间复杂度

基本思想已经在前文中概述，无非是用凸包“等价替换”原点集或原多边形，在平均情况下，大大改进时间效率。

这类问题大多是“约束最优问题”，最简单的例子如：

【例题 1】求点集的直径

点集$S=\{p_0,p_1,\cdots,p_{n-1}\}$的直径$D$定义为任意两点间的最大距离：$D^{def}=\max_{i,j}(p_i-p_j)$

很明显，原始的算法是$O(n^2)$，但是，鉴于凸包是包围这个点集的外边界，我们能否先求凸包，然后就凸包上的点两两求距离呢？这里关键问题是：直径的两端是否必在凸包上。

这其实很显然：

（1）如图 3-79 所示，如果直径的两端a、b至少有一端点不在凸包上，我们可以延长线段ab直至与凸包相交得$a'b'$，显然$|a'b'|>|ab|$；

（2）进一步，与b'邻近的两个凸包顶点b_1，b_2，取$b^*=\arg\max_{b_i}|b_ia'|$，即$b_1$与$b_2$中与$a'$远者，必有$|b^*a'|>|b'a'|$；

（3）类似地，$a1$与$a2$也有类似的性质，取$a^*=\arg\max_{a_i}|b^*a_i|$，即$a_1$与$a_2$中与$b^*$远者，必有$|a^*b^*|>|a'b^*|>|a'b'|>|ab|$。

至此，通过反证法，证明了直径两端必在凸包顶点上，注意在证明中已经暗示了顶点必定是“极点”(extreme points)，即可删除所有共线点。

关于本例　这个例子给我们的启示是：要应用凸包，首先要证明对题目的要求来说，用凸包代替原点集是等价变换。而应用了凸包以后，如果只是像本例那样，起到了缩小搜

索范围，而没有实质性的改变算法的话，可以改进平均时间复杂度。这对于实际应用显然是意义重大的，但是在理论意义上，或从竞赛的角度看，意义不是最大。理论上最坏情况下的时间复杂度最重要，在竞赛中为了体现这一点，往往专门出这样的数据。所以应用凸包，最好是在用凸包替换了原点集后，即对凸多边形的特殊性质，寻找更好的算法，降低最坏意义下的时间复杂度。

对踵点　另外，就本例来说，可能读者会发现，求了凸包以后，再在凸包顶点范围内两两求距离似乎太傻了——很明显，相邻的顶点距离较近，而“对踵点”则距离相对较远。我们可以把时间集中在“对踵点”附近的搜索，这就是利用了凸多边形的性质（进而降低最坏意义下的时间复杂度），在下一小节 3.3.5 中将详细讨论这一思想。

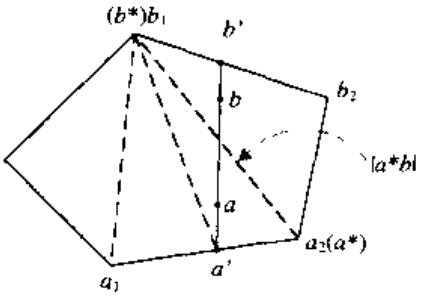


图 3-79　点集直径必在凸包顶点上取得的证明

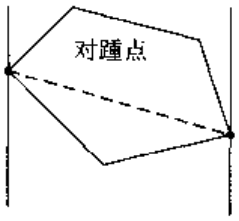


图 3-80　对踵点的例子

2. 利用凸多边形性质改进最坏情况时间复杂度

顺着上文的思想，来看两个比较简单的例子。

【例题 2】最小外接矩形[1]

对一个给定的平面点集 S，求其面积最小的外接矩形。

【分析】

我们先考虑如果不用凸包怎么做呢？乍看这个问题颇有难度，可以先考虑增加一个条件：矩形的边平行于坐标轴，则这个问题变得异常简单，只要计算水平、竖直方向的跨度即可。那么对于原问题，只要应用坐标轴旋转即可。这个思路最直观，就是模拟逼近，即枚举坐标轴的倾角 $\theta \in [0, \frac{\pi}{2})$，计算出点集在新坐标系下的坐标，再算水平、竖直方向的跨度即可。最后记录其中最小的矩形，如图 3-81 所示。

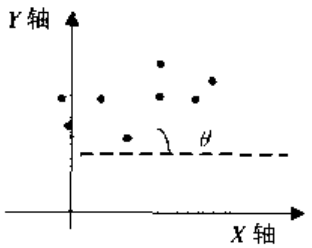


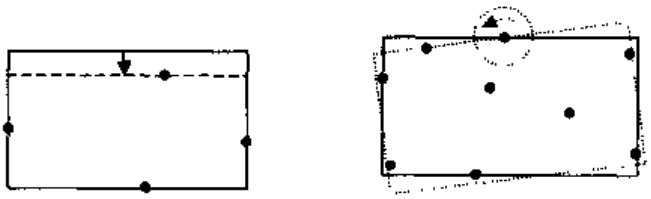
图 3-81　枚举旋转角度，模拟逼近求最小矩形

[1] 题目来源：UVA Problem Archive 10173, Smallest Enclosing Box

这个算法显然不尽如人意。因为模拟逼近算法始终无法得到精确解，而且随着精度提高，时间复杂度将大大增加。

关键问题是 θ 的变化范围是连续的，能否把它变成离散的呢？如果是离散的,我们很容易想到任意两点间的线段倾角，这就有限了（C_n^2 种倾角）。但这需要证明。

首先，最小矩形每一边上至少有一个点。这个很显然，如果一个点也没有，则可以把这条边往里推，直到碰到第一个点为止，如图 3-82 所示。然后要证明至少有一边上有至少两个点。这个证明困难一些，不过思路还是一样的，假设四个边都各只有一点，则可以对每一条边，以边上这个点为轴，转过一个小角度，直至碰到第二个点，得新矩形。四条边得到的四个新矩形中取面积最小者，证明其比原矩形面积更小即可，具体步骤留给读者。



(a) 每边上至少有一点　　(b) 至少有一边上有两点的证明

图 3-82　最小矩形

于是，只要对每对点形成的倾角 θ_i，计算 θ_i 方向上水平、竖直跨度即可。这个算法复杂度为 $O(n^3)$。

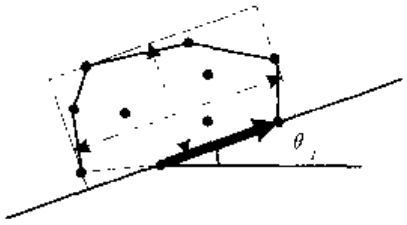


图 3-83　先求凸包，再沿凸包边计算外接矩形

相信读者肯定不耐烦了，既然最小矩形某一边必贴住两点，那么这两点之间的连边必是点集凸包上的边。于是只要求凸包，然后对凸包每一条边进行上述操作即可。这个算法在凸包求出以后，复杂度为 $O(n^2)$，这是最坏情况下的。

这个例题启示我们，以凸包代替原点集，不仅在平均情况下减少了顶点数，而且从根本上把一个无序的点集变成了有序的凸多边形，形成了线性的结构，拥有了一些优美的性质。

可能读者会问：若题中输入是一个多边形呢？是否已经是一个有序结构了呢？然而对这类问题而言，多边形与点集并无区别，因为我们只能证明最小矩形必有一边贴住两点，而无法证明必有一条边贴住原多边形一边。反例如图 3-84 所示。

图 3-84　最小矩形不一定贴住多边形一边

这与企图简单直接用 Graham Scan 在多边形凸包上所犯的错误类似。

聪明的读者可能会受前文例 1 的启发，发现“对踵点距离相对较远”的规律仍然适用，也就是说，水平、竖直范围的点可能可以在局部搜索，而不用考虑全部顶点，从而把目前复杂度 $O(n^2)$进一步改进。我们将在下一小节详细讨论这个问题。

【例题 3】点集分割[3]

有红、蓝两个点集，如图 3-85 所示，问是否存在一直线将它们分离（即在直线异侧）。

前文 3.3.1 小节介绍此题时提出了最原始的 $O(n^4)$算法。而求凸包以后，只要判断两个凸多边形是否分离，也就是两个凸多边形相交的问题。这里的做法就是简单套用一般多边形的相交：是否存在相交的边，若有，则不分离。若没有，也不代表分离，因为可能是包含关系，只需任取一红顶点，看是否也落在蓝凸包内，若没有，再取一蓝顶点，看是否落在红凸包内。凸包包含的情形如图 3-86 所示。

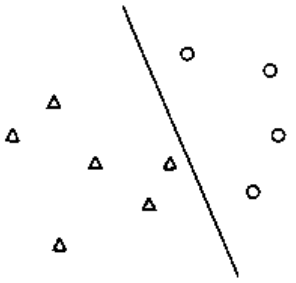


图 3-85　点集分割问题

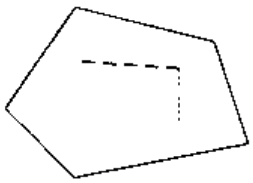


图 3-86　凸包包含的情形

整个判断费时 $O(n^2)$，这是相对于 $O(n^4)$是最坏情况下的改进，来源于多边形相对于点集所增加的“序”。当然，只有代表原点集所有点的凸多边形（凸包）才是等价变换。

当然，我们还要证凸包分离 ⇔ 原点集可被直线分离。

首先，证明“⇐”是显然的；

其次，要证“⇒”我们先考察两个凸包上距离最近处，有三种情况，如图 3-87 所示。

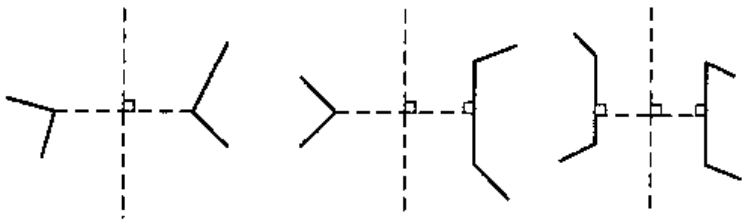


图 3-87　凸包分离的三种情况

① 两点之间，则取两点连线的中垂线为划分两个点集的直线，它显然能把两个点集分在异侧。

② 一点与一边之间，取这点到这边的垂线的中垂线。

③ 两边之间，则这两边必平行，于是取与这两边等距的平行线。

注意到这里其实只用到了一般多边形的“序”，而没有利用凸多边形的性质。我们猜

[3] 题目来源：UVA Problem Archive 10256

想，两个凸多边形相交是否应该比两个一般多边形相交容易一些呢？留待后文 3.3.5 小节讨论。

至此，完整地解决了这个问题。不过，凸包的应用尚有很多，这里举的三例只是抛砖引玉，希望读者能发现更多的实例。

在 3.3.5 小节中，我们将详细讨论这三个例题中提出的两个改进想法（例 1 和例 2 的“对踵点”、例 3 的凸多边形相交），利用凸多边形的优美性质，把平方阶的算法降为线性，同时把凸包的应用上升到一个更高的层次。

3.3.5 凸多边形的深入讨论

1. 旋转卡壳——一种作用于凸包的通用模型[7]

在 3.3.4 小节中两次提到了“对踵点”的想法，这里我们从上文[例 2]出发讨论。

运动规律 提出的问题是：对于每条凸包边，每次再求矩形另外三边贴住的点（假设是 T，L，R 三点）是否太浪费了呢？是否可以根据其运动规律局部化？

最小矩形四条边的运动规律如图 3-88 所示。

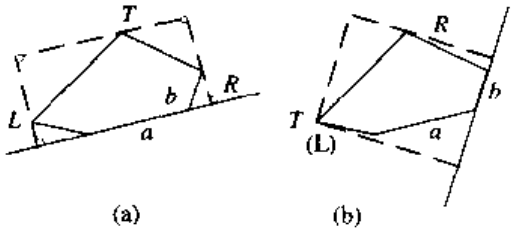


图 3-88 最小矩形四条边的运动规律：从(a)到(b)，转过了一条凸包边，T，R，L 三点也逆时针转动

考虑当从 a 边转向下一条凸包边 b 边时，T，L，R 三点的运动直觉告诉我们，它们也在向逆时针方向运动，而且当凸包边兜了一圈时，T，L，R 三点也恰好遍历了凸包顶点一圈。所以，原先 $O(n^2)$的算法是不必要的，可能只要遍历一圈，即 $O(n)$时间即可。当然，以上直觉思维需要严密的证明，现在换一种更加简单而易于证明的思路。

假设当前状态矩形下边贴住凸包边 $P_{i-1}P_i$，上侧贴住 P_k，右侧 P_j，左侧 P_l，并设 P_iP_{i+1}，P_jP_{j+1}，P_kP_{k+1}，P_lP_{l+1} 四条边与相应矩形边之间的角度为 θ_i，θ_j，θ_k，θ_l（如图 3-89 所示）。

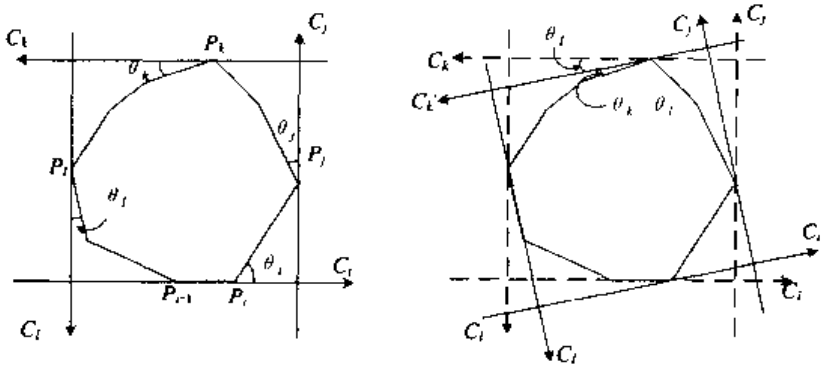


图 3-89 旋转卡壳：选择最小角前进

这时把上下左右四条矩形分别称为 C_i，C_j，C_k，C_l。下一步状态无非是选择某一个卡壳转向下一条边，例如 C_j 旋转 θ_j 角度，贴住 P_jP_{j+1} 边，同时其他三个卡壳相应旋转同样的角度，重新获得上、左、右三个方向的范围。

最小角前进　卡壳关键问题选择哪一条卡壳前进，3.3.4 小节中算法每次总是选择 C_i，而后再根据其他三个卡壳的角度，扫描一遍凸包是的点以获得其他三个卡壳所贴住的点。现在转向选择 θ 角最小的卡壳前进，例如上图中，θ_1 最小，就选择 C_l 卡壳前进,即右转 θ_1 角度，贴住 P_{i+1} 点，这时另外三个卡壳也相应向右转 θ_1 度，更新所贴点信息，但是，读者很快就会发现，转后 C_i，C_j，C_k 依然贴住三点 P_i，P_j，P_k。这是非常显然的。

即：选择转角最小的卡壳前进时，其他三个卡壳贴住的点不变，这无疑是一个极好的性质，也就是说我们只要每次更新 4 个 θ 值，选择一个最小的前进，计算当时的面积，直到旋转一圈为止。

问题是什么叫旋转一圈，或者怎样判断结束?

一种简单的可行的策略是当四个指针都走过一圈时为止，这可以保证正确，但太浪费，其实只要联系 3.3.4 小节中所提出的第一个算法模拟逼近法，就会发现卡壳的转角只要在 $[0,\frac{\pi}{2})$ 范围内即可，于是在现在这个算法中可以累计每次的转角，发现$>\frac{\pi}{2}$ 即可停止。

以上通过“旋转卡壳”（Rotating Calipers）方法得到了最小包容矩形的一个更优的算法，其复杂度为求凸包+旋转（线性），所以总时间复杂度为 $O(n\log n)$，以下是此题算法的进化历程。

$$\underset{\text{非有效、不精确}}{\text{模拟逼近法}} \xrightarrow{\text{枚举倾角}} \underset{O(n^3)}{\text{枚举倾角法}} \xrightarrow{\text{凸包}} \underset{O(n^2)}{\text{凸包贴边法}} \xrightarrow{\text{旋转卡壳}} \underset{O(n\log n)}{\text{凸包 + 卡壳}}$$

旋转卡壳与对踵点　其实，旋转卡壳法对凸包相关问题有很大的用途，下面看看 3.3.4 小节例 1：点集直径。

现在由于只要求一个方向上的距离，所以只要两个平行卡壳即可。同样选较小角转，转过 $\frac{\pi}{2}$ 即可。这时每次贴住的点叫“对踵点”（antipodal），直径总在一些对踵点处取到，这个算法加上求凸包也是 $O(n\log n)$，具体实现时有一些与最小矩形不同之处，请读者作为练习。

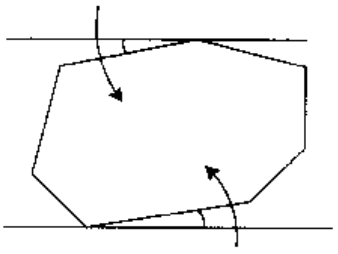


图 3-90　平行卡壳与对踵点

比较这两道题目，会发现，这类题目往往要求一个最值（与距离相关），原始的枚举法效率太低，而用了凸包之后，可把平均情况复杂度大大降低，而最坏情况下往往无能为力（如本例）。但由于凸包的优美性质，旋转卡壳可以把搜索局部化，把复杂度降为线性，只是加上求凸包才使整个复杂度为 $O(n\log n)$。

2．凸多边形相交

3.3.4 小节的例 3 最后化为两个凸多边形相交。下面继续讨论这个问题。

两个一般多边形的相交，在图形学中有非常重要的应用，最常见的例子就是各种画图软件的填充功能。但是，图形学只考虑视觉效果，而计算几何是纯粹的几何计算，其算法

是截然不同的。其最坏情况下的时间复杂度，很显然是 $O(nm)$，n，m 分别为两个多边形的顶点数，这是因为仅交点数就能有 nm 个（参考文献[86]）如图 3-91 所示。

现在，如果两个多边形都是凸的，复杂度是否可以降低呢？

追逐式赛跑文献[86]　首先，凸图形相交有个很好的性质：**交集仍是凸的**。所以，原多边形的每条边在结果多边形中至多出现一次，即结果多边形顶点数≤$n+m$，那么时间复杂度能否为 $O(n+m)$呢？我们容易想到，一开始两个多边形上各取一条边，只要两个多边形上的边，“你追我逐”地依次前进，保证每个交点都不漏掉，当每个多边形都绕行了一圈之后即可，如图 3-92 所示。

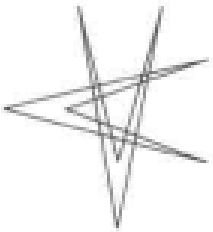


图 3-91　凹多边形相交的平方个交点

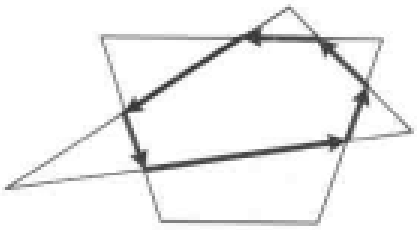


图 3-92　凸多边形相交的追逐式前进

现在的问题就是如何实现“追逐式”前进，以保证经过每个交点。容易想到，如果两边不相交，若一条边 A“指向”另一条边 B，则应前进 A，因为这样 A 的下一条边就有更大可能交到 B。图 3-93 给出了追逐式前进的几种情况。

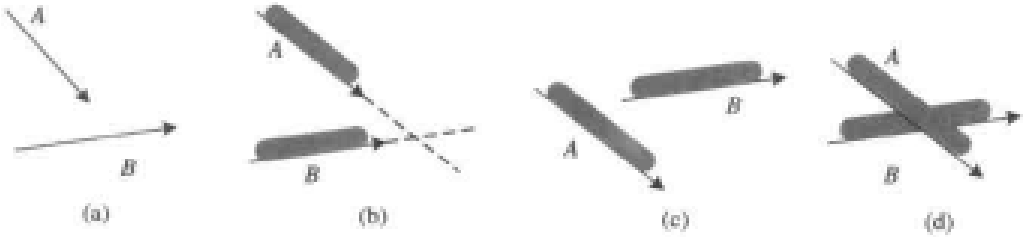


图 3-93　追逐式前进的几种情况

所谓“指向”，是指一条边 A 在边 B 所在直线的一侧，并指向此直线，如上图 a，A 指向 B，但 B 不指向 A。

如果 A，B 互相指向，则必有一边在另一边内部。所谓内部 $IH(A)$，是指 A 左侧的半平面，因为多边形正绕向时内部在每条边左侧。此时，若 $A\in IH(B)$则前进 B，否则前进 A。如上图（b），选 B 前进。

如果 A，B 互不指向，也是应该前进外侧的边。也就是终点在另一条边外部的边，如上图（c）中的 A。这些也都是为了增加相交的可能性。

如果 A，B 相交，则必有一边从另一边内部穿过外部，而另一边从其外部穿向内部，这时必须选择穿向外部者前进，如上图（d），选 A 前进。

当然，相交时还有一步很重要的操作：修改 inside 标记，一开始可定 inside=0，谁也不在对方内部，一旦相交，就改变 inside，inside=1 和 inside=2 分别表示多边形 P 在内部和多边形在 Q 内部，直到下一个交点为止，都要输出内部多边形的顶点。判断整个算法结束的条件是：再次走到第一个交点或两个多边形都兜了一圈而无交点。当然，后者需要判别的

是 $P \subset Q$，$Q \subset P$，还是 $P \cap Q = \phi$。

以上是整个算法的描述，其复杂度显然是线性的。但是，还有很多细节没有考虑，另外特别难处理的就是一些特殊情形，如交点恰好在一边上，边重叠等，如图 3-94 所示。这些问题如何简洁、统一地处理，留给读者考虑。

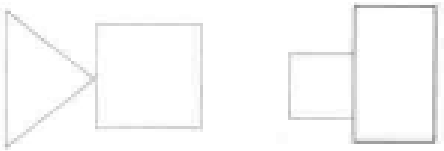


图 3-94　凸多边形相交的特殊情况举例

练　习　题

思考题：

3.3.1　我们在凸性定义一部分讲到，对于线性组合 ax_1+bx_2，几何意义是二维平面，加上 $a+b=1$ 是直线，再加上 $a \geqslant 0$，$b \geqslant 0$ 则是线段，以及推广到概率分布的情形，下面用 Σ，∀ 语言来表示，如图 3-95 所示。

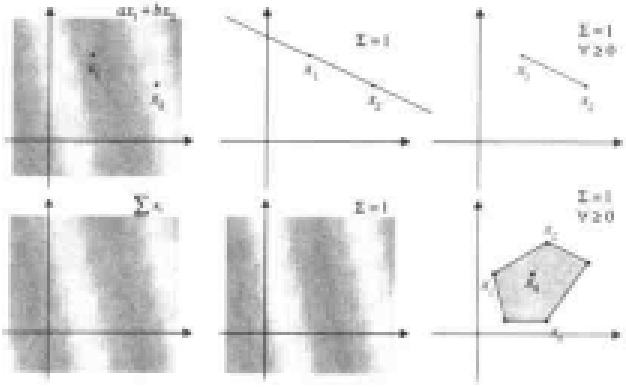


图 3-95　Σ，语言描述的凸组合和概率分布

那么现在请分别画出对于两点和多点情况下的以下要求：

（1）$\Sigma \geqslant 1$，$\forall \geqslant 0$

（2）$\Sigma \leqslant 1$，$\forall \geqslant 0$

（这个题目很简单，但对于理解凸性定义和凸包建模应用（参见习题 3.3.5，很有帮助）

3.3.2　我们知道，对于多边形的凸包，原始的 Graham-Scan 会出错。但是我们发现，直观地讲，“大多数”情况下仍然适用。那么你能否精确地刻画一下，哪种类型的简单多边形可以适用？或者说这些多边形都具有怎样的特殊性质？

3.3.3　我们曾经尝试对多边形凸包改进 Graham-Scan，但是失败了，你能否想一些其他的更加严格的改进？对你的算法证明或举反例。

WEB　多边形凸包自 70 年代提出以来，一直是一个广受关注的有趣问题。我们文中介绍的 Melkman 在线算法是迄今为止最好的算法。感谢加拿大 McGill 大学计算机系师生关于这个问题的总结性研究。关于多边形凸包的历史以及代表性的算法，请参见文献[90]。关于 Graham-scan 所适用的多边形刻画，请参见文献[91]。

3.3.4　证明包住点集的最小矩形至少有一边上有两个点。

编程题：

3.3.5　导弹和王国[①]

有若干王国，每个王国由若干城市组成。恐怖分子向这些王国里发射若干导弹，如果一颗导弹落在一个王国的领土里，导致这个王国的电力供应系统完全瘫痪。现在试问电力瘫痪的总面积。王国的边界定义为能包容所有城市的最小周长的折线。注意每个王国的城市数≤100，但导弹数可能很多。

（此题作为非常简单的综合编程练习，要求写得尽可能优化）

3.3.6　实现追逐式赛跑的凸多边形相交算法。注意有许多细节要考虑周全，如果统一地处理退化情况则最好。

3.3.7　咖啡[②]

有三种不同类型的咖啡杯，还有一些包裹，每种包裹的形式是（S_1，S_2，S_3），代表这种包裹里第 i 种类型的咖啡杯有 S_i 个（$1 \leqslant i \leqslant 3$）。每种包裹都有无限多个。现在问，能否取一部分包裹，使得这些包裹里面三种咖啡杯的数量之和是一样的。例如假设我们有 4 种包裹：（1，2，3），（1，11，5）（9，4，3）和（2，3，2），可以取 3 个（1，2，3），1 个（9，4，3）和 2 个（2，3，2），形成（16，16，16），就符合要求了。

输入 N 个包裹类型（$3 \leqslant N \leqslant 10^7$，注：原题是 1000），问能否符合要求。

（提示：联系我们本节开始处的定义，这个题目可以转化为三维凸包，然后再投影为二维凸包，化为判断一个定点是否在一个点集凸包内的问题。最后要指出，其实这个判断不需要以求凸包为前提，而存在线性做法，参见前文习题 3.1.7。当然，这适合于点集很大，而要判断的定点很少的情况，如果反过来，则显然是先求凸包的好。）

3.3.8　GIF 混合[③]

GIF 文件是一种存储图象的流行方法，但是，作为一种陈旧的文件格式，GIF 受到一些在其发明的时期所使用的显示硬件的限制。其中之一就是，调色板中的颜色最多只能有 256 种。这意味着在一个 GIF 文件中，从上百万种颜色中只能选出最多 256 种颜色表现所有的颜色。不过，通过对颜色的合理选择和用一种称作“抖动”的技术近似表现缺少的颜色，照样能生成比较高质量的图象。因为每种颜色都以三元组（r，g，b）的形式表示红、绿、蓝的亮度，所以颜色可以表示成三维坐标系中的一个点，这个坐标系分别以红、绿、蓝的亮度作为 X、Y、Z 轴。调色板中的颜色是三维坐标系中的一个固定点，而“平均颜色”可以被看作一个可移动的点，它的位置由拖拉附着在“平均颜色”和产生这个颜色的调色板中的颜色之间的弹簧来决定。如果调色板中的颜色环绕在我们期望的颜色的周围，不管离此颜色多远，将“平均颜色”拉到期望的颜色的附近的弹簧都可以附着在那儿，即“抖动”可以实现。但是，如果调色板中的颜色都在期望颜色的一侧，弹簧就不可能穿过期望颜色与调色板中的颜色之间的平面，“抖动”不能实现。现给出一个调色板的各个颜色

[①] 题目来源：UVA Problem Archive 109
[②] 题目来源：UVA Problem Archive 10089
[③] 题目来源：ACM/ICPC Regional Contest South California

（RGB），问能否调出一个特定的颜色。

（提示：本题和 3.3.7 题极其类似，基本上可以归纳为：判断一个点是否在一个 n 维凸包内，可以化为 $n-1$ 维球面投影上张角范围。然而这个问题高维还是不好解决，所以可以通过 $n-1$ 维线性规划来处理。例如本题可以化为二维线性规划（半平面相交，见后文 3.4.5 的详细讨论））。

3.4　几种常用的特殊算法

以上，我们讨论了计算机几何的基本内容：位置关系，最简单的几何体——多边形和多面体，以及最基本的计算几何模型——凸包。在这一章中，我们转向算法的讨论，主要介绍几个在计算几何中通用的特殊算法：首先是矩形几何有关的离散化及其扫除法，然后结合凸包介绍分治法和增量法，最后是一个随机增量算法的专题。

3.4.1　蛋糕被切成几块？——离散化法

本小节和 3.4.2 小节主要讨论矩形几何的两种基本方法——离散化和扫除法。当然它们在非矩形几何中也有很好的应用，参见习题。

离散化法是一种预处理方法，它能把无限空间中有限的个体映射到有限空间中去，以此提高算法的时空效率。

其基本思想很简单，就是当输入数据的范围（比如坐标的范围）是无限空间，或者输入数据有大量重复时，可以作一映射，从输入的无限空间到逻辑上的有限、有序的空间，同时避免重复。

先从一道例题讲起。

【例题 1】锡刀[1]

设一块很大的蛋糕（边界无限），现用一把刀去切，切痕水平或垂直，如图 3-96 所示问最后蛋糕没切成了几块？输入均为整点，范围[-10 000，10 000]，切痕数≤1 000。

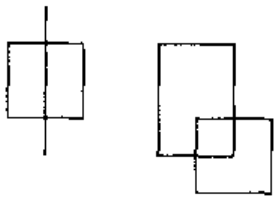


图 3-96　切蛋糕(Tin Cutter)

【分析】

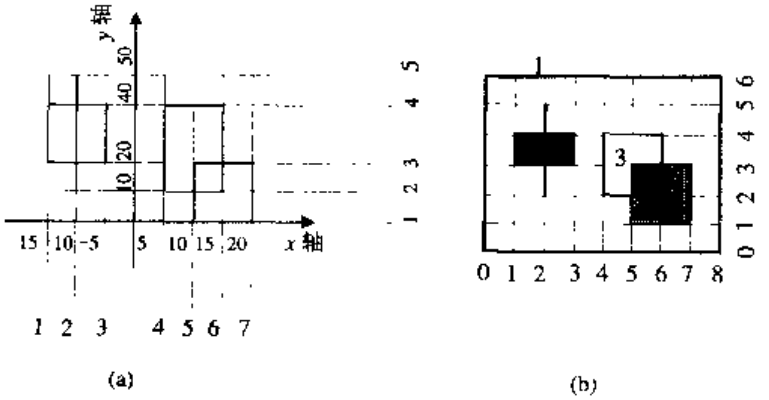
鉴于输入是整点，又是有限范围，有人就会想直接开内存，但是内存毕竟有限，再说

[1] 题目来源：UVA Problem Archive 308

如果是实点呢？所以，像这样的范围，还是只能认为“无限范围”，故必须离散化。例如图 3-97(a)分别对 x 方向和 y 方向进行了离散化。这样就把原来的输入范围 x 轴[-15,20]映射为 x'=1~7，y 轴[0,50]映射到 y'=1~5，即

$(x, y) \mapsto (x', y')$：

$[-15,20]\times[0,50] \mapsto [1,7]\times[1,5]$



(a)　　(b)

图 3-97　切蛋糕的离散化法

于是，根据新坐标重新画图（见图 3-97（b）），显然，图形拓扑性质没有改变。

这时再来解这道题就相当容易，而且事实上这也只是一个搜索而不是几何问题了。为方便起见，我们加了一个外框，然后只要用 Floodfill（参见 1.3 小节）就可以了，先从（0，0）开始扩展一遍，就遍历了所有“空白点”。再依次寻找尚未遍历的点，遍历一遍就得一“空洞”（就是题中所求的“块”）。直至全遍历完。如本例（图 3-97（b））得五个空洞，即切成了 5 块（相当于求洞的连通集）。

通过这道例题，初步能体会到离散化法的适用范围：

① 原空间是稀疏的。如本题[-10 000，10 000]，但切痕只有 1 000，相差一个数量级。

② 原空间范围是无限的，例如本题其实也可以是无限范围。

③ 原空间数据有较多重复时，也可以通过离散化来优化，例如本题。

④ 映射必须保持必要性质，某些不必要的信息，如长度，可不保持（如本例，映射后发生了伸缩）。但与解题相关的信息必须保留，如本例中的“序”，即映射后，坐标顺序不能混乱，即原空间中 $x_1 < x_2$，新空间中必有 $x_1' < x_2'$，称“保序性”。（拓扑不变）

上述四条中，④涉及题目要求和实现细节，是最需要注意的。有些题目不需要“保序性”（如单纯的频度统计），而人量几何题需要保序性（否则几何信息就没有意义了）。要“保序”，自然涉及排序，当然，还要考虑重复值的情况。两种比较高效的简单的算法如下：

① 每次两分查找，查到则不插入，否则插入，效率为 $O(n\log n)$。

② 先全部存下，再 qsort 一遍。为了排除重复，再线性扫描一遍，也是 $O(n\log n)$。

其实，有一部分几何题在保留重复的情况下，仍然成立，如本例。若不删除重复，则可能得到下图，容易发现这不影响搜索。当然，这是细节问题。

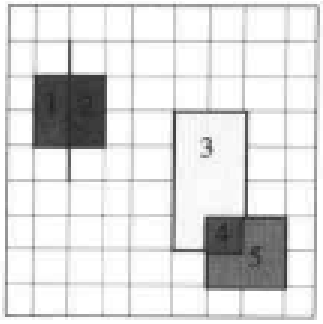


图 3-98　不删除重复的离散化

以上我们初步看到了离散化的应用，其实更多的时候，由于“保序性”的存在，离散化法更多地结合“扫除法”使用，这就是 3.4.2 小节要讨论的主题。

3.4.2　切蛋糕的周长和面积——扫除法

扫除法又名平面扫描法，一般以保序的离散化作为预处理，主要用于两类问题：矩形几何问题和线段相交问题，这里我们初步讨论矩形几何问题，更深入的讨论请参见文献[92]。

从 3.4.1 小节的“切蛋糕”问题继续讨论开去。3.4.1 小节是水平、竖直地切，这里再加上一个限制：必须用矩形的刀来切，即切痕是一个矩形，且矩形的边水平、竖直，如图 3-99 所示。

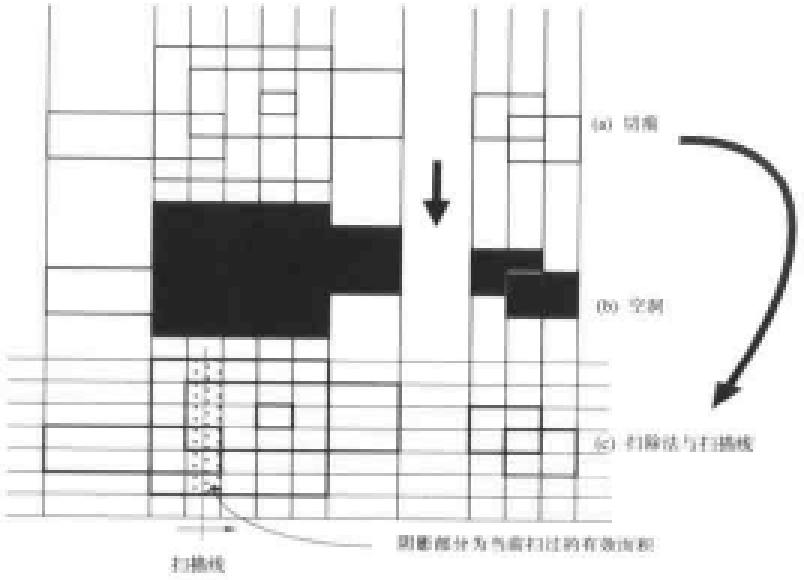


图 3-99　切蛋糕的面积和周长

如果只考虑切出的轮廓，即假想切出的蛋糕都“掉”下去了，这留下了如图 3-99（b）中的空洞（阴影部分）。现在我们不仅仅是要求“切成了几块”，而是进一步问：这些“洞”的周长、面积是多少？

这个问题可能更困难一些，不过，我们依然从离散化入手，经过保序的、删除重复的

离散化，得到图 3-99（c）。

根据离散化的基本性质，水平（竖直）方向的每一个狭长形小单元片，即 $y'_i < y' < y'_{i+1}$（或 $x'_i < x' < x'_{i+1}$）的区域中，没有水平（竖直）线段。例如上图（c）打阴影的竖直单元片中没有竖直线段（否则的话，这一片还会被继续分下去，直至单元片）。这是一个极好的性质，于是就以这样的竖直单元片作为研究对象。

想象一根竖直的扫描线，从左到右依次扫过每片竖直单元片，每扫过一片，就可计算当前片中的有效面积，即空洞面积。这只要知道哪些水平线段之间的区域是空洞就可以。

考虑一般情况的竖直单元片如图 3-100，此时由于前述性质，任何水平线段与此竖直片的关系只有两种：

① 不跨越（自然与本竖直片无关）；

② 跨越。此时水平线段左端在左边界上或更左，右端在右边界上或更右。

考虑一根水平扫描线从上向下扫过所有跨越这一竖直片的水平线段，由于水平线段分为矩形上边（+）和下边（-），有一计数器初值为 0，经过一上边，加 1，经过一下边，减 1。此时我们发现从 0→1 的跳变为空洞的上边，从 1→0 的跳变为空洞的下边，大于 0 的格子为空洞内部，其实格子上的数反映了此格被几个矩形覆盖。

这样空洞的周长和面积就很好求了：

- 对于面积，只要累计大于 0 的格子面积即可。
- 对于周长，只要累加 0→1 跳变 1→0 跳变处线段长度即可。

这个算法的实现也是比较简单的，只要一个方向离散化，另一个方向排序，然后在离散化的方向上扫描每个小区间，再从另一方向上依次扫描每条跨越这个小区间的线段，维护计数器即可。

当然，可以发现重复不删除也不影响计算，只要两个方向上都排序即可。总时间复杂度为 $O(n^2)$。这显然不是最优的结果，用 1.4 小节介绍的数据结构可提高到 $O(n\log n)$。

另外，请注意这个“扫除法”对这类问题有很好的通用性。例如如果可以改变对“空洞”的定义，可以对包含空洞的切痕数加以范围限制（本题是≥1），或奇偶性限制，例如图 3-101 中空洞定义为奇数个切痕覆盖。

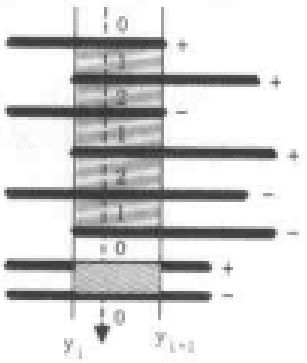


图 3-100　扫除法的竖直单元片

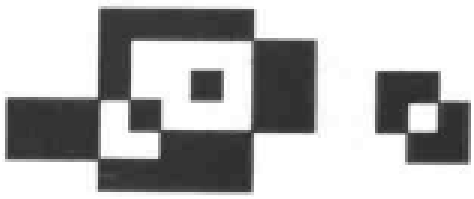


图 3-101　奇偶性的空洞

更进一步，切痕也不一定是矩形的，可以是圆形或者其他图形，参见本节习题。

3.4.3　凸包与快速排序——分治法

本小节和 3.4.4 小节结合凸包介绍两种常用的基本算法思想——分治法和增量法。

分治法的基本思想，在本书第 1 章中已经讨论过了，这里将把它广泛应用于计算几何中。

还是从凸包谈起，3.3 节已经说过，求凸包的算法非常多，而且很多算法都基于“局部凸包”这个概念，如 Graham-Scan。其实这样一种局部的思想不就正好符合分治法的化大为小的基本思路吗？于是，我们能否把点集也划分成几个“局部”，然后再求各局部（子集）的凸包，最后合并起来呢？当然，要保证这样划分出来的凸包是不相交的，如图 3-102 所示。

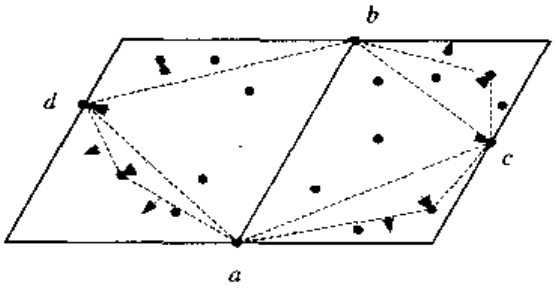


图 3-102　分治法求凸包: *Quick Hull*

分治思想　很容易想到，Graham-Scan 中根据最高点 b 和最低点 a 把凸包分成左链和右链，那么怎样把左链和右链再按最高点继续分下去呢？这里的子链的“最高点”不存在现成的判据。我们容易想到 ab 右边的点中选择相对于 ab 的极点（Extreme Point），即离 ab 最远的点 c，很显然$\triangle abc$ 中的点都不在凸包上，把它们全部忽略。而 c 显然是凸包的点，于是就再分为两个点集，ac 右下角的点集和 bc 左上角的点集。按照同样的原则继续下去，直到点集中只剩一个点为止。对左链同样处理。当然，这种（删除内点）分治的思想也可以理解为：求一个点集的凸包，化为求其三个子集的凸包，再求并的凸包，其中一个凸包是显然的三角形，其他两个继续递归求解。

Quick Hull 文献[86]　读者可能已经发现，这种分治求凸包的思想与排序中的快速排序非常相似，所以它也被称为 Quick Hull 算法，下面用类似 Quick Sort 的方法来分析其复杂度。

和 Quick Sort 一样，Quick Hull 每次递归，也分为三步：

（1）把点集 S 划分为子集 A 和 B；

（2）递归做 A；

（3）递归做 B。

设$|S|=n$，这样“划分”这一步与 Quick Sort 中一样，是线性的，记为 $O(n)$。设$|A|=\alpha$，$|B|=\beta$，显然 $\alpha+\beta\leqslant n-1$（因为 C 点不在两子集中），最坏情况下 $\alpha+\beta=n-1$（当 S 集中$\triangle abc$ 没有点时）。设计算点集 S 的时间为 $T(n)$，于是：

$$T(n)=O(n)+T(\alpha)+T(\beta)$$

最坏情况分析　这个式子与 Quick Sort 的递推式也是一致的，只不过不同在于 $\alpha+\beta\leqslant n-1$ 而 Quick Sort 中 $\alpha+\beta$ 恒为 $n-1$。当然，我们只讨论最坏情况 $\alpha+\beta=n-1$。

仿照 Quick Sort，对划分情况讨论 $T(n)$。

（1）(最坏情况中的)最好情况：均分即 $\alpha=\beta=\frac{n}{2}$（这里忽略奇偶数），则

$$T(n)=2T(\frac{n}{2})+O(n)$$

根据 Master Theorem（参见本书第 1 章 1.2.3，或者文献[93]），此时

$$T(n)=O(n\log n)$$

这是点集随机均分情况，对应 Quick Sort 中的乱序。

（2）（最坏情况中的）最坏情况：分划最不均匀，即 $\alpha=0$，$\beta=n-1$,于是：

$$T(n)=T(n-1)+O(n)$$

这时，有：

$$T(n)=O(n^2)$$

这对应 Quick Sort 恰为顺序或逆序情形。

总之，这个算法最快情况下是平方阶的，结果与 Quick Sort 一致。

下面讨论它与 Quick Sort 不同之处。

平均情况分析

很显然，所谓上述最好与最坏情况，都是已基于一个最坏情况：$\triangle abc$ 中无点。而如果每次递归都满足这一点，那么显然整个点集的点全在凸包上，再加上(2)的最快情况，事实上这种情况的概率是极低的，根据凸包的分布讨论，下面在最一般情况下，讨论其复杂度。

根据“最高点”和“极点”的定义，显然 ab 以右的点全分布于平行四边形 $abcd$ 中，则 $S_{\triangle abc}=\frac{1}{2}S_{abde}$。若均匀分布，则$\triangle abc$ 中的点只占点集一半，如图 3-103 所示，于是：

$$T(n)=O(n)+T(\alpha)+T(\beta),\quad \alpha=\beta=\frac{n}{2}$$

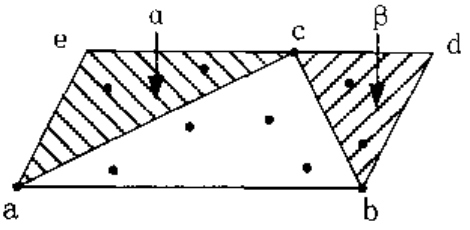


图 3-103　增量凸包算法平均时间复杂度分析：均匀分布

（1）（平均情况中的）最好情况：

$$\alpha+\beta=\frac{n}{4}\Rightarrow T(n)=2T(\frac{n}{4})+O(n)\quad T(n)=O(n)$$

（2）（平均情况中的）最坏情况：

$$\alpha=\frac{n}{2},\ \beta=0\Rightarrow T(n)=T(\frac{n}{2})+O(n)\quad T(n)=O(n)$$

所以无论哪种情况，平均时间复杂度总是线性的。

对于这个 Quick Hull 算法还有许多细节有待讨论，如：（1）凸包是共线点的两种选取对于 c 点选择的影响，（2）所有点均共线情况；（3）时间复杂度的输出敏感性（即取决于最终凸包上点的个数）。但考虑到这些问题前文 3.3.2 小节“凸包的实现”都有类似的详细讨论，这里限于篇幅，就不再讨论了，请读者自行给出解答（方法是比较巧妙的）。

然而，这还只是求凸包的一种分治法，更加典型的分治法求凸包请参见习题 3.4.4。

3.4.4　凸包的又一种求法——增量法

以上从凸包问题出发，讨论了分治算法，这一小节将介绍凸包的另一种直观的解法：增量算法（Incremental Algorithm）。

相对于分治法，增量算法可能应用得不这么广泛。所以本书第 1 章也没有作为常用算法介绍，但是其思想同样是非常直观易懂的。分治是把一个问题分为若干个相似的规模不等的小问题，而增量则是把一个问题化为规模刚好小一层的子问题。这是自顶向下（top-down）的角度，如果从自底向上（bottom-up）的角度，分治是从叶子结点开始树形增长上升，而增量只是一个线性递增，从规模为 1，每次加 1，直到规模为 n，写成递归式：

分治　$T(n)=\Sigma T(a_i)+f(n)$　　增量　$T(n)=T(n-1)+g(n)$

增量的思想很象最简单的(第一)数学归纳法证明过程，先从 $n=1$ 开始，假设作到 $n=i$，都满足这一性质（子问题），考虑加上 $n=i+1$ 的情况，最后也成立（解决原问题）。

同样以凸包为例，其实 Graham-Scan 算法已经非常类似增量，每一步都是一个局部（到目前为止已考虑的子集的）凸包，每次尝试在上一步基础上加上一个新点，构成新一步的凸包，下面讲一个更纯粹的增量算法，其增量思想更加典型。

首先，像往常一样，先排除特殊情况：假设没有三点共线。

（归纳边界）先任取三点（一般就按点集的顺序），这三点组成的三角形显然是这三个点的凸包。

（归纳假设）然后每次向现有凸包 H_{k-1} 内，尝试增加一新点 P_k，欲求 $H_k=Conv\{H_{k-1}\cup P_k\}$，则分为两种情况：

① 这个点已在凸包内，忽略；

② 这个点不在目前凸包内，则要扩充凸包,使之成立包括 H_{k-1} 和 P_k 凸包。

这样的算法框架已经非常清楚了，要继续求精的只有两件事件：（1）判断一个点是否在凸包内；（2）如何使一个凸包扩展，以包容一个外点。

对于（1），本章前文 3.2.3 小节已经作了详尽的分析。凸包的性质使这一步非常简单。而且还介绍了一个更有挑战性的两分算法，时间效率为 $O(\log n)$。

（2）这步其实也很简单：只要决定原凸包上哪一段去掉，以新点代替即可。我们只要

想从 P_k 作凸包的两根切线，而显然切点处转角相反。最简单的做法，只要考察哪个点的相邻两边相对 P_k 的转角方向相反即可。

图 3-104 右，$\overrightarrow{ab}$ 与 $\overrightarrow{bc}$ 相对 P_k 反向；e 点左右，$\overrightarrow{de}$ 与 $\overrightarrow{ef}$ 相对 P_k 反向，所以删去 b 到 e 之间的点(c,d)，代以 P_k 得 a,b,P_k,e,f 为新凸包点序列。

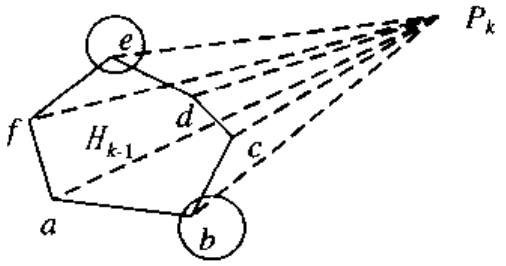


图 3-104　增量法扩展凸包——切线与切点

当然，实现时，可以循环兜凸包，发现从右转变左转（b 点），删去以后的点，直到再发现从左向右转（e 点），回到原凸包。当然，这个做法每次是线性的，总的算法就是平方的。但是聪明的读者可能已经发现，可以两分法求出这两个切点。这样做每次是 $O(\log n)$ 的，结合上面“判断点是否在凸包内”的两分法 $O(\log n)$，总的算法就是 $O(n\log n)$的，达到了凸包算法的下界。

这里只是最简单的一个例子，下文我们将看到，增量算法往往与随机算法结合，成为“随机增量算法”，在一大类问题中大大改进了平均情况时间复杂度。

3.4.5　专题——随机增量算法

上文讲了增量算法。现在我们看看如何把它应用到线性规划中去。线性规划是一类非常经典的问题：

最大化　$c_1x_1+c_2x_2+\cdots+c_dx_d$

满足　$a_{1,1}x_1+\cdots+a_{1,d}x_d\leqslant b_1$；

$a_{2,1}x_1+\cdots+a_{2,d}x_d\leqslant b_2$；

……

$a_{n,1}x_1+\cdots+a_{n,d}x_d\leqslant b_n$；

通常我们是通过单纯形法来解决，然后其算法相当复杂，现在从几何的角度来看这个问题，先从 d=2 说起。

1．几何认识

最大化　$c_1x_1+c_2x_2$

满足　$a_{1,1}x_1+a_{1,2}x_2\leqslant b_1$；

$a_{2,1}x_1+a_{2,2}x_2\leqslant b_2$；

……

$a_{n,1}x_1+a_{n,2}x_2\leqslant b_n$；

很容易看到每一条件实际是一个半平面，我们定义：

直线 l_i：　$a_{i,1}x_1+a_{i,2}x_2=b_i$；

半平面 h_i：　$a_{i,1}x_1+a_{i,2}x_2\leqslant b_i$；

目标矢量　$\vec{c}=(c_1,c_2)$

其实，线性规划是要在所有半平面交集中求 $\vec{c}$ 方向最远的点。d=2 时是平面上的线性规划，推而广之，d 维线性规划的每一个约束条件就是一个 d 维半空间，目标函数就是一个 d 维矢量代数代表最大化的方向，可行区域就是这些半空间的交集，解就是可行区域中方向最远处，如图 3-105 所示。

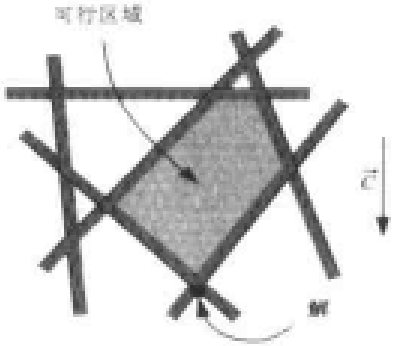


图 3-105　两维线性规划的几何解释：可行区域、目标矢量和解

一般说来，解的情况有 4 种，如图 3-106 所示。

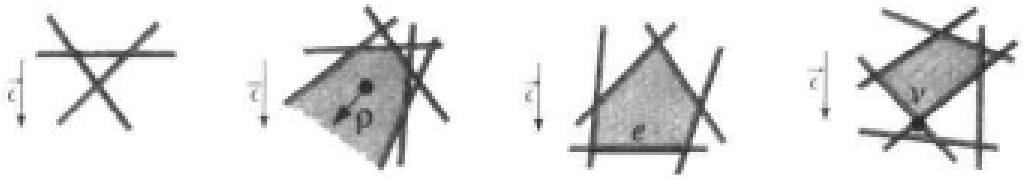


图 3-106　解的 4 种情况

为了避免(b)的情况，往往人为加上一个框[$-M$，$+M$]×[$-M$，$+M$]，M 取一个很大的数，不影响解即可。所以，这里只考虑有界情形(a)、(c)、(d)。

现在，我们只要用半平面相交求得可行区域，可行区域必是一个凸多边形（这步与求星形多边形的核一样），再对其每个顶点（最值点必在凸多边形顶点处取得，证明简单，不赘述），选出 $\vec{c}$ 方向上最远者，若有两个并列，则为(c)情形。

2．半平面相交——求多边形的核

鉴于本书前文已多次提到半平面相交这个问题，这里在讲述了分治法和增量法的基础上，对其做详细讨论，得到两个比较优的算法。

如果按常规做，每次用一根直线去截一个多边形，其复杂度应为 $O(n^2)$。但是我们很容易想到，每次的多边形都是凸的，而一根直线截一个凸多边形，最多与两条边相交。由于凸多边形的特殊性质，可以用两分法，在 $O(\log n)$时间内找到这两条边，从而整个算法是 $O(n\log n)$的。这个思路和前文 3.4.3“增量法”小节的例子——增量法求凸包非常相似。

我们曾经说过，分治法和增量法有很多相似之处，那么这里分治思想是否也适用呢？

如果用分治法，把 n 个半平面两分，分别求出各自的可行凸多边形，然后再求两个凸多边形的交，前文 3.3.5 小节“凸多边形相交”已经给出了求两个凸多边形交的线性算法 $O(n+m)$，所以，比分治法的时间复杂度为：

$$T(n)=\begin{cases}O(1), & n=1\\ O(n)+2T(\frac{n}{2}), & n>1\end{cases}$$

∴总的 $T(n)=O(n\log n)$

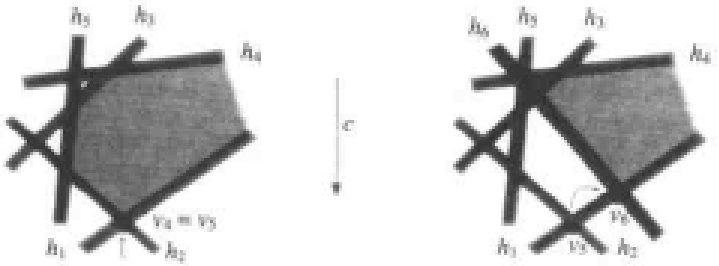
至此，得到两个最坏情况下 $O(n\log n)$的算法求解二维线性规划，它比单纯形法简单，而且在二维上还更有效。同时，这也解决了前文悬而未决的问题：求多边形的核的最快算法。

3. 增量算法

但是，毕竟目标不是求核。可能会有读者发现，求取整个可行区域（核）是否太浪费了？我们其实只要一个 $\vec{c}$ 方向最远点而已，至于其他边界的形状，并不关心。于是就有了下面的增量和随机算法[12]。

在增量算法中，就不再保留可行区域 C_i，而是只保留当前最佳点（当前解）v_i，当新增加一个半平面 h_{i+1} 时，如果 $v_i \in h_{i+1}$，则当前解不变；当 $v_i \notin h_{i+1}$ 时，说明当前最优解不在新的半平面内，要重新确定最佳点。

新增加一个半平面时的两种情况如图 3-107 所示。



(a) $v_i \in h_{i+1}$，当前解不变　　(b) $v_i \notin h_{i+1}$，重新确定最佳点

图 3-107　新增加一个半平面时的两种情况

重新确定最优解又可能有两种情况：至此无解，或 $v_{i+1} \in h_{i+1} \cap C_i$。$C_i$ 为第 i 步的可行区域。

如果排除无解情况，容易想到，新直线 l_{i+1} 的内法线 $\overrightarrow{N}_{i+1}$（指向其内部 h_{i+1}）必然有向上的分量（假设 $\vec{c}$ 竖直向下），即点积 $\overrightarrow{N}_{i+1}\vec{c}<0$。又原可行区域的最低点 $v_i \notin h_{i+1}$，所以，新直线 l_{i+1} 必然是新可行区域的一条下边，即新的最低点必在 l_{i+1} 与原可行区域的两个交点上，参见图 3-107（b）。于是得一重要结论：$v_{i+1} \in l_{i+1}$（当然也可以用反证法证明）。

所以我们只要求线段 $l_{i+1} \cap C_i$ 中 $\vec{c}$ 方向最远点。

为了求线段 $l_{i+1} \cap C_i$，只要用 $h_1, h_2, \cdots, h_i$ 去截 l_{i+1}，求交集即可，这只相当于一维线性规划。

很显然，这是又一个可行而且简单易实现的算法。但是，它是否改进了时间复杂度呢？我们非常失望地发现，它不仅没有改进，而且还更慢了——其复杂度是 $O(n^2)$（最坏情况），这与 3.4.4 小节给出的直接求交集的算法还差，尽管 $O(n\log n)$的算法的确比较难实现。

但是一个很重要的假设是：最坏情况，即对于增量算法，每一次 v_{i+1} 都不在 h_i 中。然而这种可能性是很低的，能否考察一下其平均情况下的表现呢？毕竟在现实中，大部分情况不是最坏情况。

首先考虑一下所谓最坏情况：无非是每次添加一个半平面时都发现原最优点 v_i 不在新半面内，需要重新确定最优点。一种非常容易想到的实例如下：l_i（i=3,4…）为一组平行线，于是 v_i 随着 i 的增加不断沿线 l_2 上移（每次都改变最优点），然而，如果把顺序倒一下，按照 $i=n\to 3$ 的顺序添加，我们发现 v_i 的位置一次也不改变，此所谓最佳情况。

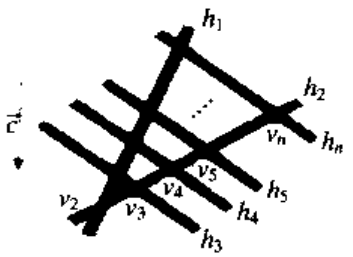


图 3-108　最坏情况的例子

以上分析使我们联想到快速排序及其复杂度分析。快速排序，其平均复杂度是 $O(n\log n)$，其最坏情况复杂度是 $O(n^2)$，且最坏情况发生在恰为顺序（或逆序）的情况（对应上例中 h_2，h_3，h_4 顺序）。

为了改进快速排序算法，人们常用的一种简便易行的策略是：随机化（Randomization），即在执行快速排序前，首先对输入数据进行一次随机洗牌（Shuffle），把原顺序洗乱，这样可以有效地避免那种非常不幸的恰为顺序或“接近顺序”，因为这些最坏（或较坏）情况发生的几率毕竟是极小的（往往是人为的）。

这种想法提醒我们，能否对本节中提出的增量算法也作一次随机化，以避免最坏情况呢？进一步地快速排序，随机化以后，期望（平均）时间复杂度可达 $O(n\log n)$，那么增量算法的期望时间复杂度可达多少？是否比我们一开始提出的最直观的半平面求交算法更好？这些问题就是下面要讨论的内容。

4. 随机增量算法

对于随机化算法，我们所要做的只有两件事情。

（1）给出一个高效的随机洗牌算法，注意其效率，至少不能差于随机化以后增量算法的效率，即不能拖整个算法的“后腿”，最好是线性的。

（2）分析随机化以后增量算法的期望效率，所谓期望效率，是指效率的数学期望。此时，随机化的结果可能是 $n!$种顺序，当然这些顺序是等概率的，所以，期望效率就是指这 $n!$种情况下的平均效率。

上述（1）显然是比较简单的，而且其效率上限还取决于（2），而（2）是这个算法的核心，所以我们先讨论（2）。

假设 p_i 代表第 i 个半平面加上去时最优点的变化，p_i=0 表示不变，p_i=1 表示变，则总时间复杂度为

$$\sum_{i=1}^{n} O(i) \cdot p(i)$$

我们要求上式的数学期望（平均值），则要用到数学期望的“线性”性质，有：

$$E(\sum_{i=1}^{n} O(i) \cdot p(i)) = \sum_{i=1}^{n} O(i) \cdot E(p(i))$$

于是问题转化为求 p_i 的数学期望。

我们可以想到，一个最优点是由两个半平面相交而得（两线交于一点），而在加上半平面 h_i 时发生最优点变动，则 h_i 必是确定这个新最优点的两个半平面之一（如图 3-109 所示），而 h_1~h_i 的顺序是随机的，所以 h_i 是两个半平面之一的几率是 $\frac{2}{i}$。因此总的复杂度的期望值为：

$$\sum_{i=1}^{n} O(i) \cdot \frac{2}{i} = O(n)$$

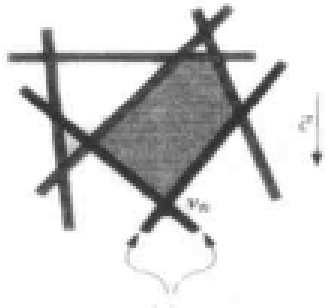


图 3-109　一个最优点由两个半平面确定

至于问题（1），一种常用且很简单的线性算法是：每次从 0 到 k-1 个数中选择一个与第 k 个交换(k=n→2)，即：

```
for k:=n downto 2 do
swap(A[k],A[random(k)]);
```

其中 random(i)函数产生一个从 0 到 i-1 的随机数。

至此，我们得到一个期望效率为 $O(n)$的算法，请注意对于它不存在最坏情况的输入，就像对快速排序一样，经验主义的观点认为，任何数据一般只要随机几次，总能得到较好的运行时间（特别适用于 ACM/ICPC 竞赛），从这点上说，它（至少在应用上）比原来的半平面相交算法更优一些。

5．应用举例：最小外接圆

上述随机增量算法，较好地（期望意义上）解决了二维线性规划问题，而且还可以推广到 d 维情形，研究表明，d 维的期望复杂度也是线性的。其实，这种随机+增量的思想是极有启发意义的，这里再举一个类似的例子——最小外接圆问题（MiniDisc）。

最小外接圆是一个很基本的计算几何问题。就是给定平面上的一组点，求一个最小的圆，包含所有这些点。

最原始的想法是：枚举所有的三点确定的圆（外接），查看每个的点是否都在圆内（或圆上）；再枚举所有的以其两点连线为直径确定的圆。这个算法显然是 $O(n^4)$的。

平面几何的改进算法　下一个进步一些的算法可能需要一些平面几何的知识。

任意枚举两点 A，B，在其连线上方所有点对此线段的张角最小者设为 C，设$\angle ACB=\alpha$，在其连线下方张角最小者为 D，设$\angle ADB=\beta$（若没有，可设为 π），只要 $\alpha+\beta\geqslant\pi$，则$\triangle ACB$ 或$\triangle ADB$ 的外接圆可包含所有点。若 $\alpha+\beta=\pi$，则 AB、CD 四点共圆，如图 3-110 所示。

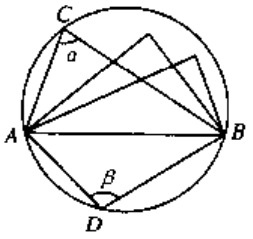


图 3-110　最小外接圆的改进算法：四点共圆

由圆周角小于圆内角定理知，$\alpha+\beta>\pi$ 时，A，B，C，D 含于一经过 A，B 的圆内。

于是，我们得到一个 $O(n^3)$的算法，这可能已经是最坏情况下最好的确定性算法了。要想继续改进，可以再应用随机增量算法。

需要指出的是，以上两个算法可以加上一个很浅显的改进——先求凸包。但是，凸包所带来的改进是与输入数据相关的，即存在最坏输入，而我们上面所述的随机化算法则与数据无关，不存在最坏输入。

改良算法　沿用前面的思路，先取两点，构成一个两点的圆，然后就像每次增加一个半平面一样，每次考察下一个点，否则就要“扩张”当前圆，使之能容纳下一个 p_i。

与前文类似，有这样一个很好的定理：p_i 不在当前圆（记 D_i 为）内时，必在“扩张”后的圆上，即包含 p_1，p_2，…，p_i 的最小圆必经过 p_i 点，这论证稍繁，由于篇幅原因，请读者自己思考或者参考文献[94]。重要的是它把最小外接圆问题引入了与前面半平面的求交问题极为类似的轨道上来了。

现在我们就要像 3.3 做的“一维线性规划”一样，重新构造一个圆，既经过 p_i，又包容所有点。这里，一种比较简单的思路是，既然 p_i 已经必是圆上一点，那么再枚举另一点，以此两点为 A，B，应用上述“张角”法。

这种算法，若不考虑随机化，复杂度是 $O(n^3)$，若考虑了随机化以后，期望复杂度是多少呢？

此处，沿用前边的分析方法，首先若每次 $p_i\in D_{i-1}$，则总复杂度为 $O(n)$。那么发生 $p_i\notin D_{i-1}$的概率多大？就像前文“两线定一点”一样，这里“三点定一圆”，所以自然是$\frac{3}{i}$，应用“张角”法的复杂度 $O(i^2)$，所以，总的复杂度为

$$O(n)+\sum_{i=2}^{n}O(i^2)\cdot\frac{3}{i}=O(n^2)$$

即这是一个期望复杂度为平方的算法，比单纯“张角法”又进了一步。可是，聪明的读者可能已经发现这种“折衷”的算法应用随机增量的思想并不彻底，其实是一种妥协性的东西，能不能抛开“张角”法，想一种纯粹的而且是更优的随机增量算法呢？

请读者顺着这个思路思考下去，最后将能得到一种期望复杂度为线性的算法。

练　习　题

思考题：

3.4.1　解决 3.4.3 小节最后遗留的 Quick Hull 的细节讨论。

3.4.2　解决 3.4.5 小节最后的问题：彻底的随机增量的最小圆算法，并仿照文中的方法分析其期望复杂度。

3.4.3　随机增量的二维线性规划算法是（期望）线性复杂度的，那么高维线性规划是否也适用这种方法？如果是，那么其（期望）时间复杂度呢？另外，如果你得到了（期望）线性的最小圆算法，能否拓展到高维空间？

编程题：

3.4.4　另一种分治法求凸包。除了我们 3.4.3 小节讲的分治法以外，还容易想到另外一种更加直观的分治法：先把点集分为左右两个子集，分别求凸包，再合并。由于两个子凸包肯定不相交，所以合并是非常方便的（至少能做到线性的，参见习题 3.2.8）。实现这个算法并详细分析其复杂度。

3.4.5　圆盘问题[①]

给出许多圆盘，如图 3-111 所示，已知它们的放置的坐标位置、半径。现在要把它们依次放到桌面上，所以一些先放的圆盘可能会被一些后放的圆盘盖住。问最后有多少圆盘可见？（提示：离散化）

3.4.6　最大内空凸多边形[②]

平面上有一些点，如图 3-112 所示，求以其中一部分点为所有顶点的最大的凸多边形。

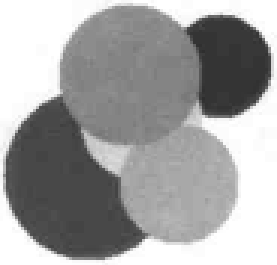


图 3-111　圆盘覆盖

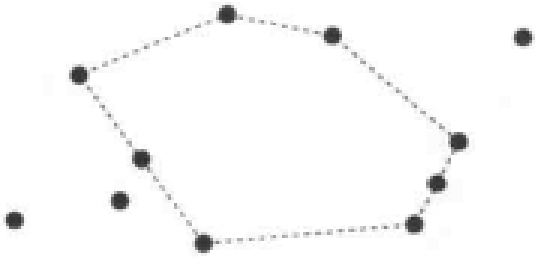


图 3-112　最大内空凸多边形

（提示：离散化/扫描法有一个很好的性质，即一个条状的区域中没有点，能否用于这道题呢？另外，能否突破水平/竖直的离散化/扫描法呢？最后，你的算法是几阶的，能否优化到立方、平方甚至线性的？）

---

① 题目来源：ACM/ICPC Regional Contest Hakodate 2002

② 题目来源：ACM/ICPC Regional Contest NWERC 2002, Picnic

参 考 文 献

英文书籍

[1] Donald EKnuth. The Art of Computer Programming. Pearson Education Inc, Addison Wesley Longman Inc, 2002

[2] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein. Introduction to Algorithms. Second Edition. The MIT Press

[3] Clifford A Shaffer. A practical Introduction to DATA STRUCTURES AND ALGORITHM ANALYSIS. Pretice-Hall Inc, 1997（中译本：张铭，刘晓丹译. 数据结构与算法分析. 北京：电子工业出版社，1998）

[4] Richard A Brualdi. Introductory Combinatorics (3rd Edition). Prentice Hall（中译本：冯舜玺，罗平，裴伟东译，卢开澄，冯舜玺校. 机械工业出版社，2002）

[5] Robert J Vanderbei. Linear Programming: Foundations and Extensions. Second Edition. Department of Operations Research and Financial Engineering, Princeton University, Princeton, NJ08544, 1996

[6] Reinhard Diestel. Graph Theory. Second edition. Springer-Verlag New York, 2000

中文书籍

[7] 吴文虎，王建德. 实用算法的分析与程序设计. 北京：电子工业出版社，1998

[8] 江文哉. 金牌之路——高中计算机竞赛辅导. 西安：陕西师范大学出版社，1999

[9] 殷人昆，陶永雷，谢若阳，盛绚华. 数据结构（面向对象方法与 C++描述）. 北京：清华大学出版社，1999

[10] 傅清祥，王晓东. 算法与数据结构. 北京：电子工业出版社，1998

[11] 《现代应用数学手册》编委会. 现代应用数学手册（离散数学卷）. 北京：清华大学出版社，2002

[12] 戴一奇，胡冠章，陈卫. 图论与代数结构. 北京：清华大学出版社，2002

[13] 卢开澄. 组合数学（第 3 版）. 北京：清华大学出版社，2002

[14] 潘承洞，潘承彪. 简明数论. 北京：北京大学出版社，1998

动态规划

[15] David Eppstein, Zvi Galil, Raffaele Giancarlo. Speeding up Dynamic Programming, 1988

[16] Zvi Galil, Kunsoo Park. Dynamic Programming with Convexity, Concavity, and Sparsity, 1992

游戏论

[17] Haurie A,.Krawczyk. J An Introduction to Dynamic Games, 2000

[18] Avierzri S Fraenkel. Arrays, Numeration Systems and Frankenstein Games, 2000

[19] Avierzri S Fraenkel. Combinatorial Game Theory Foundations Applied to Digraph Kernels, 1996

[20] Laszlo Csirmaz. Connected Graph Game.

[21] Christensen, JDaniel Mark Tilford. David Gale's Subset Take-away Game.

[22] Avierzri S Fraenkel. Heap Games, Numeration Systems and Sequences 1998

[23] Avierzri S Fraenkel. Multivision: A Game of Arbitrarily Long Play.

[24] Aleksandar Pekeč. A Winning Strategy for the Ramsey Graph Game, 1995

[25] Sripriya Venkataraman. Survey of Results in Impartial Combinatorial Games and an Extension to Three-player Game 2000

[26] Avierzri S Fraenkel Two-player Games on Cellular Automata.

[27] Avierzri S Fraenkel Virus Versus Mankind.

数学

[28] Daniel M Gordon. A Survey of Fast Exponentiation Methods. Journal of Algorithms 27, 1998, 129~146

[29] Guy Melançon. Lyndon Factorization of Infinite Words, 1995

[30] Jean-Paul Allouche, André Arnold, Jean Berstel, Srećko Brlek, William Jockusch, Simon Plouffe, Brue E Sagan. A Relative of the Thue-Morse Sequence 1999

[31] Bayreuth ABetten, Kiel. DBetten Regular Linear Spaces.

数据结构

[32] Mauricio Marín. An Empirical Comparison of Priority Queue Algorithms, 1997

[33] Gerth Stølting Brodal. Worst Case Efficient Data Structures. BRICS Dissertation Series, 1997

[34] Mauricio Marín On The Pending Event Set And Binary Tournaments 1997

[35] Diab Abuaiadh, Jeffrey H Kingston. Are Fibonacci Heaps Optimal?, 1994

[36] Ralf Hinze. A Simple Implementation Technique for Priority Search Queues, 2001

[37] Haim Kaplan, Robert E Tarjan, Kostas Tsioutsiouliklis. Faster Kinetic Heaps and Their Use in Broadcast Scheduling. 2001

[38] Brinkmann A, Graf Th, Hinrichs K. The Colored Quadrant Priority Search Tree With An Application to The All-Nearest-Foreign-Neighbors Problem, 1994

[39] Gudmund Skovbjerg Frandsen, Sven Skyum. Dynamic Maintenance of Majority Inforation in Contant Time per Update. BRICS Report Series, 1995

[40] Dany Breslauer. Saving Comparisons in the Crochemore-Perrin String Matching Algorithm,

1995

[41] Leszek Gąsieniec, Igor Potapov. Time/Space Efficient Compressed Pattern Matching, 2001

[42] Arne Andersson, Stefan Nilsson. A New Efficient Radix Sort, 1994

[43] Lau K K. Top-down Synthesis of Sorting Algorithms, 1992

[44] Kunihiko Sadakane. Unifying Text Search and Compression Suffix Sorting, Block Sorting and Suffix Arrays, 2000

[45] Roberto Grossi, Jeffrey Scott Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching, 2000

[46] Haim Kaplan, Nira Shafrir, Robert E Tarjan. Union-find with deletions, 2001

[47] J.A.La Poutré. New Techniques for the Union-Find Problem, 1989

[48] Susanne Albers, Jeffery Westbrook. Self-Organizing Data Structures. 1998

组合数学

[49] Marlán Trenkler. Connections – Magic Squares, Cubes and Matchings, 2001

[50] Hiroshi Imai, Satoru Iwata, Kyoko Sekine, Kensyu Yoshida. Combinatorial and Geometric Approaches to Counting Problems on Linear Matroids, Graphic Arrangements and Partial Orders, 1996

[51] Martin Löbbing, Ingo Wegener. The Number of Knight's Tours Equals 33,439,123,484,294 — Counting with Binary Decision Diagrams,1995

[52] Lorenz Halbeisen, Norbert Hungerbühler. Dual form of combinatorial problems and Laplace techniques.

[53] Dominique Roelants Van Baronaigien, Frank Ruskey. Efficient Generation of Subsets with a Given Sum.

[54] Aviezri S. Fraenkel. Error Correcting Codes and Combinatorial Games.

[55] Antoine Zoghbi, Ivan Stojmenović. Fast Algorithms for Generating Integer Partitions, 1994

图论

[56] Alon Efrat, Alon Itai, Matthew J Katz. Improvements on Bottleneck Matching and Related Problems Using Geometry.

[57] Gerard J Chang, Pei-Hsin Ho. The β-Assignment Problem in General Graphs, 1994

[58] Rainer E Burkard, Vladimir G Deĭneko, René van Dal, Jack AA van der Veen, Gerhard J. Woeginger. Well-Solvable Special Cases of the TSP: A Survey, 1995

[59] Rainer E Burkard. Selected Topics on Assignment Problems, 1999

[60] Ravindra K Ahuja. Solving The Convex Cost Integer Dual Network Flow Problem.

[61] Ravindra K. Ahuja. Combinatorial Algorithms for Inverse Network Flow Problems.

[62] Ching-Chih Han. A Fast Algorithm for the Minimax Flow Problem with 0/1 Weights.

[63] Therese C. Biedl, Broňa Brejová, Tomáš Vinař. Simplifying Flow Networks, 2000

[64] Kevin Daniel Wayne. Generalized Maximum Flow Algorithms, 1999

[65] Andrew V Goldberg. Recent Developments in Maximum Flow Algorithms, 1998

[66] Boris V Cherkassky, Andrew V Goldberg, Paul Martin, João C Setubal, Jorge Stolfi. Augment or Push? A computational study of Bipartite Matching and Unit Capacity Flow Algorithms, 1998

[67] Chandra S Chekuri, Andrew V Goldberg, David R Karger, Matthew S Levine, Experimental Study of Minimum Cut Algorithms, 1996

[68] Fleischer L, Tardos E. Efficient Continuous-Time Dynamic Network Flow Algorithms, 1998

[69] Fleischer L. Universally maximum flow with piecewise-constant capacities, 1998

[70] Brander A W, Sinclair M C, A Comparative Study of k-Shortest Path Algorithms

[71] Ernesto De Queirós Vieira Martins, Marta Margarida Braz Pascoal, José Luis Esteves Dos Santos. Labeling Algorithms for Ranking Shortest Paths. 1999

[72] Ernesto De Queirós Vierira Martins, Marta Margarida Braz Pascoal, José Luis Esteves Dos Santos. Deviation Algorithms for Ranking Shortest Paths, 1999

[73] Ernesto De Queirós Vierira Martins, Marta Margarida Braz Pascoal, José Luis Esteves Dos Santos. An Algorithm for Ranking Loopless paths, 1999

[74] Alon Efrat, Matthew J Katz Computing Fair and Bottleneck Matchings in Geometric Graphs, 1996

[75] Alon Efrat, Alon Itai, Matthew J Katz. Geometry Helps in Bottleneck Matching and Related Problems, 1999

[76] Tomomi Matsui, Yasufumi Saruwatari, Maiko Shigeno. An analysis of Dinkelbach's Algorithm for 0-1 Fractional Programming Problems, 1992

[77] Mohit Tawarmalani, Shabbir Ahmed, Nikolaos V Sahinidis. Reformulations of Rational Functions of 0-1 Variables, 2001

[78] Seth Pettie, Vijaya Ramachandran. An Optimal Minimum Spanning Tree Algorithm, 1999

数论

[79] Brian Dunten, Julie Jones, Jonathan Sorenson. A Space-Efficient Fast Prime Number Sieve, 1994

[80] Daniel J Bernstein. Enumerating Solutions to $p(a)+q(b)=r(c)+s(d)$, 1991

[81] Odlyzko A M, Asymptotic Enumeration Methods, 1996

[82] Jonathan P Sorenson. Trading Time for Space in Prime Number Sieves.

[83] Theodoulos Garefalakis. Primality Testing, Integer Factorization, and Discrete Logarithms, 1998

几何

[84] Aho A, Hopcoft J, Ullman J. The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974

[85] Bourke P. Polygon Area and Centroid Calculation. http://astronomy.swin.edu.au/~pbourke/geometry/polyarea/

[86] O'Rourke, J. Computational Geometry in C, 2nd edi. Cambridge University Press, 1998.

[87] Graham R.: An efficient algorithm for determining the convex hull of a finite planar set, Inform. Process. Lett. 1972, 1: 132~133

[88] Melkman A. On-line Construction of the Convex Hull of a Simple Polygon, Information Processing Letters 25, p.11, 1987

[89] Toussaint G. Solving geometric problems with the "rotating calipers". Athens, Greece: in Proc IEEE MELECON '83, pp. A 10.02/ 1-4, 1983

[90] http://cgm.cs.mcgill.ca/~athens/cs601/

[91] http://cgm.cs.mcgill.ca/~beezer/cs507/main.html

[92] 周培德. 计算几何——算法分析与设计. 北京：清华大学出版社, 1999

[93] Cormen T, Leiserson C, Rivest R, Stein C. Introduction to Algorithms. 2nd edi. MIT Press, 2001

[94] de Berg M, van Kreveld M, Overmars M, Schwarzkopf O. Computational Geometry : Algorithms and Applications. 2nd edi. Springer, 2000