ALGO

算法竞赛进阶指南

李煜东 著

ADVANCED GUIDE TO ALGORITHM CONTEST

中原出版传媒集团
中原传媒股份公司
北京出版社

ALGO 算法竞赛进阶指南

李煜东 著

中原出版传媒集团
中原传媒股份公司

大象出版社
·郑州·

图书在版编目(CIP)数据

算法竞赛进阶指南 / 李煜东著. — 郑州 : 大象出版社, 2022. 11
ISBN 978-7-5711-1651-4

Ⅰ. ①算… Ⅱ. ①李… Ⅲ. ①计算机算法 Ⅳ. ①TP301. 6

中国版本图书馆 CIP 数据核字(2022)第 218201 号

算法竞赛进阶指南

SUANFA JINGSAI JINJIE ZHINAN

李煜东 著

出 版 人 汪林中
策 划 郭孟春
责任编辑 庞 博
责任校对 安德华 万冬辉 张绍纳
特邀设计 刘 民
美术编辑 王晶晶

出版发行 大象出版社(郑州市郑东新区祥盛街 27 号 邮政编码 450016)
发行科 0371-63863551 总编室 0371-65597936
网 址 www. daxiang. cn
印 刷 新乡市龙泉印务有限公司
经 销 各地新华书店经销
开 本 787 mm×1092 mm 1/16
印 张 34
字 数 763 千字
版 次 2022 年 11 月第 1 版 2022 年 11 月第 1 次印刷
定 价 86. 00 元
若发现印、装质量问题,影响阅读,请与承印厂联系调换。
印厂地址 河南省新乡经济开发区中央大道中段
邮政编码 453731 电话 0373-5590988

0xFF 前言
——如何阅读本书

探索一门学问有三个层次：求其解，知其原因，究其思维之本。通俗地讲，就是“怎么做”“为什么是对的”“怎样才能想到这么去做”。在计算机科学中，前两者分别对应算法的步骤和证明。而长久以来，后者时常与“天赋”一词相关联。本书在全面讲解算法与数据结构知识点的同时，致力于模型构建与思路分析，帮助读者厘清思维过程的线索和脉络，建立一棵“技能树”，从容地面对算法竞赛的各项挑战，为未来更高层次的程序设计与研究奠定坚实的基础。

撰写背景

我当年准备参加信息学竞赛时，所在地区的教学资源相对不足，在大约一年半的时间里，依靠 3000 道以上的巨大刷题量，才从一名连深搜都写不对的初学者，成长为 NOI 金牌得主，并入选国家集训队。2012 年以来的每个寒、暑假，我都参与了相关竞赛的授课、命题等工作。为了把知识清晰、透彻地讲解给学生，我不得不转变自己的角色，从“会做题”的选手，变成一位“知其所以然”的教师。我发现在高中阶段的竞赛学习中，因自己死记硬背、不求甚解，导致对许多知识的理解都存在偏差。每次重新审视一个解法、更新一次课件，我都对“分析证明”的重要性和“怎样想到一种解法”的思路历程有了更加深刻的认识。

在思维的迷宫里，有的人凭天生的灵感直奔终点；有的人以持久的勤勉，铸造出适合

自己的罗盘；有的人迷失了方向，宣告失败。作为算法竞赛的过来人，我很欣喜地看到大多数选手无论结果如何，都保持着对程序设计的热情和喜爱，在日后的学习、工作中继续选择了信息科学方向。诚然，悟“道”还需个人，但准备算法竞赛的时间毕竟非常有限。我想在彻底从算法竞赛界退役之前，有必要尽一点儿微薄之力，把自己数年来的所寻所得汇集成卷，只求读者能从中受益些许。我深知作为一名指导者，想要代替时间与见识的积淀，给学习者传播“怎样才能想到这么去做”，是一件难上加难的事情。因此，希望读者始终保持独立的思考习惯和坚定的批判精神，对于每种模型，不仅能用数学语言推导其正确性，更能自然而然地“说服”自己，使整个程序设计过程完全融于自己的思维体系之中。

前期准备

《算法竞赛进阶指南》，顾名思义，是一本侧重进阶类的书籍。认真阅读本书的每一个知识点，动手完成每一道例题和习题，并加以归纳、理解，读者将会从一名算法竞赛的入门级选手，达到NOIP一等奖获得者的中、高级水平，并具有相当的自我学习与思考能力。在此基础上，读者只需涉猎一些新的领域，扩大知识范围，辅以实战训练，不难继续提升到 NOI 省队平均水平，或具有组队夺得 ACM-ICPC 区域赛金牌的能力。

在阅读本书前，读者应该先对算法竞赛的形式有所了解，掌握至少一门编程语言，能够运用朴素算法求解最基本的问题。最好是大致浏览过一本入门类教材，有 100 道题左右的练习经验，或者三个月到半年的课余学习经历。读者可以对照以下清单进行自我检验。

√ 熟悉 C++ 语言的基本语法结构，能够使用 C++ 语言编写程序。

√ 会使用朴素算法，如枚举、模拟等方法。

√ 理解“递推”的概念，能循环计算简单的递推式。

√ 掌握几种常见的排序算法，如选择排序、冒泡排序、快速排序等。

√ 熟练使用数组，知道链表、栈、队列的基本结构与用法。

√ 能够借助一些 C/C++ 内置函数，进行简单的字符串处理。

√ 了解树、图结构，以及其中的常见术语与专有名词。

√ 接触过动态规划，会做数字三角形、LIS、LCS 等入门题。

√ 具有不低于高中一年级水平的基本数学素养。

即使在上述条件中有所欠缺，也不代表一定不能读懂本书。读者可以在遇到不熟悉的内容时，随时使用 Google、Bing（国际版）等搜索引擎查阅资料，辅助学习。

阅读建议

本书由“基本算法”“基本数据结构”“搜索”“数学知识”“数据结构进阶”“动态规划”“图论”“综合技巧与实践”八章组成，每章对应着算法与数据结构中一个较为独立的领域，并进一步划分为若干节，每节都是所在领域知识体系中较为完整的一块，各章节以十六进制整数编码。整本书尽量以循序渐进的方式编写，对于不得不涉及的后续章节的内容均有所标注，读者可以从前到后依次学习。

除此之外，读者也可以按照知识点从易到难的顺序，先阅读第 0x00、0x10 和 0x70 章，再阅读第 0x21、0x22、0x25、0x31~0x33、0x36、0x41、0x51~0x53、0x61、0x62 节，接下来阅读第 0x20 章的剩余部分和第 0x42~0x46 节，最后阅读第 0x30、0x40、0x50 和 0x60 章的剩余部分。

1. 知识点讲解与例题

每节内容的主干由知识点讲解部分以及与之紧密相关的例题组成，读者应当配套进行

学习。不过，还是鼓励读者先花费适当的时间独立思考，再查看例题的详细解答。算法竞赛最终要落实在程序设计上，请读者务必动手完成每一道例题的代码实现，方能达到期望的训练效果。

2. 知识点清单与拓展

每章的最后一节都包含一份详细的知识点清单，供读者在复习时对照、检验。这些清单基本涵盖了信息学竞赛省级联赛的知识范围，尤其在动态规划、图论方面已经具有了相当的深度。在确保掌握了清单中列举的所有内容后，读者应该自主完成章末的练习题。如果遇到瓶颈，可以谨慎利用搜索引擎查阅题解。

认真阅读完全书后，若读者仍想进一步提高，可以重点关注高级数据结构、网络流、计算几何这三个本书涉及不多的领域，并进一步提升数学水平。

3. 在线资源

本书全部例题和习题都标注了来源，可以直接在 Online Judge 上提交。Acwing、NowCoder 均收录了本书的全部题目。测试数据和参考程序可以在 GitHub 的 Tedukuri 项目中下载。

同时，部分章节还配有作者亲自录制的讲解视频，读者扫描二维码即可观看。

关于在线资源的详细访问方式，请参阅 0x7F 节“附录”。

4. 关于英文

许多题目的原始题目描述为英文。对于例题，本书提供了题目大意的翻译；对于习题，仅在在线资源提供原版题面。随着在计算机科学与技术领域学习的不断深入，读者在查阅文献、进行学术交流时，遇到英文的概率将会越来越大。希望读者借此机会，提前感受英文环境，掌握一些算法与数据结构方面的英文术语，切勿对英文题目抱有逃避心态。

5. 更新与反馈

GitHub 的 Tedukuri 项目作为《算法竞赛进阶指南》社区，将随时发布有关本书的任何最新消息、内容更正等，同时也欢迎读者参与共建。读者可以通过在 GitHub 上建立 Issue 的方式，指出本书存在的问题。读者也可以 Fork 本项目，上传自己的修改，然后通过提交 Pull Request 的方式，请求合并到原项目中。

本书还有读者 QQ 群：650836280。读者可以与其他学习者进行即时交流，更快地找到问题的答案，了解更广泛的思路，追逐竞赛前沿热点。

在阅读过程中，希望读者时刻把自己视作一位科学研究者，切忌自欺欺人、口头浮夸、懒于思考、直接复制；也希望读者不仅关注结论、模型、算法本身，更要关注它们的分析证明与内在逻辑，建立自然、完整且不容置疑的思维体系。

致谢

本书自 2016 年 12 月动笔，2017 年 10 月完稿。全书约 70 万字。初稿总编辑时长达 60000 分钟，编写期间得到了许多老师、同学和朋友的支持与帮助。

本书第 0x00 章“基本算法”、第 0x10 章“基本数据结构”、第 0x30 章“数学知识”由北京大学石昊悦同学审阅，第 0x20 章“搜索”、第 0x40 章“数据结构进阶”、第 0x60 章“图论”、第 0x70 章“综合技巧与实践”由哈尔滨工业大学苏佳轩同学审阅，第 0x50 章“动态规划”由石家庄市第二中学李晶老师审阅。三位审稿者认真、负责，以严谨的科学态度尽力保证本书所讲内容的正确性，读者所看到的每一个文字都饱含着他们付出的努力。

在编写和审阅期间，北京大学李源昊同学，石家庄市第二中学郭资政、杨卓毅、杨思祺同学以及复旦大学刘炎明同学提前阅读了本书部分内容，多次提出宝贵的修改意见。另外，本书的多道例题、习题出自北京旷视科技有限公司杜宇飞先生、清华大学毕克同学之手。他们是本书的直接贡献者。

在第四次修订前，徐皓天同学（xht37）重新编写了本书全部例题、习题的程序，致力于使标程风格统一、与书中所述解法一致，为配套资源的质量提升做出了重要贡献。

本书在大象出版社再版之际，我仍然要感谢米军阳老师、郭孟春老师及出版社多名编辑的辛劳，他们自始至终推动着本书的编写进展，尤其为适应现在的学习方式，还策划在线视频课程等内容。大视野在线测评为本书开辟讨论专帖，为本书的后续服务做了许多工作。

最后，我想感谢我的父母，感谢我信息学竞赛时期的教练潘雪峰老师、ACM-ICPC竞赛时期的教练罗国杰老师，没有他们的教导，就不可能有今日本书的面世。

算法竞赛博大精深，而作者时间、能力有限，只做了一点儿微小的工作，书中难免有遗漏或不足之处，还望读者海涵并不吝赐教。期待本书再版之时，能够更臻完善。

作者　李煜东

2022 年 5 月 8 日于北京

目 录

0x00 基本算法 1
0x01 位运算 2
0x02 递推与递归 12
0x03 前缀和与差分 23
0x04 二分 27
0x05 排序 35
0x06 倍增 43
0x07 贪心 46
0x08 总结与练习 51

0x10 基本数据结构 55
0x11 栈 56
0x12 队列 62
0x13 链表与邻接表 67
0x14 Hash 72
0x15 字符串 79

0x16 Trie 87
0x17 二叉堆 91
0x18 总结与练习 100

0x20 搜索 103
0x21 树与图的遍历 105
0x22 深度优先搜索 112
0x23 剪枝 116
0x24 迭代加深 122
0x25 广度优先搜索 125
0x26 广搜变形 133
0x27 A* 138
0x28 IDA* 143
0x29 总结与练习 146

0x30 数学知识 149
0x31 质数 150
0x32 约数 155
0x33 同余 165
0x34 矩阵乘法 173
0x35 高斯消元与线性空间 177
0x36 组合计数 187
0x37 容斥原理与 Möbius 函数 193
0x38 概率与数学期望 198
0x39 0/1 分数规划 204
0x3A 博弈论之 SG 函数 206
0x3B 总结与练习 210

0x40 数据结构进阶 213
0x41 并查集 214
0x42 树状数组 224
0x43 线段树 233
0x44 分块 246
0x45 点分治 253
0x46 二叉查找树与平衡树初步 256
0x47 离线分治算法 267
0x48 可持久化数据结构 276
0x49 总结与练习 284

0x50 动态规划 287
0x51 线性 DP 290
0x52 背包 301
0x53 区间 DP 310
0x54 树形 DP 316
0x55 环形与后效性处理 323
0x56 状态压缩 DP 328
0x57 倍增优化 DP 336
0x58 数据结构优化 DP 341
0x59 单调队列优化 DP 344
0x5A 斜率优化 352
0x5B 四边形不等式 360
0x5C 计数类 DP 366
0x5D 数位统计 DP 374
0x5E 总结与练习 377

0x60 图论 381

0x61 最短路 382

0x62 最小生成树 396

0x63 树的直径与最近公共祖先 403

0x64 基环树 422

0x65 负环与差分约束 426

0x66 Tarjan 算法与无向图连通性 430

0x67 Tarjan 算法与有向图连通性 448

0x68 二分图的匹配 460

0x69 二分图的覆盖与独立集 472

0x6A 网络流初步 480

0x6B 总结与练习 493

0x70 综合技巧与实践 497

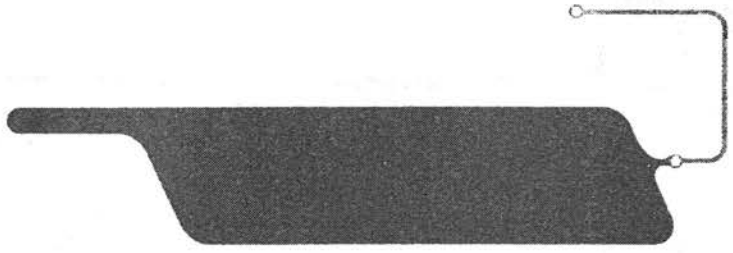
0x71 C++ STL 498

0x72 随机数据生成与对拍 510

0x7F 附 录 516

0x00 基本算法

在本章中，我们将会学习位运算、递推、递归、二分、排序、倍增、贪心等算法，以及前缀和、差分、离散化等技巧。它们简洁易懂，却又蕴含着巧妙而深刻的思想，作为构建算法与数据结构体系的基本单位，贯穿于整个程序设计竞赛的始末。永远不要小瞧这些基本思想，因为认真学习它们对于理解更加高深的算法有着极大的帮助，甚至还有可能帮你开启发现并创造新算法的门窗。



0x01 位运算

bit 是度量信息的单位，包含 0 和 1 两种状态。计算机的各种运算最后无不归结为一个个 bit 的变化。熟练掌握并利用位运算，能够帮助我们理解程序运行中的种种表现，提高程序运行的时空效率，降低编程复杂度。

读者可能已经发现，本书的章节目录是以 0x00~0xFF 这些由数字 0~9 与字母 A~F 表示的 2 位十六进制整数进行编号的，其中“0x”表示十六进制。正文即本章由 0x00 开始，前言分配序号 0xFF，后记分配序号 0x7F。这就是以最高二进制位为正负符号位的“补码”形式表示的 8 位二进制数。在 C++中，8 位二进制数对应 char 类型，范围为−128~127，其中 0xFF 代表−1，0x7F 代表最大值 127。

在阅读本节之前，读者应该对以下算术位运算有一个初步的认识[①]：

与	或	非	异或
and, &	or, |	not, ~	xor

它们不局限于逻辑运算，均可作用于二进制整数。为了避免混淆，本书统一用单词 xor **表示异或运算**，而用符号“^”**表示乘方运算**（虽然该符号在 C++中表示异或）。

另外，在 m 位二进制数中，为方便起见，通常称**最低位为第 0 位，从右到左依此类推，最高位为第 $m-1$ 位**。本书默认使用这种表示方法来指明二进制数以及整数在二进制表示下的位数。

下面我们以 32 位二进制数，即 C++的 int 与 unsigned int[②] 类型为例详细介绍计算机中的整数存储与运算。

补码

32 位无符号整数 unsigned int：

直接把这 32 位编码 C 看作 32 位二进制数 N。

① 有关“与”“或”“非”“异或”等算术位运算的基础知识，可参见配套资源提供的参考资料。

② 在信息学竞赛中，我们一般认为 int 即为 32 位有符号整数，unsigned int 即为 32 位无符号整数。在大多数现代计算机上也确实如此。不过，在 C++标准中，仅规定了 int 和 unsigned int 的位数不小于 16。请读者注意，直接把 int 和 unsigned int 与 32 位整数画等号实际上是不严谨的。



32 位有符号整数 int：

以最高位为符号位，0 表示非负数，1 表示负数。

对于最高位为 0 的每种编码 C，直接看作 32 位二进制数 S。

同时，定义该编码按位取反后得到的编码 $\sim C$ 表示的数值为 $-1-S$。

32 位补码表示	unsigned int	int
000000…000000	0	0
011111…111111	2 147 483 647	2 147 483 647
100000…000000	2 147 483 648	−2 147 483 648
111111…111111	4 294 967 295	−1

可以发现，在补码下每个数值都有唯一的表示方式，并且任意两个数值做加减法运算，都等价于在 32 位补码下做**最高位不进位**的二进制加减法运算。发生算术溢出时，32 位无符号整数相当于**自动对 2^{32} 取模**。这也解释了“有符号整数”算术上溢时出现负数的现象。

补码也被称为“二补数”。还有一种编码称为反码，也叫“一补数”，直接把 C 的每一位取反表示负 C。补码与反码在负数表示中，绝对值相差 1。例如，在上表中，第 1、4 行是一对反码，第 2、3 行是一对反码。作为整数编码、存储和运算的方式，补码与反码相比有许多优势。除了上面提到的“自然溢出取模”之外，补码重点解决了 0 的编码唯一性问题，能比反码多表示一个数，同时减少特殊判断，在电路设计中极其简单、高效。

形式	加数	加数	和
32 位补码	111…111	000…001	~~(1)~~000…000
int	−1	1	0
unsigned int	4 294 967 295	1	0 (mod 2^{32})
32 位补码	011…111	000…001	100…000
unsigned int	2 147 483 647	1	2 147 483 648
int	2 147 483 647	1	−2 147 483 648

因为用二进制表示一个 int 需要写出 32 位，比较繁琐，而用十进制表示，又不容易明显地体现出补码的每一位，所以在程序设计中，常用十六进制来表示一个常数，这样只需要书写 8 个字符，每个字符（0~9，A~F）代表补码下的 4 个二进制位。C++的十六进制常数以“0x”开头，“0x”本身只是声明了进制，“0x”后面的部分对应具体的十六进制数值。例如：

32 位补码	int（十进制）	int（十六进制）
000000…000000	0	0x0
011111…111111	2 147 483 647	0x7F FF FF FF
00111111 重复 4 次	1 061 109 567	0x3F 3F 3F 3F
111111…111111	−1	0xFF FF FF FF

上表中的 0x3F 3F 3F 3F 是一个很有用的数值，它是满足以下两个条件的最大整数。

1. 整数的两倍不超过 0x7F FF FF FF，即 int 能表示的最大正整数。

2. 整数的每 8 位（每个字节）都是相同的。

我们在程序设计中经常需要使用 `memset(a, val, sizeof(a))` 初始化一个 int 数组 a，该语句把数值 val（0x00~0xFF）填充到数组 a 的每个字节上，而 1 个 int 占用 4 个字节，所以用 memset 只能赋值出“每 8 位都相同”的 int。

综上所述，0x7F 7F 7F 7F 是用 memset 语句能初始化出的最大数值。不过，当需要把一个数组中的数值初始化成正无穷时，为了避免加法算术上溢或者繁琐的判断，我们经常用 `memset(a, 0x3f, sizeof(a))` 给数组赋 0x3F 3F 3F 3F 的值来代替。该语句在后续章节的参考程序中将会多次出现。

移位运算

左移

在二进制表示下把数字同时向左移动，低位以 0 填充，高位越界后舍弃。

$$1 \ll n = 2^n, \qquad n \ll 1 = 2n$$

算术右移

在二进制补码表示下把数字同时向右移动，高位以符号位填充，低位越界后舍弃。

$$n \gg 1 = \left\lfloor \frac{n}{2.0} \right\rfloor$$

算术右移等于除以 2 向下取整，$(-3) \gg 1 = -2$，$3 \gg 1 = 1$。

值得一提的是，“整数/2”在 C++中实现为“除以 2 向零取整”，$(-3)/2 = -1$，$3/2 = 1$。请读者自己尝试使用 C++编译器编译运行类似的语句，检查运算结果。

逻辑右移

在二进制补码表示下把数字同时向右移动，高位以 0 填充，低位越界后舍弃。

C++语法没有规定右移的实现方式，使用算术右移还是逻辑右移由编译器决定。一般的编译器（较新版本的 GNU C++与 Visual Studio C++）均使用算术右移。除非特殊提示，我们默认右移操作采用算术右移方式实现。

例题 a^b AcWing89

求 a 的 b 次方对 p 取模的值，其中 $1 \le a,b,p \le 10^9$。

相关题目：POJ1995 Raising Modulo Numbers

根据数学常识，每一个正整数可以唯一表示为若干指数不重复的 2 的次幂的和。也就是说，如果 b 在二进制表示下有 k 位，其中第 $i(0 \le i < k)$ 位的数字是 c_i，那么：

$$b = c_{k-1}2^{k-1} + c_{k-2}2^{k-2} + \cdots + c_0 2^0$$

于是：

$$a^b = a^{c_{k-1}*2^{k-1}} * a^{c_{k-2}*2^{k-2}} * \cdots * a^{c_0*2^0}$$

因为 $k = \lceil \log_2(b+1) \rceil$（其中⌈⌉表示上取整），所以上式乘积项的数量不多于 $\lceil \log_2(b+1) \rceil$ 个。又因为：

$$a^{2^i} = \left(a^{2^{i-1}}\right)^2$$

所以我们很容易通过 k 次递推求出每个乘积项，当 $c_i = 1$ 时，把该乘积项累积到答案中。b & 1 运算可以取出 b 在二进制表示下的最低位，而 b >> 1 运算可以舍去最低位，在递推的过程中将二者结合，就可以遍历 b 在二进制表示下的所有数位 c_i。整个算法的时间复杂度[①]为 $O(\log_2 b)$。

```
int power(int a, int b, int p) { // calculate (a ^ b) mod p
    int ans = 1 % p;
    for (; b; b >>= 1) {
        if (b & 1) ans = (long long)ans * a % p;
        a = (long long)a * a % p;
    }
    return ans;
}
```

在上面的代码片段中，我们通过“右移（>>）”“与（&）”运算的结合，遍历了 b 的二进制表示下的每一位。在循环到第 i 次时（从 0 开始计数），变量 a 中存储的是 a^{2^i}，若 b 该位为 1，则把此时的变量 a 累积到答案 ans 中。

值得提醒的是，在 C++语言中，两个数值执行算术运算时，以**参与运算**的最高数值类型为基准，与保存结果的变量类型无关。换言之，虽然两个 32 位整数的乘积可能超过 int 类型的表示范围，但是 CPU 只会用 1 个 32 位寄存器保存结果，造成我们常说的越界现象。因此，我们必须把**其中一个数强制转换成 64 位整数类型** long long 参与运

① 关于“时间复杂度”的定义与相关知识，可参见配套资源提供的参考资料。

算，从而得到正确的结果。最终对 p 取模以后，执行赋值操作时，该结果会被隐式转换成 int 存回 ans 中。

于是一个问题就出现了。因为 C++ 内置的最高整数类型是 64 位，若运算 $a*b \bmod p$ 中的三个变量 a,b,p 都在 10^{18} 级别，则不存在一个可供强制转换的 128 位整数类型，我们需要一些特殊的处理办法。

例题 **64 位整数乘法** AcWing90

求 a 乘 b 对 p 取模的值，其中 $1 \le a,b,p \le 10^{18}$。

方法一

类似于快速幂的思想，把整数 b 用二进制表示，即 $b = c_{k-1}2^{k-1} + c_{k-2}2^{k-2} + \cdots + c_0 2^0$，那么 $a*b = c_{k-1}*a*2^{k-1} + c_{k-2}*a*2^{k-2} + \cdots + c_0 * a * 2^0$。

因为 $a*2^i = (a*2^{i-1})*2$，若已求出 $a*2^{i-1} \bmod p$，则计算 $(a*2^{i-1})*2 \bmod p$ 时，运算过程中每一步的结果都不超过 $2*10^{18}$，仍然在 64 位整数 long long 的表示范围内，所以很容易通过 k 次递推求出每个乘积项。当 $c_i = 1$ 时，把该乘积项累加到答案中即可。时间复杂度为 $O(\log_2 b)$。

```
long long mul(long long a, long long b, long long p) {
    long long ans = 0;
    for (; b; b >>= 1) {
        if (b & 1) ans = (ans + a) % p;
        a = a * 2 % p;
    }
    return ans;
}
```

方法二

利用 $a*b \bmod p = a*b - \lfloor a*b/p \rfloor * p$，其中 $\lfloor\ \rfloor$ 表示下取整。记 $c = \lfloor a*b/p \rfloor$。

如何计算 c 呢？试想我们用浮点数直接计算 $a*b/p$，当浮点数的精度不足以保存精确数值时，它会像科学计数法一样舍弃低位，所以得到的结果是 $c.___$，其中小数点之后的部分是不准确的。long double 在十进制下的有效数字有 18~19 位，当 $a,b < p$ 时，c 一定也小于 p，即 c 在 18 位以内，故 long double 足够胜任保存整数部分的精确值 c。再把结果强制转化为 unsigned long long 类型，即可得到整数 c。值得一提的是，当 $a*b$ 恰好能被 p 整除时，由于精度误差，算出的 c 可能比实际小 1，但这在取模意义下并不影响结果的正确性。

下面我们只需要再算出 $a*b - c*p$ 就可以了。因为 $a*b - c*p$ 实际上是 $a*b \bmod p$，所以 $a*b - c*p \le p < 2^{64}$，进而 $a*b - c*p = (a*b - c*p) \bmod 2^{64}$。



因为 unsigned long long 自然溢出等价于对 2^{64} 取模，所以我们用 unsigned long long 计算 $x = a * b$ 和 $y = c * p$，再用 long long 计算 $(x \bmod p - y \bmod p) \bmod p$，即可得到最终的结果。时间复杂度为 O(1)。

```
typedef unsigned long long ull;
ull mul(ull a, ull b, ull p) {
    a %= p, b %= p;  // 当 a,b 一定在 0~p 之间时，此行不必要
    ull c = (long double)a * b / p;
    ull x = a * b, y = c * p;
    long long ans = (long long)(x % p) - (long long)(y % p);
    if (ans < 0) ans += p;
    return ans;
}
```

二进制状态压缩

二进制状态压缩，是指将一个长度为 m 的 bool 数组用一个 m 位二进制整数表示并存储的方法。利用下列位运算操作可以实现原 bool 数组中对应下标元素的存取。

操作	运算
取出整数 n 在二进制表示下的第 k 位	$(n \gg k) \& 1$
取出整数 n 在二进制表示下的第 $0 \sim k-1$ 位（后 k 位）	$n \& ((1 \ll k) - 1)$
把整数 n 在二进制表示下的第 k 位取反	$n \text{ xor } (1 \ll k)$
对整数 n 在二进制表示下的第 k 位赋值 1	$n \mid (1 \ll k)$
对整数 n 在二进制表示下的第 k 位赋值 0	$n \& (\sim(1 \ll k))$

这种方法运算简便，并且节省了程序运行的时间和空间。当 m 不太大时，可以直接使用一个整数类型存储。当 m 较大时，可以使用若干个整数类型（int 数组），也可以直接利用 C++ STL 为我们提供的 bitset 实现（第 0x71 节）。

例题 起床困难综合症 AcWing998

一个 boss 的防御战线由 n 扇防御门组成，其中第 i 扇防御门的属性包括一个运算 op_i 和一个参数 t_i，运算一定是 OR、XOR、AND 中的一种，参数是非负整数。若在未通过这扇防御门时攻击力为 x，则通过这扇防御门后攻击力将变为 $x\ op_i\ t_i$。最终 boss 受到的伤害为玩家的初始攻击力 x_0 依次经过所有 n 扇防御门后得到的攻击力。由于水平有限，玩家的初始攻击力只能为 $[0,m]$ 之间的一个整数。玩家希望通过选择合适的初始攻击力，使他的攻击能造成最大的伤害，求这个伤害值。

数据范围：$n \le 10^5$，$m, t_i \le 10^9$。

本题是让我们选择 $[0,m]$ 之间的一个整数 x_0，经过给定的 n 次位运算，使结果 ans 最大。

位运算的主要特点之一是**在二进制表示下不进位**。正因如此，在 x_0 可以任意选择的情况下，参与位运算的各个位（bit）之间是独立无关的。换言之，对于任意的 k ($0 \le k < 30$)，“ans 的第 k 位是几”只与“x_0 的第 k 位是几”有关，与其他位无关。所以我们可以从高位到低位，依次考虑 x_0 的每一位填 0 还是填 1。

x_0 的第 k 位应该填 1，当且仅当同时满足下列两个条件。

1. 已经填好的更高位构成的数值加上 $1 \ll k$ 以后不超过 m。

2. 用每个参数的第 k 位参与位运算。若初值为 1，则 n 次位运算后结果为 1；若初值为 0，则 n 次位运算后结果为 0。

如果不满足上述条件，要么填 1 会超过 m 的范围，要么填 1 不如填 0 更优。这种情况下令 x_0 的第 k 位为 0 显然更好。确定 x_0 的每一位以后，自然可以得到 ans 的值。

```
pair<string, int> a[100005];
int calc(int bit, int now) { // 用参数的第 bit 位进行 n 次运算
    for (int i = 1; i <= n; i++) {
        int x = a[i].second >> bit & 1;
        if (a[i].first == "AND") now &= x;
        else if (a[i].first == "OR") now |= x;
        else now ^= x;
    }
    return now;
}
int main() {
    cin >> n >> m;
    for (int i = 1; i <= n; i++) {
        char str[5]; int x;
        scanf("%s%d", str, &x);
        a[i] = make_pair(str, x);
    }
    int val = 0, ans = 0;
    for (int bit = 29; bit >= 0; bit--) {
        int res0 = calc(bit, 0);
        int res1 = calc(bit, 1);
        if (val + (1 << bit) <= m && res0 < res1)
            val += 1 << bit, ans += res1 << bit;
        else
```

```
            ans += res0 << bit;
        }
        cout << ans << endl;
    }
```

提醒：一些运算符优先级从高到低的顺序如下表所示。最需要注意的地方是：大小关系比较的符号优先于“位与”“异或”“位或”运算。在程序实现时，如果不确定优先级，建议加括号保证运算顺序的正确性。

加减	移位	比较大小	位与	异或	位或
+, −	≪, ≫	>, <, ==, !=	&	xor (C++ ^)	|

成对变换

通过计算可以发现，对于非负整数 n：

当 n 为偶数时，n xor 1 等于 $n+1$；

当 n 为奇数时，n xor 1 等于 $n-1$。

因此，“0 与 1”“2 与 3”“4 与 5”…关于 xor 1 运算构成“成对变换”。

这一性质经常用于图论邻接表中边集的存储。在具有无向边（双向边）的图中把一对正反方向的边分别存储在邻接表数组的第 n 与 $n+1$ 位置（**其中 n 为偶数**），就可以通过 xor 1 的运算获得与当前边 (x,y) 反向的边 (y,x) 的存储位置。详细应用我们将在讲解邻接表（第 0x13 节）时给出。

lowbit 运算

lowbit(n) 定义为非负整数 n 在二进制表示下“最低位的 1 及其后边所有的 0”构成的数值。例如，$n=10$ 的二进制表示为 $(1010)_2$，则 lowbit$(n)=2=(10)_2$。下面我们来推导 lowbit(n) 的公式。

设 $n>0$，n 的第 k 位是 1，第 $0\sim k-1$ 位都是 0。

为了实现 lowbit 运算，先把 n 取反，此时第 k 位变为 0，第 $0\sim k-1$ 位都是 1。再令 $n=n+1$，此时因为进位，第 k 位变为 1，第 $0\sim k-1$ 位都是 0。

在上面的取反加 1 操作后，n 的第 $k+1$ 到最高位恰好与原来相反，所以 $n\,\&\,(\sim n+1)$ 仅有第 k 位为 1，其余位都是 0。而在补码表示下，$\sim n=-1-n$，因此：

$$\text{lowbit}(n)=n\,\&\,(\sim n+1)=n\,\&\,(-n)$$

lowbit 运算配合 Hash（第 0x14 节）可以找出**整数二进制表示下所有是 1 的位**，所花费的时间与 1 的个数同级。为了达到这一目的，我们只需不断把 n 赋值为 $n-$

lowbit(n)，直至 $n=0$。例如，$n=9=(1001)_2$，则 lowbit(9) = 1。把 n 赋值为 $9-$ lowbit(9) $=8=(1000)_2$，则 lowbit(8) = 8 = $(1000)_2$。此时 $8-$lowbit(8) $=0$，停止循环。在整个过程中我们减掉了 1 和 $8=(1000)_2$ 两个数，它们恰好是 n 每一位上的 1 后面补 0 得到的数值。取 1 和 8 的对数 $\log_2 1$ 和 $\log_2 8$，即可得知 n 的第 0 位和第 3 位是 1。因为 C++ math.h 库的 log 函数是以 e 为底的实数运算，并且复杂度常数较大，所以我们需要预处理一个数组，通过 Hash 的方法代替 log 运算。当 n 较小时，最简单的方法是直接建立一个数组 H，令 $H[2^k]=k$，程序如下。

```
const MAX_N = 1 << 20;
int H[MAX_N + 1];
for (int i = 0; i <= 20; i++) H[1 << i] = i;  // 预处理
while (cin >> n) {  // 对多次询问进行求解
    while (n > 0) {
        cout << H[n & -n] << ' ';
        n -= n & -n;
    }
    cout << endl;
}
```

稍微复杂但效率更高的方法是建立一个长度为 37 的数组 H，令 $H[2^k \bmod 37]=k$。这里利用了一个小的数学技巧：$\forall k \in [0,35]$，$2^k \bmod 37$ 互不相等，且恰好取遍整数 1~36。修改之后的程序为：

```
int H[37];
for (int i = 0; i < 36; i++) H[(1ll << i) % 37] = i;
while (cin >> n) {
    while (n > 0) {
        cout << H[(n & -n) % 37] << ' ';
        n -= n & -n;
    }
    cout << endl;
}
```

值得指出的是，lowbit 运算也是树状数组（第 0x42 节）中的一个基本运算。

GCC 编译器还提供了一些内置函数[①]，可以高效计算 lowbit 以及二进制数中 1 的个数。不过，这些函数并非 C 语言标准，有的函数更是与机器或编译器版本相关的。另外，部分竞赛禁止使用下画线开头的库函数，故这些内置函数尽量不要随便使用。

① 关于 GNU C 编译器的部分内置函数的详细手册，可参见配套资源提供的参考资料。



```
int __builtin_ctz(unsigned int x)
int __builtin_ctzll(unsigned long long x)
```

返回 x 的二进制表示下最低位的 1 后边有多少个 0。

```
int __builtin_popcount(unsigned int x)
int __builtin_popcountll(unsigned long long x)
```

返回 x 的二进制表示下有多少位为 1。

0x02 递推与递归

一个实际问题的各种可能情况构成的集合通常称为“状态空间”，而程序的运行则是对于状态空间的遍历，算法和数据结构则通过划分、归纳、提取、抽象来帮助提高程序遍历状态空间的效率。递推和递归就是程序遍历状态空间的两种基本方式。

递推与递归的宏观描述

对于一个待求解的问题[①]，当它局限在某处边界、某个小范围或者某种特殊情形下时，其答案往往是已知的。如果能够将该解答的应用场景扩大到原问题的状态空间，并且扩展过程的每个步骤具有相似性，就可以考虑使用递推或者递归求解。

以最简单的计算阶乘为例：求 $n!$，即 $1*2*3*...*n$，特别地，定义 $0!=1$。

该问题的“边界”是 $0!=1$。从 $0!$ 计算到 $n!$ 共有 n 步，其中第 i 步需要在当前结果的基础上乘上 i，即每一步具有相似性。在不考虑数据范围的情况下，我们很容易写出下面的递推代码。

```
factorial[0] = 1;
for (int i = 1; i <= n; i++)
    factorial[i] = factorial[i - 1] * i;
```

换一种思路，我们也可以定义函数 $factorial(n)$ 表示求出 $n!$ 的值，然后考虑如何实现该函数的计算。容易发现 $factorial(n)=factorial(n-1)*n$，进而写出下面的代码。

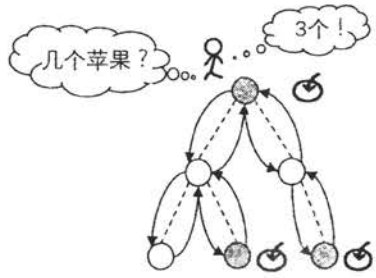
```
int factorial(int n) {
    if (n == 0) return 1;
    return factorial(n - 1) * n;
}
```

像第一份代码那样，以已知的“问题边界”为起点向“原问题”正向推导的扩展方式就是递推。像第二份代码那样，函数自身定义自身，通过函数体实现循环，以自相

① 这里的“问题”指一个抽象的数学概念，可理解为：给出一系列条件，求出满足要求的解。

似的方法重复进行的过程就被称为递归。

在这里可能有读者要问："为什么需要递归？求 $n!$ 用递推它不香吗？" 确实，因为 $n!$ 的推导路径我们已经知道了，所以直接 for 循环一遍就好了。然而在很多时候，推导的路线难以确定，这时就需要使用递归，以"原问题"为起点尝试寻找把状态空间缩小到已知的"问题边界"的路线，再通过该路线反向回溯。例如下图中的问题，一棵树，在根节点的时候，我们是不知道下边究竟长什么样的。



我们刚才也提到，使用递推或递归要求"原问题"与"问题边界"之间的每个变换步骤具有相似性，这样我们才能够设计一段程序实现这个步骤，将其重复作用于问题之中。换句话说，程序在每个步骤上应该面对**相同种类的问题**，这些问题都是**原问题的一个子问题**，可能仅在规模或者某些限制条件上有所区别，并且能够使用"求解原问题的程序"进行求解。

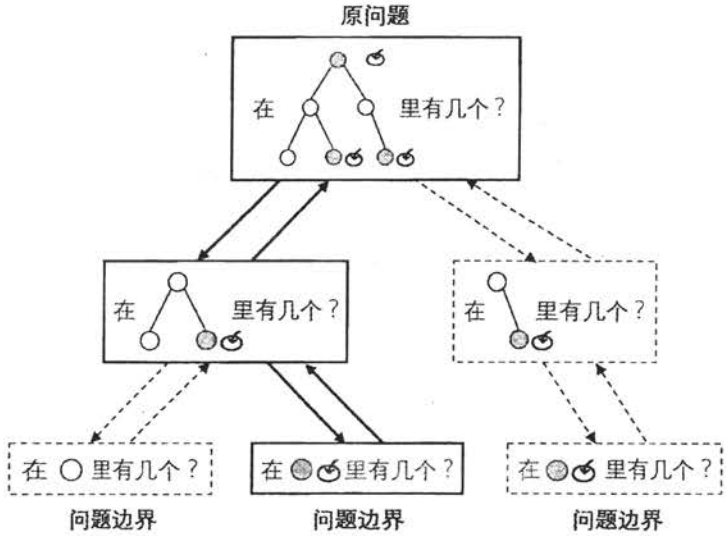
对于递归算法，有了上面这个前提，我们就可以让程序在每个变换步骤中执行三个操作。

1. 缩小问题状态空间的规模。这意味着程序尝试寻找在"原问题"与"问题边界"之间的变换路线，并向正在探索的路线上迈出一步。

2. 尝试求解规模缩小以后的问题，结果可能是成功，也可能是失败。

3. 如果成功，即找到了规模缩小后的问题的答案，那么将答案扩展到当前问题。如果失败，那么重新回到当前问题，程序可能会继续寻找当前问题的其他变换路线，直至最终确定当前问题无法求解。

上面这类程序就是“递归”的遍历方式，其整体流程如下图所示。



可以看到，递归程序的基本单元是由“缩小”“求解”“扩展”组成的一种变换步骤，只是在“求解”时因为问题的相似性，不断重复使用了这样一种变换步骤，直至在已知的问题边界上直接确定答案。对于其中任意一条从“原问题”到“问题边界”的变换路线（图中实线圈出的路径），横向来看，它的每一层是一次递归程序体的执行；纵向来看，它的向下、向上两组箭头分别是寻找路线和沿其推导的流程。

在这些操作过程中，还有一点颇为关键。注意每个问题可能有多个子问题，会向下递归多次，如果求解一个子问题失败，程序需要重新回到当前问题寻找其他的路线，因此把当前问题缩小为之前的子问题时所做的“对当前问题状态产生影响的事情”应该全部失效，这被称为“**还原现场**”。具体来说，就是在一次递归完成后，修改过的所有全局变量应该全部恢复为递归开始之前的状态。详细的方法我们会在下面的应用中讲解。

递推与递归的简单应用

在使用枚举算法蛮力探索问题的整个“状态空间”时，经常需要递归。按照规模大小，有如下几种常见的枚举形式和遍历方式：

枚举形式	状态空间规模	一般遍历方式
多项式	n^k，k 为常数	循环(for)、递推
指　数	k^n，k 为常数	递归、位运算
排　列	$n!$	递归、C++ next_permutation
组　合	C_n^m	递归+剪枝

“多项式”型的枚举在程序设计中随处可见。上一节中最短 Hamilton 路径问题的朴素做法，是一种“排列”型的枚举。本节下面的例题“费解的开关”中的枚举则是一种“指数”型的枚举。

例题 递归实现指数型枚举 AcWing92

从 $1\sim n$ 这 $n(n<20)$ 个整数中随机选取任意多个，输出所有可能的选择方案。

这等价于每个整数可以选可以不选，所有可能的方案总数共有 2^n 种。通过前两节的学习，我们已经知道可以进行一次循环，利用位运算来列举所有的选择方案。这一次我们使用递归来求解，在每次递归中分别尝试某个数“选”或“不选”两条分支，将尚未确定的整数数量减少 1，从而转化为一个规模更小的同类问题。

```
vector<int> chosen; // 被选择的数
void calc(int x) {
    if (x == n + 1) { // 问题边界
        for (int i = 0; i < chosen.size(); i++)
            printf("%d ", chosen[i]);
        puts("");
        return;
    }
    //“不选 x”分支
    calc(x + 1); // 求解子问题
    //“选 x”分支
    chosen.push_back(x); // 记录 x 已被选择
    calc(x + 1); // 求解子问题
    chosen.pop_back(); // 准备回溯到上一问题之前，还原现场
}

int main() {
    calc(1);  // 主函数中的调用入口
}
```

例题 递归实现组合型枚举 AcWing93

从 $1\sim n$ 这 n 个整数中随机选出 $m(0\le m\le n<20)$ 个，输出所有可能的选择方案。

我们只需要在上面指数型枚举的程序的 calc 函数开头添加以下这条语句即可。

```
if (chosen.size() > m || chosen.size() + (n - x + 1) < m) {
    return;
}
```

这就是所谓的“剪枝”。寻找变换路线其实就是“搜索”的过程，如果能够及时确定当前问题一定是无解的，就不需要到达问题边界才返回结果。在本题中，如果已经选择了超过 m 个数，或者即使再选上剩余所有的数也不够 m 个，就可以提前得知当前问题无解了。这条剪枝保证我们一旦进入无解的分支就会立刻返回，所以时间复杂度就从 2^n 降低为 C_n^m。关于搜索及其剪枝，我们将在 0x20 章用一章的篇幅详细讲解。

例题 递归实现排列型枚举 AcWing94

把 $1\sim n$ 这 $n(n<10)$ 个整数排成一行后随机打乱顺序，输出所有可能的次序。

该问题也被称为全排列问题，所有可能的方案总数有 $n!$ 种。在这里，递归需要求解的问题是“把指定的 n 个整数按照任意次序排列”，在每次递归中，我们尝试把每个可用的数作为数列中的下一个数，求解“把剩余的 $n-1$ 个整数按照任意次序排列”这个规模更小的子问题。

```
int order[20]; // 按顺序依次记录被选择的整数
bool chosen[20]; // 标记被选择的整数
void calc(int k) {
    if (k == n + 1) { // 问题边界
        for (int i = 1; i <= n; i++)
            printf("%d ", order[i]);
        puts("");
        return;
    }
    for (int i = 1; i <= n; i++) {
        if (chosen[i]) continue;
        order[k] = i;
        chosen[i] = 1;
        calc(k + 1);
        chosen[i] = 0;
        order[k] = 0; // 这一行可以省略
    }
}

int main() {
```

```
    calc(1);  // 主函数中的调用入口
}
```

例题 费解的开关 AcWing95

在一个 5*5 的 01 矩阵中，点击任意一个位置，该位置以及它上、下、左、右四个相邻的位置中的数字都会变化（0 变成 1，1 变成 0），问：最少需要多少次点击可以把一个给定的 01 矩阵变成全 0 矩阵？

在上述规则的 01 矩阵的点击游戏中，很容易发现三个性质。

1. 每个位置至多只会被点击一次。

2. 若固定了第一行（不能再改变第一行），则满足题意的点击方案至多只有 1 种。其原因是：当第 i 行某一位为 1 时，若前 i 行已被固定，只能点击第 $i+1$ 行该位置上的数字才能使第 i 行的这一位变成 0。从上到下按行使用归纳法可得上述结论。

3. 点击的先后顺序不影响最终结果。

于是，我们不妨先考虑第一行如何点击。在枚举第一行的点击方法（$2^5 = 32$ 种）后，就可以认为第一行“固定不动”，再考虑第 2~5 行如何点击。而按照上述性质 2，此时第 2~5 行的点击方案是确定的——从第一行开始递推，当第 i 行某一位为 1 时，点击第 $i+1$ 行该位置上的数字。若到达第 n 行时不全为 0，说明这种点击方式不合法。在所有合法的点击方式中取点击次数最少的就是答案。对第一行的 32 次枚举涵盖了该问题的整个状态空间，因此该做法是正确的。

对于第一行点击方法的枚举，可以采用位运算的方式，枚举 0~31 这 32 个 5 位二进制数，若二进制数的第 $k(0 \le k < 5)$ 位为 1，就点击 01 矩阵第 1 行第 $k+1$ 列的数字。

例题 Strange Towers of Hanoi AcWing96

解出 n 个盘子 4 座塔的 Hanoi（汉诺塔）问题最少需要多少步？关于 Hanoi 问题请参阅原题的题目描述。

首先考虑 n 个盘子 3 座塔的经典 Hanoi 问题，设 $d[n]$ 表示求解该 n 盘 3 塔问题的最少步数，显然有 $d[n] = 2 * d[n-1] + 1$，即把前 $n-1$ 个盘子从 A 柱移动到 B 柱，然后把第 n 个盘子从 A 柱移动到 C 柱，最后把前 $n-1$ 个盘子从 B 柱移动到 C 柱。

回到本题，设 $f[n]$ 表示求解 n 盘 4 塔问题的最少步数，则：

$$f[n] = \min_{1 \le i < n} \{2 * f[i] + d[n-i]\}$$

其中 $f[1] = 1$。上式的含义是，先把 i 个盘子在 4 塔模式下移动到 B 柱，然后把 $n-i$ 个盘子在 3 塔模式下移动到 D 柱，最后把 i 个盘子在 4 塔模式下移动到 D 柱。

考虑所有可能的 i 取最小值，就得到了上述递推公式。

在时间复杂度可以接受的前提下，上述做法可以推广到 n 盘 m 塔的计算。

分治

分治法是把一个问题划分为若干个规模更小的同类子问题，对这些子问题递归求解，然后在回溯时通过它们推导出原问题的解。

例题 Sumdiv AcWing97

求 A^B 的所有约数之和 mod 9901 $(0 \le A, B \le 5 * 10^7)$。

把 A 分解质因数，表示为 $p_1^{c_1} * p_2^{c_2} * \cdots * p_n^{c_n}$。那么 A^B 表示为 $p_1^{B*c_1} * p_2^{B*c_2} * \cdots * p_n^{B*c_n}$。$A^B$ 的所有约数表示为集合 $\{p_1^{k_1} * p_2^{k_2} * \cdots * p_n^{k_n}\}$，其中 $0 \le k_i \le B * c_i \, (1 \le i \le n)$。

根据乘法分配律，A^B 的所有约数之和就是：

$$\left(1 + p_1 + p_1^2 + \cdots + p_1^{B*c_1}\right) * \left(1 + p_2 + p_2^2 + \cdots + p_2^{B*c_2}\right) * \cdots$$

$$* \left(1 + p_n + p_n^2 + \cdots + p_n^{B*c_n}\right)$$

我们可以把该式展开，与约数集合比较。我们在 0x31 节和 0x32 节中将会详细介绍有关分解质因数和约数的内容。

上式中的每个括号内都是等比数列，如果使用等比数列求和公式，需要做除法。而答案还需要对 9901 取模，mod 运算只对加、减、乘有分配率，不能直接对分子、分母分别取模后再做除法。我们可以换一种思路，使用分治法进行等比数列求和。

问题：使用分治法求 $\text{sum}(p, c) = 1 + p + p^2 + \cdots + p^c = ?$

若 c 为奇数：

$$\begin{aligned}\text{sum}(p, c) &= \left(1 + p + \cdots + p^{\frac{c-1}{2}}\right) + \left(p^{\frac{c+1}{2}} + \cdots + p^c\right) \\ &= \left(1 + p + \cdots + p^{\frac{c-1}{2}}\right) + p^{\frac{c+1}{2}} * \left(1 + p + \cdots + p^{\frac{c-1}{2}}\right) \\ &= \left(1 + p^{\frac{c+1}{2}}\right) * \text{sum}(p, \frac{c-1}{2})\end{aligned}$$

若 c 为偶数，类似地：

$$\text{sum}(p, c) = \left(1 + p^{\frac{c}{2}}\right) * \text{sum}\left(p, \frac{c}{2} - 1\right) + p^c$$

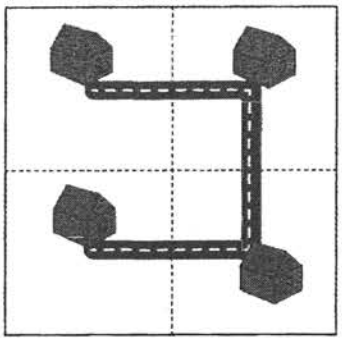
每次分治（递归之后），问题规模均会缩小一半，配合快速幂即可在 $O(\log c)$ 的时

间内求出等比数列的和。

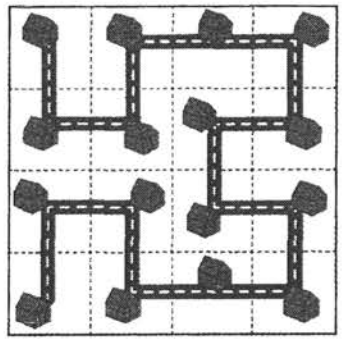
分形

例题 Fractal Streets AcWing98

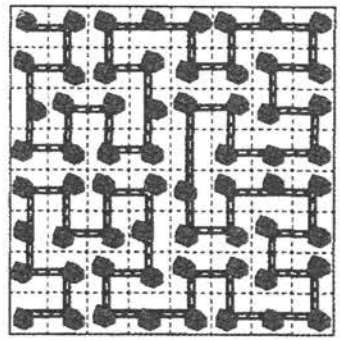
城市扩建的规划是个令人头疼的大问题。规划师设计了一个极其复杂的方案：当城市规模扩大之后，把与原来城市结构一样的区域复制或旋转 90 度之后建设在原来的城市周围（详细地说，将原来的城市复制一遍放在原城市的上方，将顺时针旋转 90 度后的城市放在原城市的左上方，将逆时针旋转 90 度后的城市放在原城市的左方），再用道路将四部分首尾连接起来，如下图所示。



等级 1



等级 2



等级 3

容易看出，扩建后的城市的各个房屋仍然由一条道路连接。定义 N 级城市为拥有 2^{2N} 座房屋的城市。对于任意等级的城市，从左上角开始沿着唯一的道路走，依次为房屋标号，就能够得到每间房屋的编号了。住在其中两间房屋里的两人想知道，如果城市发展到了一定等级，他俩各自所处的房屋之间的直线距离是多少。你可以认为图中的每个格子都是边长为 10 米的正方形，房屋均位于每个格子的中心点上（T 次询问，每次输入等级 N，两个编号 S、D，求 S 与 D 之间的直线距离，$T \le 10^4$，$N \le 31$）。

这就是著名的通过一定规律无限包含自身的“分形”图。为了计算方便，我们把题目中房屋的编号都减去 1，即从 0 开始编号，并把 S 和 D 也都减去 1。

本题关键是要解决：求编号为 M 的房屋（从 0 开始编号）在 N 级城市中的位置。把该问题记为 $\text{calc}(N,M)$，本题就是求 $\text{calc}(N,S)$ 与 $\text{calc}(N,D)$ 的距离。

不难看出，$N(N>1)$ 级城市由四座 $N-1$ 级城市组成，其中左上的 $N-1$ 级城市顺时针旋转了 90 度，左下的 $N-1$ 级城市逆时针旋转了 90 度。进一步观察，当这四座 $N-1$ 级城市首尾相接后，左上、左下的 $N-1$ 级城市的房屋编号顺序各自发生了颠倒，这相当于左上、左下两座城市发生了“水平翻转”。读者可以根据题目中等级 1 和等级 2 的两幅图验证上述结论。

在求解 $\text{calc}(N,M)$ 时，因为 $N-1$ 级城市有 2^{2N-2} 座房屋，所以我们先递归求

解 calc($N-1, M \bmod 2^{2N-2}$)，记求出的位置为 (x,y)，其中 x 为行号，y 为列号，从 0 开始编号。再根据 $M/2^{2N-2}$ 的大小，很容易确定编号为 M 的房屋处于四座 $N-1$ 级城市中的哪一座。

1. 若处于左上的 $N-1$ 级城市中，则需要把 (x,y) 所在的 $N-1$ 级城市顺时针旋转 90 度，坐标变为 $(y, 2^{N-1}-x-1)$，再水平翻转，坐标最终变为 (y,x)。这就是该房屋在 N 级城市中的位置。

2. 若处于右上的 $N-1$ 级城市中，则该房屋在 N 级城市中的位置应为 $(x, y+2^{N-1})$。

3. 若处于右下的 $N-1$ 级城市中，则该房屋在 N 级城市中的位置应为 $(x+2^{N-1}, y+2^{N-1})$。

4. 若处于左下的 $N-1$ 级城市中，则需要把 (x,y) 所在的 $N-1$ 级城市逆时针旋转 90 度再水平翻转，坐标变为 $(2^{N-1}-y-1, 2^{N-1}-x-1)$。在 N 级城市中的位置还要把行号再加 2^{N-1}，最终得到 $(2^N-y-1, 2^{N-1}-x-1)$。

```
pair<long long, long long> calc(int n, long long m) {
    if (n == 0) return make_pair(0, 0);
    long long len = 1ll << (n - 1), cnt = 1ll << (2 * n - 2);
    pair<long long, long long> pos = calc(n - 1, m % cnt);
    long long x = pos.first, y = pos.second;
    long long z = m / cnt;
    if (z == 0) return make_pair(y, x);
    if (z == 1) return make_pair(x, y + len);
    if (z == 2) return make_pair(x + len, y + len);
    if (z == 3) return make_pair(2 * len - y - 1, len - x - 1);
}
```

递归的机器实现

递归在计算机中是如何实现的？换句话说，它最终被编译成什么样的机器语言？这就要从函数调用说起。实际上，一台典型的 32 位计算机采用“堆栈结构”来实现函数调用，它在汇编语言中，把函数所需的第 k 个，第 $k-1$ 个，…，第 1 个参数依次入栈，然后执行 call (address)指令。该指令把返回地址（当前语句的下一条语句的地址）入栈，然后跳转到 address 位置的语句。在函数返回时，它执行 ret 指令。该指令把返回地址出栈，并跳转到该地址继续执行。

对于函数中定义的 C++局部变量，在每次执行 call 与 ret 指令时，也会在“栈”中相应地保存与复原，而作用范围超过该函数的变量，以及通过 new 和 malloc 函数动态分配的空间则保存在另一块称为“堆”（注意，这个堆与我们所说的二叉堆是两个不同

的概念）的结构中。栈指针、返回值、局部的运算会借助 CPU 的“寄存器”完成。

由此我们可以得知：

1. 局部变量在每层递归中都占有一份空间，声明过多或递归过深就会超过“栈”所能存储的范围，造成栈溢出；

2. 非局部变量对于各层递归都共享同一份空间，需要及时维护、还原现场，以防止在各层递归之间存储和读取的数据互相影响。

了解了递归的机器实现之后，我们就可以使用模拟的方法，把递归程序改写为非递归程序。具体来说，我们可以使用一个数组来模拟栈，使用变量来模拟栈指针和返回值，使用 switch/case 或者 goto/label 来模拟语句跳转。

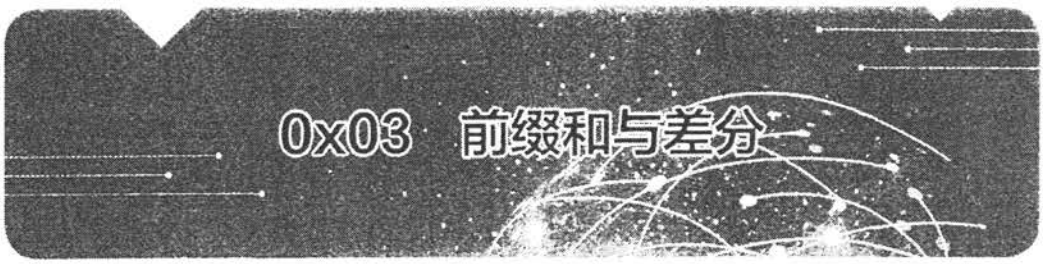
例题 非递归实现组合型枚举 AcWing93

把例题“递归实现组合型枚举”的程序通过模拟递归的方法改写为非递归版本。

```
vector<int> chosen;
int stack[100010], top = 0, address = 0, n, m;
void call(int x, int ret_addr) { // 模拟计算机汇编指令 call
    int old_top = top;
    stack[++top] = x; // 参数 x
    stack[++top] = ret_addr; // 返回地址标号
    stack[++top] = old_top; // 在栈顶记录以前的 top 值
}
int ret() { // 模拟计算机汇编指令 ret
    int ret_addr = stack[top - 1];
    top = stack[top]; // 恢复以前的 top 值
    return ret_addr;
}

int main() {
    cin >> n >> m;
    call(1, 0); // calc(1)
    while (top) {
        int x = stack[top - 2]; // 获取参数
        switch (address) {
        case 0:
            if (chosen.size()>m || chosen.size()+(n-x+1)<m) {
                address = ret(); // return
                continue;
            }
```

```
            if (x == n + 1) {
                for (int i = 0; i < chosen.size(); i++)
                    printf("%d ", chosen[i]);
                puts("");
                address = ret(); // return
                continue;
            }
            call(x+1, 1); // 相当于 calc(x+1)，返回后会从 case 1 继续
            address = 0;
            continue; // 回到 while 循环开头，相当于开始新的递归
        case 1:
            chosen.push_back(x);
            call(x+1, 2); // 相当于 calc(x+1)，返回后会从 case 2 继续
            address = 0;
            continue; // 回到 while 循环开头，相当于开始新的递归
        case 2:
            chosen.pop_back();
            address = ret(); // 相当于原 calc 函数结尾，执行 return
        }
    }
}
```



前缀和

对于一个给定的数列 A，它的前缀和数列 S 是通过递推能求出的基本信息之一：

$$S[i]=\sum_{j=1}^{i}A[j]$$

一个部分和，即数列 A 某个下标区间内的数的和，可表示为前缀和相减的形式：

$$\text{sum}(l,r)=\sum_{i=l}^{r}A[i]=S[r]-S[l-1]$$

在二维数组（矩阵）中，可类似地求出二维前缀和，进一步计算出二维部分和。

例题 激光炸弹 AcWing99

一种新型的激光炸弹，可以摧毁一个边长为 R 的正方形内的所有的目标。现在地图上有 $N(N\leq 10^4)$ 个目标，用整数 X_i,Y_i（其值在闭区间$[0,5000]$之内）表示目标在地图上的位置，每个目标都有一个价值 W_i。

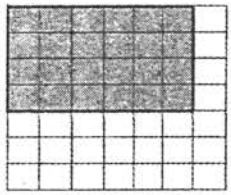
激光炸弹的投放是通过卫星定位的，但其有一个缺点，就是其爆破范围，即那个边长为 R 的正方形的边必须和 x,y 轴平行。若目标位于爆破正方形的边上，该目标不会被摧毁。求一颗炸弹最多能炸掉地图上总价值为多少的目标。

因为 X_i,Y_i 的值在 0~5000 之间，所以我们可以建立一个二维数组 A，其中 $A[i,j]$ 就等于位置 (i,j) 上的所有目标的价值之和。即对于每个目标，令 $A[X_i,Y_i]+=W_i$。

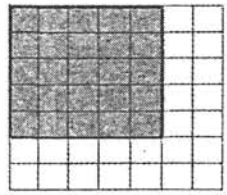
接下来我们求出 A 的二维前缀和 S，即：

$$S[i,j]=\sum_{x=1}^{i}\sum_{y=1}^{j}A[x,y]$$

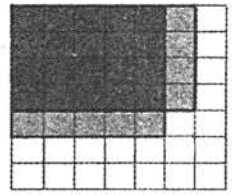
如下图所示，我们再观察 $S[i,j],S[i-1,j],S[i,j-1],S[i-1,j-1]$ 的关系。



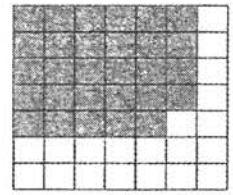
$S[i-1,j]$



$S[i,j-1]$



$S[i-1,j]+S[i,j-1]$



$S[i-1,j]+S[i,j-1]$
$-S[i-1,j-1]$

容易得到如下的递推式：

$$S[i,j]=S[i-1,j]+S[i,j-1]-S[i-1,j-1]+A[i,j]$$

同理，对于任意一个边长为 R 的正方形，我们有：

$$\sum_{x=i-R+1}^{i}\sum_{y=j-R+1}^{j}A[x,y]=S[i,j]-S[i-R,j]-S[i,j-R]+S[i-R,j-R]$$

因此，我们只需要 $O(N^2)$ 递推求出二维前缀和 S，然后 $O(N^2)$ 枚举边长为 R 的正方形的右下角坐标 (i,j)，即可通过上式 $O(1)$ 计算出该正方形内所有目标的价值之和，更新答案。上面给出的两个式子蕴含的思想其实就是“**容斥原理**”，0x37 节会详细探讨容斥原理的应用。

值得一提的是，上面我们把 (X_i,Y_i) 作为一个“格子”，而原题中 (X_i,Y_i) 是一个点，并且正方形边上的目标不会被摧毁。实际上，我们不妨认为这个点就处于“格子” (X_i,Y_i) 的中心位置，格子的左上角坐标为 $(X_i-0.5,Y_i-0.5)$，右下角坐标为 $(X_i+0.5,Y_i+0.5)$，而正方形的右下角处于“格线交点”上，即一个值为“□.5”的坐标。这个转化与原问题是等价的。另外，本题内存限制较为严格，我们可以省略 A 数组，读入时直接向 S 中累加。

差分

对于一个给定的数列 A，它的差分数列 B 定义为：

$$B[1]=A[1],\ B[i]=A[i]-A[i-1]\ (2\le i\le n)$$

容易发现，“前缀和”与“差分”是一对互逆运算，差分序列 B 的前缀和序列就是原序列 A，前缀和序列 S 的差分序列也是原序列 A。

把序列 A 的区间 $[l,r]$ 加 d（即把 $A_l,A_{l+1},\cdots,A_r$ 都加上 d），其差分序列 B 的变化为 B_l 加 d，B_{r+1} 减 d，其他位置不变。这有助于我们在很多题目中，把原序列上的“区间操作”转化为差分序列上的“单点操作”进行计算，降低求解难度。

在 0x63 节，我们还会继续探讨差分技巧在树上的应用。



例题 IncDec Sequence AcWing100

给定一个长度为 $n(n \le 10^5)$ 的数列 $\{a_1, a_2, \cdots, a_n\}$，每次可以选择一个区间 $[l,r]$，使下标在这个区间内的数都加 1 或者都减 1。

求至少需要多少次操作才能使数列中的所有数都一样，并求出在保证最少次数的前提下，最终得到的数列可能有多少种。

求出 a 的差分序列 b，其中 $b_1 = a_1$，$b_i = a_i - a_{i-1}\ (2 \le i \le n)$。令 $b_{n+1} = 0$。题目对序列 a 的操作，相当于每次可以选出 $b_1, b_2, \cdots, b_{n+1}$ 中的任意两个数，一个加 1，另一个减 1。目标是把 $b_2, b_3, \cdots, b_n$ 变为全零。最终得到的数列 a 就是由 n 个 b_1 构成的。

从 $b_1, b_2, \cdots, b_{n+1}$ 中任选两个数的方法可分为四类。

1. 选 b_i 和 b_j，其中 $2 \le i,j \le n$。这种操作会改变 $b_2, b_3, \cdots, b_n$ 中两个数的值。应该在保证 b_i 和 b_j 一正一负的前提下，尽量多地采取这种操作，更快地接近目标。

2. 选 b_1 和 b_j，其中 $2 \le j \le n$。

3. 选 b_i 和 b_{n+1}，其中 $2 \le i \le n$。

4. 选 b_1 和 b_{n+1}。这种情况没有意义，因为它不会改变 $b_2, b_3, \cdots, b_n$ 的值，相当于浪费了一次操作，一定不是最优解。

设 $b_2, b_3, \cdots, b_n$ 中正数总和为 p，负数总和的绝对值为 q。首先以正负数配对的方式尽量执行第 1 类操作，可执行 $\min(p,q)$ 次。剩余 $|p-q|$ 个未配对，每个可以选与 b_1 或 b_{n+1} 配对，即执行第 2 或第 3 类操作，共需 $|p-q|$ 次。

综上所述，最少操作次数为 $\min(p,q) + |p-q| = \max(p,q)$ 次。根据 $|p-q|$ 次第 2、第 3 类操作的选择情况，能产生 $|p-q|+1$ 种不同的 b_1 的值，即最终得到的序列 a 可能有 $|p-q|+1$ 种。

例题 Tallest Cow AcWing101

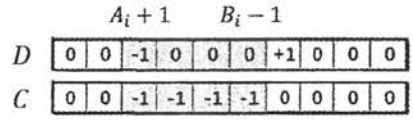
有 N 头牛站成一行。两头牛能够相互看见，当且仅当它们中间的牛身高都比它们矮。现在，我们只知道其中最高的牛是第 P 头，它的身高是 H，不知道剩余 $N-1$ 头牛的身高。但是，我们还知道 M 对关系，每对关系都指明了某两头牛 A_i 和 B_i 可以相互看见。求每头牛的身高最大可能是多少。

数据范围：$1 \le N,M \le 10^4$，$1 \le H \le 10^6$。

题目中的 M 对关系带给我们的信息实际上是牛之间身高的相对大小关系。具体来说，我们建立一个数组 C，数组中起初全为 0。若一条关系指明 A_i 和 B_i 可以互相看见（不妨设 $A_i < B_i$），则把数组 C 中下标为 A_i+1 到 B_i-1 的数都减去 1，意思是在 A_i 和 B_i 中间的牛，身高至少要比它们小 1。因为第 P 头牛是最高的，所以最终

$C[P]$ 一定为 0。其他的牛与第 P 头牛的身高差距就体现在数组 C 中。换言之，最后第 i 头牛的身高就等于 $H+C[i]$。

如果我们朴素执行把数组 C 中下标为 A_i+1 到 B_i-1 的数都减去 1 的操作，那么整个算法的时间复杂度为 $O(NM)$，复杂度过高。一个简单而高效的做法是，额外建立一个数组 D，对于每对 A_i 和 B_i，令 $D[A_i+1]$ 减去 1，$D[B_i]$ 加上 1。其含义是："身高减小 1"的影响从 A_i+1 开始，持续到 B_i-1，在 B_i 结束。最后，C 就等于 D 的前缀和，即 $C[i]=\sum_{j=1}^{i}D[j]$，如下图所示。

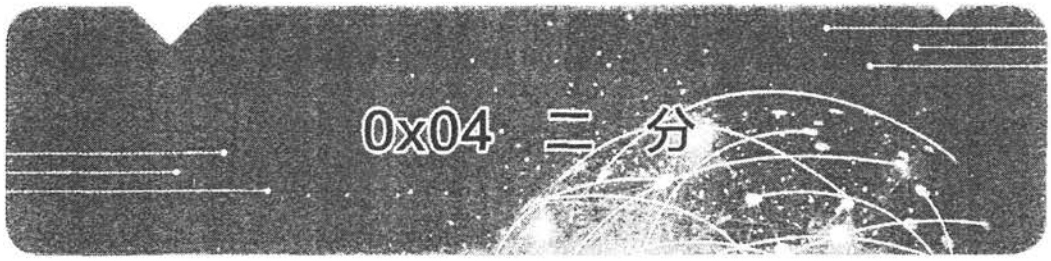


上述优化后的算法**把对一个区间的操作转化为左、右两个端点上的操作**，再通过**前缀和**得到原问题的解。这种思想很常用，我们在后面还会多次遇到。该算法的时间复杂度为 $O(N+M)$。

值得提醒的是，在本题的数据中，一条关系 (A_i,B_i) 可能会输入多次，要注意检查，对于重复的关系，只在第一次出现时执行相关操作即可。

```
map<pair<int, int>, bool> existed;
int c[10010], d[10010];

int main() {
    int n, p, h, m;
    cin >> n >> p >> h >> m;
    for (int i = 1; i <= m; i++) {
        int a, b;
        scanf("%d%d", &a, &b);
        if (a > b) swap(a, b);
        if (existed[make_pair(a, b)]) continue;
        d[a + 1]--, d[b]++;
        existed[make_pair(a, b)] = true;
    }
    for (int i = 1; i <= n; i++) {
        c[i] = c[i - 1] + d[i];
        printf("%d\n", h + c[i]);
    }
    cout << endl;
}
```



二分法是一种随处可见却非常精妙的算法，经常能为我们打开解答问题的突破口。二分的基础的用法是在单调序列或单调函数中进行查找。因此，当问题的答案具有单调性时，就可以通过二分把求解转化为判定（根据复杂度理论，判定的难度小于求解），这使得二分的运用范围变得很广泛。进一步地，我们还可以扩展到通过三分法去解决单峰函数的极值以及相关问题。

据说，只有 10% 的程序员能写对二分。二分的实现方法多种多样，但是其细节之处确实需要仔细考虑。对于整数域上的二分，需要注意终止边界、左右区间取舍时的开闭情况，避免漏掉答案或造成死循环；对于实数域上的二分，需要注意精度问题。本书讲解并通篇固定使用其中一种常见的二分实现方法，一般来说读者熟练掌握自己的一种正确写法即可。

整数集合上的二分

本书所使用的二分的写法保证最终答案处于闭区间 $[l,r]$ 以内，循环以 $l=r$ 结束，每次二分的中间值 mid 会归属于左半段与右半段二者之一。

在单调递增序列 a 中查找 $\geq x$ 的数中最小的一个（即 x 或 x 的后继）：

```
while (l < r) {
    int mid = (l + r) >> 1;
    if (a[mid] >= x) r = mid; else l = mid + 1;
}
return a[l];
```

在单调递增序列 a 中查找 $\leq x$ 的数中最大的一个（即 x 或 x 的前驱）：

```
while (l < r) {
    int mid = (l + r + 1) >> 1;
    if (a[mid] <= x) l = mid; else r = mid - 1;
}
return a[l];
```

在第一段代码中，若 $a[mid] \geq x$，则根据序列 a 的单调性，mid 之后的数会更大，所以 $\geq x$ 的最小的数不可能在 mid 之后，可行区间应该缩小为左半段。因为 mid 也可能是答案，故此时应取 $r = mid$。同理，若 $a[mid] < x$，取 $l = mid + 1$。

在第二段代码中，若 $a[mid] \leq x$，则根据序列 a 的单调性，mid 之前的数会更小，所以 $\leq x$ 的最大的数不可能在 mid 之前，可行区间应该缩小为右半段。因为 mid 也可能是答案，故此时应取 $l = mid$。同理，若 $a[mid] > x$，取 $r = mid - 1$。

如上面两段代码所示，这种二分写法可能会有**两种形式**。

1. 缩小范围时，$r = mid$，$l = mid + 1$，取中间值时，$mid = (l + r) \gg 1$。

2. 缩小范围时，$l = mid$，$r = mid - 1$，取中间值时，$mid = (l + r + 1) \gg 1$。

如果不对 mid 的取法加以区分，假如第二段代码也采用 $mid = (l + r) \gg 1$，那么当 $r - l$ 等于 1 时，就有 $mid = (l + r) \gg 1 = l$。接下来若进入 $l = mid$ 分支，可行区间未缩小，造成死循环；若进入 $r = mid - 1$ 分支，造成 $l > r$，循环不能以 $l = r$ 结束。因此对两个形式采用**配套的 mid 取法**是必要的。上面两段代码所示的两个形式共同组成了这种二分的实现方法。注意，我们在二分实现中采用了**右移运算** $\gg 1$，而不是整数除法 /2。这是因为右移运算是向下取整，而整数除法是向零取整，在二分值域包含负数时后者不能正常工作。第 0x01 节曾提到了二者的区别。

仔细分析这两种 mid 的取法，我们还发现：$mid = (l + r) \gg 1$ 不会取到 r 这个值，$mid = (l + r + 1) \gg 1$ 不会取到 l 这个值。我们可以利用这一性质来处理无解的情况，把最初的二分区间 $[1, n]$ 分别扩大为 $[1, n + 1]$ 和 $[0, n]$，把 a 数组的一个越界的下标包含进来。如果最后二分终止于扩大后的这个越界下标上，则说明 a 中不存在所求的数。

总而言之，正确写出这种二分的流程是：

1. 分析具体问题，确定左、右半段哪一个是可行区间，以及 mid 归属哪一半段；

2. 根据分析结果，选择“$r = mid, l = mid + 1, mid = (l + r) \gg 1$”和“$l = mid, r = mid - 1, mid = (l + r + 1) \gg 1$”两个配套形式之一；

3. 二分终止条件是 $l == r$，该值就是答案所在位置。

本书使用的这种二分方法的优点是始终保持答案位于二分区间内，二分结束条件对应的值恰好在答案所处位置，还可以很自然地处理无解的情况，形式优美。唯一的缺点是由两种形式共同组成，需要认真考虑实际问题选择对应的形式。

也有其他的二分写法，采用“$l = mid + 1, r = mid - 1$”或“$l = mid, r = mid$”来避免产生两种形式，但也相应地造成了丢失在 mid 点上的答案、二分结束时可行区间未缩小到确切答案等问题，需要额外加以处理，这里就不再一一讲述。

C++ STL 中的 lower_bound 与 upper_bound 函数实现了在一个序列中二分查找某个整数 x 的后继，其用法参见第 0x71 节“C++ STL”。

实数域上的二分

在实数域上二分较为简单，确定好所需的精度 eps，以 $l+eps<r$ 为循环条件，每次根据在 mid 上的判定选择 $r=mid$ 或 $l=mid$ 分支之一即可。一般需要保留 k 位小数时，则取 $eps=10^{-(k+2)}$。

```
while (l + 1e-5 < r) {
    double mid = (l + r) / 2;
    if (calc(mid)) r = mid; else l = mid;
}
```

有时精度不容易确定或表示，就干脆采用循环固定次数的二分方法，也是一种相当不错的策略。这种方法得到的结果的精度通常比设置 eps 更高。

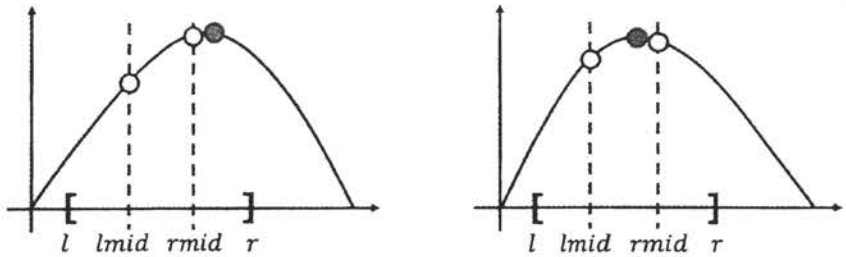
```
for (int i = 0; i < 100; i++) {
    double mid = (l + r) / 2;
    if (calc(mid)) r = mid; else l = mid;
}
```

三分求单峰函数极值

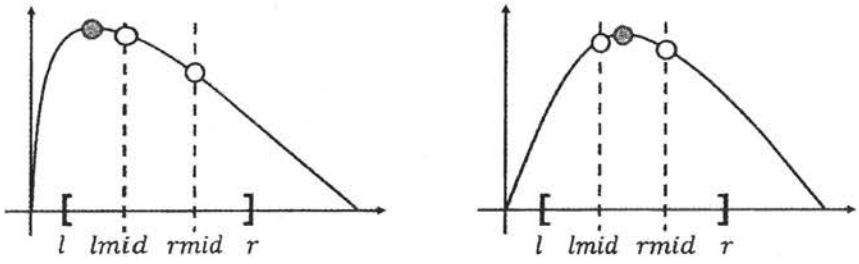
有一类函数被称为**单峰函数**，它们拥有**唯一的极大值点**，在极大值点左侧**严格单调上升**，右侧**严格单调下降**；或者拥有唯一的极小值点，在极小值点左侧严格单调下降，在极小值点右侧严格单调上升。为了避免混淆，我们也称后一种为单谷函数。对于单峰函数或单谷函数，我们可以通过三分法求其极值。

以单峰函数 f 为例，我们在函数定义域 $[l,r]$ 上任取两个点 $lmid$ 与 $rmid$，把函数分为三段。

1. 若 $f(lmid)<f(rmid)$，则 $lmid$ 与 $rmid$ 要么同时处于极大值点左侧（单调上升函数段），要么处于极大值点两侧。无论哪种情况下，极大值点都在 $lmid$ 右侧，可令 $l=lmid$。



2. 同理，若 $f(lmid) > f(rmid)$，则极大值点一定在 $rmid$ 左侧，可令 $r = rmid$。



如果我们取 $lmid$ 与 $rmid$ 为三等分点，那么定义域范围每次缩小 1/3。如果我们取 $lmid$ 与 $rmid$ 在二等分点两侧极其接近的地方，那么定义域范围每次近似缩小 1/2，通过 log 级别的时间复杂度即可在指定精度下求出极值。这就是**三分法**。

注意，我们在介绍单峰函数时特别强调了“**严格**”**单调性**。若在三分过程中遇到 $f(lmid) = f(rmid)$，当函数严格单调时，令 $l = lmid$ 或 $r = rmid$ 均可。如果函数不严格单调，即在函数中存在一段值相等的部分，那么我们无法判断定义域的左右边界如何缩小，三分法就不再适用。

二分答案转化为判定

大多数读者小时候可能都玩过“猜数字游戏”，规则如下：

主持人在 $1 \sim n$ 之间随机选择一个数字，请你猜选出的是哪个数字。如果猜错了，主持人会告诉你，你猜的数字比正确的数字是大还是小。然后你继续猜测。

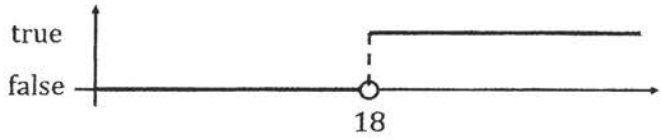
一个傻瓜式的算法是从 $1 \sim n$ 挨个猜一遍。一个高效的算法是每次猜中位数，通过主持人告知的信息，可以把数的范围每次缩小一半。这其实就是“二分答案”算法的应用。这里**求解**的是猜数字问题，主持人给的大小关系是一种**判定**。对于任何问题，如果我们有了一个判定算法，那把解空间枚举并判定一遍，当然就得到解了。而当解空间具有单调性时，我们就可以用二分代替枚举，利用二分答案并判定的方法，快速求出最终的解。

在一般的最优化问题上，二分答案也有着广泛的应用。请大家考虑如下问题：

有 N 本书排成一行，已知第 i 本的厚度是 A_i。
把它们分成**连续的** M 组，使 T 最小化，其中 T 表示厚度之和最大的一组的厚度。

本题是一道**最优化问题**，其目标是：最小化“厚度之和最大的一组的厚度”。这个问题不容易直接求解，我们可以转而考虑它对应的**判定问题**：给定一个值 T，求“把书划分为 M 组，使厚度之和最大的一组的厚度 $\leq T$”是否可行。另一种表述是：给定一个值 T，求“把书划分为 M 组，使每组的厚度都 $\leq T$”是否可行。

请大家仔细体会这里为什么是 $\leq T$ 而不是等号。举一个简单的例子，5 本书，厚度分别为 [7,2,5,10,8]，划分成连续的 2 组。一共只有四种划分方案：[7 | 2,5,10,8]，[7,2 | 5,10,8]，[7,2,5 | 10,8] 和 [7,2,5,10 | 8]，对应的最厚一组的厚度分别为 25、23、18、24。这四个数值是零散的，但我们考虑：如果要求每组的厚度都 < 18，那么这 2 组一定不能容纳这些书，必然需要更多的组才能把书分完，意味着不存在可行的分书方案。如果每组的厚度可以 ≥ 18，那么一定存在一种分书方案使得组数不会超过 2。最优解 18 就处于分书可行性的分界点上。这个解空间具有一种特殊的单调性——单调分段 0/1 函数，如下图所示。



直接求出答案 18 比较困难，但可以通过猜测一个值 T，判断 T 是否合法（true 或 false），从而得知答案是在 T 左侧还是右侧。最高效的猜法当然就是二分，而判定问题很容易通过贪心来计算——在不超过 T 的前提下，每组能多放就多放。请读者参考下面的程序。

```
// 把 n 本书分成 m 组，每组厚度之和<=size，是否可行
bool valid(int size) {
    int group = 1, rest = size;
    for (int i = 1; i <= n; i++) {
        if (rest >= a[i]) rest -= a[i];
        else group++, rest = size - a[i];
    }
    return group <= m;
}

int main() {
    // 二分答案，判定“每组厚度之和不超过二分的值”时
    // 能否在 m 组内把书分完
    int l = max_of_ai, r = sum_of_ai;
    while (l < r) {
        int mid = (l + r) / 2;
        if (valid(mid)) r = mid; else l = mid + 1;
    }
    cout << l << endl;
}
```

本题的题目描述中出现了类似于“**最大值最小**”的含义。二分答案在出现这类字眼的题目上应用尤其普遍。“最大值最小”中的“最小”是一个最优化目标，而“最大”是一个限制条件（限制每组书的厚度之和）。这类最优化问题对应的判定问题的条件通常是一个**不等式**（厚度之和的最大值 $\leq T$），这个不等式就反映了上述限制条件。**二分答案的本质是建立了一个定义域为解空间、值域为 0 或 1 的单调分段 0/1 函数，在这个函数上二分查找分界点**。该函数应以最优解为分界，计算该函数就是计算原问题对应的判定问题。二分答案通过把“最优化”转化为“判定”，降低了解决问题的难度。

例题 **Best Cow Fences** AcWing102

给定正整数数列 A，求一个平均数最大的、长度不小于 L 的（连续的）子段。

二分答案，判定“是否存在一个长度不小于 L 的子段，平均数不小于二分的值”。

如果把数列中每个数都减去二分的值，就转化为判定“**是否存在一个长度不小于 L 的子段，子段和非负**”。下面我们着重来解决以下两个问题。

1. 求一个子段，它的和最大，没有“长度不小于 L”这个限制。

无长度限制的最大子段和问题是一个经典问题，只需 $O(n)$ 扫描该数列，不断把新的数加入子段，当子段和变成负数时，把当前的整个子段清空。扫描过程中出现过的最大子段和即为所求。对这个算法不熟悉的读者建议做 To the Max(POJ1050) 这道题作为练习。

2. 求一个子段，它的和最大，子段的长度不小于 L。

子段和可以转化为前缀和相减的形式，即设 sum_i 表示 $A_1 \sim A_i$ 的和，则有：

$$\max_{i-j\geq L}\{A_{j+1}+A_{j+2}+\cdots+A_i\}=\max_{L\leq i\leq n}\left\{sum_i-\min_{0\leq j\leq i-L}\{sum_j\}\right\}$$

仔细观察上面的式子可以发现，随着 i 的增长，j 的取值范围 $0\sim i-L$ 每次只会增大 1。换言之，每次只会有一个新的取值进入 $\min\{sum_j\}$ 的候选集合，所以我们没有必要每次循环枚举 j，只需要用一个变量记录当前最小值，每次与新的取值 sum_{i-L} 取 min 就可以了。

```
double ans = -1e10;
double min_val = 1e10;
for (int i = L; i <= N; i++) {
    min_val = min(min_val, sum[i - L]);
    ans = max(ans, sum[i] - min_val);
}
```

解决了问题 2 之后，我们只需要看一下最大子段和是不是非负数，就可以确定二分上下界的变化范围了。

请读者自己思考使用二分的前提：为什么该问题的答案具有单调性？

整道题目的参考程序如下：

```
double a[100001], b[100001], sum[100001];
int main() {
    int N, L; cin >> N >> L;
    for (int i = 1; i <= N; i++) scanf("%lf", &a[i]);
    double eps = 1e-5;
    double l = -1e6, r = 1e6;
    while (r - l > eps) {
        double mid = (l + r) / 2;
        for (int i = 1; i <= N; i++) b[i] = a[i] - mid;
        for (int i = 1; i <= N; i++)
            sum[i] = (sum[i - 1] + b[i]);
        double ans = -1e10;
        double min_val = 1e10;
        for (int i = L; i <= N; i++) {
            min_val = min(min_val, sum[i - L]);
            ans = max(ans, sum[i] - min_val);
        }
        if (ans >= 0) l = mid; else r = mid;
    }
    cout << int(r * 1000) << endl;
}
```

例题 **Innovative Business** AcWing113

有 N 个元素，每一对元素之间的大小关系是确定的，关系不具有传递性。也就是说，元素的大小关系是 N 个点与 $N*(N-1)/2$ 条有向边构成的任意有向图。

然而，这是一道交互式试题，这些关系不能一次性得知，你必须通过不超过 10000 次提问，每次提问只能了解某两个元素之间的关系，把这 N 个元素排成一行，使得每个元素都小于右边与它相邻的元素。$N \leq 1000$。

根据数学归纳法，假设前 $k-1$ 个元素已经按要求排成一行，如果能确定第 k 个元素应该放在哪一个前面，即可解决此问题。

我们可以通过这样一种二分法确定第 k 个元素的位置：若第 k 个元素比第 mid 个元素小，令 $r = mid$，否则令 $l = mid + 1$。二分的初始区间可设为 $[1,k]$，区间中的 k 这个值表示放在所有 $k-1$ 个元素之后。

可以证明二分一定可以找到一个合法的位置插入，证明如下。

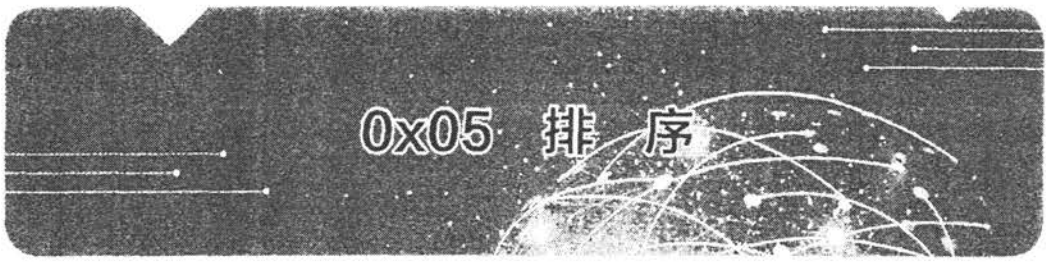
1. 如果第 k 个元素比第 mid 个元素小。

继续比较 k 与 $mid-1$ 这两个元素，若第 k 个元素比第 $mid-1$ 个元素大，则 k 可以插在 $mid-1$ 与 mid 之间；否则说明元素 k 比元素 $mid-1$ 小，那就再比较 k 与 $mid-2$ 这两个元素，依此类推，直到发现第 k 个元素比第 1 个元素小，那就放在最前边。

2. 如果第 k 个元素比第 mid 个元素大，同理。

以上只是一个证明，我们当然不会真的去依次比较 k 与每个元素[①]。实际上通过二分，我们每询问一次（k 与 mid），就可以把区间 $[l,r]$ 缩小一半，因此至多 $\log k$ 次询问就可以确定 k 应该放在哪里。把 N 个元素排成一行的总询问次数不超过 $N\log N$。

① 本题的解答过程事实上证明了：任意有向完全图（又称竞赛图）都存在 Hamilton 路径。



在程序设计中，通常会使用到以下这些排序算法，这里把它们分为三类：

1. 选择排序、插入排序、冒泡排序
2. 堆排序、归并排序、快速排序
3. 计数排序、基数排序、桶排序

前两类是基于比较的排序算法，对 n 个元素进行排序时，若元素比较大小的时间复杂度为 $O(1)$，则第一类排序算法的时间复杂度为 $O(n^2)$，第二类排序算法的时间复杂度为 $O(n\log n)$。实际上，基于比较的排序算法的时间复杂度下界为 $O(n\log n)$，因此堆排序、归并排序与快速排序已经是时间复杂度最优的基于比较的排序算法。

第三类算法换了一种思路，它们不直接比较大小，而是对被排序的数值采取按位划分、分类映射等处理方式，其时间复杂度不仅与 n 有关，还与数值的大小范围 m 有关。读者很容易在各种算法书籍或网络上找到这些排序算法的讲解与实现，这里就不再赘述，请读者自行学习理解。讨论这些排序算法的应用并以它们作为工具去解决问题才是我们的重点。

离散化

排序算法的第一个应用是离散化。通俗地讲，“离散化”就是把无穷大集合中的若干个元素映射为有限集合以便于统计的方法。例如，在很多情况下，问题的范围虽然定义在整数集合 $\mathbb{Z}$，但是只涉及其中 m 个有限数值，并且与数值的绝对大小无关（只把这些数值作为代表，或只与它们的相对顺序有关）。此时，我们就可以把整数集合 $\mathbb{Z}$ 中的这 m 个整数与 $1\sim m$ 建立映射关系。如果有一个时间、空间复杂度与数值范围 $\mathbb{Z}$ 的大小有关的算法，在离散化后，该算法的时间、空间复杂度就降低为与 m 相关。

具体地说，假设问题涉及 int 范围内的 n 个整数 $a[1]\sim a[n]$，这 n 个整数可能有重复，去重以后共有 m 个整数。我们要把每个整数 $a[i]$ 用一个 $1\sim m$ 之间的整数代替，并且保持大小顺序不变，即如果 $a[i]$ 小于（或等于、大于）$a[j]$，那么代替 $a[i]$ 的整数也小于（或等于、大于）代替 $a[j]$ 的整数。

很简单，我们可以把 a 数组排序并去掉重复的数值，得到有序数组 $b[1]\sim b[m]$，在 b 数组的下标 i 与数值 $b[i]$ 之间建立映射关系。若要查询整数 $i(1\le i\le m)$ 代

替的数值，只需直接返回 $b[i]$；若要查询整数 $a[j](1 \le j \le n)$ 被哪个 1~m 之间的整数代替，只需在数组 b 中二分查找 $a[j]$ 的位置即可。

```
void discrete() { // 离散化
    sort(a + 1, a + n + 1);
    for (int i = 1; i <= n; i++) // 也可用 STL 中的 unique 函数
        if (i == 1 || a[i] != a[i - 1])
            b[++m] = a[i];
}

int query(int x) { // 查询 x 映射为哪个 1~m 之间的整数
    return lower_bound(b + 1, b + m + 1, x) - b;
}
```

例题 Cinema AcWing103

有 m 部正在上映的电影，每部电影的语音和字幕都采用不同的语言，用一个 int 范围内的整数来表示语言。有 n 个人相约一起去看其中一部电影，每个人只会一种语言，如果一个人能听懂电影的语音，他会很高兴；如果能看懂字幕，他会比较高兴；如果语音和字幕都不懂，他会不开心。请你选择一部电影让这 n 个人一起看，使很高兴的人最多。若答案不唯一，则在此前提下再让比较高兴的人最多。$n,m \le 2 * 10^5$。

虽然语言的范围在 int 以内，但是这 m 部电影与 n 个人最多涉及 $2*m+n$ 种语言。我们把所有电影和人涉及的语言放进一个数组，排序并离散化，用一个 1~$2*m+n$ 之间的整数代替每种语言。此时我们就可以用数组直接统计会上述每种语言的人的数量，从而选择满足题目要求的电影。时间复杂度为 $O((n+m)\log(n+m))$。

中位数

在有序序列中，中位数具有一些很优美的性质，可以引出一系列与它相关的问题。动态维护序列的中位数也非常值得探讨。我们通过几道例题来感受中位数的相关应用。

例题 货仓选址 AcWing104

在一条数轴上有 N 家商店，它们的坐标分别为 $A[1]$~$A[N]$。现在需要在数轴上建立一家货仓，每天清晨，从货仓到每家商店都要运送一车商品。为了提高效率，求把货仓建在何处，可以使得货仓到每家商店的距离之和最小。

把 $A[1]$~$A[N]$ 排序，设货仓建在 X 坐标处，X 左侧的商店有 P 家，右侧的商店有 Q 家。若 $P < Q$，则每把货仓的选址向右移动 1 单位距离，距离之和就会变小 $Q-$

P。同理，若 $P > Q$，则货仓的选址向左移动会使距离之和变小。当 $P = Q$ 时为最优解。

因此货仓应该建在中位数处，即把 A 排序后，当 N 为奇数时，货仓建在 $A[(N+1)/2]$ 处最优；当 N 为偶数时，货仓建在 $A[N/2]$~$A[N/2+1]$ 之间的任何位置都是最优解。

例题 七夕祭 AcWing105

七夕节因牛郎织女的传说而被扣上了“情人节”的帽子。于是 TYVJ 今年举办了一次线下七夕祭。Vani 同学今年成功邀请到了 cl 同学陪他来共度七夕，于是他们决定去 TYVJ 七夕祭游玩。

TYVJ 七夕祭和 11 区的夏祭的形式很像。矩形的祭典会场由 N 行 M 列共计 $N*M$ 个摊点组成。虽然摊点种类繁多，不过 cl 只对其中的 T 个摊点感兴趣，比如章鱼烧、苹果糖、棉花糖、射的屋什么的。Vani 预先联系了七夕祭的负责人 zhq，希望能够通过恰当地布置会场，使得各行中 cl 感兴趣的摊点数一样多，并且各列中 cl 感兴趣的摊点数也一样多。

不过 zhq 告诉 Vani，摊点已经随意布置完毕了，如果想满足 cl 的要求，唯一的调整方式就是交换两个相邻的摊点。两个摊点相邻，当且仅当它们处在同一行或者同一列的相邻位置上。因为 zhq 率领的 TYVJ 开发小组成功地扭曲了空间，每一行或每一列的第一个位置和最后一个位置也算作相邻。现在 Vani 想知道他的两个要求最多能满足多少个；在此前提下，至少需要交换多少次摊点。$1 \le N, M \le 10^5$，$0 \le T \le \min(N*M, 10^5)$。

经过分析，我们可以发现，交换左右两个相邻的摊点只会改变某两列中 cl 感兴趣的摊点数，不会改变每行中 cl 感兴趣的摊点数。同理，交换上下两个相邻的摊点只会改变某两行中 cl 感兴趣的摊点数，不会改变每列中 cl 感兴趣的摊点数。所以我们可以把本题分成两个互相独立的部分计算。

1. 通过最少次数的左右交换使每列中 cl 感兴趣的摊点数相同。
2. 通过最少次数的上下交换使每行中 cl 感兴趣的摊点数相同。

以第 1 个问题为例进行探讨。

我们可以统计出在初始情况下，每列中 cl 感兴趣的摊点总数，记为 $C[1]$~$C[M]$。若 cl 感兴趣的摊点总数 T 不能被 M 整除，则不可能达到要求。若 T 能被 M 整除，则我们的目标就是让每列中有 T/M 个 cl 感兴趣的摊点。

思考到这里，读者可能已经想到了一个与此类似的经典问题“均分纸牌”。“均分纸牌”问题是说，有 M 个人排成一行，他们手中分别有 $C[1]$~$C[M]$ 张纸牌，在每一步操作中，可以让某个人把自己手中的一张纸牌交给他旁边的一个人，求至少需要多少

步操作才能让每个人手中持有的纸牌数相等。显然，当所有人手中持有的纸牌总数 T 能被 M 整除时，“均分纸牌”问题有解，在有解时，我们可以先考虑第一个人。

1. 若 $C[1] > T/M$，则第一个人需要给第二个人 $C[1] - T/M$ 张纸牌，即把 $C[2]$ 加上 $C[1] - T/M$。

2. 若 $C[1] < T/M$，则第一个人需要从第二个人手中拿 $T/M - C[1]$ 张纸牌，即把 $C[2]$ 减去 $T/M - C[1]$。

按照同样的方法，依次考虑第 2~M 个人。即使在某个时刻有某个 $C[i]$ 被减为负数也没有关系，因为接下来 $C[i]$ 就会从 $C[i+1]$ 处拿牌，在实际中可以认为 $C[i]$ 从 $C[i+1]$ 处拿牌发生在 $C[i-1]$ 从 $C[i]$ 处拿牌之前。按照这种方法，经过计算，达到目标所需要的最少步数其实就是：

$$\sum_{i=1}^{M} \left| i * \frac{T}{M} - G[i] \right|$$，其中 G 是 C 的前缀和，即 $G[i] = \sum_{j=1}^{i} C[j]$

其含义是每个“前缀”最初共持有 $G[i]$ 张纸牌，最终会持有 $i * T/M$ 张纸牌，多退少补，会与后边的人发生“二者之差的绝对值”张纸牌的交换。

如果我们设 $A[i] = C[i] - T/M$，即一开始就让每个人手中的纸牌数都减掉 T/M，并且最终让每个人手里都恰好有 0 张纸牌，答案显然不变，就是：

$$\sum_{i=1}^{M} |S[i]|$$，其中 S 是 A 的前缀和，即 $S[i] = \sum_{j=1}^{i} A[j]$

从数学的角度，以上两个公式也可以互相推导得到。

回到本题，如果不考虑“第 1 列与最后一列也是相邻的”这一条件，那么刚才提到的本题中的第 1 个问题与“均分纸牌”问题是等价的。若问题有解，一定存在一种适当的顺序，使得每一步传递纸牌的操作可以转化为交换一对左右相邻的摊点（其中 cl 恰好对这两个摊点之一感兴趣）。

若第 1 列与最后一列相邻，则问题等价于一个“环形均分纸牌”。仔细思考可以发现，一定存在一种最优解的方案，环上某两个相邻的人之间没有发生纸牌交换操作。于是有一种朴素的做法是，枚举这个没有发生交换的位置，把环断开看成一行，转化为一般的“均分纸牌”问题进行计算。

首先，一般的“均分纸牌”问题就相当于在第 M 个人与第 1 个人之间把环断开，此时这 M 个人写成一行，其持有的纸牌数、前缀和分别是：

$$\begin{array}{cc} A[1] & S[1] \\ A[2] & S[2] \\ \cdots & \cdots \\ A[M] & S[M] \end{array}$$

如果在第 k 个人之后把环断开写成一行，这 M 个人持有的纸牌数、前缀和分别是：

$$\begin{array}{ll} A[k+1] & S[k+1]-S[k] \\ A[k+2] & S[k+2]-S[k] \\ \cdots & \cdots \\ A[M] & S[M]-S[k] \\ A[1] & S[1]+S[M]-S[k] \\ \cdots & \cdots \\ A[k] & S[k]+S[M]-S[k] \end{array}$$

注意：此处 A 是减去最终每个人手里纸牌数 T/M 之后的数组，A 数组均分之后每个人手里都有 0 张纸牌，所以 $S[M]=0$。也就是说，从第 k 个人把环断开写成一行，前缀和数组的变化是每个位置都减去 $S[k]$。

根据我们上面推导的公式，所需最少步数为：

$\sum_{i=1}^{M}|S[i]-S[k]|$，其中 S 是 A 的前缀和，即 $S[i]=\sum_{j=1}^{i}A[j]$

当 k 取何值时上式最小？这就是上一题“货仓选址”！其中 $S[i]$ 是数轴上 M 家商店的位置，$S[k]$ 是货仓的位置，$|S[i]-S[k]|$ 就是二者之间的距离。根本不需要枚举 k，只需要把 S 从小到大排序，取中位数作为 $S[k]$ 就是最优解！至此，本题得到完美解决，时间复杂度为 $O(N\log N+M\log M)$。

综上所述，本题可类比为行、列方向上的两次“环形均分纸牌”问题。环形均分纸牌又类比为“均分纸牌”与“货仓选址”问题。其中的每一步都仅使用了基本算法和性质，最后转化为了简单而经典的问题。读者应该时刻把各种模型之间的简化、扩展和联系作为算法学习与设计的脉络，以点成线，触类旁通，才能产生数量到质量的飞跃。

例题 Running Median AcWing106

动态维护中位数问题：依次读入一个整数序列，每当已经读入的整数个数为奇数时，输出已读入的整数构成的序列的中位数。

本题有两种做法，使用“对顶堆”的在线做法（读入的同时即时计算答案）和使用“链表+Hash”的离线做法（完成所有读入后进行计算然后再统一输出）。后者将在第 0x13 节“链表与邻接表”中讲解。另外，我们将在第 0x17 节详细讨论“二叉堆”，对其概念不熟悉的读者可以先进行学习。

为了动态维护中位数，我们可以建立两个二叉堆：一个小根堆，一个大根堆。在依次读入这个整数序列的过程中，设当前序列长度为 M，我们始终保持：

1. 序列中从小到大排名为 $1\sim M/2$ 的整数存储在大根堆中；
2. 序列中从小到大排名为 $M/2+1\sim M$ 的整数存储在小根堆中。

任何时候，如果某一个堆中元素个数过多，打破了这个性质，就取出该堆的堆顶插入另一个堆。这样一来，序列的中位数就是小根堆的堆顶。

每次新读入一个数值 X 后，若 X 比中位数小，则插入大根堆，否则插入小根堆，在插入之后检查并维护上述性质即可。这就是“对顶堆”算法。

第 k 大数

给定 n 个整数，如何求出第 k 大的数？我们当然可以直接对这 n 个整数进行快速排序，然后输出从大到小排在第 k 个的数，时间复杂度为 $O(n\log n)$。实际上利用类似于快速排序的思想，只需要 $O(n)$ 的时间即可求出第 k 大数。

从大到小进行快速排序算法的思想是，在每一层递归中，随机选取一个数为基准，把比它大的数交换到“左半段”，把其余的数和基准值自身一起作为“右半段”，然后继续递归对左右两边分别进行排序，在平均情况下快速排序的复杂度为 $O(n\log n)$[①]。

实际上在每次选取基准值后，我们可以统计出大于基准值的数的数量 cnt，如果 $k \le cnt$，我们就在左半段（比基准值大的数中）寻找第 k 大数；如果 $k > cnt$，我们就在右半段（小于或等于基准值的数中）寻找第 $k-cnt$ 大数。因此，寻找第 k 大数时，我们只需要进入左右两半二者之一继续递归，在平均情况下，复杂度为 $n+n/2+n/4+\cdots+1=O(n)$。

逆序对

对于一个序列 a，若 $i<j$ 且 $a[i]>a[j]$，则称 $a[i]$ 与 $a[j]$ 构成逆序对。

使用归并排序可以在 $O(n\log n)$ 的时间里求出一个长度为 n 的序列中逆序对的个数。归并排序每次把序列二分，递归对左右两半排序，然后合并两个有序序列。

递归对左右两半排序时，可以把左右两半各自内部的逆序对数作为子问题计算。因此，我们只需要在合并时考虑“左边一半里一个较大的数”与“右边一半里一个较小的数”构成逆序对的情形，求出这种情形的个数。

合并两个有序序列 $a[l\sim mid]$ 与 $a[mid+1\sim r]$ 可以采用两个指针 i 与 j 分别对二者进行扫描的方式，不断比较两个指针所指向数值 $a[i]$ 和 $a[j]$ 的大小，将小的那个加入到排序的结果数组中。若小的那个是 $a[j]$，则 $a[i\sim mid]$ 都比 $a[j]$ 要大，它们都会与 $a[j]$ 构成逆序对，可以顺便统计到答案中。

```
void merge(int l, int mid, int r) {
    // 合并 a[l~mid]与 a[mid+1~r]
    // a 是待排序数组，b 是临时数组，cnt 是逆序对个数
```

① 关于快速排序的实现细节和时间复杂度证明，可参见配套资源提供的参考资料。

```
    int i = l, j = mid + 1;
    for (int k = l; k <= r; k++)
        if (j > r || i <= mid && a[i] <= a[j]) b[k] = a[i++];
        else b[k] = a[j++], cnt += mid - i + 1;
    for (int k = l; k <= r; k++) a[k] = b[k];
}
```

求逆序对的常用方法还有树状数组，我们将在后续的章节中讲解树状数组的应用。

例题 Ultra-QuickSort AcWing107

给定一个长度为 $n(n \le 5*10^5)$ 的序列 a，如果只允许进行比较和交换相邻两个数的操作，求至少需要多少次交换才能把 a 从小到大排序。

只通过比较和交换相邻两个数值的排序方法，实际上就是冒泡排序。在排序过程中每找到一对大小颠倒的相邻数值，把它们交换，就会使整个序列的逆序对个数减少 1。最终排好序后逆序对个数显然为 0，所以对 a 进行冒泡排序需要的最少交换次数就是序列 a 中逆序对的个数。我们直接使用归并排序求出 a 的逆序对数就是本题的答案。

例题 奇数码问题 AcWing108

你一定玩过八数码游戏，它实际上是在一个 3*3 的网格中进行的，1 个空格和 1~8 这 8 个数字恰好不重不漏地分布在这 3*3 的网格中。

```
5 2 8
1 3 _
4 6 7
```

在游戏过程中，可以把空格与其上、下、左、右四个方向之一的数字交换（如果存在）。例如在上例中，空格可与左、上、下面的数字交换，分别变成：

```
5 2 8      5 2 _      5 2 8
1 _ 3      1 3 8      1 3 7
4 6 7      4 6 7      4 6 _
```

奇数码游戏是它的一个扩展，在一个 $n*n$ 的网格中进行，其中 n 为奇数，1 个空格和 $1 \sim n*n-1$ 这 $n*n-1$ 个数恰好不重不漏地分布在 $n*n$ 的网格中。空格移动的规则与八数码游戏相同，实际上，八数码就是一个 $n=3$ 的奇数码游戏。

现在给定两个奇数码游戏的局面，请判断是否存在一种移动空格的方式，使得其中一个局面可以变化到另一个局面。奇整数 $3 \le n \le 500$。

扩展题目：POJ2893 M × N Puzzle

奇数码游戏两个局面可达，当且仅当两个局面下网格中的数依次写成 1 行 $n*n-$

1 个元素的序列后（不考虑空格），逆序对个数的奇偶性相同。例如题目描述中的第一个局面写成[5 2 8 1 3 4 6 7]。该结论的必要性很容易证明：空格左右移动时，写成的序列显然不变；空格向上（下）移动时，相当于某个数与它后（前）边的 $n-1$ 个数交换了位置，因为 $n-1$ 是偶数，所以逆序对数的变化也只能是偶数。该结论的充分性证明较为复杂，我们将不在此大篇幅讨论这样一个数学问题。

上面的结论还可以扩展到 n 为偶数的情况，此时两个局面可达，当且仅当两个局面对应网格写成序列后，“逆序对数之差”和“两个局面下空格所在的行数之差”奇偶性相同。事实上，在 $n*m$ 网格上 $(n,m\geq 2)$ 也服从上述两个结论之一（根据列数奇偶性分情况讨论）。

总而言之，$n*m$ 数码问题的有解性判定，可以转化为归并排序求逆序对来解决。



0x06 倍 增

倍增，字面意思就是“成倍增长”。这是指我们在进行递推时，如果状态空间很大，通常的线性递推无法满足时间与空间复杂度的要求，那么我们可以通过成倍增长的方式，只递推状态空间中在 2 的整数次幂位置上的值作为代表。当需要其他位置上的值时，我们通过“任意整数可以表示成若干个 2 的次幂项的和”这一性质，使用之前求出的代表值拼成所需的值。所以使用倍增算法也要求我们递推的问题的状态空间关于 2 的次幂具有可划分性。

“倍增”与“二进制划分”两个思想互相结合，降低了求解很多问题的时间与空间复杂度。我们之前学习的快速幂其实就是“倍增”与“二进制划分”思想的一种体现。在本节中，我们研究序列上的倍增问题，包括求解 RMQ（区间最值）问题的 ST 算法。关于求解最近公共祖先（LCA）等在树上的倍增应用，我们将在 0x63 节中进行探讨。

试想一个这样的问题：

给定一个长度为 N 的数列 A，然后进行若干次询问，每次给定一个整数 T，求出最大的 k，满足 $\sum_{i=1}^{k} A[i] \leq T$。你的算法必须是在线的（必须即时回答每一个询问，不能等待收到所有询问后再统一处理），假设 $0 \leq T \leq \sum_{i=1}^{N} A[i]$。

最朴素的做法当然是从前向后枚举 k，每次询问花费的时间与答案的大小有关，最坏情况下为 $O(N)$。

如果我们能够先花费 $O(N)$ 的时间预处理 A 数组的前缀和数组 S，就可以二分 k 的位置，比较 $S[k]$ 与 T 的大小来确定二分上下界的变化，每次询问花费的时间都是 $O(\log N)$。这个算法在平均情况下表现很好，但是它的缺点是如果每次询问给定的整数 T 都非常小，造成答案 k 也非常小，那么该算法可能还不如从前向后枚举更优。

我们可以设计这样一种倍增算法。

1. 令 $p = 1$，$k = 0$，$sum = 0$。

2. 比较“A 数组中 k 之后的 p 个数的和”与 T 的关系，也就是说，如果 $sum + S[k + p] - S[k] \leq T$，则令 $sum \mathrel{+}= S[k + p] - S[k]$，$k \mathrel{+}= p$，$p \mathrel{*}= 2$，即累加上这 p 个数的和，然后把 p 的跨度增长一倍。如果 $sum + S[k + p] - S[k] > T$，则令 $p \mathrel{/}= 2$。

3. 重复上一步，直到 p 的值变为 0，此时 k 就是答案。

这个算法始终在答案大小的范围内实施“倍增”与“二进制划分”思想，通过若干长度为 2 的次幂的区间拼成最后的 k，时间复杂度级别为答案的对数，能够应对 T 的各种大小情况。

例题 **Genius ACM** AcWing109

给定一个整数 M，对于任意一个整数集合 S，定义“校验值”如下：

从集合 S 中取出 M 对数（即 $2*M$ 个数，不能重复使用集合中的数，如果 S 中的整数不够 M 对，则取到不能取为止），使得“每对数的差的平方”之和最大，这个最大值就称为集合 S 的“校验值”。

现在给定一个长度为 N 的数列 A 以及一个整数 T。我们要把 A 分成若干段，使得每一段的“校验值”都不超过 T。求最少需要分成几段。

首先，对于一个集合 S，显然应该取 S 中最大的 M 个数和最小的 M 个数，最大和最小构成一对，次大和次小构成一对……这样求出的“校验值”最大。而为了让数列 A 分成的段数最少，每一段都应该在“校验值”不超过 T 的前提下，尽量包含更多的数。所以我们从头开始对 A 进行分段，让每一段尽量长，到达结尾时分成的段数就是答案。

于是，需要解决的问题为：当确定一个左端点 L 之后，右端点 R 在满足 $A[L]\sim A[R]$ 的“校验值”不超过 T 的前提下，最大能取到多少。

求长度为 N 的一段的“校验值”需要排序配对，时间复杂度为 $O(N\log N)$。当“校验值”上限 T 比较小时，如果在整个 $L\sim N$ 的区间上二分右端点 R，二分的第一步就要检验 $(N-L)/2$ 这么长的一段，最终右端点 R 却可能只扩展了一点儿，浪费了很多时间。与上一道题目一样，我们需要一个与右端点 R 扩展的长度相适应的算法——倍增。

可以采用与上一题类似的倍增过程：

1. 初始化 $p=1$，$R=L$；

2. 求出 $[L,R+p]$ 这一段区间的“校验值”，若“校验值” $\le T$，则 $R+=p$，$p*=2$，否则 $p/=2$；

3. 重复上一步，直到 p 的值变为 0，此时 R 即为所求。

上面这个过程至多循环 $O(\log N)$ 次，每次循环对长为 $O(R-L)$ 的一段进行排序完成整个题目的求解累计扩展长度为 N，所以总体时间复杂度为 $O(N\log^2 N)$。实际上我们每次求“校验值”时可以不用快速排序，而是采用类似归并排序的方法，只对新增的长度部分排序，然后合并新旧两段，这样总体时间复杂度可以降低到 $O(N\log N)$。

ST 算法

在 RMQ 问题（区间最值问题）中，著名的 ST 算法就是倍增的产物。给定一个长度为 N 的数列 A，ST 算法能在 $O(N\log N)$ 时间的预处理后，以 $O(1)$ 的时间复杂度



在线回答“数列 A 中下标在 l~r 之间的数的最大值是多少”这样的区间最值问题。

一个序列的子区间个数显然有 $O(N^2)$ 个，根据倍增思想，我们首先在这个规模为 $O(N^2)$ 的状态空间里选择一些 2 的整数次幂的位置作为代表值。

设 $F[i,j]$ 表示数列 A 中下标在子区间 $[i,i+2^j-1]$ 里的数的最大值，也就是从 i 开始的 2^j 个数的最大值。递推边界显然是 $F[i,0]=A[i]$，即数列 A 在子区间 $[i,i]$ 里的最大值。

在递推时，我们把子区间的长度成倍增长，有公式 $F[i,j]=\max(F[i,j-1],F[i+2^{j-1},j-1])$，即长度为 2^j 的子区间的最大值是左右两半长度为 2^{j-1} 的子区间的最大值中较大的一个。

```
void ST_prework() {
    for (int i = 1; i <= n; i++) f[i][0] = a[i];
    int t = log(n) / log(2) + 1;
    for (int j = 1; j < t; j++)
        for (int i = 1; i <= n - (1<<j) + 1; i++)
            f[i][j] = max(f[i][j-1], f[i + (1<<(j-1))][j-1]);
}
```

当询问任意区间 $[l,r]$ 的最值时，我们先计算出一个 k，满足 $2^k \le r-l+1 < 2^{k+1}$，也就是在 2 的 k 次幂小于区间长度的前提下最大的 k。那么“从 l 开始的 2^k 个数”和“以 r 结尾的 2^k 个数”这两段一定覆盖了整个区间 $[l,r]$，这两段的最大值分别是 $F[l,k]$ 和 $F[r-2^k+1,k]$，二者中较大的那个就是整个区间 $[l,r]$ 的最值。因为求的是最大值，所以这两段只要覆盖区间 $[l,r]$ 即可，即使有重叠也没关系。

```
int ST_query(int l, int r) {
    int k = log(r - l + 1) / log(2);
    return max(f[l][k], f[r - (1<<k) + 1][k]);
}
```

简便起见，我们在代码中使用了 cmath 库的 log 函数。该函数效率较高，一般来说对程序性能影响不大。更严格地讲，为了保证复杂度为 $O(1)$，应该 $O(N)$ 预处理出 1~N 这 N 种区间长度各自对应的 k 值，在询问时直接使用。

0x07 贪 心

贪心是一种在每次决策时采取当前意义下最优策略的算法，使用贪心法要求问题的整体最优性可以由局部最优性导出。贪心算法的正确性需要证明，常见手法如下。

1. 微扰（邻项交换）

证明在任意局面下，任何对局部最优策略的微小改变都会造成整体结果变差。经常用于以“排序”为贪心策略的证明。

2. 决策包容性

证明在任意局面下，做出局部最优决策以后，在问题状态空间中的可达集合包含了做出其他任何决策后的可达集合。换言之，这个局部最优策略提供的可能性包含其他所有策略提供的可能性。

3. 决策范围扩展

有时候不容易直接证明局部最优策略的正确性。此时可以往后多扩展一步，有助于我们对局部最优决策进行验证（验证时可以综合使用其他手法）。

4. 反证法

5. 数学归纳法

我们通过几道例题来介绍贪心算法的应用。

例题 股票买卖 II AcWing1055

给定一个长度为 N ($N \le 10^4$) 的数组，第 i 个数表示一支股票在第 i 天的价格。你可以多次买卖这支股票，但任意时刻最多持有 1 股（你必须在再次购买前出售掉之前的股票）。设计一个算法来计算你所能获取的最大利润。例如数组 7 1 5 3 6 4 可以在第 2 天买入，第 3 天卖出，第 4 天买入，第 5 天卖出，获得 $5-1+6-3=7$ 的利润。

这是一个理想的炒股状态——知道 N 天内每一天的股价。每一天只有买入、卖出或不操作三种策略。我们只需要往后多看一天，就知道当前是否该买入或卖出了。

1. 当前（第 i 天）持有股票，如果下一天还涨（$prices[i] < prices[i+1]$），那么应该继续持有，否则当然应该以 $prices[i]$ 卖出。

2. 当前（第 i 天）不持有股票，如果下一天涨（$prices[i] < prices[i+1]$），那么应该以 $prices[i]$ 买入，否则继续保持空仓。

模拟一遍这个交易过程就会发现，我们可以完美地获得每一个上涨区间的收益，所以最终答案其实就是：

$$\sum_{1 \le i < n} \max(prices[i+1] - prices[i], 0)$$

例题 **Sunscreen** AcWing110

有 C 头奶牛日光浴，第 i 头奶牛需要 minSPF[i] 和 maxSPF[i] 单位强度之间的阳光。每头奶牛在日光浴前必须涂防晒霜，防晒霜有 L 种，涂上第 i 种之后，身体接收到的阳光强度就会稳定为 SPF[i]，第 i 种防晒霜有 cover[i] 瓶。求最多可以满足多少头奶牛进行日光浴。$C, L \le 2500$。

按照 minSPF 递减的顺序把奶牛排序，依次考虑每头奶牛。

对于每头奶牛，扫描一遍所有的防晒霜，在这头奶牛能用（能用指的是该防晒霜的强度符合这头奶牛的范围，并且瓶数还有剩余）的防晒霜里找 SPF 值最大的使用。

以上算法的贪心策略是在满足条件的前提下每次选 SPF 最大的防晒霜。这个策略为什么是正确的呢？我们考虑这一步策略对后续其他奶牛的影响。每瓶防晒霜是否可用，会被 minSPF 与 maxSPF 两个条件限制。因为奶牛已被按照 minSPF 递减排序，所以每一个不低于当前奶牛 minSPF 值的防晒霜，都不会低于后面其他奶牛的 minSPF。也就是说，对于当前奶牛可用的任意两瓶防晒霜 x 与 y，如果 SPF[x] < SPF[y]，那么后面其他奶牛只可能出现“x, y 都能用”“x, y 都不能用”或者“x 能用，y 不能用”这三种情况之一，即 x 的适用性更广。因此，当前奶牛选择 maxSPF 较大的 y 去使用，把 x 留给后续的奶牛，未来的可达状态包含了选择 x 时未来的可达状态。

另外，每头奶牛对答案的贡献至多是 1。即使让当前这头奶牛放弃日光浴，留下防晒霜给后面的某一头奶牛用，对答案的贡献也不会变得更大。综上所述，尽量满足当前的奶牛，并选择 SPF 值尽量大的防晒霜是一个正确的贪心策略。

例题 **Stall Reservations** AcWing111

有 N 头牛在畜栏中吃草。每个畜栏在同一时间段只能提供给一头牛吃草，所以可能会需要多个畜栏。给定 N 头牛和每头牛开始吃草和结束吃草的时间，每头牛在给定时间段内会一直吃草，求需要的最小畜栏数目和每头牛对应的畜栏方案。$N \le 5 * 10^4$。

按照开始吃草的时间把牛排序。

维护一个数组 S，记录当前每个畜栏安排进去的最后一头牛，最初没有畜栏。

依次对于每头牛，扫描数组 S，找到任意一个畜栏，满足当前的牛开始吃草的时间不早于畜栏中最后一头牛结束吃草的时间。如果这样的畜栏不存在，则为其新建一个畜栏。

这个贪心算法的时间复杂度是 $O(N^2)$，请读者自行证明其正确性。我们可以用一个小根堆（STL priority_queue）维护每个畜栏最后一头牛结束吃草的时间，尝试把当前的牛安排在堆顶（结束时间最早）的畜栏中，时间复杂度可以降低到 $O(N \log N)$。

例题 **Radar Installation** AcWing112

校长想通过监控设备覆盖学校内的 N 座建筑物，每座建筑物被视作一个质点，在笛卡尔坐标系中给出它们的坐标 (x,y)，并且所有建筑物均处在 x 轴的上方。因为学校的供电和传输线路均沿 x 轴，所以监控设备只被允许建立在 x 轴上。每台监控设备的监控范围均为一个半径为 R 的圆形，圆心即为这台设备。现给出 N 座建筑物的坐标，问：最少需要几台这样的设备可以实现对所有建筑物的监控？$N \le 1000$。

对于 x 轴上方的每个建筑物，可以计算出 x 轴上的一段能管辖它的区间 $l[i]$~$r[i]$。问题转化为：给定 N 个区间，在 x 轴上放置最少的点，使每个区间包含至少一个点。

按照每个区间的左端点 $l[i]$ 从小到大排序，用一个变量维护已经安放的最后一台监控设备的坐标 pos，起初 pos 为负无穷。

依次考虑每个区间。如果当前区间 i 的左端点 $l[i]$ 大于最后一台监控设备的坐标 pos，则新增一台设备，令 $pos = r[i]$。否则就让最后一台已经安放的监控设备来管辖当前区间，并令 $pos = \min(r[i], pos)$。

依此类推，直至所有区间被管辖，输出安放的设备个数即可。

这个贪心算法可以用“决策包容性”来证明。

首先，对于每个区间 $l[i]$~$r[i]$，有两种选择：

1. 使用已有的监控设备管辖它；

2. 新建一台监控设备管辖它。

我们的贪心策略是，当选择 1 可行时，不会选择 2。选择 1 之后，未来可以在任意位置新建一台监控设备。而选择 2 则需要在 $l[i]$~$r[i]$ 之间新建设备，也就是说，第 1 项选择未来可能到达的状态包含了第 2 项选择未来可能到达的状态。

其次，在选择 1 之后，我们把上一台设备调整到 $\min(r[i], pos)$ 的位置，也就是在能管辖 i 的前提下尽量往后放，“尽量往后放”这个策略未来的可达状态显然也包含了“放在更靠前的位置”未来的可达状态。最后，因为所有区间已经按照 $l[i]$ 排序，所以这个调整不会影响到已经被管辖的区间，证毕。

例题 **国王游戏** AcWing114

恰逢 H 国国庆，国王邀请 n 位大臣来玩一个有奖游戏。首先，他让每位大臣在左、右手上面分别写下一个正整数，国王自己也在左、右手上各写一个正整数。然后让这 n

位大臣排成一排，国王站在队伍的最前面。排好队后，所有的大臣都会获得国王奖赏的若干金币，每位大臣获得的金币数分别是：排在该大臣前面的所有人的左手上的数的乘积除以他自己右手上的数，然后向下取整得到的结果。国王不希望某一个大臣获得特别多的奖赏，所以他想请你帮他重新安排一下队伍的顺序，使得获得奖赏最多的大臣，所获奖赏尽可能地少。注意，国王的位置始终在队伍的最前面。

按照每个大臣左、右手上的数的乘积从小到大排序，就是最优排队方案。这个贪心算法可以使用微扰（邻项交换）证明。

对于任意一种顺序，设 n 名大臣左、右手上的数分别是 $A[1]\sim A[n]$ 与 $B[1]\sim B[n]$，国王手里的数是 $A[0]$ 和 $B[0]$。

如果我们交换两个相邻的大臣 i 与 $i+1$，在交换前这两个大臣获得的奖励是：

$$\frac{1}{B[i]}*\prod_{j=0}^{i-1}A[j] \text{ 与 } \frac{1}{B[i+1]}*\prod_{j=0}^{i}A[j]$$

交换之后这两个大臣获得的奖励是：

$$\frac{1}{B[i+1]}*\prod_{j=0}^{i-1}A[j] \text{ 与 } \frac{A[i+1]}{B[i]}*\prod_{j=0}^{i-1}A[j]$$

其他大臣获得的奖励显然都不变，因此我们只需要比较上面两组式子最大值的变化。提取公因式 $\prod_{j=0}^{i-1}A[j]$ 后，实际上需要比较下面两个式子的大小关系：

$$\max\left(\frac{1}{B[i]},\frac{A[i]}{B[i+1]}\right) \quad \max\left(\frac{1}{B[i+1]},\frac{A[i+1]}{B[i]}\right)$$

两边同时乘上 $B[i]*B[i+1]$，变为比较：

$$\max(B[i+1],A[i]*B[i]) \quad \max(B[i],A[i+1]*B[i+1])$$

注意到大臣手上的数都是正整数，故 $B[i+1]\le A[i+1]*B[i+1]$，且 $A[i]*B[i]\ge B[i]$。

于是，当 $A[i]*B[i]\le A[i+1]*B[i+1]$ 时，左式 $\le$ 右式，交换前更优。当 $A[i]*B[i]\ge A[i+1]*B[i+1]$ 时，左式 $\ge$ 右式，交换后更优。也就是说，在任何局面下，减小逆序对数都不会造成整体结果变差，而增加逆序对数则不会使整体结果变好。

最后，根据冒泡排序的知识，任何一个序列都能通过邻项交换的方式变为有序序列。故当逆序对数为 0，即按上述方案排序时就是最优策略。

例题 **Color a Tree** AcWing115

一棵有 $n(1\le n\le 1000)$ 个节点的树，每个节点 $i(1\le i\le n)$ 都有一个权值 $A[i](1\le A[i]\le 1000)$。现在要把这棵树的节点全部染色，染色的规则是：根节点 R 可以随时被染色；对于其他节点，在被染色之前它的父节点必须已经染上了色。每次染色

的代价为 $T*A[i]$，其中 T 代表当前是第几次染色。求把这棵树染色的最小总代价。

有一个错误的贪心算法是“每一步在可以被染色的点里选权值最大的染色”，读者很容易构造出其反例——只要构造一棵树，让一个权值很小的节点下边有很多权值巨大的节点，另一个权值较大的节点却没有子节点。

不过从这个错误的贪心算法的思考中，我们可以提取出一个正确的性质：树中除根节点外权值最大的点，一定会在它的父节点被染色后立即染色。

于是我们可以确定的是，树中权值最大的点及其父节点的染色操作是连续进行的，我们可以把这两个点“合并起来”。合并得到的新点的权值设为这两个点的权值的平均值。

例如，有权值为 x,y,z 的三个点，我们已知 x 和 y 的染色操作是连续进行的，那么就有两种可能的染色方案：

1. 先染 x,y，再染 z，代价是 $x+2y+3z$；

2. 先染 z，再染 x,y，代价是 $z+2x+3y$。

我们主要关心这两个代价之间的大小关系，所以不妨把两个式子同时加上 $(z-y)$ 再除以 2，分别得到：

1. 代价 $(x+y)/2+2*z$；

2. 代价 $z+2*((x+y)/2)$。

这恰好就相当于有权值为 $(x+y)/2$ 和 z 的两个点的两种染色次序。换言之，下列两种情况的“最优染色次序”可以互相转化：

1. 权值为 x,y,z 的三个点；

2. 权值为 $(x+y)/2$ 和 z 的两个点。

进一步推进，我们可以得到一种“等效权值”的算法：记录每个点是由多少个点合并而成的，一个点的“等效权值”定义为：

$$\text{该点包含的原始权值总和} \div \text{该点包含的原始点数}$$

根据一开始提到的性质，我们不断在树中取“等效权值”最大的点 p，与其父节点 fa 合并。合并之前 p 与 fa 各自包含的点的染色顺序是已知的，我们就让 p 中第一个点排在 fa 中最后一个点之后紧接着被染色，把这个顺序保存在 p 与 fa 合并以后的点上。最终整棵树合并成一个点后，我们就按照这个点内保存的顺序在原始的树上把各个节点依次染色，计算出花费的总代价，即为所求。



0x08 总结与练习

在本章中，我们探讨了程序设计的一些基本算法思想。我们曾经多次提到过，任何问题都有其涉及的范围，称之为问题的“状态空间”。求解一个问题，就是在这个状态空间里的遍历与映射。递推与递归是遍历状态空间的两种基本形式。本章以及接下来的章节探讨的各种算法，则是对于“遍历顺序”“遍历过程中的决策方法”“状态空间中各状态之间的映射方式”给出的指导。枚举与模拟是按照问题的直接表述形式对状态空间进行朴素的遍历，搜索则是带有一定的选择性、决策性的遍历，贪心是在每步决策时采取局部最优策略的遍历，动态规划则是基于全局考量的分阶段、按维度、无重复遍历。二分、倍增以及与排序相关的一些算法会对状态空间实施划分、等价、代表、拼接等手段，直接降低遍历时需要面对的时空规模。

本章知识点

位运算

补码表示法，理解 C++无符号、有符号整数在计算机中的存储方式
各种按位运算，包括取位、置位、移位等，以及一些常见技巧
快速幂，64 位整数乘法
二进制状态压缩，使用二进制数对状态进行压缩、提取的方法
位运算不进位的特点以及各位之间的独立性

递推与递归

能想象问题“状态空间”，理解各种基本算法其实是对状态空间的遍历与映射
递推边界、目标、递推公式的发现与设计
理解递归思想、子问题、递归边界、回溯时还原现场
递归实现常见规模的状态空间的遍历
分治思想，对问题进行划分、递归、再合并
分形，主要练习对子问题的划分、提取、抽象
递归的机器实现，转化成非递归的通用方法

前缀和与差分

一维、二维前缀和的递推与应用

差分序列的定义与应用

二分

整数集合二分法、实数域二分法

单峰函数的三分法

二分答案，把求解转化为判定

排序

各种排序算法，插入/选择/冒泡/堆/归并/快速/计数/基数/桶排序

离散化

中位数相关问题，包括货仓选址、环形均分纸牌、动态维护中位数等

求第 k 大数的 $O(N)$ 算法

逆序对相关问题，使用归并排序求逆序对

倍增

序列上的倍增算法及其应用

RMQ-ST 算法

贪心

贪心思想及其证明手段，主要通过较多题目开拓视野、归纳总结

练习

1. 完成本章中所有例题的代码实现
2. The Pilots Brothers' Refrigerator

题目地址：AcWing116，提示：枚举/位运算

3. 占卜 DIY

题目地址：AcWing117，提示：模拟

4. Fractal

题目地址：AcWing118，提示：递归/分形

5. Raid

题目地址：AcWing119，提示：分治/平面最近点对

6. 防线

题目地址：AcWing120，提示：二分

7. Corral the Cows

题目地址：AcWing121，提示：二分/离散化/前缀和

8. 糖果传递

题目地址：AcWing122，提示：排序/中位数/环形均分纸牌

9. Soldiers

题目地址：AcWing123，提示：排序/中位数/货仓选址问题扩展

10. Number Base Conversion

题目地址：AcWing124，提示：高精度运算/进制转换

11. Cow Acrobats

题目地址：AcWing125，提示：贪心/邻项交换

12. To the Max

题目地址：AcWing126，提示：贪心

13. Task

题目地址：AcWing127，提示：贪心

0x10 基本数据结构

在本章中，我们将学习栈、队列、链表（包括邻接表）、Hash（哈希表）、字符串、Tric（字典树[①]）、二叉堆等数据结构。这些数据结构的设计思路来源于程序设计中的各种基本算法思想，同时又为我们进一步研究和实现更加深入的算法提供了工具基础。本章的前5节主要介绍线性数据结构，以及与其相关的灵活应用。本章的后2节初步涉及树形数据结构，并在第0x40章继续进行探讨。

① 树：是数据结构和图论中的基本概念。本书作为一本进阶级别的书籍，不再讲解这些入门内容。不熟悉的读者可参阅配套资源提供的参考资料。

0x11 栈

栈是一种“后进先出”的线性数据结构。栈只有一端能够进出元素，我们一般称这一端为栈顶，另一端为栈底。添加或删除栈中元素时，我们只能将其插入到栈顶（进栈），或者把栈顶元素从栈中取出（出栈）。

我们可以在 C++中用一个数组和一个变量（记录栈顶位置）来实现栈结构。在之前的章节我们也提到过，栈是机器实现递归（深度优先搜索）的基本结构。读者对于栈的基本用法应该已经比较熟悉，这里就不再赘述，而是把更多篇幅放在栈的应用上。

例题 Push, Pop, GetMin AcWing41

实现一个栈，支持 Push（入栈）、Pop（出栈并输出栈顶）和 GetMin（查询栈中最小的值）三个操作，要求时间复杂度均为 O(1)。

栈结构原本就支持 O(1) 的入栈、出栈操作，但不支持查询最小值的操作。一个比较直接的思路是，我们知道二叉堆（第 0x17 节）是一种支持插入、取出堆顶、查询最值的数据结构，如果在维护一个栈的同时再维护一个存储同样元素的二叉堆，就可以支持题目中要求的操作。然而，它的时间复杂度是 $O(\log N)$。如果我们只用一个变量记录最小值，当发生出栈操作时，如果最小值恰好被出栈，就无法得知新的最小值是什么。这启发我们使用一个线性结构来保存历史上每个时刻的最小值，这样就可以在出栈后进行还原。

我们建立两个栈，栈 A 存储原本的数据，栈 B 存储栈 A 中以栈底开头的每段数据的最小值，就像下面这样：

A: 9 2 1 5 3 0 2 ←

B: 9 2 1 1 1 0 0 ←

当执行 Push(x) 操作时，在 A 中插入 x，在 B 中插入 min(B的栈顶数据, x)。在执行 Pop 操作时，在 A、B 中分别弹出栈顶。在执行 GetMin 操作时，直接输出 B 的栈顶数据。每个操作的时间复杂度都是 O(1)。

例题 Editor AcWing128

维护一个整数序列的编辑器，有以下五种操作，操作总数不超过 10^6。



I x：在当前光标位置之后插入一个整数 x，插入以后光标移动到 x 之后；

D：删除光标之前的一个整数，即按下退格键 Backspace；

L：光标向左移动一个位置，即按下←键；

R：光标向右移动一个位置，即按下→键；

Q k：询问在位置 k 之前的最大前缀和，其中 k 不超过当前光标的位置。

本题的特殊点在于，I,D,L,R 四种操作都在光标位置处发生，并且操作完成后光标至多移动 1 个位置。根据这种“始终在序列中间某个指定位置进行修改”的性质，再联想到上一章我们动态维护中位数的“对顶堆”算法，我们不难想到一个类似的“对顶栈”做法。

建立两个栈，栈 A 存储从序列开头到当前光标位置的这一段子序列，栈 B 存储从当前光标位置到序列结尾的这一段子序列，二者都以光标所在的那一端作为栈顶。这两个栈合起来就保存了整个序列。因为查询操作的 k 不超过光标位置，所以我们用一个数组 f 维护栈 A 的前缀和的最大值即可。设 A 的栈顶位置下标是 p_A，sum 是序列 A 的前缀和数组。

对于 I x 操作：

1. 把 x 插入栈 A；
2. 更新 $sum[p_A] = sum[p_A - 1] + A[p_A]$；
3. 更新 $f[p_A] = \max(f[p_A - 1], sum[p_A])$。

对于 D 操作，把 A 的栈顶出栈。

对于 L 操作，弹出 A 的栈顶，插入到 B 中。

对于 R 操作：

1. 弹出 B 的栈顶，插入到 A 中；
2. 更新 $sum[p_A] = sum[p_A - 1] + A[p_A]$；
3. 更新 $f[p_A] = \max(f[p_A - 1], sum[p_A])$。

对于 Q k 询问，直接返回 $f[k]$。

通过这样两个“对顶栈”，我们在 $O(1)$ 的时间内实现了每种操作和询问。

例题 进出栈序列问题 AcWing129/AcWing130

给定 $1\sim N$ 这 N 个整数和一个无限大的栈，每个数都要进栈并出栈一次。如果进栈的顺序为 $1,2,\cdots,N$，那么可能的出栈序列有多少种？

方法一：搜索（枚举/递归），$O(2^N)$

一个很直观的想法是，面对任何一个状态我们只有两种选择：

1. 把下一个数进栈；
2. 把当前栈顶的数出栈（如果栈非空）。

根据上一章学到的知识，我们可以枚举每一步如何选择，用递归实现这个规模为 $O(2^N)$ 的枚举，这样就得到了所有可能的出栈序列方案。

方法二：递推，$O(N^2)$

本题只要求我们计算出可能出栈序列有多少种，并不关心具体方案，于是我们可以使用递推直接进行统计。设 S_N 表示进栈顺序为 $1,2,\cdots,N$ 时可能的出栈序列总数。根据递推的理论，我们需要想办法把问题分解成若干个类似的子问题。

考虑“1”这个数排在最终出栈序列中的位置，如果最后“1”这个数排在第 k 个，那么整个序列的进出栈过程就是：

1. 整数 1 入栈；
2. 整数 $2\sim k$ 这 $k-1$ 个数按某种顺序进出栈；
3. 整数 1 出栈，排在第 k 个；
4. 整数 $k+1\sim N$ 这 $N-k$ 个数按某种顺序进出栈。

于是整个问题就被“1”这个数划分成了“$k-1$ 个数进出栈”和“$N-k$ 个数进出栈”这两个子问题，得到递推公式：

$$S_N=\sum_{k=1}^{N}S_{k-1}*S_{N-k}$$

方法三：动态规划，$O(N^2)$

在任何一个时刻，我们实际上只关心有多少个数尚未入栈、有多少个数还在栈里，并做出一步合法的操作，并不关心这些数具体是哪些。因此，我们可以用 $F[i,j]$ 表示有 i 个数尚未进栈，目前有 j 个数在栈里，已经有 $n-i-j$ 个数出栈时的方案总数。

在最终状态下，即所有数已经出栈时，顺序已经确定，所以边界为：$F[0,0]=1$。

我们需要求出初始状态下，即所有数尚未进栈时，可以到达上述边界的方案总数，所以目标为：$F[N,0]$。

每一步的两种决策分别是“把一个数进栈”和“把栈顶的数出栈”，所以有公式：

$$F[i,j]=F[i-1,j+1]+F[i,j-1]$$

方法四：数学，$O(N)$

该问题等价于求第 N 项 Catalan 数，即 $C_{2N}^{N}/(N+1)$。我们将在第 0x36 节详细介绍。

表达式计算

栈的一大用处是做算术表达式的计算。算术表达式通常有前缀、中缀、后缀三种表示方法。

中缀表达式，是我们最常见的表达式，例如：$3*(1-2)$。



前缀表达式，又称波兰式，形如“op A B”，其中 op 是一个运算符，A,B 是另外两个前缀表达式。例如：* 3 − 12。

后缀表达式，又称逆波兰式，形如“A B op”，例如：12 − 3 *。

前缀和后缀表达式的值的定义是，先递归求出 A,B 的值，二者再做 op 运算的结果。这两种表达式不需要使用括号，其运算方案是唯一确定的。对于计算机来讲，它最容易理解后缀表达式，我们可以使用栈来 $O(N)$ 地求出它的值。

后缀表达式求值

1. 建立一个用于存数的栈，逐一扫描该后缀表达式中的元素。
   (1) 如果遇到一个数，则把该数入栈。
   (2) 如果遇到运算符，就取出栈顶的两个数进行计算，把结果入栈。
2. 扫描完成后，栈中恰好剩下一个数，就是该后缀表达式的值。

读者可以模拟 12 − 3 * 这个后缀表达式的计算过程进行体会。

如果想让计算机求解我们人类常用的中缀表达式的值，最快的办法就是把中缀表达式转化成后缀表达式，再使用上述方法求值。这个转化过程同样可以使用栈来 $O(N)$ 地完成。

中缀表达式转后缀表达式

1. 建立一个用于存运算符的栈，逐一扫描该中缀表达式中的元素。
   (1) 如果遇到一个数，输出该数。
   (2) 如果遇到左括号，把左括号入栈。
   (3) 如果遇到右括号，不断取出栈顶并输出，直到栈顶为左括号，然后把左括号出栈。
   (4) 如果遇到运算符，**只要栈顶符号的优先级不低于新符号，就不断取出栈顶并输出**，最后把新符号进栈。优先级为乘除>加减>左括号。
2. 依次取出并输出栈中的所有剩余符号，最终输出的序列就是一个与原中缀表达式等价的后缀表达式。

读者可以写一些稍微复杂的中缀表达式进行模拟，体会这个计算过程。

以上例子中包含的都是一位数，如果是多位数，并且表达式是使用字符串逐字符存储的，我们只需要稍加判断，把连续的一段数字看成一个数即可。

当然，我们也可以不转化成后缀表达式，而是使用递归法直接求解中缀表达式的值，时间复杂度为 $O(N^2)$。

中缀表达式的递归法求值

目标：求解中缀表达式 $S[1\sim N]$ 的值。

子问题：求解中缀表达式 S 的子区间表达式 $S[L\sim R]$ 的值。

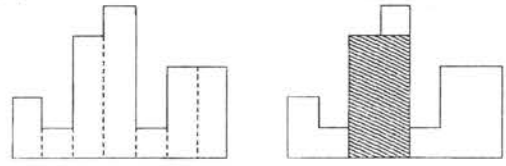
1. 在 $L\sim R$ 中考虑**没有被任何括号包含**的运算符。
   (1) 若存在加减号，选其中最后一个，分成左右两半递归，结果相加减，返回。
   (2) 若存在乘除号，选其中最后一个，分成左右两半递归，结果相乘除，返回。
2. 若不存在**没有被任何括号包含**的运算符。
   (1) 若首尾字符是括号，递归求解 $S[L+1\sim R-1]$，把结果返回。
   (2) 否则，说明区间 $S[L\sim R]$ 是一个数，直接返回数值。

读者可以用上面两种方法分别编程实现中缀表达式的求值。我们在配套资源中给出了这些方法的参考程序。

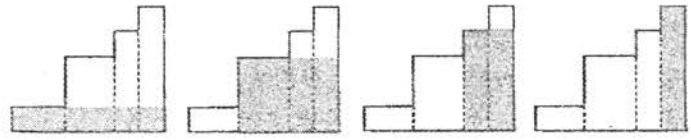
单调栈

例题 **Largest Rectangle in a Histogram** AcWing131

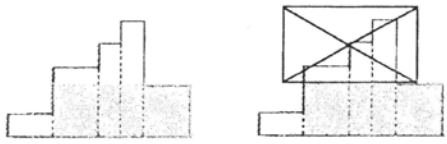
如下图所示，在一条水平线上方有若干个矩形，求包含于这些矩形的并集内部的最大矩形的面积（在下图中，答案就是阴影部分的面积），矩形个数 $\leq 10^5$。



我们先来思考这样一个问题：如果矩形的高度从左到右单调递增，那么答案是多少？显而易见，我们可以尝试以每个矩形的高度作为最终矩形的高度，并把宽度延伸到右边界，得到一个矩形，在所有这样的矩形面积中取最大值就是答案。如下图所示。



如果下一个矩形的高度比上一个小，那么该矩形想利用之前的矩形一起构成一块较大的面积时，这块面积的高度就不可能超过该矩形自己的高度。换句话说，在考虑完上图中的四种情况后，下图中打叉的那部分形状就没有丝毫用处了。



既然没有用处，为什么不把这些比该矩形高的矩形都删掉，用一个宽度累加、高度为该矩形自己的高度的新矩形（就是上图中的阴影部分）代替呢？这样并不会对后续的计算产生影响。于是我们维护的轮廓就变成了一个高度始终单调递增的矩形序列，问题变得可解了。

详细地说，我们建立一个栈，用来保存若干个矩形，这些矩形的高度是单调递增的。我们从左到右依次扫描每个矩形。

如果当前矩形比栈顶矩形高，直接进栈。

否则不断取出栈顶，直至栈为空或者栈顶矩形的高度比当前矩形小。在出栈过程中，我们累计被弹出的矩形的宽度之和，并且每弹出一个矩形，就用它的高度乘上累计的宽度去更新答案。整个出栈过程结束后，我们把一个高度为当前矩形高度、宽度为累计值的新矩形入栈。

整个扫描结束后，我们把栈中剩余的矩形依次弹出，按照与上面相同的方法更新答案。为了简化程序实现，也可以增加一个高度为 0 的矩形 $a[n+1]$，以避免在扫描结束后栈中有剩余矩形。

```
a[n + 1] = p = 0;
for (int i = 1; i <= n + 1; i++) {
    if (a[i] > s[p]) {
        s[++p] = a[i], w[p] = 1;
    } else {
        int width=0;
        while (s[p] > a[i]) {
            width += w[p];
            ans = max(ans, (long long)width * s[p]);
            p--;
        }
        s[++p] = a[i], w[p] = width + 1;
    }
}
```

这就是著名的单调栈算法，时间复杂度为 $O(N)$。借助单调性处理问题的思想在于**及时排除不可能的选项，保持策略集合的高度有效性和秩序性**，从而为我们做出决策提供更多的条件和可能的方法。

0x12 队 列

队列是一种“先进先出”的线性数据结构。一般来讲，元素从右端进入队列（入队），从左端离开队列（出队）。于是我们称队列的左端为队头，右端为队尾。可以在 C++ 中用一个数组和两个变量（记录队头和队尾的位置）来实现队列结构。C++ STL 中的 queue 也实现了一个队列。

元素进行多次入队、出队后，用于实现队列结构的数组的开头部分空间就会被严重浪费，所以我们经常将其优化为“循环队列”，也就是把队列看作一个首尾相接的环，只要队列中的元素个数在任意时刻都不超过环长，那么随着入队和出队操作的进行，存储元素的那一段位置就像沿着环不停地移动，重复利用着历史上曾被占用过的空间。

队列还有很多变体。例如两端都能插入或取出元素的双端队列（C++ STL deque），给每个元素附带一个评估值、出队时取出估值最大的、等价于一个二叉堆的优先队列（C++ STL priority_queue）。队列也是实现广度优先搜索的基本结构。对于队列的基本用法，读者应该已经比较熟悉，这里就不再赘述，而是把更多篇幅放在各种形式的队列的应用上。

例题 Team Queue AcWing132

有 n 个小组要进行排队，每个小组中有若干个人。当一个人来到队伍时，如果队伍中已经有自己小组的成员，他就直接插队排在自己小组成员的后面，否则就站在队伍的最后面。给定不超过 $2*10^5$ 个入队指令（编号为 X 的人来到队伍）和出队指令（队头的人出队），输出出队的顺序。

这是一道非常简单的题目，因为在任何时刻，同一个小组的人只要来到了队伍，就会站在一起，所以我们建立一个队列 Q_0 存储队伍中所有小组的编号，再为每个小组 i 建立一个队列 Q_i 存储队伍中这个小组的所有成员，一共 $n+1$ 个队列。

当一个编号为 X，组号为 Y 的人来到队伍时，我们直接把 X 插入 Q_Y 末尾。如果在插入之前 Q_Y 是空的，则还要把 Y 插入到 Q_0 末尾，表明队伍最后出现了一个新的小组。

当接收到出队指令时，我们通过 Q_0 得知排在最前面的小组组号 Y，然后再把 Q_Y 的队头出队。出队后如果 Q_Y 为空，就从 Q_0 开头删除 Y，表明这个小组目前所有人

已经离开。

例题 蚯蚓 AcWing133

蛐蛐国最近蚯蚓成灾了！隔壁跳蚤国的跳蚤也拿蚯蚓们没办法，蛐蛐国国王只好去请神刀手来帮他们消灭蚯蚓。

蛐蛐国里现在共有 $n(1 \le n \le 10^5)$ 只蚯蚓，第 i 只蚯蚓的长度为 a_i $(a_i \le 10^8)$，所有蚯蚓的长度都是非负整数，即可能存在长度为 0 的蚯蚓。

每一秒，神刀手会在所有的蚯蚓中，准确地找到最长的那一只，将其切成两段。若有多只最长的，则任选一只。神刀手切开蚯蚓的位置由有理数 p $(0 < p < 1)$ 决定。一只长度为 x 的蚯蚓会被切成两只长度分别为 $\lfloor px \rfloor$ 和 $x - \lfloor px \rfloor$ 的蚯蚓。特殊地，如果这两个数的其中一个等于 0，则这个长度为 0 的蚯蚓也会被保留。此外，除了刚刚产生的两只新蚯蚓，其余蚯蚓的长度都会增加一个非负整数 q $(q \le 200)$。

蛐蛐国国王知道这样不是长久之计，因为蚯蚓不仅会越来越多，还会越来越长。蛐蛐国国王决定求助于一位有着洪荒之力的神秘人物，但是救兵还需要 m $(m \le 7 * 10^6)$ 秒才能到来。

蛐蛐国国王希望知道这 m 秒内的战况。具体来说，他希望知道：

1. m 秒内，每一秒被切断的蚯蚓被切断前的长度，共有 m 个数；
2. m 秒后，所有蚯蚓的长度，共有 $n+m$ 个数。

如果 $q = 0$，即蚯蚓不会变长，那么本题相当于维护一个集合，支持查询最大值、删除最大值、插入新的值。这是二叉堆（第 0x17 节）或 STL set（第 0x71 节）能够完成的基本操作，时间复杂度约为 $O(m \log n)$。

当 $q > 0$ 时，除了最大值拆成的两个数之外，集合中的其他数都会增加 q。设最大值为 x。我们不妨认为产生了两个大小为 $\lfloor px \rfloor - q$ 和 $x - \lfloor px \rfloor - q$ 的新数，然后再把整个集合都加上 q。这与之前的操作是等价的。

于是，我们可以维护一个变量 $delta$ 表示整个集合的“偏移量”，集合中的数加上 $delta$ 是它的真实数值。起初，$delta = 0$。对于每一秒：

1. 取出集合中的最大值 x，令 $x = x + delta$；
2. 把 $\lfloor px \rfloor - delta - q$ 和 $x - \lfloor px \rfloor - delta - q$ 插入集合；
3. 令 $delta = delta + q$。

重复上述步骤 m 轮，即可得到最终集合中所有数的值。然而，本题中 m 的范围过大，我们需要一种线性算法来更快地求解。

注意到 p, q 是固定常数，$0 < p < 1$ 且 q 是非负整数。设 x_1, x_2 为非负整数，当 $x_1 \ge x_2$ 时，有 $\lfloor px_1 \rfloor + q = \lfloor px_1 + q \rfloor \ge \lfloor px_2 + pq \rfloor = \lfloor p(x_2 + q) \rfloor$。上式第一个等号成立的原因是整数可以自由移入、移出取整符号而不改变式子的值。

又因为 $x_1 - x_2 \geq p(x_1 - x_2)$，所以 $x_1 - px_1 \geq x_2 - px_2 \geq x_2 - p(x_2 + q)$。进一步有 $x_1 - \lfloor px_1 \rfloor + q = \lfloor x_1 - px_1 \rfloor + q \geq \lfloor x_2 - p(x_2 + q) \rfloor + q = x_2 + q - \lfloor p(x_2 + q) \rfloor$。

上面两段分析的意义是：若 x_1 在 x_2 之前被取出集合，则在一秒以后，x_1 分成的两个数 $\lfloor px_1 \rfloor + q$ 和 $x_1 - \lfloor px_1 \rfloor + q$ 分别不小于 $x_2 + q$ 分成的两个数 $\lfloor p(x_2 + q) \rfloor$ 和 $x_2 + q - \lfloor p(x_2 + q) \rfloor$。换言之，不仅从集合中取出的数是单调递减的，新产生的两类数值也分别随着时间单调递减。

我们可以建立三个队列 A, B, C，共同构成需要维护的集合。队列 A 保存初始的 n 个数，从大到小排序。队列 B 保存每秒新产生的 $\lfloor px \rfloor$ 那一段数值。队列 C 保存每秒新产生的 $x - \lfloor px \rfloor$ 那一段数值。起初队列 B, C 为空，新产生的数从队尾插入。根据之前的结论，B, C 单调递减。因此，每个时刻集合中最大的数就是队列 A, B, C 的三个队首之一。再配合集合的偏移量 $delta$，整个算法的时间复杂度为 $O(m + n \log n)$。

*** 例题** 双端队列 AcWing134

Sherry 现在碰到了一个棘手的问题，她需要对 $N \leq 2 * 10^5$ 个整数进行排序，但是她手头能用的工具只有若干个双端队列。

她必须依次处理这 N 个数，对于每个数 $A[i]$，Sherry 能做以下两件事之一：

1. 新建一个双端队列，并将 $A[i]$ 作为这个队列中的唯一的数；

2. 把 $A[i]$ 从已有队列的队头或队尾入队。

除此之外，对所有的数处理完成后，还要求这些队列能够按照一定的顺序连起来，得到一个非降的长度为 N 的序列。请问：Sherry 最少需要多少个双端队列？

这个问题很难直接求解，因为在不知道后面会有哪些数的情况下，我们做出的局部决策很可能造成某个数 P 与 Q 在同一个队列中，这样只要后面出现了介于 P 和 Q 之间的数，不管放到哪里都会导致无解（无法把队列连成一个有序序列）。这种情况启发我们，必须考虑按照数值的大小顺序而不是读入的顺序进行计算。

因为 Sherry 最后会把队列连成一个非降序列，所以我们不妨反过来思考，先把 N 个数 $A[1 \sim N]$ 从小到大排序，然后分成尽量少的几段，让每一段对应原问题中一个合法的双端队列。

每一段需要满足什么条件才能对应原问题中的双端队列呢？我们可以依次取出排序后的所有数在原始数组 A 中的下标，构成一个新的数组 B。

例如：样例数据 $A = [3\ 6\ 0\ 9\ 6\ 3]$，下标分别是 $[1\ 2\ 3\ 4\ 5\ 6]$。

排序后得到 $A' = [0\ 3\ 3\ 6\ 6\ 9]$，对应原数组下标 $B = [3\ 1\ 6\ 2\ 5\ 4]$。

经过分析可以发现，如果 B 中的一段满足单谷性质（先单调递减，后单调递增），那么这一段就可以对应为原问题的双端队列（B 中保存的是下标，这一段的谷点就相当于第一个入队，递减的一段相当于从队头插入，递增的一段相当于从队尾插入）。



还需要注意的一点是，如果 A 中存在几个相等的数，那么这几个数在排序时顺序是不定的，我们可以任意交换它们的顺序，使得 B 数组能够分成更少的段数。

所以我们最终的算法是，按照 A' 中数值的异同，把 B 看作若干个区间：

$A' = [[0], [3\ 3], [6\ 6], [9]]$，$B = [[3], [1\ 6], [2\ 5], [4]]$

一个区间一个区间地对 B 进行处理，最终拼成包含尽量少的段数的序列，其中每一段都具有单谷性质。我们可以用一个变量记录当前拼成的序列末尾处于递增状态还是递减状态，并用贪心策略尝试把当前区间递增或递减地接在序列末尾。以样例数据为例。

1. 第一个区间始终看作递减区间插入序列，当前序列为 [3]，正处于递减状态。

2. 第二个区间 [1 6]，因为 $6 > 3$，无法继续递减，只能以递增形式 [1 6] 插入序列，产生拐点，当前序列为 [3 1 6]，正处于递增状态。

3. 第三个区间 [2 5]，因为 $2 < 6$，无法继续递增，只能以递减形式 [5 2] 插入序列，产生拐点，单谷段数加 1。当前序列为 [3 1 6 5 2]，正处于递减状态。

4. 第四个区间 [4]，因为 $4 > 2$，无法继续递减，只能以递增形式 [4] 插入序列。

最终得到序列 [3 1 6 5 2 4]，包含 [3 1 6]，[5 2 4] 两个单谷段，所以对应到原问题，需要 2 个双端队列。

单调队列

例题 最大子序和 AcWing135

给定一个长度为 N 的整数序列（可能有负数），从中找出一段长度不超过 M 的连续子序列，使得子序列中所有数的和最大。$N, M \leq 3 * 10^5$。

计算“区间和”的问题，一般转化为“两个前缀和相减”的形式进行求解。我们先求出 $S[i]$ 表示序列里前 i 项的和，则连续子序列 $[L, R]$ 中数的和就等于 $S[R] - S[L-1]$。那么原问题可以转化为：找出两个位置 x, y，使 $S[y] - S[x]$ 最大并且 $y - x \leq M$。

首先我们枚举右端点 i，当 i 固定时，问题就变为：**找到一个左端点 j，其中 $j \in [i-m, i-1]$ 并且 $S[j]$ 最小。**

不妨比较一下任意两个位置 j 和 k，如果 $k < j < i$ 并且 $S[k] \geq S[j]$，那么对于所有大于等于 i 的右端点，k 永远不会成为最优选择。这是因为不但 $S[k]$ 不小于 $S[j]$，而且 j 离 i 更近，长度更不容易超过 M，即 j 的生存能力比 k 更强。所以当 j 出现后，k 就完全是一个无用的位置。

以上比较告诉我们，可能成为最优选择的策略集合一定是一个“**下标位置递增、对应的前缀和 S 的值也递增**”的序列。我们可以用一个队列保存这个序列。随着右端点变从前向后扫描，我们对每个 i 执行以下三个步骤。

1. 判断队头决策与 i 的距离是否超出 M 的范围，若超出则出队。

2. 此时队头就是右端点为 i 时，左端点 j 的最优选择。

3. 不断删除队尾决策，直到队尾对应的 S 值小于 $S[i]$。然后把 i 作为一个新的决策入队。

```
int l = 1, r = 1;
q[1] = 0;  // save choice j=0
for(int i = 1; i <= n; i++) {
   while (l <= r && q[l] < i - m) l++;  // step 1
   ans = max(ans, sum[i] - sum[q[l]]);  // step 2
   while (l <= r && sum[q[r]] >= sum[i]) r--;  // step 3
   q[++r] = i;
}
```

这就是著名的单调队列算法，因为每个元素至多入队一次、出队一次，所以整个算法的时间复杂度为 $O(N)$。它的思想也是**在决策集合（队列）中及时排除一定不是最优解的选择**。单调队列是优化动态规划的一个重要手段，我们在 0x59 节会详细讲解。



0x13 链表与邻接表

数组是一种支持随机访问，但不支持在任意位置插入或删除元素的数据结构。与之相对应，链表支持在任意位置插入或删除，但只能按顺序依次访问其中的元素。我们可以用一个 struct 表示链表的节点，其中可以存储任意数据。另外用 prev 和 next 两个指针指向前后相邻的两个节点，构成一个常见的双向链表结构。为了避免在左右两端或者空链表中访问越界，我们通常建立**额外的两个节点 head 与 tail 代表链表头尾**，把实际数据节点存储在 head 与 tail 之间，来减少链表边界处的判断，降低编程复杂度。

链表的正规形式一般通过动态分配内存、指针等实现，为了避免内存泄露、方便调试，使用数组模拟链表、下标模拟指针也是常见的做法。读者对于链表应该已经有所了解，这里就不再赘述。两种实现形式的参考程序：

```
struct Node {
    int value; // 数据
    Node *prev, *next; // 指针
};
Node *head, *tail;

void initialize() { // 建新链表
    head = new Node();
    tail = new Node();
    head->next = tail;
    tail->prev = head;
}

// 在 p 后插入包含数据 val 的新节点
void insert(Node *p, int val) {
    q = new Node();
    q->value = val;
    p->next->prev = q;
    q->next = p->next;
    p->next = q; q->prev = p;
}
```

```
struct Node {
    int value;
    int prev, next;
} node[SIZE];
int head, tail, tot;

int initialize() {
    tot = 2;
    head = 1, tail = 2;
    node[head].next = tail;
    node[tail].prev = head;
}

void insert(int p, int val) {
    q = ++tot;
    node[q].value = val;
    node[node[p].next].prev = q;
    node[q].next = node[p].next;
    node[p].next = q;
    node[q].prev = p;
}
```

```
void remove(Node *p) { // 删除 p
    p->prev->next = p->next;
    p->next->prev = p->prev;
    delete p;
}

void recycle() { // 链表内存回收
    while (head != tail) {
        head = head->next;
        delete head->prev;
    }
    delete tail;
}
```

```
void remove(int p) {
    node[node[p].prev].next
     = node[p].next;
    node[node[p].next].prev
     = node[p].prev;
}

void clear() { // 数组模拟链表清空
    memset(node, 0, sizeof(node))
    head = tail = tot = 0;
}
```

例题 邻值查找 AcWing136

给定一个长度为 n 的序列 A，A 中的数各不相同。对于 A 中的每一个数 A_i，求：

$$\min_{1\le j<i}|A_i - A_j|$$

以及令上式取到最小值的 j（记为 P_i）。若最小值点不唯一，则选择 A_j 较小的。

数据范围：$n \le 10^5, |A_i| \le 10^9$。

解法一：平衡树（STL set）

把 $A_1, A_2, \cdots, A_n$ 依次插入一个集合，则在插入 A_i 之前，集合中保存的就是满足 $1 \le j < i$ 的所有 A_j。根据题意，我们只需在集合中查找与 A_i 最接近的值。

若能维护一个有序集合，则集合中与 A_i 最接近的值要么是 A_i 的前驱（排在它前一名的值），要么是 A_i 的后继（排在它后一名的值），比较前驱、后继与 A_i 的差即可。

而二叉平衡树（第 0x46 节）就是一个支持动态插入、查询前驱以及查询后继的数据结构。在 C++中，STL set（第 0x71 节）也为我们提供了这些功能。

解法二：链表

把序列 A 从小到大排序，然后依次串成一个链表。注意在排序的同时，建立一个数组 B，其中 B_i 表示原始序列中的 A_i 处于链表中的哪个位置（一个指针）。

因为链表有序，所以在链表中，指针 B_n 指向的节点的 prev 和 next 就分别是 A_n 的前驱和后继。通过比较二者与 A_n 的差，我们就能求出与 A_n 最接近的值。

接下来，我们在链表中删除 B_n 指向的节点，该操作是 O(1) 的。

此时，我们按同样方法考虑 B_{n-1} 的 prev 和 next，再删除 B_{n-1}。

依此类推，最终即可求出与每个 A_i 最接近的值。

两种解法的时间复杂度都是 $O(n\log n)$，其中第二种解法的瓶颈在于排序。

例题 **Running Median** AcWing106

动态维护中位数问题：依次读入一个整数序列，每当已经读入的整数个数为奇数时，输出已读入的整数构成的序列的中位数。

在 0x05 节中，我们讨论了本题的“对顶堆”做法。这里我们再介绍一种使用链表的离线做法。我们可以先把整个序列读入进来，排序之后依次插入一个链表，此时我们可以知道整个序列的中位数 P。随后，我们倒序考虑本题中的读入过程，也就是把整数从链表中一个个删去。

与上一道题一样，为了删除一个整数，我们需要找到该整数在链表中对应的节点。上一题建立了一个指针数组，但是本题的整数范围较大，需要用 Hash 表（第 0x14 节）或者 STL map（第 0x71 节）记录这些指针。

每次删去一个整数 X 后，要么中位数不变，要么新的中位数与原来中位数 P 的位置相邻，通过 X 和 P 的大小关系以及链表中元素的总个数分情况讨论，很容易确定新的中位数 P。

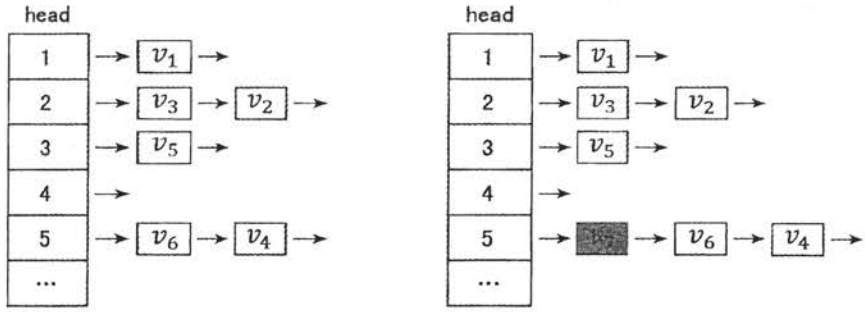
在上面两道例题中，我们都用了“倒序处理”的方法，利用链表容易执行“删除”操作的特性，快速地找到了答案。

邻接表

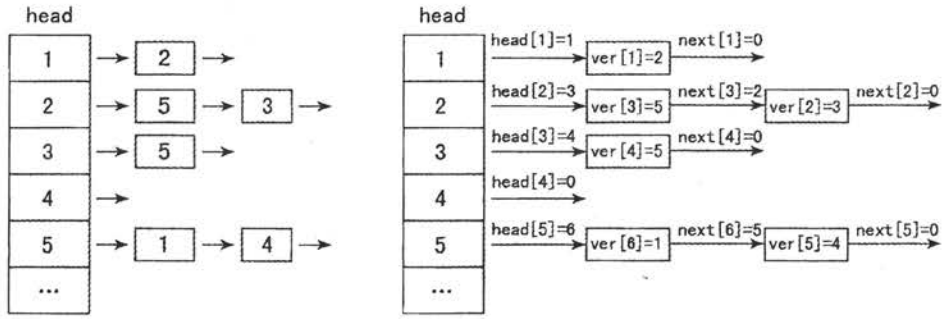
在与链表相关的诸多结构中，邻接表是相当重要的一个。它是树与图结构的一般化存储方式，还能用于实现我们在下一节中即将介绍的开散列 Hash 表。实际上，邻接表可以看成“带有索引数组的多个数据链表”构成的结构集合。在这样的结构中存储的数据被分成若干类，每一类的数据构成一个链表。每一类还有一个代表元素，称为该类对应链表的“表头”。所有“表头”构成一个表头数组，作为一个可以随机访问的索引，从而可以通过表头数组定位到某一类数据对应的链表。为了方便起见，本书将这类结构统称为“邻接表”结构。

如下图左侧所示，这是一个存储了 6 个数据节点 v_1~v_6 的邻接表结构。这 6 个数据节点被分成 4 类，通过表头数组 head 可以定位到每一类所构成的链表进行遍历访问。

当需要插入新的数据节点时，我们可以通过表头数组 head 定位到新的数据节点所属类别的链表表头，将新数据在表头位置插入。如下图右侧所示，在邻接表中插入了属于第 5 类的新数据节点 v_7。



在一个具有 n 个点 m 条边的有向图结构中，我们可以把每条边所属的“类别”定义为该边的起点标号。这样所有边被分成 n 类，其中第 x 类就由“从 x 出发的所有边”构成。通过表头 head[x]，我们很容易定位到第 x 类对应的链表，从而访问从点 x 出发的所有边。



上图是在邻接表中插入一张 5 个点、6 条边的有向图之后的状态。这 6 条边按照插入的顺序依次是 (1,2), (2,3), (2,5), (3,5), (5,4), (5,1)。上图左侧展示了这个邻接表存储的宏观信息，上图右侧展示了采用“数组模拟链表”的实现方式时，内部数据的实际存储方式。head 与 next 数组中保存的是“ver[①]数组的下标”，相当于指针，其中 0 表示指向空。ver 数组存储的是每条边的终点，是真实的图数据。

```
// 加入有向边(x, y)，权值为 z
void add(int x, int y, int z) {
    ver[++tot] = y, edge[tot] = z; // 真实数据
    next[tot] = head[x], head[x] = tot; // 在表头 x 处插入
}
```

① ver：英文 vertex（顶点）的缩写。



```
// 访问从 x 出发的所有边
for (int i = head[x]; i; i = next[i]) {
    int y = ver[i], z = edge[i];
    // 找到了一条有向边(x, y)，权值为 z
}
```

上面的代码片段用数组模拟链表的方式存储了一张带权有向图的邻接表结构。

对于无向图，我们把每条无向边看作两条有向边插入即可。有一个小技巧是，结合在第 0x01 节提到的“成对变换”的位运算性质，我们可以在程序最开始时，初始化变量 $tot = 1$。这样每条无向边看成的两条有向边会成对存储在 ver 和 edge 数组的下标“2 和 3”“4 和 5”“6 和 7”……的位置上。通过对下标进行 xor 1 的运算，就可以直接定位到与当前边反向的边。换句话说，如果 ver[i] 是第 i 条边的终点，那么 ver[i xor 1] 就是第 i 条边的起点。

0x14 Hash

Hash 表

Hash 表又称为散列表，一般由 Hash 函数（散列函数）与链表结构共同实现。与离散化思想类似，当我们要对若干复杂信息进行统计时，可以用 Hash 函数把这些复杂信息映射到一个容易维护的值域内。因为值域变简单、范围变小，有可能造成两个不同的原始信息被 Hash 函数映射为相同的值，所以我们需要处理这种冲突情况。

有一种称为“开散列”的解决方案是，建立一个邻接表结构，以 Hash 函数的值域作为表头数组 head，映射后的值相同的原始信息被分到同一类，构成一个链表接在对应的表头之后，链表的节点上可以保存原始信息和一些统计数据。

Hash 表主要包括两个基本操作：

1. 计算 Hash 函数的值；

2. 定位到对应链表中依次遍历、比较。

无论是检查任意一个给定的原始信息在 Hash 表中是否存在，还是更新它在 Hash 表中的统计数据，都需要基于这两个基本操作进行。

当 Hash 函数设计较好时，原始信息会被比较均匀地分配到各个表头之后，从而使每次查找、统计的时间降低到“原始信息总数除以表头数组长度”。若原始信息总数与表头数组长度都是 $O(N)$ 级别且 Hash 函数分散均匀，几乎不产生冲突，那么每次查找、统计的时间复杂度期望为 $O(1)$。

例如，我们要在一个长度为 N 的随机整数序列 A 中统计每个数出现了多少次。当数列 A 中的值都比较小时，我们可以直接用一个数组计数（**建立一个大小等于值域的数组进行统计和映射，其实就是最简单的 Hash 思想**）。当数列 A 中的值很大时，我们可以把 A 排序后扫描统计。这里我们换一种思路，尝试一下 Hash 表的做法。

设计 Hash 函数为 $H(x) = (x \bmod P) + 1$，其中 P 是一个比较大的质数，但不超过 N。显然，这个 Hash 函数把数列 A 分成 P 类，我们可以依次考虑数列中的每个数 $A[i]$，定位到 $\text{head}[H(A[i])]$ 这个表头所指向的链表。如果该链表中不包含 $A[i]$，我们就在表头后插入一个新节点 $A[i]$，并在该节点上记录 $A[i]$ 出现了 1 次，否则我们就直接找到已经存在的 $A[i]$ 节点将其出现次数加 1。因为整数序列 A 是随机的，所

以最终所有的 $A[i]$ 会比较均匀地分散在各个表头之后，整个算法的时间复杂度可以近似达到 $O(N)$。

上面的例子是一个非常简单的 Hash 表的直观应用。对于非随机的数列，我们可以设计更好的 Hash 函数来保证其时间复杂度。同样地，如果我们需要维护的是比大整数复杂得多的信息的某些性质（如是否存在、出现次数等），也可以用 Hash 表来解决。字符串就是一种比较一般化的信息，在本节的后半部分，我们将会介绍一个程序设计竞赛中极其常用的字符串 Hash 算法。

例题 Snowflake Snow Snowflakes AcWing137

有 N 片雪花，每片雪花由六个角组成，每个角都有长度。第 i 片雪花六个角的长度从某个角开始顺时针依次记为 $a_{i,1}, a_{i,2}, \cdots, a_{i,6}$。

因为雪花的形状是封闭的环形，所以从任何一个角开始顺时针或逆时针往后记录长度，得到的六元组都代表形状相同的雪花。例如：$a_{i,1}, a_{i,2}, \cdots, a_{i,6}$ 和 $a_{i,2}, a_{i,3}, \cdots, a_{i,6}, a_{i,1}$ 就是形状相同的雪花；$a_{i,1}, a_{i,2}, \cdots, a_{i,6}$ 和 $a_{i,6}, a_{i,5}, \cdots, a_{i,1}$ 也是形状相同的雪花。

我们称两片雪花形状相同，当且仅当它们各自从某一个角开始顺时针或逆时针记录长度，能得到两个相同的六元组。求这 N 片雪花中是否存在两片形状相同的雪花。

定义 Hash 函数 $H(a_{i,1}, a_{i,2}, \cdots, a_{i,6}) = \left(\sum_{j=1}^{6} a_{i,j} + \prod_{j=1}^{6} a_{i,j}\right) \bmod P$，其中 P 是我们自己选取的一个较大的质数。显然，对于两片形状相同的雪花，它们六个角的长度之和、长度之积都相等，因此它们的 Hash 函数值也相等。

建立一个 Hash 表，把 N 片雪花依次插入。对于每片雪花 $a_{i,1}, a_{i,2}, \cdots, a_{i,6}$，我们直接扫描表头 $H(a_{i,1}, a_{i,2}, \cdots, a_{i,6})$ 对应的链表，检查是否存在与 $a_{i,1}, a_{i,2}, \cdots, a_{i,6}$ 形状相同的雪花即可。对于随机数据，期望的时间复杂度为 $O(N^2/P)$；取 P 为最接近 N 的质数，期望的时间复杂度为 $O(N)$。在下一节中，我们将学习循环同构串的“最小表示法”，进一步提高判断两片雪花形状是否相同的效率。

```
const int SIZE = 100010;
int n, tot, P = 99991, snow[SIZE][6], head[SIZE], next[SIZE];
int H(int *a) {
    int sum = 0, mul = 1;
    for (int i = 0; i < 6; i++) {
        sum = (sum + a[i]) % P;
        mul = (long long)mul * a[i] % P;
    }
    return (sum + mul) % P;
}
```

```
bool equal(int *a, int *b) {
    for (int i = 0; i < 6; i++)
        for (int j = 0; j < 6; j++) {
            bool eq = 1;
            for (int k = 0; k < 6; k++)
                if (a[(i+k)%6] != b[(j+k)%6]) eq = 0;
            if (eq) return 1;
            eq = 1;
            for (int k = 0; k < 6; k++)
                if (a[(i+k)%6] != b[(j-k+6)%6]) eq = 0;
            if (eq) return 1;
        }
    return 0;
}

bool insert(int *a) {
    int val = H(a);
    // 遍历表头 head[val]指向的链表，寻找形状相同的雪花
    for (int i = head[val]; i; i = next[i]) {
        if (equal(snow[i], a)) return 1;
    }
    // 未找到形状相同的雪花，执行插入
    ++tot;
    memcpy(snow[tot], a, 6 * sizeof(int));
    next[tot] = head[val];
    head[val] = tot;
    return 0;
}

int main() {
    cin >> n;
    for (int i = 1; i <= n; i++) {
        int a[10];
        for (int j = 0; j < 6; j++) scanf("%d", &a[j]);
        if (insert(a)) {
            puts("Twin snowflakes found.");
            return 0;
```

```
            }
        }
        puts("No two snowflakes are alike.");
    }
```

字符串 Hash

下面介绍的字符串 Hash 函数把一个任意长度的字符串映射成一个非负整数，并且其冲突概率几乎为零。

取一固定值 P，把字符串看作 P 进制数，并分配一个大于 0 的数值，代表每种字符。一般来说，我们分配的数值都远小于 P。例如，对于小写字母构成的字符串，可以令 $a=1,b=2,\cdots,z=26$。**取一固定值 M，求出该 P 进制数对 M 的余数，作为该字符串的 Hash 值。**

一般来说，我们取 $P=131$ 或 $P=13331$，此时 Hash 值产生冲突的概率极低，只要 Hash 值相同，我们就可以认为原字符串是相等的。通常我们取 $M=2^{64}$，即**直接使用 unsigned long long 类型**存储这个 Hash 值，在计算时不处理算术溢出问题，产生溢出时相当于自动对 2^{64} 取模，这样可以避免低效的取模 (mod) 运算。

除了在极特殊构造的数据上，上述 Hash 算法很难产生冲突，一般情况下上述 Hash 算法完全可以出现在题目的标准解答中。我们还可以多取一些恰当的 P 和 M 的值（例如大质数），多进行几组 Hash 运算，当结果都相同时才认为原字符串相等，就更加难以构造出使这个 Hash 产生错误的数据。

对字符串的各种操作，都可以直接对 P 进制数进行算术运算反映到 Hash 值上。

如果我们已知字符串 S 的 Hash 值为 $H(S)$，那么在 S 后添加一个字符 c 构成的新字符串 $S+c$ 的 Hash 值就是 $H(S+c)=(H(S)*P+value[c]) \bmod M$。其中乘 P 就相当于 P 进制下的左移运算，$value[c]$ 是我们为 c 选定的代表数值。

如果我们已知字符串 S 的 Hash 值为 $H(S)$，字符串 $S+T$ 的 Hash 值为 $H(S+T)$，那么字符串 T 的 Hash 值 $H(T)=\left(H(S+T)-H(S)*P^{\text{length}(T)}\right) \bmod M$。这就相当于通过 P 进制下在 S 后边补 0 的方式，把 S 左移到与 $S+T$ 的左端对齐，然后二者相减就得到了 $H(T)$。

例如，$S=$ "abc"，$c=$ "d"，$T=$ "xyz"，则：

S 表示为 P 进制数：1 2 3

$H(S)=1*P^2+2*P+3$

$H(S+c)=1*P^3+2*P^2+3*P+4=H(S)*P+4$

$S+T$ 表示为 P 进制数：1 2 3 24 25 26

$H(S+T)=1*P^5+2*P^4+3*P^3+24*P^2+25*P+26$

S 在 P 进制下左移 length(T) 位：1 2 3 0 0 0

二者相减就是 T 表示为 P 进制数：24 25 26

$H(T) = H(S+T) - (1*P^2 + 2*P + 3)*P^3 = 24*P^2 + 25*P + 26$

根据上面两种操作，我们可以通过 O(N) 的时间预处理字符串所有前缀 Hash 值，并在 O(1) 的时间内查询它的任意子串的 Hash 值。

例题 兔子与兔子 AcWing138

很久很久以前，森林里住着一群兔子。有一天，兔子们想要研究自己的 DNA 序列。我们首先选取一个好长好长的 DNA 序列（小兔子是外星生物，DNA 序列可能包含 26 个小写英文字母），然后我们每次选择两个区间，询问如果用两个区间里的 DNA 序列分别生产出来两只兔子，这两只兔子是否一模一样。注意两只兔子一模一样只可能是它们的 DNA 序列一模一样。$1 \le \text{length}(S), Q \le 10^6$。

记我们选取的 DNA 序列为 S，根据我们刚才提到的字符串 Hash 算法，设 $F[i]$ 表示前缀子串 $S[1 \sim i]$ 的 Hash 值，有 $F[i] = F[i-1]*131 + (S[i] - \text{"}a\text{"} + 1)$。

于是可以得到任一区间 $[l,r]$ 的 Hash 值为 $F[r] - F[l-1]*131^{r-l+1}$。当两个区间的 Hash 值相同时，我们就认为对应的两个子串相等。整个算法的时间复杂度为 O($|S| + Q$)。

```
char s[1000010];
unsigned long long f[1000010], p[1000010];
int main() {
    scanf("%s", s + 1);
    int n = strlen(s + 1), q; cin >> q;
    p[0] = 1; // 131^0
    for (int i = 1; i <= n; i++) {
        f[i] = f[i-1] * 131 + (s[i]-'a'+1); // hash of 1~i
        p[i] = p[i-1] * 131; // 131^i
    }
    for (int i = 1; i <= q; i++) {
        int l1, r1, l2, r2;
        scanf("%d%d%d%d", &l1, &r1, &l2, &r2);
        if (f[r1] - f[l1-1] * p[r1-l1+1] == // hash of l1~r1
            f[r2] - f[l2-1] * p[r2-l2+1]) { // hash of l2~r2
            puts("Yes");
        } else {
            puts("No");
        }
```

```
    }
}
```

例题 **Palindrome** AcWing139

如果一个字符串正着读和倒着读是一样的，则称它是回文的。给定一个长度为 N 的字符串 S，求它的最长回文子串。

写几个回文串观察它们的性质，我们可以发现回文串分为两类。

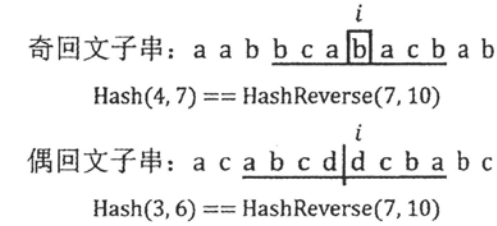
奇回文串 $A[1\sim M]$，长度 M 为奇数，并且 $A[1\sim M/2+1]=\text{reverse}(A[M/2+1\sim M])$，它的中心点是一个字符。其中 $\text{reverse}(A)$ 表示把字符串 A 倒过来。

偶回文串 $B[1\sim M]$，长度 M 为偶数，且 $B[1\sim M/2]=\text{reverse}(B[M/2+1\sim M])$，它的中心点是两个字符之间的夹缝。

于是在本题中，我们可以枚举 S 的回文子串的中心位置 $i=1\sim N$，看从这个中心位置出发向左右两侧最长可以扩展出多长的回文串。也就是说：

1. 求出一个最大的整数 p 使得 $S[i-p\sim i]=\text{reverse}(S[i\sim i+p])$，那么以 i 为中心的最长奇回文子串的长度就是 $2*p+1$；

2. 求出一个最大的整数 q 使得 $S[i-q\sim i-1]=\text{reverse}(S[i\sim i+q-1])$，那么以 $i-1$ 和 i 之间的夹缝为中心的最长偶回文子串的长度就是 $2*q$。



根据上一道题目，我们已经知道在 $O(N)$ 预处理前缀 Hash 值后，可以 $O(1)$ 计算任意子串的 Hash 值。类似地，我们可以倒着做一遍预处理，就可以 $O(1)$ 计算任意子串倒着读的 Hash 值。于是我们可以对 p 和 q 进行二分答案，用 Hash 值 $O(1)$ 比较一个正着读的子串和一个倒着读的子串是否相等，从而在 $O(\log N)$ 时间内求出最大的 p 和 q。在枚举过的所有中心位置对应的奇、偶回文子串长度中取 max 就是整道题目的答案，时间复杂度为 $O(N\log N)$。

有一个名为 Manacher 的算法可以 $O(N)$ 求解该问题，感兴趣的读者可以自行查阅相关资料。

例题 后缀数组 AcWing140

后缀数组 (SA)[①] 是一种重要的数据结构，通常使用倍增或者 DC3 算法实现，这超出了我们的讨论范围。在本题中，我们希望使用快排、Hash 与二分实现一个简单的 $O(n\log^2 n)$ 的后缀数组求法。详细地说，给定一个长度为 n 的字符串 S（下标 $0\sim n-1$），我们可以用整数 $k(0\le k<n)$ 表示字符串 S 的后缀 $S(k\sim n-1)$。把字符串 S 的所有后缀按照字典序排列，排名为 i 的后缀记为 SA[i]。额外地，我们考虑排名为 i 的后缀与排名为 $i-1$ 的后缀，把二者的最长公共前缀的长度记为 Height[i]。我们的任务就是求出 SA 与 Height 这两个数组。

S 的所有后缀的总长度在 $O(n^2)$ 级别，如果我们直接对这 n 个后缀进行快排，对于两个字符串的大小比较采取逐字符扫描的方式，最坏情况下时间复杂度将达到 $O(n^2\log n)$。

在上一道题目中，我们已经知道如何求出一个字符串的所有前缀 Hash 值，并进一步在O(1) 的时间内查询任意一个区间子串的 Hash 值。所以在快速排序需要比较两个后缀 p 和 q 时，我们就可以使用二分法，每次二分时利用 Hash 值 O(1) 地比较 $S[p\sim p+mid-1]$ 与 $S[q\sim q+mid-1]$ 这两段是否相等，最终求出 $S[p\sim n]$ 与 $S[q\sim n]$ 的最长公共前缀的长度，记为 len。于是 $S[p+len]$ 和 $S[q+len]$ 就是这两个后缀第一个不相等的位置，直接比较这两个字符的大小就可以确定 $S[p\sim n]$ 与 $S[q\sim n]$ 的大小关系。从而每次比较的总复杂度就是 $O(\log n)$，整个快排求出 SA 数组的过程的复杂度就是 $O(n\log^2 n)$。在排序完成后，我们对于每对排名相邻的后缀执行与上述相同的二分过程，就可以求出 Height 数组了。配套资源提供了参考程序。

① SA：英文 Suffix Array（后缀数组）的缩写。



0x15 字符串

KMP 模式匹配

KMP 算法，又称模式匹配算法，能够在线性时间内判定字符串 $A[1\sim N]$ 是否为字符串$B[1\sim M]$ 的子串，并求出字符串 A 在字符串 B 中各次出现的位置。

首先，一个 $O(NM)$ 的朴素做法是，尝试枚举字符串 B 中的每个位置 i，把字符串 A 与字符串 B 的后缀 $B[i\sim M]$ 对齐，向后扫描逐一比较 $A[1]$ 与 $B[i]$，$A[2]$ 与 $B[i+1]$…是否相等。我们把这种比较过程称为 A 与 B 尝试进行“匹配”。

其次，这个问题使用字符串 Hash 也能在线性时间内求解。通过上一节中的“兔子与兔子”这道例题，我们已经知道：可以在 $O(N)$ 的时间内预处理一个字符串的所有前缀 Hash 值，并在 $O(1)$ 的时间内查询该字符串任意一个子串的 Hash 值。所以，一个很直接的想法是：枚举字符串 B 中的每个位置 $i(N\le i\le M)$，检查字符串 A 的 Hash 值与字符串 B 的子串 $A[i-N+1\sim i]$ 的 Hash 值是否相同，即可得到该问题的解。

KMP 算法能够更高效、更准确地处理这个问题，并且能为我们提供一些额外的信息。详细地讲，KMP 算法分为两步。

1. 对字符串 A 进行自我“匹配”，求出一个数组 $next$，其中 $next[i]$ 表示“A 中以 i 结尾的非前缀子串”与“A 的前缀”能够匹配的最长长度，即：

$$next[i]=\max\{j\},\ 其中\ j<i\ 并且\ A[i-j+1\sim i]=A[1\sim j]$$

特别地，当不存在这样的 j 时，令 $next[i]=0$。

2. 对字符串 A 与 B 进行匹配，求出一个数组 f，其中 $f[i]$ 表示“B 中以 i 结尾的子串”与“A 的前缀”能够匹配的最长长度，即：

$$f[i]=\max\{j\},\ 其中\ j\le i\ 并且\ B[i-j+1\sim i]=A[1\sim j]$$

下面讨论 $next$ 数组的计算方法。根据定义，$next[1]=0$。接下来我们按照 $i=2\sim N$ 的顺序依次计算 $next[i]$。

假设 $next[1\sim i-1]$ 已经计算完毕，当计算 $next[i]$ 时，根据定义，我们需要找出所有满足 $j<i$ 且 $A[i-j+1\sim i]=A[1\sim j]$ 的整数 j 并取最大值。为了叙述方便，我们称满足这两个条件的 j 为 $next[i]$ 的“候选项”。

使用朴素算法计算 *next* 数组

朴素的做法是枚举 $j \in [1, i-1]$，并检查 $A[i-j+1 \sim i]$ 与 $A[1 \sim j]$ 是否相等。以下图中的字符串为例，当 $i=7$ 时，需要枚举下列 6 种情况。

$A[2 \sim 7]$ = "bababa"，它与前缀 $A[1 \sim 6]$ = "ababab" 不相等。

$A[3 \sim 7]$ = "ababa"，它与前缀 $A[1 \sim 5]$ = "ababa" 相等，长度为 5。

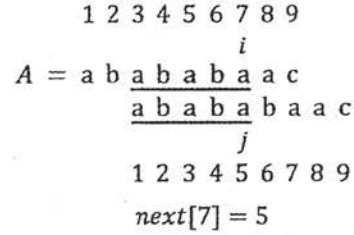
$A[4 \sim 7]$ = "baba"，它与前缀 $A[1 \sim 4]$ = "abab" 不相等。

$A[5 \sim 7]$ = "aba"，它与前缀 $A[1 \sim 3]$ = "aba" 相等，长度为 3。

$A[6 \sim 7]$ = "ba"，它与前缀 $A[1 \sim 2]$ = "ab" 不相等。

$A[7 \sim 7]$ = "a"，它与前缀 $A[1 \sim 1]$ = "a" 相等，长度为 1。

所以，以 $i=7$ 结尾，最多与 A 的前缀匹配到 5，$next[7]=5$。



该算法对每个 i 枚举了 $i-1$ 个非前缀子串，并检查与对应前缀的匹配情况，时间复杂度不会低于 $O(N^2)$。能否更快地求出 *next* 呢？

引理

若 j_0 是 $next[i]$ 的一个“候选项”，即 $j_0 < i$ 且 $A[i-j_0+1 \sim i] = A[1 \sim j_0]$，则小于 j_0 的最大的 $next[i]$ 的“候选项”是 $next[j_0]$。换言之，$next[j_0]+1 \sim j_0-1$ 之间的数都不是 $next[i]$ 的“候选项”。

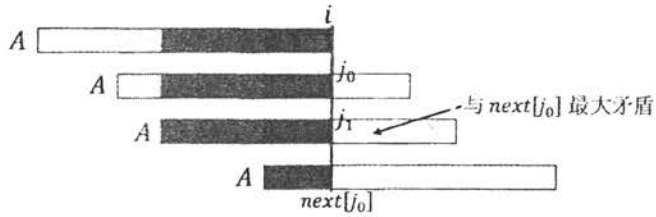
证明：

反证法。假设存在 $next[j_0] < j_1 < j_0$ 使得 j_1 为 $next[i]$ 的“候选项”，即 $A[i-j_1+1] = A[1 \sim j_1]$。如下页图所示，颜色相同的阴影部分代表相等的子串。

分别取 $A[i-j_0+1 \sim i]$ 和 $A[1 \sim j_0]$ 的后 j_1 个字符，显然也相等，即 $A[i-j_1+1 \sim i] = A[j_0-j_1+1 \sim j_0]$。从而 $A[j_0-j_1+1 \sim j_0] = A[1 \sim j_1]$，与 $next[j_0]$ “最大”的性质矛盾。故假设不成立。

另一方面，$next[j_0]$ 显然是 $next[i]$ 的“候选项”。

证毕。



使用优化的算法计算 $next$ 数组

根据引理，当 $next[i-1]$ 计算完毕时，我们即可得知 $next[i-1]$ 的所有“候选项”从大到小依次是 $next[i-1]$，$next[next[i-1]]$，$next\big[next[next[i-1]]\big]$ … 而如果一个整数 j 是 $next[i]$ 的“候选项”，那么 $j-1$ 显然也必须是 $next[i-1]$ 的“候选项”（两个字符串 $A[i-j+1{\sim}i]$ 和 $A[1{\sim}j]$ 相等的前提是 $A[i-j+1\sim i-1]$ 和 $A[1{\sim}j-1]$ 相等）。因此，在计算 $next[i]$ 时，只需把 $next[i-1]+1$，$next[next[i-1]]+1$，$next\big[next[next[i-1]]\big]+1$ … 作为 j 的选项即可。

仍然使用朴素算法中的例子。假设 $next[1{\sim}6]$ 已经被求出。按照定义，$next[6]=4$，即 $A[3{\sim}6]$ 与 $A[1{\sim}4]$ 匹配。

接下来 $A[7]=A[5]=$ "a"，在该字符上能够继续匹配，由 $next[6]$ 匹配长度的最优性可知，在字符 7 处继续匹配，$next[7]=5$ 就是最优解。

我们继续考虑 $next[8]$。此时发现 $A[8]=$ "a" 与 $A[6]=$ "b" 二者不相等，不能继续把匹配长度从 5 增长到 6。我们只好把匹配长度缩短。由引理可知，以 $i=7$ 为结尾的匹配长度除了 $j=next[7]=5$ 之外，还有 $next[5]=3$ 和 $next[3]=1$。我们依次尝试这两种稍短一些的匹配能否延伸到 $i=8$。

然而，$A[8]$ 与 $A[4]$，$A[8]$ 与 $A[2]$ 都不相等，无法延伸，所以我们只能让 $i=8$ 从 A 字符串开头重新开始匹配，$A[8]=A[1]$，匹配长度为 1，$next[8]=1$。

```
      1 2 3 4 5 6 7 8 9
              i
A = a b a b a b a c
    a b a b a b a a c          Fail  (next[7] = 5)
      a b a b a b a a c        Fail  (next[5] = 3)
        a b a b a b a a c      Fail  (next[3] = 1)
          a b a b a b a a c    Match (next[1] = 0)
          j
          1 2 3 4 5 6 7 8 9
```

在讨论优化后的算法的时间复杂度之前，我们先写出算法实现的框架和代码。

KMP 算法 *next* 数组的求法

1. 初始化 $next[1]=j=0$，假设 $next[1\sim i-1]$ 已求出，下面求解 $next[i]$。

2. 不断尝试扩展匹配长度 j，如果扩展失败（下一个字符不相等），令 j 变为 $next[j]$，直至 j 为 0（应该重新从头开始匹配）。

3. 如果能够扩展成功，匹配长度 j 就增加 1。$next[i]$ 的值就是 j。

```
next[1] = 0;
for (int i = 2, j = 0; i <= n; i++) {
    while (j > 0 && a[i] != a[j+1]) j = next[j];
    if (a[i] == a[j+1]) j++;
    next[i] = j;
}
```

因为定义的相似性，求解 f 与求解 $next$ 的过程是基本一致的。

KMP 算法 f 数组的求法

```
for (int i = 1, j = 0; i <= m; i++) {
    while (j > 0 && (j == n || b[i] != a[j+1])) j = next[j];
    if (b[i] == a[j+1]) j++;
    f[i] = j;
    // if (f[i] == n)，此时就是A在B中的某一次出现
}
```

这就是 KMP 模式匹配算法。在上面代码的 while 循环中，j 的值不断减小，$j=next[j]$ 的执行次数不会超过每层 for 循环开始时 j 的值与 while 循环结束时 j 的值之差。而在每层 for 循环中，j 的值至多增加 1。因为 j 始终非负，所以在整个计算过程中，j 减小的幅度总和不会超过 j 增加的幅度总和。故 j 的总变化次数至多为 $2(N+M)$。整个算法的时间复杂度为 $O(N+M)$。

例题 Period AcWing141

如果一个字符串 S 是由一个字符串 T 重复 K 次形成的，则称 T 是 S 的循环元。使 K 最大的字符串 T 称为 S 的最小循环元，此时的 K 称为最大循环次数。

现在给定一个长度为 N 的字符串 S，对 S 的每一个前缀 $S[1\sim i]$，如果它的最大循环次数大于 1，则输出该前缀的最小循环元长度和最大循环次数。

对字符串 S 自匹配求出 $next$ 数组，根据定义，对于每个 i，$S[i-next[i]+1\sim i]$ 与 $S[1\sim next[i]]$ 是相等的，并且不存在更大的 $next$ 值满足这个条件。



引理

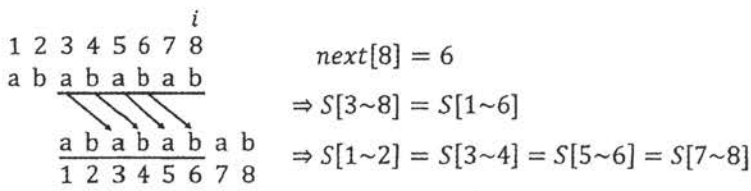
$S[1\sim i]$ 具有长度为 $len < i$ 的循环元的充要条件是 len 能整除 i 并且 $S[len+1\sim i] = S[1\sim i-len]$（即 $i-len$ 是 $next[i]$ 的“候选项”，在讲解 KMP 算法时我们提到了“候选项”的含义）。

证明：

先证必要性。设 $S[1\sim i]$ 具有长度为 len 的循环元，显然 len 能整除 i，并且 $S[1\sim i-len]$ 和 $S[len+1\sim i]$ 都是由 $i/len-1$ 个循环元构成的。故 $S[1\sim i-len] = S[len+1\sim i]$。

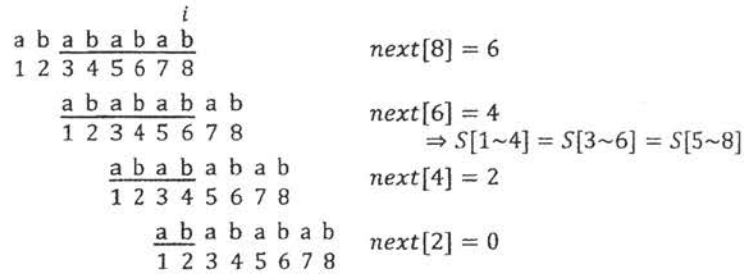
再证充分性。设 len 能整除 i 并且 $S[len+1\sim i] = S[1\sim i-len]$。因为 $len < i$，所以 $S[1\sim i-len]$ 和 $S[len+1\sim i]$ 的长度不小于 len 且是 len 的倍数。二者各取前 len 个字符，有 $S[1\sim len] = S[len+1\sim 2*len]$。依此类推，不断向后取 len 个字符，可以发现 $S[1\sim i-len]$ 和 $S[len+1\sim i]$ 是以 len 为间隔错位对齐的，如下图所示。故 $S[1\sim len]$ 是 S 的循环元。

证毕。



根据引理，当 $i-next[i]$ 能整除 i 时，$S[1\sim i-next[i]]$ 就是 $S[1\sim i]$ 的最小循环元。它的最大循环次数就是 $i/(i-next[i])$。其中 $i-next[i]$ 能整除 i 的条件是为了保证循环元每次重复的完整性。例如上图中 $i=7$ 时，最小循环元 "ab" 的第 4 次重复就尚未完成。

进一步地，如果 $i-next[next[i]]$ 能整除 i，那么 $S[1\sim i-next[next[i]]]$ 就是 $S[1\sim i]$ 的次小循环元。依此类推，我们还可以找出 $S[1\sim i]$ 所有可能的循环元。



值得指出的一个性质是，一个字符串的任意循环元的长度必然是最小循环元长度的倍数，证明留作读者思考。

```
char a[1000010];
int next[1000010], n, T;
void calc_next() {
    next[1] = 0;
    for (int i = 2, j = 0; i <= n; i++) {
        while (j > 0 && a[i] != a[j+1]) j = next[j];
        if (a[i] == a[j+1]) j++;
        next[i] = j;
    }
}
int main() {
    while (cin >> n && n) {
        scanf("%s", a + 1);
        calc_next();
        printf("Test case #%d\n", ++T);
        for (int i = 2; i <= n; i++) {
            if (i % (i - next[i]) == 0 && i / (i - next[i]) > 1)
                printf("%d %d\n", i, i / (i - next[i]));
        }
        puts("");
    }
}
```

最小表示法

给定一个字符串 $S[1\sim n]$，如果我们不断把它的最后一个字符放到开头，最终会得到 n 个字符串，称这 n 个字符串是循环同构的。这些字符串中字典序最小的一个，称为字符串 S 的最小表示。

例如 S = "abca"，那么它的 4 个循环同构字符串为 "abca","aabc","caab","bcaa"，S 的最小表示为 "aabc"。与 S 循环同构的字符串可以用该字符串在 S 中的起始下标表示，因此我们用 $B[i]$ 来表示从 i 开始的循环同构字符串，即 $S[i\sim n]+S[1\sim i-1]$。

如何求出一个字符串的最小表示？最朴素的方法是：按照定义，依次比较这 n 个循环同构的字符串，找到其中字典序最小的一个。比较两个循环同构字符串 $B[i]$ 与 $B[j]$ 时，我们也采用直接向后扫描的方式，依次取 $k=0,1,2,\cdots$，比较 $B[i+k]$ 与 $B[j+k]$ 是否相等，直至找到一个不相等的位置，从而确定 $B[i]$ 与 $B[j]$ 的大小关系。

实际上，一个字符串的最小表示可以在 $O(n)$ 的线性时间内求出。我们首先把 S 复制一份接在它的结尾，得到的字符串记为 SS。显然，$B[i]=SS[i\sim i+n-1]$。



对于任意的 i,j，我们仔细观察 $B[i]$ 与 $B[j]$ 的比较过程：

$$S = \text{"bacacabc"}, i = 2, j = 4, k = 3$$

```
   i     i+k
b a c a c a b c b a c a c a b c
b a c a c a b c b a c a c a b c
    j     j+k
```

如果在 $i+k$ 与 $j+k$ 处发现不相等，假设 $SS[i+k] > SS[j+k]$，那么我们当然可以得知 $B[i]$ 不是 S 的最小表示（因为存在一个更小的循环同构串 $B[j]$）。除此之外，我们还可以得知 $B[i+1], B[i+2], \cdots, B[i+k]$ 也都不是 S 的最小表示。这是因为对于 $1 \le p \le k$，存在一个比 $B[i+p]$ 更小的循环同构串 $B[j+p]$（从 $i+p$ 与 $j+p$ 开始向后扫描，同样会在 $p=k$ 时发现不相等，并且 $SS[i+k] > SS[j+k]$）。

同理，如果 $SS[i+k] < SS[j+k]$，那么 $B[j], B[j+1], \cdots, B[j+k]$ 都不是 S 的最小表示，直接跳过这些位置，一定不会遗漏最小表示。

最小表示法

1. 初始化 $i=1$，$j=2$。
2. 通过直接向后扫描的方法，比较 $B[i]$ 与 $B[j]$ 两个循环同构串。
   (1) 如果扫描了 n 个字符后仍然相等，说明 S 有更小的循环元（例如 catcat 有循环元 cat），并且该循环元已扫描完成，$B[\min(i,j)]$ 即为最小表示，算法结束。
   (2) 如果在 $i+k$ 与 $j+k$ 处发现不相等：
   若 $SS[i+k] > SS[j+k]$，令 $i = i+k+1$。若此时 $i=j$，再令 $i = i+1$。
   若 $SS[i+k] < SS[j+k]$，令 $j = j+k+1$。若此时 $i=j$，再令 $j = j+1$。
3. 若 $i > n$ 或 $j > n$，则 $B[\min(i,j)]$ 为最小表示；否则重复第 2 步。

该算法通过两个指针不断向后移动的形式，尝试比较每两个循环同构串的大小。利用上面的性质，及时排除掉不可能的选项。当其中一个移动到结尾时，就考虑过了所有可能的二元组 $(B[i], B[j])$，从而得到了最小表示。

如果每次比较向后扫描了 k 的长度，则 i 或 j 二者之一会向后移动 k，而 i 和 j 合计一共最多向后移动 $2n$ 的长度，因此该算法的复杂度为 $O(n)$。

```
int n = strlen(s + 1);
for (int i = 1; i <= n; i++) s[n+i] = s[i];
int i = 1, j = 2, k;
while (i <= n && j <= n) {
```

```
        for (k = 0; k < n && s[i+k] == s[j+k]; k++);
        if (k == n) break; // s 形如"catcat", 它的循环元已扫描完成
        if (s[i+k] > s[j+k]) {
            i = i + k + 1;
            if (i == j) i++;
        } else {
            j = j + k + 1;
            if (i == j) j++;
        }
    }
    ans = min(i, j); // B[ans]是最小表示
```



Trie（字典树）是一种用于实现字符串快速检索的多叉树结构。Trie 的每个节点都拥有若干个字符指针，若在插入或检索字符串时扫描到一个字符 c，就沿着当前节点的 c 字符指针，走向该指针指向的节点。下面我们来详细讨论 Trie 的基本操作过程。

初始化

一棵空 Trie 仅包含一个根节点，该点的字符指针均指向空。

插入

当需要插入一个字符串 S 时，我们令一个指针 P 起初指向根节点。然后，依次扫描 S 中的每个字符 c。

1. 若 P 的 c 字符指针指向一个已经存在的节点 Q，则令 $P = Q$。

2. 若 P 的 c 字符指针指向空，则新建一个节点 Q，令 P 的 c 字符指针指向 Q，然后令 $P = Q$。

当 S 中的字符扫描完毕时，在当前节点 P 上标记它是一个字符串的末尾。

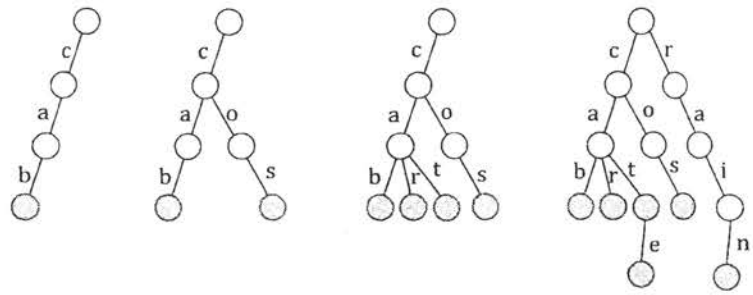
检索

当需要检索一个字符串 S 在 Trie 中是否存在时，我们令一个指针 P 起初指向根节点，然后依次扫描 S 中的每个字符 c：

1. 若 P 的 c 字符指针指向空，则说明 S 没有被插入过 Trie，结束检索。

2. 若 P 的 c 字符指针指向一个已经存在的节点 Q，则令 $P = Q$。

当 S 中的字符扫描完毕时，若当前节点 P 被标记为一个字符串的末尾，则说明 S 在 Trie 中存在，否则说明 S 没有被插入过 Trie。



在上页图所示的例子中，需要插入和检索的字符串都由小写字母构成，所以 Trie 的每个节点具有 26 个字符指针，分别为 a 到 z。上图展示了在一棵空 Trie 中依次插入“cab”“cos”“car”“cat”“cate”和“rain”后的 Trie 的形态，灰色标记了单词的末尾节点。可以看出在 Trie 中，字符数据都体现在树的边（指针）上，树的节点仅保存一些额外信息，例如单词结尾标记等。其空间复杂度是 $O(NC)$，其中 N 是节点个数，C 是字符集的大小。

```
int trie[SIZE][26], tot = 1; // 初始化，假设字符串由小写字母构成
void insert(char* str) { // 插入一个字符串
    int len = strlen(str), p = 1;
    for (int k = 0; k < len; k++) {
        int ch = str[k]-'a';
        if (trie[p][ch] == 0) trie[p][ch] = ++tot;
        p = trie[p][ch];
    }
    end[p] = true;
}
bool search(char* str) { // 检索字符串是否存在
    int len = strlen(str), p = 1;
    for (int k = 0; k < len; k++) {
        p = trie[p][str[k]-'a'];
        if (p == 0) return false;
    }
    return end[p];
}
```

例题 前缀统计 AcWing142

给定 N 个字符串 $S_1,S_2,\cdots,S_N$。接下来进行 M 次询问，每次询问给定一个字符串 T，求 S_1~S_N 中有多少个字符串是 T 的前缀。输入字符串的总长度不超过 10^6。

把这 N 个字符串插入一棵 Trie 树，Trie 树的每个节点上存储一个整数 cnt，记录该节点是多少个字符串的末尾节点。（为了处理插入重复字符串的情况，这里要记录个数，而不能只做结尾标记）

对于每个询问，在 Trie 树中检索 T，在检索过程中累加途经的每个节点的 cnt 值，就是该询问的答案。

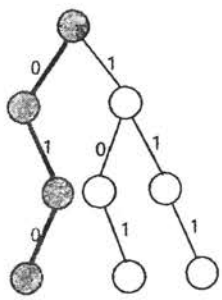
例题 The XOR Largest Pair AcWing143

在给定的 N 个整数 $A_1, A_2, \cdots, A_N$ 中选出两个进行 xor（异或）运算，得到的结果最大是多少？$N \le 10^5$，$0 \le A_i < 2^{31}$。

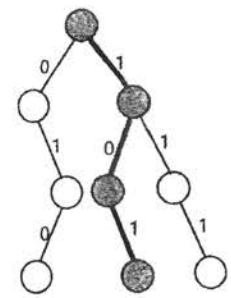
我们考虑所有的二元组 (i,j) 且 $i<j$，那么本题的目标就是在其中找到 A_i xor A_j 的最大值。也就是说，对于每个 $i(1 \le i \le N)$，我们希望找到一个 $j(1 \le j < i)$，使 A_i xor A_j 最大，并求出这个最大值。

我们可以把每个整数看作长度为 31 的二进制 01 串（数值较小时在前边补 0），并且把 $A_1 \sim A_{i-1}$ 对应的 31 位二进制串插入一棵 Trie 树（其中最低二进制位为叶子节点）。接下来，对于 A_i 对应的 31 位二进制串，我们在 Trie 中进行一次与检索类似的过程，每一步都尝试沿着“与 A_i 的当前位相反的字符指针”向下访问。若“与 A_i 的当前位相反的字符指针”指向空节点，则只好访问与 A_i 当前位相同的字符指针。根据 xor 运算“相同得 0，不同得 1”的性质，该方法即可找出与 A_i 做 xor 运算结果最大的 A_j。

如下图所示，在一棵插入了 2(010), 5(101), 7(111) 三个数的 Trie 中，分别查询与 6(110), 3(011) 做 xor 运算结果最大的数。（为了简便，图中使用了 3 位二进制数代替 31 位二进制数）



查询与 110 做 xor 运算结果最大的数



查询与 011 做 xor 运算结果最大的数

综上所述，我们可以循环 $i=1 \sim N$，对于每个 i，Trie 树中应该存储了 $A_1 \sim A_{i-1}$ 对应的 31 位二进制串（实际上每次 i 增长前，把 A_i 插入 Trie 即可）。根据我们刚才提到的“尽量走相反的字符指针”的检索策略，就可以找到所求的 A_j，更新答案。

例题 The XOR Longest Path AcWing144

给定一棵 N 个节点的树，树上的每条边都有一个权值。从树中选择两个点 x 和 y，把从 x 到 y 的路径上的所有边权 xor（异或）起来，得到的结果最大是多少？$N \le 10^5$。

设 $D[x]$ 表示根节点到 x 的路径上所有边权的 xor 值，显然有：

$$D[x] = D[\text{father}(x)] \text{ xor } \text{weight}(x, \text{father}(x))$$

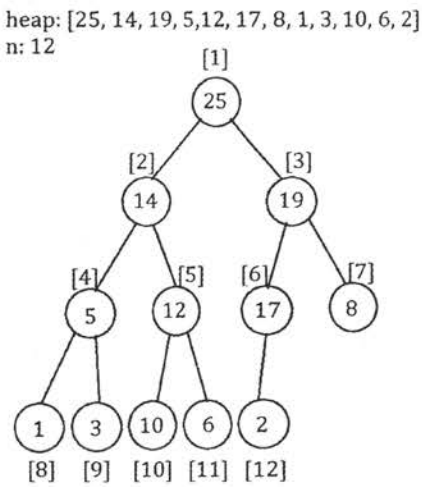
根据上式，我们可以对树进行一次深度优先遍历，求出所有的 $D[x]$。不难发现，树上 x 到 y 的路径上所有边权的 xor 结果就等于 $D[x]$ xor $D[y]$。这是因为根据 xor 运算的性质（a xor $a = 0$），“x 到根”和“y 到根”这两条路径重叠的部分恰好抵消掉。

所以，问题就变成了从 $D[1]$~$D[N]$ 这 N 个数中选出两个，xor 的结果最大，即上一道例题。可以用 Trie 树来快速求解。

0x17 二叉堆

二叉堆是一种支持插入、删除、查询最值的数据结构。它其实是一棵满足“堆性质”的完全二叉树[①]，树上的每个节点带有一个权值。若树中的任意一个节点的权值都小于等于其父节点的权值，则称该二叉树满足“大根堆性质”。若树中任意一个节点的权值都大于等于其父节点的权值，则称该二叉树满足“小根堆性质”。满足“大根堆性质”的完全二叉树就是“大根堆”，而满足“小根堆性质”的完全二叉树就是“小根堆”，二者都是二叉堆的形态之一。

根据完全二叉树的性质，我们可以采用层次序列存储方式，直接用一个数组来保存二叉堆。层次序列存储方式，就是逐层从左到右为树中的节点依次编号，把此编号作为节点在数组中存储的位置（下标）。在这种存储方式中，父节点编号等于子节点编号除以 2，左子节点编号等于父节点编号乘 2，右子节点编号等于父节点编号乘 2 加 1，如下图所示。

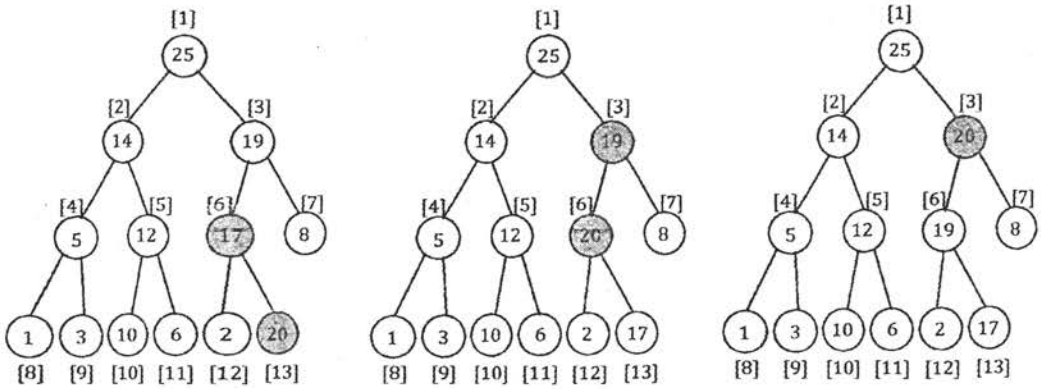


我们以大根堆为例探讨堆支持的几种常见操作的实现。

① 完全二叉树：叶子节点都在最后两层，且在最后一层集中于左侧的二叉树。本页图就是一棵完全二叉树。

Insert

Insert(val) 操作向二叉堆中插入一个带有权值 val 的新节点。我们把这个新节点直接放在存储二叉堆的数组末尾，然后通过交换的方式向上调整，直至满足堆性质。其时间复杂度为堆的深度，即 $O(\log N)$。




```
int heap[SIZE], n;
void up(int p) { // 向上调整
    while (p > 1) {
        if (heap[p]>heap[p/2]) { // 子节点>父节点，不满足大根堆性质
            swap(heap[p], heap[p/2]);
            p/=2;
        }
        else break;
    }
}
void Insert(int val) {
    heap[++n] = val;
    up(n);
}
```

GetTop

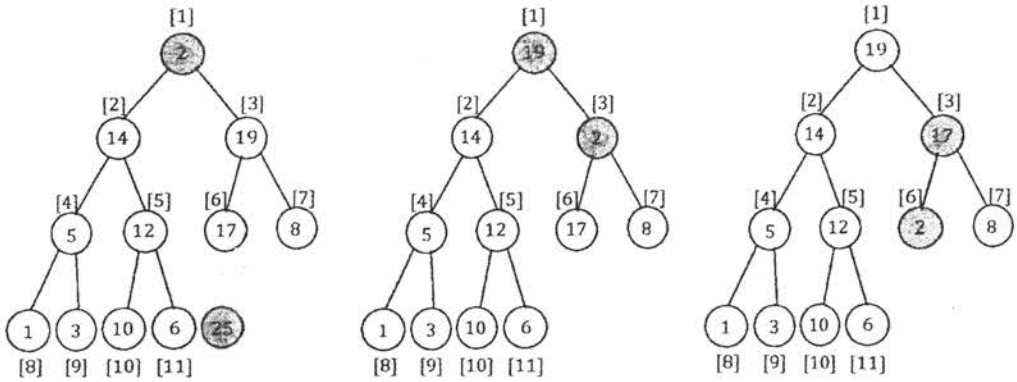
GetTop 操作返回二叉堆的堆顶权值，即最大值 $heap[1]$，复杂度为 $O(1)$。

```
int GetTop() {
    return heap[1];
}
```

Extract

Extract 操作把堆顶从二叉堆中移除。我们把堆顶 $heap[1]$ 与存储在数组末尾的

节点 $heap[n]$ 交换，然后移除数组末尾节点（令 n 减小 1），最后把堆顶通过交换的方式向下调整，直至满足堆性质。其时间复杂度为堆的深度，即 $O(\log N)$。




```
void down(int p) {
    int s = p*2; // p 的左子节点
    while (s <= n) {
        if (s<n && heap[s]<heap[s+1]) s++; // 左右子节点中取较大者
        if (heap[s]>heap[p]) { // 子节点大于父节点，不满足大根堆性质
            swap(heap[s], heap[p]);
            p = s, s = p*2;
        }
        else break;
    }
}
void Extract() {
    heap[1] = heap[n--];
    down(1);
}
```

Remove

Remove(p) 操作把存储在数组下标 p 位置的节点从二叉堆中删除。与 Extract 相类似，我们先把 $heap[p]$ 与 $heap[n]$ 交换，然后令 n 减小 1。注意此时 $heap[p]$ 既有可能需要向下调整，也有可能需要向上调整，需要分别进行检查和处理。时间复杂度为 $O(\log N)$。

```
void Remove(int k) {
    heap[k] = heap[n--];
    up(k), down(k);
}
```

C++ STL 中的 priority_queue（优先队列）为实现一个大根堆，支持 push（Insert），top（GetTop）和 pop（Extract）操作，不支持 Remove 操作，详细用法参见第 0x71 节。

例题 **Supermarket** AcWing145

给定 N 个商品，每个商品有利润 p_i 和过期时间 d_i，每天只能卖一个商品，过期商品不能再卖，求如何安排每天卖的商品，可以使收益最大。$1 \le N, p_i, d_i \le 10^4$。

容易想到一个贪心策略：在最优解中，对于每个时间（天数）t，应该在保证不卖出过期商品的前提下，尽量卖出利润前 t 大的商品。因此，我们可以依次考虑每个商品，动态维护一个满足上述性质的方案。

详细地说，我们把商品按照过期时间排序，建立一个初始为空的小根堆（节点权值为商品利润），然后扫描每个商品。

1. 若当前商品的过期时间（天数）t 等于当前堆中的商品个数，则说明在目前方案下，前 t 天已经安排了 t 个商品卖出。此时，若当前商品的利润大于堆顶权值（即已经安排的 t 个商品中的最低利润），则替换掉堆顶（用当前商品替换掉原方案中利润最低的商品）。

2. 若当前商品的过期时间（天数）大于当前堆中的商品个数，直接把该商品插入堆。

最终，堆里的所有商品就是我们需要卖出的商品，它们的利润之和就是答案。该算法的时间复杂度为 $O(N \log N)$。

例题 **Sequence** AcWing146

给定 M 个长度为 N 的序列，从每个序列中任取一个数求和，可以构成 N^M 个和，求其中最小的 N 个和。$N \le 2000$，$M \le 100$。

先来考虑当 $M = 2$ 时的简化问题，即从 2 个序列中任取一个数相加构成的 N^2 个和中求出前 N 小的和。设这两个序列为 A 和 B，把它们分别排序。

可以发现，最小和一定是 $A[1] + B[1]$，次小和是 $\min(A[1] + B[2], A[2] + B[1])$。假设次小和是 $A[2] + B[1]$，那么第 3 小和就是 $A[1] + B[2]$，$A[2] + B[2]$，$A[3] + B[1]$ 三者之一。也就是说，当确定 $A[i] + B[j]$ 为第 k 小和后，$A[i+1] + B[j]$ 与 $A[i] + B[j+1]$ 就加入了第 $k+1$ 小和的备选答案集合。读者可以类比有两个指针分别指向 $A[i]$ 和 $B[j]$，把其中一个指针向后移动一位，就可能产生下一个和。

需要注意的是，$A[1] + B[2]$ 与 $A[2] + B[1]$ 都能产生 $A[2] + B[2]$ 这个备选答案。为了避免重复，我们可以规定：如果把 j 加 1 产生新的备选答案，那么以后只能再增加 j，不能再增加 i。也就是说，从 $A[1] + B[1]$ 到任何 $A[i] + B[j]$ 必须先向后移动指向 A 的指针 i，再向后移动指向 B 的指针 j，这样就可以保证备选答案产生路线的

唯一性。

我们建立一个小根堆，堆中每个节点存储一个三元组 $(i,j,last)$，其中 $last$ 表示上一次移动的指针是不是 j，堆的比较操作以 $A[i]+B[j]$ 作为节点权值。

起初，堆中只有 (1,1,false)。

取出堆顶 $(i,j,last)$，然后把 $(i,j+1,\text{true})$ 插入堆，如果 $last$ 为 false，再把 $(i+1,j,\text{false})$ 也插入堆。

重复上一步 N 次，每次取出的堆顶节点的权值 $A[i]+B[j]$ 一起构成前 N 小和。该算法的时间复杂度为 $O(N\log N)$。

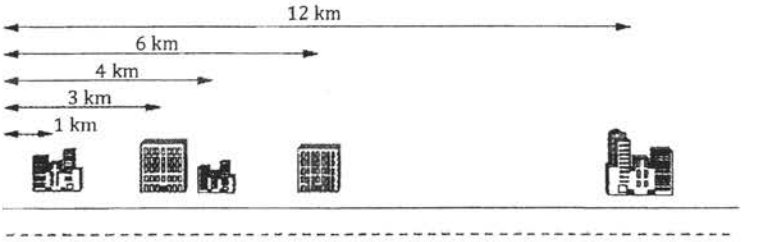
回到本题，根据数学归纳法，我们可以先求出前 2 个序列中任取一个数相加构成的前 N 小和，把这 N 个和作为一个序列，再与第 3 个序列求新的前 N 小和，依此类推，最终得到 M 个序列任取一个数相加构成的前 N 小和。整个算法的时间复杂度为 $O(MN\log N)$。

例题 数据备份 AcWing147

你在一家 IT 公司为大型写字楼或办公楼（offices）的计算机数据做备份。然而，数据备份的工作是枯燥乏味的。因此，你想设计一个系统让不同的办公楼彼此之间互相备份，而你则坐在家中尽享计算机游戏的乐趣。

已知办公楼都位于同一条街上。你决定给这些办公楼配对（两个一组）。每一对办公楼可以通过在这两个建筑物之间铺设网络电缆使得它们可以互相备份。然而，网络电缆的费用很高。当地电信公司仅能为你提供 K 条网络电缆，这意味着你仅能为 K 对办公楼（或总计 $2K$ 个办公楼）安排备份。任一个办公楼都属于唯一的配对组（换句话说，这 $2K$ 个办公楼一定是相异的）。此外，电信公司需按网络电缆的长度（公里数）收费。因而，你需要选择这 K 对办公楼使得电缆的总长度尽可能短。换句话说，你需要选择这 K 对办公楼，使得每一对办公楼之间的距离之和（总距离）尽可能小。

下面给出一个示例，假定你有 5 个客户，其办公楼都在一条街上，如下图所示。这 5 个办公楼分别位于距离大街起点 1km, 3km, 4km, 6km 和 12km 处。电信公司仅为你提供 $K=2$ 条电缆。

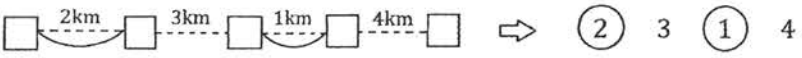


上例最好的配对方案是将第 1 个和第 2 个办公楼相连，第 3 个和第 4 个办公楼相连。这样可按要求使用 $K=2$ 条电缆。第 1 条电缆的长度是 $3\text{km}-1\text{km}=2\text{km}$，第

2 条电缆的长度是 6km − 4km = 2km。这种配对方案需要总长 4km 的网络电缆，满足距离之和最小的要求。

办公楼数目 N 满足 $2 \le N \le 10^5$，电缆数 K 满足 $1 \le K \le N/2$。

我们很容易发现，最优解中每两个配对的办公楼一定是相邻的。我们求出每两个相邻办公楼之间的距离，记为 $D_1, D_2, D_3, \cdots, D_{N-1}$。于是问题可以转化为：从数列 D 中选出不超过 K 个数（对应办公楼的 K 个配对），使它们的和最小，并且相邻的两个数不能同时被选（任一办公楼都属于唯一的配对组）。



如果 $K = 1$，答案显然是 D 数列中的最小值。

如果 $K = 2$，答案一定是以下两种情况之一。

1. 选择最小值 D_i，以及除了 D_{i-1}，D_i，D_{i+1} 之外其他数中的最小值。

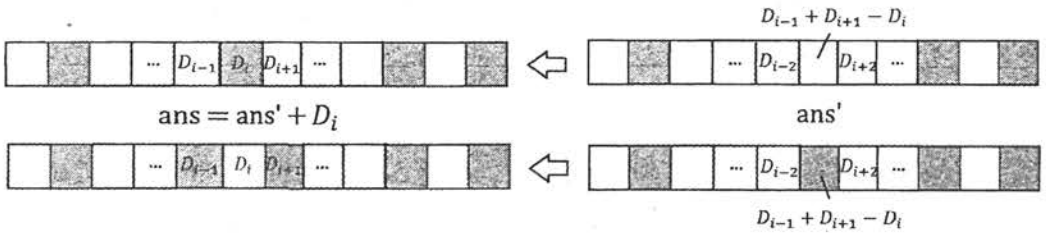
2. 选择最小值 D_i 左右两侧的两个数，即 D_{i-1} 和 D_{i+1}。

这很容易证明：如果 D_{i-1} 和 D_{i+1} 都没有选，那么不选最小值 D_i 一定不优；如果 D_{i-1} 和 D_{i+1} 选了一个，那么把选了的那个换成 D_i，答案也会变小，所以最优解必定是上面两种情况之一。

通过上述证明，我们也可以得到一个推论：在最优解中，最小值左右两侧的数要么同时选，要么都不选。

因此，我们可以先选上 D 数列中的最小值，然后把 D_{i-1}，D_i，D_{i+1} 从 D 数列中删除，把 $D_{i-1} + D_{i+1} - D_i$ 插入到 D 数列中刚才执行删除的位置。最后，求解“从新的 D 数列中选出不超过 $K-1$ 个数，使它们的和最小，并且相邻两个数不能同时被选”这个子问题。

在这个子问题中，如果选了 $D_{i-1} + D_{i+1} - D_i$ 这个数，相当于去掉 D_i，换上 D_{i-1} 和 D_{i+1}；如果没选，那么刚才选择最小值 D_i 显然是一步最优策略。这恰好涵盖了我们在推论中提到的最优解当中的两种情况。



综上所述，我们得到了这样一个算法：

建立一个链表 L，连接 $N-1$ 个节点，节点上分别记录数值 $D_1, D_2, D_3, \cdots, D_{N-1}$，即每两个相邻办公楼之间的距离。再建立一个小根堆，与链表构成映射关系（就是说堆中也有 $N-1$ 个节点，节点权值分别是 $D_1, D_2, D_3, \cdots, D_{N-1}$，并且同时记录对应的链表

节点的指针）。

取出堆顶，把权值累加到答案中。设堆顶对应链表节点的指针为 p，数值为 $L(p)$。在链表中删除 p，p->prev 和 p->next，在同样的位置插入一个新节点 q，记录数值 $L(q) = L(p\text{->prev}) + L(p\text{->next}) - L(p)$。在堆中也同时删除对应 p->prev 和 p->next 的节点，插入对应链表节点 q，权值为 $L(q)$ 的新节点。

重复上述操作 K 次，就得到了最终的答案。

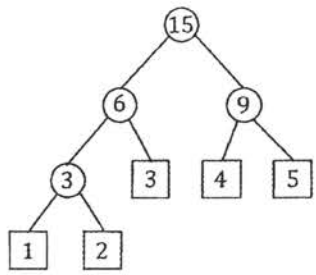
Huffman 树

考虑这样一个问题：构造一棵包含 n 个叶子节点的 k 叉树，其中第 i 个叶子节点带有权值 w_i，要求最小化 $\sum w_i * l_i$，其中 l_i 表示第 i 个叶子节点到根节点的距离。该问题的解被称为 k 叉 Huffman 树（哈夫曼树）。

为了最小化 $\sum w_i * l_i$，应该让权值大的叶子节点的深度尽量小。当 $k = 2$ 时，我们很容易想到用下面这个贪心算法来求出二叉 Huffman 树。

1. 建立一个小根堆，插入这 n 个叶子节点的权值。
2. 从堆中取出最小的两个权值 w_1 和 w_2，令 ans += $w_1 + w_2$。
3. 建立一个权值为 $w_1 + w_2$ 的树节点 p，令 p 成为权值为 w_1 和 w_2 的树节点的父亲。
4. 在堆中插入权值 $w_1 + w_2$。
5. 重复第 2~4 步，直至堆的大小为 1。

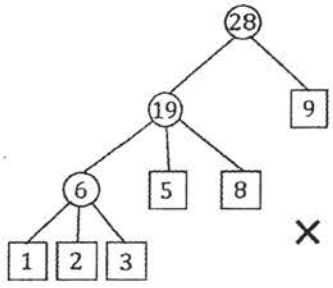
最后，由所有新建的 p 与原来的叶子节点构成的树就是 Huffman 树，变量 ans 就是 $\sum w_i * l_i$ 的最小值。



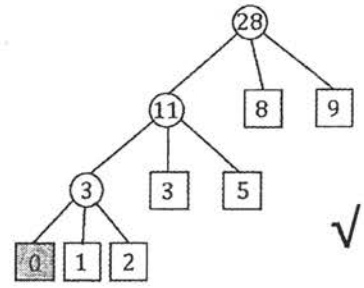
对于 $k(k > 2)$ 叉 Huffman 树的求解，直观的想法是在上述贪心算法的基础上，改为每次从堆中取出最小的 k 个权值。然而，仔细思考可以发现，如果在执行最后一轮循环时，堆的大小在 $2 \sim k-1$ 之间（不足以取出 k 个），那么整个 Huffman 树的根的子节点个数就小于 k。这显然不是最优解——我们任意取 Huffman 树中一个深度最大的节点，把它改为树根的子节点，就会使 $\sum w_i * l_i$ 变小。

因此，我们应该在执行上述贪心算法之前，补加一些额外的权值为 0 的叶子节点，使叶子节点的个数 n 满足 $(n-1) \bmod (k-1) = 0$。也就是说，我们让子节点不足 k

个的情况发生在最底层，而不是根节点处。在 $(n-1) \bmod (k-1) = 0$ 时，执行“每次从堆中取出最小的 k 个权值”的贪心算法就是正确的。



ans=(1+2+3)*3+(5+8)*2+9=53



ans=(1+2)*3+(3+5)*2+(8+9)=42

例题 合并果子 AcWing148

在一个果园里，多多已经将所有的果子打了下来，而且按果子的不同种类分成了不同的堆。多多决定把所有的果子合成一堆。

每一次合并，多多可以把任意两堆果子合并到一起，消耗的体力等于两堆果子的重量之和。可以看出，所有的果子经过 $n-1$ 次合并之后，就只剩下一堆了。多多在合并果子时总共消耗的体力等于每次合并所耗体力之和。

因为还要花大力气把这些果子搬回家，所以多多在合并果子时要尽可能地节省体力。假定每个果子重量都为 1，并且已知果子的种类数和每种果子的数目，你的任务是设计出合并的次序方案，使多多耗费的体力最少，并输出这个最小的体力耗费值。

例如有 3 种果子，数目依次为 1，2，9。可以先将 1，2 堆合并，新堆数目为 3，耗费体力为 3。接着，将新堆与原先的第三堆合并，又得到新的堆，数目为 12，耗费体力为 12。所以多多总共耗费体力 $=3+12=15$。可以证明 15 为最小的体力耗费值。

因为每次合并消耗的体力等于两堆果子的重量之和，所以最终消耗的体力总和就是每堆果子的重量乘它参与合并的次数。这恰好对应一个二叉 Huffman 树问题，果子堆的重量就是叶子节点的权值，参与合并的次数就是叶子到根的距离。

建立一个小根堆，插入所有果子堆的重量。不断取出堆中最小的两个值，把它们的和插入堆，同时累加到答案中。直至最后堆的大小为 1 时，输出答案即可。

例题 荷马史诗 AcWing149

追逐影子的人，自己就是影子。 ——荷马

Allison 最近迷上了文学。她喜欢在一个慵懒的午后，细细地品上一杯卡布奇诺，静静地阅读她爱不释手的《荷马史诗》。但是由《奥德赛》和《伊利亚特》组成的鸿篇巨制《荷马史诗》实在是太长了，Allison 想通过一种编码方式使得它变得短一些。

一部《荷马史诗》中有 n 种不同的单词，从 1 到 n 进行编号。其中第 i 种单词出现的总次数为 w_i。Allison 想要用 k 进制串 s_i 来替换第 i 种单词，使得其满足如下要求：

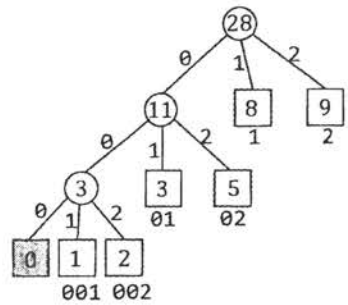
对于任意的 $1 \le i,j \le n$，$i \ne j$，都有：s_i 不是 s_j 的前缀。

现在 Allison 想要知道，如何选择 s_i，才能使替换以后得到的新的《荷马史诗》长度最小。在确保总长度最小的情况下，Allison 还想知道最长的 s_i 的最短长度是多少。

一个字符串被称为 k 进制字符串，当且仅当它的每个字符是 0 到 $k-1$ 之间（包括 0 和 $k-1$）的整数。字符串 $Str1$ 被称为字符串 $Str2$ 的前缀，当且仅当：存在 $1 \le t \le m$，使得 $Str1 = Str2[1..t]$。其中，m 是字符串 $Str2$ 的长度，$Str2[1..t]$ 表示 $Str2$ 的前 t 个字符组成的字符串。

本题所构造的编码方式其实就是 Huffman 编码。我们把单词的出现次数 $w_1 \sim w_n$ 作为 Huffman 树叶子节点的权值，然后求出 k 叉 Huffman 树。对于 Huffman 树每个节点的 k 个分支，分别在边上标记字符 $0 \sim k-1$。

此时，如果把这棵 Huffman 树看作一棵 Trie 树，就得到了使总长度最小的编码方式——单词 i 的编码就是从根节点到叶子节点 i 的路径上各条边的字符相连。一个单词不是另一个的前缀，其实就对应着：在 Trie 树中，所有单词编码的末尾都在叶子节点上，而不在 Trie 树的一个内部节点上。我们恰好满足了这个性质。



本题还要求最长的 s_i 长度最短，我们只需要在求 Huffman 树时，对于权值相同的节点，优先考虑当前深度最小（已合并次数最少）的进行合并即可。

0x18 总结与练习

在本章中，我们探讨了程序设计以及算法实现中可能用到的一些基本数据结构，如栈、队列、链表（邻接表结构）、Trie、二叉堆以及与 Hash、字符串相关的问题。通过对这些数据结构的理解和分析，读者对于上一章中学到的各种算法思想也应该有了更加深入的体会。对于这两章的内容，希望读者能够仔细思考各种模型的每个细节，主动寻找和归纳基本算法与基本数据结构之间的结合点。

如果读者认真阅读并实现了这两章的每一个算法、数据结构以及每一道例题，那么读者应该已经初步迈进了算法领域的大门，能够使用较为贴切的术语与其他探索者进行思维碰撞和交流，能够循着正确的方向独立提出一个问题或是扩展一个解答，而不是敲着死记硬背的模板，做着一知半解的试题。从下一章起，读者将会开始被当作专业人士对待，一些面向初学者的解释和过于细节的代码实现可能不会再出现于各个知识点和题解中。因此，建议读者先花一些时间，回顾一遍前两章所学内容和所做习题，检查一遍知识点列表，最好能够辅以网络上各大题库中更多题目的训练。

本章知识点

栈

栈的基本实现，使用数组和栈顶位置变量模拟一个栈
栈的灵活应用，例如使用辅助栈保存额外信息、对顶栈等
表达式计算，后缀表达式、中缀转后缀、中缀表达式递归求值
单调栈

队列

一般队列、双端队列、循环队列的基本实现
单调队列，理解使用单调性处理问题的思想

链表与邻接表

双向链表的实现与操作，以及数组模拟链表
邻接表结构，图和树的邻接表存储与遍历



Hash

Hash 表，使用邻接表结构实现开散列法

字符串 Hash，前缀与区间 Hash 值、二分法的结合

字符串

KMP 模式匹配算法，*next* 数组的灵活运用

最小表示法，循环同构问题

Trie

Trie 的插入、检索等基本操作

Trie 与 xor 问题

二叉堆

二叉堆的基本操作及其实现，Insert、GetTop、Extract、Remove 等

二叉堆的灵活应用，与贪心算法相结合，数据结构间“建立映射”的思想

k 叉 Huffman 树与 Huffman 编码

练习

1. 完成本章中所有例题的代码实现
2. 括号画家

题目地址：AcWing150，提示：栈

3. 表达式计算 4

题目地址：AcWing151，提示：栈/中缀表达式计算

4. City Game

题目地址：AcWing152，提示：单调栈

5. 双栈排序

题目地址：AcWing153，提示：栈

6. Sliding Window

题目地址：AcWing154，提示：单调队列

7. 内存分配

题目地址：AcWing155，提示：链表/二叉堆

8. Matrix

题目地址：AcWing156，提示：字符串 Hash

9. Subway tree systems

题目地址：AcWing157，提示：Hash/树的最小表示

10. Necklace

题目地址：AcWing158，提示：字符串/最小表示法

11. Milking Grid

题目地址：AcWing159，提示：字符串/KMP 模式匹配

12. 匹配统计

题目地址：AcWing160，提示：字符串 Hash 或 KMP 模式匹配

13. Phone List

题目地址：AcWing161，提示：Trie

14. Black Box

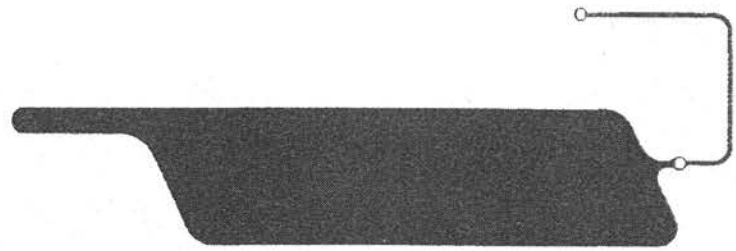
题目地址：AcWing162，提示：二叉堆

15. 生日礼物

题目地址：AcWing163，提示：二叉堆/链表/贪心

0x20 搜 索

在本章中，我们将学习搜索算法。早在第 0x00 章“基本算法”中我们就多次提到，对问题进行求解，实际上是在问题对应的状态空间中进行映射与遍历。在阅读本章之前，请读者再次确保自己对于递归思想已经有了一定的认识，并懂得利用在第 0x10 章中介绍的栈、队列、链表、Hash、二叉堆等存储和操作“状态”的基本工具。这些内容会贯穿于我们搜索算法学习的始末。



在 0x21 节中，我们以树与图的遍历引入本章内容，尝试在树与图结构上进行访问、记录与推导。我们将介绍树的遍历产生的几种序列，树上信息的基本统计方法，图的拓扑排序等算法。这节内容会帮助我们建立“问题状态空间”与图之间的类比联系，从而更加直观地描绘后边几节的各个搜索过程。

在 0x22~0x24 节中，我们探讨搜索的第一种基本实现形式——深度优先搜索（简称深搜，缩写 DFS）。深搜是一种与递归和栈紧密结合的算法。我们会完善“搜索树”这一概念，帮助读者进一步规范和理解问题状态空间的遍历过程。除此之外，我们会用较大篇幅介绍深搜的各种剪枝方法，并辅以大量例题，帮助读者掌握提高搜索时间、空间效率的途径。迭代加深搜索也是我们重点讨论的对象。

在 0x25~0x26 节中，我们探讨搜索的第二种基本实现形式——广度优先搜索（简称广搜，缩写 BFS）。广搜又称为宽度优先搜索，是一种与队列紧密结合的算法。它还有一些使用双端队列、优先队列或其他数据结构的变形版本，我们也会选择重要的几项进行介绍。无论是深搜还是广搜，如何使用 Hash 等方法快速记录并检索搜索历史和已遍历状态，也是一个重要的研究话题。

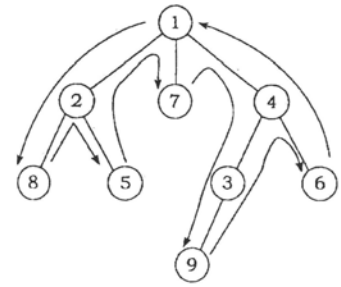
在 0x27~0x28 节中，我们初探启发式搜索。启发式搜索的算法多种多样，也是解决很多复杂问题的强有力手段。对此，我们选择了 A*算法进行讲解，包括其理论和应用，以及 IDA*（迭代加深 A*）这种变化形式。通过搜索类题目的大量训练，读者不仅能初步掌握“非多项式时间可解问题”的处理方法，还能大幅提高代码实现能力，快速提升程序设计水平。

0x21 树与图的遍历一

树与图最常见的存储方式就是使用一个邻接表保存它们的边集。除非特殊声明，本书在接下来的各章节中默认，给定 N 个点的树或图时，其节点编号均为 1~N，无向图中的边看作成对出现的双向边，树看作一张具有 $N-1$ 条边的无向图，它们的边都存储在一个邻接表中，邻接表以 head 数组为表头，使用 ver 和 edge 数组分别存储边的终点和权值，使用 next 数组模拟链表指针（就像我们在 0x13 节中讲解邻接表时所给出的代码那样）。

树与图的深度优先遍历，树的 DFS 序、深度和重心

深度优先遍历，就是在每个点 x 上面对多条分支时，任意选一条边走下去，执行递归，直至回溯到点 x 后，再考虑走向其他的边，如右图所示。根据上面提到的存储方式，我们可以采用下面的代码，调用 dfs(1)，对一张图进行深度优先遍历。




```
void dfs(int x) {
    v[x] = 1; // 记录点 x 被访问过，v 是 visit 的缩写
    for (int i = head[x]; i; i = next[i]) {
        int y = ver[i];
        if (v[y]) continue;  // 点 y 已经被访问过了
        dfs(y);
    }
}
```

这段代码访问每个点和每条边恰好 1 次（如果是无向边，正反向各访问 1 次），其时间复杂度为 $O(N+M)$，其中 M 为边数。以这段代码为框架，我们可以统计许多关于树和图的基本信息。

时间戳

按照上述深度优先遍历的过程，以每个节点第一次被访问（$v[x]$ 被赋值为 1 时）的顺序，依次给予这 N 个节点 1~N 的整数标记，该标记就被称为时间戳，记为 dfn。

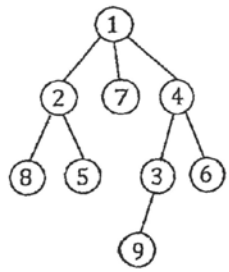
例如，在本页的图中，$dfn[1]=1$，$dfn[2]=2$，$dfn[8]=3$，$dfn[5]=4$，$dfn[7]=5$，$dfn[4]=6$，$dfn[3]=7$，$dfn[9]=8$，$dfn[6]=9$。

树的 DFS 序

一般来讲，我们在对树进行深度优先遍历时，对于每个节点，在刚进入递归后以及即将回溯前各记录一次该点的编号，最后产生的长度为 $2N$ 的节点序列就称为树的 DFS 序。

```
void dfs(int x) {
    a[++m] = x;  // a 数组存储 DFS 序
    v[x] = 1; // 记录点 x 被访问过
    for (int i = head[x]; i; i = next[i]) {
        int y = ver[i];
        if (v[y]) continue;
        dfs(y);
    }
    a[++m] = x;
}
```

DFS 序的特点是：每个节点 x 的编号在序列中恰好出现两次。设这两次出现的位置为 $L[x]$ 与 $R[x]$，那么闭区间 $[L[x],R[x]]$ 就是以 x 为根的子树的 DFS 序。这使我们在很多与树相关的问题中，可以通过 DFS 序把子树统计转化为序列上的区间统计。我们在后续的章节中会涉及此类题目。



DFS 序：1, 2, 8, 8, 5, 5, 2, 7, 7, 4, 3, 9, 9, 3, 6, 6, 4, 1
子树 2　子树 7　子树 4

另外，二叉树的先序、中序与后序遍历序列，也是通过深度优先遍历产生的，大多数程序设计入门级的书籍上都有详细讲解，在此就不再赘述。读者应该掌握这几种遍历，以及它们之间的联系和转化。

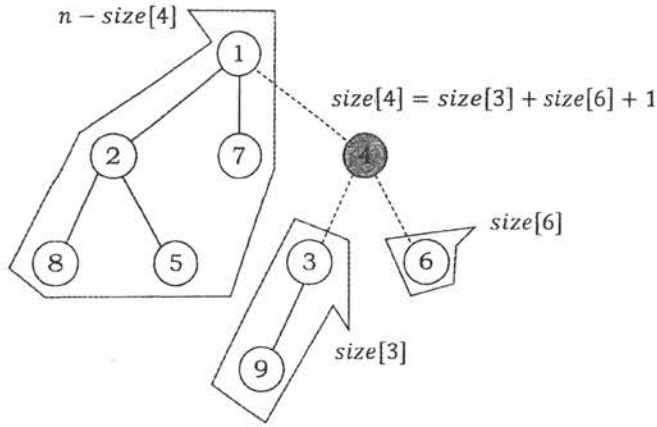
树的深度

树中各个节点的深度是一种自顶向下的统计信息。起初，我们已知根节点的深度为 0。若节点 x 的深度为 $d[x]$，则它的子节点 y 的深度就是 $d[y]=d[x]+1$。在深度优先遍历的过程中结合自顶向下的递推，就可以求出每个节点的深度 d。请读者尝试列举还有哪些信息一般是自顶向下进行统计的。

```
void dfs(int x) {
    v[x] = 1;
    for (int i = head[x]; i; i = next[i]) {
        int y = ver[i];
        if (v[y]) continue;
        d[y] = d[x] + 1;      // 从父节点 x 到子节点 y 递推，计算深度
        dfs(y);
    }
}
```

树的重心

当然，也有许多信息是自底向上进行统计的，比如以每个节点 x 为根的子树大小 $size[x]$。对于叶子节点，我们已知“以它为根的子树”大小为 1。若节点 x 有 k 个子节点 $y_1 \sim y_k$，并且以 $y_1 \sim y_k$ 为根的子树大小分别是 $size[y_1], size[y_2], \cdots, size[y_k]$，则以 x 为根的子树的大小就是 $size[x] = size[y_1] + size[y_2] + \cdots + size[y_k] + 1$。请读者尝试列举还有哪些信息一般是自底向上进行统计的。



对于一个节点 x，如果我们把它从树中删除，那么原来的一棵树可能会分成若干个不相连的部分，其中每一部分都是一棵子树。设 max_part(x) 表示在删除节点 x 后产生的子树中最大的一棵的大小。使 max_part 函数取到最小值的节点 p 就称为整棵树的重心。例如上图中的树的重心应该是节点 1。通过下面的代码，我们可以统计出 size 数组，并求出树的重心。

```
void dfs(int x) {
    v[x] = 1; size[x] = 1;     // 子树 x 的大小
    int max_part = 0;          // 删掉 x 后分成的最大子树的大小
    for (int i = head[x]; i; i = next[i]) {
        int y = ver[i];
        if (v[y]) continue;    // 点 y 已经被访问过了
```

```
        dfs(y);
        size[x] += size[y]; // 从子节点向父节点递推
        max_part = max(max_part, size[y]);
    }
    max_part = max(max_part, n - size[x]); // n为整棵树的节点数目
    if (max_part < ans) {
        ans = max_part;      // 全局变量ans记录重心对应的max_part值
        pos = x;             // 全局变量pos记录了重心
    }
}
```

图的连通块划分

上面的代码每从 x 开始一次遍历，就会访问 x 能够到达的所有的点与边。因此，通过多次深度优先遍历，可以划分出一张无向图中的各个连通块[①]。同理，对一个森林进行深度优先遍历，可以划分出森林中的每棵树。如下面的代码所示，cnt 就是无向图包含的连通块的个数，v 数组标记了每个点属于哪一个连通块。

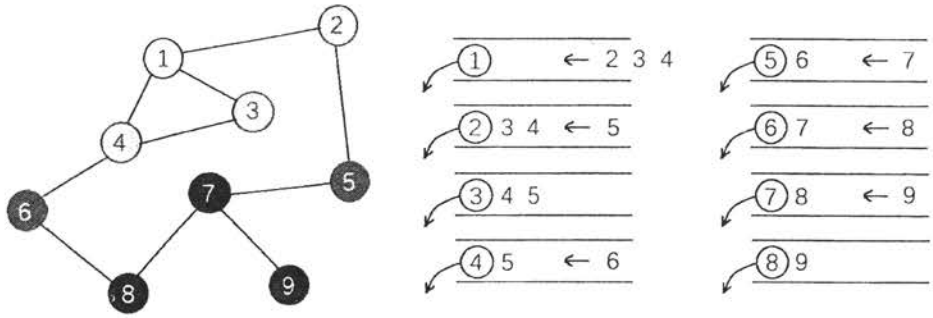
```
void dfs(int x) {
    v[x] = cnt;
    for (int i = head[x]; i; i = next[i]) {
        int y = ver[i];
        if (v[y]) continue;
        dfs(y);
    }
}
for (int i = 1; i <= n; i++)  // 在int main()中
    if (!v[i]) {
        cnt++;
        dfs(i);
    }
```

树与图的广度优先遍历，拓扑排序

树与图的广度优先遍历需要使用一个队列来实现。起初，队列中仅包含一个起点（例如 1 号节点）。在广度优先遍历的过程中，我们不断从队头取出一个节点 x，对于

① 连通块：若在无向图的一个子图中，任意两个节点之间都存在一条路径（可以互相到达），并且这个子图是“极大”的（不能再扩张），则称该子图为无向图的一个连通块。一张不连通的无向图由 2 个或 2 个以上连通块组成，而一张无向连通图整体是一个连通块。

x 面对的多条分支，把沿着每条分支到达的下一个节点（如果尚未访问过）插入队尾。重复执行上述过程直到队列为空。



我们可以采用下面的代码对一张图进行广度优先遍历（关于代码中的 STL queue，参见 0x71 节）。

```
void bfs() {
    memset(d, 0, sizeof(d));
    queue<int> q;
    q.push(1); d[1] = 1;
    while (q.size() > 0) {
        int x = q.front(); q.pop();
        for (int i = head[x]; i; i = next[i]) {
            int y = ver[i];
            if (d[y]) continue;
            d[y] = d[x] + 1;
            q.push(y);
        }
    }
}
```

在上面的代码中，我们在广度优先遍历的过程中顺便求出了一个 d 数组。对于一棵树来讲，$d[x]$ 就是点 x 在树中的深度。对于一张图来讲，$d[x]$ 被称为点 x 的层次（从起点 1 走到点 x 需要经过的最少点数）。从代码和示意图中我们可以发现，广度优先遍历是一种按照层次顺序进行访问的方法，它具有如下两个重要性质：

1. **在访问完所有的第 i 层节点后，才会开始访问第 $i+1$ 层节点。**

2. 任意时刻，队列中至多有两个层次的节点。若其中一部分节点属于第 i 层，则另一部分节点属于第 $i+1$ 层，并且所有第 i 层节点排在第 $i+1$ 层节点之前。也就是说，**广度优先遍历队列中的元素关于层次满足“两段性”和“单调性”。**

这两条性质是所有广度优先思想的基础，请读者务必牢记，我们在 0x26 节的“广搜变形”中会再次提及并探讨。与深度优先遍历一样，上页这段代码的时间复杂度也是 $O(N+M)$。

拓扑排序

给定一张有向无环图[①]，若一个由图中所有点构成的序列 A 满足：对于图中的每条边 (x,y)，x 在 A 中都出现在 y 之前，则称 A 是该有向无环图顶点的一个拓扑序。求解序列 A 的过程就称为拓扑排序。

拓扑排序过程的思想非常简单，我们只需要不断选择图中入度[②]为 0 的节点 x，然后把 x 连向的点的入度减 1。我们可以结合广度优先遍历的框架来高效地实现这个过程。

1. 建立空的拓扑序列 A。
2. 预处理出所有点的入度 $\deg[i]$，起初把所有入度为 0 的点入队。
3. 取出队头节点 x，把 x 加入拓扑序列 A 的末尾。
4. 对于从 x 出发的每条边 (x,y)，把 $\deg[y]$ 减 1。若被减为 0，则把 y 入队。
5. 重复第 3~4 步直到队列为空，此时 A 即为所求。

拓扑排序可以判定有向图中是否存在环。我们可以对任意有向图执行上述过程，在完成后检查 A 序列的长度。若 A 序列的长度小于图中点的数量，则说明某些节点未被遍历，进而说明图中存在环。读者可以参考下面的程序，画图模拟拓扑排序算法。

```
void add(int x, int y) { // 在邻接表中添加一条有向边
    ver[++tot] = y, next[tot] = head[x], head[x] = tot;
    deg[y]++;
}
void topsort() { // 拓扑排序
    queue<int> q;
    for (int i = 1; i <= n; i++)
        if (deg[i] == 0) q.push(i);
    while (q.size()) {
        int x = q.front(); q.pop();
        a[++cnt] = x;
        for (int i = head[x]; i; i = next[i]) {
            int y = ver[i];
```

① 有向无环图：在有向图中，从一个节点出发，最终回到它自身的路径被称为“环”。不存在环的有向图即为有向无环图。

② 入度：在有向图中，以节点 x 为终点的有向边的条数被称为 x 的入度，以节点 x 为起点的有向边的条数被称为 x 的出度。在无向图中，以 x 为端点的无向边的条数被称为 x 的度。

```
            if (--deg[y] == 0) q.push(y);
        }
    }
}
int main() {
    cin >> n >> m; // 点数、边数
    for (int i = 1; i <= m; i++) {
        int x, y;
        scanf("%d%d", &x, &y);
        add(x, y);
    }
    topsort();
    for (int i = 1; i <= cnt; i++)
        printf("%d ", a[i]);
    cout << endl;
}
```

例题 可达性统计 AcWing164

给定一张 N 个点 M 条边的有向无环图，分别统计从每个点出发能够到达的点的数量。$N, M \leq 30000$。

设从点 x 出发能够到达的点构成的集合是 $f(x)$，则显然有：

$$f(x) = \{x\} \cup \left(\bigcup_{\text{存在有向边}(x,y)} f(y) \right)$$

也就是说，从 x 出发能够到达的点，是从“x 的各个后继节点 y”出发能够到达的点的并集，再加上点 x 自身。所以，在计算出一个点的所有后继节点的 f 值之后，就可以计算出该点的 f 值。这启发我们先用拓扑排序算法求出一个拓扑序，然后按照拓扑序的倒序进行计算——因为在拓扑序中，对任意的一条边 (x, y)，x 都排在 y 之前。

回想起在第 0x01 节中学到的状态压缩方法，我们可以使用一个 N 位二进制数存储每个$f(x)$，其中第 i 位是 1 表示 x 能到 i，是 0 表示不能到 i。这样一来，对若干个集合求并，就相当于对若干个 N 位二进制数做“按位或”运算。最后，每个 $f(x)$ 中 1 的个数就是从 x 出发能够到达的节点数量。

这个 N 位二进制数可以压缩成 $N/32 + 1$ 个无符号 32 位整数 unsigned int 进行存储，更简便的方法是直接使用 C++ STL 中为我们提供的 bitset（参见第 0x71 节），bitset 支持我们上述所需的所有运算。该拓扑排序+状态压缩算法花费的时间规模约为 $N(N + M)/32$，所需空间大小约为 $N^2/8$ 字节。

0x22 深度优先搜索

深度优先搜索（DFS，Depth First Search），顾名思义，就是按照深度优先的顺序对“问题状态空间”进行搜索的算法。在第 0x00 章中，我们多次把一个问题的求解看作对问题状态空间的遍历与映射。从本章起，我们可以进一步把“问题空间”类比为一张图，其中的状态类比为节点，状态之间的联系与可达性就用图中的边来表示，那么使用深度优先搜索算法求解问题，就相当于在一张图上进行深度优先遍历。

读者可能发现，深度优先搜索与“递归”和“栈”密切相关。我们倾向于认为“递归”是与递推相对的一种单纯的遍历方法，除了搜索之外，还有许多算法都可以用递归实现。而“深搜”是一类包括遍历形式、状态记录与检索、剪枝优化等算法整体设计的统称。

在研究深度优先搜索算法之前，我们先来定义该过程产生的“**搜索树**”结构。在对图进行深度优先遍历处于点 x 时，对于某些边 (x,y)，y 是一个尚未访问过的节点，程序从 x 成功进入了更深层的对 y 的递归；对于另外的一些边 (x,y)，y 已经被访问过，从而程序继续考虑其他分支。我们称所有点（问题空间中的状态）与成功发生递归的边（访问两个状态之间的移动）构成的树为一棵“搜索树”。整个深搜算法就是基于该搜索树完成的——为了避免重复访问，我们对状态进行记录和检索；为了使程序更加高效，我们提前剪除搜索树上不可能是答案的子树和分支，不去进行遍历。

我们在 0x03 节中使用递归实现的指数型、排列型和组合型枚举，其实就是深搜的三种最简单的形式。与之相关的**子集和问题**、**全排列问题**、**N 皇后问题**等都是可以用深搜求解的经典 NPC 问题。许多程序设计的入门书籍均以这几个问题作为例子讲解深搜算法，故本书就不再赘述，请读者确保自己已经掌握并能够实现这几个经典问题的深搜解法。在本节中，我们先通过几道更加综合性的例题直观地感受一下深度优先搜索算法。下节我们将进一步系统地讨论各类剪枝技巧。

例题 小猫爬山 AcWing165

Freda 和 Rainbow 运送 N 只小猫坐索道下山。索道上的缆车最大承重量为 W，而 N 只小猫的重量分别是 $C_1, C_2, \cdots, C_N$。当然，每辆缆车上的小猫的重量之和不能超过 W。每租用一辆缆车，Freda 和 Rainbow 就要付 1 美元，所以他们想知道，最少需要付多少美元才能把这 N 只小猫都运送下山？$1 \le N \le 18$，$1 \le C_i \le W \le 10^8$。

可以使用深度优先搜索算法解决本题。在搜索的过程中，我们可以尝试依次把每一只小猫分配到一辆已经租用的缆车上，或者新租一辆缆车安置这只小猫。于是，我们实时关心的状态有：已经运送的小猫有多少只，已经租用的缆车有多少辆，每辆缆车上当前搭载的小猫重量之和。

编写函数 dfs(now, cnt) 处理第 now 只小猫的分配过程（前 $now-1$ 只已经装载），并且目前已经租用了 cnt 辆缆车。对于已经租用的这 cnt 辆缆车的当前搭载量，我们使用一个全局数组 $cab[]$ 来记录。

now, cnt和 cab 数组共同标识着问题状态空间所类比的“图”中的一个“节点”。在这个“节点”上，我们至多有 $cnt+1$ 个可能的分支。

1. 尝试把第 now 只小猫分配到已经租用的第 $i(1 \le i \le cnt)$ 辆缆车上。如果第 i 辆缆车还装得下，我们就在 $cab[i]$ 中累加 C_{now}，然后递归 dfs($now+1, cnt$)。

2. 尝试新租一辆缆车来安置这只小猫，也就是令 $cab[cnt+1]=C_{now}$，然后递归 dfs($now+1, cnt+1$)。

当 $now=N+1$ 时，说明搜索到了递归边界，此时就可以用 cnt 更新答案。

为了让搜索过程更加高效，我们可以加入一个很显然的优化：如果在搜索的任何时刻发现 cnt 已经大于或等于已经搜到的答案，那么当前分支就可以立即回溯了。另外，重量较大的小猫显然比重量较轻的小猫更“难”运送，我们还可以在整个搜索前把小猫按照重量递减排序，优先搜索重量较大的小猫，减少搜索树“分支”的数量。读者可以比较加入这两条优化前后，程序运行时间的差异。

```
int c[20], cab[20], n, w, ans;
void dfs(int now, int cnt) {
    if (cnt >= ans) return;
    if (now == n + 1) {
        ans = min(ans, cnt);
        return;
    }
    for (int i = 1; i <= cnt; i++) {  // 分配到已租用缆车
        if (cab[i] + c[now] <= w) {   // 能装下
            cab[i] += c[now];
            dfs(now + 1, cnt);
            cab[i] -= c[now];         // 还原现场
        }
    }
    cab[cnt + 1] = c[now];
    dfs(now + 1, cnt + 1);
    cab[cnt + 1] = 0;
```

```
}
int main() {
    cin >> n >> w;
    for (int i = 1; i <= n; i++) cin >> c[i];
    sort(c + 1, c + n + 1); reverse(c + 1, c + n + 1);
    ans = n;    // 最多用 n 辆缆车，每只猫一辆
    dfs(1, 0);  // 搜索入口
    cout << ans << endl;
}
```

例题 **Sudoku** AcWing166

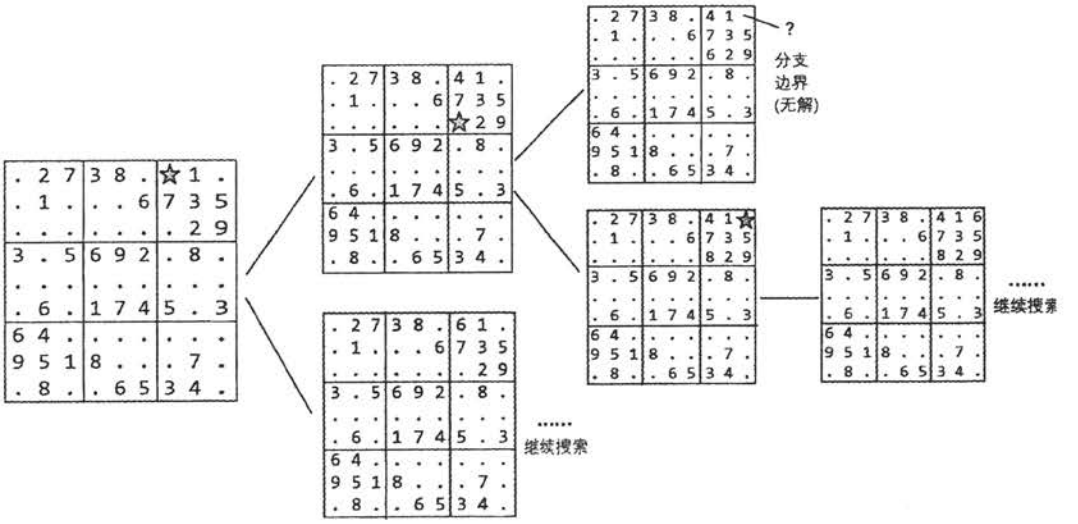
数独是一种传统益智游戏，你需要把一个 9*9 的数独（如右图）补充完整，使得图中每行、每列、每个 3*3 的九宫格内数字 1~9 均恰好出现一次。请编写一个程序填写数独。POJ3074 的测试数据难度要高于 POJ2676。

```
. 2 7 | 3 8 . | . 1 .
. 1 . | . . 6 | 7 3 5
. . . | . . . | . 2 9
-------+-------+-------
3 . 5 | 6 9 2 | . 8 .
. . . | . . . | . . .
. 6 . | 1 7 4 | 5 . 3
-------+-------+-------
6 4 . | . . . | . . .
9 5 1 | 8 . . | . 7 .
. 8 . | . 6 5 | 3 4 .
```

数独问题的搜索框架非常简单，我们关心的“状态”就是数独的每个位置上填了什么数。在每个状态下，我们找出一个还没有填的位置，检查有哪些合法的数字可以填。这些合法的数字就构成该状态向下继续递归的“分支”。

搜索边界分为两种：一是如果所有位置都被填满，就找到了一个解；二是如果发现某个位置没有能填的合法数字，说明当前分支搜索失败，应该回溯去尝试其他分支。

如下图所示，就是用深度优先搜索算法求解 Sudoku 问题产生的一棵搜索树。



提醒：在任意一个状态下，我们只需要找出 1 个位置，考虑该位置上填什么数（如

上图搜索树的根节点只有 2 个分支），不需要枚举所有的位置和可填的数字向下递归（因为其他位置在更深的层次会被搜索到）。新手常犯的错误就是重叠、混淆“层次”和“分支”，造成重复遍历若干棵覆盖同一状态空间的搜索树，致使搜索的复杂度大规模增长。

然而，数独问题的“搜索树”规模仍然很大，直接盲目搜索的效率实在不能接受。试想，如果是人类来玩数独，策略一定是先填上“已经能够唯一确定的位置”，然后从那些填得比较满、选项比较少的位置实施突破。所以，在搜索算法中，也应该采取类似的策略：在每个状态下，从所有未填的位置里**选择“能填的合法数字”最少的位置**，考虑该位置上填什么数，作为搜索的分支，而不是任意找出一个位置。

在搜索程序中，影响时间效率的因素除了搜索树的规模（影响算法的时间复杂度），还有在每个状态上记录、检索、更新的开销（影响程序运行的“常数”时间）。我们可以使用位运算来代替数组执行“对数独各个位置所填数字的记录”以及“可填性的检查与统计”。这就是我们所说的程序“常数优化”。具体地说，本题有以下优化方法。

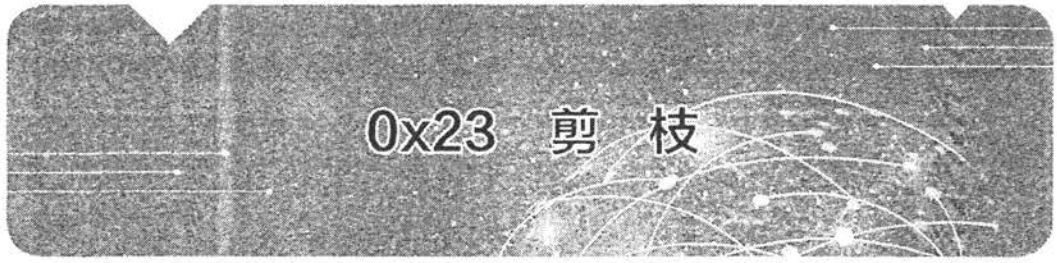
1. 对于每行、每列、每个九宫格，分别用一个 9 位二进制数（全局整数变量）保存哪些数字还可以填。

2. 对于每个位置，把它所在行、列、九宫格的 3 个二进制数做位与(&)运算，就可以得到该位置能填哪些数，用 lowbit 运算就可以把能填的数字取出（在 0x01 节已经探讨了这一技巧）。

3. 当一个位置填上某个数后，把该位置所在的行、列、九宫格记录的二进制数的对应位改为 0，即可更新当前状态；回溯时改回 1，即可还原现场。

上述算法已经能够快速求解 9*9 的数独问题，并且程序的实现十分简洁。配套资源中给出了一份参考程序。

通过本节的内容，读者可能已经发现，一般搜索解法的基本框架并不难，但如何减小搜索树的规模并快速遍历搜索树却是一门高深的学问。在接下来的两节中，我们就重点研究搜索的优化。我们还会探究 16*16 的数独问题，进一步提高求解数独的效率。



剪枝，就是减小搜索树规模、尽早排除搜索树中不必要的分支的一种手段。形象地看，就好像剪掉了搜索树的枝条，故被称为“剪枝”。在深度优先搜索中，有以下几类常见的剪枝方法。

1. 优化搜索顺序

在一些搜索问题中，搜索树的各个层次、各个分支之间的顺序不是固定的。不同的搜索顺序会产生不同的搜索树形态，其规模大小也相差甚远。例如：

(1) 在上一节的“小猫爬山”问题中，把小猫按照重量递减的顺序进行搜索；

(2) 在上一节的“Sudoku”问题中，优先搜索“能填的合法数字”最少的位置。

2. 排除等效冗余

在搜索过程中，如果我们能够判定从搜索树的当前节点上沿着某几条不同分支到达的子树是等效的，那么只需要对其中的一条分支执行搜索。我们会在本节的“Sticks”问题中看到该剪枝的应用。

另外，就如我们在上一节的“Sudoku”问题中提出的，初学者一定要避免重叠、混淆“层次”与“分支”，避免遍历若干棵覆盖同一状态空间的等效搜索树。

3. 可行性剪枝

在搜索过程中，及时对当前状态进行检查，如果发现分支已经无法到达递归边界，就执行回溯。这就好比我们在道路上行走时，远远看到前方是一条死胡同，就应该立即折返绕路，而不是走到路的尽头再返回。

某些题目条件的范围限制是一个区间，此时可行性剪枝也被称为“上下界剪枝”。

4. 最优性剪枝

在最优化问题的搜索过程中，如果当前花费的代价已经超过了当前搜到的最优解，那么无论采取多么优秀的策略到达递归边界，都不可能更新答案。此时可以停止对当前分支的搜索，执行回溯。

5. 记忆化

可以记录每个状态的搜索结果，在重复遍历一个状态时直接检索并返回。这就好比我们对图进行深度优先遍历时，标记一个节点是否已经被访问过。

不过，读者可能已经发现，在“小猫爬山”与“Sudoku”问题中，搜索算法遍历的状态空间其实是“树”形，不会重复访问，所以不需要进行记录。

在本节中，我们通过几道例题来了解这些剪枝方法的具体应用。

例题 Sticks AcWing167

乔治拿来一组等长的木棒，将它们随机地砍断，得到若干根小木棍，并使每一节木棍的长度都不超过 50 个长度单位。然后他又想把这些木棍拼接起来，恢复到裁剪前的状态，但他忘记了初始时有多少木棒以及木棒的初始长度。请你设计一个程序，帮助乔治计算木棒的可能最小长度。每一节木棍的长度都用大于零的整数表示。

输入包含多组数据，每组数据包括两行。第一行是一个不超过 64 的整数，表示砍断之后共有多少节木棍。第二行是截断以后，所得到的各节木棍的长度。在最后一组数据之后，是一个零。

对于每组数据，分别输出原始木棒的可能最小长度。

我们可以从小到大枚举原始木棒的长度 len（也就是枚举答案）。当然，len 应该是所有木棍长度总和 sum 的约数，并且原始木棒的根数 cnt 就等于 sum/len。

对于枚举的每个 len，我们可以依次搜索每根原始木棒由哪些木棍拼成。具体地讲，搜索所面对的状态包括：**已经拼好的原始木棒根数，正在拼的原始木棒的当前长度，每个木棍的使用情况**。在每个状态下，我们**从尚未使用的木棍中选择一个，尝试拼到当前的原始木棒里**，然后递归到新的状态。递归边界就是成功拼好 cnt 根原始木棒，或者因无法继续拼接而宣告失败。

这个算法的效率比较低，我们来依次考虑几类剪枝。

1. 优化搜索顺序

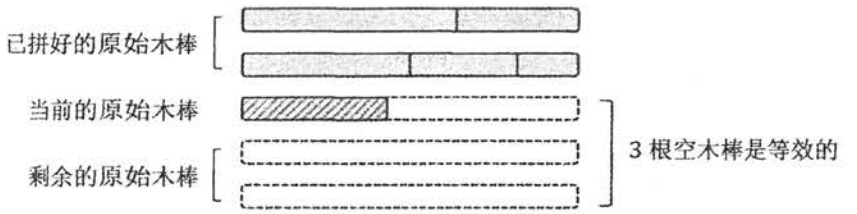
把木棍长度从大到小排序，优先尝试较长的木棍。

2. 排除等效冗余

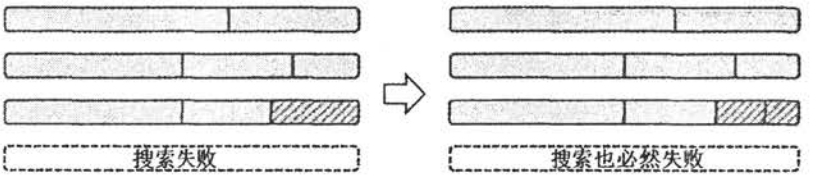
(1) 可以限制先后加入一根原始木棒的木棍长度是递减(≥)的。这是因为先拼上一根长度为 x 的木棍，再拼上一根长为 y 的木棍（$x < y$），与先拼上 y 再拼上 x 显然是等效的，只需要搜索其中一种。

(2) 对于当前原始木棒，记录最近一次尝试拼接的木棍长度。如果分支搜索失败回溯，不再尝试向该木棒中拼接其他相同长度的木棍（必定也会失败）。

(3) 如果在当前原始木棒中“尝试拼入的第一根木棍”的递归分支就返回失败，那么直接判定当前分支失败，立即回溯。这是因为在拼入这根木棍前，面对的原始木棒都是“空”的（还没有进行拼接），这些木棒是等效的。木棍拼在当前的木棒中失败，拼在其他木棒中一样会失败。如下图所示。



(4) 如果在当前原始木棒中拼入一根木棍后，木棒恰好被拼接完整，并且“接下来拼接剩余原始木棒”的递归分支返回失败，那么直接判定当前分支失败，立即回溯。该剪枝可以用贪心来解释，“再用 1 根木棍恰好拼完当前原始木棒”必然比“再用若干根木棍拼完当前原始木棒”更好。如下图所示。



上述(1)至(4)分别利用“同一根木棒上木棍顺序的等效性”“等长木棍的等效性”“空木棒的等效性”和“贪心”，剪掉了搜索树上诸多分支，使得搜索效率大大提升。

```
int a[100], v[100], n, len, cnt;
// 正在拼第 stick 根原始木棒（已经拼好了 stick-1 根）
// 第 stick 根木棒的当前长度为 cab
// 拼接到第 stick 根木棒中的上一根小木棍为 last
bool dfs(int stick, int cab, int last) {
    // 所有原始木棒已经全部拼好，搜索成功
    if (stick > cnt) return true;
    // 第 stick 根木棒已经拼好，去拼下一根
    if (cab == len) return dfs(stick + 1, 0, 1);
    int fail = 0; // 剪枝(2)
    // 剪枝(1)：小木棍长度递减（从 last 开始枚举）
    for (int i = last; i <= n; i++)
        if (!v[i] && cab + a[i] <= len && fail != a[i]) {
            v[i] = 1;
            if (dfs(stick, cab + a[i], i + 1)) return true;
            fail = a[i];
            v[i] = 0; // 还原现场
            if (cab==0 || cab+a[i]==len) return false; // 剪枝(3,4)
        }
```

```
    return false; // 所有分支均尝试过，搜索失败
}

int main() {
    while (cin >> n && n) {
        int sum = 0, val = 0;
        for (int i = 1; i <= n; i++) {
            scanf("%d", &a[i]);
            sum += a[i]; val = max(val, a[i]);
        }
        sort(a + 1, a + n + 1);
        reverse(a + 1, a + n + 1);
        for (len = val; len <= sum; len++) {
            if (sum % len) continue;
            cnt = sum / len; // 原始木棒长度为len，共cnt根
            memset(v, 0, sizeof(v));
            if (dfs(1, 0, 1)) break;
        }
        cout << len << endl;
    }
}
```

例题 生日蛋糕 AcWing168

Mr. W 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 $i(1 \le i \le M)$ 层蛋糕是半径为 R_i，高度为 H_i 的圆柱。当 $i < M$ 时，要求 $R_i > R_{i+1}$ 且 $H_i > H_{i+1}$。

因为要在蛋糕上抹奶油，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。

记 $Q = S\pi$，请编程对给出的 N 和 M，找出一种蛋糕的制作方案（适当的 R_i 和 H_i 的值），使 S 最小。

除 Q 外，以上所有符号皆表示正整数，$N \le 10000$，$M \le 20$，圆柱体积 $V = \pi R^2 H$，侧面积 $A' = 2\pi RH$，底面积 $A = \pi R^2$。

搜索框架：从下往上搜索，枚举每层的半径和高度作为分支。

搜索面对的状态有：正在搜索蛋糕第 dep 层，当前外表面面积 s，当前体积 v，第 $dep+1$ 层的高度和半径。不妨用数组 h 和 r 分别记录每层的高度和半径。

整个蛋糕的“上表面”面积之和等于最底层的圆面积，可以在第 M 层直接累加到 s 中。这样在第 $M-1$ 层往上的搜索中，只需要计算侧面积。

剪枝：

1. 上下界剪枝

在第 dep 层时，只在下面的范围内枚举半径和高度即可。

首先，枚举 $R \in \left[dep, \min\left(\left\lfloor\sqrt{N-v}\right\rfloor, r[dep+1]-1\right)\right]$。

其次，枚举 $H \in [dep, \min(\lfloor(N-v)/R^2\rfloor, h[dep+1]-1)]$。

上面两个区间右边界中的式子可以通过圆柱体积公式 $\pi R^2 H = \pi(N-v)$ 得到。

2. 优化搜索顺序

在上面确定的范围中，使用倒序枚举。

3. 可行性剪枝

可以预处理出从上往下前 $i(1 \le i \le M)$ 层的最小体积和侧面积。显然，当第 $1 \sim i$ 层的半径分别取 $1,2,3,\cdots,i$，高度也分别取 $1,2,3,\cdots,i$ 时，有最小体积与侧面积。

如果当前体积 v 加上 $1 \sim dep-1$ 层的最小体积大于 N，可以剪枝。

4. 最优性剪枝一

如果当前表面积 s 加上 $1 \sim dep-1$ 层的最小侧面积大于已经搜到的答案，剪枝。

5. 最优性剪枝二

利用 h 与 r 数组，$1 \sim dep-1$ 层的体积可表示为 $n-v=\sum_{k=1}^{dep-1} h[k]*r[k]^2$，$1 \sim dep-1$ 层的表面积可表示为 $2\sum_{k=1}^{dep-1} h[k]*r[k]$。

因为 $2\sum_{k=1}^{dep-1} h[k]*r[k] = \frac{2}{r[dep]}*\sum_{k=1}^{dep-1} h[k]*r[k]*r[dep] \ge \frac{2}{r[dep]}*\sum_{k=1}^{dep-1} h[k]*r[k]^2 \ge \frac{2(n-v)}{r[dep]}$，所以当 $\frac{2(n-v)}{r[dep]}+s$ 大于已经搜到的答案时，可以剪枝。

加入以上五个剪枝后，搜索算法就可以快速求出该问题的最优解了。

实际上，**搜索算法面对的状态可以看作一个多元组，其中每一元都是问题状态空间中的一个“维度”**。例如本题中，层数 dep、表面积 s、体积 v、第 $dep+1$ 层的高度和半径就构成状态空间中的五个维度，其中每一项发生变化，都会移动到状态空间中的另一个“点”。**这些维度通常在题目描述中也有所体现**，它们一般在输入变量、限制条件、待求解变量等非常关键的位置出现。读者一定要注意提取这些“维度”，从而设计出合适的搜索框架。

搜索过程中的剪枝，其实就是**针对每个“维度”与该维度的边界条件，加以缩放、推导，得出一个相应的不等式，来减少搜索树分支的扩张**。例如本题中的剪枝 1、剪枝 3 和剪枝 4，就是考虑与半径、高度、体积、表面积这些维度的上下界进行比较后直接得到的。

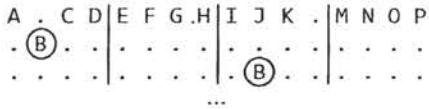
为了进一步提高剪枝的效果，除了当前花费的“代价”之外，我们**还可以对未来至少需要花费的代价进行预算**，这样更容易接近每个维度的上下界。例如本题中的前 $dep-1$ 层最小体积、最小侧面积就是这种想法。剪枝 5 则通过表面积与体积之间的

关系，对不等式进行缩放，$2(n-v)/r[dep]$ 这个式子也是对前 $dep-1$ 层侧面积的一个估计。这告诉我们在一般的剪枝不足以应对问题的时候，也可以**结合各维度之间的联系**得到更加精准的剪枝。在 A*算法中，我们会更详细地讲解这种“未来预算”理论。

例题 **Sudoku** AcWing169

在上一节中，我们解决了 9*9 的传统数独问题。在这道题目中，你需要填写一个 16*16 的数独，使得每行、每列、每个 4*4 的十六宫格内字母 A~P 均恰好出现一次。

在上一节 9*9 的数独问题中，我们只使用了“优先选择能填的数字最少的位置”这一策略，并且只有当某个位置无法填数时才判定失败进行回溯。不过，请读者考虑下面这个 16*16 数独的局部情况。



如上图所示，虽然每个位置都还有能填的数，但是因为图中圈出的两个 B 的影响，导致 B 不可能填入第一行中的任何一个空位。类似的还有其他更加复杂的情况。也就是说，我们需要对数独进行更加全面的可行性判定，尽早发现无解的分支执行回溯。

我们加入以下的可行性剪枝。

1. 遍历当前所有的空格

(1) 若某个位置 A~P 都不能填，立即回溯。

(2) 若某个位置只有 1 个字母可填，立即填上这个字母。

2. 考虑所有的行

(1) 若某个字母不能填在该行的任何一个空位上，立即回溯。

(2) 若某个字母只能填在该行的某一个空位上，立即填写。

3. 考虑所有的列，执行与第 2 步类似的过程。

4. 考虑所有的十六宫格，执行类似的过程。

之后，我们再选择可填的字母最少的位置，枚举填写哪个字母作为分支。

使用位运算来进行常数优化仍然是必要的。不过，因为上述剪枝较为复杂，按照上一节中 9*9 数独的位运算记录方法，实现起来比较困难，所以在本题中，我们直接对每个位置保存一个 16 位二进制数，存储该位置能填的数字情况，一共 16*16=256 个这样的二进制数。在每次递归前，我们简单地把这 256 个数的副本记录在局部变量上，在还原现场时直接恢复即可。本书配套资源中提供了本题的参考代码。

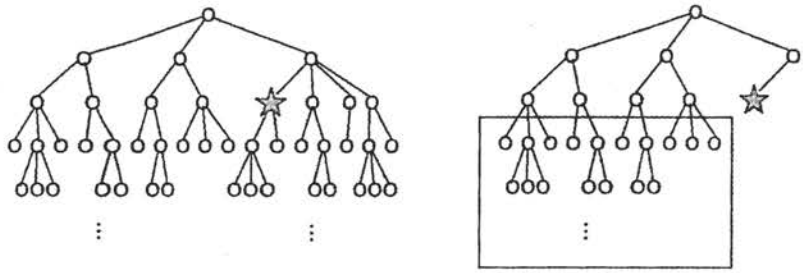
数独可以转化为精确覆盖问题，使用一种叫作 Dancing Links 的数据结构求解。这超出了我们的讨论范围，学有余力的读者可以自行查阅相关资料。不过，建议读者还是使用搜索算法实现上一节与本节中的共计三道数独问题，因为算法思维能力的训练远比学习几个能直接拿来解决问题的数据结构重要。

0x24 迭代加深

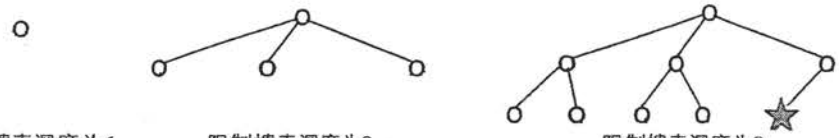
迭代加深

深度优先搜索每次选定一个分支，不断深入，直至到达递归边界才回溯。这种策略带有一定的缺陷。试想以下情况：搜索树每个节点的分支数目非常多，并且问题的答案在某个较浅的节点上。如果深搜在一开始选错了分支，就很可能在不包含答案的深层子树上浪费许多时间。

如果下图左半部分是问题的状态空间，五角星标识着答案，那么深度优先搜索算法产生的搜索树就如下图右半部分所示，算法在矩形圈出的深层子树上浪费了很多时间。



此时，我们可以从小到大限制搜索的深度，如果在当前深度限制下搜不到答案，就把深度限制增加，重新进行一次搜索，这就是**迭代加深**思想。所谓“迭代”，就是以上一次的结果为基础，重复执行以逼近答案的意思。迭代加深 DFS 的过程如下图所示。



虽然该过程在深度限制为 d 时，会重复搜索第 $1\sim d-1$ 层的节点，但是当搜索树节点分支数目较多时，随着层数的深入，每层节点数会呈指数级增长，这点重复搜索与深层子树的规模相比，实在是小巫见大巫了。

总而言之，当**搜索树规模随着层次的深入增长很快，并且我们能够确保答案在一个**

较浅层的节点时，就可以采用迭代加深的深度优先搜索算法来解决问题。读者可以进行大致的估算，有些题目描述甚至会包含“如果10步以内搜不到结果就算无解”的字样。

例题 Addition Chains AcWing170

满足如下条件的序列 X 被称为“加成序列”。

1. $X[1] = 1$

2. $X[m] = n$

3. $X[1] < X[2] < \cdots < X[m-1] < X[m]$

4. 对于每个 $k(2 \le k \le m)$ 都存在两个整数 i 和 j $(1 \le i,j \le k-1$，i 和 j 可以相等)，使得 $X[k] = X[i] + X[j]$。

你的任务是：给定一个整数 n，找出符合上述条件的长度 m 最小的“加成序列”。如果有多个满足要求的答案，只需要找出任意一个可行解。$n \le 100$。

搜索框架：依次搜索序列中的每个位置 k，枚举 i 和 j 作为分支，把 $X[i]$ 和 $X[j]$ 的和填到 $X[k]$ 上，然后递归填写下一个位置。

加入以下剪枝：

1. 优化搜索顺序

为了让序列中的数尽快逼近 n，在枚举 i 和 j 时从大到小枚举。

2. 排除等效冗余

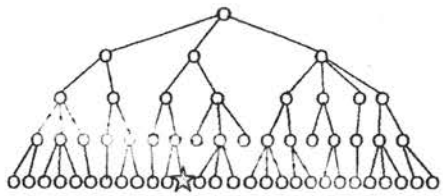
对于不同的 i 和 j，$X[i] + X[j]$ 可能是相等的。我们可以在枚举时用一个 bool 数组对 $X[i] + X[j]$ 进行判重，避免重复搜索某一个和。

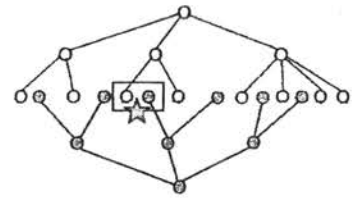
经过观察分析可以发现，m 的值（序列长度）不会太大（≤ 10），而每次枚举两个数的和，分支很多。所以我们采用迭代加深的搜索方式，从 1 开始限制搜索深度，若搜索失败就增加深度限制重新搜索，直到找到一组解时即可输出答案。

双向搜索

除了迭代加深之外，双向搜索也可以避免在深层子树上浪费时间。在一些题目中，问题不但具有“初态”，还具有明确的“终态”，并且从初态开始搜索与从终态开始逆向搜索产生的搜索树都能覆盖整个状态空间。在这种情况下，就可以采用**双向搜索——从初态和终态出发各搜索一半状态，产生两棵深度减半的搜索树，在中间交会、组合成最终的答案。**

如下页图所示，左侧是直接进行一次搜索产生的搜索树，右侧是双向搜索的两棵搜索树，避免了层数过深时分支数量大规模增长。





例题 送礼物 AcWing171

作为惩罚，小 G 被遣送去帮助某神牛给女生送礼物（小 G：貌似是个好差事），但是在小 G 看到礼物之后，他就不这么认为了。该神牛有 $N(N \leq 45)$ 个礼物，其中第 i 个礼物的重量是 $G[i]$。小 G 的力气也异常大，他一次可以搬动重量之和不超过 $W(W \leq 2^{31}-1)$ 的任意多个物品。小 G 希望一次搬掉尽量重的一些物品。请你告诉小 G 在他的力气范围内一次性能搬动的最大重量是多少。

这道题目就是“子集和”问题的扩展——从给定的 N 个数中选择几个，使它们的和最接近 W。了解“背包”问题的读者可能已经发现，这道题目也是一个“大体积”的背包问题。这类问题的直接解法就是进行“指数型”的枚举——搜索每个礼物选还是不选，时间复杂度为 $O(2^N)$。当然，若搜索过程中已选的礼物重量之和已经大于 W，可以及时剪枝。

此题 $N \leq 45$，2^{45} 的复杂度过高。这时我们就可以利用双向搜索的思想，把礼物分成两半。

首先，我们搜索出从前一半礼物中选出若干个，可能达到的 $0 \sim W$ 之间的所有重量值，存放在一个数组 A 中，并对数组 A 进行排序、去重。

然后，我们进行第二次搜索，尝试从后一半礼物中选出一些。对于每个可能达到的重量值 t，在第一部分得到的数组 A 中二分查找 $\leq W-t$ 的数值中最大的一个，用二者的和更新答案。

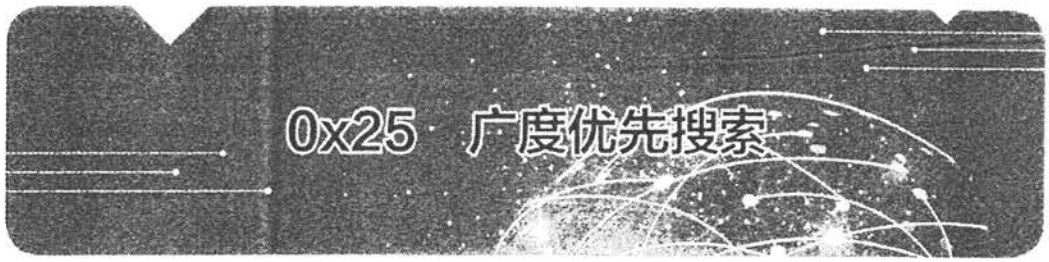
这个算法的时间复杂度就只有 $O(2^{N/2}\log 2^{N/2}) = O(N * 2^{N/2})$ 了。我们还可以加入一些优化，进一步提高算法的效率。

1. 优化搜索顺序

把礼物按照重量降序排序后再分半、搜索。

2. 选取适当的“折半划分点”

因为第二次搜索需要在第一次搜索得到的数组中进行二分查找，效率相对较低，所以我们应该稍微增加第一次搜索的礼物数，减少第二次搜索的礼物数。经过本地随机数据的实验，我们发现取第 $1 \sim N/2+2$ 个礼物为“前一半”，取第 $N/2+3 \sim N$ 个礼物为“后一半”时，搜索的速度最快。具体实现请参阅配套资源中的程序。



0x25 广度优先搜索

在 0x21 节中，我们介绍了图的广度优先遍历，在 0x22 节中，我们又定义了深度优先搜索过程产生的“搜索树”结构。如果我们把问题状态空间类比成一张图，那么广度优先搜索就相当于对这张图的广度优先遍历。类似地，我们依然借助一个队列来实现广度优先搜索，起初，队列中仅包含起始状态。在广度优先搜索的过程中，我们不断从队头取出状态，对于该状态面临的所有分支，把沿着每条分支到达的下一个状态（如果尚未访问过或者能够被更新成更优的解）插入队尾。重复执行上述过程直到队列为空。

例题 **Bloxorz** AcWing172

Bloxorz 是一个风靡世界的小游戏。Bloxorz 的地图是一个 N 行 M 列的矩阵，每个位置可能是硬地（用“.”表示）、易碎地面（用“E”表示）、禁地（用“#”表示）、起点（用“X”表示）或终点（用“O”表示）。你的任务是操作一个 1*1*2 的长方体。这个长方体在地面上有两种放置形式，“立”在地面上（1*1 的面接触地面）或者“躺”在地面上（1*2 的面接触地面）。在每一步操作中，可以按上、下、左、右四个键之一。按下按键之后，长方体向对应的方向沿着棱滚动 90 度。任意时刻，长方体不能有任何部位接触禁地（否则就会掉下去），并且不能立在易碎地面上（否则会因为压强太大掉下去）。字符“X”标识长方体的起始位置，地图上可能有一个“X”或者两个相邻的“X”。地图上唯一的一个字符“O”标识目标位置。求把长方体移动到目标位置（即立在“O”上）所需要的最少步数。如果无解，输出“Impossible”。在移动过程中，“X”和“O”标识的位置都可以看作是硬地被利用。$3 \le N, M \le 500$。

这是一道典型的“**走地图**”**类问题**，也就是形如“给定一个矩形地图，控制一个物体在地图中按要求移动，求最少步数”的问题。广度优先搜索很擅长解决这种问题——地图的整体形态是固定不变的，只有少数个体或特征随着每一步操作发生改变。我们只需要把这些**变化的部分提取为状态**，把起始状态加入队列，使用广度优先搜索不断取出队头状态，沿着分支扩展、入队即可。**广度优先搜索是逐层遍历搜索树的算法，所有状态按照入队的先后顺序具有层次单调性**（也就是步数单调性）。**如果每一次扩展恰好对应一步，那么当一个状态第一次被访问（入队）时，就得到了从起始状态到达该状态的最少步数。**

在这道题目中，不变的是整个地图，变化的部分有长方体的位置和放置形态。我们可以用一个三元组 (x,y,lie) 代表一个状态（搜索树中的一个节点），其中 $lie=0$ 表示长方体立在 (x,y)；$lie=1$ 表示长方体横向躺着，左半部分位置在 (x,y)；$lie=2$ 表示长方体纵向躺着，上半部分在 (x,y)，并用数组 $d[x][y][lie]$ 记录从起始状态到达每个状态的最少步数。然后执行广度优先搜索即可。为了程序实现方便，我们用数字0~3分别代指左、右、上、下四个方向。

```
struct rec { int x, y, lie; };  // 状态
char s[510][510];  // 地图
rec st, ed;  // 起始状态和目标状态
int n, m, d[510][510][3];  // 最少步数记录数组
queue<rec> q;  // 队列
const int dx[4] = { 0,0,-1,1 }, dy[4] = { -1,1,0,0 }; // 方向数组

void parse_st_ed() {  // 处理起点和终点
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            if (s[i][j] == 'O') {
                ed.x = i, ed.y = j, ed.lie = 0, s[i][j] = '.';
            }
            else if (s[i][j] == 'X') {
                for (int k = 0; k < 4; k++) {
                    int x = i + dx[k], y = j + dy[k];
                    if (valid(x, y) && s[x][y] == 'X') {
                        st.x = min(i, x), st.y = min(j, y);
                        st.lie = k<2 ? 1 : 2;
                        s[i][j] = s[x][y] = '.';
                        break;
                    }
                }
                if (s[i][j] == 'X') st.x = i, st.y = j, st.lie = 0;
            }
}

bool valid(int x, int y) { // 是否在地图边界内
    return x>=1 && y>=1 && x<=n && y<=m;
}
bool valid(rec next) {  // 滚动是否合法
```

```
    if (!valid(next.x, next.y)) return 0;
    if (s[next.x][next.y] == '#') return 0;
    if (next.lie == 0 && s[next.x][next.y] != '.') return 0;
    if (next.lie == 1 && s[next.x][next.y + 1] == '#') return 0;
    if (next.lie == 2 && s[next.x + 1][next.y] == '#') return 0;
    return 1;
}

// next_x[i][j]表示在 lie=i 时朝方向 j 滚动后 x 的变化情况
const int next_x[3][4] = { { 0,0,-2,1 },{ 0,0,-1,1 },{ 0,0,-1,2 } }
// next_y[i][j]表示在 lie=i 时朝方向 j 滚动后 y 的变化情况
const int next_y[3][4] = { { -2,1,0,0 },{ -1,2,0,0 },{ -1,1,0,0 } }
// next_lie[i][j]表示在 lie=i 时朝方向 j 滚动后 lie 的新值
const int next_lie[3][4] = { { 1,1,2,2 },{ 0,0,1,1 },{ 2,2,0,0 } };

int bfs() {  // 广搜
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            for (int k = 0; k < 3; k++) d[i][j][k] = -1;
    while (q.size()) q.pop();
    d[st.x][st.y][st.lie] = 0; q.push(st);
    while (q.size()) {
        rec now = q.front(); q.pop();  // 取出队头
        for (int i = 0; i < 4; i++) {  // 向 4 个方向滚动
            rec next;
            next.x = now.x + next_x[now.lie][i];
            next.y = now.y + next_y[now.lie][i];
            next.lie = next_lie[now.lie][i];
            if (!valid(next)) continue;
            if (d[next.x][next.y][next.lie] == -1) {  // 尚未访问过
                d[next.x][next.y][next.lie]
                    = d[now.x][now.y][now.lie] + 1;
                q.push(next);
                if (next.x == ed.x && next.y == ed.y
                                   && next.lie == ed.lie)
                    return d[next.x][next.y][next.lie];  // 到达目标
            }
        }
    }
```

```
    return -1;  // 无解
}

int main() {
    while (cin >> n >> m && n) {
        for (int i = 1; i <= n; i++) scanf("%s", s[i] + 1);
        parse_st_ed();
        int ans = bfs();
        if (ans == -1) puts("Impossible"); else cout << ans << endl;
    }
}
```

提示：在上述代码中，我们使用 next_x, next_y, next_lie 等常数数组预先保存了长方体沿四个方向滚动的变化情况，避免了使用大量 if 语句造成的混乱。这是广搜程序实现中的一个常见技巧。

例题 矩阵距离 AcWing173

给定一个 N 行 M 列的 01 矩阵 A，$A[i][j]$ 与 $A[k][l]$ 之间的曼哈顿距离定义为：

$$\text{dist}(A[i][j], A[k][l]) = |i-k| + |j-l|$$

输出一个 N 行 M 列的整数矩阵 B，其中：

$$B[i][j] = \min_{1 \le x \le N, 1 \le y \le M, A[x][y]=1} \{\text{dist}(A[i][j], A[x][y])\}$$

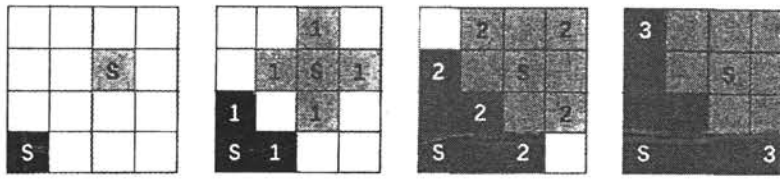
$N, M \le 1000$。

先来思考这样一个问题：给定一个 $N * M$ 的 01 矩阵和一个起点（0 表示可通行点，1 表示禁止点），每一步可以向上、下、左、右四个方向之一移动 1 单位距离，求从起点出发能够到达哪些位置，以及到达每个位置的最少步数。这是广度优先搜索的最基本应用，读者应该都不陌生，我们也称 BFS 求解该问题的算法为 flood-fill，就好像在起点倒水，看能覆盖多大的一片区域。

本题可以看作一道**有多个起始状态**的 flood-fill 问题。我们把矩阵 A 中**每一个 1 都看作起点**，整个矩阵的所有位置都可以通行，对于每个位置，在从任何一个起点出发都可以的情况下，求到达该位置所需要的最少步数（也就是距离该位置最近的起点的距离）。

在这种具有多个等价的起始状态的问题中，我们只需要在 BFS 开始之前把这些**起始状态全部插入队列**。根据 BFS 逐层搜索的性质，BFS 的过程就相当于每个起点先扩展 1 层，再扩展 2 层、3 层，依此类推。所以当每个位置 (x,y) 第一次被访问时，就相当于距离它最近的那个起点扩展到了它，此时从那个起点到 (x,y) 经历的步数就是

最短距离 $B[x][y]$。如下图所示。



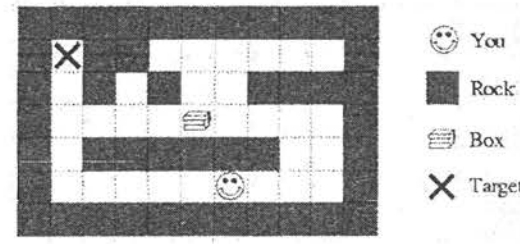
编写广搜代码时，再次提醒读者注意地图**边界的检查**、记录数组 d 的**初始化**，并用**方向常数数组** dx, dy 避免多个 if 分支，用 0~3 代指上、下、左、右四个方向。

```
const int dx[4] = { -1,1,0,0 }, dy[4] = { 0,0,-1,1 };
char s[1020][1020]; int d[1020][1020], n, m;
queue<pair<int, int>> q;
int main() {
    cin >> n >> m;
    for (int i = 1; i <= n; i++) scanf("%s", s[i] + 1);
    memset(d, -1, sizeof(d));
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            if (s[i][j]=='1') q.push(make_pair(i,j)), d[i][j] = 0;
    while (q.size()) {
        pair<int, int> now = q.front(); q.pop();
        for (int k = 0; k < 4; k++) {
            pair<int,int> next(now.first+dx[k], now.second+dy[k]);
            if (next.first<1 || next.second<1 ||
                next.first>n || next.second>m) continue;
            if (d[next.first][next.second] == -1) {
                d[next.first][next.second]
                    = d[now.first][now.second] + 1;
                q.push(next);
            }
        }
    }
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) printf("%d ", d[i][j]);
        puts("");
    }
}
```

例题 **Pushing Boxes** AcWing174

推箱子游戏相信大家都不陌生，在本题中，你将控制一个人把一个箱子推到目的地。给定一张 N 行 M 列的地图，用字符“.”表示空地，字符“#”表示墙，字符“S”表示人的起始位置，字符“B”表示箱子的起始位置，字符“T”表示箱子的目标位置。求一种移动方案，使箱子移动的次数最少，在此基础上再让人移动的总步数最少。方案中使用大写的“EWSN”（东西南北）表示箱子的推动，使用小写的“ewsn”（东西南北）表示人的移动。$N, M \leq 20$。

样例输入：
```
7 11
###########
#T##......#
#.#.#..####
#....B....#
#.######..#
#.....S...#
###########
```



样例输出：
```
eennwwWWWWeeeeeesswwwwwwwnNN
```

在这道题目中，箱子和人都会移动，并且二者的移动可能不是同步的。一个直接的想法是把人和箱子的位置一起组成一个状态，这样的状态是一个四元组，规模是 $O(N^2M^2)$。如果直接使用这个算法进行广搜，会出现一些问题。

1. 题目要求先保证箱子移动的总步数 $step_box$ 最少，再保证人移动的总步数 $step_man$ 最少。根据我们之前提到过的广搜的性质，应该让队列中的状态对应的步数二元组 $(step_box, step_man)$ 保持单调性（即队列中的状态满足第一关键字 $step_box$ 递增，当 $step_box$ 相同时第二关键字 $step_man$ 递增）。

然而，在对每个状态进行扩展时，有的时候箱子和人移动的步数都会增加，有的时候只有人移动的步数会增加。如果像往常一样简单地把新状态放在队尾，就可能会破坏队列中的状态对应“步数二元组”的单调性。

2. 在队列状态不满足步数单调性时，每个状态只访问一次的广搜会求出错误的答案。我们在下一节 0x26“广搜变形”中会提到几种解决方案，比如使用迭代的思想多次更新一个状态，或者改用优先队列进行广搜。然而这些方案都会进一步增加算法的时间复杂度（至少增加一个 log）。本题每个测试点包含很多组数据，上述算法都会导致超时。

进一步分析本问题，我们可以发现，在每次箱子移动后，人一定位于箱子之前处于的位置上。于是，我们可以把每次箱子刚刚移动后，箱子与人的位置打包构成一个状态，

记为 (x,y,dir)，其中 (x,y) 表示箱子处于第 x 行第 y 列，dir 是一个 0~3 之间的整数值，指示人处于箱子旁边四个位置中的哪一个。如果我们像上一道题目一样使用方向常数数组 dx 和 dy，那么人的位置就可以表示为 $(x-dx[dir],\ y-dy[dir])$。

另外，我们使用数组 $f_box[x][y][dir]$ 和 $f_man[x][y][dir]$ 分别记录每个状态下箱子和人已经移动的步数。

对于任意一个状态 (x,y,dir)，都可能有四个分支，代表箱子可能向四个方向移动。我们仍使用方向常数数组，然后枚举方向 $k=0$~3，则扩展到的下一个状态就是 $(x+dx[k],\ y+dy[k],k)$。移动的方式是：

1. 箱子在 (x,y) 处不动，人用尽量少的步数从起点 $(x-dx[dir],\ y-dy[dir])$ 移动到终点 $(x-dx[k],\ y-dy[k])$，并且途中不能经过 (x,y)；

2. 人沿着 k 方向推一次箱子，箱子移动到 $(x+dx[k],\ y+dy[k])$，人移动到 (x,y)。

此后，我们把新状态 $(x+dx[k],\ y+dy[k],k)$ 入队，并存储新状态对应的 f_box 与 f_man 的值。

这是一个**双重** BFS 算法。从整体上看，算法在对每次箱子移动后的“箱子与旁边的人”这个合体进行 BFS。而在这个 BFS 的每一次状态扩展时，我们再用“对人进行的另一个 BFS”求出人在两个位置之间移动的最少步数。

在程序设计时我们常常讲究代码的“模块性”。如果我们把 (x,y,dir) 这个状态扩展时发生的移动写成一个函数 expand，该函数返回箱子和人移动的步数，那么算法本身其实就是一个普普通通的 BFS，只不过它每一步扩展时使用的计算函数 expand 内，恰好又需要使用 BFS 而已。

除了代码编写的“模块性”，读者在理解、设计算法时，也要时刻秉承“模块化”的思想。整个问题有一个大的解决框架，框架的每个细节可能又是一个“小问题”，小问题的解决框架的细节可能还有其他更小的“子问题”，就像一棵树一样逐层深入。读者在设计其中每个问题的算法时，要专注于整体框架，不妨假设细节部分的“小问题”已经处理完毕，争取把每个框架都转化成我们学过的类似“模型”。这样的话，无论多么复杂的问题都会迎刃而解了。

最后，我们还需要讨论上述算法的正确性。对于状态 (x,y,dir) 进行的外层 BFS，在每次状态扩展的移动过程中，箱子一定移动了 1 步，人可能移动了若干步，并且状态中的 dir 相当于移动的方向（还相当于分支来源）。所以，在第二次访问某个状态时，箱子移动的步数一定比第一次多（请读者仔细思考为什么带有 dir 的状态就满足这个性质）。因此，每个状态 (x,y,dir) 只访问一次的这个双重 BFS 算法就可以求出最优解，其状态规模为 $O(NM)$，每次扩展最多花 $O(NM)$ 的时间进行内层 BFS，总体时间复杂度不超过 $O(N^2M^2)$。

本题还要求输出一种具体方案，我们可以额外用数组记录最终的每个 f_box 与 f_man 值是从**上一步的哪个状态更新而来的**（也就是记录搜索树上的父节点），在**求出最优解后向前逆推**得到箱子的运动轨迹。最后我们按照箱子的运动轨迹，在相邻的两次移动之间重新对人进行 BFS，组合起来即可得到最终的方案。这种记录和逆推的方法是各种搜索、动态规划题目输出方案的常用做法。

记为 (x,y,dir)，其中 (x,y) 表示箱子处于第 x 行第 y 列，dir 是一个 0~3 之间的整数值，指示人处于箱子旁边四个位置中的哪一个。如果我们像上一道题目一样使用方向常数数组 dx 和 dy，那么人的位置就可以表示为 $(x-dx[dir],\ y-dy[dir])$。

另外，我们使用数组 $f_box[x][y][dir]$ 和 $f_man[x][y][dir]$ 分别记录每个状态下箱子和人已经移动的步数。

对于任意一个状态 (x,y,dir)，都可能有四个分支，代表箱子可能向四个方向移动。我们仍使用方向常数数组，然后枚举方向 $k=0$~3，则扩展到的下一个状态就是 $(x+dx[k],\ y+dy[k],k)$。移动的方式是：

1. 箱子在 (x,y) 处不动，人用尽量少的步数从起点 $(x-dx[dir],\ y-dy[dir])$ 移动到终点 $(x-dx[k],\ y-dy[k])$，并且途中不能经过 (x,y)；

2. 人沿着 k 方向推一次箱子，箱子移动到 $(x+dx[k],\ y+dy[k])$，人移动到 (x,y)。

此后，我们把新状态 $(x+dx[k],\ y+dy[k],k)$ 入队，并存储新状态对应的 f_box 与 f_man 的值。

这是一个**双重** BFS 算法。从整体上看，算法在对每次箱子移动后的“箱子与旁边的人”这个合体进行 BFS。而在这个 BFS 的每一次状态扩展时，我们再用“对人进行的另一个 BFS”求出人在两个位置之间移动的最少步数。

在程序设计时我们常常讲究代码的“模块性”。如果我们把 (x,y,dir) 这个状态扩展时发生的移动写成一个函数 expand，该函数返回箱子和人移动的步数，那么算法本身其实就是一个普普通通的 BFS，只不过它每一步扩展时使用的计算函数 expand 内，恰好又需要使用 BFS 而已。

除了代码编写的“模块性”，读者在理解、设计算法时，也要时刻秉承“模块化”的思想。整个问题有一个大的解决框架，框架的每个细节可能又是一个“小问题”，小问题的解决框架的细节可能还有其他更小的“子问题”，就像一棵树一样逐层深入。读者在设计其中每个问题的算法时，要专注于整体框架，不妨假设细节部分的“小问题”已经处理完毕，争取把每个框架都转化成我们学过的类似“模型”。这样的话，无论多么复杂的问题都会迎刃而解了。

最后，我们还需要讨论上述算法的正确性。对于状态 (x,y,dir) 进行的外层 BFS，在每次状态扩展的移动过程中，箱子一定移动了 1 步，人可能移动了若干步，并且状态中的 dir 相当于移动的方向（还相当于分支来源）。所以，在第二次访问某个状态时，箱子移动的步数一定比第一次多（请读者仔细思考为什么带有 dir 的状态就满足这个性质）。因此，每个状态 (x,y,dir) 只访问一次的这个双重 BFS 算法就可以求出最优解，其状态规模为 $O(NM)$，每次扩展最多花 $O(NM)$ 的时间进行内层 BFS，总体时间复杂度不超过 $O(N^2M^2)$。

本题还要求输出一种具体方案，我们可以额外用数组记录最终的每个 f_box 与 f_man 值是从上一步的哪个状态更新而来的（也就是记录搜索树上的父节点），在求出最优解后向前逆推得到箱子的运动轨迹。最后我们按照箱子的运动轨迹，在相邻的两次移动之间重新对人进行 BFS，组合起来即可得到最终的方案。这种记录和逆推的方法是各种搜索、动态规划题目输出方案的常用做法。

0x26 广搜变形

双端队列 BFS

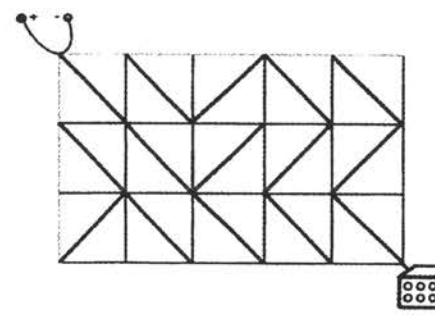
在最基本的广度优先搜索中，每次沿着分支的扩展都记为“一步”，我们通过逐层搜索，解决了求从起始状态到每个状态的最少步数的问题。这其实等价于在一张**边权均为 1 的图**上执行广度优先遍历，求出每个点相对于起点的最短距离（层次）。在第 0x21 节中我们曾讨论过这个问题，并得到了“队列中的状态的层数满足两段性和单调性”的结论。从而我们可以知道，每个状态在**第一次被访问并入队时**，计算出的步数即为所求。

然而，如果图上的边权不全是 1 呢？换句话说，如果每次扩展都有各自不同的“代价”，我们想求出起始状态到每个状态的最小代价，应该怎么办呢？我们不妨先来讨论图上的边权要么是 1、要么是 0 的情况。

例题 电路维修 AcWing175

Ha'nyu 是来自异世界的魔女，她在漫无目的地四处漂流的时候，遇到了善良的少女 Rika，从而被收留在地球上。Rika 的家里有一辆飞行车。有一天飞行车的电路板突然出现了故障，导致无法启动。

电路板的整体结构是一个 R 行 C 列的网格（$R, C \leq 500$），如右图所示。每个格点都是电线的接点，每个格子都包含一个电子元件。电子元件的主要部分是一个可旋转的、连接一条对角线上的两个接点的短电缆。在旋转之后，它就可以连接另一条对角线的两个接点。电路板左上角的接点接入直流电源，右下角的接点接入飞行车的发动装置。



Ha'nyu 发现因为某些元件的方向不小心发生了改变，电路板可能处于断路的状态。她准备通过计算，旋转最少数量的元件，使电源与发动装置通过若干条短缆相连。不过，电路的规模实在是太大了，Ha'nyu 并不擅长编程，希望你能够帮她解决这个问题。

我们可以把电路板上的每个格点（横线与竖线的交叉点）看作无向图中的节点。若两个节点 x 和 y 是某个小方格的两个对角，则在 x 与 y 之间连边。若该方格中的标准件（对角线）与 x 到 y 的线段重合，则边权为 0；若垂直相交，则边权为 1（说明需要旋转 1 次才能连上）。然后，我们在这个无向图中求出左上角到右下角的最短距离，就得到了答案。

这是一张**边权要么是 0、要么是 1** 的无向图。在这样的图上，我们可以通过**双端队列广搜**来计算。算法的整体框架与一般的广搜类似，只是在每个节点上沿分支扩展时稍作改变。如果这条分支是**边权为 0** 的边，就把沿该分支到达的新节点**从队头入队**；如果这条分支是**边权为 1** 的边，就像一般的广搜一样**从队尾入队**。这样一来，我们就仍然能保证，任意时刻广搜队列中的节点对应的距离值都具有“两段性”和“单调性”，每个节点虽然可能被更新（入队）多次，但是它第一次被扩展（出队）时，就能得到从左上角到该节点的最短距离，之后再被取出可以直接忽略，时间复杂度为 $O(R*C)$。

优先队列 BFS

对于更加具有普适性的情况，也就是每次扩展都有各自不同的“代价”时，求出起始状态到每个状态的最小代价，就相当于在一张带权图上求出从起点到每个节点的最短路。此时，我们有两个解决方案。

方法一

仍然使用一般的广搜，采用一般的队列。

这时我们不再能保证每个状态第一次入队时就能得到最小代价，所以只能允许一个状态被多次更新、多次进出队列。我们不断执行搜索，直到队列为空。

整个广搜算法对搜索树进行了重复遍历与更新，直至“收敛”到最优解，其实也就是“迭代”的思想。最坏情况下，该算法的时间复杂度会从一般广搜的 $O(N)$ 增长到 $O(N^2)$。对应在最短路问题中，就是我们将在 0x61 节介绍的 SPFA 算法。

方法二

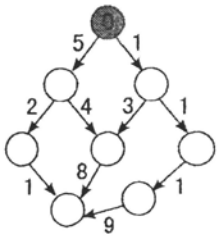
改用优先队列进行广搜。

这里的“优先队列”就相当于一个二叉堆。我们可以每次从队列中**取出当前代价最小的状态进行扩展**（该状态一定已经是最优的，因为队列中其他状态的当前代价都不小于它，所以以后就不可能再更新它了），沿着各条分支把到达的新状态加入优先队列。不断执行搜索，直到队列为空。

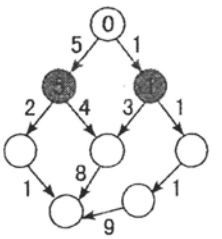
在优先队列 BFS 中，每个状态也会被多次更新、多次进出队列，一个状态也可能以不同的代价在队列中同时存在。不过，当每个状态**第一次从队列中被取出时**，就得到了从起始状态到该状态的最小代价。之后若再被取出，则可以直接忽略，不进行扩展。

所以，优先队列 BFS 中每个状态只扩展一次，时间复杂度只多了维护二叉堆的代价。若一般广搜复杂度为 $O(N)$，则优先队列 BFS 的复杂度为 $O(N \log N)$。对应在最短路问题中，就是我们将在 0x61 节介绍的堆优化的 Dijkstra 算法。

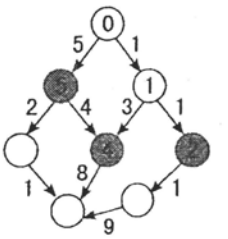
下图就展示了优先队列 BFS 的详细过程，每条边上的数值标明了每个状态向下扩展所需的代价，节点中的数值标明了每个状态上求出的“当前代价”，灰色节点标明了每个时刻在堆中出现至少一次的节点。



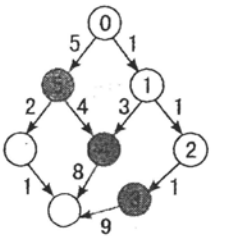
队列：0
取出：0
插入：1 5



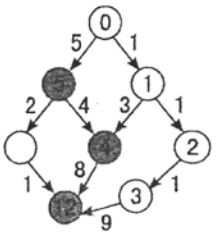
队列：1 5
取出：1
插入：2 4



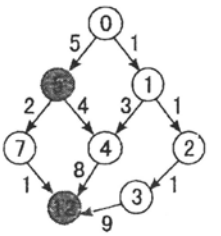
队列：2 4 5
取出：2
插入：3



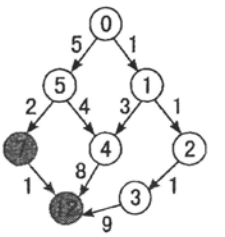
队列：3 4 5
取出：3
插入：12



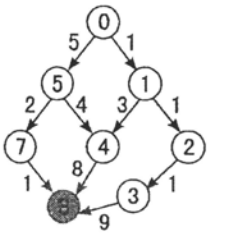
队列：4 5 12
取出：4
插入：12



队列：5 12 12
取出：5
插入：7



队列：7 12 12
取出：7
插入：8



队列：8 12 12
取出：8
得到答案

建议不熟悉最短路相关算法的读者，简要对 0x61 节的两个单源最短路径算法进行阅读，这对于理解并写出正确的优先队列 BFS 有很大的帮助。

至此，我们就可以对 BFS 的形式，按照对应在图上的边权情况进行分类总结。

1. 问题只计最少步数，等价于在边权都为 1 的图上求最短路。

使用普通的 BFS，时间复杂度 $O(N)$。

每个状态只访问（入队）一次，第一次入队时即为该状态的最少步数。

2. 问题每次扩展的代价可能是 0 或 1，等价于在边权只有 0 和 1 的图上求最短路。

使用双端队列 BFS，时间复杂度 $O(N)$。

每个状态被更新（入队）多次，只扩展一次，第一次出队时即为该状态的

最小代价。

3. 问题每次扩展的代价是任意数值，等价于一般的最短路问题。

(1) 使用优先队列 BFS，时间复杂度 $O(N \log N)$。

每个状态被更新（入队）多次，只扩展一次，第一次出队时即为该状态的最小代价。

(2) 使用迭代思想+普通的 BFS，时间复杂度 $O(N^2)$。

每个状态被更新（入队）、扩展（出队）多次，最终完成搜索后，记录数组中保存了最小代价。

例题 Full Tank AcWing176

有 $N(1 \le N \le 1000)$ 个城市和 $M(1 \le M \le 10000)$条道路，构成一张无向图。在每个城市里边都有一个加油站，不同加油站的价格不一样。通过一条道路的油耗就是该道路的边权。现在你需要回答不超过 100 个问题，在每个问题中，请计算出一架油箱容量为 $C(1 \le C \le 100)$ 的车子，从起点 S 开到终点 T 至少要花多少油钱。

我们使用二元组 $(city, fuel)$ 来表示每个状态，其中 $city$ 为城市编号，$fuel$ 为油箱中剩余的汽油量，并使用记录数组 $d[city][fuel]$ 存储最少花费。

对于每个问题，我们都单独进行一次优先队列 BFS。起始状态为 $(S, 0)$。每个状态 $(city, fuel)$ 的分支有：

1. 若 $fuel < C$，可以加 1 升油，扩展到新状态 $(city, fuel + 1)$，花费在城市 $city$ 加 1 升油的钱；

2. 对于每条从 $city$ 出发的边 $(city, next)$，若边权大小 w 不超过 $fuel$，可以开车前往城市 $next$，扩展到新状态 $(next, fuel - w)$。

我们不断取出优先队列中“当前花费最少”的状态（堆顶）进行扩展，更新扩展到的新状态在记录数组 d 中存储的值，直到终点 T 的某个状态第一次被取出，即可停止 BFS，输出答案。

双向 BFS

在 0x24 节中我们介绍了双向搜索的思想，并讲解了一道双向 DFS 的例题。双向 BFS 的思想与之完全相同，我们就不再重复描述。因为 BFS 本身就是逐层搜索的算法，所以双向 BFS 的实现更加自然、简便。以普通的求最少步数的双向 BFS 为例，我们只需从起始状态、目标状态分别开始，**两边轮流进行，每次各扩展一整层**。当两边各自有一个状态在记录数组中发生重复时，就说明这两个搜索过程相遇了，可以合并得出起点到终点的最少步数。

例题 **Nightmare II** AcWing177

给定一张 $N*M$ 的地图，地图中有 1 个男孩、1 个女孩和 2 个鬼。字符“.”表示道路，字符“X”表示墙，字符“M”表示男孩的位置，字符“G”表示女孩的位置，字符“Z”表示鬼的位置。男孩每秒可以在道路上移动 3 个单位距离，女孩每秒可以在道路上移动 1 个单位距离。每个鬼占据的区域每秒可以向四周扩张 2 个单位距离，并且无视墙的阻挡，也就是在第 k 秒后所有与鬼的曼哈顿距离不超过 $2k$ 的位置都会被鬼占领。

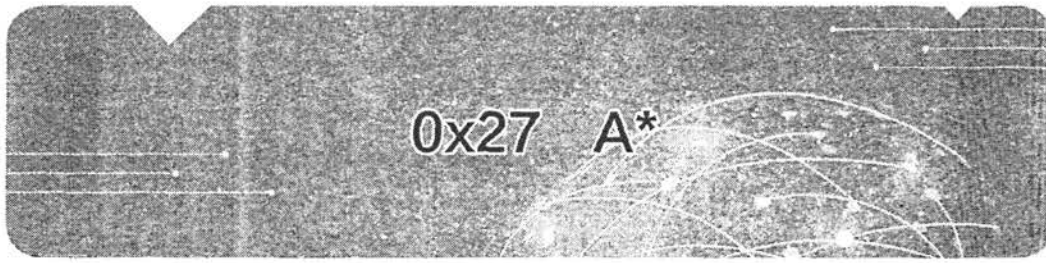
求在不进入鬼的占领区的前提下，男孩与女孩能否会合，若能会合，求出最短会合时间。$1 \le N, M \le 800$。

使用双向 BFS 算法。建立两个队列，分别从男孩的初始位置、女孩的初始位置开始进行 BFS，两边轮流进行。

在每一轮中，男孩这边 BFS 三层（可以移动三步），女孩这边 BFS 一层（可以移动一步），使用数组 d 记录每个位置对于男孩和女孩的可达性。

当然，在 BFS 的每次扩展时，注意实时计算新状态与鬼之间的曼哈顿距离，如果已经小于等于当前轮数（即秒数）的 2 倍，那么就判定这个新状态不合法，不再记录或入队。

在 BFS 的过程中，第一次出现某个位置 (x, y) 既能被男孩到达，也能被女孩到达时，当前轮数就是两人的最短会合时间。



在探讨 A* 算法之前，我们先来回顾一下优先队列 BFS 算法。该算法维护了一个优先队列（二叉堆），不断从堆中取出“当前代价最小”的状态（堆顶）进行扩展。每个状态第一次从堆中被取出时，就得到了从初态到该状态的最小代价。

如果**给定一个“目标状态”**，需要求出从初态到目标状态的最小代价，那么优先队列 BFS 的这个“优先策略”显然是不完善的。一个状态的当前代价最小，只能说明从起始状态到该状态的代价很小，而在未来的搜索中，从该状态到目标状态可能会花费很大的代价。另外，一些状态虽然当前代价略大，但是未来到目标状态的代价可能会很小，于是从起始状态到目标状态的总代价反而更优。优先队列 BFS 会优先选择前者的分支，导致求出最优解的搜索量增大。比如在上一节优先队列 BFS 的示意图中，产生最优解的搜索路径（5+2+1）的后半部分就很晚才得以扩展。

为了提高搜索效率，我们很自然地想到，可以对未来可能产生的代价进行预估。详细地讲，我们设计一个**“估价函数”，以任意“状态”为输入，计算出从该状态到目标状态所需代价的估计值**。在搜索中，仍然维护一个堆，不断从堆中取出**“当前代价+未来估价”**最小的状态进行扩展。

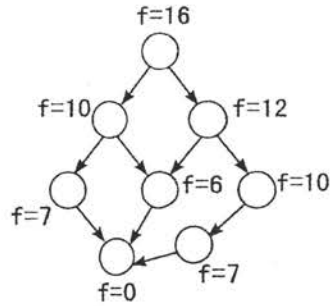
为了保证第一次从堆中取出目标状态时得到的就是最优解，我们设计的估价函数需要满足一个基本准则。

> 设当前状态 state 到目标状态所需代价的估计值为 f(state)。
>
> 设在未来的搜索中，实际求出的从当前状态 state 到目标状态的最小代价为 g(state)。
>
> 对于任意的 state，应该有 $f(state) \le g(state)$。

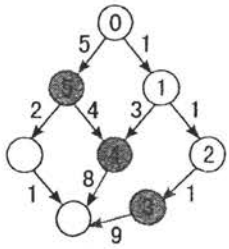
也就是说，估价函数的**估值不能大于未来实际代价**，估价比实际代价更优。

为什么要遵守这个准则呢？我们不妨看看，如果某些估值大于未来实际代价，那么将发生什么情况。

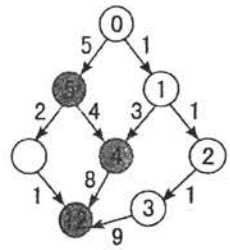
假设我们的估价函数 f 在每个状态上的值如下图所示：



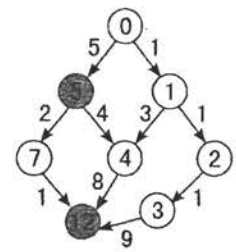
根据“每次取出当前代价+未来估价最小的状态”的策略，会得到如下过程：



队列：3+7, 4+6, 5+10
取出：3+7
插入：12+0



队列：4+6, 12+0, 5+10
取出：4+6
插入：12+0



队列：12+0, 12+0, 5+10
取出：12+0
得到错误答案 12

我们可以看到，本来在最优解搜索路径上的状态被错误地估计了较大的代价，被压在堆中无法取出，从而导致非最优解搜索路径上的状态不断扩展，直至在目标状态上产生错误的答案。

如果我们设计估价函数时遵守上述准则，保证估值不大于未来实际代价，那么即使估价不太准确，导致非最优解搜索路径上的状态 s 先被扩展，但是随着“当前代价”的不断累加，在目标状态被取出之前的某个时刻：

1. 根据 s 并非最优，s 的“当前代价”就会大于从起始状态到目标状态的最小代价；

2. 对于最优解搜索路径上的状态 t，因为 $f(t) \le g(t)$，所以 t 的“当前代价”加上 $f(t)$ 必定小于等于 t 的“当前代价”加上 $g(t)$，而后者的含义就是从起始状态到目标状态的最小代价。

结合以上两点，可知“t 的当前代价加上 $f(t)$”小于 s 的当前代价。因此，t 就会被从堆中取出进行扩展，最终更新到目标状态上，产生最优解。

这种带有估价函数的优先队列 BFS 就称为 A*算法。只要保证对于任意状态 state，都有 $f(state) \le g(state)$，A*算法就一定能在目标状态第一次从堆中被取出时得到最优

解，并且在搜索过程中每个状态只需要被扩展一次（之后再被取出就可以直接忽略）。估价 f(state)越准确、越接近 g(state)，A*算法的效率就越高。如果估价始终为 0，就等价于普通的优先队列 BFS。

A*算法提高搜索效率的关键，就在于能否设计出一个优秀的估价函数。估价函数在满足上述设计准则的前提下，还应该尽可能反映未来实际代价的变化趋势和相对大小关系，这样搜索才会较快地逼近最优解。接下来我们通过两道例题来具体地感受一下 A*算法。

例题 **第 K 短路 Remmarguts' Date** AcWing178

给定一张 N 个点，M 条边的有向图，求从起点 S 到终点 T 的第 K 短路的长度，路径允许重复经过点或边。其中 $1 \le S,T \le N \le 1000$, $0 \le M \le 10^5$, $1 \le K \le 1000$。

注：不熟悉最短路的读者可以简单浏览第 0x61 节，或者假设我们能够在一张图上求出单源最短路（从一个点出发到其他所有点的最短路径长度），然后在此基础上考虑本题。

一个比较直接的想法是使用优先队列 BFS 进行求解。优先队列（堆）中保存一些二元组$(x, dist)$，其中 x 为节点编号，$dist$ 表示从 S 沿着某条路径到 x 的距离。

起初，堆中只有 $(S,0)$。我们不断从堆中取出 $dist$ 值最小的二元组 $(x, dist)$，然后沿着从 x 出发的每条边 (x,y) 进行扩展，把新的二元组 $\big(y, dist + \text{length}(x,y)\big)$ 插入到堆中（无论堆中是否已经存在一个节点编号为 y 的二元组）。

上一节我们已经讲到，**在优先队列 BFS 中**，某个状态第一次从堆中被取出时，就得到了从初态到它的最小代价。读者用数学归纳法很容易得到一个推论：**对于任意正整数 i 和任意节点 x，当第 i 次从堆中取出包含节点 x 的二元组时，对应的 $dist$ 值就是从 S 到 x 的第 i 短路**。所以，当扩展到的节点 y 已经被取出 K 次时，就没有必要再插入堆中了。最后当节点 T 第 K 次被取出时，就得到了 S 到 T 的第 K 短路。

使用优先队列 BFS 在最坏情况下的复杂度为 $O(K*(N+M)*\log(N+M))$。这道题目给定了起点和终点，求长度最短（代价最小）的路径，可以考虑使用 A* 算法提高搜索效率。

根据估价函数的设计准则，在第 K 短路中从 x 到 T 的估计距离 $f(x)$ 应该不大于第 K 短路中从 x 到 T 的实际距离 $g(x)$。于是，我们可以**把估价函数 $f(x)$ 定为从 x 到 T 的最短路长度**，这样不但能保证 $f(x) \le g(x)$，还能顺应 $g(x)$ 的实际变化趋势。最终我们得到了以下 A* 算法。

1. 预处理出各个节点 x 到终点 T 的最短路长度 $f(x)$ ——这等价于在反向图上以 T 为起点求解单源最短路径问题，可以在 $O\big((N+M)\log(N+M)\big)$ 的时间内完成。

2. 建立一个二叉堆，存储一些二元组 $(x, dist + \mathrm{f}(x))$，其中 x 为节点编号，$dist$ 表示从 S 到 x 当前走过的距离。起初堆中只有 $(S, 0 + \mathrm{f}(0))$。

3. 从二叉堆中取出 $dist + \mathrm{f}(x)$ 值最小的二元组 $(x, dist + \mathrm{f}(x))$，然后沿着从 x 出发的每条边 (x, y) 进行扩展。如果节点 y 被取出的次数尚未达到 K，就把新的二元组 $(y, dist + \mathrm{length}(x, y) + \mathrm{f}(y))$ 插入堆中。

4. 重复第 2~3 步，直至第 K 次取出包含终点 T 的二元组，此时二元组中的 $dist$ 值就是从 S 到 T 的第 K 短路。

A* 算法的复杂度上界与优先队列 BFS 相同。不过，因为估价函数的作用，图中很多节点访问次数都远小于 K，上述 A* 算法已经能够比较快速地求出结果。

例题 八数码 Eight AcWing179

在 0x05 节的“奇数码问题”中，我们已经描述了八数码的规则。在本题中，你需要求出八数码的最少移动步数，并给出一种方案。

先进行可解性判定。我们在 0x05 节中已经提到过，把除空格之外的所有数字排成一个序列，求出该序列的逆序对数。如果初态和终态的逆序对数奇偶性相同，那么这两个状态互相可达，否则一定不可达。

若问题有解，我们就采用 A* 算法搜索一种移动步数最少的方案。通过观察可以发现，每次移动只能把一个数字与空格交换位置，这样至多把一个数字向它在目标状态中的位置移近一步。即使每一步移动都是有意义的，从任何一个状态到目标状态的移动步数也不可能小于所有数字当前位置与目标位置的曼哈顿距离之和。

于是，对于任意状态 $state$，我们可以把估价函数设计为所有数字在 $state$ 中的位置与目标状态 end 中的位置的曼哈顿距离之和，即：

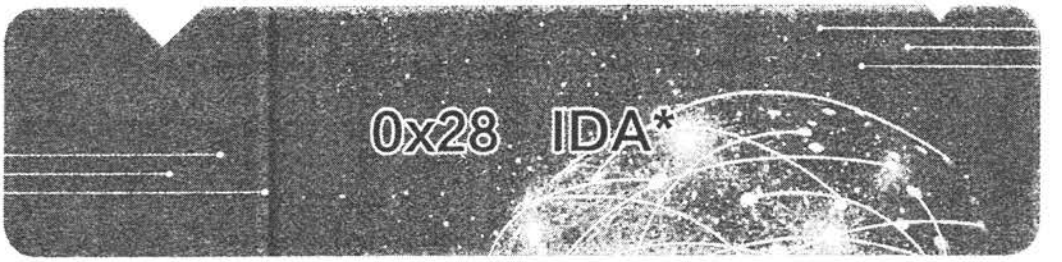
$$f(state) = \sum_{num=1}^{9} (|state_x_{num} - end_x_{num}| + |state_y_{num} - end_y_{num}|)$$

其中 $state_x_{num}$ 表示在状态 $state$ 下数字 num 的行号，$state_y_{num}$ 为列号。

我们不断从堆中取出“从初态到当前状态 $state$ 已经移动的步数+$f(state)$”最小的状态进行扩展，当终态第一次被从堆中取出时，就得到了答案。

在 A* 算法中，为了保证效率，每个状态只需要在第一次被取出时扩展一次。本题中的状态是一个八数码，并非一个简单的节点编号，所以需要使用 Hash 来记录每个八数码是否已经被取出并扩展过一次。我们可以选择取模配合开散列处理冲突，或 STL map 等 Hash 方法。另外，有一种名为“康托展开”的 Hash 方法，可以对全排列进行编码和解码，把 $1 \sim N$ 排成的序列与 $1 \sim N!$ 之间的整数建立一一映射关系，非常适合八数码的 Hash，感兴趣的读者可以自行查阅相关资料。

本书配套资源中的参考程序实现了一个最基本的 A* 算法。虽然它没有用“逆序对的结论判断是否有解，也没有用高效的线性 Hash 方法（直接采用了 STL map，参见第 0x71 节），但足以在规定的时限内完成求解。读者可以进一步优化，提高程序的效率。



在上一节中我们提到，A* 算法本质上是带有估价函数的优先队列 BFS 算法。故 A* 算法有一个显而易见的缺点，就是需要维护一个二叉堆（优先队列）来存储状态及其估价，耗费空间较大，并且对堆进行一次操作也要花费 $O(\log N)$ 的时间。

我们也提到了 A* 算法的关键在于设计估价函数。既然估价函数与优先队列 BFS 结合可以产生 A* 算法，那么估价函数能否与 DFS 结合呢？当然，DFS 也有一个缺点，就是一旦估价出现失误，容易向下递归深入一个不能产生最优解的分支，浪费许多时间。

综合以上讨论，我们最终选择把估价函数与迭代加深的 DFS 算法相结合。请读者回忆 0x24 节中学习的迭代加深 DFS 算法。该算法限定一个深度，在不超过该深度的前提下执行 DFS，若找不到解就扩大深度限制，重新进行搜索。

我们设计一个估价函数，估算从每个状态到目标状态需要的步数。当然，与 A* 算法一样，估价函数需要遵守“估计值不大于未来实际步数”的准则。然后，以迭代加深 DFS 的搜索框架为基础，把原来简单的深度限制加强为：**若当前深度+未来估计步数>深度限制，则立即从当前分支回溯。**

这就是 IDA* 算法（迭代加深的 A* 算法）。IDA* 算法在许多场景下表现出了优秀的效率，并且程序实现的难度低于 A* 算法。

例题 **Booksort** AcWing180

给定 $n(1 \le n \le 15)$ 本书，编号为 $1 \sim n$。在初始状态下，书是任意排列的。在每一次操作中，可以抽取其中连续的一段，再把这段插入到其他某个位置。我们的目标状态是把书按照 $1 \sim n$ 的顺序依次排列，求最少需要多少次操作。若操作次数 ≥ 5，则直接输出字符串“5 or more”即可。

我们先来估算一下每个状态的分支数量。在每个状态下，我们可以抽取连续的一段书，移动到另一个位置。对于任意整数 i，当抽取长度为 i 时，有 $n-i+1$ 种选择方法，有 $n-i$ 个可插入的位置。另外，把一段书移动到更靠前的某个位置，等价于把“跳过”的那段书移动到靠后的某个位置，所以上面的计算方法把每种移动情况算了两遍。每个状态的分支数量约为：$\sum_{i=1}^{n}(n-i)*(n-i+1)/2 \le (15*14+14*13+\cdots+2*1)/2 = 560$。

根据题目要求，我们只需要考虑在 4 次操作以内是否能实现目标，也就是我们只需要考虑搜索树的前 4 层。4 层的搜索树的规模能够达到 560^4，无法承受。

第一种解法是采用双向 BFS，从起始状态、目标状态开始各搜索 2 步，看能否到达相同的状态进行衔接，复杂度降低为 560^2。

第二种解法是采用 IDA*。

在目标状态下，第 i 本书后边应该是第 $i+1$ 本书，我们称 $i+1$ 是 i 的正确后继。

对于任意状态，考虑整个排列中书的错误后继的总个数（记为 tot），可以发现每次操作至多更改 3 本书的后继。即使在最理想的情况下，每次操作都能把某 3 个错误后继全部改对，消除所有错误后继的操作次数也至少需要 $\lceil tot/3 \rceil$（其中 $\lceil\ \rceil$ 表示向上取整）次。

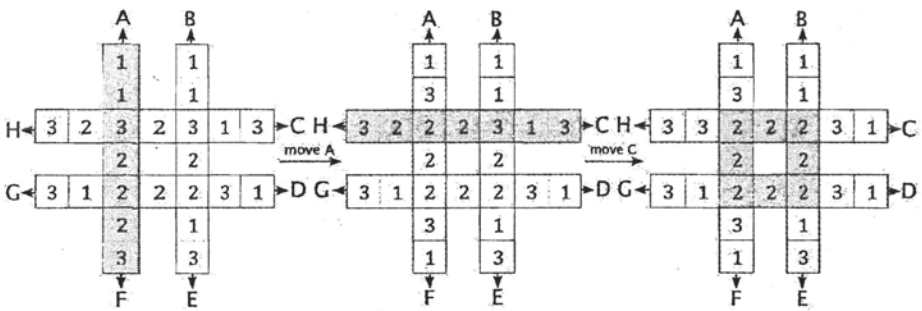
因此，我们把一个状态 s 的估价函数设计为 $f(s)=\lceil tot/3 \rceil$，其中 tot 表示在状态 s 下书的错误后继总数。

我们采用迭代加深的方法，从 1~4 依次限制搜索深度，然后从起始状态出发 DFS。DFS 时，在每个状态下直接枚举抽取哪一段、移动到更靠后的哪个位置，沿着该分支深入即可。注意在进入任何状态 s 后，我们先进行判断，如果当前操作次数加上 $f(s)$ 已经大于深度限制，直接从当前分支回溯。在实际测试中，上述 IDA* 算法能够比双向 BFS 更快地求出答案。

例题 **The Rotation Game** AcWing181

如下图所示，有一个“#”形的棋盘，上面有 1，2，3 三种数字各 8 个。给定 8 种操作，分别为图中的 A~H。这些操作会按照图中字母和箭头所指明的方向，把一条长为 8 的序列循环移动 1 个单位。例如下图最左边的“#”形棋盘执行操作 A 后，会变为下图中间的“#”形棋盘，再执行操作 C 后会变成下图最右边的“#”形棋盘。

给定一个初始状态，请使用最少的操作次数，使“#”形棋盘最中间的 8 个格子里的数字相同。



可以使用 IDA* 算法求解。首先我们来确定 DFS 的框架——在每个状态下枚举执

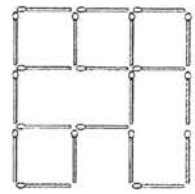
行哪种操作，然后沿着该分支深入即可。有一个很明显的剪枝是记录上一次的操作，不执行上次操作的逆操作，避免来回搜索。

接下来我们设计估价函数。通过仔细观察可以发现，在每个状态下，如果中间 8 个格子里出现次数最多的数字是 k，而中间 8 个格子里剩下的数字有 m 个与 k 不同，那么把中间 8 个格子里的数字都变为 k 至少需要 m 次操作。因此，我们就以这个 m 为估价即可。

总而言之，我们采用迭代加深，由 1 开始从小到大依次限制操作次数（搜索深度），在 DFS 的每个状态下，若“当前操作次数+估价>深度限制”，则从当前分支回溯。

例题 **Square Destroyer** AcWing182

给定一个“在由火柴棒拼成的 $n*n(n\leq 5)$ 的网格图中去掉一些边”构成的图形（如右图所示）。求至少再去掉多少根火柴棒，可以使得图形中不含有正方形（内部有其他火柴的大正方形也算，例如右图中有 5 个正方形）。



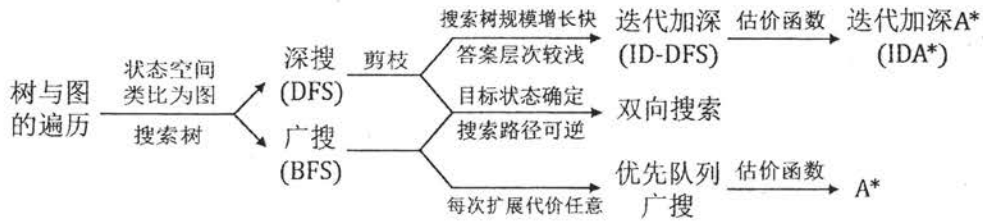
DFS 框架：在每个状态下找出一个最小的正方形，枚举去掉它边界上的哪一根火柴棒，然后沿着该分支深入。

估价函数设计：不断从当前图形中任选一个还没有被破坏的正方形，去掉它边界上的所有火柴棒，但是只记作一次操作。按照上述方法统计出破坏所有正方形需要多少次操作，作为“从当前图形到不含有正方形的图形需要去掉的火柴棒数量”的估计值。这个估值显然不会大于未来实际需要去掉的火柴棒数。

最后执行迭代加深的 A* 算法（IDA*）即可快速求出答案。值得提醒的是，本题作为一道选做题，代码实现中火柴棒编号、图形样式的处理较为繁琐，读者需要格外仔细。

0x29 总结与练习

在本章中，我们主要按照以下脉络探讨了各种搜索算法：



树与图的遍历是学习第 0x40、0x50、0x60 章的基础，读者务必熟练掌握在树上利用深度优先遍历和在有向无环图上按照拓扑序进行若干信息的统计。

深度优先搜索和广度优先搜索是解决各类问题的一般性方法，是搜索的最基础内容。设计并实现基本的 DFS/BFS 框架，也是程序设计的基本能力。为了提高搜索算法的效率，我们在 0x23 节提到了五类常见剪枝手法，其主要思想是降低搜索树的分支数量，及时发现并避免重复和无效的分支搜索。基于搜索树的层次限制，我们又在 0x24 节提及了迭代加深和双向搜索两种思想。另外，请读者再次回顾 0x26 节中我们列出的根据每次扩展代价的情况对应的几种不同的广搜形式，这对于读者理解广搜和 A*算法有很大的帮助。

A* 和 IDA* 是入门级的启发式搜索算法，虽然搜索框架基于 BFS/DFS，但关键在于设计出优秀的估价函数。在满足估价不大于未来实际代价这个基本准则的前提下，如何想出更确切的估价函数，就只能考验读者的见识和思维水平了。

值得提出的是，本章的部分例题和习题代码量较大，对于初学者来说不容易实现。希望读者在学习本章知识点的同时，多花时间尝试独立实现各道题目的具体编程工作，提高自己的代码实现能力——这也是我们设计本章内容的初衷之一。

本章知识点

树与图的遍历

树与图的深度优先遍历，树的 DFS 序、深度、重心，图的连通块划分

树与图的广度优先遍历，拓扑排序，bitset 优化的可达性统计

深度优先搜索

深搜的三种基本递归形式，经典的子集和、全排列、N 皇后问题等

深搜框架的设计与实现，搜索树理论

剪枝

剪枝的设计思想与实现

优化搜索顺序、排除等效冗余、可行性与最优性剪枝、记忆化等常见剪枝手法

迭代加深

迭代加深思想，迭代加深 DFS（ID-DFS）

双向搜索思想，双向 DFS

广度优先搜索

广搜框架的设计与实现，熟练使用记录数组判重、方向常数数组等

广搜的常见问题类型，如走地图、多起点 BFS、双重 BFS 等

广搜变形

双端队列 BFS，双向 BFS

优先队列 BFS，理解并能根据每次扩展代价的实际情况选择正确的 BFS 形式

A*

估价函数的设计准则，估值 f(state) ≤ 未来实际代价 g(state)

A* 算法的实现，A* = 优先队列 BFS + 估价函数

IDA*

IDA* 算法的实现，IDA* = 迭代加深 DFS + 估价函数

练习

1. 完成本章中所有例题的代码实现（标有*的题目选做）
2. 靶形数独

题目地址：AcWing183，提示：深度优先搜索/剪枝/位运算优化

3. 虫食算

题目地址：AcWing184，提示：深度优先搜索/剪枝

4. Mayan 游戏

题目地址：AcWing185，提示：深度优先搜索

5. *The Buses

题目地址：AcWing186，提示：深度优先搜索/迭代加深

6. *Missile Defending System

题目地址：AcWing187，提示：深度优先搜索/迭代加深

7. 武士风度的牛

题目地址：AcWing188，提示：广度优先搜索

8. 乳草的入侵

题目地址：AcWing189，提示：广度优先搜索

9. 字串变换

题目地址：AcWing190，提示：广度优先搜索/双向广搜

10. Weather Forecast

题目地址：AcWing191，提示：广度优先搜索

11. *Bloxorz II

题目地址：AcWing192，提示：广度优先搜索/分情况讨论/数学计算

12. Power Hungry Cow

题目地址：AcWing193，提示：A*

13. Flood-it!

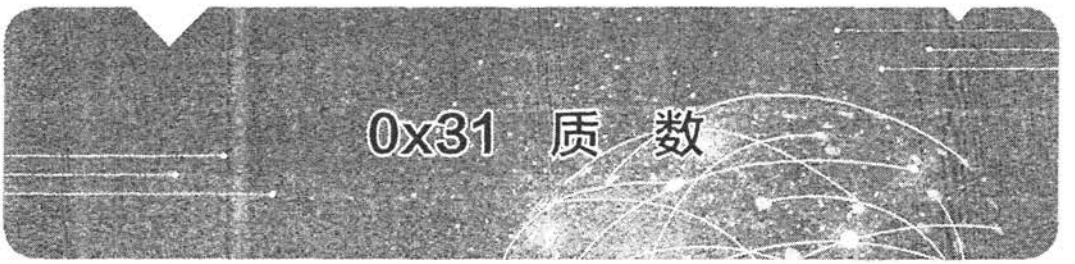
题目地址：AcWing194，提示：IDA*

14. *骑士精神

题目地址：AcWing195，提示：IDA*

0x30 数学知识

本章讲解算法竞赛中常用的数学知识，主要分为数论、线性代数、组合数学、概率论、博弈论五个部分。0x31~0x33 节介绍数论，包括质数、约数、同余以及相关内容。0x34~0x35 节介绍线性代数，包括矩阵、消元、线性变换、线性空间等。0x36~0x37 节介绍组合数学，包括组合计数、容斥原理以及相关的重要数列、函数。0x38 节介绍概率和数学期望。0x39 节介绍 0/1 分数规划模型。0x3A 节介绍博弈的基本概念和 SG 函数的应用。与传统的数学学科教材相比，本章比较注重数学内容与程序设计的结合应用，故而会忽略部分证明过程。如果读者有充足的时间并希望进一步深入，选购一本数学类工具书或利用网络学习资源作为配套资料，将有助于构建更加严谨的数学思维和逻辑体系。



定义

若一个正整数无法被除了 1 和它自身之外的任何自然数整除，则称该数为质数（或素数），否则称该正整数为合数。

在整个自然数集合中，质数的数量不多，分布比较稀疏，对于一个足够大的整数 N，不超过 N 的质数大约有 $N/\ln N$ 个，即每 $\ln N$ 个数中大约有 1 个质数。

质数的判定

试除法

若一个正整数 N 为合数，则存在一个能整除 N 的数 T，其中 $2 \le T \le \sqrt{N}$。

证明：

由定义可得，因为 N 是合数，所以存在一个能整除 N 的数 M，其中 $2 \le M \le N-1$。

反证法。假设命题不成立，那么这样的数 M 一定满足 $\sqrt{N}+1 \le M \le N-1$。

因为 M 能整除 N，所以它们的商 N/M 也能整除 N。而 $2 \le N/M \le \sqrt{N}$，令 $T = N/M$，这与假设矛盾。故假设不成立，原命题成立。

证毕。

根据上述命题，我们只需要扫描 $2 \sim \sqrt{N}$ 之间的所有整数，依次检查它们能否整除 N，若都不能整除，则 N 是质数，否则 N 是合数。试除法的时间复杂度为 $O(\sqrt{N})$。当然，我们需要特判 0 和 1 这两个整数，它们既不是质数，也不是合数。

```
bool is_prime(int n) {
    if (n < 2) return false;
    for (int i = 2; i <= sqrt(n); i++)
        if (n % i == 0) return false;
    return true;
}
```

“试除法”作为最简单也最经典的确定性算法，是我们在算法竞赛中通常会使用的方法。有一些效率更高的随机性算法，例如“Miller-Robbin”等，有较小的概率把合数错误判定为质数，但多次判定合起来的错误概率趋近于零，感兴趣的读者可以自行查阅并学习。

质数的筛选

给定一个整数 N，求出 $1\sim N$ 之间的所有质数，称为质数的筛选问题。

Eratosthenes 筛法

Eratosthenes 筛法基于这样的想法：任意整数 x 的倍数 $2x, 3x, \cdots$ 都不是质数。根据质数的定义，上述命题显然成立。

我们可以从 2 开始，由小到大扫描每个数 x，把它的倍数 $2x, 3x, \cdots, \lfloor N/x \rfloor * x$ 标记为合数。当扫描到一个数时，若它尚未被标记，则它不能被 $2\sim x-1$ 之间的任何数整除，该数就是质数。

Eratosthenes 筛法的进行过程如下：

2, 3, ~~4,~~ 5, ~~6,~~ 7, ~~8,~~ 9, ~~10,~~ 11, ~~12,~~ …
2, 3, ~~4,~~ 5, ~~6,~~ 7, ~~8, 9, 10,~~ 11, ~~12,~~ …
2, 3, ~~4,~~ 5, ~~6,~~ 7, ~~8, 9, 10,~~ 11, ~~12,~~ …
2, 3, ~~4,~~ 5, ~~6,~~ 7, ~~8, 9, 10,~~ 11, ~~12,~~ …
2, 3, ~~4,~~ 5, ~~6,~~ 7, ~~8, 9, 10,~~ 11, ~~12,~~ …
…

另外，我们可以发现，2 和 3 都会把 6 标记为合数。实际上，小于 x^2 的 x 的倍数在扫描更小的数时就已经被标记过了。因此，我们可以对 Eratosthenes 筛法进行优化，对于每个数 x，我们只需要从 x^2 开始，把 $x^2, (x+1)*x, \cdots, \lfloor N/x \rfloor * x$ 标记为合数即可。

```
void primes(int n) {
    memset(v, 0, sizeof(v)); // 合数标记
    for (int i = 2; i <= n; i++) {
        if (v[i]) continue;
        cout << i << endl; // i 是质数
        for (int j = i; j <= n/i; j++) v[i*j] = 1;
    }
}
```

Eratosthenes 筛法的时间复杂度为 $O\left(\sum_{质数 p\le N}\frac{N}{p}\right) = O(N \log\log N)$。该算法实现

简单，效率已经非常接近于线性，是算法竞赛中最常用的质数筛法。

线性筛法

即使在优化后（从 x^2 开始），Eratosthenes 筛法仍然会重复标记合数。例如：12 既会被 2 又会被 3 标记，在标记 2 的倍数时，12=6*2；在标记 3 的倍数时，12=4*3。其根本原因是我们没有确定出唯一的产生 12 的方式。

线性筛法通过“从大到小累积质因子”的方式标记每个合数，即让 12 只有 3*2*2 一种产生方式。设数组 v 记录每个数的最小质因子，我们按照以下步骤维护 v。

1. 依次考虑 $2\sim N$ 之间的每一个数 i。

2. 若 $v[i]=0$，说明 i 是质数，把它保存下来。

3. 扫描不大于 $v[i]$ 的每个质数 p，令 $v[i*p]=p$。也就是在 i 的基础上累积一个质因子 p。因为 $p\le v[i]$，所以 p 就是合数 $i*p$ 的最小质因子。

i	2	3	4	5	6	7	8	9	10
$p\le v[i]$	2	2,3	2	2,3,5	2	2,3,5,7	2	2,3	2
$i*p$	4	6,9	8	10,15,25	12	14,21,35,49	16	18,27	20

每个合数 $i*p$ 只会被它的最小质因子 p 筛一次，时间复杂度为 $O(N)$。

```
int v[MAX_N], prime[MAX_N];
void primes(int n) {
    memset(v, 0, sizeof(v)); // 最小质因子
    m = 0; // 质数数量
    for (int i = 2; i <= n; i++) {
        if (v[i] == 0) { v[i] = i; prime[++m] = i; } // i 是质数
        // 给当前的数 i 乘上一个质因子
        for (int j = 1; j <= m; j++) {
            // i 有比 prime[j]更小的质因子，或者超出 n 的范围，停止循环
            if (prime[j] > v[i] || prime[j] > n/i) break;
            // prime[j]是合数 i*prime[j]的最小质因子
            v[i*prime[j]] = prime[j];
        }
    }
    for (int i = 1; i <= m; i++) cout << prime[i] << endl;
}
```

质因数分解

算术基本定理

任何一个大于 1 的正整数都能唯一分解为有限个质数的乘积，可写作：

$$N = p_1^{c_1} p_2^{c_2} \cdots p_m^{c_m}$$

其中 c_i 都是正整数，p_i 都是质数，且满足 $p_1 < p_2 < \cdots < p_m$。

试除法

结合质数判定的“试除法”和质数筛选的“Eratosthenes 筛法”，我们可以扫描 $2 \sim \lfloor \sqrt{N} \rfloor$ 的每个数 d，若 d 能整除 N，则从 N 中除掉所有的因子 d，同时累计除去的 d 的个数。

因为一个合数的因子一定在扫描到这个合数之前就从 N 中被除掉了，所以在上述过程中能整除 N 的一定是质数。最终就得到了质因数分解的结果，易知时间复杂度为 $O(\sqrt{N})$。

特别地，若 N 没有被任何 $2 \sim \sqrt{N}$ 的数整除，则 N 是质数，无需分解。

```
void divide(int n) {
    m = 0;
    for (int i = 2; i <= sqrt(n); i++) {
        if (n % i == 0) { // i 是质数
            p[++m] = i, c[m] = 0;
            while (n % i == 0) n /= i, c[m]++; // 除掉所有的 i
        }
    }
    if (n > 1) // n 是质数
        p[++m] = n, c[m] = 1;
    for (int i = 1; i <= m; i++)
        cout << p[i] << '^' << c[i] <<endl;
}
```

“Pollard's Rho”算法是一个比“试除法”效率更高的质因数分解算法，不过这超出了我们的探讨范围，感兴趣的读者可以自行查阅并学习。

例题 **Prime Distance** AcWing196

给定两个整数 L, R $(1 \le L \le R \le 2^{31}, R - L \le 10^6)$，求闭区间 $[L, R]$ 中相邻两个质数的差最大是多少，输出这两个质数。

L,R 的范围很大，任何已知算法都无法在规定时间内生成 $[1,R]$ 中的所有质数。但是 $R-L$ 的值很小，并且任何一个合数 n 必定包含一个不超过 $\sqrt{n}$ 的质因子。

所以，我们只需要用筛法求出 $2\sim\sqrt{R}$ 之间的所有质数。对于每个质数 p，把 $[L,R]$ 中能被 p 整除的数标记，即标记 $i*p\left(\left\lceil\frac{L}{p}\right\rceil\le i\le\left\lfloor\frac{R}{p}\right\rfloor\right)$ 为合数。

最终所有未被标记的数就是 $[L,R]$ 中的质数。对相邻的质数两两比较，找出差最大的即可。

时间复杂度为 $O\left(\sum_{质数p\le\sqrt{R}}\frac{R-L}{p}\right)=O\left(\sqrt{R}\log\log\sqrt{R}+(R-L)\log\log R\right)$。

例题 阶乘分解 AcWing197

给定整数 $N(1\le N\le 10^6)$，试把阶乘 $N!$ 分解质因数，按照算术基本定理的形式输出分解结果中的 p_i 和 c_i 即可。

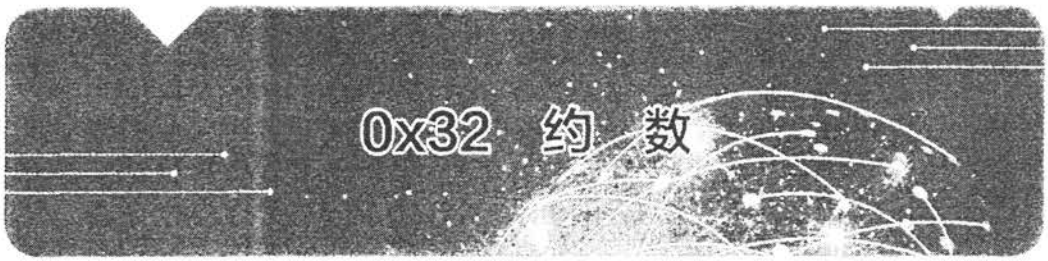
若把 $1\sim N$ 每个数分别分解质因数，再把结果合并，时间复杂度过高，为 $O(N\sqrt{N})$。显然，$N!$ 的每个质因子都不会超过 N，我们可以先筛选出 $1\sim N$ 的每个质数 p，然后考虑阶乘 $N!$ 中一共包含多少个质因子 p。

$N!$ 中质因子 p 的个数就等于 $1\sim N$ 每个数包含质因子 p 的个数之和。在 $1\sim N$ 中，p 的倍数，即至少包含 1 个质因子 p 的显然有 $\lfloor N/p\rfloor$ 个。而 p^2 的倍数，即至少包含 2 个质因子 p 的有 $\lfloor N/p^2\rfloor$ 个。不过其中的 1 个质因子已经在 $\lfloor N/p\rfloor$ 里统计过，所以只需要再统计第 2 个质因子，即累加上 $\lfloor N/p^2\rfloor$，而不是 $2*\lfloor N/p^2\rfloor$。

综上所述，$N!$ 中质因子 p 的个数为：

$$\left\lfloor\frac{N}{p}\right\rfloor+\left\lfloor\frac{N}{p^2}\right\rfloor+\left\lfloor\frac{N}{p^3}\right\rfloor+\cdots+\left\lfloor\frac{N}{p^{\lfloor\log_p N\rfloor}}\right\rfloor=\sum_{p^k\le N}\left\lfloor\frac{N}{p^k}\right\rfloor$$

$1\sim N$ 中质数个数约为 $O(N/\log N)$ 个，对于每个 p，只需要 $O(\log N)$ 的时间计算上述和式。故对整个 $N!$ 分解质因数的时间复杂度为 $O(N)$。



定义

若整数 n 除以整数 d 的余数为 0，即 d 能整除 n，则称 d 是 n 的约数，n 是 d 的倍数，记为 $d|n$。

算术基本定理的推论

在算术基本定理中，若正整数 N 被唯一分解为 $N = p_1^{c_1} p_2^{c_2} \cdots p_m^{c_m}$，其中 c_i 都是正整数，p_i 都是质数，且满足 $p_1 < p_2 < \cdots < p_m$，则 N 的正约数集合可写作：

$$\{p_1^{b_1} p_2^{b_2} \cdots p_m^{b_m}\}, \text{其中 } 0 \le b_i \le c_i$$

N 的正约数个数为（$\prod$ 表示连乘积符号，与 $\sum$ 类似）：

$$(c_1 + 1) * (c_2 + 1) * \cdots * (c_m + 1) = \prod_{i=1}^{m} (c_i + 1)$$

N 的所有正约数的和为：

$$\left(1 + p_1 + p_1^2 + \cdots + p_1^{c_1}\right) * \cdots * \left(1 + p_m + p_m^2 + \cdots + p_m^{c_m}\right) = \prod_{i=1}^{m} \left(\sum_{j=0}^{c_i} (p_i)^j \right)$$

求 N 的正约数集合——试除法

若 $d \ge \sqrt{N}$ 是 N 的约数，则 $N/d \le \sqrt{N}$ 也是 N 的约数。换言之，约数总是成对出现的（除了对于完全平方数，$\sqrt{N}$ 会单独出现）。

因此，只需要扫描 $d = 1 \sim \sqrt{N}$，尝试 d 能否整除 N，若能整除，则 N/d 也是 N 的约数。时间复杂度为 $O(\sqrt{N})$。

```
int factor[1600], m = 0;
for (int i = 1; i*i <= n; i++) {
    if (n % i == 0) {
        factor[++m] = i;
        if (i != n/i) factor[++m] = n/i;
```

```
        }
    }
    for (int i = 1; i <= m; i++)
        cout << factor[i] <<endl;
```

试除法的推论

一个整数 N 的约数个数上界为 $2\sqrt{N}$。

求 $1\sim N$ 每个数的正约数集合——倍数法

若用“试除法”分别求出 $1\sim N$ 每个数的正约数集合，时间复杂度过高，为 $O(N\sqrt{N})$。可以反过来考虑，对于每个数 d，$1\sim N$ 中以 d 为约数的数就是 d 的倍数 $d,2d,3d,\cdots,\lfloor N/d\rfloor * d$。以下程序采用“倍数法”求出 $1\sim N$ 每个数的正约数集合。

```
vector<int> factor[500010];
for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n/i; j++)
        factor[i*j].push_back(i);
for (int i = 1; i <= n; i++) {
    for (int j = 0; j < factor[i].size(); j++)
        printf("%d ", factor[i][j]);
    puts("");
}
```

上述算法的时间复杂度为 $O(N+N/2+N/3+\cdots+N/N)=O(N\log N)$。

倍数法的推论

$1\sim N$ 每个数的约数个数的总和大约为 $N\log N$。

例题 反素数 AcWing198

对于任何正整数 x，其约数的个数记作 $g(x)$。例如：$g(1)=1$，$g(6)=4$。

如果某个正整数 x 满足：对于任意的 $0<i<x$，都有 $g(x)>g(i)$，那么称 x 为反质数。例如整数 $1,2,4,6$ 等都是反质数。

现在给定一个数 $N(1\le N\le 2*10^9)$，请求出不超过 N 的最大的反质数。

引理 1：

$1\sim N$ 中最大的反质数，就是 $1\sim N$ 中约数个数最多的数中最小的一个。

证明：

设 m 是 $1\sim N$ 中约数个数最多的数中最小的一个。根据 m 的定义，m 显然满足：

1. $\forall x<m$，$g(x)<g(m)$

2. $\forall x > m,\ g(x) \le g(m)$

根据反质数的定义，第一条性质说明 m 是反质数，第二条性质说明大于 m 的数都不是反质数，故 m 即为所求。

引理 2：

$1\sim N$ 中任何数的不同质因子都不会超过 10 个，且所有质因子的指数总和不超过 30。

证明：

因为最小的 11 个质数的乘积 $2*3*5*7*11*13*17*19*23*29*31 > 2*10^9$，所以 $N \le 2*10^9$ 不可能有多于 10 个不同的质因子。

因为即使只包含最小的质数，仍然有 $2^{31} > 2*10^9$，所以 $N \le 2*10^9$ 的质因子指数总和不可能超过 30。

引理 3：

$\forall x \in [1,N]$，x 为反质数的必要条件是：x 分解质因数后可写作 $2^{c_1}*3^{c_2}*5^{c_3}*7^{c_4}*11^{c_5}*13^{c_6}*17^{c_7}*19^{c_8}*23^{c_9}*29^{c_{10}}$，并且 $c_1 \ge c_2 \ge \cdots \ge c_{10} \ge 0$。

通俗地讲，x 的质因子是连续的若干个最小的质数，并且指数单调递减。

证明：

反证法。由引理 2，若 x 的质因数分解式中存在一项 p^k $(p > 29)$，则必定有一个不超过 29 的质因子 p' 不能整除 x。根据算术基本定理的推论，$x/p^k * p'^k$ 的约数个数与 x 的约数个数相等，但前者更小，这与反质数的定义矛盾。故 x 只包含 29 以内的质因子。

同理，若 x 的质因子不是连续若干个最小的，或者指数不单调递减，我们也可以通过上述交换质因子的方法，得到一个比 x 更小但约数个数相等的整数。因此假设不成立，原命题成立。

综上所述，我们可以使用深度优先搜索 (DFS)，尝试依次确定前 10 个质数的指数，并满足指数单调递减、总乘积不超过 N，同时记录约数的个数。

在这两个限制条件下，搜索量实际上非常小。每当搜索出一个满足条件的整数时，我们就按照引理 1 的结论更新答案，最终得到约数个数最多的数中最小的一个。

例题 余数之和 AcWing199

给定正整数 n 和 k，计算 $(k \bmod 1) + (k \bmod 2) + \cdots + (k \bmod n)$ 的值。$1 \le n, k \le 10^9$。

注意到 $k \bmod i = k - \lfloor k/i \rfloor * i$，故可转化为计算 $n*k - \sum_{i=1}^{n} \lfloor k/i \rfloor * i$。

对于任意整数 $x \in [1,k]$，设 $g(x) = \lfloor k/\lfloor k/x \rfloor \rfloor$。显然，函数 $f(x) = k/x$ 单调递减，而 $g(x) \geq \lfloor k/(k/x) \rfloor = x$，故 $\lfloor k/g(x) \rfloor \leq \lfloor k/x \rfloor$。

另外，$\lfloor k/g(x) \rfloor \geq \lfloor k/(k/\lfloor k/x \rfloor) \rfloor = \lfloor k/k * \lfloor k/x \rfloor \rfloor = \lfloor k/x \rfloor$，故 $\lfloor k/g(x) \rfloor = \lfloor k/x \rfloor$。进一步可得，$\forall i \in [x, \lfloor k/\lfloor k/x \rfloor \rfloor]$，$\lfloor k/i \rfloor$ 的值都相等。

$\forall i \in [1,k]$，$\lfloor k/i \rfloor$ 至多只有 $2\sqrt{k}$ 个不同的值。这是因为当 $i \leq \sqrt{k}$ 时，i 只有 $\sqrt{k}$ 种选择，故 $\lfloor k/i \rfloor$ 至多只有 $\sqrt{k}$ 个不同的值。而当 $i > \sqrt{k}$ 时，$\lfloor k/i \rfloor < \sqrt{k}$，故 $\lfloor k/i \rfloor$ 也至多只有 $\sqrt{k}$ 个不同的值。

综上所述，对于 $i = 1 \sim k$，$\lfloor k/i \rfloor$ 由不超过 $2\sqrt{k}$ 段组成，每一段 $i \in [x, \lfloor k/\lfloor k/x \rfloor \rfloor]$ 中 $\lfloor k/i \rfloor$ 的值都等于 $\lfloor k/x \rfloor$。在该段中，算式 $\lfloor k/i \rfloor * i = \lfloor k/x \rfloor * i$ 就是一个公差为 $\lfloor k/x \rfloor$ 的等差数列，可以直接用等差数列求和公式计算。

整个算法的时间复杂度为 $O(\sqrt{k})$。

```
long long n, k, ans;
int main() {
    cin >> n >> k;  ans = n * k;
    for (int x = 1, gx; x <= n; x = gx + 1) {
        gx = k/x ? min(k/(k/x), n) : n;
        ans -= (k/x) * (x + gx) * (gx - x + 1) / 2;
    }
    cout << ans << endl;
    return 0;
}
```

最大公约数

定义

若自然数 d 同时是自然数 a 和 b 的约数，则称 d 是 a 和 b 的公约数。在所有 a 和 b 的公约数中最大的一个，称为 a 和 b 的最大公约数，记为 $\gcd(a,b)$。

若自然数 m 同时是自然数 a 和 b 的倍数，则称 m 是 a 和 b 的公倍数。在所有 a 和 b 的公倍数中最小的一个，称为 a 和 b 的最小公倍数，记为 $\mathrm{lcm}(a,b)$。

同理，我们也可以定义三个数以及更多个数的最大公约数、最小公倍数。

定理

$$\forall a,b \in \mathbb{N}, \quad \gcd(a,b) * \mathrm{lcm}(a,b) = a * b$$

证明：

设 $d = \gcd(a,b), a_0 = a/d, b_0 = b/d$。根据最大公约数的定义，有 $\gcd(a_0,b_0) =$

1。再根据最小公倍数的定义，有 $\text{lcm}(a_0, b_0) = a_0 * b_0$。

于是 $\text{lcm}(a, b) = \text{lcm}(a_0 * d, b_0 * d) = \text{lcm}(a_0, b_0) * d = a_0 * b_0 * d = a * b/d$。

证毕。

九章算术 · 更相减损术

$\forall a, b \in \mathbb{N}, a \geq b$，有 $\gcd(a, b) = \gcd(b, a - b) = \gcd(a, a - b)$

$\forall a, b \in \mathbb{N}$，有 $\gcd(2a, 2b) = 2\gcd(a, b)$

证明：

根据最大公约数的定义，后者显然成立，我们主要证明前者。

对于 a, b 的任意公约数 d，因为 $d|a, d|b$，所以 $d|(a - b)$。因此 d 也是 $b, a - b$ 的公约数。反之亦成立。故 a, b 的公约数集合与 $b, a - b$ 的公约数集合相同。于是它们的最大公约数自然也相等。对于 $a, a - b$ 同理。

证毕。

欧几里得算法

$$\forall a, b \in \mathbb{N}, b \neq 0, \qquad \gcd(a, b) = \gcd(b, a \bmod b)$$

证明：

若 $a < b$，则 $\gcd(b, a \bmod b) = \gcd(b, a) = \gcd(a, b)$，命题成立。

若 $a \geq b$，不妨设 $a = q * b + r$，其中 $0 \leq r < b$。显然 $r = a \bmod b$。对于 a, b 的任意公约数 d，因为 $d|a$, $d|q * b$，故 $d|(a - qb)$，即 $d|r$，因此 d 也是 b, r 的公约数。反之亦成立。故 a, b 的公约数集合与 $b, a \bmod b$ 的公约数集合相同。于是它们的最大公约数自然也相等。

证毕。

```
int gcd(int a, int b) {
    return b ? gcd(b, a % b) : a;
}
```

使用欧几里得算法求最大公约数的复杂度为 $O(\log(a + b))$。欧几里得算法是最常用的求最大公约数的方法。不过，因为高精度除法（取模）不容易实现，需要做高精度运算时，可考虑用更相减损术代替欧几里得算法。

例题 **Hankson 的趣味题** AcWing200

有 n 个询问，在每个询问中，先给定四个自然数 a, b, c, d，然后求有多少个 x 满足 $\gcd(a, x) = c$ 并且 $\text{lcm}(b, x) = d$。数据范围：$n \leq 2000, 1 \leq a, b, c, d \leq 2 * 10^9$。

解法一

从 $\mathrm{lcm}(b,x)=d$ 可知 x 一定是 d 的约数。于是我们可以立即得到一个朴素算法：用试除法求出 d 的所有约数，逐一判断是否满足这两个条件。时间复杂度为 $O(n\sqrt{d}\log d)$。在实际测试中可以得到 90 分。

我们提到过，虽然 d 的约数个数上界大约是$\sqrt{d}$，但是 $1\sim d$ 中平均每个数的约数个数大约只有 $\log d$。换言之，约数个数在通常情况下远远达不到上界。根据实际测算，10^9之内的自然数中，约数个数最多的自然数仅有 1536 个约数。

为了尽量避免试除法中不能整除时耗费的时间，我们预处理出 $1\sim\sqrt{2*10^9}$ 之间的所有质数，用搜索算法组成 d 的所有约数，再判断题目中的两个条件是否满足，即可得到 100 分。当然，我们也有更加标准的解法。

解法二

因为 x 是 d 的约数，所以 x 的质因子一定也是 d 的质因子。我们可以对 d 的每个质因子 p，计算 x 可能包含多少个 p。

设 a,b,c,d,x 分别包含 m_a,m_b,m_c,m_d,m_x 个质因子 p，其中 m_x 是未知量。

根据最大公约数的定义，在 $\gcd(a,x)=c$ 中：

1. 若 $m_a>m_c$，则 m_x 只能等于 m_c；
2. 若 $m_a=m_c$，则只需满足 $m_x\geq m_c$ 即可；
3. 若 $m_a<m_c$，则 m_x 无解。

同理，根据最小公倍数的定义，在 $\mathrm{lcm}(b,x)=d$ 中：

1. 若 $m_b<m_d$，则 m_x 只能等于 m_d；
2. 若 $m_b=m_d$，则只需满足 $m_x\leq m_d$ 即可；
3. 若 $m_b>m_d$，则 m_x 无解。

结合以上两种情况，有以下结论：

1. 若 $m_a>m_c,m_b<m_d,m_c=m_d$，则 m_x 只有一种取法，即 $m_x=m_c=m_d$；
2. 若 $m_a>m_c,m_b=m_d,m_c\leq m_d$，则 m_x 只有一种取法，即 $m_x=m_c$；
3. 若 $m_a=m_c,m_b<m_d,m_c\leq m_d$，则 m_x 只有一种取法，即 $m_x=m_d$；
4. 若 $m_a=m_c,m_b=m_d,m_c\leq m_d$，则 m_x 可取 $m_c\sim m_d$ 之间的任意值，共有 m_d-m_c+1 种取法；
5. 其他情况，m_x 无解。

我们把 m_x 的取法数记为 cnt_p，也就是 x 包含质因子 p 的方案有 cnt_p 种。根据乘法原理，满足题意的 x 数量即为连乘积：

$$\prod_{\text{质数}p\mid d} cnt_p$$

为了更高效地计算，可以预处理出 $1\sim\sqrt{2*10^9}$ 之间的所有质数，对于每个询问，逐一检查每个质数是不是 d 的约数，若是，按照上述方法计算 cnt_p。若 d 不能被上述任何质数整除，则说明 d 本身是质数，应计算 cnt_d。在“反素数”一题中我们已经证明过，d 至多包含 10 个质因子，所以计算量很小，时间主要花费在找 d 的质因子上。根据 0x31 节提到的质数分布密度，整个算法的时间复杂度约为 $O(n\sqrt{d}/\log d)$。

互质与欧拉函数

定义

$\forall a,b\in\mathbb{N}$，若 $\gcd(a,b)=1$，则称 a,b 互质。

对于三个数或更多个数的情况，我们把 $\gcd(a,b,c)=1$ 的情况称为 a,b,c 互质。把 $\gcd(a,b)=\gcd(a,c)=\gcd(b,c)=1$ 称为 a,b,c 两两互质。后者显然是一个更强的条件。

欧拉函数

$1\sim N$ 中与 N 互质的数的个数被称为欧拉函数，记为 $\varphi(N)$。

若在算术基本定理中，$N=p_1{}^{c_1}p_2{}^{c_2}\cdots p_m{}^{c_m}$，则：

$$\varphi(N)=N*\frac{p_1-1}{p_1}*\frac{p_2-1}{p_2}*\cdots*\frac{p_m-1}{p_m}=N*\prod_{\text{质数}p|N}\left(1-\frac{1}{p}\right)$$

证明：

设 p 是 N 的质因子，$1\sim N$ 中 p 的倍数有 $p,2p,3p,\cdots,(N/p)*p$，共 N/p 个。同理，若 q 也是 N 的质因子，则 $1\sim N$ 中 q 的倍数有 N/q 个。如果我们把这 $N/p+N/q$ 个数去掉，那么 $p*q$ 的倍数被排除了两次，需要加回来一次。因此，$1\sim N$ 中不与 N 含有共同质因子 p 或 q 的数的个数为：

$$N-\frac{N}{p}-\frac{N}{q}+\frac{N}{pq}=N*\left(1-\frac{1}{p}-\frac{1}{q}+\frac{1}{pq}\right)=N\left(1-\frac{1}{p}\right)\left(1-\frac{1}{q}\right)$$

实际上，上述思想被称为容斥原理，我们将在 0x37 节详细介绍。类似地，可以在 N 的全部质因子上使用容斥原理，即可得到 $1\sim N$ 中不与 N 含有任何共同质因子的数的个数，也就是与 N 互质的数的个数。

证毕。

根据欧拉函数的计算式，我们只需要分解质因数，即可顺便求出欧拉函数。

```
int phi(int n) {
    int ans = n;
    for (int i = 2; i <= sqrt(n); i++)
        if (n % i == 0) {
```

```
            ans = ans / i * (i-1);
            while (n % i == 0) n /= i;
        }
    if (n > 1) ans = ans / n * (n-1);
    return ans;
}
```

性质 1~2

1. $\forall n>1$，$1\sim n$ 中与 n 互质的数的和为 $n*\varphi(n)/2$。
2. 若 a,b 互质，则 $\varphi(ab)=\varphi(a)\varphi(b)$。

证明：

因为 $\gcd(n,x)=\gcd(n,n-x)$，所以与 n 不互质的数 $x,n-x$ 成对出现，平均值为 $n/2$。因此，与 n 互质的数的平均值也是 $n/2$，进而得到性质 1。

根据欧拉函数的计算式，对a,b分解质因数，直接可得性质 2。把性质 2 推广到一般的函数上，可以得到“积性函数”的概念。

证毕。

积性函数

如果当 a,b 互质时，有 $f(ab)=f(a)*f(b)$，那么称函数 f 为积性函数。

性质 3~6

3. 若 f 是积性函数，且在算术基本定理中 $n=\prod_{i=1}^{m}p_i^{c_i}$，则 $f(n)=\prod_{i=1}^{m}f(p_i^{c_i})$。
4. 设 p 为质数，若 $p|n$ 且 $p^2|n$，则 $\varphi(n)=\varphi(n/p)*p$[①]。
5. 设 p 为质数，若 $p|n$ 但 $p^2\nmid n$，则 $\varphi(n)=\varphi(n/p)*(p-1)$。
6. $\sum_{d|n}\varphi(d)=n$

证明：

把 n 分解质因数，按照积性函数的定义，性质 3 显然成立。

若 $p|n$ 且 $p^2|n$，则 $n,n/p$ 包含相同的质因子，只是 p 的指数不同。直接把 $\varphi(n)$ 与 $\varphi(n/p)$ 按照欧拉函数的计算公式写出，二者相除，商为 p，所以性质 4 成立。

若 $p|n$ 但 $p^2\nmid n$，则 $p,n/p$ 互质，由 φ 是积性函数得 $\varphi(n)=\varphi(n/p)*\varphi(p)$，而 $\varphi(p)=p-1$，所以性质 5 成立。

设 $f(n)=\sum_{d|n}\varphi(d)$。用乘法分配律展开比较，再利用 φ 是积性函数，得到：若 n,m 互质，则 $f(nm)=\sum_{d|nm}\varphi(d)=\left(\sum_{d|n}\varphi(d)\right)*\left(\sum_{d|m}\varphi(d)\right)=f(n)*f(m)$。即 $\sum_{d|n}\varphi(d)$ 是积性函数。对于单个质因子，$f(p^m)=\sum_{d|p^m}\varphi(d)=\varphi(1)+\varphi(p)+$

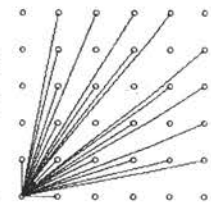
① 请注意，性质 4~6 只是欧拉函数的性质，并非所有积性函数都满足。

$\varphi(p^2)+\cdots+\varphi(p^m)$ 是一个等比数列求和再加 1，结果为 p^m。所以 $f(n)=\prod_{i=1}^{m} f(p_i^{c_i})=\prod_{i=1}^{m} p_i^{c_i}=n$，性质 6 成立。

证毕。

有关积性函数还有许多内容，并可延伸出狄利克雷卷积、莫比乌斯反演以及一系列相关的快速求和问题。这些内容比较高深，超出了我们的探讨范围，学有余力的读者可以自行查阅相关资料。

例题 **Visible Lattice Points** AcWing201



在一个平面直角坐标系的以 $(0,0)$ 为左下角、(N,N) 为右上角的矩形中，除了 $(0,0)$ 之外，每个坐标上都插着一个钉子，如右图所示。

求在原点向四周看去，能够看到多少个钉子。一个钉子能被看到，当且仅当连接它和原点的线段上没有其他钉子。右图也画出了所有能看到的钉子以及视线。$1\le N\le 1000$。

分析题目容易发现，除了 $(1,0)$，$(0,1)$ 和 $(1,1)$ 这三个钉子外，一个钉子 (x,y) 能被看到，当且仅当 $1\le x,y\le N$，$x\ne y$ 并且 $\gcd(x,y)=1$。

在 $1\le x,y\le N$，$x\ne y$ 中能看到的钉子关于过 $(0,0)$ 和 (N,N) 的直线对称。我们可以考虑其中的一半，即 $1\le x<y\le N$。换言之，对于每个 $2\le y\le N$，我们需要统计有多少个 x 满足 $1\le x<y$ 并且 $\gcd(x,y)=1$。这样的 x 的数量恰好就是 $\varphi(y)$。

综上所述，本题的答案就是 $3+2*\sum_{i=2}^{N}\varphi(i)$。我们要做的事情就是求出 $2\sim N$ 中每个数的欧拉函数。

利用 Eratosthenes 筛法，我们可以按照欧拉函数的计算式，在 $O(N\log N)$ 的时间内求出 $2\sim N$ 中每个数的欧拉函数。

```
void euler(int n) {
    for (int i = 2; i <= n; i++) phi[i] = i;
    for (int i = 2; i <= n; i++)
        if (phi[i] == i)
            for (int j = i; j <= n; j += i)
                phi[j] = phi[j] / i * (i - 1);
}
```

至此，本题已经得到解决。不过我们还可以进一步优化，利用线性筛法的思想在 $O(N)$ 的时间内快速递推出 $2\sim N$ 中每个数的欧拉函数。

由前面提到的性质 4 和性质 5：

1. 若 $p|n$ 且 $p^2|n$，则 $\varphi(n) = \varphi(n/p) * p$；

2. 若 $p|n$ 但 $p^2 \nmid n$，则 $\varphi(n) = \varphi(n/p) * (p-1)$。

在线性筛法中，每个合数 n 只会被它的最小质因子 p 筛一次。我们恰好可以在此时执行上面两条判断，从 $\varphi(n/p)$ 递推到 $\varphi(n)$。

```
int v[MAX_N], prime[MAX_N], phi[MAX_N];
void euler(int n) {
    memset(v, 0, sizeof(v)); // 最小质因子
    m = 0; // 质数数量
    for (int i = 2; i <= n; i++) {
        if (v[i] == 0) { // i是质数
            v[i] = i, prime[++m] = i;
            phi[i] = i - 1;
        }
        // 给当前的数 i 乘上一个质因子
        for (int j = 1; j <= m; j++) {
            // i有比 prime[j]更小的质因子，或者超出 n 的范围，停止循环
            if (prime[j] > v[i] || prime[j] > n / i) break;
            // prime[j]是合数 i*prime[j]的最小质因子
            v[i*prime[j]] = prime[j];
            phi[i*prime[j]] =
                phi[i] * (i%prime[j] ? prime[j]-1 : prime[j]);
        }
    }
}
```



0x33 同余

定义

若整数 a 和整数 b 除以正整数 m 的余数相等，则称 a,b 模 m **同余**，记为 $a \equiv b \pmod{m}$。

同余类与剩余系

对于 $\forall a \in [0, m-1]$，集合 $\{a+km\}(k \in \mathbb{Z})$ 的所有数模 m 同余，余数都是 a。该集合称为一个模 m 的**同余类**，简记为 $\overline{a}$。

模 m 的同余类一共有 m 个，分别为 $\overline{0}, \overline{1}, \overline{2}, \cdots, \overline{m-1}$。它们构成 m 的**完全剩余系**。

1~m 中与 m 互质的数代表的同余类共有 $\varphi(m)$ 个，它们构成 m 的**简化剩余系**。例如，模 10 的简化剩余系为 $\{\overline{1}, \overline{3}, \overline{7}, \overline{9}\}$。

简化剩余系关于模 m 乘法封闭。这是因为若 $a, b(1 \le a, b \le m)$ 与 m 互质，则 $a*b$ 也不可能与 m 含有相同的质因子，即 $a*b$ 也与 m 互质。再由余数的定义即可得到 $a*b \bmod m$ 也与 m 互质，即 $a*b \bmod m$ 也属于 m 的简化剩余系。

费马小定理

若 p 是质数，则对于任意整数 a，有 $a^p \equiv a \pmod{p}$。

欧拉定理

若正整数 a, n 互质，则 $a^{\varphi(n)} \equiv 1 \pmod{n}$，其中 $\varphi(n)$ 为欧拉函数。

证明：

设 n 的简化剩余系为 $\{\overline{a_1}, \overline{a_2}, \cdots, \overline{a_{\varphi(n)}}\}$。对 $\forall a_i, a_j$，若 $a*a_i \equiv a*a_j \pmod{n}$，则 $a*(a_i - a_j) \equiv 0$。因为 a, n 互质，所以 $a_i - a_j \equiv 0$，即 $a_i \equiv a_j$。故当 $a_i \ne a_j$ 时，aa_i, aa_j 也代表不同的同余类。

又因为简化剩余系关于模 n 乘法封闭，故 $\overline{aa_i}$ 也在简化剩余系集合中。因此，集合 $\{\overline{a_1}, \overline{a_2}, \cdots, \overline{a_{\varphi(n)}}\}$ 与集合 $\{\overline{aa_1}, \overline{aa_2}, \cdots, \overline{aa_{\varphi(n)}}\}$ 都能表示 n 的简化剩余系。综上所述：

$$a^{\varphi(n)} a_1 a_2 \cdots a_{\varphi(n)} \equiv (aa_1)(aa_2)\cdots(aa_{\varphi(n)}) \equiv a_1 a_2 \cdots a_{\varphi(n)} \pmod{n}$$

因此 $a^{\varphi(n)} \equiv 1 \pmod n$。

当 p 是质数时，$\varphi(p) = p-1$，并且只有 p 的倍数与 p 不互质。所以，只要 a 不是 p 的倍数，就有 $a^{p-1} \equiv 1 \pmod p$，两边同乘 a 就是费马小定理。另外，若 a 是 p 的倍数，费马小定理显然成立。

证毕。

欧拉定理的推论

若正整数 a,n 互质，则对于任意正整数 b，有 $a^b \equiv a^{b \bmod \varphi(n)} \pmod n$。

证明：

设 $b = q * \varphi(n) + r$，其中 $0 \le r < \varphi(n)$，即 $r = b \bmod \varphi(n)$。于是：

$$a^b \equiv a^{q*\varphi(n)+r} \equiv \left(a^{\varphi(n)}\right)^q * a^r \equiv 1^q * a^r \equiv a^r \equiv a^{b \bmod \varphi(n)} \pmod n$$

证毕。

许多计数类的题目要求我们把答案对一个质数 p 取模后输出。面对 $a+b, a-b, a*b$ 这样的算式，可以在计算前先把 a,b 对 p 取模。面对乘方算式，根据欧拉定理的推论，可以先把底数对 p 取模、指数对 $\varphi(p)$ 取模，再计算乘方。

特别地，当 a,n 不一定互质且 $b > \varphi(n)$ 时，有 $a^b \equiv a^{b \bmod \varphi(n)+\varphi(n)} \pmod n$。这意味着即使底数和模数不互质，我们也有办法把指数的规模缩小到容易计算的范围内。上式可以通过寻找 $a^b \bmod n$ 的**指数循环节**证明，留给学有余力的读者思考。

例题 **The Luckiest Number** AcWing202

给定一个正整数 L，$L \le 2 * 10^9$。

问：至少多少个 8 连在一起组成的正整数是 L 的倍数？

x 个 8 连在一起组成的正整数可写作 $8(10^x-1)/9$。题目就是让我们求出一个最小的 x，满足 $L \mid 8(10^x-1)/9$。设 $d = \gcd(L,8)$。

$$L \mid \frac{8(10^x-1)}{9} \Leftrightarrow 9L \mid 8(10^x-1) \Leftrightarrow \frac{9L}{d} \mid 10^x-1 \Leftrightarrow 10^x \equiv 1 \left(\bmod \frac{9L}{d}\right)$$

引理：

若正整数 a,n 互质，则满足 $a^x \equiv 1 \pmod n$ 的最小正整数 x_0 是 $\varphi(n)$ 的约数。

证明：

反证法。假设满足 $a^x \equiv 1 \pmod n$ 的最小正整数 x_0 不能整除 $\varphi(n)$。

设 $\varphi(n) = qx_0 + r\ (0 < r < x_0)$。因为 $a^{x_0} \equiv 1 \pmod n$，所以 $a^{qx_0} \equiv 1 \pmod n$。根据欧拉定理，有 $a^{\varphi(n)} \equiv 1 \pmod n$，所以 $a^r \equiv 1 \pmod n$。这与 x_0 最小矛盾。故

假设不成立，原命题成立。

证毕。

根据以上引理，我们只需求出欧拉函数 $\varphi(9L/d)$，枚举它的所有约数，用快速幂逐一检查是否满足条件即可。时间复杂度为 $O(\sqrt{L}\log L)$。

扩展欧几里得算法

Bézout 定理

对于任意整数 a,b，存在一对整数 x,y，满足 $ax+by=\gcd(a,b)$。

证明：

在欧几里得算法的最后一步，即 $b=0$ 时，显然有一对整数 $x=1,y=0$，使得 $a*1+0*0=\gcd(a,0)$。

若 $b>0$，则 $\gcd(a,b)=\gcd(b,a \bmod b)$。假设存在一对整数 x,y，满足 $b*x+(a \bmod b)*y=\gcd(b,a \bmod b)$，因为 $bx+(a \bmod b)y=bx+(a-b\lfloor a/b\rfloor)y=ay+b(x-\lfloor a/b\rfloor y)$，所以令 $x'=y$，$y'=x-\lfloor a/b\rfloor y$，就得到了 $ax'+by'=\gcd(a,b)$。

对欧几里得算法的递归过程应用数学归纳法，可知 Bézout 定理成立。

证毕。

Bézout 定理是按欧几里得算法的思路证明的，且上述证明同时给出了整数 x 和 y 的计算方法。这种计算方法被称为**扩展欧几里得算法**。

```
int exgcd(int a, int b, int &x, int &y) {
    if (b == 0) { x = 1, y = 0; return a; }
    int d = exgcd(b, a%b, x, y);
    int z = x; x = y; y = z - y * (a / b);
    return d;
}
```

定义变量 d,x_0,y_0，调用 `d=exgcd(a,b,x0,y0)`。注意在上述代码中，x_0,y_0 是以引用的方式传递的。上述程序求出方程 $ax+by=\gcd(a,b)$ 的一组特解 x_0,y_0，并返回 a,b 的最大公约数 d。

对于更为一般的方程 $ax+by=c$，它有解当且仅当 $d|c$。我们可以先求出 $ax+by=d$ 的一组特解 x_0,y_0，然后令 x_0,y_0 同时乘上 c/d，就得到了 $ax+by=c$ 的一组特解 $(c/d)x_0,(c/d)y_0$。

事实上，方程 $ax+by=c$ 的通解可以表示为：

$$x=\frac{c}{d}x_0+k\frac{b}{d},\qquad y=\frac{c}{d}y_0-k\frac{a}{d}\ (k\in\mathbb{Z})$$

其中 k 取遍整数集合，$d=\gcd(a,b)$，x_0,y_0 是 $ax+by=\gcd(a,b)$ 的一组特解。

乘法逆元

若整数 b,m 互质，并且 $b|a$，则存在一个整数 x，使得 $a/b\equiv a*x\pmod m$。称 x 为 b 的模 m 乘法逆元，记为 $b^{-1}\pmod m$。

因为 $a/b\equiv a*b^{-1}\equiv a/b*b*b^{-1}\pmod m$，所以 $b*b^{-1}\equiv 1\pmod m$。

如果 m 是质数（此时我们用符号 p 代替 m）并且 $b<p$，根据费马小定理，$b^{p-1}\equiv 1\pmod p$，即 $b*b^{p-2}\equiv 1\pmod p$。因此，**当模数 p 为质数时，b^{p-2} 即为 b 的乘法逆元。**

如果只是保证 b,m 互质，那么乘法逆元可通过**求解同余方程** $b*x\equiv 1\pmod m$ 得到。下个部分我们就来介绍线性同余方程及其求解方法。

有了乘法逆元，我们在计数类问题中即使遇到 a/b 这样的除法算式，也可以先把 a,b 各自对模数 p 取模，再计算 $a*b^{-1}\bmod p$ 作为最终的结果。当然，前提是必须保证 b,p 互质（当 p 是质数时，等价于 b 不是 p 的倍数）。

到此为止，我们在模 p 运算下对加、减、乘、除、乘方运算都已经有了适当的处理方法，请读者回顾并加以思考。

例题 **Sumdiv** AcWing97

求 A^B 的所有约数之和 $\bmod 9901$ $(1\le A,B\le 5*10^7)$。

在 0x03 节中，我们已经用递归和分治算法解决了取模乘法下等比数列的求和问题。在本题中我们将直接借助乘法逆元计算等比数列的求和公式。

把 A 分解质因数，表示为 $p_1^{c_1}*p_2^{c_2}*\cdots*p_n^{c_n}$，由算术基本定理的“约数和”推论，$A^B$ 的所有约数之和为：$(1+p_1+p_1^2+\cdots+p_1^{B*c_1})*(1+p_2+p_2^2+\cdots+p_2^{B*c_2})*\cdots*(1+p_n+p_n^2+\cdots+p_n^{B*c_n})$。

上式的每一项都是一个等比数列。以第一项为例，$1+p_1+p_1^2+\cdots+p_1^{B*c_1}=(p_1^{B*c_1+1}-1)/(p_1-1)$。可以先用快速幂计算分子 $(p_1^{B*c_1+1}-1)\bmod 9901$ 和分母 $(p_1-1)\bmod 9901$。因为 9901 是质数，只要 p_1-1 不是 9901 的倍数，就只需计算 p_1-1 的乘法逆元 inv，用乘 inv 代替除以 (p_1-1)，直接计算出等比数列求和公式的结果。

特别地，若 p_1-1 是 9901 的倍数，此时乘法逆元不存在，但是 $p_1\bmod 9901=1$，所以 $1+p_1+p_1^2+\cdots+p_1^{B*c_1}\equiv 1+1+1^2+\cdots+1^{B*c_1}\equiv B*c_1+1\pmod{9901}$。

```
int a, b, m, ans = 1, mod = 9901;
int p[20], c[20];
void divide(int n) {
```

```
    m = 0;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            p[++m] = i, c[m] = 0;
            while (n % i == 0) n /= i, c[m]++;
        }
    }
    if (n > 1)
        p[++m] = n, c[m] = 1;
}
int power(int a, long long b) {
    int c = 1;
    for (; b; b >>= 1) {
        if (b & 1) c = (long long)c * a % mod;
        a = (long long) a * a % mod;
    }
    return c;
}
int main() {
    cin >> a >> b;
    divide(a); // 分解质因数
    for (int i = 1; i <= m; i++) {
        if ((p[i] - 1) % mod == 0) { // 没有逆元时，特判
            ans = ((long long)b * c[i] + 1) % mod * ans % mod;
            continue;
        }
        int x = power(p[i], (long long)b * c[i] + 1); // 分子
        x = (x - 1 + mod) % mod;
        int y = p[i] - 1; // 分母
        y = power(y, mod - 2); // 根据费马小定理求乘法逆元
        ans = (long long)ans * x % mod * y % mod;
    }
    cout << ans <<endl;
}
```

线性同余方程

给定整数 a,b,m，求一个整数 x 满足 $a*x\equiv b\,(\mathrm{mod}\,m)$，或者给出无解。因为

未知数的指数为 1，所以我们称之为一次同余方程，也称线性同余方程。

$a*x \equiv b \pmod m$ 等价于 $a*x-b$ 是 m 的倍数，不妨设为 $-y$ 倍。于是，该方程可以改写为 $a*x+m*y=b$。

根据 Bézout 定理及其证明过程，线性同余方程有解当且仅当 $\gcd(a,m) \mid b$。

在有解时，先用欧几里得算法求出一组整数 x_0, y_0，满足 $a*x_0+m*y_0=\gcd(a,m)$。然后 $x=x_0*b/\gcd(a,m)$ 就是原线性同余方程的一个解。

方程的通解则是所有模 $m/\gcd(a,m)$ 与 x 同余的整数。

例题 同余方程 AcWing203

求关于 x 的同余方程 $a*x \equiv 1 \pmod b$ 的最小正整数解。输入数据保证一定有解。

由上述线性同余方程的知识可得，$a*x \equiv 1 \pmod b$ 有解当且仅当 $\gcd(a,b)=1$。方程可改写为 $a*x+b*y=1$，用欧几里得算法求出一组特解 x_0, y_0，则 x_0 就是原方程的一个解，通解为所有模 b 与 x_0 同余的整数。通过取模操作把解的范围移动到 $1\sim b$ 之间，就得到了最小正整数解。

```
long long a, b, x, y;
long long exgcd(long long a, long long b,
          long long &x, long long &y) {
    if (!b) { x=1, y=0; return a; }
    long long d = exgcd(b, a%b, x, y);
    long long z = x; x = y; y = z - y*(a/b);
    return d;
}
int main() {
    cin >> a >> b;
    exgcd(a, b, x, y);
    cout << (x % b + b) % b << endl;
}
```

中国剩余定理

设 $m_1, m_2, \cdots, m_n$ 是两两互质的整数，$m=\prod_{i=1}^{n} m_i$，$M_i=m/m_i$，t_i 是线性同余方程 $M_i t_i \equiv 1 \pmod{m_i}$ 的一个解。对于任意的 n 个整数 $a_1, a_2, \cdots, a_n$，方程组

$$\begin{cases} x \equiv a_1 \pmod{m_1} \\ x \equiv a_2 \pmod{m_2} \\ \quad\vdots \\ x \equiv a_n \pmod{m_n} \end{cases}$$

有整数解，解为 $x=\sum_{i=1}^{n}a_iM_it_i$。

证明：

因为 $M_i=m/m_i$ 是除 m_i 之外所有模数的倍数，所以 $\forall k\neq i,\ a_iM_it_i\equiv 0\ (\mathrm{mod}\ m_k)$。又因为 $a_iM_it_i\equiv a_i\ (\mathrm{mod}\ m_i)$，所以代入 $x=\sum_{i=1}^{n}a_iM_it_i$，原方程组成立。

证毕。

中国剩余定理给出了模数两两互质的线性同余方程组的一个特解。方程组的通解可以表示为 $x+km\ (k\in\mathbb{Z})$。有些题目要求我们求出最小的非负整数解，只需把 x 对 m 取模，并让 x 落在 $0\sim m-1$ 的范围内即可。

另外，即使模数不满足两两互质，我们也有方法判断线性同余方程组是否有解，并求出方程组的解。

例题 **Strange Way to Express Integers** AcWing204

给定 $2n$ 个正整数 $a_1,a_2,\cdots,a_n$ 和 $m_1,m_2,\cdots,m_n$，求一个最小的正整数 x，满足 $\forall i\in[1,n],\ x\equiv a_i\ (\mathrm{mod}\ m_i)$，或者给出无解。

本题中 m_i 不一定两两互质，中国剩余定理不再适用。可以考虑使用数学归纳法，假设已经求出了前 $k-1$ 个方程构成的方程组的一个解 x。记 $m=\mathrm{lcm}(m_1,m_2,\cdots,m_{k-1})$，则 $x+i*m\ (i\in\mathbb{Z})$ 是前 $k-1$ 个方程的通解。

考虑第 k 个方程，求出一个整数 t，使得 $x+t*m\equiv a_k\ (\mathrm{mod}\ m_k)$。该方程等价于 $m*t\equiv a_k-x\ (\mathrm{mod}\ m_k)$，其中 t 是未知量。这就是一个线性同余方程，可以用扩展欧几里得算法判断是否有解，并求出它的解。若有解，则 $x'=x+t*m$ 就是前 k 个方程构成的方程组的一个解。

综上所述，我们使用 n 次扩展欧几里得算法，就求出了整个方程组的解。

高次同余方程

关于高次同余方程，有 $a^x\equiv b(\mathrm{mod}\ p)$ 和 $x^a\equiv b(\mathrm{mod}\ p)$ 两类问题。不过后者超出了我们的探讨范围，学有余力的读者可以查阅“原根”“阶”“指标”的相关资料。我们重点来解决前者。

问题：

给定整数 a,b,p，其中 a,p 互质，求一个非负整数 x，使得 $a^x\equiv b(\mathrm{mod}\ p)$。

Baby Step, Giant Step 算法

因为 a,p 互质，所以可以在模 p 意义下执行关于 a 的乘、除法运算。

设 $x=i*t-j$，其中 $t=\lceil\sqrt{p}\rceil$，$0\leq j\leq t-1$，则方程变为 $a^{i*t-j}\equiv b(\mathrm{mod}\ p)$。

即 $(a^t)^i \equiv b * a^j \pmod p$。

对于所有的 $j \in [0, t-1]$，把 $b * a^j \bmod p$ 插入一个 Hash 表。

枚举 i 的所有可能取值，即 $i \in [0, t]$，计算出 $(a^t)^i \bmod p$，在 Hash 表中查找是否存在对应的 j，更新答案即可。时间复杂度为 $O(\sqrt{p})$。

下面的参考程序实现了 Baby Step, Giant Step 算法，计算同余方程 $a^x \equiv b \pmod p$ 的最小非负整数解，无解时返回 -1。

```
int baby_step_giant_step(int a, int b, int p) {
    map<int, int> hash; hash.clear();
    b %= p;
    int t = (int)sqrt(p) + 1;
    for (int j = 0; j < t; j++) {
        int val = (long long)b * power(a, j, p) % p; // b*a^j
        hash[val] = j;
    }
    a = power(a, t, p); // a^t
    if (a == 0) return b == 0 ? 1 : -1;
    for (int i = 0; i <= t; i++) {
        int val = power(a, i, p); // (a^t)^i
        int j = hash.find(val) == hash.end() ? -1 : hash[val];
        if (j >= 0 && i * t - j >= 0) return i * t - j;
    }
    return -1;
}
```



0x34 矩阵乘法

一个 $n*m$ 的矩阵可看作一个 $n*m$ 的二维数组。矩阵的加法和减法就是把两个矩阵对应位置上的数相加减，即 $C=A\pm B \Leftrightarrow \forall i\in[1,n],\ \forall j\in[1,m],\ C_{i,j}=A_{i,j}\pm B_{i,j}$，其中 A,B,C 都是 $n*m$ 矩阵。

矩阵乘法的定义略微复杂一些。设 A 是 $n*m$ 矩阵，B 是 $m*p$ 矩阵，则 $C=A*B$ 是 $n*p$ 矩阵，并且 $\forall i\in[1,n],\ \forall j\in[1,p]$：

$$C_{i,j}=\sum_{k=1}^{m}A_{i,k}*B_{k,j}$$

也就是说，参与矩阵乘法运算的第一个矩阵的列数必须等于第二个矩阵的行数，所得到的结果矩阵的行数、列数分别等于第一个矩阵的行数、第二个矩阵的列数。结果矩阵 C 第 i 行第 j 列的数，是由 A 的第 i 行的 m 个数与 B 的第 j 列的 m 个数分别相乘再相加得到的。

矩阵乘法满足结合律，即 $(A*B)*C=A*(B*C)$，满足分配律，即 $(A+B)*C=A*C+B*C$，不满足交换律，即 $A*B$ 不一定等于 $B*A$。

考虑矩阵乘法中一种特殊的情形，F 是 $1*n$ 矩阵，A 是 $n*n$ 矩阵，则 $F'=F*A$ 也是 $1*n$ 矩阵。F 和 F' 可看作一维数组，省略它们的行下标 1。按照矩阵乘法的定义，$\forall j\in[1,n]$，有 $F'_j=\sum_{k=1}^{n}F_k*A_{k,j}$。它的含义是，经过一次与 A 的矩阵乘法运算后，F 数组中的第 k 个值会以 $A_{k,j}$ 为系数累加到 F' 数组的第 j 个值上，等价于在一个向量的 k,j 两个状态之间发生了递推。

例题 **Fibonacci** AcWing205

在斐波那契数列中，$Fib_0=0,\ Fib_1=1,\ Fib_n=Fib_{n-1}+Fib_{n-2}(n>1)$。现给定整数 $n,m\ (0\le n\le 2*10^9,\ m=10000)$，求 $Fib_n \bmod m$。

直接循环递推计算斐波那契数列，时间复杂度显然为 $O(n)$。不过，Fib_n 只与 Fib_{n-1} 和 Fib_{n-2} 有关，我们在递推时只需要保存最近的两个斐波那契数，即可得到下一个斐波那契数。

设 $F(n)$ 表示一个 $1*2$ 的矩阵，$F(n)=[Fib_n \quad Fib_{n+1}]$。

我们希望根据 $F(n-1)=[Fib_{n-1}\quad Fib_n]$ 计算出 $F(n)$。也就是说，要把 $F(n-1)$ 第 2 列上的数作为 $F(n)$ 第 1 列上的数，并把 $F(n-1)$ 第 1、2 列上的数都累加到 $F(n)$ 的第 2 列上。因此，我们令矩阵 A 第 2 行第 1 列、第 1 行第 2 列、第 2 行第 2 列的数都是 1，第 1 行第 1 列的数是 0。

$$F(n)=F(n-1)*A \Leftrightarrow [Fib_n\quad Fib_{n+1}]=[Fib_{n-1}\quad Fib_n]*\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix}$$

请读者根据矩阵乘法的定义对上式进行验证。

上式是一个矩阵乘法递推式，初值为 $F(0)=[0\quad 1]$，目标为 $F(n)=F(0)*A^n$。因为矩阵乘法满足结合律，所以我们可以用快速幂计算 $F(0)*A^n$，得到的矩阵第 1 列上的数就是 Fib_n。算法的时间复杂度为 $O(2^3\log n)$，其中 2^3 是矩阵乘法所消耗的时间。

题目还要求我们对 m 取模，可以直接在执行矩阵乘法时，对各个整数的加法、乘法运算加入取模操作。详细过程请参阅下面的参考程序：

```
const int mod = 10000;
int n;
void mul(int f[2], int a[2][2]) {
    int c[2];
    memset(c, 0, sizeof(c));
    for (int j = 0; j < 2; j++)
        for (int k = 0; k < 2; k++)
            c[j] = (c[j] + (long long)f[k] * a[k][j]) % mod;
    memcpy(f, c, sizeof(c));
}
void mulself(int a[2][2]) {
    int c[2][2];
    memset(c, 0, sizeof(c));
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 2; j++)
            for (int k = 0; k < 2; k++)
                c[i][j] = (c[i][j]+(long long)a[i][k]*a[k][j])%mod;
    memcpy(a, c, sizeof(c));
}
int main() {
    while (cin >> n && n != -1) {
        int f[2] = {0,1};
        int a[2][2] = {{0,1}, {1,1}};
        for (; n; n >>= 1) {
```

```
            if (n & 1) mul(f, a);
            mulself(a);
        }
        cout << f[0] << endl;
    }
}
```

这道题使我们对“矩阵乘法加速递推”有了初步的认识。一般来说，如果一类问题具有以下特点：

1. 可以抽象出一个长度为 n 的一维向量，该向量在每个单位时间发生一次变化；
2. 变化的形式是一个线性递推（只有若干“加法”或“乘一个系数”的运算）；
3. 该递推式在每个时间可能作用于不同的数据上，但本身保持不变；
4. 向量变化时间（即递推的轮数）很长，但向量长度 n 不大。

那么可以考虑采用矩阵乘法进行优化。我们把这个长度为 n 的一维向量称为“**状态矩阵**”，把用于与“状态矩阵”相乘的固定不变的矩阵 A 称为“**转移矩阵**”。若状态矩阵中的第 x 个数对下一单位时间状态矩阵中的第 y 个数产生影响，则把**转移矩阵的第 x 行第 y 列**赋值为适当的系数。

矩阵乘法加速递推的关键在于定义出“状态矩阵”，并根据递推式构造出正确的“转移矩阵”，之后就可以利用快速幂和矩阵乘法完成程序实现了。时间复杂度为 $O(n^3 \log T)$，其中 T 为递推总轮数。

例题 石头游戏 AcWing206

石头游戏在一个 n 行 m 列 $(1 \le n, m \le 8)$ 的网格上进行，每个格子对应一种操作序列，操作序列至多有 10 种，分别用 0~9 这 10 个数字指明。

操作序列是一个长度不超过 6 且循环执行、每秒执行一个字符的字符串。每秒钟，所有格子同时执行各自操作序列里的下一个字符。序列中的每个字符是以下格式之一。

1. 数字 0~9：表示拿 0~9 个石头到该格子。
2. NWSE：表示把这个格子内所有的石头推到相邻的格子，N 表示上方，W 表示左方，S 表示下方，E 表示右方。
3. D：表示拿走这个格子的所有石头。

给定每种操作序列对应的字符串，以及网格中每个格子对应的操作序列，求石头游戏进行了 t 秒之后，石头最多的格子里有多少个石头。在游戏开始时，网格是空的。

设 $\forall i \in [1,n], j \in [1,m]$，$\text{num}(i,j) = (i-1) * m + j$。

把网格看作长度为 $n * m$ 的一维向量，定义 1 行 $n * m + 1$ 列的“状态矩阵” F，下标为 $0 \sim n * m$，其中 $F[\text{num}(i,j)]$ 记录格子 (i,j) 中石头的个数。

特别地，令 $F[0]$ 始终等于 1。

F 会随着时间的增长而不断变化。设 F_k 表示 k 秒之后的“状态矩阵”。在游戏开始时，根据题意和 F 的定义，有 $F_0 = [1 \quad 0 \quad 0 \quad \cdots \quad 0]$。

注意到操作序列的长度不超过 6，而 1~6 的最小公倍数是 60，所以每经过 60 秒，所有操作序列都会重新处于最开始的字符处。我们可以统计出第 k 秒 $(1 \le k \le 60)$ 各个格子执行了什么字符，第 $k+60$ 秒执行的字符与第 k 秒一定是相同的。

对于 1~60 之间的每个 k，各个格子在第 k 秒执行的操作字符可以构成一个“转移矩阵” A_k，矩阵行、列下标都是 $0 \sim n*m$。构造方法如下。

1. 若网格 (i,j) 第 k 秒的操作字符为“N”，且 $i>1$，则令 $A_k[\text{num}(i,j), \text{num}(i-1,j)] = 1$，表示把石头推到上边的格子。字符“W”“S”“E”类似。

2. 若网格 (i,j) 第 k 秒的操作字符是一个数字 x，则令 $A_k[0, \text{num}(i,j)] = x$，$A_k[\text{num}(i,j), \text{num}(i,j)] = 1$，表示格子里本来就有的石头不动，再拿 x 块石头。

3. 令 $A_k[0,0] = 1$。

4. A_k 的其他部分均赋值为 0。

上述构造实际上把“状态矩阵”的第 0 列作为“石头来源”。$A_k[0,0] = 1$ 保证了 $F[0]$ 始终等于 1。$A_k[0, \text{num}(i,j)] = x$ 相当于从“石头来源”获取 x 个石头，放到格子 (i,j) 中。

使用矩阵乘法加速递推，遇到常数项时，经常需要在“状态矩阵”中添加一个额外的位置，始终存储常数 1，并乘上“转移矩阵”中适当的系数，累加到“状态矩阵”的其他位置。

设 $A = \prod_{i=1}^{60} A_i$，$t = q*60 + r \ (0 \le r < 60)$。根据上面的讨论：$F_t = F_0 * A^q * \prod_{i=1}^{r} A_i$。使用矩阵乘法和快速幂计算该式，$F_t$ 第 $1 \sim n*m$ 列中的最大值即为所求。



0x35 高斯消元与线性空间

高斯消元

高斯消元是一种求解线性方程组的方法。所谓线性方程组，是由 M 个 N 元一次方程共同构成的。线性方程组的所有系数可以写成一个 M 行 N 列的“系数矩阵”，再加上每个方程等号右侧的常数，可以写成一个 M 行 $N+1$ 列的“增广矩阵”。例如：

$$\begin{cases} x_1 + 2x_2 - x_3 = -6 \\ 2x_1 + x_2 - 3x_3 = -9 \\ -x_1 - x_2 + 2x_3 = 7 \end{cases} \Rightarrow \left[\begin{array}{ccc|c} 1 & 2 & -1 & -6 \\ 2 & 1 & -3 & -9 \\ -1 & -1 & 2 & 7 \end{array}\right]$$

求解这种方程组的步骤可以概括成对“增广矩阵”的三类操作。

1. 用一个非零的数乘某一行。
2. 把其中一行的若干倍加到另一行上。
3. 交换两行的位置。

我们把这三类操作称为矩阵的“初等行变换”。同理，我们也可以定义矩阵的“初等列变换”。用若干次初等行变换求解上面的方程组，过程如下：

$$\begin{bmatrix} 1 & 2 & -1 & -6 \\ 2 & 1 & -3 & -9 \\ -1 & -1 & 2 & 7 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 2 & -1 & -6 \\ 0 & -3 & -1 & 3 \\ -1 & -1 & 2 & 7 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 2 & -1 & -6 \\ 0 & -3 & -1 & 3 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} 1 & 2 & -1 & -6 \\ 0 & 1 & 1 & 1 \\ 0 & -3 & -1 & 3 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 2 & -1 & -6 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 2 & 6 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 2 & -1 & -6 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 3 \end{bmatrix}$$

最后得到的矩阵被称为“阶梯形矩阵”，它的系数矩阵部分被称为“上三角矩阵”，名字来源于其形状。这个矩阵表达的信息是：

$$\left[\begin{array}{ccc|c} 1 & 2 & -1 & -6 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 3 \end{array}\right] \Rightarrow \begin{cases} x_1 + 2x_2 - x_3 = -6 \\ x_2 + x_3 = 1 \\ x_3 = 3 \end{cases}$$

因此，我们已经知道了最后一个未知量的值，从下往上依次代回方程组，即可得到每个未知量的解。事实上，该矩阵也可以进一步化简：

$$\begin{bmatrix}1 & 2 & -1 & -6\\0 & 1 & 1 & 1\\0 & 0 & 1 & 3\end{bmatrix}\Rightarrow\begin{bmatrix}1 & 2 & 0 & -3\\0 & 1 & 0 & -2\\0 & 0 & 1 & 3\end{bmatrix}\Rightarrow\left[\begin{array}{ccc|c}1 & 0 & 0 & 1\\0 & 1 & 0 & -2\\0 & 0 & 1 & 3\end{array}\right]$$

最后得到的矩阵被称为“简化阶梯形矩阵”，它的系数矩阵部分是一个“对角矩阵”，名字来源于其形状。该矩阵已经直接给出了方程组的解。

通过初等行变换把增广矩阵变为简化阶梯形矩阵的线性方程组求解算法就是高斯消元算法。高斯消元算法的思想就是，对每个未知量 x_i，找到一个 x_i 的系数非 0，但 $x_1 \sim x_{i-1}$ 的系数都是 0 的方程，然后用初等行变换把其他方程的 x_i 的系数全部消成 0。

上面给出的例子是一种比较理想的情况。事实上，在高斯消元的过程中，可能会遇到各种各样的特殊情形，需要我们分情况讨论。

首先，在高斯消元过程中，可能出现 $0=d$ 这样的方程，其中 d 是一个非零常数。这表明某些方程之间存在矛盾，方程组无解。

其次，有可能找不到一个 x_i 的系数非零，但 $x_1 \sim x_{i-1}$ 的系数都是零的方程。这是我们要重点讨论的情况，例如：

$$\begin{cases}x_1+2x_2-x_3=3\\2x_1+4x_2-8x_3=0\\-x_1-2x_2+6x_3=2\end{cases}\Rightarrow\begin{bmatrix}1 & 2 & -1 & 3\\0 & 0 & -6 & -6\\0 & 0 & 5 & 5\end{bmatrix}\Rightarrow\begin{bmatrix}1 & 2 & -1 & 3\\0 & 0 & 1 & 1\\0 & 0 & 0 & 0\end{bmatrix}\Rightarrow\left[\begin{array}{ccc|c}1 & 2 & 0 & 4\\0 & 0 & 1 & 1\\0 & 0 & 0 & 0\end{array}\right]$$

在上例中，找不到 x_2 的系数非 0 但 x_1 的系数为 0 的方程。方程组的解可以写作：

$$\begin{cases}x_1=4-2x_2\\x_3=1\end{cases}$$

x_2 取任何值，都可以计算出一个对应的 x_1，并且满足原方程组。换言之，方程组有无穷多个解。我们把 x_1, x_3 这样的未知量称为“**主元**”，把 x_2 这样的未知量称为“**自由元**”。

仔细分析可以发现，对于每个主元，整个简化阶梯形矩阵中有且仅有一个位置 (i,j)，满足该主元的系数非零。第 j 列的其他位置都是 0，第 i 行的第 $1 \sim j-1$ 列都是 0。

综上所述，在高斯消元完成后，若存在系数全为 0、常数不为 0 的行，则方程组无解。若系数不全为 0 的行恰好有 n 个，则说明主元有 n 个，方程组有唯一解。若系数不全为 0 的行有 $k<n$ 个，则说明主元有 k 个，自由元有 $n-k$ 个，方程组有无穷多个解。

例题 球形空间产生器 AcWing207

有一个球形空间产生器能够在 n 维空间 $(1 \le n \le 10)$ 中产生一个坚硬的球体。

现在，你被困在了这个 n 维球体中，你只知道球面上 $n+1$ 个点的坐标，你需要以最快的速度确定这个 n 维球体的球心坐标，以便于摧毁这个球形空间产生器。数据保证有解，答案精确到小数点后三位。

一个球体上的所有点到球心的距离相等，因此只需求出一个点 $(x_1, x_2, \cdots, x_n)$，使得：

$$\sum_{j=1}^{n}(a_{i,j}-x_j)^2 = C$$

其中 C 为常数，$i \in [1, n+1]$，球面上第 i 个点的坐标为 $(a_{i,1}, a_{i,2}, \cdots, a_{i,n})$。该方程组由 $n+1$ 个 n 元二次方程构成，不是线性方程组。但是我们可以通过相邻两个方程作差，把它变成 n 个 n 元一次方程，同时消去常数 C。

$$\sum_{j=1}^{n}\left(a_{i,j}^2 - a_{i+1,j}^2 - 2x_j(a_{i,j}-a_{i+1,j})\right) = 0 \quad (i = 1,2,\cdots,n)$$

把变量放在左边，常数放在右边。

$$\sum_{j=1}^{n}2(a_{i,j}-a_{i+1,j})x_j = \sum_{j=1}^{n}(a_{i,j}^2 - a_{i+1,j}^2) \quad (i = 1,2,\cdots,n)$$

这就变成了一个线性方程组。题目保证方程组有唯一解，我们直接对下面的增广矩阵进行高斯消元，变为简化阶梯形矩阵，即可得到每个 x_j 的值。

$$\begin{bmatrix} 2(a_{1,1}-a_{2,1}) & 2(a_{1,2}-a_{2,2}) & \cdots & 2(a_{1,n}-a_{2,n}) & \sum_{j=1}^{n}(a_{1,j}^2-a_{2,j}^2) \\ 2(a_{2,1}-a_{3,1}) & 2(a_{2,2}-a_{3,2}) & \cdots & 2(a_{2,n}-a_{3,n}) & \sum_{j=1}^{n}(a_{2,j}^2-a_{3,j}^2) \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 2(a_{n,1}-a_{n+1,1}) & 2(a_{n,2}-a_{n+1,2}) & \cdots & 2(a_{n,n}-a_{n+1,n}) & \sum_{j=1}^{n}(a_{n,j}^2-a_{n+1,j}^2) \end{bmatrix}$$

```
double a[20][20], b[20], c[20][20];
int n;
int main() {
    cin >> n;
    for (int i = 1; i <= n + 1; i++)
        for (int j = 1; j <= n; j++) scanf("%lf", &a[i][j]);
    // c: 系数矩阵，b: 常数，二者一起构成增广矩阵
```

```
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++) {
                c[i][j] = 2 * (a[i][j] - a[i + 1][j]);
                b[i] += a[i][j] * a[i][j] - a[i + 1][j] * a[i + 1][j];
            }
        // 高斯消元（数据保证有唯一解）
        for (int i = 1; i <= n; i++) {
            // 找到x[i]的系数不为0的一个方程
            for (int j = i; j <= n; j++) {
                if (fabs(c[j][i]) > 1e-8) {
                    for (int k = 1; k <= n; k++) swap(c[i][k], c[j][k]);
                    swap(b[i], b[j]);
                }
            }
            // 消去其他方程的x[i]的系数
            for (int j = 1; j <= n; j++) {
                if (i == j) continue;
                double rate = c[j][i] / c[i][i];
                for (int k = i; k <= n; k++) c[j][k] -= c[i][k] * rate;
                b[j] -= b[i] * rate;
            }
        }
        for (int i = 1; i < n; i++) printf("%.3f ", b[i] / c[i][i]);
        printf("%.3f\n", b[n] / c[n][n]);
    }
```

例题 开关问题 AcWing208

有 N 个相同的开关，每个开关都与某些开关有着联系。每当你打开或者关闭某个开关的时候，其他的与此开关相关联的开关也会相应地发生变化，即这些相联系的开关的状态如果原来为开就变为关，如果为关就变为开。

你的目标是经过若干次开关操作后使得最后 N 个开关达到一个特定的状态。对于任意一个开关，最多只能进行一次开关操作。你的任务是，计算有多少种可以达到指定状态的方法（不计开关操作的顺序）。

输入数据的第一行是一个数 $N(0 < N < 29)$；第二行有 N 个 0 或者 1 的数字，表示 N 个开关的初始状态；第三行有 N 个 0 或者 1 的数字，表示 N 个开关的目标状态。接下来每行两个数 i,j，表示如果操作第 i 个开关，第 j 个开关的状态也会变化。

设 x_i 表示第 i 个开关的操作情况，$x_i = 1$ 表示按了这个开关，$x_i = 0$ 表示没有

按。再统计 $a_{i,j}$ 表示第 i 个开关和第 j 个开关的联系情况，$a_{i,j}=1$ 表示按下 j 会影响 i 的状态，$a_{i,j}=0$ 表示不会影响，特别的，令 $a_{i,i}=1$。

一个开关最后的状态 dst_i，取决于它最初的状态 src_i，以及所有与它有联系的开关的操作情况执行异或运算得到的结果。可列出异或方程组：

$$\begin{cases} a_{1,1}x_1 \text{ xor } a_{1,2}x_2 \text{ xor } \cdots \text{ xor } a_{1,n}x_n = src_1 \text{ xor } dst_1 \\ a_{2,1}x_1 \text{ xor } a_{2,2}x_2 \text{ xor } \cdots \text{ xor } a_{2,n}x_n = src_2 \text{ xor } dst_2 \\ \vdots \\ a_{n,1}x_1 \text{ xor } a_{n,2}x_2 \text{ xor } \cdots \text{ xor } a_{n,n}x_n = src_n \text{ xor } dst_n \end{cases}$$

异或其实就是不进位加法，我们仍然可以写出增广矩阵，矩阵中的每个值要么是 0，要么是 1。然后，在执行高斯消元的过程中，把加、减法替换成异或，且不需要执行乘法。最终我们可以得到该异或方程组对应的简化阶梯形矩阵。若存在形如 $0=1$ 的方程，则方程组无解。否则，因为自由元可以取 0 或 1，所以方程组解的数量就是 2^{cnt}，其中 cnt 为自由元的个数。

在编写程序时，为了简便、高效，可以把增广矩阵的每一行进行状态压缩，用一个 int 类型的整数表示 $n+1$ 位二进制数，其中第 0 位为增广矩阵最后一列的常数，第 $1\sim n$ 位分别为增广矩阵第 $1\sim n$ 列的系数。

```
int a[100], n, t, ans;
int main() {
    cin >> t;
    while (t--) {
        cin >> n;
        for (int i = 1; i <= n; i++) scanf("%d", &a[i]);
        for (int i = 1, j; i <= n; i++) {
            scanf("%d", &j);
            a[i] ^= j;
            a[i] |= 1 << i; // a[i][i] = 1;
        }
        int x, y;
        while (~scanf("%d%d", &x, &y) && x && y) {
            a[y] |= 1 << x; // a[y][x] = 1;
        }
        ans = 1;
        for (int i = 1; i <= n; i++) {
            // 找到最大的一个 a[i]，即主元位数最高的 a[i]
            for (int j = i + 1; j <= n; j++)
                if (a[j] > a[i]) swap(a[i], a[j]);
            // 消元完毕，有 i-1 个主元，n-i+1 个自由元
```

```
            if (a[i] == 0) { ans = 1 << (n - i + 1); break; }
            // 出现 0=1 的方程，无解
            if (a[i] == 1) { ans = 0; break; }
            // a[i]最高位的 1 作为主元，消去其他方程该位的系数
            for (int k = n; k; k--)
                if (a[i] >> k & 1) {
                    for (int j = 1; j <= n; j++)
                        if (i != j && (a[j] >> k & 1)) a[j] ^= a[i];
                    break;
                }
        }
        if (ans == 0) puts("Oh,it's impossible~!!");
        else cout << ans << endl;
    }
}
```

线性空间

线性空间是一个关于以下两个运算封闭的向量集合。

1. 向量加法 $a+b$，其中 a,b 均为向量。

2. 标量乘法 $k*a$，也称数乘运算，其中 a 是向量，k 是常数（标量）。

给定若干个向量 $a_1,a_2,\cdots,a_k$，若向量 b 能由 $a_1,a_2,\cdots,a_k$ 经过向量加法和标量乘法运算得出，则称向量 b 能被向量 $a_1,a_2,\cdots,a_k$ **表出**。显然，$a_1,a_2,\cdots,a_k$ 能表出的所有向量构成一个线性空间，$a_1,a_2,\cdots,a_k$ 被称为这个线性空间的**生成子集**。

任意选出线性空间中的若干个向量，如果其中存在一个向量能被其他向量表出，则称这些向量**线性相关**，否则称这些向量**线性无关**。

线性无关的生成子集被称为线性空间的**基底**，简称**基**。基的另一种定义是线性空间的极大线性无关子集。一个线性空间的所有基包含的向量个数都相等，这个数被称为线性空间的**维数**。

例如，平面直角坐标系中的所有向量构成一个二维线性空间，它的一个基就是单位向量集合 {(0,1), (1,0)}。平面直角坐标系的 x 轴上的所有向量构成一个一维线性空间，它的一个基就是 {(1,0)}。

对于一个 n 行 m 列的矩阵，我们可以把它的每一行看作一个长度为 m 的向量，称为“行向量”。矩阵的 n 个行向量能够表出的所有向量构成一个线性空间，这个线性空间的维数被称为矩阵的“行秩”。类似地，我们可以定义列向量和列秩。实际上，矩阵的行秩一定等于列秩，它们都被称为矩阵的**秩**。

把这个 $n*m$ 的矩阵看作“系数矩阵”进行高斯消元（增广矩阵的最后一列全看作零），得到一个简化阶梯形矩阵。显然，简化阶梯形矩阵的所有非零行向量线性无关。因为初等行变换就是行向量之间进行的向量加法与标量乘法运算，所以高斯消元不改变矩阵的行向量表出的线性空间。于是，简化阶梯形矩阵的所有非零行向量就是该线性空间的一个基，非零行向量的个数就是矩阵的秩。

例题 装备购买 AcWing209

脸哥最近在玩一款神奇的游戏，这个游戏里有 n 件装备，每件装备有 m 个属性，用向量 $z[i]=(a_{i,1},a_{i,2},\cdots,a_{i,m})$ 表示，每件装备需要花费 c_i。现在脸哥想买一些装备，但是脸哥很穷，所以总是盘算着怎样才能花尽量少的钱买尽量多的装备。对于脸哥来说，如果一件装备的属性能用购买的其他装备组合出（也就是说，脸哥可以利用手上的这些装备组合出这件装备的效果），那么这件装备就没有买的必要了。严格的定义是，如果脸哥买了 $z[i_1],z[i_2],\cdots,z[i_p]$ 这 p 件装备，并且不存在实数 $b_1,b_2,\cdots,b_p$ 使得 $z[k]=b_1z[i_1]+b_2z[i_2+\cdots+b_pz[i_p]$，那么脸哥就会买 $z[k]$，否则 $z[k]$ 对脸哥就是无用的。脸哥想要在买下最多数量的装备的前提下花最少的钱，你能帮他算一下吗？$1\le n,m\le 500$，$0\le a_{i,j}\le 1000$。

把 n 件装备看作 n 个长度为 m 的向量，根据题意，购买的装备对应的向量应该是线性无关的。要买下最多数量的装备，就是求这 n 个向量表出的线性空间的基。

把 $a_{i,j}(1\le i\le n,1\le j\le m)$ 看作系数矩阵，则每个装备 z_i 都是一个行向量。用高斯消元求出该矩阵的秩，就得到了能买下的装备的最多数量。

本题还要求花最少的钱。我们只需要在高斯消元的过程中，使用贪心策略，对于每个主元 x_i，在前 $i-1$ 列为 0、第 i 列不为 0 的行向量中，选择价格最低的一个，消去其他行中第 i 列的值。

该贪心策略可用反证法证明。假设花费价钱最少的基底是 $z[i_1],z[i_2],\cdots,z[i_p]$，其中不包含价格最低的行向量 $z[k]$。因为基底是极大线性无关子集，所以 $z[k]$ 能被 $z[i_1],z[i_2],\cdots,z[i_p]$ 表出，不妨设 $z[k]=b_1z[i_1]+b_2z[i_2]+\cdots+b_pz[i_p]$。

移项变换可得 $z[i_p]=(z[k]-b_1z[i_1]-\cdots-b_{p-1}z[i_{p-1}])/b_p$，即 $z[i_p]$ 能被 $z[k]$ 和 $z[i_1],z[i_2],\cdots,z[i_{p-1}]$ 表出。故 $z[k],z[i_1],z[i_2],\cdots,z[i_{p-1}]$ 能与 $z[i_1],z[i_2],\cdots,z[i_p]$ 表出相同的线性空间，是一个总价格更低的基底，与假设矛盾。实现细节请参阅配套资源中的程序。

线性空间的概念可以进一步推广，不仅限于向量、向量加法和标量乘法。例如“异或空间”就是一个很常见的形式。异或空间是一个关于异或运算封闭的非负整数集合。可以在异或空间中用类似的方法定义“表出”“线性无关”“基底”等。

若整数 b 能由整数 $a_1,a_2,\cdots,a_k$ 经异或运算得出，则称 b 能被 $a_1,a_2,\cdots,a_k$ 表

出。$a_1,a_2,\cdots,a_k$ 能表出的所有整数构成一个异或空间，$a_1,a_2,\cdots,a_k$ 被称为这个异或空间的**生成子集**。

任意选出异或空间中的若干个整数，如果其中存在一个整数能被其他整数表出，则称这些整数**线性相关**，否则称这些整数**线性无关**。异或空间的**基**就是异或空间中一个线性无关的生成子集，或者定义为异或空间的极大线性无关子集。

给定 n 个 $0\sim2^m-1$ 之间的整数 $a_1,a_2,\cdots,a_n$，如何求出这 n 个整数表出的异或空间的基？我们可以把它们看作 m 位二进制数，写成一个 n 行 m 列的 01 矩阵，矩阵中第 i 行从左到右依次是 a_i 的第 $m-1,m-2,\cdots,1,0$ 位（二进制数的最低位称为第 0 位）。把矩阵作为系数矩阵，用高斯消元求解异或方程组的方法，将其转化为简化阶梯形矩阵。简化阶梯形矩阵的每一行也是一个整数，所有非 0 整数一起构成异或空间的基。

例题 XOR AcWing210

有 n 个整数 $a_1,a_2,\cdots,a_n$ 和 m 个询问，每个询问给出一个整数 k，求从 $a_1,a_2,\cdots,a_n$ 中选出若干个数执行异或运算能够得到的整数集合中（去掉重复的数），第 k 小的整数是多少。$1\le n,m\le 10000,1\le a_i,k\le 10^{18}$。

用高斯消元求出 $a_1,a_2,\cdots,a_n$ 构成的异或空间的基。不妨设这个基由整数 $b_1,b_2,\cdots,b_t$ 构成，其中 $b_1>b_2>\cdots>b_t$。从基底中选取若干个整数，显然有 2^t 种取法。因此，t 维异或空间中共有 2^t 个整数，与这 2^t 种取法一一对应。

在上面的高斯消元过程中，共有 t 个主元，分别是 $b_1,b_2,\cdots,b_t$ 的最高位。设它们的最高位分别是第 $c_1,c_2,\cdots,c_t$ 位，显然 $c_1>c_2>\cdots>c_t$。

我们之前提到过，在简化阶梯形矩阵中，对于每个主元，有且仅有一个位置 (i,j)，满足该主元的系数非 0。第 j 列的其他位置都是 0，第 i 行的第 $1\sim j-1$ 列都是 0。也就是说，在 $b_1,b_2,\cdots,b_t$ 中，有且仅有 b_i 的第 c_i 位是 1，其他数的第 c_i 位都是 0。

例如整数 5,12,2,7,9 写成的 01 矩阵，以及高斯消元得到的简化阶梯形矩阵如下：

$$\begin{matrix}0101\\1100\\0010\\0111\\1001\end{matrix}\quad\Longrightarrow\quad\begin{matrix}1001\\0101\\0010\\0000\\0000\end{matrix}$$

异或空间的基是 9,5,2，最高位的 1 分别在第 3 位、第 2 位、第 1 位，自由元是第 0 位。可以看到，只有 9 的第 3 位是 1，5 和 2 的第 3 位都是 0，即满足“主元所在位为 1”的整数是唯一的。

因为基底 $b_1,b_2,\cdots,b_t$ 能与 $a_1,a_2,\cdots,a_n$ 表出相同的异或空间，所以异或空间中第 k 小的整数，就是从 $b_1,b_2,\cdots,b_t$ 中选出若干个数做异或运算，能表出的第 k 小的

整数。而第 c_1 位是 1 的只有 b_1，所以选取 b_1 一定比不选 b_1 得到的数更大。同理，在选了 b_1 之后，选取 b_2 一定比不选 b_2 得到的数更大。异或空间中最大的数就是 $b_1, b_2, \cdots, b_t$ 全部异或起来得到的数。

综上所述，我们把询问中给出的 $k-1$ 进行二进制分解（减掉 1 是为了把最小的数定义为第 0 小，题目中是第 1 小），如果 $k-1$ 的第 $j\ (0 \le j < t)$ 位等于 1，就选取 b_{t-j}，最后把所有选出的 b 异或起来，就得到了答案。特别地，若 $k > 2^t$，则无解。

值得注意的一点边界情况是，异或空间中第 k 小的整数与题目中所求的第 k 小的整数稍有不同。题目不允许 $a_i \text{ xor } a_i$ 这样的运算，而异或空间中允许。造成的差别是异或空间一定包含整数 0，但从 $a_1, a_2, \cdots, a_n$ 选出几个不同的数异或，可能得不到 0。实际上，我们检查简化阶梯形矩阵中是否存在“零行”，即可判断能否得到 0。若不能得到 0，就把 k 而不是 $k-1$ 进行二进制分解（相当于在统计顺序时跳过“0”这个数）。

```
unsigned long long a[10010];
int n, m, t, T;
int main() {
    cin >> T;
    for (int C = 1; C <= T; C++) {
        cin >> n;
        for (int i = 1; i <= n; i++) scanf("%I64u", &a[i]);
        bool zero = 0;
        t = n;
        for (int i = 1; i <= n; i++) {
            for (int j = i + 1; j <= n; j++)
                if (a[j] > a[i]) swap(a[i], a[j]);
            if (a[i] == 0) { zero = 1, t = i - 1; break; }
            for (int k = 63; k >= 0; k--)
                if (a[i] >> k & 1) {
                    for (int j = 1; j <= n; j++)
                        if (i != j && (a[j] >> k & 1)) a[j] ^= a[i];
                    break;
                }
        }
        cin >> m;
        printf("Case #%d:\n", C);
        while (m--) {
            unsigned long long k, ans = 0;
```

```
            scanf("%I64u", &k);
            if (zero) k--;
            if (k >= 1llu << t) puts("-1");
            else {
                for (int i = t - 1; i >= 0; i--)
                    if (k >> i & 1) ans ^= a[t - i];
                printf("%I64u\n", ans);
            }
        }
    }
}
```

本题中讨论的“异或空间”是一个“去重异或集合”，它包含 2^t 个互不相等的整数。我们还可以进一步关注包含 2^n 个整数（可能相等）的“不去重异或集合”的形态。

一定能从 $a_1, a_2, \cdots, a_n$ 中找出 t 个线性无关的整数，构成异或空间的一个基。不在基底中的整数还有 $n-t$ 个，从这 $n-t$ 个整数中选出若干个，显然有 2^{n-t} 种选法。

对于任意一种选法，设选取的整数异或起来得到 x，把基底能表出的 2^t 个互不相等的整数与 x 分别异或，又得到 2^t 个整数。因为当 $y \neq z$ 时，必然有 $x \operatorname{xor} y \neq x \operatorname{xor} z$，所以刚刚得到的 2^t 个整数也互不相等。换言之，每种取法与基底相组合，恰好遍历“去重异或集合”一次。

综上所述，“不去重异或集合”就是“去重异或集合”中的 2^t **个整数各重复** 2^{n-t} **次**形成的。



0x36 组合计数

加法原理

若完成一件事的方法有 n 类，其中第 i 类方法包括 a_i 种不同的方法，且这些方法互不重合，则完成这件事共有 $a_1 + a_2 + \cdots + a_n$ 种不同的方法。

乘法原理

若完成一件事需要 n 个步骤，其中第 i 个步骤有 a_i 种不同的完成方法，且这些步骤互不干扰，则完成这件事共有 $a_1 * a_2 * \cdots * a_n$ 种不同的方法。

排列数

从 n 个不同元素中依次取出 m 个元素排成一列，产生的不同排列的数量为：

$$A_n^m \text{（也可记作 } P_n^m\text{）} = \frac{n!}{(n-m)!} = n * (n-1) * \cdots * (n-m+1)$$

组合数

从 n 个不同元素中取出 m 个组成一个集合（不考虑顺序），产生的不同集合数量为：

$$C_n^m = \frac{n!}{m!\,(n-m)!} = \frac{n * (n-1) * \cdots * (n-m+1)}{m * (m-1) * \cdots * 2 * 1}$$

性质

1. $C_n^m = C_n^{n-m}$
2. $C_n^m = C_{n-1}^m + C_{n-1}^{m-1}$
3. $C_n^0 + C_n^1 + C_n^2 + \cdots + C_n^n = 2^n$

证明：

由组合数的定义，对于从 n 个不同元素中取出 m 个组成的每个集合，剩余未取出的元素也构成一个集合，两个集合一一对应，所以性质 1 成立。

从 n 个不同元素中取出 m 个组成一个集合有两类方法：取 n 号元素，不取 n 号元素。若取 n 号元素，则应在剩余 $n-1$ 个元素中选出 $m-1$ 个，有 C_{n-1}^{m-1} 种取法。若不取 n 号元素，则应在剩余 $n-1$ 个元素中选出 m 个，有 C_{n-1}^m 种取法。根

据加法原理，性质 2 成立。

从 n 个不同元素中取出若干个元素组成一个集合，有 $n+1$ 类方法，分别是取出 $0,1,2,\cdots,n$ 个。根据加法原理，共有 $C_n^0+C_n^1+C_n^2+\cdots+C_n^n$ 种方法。从另一个方面去想，n 个元素每个元素可以取或不取，总方法数为 2^n，二者相等，故性质 3 成立。

证毕。

组合数的求法

根据性质2，用递推法可求出 $0\le y\le x\le n$ 的所有组合数 C_x^y，时间复杂度 $O(n^2)$。

组合数的结果一般较大，若题目要求出 C_n^m 对一个数 p 取模后的值，并且 $1\sim n$ 都存在模 p 乘法逆元，则可以先计算分子 $n!\bmod p$，再计算分母 $m!(n-m)!\bmod p$ 的逆元，乘起来得到 $C_n^m \bmod p$，复杂度为 $O(n)$。

若在计算阶乘的过程中，把 $0\le k\le n$ 的每个 $k!\bmod p$ 及其逆元分别保存在两个数组 jc 和 jc_inv 中，则可以在 $O(n\log n)$ 的预处理后，以 $O(1)$ 的时间回答 $0\le y\le x\le n$ 的所有组合数 $C_x^y \bmod p = jc[x]*jc_inv[y]*jc_inv[x-y]\bmod p$。

若题目要求我们对 C_n^m 进行高精度运算，为避免除法，可用 0x31 节中“阶乘分解”的做法，把分子、分母快速分解质因数，在数组中保存各项质因子的指数。然后把分子、分母各个质因子的指数对应相减（把分母消去），最后把剩余质因子乘起来，时间复杂度为 $O(n\log n)$。

二项式定理

$$(a+b)^n=\sum_{k=0}^{n}C_n^k a^k b^{n-k}$$

证明：

数学归纳法。当 $n=1$ 时，$(a+b)^1=C_n^0a^0b^1+C_n^1a^1b^0=a+b$ 成立。

假设当 $n=m$ 时命题成立，当 $n=m+1$ 时：

$$(a+b)^{m+1}=(a+b)(a+b)^m=(a+b)\sum_{k=0}^{m}C_m^k a^k b^{m-k}$$

$$=\sum_{k=0}^{m}C_m^k a^{k+1}b^{m-k}+\sum_{k=0}^{m}C_m^k a^k b^{m-k+1}=\sum_{k=1}^{m+1}C_m^{k-1}a^k b^{m-k+1}+\sum_{k=0}^{m}C_m^k a^k b^{m-k+1}$$

$$=\sum_{k=0}^{m+1}\left(C_m^{k-1}+C_m^k\right)a^k b^{m-k+1}=\sum_{k=0}^{m+1}C_{m+1}^k a^k b^{m+1-k}$$

证毕。

例题 计算系数 AcWing211

给定一个多项式 $(ax+by)^k$，求出多项式展开后 $x^n y^m$ 项的系数，对 10007 取模。$0 \le n,m \le k \le 1000$，$n+m=k$，$0 \le a,b \le 10^6$。

根据二项式定理，有 $(ax+by)^k = \sum_{i=0}^{k} C_k^i a^i b^{k-i} x^i y^{k-i}$，所以 $x^n y^m$ 项的系数即为 $C_k^n a^n b^m$，直接计算 $a^n b^m$ 和组合数即可。因为 10007 是质数，所以 1~1000 的整数都存在模 10007 的乘法逆元，组合数中的除法可用逆元转化为乘法计算。

多重集的排列数

多重集是指包含重复元素的广义集合。设 $S=\{n_1 \cdot a_1, n_2 \cdot a_2, \cdots, n_k \cdot a_k\}$ 是由 n_1 个 a_1，n_2 个 a_2，…，n_k 个 a_k 组成的多重集，记 $n=n_1+\cdots+n_k$。S 的全排列个数为：

$$\frac{n!}{n_1!n_2!\cdots n_k!}$$

多重集的组合数

设 $S=\{n_1 \cdot a_1, n_2 \cdot a_2, \cdots, n_k \cdot a_k\}$ 是由 n_1 个 a_1，n_2 个 a_2，…，n_k 个 a_k 组成的多重集，设整数 $r \le n_i\ (\forall i \in [1,k])$。从 S 中取出 r 个元素组成一个多重集（不考虑元素的顺序），产生的不同多重集的数量为：

$$C_{k+r-1}^{k-1}$$

证明：

原问题等价于统计下列集合的数量：$\{x_1 \cdot a_1, x_2 \cdot a_2, \cdots, x_k \cdot a_k\}$，其中 $\sum_{i=1}^{k} x_i = r$ 并且 $x_i \le n_i$。因为 $r \le n_i$，必定有 $x_i \le n_i$，所以只需考虑 $\sum_{i=1}^{k} x_i = r$ 这一个条件。故原问题等价于 r 个 0，$k-1$ 个 1 构成的全排列数—— $k-1$ 个 1 把 r 个 0 分成 k 组，每组的 0 的数量对应 x_i。而多重集 $\{r \cdot 0, (k-1) \cdot 1\}$ 的全排列数为：

$$\frac{(r+k-1)!}{r!\,(k-1)!} = C_{k+r-1}^{r} = C_{k+r-1}^{k-1}$$

对于更为一般的 r 的情况，我们将在下一节继续讨论。

证毕。

例题 Counting Swaps AcWing212

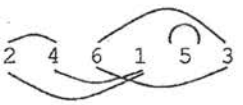
给定一个 $1 \sim n$ 的排列 $p_1, p_2, \cdots, p_n$，可进行若干次操作，每次选择两个整数 x, y，交换 p_x, p_y。设把 $p_1, p_2, \cdots, p_n$ 变成单调递增的排列 $1, 2, \cdots, n$ 至少需要 m 次交换。求有多少种操作方法可以只用 m 次交换达到上述目标。因为结果可能很大，你只需要输出对 10^9+9 取模之后的值。$1 \le n \le 10^5$。

例如排列 2,3,1 至少需要 2 次交换才能变为 1,2,3。操作方法共有三种，分别是：

1. 先交换数字 2,3，变成 3,2,1，再交换数字 3,1，变成 1,2,3；
2. 先交换数字 2,1，变成 1,3,2，再交换数字 3,2，变成 1,2,3；
3. 先交换数字 3,1，变成 2,1,3，再交换数字 2,1，变成 1,2,3。

对于一个排列 $p_1,p_2,\cdots,p_n$，如果从每个 i 向 p_i 连一条边，那么可以得到一张 n 个点 n 条边的图，并且这张图由若干个环构成。

例如排列 2,4,6,1,5,3 对应右图，有 1-2-4，3-6 和 5 三个环。



最后的目标排列为 $1,2,\cdots,n$，显然由 n 个自环构成。

引理：

把一个长度为 n 的环变成 n 个自环，最少需要 $n-1$ 次交换操作。

证明：

首先，把长度为 2 的环变为 2 个自环，显然需要 1 次交换操作。

假设 $\forall k \le n-1$，把长度不超过 k 的环变成 k 个自环最少需要 $k-1$ 次交换操作。当 $k=n$ 时，设该环为 $v_1 \to v_2 \to v_3 \to \cdots \to v_n \to v_1$，交换任意 $v_i,v_j(i<j)$ 的出边后，该环被拆成 $v_{i+1} \to v_{i+2} \to \cdots \to v_j \to v_{i+1}$ 和 $v_1 \to v_2 \to \cdots \to v_i \to v_{j+1} \to v_{j+2} \to \cdots \to v_n \to v_1$ 两个环，二者的长度分别是 $j-i$ 和 $n-(j-i)$。把二者分别拆成自环，所需的最小交换次数为 $(j-i-1)+(n-(j-i)-1)=n-2$，再加上刚才 v_i,v_j 的一次交换，共需 $n-1$ 次交换操作。最后由数学归纳法可知，原命题成立。

证毕。

设 F_n 表示用最少的步数把一个长度为 n 的环变成 n 个自环，共有多少种操作方法。由上面的证明过程可知，把一个长度为 n 的环变成 n 个自环的过程中，可以把该环拆成长度为 x 和 y 的两个环，其中 $x+y=n$。设 $T(x,y)$ 表示有多少种交换方法可以把长度为 n 的环变成长度为 x 和 y 的两个环，容易发现：

$$T(x,y)=\begin{cases} n/2 & n\text{是偶数并且}x=y \\ n & n\text{是奇数或}x\neq y \end{cases}$$

另外，二者各自变为自环的方法数为 F_x 和 F_y，步数为 $x-1$ 和 $y-1$。

根据多重集的排列数、加法原理和乘法原理：

$$F_n=\sum_{x+y=n} T(x,y)*F_x*F_y*\frac{(n-2)!}{(x-1)!(y-1)!}$$

如果最初的排列 $p_1,p_2,\cdots,p_n$ 由长度为 $l_1,l_2,\cdots,l_k$ 的 k 个环构成，其中 $l_1+l_2+\cdots+l_k=n$，那么最终的答案就是：

$$F_{l_1} * F_{l_2} * \cdots * F_{l_k} * \frac{(n-k)!}{(l_1-1)! * (l_2-1)! * \cdots * (l_k-1)!}$$

10^9+9 是一个质数，可以用乘法逆元处理上面公式中的除法。整个算法的时间复杂度为 $O(n^2)$。事实上，我们递推求出 F_n 的前几项找规律，或者把前几项输入到 OEIS 网站[①]中，可发现通项公式 $F_n = n^{n-2}$，从而进一步优化到 $O(n \log n)$。

Lucas 定理

若 p 是质数，则对于任意整数 $1 \le m \le n$，有：

$$C_n^m \equiv C_{n \bmod p}^{m \bmod p} * C_{n/p}^{m/p} \pmod p$$

也就是把 n 和 m 表示成 p 进制数，对 p 进制下的每一位分别计算组合数，最后再乘起来。证明一般需要用到生成函数知识，此处省略，感兴趣的读者可查阅相关资料。

例题 古代猪文 AcWing213

给定整数 $q,n\ (1 \le q,n \le 10^9)$，计算 $q^{\sum_{d|n} C_n^d} \bmod 999911659$。

若 $q = 999911659$，则上式结果为 0。否则，因为 999911659 是质数，所以 q,n 互质。由欧拉定理的推论得：

$$q^{\sum_{d|n} C_n^d} \equiv q^{\sum_{d|n} C_n^d \bmod 999911658} \pmod{999911659}$$

因此，本题的关键是计算 $\sum_{d|n} C_n^d \bmod 999911658$。

尝试分解质因数，可以发现 $999911658 = 2 \times 3 \times 4679 \times 35617$。我们称这样的数为 square-free-number，它的所有质因子的指数都是 1。

我们可以枚举 n 的约数 d，然后运用 Lucas 定理求组合数 C_n^d，分别计算出 $\sum_{d|n} C_n^d$ 对 2,3,4679,35617 四个质数取模的结果，记为 a_1, a_2, a_3, a_4。在计算过程中，对于一个质数 p，预处理 p 以内的所有阶乘以及阶乘的模 p 乘法逆元，就能快速计算按照 p 进制表示之后，每一位对应的组合数。

最后，用中国剩余定理求解线性同余方程组：

$$\begin{cases} x \bmod 2 = a_1 \\ x \bmod 3 = a_2 \\ x \bmod 4679 = a_3 \\ x \bmod 35617 = a_4 \end{cases}$$

① oeis.org 是一个数列检索网站，它的数据库中保存了大量的数列信息。只需输入若干个数值，OEIS 会自动检索匹配这些数值的数列，并给出数列的结果、递推与通项公式、相关性质等。

即可得到 $\sum_{d|n} C_n^d \bmod 999911658$ 的最小非负整数解 x。再用快速幂求 q^x 即可得到原问题的答案。

Catalan 数列

给定 n 个 0 和 n 个 1，它们按照某种顺序排成长度为 $2n$ 的序列，满足任意前缀中 0 的个数都不少于 1 的个数的序列的数量为：

$$Cat_n = \frac{C_{2n}^n}{n+1}$$

证明：

令 n 个 0 和 n 个 1 随意排成一个长度为 $2n$ 的序列 S，若 S 不满足任意前缀中 0 的个数都不少于 1 的个数，则存在一个最小的位置 $2p+1 \in [1,2n]$，使得 $S[1\sim 2p+1]$ 中有 p 个 0、$p+1$ 个 1。而把 $S[2p+2\sim 2n]$ 中的所有数字取反后，包含 $n-p-1$ 个 0 和 $n-p$ 个 1。于是我们得到了由 $n-1$ 个 0 和 $n+1$ 个 1 排成的序列。

同理，令 $n-1$ 个 0 和 $n+1$ 个 1 随意排成一个长度为 $2n$ 的序列 S'，也必定存在一个最小的位置 $2p'+1$，使得 $S'[1\sim 2p'+1]$ 中有 p' 个 0、$p'+1$ 个 1。把 S' 后面剩下的一半取反，就得到了由 n 个 0 和 n 个 1 排成的、存在一个前缀 0 比 1 多的序列。

因此，以下两种序列构成一个双射。

1. 由 n 个 0 和 n 个 1 排成的、存在一个前缀 0 比 1 多的序列。

2. 由 $n-1$ 个 0 和 $n+1$ 个 1 排成的序列。

根据组合数的定义，后者显然有 C_{2n}^{n-1} 个。

综上所述，由 n 个 0 和 n 个 1 排成的，任意前缀中 0 的个数都不少于 1 的个数的序列数量为：

$$C_{2n}^n - C_{2n}^{n-1} = \frac{(2n)!}{n!\,n!} - \frac{(2n)!}{(n-1)!\,(n+1)!} = \frac{C_{2n}^n}{n+1} = Cat_n$$

证毕。

推论

以下问题都与 Catalan 数有关。

1. n 个左括号和 n 个右括号组成的合法括号序列的数量为 Cat_n。

2. $1,2,\cdots,n$ 经过一个栈，形成的合法出栈序列的数量为 Cat_n。

3. n 个节点构成的不同二叉树的数量为 Cat_n。

4. 在平面直角坐标系上，每一步只能向上或向右走，从 $(0,0)$ 走到 (n,n) 并且除两个端点外不接触直线 $y=x$ 的路线数量为 $2Cat_{n-1}$。

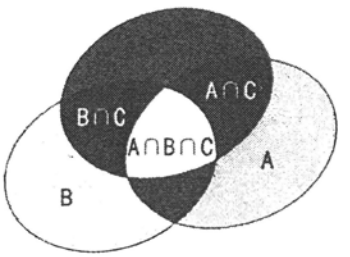
0x37 容斥原理与 Möbius 函数

容斥原理

设 $S_1, S_2, \cdots, S_n$ 为有限集合，$|S|$ 表示集合 S 的大小，则：

$$\left|\bigcup_{i=1}^{n} S_i\right| = \sum_{i=1}^{n}|S_i| - \sum_{1\le i<j\le n}|S_i \cap S_j| + \sum_{1\le i<j<k\le n}|S_i \cap S_j \cap S_k| + \cdots + (-1)^{n+1}|S_1 \cap \cdots \cap S_n|$$

我们可以简单地用文氏图来宏观地描述容斥原理，如下图所示。



多重集的组合数

设 $S = \{n_1 \cdot a_1, n_2 \cdot a_2, \cdots, n_k \cdot a_k\}$ 是由 n_1个a_1，n_2个a_2，…，n_k个a_k 组成的多重集。设 $n = \sum_{i=1}^{k} n_i$，对于任意整数 $r \le n$，从 S 中取出 r 个元素组成一个多重集（不考虑顺序），产生的不同多重集的数量为：

$$C_{k+r-1}^{k-1} - \sum_{i=1}^{k} C_{k+r-n_i-2}^{k-1} + \sum_{1\le i<j\le k} C_{k+r-n_i-n_j-3}^{k-1} - \cdots + (-1)^k C_{k+r-\sum_{i=1}^{k} n_i-(k+1)}^{k-1}$$

证明：

不考虑 n_i 的限制，从 S 中任选 r 个元素，相当于从多重集 $\{\infty \cdot a_1, \infty \cdot a_2, \cdots, \infty \cdot a_k\}$ 中取出 r 个元素。根据我们在 0x36 节中对多重集组合数的讨论，方法数为 C_{k+r-1}^{k-1}。

设 S_i $(1 \le i \le k)$ 表示至少包含 $n_i + 1$ 个 a_i 的多重集。我们先从 S 中取出

n_i+1 个 a_i，然后再任选 $r-n_i-1$ 个元素，即可构成 S_i。与上面同理，可以构成的不同 S_i 的数量为 $C_{k+r-n_i-2}^{k-1}$。

进一步地，先从 S 中取出 n_i+1 个 a_i 和 n_j+1 个 a_j，然后再任选 $r-n_i-n_j-2$ 个元素，即可构成 $S_i \cap S_j$，方法数为：

$$C_{k+r-n_i-n_j-3}^{k-1}$$

根据容斥原理，至少有一种 a_i 选取的数量超过 n_i 限制的多重集共有：

$$\left|\bigcup_{i=1}^{k} S_i\right| = \sum_{i=1}^{k} C_{k+r-n_i-2}^{k-1} - \sum_{1\le i<j\le k} C_{k+r-n_i-n_j-3}^{k-1} + \cdots + (-1)^{k+1} C_{k+r-\sum_{i=1}^{k} n_i-(k+1)}^{k-1}$$

故满足所有限制的合法多重集共有 $C_{k+r-1}^{k-1} - \left|\bigcup_{i=1}^{k} S_i\right|$ 个，两式相减即得原命题。证毕。

例题 **Devu and Flowers** AcWing214

Devu 有 N 个盒子，第 i 个盒子中有 A_i 枝花。同一个盒子内的花颜色相同，不同盒子内的花颜色不同。Devu 要从这些盒子中选出 M 枝花组成一束，求共有多少种方案。若两束花每种颜色的花的数量都相同，则认为这两束花是相同的方案。输出对 10^9+7 取模之后的结果即可。$1 \le N \le 20, 0 \le M \le 10^{14}, 0 \le A_i \le 10^{12}$。

设第 i 个盒子中的花颜色为 B_i，则本题等价于从多重集 $S=\{A_1 \cdot B_1, A_2 \cdot B_2, \cdots, A_n \cdot B_n\}$ 中选出 M 个元素能够产生的不同多重集的数量。根据多重集组合数的结论，本题的答案为：

$$C_{N+M-1}^{N-1} - \sum_{i=1}^{N} C_{N+M-A_i-2}^{N-1} + \sum_{1\le i<j\le N} C_{N+M-A_i-A_j-3}^{N-1} - \cdots + (-1)^N C_{N+M-\sum_{i=1}^{N} A_i-(N+1)}^{N-1}$$

在具体实现时，我们可以枚举 $x=0\sim 2^N-1$，若 x 在二进制表示下共有 p 位为 1，分别是第 $i_1, i_2, \cdots, i_p$ 位，则这个 x 就代表上式中的一项：

$$(-1)^p C_{N+M-A_{i_1}-A_{i_2}-\cdots-A_{i_p}-(p+1)}^{N-1}$$

特别地，$x=0$ 代表 C_{N+M-1}^{N}。这样我们就可以得到容斥原理计算多重集组合数的公式的每一项。

对于每一项组合数，N 的范围很小，但 M 的范围很大。我们以 C_{N+M-1}^{N-1} 为例，因为 $C_{N+M-1}^{N-1} = P_{N+M-1}^{N-1}/(N-1)!$，所以可以先计算排列数 $P_{N+M-1}^{N-1} = (N+M-1) * (N+M-2) * \cdots * (M+1)$，然后再乘上 $(N-1)!$ 的乘法逆元。我们还可以应用 Lucas 定理，先把 $N+M-1$ 对 10^9+7 取模，再计算组合数，避免乘法结果超出 64 位整数的表示范围。

```
const int mod = 1000000007;
long long a[22], m, ans = 0;
int inv[22], n;

int power(int a, int b) {  // a^b
    int c = 1;
    for (; b; b>>=1) {
        if (b&1) c = (long long)c * a % mod;
        a = (long long)a * a % mod;
    }
    return c;
}

// C(y,x)
int C(long long y, int x) {
    if (y < 0 || x < 0 || y < x) return 0;
    y %= mod;
    if (y == 0 || x == 0) return 1;
    int ans = 1;
    for (int i = 0; i < x; i++)
        ans = (long long)ans * (y - i) % mod;
    for (int i = 1; i <= x; i++)
        ans = (long long)ans * inv[i] % mod;
    return ans;
}
int main() {
    for (int i = 1; i <= 20; i++)
        inv[i] = power(i, mod-2);
    cin >> n >> m;
    for (int i = 1; i <= n; i++) cin >> a[i];
    for (int x = 0; x < 1<<n; x++) {
        if (x == 0) {
            ans = (ans + C(n+m-1, n-1)) % mod;
        }
        else {
            long long t = n + m;
            int p = 0;
            for (int i = 0; i < n; i++)
```

```
                if (x >> i & 1) {
                    p++;
                    t -= a[i + 1];
                }
            t -= p+1;
            if (p&1) {
                ans = (ans - C(t, n-1)) % mod;
            }
            else {
                ans = (ans + C(t, n-1)) % mod;
            }
        }
    }
    cout << (ans + mod) % mod << endl;
}
```

Möbius 函数

设正整数 N 按照算术基本定理分解质因数为 $N = p_1^{c_1}p_2^{c_2}\cdots p_m^{c_m}$，定义函数

$$\mu(N) = \begin{cases} 0 & \exists i \in [1,m], c_i > 1 \\ 1 & m \equiv 0 \pmod 2, \forall i \in [1,m], c_i = 1 \\ -1 & m \equiv 1 \pmod 2, \forall i \in [1,m], c_i = 1 \end{cases}$$

称 $\mu(N)$ 为 Möbius 函数（莫比乌斯函数）。

通俗地讲，当 N 包含相等的质因子时，$\mu(N) = 0$。当 N 的所有质因子各不相等时，若 N 有偶数个质因子，$\mu(N) = 1$；若 N 有奇数个质因子，$\mu(N) = -1$。

若只求一项 Möbius 函数，则分解质因数即可计算。若求 $1 \sim N$ 的每一项 Möbius 函数，可以利用 Eratosthenes 筛法计算。把所有 μ 值初始化为 1。接下来，对于筛出的每个质数 p，令 $\mu(p) = -1$，并扫描 p 的倍数 $x = 2p, 3p, \cdots, \lfloor n/p \rfloor * p$，检查 x 能否被 p^2 整除。若能，则令 $\mu(x) = 0$，否则令 $\mu(x) = -\mu(x)$。

```
for (int i = 1; i <= n; i++) miu[i] = 1, v[i] = 0;
for (int i = 2; i <= n; i++) {
    if (v[i]) continue;
    miu[i] = -1;
    for (int j = 2 * i; j <= n; j += i) {
        v[j] = 1;
        if ((j / i) % i == 0) miu[j] = 0;
        else miu[j] *= -1;
    }
}
```

例题 Zap AcWing215

有 50000 组询问，每次询问给定三个整数 a, b, k，求有多少对二元组 (x, y) 满足 $x \le a$，$y \le b$ 并且 $\gcd(x, y) = k$。$1 \le k \le a, b \le 50000$。

求有多少对二元组 (x,y) 满足 $x \le a$，$y \le b$ 并且 $\gcd(x,y) = k$，等价于求有多少对二元组 (x,y) 满足 $x \le a/k$，$y \le b/k$ 并且 x,y 互质。

设 $D[a,b,k]$ 表示满足 $x \le a$，$y \le b$ 且 $k|\gcd(x,y)$ 的二元组有多少对。显然只要 x,y 都是 k 的倍数即可。$1 \sim a$ 之间 k 的倍数有 $\lfloor a/k \rfloor$ 个。故 $D[a,b,k] = \lfloor a/k \rfloor \lfloor b/k \rfloor$。

设 $F[a,b]$ 表示满足 $x \le a$，$y \le b$ 且 x,y 互质的二元组有多少对，由容斥原理：

$$F[a,b] = \sum_{i=1}^{\min(a,b)} \mu(i) * D[a,b,i]$$

上式的意思是，没有任何限制的二元组总数为 $D[a,b,1] = a * b$。应当减去 $\gcd(x,y)$ 是 2，3，5，…的倍数的二元组数量。这样又重复减掉了 $\gcd(x,y)$ 既是 2 的倍数又是 3 的倍数的二元组数量，应该加回来。依此类推，$D[a, b, i]$ 的系数恰好就是 Möbius 函数。

回顾 0x32 节中例题“余数之和”的讨论，我们知道：$\forall i \in [x, \min(\lfloor a/\lfloor a/x \rfloor \rfloor, \lfloor b/\lfloor b/x \rfloor \rfloor)]$，$D[a,b,i] = \lfloor a/i \rfloor \lfloor b/i \rfloor$ 的值都相等，预处理出 Möbius 函数的前缀和，即可直接累加这一段的答案。这样的段只有 $O(\sqrt{a} + \sqrt{b})$ 个。

0x38 概率与数学期望

在概率论中，我们把一个随机实验的某种可能结果称为“样本点”，把所有可能结果构成的集合称为“样本空间”。在一个给定的样本空间中，**随机事件**就是样本空间的子集，即由若干个样本点构成的集合，**随机变量**就是把样本点映射为实数的函数。随机变量分为离散型与连续型两种，我们主要讨论离散型随机变量，也就是取值有限或可数的随机变量。

定义

设样本空间为 Ω，若对于 Ω 中的每一个随机事件 A，都存在实值函数 $P(A)$，满足：

1. $P(A) \geq 0$；
2. $P(\Omega) = 1$；
3. 对于若干个两两互斥事件 $A_1, A_2, \cdots$，有 $\sum P(A_i) = P(\bigcup A_i)$，其中 $\cup$ 表示并集。

则称 $P(A)$ 为随机事件 A 发生的**概率**。通俗地讲，概率是对随机事件发生可能性的度量，是一个 0~1 之间的实数。

定义

若随机变量 X 的取值有 $x_1, x_2, \cdots$，一个随机事件可表示为 $X = X_i$，其概率为 $P(X = X_i) = p_i$，则称 $E(X) = \sum p_i x_i$ 为随机变量 X 的**数学期望**。通俗地讲，数学期望是随机变量取值与概率的乘积之和。

例如，在掷两枚骰子的点数实验中，样本空间是由 36 个样本点构成的集合，每个样本点可写作 (a, b)，其中 $1 \leq a, b \leq 6$。随机变量有多种，不妨以“掷出的点数之和 X”为例，则随机变量 X 的取值为 2~12。随机事件可描述为“掷出 X 点”，即由 $a + b = X$ 的样本点 (a, b) 构成的子集。掷出 8 点的概率 $P(X = 8) = 5/36$。掷出的点数的数学期望为：

$$\frac{1}{36} * (2 + 12) + \frac{1}{18} * (3 + 11) + \frac{1}{12} * (4 + 10) + \frac{1}{9} * (5 + 9) + \frac{5}{36} * (6 + 8) + \frac{1}{6} * 7 = 7$$

对于条件概率、方差以及各种常见的离散型随机变量的分布，读者在高中数学课本中应该已经学到，这里就不再赘述。

性质

数学期望是线性函数，满足 $E(aX+bY)=a*E(X)+b*E(Y)$。

该性质非常重要，是我们能够对数学期望进行递推求解的基本依据，请务必牢记。有了这条性质，在上面的例子中计算“掷两枚骰子的点数”的数学期望会非常容易。设随机变量 X 表示掷 1 枚骰子的点数，显然期望值 $E(X)=(1+2+3+4+5+6)/6=3.5$。掷 2 枚骰子的点数可表示为随机变量 $2X$，于是 $E(2X)=2E(X)=2*3.5=7$。

例题 **Rainbow 的信号** AcWing216

Freda 发明了传呼机之后，Rainbow 进一步改进了传呼机发送信息所使用的信号。因为现在是数字信息时代，Rainbow 发明的信号可以用 N 个自然数 $A_1,A_2,\cdots,A_N$ 表示。为了避免两个人的对话被大坏蛋 VariantF 偷听，Rainbow 把对话分成 A，B，C 三部分，分别用 a，b，c 三个密码加密。现在 Freda 接到了 Rainbow 的信息，她的首要工作就是解密。Freda 了解到，这三部分的密码计算方式如下：

在 $1\sim N$ 这 N 个数中，等概率地选取两个数 l,r，如果 $l>r$，则交换 l,r。把信号中的第 l 个数到第 r 个数取出来，构成一个数列 P。

A 部分对话的密码是数列 P 的 xor 和的数学期望值。xor 和就是数列 P 中所有数异或起来得到的数。

B 部分对话的密码是数列 P 的 and 和的期望，and 和的定义类似于 xor 和。

C 部分对话的密码是数列 P 的 or 和的期望，or 和的定义类似于 xor 和。

请你帮 Freda 计算这三个密码。

$1\le N\le 10^5$，N 个自然数均不超过 10^9。

位运算是不进位的，各位之间互不影响。因此，我们可以把 N 个自然数 $A_1,A_2,\cdots,A_N$ 都分成 31 位，对每一位分别进行处理。设 B 是一个 01 序列，B_i 等于 A_i 的第 k 位。

按照题目中 l,r 的选取方式，长度为 1 的区间（即 $l=r$ 时）被选出的概率为 $1/N^2$，其他区间被选出的概率为 $2/N^2$。我们先 $O(N)$ 检查序列 B 中的每个数是否为 1，若是，则累加 $2^k*1/N^2$ 到答案中。于是，接下来我们只需统计 and 和、or 和或 xor 和为 1，并且长度 ≥ 2 的区间个数，即可进一步得到数学期望。

依次枚举右端点 r，设 $last[k]$ $(k=0,1)$ 表示数字 k 上一次出现的位置。

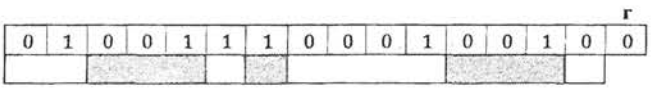
对于 and 和，当 $B[r]=1$ 并且 $l\in[last[0]+1,r-1]$ 时，$B[l\sim r]$ 的 and 和为 1，否则为 0。所以，若 $B[r]=1$，则可以累加 $2^k*\big((r-1)-(last[0]+1)+1\big)*2/N^2$ 到答案中。

对于 or 和，当 $B[r]=1$ 时，l 取任何值，$B[l\sim r]$ 的 or 和都是 1，此时可累加 $2^k*(r-1)*2/N^2$ 到答案中。当 $B[r]=0$ 时，若 $l\in[1,last[1]]$，则 $B[l\sim r]$ 的 or

和为 1，否则为 0。所以，若 $B[r]=0$，则可以累加 $2^k * last[1] * 2/N^2$ 到答案中。

对于 xor 和，试想，我们枚举 l 并向左扫描，则遇到数字 0 时 $B[l\sim r]$ 的 xor 和不变，遇到数字 1 时 $B[l\sim r]$ 的 xor 和取反。如果我们以所有的数字 1 为分界，把序列 B 分成几段，那么同一段中的位置作为 l 时，$B[l\sim r]$ 的 xor 和也相等。来自相邻两段的两个位置分别作为 l 时，$B[l\sim r]$ 的 xor 和恰好相反。

如下图所示，当 l 取灰色位置时，$B[l\sim r]$ 的 xor 和为 1。当 l 取白色位置时，$B[l\sim r]$ 的 xor 和为 0。



我们建立两个变量 c_1, c_2，c_1 记录从 $r-1$ 倒着往前数，第 1,3,5,… 段的总长度（图中白色部分的总长度），c_2 记录第 2,4,6,… 段的总长度（图中灰色部分的总长度）。当 $B[r]=0$ 时，c_2 就是使得 $B[l\sim r]$ 的 xor 和为 1 的 l 的数量，可把 $2^k * c_2 * 2/N^2$ 累加到答案中。当 $B[r]=1$ 时，c_1 就是使得 $B[l\sim r]$ 的 xor 和为 1 的 l 的数量，可把 $2^k * c_1 * 2/N^2$ 累加到答案中。

在右端点 r 增长为 $r+1$ 时，令 $c_1 = c_1 + 1$。若 $B[r]=1$，则还需交换 c_1, c_2。

整个算法的时间复杂度为 $O(N \log \text{MAX_INT})$，实现细节请参阅配套资源中的程序。

例题 绿豆蛙的归宿 AcWing217

给出一个有向无环图，起点为 1，终点为 N，图中每条边都有一个长度。另外，数据保证从起点出发能够到达图中所有的点，图中所有的点也都能够到达终点。

一只绿豆蛙从起点出发，走向终点。到达每一个顶点时，若有 K 条离开该点的道路，它会等概率地选择任意一条道路离开该点，即走向每条路的概率都是 $1/K$。求它从起点走到终点所经过的路径的期望长度是多少。

$1 \le N \le 10^5$，$1 \le M \le 2N$。四舍五入保留两位小数输出。

设 $F[x]$ 表示从节点 x 走到终点所经过的路径的期望长度。若从 x 出发有 k 条边，分别到达 $y_1, y_2, \cdots, y_k$，边长分别为 $z_1, z_2, \cdots, z_k$，则根据数学期望的定义和性质，有：

$$F[x] = \frac{1}{k}\sum_{i=1}^{k}(F[y_i] + z_i)$$

显然 $F[N]=0$，我们的目标是求出 $F[1]$。故我们从终点出发，在反图上（每条边方向取反的图上）执行拓扑排序，在拓扑排序的过程中顺便计算 $F[x]$ 即可。

整个算法的时间复杂度为 $O(N+M)$。

```
int ver[200002], edge[200002], next[200002], head[100002];
int out[100002], deg[100002];
int n, m, tot;
double dis[100002];
queue<int> q;
void add(int x, int y, int z) {
    ver[++tot] = y, edge[tot] = z, next[tot] = head[x], head[x] =
tot;
}

int main() {
    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        int x, y, z;
        scanf("%d%d%d", &x, &y, &z);
        add(y, x, z); // 建反图
        deg[x]++; out[x]++;
    }
    q.push(n);
    while (q.size()) {
        int x = q.front(); q.pop();
        for (int i = head[x]; i; i = next[i]) {
            int y = ver[i];
            dis[y] += (dis[x] + edge[i]) / deg[y];
            out[y]--;
            if (out[y] == 0) q.push(y);
        }
    }
    printf("%.2f\n", dis[1]);
}
```

例题 扑克牌 AcWing218

Zhq 生日那天，Rainbow 来找 Zhq 玩扑克牌。玩着玩着 Rainbow 觉得太没意思了，于是决定给 Zhq 一个考验。

Rainbow 把一副扑克牌（54 张）随机洗匀，倒扣着放成一摞。然后 Zhq 从上往下依次翻开每张牌，每翻开一张黑桃、红桃、梅花或者方块，就把它放到对应花色的堆里去。

Rainbow 想问问 Zhq，得到 A 张黑桃、B 张红桃、C 张梅花、D 张方块需要翻开的牌的张数的期望值 E 是多少。

特殊地，如果翻开的牌是大王或者小王，那么 Zhq 可以把它作为某种花色的牌放入对应堆中。Zhq 会采用最优选择，使得放入之后 E 的值尽可能小。

因为 Zhq 和 Rainbow 还在玩扑克，所以这个程序就交给你来写了。

$0 \le A, B, C, D \le 15$。四舍五入保留三位小数输出。

本题是一道数学期望与动态规划结合的题目，可采用记忆化搜索来实现。对动态规划不熟悉的读者可在学习第 0x50 章“动态规划”的前三节后再尝试求解本题。

设 $F[a,b,c,d,x,y]$ 表示当前已经翻开了 a 张黑桃、b 张红桃、c 张梅花、d 张方块，并且小王状态为 x，大王状态为 y 时的期望值。具体来说，$x=4$ 表示没有用过小王，$x=0\sim3$ 表示用过小王，且把小王作为相应的花色（0 代表黑桃，1 代表红桃，2 代表梅花，3 代表方块）。大王的记录方法同理。

当前已经翻开的牌总数 $sum=(a+b+c+d+(x\neq4)+(y\neq4))$。到目前为止，还剩下 $54-sum$ 张牌，其中有 $13-a$ 张黑桃。故翻开一张黑桃的概率为 $(13-a)/(54-sum)$。翻开这张黑桃后，还需要翻开的牌的期望张数为 $F[a+1,b,c,d,x,y]$。对于红桃、梅花、方块，情况类似。

特别地，当 $x=4$ 时，有 $1/(54-sum)$ 的概率翻开小王。根据题意，应选择把小王看作某种花色，使期望值尽量小，即 $\min_{0\le x'\le3}\{F[a,b,c,d,x',y]\}$。对于大王，情况类似。

综上所述，有：$F[a,b,c,d,x,y]=1+$

$$\frac{13-a}{54-sum}*F[a+1,b,c,d,x,y]+\frac{13-b}{54-sum}*F[a,b+1,c,d,x,y]+$$

$$\frac{13-c}{54-sum}*F[a,b,c+1,d,x,y]+\frac{13-d}{54-sum}*F[a,b,c,d+1,x,y]+$$

$$\frac{1}{54-sum}*\min_{0\le x'\le3}\{F[a,b,c,d,x',y]\}\ (if\ x=4)+$$

$$\frac{1}{54-sum}*\min_{0\le y'\le3}\{F[a,b,c,d,x,y']\}\ (if\ y=4)$$

初值（边界）：若已经翻开的牌数达到了题目要求的数量，则期望值为 0。例如已经翻开的黑桃张数为 $a+(x==0)+(y==0)$，其余同理。若 54 张牌被翻完仍未达到，即 $54-sum\le0$，则期望值为正无穷。

目标：$F[0,0,0,0,4,4]=?$

值得一提的是，在数学期望递推、数学期望动态规划中，我们通常把终止状态（翻开的牌数达到要求）作为初值，把起始状态（尚未翻开任何牌）作为目标，倒着进行计算。这是因为在很多情况下，起始状态是唯一的（0,0,0,0,4,4），而终止状态很多（只要

各花色翻开的牌数都足够即可）。根据数学期望的定义，若我们正着计算，则还需求出从起始状态到达每个终止状态的概率，与 F 值相乘求和才能得到答案，增加了难度，也容易出错。而如果倒着计算，因为起始状态 0,0,0,0,4,4 唯一，所以它的概率一定是 1，直接输出 $F[0,0,0,0,4,4]$ 即为所求。

0x39 0/1 分数规划

0/1 分数规划模型是指，给定整数 $a_1, a_2, \cdots, a_n$ 以及 $b_1, b_2, \cdots, b_n$，求一组解 x_i ($1 \le i \le n$, $x_i = 0$ 或 1)，使下式最大化：

$$\frac{\sum_{i=1}^{n} a_i * x_i}{\sum_{i=1}^{n} b_i * x_i}$$

通俗地说，就是给定 n 对整数 a_i, b_i，从中选出若干对，使选出的数对的 a 之和与 b 之和的商最大。

我们不妨随便猜测一个值 L，然后考虑这样一个问题：是否存在一组解 $\{x_1, x_2, \cdots, x_n\}$ 满足 $\sum_{i=1}^{n}(a_i - L * b_i) * x_i \ge 0$。

1. 如果存在一组解，使得 $\sum_{i=1}^{n}(a_i - L * b_i) * x_i \ge 0$，那么变形得 $(\sum_{i=1}^{n} a_i * x_i) - L * (\sum_{i=1}^{n} b_i * x_i) \ge 0$。进一步可知：

$$\exists \{x_1, x_2, \cdots, x_n\}, \text{使得} \frac{\sum_{i=1}^{n} a_i * x_i}{\sum_{i=1}^{n} b_i * x_i} \ge L$$

也就是说，L 比我们求的最大值要小。

2. 如果对于任意一组解，都有 $\sum_{i=1}^{n}(a_i - L * b_i) * x_i < 0$，那么同理可知：

$$\forall \{x_1, x_2, \cdots, x_n\}, \text{都有} \frac{\sum_{i=1}^{n} a_i * x_i}{\sum_{i=1}^{n} b_i * x_i} < L$$

也就是说，L 比我们求的最大值要大。

这与“二分答案”的情形非常相似。虽然最终的答案是未知的，但是如果我们猜测一个值 L，就能通过判定“是否存在一组解满足 $\sum_{i=1}^{n}(a_i - L * b_i) * x_i \ge 0$”，得到 L 应该变得更大还是更小。也就是说，这个 L 关于“解的存在性”具有“单调性”。

如何判定“是否存在一组解满足 $\sum_{i=1}^{n}(a_i - L * b_i) * x_i \ge 0$”呢？我们考虑这样一个问题：给定整数 $a_1, a_2, \cdots, a_n, b_1, b_2, \cdots, b_n$ 以及 L，求一组解 x_i ($1 \le i \le n$, $x_i = 0$ 或 1)，使 $\sum_{i=1}^{n}(a_i - L * b_i) * x_i$ 最大化。显然，只需判定这个最大值是否非负即可。

这个判定问题显然比原问题要简单得多——因为 $a_i - L * b_i$ 可以直接被计算出来。这就等价于从 n 个数里选出若干个，使它们的和最大。把所有的正数加起来就可以了。

综上所述，我们可以二分答案（实数）。当二分的值为 mid 时，我们就计算

$\sum_{i=1}^{n}(a_i - mid * b_i) * x_i$ 的最大值，检查最大值是否非负。若非负，则令 $l = mid$，否则令 $r = mid$。当二分停止时，就得到了 0/1 分数规划问题的解。

0x3A 节的练习中包含一道 0/1 分数规划的基本题目。另外，许多 0/1 分数规划题目与图论有所结合，0x62 和 0x65 节还会继续探讨两道 0/1 分数规划的例题。

除了二分法外，0/1 分数规划还有一种求解方法——Dinkelbach 迭代法。学有余力的读者可以自行查阅相关资料。

0x3A 博弈论之 SG 函数

在许多地方曾经流行过这样一个小游戏：摆出三堆硬币，分别包含 3 枚、5 枚、7 枚。两人轮流行动，每次可以任选一堆，从中取走任意多枚硬币，可把一堆取光，但不能不取。取走最后一枚硬币者获得胜利。

这类游戏可以推广为更加一般的形式：

给定 n 堆物品，第 i 堆物品有 A_i 个。两名玩家轮流行动，每次可以任选一堆，取走任意多个物品，可把一堆取光，但不能不取。取走最后一件物品者获胜。两人都采取最优策略，问先手能否必胜。

我们把这种游戏称为 NIM **博弈**。把游戏过程中面临的状态称为**局面**。整局游戏第一个行动的称为**先手**，第二个行动的称为**后手**。若在某一局面下无论采取何种行动，都会输掉游戏，则称该局面**必败**。所谓采取最优策略是指，若在某一局面下存在某种行动，使得行动后对手面临必败局面，则优先采取该行动。同时，这样的局面被称为**必胜**。我们讨论的博弈问题一般都只考虑理想情况，即两人均无失误，都采取最优策略行动时游戏的结果。NIM 博弈不存在平局，只有**先手必胜**和**先手必败**两种情况。

定理

NIM 博弈先手必胜，当且仅当 $A_1 \operatorname{xor} A_2 \operatorname{xor} \cdots \operatorname{xor} A_n \neq 0$。

证明：

所有物品都被取光是一个必败局面（对手取走最后一件物品，已经获得胜利），此时显然有 $A_1 \operatorname{xor} A_2 \operatorname{xor} \cdots \operatorname{xor} A_n = 0$。

对于任意一个局面，如果 $A_1 \operatorname{xor} A_2 \operatorname{xor} \cdots \operatorname{xor} A_n = x \neq 0$，设 x 的二进制表示下最高位的 1 在第 k 位，那么至少存在一堆物品 A_i，它的第 k 位是 1。显然 $A_i \operatorname{xor} x < A_i$，我们就从 A_i 堆中取走若干物品，使其变为 $A_i \operatorname{xor} x$，就得到了一个各堆物品数异或起来等于 0 的局面。

对于任意一个局面，如果 $A_1 \operatorname{xor} A_2 \operatorname{xor} \cdots \operatorname{xor} A_n = 0$，那么无论如何取物品，得到的局面下各堆物品异或起来都不等于 0。可用反证法证明，假设 A_i 被取成了 A_i'，并且 $A_1 \operatorname{xor} A_2 \operatorname{xor} \cdots \operatorname{xor} A_i' \operatorname{xor} \cdots \operatorname{xor} A_n = 0$。由异或运算的消去律得 $A_i' = A_i$，与“不能不取物品”的规则矛盾。

综上所述，再由数学归纳法可知，$A_1 \text{ xor } A_2 \text{ xor } \cdots \text{ xor } A_n \neq 0$ 为必胜局面，一定存在一种行动让对手面临“各堆物品异或起来等于 0”。$A_1 \text{ xor } A_2 \text{ xor } \cdots \text{ xor } A_n = 0$ 为必败局面，无论如何行动，都会让对手面临一个“各堆物品异或起来不等于 0”的必胜局面。

证毕。

公平组合游戏 ICG

若一个游戏满足：

1. 由两名玩家交替行动；
2. 在游戏进程的任意时刻，可以执行的合法行动与轮到哪名玩家无关；
3. 不能行动的玩家判负。

则称该游戏为一个公平组合游戏。

NIM 博弈属于公平组合游戏，但常见的棋类游戏，比如围棋，就不是公平组合游戏。因为围棋交战双方分别只能落黑子和白子，胜负判定也比较复杂，不满足条件 2 和条件 3。

有向图游戏

给定一个有向无环图，图中有一个唯一的起点，在起点上放有一枚棋子。两名玩家交替地把这枚棋子沿有向边进行移动，每次可以移动一步，无法移动者判负。该游戏被称为**有向图游戏**。

任何一个公平组合游戏都可以转化为有向图游戏。具体方法是：把每个局面看成图中的一个节点，并且从每个局面向沿着合法行动能够到达的下一个局面连有向边。

Mex 运算

设 S 表示一个非负整数集合。定义 $\text{mex}(S)$ 为求出不属于集合 S 的最小非负整数的运算，即：

$$\text{mex}(S) = \min_{x \in \mathbb{N}, x \notin S} \{x\}$$

SG 函数

在有向图游戏中，对于每个节点 x，设从 x 出发共有 k 条有向边，分别到达节点 $y_1, y_2, \cdots, y_k$，定义 $\text{SG}(x)$ 为 x 的后继节点 $y_1, y_2, \cdots, y_k$ 的 SG 函数值构成的集合再执行 mex 运算的结果，即：

$$\text{SG}(x) = \text{mex}(\{\text{SG}(y_1), \text{SG}(y_2), \cdots, \text{SG}(y_k)\})$$

特别地，整个有向图游戏 G 的 SG 函数值被定义为有向图游戏起点 s 的 SG 函数值，即 $\text{SG}(G) = \text{SG}(s)$。

有向图游戏的和

设 $G_1, G_2, \cdots, G_m$ 是 m 个有向图游戏。定义有向图游戏 G，它的行动规则是任选某个有向图游戏 G_i，并在 G_i 上行动一步。G 被称为有向图游戏 $G_1, G_2, \cdots, G_m$ 的和。

有向图游戏的和的 SG 函数值等于它包含的各个子游戏 SG 函数值的异或和，即：

$$\mathrm{SG}(G) = \mathrm{SG}(G_1) \text{ xor } \mathrm{SG}(G_2) \text{ xor } \cdots \text{ xor } \mathrm{SG}(G_m)$$

定理

有向图游戏的某个局面必胜，当且仅当该局面对应节点的 SG 函数值大于 0。

有向图游戏的某个局面必败，当且仅当该局面对应节点的 SG 函数值等于 0。

我们不再详细证明该定理。读者可以这样理解：

在一个没有出边的节点上，棋子不能移动，它的 SG 值为 0，对应必败局面。

若一个节点的某个后继节点 SG 值为 0，在 mex 运算后，该节点的 SG 值大于 0。这等价于，若一个局面的后继局面中存在必败局面，则当前局面为必胜局面。

若一个节点的后继节点 SG 值均不为 0，在 mex 运算后，该节点的 SG 值为 0。这等价于，若一个局面的后继局面全部为必胜局面，则当前局面为必败局面。

对于若干个有向图游戏的和，其证明方法与 NIM 博弈类似。

例题 **Cutting Game** AcWing219

给定一张 $N * M$ 的矩形网格纸，两名玩家轮流行动。在每一次行动中，可以任选一张矩形网格纸，沿着某一行或者某一列的格线，把它剪成两部分。首先剪出 1*1 的玩家获胜。两名玩家都采取最优策略行动，求先手是否必胜。$1 \le N, M \le 200$。

在游戏开始时，只有一张 $N * M$ 的矩形网格纸。但在游戏过程中，可能会有若干张大小不同的矩形网格纸。我们可以把每张矩形网格纸作为一个“游戏”，所有矩形网格纸构成一个“游戏的和”。

这个剪纸游戏与有向图游戏稍有不同。在剪纸游戏中，不能行动的局面，即 1*1 的纸张，是一个必胜局面。而有向图要求在不能行动时判负。因此，我们需要做出一些转化。

首先，若两人都采取最优策略行动，则他们一定不会剪出 $1 * X$ 或 $X * 1$ 的纸张。否则对手就可以剪出 1*1 从而立即获胜。除了 $1 * X$ 和 $X * 1$ 的纸张以外，能够剪出 1*1 的方法必定要经过 2*2，2*3 和 3*2 三种局面之一。而在 2*2，2*3 和 3*2 局面下，先手无论如何行动，都会剪出 $1 * X$ 或 $X * 1$ 的形状。故 2*2，2*3 和 3*2 是必败局面。我们把这三者作为终止局面判负，剪纸游戏就符合有向图游戏的特点了。

对于一张 $N * M$ 的矩形网格纸，我们可以枚举如何行动，把这张纸剪成两部分。这两部分是两个“子剪纸游戏”，二者的 SG 值执行 xor 运算，可得到这个行动之后局



面的 SG 值。对所有合法行动产生的子局面构成的集合做 mex 运算，即可得到这张 $N*M$ 网格纸的 SG 函数值，从而判断它是否先手必胜。即：

$$\mathrm{SG}(N,M)=\mathrm{mex}\big(\{\mathrm{SG}(i,M)\ \mathrm{xor}\ \mathrm{SG}(N-i,M)，\text{其中}\ 1\le i<N\}$$
$$\cup\{\mathrm{SG}(N,i)\ \mathrm{xor}\ \mathrm{SG}(N,M-i)，\text{其中}\ 1\le i<M\}\big)$$

$\mathrm{SG}(i,M)\ \mathrm{xor}\ \mathrm{SG}(N-i,M)$ 为沿着第 i 行下边的格线剪开，两个子游戏 SG 值 xor 的结果。$\mathrm{SG}(N,i)\ \mathrm{xor}\ \mathrm{SG}(N,M-i)$ 为沿着第 i 列右边的格线剪开，两个子游戏 SG 值 xor 的结果。$\cup$ 表示集合的并集运算。最后执行 mex 运算，就得到了 $\mathrm{SG}(N,M)$。

0x3B 总结与练习

本章知识点

质数

质数的判定，试除法

质数的筛选，Eratosthenes 筛法、线性筛法

算术基本定理，试除法分解质因数

约数

算术基本定理的约数个数推论、约数和推论

试除法求约数，倍数法求 $1\sim N$ 每个数的约数集合

最大公约数，最小公倍数，更相减损术，欧几里得算法

欧拉函数的定义与基本性质，积性函数的定义与基本性质

试除法计算欧拉函数，Eratosthenes 筛法与线性筛法快速递推欧拉函数

同余

同余、同余类、剩余系的定义

费马小定理，欧拉定理及其推论，指数循环节

Bézout 定理，扩展欧几里得算法

乘法逆元的计算与应用

线性同余方程、方程组的求解，中国剩余定理

高次同余方程中指数的求解，Baby Step, Giant Step 算法

矩阵乘法

矩阵乘法运算与基本性质

矩阵乘法加速递推，状态矩阵、转移矩阵的构造方法

高斯消元与线性空间

系数矩阵、增广矩阵、主元、自由元、初等行变换等概念

高斯消元的实现，方程组唯一解、多解、无解的判断

线性空间的相关概念，高斯消元求线性空间的基
线性空间的推广，异或空间的性质与应用，去重与不去重异或空间的形态

组合计数
加法原理，乘法原理，排列数，组合数及其性质
组合数的递推求法、逆元求法、分解质因数约分求法
二项式定理，Lucas 定理
多重集的排列数和组合数
Catalan 数列的定义和应用

容斥原理与 Möbius 函数
容斥原理的理解与应用
多重集组合数的完整解析
Möbius 函数的定义、计算与应用

概率与数学期望
随机变量、概率、数学期望等相关定义
数学期望的线性性质，数学期望的递推计算，数学期望与动态规划的结合

0/1 分数规划
0/1 分数规划模型与二分法求解

博弈论之 SG 函数
NIM 游戏等简单博弈模型
SG 函数的计算与应用

练习

1. 完成本章中所有例题的代码实现
2. GCD
题目地址：AcWing220，提示：线性筛法/最大公约数/欧拉函数递推
3. Longge's problem
题目地址：AcWing221，提示：最大公约数/欧拉函数/积性函数
4. 青蛙的约会
题目地址：AcWing222，提示：扩展欧几里得算法
5. Xiao 9*大战朱最学
题目地址：AcWing223，提示：中国剩余定理

6. 计算器

题目地址：AcWing224，提示：线性同余方程/高次同余方程

7. Matrix Power Series

题目地址：AcWing225，提示：矩阵乘法/分治

8. 233 Matrix

题目地址：AcWing226，提示：矩阵乘法/递推

9. Widget Factory

题目地址：AcWing227，提示：高斯消元/同余

10. XOR

题目地址：AcWing228，提示：线性空间/高斯消元/生成树

11. 新 NIM 游戏

题目地址：AcWing229，提示：线性空间/高斯消元/博弈论

12. 排列计数

题目地址：AcWing230，提示：组合/错排问题

13. Sky Code

题目地址：AcWing231，提示：容斥原理

14. 守卫者的挑战

题目地址：AcWing232，提示：概率/动态规划

15. 换教室

题目地址：AcWing233，提示：数学期望/动态规划

16. Dropping tests

题目地址：AcWing234，提示：0/1 分数规划

17. 魔法珠

题目地址：AcWing235，提示：博弈论/SG 函数

18. Georgia and Bob

题目地址：AcWing236，提示：博弈论

0x40 数据结构进阶

在 0x10 章中，我们介绍了基本数据结构及其应用，包括一些线性数据结构，以及 Trie 和二叉堆两种简单的树形数据结构。在本章中，我们将会把重点放在更为复杂的树形数据结构上，主要学习并查集、树状数组、线段树、二叉查找树、平衡树，同时还有分块和点分治这两种与数据结构密切相关的思想。除了介绍这些数据结构的基本理论与实现之外，我们仍然会通过较大篇幅讨论它们的变形与灵活运用。我们还会从“时间轴”和“值域”切入，引入两种离线分治算法以及可持久化数据结构。本章内容具有一定的代码量，读者在阅读和思考的同时，务必多动手练习，掌握每种数据结构的“模板”，并在此基础上熟练使用它们解决各种常见的问题。

0x41 并查集

并查集（Disjoint-Set）是一种可以动态维护若干个不重叠的集合，并支持合并与查询的数据结构。详细地说，并查集包括如下两个基本操作。

1. Get，查询一个元素属于哪一个集合。

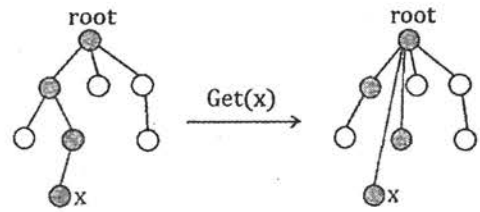
2. Merge，把两个集合合并成一个大集合。

为了具体实现并查集这种数据结构，我们首先需要定义集合的表示方法。在并查集中，我们采用“代表元”法，即为**每个集合选择一个固定的元素，作为整个集合的“代表”**。

其次，我们需要定义归属关系的表示方法。第一种思路是维护一个数组 f，用 $f[x]$ 保存元素 x 所在集合的“代表”。这种方法可以快速查询元素的归属集合，但在合并时需要修改大量元素的 f 值，效率很低。第二种思路是**使用一个树形结构存储每个集合，树上的每个节点都是一个元素，树根是集合的代表元素**。整个并查集实际上是一个森林（若干棵树）。我们仍然可以维护一个数组 fa 来记录这个森林，用 $fa[x]$ 保存 x 的父节点。特别地，令树根的 fa 值为它自己。这样一来，在合并两个集合时，只需连接两个树根（令其中一个树根为另一个树根的子节点，即 $fa[root_1] = root_2$）。不过在查询元素的归属时，需要从该元素开始通过 fa 存储的值不断递归访问父节点，直至到达树根。为了提高查询效率，并查集引入了路径压缩与按秩合并两种思想。

路径压缩与按秩合并

读者可能已经注意到，第一种思路（直接用数组 f 保存代表）的查询效率很高，我们不妨考虑把两种思路进行结合。实际上，我们只关心每个集合对应的“树形结构”的根节点是什么，并不关心这棵树的具体形态——这意味着下图中的两棵树是等价的。





因此，我们可以在每次执行 Get 操作的同时，把访问过的每个节点（也就是所查询元素的全部祖先）都直接指向树根，即把上图中左边那棵树变成右边那棵。这种优化方法被称为**路径压缩**。采用路径压缩优化的并查集，每次 Get 操作的均摊复杂度为 $O(\log N)$。

还有一种优化方法被称为**按秩合并**。所谓“秩”，一般有两种定义。有的资料把并查集中集合的“秩”被定义为树的深度（未路径压缩时）。有的资料把集合的“秩”定义为集合的大小。无论采取哪种定义，我们都可以把集合的“秩”记录在“代表元素”也就是树根上。在合并时都把“秩”较小的树根作为“秩”较大的树根的子节点。

值得一提的是，当“秩”被定义为集合的大小时，“按秩合并”也称为“**启发式合并**”，它是数据结构相关问题中一种重要的思想，应用非常广泛，不只局限于并查集中。启发式合并的原则是：把“小的结构”合并到“大的结构”中，并且只增加“小的结构”的查询代价。这样一来，把所有结构全部合并起来，增加的总代价不会超过 $N \log N$。故单独采用“按秩合并”优化的并查集，每次 Get 操作的均摊复杂度也是 $O(\log N)$。

同时采用“路径压缩”和“按秩合并”优化的并查集，每次 Get 操作的均摊复杂度可以进一步降低到 $O(\alpha(N))$，其中 $\alpha(N)$ 为反阿克曼函数，它是一个比“对数函数” $\log N$ 增长还要慢的函数，对于 $\forall N \leq 2^{2^{2^{19729}}}$，都有 $\alpha(N) < 5$，故 $\alpha(N)$ 可近似为一个常数。著名计算机科学家 R. E. Tarjan 于 1975 年发表的论文[①]中给出了证明。

在实际应用中，我们一般只用路径压缩优化就足够了。接下来，我们对并查集的具体代码实现作一下具体说明。

1. 并查集的存储

使用一个数组 fa 保存父节点（根的父节点设为自己）。

```
int fa[SIZE];
```

2. 并查集的初始化

设有 n 个元素，起初所有元素各自构成一个独立的集合，即有 n 棵 1 个点的树。

```
for (int i = 1; i <= n; i++) fa[i] = i;
```

3. 并查集的 Get 操作

若 x 是树根，则 x 就是集合代表，否则递归访问 $fa[x]$ 直至根节点。

```
int get(int x) {
    if (x == fa[x]) return x;
    return fa[x] = get(fa[x]); // 路径压缩，fa 直接赋值为代表元素
}
```

① Tarjan, Robert Endre (1975), Efficiency of a Good But Not Linear Set Union Algorithm. Journal of the ACM. 22 (2): 215–225 “http://dl.acm.org/citation.cfm?id=321884”.

4. 并查集的 Merge 操作

合并元素 x 和元素 y 所在的集合，等价于让 x 的树根作为 y 的树根的子节点。

```
void merge(int x, int y) {
    fa[get(x)] = get(y);
}
```

例题 程序自动分析 AcWing237

在实现程序自动分析的过程中，常常需要判定一些约束条件是否能被同时满足。

考虑一个约束满足问题的简化版本：假设 $x_1, x_2, x_3, \cdots$ 代表程序中出现的变量，给定 n 个形如 $x_i = x_j$ 或 $x_i \neq x_j$ 的变量相等/不等的约束条件，请判定是否可以分别为每一个变量赋予恰当的值，使得上述所有约束条件同时被满足。例如一个问题中的约束条件为 $x_1 = x_2, x_2 = x_3, x_3 = x_4, x_1 \neq x_4$，这些约束条件显然是不可能同时被满足的，因此这个问题应判定为不可被满足。

现在给出一些约束满足问题，请分别对它们进行判定。

$1 \leq n \leq 10^5$，$1 \leq x \leq 10^9$。

先满足所有“相等”类型的约束条件。容易发现，如果把每个变量看作无向图中的一个节点，每个“相等”的约束条件看作无向图中的一条边，那么两个变量相等，当且仅当它们连通。于是，我们可以把变量分成若干个集合，每个集合都对应无向图中的一个连通块。

有两种方法求出这些集合。第一种是建出上面提到的无向图，执行深度优先遍历，划分出无向图中的每个连通块。第二种是使用并查集动态维护。起初，所有变量各自构成一个集合；对于每条“相等”的约束条件，合并它约束的两个变量所在的集合即可。最后，扫描所有“不等”类型的约束条件。若存在一条“不等”的约束条件，它约束的两个变量处于同一个集合内，则不可能被满足。若不存在这样的“不等”约束，则全部条件可以被满足。

值得注意的是，本题中变量 x 的范围很大。我们可以首先使用 0x05 节中提到的“离散化”方法把变量 x 的范围映射到 $1 \sim 2n$ 之内，再用上述算法解决。

从这道题目我们看到，并查集能**在一张无向图中维护节点之间的连通性**，这是它的基本用途之一。实际上，**并查集擅长动态维护许多具有传递性的关系**。所谓传递性，顾名思义，就是指如果 A 与 B 有某种关系，B 与 C 有某种关系，那么 A 与 C 也有某种确定的关系。本题中的“等于”就是一种传递关系，而“不等于”则显然不具有传递性。

例题 Supermarket AcWing145

给定 N 个商品，每个商品有利润 p_i 和过期时间 d_i，每天只能卖一个商品，过期



商品不能再卖，求如何安排每天卖的商品，可以使收益最大。$1 \le N, p_i, d_i \le 10000$。

在第 0x17 节中，我们给出了一个利用二叉堆实现的算法。在本节中，请读者尝试换一种思路，使用并查集配合贪心解决本题。

在第 0x17 节的二叉堆做法中，我们把商品按照过期时间从前到后排序，然后依次尝试用每个商品替换掉堆中利润较低的商品。

另一种显而易见的贪心策略则是，优先考虑卖出利润大的商品，并且对每个商品，在它过期之前尽量晚卖出——占用较晚的时间，显然对其他商品具有“决策包容性”。第 0x07 节曾多次使用这一性质证明过贪心算法。

于是我们可以把商品按照利润从大到小排序，并建立一个关于“天数”的并查集，起初每一天各自构成一个集合。对于每个商品，若它在 d 天之后过期，就在并查集中查询 d 的树根（记为 r）。若 r 大于 0，则把该商品安排在第 r 天卖出，合并 r 与 $r-1$（令 r 为 $r-1$ 的子节点），答案累加该商品的利润。

这个并查集实际上**维护了一个数组中“位置”的占用情况**。每个“位置”所在集合的代表就是从它开始往前数第一个空闲的位置（包括它本身）。当一个“位置”被占用时（某一天安排了商品），就把该“位置”在并查集中指向它前一个“位置”。利用并查集的路径压缩，就可以快速找到最晚能卖出的时间（从过期时间**往前数第一个空闲**的天数）。

“扩展域”与“边带权”的并查集

并查集实际上是由若干棵树构成的森林，我们可以在树中的每条边上记录一个权值，即维护一个数组 d，用 $d[x]$ 保存节点 x 到父节点 $fa[x]$ 之间的边权。在每次路径压缩后，每个访问过的节点都会直接指向树根，如果我们同时更新这些节点的 d 值，就可以利用路径压缩过程来统计每个节点到树根之间的路径上的一些信息。这就是所谓“边带权”的并查集。

例题 银河英雄传说 AcWing238

有一个划分成 N 列的星际战场，各列依次编号为 $1,2,\cdots,N$。有 N 艘战舰，也依次编号为 $1,2,\cdots,N$，其中第 i 号战舰处于第 i 列。

有 M 条指令，每条指令格式为以下两种之一。

1. M i j，表示让第 i 号战舰所在列的全部战舰保持原有顺序，接在第 j 号战舰所在列的尾部。

2. C i j，表示询问第 i 号战舰与第 j 号战舰当前是否处于同一列中，如果在同一列中，它们之间间隔了多少艘战舰。

$N \le 30000$，$M \le 5 * 10^5$。

一条“链”也是一棵树，只不过是树的特殊形态。因此可以把每一列战舰看作一个集合，用并查集维护。最初，N 个战舰构成 N 个独立的集合。

在没有路径压缩的情况下，$fa[x]$ 就表示排在第 x 号战舰前面的那个战舰的编号，一个集合的代表就是位于最前边的战舰。另外，让树上每条边带权值 1，这样树上两点之间的距离减 1 就是二者之间间隔的战舰数量。

在考虑路径压缩的情况下，我们额外建立一个数组 d，$d[x]$ 记录战舰 x 与 $fa[x]$ 之间的边的权值。在路径压缩把 x 直接指向树根的同时，我们把 $d[x]$ 更新为从 x 到树根的路径上的所有边权之和。下面的代码对 Get 函数稍加修改，即可实现对 d 数组的维护。

```
int get(int x) {
    if (x == fa[x]) return x;
    int root = get(fa[x]);  // 递归计算集合代表
    d[x] += d[fa[x]];       // 维护 d 数组——对边权求和
    return fa[x] = root;    // 路径压缩
}
```

当接收到一个 C x y 指令时，分别执行 get(x) 和 get(y) 完成查询和路径压缩。若二者的返回值相同，则说明 x 和 y 处于同一列中。因为 x 和 y 此时都已经指向树根，所以 $d[x]$ 保存了位于 x 之前的战舰数量，$d[y]$ 保存了位于 y 之前的战舰数量。二者之差的绝对值再减 1，就是 x 和 y 之间间隔的战舰数量。

当接收到一个 M x y 指令时，把 x 的树根作为 y 的树根的子节点，连接的新边的权值应该设为合并之前集合 y 的大小（根据题意，集合 y 中的全部战舰都排在集合 x 之前）。因此，我们还需要一个 *size* 数组在每个树根上记录集合大小。下面这段对 Merge 函数稍加修改的代码实现了这条指令：

```
void merge(int x, int y) {
    x = get(x), y = get(y);
    fa[x] = y, d[x] = size[y];
    size[y] += size[x];
}
```

刚才我们在“程序自动分析”一题中提到，并查集擅长维护具有传递性的关系及其连通性。在某些问题中，“传递关系”不止一种，并且这些“传递关系”能够互相导出。此时可以使用“扩展域”或者“边带权”的并查集来解决。我们通过几道例题来详细说明。另外，我们会在 0x67 节探讨 2-SAT 模型，学习之后读者就会更加深刻地理解并查集能够处理的“关系”本质。

例题 Parity game AcWing239

小 A 和小 B 在玩一个游戏。首先，小 A 写了一个由 0 和 1 组成的序列 S，长度为 N。

然后，小 B 向小 A 提出了 M 个问题。在每个问题中，小 B 指定两个数 l 和 r，小 A 回答 $S[l\sim r]$ 中有奇数个 1 还是偶数个 1。

机智的小 B 发现小 A 有可能在撒谎。例如，小 A 曾经回答过 $S[1\sim3]$ 中有奇数个 1，$S[4\sim6]$ 中有偶数个 1，现在又回答 $S[1\sim6]$ 中有偶数个 1，显然小 A 是自相矛盾的。请你帮助小 B 检查这 M 个答案，并指出在至少多少个回答之后可以确定小 A 一定在撒谎。即求出一个最小的 k，使得存在一个 01 序列满足第 $1\sim k$ 个回答，但不存在满足第 $1\sim k+1$ 个回答的 01 序列。$N \le 10^9$，$M <= 10000$。

如果我们用 sum 数组表示序列 S 的前缀和，那么在每个回答中：

1. $S[l\sim r]$ 有偶数个 1，等价于 $sum[l-1]$ 与 $sum[r]$ 奇偶性相同；

2. $S[l\sim r]$ 有奇数个 1，等价于 $sum[l-1]$ 与 $sum[r]$ 奇偶性不同。

注意，这里没有求出 sum 数组，我们只是把 sum 看作变量。读者可以发现，此时本题与本节中“程序自动分析”一题非常类似——都是给定若干个变量和关系，判定这些关系可满足性的问题。不同点是本题的传递关系不止一种。

1. 若 x_1 与 x_2 奇偶性相同，x_2 与 x_3 奇偶性也相同，那么 x_1 与 x_3 奇偶性相同。这种情况与“程序自动分析”中的等于关系一样。

2. 若 x_1 与 x_2 奇偶性相同，x_2 与 x_3 奇偶性不同，那么 x_1 与 x_3 奇偶性不同。

3. 若 x_1 与 x_2 奇偶性不同，x_2 与 x_3 奇偶性也不同，那么 x_1 与 x_3 奇偶性相同。

另外，序列长度 N 很大，但问题数 M 较少，可以先用 0x05 节中学习的离散化方法把每个问题的两个整数 $l-1$ 和 r 缩小到等价的 $1\sim2M$ 以内的范围。

```
struct { int l, r, ans; } query[10010];
int a[20010], fa[20010], d[20010], n, m, t;
void read_discrete() { // 读入、离散化
    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        char str[5];
        scanf("%d%d%s", &query[i].l, &query[i].r, str);
        query[i].ans = (str[0] == 'o' ? 1 : 0);
        a[++t] = query[i].l - 1;
        a[++t] = query[i].r;
    }
```

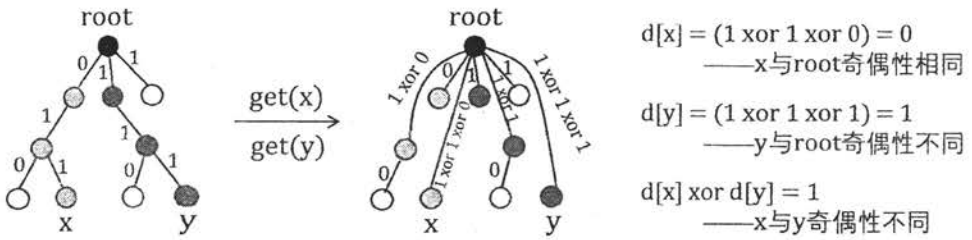
```
    sort(a + 1, a + t + 1);
    n = unique(a + 1, a + t + 1) - a - 1;
}
```

为了处理本题的多种传递关系，第一种解决方案是使用“边带权”的并查集。

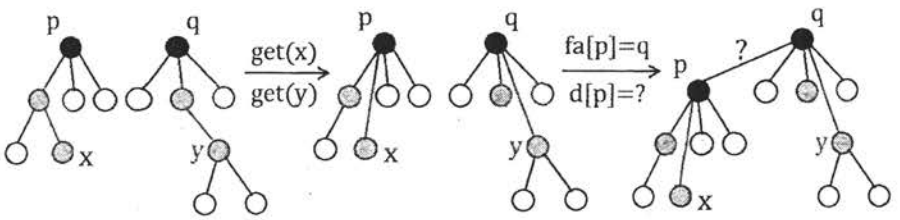
边权 $d[x]$ 为 0，表示 x 与 $fa[x]$ 奇偶性相同；为 1，表示 x 与 $fa[x]$ 奇偶性不同。在路径压缩时，对 x 到树根路径上的所有边权做异或(xor)运算，即可得到 x 与树根的奇偶性关系。

对于每个问题，设在离散化后 $l-1$ 和 r 的值分别是 x 和 y，设 ans 表示该问题的回答（0 代表偶数个，1 代表奇数个）。

先检查 x 和 y 是否在同一个集合内（奇偶关系是否已知）。get(x)、get(y) 都执行完成后，$d[x]$ xor $d[y]$ 即为 x 和 y 的奇偶性关系（如下图）。若 $d[x]$ xor $d[y] \neq ans$（该关系与回答矛盾），则在该问题之后即可确定小 A 撒谎。



若 x 和 y 不在同一个集合内，则合并两个集合。此时应该先通过 get 操作得到两个集合的树根（设为 p 和 q），令 p 为 q 的子节点。如下图所示。已知 $d[x]$ 与 $d[y]$ 分别表示路径 x~p 与 y~q 之间所有边权的“xor 和”，p~q 之间的边权 $d[p]$ 是待求的值。显然，路径 x~y 由路径 x~p，p~q 与 q~y 组成，因此 x 与 y 的奇偶性关系 $ans = d[x]$ xor $d[y]$ xor $d[p]$。进而推出新连接的边权 $d[p] = d[x]$ xor $d[y]$ xor ans。



“边带权”并查集解法的主要代码片段：

```
int get(int x) {
    if (x == fa[x]) return x;
    int root = get(fa[x]); d[x] ^= d[fa[x]];
```

```
    return fa[x] = root;
}
int main() {
    read_discrete();
    for (int i = 1; i <= n; i++) fa[i] = i;
    for (int i = 1; i <= m; i++) {
        // 求出 l-1 和 r 离散化之后的值
        int x = lower_bound(a + 1, a + n + 1, query[i].l - 1) - a;
        int y = lower_bound(a + 1, a + n + 1, query[i].r) - a;
        // 执行 get 函数，得到树根，并进行路径压缩
        int p = get(x), q = get(y);
        if (p == q) { // 已经在同一集合内
            if ((d[x] ^ d[y]) != query[i].ans) // 矛盾，输出
                { cout << i - 1 << endl; return 0; }
        }
        else // 不在同一集合，合并
            fa[p] = q, d[p] = d[x] ^ d[y] ^ query[i].ans;
    }
    cout << m << endl; // 没有矛盾
}
```

第二种解决方案是使用“扩展域”的并查集。

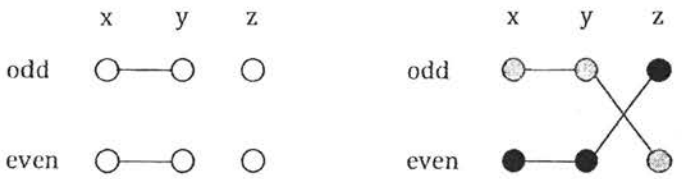
把每个变量 x 拆成两个节点 x_{odd} 和 x_{even}，其中 x_{odd} 表示 $sum[x]$ 是奇数，x_{even} 表示 $sum[x]$ 是偶数。我们也经常把这两个节点称为 x 的“奇数域”和“偶数域”。

对于每个问题，设在离散化后 $l-1$ 和 r 的值分别是 x 和 y，设 ans 表示该问题的回答（0 代表偶数个，1 代表奇数个）。

1. 若 $ans=0$，则合并 x_{odd} 与 y_{odd}，x_{even} 与 y_{even}。这表示“x 为奇数”与“y 为奇数”可以互相推出，“x 为偶数”与“y 为偶数”也可以互相推出，它们是等价的信息。

2. 若 $ans=1$，则合并 x_{odd} 与 y_{even}，x_{even} 与 y_{odd}。这表示“x 为奇数”与“y 为偶数”可以互相推出，“x 为偶数”与“y 为奇数”也可以互相推出，它们是等价的信息。

上述合并同时还维护了关系的传递性。试想，在处理完 $(x,y,0)$ 和 $(y,z,1)$ 两个回答之后，x 和 z 之间的关系也就已知了，如下页图所示。这种做法就相当于在无向图上维护节点之间的连通情况，只是扩展了多个域来应对多种传递关系。



在处理每个问题之前，我们当然也要先检查是否存在矛盾。若两个变量 x 和 y 对应的 x_{odd} 和 y_{odd} 节点在同一集合内，则已知二者奇偶性相同。若两个变量 x 和 y 对应的 x_{odd} 和 y_{even} 节点在同一集合内，则已知二者奇偶性不同。

“扩展域”并查集解法的主要代码片段：

```
int get(int x) {
    if (x == fa[x]) return x;
    return fa[x] = get(fa[x]);
}
int main() {
    read_discrete();
    for (int i = 1; i <= 2 * n; i++) fa[i] = i;
    for (int i = 1; i <= m; i++) {
        // 求出 l-1 和 r 离散化之后的值
        int x = lower_bound(a + 1, a + n + 1, query[i].l - 1) - a;
        int y = lower_bound(a + 1, a + n + 1, query[i].r) - a;
        int x_odd = x, x_even = x + n;
        int y_odd = y, y_even = y + n;
        if (query[i].ans == 0) { // 回答奇偶性相同
            if (get(x_odd) == get(y_even)) { // 与已知情况矛盾
                cout << i - 1 << endl;
                return 0;
            }
            fa[get(x_odd)] = get(y_odd); // 合并
            fa[get(x_even)] = get(y_even);
        }
        else { // 回答奇偶性不同
            if (get(x_odd) == get(y_odd)) { // 与已知情况矛盾
                cout << i - 1 << endl;
                return 0;
            }
            fa[get(x_odd)] = get(y_even); // 合并
            fa[get(x_even)] = get(y_odd);
        }
```

```
    }
    cout << m << endl; // 没有矛盾
}
```

例题 食物链 AcWing240

动物王国中有三类动物A、B、C，这三类动物的食物链构成了有趣的环形。其中A吃B，B吃C，C吃A。

现有N个动物，以1~N编号。每个动物都是A、B、C中的一种，但是我们并不知道它到底是哪一种。有人用两种说法对这N个动物所构成的食物链关系进行了描述。

1. 第一种说法是“1 X Y”，表示X和Y是同类。

2. 第二种说法是“2 X Y”，表示X吃Y。

此人对N个动物，用上述两种说法，一句接一句地说出K句话，这K句话有的是真的，有的是假的。当一句话满足下列三条之一时，这句话就是假话，否则就是真话。

1. 当前的话与前面的某些真的话冲突，就是假话。

2. 当前的话中X或Y比N大，就是假话。

3. 当前的话表示X吃X，就是假话。

你的任务是根据给定的 $N(1 \le N \le 50000)$ 和 K 句话 $(0 \le K \le 10^5)$，输出假话的总数。

本题仍然可以用“边带权”或者“扩展域”的并查集解决。我们以“扩展域”并查集为例进行讲解。

把每个动物 x 拆成三个节点，同类域 x_{self}、捕食域 x_{eat} 和天敌域 x_{enemy}。

若一句话说“x 与 y 是同类”，则说明“x 的同类”与“y 的同类”一样，“x 捕食的物种”与“y 捕食的物种”一样，“x 的天敌”与“y 的天敌”一样。此时，我们合并 x_{self} 与 y_{self}，x_{eat} 与 y_{eat}，x_{enemy} 与 y_{enemy}。

若一句话说“x 吃 y”，说明“x 捕食的物种”就是“y 的同类”，“x 的同类”都是“y 的天敌”。又因为题目中告诉我们食物链是长度为3的环形，所以“x 的天敌”就是“y 捕食的物种”（x 吃 y，y 吃 z，z 就吃 x）。此时，应合并 x_{eat} 与 y_{self}，x_{self} 与 y_{enemy}，x_{enemy} 与 y_{eat}。

在处理每句话之前，都要检查这句话的真假。

有两种信息与“x 与 y 是同类”矛盾：

1. x_{eat} 与 y_{self} 在同一集合，说明 x 吃 y；

2. x_{self} 与 y_{eat} 在同一集合，说明 y 吃 x。

有两种信息与“x 吃 y”矛盾：

1. x_{self} 与 y_{self} 在同一集合，说明 x 与 y 是同类；

2. x_{self} 与 y_{eat} 在同一集合，说明 y 吃 x。

0x42 树状数组

在 0x00 章的基本算法中，我们详细探讨了位运算、二分、倍增等贯穿于各种数据结构设计始末的思想。无论是二进制、折半还是翻倍，都与“二”这个数字密切相关。请读者回想 0x06 节的 ST 表，在对一个较大的连续线性范围进行统计时，我们把它按照 2 的整数次幂分成若干个小范围进行预处理和计算。再回想 0x01 节的快速幂算法，根据任意正整数关于 2 的不重复次幂的唯一分解性质，若一个正整数 x 的二进制表示为 $a_{k-1}a_{k-2}\cdots a_2a_1a_0$，其中等于 1 的位是 $\{a_{i_1}, a_{i_2}, \cdots, a_{i_m}\}$，则正整数 x 可以被“二进制分解”成：

$$x = 2^{i_1} + 2^{i_2} + \cdots + 2^{i_m}$$

不妨设 $i_1 > i_2 > \cdots > i_m$，进一步地，区间 $[1,x]$ 可以分成 $O(\log x)$ 个小区间：

1. 长度为 2^{i_1} 的小区间 $[1,\ 2^{i_1}]$
2. 长度为 2^{i_2} 的小区间 $[2^{i_1}+1,\ 2^{i_1}+2^{i_2}]$
3. 长度为 2^{i_3} 的小区间 $[2^{i_1}+2^{i_2}+1,\ 2^{i_1}+2^{i_2}+2^{i_3}]$

……

m. 长度为 2^{i_m} 的小区间 $[2^{i_1}+2^{i_2}+\cdots+2^{i_{m-1}}+1,\ 2^{i_1}+2^{i_2}+\cdots+2^{i_m}]$

这些小区间的共同特点是：若区间结尾为 R，则区间长度就等于 R 的“二进制分解”下最小的 2 的次幂，即 lowbit(R)。例如 $x = 7 = 2^2 + 2^1 + 2^0$，区间 [1,7] 可以分成 [1,4]，[5,6] 和 [7,7] 三个小区间，长度分别是 lowbit(4) = 4，lowbit(6) = 2 和 lowbit(7) = 1。

我们在 0x01 节中探讨过 lowbit 运算，并介绍了如何利用 lowbit 运算找出整数在二进制表示下所有等于 1 的位。类似地，给定一个整数 x，下面这段代码可以计算出区间 $[1,x]$ 分成的 $O(\log x)$ 个小区间。

```
while (x > 0) {
    printf("[%d, %d]\n", x - (x & -x) + 1, x);
    x -= x & -x;
}
```

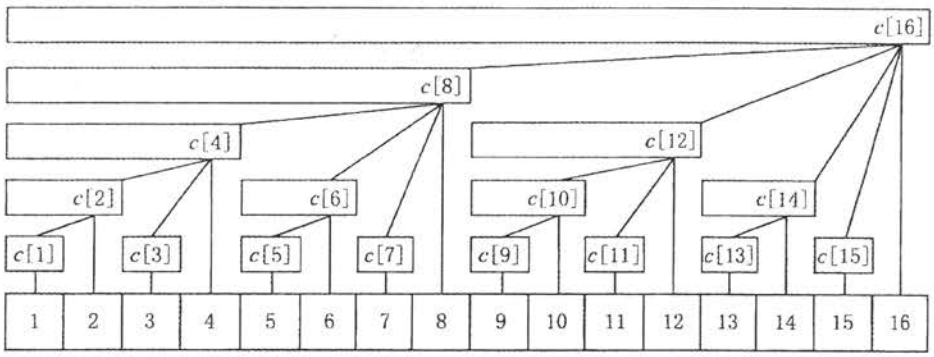
树状数组 (Binary Indexed Trees) 就是一种基于上述思想的数据结构，其基本用途是**维护序列的前缀和**。对于给定的序列 a，我们建立一个数组 c，其中 $c[x]$ 保存序

列 a 的区间 $[x-\text{lowbit}(x)+1,\ x]$ 中所有数的和，即 $\sum_{i=x-\text{lowbit}(x)+1}^{x} a[i]$。

事实上，数组 c 可以看作一个如下图所示的树形结构，图中最下边一行是 N 个叶节点（$N=16$），代表数值 $a[1\sim N]$。该结构满足以下性质：

1. 每个内部节点 $c[x]$ 保存以它为根的子树中所有叶节点的和；
2. 每个内部节点 $c[x]$ 的子节点个数等于 $\text{lowbit}(x)$ 的位数；
3. 除树根外，每个内部节点 $c[x]$ 的父节点是 $c[x+\text{lowbit}(x)]$；
4. 树的深度为 $O(\log N)$。

如果 N 不是 2 的整次幂，那么树状数组就是一个具有同样性质的森林结构。



树状数组支持的基本操作有两个。第一个操作是**查询前缀和**，即序列 a 第 $1\sim x$ 个数的和。按照我们刚才提出的方法，应该求出 x 的二进制表示中每个等于 1 的位，把 $[1,x]$ 分成 $O(\log N)$ 个小区间，而每个小区间的区间和都已经保存在数组 c 中。所以，把上面的代码稍加改写即可在 $O(\log N)$ 的时间内查询前缀和。

```
int ask(int x) {
    int ans = 0;
    for (; x; x -= x & -x) ans += c[x];
    return ans;
}
```

当然，若要查询序列 a 的区间 $[l,r]$ 中所有数的和，只需计算 $\text{ask}(r)-\text{ask}(l-1)$ 即可。

树状数组支持的第二个基本操作是**单点增加**，意思是给序列中的一个数 $a[x]$ 加上 y，同时正确维护序列的前缀和。根据上面给出的树形结构和它的性质，只有节点 $c[x]$ 及其所有祖先节点保存的“区间和”包含 $a[x]$，而任意一个节点的祖先至多只有 $\log N$ 个，我们逐一对它们的 c 值进行更新即可。下面的代码在 $O(\log N)$ 时间内执行单点增加操作。

```
void add(int x, int y) {
    for (; x <= N; x += x & -x) c[x] += y;
}
```

在执行所有操作之前，我们需要对树状数组进行**初始化**——针对原始序列 a 构造一个树状数组。

为了简便起见，比较一般的初始化方法是：直接建立一个全为 0 的数组 c，然后对每个位置 x 执行 $\text{add}(x, a[x])$，就完成了对原始序列 a 构造树状数组的过程，时间复杂度为 $O(N \log N)$。通常采用这种初始化方法已经足够。

更高效的初始化方法是：从小到大依次考虑每个节点 x，借助 lowbit 运算扫描它的子节点并求和。若采用这种方法，上面树形结构中的每条边只会被遍历一次，时间复杂度为 $O\left(\sum_{k=1}^{\log N} k * N/2^k\right) = O(N)$。

树状数组与逆序对

任意给定一个集合 a，如果用 $t[val]$ 保存数值 val 在集合 a 中出现的次数，那么数组 t 在 $[l,r]$ 上的区间和（即 $\sum_{i=l}^{r} t[i]$）就表示集合 a 中范围在 $[l,r]$ 内的数有多少个。

我们可以在集合 a 的**数值范围上**建立一个树状数组，来维护 t 的前缀和。这样即使在集合 a 中插入或删除一个数，也可以高效地进行统计。

我们在 0x05 节中提到了逆序对问题以及使用归并排序的解法。对于一个序列 a，若 $i < j$ 且 $a[i] > a[j]$，则称 $a[i]$ 与 $a[j]$ 构成逆序对。按照上述思路，利用树状数组也可以求出一个序列的逆序对个数。

1. 在序列 a 的数值范围上建立树状数组，初始化为全零。
2. 倒序扫描给定的序列 a，对于每个数 $a[i]$：
   (1) 在树状数组中查询前缀和 $[1, a[i]-1]$，累加到答案 ans 中。
   (2) 执行“单点增加”操作，即把位置 $a[i]$ 上的数加 1（相当于 $t[a[i]]$++），同时正确维护 t 的前缀和。这表示数值 $a[i]$ 又出现了 1 次。
3. ans 即为所求。

```
for (int i = n; i; i--) {
    ans += ask(a[i] - 1);
    add(a[i], 1);
}
```

在这个算法中，因为倒序扫描，“已经出现过的数”就是在 $a[i]$ 后边的数，所以

我们通过树状数组查询的内容就是“每个 $a[i]$ 后边有多少个比它小”。每次查询的结果之和当然就是逆序对个数。时间复杂度为 $O\big((N+M)\log M\big)$，M 为数值范围的大小。

当数值范围较大时，当然可以先进行离散化，再用树状数组进行计算。不过因为离散化本身就要通过排序来实现，所以在这种情况下就不如直接用归并排序来计算逆序对数了。

例题 楼兰图腾 AcWing241

平面上有 $N(N \le 2*10^5)$ 个点，每个点的横、纵坐标的范围都是 $1\sim N$，任意两个点的横、纵坐标都不相同。

若三个点 $(x_1,y_1),(x_2,y_2),(x_3,y_3)$ 满足 $x_1<x_2<x_3$，$y_1>y_2$ 并且 $y_3>y_2$，则称这三个点构成“v”字图腾。

若三个点 $(x_1,y_1),(x_2,y_2),(x_3,y_3)$ 满足 $x_1<x_2<x_3$，$y_1<y_2$ 并且 $y_3<y_2$，则称这三个点构成“^”字图腾。

求平面上“v”和“^”字图腾的个数。

题目描述的第一句话实际上告诉我们，如果把这 N 个点按照横坐标排序，那么它们的纵坐标是 $1\sim N$ 的一个排列。我们把这个排列记为 a。

在树状数组求逆序对的算法中，我们已经知道如何在一个序列中计算每个数后边有多少个数比它小。类似地，我们可以：

1. 倒序扫描序列 a，利用树状数组求出每个 $a[i]$ 后边有几个数比它大，记为 $right[i]$；

2. 正序扫描序列 a，利用树状数组求出每个 $a[i]$ 前边有几个数比它大，记为 $left[i]$。

依次枚举每个点作为中间点，以该点为中心的“v”字图腾个数显然是 $left[i]*right[i]$。所以“v”字图腾的总数就是 $\sum_{i=1}^{N} left[i]*right[i]$。

按照同样的方法，我们可以统计出“^”字图腾的个数。

树状数组的扩展应用

例题 A Tiny Problem with Integers AcWing242

给定长度为 $N(N \le 10^5)$ 的数列 A，然后输入 $Q(Q \le 10^5)$ 行操作指令。

第一类指令形如“C l r d”，表示把数列中第 $l\sim r$ 个数都加 d。

第二类指令形如“Q x”，表示询问数列中第 x 个数的值。

本题的指令有“区间增加”和“单点查询”，而树状数组仅支持“单点增加”，需要做出一些转化来解决问题。

新建一个数组 b，起初为全零。对于每条指令“C l r d”，我们把它转化成以下两条指令：

1. 把 $b[l]$ 加上 d。

2. 再把 $b[r+1]$ 减去 d。

执行上面两条指令之后，我们来考虑一下 b 数组的前缀和（$b[1\sim x]$ 的和）的情况。

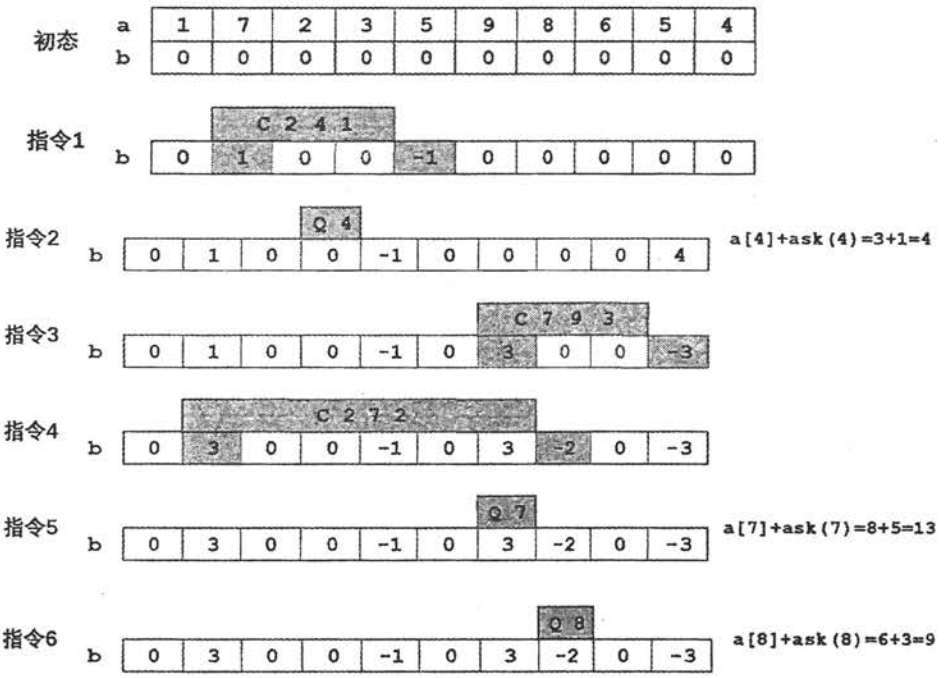
1. 对于 $1 \le x < l$，前缀和不变。

2. 对于 $l \le x \le r$，前缀和增加了 d。

3. 对于 $r < x \le N$，前缀和不变（l 处加 d，$r+1$ 处减 d，抵消）。

我们发现，b 数组的前缀和 $b[1\sim x]$ 就反映了指令“C l r d”对 $a[x]$ 产生的影响。

于是，我们可以用树状数组来维护数组 b 的前缀和（对 b 只有单点增加操作）。因为各次操作之间具有可累加性，所以在树状数组上查询前缀和 $b[1\sim x]$，就得出了到目前为止所有“C”指令在 $a[x]$ 上增加的数值总和。再加上 $a[x]$ 的初始值，就得到了“Q x”的答案。



该做法把“维护数列的具体值”转化为“维护指令的累积影响”。每次修改的“影响”在 l 处产生，在 $r+1$ 处消除。“影响”的前缀和 $b[1\sim x]$ 就代表了数值 $a[x]$ 的变化情况，如上图所示。该做法巧妙地把“区间增加”+“单点查询”变为树状数组擅长的“单点增加”+“区间查询”进行处理。

例题 **A Simple Problem with Integers** AcWing243

给定长度为 $N(N \le 10^5)$ 的数列 A，然后输入 $Q(Q \le 10^5)$ 行操作指令。

第一类指令形如“C l r d”，表示把数列中第 l~r 个数都加 d。

第二类指令形如“Q l r”，表示询问数列中第 l~r 个数的和。

在上一题中，我们用树状数组维护了一个数组 b，对于每条指令“C l r d”，把 $b[l]$ 加上 d，再把 $b[r+1]$ 减去 d。

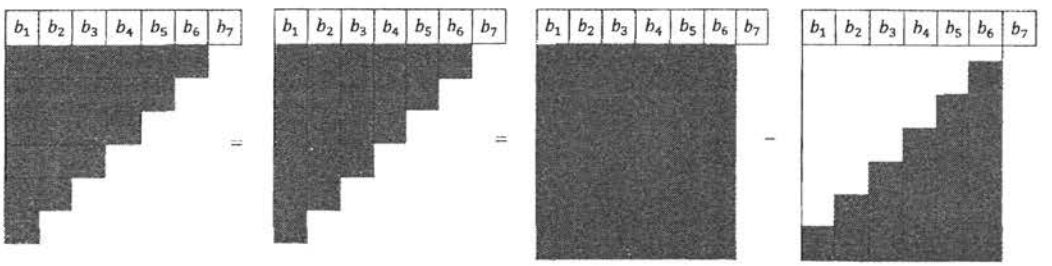
我们已经讨论过，b 数组的前缀和 $\sum_{i=1}^{x} b[i]$ 就是经过这些指令后 $a[x]$ 增加的值，那么序列 a 的前缀和 $a[1\sim x]$ 整体增加的值就是：

$$\sum_{i=1}^{x}\sum_{j=1}^{i} b[j]$$

上式可以改写为：

$$\sum_{i=1}^{x}\sum_{j=1}^{i} b[j] = \sum_{i=1}^{x}(x-i+1)*b[i] = (x+1)\sum_{i=1}^{x} b[i] - \sum_{i=1}^{x} i*b[i]$$

这个推导也可以用以下图形来直观描绘：



在本题中，我们增加一个树状数组，用于维护 $i*b[i]$ 的前缀和 $\sum_{i=1}^{x} i*b[i]$，上式就可以直接查询、计算了。

具体来说，我们建立两个树状数组 c_0 和 c_1，起初全部赋值为零。对于每条指令“C l r d”，执行 4 个操作。

1. 在树状数组 c_0 中，把位置 l 上的数加 d。
2. 在树状数组 c_0 中，把位置 $r+1$ 上的数减 d。
3. 在树状数组 c_1 中，把位置 l 上的数加 $l*d$。
4. 在树状数组 c_1 中，把位置 $r+1$ 上的数减 $(r+1)*d$。

另外，我们建立数组 sum 存储序列 a 的原始前缀和。对于每条指令“Q l r”，当然还是拆成 $1\sim r$ 和 $1\sim l-1$ 两个部分，二者相减。写成式子就是：

$$\big(sum[r] + (r+1)*ask(c_0,r) - ask(c_1,r)\big)$$

$$-\left(sum[l-1]+l*ask(c_0,l-1)-ask(c_1,l-1)\right)$$

```
const int SIZE=100010;
int a[SIZE],n,m;
long long c[2][SIZE],sum[SIZE];
long long ask(int k,int x) {
    long long ans=0;
    for(;x;x-=x&-x) ans+=c[k][x];
    return ans;
}

void add(int k,int x,int y) {
    for(;x<=n;x+=x&-x) c[k][x]+=y;
}

int main() {
    cin>>n>>m;
    for(int i=1;i<=n;i++) {
        scanf("%d",&a[i]);
        sum[i]=sum[i-1]+a[i];
    }
    while(m--) {
        char op[2]; int l,r,d;
        scanf("%s%d%d",op,&l,&r);
        if(op[0]=='C') {
            scanf("%d",&d);
            add(0,l,d);
            add(0,r+1,-d);
            add(1,l,l*d);
            add(1,r+1,-(r+1)*d);
        }
        else {
            long long ans=sum[r]+(r+1)*ask(0,r)-ask(1,r);
            ans-=sum[l-1]+l*ask(0,l-1)-ask(1,l-1);
            printf("%lld\n",ans);
        }
    }
}
```



值得指出的是，为什么我们把 $\sum_{i=1}^{x}(x-i+1)*b[i]$ 变成 $(x+1)\sum_{i=1}^{x}b[i]-\sum_{i=1}^{x}i*b[i]$ 进行统计呢？仔细观察该式的定义，这里的 x 是关于“前缀和 $a[1\sim x]$”这个询问的变量，而 i 是在每次修改时影响的对象。

对于前者来说，求和式中的每一项同时包含 x 和 i，在修改时无法确定 $(x-i+1)$ 的值，只能维护 $b[i]$ 的前缀和。在询问时需要面临一个“系数为等差数列”的求和式，计算起来非常困难。

对于后者来说，求和式中的每一项只与 i 有关。它通过一次容斥，把 $(x+1)$ 提取为常量，使得 $b[i]$ 的前缀和与 $i*b[i]$ 的前缀和可以分别用树状数组进行维护。这种**分离包含有多个变量的项，使公式中不同变量之间互相独立**的思想非常重要，我们在下一章讨论动态规划的优化策略时会多次用到。

例题 **Lost Cows** AcWing244

有 n 头奶牛 $(n\le 10^5)$，已知它们的身高为 $1\sim n$ 且各不相同，但不知道每头奶牛的具体身高。

现在这 n 头奶牛站成一列，已知第 i 头奶牛前面有 A_i 头牛比它低，求每头奶牛的身高。

如果最后一头奶牛前面有 A_n 头牛比它低，那么显然它的身高 $H_n=A_n+1$。

如果倒数第二头奶牛前有 A_{n-1} 头牛比它低，那么：

1. 若 $A_{n-1}<A_n$，则它的身高 $H_{n-1}=A_{n-1}+1$；

2. 若 $A_{n-1}\ge A_n$，则它的身高 $H_{n-1}=A_{n-1}+2$。

依此类推，如果第 k 头牛前面有 A_k 头比它低，那么它的身高 H_k 是数值 $1\sim n$ 中第 A_k+1 小的没有在 $\{H_{k+1},H_{k+2},\cdots,H_n\}$ 中出现过的数。

具体来说，我们建立一个长度为 n 的 01 序列 b，起初全部为 1。然后，从 n 到 1 倒序扫描每个 A_i，对每个 A_i 执行以下两个操作。

1. 查询序列 b 中第 A_i+1 个 1 在什么位置，这个位置号就是第 i 头奶牛的身高 H_i。

2. 把 $b[H_i]$ 减 1（从 1 变为 0）。

也就是说，我们需要**实时维护一个 01 序列，支持查询第 k 个 1 的位置（k 为任意整数），以及修改序列中的一个数值。**

方法一：树状数组+二分，单次操作 $O(\log^2 n)$

用树状数组 c 维护 01 序列 b 的前缀和，在每次查询时二分答案，通过 ask(mid) 即可得到前 mid 个数中有多少个 1，与 k 比较大小，即可确定二分上下界的变化。

方法二：树状数组+倍增，单次操作 $O(\log n)$

用树状数组 c 维护 01 序列 b 的前缀和，在每次查询时：

1. 初始化两个变量 $ans = 0$ 和 $sum = 0$。

2. 从 $\log n$（下取整）到 0 倒序考虑每个整数 p。

对于每个 p，若 $ans + 2^p \le n$ 且 $sum + c[ans + 2^p] < k$，则令 $sum \mathrel{+}= c[ans + 2^p]$，$ans \mathrel{+}= 2^p$。

3. 最后，$H_i = ans + 1$ 即为所求。

该做法与 0x06 节“倍增”中给出的第一个小问题采用了类似的思想，都是“**以 2 的整数次幂为步长，能累加则累加**”。树状数组 c 恰好已经为我们维护了区间长度为 2 的次幂的一些信息，直接结合树状数组使用这种思想，就得到了上面这个高效算法。



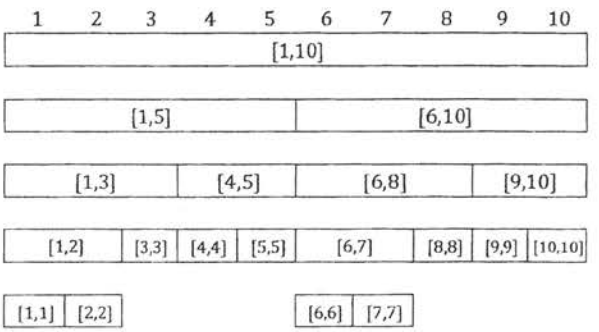
线段树（Segment Tree）是一种基于分治思想的二叉树结构，用于在区间上进行信息统计。与按照二进制位（2 的次幂）进行区间划分的树状数组相比，线段树是一种更加通用的结构。

1. 线段树的每个节点都代表一个区间。

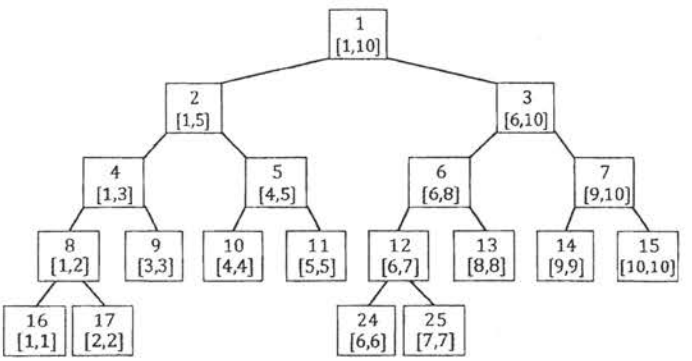
2. 线段树具有唯一的根节点，代表的区间是整个统计范围，如 $[1,N]$。

3. 线段树的每个叶节点都代表一个长度为 1 的元区间 $[x,x]$。

4. 对于每个内部节点 $[l,r]$，它的左子节点是 $[l,mid]$，右子节点是 $[mid+1,r]$，其中 $mid=(l+r)/2$（向下取整）。



区间视角



二叉树视角

上图展示了一棵线段树。可以发现，除去树的最后一层，整棵线段树一定是一棵完

全二叉树，树的深度为 $O(\log N)$。因此，我们可以按照与二叉堆类似的“父子 2 倍”节点编号方法。

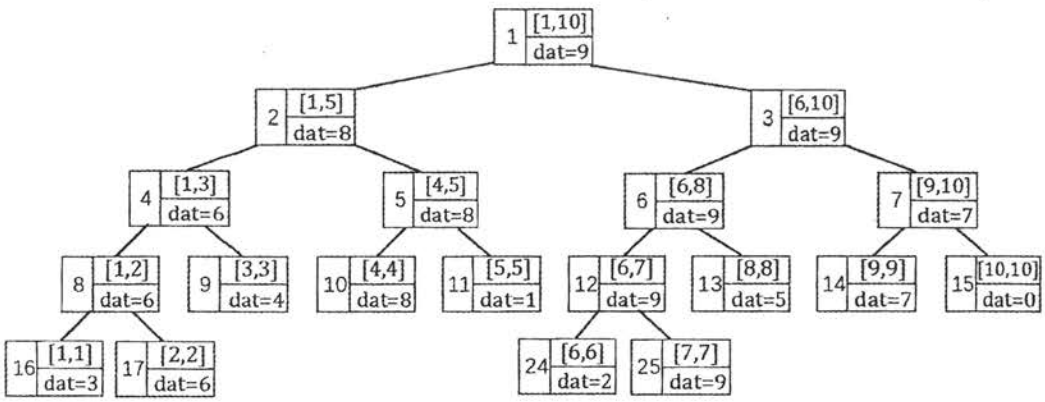
1. 根节点编号为 1。

2. 编号为 x 的节点的左子节点编号为 $x*2$，右子节点编号为 $x*2+1$。

这样一来，我们就能简单地使用一个 struct 数组来保存线段树。当然，树的最后一层节点在数组中保存的位置不是连续的，直接空出数组中多余的位置即可。在理想情况下，N 个叶节点的满二叉树有 $N+N/2+N/4+\cdots+2+1=2N-1$ 个节点。因为在上述存储方式下，最后还有一层产生了空余，所以保存线段树的数组长度要不小于 $4N$，才能保证不会越界。

线段树的建树

线段树的基本用途是对序列进行维护，支持查询与修改指令。给定一个长度为 N 的序列 A，我们可以在区间 $[1,N]$ 上建立一棵线段树，每个叶节点 $[i,i]$ 保存 $A[i]$ 的值。线段树的二叉树结构可以很方便地从下往上传递信息。以区间最大值问题为例，记 $\mathrm{dat}(l,r)$ 等于 $\max_{l\le i\le r}\{A[i]\}$，显然 $\mathrm{dat}(l,r)=\max(\mathrm{dat}(l,mid),\ \mathrm{dat}(mid+1,r))$。



Build(1,1,10)　　A={3,6,4,8,1,2,9,5,7,0}

下面这段代码建立了一棵线段树，并在每个节点上保存了对应区间的最大值。

```
struct SegmentTree {
    int l, r;
    int dat;
} t[SIZE * 4]; // struct 数组存储线段树

void build(int p, int l, int r) {
    t[p].l = l, t[p].r = r;                    // 节点 p 代表区间[l,r]
    if (l==r) { t[p].dat = a[l]; return; } // 叶节点
    int mid = (l + r) / 2;                     // 折半
```

```
    build(p*2, l, mid);              // 左子节点[l,mid]，编号 p*2
    build(p*2+1, mid+1, r);          // 右子节点[mid+1,r]，编号 p*2+1
    t[p].dat = max(t[p*2].dat, t[p*2+1].dat); // 从下往上传递信息
}

build(1, 1, n); // 调用入口
```

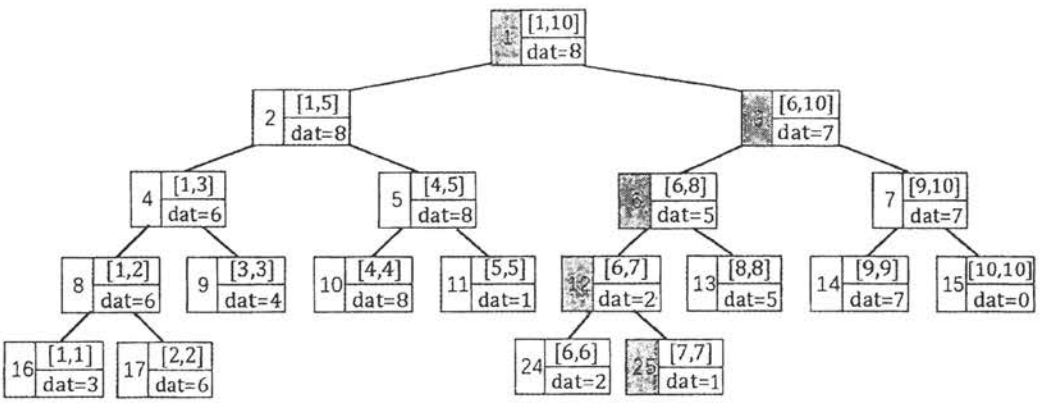
线段树的单点修改

单点修改是一条形如“C x v”的指令，表示把 $A[x]$ 的值修改为 v。

在线段树中，根节点（编号为 1 的节点）是执行各种指令的入口。我们需要从根节点出发，递归找到代表区间 $[x,x]$ 的叶节点，然后从下往上更新 $[x,x]$ 以及它的所有祖先节点上保存的信息，如下图所示。时间复杂度为 $O(\log N)$。

```
void change(int p, int x, int v) {
    if (t[p].l == t[p].r) { t[p].dat = v; return; } // 找到叶节点
    int mid = (t[p].l + t[p].r) / 2;
    if (x <= mid) change(p*2, x, v);          // x 属于左半区间
    else change(p*2+1, x, v);                 // x 属于右半区间
    t[p].dat = max(t[p*2].dat, t[p*2+1].dat); // 从下往上更新信息
}

change(1, x, v); // 调用入口
```



Change(1,7,1)

线段树的区间查询

区间查询是一条形如“Q l r”的指令，例如查询序列 A 在区间 $[l,r]$ 上的最大值，即 $max_{l\le i\le r}\{A[i]\}$。我们只需要从根节点开始，递归执行以下过程。如下页图所示。

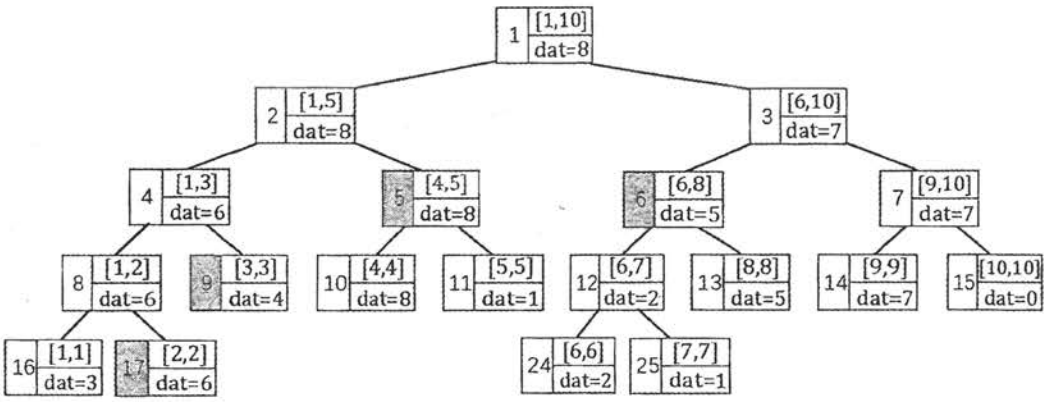
1. 若 $[l,r]$ 完全覆盖了当前节点代表的区间，则立即回溯，并且该节点的 dat 值为候选答案。

2. 若左子节点与 $[l,r]$ 有重叠部分，则递归访问左子节点。

3. 若右子节点与 $[l,r]$ 有重叠部分，则递归访问右子节点。

```
int ask(int p, int l, int r) {
    if (l <= t[p].l && r >= t[p].r) return t[p].dat; // 完全包含
    int mid = (t[p].l + t[p].r) / 2;
    int val = -(1<<30); // 负无穷大
    if (l<=mid) val = max(val, ask(p*2, l, r)); // 左子节点有重叠
    if (r>mid) val = max(val, ask(p*2+1, l, r)); // 右子节点有重叠
    return val;
}

cout << ask(1, l, r) << endl; // 调用入口
```



Ask(1,2,8)=max{6,4,8,5}=8

该查询过程会**把询问区间 $[l,r]$ 在线段树上分成 $O(\log N)$ 个节点**，取它们的最大值作为答案。为什么是 $O(\log N)$ 个呢？仔细分析上述过程，在每个节点 $[p_l,p_r]$ 上，设 $mid=(p_l+p_r)/2$（向下取整），可能会出现以下几种情况。

1. $l \le p_l \le p_r \le r$，即完全覆盖了当前节点，直接返回。
2. $p_l \le l \le p_r \le r$，即只有 l 处于节点之内。
   (1) $l > mid$，只会递归右子树。
   (2) $l \le mid$，虽然递归两棵子树，但是右子节点会在递归后直接返回。
3. $l \le p_l \le r \le p_r$，即只有 r 处于节点之内，与情况 2 类似。
4. $p_l \le l \le r \le p_r$，即 l 与 r 都位于节点之内。
   (1) l,r 都位于 mid 的一侧，只会递归一棵子树。

(2) l,r 分别位于 mid 的两侧，递归左右两棵子树。

也就是说，只有情况 4(2) 会真正产生对左右两棵子树的递归。请读者思考，这种情况至多发生一次，之后在子节点上就会变成情况 2 或 3。因此，上述查询过程的时间复杂度为 $O(2\log N)=O(\log N)$。从宏观上理解，相当于 l,r 两个端点分别在线段树上划分出一条递归访问路径，情况 4(2) 在两条路径于从下往上的第一次交会处产生。

至此，线段树已经能够像 0x06 节的 ST 算法一样处理区间最值问题，并且还支持动态修改某个数的值。同时，线段树也已经能支持上一节中树状数组的单点增加与查询前缀和指令。在讨论区间修改之前，我们先来通过几道例题加深对线段树的印象。

例题 **Can you answer on these queries III** AcWing245

给定长度为 N 的数列 A，以及 M 条指令 $(N\le 5*10^5, M\le 10^5)$，每条指令可能是以下两种之一。

1. “1 x y”，查询区间 $[x,y]$ 中的最大连续子段和，即 $\max\limits_{x\le l\le r\le y}\{\sum_{i=l}^{r}A[i]\}$。
2. “2 x y”，把 $A[x]$ 改成 y。

对于每个询问，输出一个整数表示答案。

在线段树上的每个节点上，除了区间端点外，再维护 4 个信息：区间和 sum，区间最大连续子段和 dat，紧靠左端的最大连续子段和 lmax，紧靠右端的最大连续子段和 rmax。

线段树的整体框架不变，我们只需完善在 build 和 change 函数中从下往上传递的信息。

$$t[p].\text{sum}=t[p*2].\text{sum}+t[p*2+1].\text{sum}$$
$$t[p].\text{lmax}=\max(t[p*2].\text{lmax},t[p*2].\text{sum}+t[p*2+1].\text{lmax})$$
$$t[p].\text{rmax}=\max(t[p*2+1].\text{rmax},t[p*2+1].\text{sum}+t[p*2].\text{rmax})$$
$$t[p].\text{dat}=\max(t[p*2].\text{dat},t[p*2+1].\text{dat},t[p*2].\text{rmax}+t[p*2+1].\text{lmax})$$

从这道题目我们也可以看出，线段树作为一种比较通用的数据结构，能够维护各式各样的信息，前提是这些信息容易按照区间进行划分与合并（又称满足区间可加性）。我们只需要在父子传递信息和更新答案时稍作变化即可。

例题 **Interval GCD** AcWing246

给定一个长度为 N 的数列 A，以及 M 条指令 $(N\le 5*10^5, M\le 10^5)$，每条指令可能是以下两种之一。

1. “C l r d”，表示把 $A[l],A[l+1],\cdots,A[r]$ 都加上 d。
2. “Q l r”，表示询问 $A[l],A[l+1],\cdots,A[r]$ 的最大公约数(GCD)。

对于每个询问，输出一个整数表示答案。

根据《九章算术之“更相减损术”》，我们知道 $\gcd(x,y)=\gcd(x,y-x)$。它可以进一步扩展到三个数的情况：$\gcd(x,y,z)=\gcd(x,y-x,z-y)$。实际上，读者用数学归纳法容易证明，该性质对任意多个整数都成立。

因此，我们可以构造一个长度为 N 的新数列 B，其中 $B[i]=A[i]-A[i-1]$，$B[1]$ 可为任意值。数列 B 称为 A 的**差分序列**。用线段树维护序列 B 的区间最大公约数。

这样一来，询问“Q l r”，就等于求出 $\gcd\big(A[l],ask(1,l+1,r)\big)$。

在指令“C l r d”下，只有 $B[l]$ 加了 d，$B[r+1]$ 被减掉了 d，所以在维护 B 的线段树上只需两次单点修改即可。另外，询问时需要数列 A 中的值，可额外用一个支持“区间增加、单点查询”的树状数组对 A 进行维护。参考程序请查阅配套资源。

延迟标记

在线段树的“区间查询”指令中，每当遇到被询问区间 $[l,r]$ 完全覆盖的节点时，可以立即把该节点上存储的信息作为候选答案返回。我们已经证明，被询问区间 $[l,r]$ 在线段树上会被分成 $O(\log N)$ 个小区间（节点），从而在 $O(\log N)$ 的时间内求出答案。不过，在“区间修改”指令中，如果某个节点被修改区间 $[l,r]$ 完全覆盖，那么以该节点为根的整棵子树中的所有节点存储的信息都会发生变化，若逐一进行更新，将使得一次区间修改指令的时间复杂度增加到 $O(N)$，这是我们不能接受的。

试想，如果我们在一次修改指令中发现节点 p 代表的区间 $[p_l,p_r]$ 被修改区间 $[l,r]$ 完全覆盖，并且逐一更新了子树 p 中的所有节点，但是在之后的查询指令中却根本没有用到 $[l,r]$ 的子区间作为候选答案，那么更新 p 的整棵子树就是徒劳的。

换言之，我们在执行修改指令时，同样可以在 $l\le p_l\le p_r\le r$ 的情况下立即返回，只不过在回溯之前向节点 p 增加一个标记，标识“该节点曾经被修改，但其子节点尚未被更新”。

如果在后续的指令中，需要从节点 p 向下递归，我们再检查 p 是否具有标记。若有标记，就根据标记信息更新 p 的两个子节点，同时为 p 的两个子节点增加标记，然后清除 p 的标记。

也就是说，除了在修改指令中直接划分成的 $O(\log N)$ 个节点之外，对任意节点的修改都延迟到“在后续操作中递归进入它的父节点时”再执行。这样一来，每条查询或修改指令的时间复杂度都降低到了 $O(\log N)$。这些标记被称为“延迟标记”。延迟标记提供了线段树中从上往下传递信息的方式[①]。这种“延迟”也是设计算法与解决问题的一个重要思路。

① 有一种称为“标记永久化”的技巧，可以不下传标记，而是在查询时累计递归经过的所有标记产生的影响。这种做法局限性非常大，对线段树维护的信息和标记的性质有特殊要求，仅在二维线段树和可持久化线段树难以下传标记的情况下有所应用，超出了我们的讨论范围。

例题 **A Simple Problem with Integers** AcWing243

给定长度为 $N(N \le 10^5)$ 的数列 A，然后输入 $Q(Q \le 10^5)$ 行操作指令。

第一类指令形如“C l r d”，表示把数列中第 l~r 个数都加 d。

第二类指令形如“Q l r”，表示询问数列中第 l~r 个数的和。

在上一节中我们用树状数组解决了该问题，本节我们改用线段树来求解。除了左右端点 l,r 以外，线段树的每个节点上保存了 sum（区间和）、add（增量延迟标记）两个值。建树、查询和修改的框架仍然不变，spread 函数实现了延迟标记的向下传递。

需要指出的是，延迟标记的含义为“该节点曾经被修改，但其子节点尚未被更新”，即延迟标记标识的是子节点等待更新的情况。因此，一个节点被打上“延迟标记”的同时，它自身保存的信息应该已经被修改完毕。读者在编写代码时，一定要注意“更新信息”与“打标记”之间的关系，避免出现错误。

```
struct SegmentTree {
    int l, r;
    long long sum, add;
    #define l(x) tree[x].l
    #define r(x) tree[x].r
    #define sum(x) tree[x].sum
    #define add(x) tree[x].add
} tree[100010*4];
int a[100010], n, m;

void build(int p,int l,int r) {
    l(p) = l, r(p) = r;
    if (l==r) { sum(p)=a[l]; return; }
    int mid = (l+r)/2;
    build(p*2, l, mid);
    build(p*2+1, mid+1, r);
    sum(p) = sum(p*2) + sum(p*2+1);
}

void spread(int p) {
    if (add(p)) {                    // 节点 p 有标记
        sum(p*2) += add(p)*(r(p*2)-l(p*2)+1);    // 更新左子节点信息
        sum(p*2+1) += add(p)*(r(p*2+1)-l(p*2+1)+1); // 更新右子节点
        add(p*2) += add(p);          // 给左子节点打延迟标记
```

```
        add(p*2+1) += add(p);    // 给右子节点打延迟标记
        add(p) = 0;              // 清除 p 的标记
    }
}

void change(int p,int l,int r,int d) {
    if (l <= l(p) && r >= r(p)) {          // 完全覆盖
        sum(p) += (long long)d * (r(p)-l(p)+1); // 更新节点信息
        add(p) += d;                       // 给节点打延迟标记
        return;
    }
    spread(p);                                  // 下传延迟标记
    int mid = (l(p)+r(p))/2;
    if (l <= mid) change(p*2, l, r, d);
    if (r > mid) change(p*2+1, l, r, d);
    sum(p) = sum(p*2) + sum(p*2+1);
}

long long ask(int p,int l,int r) {
    if (l <= l(p) && r >= r(p)) return sum(p);
    spread(p);                                  // 下传延迟标记
    int mid = (l(p)+r(p))/2;
    long long val = 0;
    if (l <= mid) val += ask(p*2,l,r);
    if (r > mid) val += ask(p*2+1,l,r);
    return val;
}

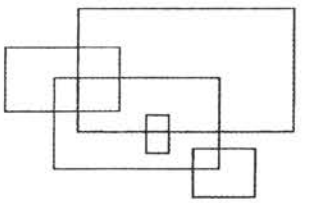
int main() {
    cin >> n >> m;
    for (int i=1; i<=n; i++) scanf("%d",&a[i]);
    build(1, 1, n);
    while (m--) {
        char op[2]; int l, r, d;
        scanf("%s%d%d", op, &l, &r);
        if(op[0] == 'C') {
            scanf("%d", &d);
            change(1, l, r, d);
```

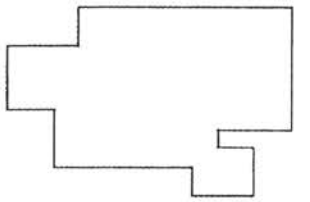
```
        }
        else printf("%lld\n", ask(1, l, r));
    }
}
```

扫描线

例题 **Atlantis** AcWing247

给定平面直角坐标系中的 N 个矩形，求它们的面积并，即这些矩形的并集在坐标系中覆盖的总面积，如下图所示。

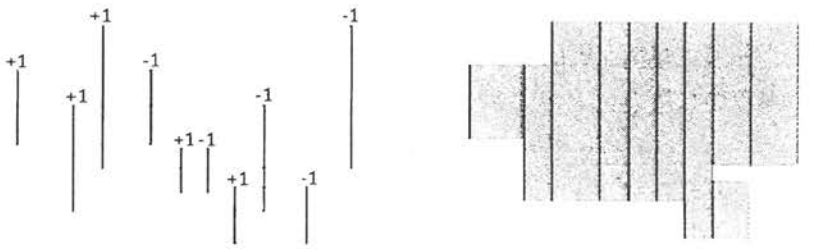




试想，如果我们用一条竖直直线从左到右扫过整个坐标系，那么直线上被并集图形覆盖的长度只会在每个矩形的左右边界处发生变化。

换言之，整个并集图形可以被分成 $2*N$ 段，每一段在直线上覆盖的长度（记为 L）是固定的，因此该段的面积就是 $L*$该段的宽度，各段面积之和即为所求。这条直线就被称为扫描线，这种解题思路被称为扫描线法。

具体来说，我们可以取出 N 个矩形的左右边界。若一个矩形的两个对角顶点坐标为 (x_1,y_1) 和 (x_2,y_2)，不妨设 $x_1<x_2,y_1<y_2$，则左边界记为四元组 $(x_1,y_1,y_2,1)$，右边界记为四元组 $(x_2,y_1,y_2,-1)$。把这 $2N$ 个四元组按照 x 递增排序，如下图所示。



注意到本题中的 y 坐标范围较大且不一定是整数，我们先把输入数据中出现的所有 y 坐标放入一个数组，排序、去重，完成**离散化**。设 $val(y)$ 表示 y 被离散化之后映射到的整数值，$raw(i)$ 表示整数值 i 对应的原始 y 坐标值。

在离散化后，若有 M 个不同的 y 坐标值，分别对应 $raw(1),raw(2),\cdots,raw(M)$，则扫描线至多被分成 $M-1$ 段，其中第 i 段为区间 $[raw(i),raw(i+1)]$。建立数组

c，用 $c[i]$ 记录扫描线上第 i 段被覆盖的次数。起初 c 数组中的元素全为 0。

逐一扫描排序后的 $2N$ 个四元组，设当前四元组为 (x,y_1,y_2,k)。我们把数组 c 中 $c[val(y_1)]$, $c[val(y_1)+1],\cdots,c[val(y_2)-1]$ 这些值都加 k，相当于覆盖了 $[y_1,y_2]$ 这个区间。此时，如果下一个四元组的横坐标为 x_2，则扫描线从 x 扫到 x_2 的过程中，被覆盖的长度就固定为 $\sum_{c[i]>0}\big(raw(i+1)-raw(i)\big)$，即数组 c 中至少被覆盖一次的“段”的总长度。于是，我们就让最终的答案 ans 累加上 $(x_2-x)*\sum_{c[i]>0}\big(raw(i+1)-raw(i)\big)$。

对于每个四元组，采用朴素算法在 c 数组上执行修改与统计，即可在 $O(N^2)$ 的时间内求出并集图形的面积。

值得说明的是，四元组中的 y_1,y_2 都是坐标，是一个“点”。我们需要维护的是扫描线上每一段被覆盖的次数及其长度，对“点”的覆盖次数进行统计是没有意义的。因此，我们把 c 数组中的每个值 $c[i]$ 定义成扫描线上一个区间的覆盖次数，四元组 (x,y_1,y_2,k) 对 $c[val(y_1)\sim val(y_2)-1]$ 产生影响。读者在解题时一定要多加留意此类边界情况。

另外，我们可以用线段树维护 c 数组，把算法优化到 $O(N\log N)$。

在本题中，我们只关心整个扫描线（线段树根节点）上被矩形覆盖的长度。而且，因为四元组 $(x,y_1,y_2,1)$ 和 $(x,y_1,y_2,-1)$ 成对出现，所以线段树区间修改也是成对出现的。在这种特殊情形下，我们没有必要下传延迟标记，可以采用更为简单的做法。

除左右端点 l,r 之外，在线段树的每个节点上维护两个值：该节点代表的区间被矩形覆盖的长度 len、该节点自身被覆盖的次数 cnt。最初，二者均为 0。

对于一个四元组 (x,y_1,y_2,k)，我们在 $[val(y_1),val(y_2)-1]$ 上执行区间修改。该区间被线段树划分成 $O(\log N)$ 个节点，我们把这些节点的 cnt 都加 k。

对于线段树中任意一个节点 $[l,r]$，若 $cnt>0$，则 len 等于 $raw(r+1)-raw(l)$，否则，该点 len 等于两个子节点的 len 之和。在一个节点的 cnt 值被修改，以及线段树从下往上传递信息时，我们都按照该方法更新 len 值。根节点的 len 值就是整个扫描线上被覆盖的长度。

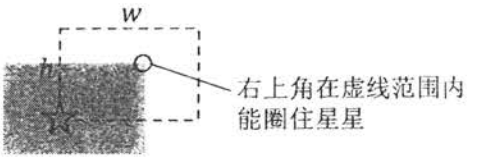
例题 **Stars in Your Window** AcWing248

在天空中有很多星星（看作平面直角坐标系），已知每颗星星的坐标和亮度（都是整数）。求用宽为 w、高为 h 的矩形（w,h 为正整数）能圈住的星星的亮度总和最大是多少（矩形边界上的星星不算）。

因为矩形的大小固定，所以矩形可以由它的任意一个顶点唯一确定。我们可以考虑把矩形的右上角顶点放在什么位置，圈住的星星亮度总和最大。

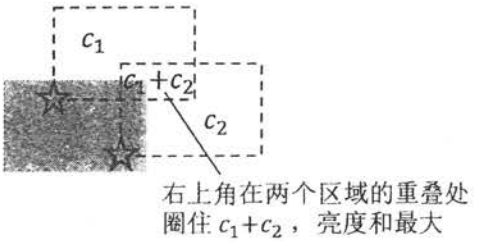
对于一颗星星 (x,y,c)，其中 x,y 为坐标，c 为亮度，“能圈住这颗星星的矩形

右上角顶点坐标的范围”如下图所示。该范围也是一个矩形，左下角顶点为 (x,y)，右上角顶点为 $(x+w,y+h)$。为了避免歧义，在接下来的讨论中，我们用“区域”一词来代指这个范围。



题目中说“矩形边界上的星星不算”。为了处理这种边界情况，不妨把所有星星向左、向下各移动 0.5 的距离，坐标从 (x,y) 变为 $(x-0.5,\ y-0.5)$。在此基础上，**不妨假设圈住星星的矩形顶点坐标都是整数**。于是，上图虚线“区域”的左下角可看作 (x,y)，右上角可看作 $(x+w-1,y+h-1)$，边界也算在内。容易证明这些假设不会影响答案。

此时，问题转化为：**平面上有若干个区域，每个区域都带有一个权值，求在哪个坐标上重叠的区域权值和最大**。其中，每一个区域都是由一颗星星产生的，权值等于星星的亮度，把原问题中的矩形右上角放在该区域中，就能圈住这颗星星。



在转化后的问题中，我们使用扫描线算法，取出每个区域的左右边界，保存成两个四元组 $(x,\ y,\ y+h-1,\ c)$ 和 $(x+w,\ y,\ y+h-1,\ -c)$。把这些四元组按照横坐标（第一维的值）排序。

同时，关于纵坐标建立一棵线段树，维护区间最大值 dat，起初全为 0。因为我们已经平移了星星的坐标，并假设矩形只能放在整点上，所以线段树维护的相当于是若干个数值（纵坐标整点上的权值和）构成的序列。逐一扫描每个四元组 (x,y_1,y_2,c)，在线段树中执行区间修改（可用延迟标记实现），把 $[y_1,y_2]$ 中的每个数都加 c，然后用根节点的 dat 值更新答案即可。

动态开点与线段树合并

在一些计数问题中，线段树用于维护值域（一段权值范围），这样的线段树也称为权值线段树。为了降低空间复杂度，我们可以不建出整棵线段树的结构，而是在最初只

建立一个根节点，代表整个区间，当需要访问线段树的某棵子树（某个子区间）时，再建立代表这个子区间的节点。采用这种方法维护的线段树被称为**动态开点**的线段树。动态开点的线段树抛弃了完全二叉树父子节点的 2 倍编号规则，改为使用变量记录左右子节点的编号（相当于指针）。同时，它也不再保存每个节点代表的区间，而是在每次递归访问的过程中作为参数传递。下面是一个动态开点的线段树的节点结构。

```
struct SegmentTree {
    int lc, rc; // 左右子节点的编号
    int dat; // 区间最大值
} tr[SIZE * 2];
int root, tot;

int build() { // 新建一个节点
    tot++;
    tr[tot].lc = tr[tot].rc = tr[tot].dat = 0;
    return tot;
}

// 在 main 函数中
tot = 0;
root = build(); // 根节点
```

下面的代码对线段树单点修改的过程稍加变动，实现了在动态开点的线段树中把 *val* 位置上的值加 *delta*，同时维护区间最大值的操作。

```
void insert(int p, int l, int r, int val, int delta) {
    if (l == r) {
        tr[p].dat += delta;
        return;
    }
    int mid = (l + r) >> 1; // 代表的区间[l,r]作为递归参数传递
    if (val <= mid) {
        if (!tr[p].lc) tr[p].lc = build(); // 动态开点
        insert(tr[p].lc, l, mid, val, delta);
    }
    else {
        if (!tr[p].rc) tr[p].rc = build(); // 动态开点
        insert(tr[p].rc, mid + 1, r, val, delta);
    }
```

```
        tr[p].dat = max(tr[tr[p].lc].dat, tr[tr[p].rc].dat);
    }
    insert(root, 1, n, val, delta); // 调用
```

常规线段树的其他操作也可以通过类似的变动，在动态开点的线段树上实现，这里就不再赘述。一棵维护值域 $[1,n]$ 的动态开点的线段树在经历 m 次单点操作后，节点数量的规模为 $O(m\log n)$，最终至多有 $2n-1$ 个节点。

如果有若干棵线段树，它们都维护相同的值域 $[1,n]$，那么它们对各个子区间的划分显然是一致的。假设有 m 次单点修改操作，每次操作在某一棵线段树上执行。所有操作完成后，我们希望把这些线段树对应位置上的值相加，同时维护区间最大值。

该问题可以通过**线段树合并**算法实现。我们依次合并这些线段树。合并两棵线段树时，用两个指针 p,q 从两个根节点出发，以递归的方式**同步遍历**两棵线段树。换句话说，p,q 指向的节点总是代表相同的子区间。

1. 若 p,q 之一为空，则以非空的那个作为合并后的节点。

2. 若 p,q 均不为空，则递归合并两棵左子树和两棵右子树，然后删除节点 q，以 p 为合并后的节点，自底向上更新最值信息。若已到达叶子节点，则直接把两个最值相加即可。

```
int merge(int p, int q, int l, int r) {
    if (!p) return q; // p,q 之一为空
    if (!q) return p;
    if (l == r) { // 到达叶子
        tr[p].dat += tr[q].dat;
        return p;
    }
    int mid = (l + r) >> 1;
    tr[p].lc = merge(tr[p].lc,tr[q].lc,l,mid); // 递归合并左子树
    tr[p].rc = merge(tr[p].rc,tr[q].rc,mid+1,r); // 递归合并右子树
    tr[p].dat = max(tr[tr[p].lc].dat, tr[tr[p].rc].dat); // 更新
    return p; // 以 p 为合并后的节点，相当于删除 q
}
```

仔细观察可以发现，若线段树在合并过程中发生递归，则必定会导致 p,q 之一被删除。因此，在完成所有线段树的合并后，merge 函数被执行的次数不会超过所有线段树的节点总数加 1 。故整个合并过程的时间复杂度为 $O(m\log n)$，与完成所有单点修改操作的时间复杂度相等，是一个很高效的过程。

我们会在 0x63 节结合具体例题，探讨线段树合并的应用并完成代码实现。

0x44 分 块

在前面两节中，我们探讨了树状数组与线段树两种数据结构。树状数组基于二进制划分与倍增思想，线段树基于分治思想。它们之所以能够高效地在一个序列上执行指令并统计信息，就是因为它们把序列中的元素聚合成大大小小的“段”，花费额外的代价对这些“段”进行维护，从而使得每个区间的信息可以快速由几个已有的“段”结合而成。

当然，树状数组与线段树也有其缺点。比如在维护较为复杂的信息（尤其是不满足区间可加、可减性的信息）时显得吃力，代码实现也不是那么简单、直观，需要深入理解并注意许多细节。在本节中，我们将介绍分块算法。分块的基本思想是通过适当的划分，预处理一部分信息并保存下来，用空间换取时间，达到时空平衡。事实上，分块更接近于“朴素”，效率往往比不上树状数组与线段树，但是它更加通用、容易实现。我们通过几道例题详细探讨各种形式的分块算法及其应用。

例题 **A Simple Problem with Integers** AcWing243

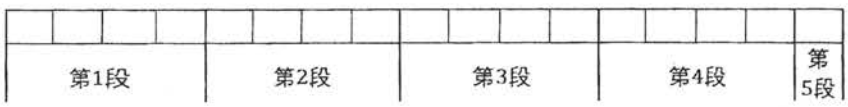
给定长度为 $N(N \le 10^5)$ 的数列 A，然后输入 $Q(Q \le 10^5)$ 行操作指令。

第一类指令形如“C l r d”，表示把数列中第 l~r 个数都加 d。

第二类指令形如“Q l r”，表示询问数列中第 l~r 个数的和。

我们已经用树状数组和线段树在 $O\big((N+Q)\log N\big)$ 的时间内解决过该问题。现在我们用分块来再次求解。

把数列 A 分成若干个长度不超过 $\lfloor\sqrt{N}\rfloor$ 的段，其中第 i 段左端点为 $(i-1)\lfloor\sqrt{N}\rfloor+1$，右端点为 $\min\big(i\lfloor\sqrt{N}\rfloor, N\big)$，如下图所示。



另外，预处理出数组 sum，其中 $sum[i]$ 表示第 i 段的区间和。设 $add[i]$ 表示第 i 段的“增量标记”，起初 $add[i]=0$。

对于指令“C l r d”：

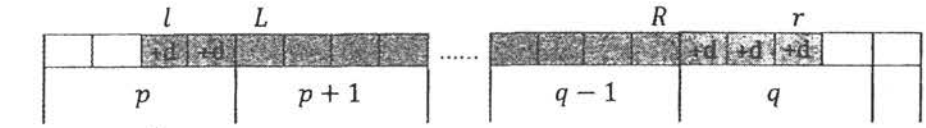
1. 若 l 与 r 同时处于第 i 段内，则直接把 $A[l], A[l+1], \cdots, A[r]$ 都加 d，同时令 $sum[i] += d * (r - l + 1)$。

2. 否则，设 l 处于第 p 段，r 处于第 q 段。

(1) 对于 $i \in [p+1, q-1]$，令 $add[i] += d$。

(2) 对于开头、结尾不足一整段的两部分，按照与第 1 种情况相同的方法朴素地更新。

该指令的操作方法如下图所示。



$[L, R]$ 分成若干个整段处理，$[l, L), (R, r]$ 不足整段，朴素处理

对于指令“Q l r”：

1. 若 l 与 r 同时处于第 i 段内，则 $(A[l] + A[l+1] + \cdots + A[r]) + (r - l + 1) * add[i]$ 就是答案。

2. 否则，设 l 处于第 p 段，r 处于第 q 段，初始化 $ans = 0$。

(1) 对于 $i \in [p+1, q-1]$，令 $ans += sum[i] + add[i] * len[i]$，其中 $len[i]$ 表示第 i 段的长度。

(2) 对于开头、结尾不足一整段的两部分，按照与第 1 种情况相同的方法朴素地累加。

这种分块算法对于整段的修改用标记 add 记录，对于不足整段的修改采取朴素算法。因为段数和段长都是 $O(\sqrt{N})$，所以整个算法的时间复杂度为 $O((N+Q) * \sqrt{N})$。大部分常见的分块思想都可以用“**大段维护、局部朴素**”来形容。

```
long long a[100010], sum[100010], add[100010];
int L[100010], R[100010]; // 每段左右端点
int pos[100010]; // 每个位置属于哪一段
int n, m, t;

void change(int l, int r, long long d) {
    int p = pos[l], q = pos[r];
    if (p == q) {
        for (int i = l; i <= r; i++) a[i] += d;
        sum[p] += d*(r - l + 1);
    }
    else {
```

```
        for (int i = p + 1; i <= q - 1; i++) add[i] += d;
        for (int i = l; i <= R[p]; i++) a[i] += d;
        sum[p] += d*(R[p] - l + 1);
        for (int i = L[q]; i <= r; i++) a[i] += d;
        sum[q] += d*(r - L[q] + 1);
    }
}

long long ask(int l, int r) {
    int p = pos[l], q = pos[r];
    long long ans = 0;
    if (p == q) {
        for (int i = l; i <= r; i++) ans += a[i];
        ans += add[p] * (r - l + 1);
    }
    else {
        for (int i = p + 1; i <= q - 1; i++)
            ans += sum[i] + add[i] * (R[i] - L[i] + 1);
        for (int i = l; i <= R[p]; i++) ans += a[i];
        ans += add[p] * (R[p] - l + 1);
        for (int i = L[q]; i <= r; i++) ans += a[i];
        ans += add[q] * (r - L[q] + 1);
    }
    return ans;
}

int main() {
    cin >> n >> m;
    for (int i = 1; i <= n; i++) scanf("%lld", &a[i]);
    // 分块
    t = sqrt(n);
    for (int i = 1; i <= t; i++) {
        L[i] = (i - 1)*sqrt(n) + 1;
        R[i] = i*sqrt(n);
    }
    if (R[t] < n) t++, L[t] = R[t - 1] + 1, R[t] = n;
    // 预处理
    for (int i = 1; i <= t; i++)
```

```
        for (int j = L[i]; j <= R[i]; j++) {
            pos[j] = i;
            sum[i] += a[j];
        }
    // 指令
    while (m--) {
        char op[3];
        int l, r, d;
        scanf("%s%d%d", op, &l, &r);
        if (op[0] == 'C') {
            scanf("%d", &d);
            change(l, r, d);
        }
        else printf("%lld\n", ask(l, r));
    }
}
```

POJ3468	复杂度	时间	内存	代码	优劣
树状数组	$O((N+Q)\log N)$	1.0s	3MB	850B	效率高、代码短 不易扩展、不太直观
线段树	$O((N+Q)\log N)$	1.5s	7MB	1700B	效率较高、扩展性好 代码较长、直观性一般
分块	$O\left((N+Q)\sqrt{N}\right)$	1.9s	1.5MB	1500B	通用、直观 效率偏低、码长一般
朴素	$O((N+Q)*N)$	TLE	1MB	500B	略

例题 蒲公英 AcWing249

在乡下的小路旁种着许多蒲公英，而我们的问题正是与这些蒲公英有关。

为了简化起见，我们把所有的蒲公英看成一个长度为 n 的序列 $a_1, a_2, \cdots, a_n$。其中 a_i 为一个正整数，表示第 i 棵蒲公英的种类编号。

接下来有 m 次询问，每次询问一个区间 $[x, y]$，你需要回答 $a_x, a_{x+1}, \cdots, a_y$ 里出现次数最多的是哪种蒲公英，如果有若干种蒲公英出现次数相同，则输出种类编号最小的那个。

注意，你的算法必须是在线的。$n \le 4*10^4, m \le 5*10^5, a_i \le 10^9$。

本题是经典的在线求区间众数问题。因为众数不具有“区间可加性”（已知序列

a 中区间 $[x,y]$ 的众数和区间 $[y+1,z]$ 的众数，不能直接得到区间 $[x,z]$ 的众数），所以用树状数组或线段树维护就十分困难。下面我们介绍两种常见的分块做法。

若把序列 a 分成 T 块，则每块的长度 $L=N/T$，我们稍后会讨论 T 的取值。

对于每个询问 $[l,r]$，设 l 处于第 p 块，r 处于第 q 块。与上一道题一样，我们把区间 $[l,r]$ 分成三部分。

1. 开头不足一整段的 $[l,L)$。

2. 第 $p+1\sim q-1$ 块构成的区间 $[L,R]$。

3. 结尾不足一整段的 $(R,r]$。

显然，a 序列在区间 $[l,r]$ 中的众数只可能来自以下两种情况。

1. 区间 $[L,R]$ 的众数。

2. 出现在 $[l,L)$ 与 $(R,r]$ 之间的数。

方法一

预处理出所有以“段边界”为端点的区间 $[L,R]$ 中每个数出现的次数，以及区间众数。这样的区间共有 $O(T^2)$ 个，并且保存每个数出现的次数需要长度为 $O(N)$ 的数组（记为 $cnt_{L,R}$）。

对于每个询问中的 $[l,L)$ 与 $(R,r]$，可以通过朴素扫描，在数组 $cnt_{L,R}$ 的基础上累加次数，从而更新答案。回答询问后再进行一次朴素扫描，从 $cnt_{L,R}$ 中减少次数，把数组复原。

这个算法的时间为 $O(NT^2+MN/T)$，空间为 $O(NT^2)$。不妨设 N,M 为相同数量级，通过方程 $NT^2=MN/T$，可解得 $T\approx\sqrt[3]{N}$，此时整个算法的复杂度在 $O(N^{5/3})$ 级别。

方法二

在预处理时，只保存所有以“段边界”为端点的区间 $[L,R]$ 的众数。另外，对每个数值建立一个 STL vector，按顺序保存该数值在序列 a 中每次出现的位置。

对于每个询问，扫描 $[l,L)$ 与 $(R,r]$ 中的每个数 x，在对应的 vector 里二分查找即可得到 x 在 $[l,r]$ 中出现的次数，从而更新答案。

例如序列 $a=\{1,4,2,3,2,4,3,2,1,4\}$，数值 1 出现的位置为 $\{1,9\}$，数值 2 出现的位置为$\{3,5,8\}$，数值 3 出现的位置为 $\{4,7\}$，数值 4 出现的位置为 $\{2,6,10\}$。假设 $l=2$，$r=8$，若要求数值 2 在序列 a 的区间 $[l,r]$ 中出现多少次，则应该在 $\{3,5,8\}$ 中二分查找第一个 $\geq l$ 的数，得到 3（下标为 0）；二分查找最后一个 $\leq r$ 的数，得到 8（下标为 2）。把两个下标相减再加 1，得到答案：3 次。

这个算法的时间为 $O(NT+MN/T*\log N)$，空间为 $O(T^2)$。应取 $T=\sqrt{N\log N}$，此时整个算法的复杂度在 $O(N\sqrt{N\log N})$ 级别。

例题 磁力块 AcWing250

在一片广袤无垠的原野上，散落着 N 块磁石。每个磁石的性质可以用一个五元组 (x,y,m,p,r) 描述，其中 x,y 表示其坐标，m 是磁石的质量，p 是磁力，r 是吸引半径。

小 Q 带着一块自己的磁石 L 来到了这片原野的 (x_0,y_0) 处。他手持磁石 L 并保持原地不动，所有可以被 L 吸引的磁石将会被吸引过来。在每个时刻，他可以选择更换任意一块自己已经获得的磁石（当然也可以是最初携带的 L 磁石）在 (x_0,y_0) 处吸引更多的磁石。

磁石 A 可以吸引磁石 B 的条件是：磁石 A 位于 (x_0,y_0)，并且磁石 A 与磁石 B 的距离不大于磁石 A 的吸引半径，磁石 B 的质量不大于磁石 A 的磁力。特别地，只有小 Q 手持的磁石才会吸引其他磁石，原野上散落的磁石之间不会互相吸引。

小 Q 想知道，他最多能获得多少块磁石呢？$1 \le N \le 2.5 * 10^5$。

我们可以执行一个 BFS 框架，建立一个队列，最初仅包含磁石 L，每次从队头取出磁石，把所有能被吸引的磁石加入队尾。

接下来我们需要解决的问题就是判断哪些磁石能被吸引。本题中，磁石吸引条件包含两个维度，即质量 $\le$ 磁力、距离 $\le$ 吸引半径，使用平衡树等数据结构进行维护会比较复杂。这时分块算法就是一个不错的选择。

把 N 块磁石按照质量排序，分成 $\sqrt{N}$ 段。然后在每段内部，再重新按照距离排序。

每次用队头磁石（设为 H）吸引其他磁石时，根据排序的结果，一定存在一个整数 k，满足：

1. 第 $1\sim k-1$ 段中的所有磁石质量都不大于 H 的磁力；
2. 第 $k+1$ 段之后的磁石质量都大于 H 的磁力，不可能被吸引。

因为每一段内部已经按照距离重新排序，所以第 $1\sim k-1$ 段中与 (x_0,y_0) 的距离小于 H 吸引半径的磁石必定处于每段的开头部分。我们直接在第 $1\sim k-1$ 段中，从每段左端开始逐一扫描，把能吸引的磁石加入队尾，直至第一个不能被吸引的磁石（记为 P）。之后，我们把该段的开头位置移到 P 处。这样一来，被吸引的磁石就不会被重复扫描。所以对于每一段来说，上述扫描过程的均摊复杂度为 $O(1)$，处理全部 $k-1$ 段的复杂度为 $O(\sqrt{N})$。

对于第 k 段，朴素扫描该段，把能被吸引的磁石取走，时间复杂度为 $O(\sqrt{N})$。

整个算法的时间复杂度为 $O(N\sqrt{N})$。

例题 小 Z 的袜子 AcWing251

作为一个生活散漫的人，小 Z 每天早上都要耗费很久从一堆五颜六色的袜子中找

出一双来穿。终于有一天，小 Z 再也无法忍受这恼人的找袜子过程，于是他决定听天由命。

具体来说，小 Z 把这 N 只袜子从 1 到 N 编号，然后从编号 L 到 R 的袜子中随机选出两只来穿。尽管小 Z 并不在意两只袜子是不是完整的一双，他却很在意袜子的颜色，毕竟穿两只不同色的袜子会很尴尬。

你的任务便是告诉小 Z，他有多大的概率抽到两只颜色相同的袜子。当然，小 Z 希望这个概率尽量高，所以他可能会询问多个 (L,R) 以方便自己选择。

$N \leq 5*10^4$，询问次数 $\leq 5*10^4$，袜子颜色数 $\leq N$。

在本题中，我们将介绍分块算法的一种重要形式——对“询问”进行分块。这是一种离线做法，又被称为**莫队算法**（来源于出题人的名字）。

先把所有询问 $[l,r]$ 读入，**把这些询问按照左端点递增排序，然后分成 $\sqrt{N}$ 块。每块内部再按照右端点排序**。

这样一来，在每一块内部，相邻两个询问的**左端点变化在 $\sqrt{N}$ 以内**，右端点变化是单调的。如果我们以**上一次询问的回答为基础**，那么每次只需要花费 $O(\sqrt{N})$ 的时间处理左端多出或减少的部分。而整块中右端点增长的范围之和为 $O(N)$，所以在 $O(N\sqrt{N})$ 的时间内即可求解本题。

具体来说，对于每一块询问，我们先朴素处理该块的第一个询问，得到一个数组 num，表示在该块的第一个询问区间 $[l,r]$ 中，颜色为 c 的袜子有 $num[c]$ 只。另外，我们再维护一个变量 ans，存储 $\sum_c num[c]*(num[c]-1)/2$。

然后，我们依次考虑该块的其余询问。当左、右端点发生变化时，朴素地在 num 数组中进行增减，同时更新 ans 的值。显然，抽到同色袜子的概率就是 ans/C_{r-l+1}^2。



到目前为止，我们用数据结构处理的大多是序列上的问题。这些问题的形式一般是给定序列中的两个位置 l 和 r，在区间 $[l,r]$ 上执行查询或修改指令。如果给定一棵树，以及树上两个节点 x 和 y，那么与“序列上的区间”相对应的就是“树上两点之间的路径”。我们先不考虑对路径进行修改的操作。本节中介绍的点分治就是在一棵树上，对具有某些限定条件的路径静态地进行统计的算法。

例题 Tree AcWing252

给一棵有 N 个点的树，每条边都有一个权值。树上两个节点 x 与 y 之间的路径长度就是路径上各条边的权值之和。求长度不超过 K 的路径有多少条。

本题树中的边是无向的，即这棵树是一个由 N 个点、$N-1$ 条边构成的无向连通图。我们把这种树称为“**无根树**”，也就是说可以任意指定一个节点为根节点，而不影响问题的答案。

若指定节点 p 为根，则对 p 而言，树上的路径可以分为两类。

1. 经过根节点 p。

2. 包含于 p 的某一棵子树中（不经过根节点）。

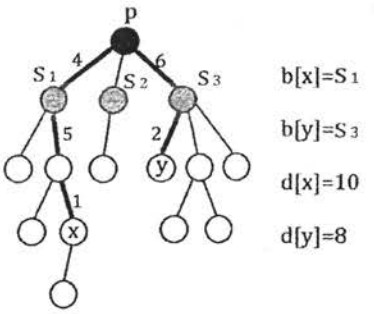
根据分治的思想，对于第 2 类路径，显然可以把 p 的每棵子树作为子问题，递归进行处理。

而对于第 1 类路径，可以从根节点 p 分成“$x\sim p$”与“$p\sim y$”两段。回顾在 0x21 节所学的知识，我们可以从 p 出发对整棵树进行 DFS，求出数组 d，其中 $d[x]$ **表示点 x 到根节点 p 的距离**。同时还可以求出数组 b，其中 $b[x]$ **表示点 x 属于根节点 p 的哪一棵子树**，特别地，令 $b[p]=p$。

此时满足题目要求的第 1 类路径就是满足以下两个条件的点对 (x,y) 的个数。

1. $b[x]\neq b[y]$。

2. $d[x]+d[y]\leq K$。如下页图所示。



定义 $Calc(p)$ 表示在以 p 为根的树中统计上述点对的个数(第 1 类路径的条数)。$Calc(p)$ 有两种常见的实现方式。针对不同的题目，二者各有优劣。

方法一：树上直接统计

设 p 的子树为 $s_1, s_2, \cdots, s_m$。

对于 s_i 中的每个节点 x，把在子树 $s_1, s_2, \cdots, s_{i-1}$ 中满足 $d[x]+d[y] \le K$ 的节点 y 的个数累加到答案中即可。

具体来说，可以建立一个树状数组，依次处理每棵子树 s_i。

1. 对 s_i 中的每个节点 x，查询前缀和 $\text{ask}(K-d[x])$，即为所求的 y 的个数。

2. 对 s_i 中的每个节点 x，执行 $\text{add}(d[x],1)$，表示与 p 距离为 $d[x]$ 的节点增加了 1 个。

按子树一棵棵进行处理保证了 $b[x] \ne b[y]$，查询前缀和保证了 $d[x]+d[y] \le K$。

需要注意的是，树状数组的范围与路径长度有关，这个范围远比 N 要大。而本题中不易进行离散化。一种解决方案是用平衡树代替树状数组，以保证 $O(N \log N)$ 的复杂度，但代码复杂程度显著增加。所以本题更适用下一种方法。

方法二：指针扫描数组

把树中每个点放进一个数组 a，并把数组 a 按照节点的 d 值排序。

使用两个指针 L, R 分别从前、后开始扫描 a 数组。

容易发现，在指针 L 从左向右扫描的过程中，恰好使得 $d[a[L]]+d[a[R]] \le K$ 的指针 R 的范围是从右向左单调递减的。

另外，我们用数组 $cnt[s]$ 维护在 $L+1$ 与 R 之间满足 $b[a[i]]=s$ 的位置 i 的个数。

于是，当路径的一端 x 等于 $a[L]$ 时，满足题目要求的路径另一端 y 的个数就是 $R-L-cnt[b[a[L]]]$。

总而言之，整个点分治算法的过程就是：

1. 任选一个根节点 p（后面我们将说明，p **应该取树的重心**）；

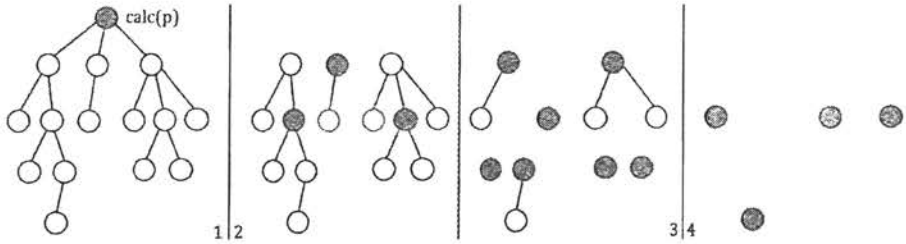
2. 从 p 出发进行一次 DFS，求出 d 数组和 b 数组；

3. 执行 $Calc(p)$；

4. 删除根节点 p，对 p 的每棵子树（看作无根树）递归执行 1~4 步。

在点分治过程中，每一层的所有递归过程合计对每个节点处理 1 次。因此，若递归最深到达第 T 层，则整个算法的时间复杂度就是 $O(TN\log N)$。

如果问题中的树是一条链，最坏情况下每次都以链的一端为根，那么点分治将需要递归 N 层，时间复杂度退化到 $O(N^2\log N)$。为了避免这种情况，我们**每次选择树的重心**（曾在 0x21 节中提及）**作为根节点** p。此时容易证明 p 的每棵子树的大小都不会超过整棵树大小的一半，点分治就至多递归 $O(\log N)$ 层，整个算法的时间复杂度为 $O(N\log^2 N)$。如下图所示。



0x46 二叉查找树与平衡树初步

在二叉树中，有两组非常重要的条件，分别是两类数据结构的基础性质。其一是“堆性质”，我们曾在 0x17 节中提及。二叉堆以及高级数据结构中的所有可合并堆，都满足“堆性质”。其二就是本节即将探讨的“BST 性质”，它是二叉查找树（Binary Search Tree）以及所有平衡树的基础。

BST

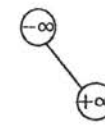
给定一棵二叉树，树上的每个节点带有一个数值，称为节点的“**关键码**”。所谓“BST 性质”是指，对于树中的任意一个节点：

1. 该节点的关键码不小于它的左子树中任意节点的关键码；
2. 该节点的关键码不大于它的右子树中任意节点的关键码。

满足上述性质的二叉树就是一棵“**二叉查找树**”(BST)。显然，二叉查找树的中序遍历是一个关键码非严格单调递增的节点序列。

BST 的建立

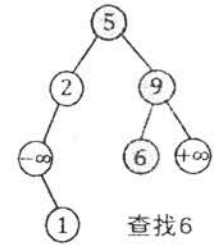
为了避免越界，减少边界情况的特殊判断，我们一般在 BST 中额外插入一个关键码为正无穷（一个很大的正整数）和一个关键码为负无穷的节点。仅由这两个节点构成的 BST 就是一棵初始的空 BST。如右图所示。



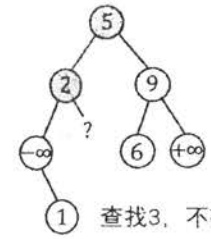
为了简便起见，在接下来的操作中，假设 BST 不会含有关键码相同的节点。

```
struct BST {
    int l, r; // 左右子节点在数组中的下标
    int val;  // 节点关键码
} a[SIZE]; // 数组模拟链表
int tot, root, INF = 1<<30;
```

```
int New(int val) {
    a[++tot].val = val;
    return tot;
}
void Build() {
    New(-INF), New(INF);
    root = 1, a[1].r = 2;
}
```



查找6



查找3，不存在

BST 的检索

在 BST 中检索是否存在关键码为 *val* 的节点。

设变量 *p* 等于根节点 *root*，执行以下过程。

1. 若 *p* 的关键码等于 *val*，则已经找到。
2. 若 *p* 的关键码大于 *val*
   (1) 若 *p* 的左子节点为空，则说明不存在 *val*。
   (2) 若 *p* 的左子节点不为空，在 *p* 的左子树中递归进行检索。
3. 若 *p* 的关键码小于 *val*
   (1) 若 *p* 的右子节点为空，则说明不存在 *val*。
   (2) 若 *p* 的右子节点不为空，在 *p* 的右子树中递归进行检索。

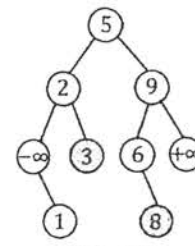
```
int Get(int p, int val) {
    if (p == 0) return 0; // 检索失败
    if (val == a[p].val) return p; // 检索成功
    return val < a[p].val ? Get(a[p].l, val) : Get(a[p].r, val);
}
```

BST 的插入

在 BST 中插入一个新的值 *val*（假设目前 BST 中不存在关键码为 *val* 的节点）。

与 BST 的检索过程类似。

在发现要走向的 *p* 的子节点为空，说明 *val* 不存在时，直接建立关键码为 *val* 的新节点作为 *p* 的子节点。



插入3和8

```
void Insert(int &p, int val) {
    if (p == 0) {
        p = New(val); // 注意 p 是引用，其父节点的 l 或 r 值会被同时更新
        return;
```

```
    }
    if (val == a[p].val) return;
    if (val < a[p].val) Insert(a[p].l, val);
    else Insert(a[p].r, val);
}
```

BST 求前驱/后继

以“后继”为例。*val* 的“后继”指的是在 BST 中关键码大于 *val* 的前提下，关键码最小的节点。

初始化 *ans* 为具有正无穷关键码的那个节点的编号。然后，在 BST 中检索 *val*。在检索过程中，每经过一个节点，都检查该节点的关键码，判断能否更新所求的后继 *ans*。

检索完成后，有三种可能的结果。

1. 没有找到 *val*。

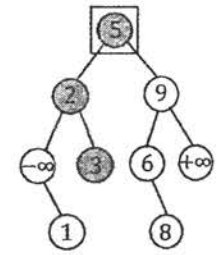
此时 *val* 的后继就在已经经过的节点中，*ans* 即为所求。

2. 找到了关键码为 *val* 的节点 *p*，但 *p* 没有右子树。

与上一种情况相同，*ans* 即为所求。

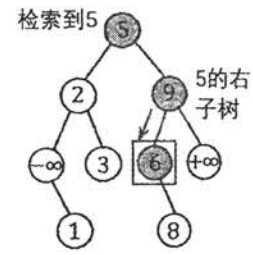
3. 找到了关键码为 *val* 的节点 *p*，且 *p* 有右子树。

从 *p* 的右子节点出发，一直向左走，就找到了 *val* 的后继。



求3的后继

```
int GetNext(int val) {
    int ans = 2; // a[2].val==INF
    int p = root;
    while (p) {
        if (val == a[p].val) { // 检索成功
            if (a[p].r > 0) { // 有右子树
                p = a[p].r;
                // 右子树上一直向左走
                while (a[p].l > 0) p = a[p].l;
                ans = p;
            }
            break;
        }
        // 每经过一个节点，都尝试更新后继
        if (a[p].val > val && a[p].val < a[ans].val) ans = p;
        p = val < a[p].val ? a[p].l : a[p].r;
    }
```



求5的后继

```
    return ans;
}
```

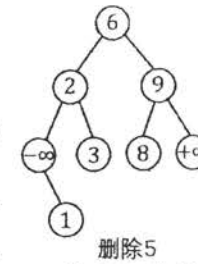
BST 的节点删除

从 BST 中删除关键码为 *val* 的节点。

首先，在 BST 中检索 *val*，得到节点 *p*。

若 *p* 的子节点个数小于 2，则直接删除 *p*，并令 *p* 的子节点代替 *p* 的位置，与 *p* 的父节点相连。

若 *p* 既有左子树又有右子树，则在 BST 中求出 *val* 的后继节点 *next*。因为 *next* 没有左子树，所以可以直接删除 *next*，并令 *next* 的右子树代替 *next* 的位置。最后，再让 *next* 节点代替 *p* 节点，删除 *p* 即可。如右图所示。



删除5
5的后继6代替5
6的右子树8代替6

```
void Remove(int &p, int val) { // 从子树 p 中删除值为 val 的节点
    if (p == 0) return;
    if (val == a[p].val) { // 已经检索到值为 val 的节点
        if (a[p].l == 0) { // 没有左子树
            p = a[p].r; // 右子树代替 p 的位置，注意 p 是引用
        }
        else if (a[p].r == 0) { // 没有右子树
            p = a[p].l; // 左子树代替 p 的位置，注意 p 是引用
        }
        else { // 既有左子树又有右子树
            // 求后继节点
            int next = a[p].r;
            while (a[next].l > 0) next = a[next].l;
            // next 一定没有左子树，直接删除
            Remove(a[p].r, a[next].val);
            // 令节点 next 代替节点 p 的位置
            a[next].l = a[p].l, a[next].r = a[p].r;
            p = next; // 注意 p 是引用
        }
        return;
    }
    if (val < a[p].val) {
        Remove(a[p].l, val);
    } else {
        Remove(a[p].r, val);
```

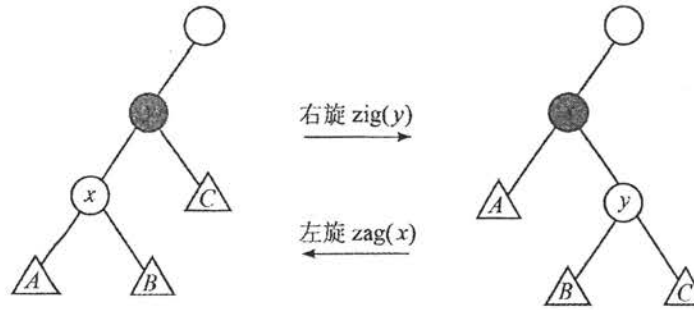
```
    }
}
```

在随机数据中，BST 一次操作的期望复杂度为 $O(\log N)$。然而，BST 很容易退化。例如，在 BST 中依次插入一个有序序列，将会得到一条链，平均每次操作的复杂度为 $O(N)$。我们称这种左右子树大小相差很大的 BST 是“不平衡”的。有很多方法可以维持 BST 的平衡，从而产生了各种平衡树。本节我们介绍一种入门级平衡树——Treap。

Treap

满足 BST 性质且中序遍历为相同序列的二叉查找树是不唯一的。这些二叉查找树是等价的，它们维护的是相同的一组数值。在这些二叉查找树上执行同样的操作，将得到相同的结果。因此，我们可以在维持 BST 性质的基础上，通过改变二叉查找树的形态，使得树上每个节点的左右子树大小达到平衡，从而使整棵树的深度维持在 $O(\log N)$ 级别。

改变形态并保持 BST 性质的方法就是“**旋转**”。最基本的旋转操作称为“单旋转”，它又分为“左旋”和“右旋”。如下图所示。



注意：某些书中把左、右旋操作定义为一个节点绕其父节点向左或向右旋转。本书后面即将讲解的 Treap 代码仅记录左右子节点，没有记录父节点，为了方便起见，统一以“旋转前处于父节点位置”（旋转后处于子节点位置）的节点作为左、右旋的作用对象（函数参数）。

以右旋为例。在初始情况下，x 是 y 的左子节点，A 和 B 分别是 x 的左右子树，C 是 y 的右子树。

“右旋”操作在保持 BST 性质的基础上，把 x 变为 y 的父节点。因为 x 的关键码小于 y 的关键码，所以 y 应该作为 x 的右子节点。

当 x 变成 y 的父节点后，y 的左子树就空了出来，于是 x 原来的右子树 B 就恰好作为 y 的左子树。

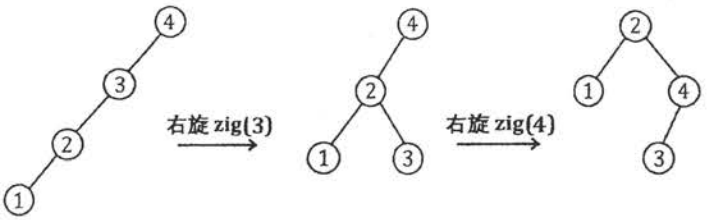
右旋操作的代码如下，$\text{zig}(p)$ **可以理解成把** p **的左子节点绕着** p **向右旋转。**

```
void zig(int &p) {
    int q = a[p].l;
    a[p].l = a[q].r, a[q].r = p;
    p = q; // 注意 p 是引用
}
```

左旋操作的代码如下，zag(p) 可以理解成把 p 的右子节点绕着 p 向左旋转。

```
void zag(int &p) {
    int q = a[p].r;
    a[p].r = a[q].l, a[q].l = p;
    p = q; // 注意 p 是引用
}
```

合理的旋转操作可使 BST 变得更“平衡”。如下图所示，对形态为一条链的 BST 进行一系列单旋转操作后，这棵 BST 变得比较平衡了。



现在，我们的问题是，怎样才算“合理”的旋转操作呢？我们发现，在随机数据下，普通的 BST 是趋近平衡的。Treap 的思想就是利用“随机”来创造平衡条件。因为在旋转过程中必须维持 BST 性质，所以 Treap 就把“随机”作用在堆性质上。

Treap 是英文 Tree 和 Heap 的合成词。Treap 在插入每个新节点时，给该节点随机生成一个额外的权值。然后像二叉堆的插入过程一样，自底向上依次检查，当某个节点不满足大根堆性质时，就执行单旋转，使该节点与其父节点的关系发生对换。

特别地，对于删除操作，因为 Treap 支持旋转，我们可以直接找到需要删除的节点，并把它**向下旋转成叶节点，最后直接删除**。这样就避免了采取类似普通 BST 的删除方法可能导致的节点信息更新、堆性质维护等复杂问题。

总而言之，Treap 通过适当的单旋转，在**维持节点关键码满足 BST 性质的同时，还使每个节点上随机生成的额外权值满足大根堆性质**。Treap 是一种平衡二叉查找树，检索、插入、求前驱后继以及删除节点的时间复杂度都是 $O(\log N)$。

例题 普通平衡树 AcWing253

你需要写一种数据结构（可参考本题标题），来维护一些数，其中需要提供以下操作：

1. 插入数值 x。

2. 删除数值 x (若有多个相同的数，应只删除一个)。
3. 查询数值 x 的排名 (若有多个相同的数，应输出最小的排名)。
4. 查询排名为 x 的数值。
5. 求数值 x 的前驱 (前驱定义为小于 x 的最大的数)。
6. 求数值 x 的后继 (后继定义为大于 x 的最小的数)。

这是一道平衡树的模板题，我们直接用 Treap 实现即可。

根据题意，数据中可能有相同的数值，我们可以给每个节点**增加一个域 *cnt*，记录该节点的“副本数”**，初始为 1。若插入已经存在的数值，就直接把“副本数”加 1。在删除时，减少节点的“副本数”，当减为 0 时删除该节点。这样就可以比较容易地处理关键码相同的问题。

题目还要求查询排名，我们可以给每个节点**增加一个域 *size*，记录以该节点为根的子树中所有节点的“副本数”之和**。当不存在重复数值时，*size* 其实就是子树大小。

与线段树一样，我们需要在插入或删除时从下往上更新 *size* 信息。另外，在发生旋转操作时，也需要同时修改 *size*。最后，在 BST 检索的基础上，通过判断左、右子树 *size* 的大小，选择适当的一侧递归，就很容易查询排名了。

因为在插入和删除操作时，Treap 的形态会发生变化，所以我们一般使用递归实现，以便于**在回溯时更新 Treap 上存储的 *size* 等信息**。

```
const int SIZE = 100010;
struct Treap {
    int l, r; // 左右子节点在数组中的下标
    int val, dat; // 节点关键码、权值
    int cnt, size; // 副本数、子树大小
} a[SIZE]; // 数组模拟链表
int tot, root, n, INF = 0x7fffffff;

int New(int val) {
    a[++tot].val = val;
    a[tot].dat = rand();
    a[tot].cnt = a[tot].size = 1;
    return tot;
}

void Update(int p) {
    a[p].size = a[a[p].l].size + a[a[p].r].size + a[p].cnt;
}
```

```
void Build() {
    New(-INF), New(INF);
    root = 1, a[1].r = 2;
    Update(root);
}

int GetRankByVal(int p, int val) {
    if (p == 0) return 0;
    if (val == a[p].val) return a[a[p].l].size + 1;
    if (val < a[p].val) return GetRankByVal(a[p].l, val);
    return GetRankByVal(a[p].r, val) + a[a[p].l].size + a[p].cnt;
}

int GetValByRank(int p, int rank) {
    if (p == 0) return INF;
    if (a[a[p].l].size >= rank) return GetValByRank(a[p].l, rank);
    if (a[a[p].l].size + a[p].cnt >= rank) return a[p].val;
    return GetValByRank(a[p].r, rank - a[a[p].l].size - a[p].cnt);
}

void zig(int &p) {
    int q = a[p].l;
    a[p].l = a[q].r, a[q].r = p, p = q;
    Update(a[p].r), Update(p);
}

void zag(int &p) {
    int q = a[p].r;
    a[p].r = a[q].l, a[q].l = p, p = q;
    Update(a[p].l), Update(p);
}

void Insert(int &p, int val) {
    if (p == 0) {
        p = New(val);
        return;
    }
```

```
    if (val == a[p].val) {
        a[p].cnt++, Update(p);
        return;
    }
    if (val < a[p].val) {
        Insert(a[p].l, val);
        if (a[p].dat < a[a[p].l].dat) zig(p); // 不满足堆性质，右旋
    }
    else {
        Insert(a[p].r, val);
        if (a[p].dat < a[a[p].r].dat) zag(p); // 不满足堆性质，左旋
    }
    Update(p);
}

int GetPre(int val) {
    int ans = 1; // a[1].val==-INF
    int p = root;
    while (p) {
        if (val == a[p].val) {
            if (a[p].l > 0) {
                p = a[p].l;
                while (a[p].r > 0) p = a[p].r; // 左子树上一直向右走
                ans = p;
            }
            break;
        }
        if (a[p].val < val && a[p].val > a[ans].val) ans = p;
        p = val < a[p].val ? a[p].l : a[p].r;
    }
    return a[ans].val;
}

int GetNext(int val) {
    int ans = 2; // a[2].val==INF
    int p = root;
    while (p) {
        if (val == a[p].val) {
```



```
            if (a[p].r > 0) {
                p = a[p].r;
                while (a[p].l > 0) p = a[p].l; // 右子树上一直向左走
                ans = p;
            }
            break;
        }
        if (a[p].val > val && a[p].val < a[ans].val) ans = p;
        p = val < a[p].val ? a[p].l : a[p].r;
    }
    return a[ans].val;
}
void Remove(int &p, int val) {
    if (p == 0) return;
    if (val == a[p].val) { // 检索到了 val
        if (a[p].cnt > 1) { // 有重复，减少副本数即可
            a[p].cnt--, Update(p);
            return;
        }
        if (a[p].l || a[p].r) { // 不是叶子节点，向下旋转
            if (a[p].r == 0 || a[a[p].l].dat > a[a[p].r].dat)
                zig(p), Remove(a[p].r, val);
            else
                zag(p), Remove(a[p].l, val);
            Update(p);
        }
        else p = 0; // 叶子节点，删除
        return;
    }
    val < a[p].val ? Remove(a[p].l, val) : Remove(a[p].r, val);
    Update(p);
}

int main() {
    Build();
    cin >> n;
    while (n--) {
        int opt, x;
```

```
        scanf("%d%d", &opt, &x);
        switch (opt) {
        case 1:
            Insert(root, x);
            break;
        case 2:
            Remove(root, x);
            break;
        case 3:
            printf("%d\n", GetRankByVal(root, x) - 1);
            break;
        case 4:
            printf("%d\n", GetValByRank(root, x + 1));
            break;
        case 5:
            printf("%d\n", GetPre(x));
            break;
        case 6:
            printf("%d\n", GetNext(x));
            break;
        }
    }
}
```

除了 Treap 以外，常见的平衡二叉查找树还有 Splay、红黑树、AVL、SBT、替罪羊树等。其中，C++ STL 的 set、map 等就采用了效率很高的红黑树的一种变体。不过，大多数平衡树因为实现比较复杂，或者应用范围能被其他平衡树替代，在算法竞赛等短时间程序设计中并不常用。在这些平衡树中，Splay（伸展树）灵活多变，应用广泛，能够很方便地支持各种动态的区间操作，是用于解决复杂问题的一个重要的高级数据结构。学有余力的读者可以自行查阅相关资料，并尝试用 Splay 完成本章末尾的一道标有“*”号的练习题。



0x47 离线分治算法

大多数数据结构问题都可以抽象为“维护一系列数据，并对一系列操作依次做出响应”的形式。这些操作一般分为统计数据信息的“查询”型操作和更新数据状态的“修改”型操作。其中，执行各项操作的“顺序”是一个要点，我们把它称为“时间轴”。

根据“查询”响应时间的不同，可以把解决上述数据结构问题的算法分为“在线”和“离线”两类。一个**在线算法**依次获得每项操作，并能够在每次“查询”时立即回答正确的结果，然后再继续执行下一次操作。一个**离线算法**需要预先知晓整个操作序列，经过一系列计算，最后再批量回答所有“查询”的结果。

根据两种操作在“时间”轴上分布的不同，可以把上述数据结构问题分成“动态”和“静态”两类。只包含“查询”型操作，或一切“查询”型操作都在一切“修改”型操作之后的问题称为**静态问题**，其余问题称为**动态问题**。

例如，第 0x44 节“分块”中，例题“A Simple Problem with Integers”是一道动态问题，其解法是一个在线算法。例题“蒲公英”是一道静态问题，其解法是一个在线算法。例题“小 Z 的袜子”是一道静态问题，其解法是一个离线算法。

本节介绍的第一类分治算法提供了一种把动态问题划分为若干个静态子问题，并使用离线算法进行求解的框架。

基于时间的分治算法

我们可以这样想：对于操作序列中的每个“查询”，计算其结果等价于计算“初始数据”以及“之前的所有修改”对该查询造成的影响。

设共有 M 项操作，$\forall l,r \in [1,\mathrm{M}]$，$l \le r$，定义 $\mathrm{solve}(l,r)$ 为：$\forall k \in [l,r]$，若第 k 项操作是“查询”，则计算第 $l \sim k-1$ 项操作中的“修改”对它造成的影响。计算方法如下。

1. 设 $mid = (l+r) \gg 1$，递归计算 $\mathrm{solve}(l, mid)$。
2. 递归计算 $\mathrm{solve}(mid+1, r)$。
3. 计算第 $l \sim mid$ 项操作中的所有“修改”对第 $mid+1 \sim r$ 项操作中所有“查询”造成的影响。

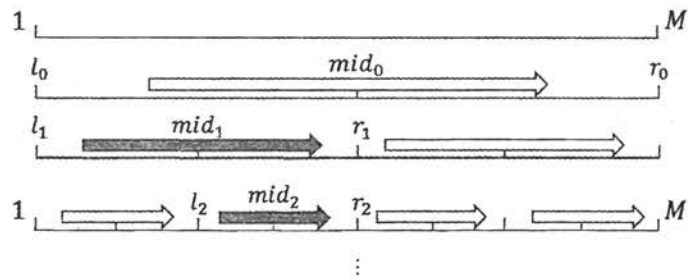
上述方法正确性的证明：

设第 k 项操作是“查询”。根据定义，若 $k \le mid$，则 solve(l, mid) 已经计算了第 $l \sim k-1$ 项操作中的“修改”对它的影响。若 $k > mid$，则 solve$(mid+1, r)$ 计算了第 $mid+1 \sim k-1$ 项操作中的“修改”对它的影响，再加上第 3 部分的计算，即得到了第 $l \sim k-1$ 项操作中的“修改”对它的影响。故上述计算方法是正确的。

证毕。

solve(l, r) 的计算方法显然是一个分治的过程。在不考虑“初始数据”的情况下，solve$(1, M)$ 即为我们的原始问题。当 $l = r$ 时，solve(l, r) 只包含一项操作，不需要进行任何计算，可作为递归边界。

当 $l < r$ 时，除了两次递归外，还需要执行第 3 部分的计算——左半部分的“修改”对右半部分的“查询”造成的影响。回顾本节开头部分的叙述，第 3 部分显然是一个“静态问题”。如下图所示，该分治算法把一个动态问题划分为 $O(M)$ 个静态问题，其中原始问题中每个“查询”的结果是由 $O(\log M)$ 个静态问题的结果共同组成的。



一般来说，一个静态问题要比动态问题简单许多。如果我们能在仅与 $r-l$ 的规模相关（与 M 或整个数据集的规模无关）的时间复杂度内解决 solve(l, r) 的第 3 部分，那么整个分治算法解决动态问题的时间复杂度就仅比解决同类静态问题的时间复杂度多了 $O(\log M)$。

这种离线分治算法是基于“时间”顺序对操作序列进行分治的，因此我们称它为**基于时间的分治算法**，又称为 **CDQ 分治**[①]算法。下面我们结合例题来理解这一算法。

例题 天使玩偶 AcWing254

Ayu 在七年前曾经收到过一个天使玩偶，当时她把它当作时间囊埋在了地下。而七年后的今天，Ayu 却忘了她把天使玩偶埋在了哪里，所以她决定仅凭一点模糊的记忆来寻找它。

① CDQ 分治，由 2008 年国际信息学奥林匹克竞赛（IOI）金牌女选手陈丹琦在国家集训队中引入而得名，为算法竞赛界中的一个广泛称呼。原论文请参阅本书配套资源中的资料。



我们把 Ayu 生活的小镇看作一个二维平面直角坐标系，而 Ayu 会不定时地记起可能在某个点 (x,y) 埋下了天使玩偶。或者 Ayu 会询问你，假如她在 (x,y)，那么她离最近的天使玩偶可能埋下的地方有多远。

因为 Ayu 只会沿着平行坐标轴的方向来行动，所以在这个问题里我们定义两个点之间的距离为曼哈顿距离 $\mathrm{dist}(A,B)=|A_x-B_x|+|A_y-B_y|$。其中 A_x 表示点 A 的横坐标，其余类似。

输入数据的第一行包含两个整数 n 和 m，表示在刚开始时，Ayu 已经知道有 n 个点可能埋着天使玩偶，接下来 Ayu 要进行 m 次操作。

接下来 n 行，每行两个非负整数 x_i,y_i，表示初始 n 个点的坐标。

再接下来 m 行，每行三个非负整数 t,x,y。

如果 $t=1$，则表示 Ayu 又回忆起了一个可能埋着玩偶的点 (x,y)。

如果 $t=2$，则表示 Ayu 询问如果她在坐标 (x,y)，那么在已经回忆出的点里，离她最近的那个点有多远。

数据范围为 $n,m\le 5*10^5$，坐标范围为 $0\sim10^6$。

先来解决该问题的简化版——假设没有 $t=1$ 的操作。这时，题目变为：平面上有 n 个点 (x_i,y_i)，执行 m 次查询，每次询问与给定坐标 (x,y) 最近的点有多远。根据题意，答案为：

$$\min_{1\le i\le n}\{|x-x_i|+|y-y_i|\}$$

为了去掉绝对值符号，我们不妨把原来的询问分成四个，分别询问在 (x,y) 的左下、左上、右上、右下方向上距离最近的点有多远，四个结果取最小值即为答案。

以左下方向为例，此时要求的式子变为：

$$\min_{1\le i\le n}\{(x-x_i)+(y-y_i)\}$$

进一步简化为：

$$(x+y)-\max_{1\le i\le n}\{x_i+y_i\},\ \text{其中}\ x_i\le x,\ y_i\le y$$

我们可以把 n 个点的坐标和 m 个询问的坐标一起按照横坐标从小到大排序，然后依次进行扫描。

1. 若扫描到一个点 (x_i,y_i)，则在树状数组（或线段树）中把第 y_i 个位置上的值与 x_i+y_i 取 max，同时维护前缀（或区间）最大值。

2. 若扫描到一个询问 (x,y)，则在树状数组（或线段树）中查询 $[0,y]$ 上的最大值 val。该询问（左下方向）的答案就是 $x+y-val$。

在上面的做法中，排序满足了 $x_i\le x$ 的条件，树状数组控制了 $y_i\le y$ 的条件并维护了 x_i+y_i 的最大值。

对于左上、右上和右下三个方向，做法类似。例如左上方向，仍按照横坐标从小到大排序，树状数组改为维护 $x_i - y_i$ 的最大值，修改时的位置变为 $10^6 - y_i$，查询时的前缀变为 $[0, 10^6 - y]$ 即可。剩下的两个方向请读者自行推导。

时间复杂度为 $O\big((n+m)\log(n+m)\big)$。

下面我们回到“天使玩偶”一题，即带有 $t=1$ 操作的情况。

我们可以把初始的 n 个点看作 n 次加入新点的操作，这样题目就变为：有一个长度为 $n+m$ 的操作序列，每项操作要么是在平面上新增一个点，要么是查询与给出的坐标最近的点有多远。

对整个操作序列应用基于时间的离线分治算法（CDQ 分治），我们还需解决的是每个 solve(l,r) 的第 3 部分，即把第 l~mid 项操作中新增的点依次加入一个初始为空的平面，再考虑第 $mid+1 \sim r$ 项操作中的查询，找到平面上距离最近的点，更新答案。这是一个静态的问题，只需把第 l~mid 项操作中新增的点作为初始点，运用刚才讨论的“简化版”问题的解法即可。

注意，为了保证时间复杂度仅与 $r-l$ 相关，不能每次建立一个新的树状数组，必须在计算每个 solve(l,r) 后，把对树状数组的修改依次撤销，保证在每次进入和离开 solve 函数时，树状数组处于空的状态，能被重复利用。整个算法的时间复杂度为 $O\big((n+m)\log^2(n+m)\big)$。

CDQ 分治算法能够简化问题的关键之处在于能够对操作序列按照“时间”分治，让转化后的静态问题不必考虑“修改”和“查询”的时间顺序，使按照其他信息（例如横坐标）排序成为可能，降低了数据结构需要控制的限制条件的维度。

一种称为 KD-Tree 的数据结构可以在线维护本题，时间复杂度为 $O((n+m)\sqrt{n})$。学有余力的读者可以自行查阅相关资料。配套资源中提供了 CDQ 分治和 KD-Tree 两种解法的参考程序。

基于值域的整体分治算法

试考虑这样一个问题：

给定一个长度为 N 的整数序列 A 和一个固定的整数 S，执行 M 次操作，每次操作输入三个整数，格式为以下两种之一。

1. “$0\ l\ r$”，代表一次查询操作，求序列 A 的第 l 到 r 个数中不大于 S 的数（即满足 $l \le i \le r$ 且 $A_i \le S$ 的数）有多少个。

2. “$1\ x\ y$”，代表一次修改操作，把 A 的第 x 个数改为 y（即执行 $A[x]=y$）。

要求对于每次查询，输出正确的结果。



这个问题很容易用树状数组解决，我们只需要在树状数组中记录不大于 S 的整数的变化。建立一个空的树状数组，对于任意 $A[i] \le S$，在树状数组中执行 $\text{add}(i,1)$，表示在第 i 个位置上有一个不大于 S 的数。此时树状数组就维护了序列 A 的每个前缀 $A[1\sim i]$ 中不大于 S 的数的个数。

对于每次查询，在树状数组中执行 $\text{ask}(r)-\text{ask}(l-1)$ 即可得到答案。

对于每次修改，等价于在第 x 个位置上去掉了一个值为 $A[x]$ 的数，增加了一个值为 y 的数。若 $A[x] \le S$，则需执行 $\text{add}(x,-1)$，表示修改以后，不大于 S 的数在第 x 个位置上减少了一个。若 $y \le S$，则再执行 $\text{add}(x,1)$，表示修改以后，不大于 S 的数在第 x 个位置上增加了一个。当然，还要赋值 $A[x]=y$，正确维护序列 A。

再考虑一个这样的问题：

给定一个长度为 N 的整数序列 A，求序列 A 中第 K 小的数是多少。

这个问题的做法非常多。最直观的做法是 $O(N\log N)$ 排序后输出第 K 个数。采用 0x05 节介绍的每次只递归一半的快速排序框架，可以做到 $O(N)$。当然，以下做法也是可行的。

二分答案，设当前二分的值为 mid，统计序列中有多少个数 $\le mid$，记为 cnt。

1. 若 $K \le cnt$，则说明第 K 小数的值一定 $\in [l,mid]$，可在左半区间继续二分。

2. 若 $K > cnt$，则说明第 K 小数的值一定 $\in [mid+1,r]$，等价于在值域 $[mid+1,r]$ 限制下寻找第 $K-cnt$ 小的数（序列 A 落在左半区间的 cnt 个数都比所求答案小，全部排在所求答案之前），可令 $K-=cnt$ 并在右半区间继续二分。

时间复杂度为 $O(N\log SIZE)$，其中 $SIZE$ 为值域的大小。

倘若有 M 个如“求序列 A 的第 l 到 r 个数中第 k 小的数”的询问，我们当然可以对每个询问都按照上述做法二分求解，时间复杂度为 $O(MN\log SIZE)$。实际上，每次对序列进行统计时，我们都能获得一些对所有询问有价值的信息，却只用在了一次询问上，造成了许多冗余计算。在下面的例题中，我们将综合刚才两道题目的解法，在对值域进行二分的同时，对整个“操作序列”也进行分治计算。

例题 K-th Number AcWing255

给定一个长度为 N 的整数序列 A，执行 M 次操作，其中第 i 次操作给出三个整数 l_i,r_i,k_i，求 $A[l_i],A[l_i+1],\cdots,A[r_i]$（即 A 的下标区间 $[l_i,r_i]$）中第 k_i 小的数是多少。$N \le 10^5, M \le 10^4, |A[i]| \le 10^9$。

设 A 中最小的数为 $MINA$，最大的数为 $MAXA$，我们仍然尝试在值域 $[MINA,MAXA]$ 上进行二分答案，设第一次二分的值为 mid。

在二分的过程内部，扫描每个询问 $i=1\sim M$，统计在下标区间 $[l_i,r_i]$ 中不大于

mid 的数有多少个，记为 c_i。因为对于每个 $[l_i,r_i]$ 来说，mid 是一个固定的值，所以可直接使用我们上一页讨论过的第一道题目的解法。

然后，我们套用上一页讨论过的第二道题目的解法，对这些询问进行分类。

1. 若 $k_i \le c_i$，则说明第 i 个询问的答案在 $[MINA,mid]$ 中。

2. 若 $k_i > c_i$，则说明第 i 个询问的答案在 $[mid+1,MAXA]$ 中，并且等价于在值域 $[mid+1,MAXA]$ 的限制下寻找第 $k_i - c_i$ 小的数。

我们把序列 A 中的数也进行分类，不大于 mid 的数构成一个子序列 LA，大于 mid 的数构成一个子序列 RA。

考虑值域 $[MINA,mid]$，序列 LA 和第 1 类询问（$k_i \le c_i$）。容易发现，它们构成一个子问题，序列从 A 变为 LA，值域上界减小，询问的数量也变少了。

考虑值域 $[mid+1,MAXA]$、序列 RA 和第 2 类询问（$k_i > c_i$）容易发现，它们也构成一个子问题，并且令 $k_i -= c_i$ 后，该问题与 LA 上的子问题互不相关。

综上所述，我们得到一个分治算法。设 solve(L,R,a,q) 表示有一个值域为 $[L,R]$ 的整数序列 a，对询问序列 q（q 中存储若干个三元组 l_i,r_i,k_i，表示求 a 的下标区间 $[l_i,r_i]$ 中的第 k_i 小的数）当中的每个询问做出回答。

1. 设 $mid = (L+R) \gg 1$。

2. 利用树状数组，对于 q 中的每个询问，统计在序列 a 的下标区间 $[l_i,r_i]$ 中不大于 mid 的数有多少个，记为 c_i。

3. 若 $k_i \le c_i$，则把该询问加入到序列 lq 中。否则，令 $k_i -= c_i$，将其加入到序列 rq 中。

4. 令序列 a 中 $\le mid$ 的数构成序列 la，$> mid$ 的数构成序列 ra。

5. 递归求解 solve(L,mid,la,lq) 和 solve$(mid+1,R,ra,rq)$。

递归的边界为：当 q 为空时，直接返回。当 $L=R$ 时，直接把 L 作为 q 中每个询问的答案。读者可以利用样例数据在纸上模拟，理解上述过程。

与 CDQ 分治一样，还应该在 solve 的第 5 步之前撤销对树状数组的所有修改，不能每次建立一个新的树状数组。另外，在程序实现中，为了避免空间浪费，可以直接把 la,ra,lq,rq 拷贝回原来的数组 a,q，在 solve 函数的参数中用两个下标表示有效范围。还可以把初始序列 A 中的 N 个整数看作 N 次“赋值”操作，合并 a 和 q 两个序列，进一步简化代码。因为分治至多进行 $\log SIZE$ 层（$SIZE$ 为值域的大小），当操作序列 q 为空时会立即返回，再算上树状数组的开销，所以整个算法的时间复杂度为 $O((N+M)\log SIZE \log N)$。如果加上离散化，可以做到 $O((N+M)\log^2 N)$。

```
const int N = 100010, INF = 1e9;
struct rec {int op, x, y, z;} q[2 * N], lq[2 * N], rq[2 * N];
int n, m, t, c[N], ans[N];
```



```
int ask(int x) {
    int y = 0;
    for (; x; x -= x & -x) y += c[x];
    return y;
}
void change(int x, int y) {
    for (; x <= n; x += x & -x) c[x] += y;
}

void solve(int lval, int rval, int st, int ed) {
    if (st > ed) return; // 操作序列为空
    if (lval == rval) { // 值域到达边界
        for (int i = st; i <= ed; i++)
            if (q[i].op > 0) ans[q[i].op] = lval;
        return;
    }
    int mid = (lval + rval) >> 1;
    int lt = 0, rt = 0;
    for (int i = st; i <= ed; i++) {
        if (q[i].op == 0) { // 是一次赋值操作
            if (q[i].y <= mid) change(q[i].x, 1), lq[++lt] = q[i];
            else rq[++rt] = q[i];
        } else { // 是一次询问
            int cnt = ask(q[i].y) - ask(q[i].x - 1);
            if (cnt >= q[i].z) lq[++lt] = q[i];
            else q[i].z -= cnt, rq[++rt] = q[i];
        }
    }
    for (int i = ed; i >= st; i--) { // 还原树状数组
        if (q[i].op == 0 && q[i].y <= mid) change(q[i].x, -1);
    }
    for (int i = 1; i <= lt; i++) q[st + i - 1] = lq[i];
    for (int i = 1; i <= rt; i++) q[st + lt + i - 1] = rq[i];
    solve(lval, mid, st, st + lt - 1);
    solve(mid + 1, rval, st + lt, ed);
}
```

```
int main() {
    cin >> n >> m;
    for (int i = 1; i <= n; i++) {
        int val; scanf("%d", &val);
        // a[i]=val 等价于一次把第 i 个数赋值为 val 的操作
        q[++t].op = 0, q[t].x = i, q[t].y = val;
    }
    for (int i = 1; i <= m; i++) {
        int l, r, k; scanf("%d%d%d", &l, &r, &k);
        // 记录一次询问
        q[++t].op = i, q[t].x = l, q[t].y = r, q[t].z = k;
    }
    // 基于值域对 t=n+m 个操作进行整体分治
    solve(-INF, INF, 1, t);
    for (int i = 1; i <= m; i++) printf("%d\n", ans[i]);
}
```

最后，我们来讨论本题的扩展—— M 次操作中既有区间第 k 小数的查询，还有“把 $A[x]$ 变为 y”的修改。

与本小节讨论的第一道题目一样，我们先不考虑查询，在序列 A 上模拟一遍所有修改操作，同时把每次修改拆成“在第 x 个位置上去掉了一个值为 $A[x]$ 的数”与“增加了一个值为 y 的数”两次操作。序列 A 最初的 N 个数可以看作最早的 N 次“在第 i 个位置上增加了一个值 $A[i]$”的操作。我们把这些操作和所有“查询”一起，按照时间顺序组成一个操作序列，并把“去掉一个值”的修改操作标记为 -1，“增加一个值”的修改操作标记为 0，查询操作标记为它是第几次查询（一个正整数）。该标记用于区分操作的类型。随后，我们基于值域对操作序列进行整体分治，即上面程序中的 $\text{solve}(-INF, INF, 1, t)$，其中 t 表示新操作序列的总长度。

设 $\text{solve}(L, R, st, ed)$ 表示有一个值域为 $[L, R]$ 且初始为空的整数序列，执行操作序列的第 st~ed 项操作，并对其中的询问计算结果。

1. 设 $mid = (L + R) \gg 1$。

2. 利用树状数组，对于第 st~ed 项操作，执行与本小节讨论的第一道题目类似的做法（该做法已经能处理修改），统计出对于其中的询问，$A[l_i$~$r_i]$ 中不大于 mid 的数有多少个，记为 c_i。

3. 把涉及对 $\leq mid$ 的数值进行修改的操作，以及满足 $k_i \leq c_i$ 的询问加入到序列 lq 中。把涉及对 $> mid$ 的数值进行修改的操作，以及满足 $k_i > c_i$ 的执行 $k_i -= c_i$ 后的询问加入到序列 rq 中。注意保持操作的时间顺序。

4. 把 lq 和 rq 拷贝回原操作序列的 st~ed 位置，设 lq 的长度为 lt。



5. 还原树状数组（再次扫描操作序列，在树状数组上撤销修改）以便重复利用。

6. 递归求解 solve$(L, mid, st, st + lt - 1)$ 和 solve$(mid + 1, R, st + lt, ed)$。

新的算法只是增加了对“修改”操作的划分方式，使得每次二分后，值域和操作序列都能分成互相独立的两段，构成子问题。时间复杂度为 $O((N + M)\log^2 N)$。读者可以 Dynamic Rankings (AcWing1782)作为模板题，并参阅配套资源中的程序。

这类离线分治算法基于值域（答案域）对整个操作序列进行分治，使操作序列在保持时间顺序的基础上能够同值域一起划分为独立的子问题，故被称为**基于值域的整体分治算法**，简称**整体分治**。

此外，本题还有归并树、可持久化线段树、各类“树套树”等多种解法，我们将在下一节继续研究。在本章习题中，我们还要求读者应用 CDQ 分治和整体分治算法解决另外两道题目。

0x48 可持久化数据结构

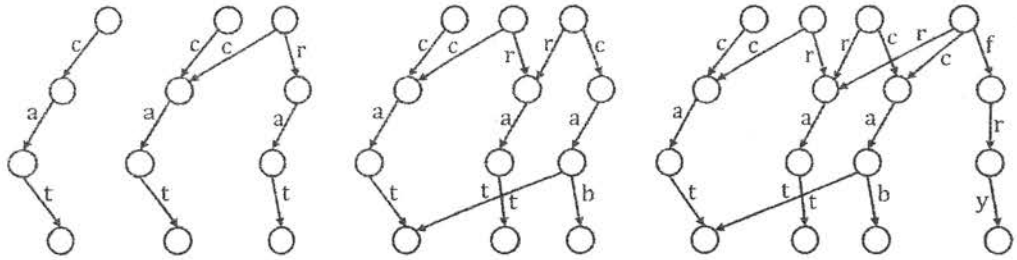
到目前为止，我们学习的数据结构维护的都是“数据集的最新状态”。若想知道数据集在任意时间的历史状态（即 $\forall i \in [1, M]$，执行完操作序列中第 i 项操作后数据集的状态），一种朴素的做法是 $\forall i \in [1, M]$，在第 i 项操作结束后，把整个数据结构拷贝一遍，存储在 $history[i]$ 中，多耗费 M 倍的空间。“可持久化”提供了一种思想，在每项操作结束后，仅创建数据结构中发生改变的部分的副本，不拷贝其他部分。这样一来，维护数据结构的时间复杂度没有增加，空间复杂度仅增长为与时间同级的规模。换言之，可持久化数据结构能够高效地记录一个数据结构的所有历史状态。

可持久化 Trie

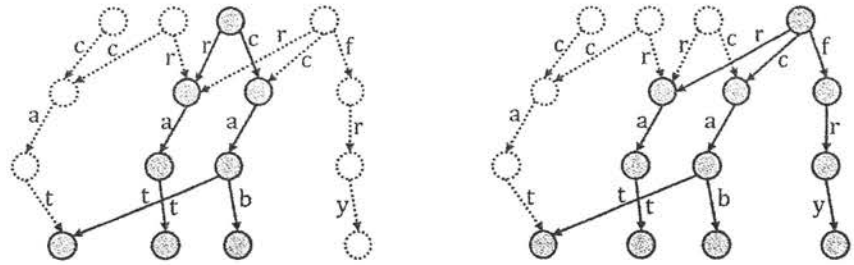
与 Trie 的节点一样，可持久化 Trie 的每个节点也有若干字符指针指向子节点，可以用 $trie[x, c]$ 保存节点 x 的字符指针 c 指向的子节点的编号（0 代表指向空）。可持久化 Trie 按照以下步骤插入一个新的字符串 s。

1. 设当前可持久化 Trie 的根节点为 $root$，令 $p = root$, $i = 0$。
2. 建立一个新的节点 q，令 $root' = q$。
3. 若 $p \neq 0$，则对于每种字符 c，令 $trie[q, c] = trie[p, c]$。
4. 建立一个新的节点 q'，令 $trie[q, s_i] = q'$。换言之，除了字符指针 s_i 不同之外，节点 q 与节点 p 的其他信息完全相同。
5. 令 $p = trie[p, s_i]$, $q = trie[q, s_i]$, $i = i + 1$。
6. 重复步骤 3~5，直至 i 到达字符串末尾。

从 $root$ 出发沿着字符指针向下能够访问到的节点构成插入 s 之前的 Trie，而从 $root'$ 出发沿着字符指针向下能够访问到的节点构成插入 s 之后的 Trie。下页图展示了在可持久化 Trie 中依次插入字符串 cat, rat, cab, fry 的过程。



可以看出，可持久化 Trie 的本质是一张有向图。我们把每次插入字符串后得到的新的根节点记录在数组 $root[1\sim4]$ 中，那么在可持久化 Trie 中从 $root[i]$ 出发能够访问到的点和字符指针就组成了由前 i 个字符串构成的 Trie。如下图所示。



构建可持久化 Trie 所需的空间和时间复杂度都是字符串的总长度的线性函数。

例题 最大异或和 AcWing256

给定一个非负整数序列 a，初始长度为 N。有 M 个操作，每个操作为以下两种类型之一。

1. “A x”，添加操作，表示在序列末尾插入一个数 x，序列的长度 N 增大 1。

2. “Q l r x”，询问操作，求一个位置 p，满足 $l\le p\le r$，使得 $a[p]$ xor $a[p+1]$ xor $\cdots$ xor $a[N]$ xor x 最大，输出这个最大值。

数据范围：$N,M\le 3*10^5$，$0\le a[i]\le 10^7$。

设 $s[i]$ 表示序列 a 的前 i 个数 xor 起来得到的结果。特别地，设 $s[0]=0$。根据异或运算的性质：

$$a[p] \text{ xor } a[p+1] \text{ xor } \cdots \text{ xor } a[N] \text{ xor } x = s[p-1] \text{ xor } s[N] \text{ xor } x$$

对于“添加操作”，序列 s 很容易维护。对于“询问操作”，问题变为：已知一个整数 $val=s[N]$ xor x，求一个位置 p，满足 $l-1\le p\le r-1$，使得 $s[p]$ xor val 最大。

如果不考虑 $l-1$ 和 $r-1$ 的限制，那么就是初始有 $N+1$ 个数 $s[0]\sim s[N]$，每次操作可以“在末尾添加一个数 $s[N+1]=s[N]$ xor x 并令 N 增大 1”或者“询问序列 s 中的哪个数与 val 异或的结果最大”。我们在 0x16 节“Trie”的例题“The XOR

Largest Pair”中已经讨论过类似问题的解法。只需把序列 s 的每个数看作二进制数，以字符串的形式插入 Trie，询问时在 Trie 中优先尝试访问与 val 的每一位相反的指针即可，请读者回顾。

如果只有 $p \le r-1$ 的限制，那么只需用可持久化 Trie 代替上述解法中的 Trie。因为可持久化 Trie 保存了 $\forall i \in [0,N]$，$s[0\sim i]$ 这 $i+1$ 个二进制数构成的 Trie，所以每次询问从 $root[r-1]$ 出发，优先尝试访问与 val 的每一位相反的指针即可。

回到原题，为了要满足 $l-1 \le p \le r-1$，我们可以在可持久化 Trie 的每个节点上增加一些信息。设 $end[x]$ 表示可持久化 Trie 的节点 x 是序列 s 中第几个二进制数的末尾节点（若不是末尾节点，$end[x]=-1$）。设 $latest[x]$ 表示在可持久化 Trie 以 x 为根的子树中 end 的最大值。从 $root[r-1]$ 出发检索 val 时，仅考虑 $latest$ 值不小于 $l-1$ 的节点即可。整个算法的时间复杂度为 $O((N+M)\log 10^7)$。

```
const int N = 600010;
int trie[N*24][2], latest[N*24]; // latest 和 end 可合并为一个数组
int s[N], root[N], n, m, tot;
// 本题需要统计子树 latest，故使用递归插入 s[i]，当前为 s[i]的第 k 位
void insert(int i, int k, int p, int q) {
    if (k < 0) {
        latest[q] = i;
        return;
    }
    int c = s[i] >> k & 1;
    if (p) trie[q][c ^ 1] = trie[p][c ^ 1];
    trie[q][c] = ++tot;
    insert(i, k - 1, trie[p][c], trie[q][c]);
    latest[q] = max(latest[trie[q][0]], latest[trie[q][1]]);
}

int ask(int now, int val, int k, int limit) {
    if (k < 0) return s[latest[now]] ^ val;
    int c = val >> k & 1;
    if (latest[trie[now][c ^ 1]] >= limit)
        return ask(trie[now][c ^ 1], val, k - 1, limit);
    else
        return ask(trie[now][c], val, k - 1, limit);
}

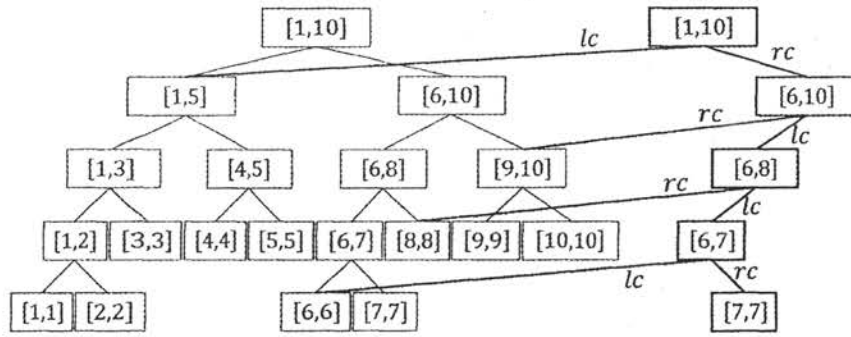
int main() {
```

```
    cin >> n >> m;
    latest[0] = -1;
    root[0] = ++tot;
    insert(0, 23, 0, root[0]);
    for (int i = 1; i <= n; i++) {
        int x; scanf("%d", &x);
        s[i] = s[i - 1] ^ x;
        root[i] = ++tot;
        insert(i, 23, root[i - 1], root[i]);
    }
    for (int i = 1; i <= m; i++) {
        char op[2]; scanf("%s", op);
        if (op[0] == 'A') {
            int x; scanf("%d", &x);
            root[++n] = ++tot;
            s[n] = s[n - 1] ^ x;
            insert(n, 23, root[n - 1], root[n]);
        }
        else {
            int l, r, x; scanf("%d%d%d", &l, &r, &x);
            printf("%d\n", ask(root[r - 1], x ^ s[n], 23, l - 1));
        }
    }
}
```

可持久化线段树

基于可持久化 Trie 的思想，我们很容易设计出多种数据结构的可持久化版本——前提是数据结构的“内部结构”在操作过程中不发生变化（例如平衡树的旋转就改变了平衡树的“内部结构”）。**可持久化线段树**，又称函数式线段树，是最重要的可持久化数据结构之一。

仍以 0x43 节的区间最大值问题为例，线段树的“单点修改”只会使 $O(\log N)$ 个区间对应的节点发生更新。对于每个被更新的节点 p，我们创建该节点的副本 p'。只要 p 不是叶子节点，p 的左、右子树之一必然发生了更新。不妨设发生更新的是 p 的左子树 lc，我们在 lc 中递归，返回时令 p' 的左子树为 lc'，令 p' 的右子树与 p 的右子树相同，然后更新节点 p' 上保存的区间最大值即可。下页图展示了在一棵线段树中对位置 7 进行“单点修改”后，产生的 $O(\log N)$ 个新节点。图中省略了区间最大值的信息。



因为可持久化线段树不再是一棵完全二叉树，所以我们不能再用层次序编号，而是改为直接记录每个节点的左、右子节点编号，与 Trie 的字符指针类似。因为每次修改都会创建 $O(\log N)$ 个新节点，所以可持久化线段树的空间复杂度为 $O(N + M \log N)$。为了节省空间，我们不再记录每个节点代表的区间 $[l, r]$，而是作为递归参数传递。下面的代码定义并在初始序列 a 上创建了一棵可持久化线段树（目前为止，它还只是一棵普通的线段树）。

```
struct SegmentTree {
    int lc, rc; // 左右子节点编号
    int dat; // 区间最大值
} tree[MAX_MLOGN];
int tot, root[MAX_M]; // 可持久化线段树的总点数和每个根
int n, a[MAX_N];

int build(int l, int r) {
    int p = ++tot;
    if (l == r) { tree[p].dat = a[l]; return p; }
    int mid = (l + r) >> 1;
    tree[p].lc = build(l, mid);
    tree[p].rc = build(mid + 1, r);
    tree[p].dat = max(tree[tree[p].lc].dat, tree[tree[p].rc].dat);
    return p;
}
// 在 main 函数中
root[0] = build(1, n);
```

对于第 i 次修改，以可持久化线段树的第 $i-1$ 个版本为基础，执行如上图所示的步骤。下面的代码实现了可持久化线段树的单点修改 $a[x] = val$。



```
int insert(int now, int l, int r, int x, int val) {
    int p = ++tot;
    tree[p] = tree[now];
    if (l == r) {
        tree[p].dat = val;
        return p;
    }
    int mid = (l + r) >> 1;
    if (x <= mid)
        tree[p].lc = insert(tree[now].lc, l, mid, x, val);
    else
        tree[p].rc = insert(tree[now].rc, mid + 1, r, x, val);
    tree[p].dat = max(tree[tree[p].lc].dat, tree[tree[p].rc].dat)
    return p;
}
// 在 main 函数中
root[i] = insert(root[i - 1], 1, n, x, val);
```

容易发现，可持久化线段树维护了每次操作之后线段树的历史形态。$\forall i \in [1,M]$，从 $root[i]$ 出发向下能够访问到的节点就构成了执行完前 i 次修改操作时，维护 a 序列区间最大值的线段树。可持久化线段树的查询操作与一般线段树类似，这里就不再赘述。

可持久化线段树难以支持大部分“区间修改”。当一个节点下传延迟标记时，一旦我们创建它左右子节点 lc,rc 的副本 lc',rc' 并把标记更新，那么所有依赖 lc,rc 的“线段树版本”都要改为依赖 lc',rc'，甚至还要自底向上重新计算某些信息。这样的后果是灾难性的。在一些特殊的题目中，可以使用“标记永久化”代替标记的下传，但应用的局限性很大。可持久化线段树的“区间修改”不是我们讨论的重点，学有余力的读者可以尝试用可持久化线段树配合“标记永久化”，完成 HDOJ4348 To the Moon。

例题 **K-th Number** AcWing255

给定一个长度为 N 的整数序列 A，执行 M 次操作，其中第 i 次操作给出三个整数 l_i,r_i,k_i，求 $A[l_i],A[l_i+1],\cdots,A[r_i]$（即 A 的下标区间 $[l_i,r_i]$）中第 k_i 小的数是多少。$N \le 10^5, M \le 10^4, |A[i]| \le 10^9$。

在 0x47 节“离线分治算法”中，我们讨论了经典的“区间第 k 小数”问题的整体分治算法，其时间复杂度为 $O\big((N+M)\log^2 N\big)$，空间复杂度为 $O(N+M)$。本节我们再来介绍三种在线算法，并把重点放在可持久化线段树的解法上，其他两种做法只作

简单叙述。

解法一：归并树

在 $[1,N]$ 上建立一棵线段树，区间 $[l,r]$ 对应的节点上保存一个长度为 $r-l+1$ 的数组，存储 $A_l,A_{l+1},\cdots,A_r$ 排序后的结果。我们只需一起执行线段树建树和归并排序的过程，即可实现上述结构的构造。这也是“归并树”名称的由来。

对于每次询问 l_i,r_i,k_i，区间 $[l_i,r_i]$ 在线段树上对应 $O(\log N)$ 个节点。这些节点保存的 $O(\log N)$ 个有序数组共同组成 A 序列的第 l_i~r_i 个数。我们在 0x47 节“离线分治算法”的“基于值域的整体分治算法”小节中讨论过“求序列第 K 小的数”的二分答案做法。对于本题的每个询问，我们也二分答案 mid，然后在 $O(\log N)$ 个有序数组中分别二分查找最后一个 $\le mid$ 的数所处的下标 p。$\sum p$ 即为 $A[l_i$~$r_i]$ 中 $\le mid$ 的数的个数，与 k_i 比较就可以确定二分答案上下界的变化情况。

该算法的时间复杂度为 $O(N\log N+M\log^3 N)$，空间复杂度为 $O(N\log N)$，且不能扩展到支持“修改”操作的动态“区间第 k 小数”问题。

解法二：可持久化线段树

归并树解法的时间复杂度较高，其原因在于线段树没有维护“值域”的划分信息，需要经过两重二分才能与 k_i 执行比较，这启发我们在“值域”而不是“下标”上建立线段树。

暂时不考虑 l_i,r_i。如果能够在线段树上维护“序列 A 有多少个数落在值域区间 $[L,R]$ 内”（记为 $cnt_{L,R}$），那么只需比较 $cnt_{L,mid}$ 与 k_i 的大小关系，即可确定 A 的第 k_i 小数是 $\le mid$ 还是 $>mid$，从而可以进入线段树的左、右子树之一。换言之，线段树对值域的划分代替了对答案的二分，请读者务必理解再继续阅读。在有 l_i,r_i 限制的情况下，我们可以使用可持久化线段树。

我们首先对 A 序列进行离散化（0x05 节），设离散化之后 $A[i]$ 的值为 $H[A[i]]\in[1,T]$。在区间 $[1,T]$ 上建立可持久化线段树，线段树的每个节点上保存一个值 cnt，表示该节点代表的值域区间 $[L,R]$ 中一共插入过多少个数，起初 cnt 值均为 0。

然后我们对每个 $A[i]$，在可持久化线段树上执行 $H[A[i]]$ 的“单点修改”，将其 cnt 加 1。线段树每个内部节点的 cnt 值等于左右子节点的 cnt 值之和。

此时，可持久化线段树中“以 $root[i]$ 为根的线段树”的值域区间 $[L,R]$ 就保存了 A 的前 i 个数有多少个落在值域 $[L,R]$ 内。

接下来考虑每个询问 l_i,r_i,k_i。有一条重要的性质：以 $root[l_i]$ 和 $root[r_i]$ 为根的两棵线段树对值域的划分是相同的。换言之，除了 cnt 值不同外，两棵线段树的内部结构和每个节点代表的值域区间完全对应。这意味着：“$root[r_i]$ 的值域区间 $[L,R]$ 的 cnt 值”减去“$root[l_i-1]$ 的值域区间 $[L,R]$ 的 cnt 值”就等于 $A[l_i$~$r_i]$ 中有多



少个数落在值域 $[L,R]$ 内，也就是可持久化线段树中两个代表相同值域的节点具有可减性。下面的代码同时从 $root[l_i-1]$ 和 $root[r_i]$ 出发执行 $O(\log N)$ 的访问，利用上述可减性计算出询问的答案。

```
// 在 p, q 两个节点上，值域为 [l,r]，求第 k 小数
int ask(int p, int q, int l, int r, int k) {
    if (l == r) return l; // 找到答案
    int mid = (l + r) >> 1;
    // 有多少个数落在值域 [l,mid] 内
    int lcnt = tree[tree[p].lc].cnt - tree[tree[q].lc].cnt;
    if (k <= lcnt) return ask(tree[p].lc, tree[q].lc, l, mid, k);
    else return ask(tree[p].rc, tree[q].rc, mid + 1, r, k - lcnt);
}
// 在 main 函数中，询问为(li,ri,ki)
int ans = ask(root[ri], root[li - 1], 1, t, ki);
```

该算法的时间复杂度为 $O\big((N+M)\log N\big)$，空间复杂度为 $O(N\log N)$。配套资源提供了完整的程序。若要扩展到支持“修改”操作的动态“区间第 k 小数”问题，则需在最外层套一个树状数组，本书不推荐这种做法（空间复杂度较高），故不再介绍。

解法三：线段树套平衡树

对序列 A 离散化之后，在值域 $[1,T]$ 上建立线段树，在线段树的每个节点 $[L,R]$ 上建立一棵平衡树（例如 Treap），保存序列 A 落在值域区间 $[L,R]$ 内的数的下标。因为平衡树本身支持查询“$\le x$ 的数有多少个”，所以我们也能知道序列 A 的前 x 个数有多少个落在值域区间 $[L,R]$ 内。

对于每个询问 l_i,r_i,k_i，与解法二开头讨论的方式一样，在线段树中进行访问。只需在线段树左子节点 $[L,mid]$ 对应的平衡树中查询 $\le r_i$ 和 $\le l_i-1$ 的数的个数，二者相减与 k_i 比较，即可确定应该递归进入线段树的左子树还是右子树。

该算法的时间复杂度为 $O\big((N+M)\log^2 N\big)$，空间复杂度为 $O(N\log N)$。因为平衡树本身支持插入和删除，所以可以直接扩展到支持“修改”操作的动态“区间第 k 小数”问题。当把 $A[x]$ 赋值为 y 时，从包含 $A[x]$ 的值域区间对应的平衡树内删除下标 x，在包含 y 的值域区间对应的平衡树内插入下标 x 即可。树套树也是解决此类动态问题的最传统解法。

可持久化线段树、树套树和离线分治算法还有许多广泛的应用，例如：可以用可持久化线段树实现“可持久化数组”。因本书编写初衷和目标难度的限制，我们再三斟酌后，选择了经典的“区间第 k 小数”大致引入了这些高级数据结构，但离灵活运用还差很远。感兴趣的读者可以尝试完成本章标有*号的习题，并自己进行深入研究。

0x49 总结与练习

本章讲解了并查集、树状数组、线段树、二叉查找树、平衡树等数据结构，以及分块、点分治等思想，还探讨了基于时间或值域的离线分治算法，以及两种可持久化数据结构。这些数据结构大多是位运算、递归、分治、倍增等基本算法在树上或复杂序列上的进一步应用。除了熟练掌握这些数据结构的模板之外，读者更应该多加留意它们如何在各种实际情境下解决问题。

本章知识点

并查集

并查集的基本实现与应用，路径压缩

并查集对传递性关系的维护，扩展域并查集，边带权并查集

树状数组

支持单点增加、区间和查询的树状数组

支持区间增加、单点查询的树状数组

支持区间增加、区间和查询的树状数组

树状数组的应用，求逆序对、实时维护 01 序列中的第 k 个 1

线段树

支持单点修改、区间查询的线段树

线段树延迟标记，支持区间修改、区间查询

扫描线思想，线段树维护扫描线

动态开点的线段树

线段树的合并算法

分块

分块的“大段维护、局部朴素”思想

两种常见的分块形式，在线求区间众数

离线对询问进行分块的算法



点分治

点分治框架，以树的重心为根防止退化

两种常见的子树统计方法：树上直接统计、指针扫描数组

二叉查找树与平衡树初步

BST 的实现与基本操作

平衡树初步：单旋转的概念，Treap 的实现与基本操作，*Splay

离线分治算法

在线与离线算法、动态与静态问题的定义

基于时间的分治算法，把动态问题划分成静态子问题并用离线算法求解

基于值域的分治算法，按照涉及的值域实施对操作序列的整体划分求解

可持久化数据结构

可持久化 Trie

可持久化线段树，*线段树套平衡树



1. 完成本章中所有例题的代码实现（标有*的题目选做）

2. 关押罪犯

题目地址：AcWing257，提示：并查集/扩展域或边带权

3. Rochambeau

题目地址：AcWing258，提示：并查集/扩展域或边带权

4. True Liars

题目地址：AcWing259，提示：并查集+背包 (0x52 节)

5. Buy Tickets

题目地址：AcWing260，提示：树状数组

6. Hotel

题目地址：AcWing261，提示：线段树/延迟标记

7. Picture

题目地址：AcWing262，提示：扫描线

8. *作诗

题目地址：AcWing263，提示：分块

9. Race

题目地址：AcWing264，提示：点分治

10. 营业额统计

题目地址：AcWing265，提示：平衡树/Treap

11. *SuperMemo

题目地址：AcWing266，提示：平衡树/Splay

12. Mokia

题目地址：AcWing267，提示：基于时间的分治算法

13. *Meteors

题目地址：AcWing268，提示：基于值域的整体分治算法

14. *Fotile 模拟赛 L

题目地址：AcWing269，提示：分块+可持久化 Trie

15. *可持久化并查集加强版

题目地址：AcWing270，提示：可持久化线段树+并查集按秩合并

0x50 动态规划

通过第 0x00 章“基本算法”和第 0x20 章“搜索”的学习，读者应该已经对“问题与状态空间”之间的类比有了更深入的认识。我们提及了递归和递推两种遍历状态空间的基本方法，以及各种降低状态空间规模的手段。对于指数级别等非多项式复杂度的问题，我们介绍了搜索算法，以及提高搜索效率的各种剪枝。本章的主要内容是动态规划算法（DP，Dynamic Programming），它针对满足特定条件的一类问题，对各状态维度进行分阶段、有顺序、无重复、决策性的遍历求解。

本章有20小时详细视频讲解，读者可以扫描右侧二维码或访问 https://www.acwing.com/activity/content/35/ 了解相关信息。



我们之前探讨过的很多算法都是利用问题的可划分性以及子问题之间的相似性来进行归纳，降低求解的复杂度，动态规划也不例外。动态规划算法把原问题视作若干个重叠子问题的逐层递进，每个子问题的求解过程都构成一个“**阶段**”。在完成前一个阶段的计算后，动态规划才会执行下一阶段的计算。

为了保证这些计算能够按顺序、不重复地进行，动态规划要求已经求解的子问题不受后续阶段的影响。这个条件也被叫作“**无后效性**”。换言之，**动态规划对状态空间的遍历构成一张有向无环图，遍历顺序就是该有向无环图的一个拓扑序**。有向无环图中的节点对应问题中的“状态”，图中的边则对应状态之间的“转移”，转移的选取就是动态规划中的“决策”。

在很多情况下，动态规划用于求解最优化问题。此时，下一阶段的最优解应该能够由前面各阶段子问题的最优解导出。这个条件被称为“**最优子结构性质**”。当然，这是一种比较片面的说法，它其实告诉我们，动态规划在阶段计算完成时，只会在每个状态上保留与最终解集相关的部分代表信息，这些代表信息应该具有可重复的求解过程，并能够导出后续阶段的代表信息。这样一来，动态规划对状态的抽象和子问题的重叠递进才能够起到优化作用。

“**状态**”“**阶段**”和“**决策**”是构成动态规划算法的三要素，而“子问题重叠性”“无后效性”和“最优子结构性质”是问题能用动态规划求解的三个基本条件。

动态规划算法把相同的计算过程作用于各阶段的同类子问题，就好像把一个固定的公式在格式相同的若干输入数据上运行。因此，我们一般只需要定义出 DP 的计算过程，就可以编程实现了。这个计算过程就被称为“**状态转移方程**”。在大多数例题中，我们讨论状态转移方程的设计思路，并给出最终的方程、边界和目标，而不再讨论程序的实现细节。

0x51~0x53 节主要侧重最常见的 DP，也就是具有线性阶段划分的 DP 问题。在 0x51 节中，我们先介绍最基本的线性 DP，帮助读者逐步理解动态规划思想，掌握设计 DP 算法基本框架的几条准则。另外，我们在 0x52 节与 0x53 节中会单独讲解背包与区间 DP 这两个重要的部分。我们将探讨背包的各种变形与扩展，在区间 DP 中还会结合讨论“递推”与“记忆化搜索”这两种 DP 基本实现形式。

0x54~0x55 节把视野扩展到更加一般化的问题。我们将在 0x54 节介绍各种树上的动态规划算法，并在 0x55 节中研究使用 DP 遇到环形、分阶段带环或者具有局部后效性的问题时常见的处理方法。

0x56~0x5B 节包括动态规划的各类优化手段。从状态入手，我们有状态压缩、倍增等优化算法。从转移入手，我们可以使用数据结构进行一般的优化，或者挖掘决策单调性，实施单调队列、斜率优化、四边形不等式等比较高级的优化策略。

0x5C~0x5D 节与数学内容有所交叉，我们将简单探讨计数类 DP 和数位统计 DP 的

一些例题，进一步扩充读者的知识储备。

动态规划的艺术在于状态设计和子结构的挖掘。多年来，学界探讨了诸多 DP 转移的方法和优化手段，然而如何把问题形式化为状态空间，进一步抽象出 DP 的状态表示和阶段划分，往往是一件考查智力而非套路的事情。本章努力尝试把读者领进大门，但 DP 的这一难点仍然需要读者自己多加思考。练习设计 DP 算法并熟练使用它求解问题，也是提升读者在算法竞赛领域思维水平的一个重要途径。

0x51 线性 DP

在本书中，具有线性“阶段”划分的动态规划算法被统称为线性 DP[①]。读者对最长上升子序列（LIS）、最长公共子序列（LCS）、数字三角形（IOI1994）等 DP 的经典入门例题应该并不陌生，在动态规划的入门书籍或者网络上很容易找到这些问题的描述和解答。本书不会花费时间再次详细解释这些问题，但本着见微知著的思想，我们可以简单回顾它们，并尝试分析一下 DP 要素在其中的体现。

LIS 问题	问题描述	最长上升子序列。给定一个长度为 N 的数列 A，求数值单调递增的子序列的长度最长是多少。A 的任意子序列 B 可表示为 $B=\{A_{k_1},A_{k_2},\cdots,A_{k_p}\}$，其中 $k_1<k_2<\cdots<k_p$
	状态表示	$F[i]$ 表示以 $A[i]$ 为结尾的“最长上升子序列”的长度
	阶段划分	子序列的结尾位置（数列 A 中的位置，从前到后）
	转移方程	$F[i]=\max\limits_{0\le j<i,A[j]<A[i]}\{F[j]+1\}$
	边界	$F[0]=0$
	目标	$\max\limits_{1\le i\le N}\{F[i]\}$
LCS 问题	问题描述	最长公共子序列。给定两个长度分别为 N 和 M 的字符串 A 和 B，求既是 A 的子序列又是 B 的子序列的字符串的最长长度
	状态表示	$F[i,j]$ 表示前缀子串 $A[1\sim i]$ 与 $B[1\sim j]$ 的“最长公共子序列”的长度
	阶段划分	已处理的前缀长度（两个字符串中的位置，即一个二维坐标）
	转移方程	$F[i,j]=\max\begin{cases}F[i-1,j]\\F[i,j-1]\\F[i-1,j-1]+1 & \text{if } A[i]=B[j]\end{cases}$

① 线性 DP：线性动态规划，在本节中是一个非常广义的概念，不局限于“线性时间复杂度”的一维动态规划。与数学中的“线性空间”类似，如果一个动态规划算法的“状态”包含多个维度，但在每个维度上都具有“线性”变化的“阶段”，那么该动态规划算法同样称为“线性 DP”。

续表

LCS 问题	边界	$F[i,0]=F[0,j]=0$
	目标	$F[N,M]$
数字三角形	问题描述	给定一个共有 N 行的三角矩阵 A，其中第 i 行有 i 列。从左上角出发，每次可以向下方或右下方走一步，最终到达底部。求把经过的所有位置上的数加起来，和最大是多少
	状态表示	$F[i,j]$ 表示从左上角走到第 i 行第 j 列，和最大是多少
	阶段划分	路径的结尾位置（矩阵中的行、列位置，即一个二维坐标）
	转移方程	$F[i,j]=A[i,j]+\max\begin{cases}F[i-1,j]\\F[i-1,j-1] & \text{if } j>1\end{cases}$
	边界	$F[1,1]=A[1,1]$
	目标	$\max\limits_{1\le j\le N}\{F[N,j]\}$

容易发现，状态表示无论是一维还是多维，DP 算法在这些问题上都体现为“作用在线性空间上的递推”——DP 的阶段沿着各个维度线性增长，从一个或多个边界点开始有方向地向整个状态空间转移、扩展，最后每个状态上都保留了以自身为“目标”的子问题的最优解。

这几个问题也是线性 DP 中最简单的一类。在这类问题中，需要计算的对象表现出明显的维度以及有序性，每个状态的求解直接构成一个阶段，这使得 DP 的状态表示就是阶段的表示。因此，我们只需要在每个维度上各取一个坐标值作为 DP 的状态，自然就可以描绘出“已求解部分”在状态空间中的轮廓特征，该轮廓的进展就是阶段的推移。另外，每个状态的求解显然只与之前阶段的最优解有关，最优子结构性质也得以验证。接下来，我们按顺序依次循环每个维度，根据问题要求递推求解的具体实现过程也就顺理成章了。

例题 **Mr. Young's Picture Permutations** AcWing271

有 N 个学生合影，站成左端对齐的 k 排，每排分别有 $N_1,N_2,\cdots,N_k$ 个人，第 1 排站在最后边，第 k 排站在最前边。学生的身高互不相同，把他们从高到低依次标记为 $1,2,\cdots,N$。在合影时要求每一排从左到右身高递减，每一列从后到前身高也递减，问：一共有多少种安排合影位置的方案？$N\le 30,k\le 5$。

下面的一排三角矩阵给出了当 $N=6,k=3,N_1=3,N_2=2,N_3=1$ 时的全部 16 种合影方案。注意身高最高的是 1，最低的是 6。

```
123 123 124 124 125 125 126 126 134 134 135 135 136 136 145 146
45  46  35  36  34  36  34  35  25  26  24  26  24  25  26  25
6   5   6   5   6   4   5   4   6   5   6   4   5   4   3   3
```

因为在合法的合影方案中每行、每列的身高都是单调的，所以我们可以从高到低依次考虑标记为 $1,2,\cdots,N$ 的学生站的位置。这样一来，在任意时刻，已经安排好位置的学生在每一行中占据的一定是从左端开始的连续若干个位置，用一个 k 元组 $(a_1,a_2,\cdots,a_k)$ 表示每一行已安排的学生人数，即可描绘出“已经处理的部分”的轮廓。

当安排一名新的学生时，我们考虑所有满足如下条件的行号 i：

1. $a_i < N_i$。

2. $i=1$ 或 $a_{i-1} > a_i$。

只要该学生站在这样一行中，每列学生的身高单调性也就得以满足。换言之，我们不需要关心已经站好的 $(a_1+a_2+\cdots+a_k)$ 名学生的具体方案。k 元组 $(a_1,a_2,\cdots,a_k)$ 描绘的轮廓内的合影方案总数就足以构成一个子问题。因此，我们可以把 $a_1,a_2,\cdots,a_k$ 作为阶段，当安排一名新的学生时，$a_1,a_2,\cdots,a_k$ 其中之一会增加 1，从而转移到后续的阶段，符合各维度线性增长的形式。

简便起见，我们假设 $k=5$。当 $k<5$ 时，我们可以增加人数为 0 的排，使其等价于 $k=5$ 的问题。动态规划算法如下。

$F[a_1,a_2,a_3,a_4,a_5]$ 表示各排从左端起分别站了 a_1,a_2,a_3,a_4,a_5 个人时的合影方案数量。

边界：$F[0,0,0,0,0]=1$，其余均为 0。

目标：$F[N_1,N_2,N_3,N_4,N_5]$。

转移：若 $a_1<N_1$，则令 $F[a_1+1,a_2,a_3,a_4,a_5]+=F[a_1,a_2,a_3,a_4,a_5]$。若 $a_2<N_2$ 且 $a_1>a_2$，则令 $F[a_1,a_2+1,a_3,a_4,a_5]+=F[a_1,a_2,a_3,a_4,a_5]$。第 3~5 排同理。

本题还有一种数学解法：直接用“杨氏矩阵”和“勾长公式”进行计算。这里就不再赘述，感兴趣的读者可以查阅相关资料。

从该题给出的解法中我们发现，设计动态规划的状态转移方程，不一定要以“如何计算出一个状态”的形式给出，也可以考虑“一个已知状态应该更新哪些后续阶段的未知状态”。当然，对于本题来讲，两种方式没有什么差别。在有的题目中，其中一种可能会比另一种思考起来更加自然、简便。

例题 LCIS 最长公共上升子序列 AcWing272

熊大妈的奶牛在小沐沐的熏陶下开始研究信息题目。小沐沐先让奶牛研究了最长上升子序列，再让他们研究了最长公共子序列，现在又让他们研究最长公共上升子序列了。

小沐沐说，对于两个数列 A 和 B，如果它们都包含一段位置不一定连续的数，且数值是严格递增的，那么称这一段数是两个数列的公共上升子序列，而所有的公共上升子序列中最长的就是最长公共上升子序列了。

奶牛半懂不懂，小沐沐要你来告诉奶牛什么是最长公共上升子序列。不过，只要告



诉奶牛它的长度就可以了。数列 A 和 B 的长度均不超过 3000。

这道题目是 LIS 与 LCS 的综合。请读者回顾在本节开头部分列举的 LIS 与 LCS 问题的动态规划状态表示，把二者相结合，容易想到以下解法：

$F[i,j]$ 表示 $A_1 \sim A_i$ 与 $B_1 \sim B_j$ 可以构成的以 B_j 为结尾的 LCIS 的长度。不妨假设 $A_0 = B_0 = -\infty$。

当 $A_i \neq B_j$ 时，有 $F[i,j] = F[i-1,j]$

当 $A_i = B_j$ 时，有

$$F[i,j] = \max_{0 \le k < j, B_k < B_j} \{F[i-1,k]\} + 1 = \max_{0 \le k < j, B_k < A_i} \{F[i-1,k]\} + 1$$

显然，上面的状态转移方程可以直接用三重循环的程序计算（初始化部分省略）。

```
for (int i = 1; i <= n; i++)
    for (int j = 1; j <= m; j++)
        if (a[i] == b[j]) {
            for (int k = 0; k < j; k++)
                if (b[k] < a[i])
                    f[i][j] = max(f[i][j], f[i - 1][k] + 1);
        }
        else f[i][j] = f[i - 1][j];
```

在转移过程中，我们把满足 $0 \le k < j, B_k < A_i$ 的 k 构成的集合称为 $F[i,j]$ 进行状态转移时的决策集合，记为 $S(i,j)$。注意到，在第二层循环 j 从 1 增加到 m 时，第一层循环 i 是一个定值，这使得条件 $B_k < A_i$ 是固定的。因此，当变量 j 增加 1 时，k 的取值范围从 $0 \le k < j$ 变为 $0 \le k < j+1$，即整数 j 可能会进入新的决策集合。也就是说，我们只需要 $O(1)$ 地检查条件 $B_j < A_i$ 是否满足，已经在决策集合中的数则一定不会被去除。

$$S(i,j+1) = \begin{cases} S(i,j) & B_j \ge A_i \\ S(i,j) \cup \{j\} & B_j < A_i \end{cases}$$

所以上面的状态转移方程只需要两重循环即可求解。最终的目标是 $\max_{1 \le j \le m} F[n,j]$。

```
for (int i = 1; i <= n; i++) {
    // val是决策集合S(i,j)中f[i-1][k]的最大值
    int val = 0;
    // j=1时，0可以作为k的取值
    if (b[0] < a[i]) val = f[i - 1][0];
    for (int j = 1; j <= m; j++) {
        if (a[i] == b[j]) f[i][j] = val + 1;
```

```
            else f[i][j] = f[i - 1][j];
            // j 即将增大为 j+1，检查 j 能否进入新的决策集合
            if (b[j] < a[i]) val = max(val, f[i - 1][j]);
        }
    }
```

这道题转移部分的优化告诉我们，在实现状态转移方程时，要注意观察决策集合的范围随着状态的变化情况。对于“决策集合中的元素只增多不减少”的情景，就可以像本题一样维护一个变量来记录决策集合的当前信息，避免重复扫描，把转移的复杂度降低一个量级。

例题 **Making the Grade** AcWing273

给定长度为 N 的序列 A，构造一个长度为 N 的序列 B，满足：

1. B 非严格单调，即 $B_1 \le B_2 \le \cdots \le B_N$ 或 $B_1 \ge B_2 \ge \cdots \ge B_N$。

2. 最小化 $S = \sum_{i=1}^{N} |A_i - B_i|$。

只需要求出这个最小值 S。$1 \le N \le 2000$, $1 \le |A_i| \le 10^9$。

序列 B 可能有“非严格单调递增”和“非严格单调递减”两种情况，我们分别进行一次求解，二者的结果中较小的就是本题的答案。下面以递增为例进行讲解，递减同理。

引理

在满足 S 最小化的前提下，一定存在一种构造序列 B 的方案，使得 B 中的数值都在 A 中出现过。

证明：

数学归纳法。命题对 $N = 1$ 显然成立。

设引理对 $N = k - 1$ 成立，此时构造出的序列为 $B_1 \sim B_{k-1}$。

当 $N = k$ 时，若 $B_{k-1} \le A_k$，则令 $B_k = A_k$，满足单调性且命题仍成立。

否则，要么令 $B_k = B_{k-1}$，命题仍成立；要么存在一个 j，令 $B_j, B_{j+1}, \cdots, B_k$ 为同一个值 v。根据第 0x05 节中提到的“货仓选址”问题，设 $A_j, A_{j+1}, \cdots, A_k$ 的中位数是 mid，若 $mid \ge B_{j-1}$，则有 $v = mid$，否则应该有 $v = B_{j-1}$。无论哪种情况，$B_1 \sim B_k$ 中的数都在 A 中出现过。

证毕。

回到本题。我们依次考虑：完成前 $i(1 \le i \le N)$ 个数的构造时，$\sum_{j=1}^{i} |A_j - B_j|$ 的最小值。也就是把 DP 的“阶段”设为已经处理完的前缀序列的长度。在转移时，为了确定 B_i 定为多少才能保证单调，我们还需要知道 B_{i-1} 的大小。有两种常见的解决方案：

方法一

仿照 LIS 问题，设 $F[i]$ 表示完成前 i 个数的构造，并且 $B_i = A_i$ 时，S 的最小值。

$$F[i] = \min_{0 \le j < i, A[j] \le A[i]} \{F[j] + cost(j+1, i-1)\}$$

其中，$cost(j+1, i-1)$ 表示构造 $B_{j+1}, \cdots, B_{i-1}$ 同时满足 $A_j \le B_{j+1} \le \cdots \le B_{i-1} \le A_i$ 时，$\sum_{k=j+1}^{i-1} |A_k - B_k|$ 的最小值。

该解法的思路是，根据引理可知，最终 B 序列由一段段 A 中的数构成。上面 DP 中的 i 就是当前最新的一段数中与 A 相同的位置，j 是上一段数中这样的位置。因此，$cost(j+1, i-1)$ 就是把区间 $[j+1, i-1]$ 中前面的一部分变为 A_j，后面的一部分变为 A_i，所需要的最小代价。

注意，我们不需要考虑存在一个 $k \in [j+1, i-1]$，$[j+1, i-1]$ 中的一部分数变为 A_k 的情况，因为该情况已经在 $j = \mathrm{k}$ 时被上述 DP 方程涵盖。

采用朴素算法计算 $cost$ 值，总的时间复杂度为 $O(N^3)$。

方法二

既然仅把 DP 的“阶段”要素（即已经处理的序列长度）放在 DP 状态中不足以执行转移，一个直接的想法就是把 B 序列的最后一个值也记录在 DP 状态里。设 $F[i, j]$ 表示完成前 i 个数的构造，其中 $B_i = j$ 时，S 的最小值。

$$F[i, j] = \min_{0 \le k \le j} \{F[i-1, k] + |A_i - j|\}$$

根据引理，我们可以把 A 序列中出现的数进行**离散化**，把 DP 状态中第二维 j 的范围降低到 $O(N)$。另外，本题的转移与上一道例题 LCIS 非常相似，决策集合只增多不减少，$O(1)$ **即可实现转移**。故该算法的时间复杂度为 $O(N^2)$。

【思考题】

把一个序列 A 变成非严格单调递增的（单调不下降的），至少需要修改多少个数？

提示：序列 A 的总长度减去 A 的最长不下降子序列长度。

把一个序列 A 变成严格单调递增的，至少需要修改多少个数？

提示：构造序列 $B[i] = A[i] - i$，答案为序列总长度减去 B 的最长不下降子序列长度。

请读者思考：为什么这样是正确的？为什么用序列总长度减去 A 的 LIS 长度不对？

例题 **Mobile Service** AcWing274

一个公司有三个移动服务员，最初分别在位置 1，2，3 处。

如果某个位置（用一个整数表示）有一个请求，那么公司必须指派某名员工赶到那个地方去。某一时刻只有一个员工能移动，且不允许在同样的位置出现两个员工。从 p 到 q 移动一个员工，需要花费 $c(p,q)$。这个函数不一定对称，但保证 $c(p,p)=0$。

给出 N 个请求，请求发生的位置分别为 p_1~p_N。公司必须按顺序依次满足所有请求，目标是最小化公司花费，请你帮忙计算这个最小花费。$N \leq 1000$，位置是 1~200 的整数。

使用动态规划解决本题，容易发现 DP 的“阶段”就是“已经完成的请求数量”，通过指派一名服务员，可以把一个“完成 $i-1$ 个请求”的状态转移到“完成 i 个请求”的状态。

为了计算指派服务员的花费，就必须要知道状态转移时每个服务员的位置。最直接的想法就是把三个服务员的位置也放在 DP 的“状态”中。设 $F[i,x,y,z]$ 表示完成了前 i 个请求，三个员工分别位于 x,y,z 时，公司当前最小花费。

考虑 $F[i,x,y,z]$ 能够更新 $i+1$ 阶段的哪些状态。转移显然有三种，就是派三个员工之一去第 $i+1$ 个请求处。

$$F[i+1,p_{i+1},y,z]=\min\left(F[i+1,p_{i+1},y,z],F[i,x,y,z]+c(x,p_{i+1})\right)$$
$$F[i+1,x,p_{i+1},z]=\min\left(F[i+1,x,p_{i+1},z],F[i,x,y,z]+c(y,p_{i+1})\right)$$
$$F[i+1,x,y,p_{i+1}]=\min\left(F[i+1,x,y,p_{i+1}],F[i,x,y,z]+c(z,p_{i+1})\right)$$

题目要求同一位置不能出现两个员工，注意加上 if 语句判断状态转移的合法性。

该算法的规模大约在 $1000*200^3$，不能承受。仔细观察可以发现，在第 i 个请求完成时，一定有某个员工位于位置 p_i，只需要知道阶段 i 和另外两个员工的位置即可描述一个状态，处于 p_i 的员工位置对 DP 来说是冗余信息。

因此，可用 $F[i,x,y]$ 表示完成了前 i 个请求，其中一个员工位于 p_i，另外两个员工位于 x 和 y 时，公司当前最小花费。三种转移则分别是让位于 p_i，x 和 y 的员工前往 p_{i+1} 处理请求。

$$F[i+1,x,y]=\min\left(F[i+1,x,y],F[i,x,y]+c(p_i,p_{i+1})\right)$$
$$F[i+1,p_i,y]=\min\left(F[i+1,p_i,y],F[i,x,y]+c(x,p_{i+1})\right)$$
$$F[i+1,x,p_i]=\min\left(F[i+1,x,p_i],F[i,x,y]+c(y,p_{i+1})\right)$$

不妨设 $p_0=3$，于是初值可设为 $F[0,1,2]=0$，目标为 $F[N,?,?]$。与优化之前一样，也要根据题意判断状态转移的合法性，例如第一个转移应满足 x,y,p_{i+1} 不相等。

这道题给我们两点启发。

1. 求解线性 DP 问题，一般先确定“阶段”。若“阶段”不足以表示一个状态，则可以把所需的附加信息也作为状态的维度。

在转移时，若总是从一个阶段转移到下一个阶段（本题 i 到 i+1），则没有必要

关心附加信息维度的大小变化情况（本题 x,y,z 在转移前后大小不定），因为无后效性已经由“阶段”保证。

2. 在确定 DP 状态时，要选择最小的能够覆盖整个状态空间的“维度集合”。

若 DP 状态由多个维度构成，则应检查这些维度之间能否相互导出，用尽量少的维度覆盖整个状态空间，排除冗余维度。可以类比第 0x36 节中提到的“线性无关组”。例如本题，阶段 i 和两个员工位置就可表明状态，另一个员工的位置可直接得出。

例题 传纸条 AcWing275

给定一个 $N*M$ 的矩阵 A，每个格子中有一个整数。现在需要找到两条从左上角 $(1,1)$ 到右下角 (N,M) 的路径，路径上的每一步只能向右或向下走。路径经过的格子中的数会被取走，若两条路径同时经过一个格子，只算一次。求取得的数之和最大是多少。$N,M \leq 50$。

首先尝试寻找一个线性的“阶段”。考虑路径形成的过程，我们需要从左上角开始依次确定两条路径上的每一步如何走，直至到达右下角。把路径与前面问题中的序列相类比，自然想到可以把“路径长度”，即当前走过的步数，作为 DP 的“阶段”。在每个阶段中，我们把两条路径同时扩展一步，路径长度增加 1，从而转移到下一个阶段。

除了“路径长度”外，还需要确定两条路径当前的末尾位置。设路径长度为 i，第一条路径末尾位于 (x_1,y_1)，第二条路径末尾位于 (x_2,y_2)。根据上一道例题的启发，为了找到能够覆盖整个状态空间的最小的“维度集合”，我们检查上面五个值有何种关系，能否互相导出。容易发现有以下等式成立：

$$x_1+y_1=x_2+y_2=i+2$$

故 i，x_1，x_2 三个维度即可描述一个状态，其中 $1 \leq x_1,x_2 \leq \min(N,i+1)$。

设 $F[i,x_1,x_2]$ 表示两条路径长度均为 i，第一条路径末尾在第 x_1 行，第二条路径末尾在第 x_2 行时，已经取得的数之和的最大值。此时可推出 $y_1=i+2-x_1$，$y_2=i+2-x_2$。

每条路径有向右、向下两种扩展方法，故共有 $2*2=4$ 种转移。以两条路径均往右扩展为例，其余三种情况同理。

如果 $x_1=x_2$ 并且 $y_1+1=y_2+1$，那么两条路径进入同一个格子，只累加一次：

$$F[i+1,x_1,x_2]=\max(F[i+1,x_1,x_2],F[i,x_1,x_2]+A[x_1,y_1+1])$$

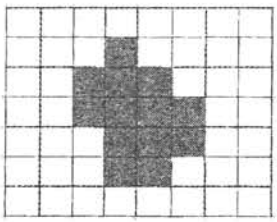
否则两条路径分别扩展到不同的格子，两个格子中的数都进行累加：

$$F[i+1,x_1,x_2]=\max(F[i+1,x_1,x_2],F[i,x_1,x_2]+A[x_1,y_1+1]+A[x_2,y_2+1])$$

初值为 $F[0,1,1]=\mathrm{A}[1,1]$。

目标为 $F[N+M-2,N,N]$。

***例题 I-country** AcWing276



在 $N*M$ 的矩阵中，每个格子有一个权值，要求寻找一个包含 K 个格子的凸连通块（连通块中间没有空缺，并且轮廓是凸的，如右图所示），使这个连通块中的格子的权值和最大。求出这个最大的权值和，并给出连通块的具体方案。$N,M \le 15$，$K \le 225$。

任何一个凸连通块可以划分成连续的若干行，每行的左端点列号先递减、后递增，右端点列号先递增、后递减。我们可以依次考虑从 $N*M$ 矩阵的每一行中选择哪些格子来构成所求的凸连通块。需要关注的信息如下。

1. 当前已经处理完的行数。
2. 已经选出的格子数。
3. 当前行已选格子的左端位置——为了确定下一行左端点的范围，以满足单调性。
4. 当前行已选格子的右端位置——为了确定下一行右端点的范围，以满足单调性。
5. 当前左侧轮廓的单调性类型——列号是该递增还是递减。
6. 当前右侧轮廓的单调性类型——列号是该递增还是递减。

行数和格子数可以作为 DP 的“阶段”，每次转移到下一行，同时选出的格子数增加，在这两个维度上都符合“阶段线性增长”的特点。因此，可以设计以下的状态表示：

$F[i,j,l,r,x,y]$ 表示前 i 行选择了 j 个格子，其中第 i 行选择了第 l 到 r 个格子（若不选则均为 0），左边界的单调性类型为 x，右边界的单调性类型为 y（0 表示递增，1 表示递减）时，能构成的凸连通块的最大权值和。

状态转移方程如下。

1. 左边界列号递减，右边界列号递增（两个边界都处于扩张状态）。

$$F[i,j,l,r,1,0] = \sum_{p=l}^{r} A[i,p] + \begin{cases} F[i-1,0,0,0,1,0] & \text{if } j = r-l+1 > 0 \\ \max\limits_{l \le p \le q \le r} \{F[i-1,j-(r-l+1),p,q,1,0]\} & \text{if } j > r-l+1 > 0 \end{cases}$$

2. 左、右边界列号都递减（左边界扩张，右边界收缩）。

$$F[i,j,l,r,1,1] = \sum_{p=l}^{r} A[i,p] + \max_{l \le p \le r \le q} \left\{ \max_{0 \le y \le 1} \{F[i-1,j-(r-l+1),p,q,1,y]\} \right\}$$

3. 左、右边界列号都递增（左边界收缩，右边界扩张）。

$$F[i,j,l,r,0,0]=\sum_{p=l}^{r}A[i,p]+\max_{p\le l\le q\le r}\left\{\max_{0\le x\le 1}\{F[i-1,j-(r-l+1),p,q,x,0]\}\right\}$$

4. 左边界列号递增，右边界列号递减（两个边界都处于收缩状态）。

$$F[i,j,l,r,0,1]=\sum_{p=l}^{r}A[i,p]+\max_{p\le l\le r\le q}\left\{\max_{0\le x\le 1}\left\{\max_{0\le y\le 1}\{F[i-1,j-(r-l+1),p,q,x,y]\}\right\}\right\}$$

初态：$F[i,0,0,0,1,0]=0$，目标：$\max\{F[i,K,l,r,x,y]\}$，时间复杂度：$O(NM^4K)=O(N^2M^5)$。

本题还要输出方案。在动态规划问题需要给出方案时，通常做法是额外使用一些与 DP 状态大小相同的数组记录下来每个状态的“最优解”是从何处转移而来的。最终，在 DP 求出最优解后，通过一次递归，沿着记录的每一步“转移来源”回到初态，即可得到一条从初态到最优解的转移路径，也就是所求的具体方案。

例题 **Cookies** AcWing277

圣诞老人共有 M 个饼干，准备全部分给 N 个孩子。每个孩子有一个贪婪度，第 i 个孩子的贪婪度为 $g[i]$。如果有 $a[i]$ 个孩子拿到的饼干数比第 i 个孩子多，那么第 i 个孩子会产生 $g[i]*a[i]$的怨气。给定 N、M 和序列 g，圣诞老人请你帮他安排一种分配方式，使得每个孩子至少分到一块饼干，并且所有孩子的怨气总和最小。$1\le N\le 30, N\le M\le 5000$。

在这道题目中，“已经获得饼干的孩子数”和“已经发放的饼干数”显然应该作为 DP 的“阶段”。不过，一个孩子的怨气大小与其他孩子获得的饼干数有关，这让我们很难对状态进行划分以产生“子结构”，同时也很难计算出每个孩子的怨气值。

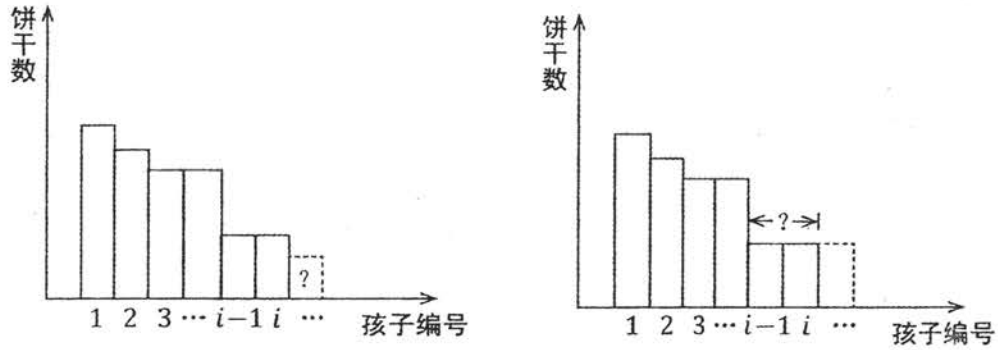
仔细思考可以发现，贪婪度大的孩子应该获得更多的饼干。读者用 0x07 节贪心算法中邻项交换的方法很容易证明这一点。因此，可以把 N 个孩子按照贪婪度从大到小排序，他们分配到的饼干数将是单调递减的。

设 $F[i,j]$ 表示前 i 个孩子一共分配 j 块饼干时，这 i 个孩子的怨气总和的最小值。直观的思想是考虑分给第 $i+1$ 个孩子多少块饼干，然后进行转移。转移时有两种情况。

1. 第 $i+1$ 个孩子获得的饼干数比第 i 个孩子少，此时 $a[i+1]=i$。

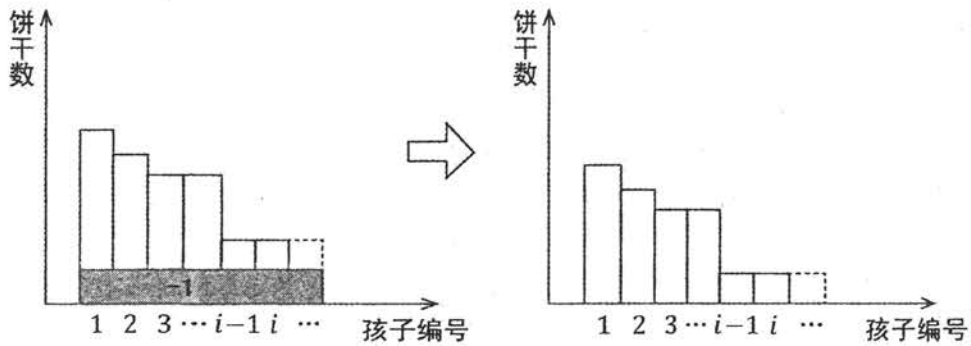
2. 第 $i+1$ 个孩子获得的饼干数与第 i 个孩子相同，此时还需要知道 i 前面有几个孩子与 i 获得的饼干数也相同，才能计算出 $a[i+1]$。

总而言之，无论哪种情况，我们都需要知道第 i 个孩子获得的饼干数，以及 i 前面有多少个孩子与 i 获得的饼干数相同，如下图所示。然而，在现有的 DP 状态下，很难高效地维护这两种信息。



观察图中的形状，我们不妨对状态转移做一个等价转换。

1. 若第 i 个孩子获得的饼干数大于 1，则等价于分配 $j-i$ 个饼干给前 i 个孩子，每人少拿 1 块饼干，获得的饼干数的相对大小顺序不变，从而怨气之和也不变。



2. 若第 i 个孩子获得的饼干数为 1，则枚举 i 前面有多少个孩子也获得 1 块饼干。

整个 DP 算法的状态转移方程：

$$F[i,j]=\min\begin{cases}F[i,j-i]\\ \min\limits_{0\le k<i}\left\{F[k,j-(i-k)]+k*\sum\limits_{p=k+1}^{i}g[p]\right\}\end{cases}$$

初态：$F[0,0]=0$，目标：$F[N,M]$。

这道题启发我们，有时可以通过额外的算法确定 DP 状态的计算顺序，有时可以在状态空间中运用等效手法对状态进行缩放。在本题中，我们就利用贪心策略，在 DP 前对 N 个孩子执行排序，使他们获得的饼干数单调递减。我们还利用相对大小的不变性，把第 $i+1$ 个孩子获得的饼干数先缩放到 1，再考虑 i 前面有几个孩子获得的饼干数量相等，使需要计算的问题得到了极大的简化，容易进行维护、转移。



背包是线性 DP 中一类重要而特殊的模型，我们把它单独作为一节进行讲解。

0/1 背包

0/1 背包问题的模型如下：

给定 N 个物品，其中第 i 个物品的体积为 V_i，价值为 W_i。有一容积为 M 的背包，要求选择一些物品放入背包，使得物品总体积不超过 M 的前提下，物品的价值总和最大。

根据上一节线性 DP 的知识，读者应该很容易想到依次考虑每个物品是否放入背包，用“已经处理的物品数”作为 DP 的“阶段”，以“背包中已经放入的物品总体积”作为附加维度。

$F[i,j]$ 表示从前 i 个物品中选出了总体积为 j 的物品放入背包，物品的最大价值和。

$$F[i,j]=\max\begin{cases}F[i-1,j] & \text{不选第 } i \text{ 个物品}\\ F[i-1,j-V_i]+W_i \quad \text{if } j\geq V_i & \text{选第 } i \text{ 个物品}\end{cases}$$

初值：$F[0,0]=0$，其余均为负无穷。目标：$\max\limits_{0\leq j\leq M}\{F[N][j]\}$。

```
memset(f, 0xcf, sizeof(f)); // -INF
f[0][0] = 0;
for (int i = 1; i <= n; i++) {
    for (int j = 0; j <= m; j++)
        f[i][j] = f[i - 1][j];
    for (int j = v[i]; j <= m; j++)
        f[i][j] = max(f[i][j], f[i - 1][j - v[i]] + w[i]);
}
```

通过 DP 的状态转移方程，我们发现，每一阶段 i 的状态只与上一阶段 $i-1$ 的状态有关。在这种情况下，可以使用被称为“**滚动数组**”的优化方法，降低空间开销。

```
int f[2][MAX_M+1];
memset(f, 0xcf, sizeof(f)); // -INF
f[0][0] = 0;
for (int i = 1; i <= n; i++) {
    for (int j = 0; j <= m; j++)
        f[i & 1][j] = f[(i-1) & 1][j];
    for (int j = v[i]; j <= m; j++)
        f[i & 1][j] = max(f[i & 1][j],
                          f[(i-1) & 1][j - v[i]] + w[i]);
}
int ans = 0;
for (int j = 0; j <= m; j++)
    ans = max(ans, f[n & 1][j]);
```

在上面的程序中，我们把阶段 i 的状态存储在第一维下标为 i&1 的二维数组中。当 i 为奇数时，i&1 等于 1；当 i 为偶数时，i&1 等于 0。因此，DP 的状态就相当于在 $F[0][]$ 和 $F[1][]$ 两个数组中交替转移，空间复杂度从 $O(NM)$ 降低为 $O(M)$。

进一步分析上面的代码，容易发现，在每个阶段开始时，实际上执行了一次从 $F[i-1][]$ 到 $F[i][]$ 的拷贝操作。这提示我们可以进一步省略掉 F 数组的第一维，只用一维数组，即当外层循环到第 i 个物品时，$F[j]$ 表示背包中放入总体积为 j 的物品的最大价值和。

```
int f[MAX_M+1];
memset(f, 0xcf, sizeof(f)); // -INF
f[0] = 0;
for (int i = 1; i <= n; i++)
    for (int j = m; j >= v[i]; j--)
        f[j] = max(f[j], f[j - v[i]] + w[i]);
int ans = 0;
for (int j = 0; j <= m; j++)
    ans = max(ans, f[j]);
```

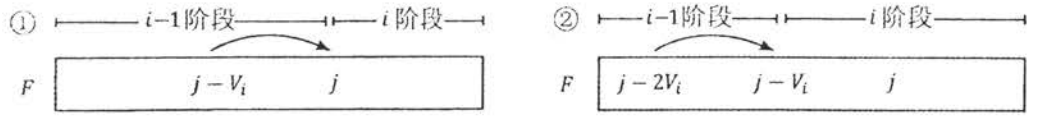
请注意上面的代码片段中特别标注的部分——我们使用了**倒序循环**。循环到 j 时：

1. F 数组的后半部分 $F[j\sim M]$ 处于“第 i 个阶段”，也就是已经考虑过放入第 i 个物品的情况；

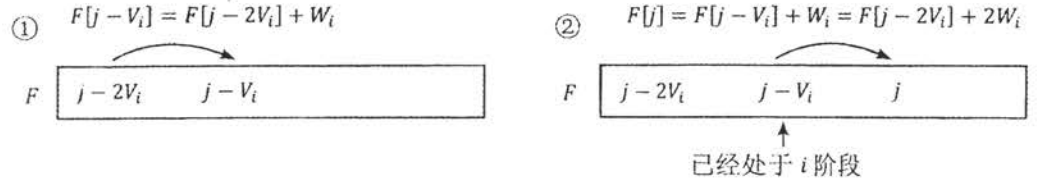
2. 前半部分 $F[0\sim j-1]$ 处于“第 $i-1$ 个阶段”，也就是还没有第 i 个物品更新。

接下来 j 不断减小，意味着我们总是用“第 $i-1$ 个阶段”的状态向“第 i 个阶

段”的状态进行转移，符合线性 DP 的原则，进而保证了第 i 个物品只会被放入背包一次。如下图所示。



然而，如果使用正序循环，假设 $F[j]$ 被 $F[j-V_i]+W_i$ 更新，接下来 j 增大到 $j+V_i$ 时，$F[j+V_i]$ 又可能被 $F[j]+W_i$ 更新。此时，两个都处于“第 i 个阶段”的状态之间发生了转移，违背了线性 DP 的原则，相当于第 i 个物品被使用了两次。如下图所示。



所以，在上面的代码中必须用倒序循环，才符合 0/1 背包问题中每个物品是唯一的、只能放入背包一次的要求。

例题 数字组合 AcWing278

给定 N 个正整数 $A_1, A_2, A_3, \cdots, A_N$，从中选出若干个数，使它们的和是 M，求有多少种选择方案。$1 \le N \le 100$，$1 \le M \le 10000$，$1 \le A_i \le 1000$。

这是一个典型的 0/1 背包模型，N 个正整数就是 N 个物品，M 就是背包的容积。在外层循环到 i 时（表示从前 i 个数中选），设 $F[j]$ 表示“和为 j”有多少种方案。在具体实现中，只需要把上面代码中求 max 的函数改为求和即可。

```
int f[MAX_M+1];
memset(f, 0, sizeof(f));
f[0] = 1;
for (int i = 1; i <= n; i++)
    for (int j = m; j >= a[i]; j--)
        f[j] += f[j - a[i]];
cout << f[m] <<endl;
```

完全背包

完全背包问题的模型如下：

给定 N 种物品，其中第 i 种物品的体积为 V_i，价值为 W_i，并且有无数个。有一

容积为 M 的背包，要求选择若干个物品放入背包，使得物品总体积不超过 M 的前提下，物品的价值总和最大。

先来考虑使用传统的二维线性 DP 的做法。设 $F[i,j]$ 表示从前 i 种物品中选出了总体积为 j 的物品放入背包，物品的最大价值和。

$$F[i,j]=\max\begin{cases}F[i-1,j] & \text{尚未选过第 } i \text{ 种物品}\\ F[i,j-V_i]+W_i \quad \text{if } j\geq V_i & \text{从第 } i \text{ 种物品中选一个}\end{cases}$$

初值：$F[0,0]=0$，其余均为负无穷。目标：$\max\limits_{0\leq j\leq M}\{F[N][j]\}$。

与 0/1 背包一样，我们也可以省略 F 数组的 i 这一维。根据我们在 0/1 背包中对循环顺序的分析，当采用**正序循环**时，就对应着每种物品可以使用无限次，也对应着 $F[i,j]=F[i,j-V_i]+W_i$ 这个在两个均处于 i 阶段的状态之间进行转移的方程。

```
int f[MAX_M+1];
memset(f, 0xcf, sizeof(f)); // -INF
f[0] = 0;
for (int i = 1; i <= n; i++)
    for (int j = v[i]; j <= m; j++)
        f[j] = max(f[j], f[j - v[i]] + w[i]);
int ans = 0;
for (int j = 0; j <= m; j++)
    ans = max(ans, f[j]);
```

例题 自然数拆分 Lunatic 版 AcWing279

给定一个自然数 N，要求把 N 拆分成若干个正整数相加的形式，参与加法运算的数可以重复。求拆分的方案数 mod 2147483648 的结果。$1\leq N\leq 4000$。

这是一个典型的完全背包模型，1~N 这 N 个自然数构成 N 种物品，每种物品都可以无限次使用，背包容积也是 N。与上一道例题类似，本题也是要求方案数，我们在完全背包程序模板的基础上，把求 max 的函数改为求和即可。

```
unsigned int f[MAX_M+1];
memset(f, 0, sizeof(f));
f[0] = 1;
for (int i = 1; i <= n; i++)
    for (int j = i; j <= n; j++)
        f[j] = (f[j] + f[j - i]) % 2147483648u;
cout << (f[n] > 0 ? f[n] - 1 : 2147483647) <<endl;
```

例题 Jury Compromise AcWing280

在一个遥远的国家，一名嫌犯是否有罪需要由陪审团来决定。陪审团是由法官从公民中挑选的。法官先随机挑选 N 个人作为陪审团的候选人，然后再从这 N 个人中按照下列方法选出 M 人组成陪审团。

首先，参与诉讼的控方和辩方会给所有候选人打分，分值在 0 到 20 之间。第 i 个人的得分分别记为 $a[i]$ 和 $b[i]$。为了公平起见，法官选出的 M 个人必须满足：辩方总分 D 和控方总分 P 的差的绝对值 $|D-P|$ 最小。如果选择方法不唯一，那么再从中选择辩、控双方总分之和 $D+P$ 最大的方案。

求最终的陪审团获得的辩方总分 D、控方总值 P，以及陪审团人选的编号（方案）。$1 \le N \le 200$，$1 \le M \le 20$。

这是一道具有**多个“体积维度”**的 0/1 背包问题。把 N 个候选人看作 N 个物品，那么每个物品有如下三种“体积”。

1. “人数”，每个“候选人”的“人数”都是 1，最终要填满容积为 M 的背包。
2. “辩方得分”，即辩方给该候选人打的分数 $a[i]$，一个 0~20 的整数。
3. “控方得分”，即控方给该候选人打的分数 $b[i]$，一个 0~20 的整数。

因此，我们依次考虑每个候选人是否被选入评审团，当外层循环到阶段 i 时，表示已经考虑了前 i 个候选人的入选情况，此时用 Boolean 数组 $F[j,d,p]$ 表示已有 j 人被选入评审团，当前辩方总分为 d、控方总分为 p 的状态是否可行。

$$F[j,d,p] = F[j,d,p] \text{ or } F[j-1,d-a[i],p-b[i]]$$

初值：$F[0,0,0] = \text{true}$，其余均为 false。

目标：找到一个状态 $F[M,d,p]$，满足 $F[M,d,p] = \text{true}$，并且 $|d-p|$ 尽量小，$|d-p|$相同时 $d+p$ 尽量大。

上述算法没有很好地利用背包“价值”这一维度，还有很大的优化空间。实际上，我们可以把每个候选人辩、控双方得分的差 $a[i]-b[i]$ 作为该物品的“体积”之一，把辩、控双方得分的和作为该物品的价值。

当外层循环到 i 时，设 $F[j,k]$ 表示已经在前 i 个候选人中选出了 j 个，此时辩方总分与控方总分的差为 k 时，辩方总分与控方总分的和的最大值。

$$F[j,k] = \max(F[j,k], F[j-1,k-(a[i]-b[i])] + a[i] + b[i])$$

初值：$F[0,0] = 0$，其余均为负无穷。

目标：找到一个状态 $F[M,k]$，满足 $|k|$ 尽量小，当 $|k|$ 相同时 $F[M,k]$ 尽量大。

在转移中，注意对 j 这一维执行**倒序循环**，即可保证每个候选人只会被选一次，符合 0/1 背包的特征。

本题还要求输出评审团人选的具体方案。正如在 0x51 节中提到的，我们采用记录

转移路径的方法，额外建立一个数组 D，其中 $D[i,j,k]$ 表示外层循环执行到 i 时，状态 $F[j,k]$ 的最大值是选了哪一名候选人而得到的。这里我们必须采用三维数组，否则后续对 i 的循环可能会覆盖之前 $D[j,k]$ 保存的值。求出最优解后，可以沿着数组 D 记录的转移路径，递归得到方案。设 $last = D[i,j,k]$，不断从状态 (i,j,k) 递归到 $(last-1, j-1, k-(a[last]-b[last]))$，直至 $j=0$。递归过程中所有的 $last$ 值就构成了评审团的人选。具体实现请查阅配套资源中的参考程序。

多重背包

多重背包问题的模型如下：

给定 N 种物品，其中第 i 种物品的体积为 V_i，价值为 W_i，并且有 C_i 个。有一容积为 M 的背包，要求选择若干个物品放入背包，使得物品总体积不超过 M 的前提下，物品的价值总和最大。

直接拆分法

求解多重背包问题最直接的方法是把第 i 种物品看作独立的 C_i 个物品，转化为共有 $\sum_{i=1}^{N} C_i$ 个物品的 0/1 背包问题进行计算，时间复杂度为 $O(M * \sum_{i=1}^{N} C_i)$。该算法把每种物品拆成了 C_i 个，效率较低。

```
unsigned int f[MAX_M+1];
memset(f, 0xcf, sizeof(f)); // -INF
f[0] = 0;
for (int i = 1; i <= n; i++)
    for (int j = 1; j <= c[i]; j++)
        for (int k = m; k >= v[i]; k--)
            f[k] = max(f[k], f[k - v[i]] + w[i]);
int ans = 0;
for (int i = 0; i <= m; i++) ans = max(ans, f[i]);
```

二进制拆分法

众所周知，从 $2^0, 2^1, 2^2, \cdots, 2^{k-1}$ 这 k 个 2 的整数次幂中选出若干个相加，可以表示出 $0 \sim 2^k - 1$ 之间的任何整数。进一步地，我们求出满足 $2^0 + 2^1 + \cdots + 2^p \le C_i$ 的最大的整数 p，设 $R_i = C_i - 2^0 - 2^1 - \cdots - 2^p$，那么：

1. 根据 p 的最大性，有 $2^0 + 2^1 + \cdots + 2^{p+1} > C_i$，可推出 $2^{p+1} > R_i$，因此 $2^0, 2^1, \cdots, 2^p$ 选出若干个相加可以表示出 $0 \sim R_i$ 之间的任何整数；

2. 从 $2^0, 2^1, \cdots, 2^p$ 以及 R_i 中选出若干个相加，可以表示出 $R_i \sim R_i + 2^{p+1} - 1$ 之间的任何整数，而根据 R_i 的定义，$R_i + 2^{p+1} - 1 = C_i$，因此 $2^0, 2^1, \cdots, 2^p, R_i$ 选出若干个相加可以表示出 $R_i \sim C_i$ 之间的任何整数。

综上所述，我们可以把数量为 C_i 的第 i 种物品拆成 $p+2$ 个物品，它们的体积分别为：

$$2^0 * V_i, 2^1 * V_i, \cdots, 2^p * V_i, R_i * V_i$$

这 $p+2$ 个物品可以凑成 $0 \sim C_i * V_i$ 之间所有能被 V_i 整除的数，并且不能凑成大于 $C_i * V_i$ 的数。这等价于原问题中体积为 V_i 的物品可以使用 $0 \sim C_i$ 次。该方法仅把每种物品拆成了 $O(\log C_i)$ 个，效率较高。

单调队列

使用单调队列优化的动态规划算法求解多重背包问题，时间复杂度可以进一步降低到 $O(NM)$，与 0/1 背包和完全背包中 DP 算法的效率相同，我们将在 0x59 节中进行讲解。

例题 **Coins** AcWing281

给定 N 种硬币，其中第 i 种硬币的面值为 A_i，共有 C_i 个。从中选出若干个硬币，把面值相加，若结果为 S，则称“面值 S 能被拼成”。求 $1 \sim M$ 之间能被拼成的面值有多少个。$1 \le N \le 100, 1 \le M \le 10^5, 1 \le A_i \le 10^5, 1 \le C_i \le 1000$。

本题是一个多重背包模型，“硬币”为物品，“面值”为体积，M 为背包总容积。这道题目中没有“物品价值”属性，不是一个最优化问题，而是一个可行性问题。按照我们方才介绍的 DP 算法，可以依次考虑每种硬币是否被用于拼成最终的面值（是否放入背包），以“已经考虑过的物品种数” i 作为 DP 的“阶段”，在阶段 i 时，$F[j]$ 表示前 i 种硬币能否拼成面值 j。用“直接拆分法”朴素求解的代码片段如下：

```
bool f[100010];
memset(f, 0, sizeof(f));
f[0] = true;
for (int i = 1; i <= n; i++)
    for (int j = 1; j <= c[i]; j++)
        for (int k = m; k >= a[i]; k--)
            f[k] |= f[k - a[i]];
int ans = 0;
for (int i = 1; i <= m; i++)
    ans += f[i];
```

上述算法的时间复杂度过高，不能通过本题的测试数据。当然，读者可以采用“二进制拆分法”或者“单调队列”进行优化。不过，本题**仅关注“可行性”（面值能否拼成）而不是“最优性”**，这是一个特殊之处。仔细分析动态规划的过程，我们可以发现，

若前 i 种硬币能够拼成面值 j，只有两类可能情况。

1. 前 $i-1$ 种硬币就能拼成面值 j，即在第 i 阶段开始前，变量 $F[j]$ 已经为 true。

2. 使用了第 i 种硬币，即在第 i 阶段的递推中，发现 $F[j-a[i]]$ 为 true，从而变量 $F[j]$ 变为 true。

于是我们可以考虑一种贪心策略：设 $used[j]$ 表示 $F[j]$ 在阶段 i 时为 true 至少需要用多少枚第 i 种硬币，并且尽量选择第一类情况。也就是说，在 $F[j-a[i]]$ 为 true 时，如果 $F[j]$ 已经为 true，则不执行 DP 的转移，并令 $used[j]=0$，否则才执行 $F[j]=F[j]$ or $F[j-a[i]]$ 的转移，并令 $used[j]=used[j-a[i]]+1$。

在“直接拆分法”程序的基础上，把核心部分作如下修改：

```
int used[100010];
for (int i = 1; i <= n; i++) {
    for (int j = 0; j <= m; j++) used[j] = 0;
    for (int j = a[i]; j <= m; j++)
        if (!f[j] && f[j - a[i]] && used[j - a[i]] < c[i])
            f[j] = true, used[j] = used[j - a[i]] + 1;
}
```

该程序使用了“完全背包”模型中 j 的循环顺序，但通过 $used$ 数组实现多重背包中“物品个数”的限制，同时根据贪心策略，仅当 $F[j]$ 为 false，不得不利用第 i 种硬币时才执行转移，保证不会漏掉可行解。

分组背包

分组背包问题的模型如下：

给定 N 组物品，其中第 i 组有 C_i 个物品。第 i 组的第 j 个物品的体积为 V_{ij}，价值为 W_{ij}。有一容积为 M 的背包，要求选择若干个物品放入背包，使得每组至多选择一个物品并且物品总体积不超过 M 的前提下，物品的价值总和最大。

仍然先考虑原始线性 DP 的做法。为了满足“每组至多选择一个物品”，很自然的想法就是利用“阶段”线性增长的特征，把“物品组数”作为 DP 的“阶段”，只要使用了一个第 i 组的物品，就从第 i 个阶段的状态转移到第 $i+1$ 个阶段的状态。设 $F[i,j]$ 表示从前 i 组中选出总体积为 j 的物品放入背包，物品的最大价值和。

$$F[i,j]=\max\begin{cases}F[i-1,j] & \text{不选第 } i \text{ 组的物品}\\ \max\limits_{1\le k\le C_i} F[i-1,j-V_{ik}]+W_{ik} & \text{选第 } i \text{ 组的某个物品 } k\end{cases}$$

与前面几个背包模型一样，我们可以省略掉 F 数组的第一维，用 j 的倒序循环来



控制“阶段 i”的状态只能从“阶段 $i-1$”转移而来。

```
memset(f, 0xcf, sizeof(f));
f[0] = 0;
for (int i = 1; i <= n; i++)
    for (int j = m; j >= 0; j--)
        for (int k = 1; k <= c[i]; k++)
            if (j >= v[i][k])
                f[j] = max(f[j], f[j - v[i][k]] + w[i][k]);
```

除了**倒序循环** j 之外，请读者格外留意，对于**每一组内** $c[i]$ **个物品的循环** k 应该放在 j 的**内层**。从背包的角度看，这是因为每组内至多选择一个物品，若把 k 置于 j 的外层，就会类似于多重背包，每组物品在 F 数组上的转移会产生累积，最终可以选择超过一个物品。从动态规划的角度，i 是“阶段”，i 与 j 共同构成“状态”，而 k 是“决策”——在第 i 组内使用哪一个物品，这三者的顺序绝对不能混淆。

另外，分组背包是许多树形 DP 问题中状态转移的基本模型，读者在 0x54 节中将会进一步接触到它。

在本节中，我们介绍了 0/1 背包、完全背包、多重背包和分组背包。除了以传统的线性 DP 求解之外，我们还尽量缩减了空间复杂度，省去了“阶段”的存储，用适当的循环顺序控制状态在原有基础上直接转移和累积。请读者不要只把正确的循环顺序记住了事，务必要理解清楚其中的本质和内在联系。

0x53 区间 DP

到目前为止，我们介绍的线性 DP 一般从初态开始，沿着阶段的扩张向某个方向递推，直至计算出目标状态。区间 DP 也属于线性 DP 中的一种，它以“区间长度”作为 DP 的“阶段”，使用两个坐标（区间的左、右端点）描述每个维度。在区间 DP 中，一个状态由若干个比它更小且包含于它的区间所代表的状态转移而来，因此，区间 DP 的决策往往就是划分区间的方法。区间 DP 的初态一般就由长度为 1 的“元区间”构成。这种向下划分再向上递推的模式与某些树形结构，例如第 0x43 节的线段树，有很大的相似之处。我们把区间 DP 作为线性 DP 中一类重要的分支单独进行讲解，将使读者更容易理解下一节树形 DP 的内容。同时，借助区间 DP 这种与树形相关的结构，我们也将提及记忆化搜索——其本质是动态规划的递归实现方法。

例题 石子合并 CH5301/AcWing282

有 N 堆石子排成一排，其中第 i 堆石子的重量为 A_i。每次可以选择其中相邻的两堆石子合并成一堆，形成的新石子堆的重量以及消耗的体力都是两堆石子的重量之和。求把全部 N 堆石子合成一堆最少需要消耗多少体力。$1 \le N \le 300$。

若最初的第 l 堆石子和第 r 堆石子被合并成一堆，则说明 l~r 之间的每堆石子也已经被合并，这样 l 和 r 才有可能相邻。因此，在任意时刻，任意一堆石子均可以用一个闭区间 $[l,r]$ 来描述，表示这堆石子是由最初的第 l~r 堆石子合并而成的，其重量为 $\sum_{i=l}^{r} A_i$。另外，一定存在一个整数 $k(l \le k < r)$，在这堆石子形成之前，先有第 l~k 堆石子（闭区间 $[l,k]$）被合并成一堆，第 $k+1$~r 堆石子（闭区间 $[k+1,r]$）被合并成一堆，然后这两堆石子才合并成 $[l,r]$。

对应到动态规划中，就意味着两个长度较小的区间上的信息向一个更长的区间发生了转移，划分点 k 就是转移的决策。自然地，应该把区间长度 len 作为 DP 的阶段。不过，区间长度可以由左端点和右端点表示出，即 $len = r - l + 1$。本着动态规划“选择最小的能覆盖状态空间的维度集合”的思想，可以只用左、右端点表示 DP 的状态。

设 $F[l,r]$ 表示把最初的第 l 堆到第 r 堆石子合并成一堆，需要消耗的最少体力，根据上述分析，容易写出状态转移方程：

$$F[l,r] = \min_{l \le k < r} \{F[l,k] + F[k+1,r]\} + \sum_{i=l}^{r} A_i$$

初值：$\forall l \in [1,N], F[l,l]=0$，其余为正无穷。

目标：$F[1,N]$。

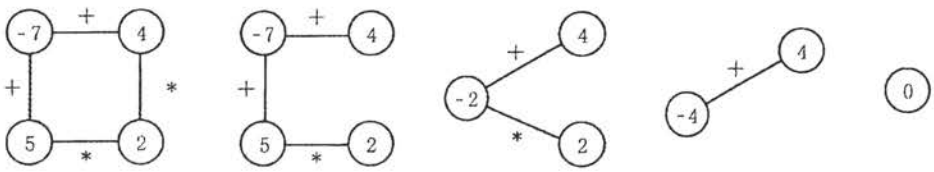
我们在分组背包最后强调，编程实现动态规划的状态转移方程时，务必**分清阶段、状态与决策，三者应该按照从外到内的顺序依次循环**。而 $\sum_{i=l}^{r} A_i$ 可使用前缀和计算。

```
memset(f, 0x3f, sizeof(f)); // INF
for (int i = 1; i <= n; i++) {
    f[i][i] = 0;
    sum[i] = sum[i-1] + a[i]; // 前缀和
}
for (int len = 2; len <= n; len++) // 阶段
    for (int l = 1; l <= n - len + 1; l++) { // 状态：左端点
        int r = l + len - 1; // 状态：右端点
        for (int k = l; k < r; k++) // 决策
            f[l][r] = min(f[l][r], f[l][k] + f[k+1][r]);
        f[l][r] += sum[r] - sum[l-1];
    }
```

例题 **Polygon** AcWing283

“多边形游戏”是一款单人益智游戏。在游戏开始时，系统给定玩家一个 N 边形，该 N 边形由 N 个顶点和 N 条边构成，每条边连接两个相邻的顶点。在每个顶点上写有一个整数，可正可负。在每条边上标有一个运算符“+”（加号）或“*”（乘号）。

第一步，玩家需要选择一条边，将它删除。接下来再进行 $N-1$ 步，在每一步中，玩家选择一条边，把这条边以及该边连接的两个顶点用一个新的顶点代替，新顶点上的整数值等于删去的两个顶点上的数按照删去的边上标有的符号进行计算得到的结果。如下图所示，就是一盘游戏的过程。



最终，游戏仅剩一个顶点，顶点上的数值就是玩家的得分，上图玩家得 0 分。

请计算对于给定的 N 边形，玩家最高能获得多少分，以及第一步有哪些策略可以使玩家获得最高得分。

$1 \le N \le 50$，保证玩家无论如何操作，顶点上的数值均在 $[-32768, 32767]$ 之内。

在枚举第一步删除哪条边后，这道题就与上一题“石子合并”非常相似，仍然是在

每一步中对两个相邻的元素做某种运算合成一个。简便起见，我们把被删除的边逆时针方向的顶点称为“第 1 个顶点”，依此类推。读者容易想到使用 $F[l,r]$ 表示把第 l 到 r 个顶点合成一个顶点后，顶点上的数值最大是多少。

然而，请读者在使用动态规划解决每一道问题时，都时刻牢记动态规划的“三要素”和使用动态规划的“三前提”。把“顶点上的最大数值”作为每一个子问题 $[l,r]$ 的代表信息，不符合动态规划的“最优子结构”性质。因为负数的存在，进行乘法运算时，大区间 $[l,r]$ 合成的顶点的最大数值不能由区间 $[l,k]$ 和区间 $[k+1,r]$ 合成的两个顶点的最大数值导出——因为区间 $[l,k]$ 和区间 $[k+1,r]$ 合成的两个顶点的最小数值可能是很小的负数，负负相乘得正，运算结果可能更大。

不过，上面的反例也启发我们，如果把一个区间 $[l,r]$ 能够合成的顶点上的最大和最小数值同时作为子问题 $[l,r]$ 的代表信息，是否满足最优子结构性质？答案是肯定的。最大值的来源只可能是两个最大值相加、相乘，或两个最小值相乘（负负得正），或一个最大值与一个最小值相乘（当其中一个子区间的最大、最小值都是正数，而另一个子区间的最大、最小值都是负数时）。最小值的来源可能是两个最小值相加、相乘，或一个最大值与一个最小值相乘（正负得负），或两个最大值相乘（当两个子区间的最大值、最小值都是负数时）。请读者尝试给出完整的证明。

因此，可以设 $F[l,r,0]$ 表示把第 l 到 r 个顶点合成一个顶点后，顶点上的数值最大是多少，设 $F[l,r,1]$ 表示把第 l 到 r 个顶点合成一个顶点后，顶点上的数值最小是多少。枚举区间的划分点 k（决策），状态转移方程如下：

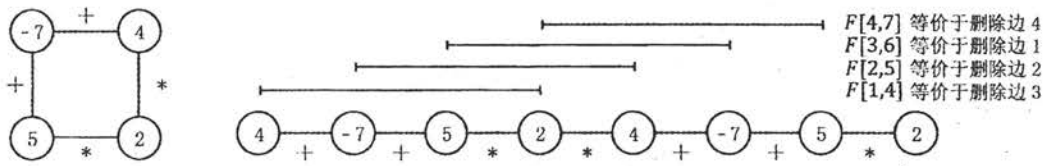
$$F[l,r,0]=\max_{l\le k<r}\left\{\max\begin{cases}F[l,k,0]\ \mathrm{op}\ F[k+1,r,0] & \mathrm{op}\in\{+,*\}\\ F[l,k,p]*F[k+1,r,q] & p,q\in\{0,1\}\end{cases}\right\}$$

$$F[l,r,1]=\min_{l\le k<r}\left\{\min\begin{cases}F[l,k,1]\ \mathrm{op}\ F[k+1,r,1] & \mathrm{op}\in\{+,*\}\\ F[l,k,p]*F[k+1,r,q] & p,q\in\{0,1\}\end{cases}\right\}$$

初值：$\forall l\in[1,N],F[l,l,0]=F[l,l,1]=A_l$，其余为正或负无穷。

目标：$F[1,N,0]$。

上述算法的时间复杂度为 $O(N^4)$。实际上我们还可以进一步优化掉枚举第一步删除哪条边耗费的时间。在游戏最初，我们任意选择一条边删除，然后把剩下的“链”复制 1 倍接在末尾（以被删除的边逆时针方向的第一个顶点为开头），如下图所示。



在这个长度为 $2N$ 的“链”上，$\forall i\in[1,N]$，把长度为 N 的区间 $[i,i+N-1]$ 合并成一个顶点，就等价于原游戏的第一步删除第 i 个顶点逆时针一侧的边，然后把剩余

的部分合并成一个顶点。因为区间长度是 DP 的阶段，我们只需要对前 N 个阶段进行 DP，每个阶段只有不超过 $2N$ 个状态，总时间复杂度降低为 $O(N^3)$。最后的答案是 $\max_{1\le i\le N}\{F[i,i+N-1]\}$。

这种“**任意选择一个位置断开，复制形成 2 倍长度的链**”的方法，是解决 DP 中环形结构的常用手段之一，我们会在 0x55 节进一步探讨。

例题 金字塔 AcWing284

虽然探索金字塔是极其老套的剧情，但是有一队探险家还是到了某金字塔脚下。经过多年的研究，科学家对这座金字塔的内部结构已经有所了解。首先，金字塔由若干房间组成，房间之间连有通道。如果把房间看作节点，通道看作边的话，整个金字塔呈现一个有根树结构，节点的子树之间有序，金字塔有唯一的一个入口通向树根。并且，每个房间的墙壁都涂有若干种颜色的一种。

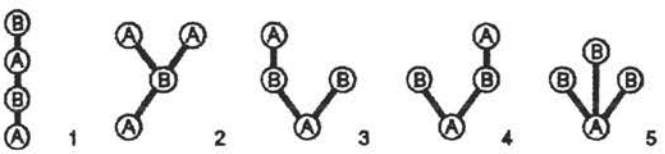
探险队员打算进一步了解金字塔的结构，为此，他们使用了一种特殊设计的机器人。这种机器人会从入口进入金字塔，之后对金字塔进行深度优先遍历。机器人每进入一个房间（无论是第一次进入还是返回），都会记录这个房间的颜色。最后，机器人会从入口退出金字塔。

显然，机器人会访问每个房间至少一次，并且穿越每条通道恰好两次（两个方向各一次），然后，机器人会得到一个颜色序列。但是，探险队员发现这个颜色序列并不能唯一确定金字塔的结构。现在他们想请你帮助他们计算，对于一个给定的颜色序列，有多少种可能的结构会得到这个序列。因为结果可能会非常大，你只需要输出答案对 10^9 取模之后的值。

输入文件包含一行，一个字符串 S，长度不超过 300，表示机器人得到的颜色序列。

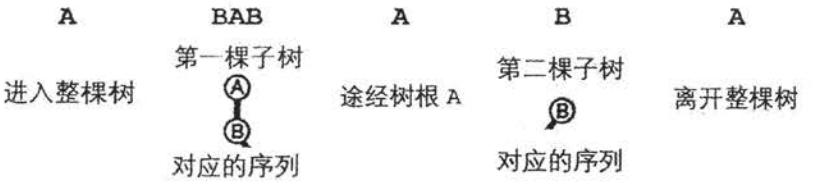
输出一个整数表示答案。

例如序列“ABABABA”对应 5 种金字塔结构，最底部是树根。我们认为子树之间是有序的，所以方案 3 和 4 是两种不同的方案。如下图所示。



在 0x21 节中我们提到过，一棵树的每棵子树都对应着这棵树 DFS 序中的一个区间。本题中记录的序列虽然不是 DFS 序，但仍然满足这条性质。因此，这道题目在“树形结构”与“字符串”之间通过“子树”和“区间”建立了联系。结合本节前半部分对区间 DP 的分析，读者不难想到用 $F[l,r]$ 表示子串 $S[l\sim r]$ 对应着多少种可能的金字塔结构（树形结构）。

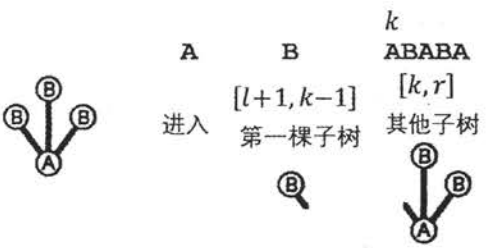
接下来我们考虑对区间的划分。以上页图中的方案 3 为例，序列“ABABABA”被分成五个部分：



同理，方案 5 把序列分成“A|B|A|B|A|B|A”七个部分。也就是说，若子串 $S[l\sim r]$ 对应一棵子树，则 $S[l+1]$，$S[r-1]$ 两个字符是进入和离开时产生的。除此之外，$[l,r]$ 包含的每棵更深的子树都对应着一个子问题，会产生 $[l,r]$ 中的一段。相邻两段之间还有途经树根产生的一个字符。因为 $[l,r]$ 包含的子树个数可能不止两个，如果我们像前面的题目一样，采用朴素算法枚举子串 $S[l\sim r]$ 划分点的数量和**所有划分点的位置**，那么时间复杂度会变得非常高。

读者可能会想到，把子串 $S[l\sim r]$ 分成两部分，每部分可由**若干棵子树**组成。不过这样可能会产生重复计数。如果每段可以由多棵子树构成，那么划分方案“A|BAB|A|B|A 和“A|B|A|BAB|A”中的“BAB”都能产生“B|A|B”两棵子树，最终归为同一结果——方案 5。

实际上，为了解决让计数不重不漏，我们可以只考虑子串 $S[l\sim r]$ 的**第一棵子树**是由哪一段构成的。枚举划分点 k，令子串 $S[l+1\sim k-1]$ 构成 $[l,r]$ 的第一棵子树，$S[k\sim r]$ 构成 $[l,r]$ 的剩余部分（其他子树）。如下图所示。



如果 k 不相同，那么子串 $S[l+1\sim k-1]$ 代表的子树的大小也不相同，就不可能产生重复计算的结构。于是，我们可以得到状态转移方程。

$$F[l,r]=\begin{cases}F[l+1,r-1]+\sum\limits_{l+2\le k\le r-2,S[l]=S[k]}^{0}F[l+1,k-1]*F[k,r] & S[l]=S[r]\\ 0 & S[l]\neq S[r]\end{cases}$$

初值：$\forall l\in[1,N],F[l,l]=1$，其余均为 0。目标：$F[1,N]$。

这道题告诉我们，对于方案计数类的动态规划问题，通常一个**状态的各个决策之间满足“加法原理”**，而每个**决策划分的几个子状态之间满足“乘法原理”**。在设计状态转移方程的决策方式与划分方法时，一个状态的所有决策之间必须具有**互斥性**，才能保



证不会出现重复问题。在 0x5C 节我们会进一步探讨计数类 DP 的相关模型与求解策略。

在具体的程序编写中，区间 DP 不仅可以用递推（若干循环）来实现，也可以用递归（记忆化搜索）来实现。把子问题的求解过程写成一个函数 solve(l,r)，枚举划分点 k，递归求解 solve$(l+1,k-1)$ 和 solve(k,r)，回溯时把二者的结果相乘，加到 solve(l,r) 的结果中。在上述过程中，一个区间 $[l,r]$ 对应的函数 solve(l,r) 可能会被调用多次，我们可以建立一个全局数组 F，在第一次计算完 solve(l,r) 时把结果保存在 $F[l,r]$ 中，之后 solve(l,r) 再被调用时就可以直接返回 $F[l,r]$。这样带有记忆化的搜索就保证了每个区间只会被求解一次，时间复杂度仍然是 $O(N^3)$。

```
int f[310][310], MOD = 1000000000; // 对 MOD 取模

int solve(int l, int r) {
    if (l > r) return 0; // 递归边界
    if (l == r) return 1; // 递归边界
    if (f[l][r] != -1) return f[l][r]; // 记忆化
    f[l][r] = 0;
    for (int k = l + 2; k <= r; k++)
        f[l][r] = (f[l][r] +
            (long long)solve(l+1, k-1) * solve(k, r)) % MOD;
    return f[l][r];
}

memset(f, -1, sizeof(f)); // -1 表示没有被计算过
solve(1, n);
```

0x54 树形 DP

通过上一节的学习，读者应该对动态规划在树形结构上的实现方式有了初步的认识。给定一棵有 N 个节点的树（通常是无根树，也就是有 $N-1$ 条无向边），我们可以任选一个节点为根节点，从而定义出每个节点的深度和每棵子树的根。在树上设计动态规划算法时，一般就以**节点从深到浅（子树从小到大）的顺序作为 DP 的“阶段”**。DP 的状态表示中，**第一维通常是节点编号**（代表以该节点为根的子树）。大多数时候，我们采用递归的方式实现树形动态规划。对于每个节点 x，**先递归在它的每个子节点上进行 DP，在回溯时，从子节点向节点 x 进行状态转移。**

例题 没有上司的舞会 AcWing285

Ural 大学有 N 名职员，编号为 $1\sim N$。他们的关系就像一棵以校长为根的树，父节点就是子节点的直接上司。每个职员有一个快乐指数，用整数 H_i 给出，其中 $1\le i\le N$。现在要召开一场周年庆宴会，不过，没有职员愿意和直接上司一起参会。在满足这个条件的前提下，主办方希望邀请一部分职员参会，使得所有参会职员的快乐指数总和最大，求这个最大值。

根据本节开始时的讨论，我们就以节点编号（子树的根）作为 DP 状态的第一维。因为职员是否愿意参加只跟他的直接上司是否参加有关，所以我们在每棵子树递归完成时，保留两个“代表信息”——根节点参加时整棵子树的最大快乐指数和，以及根节点不参加时整棵子树的最大快乐指数和，就可以满足“最优子结构”性质了。

设 $F[x,0]$ 表示从以 x 为根的子树中邀请一部分职员参会，并且 x 不参加舞会时，快乐指数总和的最大值。此时，x 的子节点（直接下属）可以参会，也可以不参会。

$$F[x,0]=\sum_{s\in Son(x)}\max(F[s,0],F[s,1])$$

设 $F[x,1]$ 表示从以 x 为根的子树中邀请一部分职员参会，并且 x 参加舞会时，快乐指数总和的最大值。此时，x 的所有子节点（直接下属）都不能参会。

$$F[x,1]=H[x]+\sum_{s\in Son(x)}F[s,0]$$

其中，$Son(x)$ 表示 x 的子节点集合。

本题输入的是一棵有根树（指定了节点间的上下属关系），故我们需要先找出没有

上司的节点 $root$ 作为根，DP 的目标为 $\max(F[root,0],F[root,1])$。时间复杂度为 $O(N)$。

```
vector<int> son[10010];
int f[10010][2], v[10010], h[10010], n;
void dp(int x) {
    f[x][0] = 0;
    f[x][1] = h[x];
    for (int i = 0; i < son[x].size(); i++) {
        int y = son[x][i];
        dp(y);
        f[x][0] += max(f[y][0], f[y][1]);
        f[x][1] += f[y][0];
    }
}

int main() {
    cin >> n;
    for (int i = 1; i <= n; i++) scanf("%d", &h[i]);
    for (int i = 1; i < n; i++) {
        int x, y;
        scanf("%d %d", &x, &y);
        v[x] = 1; // x has a father
        son[y].push_back(x); // x is a son of y
    }
    int root;
    for (int i = 1; i <= n; i++)
        if (!v[i]) { // i doesn't have a father
            root = i;
            break;
        }
    dp(root);
    cout << max(f[root][0], f[root][1]) << endl;
}
```

正如深度优先和广度优先都可以对树或图进行遍历一样，除了**自顶向下的递归**外，我们也可以用**自底向上的拓扑排序**来执行树形 DP。通常前者已经足够，这里我们就不再赘述。

在更多的题目中，树是以一张 N 个点、$N-1$ 条边的无向图的形式给出的。在这

种情况下使用树形 DP 算法时，我们就要用邻接表存下来这 $N-1$ 条无向边，任选一个点出发执行深度优先遍历，并注意标记节点是否已经被访问过，以避免在遍历中沿着反向边回到父节点。

背包类树形DP

例题 选课 AcWing286

学校实行学分制。每门的必修课都有固定的学分，同时还必须获得相应的选修课程学分。学校开设了 $N(N \leq 300)$ 门的选修课程，每个学生可选课程的数量 M 是给定的。学生选修了这 M 门课并考核通过就能获得相应的学分。

在选修课程中，有些课程可以直接选修，有些课程需要一定的基础知识，必须在选了其他的一些课程的基础上才能选修。例如《Windows 程序设计》必须在选修了《Windows 操作基础》之后才能选修。我们称《Windows 操作基础》是《Windows 程序设计》的先修课。每门课的直接先修课最多只有一门。两门课可能存在相同的先修课。

你的任务是为自己确定一个选课方案，使得你能得到的学分最多，并且必须满足先修条件。假定课程之间不存在时间上的冲突。

因为每门课的先修课最多只有 1 门（对应着树中每个节点至多只有 1 个父亲），所以这 N 门课程构成了森林结构（若干棵树，因为可能有不止一门课没有先修课）。简便起见，我们可以新建一个“虚拟课程”——0 号节点，作为“实际上没有先修课的课程”的先修课，把包含 N 个节点的森林转化为包含 $N+1$ 个节点的树，其中节点 0 为根节点。

设 $F[x,t]$ 表示在以 x 为根的子树中选 t 门课能够获得的最高学分，设 x 的子节点集合为 $Son(x)$，子节点个数 $p=|Son(x)|$。修完 x 这门课后，对于 x 的每个子节点 $y_i \in Son(x)$，我们可以在以 y_i 为根的子树中选修若干门课（记为 c_i），在满足 $\sum c_i = t-1$ 的基础上获得尽量多的学分。

当 $t=0$ 时，显然 $F[x,t]=0$。当 $t>0$ 时，根据以上分析，状态转移方程如下：

$$F[x,t]=\max_{\sum_{i=1}^{p} c_i=t-1}\left\{\sum_{i=1}^{p} F[y_i,c_i]\right\}+score[x]$$

该方程其实是一个**分组背包**模型。有 $p=|Son(x)|$ 组物品，每组物品都有 $t-1$ 个，其中第 i 组的第 j 个物品的体积为 j，价值为 $F[y_i,j]$，背包的总容积为 $t-1$。我们要从每组中选出不超过 1 个物品（每个子节点 y 只能选一个状态转移到 x），使得物品体积不超过 $t-1$ 的前提下（在修完 x 后，还能选修 $t-1$ 门课），物品价值总和最大（获得最多学分）。当然，$x=0$ 是一个特例，因为虚拟的根节点实际上不需要被选修，此时背包总容积应为 t。我们用分组背包进行树形 DP 的状态转移，关键部

分的代码如下：

```
void dp(int x) {
    f[x][0] = 0;
    for (int i = 0; i < son[x].size(); i++) { // 循环子节点（物品）
        int y = son[x][i];
        dp(y);
        // 倒序循环当前选课总门数（当前背包体积）
        for (int t = m; t >= 0; t--)
            // 循环更深子树上的选课门数（组内物品）
            for (int j = 0; j <= t; j++)
                f[x][t] = max(f[x][t], f[x][t-j] + f[y][j]);
    }
    if (x != 0) // x不为0时，选修x本身需要占用1门课，获得相应学分
        for(int t = m; t > 0; t--)
            f[x][t] = f[x][t-1] + score[x];
}
```

这类题目被称为**背包类树形 DP**，又称**有树形依赖的背包问题**。它实际上是背包与树形 DP 的结合。除了以“节点编号”作为树形 DP 的阶段，通常我们也像线性 DP 一样，把当前背包的“体积”作为第二维状态。在状态转移时，我们要处理的实际上就是一个**分组背包**问题。

值得一提的是，这类题目还可以先按照一个被称为“**左儿子右兄弟**”的方法，把多叉树转化成二叉树，再进行计算。这样一来，在状态转移时就只需要枚举左右两棵子树中的一棵选了多少门课。不过，这种方法需要做额外的转换，在复杂的题目中很容易把节点之间的父子关系、兄弟关系混淆，容易出错。因此，**本书提倡读者直接在原始的多叉树上使用分组背包模型进行背包类树形 DP。**

二次扫描与换根法

例题 **Accumulation Degree** AcWing287

有一个树形的水系，由 $N-1$ 条河道和 N 个交叉点组成。我们可以把交叉点看作树中的节点，编号为 $1\sim N$，河道则看作树中的无向边。每条河道都有一个容量，连接 x 与 y 的河道的容量记为 $c(x,y)$。河道中单位时间流过的水量不能超过河道的容量。

有一个节点是整个水系的发源地，可以源源不断地流出水，我们称之为源点。除了源点之外，树中所有度数为 1 的节点都是入海口，可以吸收无限多的水，我们称之为汇点。也就是说，水系中的水从源点出发，沿着每条河道，最终流向各个汇点。

在整个水系稳定时，每条河道中的水都以单位时间固定的水量流向固定的方向。除源点和汇点之外，其余各点不贮存水，也就是流入该点的河道水量之和等于从该点流出的河道水量之和。整个水系的流量就定义为源点单位时间发出的水量。

在流量不超过河道容量的前提下，求哪个点作为源点时，整个水系的流量最大，输出这个最大值。$N \leq 2 * 10^5$。

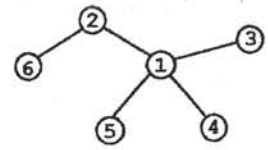
本题是一个“不定根”的树形 DP 问题。这类题目的特点是，给定一个树形结构，需要以**每个节点为根**进行一系列统计。我们一般通过两次扫描来求解此类题目：

1. 第一次扫描时，任选一个点为根，在“有根树”上执行一次**树形 DP**，也就是在回溯时发生的、自底向上的状态转移。

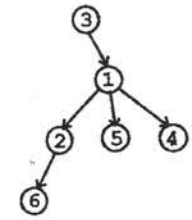
2. 第二次扫描时，从刚才选出的根出发，对整棵树执行一次**深度优先遍历**，在每次递归前进行自顶向下的推导，计算出“换根”后的解。

本书以“**二次扫描与换根法**”来指明与上述过程类似的算法。

我们先来考虑本题的朴素解法——枚举源点 s，然后计算水系流量。把源点作为树根，整个水系就变成了一棵有根树，每条河道的方向也可以直接得出。如下图所示。



无根树
(n个点、n-1条边的无向连通图)



有根树

因此，以树根为源点时，每个节点都只会从父节点获得水源，并流向它的子节点。每个节点的“流域”就是以该点为根的子树。这非常符合树形 DP 的应用场景——每棵子树都是一个“子问题”。

设 $D_s[x]$ 表示在以 x 为根的子树中，把 x 作为源点，从 x 出发流向**子树**的流量最大是多少。

$$D_s[x] = \sum_{y \in Son(x)} \begin{cases} \min(D_s[y], c(x,y)) & y\text{的度数} > 1 \\ c(x,y) & y\text{的度数} = 1 \end{cases}$$

对于枚举的每个源点 s，可以用树形 DP 在 $O(N)$ 的时间内求出 D_s 数组，并用 $D_s[s]$ 更新答案。所以，最终的时间复杂度为 $O(N^2)$。下面的代码给出了一次树形 DP 的过程。在主函数中调用 dp(s)，完成后 $d[s]$ 就是所求的 $D_s[s]$。

```
void dp(int x) {
   v[x] = 1; // 访问标记
```

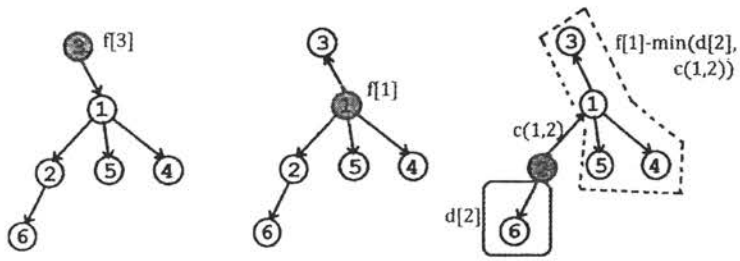
```
    d[x] = 0;
    for (int i = head[x]; i; i = Next[i]) { // 邻接表存储
        int y = ver[i];
        if (v[y]) continue;
        dp(y);
        if (deg[y] == 1) d[x] += edge[i]; // edge[i]保存 c(x,y)
        else d[x] += min(d[y], edge[i]);
    }
}
```

用“二次扫描与换根法”代替源点的枚举，可以在 $O(N)$ 的时间内解决整个问题。首先，我们任选一个源点作为根节点，记为 $root$，然后采用上面的代码进行一次树形 DP，求出 D_{root} 数组，简记为 D 数组。

设 $F[x]$ 表示把 x 作为源点，流向整个水系，流量最大是多少。对于根节点 $root$，显然有 $F[root]=D[root]$。

假设 $F[x]$ 已被正确求出，考虑其子节点 y，$F[y]$ 尚未计算。$F[y]$ 包含两部分。

1. 从 y 流向以 y 为根的子树的流量，已经计算并保存在 $D[y]$ 中。
2. 从 y 沿着到父节点 x 的河道，进而流向水系中其他部分的流量。



以3为根　　以1为根　　以2为根

因为把 x 作为源点的总流量为 $F[x]$，从 x 流向 y 的流量为 $\min\big(D[y],c(x,y)\big)$ 所以从 x 流向除 y 以外其他部分的流量就是二者之差。于是，把 y 作为源点，先流到 x，再流向其他部分的流量就是把这个“差”再与 $c(x,y)$ 取最小值后的结果。

当然，对于度数为 1 的节点 x，需要进行特殊判断。

$$F[y]=D[y]+\begin{cases}\min\Big(F[x]-\min\big(D[y],c(x,y)\big),c(x,y)\Big) & x,y\text{ 的度数}>1\\ \min\big(F[x]-c(x,y),c(x,y)\big) & x\text{ 度数}>1,y\text{ 度数}=1\\ c(x,y) & x\text{ 的度数}=1\end{cases}$$

$F[y]$ 就是把源点（树根）从 x 换成 y 后，流量的计算结果。这是一个自顶向下的递推方程，我们通过一次深度优先遍历即可实现。

```
void dfs(int x) {
    v[x] = 1;
    for (int i = head[x]; i; i = Next[i]) {
        int y = ver[i];
        if (v[y]) continue;
        if (deg[x] == 1) f[y] = d[y] + edge[i];
        else if (deg[y] == 1)
            f[y] = d[y] + min(f[x] - edge[i], edge[i]);
        else
            f[y] = d[y] + min(f[x] - min(d[y], edge[i]), edge[i]);
        dfs(y);
    }
}

// 主函数中的调用
int root = 1;
dp(root);
memset(v, 0, sizeof(v));
f[root] = d[root];
dfs(root);
int ans = 0;
for (int i = 1; i <= n; i++) ans = max(ans, f[i]);
cout << ans <<endl;
```

0x55 环形与后效性处理

环形结构上的动态规划问题

在许多环形结构的问题中，我们都能够通过枚举法，选择一个位置把环断开，变成线性结构进行计算，最后根据每次枚举的结果求出答案。我们把能通过上述枚举方式求解的环形问题称为“可拆解的环形问题”，这也是本节的主要研究对象。我们的目标是采取适当的策略避免枚举，从而使时间复杂度得到降低。

例题 Naptime AcWing288

在某个星球上，一天由 N 个小时构成，我们称 0 点到 1 点为第 1 个小时、1 点到 2 点为第 2 个小时，依此类推。在第 i 个小时睡觉能够恢复 U_i 点体力。在这个星球上住着一头牛，它每天要休息 B 个小时。它休息的这 B 个小时不一定连续，可以分成若干段，但是在每段的第 1 个小时，它需要从清醒逐渐入睡，不能恢复体力，从下一个小时开始才能睡着。

为了身体健康，这头牛希望遵循生物钟，每天采用相同的睡觉计划。另外，因为时间是连续的，即每一天的第 N 个小时和下一天的第 1 个小时是相连的（N 点等于 0 点），这头牛只需要在每 N 个小时内休息够 B 个小时就可以了（就像地球上的人类，如果每天需要休息 9 个小时，那么可以 23 点入睡、6 点起床，13 点午睡、15 点起床，这样能够恢复体力的时间就是从 0~6 点和 14~15 点，共 7 个小时）。

请你帮忙给这头牛安排一个睡觉计划，使它每天恢复的体力最多。$1 \le N \le 3830$。

先来考虑一个简化的问题。假设第 N 个小时和下一天的第 1 个小时不是相连的，这样星球上每天的时间就是线性的，从第 1 个小时开始，到第 N 个小时结束，可以用一个长度为 N 的序列 $\{U_i\}$ 来描述一天内每小时的体力恢复情况。我们从这个序列中选出 B 项，按照题目中的计算方法，让一天里恢复的体力尽量多。利用前面几节学到的线性 DP 的知识，容易设计出状态表示和转移方程。

设 $F[i,j,1]$ 表示前 i 个小时休息了 j 个小时，并且第 i 个小时正在休息，累计恢复的体力的最大值。$F[i,j,0]$ 表示前 i 个小时休息了 j 个小时，并且第 i 个小时没

有在休息，累计恢复的体力的最大值。

$$F[i,j,0]=\max(F[i-1,j,0],F[i-1,j,1])$$

$$F[i,j,1]=\max(F[i-1,j-1,0],F[i-1,j-1,1]+U_i)$$

在这个简化的问题中，第 1 个小时一定是每天的开始，不可能处于熟睡状态。因此 DP 的初值为 $F[1,0,0]=0$，$F[1,1,1]=0$，其余均为负无穷。

目标为：$\max(F[N,B,0],\ F[N,B,1])$。

到目前为止，我们解决的“线性问题”仅比原来的“环形问题”少了一种情况，即在第 1 个小时熟睡，并获得 U_1 点体力。如果在原问题的最优解中，第 N 个小时与第 1 个小时不都在休息，那么上面简化问题的答案即为所求。为了补充这个缺少的情况，可以强制令第 N 个小时和第 1 个小时都在休息，使第 1 个小时能够恢复体力。我们仍然采用上述线性 DP 的方法，只需把初值和目标修改为：

初值：$F[1,1,1]=U_1$，其余均为负无穷。

目标：$F[N,B,1]$。

两次动态规划结果中的较优者就是原问题的答案。

本题的解法本质上是把问题拆成了两部分。第一部分不可能在第 1 小时恢复体力 U_1，而第二部分强制在第 1 小时获得 U_1 点体力。这两部分合起来能够覆盖整个问题。无论是哪一部分，因为第 N 小时和第 1 小时之间的特殊关系被确定，所以我们就能把环拆开，用线性 DP 计算。这就是我们要介绍的第一种策略——执行**两次** DP，第一次在任意位置把环断开成链，按照线性问题求解；第二次通过适当的条件和赋值，保证计算出的状态等价于把断开的位置强制相连。

我们要介绍的第二种策略是在任意位置**把环断开成链**，然后**复制 1 倍接在末尾**。我们在 0x53 节“区间 DP”探讨过一道这样的例题，请读者回顾。下面再给出一道采用本策略的例题。

例题 环路运输 AcWing289

在一条环形公路旁均匀地分布着 N 座仓库，编号为 1~N，编号为 i 的仓库与编号为 j 的仓库之间的距离定义为 $dist(i,j)=\min(|i-j|,N-|i-j|)$，也就是逆时针或顺时针从 i 到 j 中较近的一种。每座仓库都存有货物，其中编号为 i 的仓库库存量为 A_i。在 i 和 j 两座仓库之间运送货物需要的代价为 $A_i+A_j+dist(i,j)$。求在哪两座仓库之间运送货物需要的代价最大。$1\le N\le 10^6$。

我们在任意位置（例如仓库 1 和 N 之间）把环断开，复制 1 倍接在末尾，形成长度为 $2N$ 的直线公路。在转化之后的模型中，公路旁均匀分布着 $2N$ 座仓库，其中 $A_i=A_{i+N}\ (1\le i\le N)$。

对于原来环形公路上的任意两座仓库 i 和 $j(1\le j<i\le N)$，如果 $i-j\le N/2$，



那么在新的直线公路上，仍然可以对应成在 i 和 j 之间运送货物，代价为 $A_i + A_j + i - j$。

如果 $i - j > N/2$，那么可以对应成在 i 和 $j + N$ 之间运送货物，代价为 $A_i + A_{j+N} + j + N - i$，其中 $j + N - i = N - (i - j) \le N/2$。

综上所述，原问题可以等价转化为：长度为 $2N$ 的直线公路上，在满足 $1 \le j < i \le 2N$ 并且 $i - j \le N/2$ 的哪两座仓库 i 和 j 之间运送货物，运送代价 $A_i + A_j + i - j$ 最大？

我们可以枚举 i，对于每个 i，需要找到一个 $j \in [i - N/2, i - 1]$，使 $A_j - j$ 尽量大。在 0x12 节“最大子序和”这道例题的解答中，我们已经探讨过同样的问题。使用单调队列进行维护，可以在均摊 $O(1)$ 的时间内找到这样的 j。整个算法的时间复杂度为 $O(N)$。

有后效性的状态转移方程

从最初学习 DP 开始，我们就多次强调，“阶段”是动态规划的三要素之一，“无后效性”是应用动态规划算法的三前提之一。事实上，在一些题目中，当我们根据题目的关键点抽象出“状态维度”，并设计出状态表示和状态转移方程后，却发现这道形似 DP 的题目不满足“无后效性”这一基本条件——部分状态之间互相转移、互相影响，构成了环形，无法确定出一个合适的 DP“阶段”，从而沿着某个方向执行递推。

事实上，我们可以把动态规划的各状态看作未知量，状态的转移看作若干个方程。如果仅仅是“无后效性”这一条前提不能满足，并且状态转移方程都是一次方程，那么我们可以不进行线性递推，而是用高斯消元直接求出状态转移方程的解。

在更多的题目中，动态规划的状态转移“分阶段带环”——我们需要把 DP 和高斯消元相结合，在整体层面采用动态规划框架，而在局部使用高斯消元解出互相影响的状态。我们用一道例题来具体说明这类情况。

例题 Broken Robot AcWing290

给定一张 $N * M$ 的棋盘，有一个机器人处于 (x, y) 位置。这个机器人可以进行很多轮行动，每次等概率地随机选择停在原地、向左移动一格、向右移动一格或向下移动一格。当然机器人不能移出棋盘。求机器人从起点走到最后一行的任意一个位置上，所需行动次数的数学期望值。$1 \le N, M \le 1000$。

在 0x51 节“线性 DP”的“传纸条”一题中，因为只能往右和往下传递，所以 DP 状态在行、列两个维度上都构成线性增长的形式。最后我们以传递的步数作为 DP 的阶段。在本题中，我们容易想到，设 $F[i, j]$ 表示机器人从位置 (i, j) 走到最后一行，所需行动次数的期望值。

机器人在第 1 列时不能再向左走，$F[i,1]=\frac{1}{3}(F[i,1]+F[i,2]+F[i+1,1])+1$。

同理，在第 M 列时，$F[i,M]=\frac{1}{3}(F[i,M]+F[i,M-1]+F[i+1,M])+1$。

当 $2\le j\le M-1$ 时，$F[i,j]=\frac{1}{4}(F[i,j]+F[i,j-1]+F[i,j+1]+F[i+1,j])+1$

初值：$\forall j\in[1,M]$，$F[N,j]=0$。目标：$F[x,y]$。

观察上面的状态转移方程，我们发现，因为第 i 行的状态只能转移到第 $i+1$ 行，所以“行”维度仍然满足“无后效性”。但是在同一行内，机器人既能向左走，也能向右走，甚至可以原地不动，各状态之间互相转移，无法确定一个递推顺序。故我们采用 DP 套高斯消元的算法来求解本题。

我们先以行号为阶段，从 M 到 x 倒序扫描每一行，依次计算以该行的每个位置为起点走到最后一行，所需行动次数的数学期望值。

接下来我们考虑每行内的计算方法。在计算第 i 行的状态时，因为第 $i+1$ 行的状态已经计算完毕，我们可以把 $F[i+1,j]$ 看作已知数。于是，状态转移方程中就只剩下 $F[i,1],F[i,2],\cdots,F[i,M]$ 这 M 个未知量。第 i 行的每个位置可以列出一个方程，共 M 个方程。因此，我们可以用高斯消元解出 $F[i,1],F[i,2],\cdots,F[i,M]$ 的值。

列出上述 M 个方程，移项并仔细观察，可以发现高斯消元的系数矩阵除主对角线和对角线两侧的斜线之外，其余部分均为 0。下面就是一个 $M=5$ 时的系数矩阵。

$$\begin{matrix}-2/3 & 1/3 & 0 & 0 & 0\\ 1/4 & -3/4 & 1/4 & 0 & 0\\ 0 & 1/4 & -3/4 & 1/4 & 0\\ 0 & 0 & 1/4 & -3/4 & 1/4\\ 0 & 0 & 0 & 1/3 & -2/3\end{matrix}$$

在这样的矩阵中执行高斯消元，在每一行中实际上只有 2~3 个位置需要相减，只需 $O(M)$ 的时间即可求出各个未知量的解。整个算法的时间复杂度为 $O(NM)$。注意 $M=1$ 时，上述 DP 算法有边界问题，需要进行特判。

值得一提的是，为什么我们用 $F[i,j]$ 表示从 (i,j) 走到最后一行的期望步数，按行号倒序进行递推，而不是用 $F[i,j]$ 表示 (x,y) 到 (i,j) 的期望步数，并正序递推？原因是，根据数学期望的定义，若正序递推，则还需求出 (x,y) 到最后一行每个位置的概率 $p[N,j]$，计算 $\sum_{j=1}^{M}p[N,j]*F[N,j]$ 才能得到答案，较为复杂。**事实上，很多数学期望 DP 都会采取倒推的方式执行**，请读者留心思考。

通过本章前五节的学习，读者应该对动态规划的本质、动态规划的设计与实现方法有了更加深刻的认识。读者应能在线性（包括背包、区间）、树形等各种结构上初步判断 DP 是否适用，解释 DP 的思想，并熟练使用 DP 算法求解不太困难的问题。遇到简



单的环形问题和后效性问题，读者也能够从容应对。

从下一节开始，我们将把重点放在动态规划的优化上。大部分情况下，我们会直接给出 DP 状态表示与转移方程，切入 DP 优化的正题，不再花费过多篇幅解释设计思路。因此，请读者根据自身水平，尽量完成 0x5E 节的第 2~16 题，务必把所学知识掌握扎实。

0x56 状态压缩 DP

在 0x51 节“线性 DP”中，我们提到，动态规划的过程是随着“阶段”的增长，在每个状态维度上不断扩展的。在任意时刻，已经求出最优解的状态与尚未求出最优解的状态在各维度上的分界点组成了 DP 扩展的“轮廓”。对于某些问题，我们需要在动态规划的“状态”中记录一个集合，保存这个“轮廓”的详细信息，以便进行状态转移。若集合大小不超过 N，集合中每个元素都是小于 K 的自然数，则我们可以把这个集合看作一个 N 位 K 进制数，以一个 $[0,K^N-1]$ 之间的十进制整数的形式作为 DP 状态的一维。这种**把集合转化为整数记录在 DP 状态中**的一类算法，被称为**状态压缩动态规划**算法。

例题 最短 Hamilton 路径 AcWing91

给定一张 $n(n \le 20)$ 个点的带权无向图，点从 $0 \sim n-1$ 标号，求起点 0 到终点 $n-1$ 的最短 Hamilton 路径。

Hamilton 路径的定义是从 0 到 $n-1$ 不重不漏地经过每个点恰好一次。

很容易想到本题的一种朴素做法，就是枚举 n 个点的全排列，计算路径长度取最小值，时间复杂度为 $O(n*n!)$，使用二进制状态压缩 DP 可以优化到 $O(n^2*2^n)$。

在任意时刻如何表示哪些点已经被经过，哪些点没有被经过？可以使用一个 n 位二进制数，若其第 i 位 $(0 \le i < n)$ 为 1，则表示第 i 个点已经被经过，反之未被经过。在任意时刻还需要知道当前所处的位置，我们可以使用 $F[i,j]$ $(0 \le i < 2^n, 0 \le j < n)$ 表示“点被经过的状态”对应的二进制数为 i，且目前处于点 j 时的最短路径。

整个状态空间可看作 N 维，每维代表一个节点，只有 0（尚未访问）和 1（已访问）两个值。我们可以想象 DP 的“轮廓”以“访问过的节点数目”为阶段，从 $(0,0,\cdots,0)$ 扩展到 $(1,1,\cdots,1)$。为了记录当前状态在每个维度上的坐标是 0 还是 1，我们使用了一个 N 位二进制数，即 $[0,2^N-1]$ 之间的十进制整数存储节点的访问情况。另外，为了知道最后经过的节点是哪一个，我们把该节点编号作为附加信息，也保存在 DP 的状态中。因此，该状态压缩 DP 的状态数组就由大小分别为 2^N 和 N 的两个维度构成。

在起点时，有 $F[1,0]=0$，即只经过了点 0（i 只有第 0 位为 1），目前处于起点 0，最短路径长度为 0。为了方便起见，我们将 F 数组其他的值设为无穷大。目标是

$F[(1 \ll n)-1, n-1]$，即经过所有点（i 的所有位都是 1），处于终点 $n-1$ 的最短路径。

在任意时刻，有状态转移方程 $F[i,j] = \min\{F[i \text{ xor } (1 \ll j), k] + \text{weight}(k,j)\}$，其中 $0 \le k < n$ 并且 $((i \gg j) \& 1) = 1$，即当前时刻“被经过的点的状态”对应的二进制数为 i，处于点 j。因为 j 只能被恰好经过一次，所以一定是刚刚经过的，故在上一时刻“被经过的点的状态”对应的二进制数的第 j 位应该赋值为 0，也就是 $i \text{ xor } (1 \ll j)$。另外，上一时刻所处的位置可能是 $i \text{ xor } (1 \ll j)$ 中任意一个是 1 的数位 k，从 k 走到 j 需经过 $\text{weight}(k,j)$ 的路程，可以考虑所有这样的 k 取最小值。

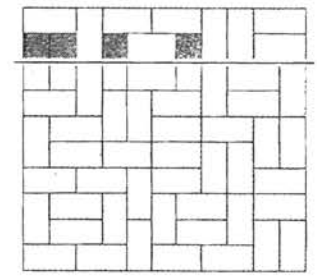
```
int f[1 << 20][20];
int hamilton(int n, int weight[20][20]) {
    memset(f, 0x3f, sizeof(f));
    f[1][0] = 0;
    for (int i = 1; i < 1 << n; i++)
        for (int j = 0; j < n; j++) if (i >> j & 1)
            for (int k = 0; k < n; k++) if ((i^1<<j) >> k & 1)
                f[i][j] = min(f[i][j], f[i^1<<j][k]+weight[k][j]);
    return f[(1 << n) - 1][n - 1];
}
```

例题 **Mondriaan's Dream** AcWing291

求把 $N * M (1 \le N, M \le 11)$ 的棋盘分割成若干个 1*2 的长方形，有多少种方案。例如：当 $N=2$, $M=4$ 时，共有 5 种方案；当 $N=2$, $M=3$ 时，有 3 种方案。如下图所示。



对于任意一种方案，考虑以某一行为界，把整个棋盘横着分成两半，如右图所示。在上半部分最后一行中：



1. 每个灰色背景的部分都是一个竖着的 1*2 长方形的一半，决定了下一行必须继续补全该长方形。

2. 其余部分对下半部分的分割方法没有影响。下一行既可以在连续两个位置安排一个横着的 1*2 长方形，也可以让某个位置作为一个竖着的 1*2 长方形的一半。

综上所述，我们可以把“行号”作为 DP 的“阶段”，把“上半部分”不断向下扩展，直至确定整个棋盘的分割方法。为了描述上半部分最后一行的详细形态，我们可以使用一个 M 位二进制数，其中第 $k (0 \le k < M)$ 位为 1 表示第 k 列是一个竖着的 1*2 长方形的上面一半，第 k 位为 0 表示其他情况。

设 $F[i,j]$ 表示第 i 行的形态为 j 时，前 i 行分割方案的总数。j 是用十进制整数记录的 M 位二进制数。

第 $i-1$ 行的形态 k 能转移到第 i 行的形态 j，当且仅当：

1. j 和 k 执行按位与运算的结果是 0。

这保证了每个数字 1 的下方必须是数字 0，代表继续补全竖着的 1*2 长方形。

2. j 和 k 执行按位或运算的结果的二进制表示中，每一段连续的 0 都必须有偶数个。

这些 0 代表若干个横着的 1*2 长方形，奇数个 0 无法分割成这种形态。

我们可以在 DP 前预处理出 $[0,2^M-1]$ 内所有满足“二进制表示下每一段连续的 0 都有偶数个”的整数，记录在集合 S 中。

$$F[i,j]=\sum_{j\&k=0\ 并且\ j|k\in S} F[i-1,k]$$

初值：$F[0,0]=1$，其余均为 0。

目标：$F[N,0]$。

时间复杂度为 $O(2^M 2^M N)=O(4^M N)$。

```
int n, m;
long long f[12][1 << 11];
bool in_s[1 << 11];
int main()  {
   while (cin >> n >> m && n) {
      for (int i = 0; i < 1 << m; i++) {
         bool cnt = 0, has_odd = 0;
         for (int j = 0; j < m; j++)
            if (i >> j & 1) has_odd |= cnt, cnt = 0;
            else cnt ^= 1;
         in_s[i] = has_odd | cnt ? 0 : 1;
      }
      f[0][0] = 1;
      for (int i = 1; i <= n; i++)
         for (int j = 0; j < 1 << m; j++) {
            f[i][j] = 0;
            for (int k = 0; k < 1 << m; k++)
               if ((j&k) == 0 && in_s[j | k])
                  f[i][j] += f[i - 1][k];
         }
```

```
        cout << f[n][0] << endl;
    }
}
```

例题 炮兵阵地 AcWing292

司令部的将军们打算在 $N*M$ 的网格地图上部署他们的炮兵部队。一个 $N*M$ 的地图由 N 行 M 列组成，地图的每一格可能是山地（用“H”表示），也可能是平原（用“P”表示），如右图所示。在每一格平原地形上最多可以部署一支炮兵部队（山地上不能够部署炮兵部队），一支炮兵部队在地图上的攻击范围如图中黑色区域所示。

P	P	H	P	H	H	P	P
P	H	P	P	P	H	P	P
P	P	P	P	H	H	P	H
H	H	H	P	P	P	P	H
H	P	P	P	P	H	P	H
H	P	P	P	P	H	H	P
H	H	H	P	P	P	P	H

如果在地图中的灰色所标识的平原上部署一支炮兵部队，则图中的黑色网格表示它能够攻击到的区域：沿横向左右各两格，沿纵向上下各两格。图上其他白色网格均攻击不到。从图上可见炮兵的攻击范围不受地形的影响。

现在，将军们规划如何部署炮兵部队，在防止误伤的前提下（保证任何两支炮兵部队之间不能互相攻击，即任何一支炮兵部队都不在其他支炮兵部队的攻击范围内），在整个地图区域内最多能够部署多少支我军的炮兵部队。$N \le 100$, $M \le 10$。

本题与上一题类似，都是在矩形网格中放置图形的问题。上一题是求放满 1*2 的长方形的方案数，本题则是求最多能放多少个“十字形状”，并且每个“十字”的中心都不被其他“十字”覆盖。因此，我们还是采用按“行号”为阶段的 DP 方法。

解法一

因为每个位置能否放置炮兵与它上面两行对应位置上是否放置炮兵有关，所以在向第 i 行的状态转移时，需要知道第 $i-1$ 行和第 $i-2$ 行的状态。我们把每一行的状态看作一个 M 位二进制数，用一个 $0 \sim 2^M-1$ 之间的十进制整数存储，其中第 $p(0 \le p < M)$ 位为 1 表示该行第 p 列放置了炮兵，为 0 则表示没有放置炮兵。

我们在 DP 前预处理出集合 S，存储“相邻两个 1 的距离不小于 3”的所有 M 位二进制数，这些二进制数代表每一行中两个炮兵的距离不能小于 3。

设 $\text{count}(x)$ 表示 M 位二进制数 x 中 1 的个数。

设 $\text{valid}(i,x)$ 表示 M 位二进制数 x 属于集合 S，并且 x 中的每个 1 对应在地图第 i 行中的位置都是平原。

设 $F[i,j,k]$ 表示第 i 行压缩后的状态为 j，第 $i-1$ 行压缩后的状态为 k 时，前 i 行最多能放置多少个炮兵。

$F[i,j,k]$

$$= \begin{cases} \max\limits_{j\&l=0}\{F[i-1,k,l]\} + \text{count}(j) & \text{如果 valid}(i,j), \text{valid}(i-1,k) \text{ 并且 } j\&k=0 \\ -\infty & \text{否则} \end{cases}$$

初值：$F[0,0,0]=0$，其余为负无穷。目标：$\max\limits_{j\&k=0,\text{valid}(N,j),\text{valid}(N-1,k)}\{F[N,j,k]\}$。

虽然 M 位二进制数有 2^M 个，但只有集合 S 中的数才可能是合法状态。通过写程序预处理可以发现，事实上 S 集合非常小，仅包含不到 100 个数。我们可以对集合 S 中的数离散化，只对这些状态进行存储和遍历。时间和空间复杂度均为 $O(N|S|^3)$。

解法二

上一个解法状态压缩以后的规模本来是 $O(N*4^M)$，我们利用题目的特殊性排除掉了不合法的状态，才使解法变得高效。接下来我们再介绍一种更通用的做法。这种做法仍以“行”作为动态规划的阶段，但用更高进制的状态压缩代表一行的状态，从而区分网格中不同位置的不同属性。

根据题意，一个格子放置炮兵以后，该格子上下两行的同一列都不能再放置炮兵。可以用数字 2 表示放置炮兵的格子，规定 2 的下面必须是 1，1 的下边必须是 0，只有 0 的下边可以放置新的炮兵。在炮兵不会误伤的情况下，每个“十字”攻击范围形如：

$$\begin{array}{ccccc} & & \le 1 & & \\ & & 0 & & \\ \le 1 & \le 1 & 2 & \le 1 & \le 1 \\ & & 1 & & \\ & & 0 & & \end{array}$$

把每行的状态压缩为一个 M 位三进制数，用 $0\sim3^M-1$ 之间的十进制整数存储。设 $F[i,j]$ 表示第 i 行压缩后的状态为 j 时，前 i 行最多能放多少个炮兵，并且前 i 行的炮兵不会误伤。状态规模为 $O(N*3^M)$，不会超过内存限制。

对于每个已经求出的合法的 $F[i,j]$，我们考虑它能转移到哪些状态。这等价于依次考虑第 $i+1$ 行的每个位置填写什么数字。需要满足四个条件。

1. 当第 i 行第 j 列为 2 时，第 $i+1$ 行第 j 列必须填 1。
2. 当第 i 行第 j 列为 1 时，第 $i+1$ 行第 j 列必须填 0。
3. 山地格子不能填 2。
4. 一个格子填 2 以后，它右边的两个格子不能再填 2。

像本题这种状态表示比较复杂、冗余较多的题目，我们不一定非要写出确切的状态转移方程。可以通过深度优先搜索（DFS），在保证上述四个条件的前提下，搜索第 $i+1$ 行的每个位置填写什么数字。在到达搜索边界时（M 个位置都填完），就得到了一个状态 k，从而可以从 $F[i,j]$ 转移到 $F[i+1,k]$。总而言之，整个动态规划算法使用三进制压缩的状态表示，以“行号”为阶段，在相邻两行之间使用 DFS 进行转移。

上面两道例题都是“填充网格图形”类的题目，并且填充的图形仅与相邻的若干



行有关，容易按照“行号”划分阶段，从而应用状态压缩动态规划算法。有些更为复杂的题目，填充的图形与整个网格有关，就需要进一步提炼动态规划过程中“轮廓”的特点。这超出了我们的讨论范围，学有余力的读者可以自行查阅论文[①]，学习“基于连通性的状态压缩动态规划”（简称“插头 DP”）。

例题 宝藏 AcWing529

参与考古挖掘的小明得到了一份藏宝图，藏宝图上标出了 n 个深埋在地下的宝藏屋，也给出了这 n 个宝藏屋之间可供开发的 m 条道路和它们的长度。

小明决心亲自前往挖掘所有宝藏屋中的宝藏。但是，每个宝藏屋距离地面都很远，也就是说，从地面打通一条到某个宝藏屋的道路是很困难的，而开发宝藏屋之间的道路则相对容易很多。

小明的决心感动了考古挖掘的赞助商，赞助商决定免费赞助他打通一条从地面到某个宝藏屋的通道，通往哪个宝藏屋则由小明来决定。

在此基础上，小明还需要考虑如何开凿宝藏屋之间的道路。已经开凿出的道路可以任意通行且不消耗代价。每开凿出一条新道路，小明就会与考古队一起挖掘出由该条道路所能到达的宝藏屋的宝藏。另外，小明不想开发无用道路，即两个已经被挖掘过的宝藏屋之间的道路无需再开发。

新开发一条道路的代价等于以下二者之积。

1. 这条道路的长度。

2. 从“赞助商帮你打通的宝藏屋”到“这条道路起点的宝藏屋”所经过的宝藏屋的数量（包括两端）。

请你编写程序为小明选定由赞助商打通的宝藏屋和之后开凿的道路，使得工程总代价最小，并输出这个最小值。

数据范围：$1 \le n \le 12$，$0 \le m \le 1000$。道路长度：≤ 500000。

根据题意，已经开凿出的道路可以任意通行而不消耗代价，故只需花费最小的代价让所有宝藏屋连通即可。因此，最终修建的道路必定构成一棵树。如若不然，一定存在一条多余的道路，删去它之后宝藏屋仍然连通。而“赞助商帮你打通的宝藏屋”就是这棵树的根。

于是，我们很容易想到搜索算法。

1. 任选一个点为根，即“由赞助商帮你打通的宝藏屋”。

2. 任选一个已经打通的宝藏屋 x，从它出发开凿一条道路，到达一个尚未打通的宝藏屋 y。打通 y 的代价为道路 (x,y) 的长度乘节点 x 的深度。然后继续搜索。

如果在每次递归过程中，都扫描所有已经打通的宝藏屋，尝试开凿一条道路，那么

① 陈丹琦《基于连通性的状态压缩动态问题》，https://cs.stanford.edu/~danqi/misc/dynamic-programming.pdf，配套资源提供了该网页的快照。

该算法显然遍历了相当多的重复状态，时间复杂度非常高。为此，可以做出两点限制。

1. 先打通浅的宝藏屋，再打通深的宝藏屋。这是因为在合法的前提下，对于同一个树形结构，打通宝藏屋的代价与顺序是无关的。

2. 设已经打通的宝藏屋集合为 S。对于相同的集合 S，我们只关心代价最小的那一个。这是因为打通后续宝藏屋的代价与 S 的内部连接方式无关，只要集合 S 是连通的就足够了。

不难观察到，第一点限制从浅到深形成了动态规划的“阶段”，而第二点限制正是动态规划的“最优子结构”性质。为了把“集合”作为动态规划的“状态”，可以采取状态压缩的方法。这启发我们使用状态压缩动态规划算法解决本题。

解法一：二进制状态压缩 DP，每次扩展一层

设 $F[i,j]$ 表示已经打通的宝藏屋最大深度为 i，宝藏屋的打通状态为 n 位二进制数 j（其中对应位为 1 表示该宝藏屋已经打通，0 表示未打通）时，目前花费的最小代价。状态转移方程：

$$F[i,j]=\min_{\text{valid}(k,j)}\{F[i-1,k]+(i-1)*\text{cost}(k,j)\}$$

初值：$\forall x\in[1,n]$，$F[1,1\ll(x-1)]=0$，其余为正无穷。

目标：$\min\limits_{1\le i\le n}\{F[i,(1\ll n)-1]\}$。

这是一个按层 DP 的方程，每次考虑第 i 层相比第 $i-1$ 层新增了哪些节点。其中 $\text{valid}(k,j)$ 表示从状态 k 能否通过增加若干条道路变为状态 j，且只扩展了一层。$\text{cost}(k,j)$ 表示增加的道路长度总和的最小值。$\text{valid}(k,j)$ 为真当且仅当满足下列两个条件。

1. 状态 k 对应的节点集合是 j 的子集，即 $j\,\&\,k=k$，& 表示按位与运算。

2. 状态 j 对应的节点集合是 $\text{expand}(k)$ 的子集，即 $j\,\&\,\text{expand}(k)=j$。其中，集合 $\text{expand}(k)$ 表示从 k 中所有节点出发，向下开凿一层以内的所有道路之后，处于被打通状态的所有节点构成的集合。

函数 expand 可以通过预处理得到。只需枚举 k，遍历 k 中节点出发的所有边即可。同时可以求出从 k 中节点扩展到新打通节点 x 的最短道路，记为 $\text{road}(k,x)$。

对于函数 valid 和 cost，首先枚举 j，然后枚举 j 的子集 k，利用 expand 函数可以预处理出对于每个 j，所有使 $\text{valid}(k,j)$ 为真的 k，存储在 STL vector 中。比较 j 和 k 的差别，对若干个 road 值求和，即可进一步预处理出对应的 $\text{cost}(k,j)$，也存储在 STL vector 中。详细实现可以查阅配套资源中的参考程序。有了这些预处理的函数值，我们就能快速地进行动态规划了。

最后，有些读者可能还有一个疑问：如何得知 k 中哪些节点深度为 $i-1$，哪些节点深度更小呢？我们不妨认为 k 中所有节点都可以扩展，并且扩展代价的深度部分

都是 $i-1$，就得到了上面的状态转移方程。这不会影响答案，因为深度小于 $i-1$ 的节点在更早的时候扩展的情况已经在其他状态中计算过了，此时即使这些节点发生扩展，并把深度代价视为 $i-1$，也一定不优，不会干扰最小值的计算。这也是最优化问题常用的等效手法之一。

整个算法的时间复杂度为 $O(N*3^N+M*2^N)$。其中 3^N 为所有 2^N 个状态的子集总数，这是由二项式定理 $2^N+C_N^1*2^{N-1}+C_N^2*2^{N-2}+\cdots+1=(1+2)^N=3^N$ 得到的。$M*2^N$ 是预处理 $\text{expand}(k)$ 和 $\text{road}(k,x)$ 消耗的时间。

解法二：三进制状态压缩 DP，每次扩展一条道路

上一个做法一次转移一层，枚举了每层内打通的节点（开凿的道路）。为了高效地转移，需要预处理多个函数值辅助计算。我们自然会想到：能否重新设计一个状态转移方程，每次只打通一个节点（开凿一条道路）？

因为计算代价时需要知道道路起点的深度，所以我们仍然以深度为阶段按层 DP，每次尝试扩展一个前 $i-1$ 层的节点（深度代价可以视为 $i-1$）。为了区分已经打通的节点中，哪些是前 $i-1$ 层的（可以扩展），哪些是第 i 层的（刚打通，不能以 $i-1$ 的深度代价扩展），可以采用三进制状态压缩动态规划。

设 $F[i,j]$ 表示前 i 层，节点的打通状态为 j 时，已经耗费的最小代价。j 是一个三进制数，它的第 $x-1$ 位为 0 表示节点 x 尚未打通，为 1 表示节点 x 是在前 i 层打通的，为 2 表示节点 x 是开凿了一条第 i 层通向第 $i+1$ 层的道路而打通的。

初值：$\forall x\in[1,n]$，$F[1,3^{x-1}]=0$，其余为正无穷。

目标：$F[n+1,(111\cdots1)_3]$。

对于每个已经求出的 $F[i,j]$，考虑它能转移到哪些状态。状态转移分为三类。

1. 不再从第 i 层出发开凿道路，进入下一层。设 s 是把 j 中所有为 2 的位改为 1 得到的三进制数，用 $F[i,j]$ 更新 $F[i+1,s]$。

2. 考虑扩展 j 最高位的 1 对应的节点 x。枚举从 x 出发的所有道路，若到达一个尚未打通的节点 y（即 j 的第 $y-1$ 位为 0），则用 $F[i,j]+i*\text{length}(x,y)$ 更新 $F[i,j+2*3^{y-1}]$。

3. 不再扩展 j 最高位的 1 对应的节点 x。此时可以把该位赋值为 2 表示不能扩展，也就是用 $F[i,j]$ 更新 $F[i,j+3^{x-1}]$。

之所以只考虑 j 最高位的 1，是因为 j 所有为 1 的位是等价的，无论如何处理最高位，都会转移到同一层内一个压缩值更大的状态，会在以后再次循环到，从而继续扩展较低的位。故此时没有必要全部枚举，以防造成许多冗余的转移。我们还可以使用滚动数组优化空间复杂度。最终的时间复杂度为 $O(N^2*3^N)$，刚好通过本题。三进制状态压缩不能使用位运算，需要预处理 3 的次幂并使用乘除法计算，速度较慢，程序实现时要注意时间常数。

0x57 倍增优化 DP

在 0x06 节中，我们已经结合递推介绍了倍增思想，请读者回顾。因为动态规划经常采用按阶段的递推形式实现，所以也可以按照类似的方法，使用倍增把阶段的线性增长优化为成倍增长。

事实上，0x06 节提到的区间最值问题的 ST 算法也可以用倍增优化 DP 的方式理解。在 ST 算法中，$F[i,j]$ 表示数列 A 在子区间 $[i,i+2^j-1]$ 里的最大值，即 $\max\limits_{i\le k<i+2^j}\{A_k\}$。这个动态规划的“阶段”就是区间的长度（成倍增长），区间的左端点 i 是一个附加维度。因此在 ST 算法的实现中，外层循环为区间长度以 2 为底的对数，内层循环为左端点。在状态转移时，从长度为 2^{i-1} 的阶段转移到长度为 2^i 的阶段。

在本节中，我们再以两道例题对动态规划中倍增优化的运用进行探讨。

例题 开车旅行 AcWing293

小 A 和小 B 决定利用假期外出旅行，他们将想去的城市从 1 到 N 编号，且编号较小的城市在编号较大的城市的西边，已知各个城市的海拔高度互不相同，记城市 i 的海拔高度为 H_i，城市 i 和城市 j 之间的距离 $d[i,j]$ 恰好是这两个城市海拔高度之差的绝对值，即 $d[i,j]=|H_i-H_j|$。

旅行过程中，小 A 和小 B 轮流开车，第一天小 A 开车，之后每天轮换一次。他们计划选择一个城市 S 作为起点，一直向东行驶，并且最多行驶 X 公里就结束旅行。小 A 和小 B 的驾驶风格不同，小 B 总是沿着前进方向选择一个最近的城市作为目的地，而小 A 总是沿着前进方向选择第二近的城市作为目的地（注意：本题中如果当前城市到两个城市的距离相同，则认为离海拔低的那个城市更近）。如果其中任何一人无法按照自己的原则选择目的城市，或者到达目的地会使行驶的总距离超出 X 公里，他们就会结束旅行。

在启程之前，小 A 想知道两个问题。

1. 对于一个给定的 $X=X_0$，从哪一个城市出发，小 A 开车行驶的路程总数与小 B 行驶的路程总数的比值最小（如果小 B 的行驶路程为 0，此时的比值可视为无穷大，且两个无穷大视为相等）。如果从多个城市出发，小 A 开车行驶的路程总数与小 B 行驶的路程总数的比值都最小，则输出海拔最高的那个城市。



2. 对任意给定的 $X=X_i$ 和出发城市 S_i，求出小 A 开车行驶的路程总数以及小 B 行驶的路程总数。

$1 \le N \le 10^5$，$1 \le M \le 10^4$，$-10^9 \le H_i \le 10^9$，$0 \le X_0 \le 10^9$，$1 \le S_i \le N$，$0 \le X_i \le 10^9$，数据保证 H_i 互不相同。

先预处理出小 A 和小 B 从每个城市 i 出发，沿前进方向行驶到的下一个城市，分别记为 ga(i) 和 gb(i)。根据题意，ga(i) 就等于 $i+1\sim N$ 中使 dist(i,j) 取到最小值的城市 j，gb(i) 则等于 $i+1\sim N$ 中使 dist(i,j) 取到次小值的城市 j。我们已经在第 0x13 节的例题“邻值查找”中，用平衡树（STL set）和链表两种方法解决了该问题，请读者回顾。

本题有三个关键信息——所在城市、已行驶的天数、小 A 和小 B 各行驶的路程长度。若已知出发城市和天数，就能计算出到达城市和小 A、小B 行驶的路程长度，并且天数还能反映轮到谁开车。因此，我们在 DP 中就把“天数”作为“阶段”，所在城市作为另一维状态，并使用倍增对 DP 进行优化。

设 $f[i,j,k]$ 表示从城市 j 出发，两人共行驶 2^i 天，k 先开车，最终会到达的城市。其中 $k=0$ 或 1，0 代表轮到小 A 先开车，1 代表轮到小 B 先开车。

初值：$f[0,j,0]=\text{ga}(j)$，$f[0,j,1]=\text{gb}(j)$。

当 $i=1$ 时，因为 2^0 是奇数，所以两人从 j 出发开2^1天到达的城市，等于 k 先开2^0天，另一人 $1-k$ 再开2^0天到达的城市。

$$f[1,j,k]=f[0,f[0,j,k],1-k]$$

当 $i>1$ 时，因为 2^{i-1} 是偶数，所以前后两半路程都轮到 k 先开车。

$$f[i,j,k]=f[i-1,f[i-1,j,k],k]$$

在具体实现时，读者需要注意 ga、gb 和 f 数组到达的城市超出第 N 个城市的边界情况，这里就不再赘述。

设 $da[i,j,k]$ 表示从城市 j 出发，两人共行驶 2^i 天，k 先开车，小 A 行驶的路程总长度。

初值：$da[0,j,0]=\text{dist}(j,\text{ga}(j))$，$da[0,j,1]=0$。

当 $i=1$ 时，$da[1,j,k]=da[0,j,k]+da[0,f[0,j,k],1-k]$。

当 $i>1$ 时，$da[i,j,k]=da[i-1,j,k]+da[i-1,f[i-1,j,k],k]$。

设 $db[i,j,k]$ 表示从城市 j 出发，两人共行驶 2^i 天，k 先开车，小 B 行驶的路程总长度。

初值：$db[0,j,0]=0$，$db[0,j,1]=\text{dist}(j,\text{gb}(j))$。

当 $i=1$ 时，$db[1,j,k]=db[0,j,k]+db[0,f[0,j,k],1-k]$。

当 $i>1$ 时，$db[i,j,k]=db[i-1,j,k]+db[i-1,f[i-1,j,k],k]$。

上述动态规划算法在 $O(N\log N)$ 的时间内计算出了所有“行驶天数为 2 的整数次幂”的状态。接下来我们考虑问题 calc(S,X)，意为“从城市 S 出发最多行驶 X 公里”时，小 A 和小 B 分别行驶了多长的路程。

我们按递减顺序枚举所有 2 的整数次幂，作为行驶天数，并用动态规划中得到的状态，基于二进制划分思想来拼成 X。具体来说：

1. 初始化当前城市 $p=S$，小 A、小 B 累计行驶路程 $la=0$，$lb=0$；

2. 倒序循环 $i=\log N\sim 0$；

3. 对于每个 i，若两人从 p 出发行驶 2^i 天，累计路程仍未超过 X，即 $la+lb+da[i,p,0]+db[i,p,0]\le X$，则令 $la=la+da[i,p,0]$，$lb=lb+db[i,p,0]$，$p=f[i,p,0]$；

4. 循环结束后，la 与 lb 即为所求。

枚举起点 S_i，取小 A、小 B 行驶路程比值最小的 calc(S_i,X_0)，即可求出原问题一。原问题二就是多次询问 calc(S_i,X_i)，也可以直接计算。整个算法的时间复杂度为 $O\big((N+M)\log N\big)$。

例题 **Count The Repetitions** AcWing294

定义 conn(s,n) 为 n 个字符串 s 首尾相接形成的字符串，例如：

$$\text{conn("abc", 2)} = \text{"abcabc"}$$

称字符串 a 能由字符串 b 生成，当且仅当从字符串 b 中删除某些字符后可以得到字符串 a。例如“abdbec”可以生成“abc”，但是“acbbe”不能生成“abc”。

给定两个字符串 s_1 和 s_2，以及两个整数 n_1 和 n_2，求一个最大的整数 m，满足 conn(conn$(s_2,n_2),m)$ 能由 conn(s_1,n_1) 生成。

s_1 和 s_2 长度不超过 100，n_1 和 n_2 不大于 10^6。

首先，conn(conn$(s_2,n_2),m)=$ conn(s_2,n_2*m)。我们可以求出一个最大的整数 m' 满足 conn(s_2,m') 能由 conn(s_1,n_1) 生成。之后我们求出满足 $n_2*m\le m'$ 的最大整数 m，即为本题的答案。

m' 可能很大，其上界为 $|s_1|*n_1/|s_2|$。为了提高效率，可以用二进制拆分思想，若 $m'=2^{p_{t-1}}+\cdots+2^{p_1}+2^{p_0}$，则把 conn$(s_2,m')$ 看作由 conn$(s_2,2^{p_{t-1}})$，conn$(s_2,2^{p_{t-2}}),\cdots,$conn$(s_2,2^{p_0})$ 这 t 个字符串首尾连接形成。

注意到这 t 个字符串都是由 2 的整数次幂个 s_2 构成的。我们可以换一种思路，对于每个 $k\in[0,\log_2(|s_1|*n_1/|s_2|)]$，求出从 conn$(s_1,n_1)$ 的每个位置开始，至少需要多少个字符，才能生成 conn$(s_2,2^k)$。最后在总字符数不超过 $|s_1|*n_1$ 的前提下，用这些 2 的整数次幂拼成尽量大的 m'。

还有一个问题是 n_1 的范围也非常大。不过，conn(s_1,n_1) 是 s_1 重复 n_1 次形成的。因此，只要没有到达结尾，从 $s_1[i]$ 开始往后的字符与从 $s_1[i+k*n_1]$ 开始往后



的字符是相同的。我们不妨先假设 n_1 足够大，即在 s_1 重复无限次得到的字符串中执行计算。

综合上面的讨论，我们设 $F[i,j]$ 表示从 $s_1[i]$ 开始，至少需要多少个字符，才能生成 $\text{conn}(s_2, 2^j)$。状态转移方程为：

$$F[i,j] = F[i,j-1] + F[(i + F[i,j-1]) \bmod |s_1|, j-1]$$

其含义是，在 s_1 无限重复的字符串中，从 $s_1[i]$ 开始，先用尽量少的字符生成目标串的前半部分，紧接着在后边再用尽量少的字符生成目标串的后半部分。第二维 j 是 DP 的“阶段”，第一维 i 是附加状态。复杂度为 $O(|s_1| * \log(|s_1| * n_1/|s_2|))$。

DP 的初值 $F[i,0]$ 就是从 $s_1[i]$ 开始生成 s_2 需要的最少字符数。s_1 和 s_2 都很短，可以采用朴素算法直接计算。

在整个动态规划过程完成后，我们对 DP 得到的状态进行拼接。

1. 初始化 $x = 0$，$ans = 0$，然后从 $\log(|s_1| * n_1/|s_2|)$ 到 0 倒序循环 k。

2. 如果 $x + F[x \bmod |s_1|, k] \le |s_1| * n_1$，则令 $x = x + F[x \bmod |s_1|, k]$，$ans = ans + 2^k$。

3. 在循环完成后，ans 即为 m' 的最大值。

```
string s1, s2;
int n1, n2;
long long f[105][32];
int main() {
    cin >> s2 >> n2 >> s1 >> n1;
    for (int i = 0; i < s1.size(); i++) {
        int pos = i;
        f[i][0] = 0;
        for (int j = 0; j < s2.size(); j++) {
            int cnt = 0;
            while (s1[pos] != s2[j]) {
                pos = (pos + 1) % s1.size();
                if (++cnt >= s1.size()) {
                    cout << 0 << endl;
                    return 0;
                }
            }
            pos = (pos + 1) % s1.size();
            f[i][0] += cnt + 1;
        }
```

```
    }
    for (int j = 1; j <= 30; j++)
        for (int i = 0; i < s1.size(); i++)
            f[i][j] = f[i][j-1] + f[(i+f[i][j-1])%s1.size()][j-1];
    long long x = 0, ans = 0;
    for (int k = 30; k >= 0; k--) {
        if (x + f[x % s1.size()][k] <= s1.size() * n1) {
            x += f[x % s1.size()][k];
            ans += 1 << k;
        }
    }
    cout << ans / n2 << endl;
}
```

对这两道题目进行总结，我们可以发现，用倍增优化的动态规划算法求解问题时，一般分为两部分。第一部分是**预处理**，用“阶段”成倍增长的 DP，计算出若干与 2 的整数次幂相关的代表状态。第二部分是**拼凑**，基于“二进制划分”思想，用上一步得到的代表状态组合成最终的答案。



0x58 数据结构优化 DP

状态压缩 DP 和倍增 DP 都是从状态表示入手对 DP 进行的优化。当状态表示和状态转移方程确定后，如何高效地按照公式执行计算也是一个非常值得探讨的问题。从本节开始，我们将从转移入手，探讨 DP 程序实现中的优化手段。

在 0x04 节的例题“Best Cow Fences”、0x51 节的例题“LCIS 最长公共上升子序列”中，我们已经接触到了对于状态转移的优化。这两道题目的动态规划算法在转移时，都需要枚举一个决策，在所有可能情况中取最大或最小值。然而，随着 DP 阶段的增长，围的下界不变，上界每次增大 1。更一般地，只要这个**决策的候选集合只扩大、不缩小**，该决策的取值范我们就可以**仅用一个变量维护最值**，不断与新加入候选集合的元素比较，即可直接得到最优决策，O(1) 地执行转移。

在更复杂的情况下，我们就要用更加高级的数据结构（而不是仅仅一个变量）维护 DP 决策的候选集合，以便快速执行插入元素、删除元素、查询最值等操作，把朴素的在取值范围中枚举决策的时间优化为维护数据结构的时间。这就是本节探讨的主要内容。

例题 **Cleaning Shifts** AcWing295/AcWing296

有一条很长的白色纸带，被划分成一个个长度为 1 的网格，其中第 L 到第 R 个网格不慎被染上了黑色墨水。现在有 N 条贴纸，第 i 条贴纸可以覆盖第 a_i 到第 b_i 个格子，售价为 c_i。求用若干条贴纸覆盖纸带上的第 L 到第 R 个格子，至少要花费多少钱。$1 \le N \le 25000$，$1 \le L \le R \le 10^6$。

设 $f[x]$ 表示覆盖 $[L,x]$ 需要花费的最小代价。

把所有贴纸按照右端点 b_i 递增排序，按顺序扫描这些贴纸。设当前贴纸为 $[a_i,b_i]$，价格为 c_i。状态转移方程为：

$$f[b_i] = \min_{a_i-1 \le x < b_i} \{f[x]\} + c_i$$

初值：$f[L-1]=0$，其余为正无穷。目标：$\min_{b_i \ge R}\{f[b_i]\}$。

在这个状态转移方程中，需要查询 f 数组在 $[a_i-1,b_i]$ 上的最小值，同时 f 数组会不断发生更新。这是一个带有修改的区间最值问题，使用线段树维护 f 数组即可在

$O(\log N)$ 的时间内执行查询、更新操作。对线段树和区间最值问题不熟悉的读者请回顾 0x43 节。

本题中网格位置的坐标都很小，可以直接在 $[L-1,R]$ 上建立线段树。当坐标较大时也可以先离散化，再用线段树求解。另外，请读者在程序实现时，注意处理贴纸左、右端点超出 $[L,R]$ 的边界情况。

例题 The Battle of Chibi AcWing297

给定一个长度为 N 的数列 A，求 A 有多少个长度为 M 的严格递增子序列。$1 \le M \le N \le 1000$，序列 A 中的数的绝对值不超过 10^9。因为答案可能很大，你只需要输出对 10^9+7 取模后的结果。

设 $F[i,j]$ 表示 A 的前 j 个数构成的以 A_j 为结尾的数列中，长度为 i 的严格递增子序列有多少个。

$$F[i,j]=\sum_{k<j \text{ 并且 } A_k<A_j} F[i-1,k]$$

在上式中，i 和 j 都可看作“阶段”，只会从小往大转移。k 是 DP 的“决策”，有两个限制条件 $k<j$ 和 $A_k<A_j$。我们先写出枚举 k 的朴素程序：

```
const int mod = 1000000007;
memset(f, 0, sizeof(f));
a[0] = -(1<<30); // -INF
f[0][0] = 1;
for (int i = 1; i <= m; i++)
    for (int j = 1; j <= n; j++)
        for (int k = 0; k < j; k++)
            if (a[k] < a[j])
                f[i][j] = (f[i][j] + f[i-1][k]) % mod;
int ans = 0;
for (int i = 1; i <= n; i++)
    ans = (ans + f[m][i]) % mod;
```

在内层循环 j 和 k 进行时，可以把外层循环变量 i 看作定值。我们可以发现，当 j 增加 1 时，k 的取值范围从 $0 \le k < j$ 变为 $0 \le k < j+1$，也就是只多了 $k=j$ 这 1 个新决策。所以，我们需要维护一个支持如下操作的决策候选集合，其中每个决策可以用一个二元组 $(A_k, F[i-1,k])$ 来存储。

1. 插入一个新的决策。具体来说，在 j 增加 1 前，把 $(A_j, F[i-1,j])$ 加入集合。
2. 给定一个值 A_j，查询满足 $A_k < A_j$ 的二元组对应的 $F[i-1,k]$ 的和。



若把二元组的 A_k 看作关键码，$F[i-1,k]$ 看作权值，这就是一个典型的支持插入、查询前缀和的数据结构问题，用平衡树可直接解决。

当然，因为平衡树实现起来难度较大，我们可以把数列 A 中的所有数值离散化到 $[2,N+1]$ 之间，设 $\mathrm{val}(x)$ 表示 x 离散化后的值。特殊地，我们令 $A_0=-\infty$，$\mathrm{val}(A_0)=1$。然后在 $[1,N+1]$ 上建立树状数组，起初所有值为 0 。

1. 对于插入决策的操作，就把 $\mathrm{val}(A_k)$ 位置上的值增加 $F[i-1,k]$。

2. 对于查询操作，就在树状数组中计算 $[1,\mathrm{val}(A_j)-1]$ 的前缀和。

综上所述，我们简单地使用树状数组来维护前缀和，就把动态规划的时间复杂度优化到了$O(MN\log N)$。

```
for (int i = 1; i <= m; i++) {
    memset(c, 0, sizeof(c)); // c 为存储树状数组数据的数组
    add(val(a[0]), f[i-1][0]); // 树状数组 add 操作
    for (int j = 1; j <= n; j++) {
        f[i][j] = ask(val(a[j]) - 1); // 树状数组 ask 操作
        add(val(a[j]), f[i-1][j]); // 树状数组 add 操作
    }
}
```

本节提到的三个问题是循序渐进的。这些题目中 DP 决策的取值范围都可以用简单的不等式来表示。在开头提到的题目中，不等式只有上界变化，且只增大、不减小，于是我们采用变量维护法。在第一道例题中，不等式除上界不断增大外，下界变化没有什么规律，此时我们采用更加灵活的支持区间最值维护的数据结构。在第二道例题中，取值范围有两个限制条件：一个是关于“数组下标”的位置，一个是关于“数列 A 的数值”的位置，它们实际上是两种“坐标”。DP 循环顺序保证了第一个条件的满足。对于第二个条件，我们在“数列 A”这个“坐标轴”上建立了以 F 数组中的状态为值的数据结构。

总而言之，无论 DP 决策的限制条件是多还是少，我们都要尽量对其进行分离。多维 DP 状态在执行内层循环时，把外层循环变量看作定值。状态转移取最优决策时，简单的限制条件用循环顺序处理，复杂的限制条件用数据结构维护，注重二者之间的配合。在更难的题目中，还经常出现数据结构之间的嵌套，以同时满足更多的限制条件。

0x59 单调队列优化 DP

在 0x11 节讲解“栈”和 0x12 节讲解“队列”时，我们已经引入了“单调栈”和“单调队列”两种思想。它们的本质都是借助单调性，及时排除不可能的决策，保持候选集合的高度有效性和秩序性。请读者仔细回顾 0x12 节“最大子序和”这道例题，该问题的答案可以形式化表述为：

$$ans = \max_{1 \le i \le N} \left\{ S[i] - \min_{i-M \le j \le i-1} \{S[j]\} \right\}$$

此处的 i 非常类似于动态规划的“状态”，j 则类似于动态规划的“决策”。我们从小到大枚举每个 $i \in [1, N]$，当 i 增大 1 时，j 的取值范围 $[i-M, i-1]$ 的上、下界同时增大 1，变为 $[i-M+1, i]$。这意味着不仅有一个新的决策 $j = i$ 进入了候选集合，也应该把过时的决策 $j = i - M$ 从候选集合中排除。单调队列非常适合优化这种决策取值范围的上、下界均单调变化，每个决策在候选集合中插入或删除至多一次的问题。

类比“最大子序和”一题的思想，请读者尝试解答下面几道例题。

例题 **Fence** AcWing298

有 N 块木板从左到右排成一行，有 M 个工匠对这些木板进行粉刷，每块木板至多被粉刷一次。第 i 个工匠要么不粉刷，要么粉刷包含木板 S_i 的、长度不超过 L_i 的连续的一段木板，每粉刷一块可以得到 P_i 的报酬。求如何安排能使工匠们获得的总报酬最多。$1 \le N \le 16000$, $1 \le M \le 100$。

先把所有工匠按照 S_i 排序，这样一来，每个工匠粉刷的木板一定在上一个工匠粉刷的木板之后，使我们能够按顺序进行线性 DP。

设 $F[i, j]$ 表示安排前 i 个工匠粉刷前 j 块木板（可以有空着不刷的木板），工匠们能获得的最多报酬。

1. 第 i 个工匠可以什么也不刷，此时 $F[i, j] = F[i-1, j]$。

2. 第 j 块木板可以空着不刷，此时 $F[i, j] = F[i, j-1]$。

3. 第 i 个工匠粉刷第 $k+1$ 块到第 j 块木板。根据题意，该工匠粉刷总数不能超过 L_i，且必须粉刷 S_i，所以需要满足：$k+1 \le S_i \le j$ 并且 $j - k \le L_i$。于是，有状



态转移方程：

$$F[i,j]=\max_{j-L_i\le k\le S_i-1}\{F[i-1,k]+P_i*(j-k)\}，其中 j\ge S_i$$

我们重点来看这个方程如何优化。首先，我们多次提到，在考虑内层循环 j 以及决策 k 时，可把外层循环变量 i 看作定值。这样一来，状态转移方程中的各项可分为两部分。

1. P_i*j，除定值 i 外，只有状态变量 j。

2. $F[i-1,k]-P_i*k$，除定值 i 外，只有决策变量 k。

状态转移方程可改写为：

$$F[i,j]=P_i*j+\max_{j-L_i\le k\le S_i-1}\{F[i-1,k]-P_i*k\}，其中 j\ge S_i$$

当 j 增大时，k 的取值范围上界 S_i-1 不变，下界 $j-L_i$ 变大。按与“最大子序和”一题类似的想法，我们比较一下任意两个决策 k_1 和 k_2。不妨设 $k_1<k_2\le S_i-1$。

因为 k_2 比 k_1 更靠后，所以随着 j 的增加，k_1 会比 k_2 更早从范围 $[j-L_i,S_i-1]$ 中被排除。如果还满足 $F[i-1,k_1]-P_i*k_1\le F[i-1,k_2]-P_i*k_2$，那么就意味着 k_2 不但比 k_1 更优，还比 k_1 的存活时间更长。在这种情况下，k_1 就是一个无用的决策，应该被直接排除出候选集合。

综上所述，我们可以维护一个决策点 k 单调递增，数值 $F[i-1,k]-P_i*k$ 单调递减的队列。只有这个队列中的决策才有可能在某一时刻成为最优决策。这个单调队列支持如下几种操作。

1. 当 j 变大时，检查队头元素，把小于 $j-L_i$ 的决策出队。

2. 需要查询最优决策时，队头即为所求。

3. 有一个新的决策要加入候选集合时，在队尾检查 $F[i-1,k]-P_i*k$ 的单调性，把无用决策从队尾直接出队，最后把新决策加入队列。

在本题中具体来说，当内层循环开始时（$j=S_i$），建立一个空的单调队列，把 $[\max(S_i-L_i,0),S_i-1]$ 中的决策依次加入候选集合（执行操作 3）。对于每个 $j=S_i\sim N$ 先在队头检查决策合法性（操作 1），然后取队头为最优决策（操作 2）进行状态转移。因为每个决策至多入队、出队一次，故转移的时间复杂度均摊 $O(1)$。整个算法的时间复杂度为 $O(NM)$。

```
struct rec{ int L, P, S; } a[110];
int n, m, f[110][16010], q[16010];

bool operator <(rec a, rec b) {
    return a.S < b.S;
```

```
}

int calc(int i, int k) {
    return f[i - 1][k] - a[i].P * k;
}

int main()  {
    cin >> n >> m;
    for (int i = 1; i <= m; i++)
        scanf("%d%d%d", &a[i].L, &a[i].P, &a[i].S);
    sort(a + 1, a + m + 1);
    for (int i = 1; i <= m; i++) {
        // 初始化单调队列
        int l = 1, r = 0;
        // 把最初的候选集合插入队列
        for (int k = max(0, a[i].S-a[i].L); k <= a[i].S-1; k++) {
            // 插入新决策，维护队尾单调性
            while (l <= r && calc(i, q[r]) <= calc(i, k)) r--;
            q[++r] = k;
        }
        for (int j = 1; j <= n; j++) {
            // 不粉刷时的转移
            f[i][j] = max(f[i - 1][j], f[i][j - 1]);
            // 粉刷第 k+1~j 块木板时的转移
            if (j >= a[i].S) {
                // 排除队头不合法决策
                while (l <= r && q[l] < j - a[i].L) l++;
                // 队列非空时，取队头进行状态转移
                if (l <= r)
                    f[i][j] = max(f[i][j],
                                  calc(i, q[l]) + a[i].P * j);
            }
        }
    }
    cout << f[m][n] << endl;
}
```

例题 Cut the Sequence AcWing299

给定一个长度为 N 的序列 A，要求把该序列分成若干段，在满足“每段中所有数的和”不超过 M 的前提下，让“每段中所有数的最大值”之和最小。试计算这个最小值。$N \le 10^5$，数列 A 中的数非负，且不超过 10^6，$M \le 10^{11}$。

设 $F[i]$ 表示把前 i 个数分成若干段，在满足每段中所有数的和不超过 M 的前提下，各段的最大值之和最小是多少。容易写出状态转移方程：

$$F[i] = \min_{0 \le j < i \text{并且} \sum_{k=j+1}^{i} A_k \le M} \left\{ F[j] + \max_{j+1 \le k \le i} \{A_k\} \right\}$$

若采用枚举决策 j 的方法，时间复杂度为 $O(N^2)$。然而，该方程似乎很难进行优化，因为 $\max_{j+1 \le k \le i}\{A_k\}$ 不容易用一个简单的多项式来表示，不容易找到单调性。我们曾多次强调，DP 转移优化的指导思想就是及时排除不可能的决策，保持候选集合的高度有效性和秩序性。本着这个原则，我们考虑一个决策 j 何时是必要的。

引理：

在上述方程中，若 $j(0 \le j < i)$ 可能成为最优决策，则除了 $\sum_{k=j+1}^{i} A_k \le M$ 外，必然还满足以下**两个条件之一**。

1. $A_j = \max_{j \le k \le i}\{A_k\}$。

2. $\sum_{k=j}^{i} A_k > M$（即 j 是满足 $\sum_{k=j+1}^{i} A_k \le M$ 的最小的 j）。

证明：

反证法。假设两个条件都不满足，即 $A_j < \max_{j \le k \le i}\{A_k\}$ 并且 $\sum_{k=j}^{i} A_k \le M$。由此可知 $\max_{j \le k \le i}\{A_k\} = \max_{j+1 \le k \le i}\{A_k\}$。另外，$F$ 数组显然具有单调性，即 $F[j-1] \le F[j]$（多一个数，答案肯定更大）。于是：

$$F[j-1] + \max_{j \le k \le i}\{A_k\} \le F[j] + \max_{j+1 \le k \le i}\{A_k\}$$

上式的含义就是决策 $j-1$ 比 j 更优。而决策的取值范围是 $0 \le j < i$，$j-1$ 比 j 更容易满足这个条件。所以，上式与 j 可能成为最优决策矛盾，故假设不成立。

证毕。

引理中的第 2 个条件比较简单，只需预处理出对于每个 i，满足 $\sum_{k=j+1}^{i} A_k \le M$ 的最小的 j，记为 $C[i]$。在计算 $F[i]$ 时，从 $C[i]$ 进行一次转移即可。下面我们单独讨论满足引理中第 1 个条件的决策 j 的维护方法。

根据引理，当一个新的决策 j_2 插入候选集合时，若候选集合中已有的决策 j_1 满足条件 $j_1 < j_2$ 并且 $A_{j_1} \le A_{j_2}$，则 j_1 就是无用决策，可被排除。

综上所述，我们可以维护一个**决策点 j 单调递增、数值 A_j 单调递减**的队列，只有该队列中的元素才可能成为最优决策。

与上一道题目不同的是，该队列只是一个 A_j 单调递减的队列，关于转移方程等式右边的 $F[j]+\max\limits_{j+1\le k\le i}\{A_k\}$ 并没有单调性。所以我们不能简单地取队头为最优决策，而是要再加一个额外的数据结构（例如二叉堆）。二叉堆与单调队列保存相同的候选集合，该插入时一起插入，该删除时一起删除（或用“懒惰删除法”，参见 0x71 节），只不过单调队列以 A_j 递减作为比较大小的依据，二叉堆以 $F[j]+\max\limits_{j+1\le k\le i}\{A_k\}$ 作为比较大小的依据，保证能快速在候选集合中查询最值。事实上，我们一般称这种做法为“二叉堆与单调队列建立了映射关系”。

最后，关于 $\max\limits_{j+1\le k\le i}\{A_k\}$ 的计算有两种方法：第一种是按照区间最值问题，直接用 ST 算法预处理，支持 O(1) 查询；第二种是仔细发掘本题中单调队列的性质。我们可以发现，队列中某一项的 $\max\limits_{j+1\le k\le i}\{A_k\}$ 的结果其实就是队列中下一个元素的 A 值。

在整个算法中，每个 j 至多在单调队列和二叉堆中插入和删除一次，故时间复杂度为 $O(N\log N)$。

单调队列优化多重背包

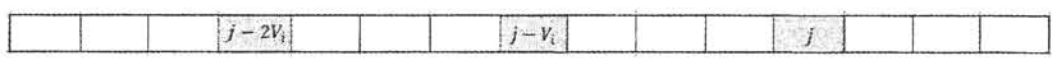
在 0x52 节“背包”中，我们已经介绍了多重背包问题的朴素解法和二进制拆分解法。若使用单调队列，可以把求解多重背包问题的复杂度进一步优化到 $O(NM)$。先来回顾一下多重背包问题的模型：

给定 N 种物品，其中第 i 种物品的体积为 V_i，价值为 W_i，并且有 C_i 个。有一容积为 M 的背包，要求选择若干个物品放入背包，使得物品总体积不超过 M 的前提下，物品的价值总和最大。

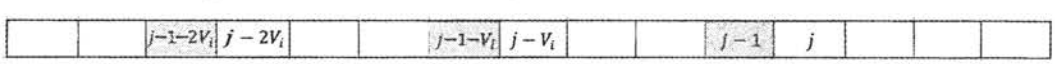
在 0x52 节背包的解法中，DP 数组省略了“阶段”这一维。当外层循环进行到 i 时，$F[j]$ 表示从前 i 种物品中选出若干个放入背包，体积之和为 j 时，价值之和最大是多少。**倒序循环** j，在状态转移时，考虑选取第 i 个物品的个数 cnt：

$$F[j]=\max_{1\le cnt\le C_i}\{F[j-cnt*V_i]+cnt*W_i\}$$

画出能够转移到状态 j 的决策候选集合 $\{j-cnt*V_i\mid 1\le cnt\le C_i\}$：



当循环变量 j 减小 1 时：



可以发现，相邻两个状态 j 和 $j-1$ 对应的决策候选集合没有重叠，很难快速地从 $j-1$ 对应的集合得到 j 对应的集合。

但是，我们试着考虑一下状态 j 和 $j-V_i$：

$C_i=3$

				$j-3V_i$			$j-2V_i$			$j-V_i$			j	
	$j-4V_i$			$j-3V_i$			$j-2V_i$			$j-V_i$				

这两者对应的决策候选集合之间的关系，与我们前面讲解的单调队列题目非常相似，只有一个新决策加入候选集合、一个已有决策被排除。所以，我们应该把**状态 j 按照除以 V_i 的余数分组**，对每一组分别进行计算，不同组之间的状态在阶段 i 不会互相转移。

余数为 0 —— $0, V_i, 2V_i, \cdots$

余数为 1 —— $1, 1+V_i, 1+2V_i, \cdots$

……

余数为 V_i-1 —— $(V_i-1), (V_i-1)+V_i, (V_i-1)+2V_i, \cdots$

把“倒序循环 j”的过程，改为对每个余数 $u \in [0, V_i-1]$，倒序循环 $p=\lfloor (M-u)/V_i \rfloor \sim 0$，对应的状态就是 $j=u+p*V_i$。第 i 种物品只有 C_i 个，故能转移到 $j=u+p*V_i$ 的决策候选集合就是 $\{u+k*V_i \mid p-C_i \le k \le p-1\}$。写出新的状态转移方程[①]：

$$F[u+p*V_i]=\max_{p-C_i \le k \le p-1}\{F[u+k*V_i]+(p-k)*W_i\}$$

与本节的“Fence”一题类似，把外层循环 i 和 u 看作定值，当内层循环变量 p 减小 1 时，决策 k 的取值范围 $[p-C_i, p-1]$ 的上、下界均单调减小。状态转移方程等号右侧的式子仍然分为两部分，仅包含变量 p 的 $p*W_i$ 和仅包含变量 k 的 $F[u+k*V_i]-k*W_i$。综上所述，我们可以建立一个决策点 k 单调递减、数值 $F[u+k*V_i]-k*W_i$ 单调递减的队列，用于维护候选集合。对于每个 p，执行单调队列有以下三个惯例操作。

1. 检查队头合法性，把大于 $p-1$ 的决策点出队。
2. 取队头为最优决策，更新 $F[u+p*V_i]$。
3. 把新决策 $k=p-C_i-1$ 插入队尾，入队前检查队尾单调性，排除无用决策。

整个算法的时间复杂度为 $O(NM)$。

```
int calc(int i, int u, int k) {
    return f[u + k*V[i]] - k*W[i];
}
```

① k 的取值范围：因为省略了“状态”维度，所以我们可以在上一阶段 f 数组的基础上更新最大值，这样就不必考虑“不选第 i 个物品”即 $k=p$ 的情况。

```
int main() {
    cin >> n >> m;
    memset(f, 0xcf, sizeof(f)); // -INF
    f[0] = 0;
    // 物品种类
    for (int i = 1; i <= n; i++) {
        scanf("%d%d%d", &V[i], &W[i], &C[i]);
        // 除以V[i]的余数
        for (int u = 0; u < V[i]; u++) {
            // 建立单调队列
            int l = 1, r = 0;
            // 把最初的候选集合插入队列
            int maxp = (m - u) / V[i];
            for (int k = maxp-1; k >= max(maxp-C[i], 0); k--) {
                while (l<=r && calc(i,u,q[r]) <= calc(i,u,k)) r--;
                q[++r] = k;
            }
            // 倒序循环每个状态
            for (int p = maxp; p >= 0; p--) {
                // 排除过时决策
                while (l <= r && q[l] > p - 1) l++;
                // 取队头进行状态转移
                if (l <= r)
                    f[u + p*V[i]] = max(f[u + p*V[i]],
                                       calc(i,u,q[l]) + p*W[i]);
                // 插入新决策，同时维护队尾单调性
                if (p - C[i] - 1 >= 0) {
                    while (l <= r &&
                        calc(i,u,q[r]) <= calc(i,u,p-C[i]-1)) r--;
                    q[++r] = p - C[i] - 1;
                }
            }
        }
    }
    int ans = 0;
    for (int i = 1; i <= m; i++) ans = max(ans, f[i]);
    cout << ans << endl;
}
```

最后，我们对单调队列优化 DP 的模型进行总结。我们重新写出上面三道例题的状态转移方程：

状态转移方程	定值(外层循环)	状态变量(内层循环)	决策变量
$ans=\max\limits_{1\le i\le N}\left\{S[i]-\min\limits_{i-M\le j\le i-1}\{S[j]\}\right\}$		i	j
$F[i,j]=\max\limits_{j-L_i\le k\le S_i-1}\{F[i-1,k]+P_i*(j-k)\}$	i	j	k
$F[i]=\min\limits_{0\le j<i\text{并且}\sum_{k=j+1}^{i}A_k\le M}\left\{F[j]+\max\limits_{j+1\le k\le i}\{A_k\}\right\}$		i	j
$F[u+p*V_i]=\max\limits_{p-C_i\le k\le p-1}\{F[u+k*V_i]+(p-k)*W_i\}$	i,u	p	k

只关注“状态变量”“决策变量”及其所在的维度，这些状态转移方程都可以大致归为如下形式（除第三个方程稍有不同）：

$$F[i]=\min_{L(i)\le j\le R(i)}\{F[j]+val(i,j)\}$$

上式所代表的问题覆盖范围广泛，是 DP 中一类非常基本、非常重要的模型。这种模型也被称为 1D/1D **的动态规划**。它是一个最优化问题，$L(i)$ 和 $R(i)$ 是关于变量 i 的一次函数，限制了决策 j 的取值范围，并保证其上下界变化具有单调性。$val(i,j)$ 是一个关于变量 i 和 j 的多项式函数，通常是决定我们采用何种优化策略的关键之处。

回想本节三道例题的解法，我们都**把 $val(i,j)$ 分成了两部分，第一部分仅与 i 有关，第二部分仅与 j 有关**。对于每个 i，无论采取哪个 j 作为最优决策，第一部分的值都是相等的，可以在选出最优决策更新 $F[i]$ 时再进行计算、累加。而当 i 发生变化时，第二部分的值不会发生变化，从而保证原来较优的决策，在 i 改变后仍然较优，不会产生乱序的现象。于是，我们就可以在队列中维护第二部分的单调性，及时排除不可能的决策，让 DP 算法得以高效进行。所以，**在上述模型中，多项式 $val(i,j)$ 的每一项仅与 i 和 j 中的一个有关，是使用单调队列进行优化的基本条件。**

0x5A 斜率优化

在上一节末尾我们给出了动态规划的一个经典模型 $F[i]=\min_{L(i)\le j\le R(i)}\{F[j]+val(i,j)\}$，并总结了单调队列优化的基本条件——多项式 $val(i,j)$ 的每一项仅与 i 和 j 中的一个有关。在本节中，我们将讨论**多项式 $val(i,j)$ 包含 i,j 的乘积项**，即存在一个同时与 i 和 j 有关的部分时，该如何进行优化。

例题 任务安排 1 AcWing300

有 N 个任务排成一个序列在一台机器上等待执行，它们的顺序不得改变。机器会把这 N 个任务分成若干批，每一批包含连续的若干个任务。从时刻 0 开始，任务被分批加工，执行第 i 个任务所需的时间是 T_i。另外，在每批任务开始前，机器需要 S 的启动时间，故执行一批任务所需的时间是启动时间 S 加上每个任务所需时间之和。

一个任务执行后，将在机器中稍作等待，直至该批任务全部执行完毕。也就是说，同一批任务将在同一时刻完成。每个任务的费用是它的完成时刻乘以一个费用系数 C_i。请为机器规划一个分组方案，使得总费用最小。

数据范围：$1\le N\le 5000, 1\le S\le 50, 1\le T_i, C_i\le 100$。

解法一

求出 T,C 的前缀和 $sumT, sumC$，即 $sumT[i]=\sum_{j=1}^{i}T[j]$，$sumC[i]=\sum_{j=1}^{i}C[j]$。设 $F[i,j]$ 表示把前 i 个任务分成 j 批执行的最小费用，则第 j 批任务完成的时间就是 $j*S+sumT[i]$。以第 $j-1$ 批和第 j 批任务的分界点为 DP 的“决策”，状态转移方程为：

$$F[i,j]=\min_{0\le k<i}\{F[k,j-1]+(S*j+sumT[i])*(sumC[i]-sumC[k])\}$$

该解法的时间复杂度为 $O(N^3)$。

解法二

本题并没有规定需要把任务分成多少批，在上一个解法中之所以需要批数 j，是因为我们需要知道机器启动了多少次（每次启动都要 S 单位时间），从而计算出 i 所在的一批任务的完成时刻。

事实上，在执行一批任务时，我们不容易直接得知在此之前机器启动过几次。但我



们知道，机器因执行这批任务而花费的启动时间 S，会累加到在此之后所有任务的完成时刻上。设 $F[i]$ 表示把前 i 个任务分成若干批执行的最小费用，状态转移方程为：

$$F[i]=\min_{0\le j<i}\{F[j]+sumT[i]*(sumC[i]-sumC[j])+S*(sumC[N]-sumC[j])\}$$

在上式中，第 $j+1\sim i$ 个任务在同一批内完成，$sumT[i]$ 是忽略机器的启动时间时，这批任务的完成时刻。因为这批任务的执行，机器的启动时间 S 会对第 $j+1$ 个之后的所有任务产生影响，故我们把这部分补充到费用中。

也就是说，我们没有直接求出每批任务的完成时刻，而是在一批任务“开始”对后续任务产生影响时，就先把费用累加到答案中。这是一种名为“**费用提前计算**”的经典思想。

该解法的时间复杂度为 $O(N^2)$。

```
long long f[5010], sumt[5010], sumc[5010];
int n, s;
int main() {
    cin >> n >> s;
    for (int i = 1; i <= n; i++) {
        int t, c;
        scanf("%d%d", &t, &c);
        sumt[i] = sumt[i - 1] + t;
        sumc[i] = sumc[i - 1] + c;
    }
    memset(f, 0x3f, sizeof(f));
    f[0] = 0;
    for (int i = 1; i <= n; i++)
        for (int j = 0; j < i; j++)
            f[i] = min(f[i], f[j] + sumt[i] * (sumc[i] - sumc[j])
                             + s*(sumc[n] - sumc[j]));
    cout << f[n] << endl;
}
```

例题 任务安排 2 AcWing301

题意同“任务安排 1”。

数据范围：$1\le N\le 3*10^5, 1\le S,T_i,C_i\le 512$。

对上一题的解法二进行优化，先对状态转移方程稍作变形，把常数、仅与 i 有关的项、仅与 j 有关的项以及 i,j 的乘积项分开。

$$F[i]=\min_{0\le j<i}\{F[j]-(S+sumT[i])*sumC[j]\}$$

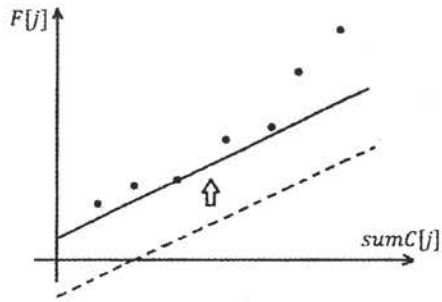
$$+sumT[i]*sumC[i]+S*sumC[N]$$

把 min 函数去掉，把关于 j 的值 $F[j]$ 和 $sumC[j]$ 看作变量，其余部分看作常数，得到：

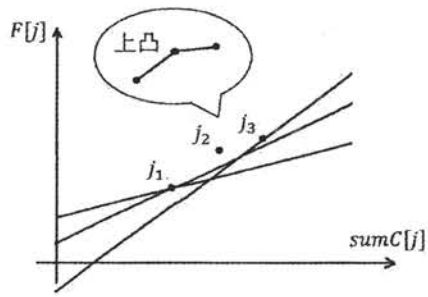
$$F[j]=(S+sumT[i])*sumC[j]+F[i]-sumT[i]*sumC[i]-S*sumC[N]$$

在 $sumC[j]$ 为横坐标，$F[j]$ 为纵坐标的平面直角坐标系中，这是一条以 $S+sumT[i]$ 为斜率，$F[i]-sumT[i]*sumC[i]-S*sumC[N]$ 为截距的直线。也就是说，决策候选集合是坐标系中的一个点集，每个决策 j 都对应着坐标系中的一个点 $(sumC[j],F[j])$。每个待求解的状态 $F[i]$ 都对应着一条直线的截距，直线的斜率是一个固定的值 $S+sumT[i]$，截距未知。当截距最小化时，$F[i]$ 也取到最小值。

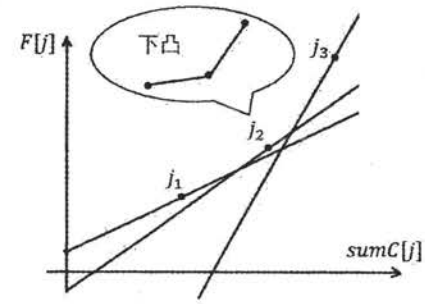
该问题实际上是一个**线性规划问题**，高中数学课本中应该有所涉及。令直线过每个决策点 $(sumC[j],F[j])$，都可以解出一个截距，其中使截距最小的一个就是最优决策。体现在坐标系中，就是用一条斜率为固定正整数的直线自下而上平移，第一次接触到某个决策点时，就得到了最小截距。如下图所示。



对于任意三个决策点 $(sumC[j_1],F[j_1])$，$(sumC[j_2],F[j_2])$ 和 $(sumC[j_3],F[j_3])$，不妨设 $j_1<j_2<j_3$，因为 T,C 均为正整数，亦有 $sumC[j_1]<sumC[j_2]<sumC[j_3]$。根据“及时排除无用决策”的思想，我们来考虑 j_2 有可能成为最优决策的条件。



连接j_1,j_2的线段与连接j_2,j_3的线段构成上凸形状
无论直线斜率是多少，j_2都不是最优决策

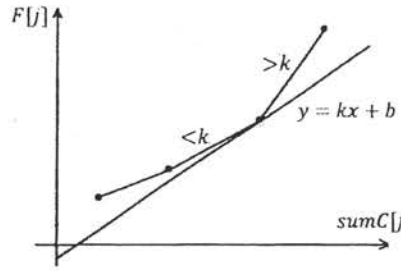


连接j_1,j_2的线段与连接j_2,j_3的线段构成下凸形状
j_2有可能成为最优决策

从上图中我们发现，j_2 有可能成为最优决策，当且仅当：

$$\frac{F[j_2]-F[j_1]}{sumC[j_2]-sumC[j_1]}<\frac{F[j_3]-F[j_2]}{sumC[j_3]-sumC[j_2]}$$

不等号两侧实际上都是连接两个决策点的线段的斜率。通俗地讲，我们应该维护“连接相邻两点的线段斜率”单调递增的一个“下凸壳”[①]，只有这个“下凸壳”的顶点才有可能成为最优决策。实际上，对于一条斜率为 k 的直线，若某个顶点左侧线段的斜率比 k 小、右侧线段的斜率比 k 大，则该顶点就是最优决策。换言之，如果把这条直线和所有线段组成一个序列，那么令直线截距最小化的顶点就出现在按照斜率大小排序时，直线应该排在的位置上。如下图所示。



在本题中，j 的取值范围是 $0\le j<i$，随着 i 的增大，每次会有一个新的决策进入候选集合。因为 $sumC$ 的单调性，新决策在坐标系中的横坐标一定大于之前的所有决策，出现在整个凸壳的最右端。另外，因为 $sumT$ 的单调性，每次求解“最小截距”的直线斜率 $S+sumT[i]$ 也单调递增，如果我们只保留凸壳上“连接相邻两点的线段斜率”大于 $S+sumT[i]$ 的部分，那么凸壳的最左端顶点就一定是最优决策。

综上所述，我们可以建立单调队列 q，维护这个下凸壳。队列中保存若干个决策变量，它们对应凸壳上的顶点，且满足横坐标 $sumC$ 递增、连接相邻两点的线段斜率也递增。需要支持的操作与一般的单调队列题目类似，对于每个状态变量 i：

1. 检查队头的两个决策变量 $q[l]$ 和 $q[l+1]$，若斜率 $\big(F[q[l+1]]-F[q[l]]\big)/\big(sumC[q[l+1]]-sumC[q[l]]\big)\le S+sumT[i]$，则把 $q[l]$ 出队，继续检查新的队头；

2. 直接取队头 $j=q[l]$ 为最优决策，执行状态转移，计算出 $F[i]$；

3. 把新决策 i 从队尾插入，在插入之前，若三个决策点 $j_1=q[r-1],j_2=q[r],j_3=i$ 不满足斜率单调递增（不满足下凸性，即 j_2 是无用决策），则直接从队尾把 $q[r]$ 出队，继续检查新的队尾。

整个算法的时间复杂度为 $O(N)$。

---

① 凸壳，意思是点集的边界。在斜率优化中，我们一般称“连接相邻两点的线段斜率”单调递增的一组顶点构成“下凸壳”，称“连接相邻两点的线段斜率”单调递减的一组顶点构成“上凸壳”。事实上，在高等数学中，因为斜率对应一阶导数，斜率的单调性对应二阶导数，所以我们称在定义域上二阶导数大于 0 的函数为凸函数，即“下凸壳”，二阶导数小于 0 的函数为凹函数，即“上凸壳”。关于函数凹凸性更加严谨的定义，请参阅配套资源中的参考资料。

```
int main() {
    cin >> n >> s;
    for (int i = 1; i <= n; i++) {
        int t, c;
        scanf("%d%d", &t, &c);
        sumt[i] = sumt[i - 1] + t;
        sumc[i] = sumc[i - 1] + c;
    }
    memset(f, 0x3f, sizeof(f));
    f[0] = 0;
    int l = 1, r = 1;
    q[1] = 0;
    for (int i = 1; i <= n; i++) {
        while (l < r && (f[q[l+1]] - f[q[l]])
            <= (s + sumt[i]) * (sumc[q[l+1]] - sumc[q[l]])) l++;
        f[i] = f[q[l]] - (s + sumt[i]) * sumc[q[l]]
            + sumt[i] * sumc[i] + s * sumc[n];
        while (l < r && (f[q[r]]-f[q[r-1]]) * (sumc[i]-sumc[q[r]])
            >= (f[i]-f[q[r]]) * (sumc[q[r]]-sumc[q[r-1]])) r--;
        q[++r] = i;
    }
    cout << f[n] << endl;
}
```

与一般的单调队列优化 DP 的模型相比，本题维护的“单调性”依赖于队列中相邻两个元素之间的某种“比值”。因为这个值对应线性规划的坐标系中的斜率，所以我们把本题中使用的优化方法称为**“斜率优化”**，英文称为 convex hull trick（直译为凸壳优化策略）。

* 例题 任务安排 3 AcWing302

题意同“任务安排 1”。

数据范围：$1 \le N \le 3 * 10^5$，$0 \le S, C_i \le 512$，$-512 \le T_i \le 512$。

与上一题不同的是，任务的执行时间 T 可能是负数。这意味着 $sumT$ 不具有单调性，从而需要最小化截距的直线的斜率 $S + sumT[i]$ 不具有单调性。所以，我们不能在单调队列中只保留凸壳上“连接相邻两点的线段斜率”大于 $S + sumT[i]$ 的部分，而是必须维护整个凸壳。这样一来，我们就不需要在队头把斜率与 $S + sumT[i]$ 比较。

队头也不一定是最优决策，我们可以在单调队列中二分查找，求出一个位置 p，p



左侧线段的斜率比 $S + sumT[i]$ 小，右侧线段的斜率比 $S + sumT[i]$ 大，p 就是最优决策。

队尾维护斜率单调性的方法与上一题相同。

```
int binary_search(int i, int k) {
    if (l == r) return q[l];
    int L = l, R = r;
    while (L < R) {
        int mid = (L + R) >> 1;
        if (f[q[mid + 1]] - f[q[mid]] <= k *
            (sumc[q[mid + 1]] - sumc[q[mid]])) L = mid + 1;
        else R = mid;
    }
    return q[L];
}

int main() {
    cin >> n >> s;
    for (int i = 1; i <= n; i++) {
        int t, c; scanf("%d%d", &t, &c);
        sumt[i] = sumt[i - 1] + t, sumc[i] = sumc[i - 1] + c;
    }
    l = r = 1, q[1] = 0;
    for (int i = 1; i <= n; i++) {
        int p = binary_search(i, s + sumt[i]);
        f[i] = f[p] - (s + sumt[i]) * sumc[p]
            + sumt[i] * sumc[i] + s * sumc[n];
        while (l<r && (f[q[r]]-f[q[r-1]]) * (sumc[i]-sumc[q[r]])
            >= (f[i]-f[q[r]]) * (sumc[q[r]]-sumc[q[r-1]])) r--;
        q[++r] = i;
    }
    cout << f[n] << endl;
}
```

*【思考题】任务安排 4

如果 T_i 是正整数，但 C_i 可能是负数，该怎么办？

如果 T_i 和 C_i 都有可能是负数呢？

提示：

若 C_i 可能是负数，可以倒序 DP，设计一个状态转移方程，让 $sumT$ 是横坐标，$sumC$ 是斜率中的一项。仍然可以用单调队列维护凸壳，用二分法求出最优决策。

若二者都可能是负数，则意味着要在凸壳任意位置动态插入顶点、动态查询。此时要用平衡树来维护凸壳。

例题 **Cats Transport** AcWing303

小 S 是农场主，他养了 M 只猫，雇了 P 位饲养员。农场中有一条笔直的路，路边有 N 座山，从 1 到 N 编号。第 i 座山与第 $i-1$ 座山之间的距离是 D_i。饲养员都住在 1 号山。

有一天，猫出去玩。第 i 只猫去 H_i 号山玩，玩到时刻 T_i 停止，然后在原地等饲养员来接。饲养员们必须回收所有的猫。每个饲养员沿着路从 1 号山走到 N 号山，把各座山上已经在等待的猫全部接走。饲养员在路上行走需要时间，速度为 1 米/单位时间。饲养员在每座山上接猫的时间可以忽略，可以携带的猫的数量为无穷大。

例如有两座相距为 1 的山，一只猫在 2 号山玩，玩到时刻 3 开始等待。如果饲养员从 1 号山在时刻 2 或 3 出发，那么他可以接到猫，猫的等待时间为 0 或 1。而如果他于时刻 1 出发，那么他将于时刻 2 经过 2 号山，不能接到当时仍在玩的猫。

你的任务是规划每个饲养员从 1 号山出发的时间，使得所有猫等待时间的总和尽量小。饲养员出发的时间可以为负。

数据范围：$2 \le N \le 10^5, 1 \le M \le 10^5, 1 \le P \le 100$。

对于每只猫，设 $A_i = T_i - \sum_{j=1}^{H_i} D_j$。一名饲养员如果想接到第 i 只猫，就必须在 A_i 时刻之后从 1 号山出发。若出发时刻为 t，则这只猫的等待时间就是 $t - A_i$。

把 A_i 从小到大排序，求出排好序的 A 数组的前缀和，记录在数组 S 中。根据贪心策略，每个饲养员带走的猫一定是按照 A 排序后连续的若干只，请读者尝试给出证明，这里就不再赘述。

此时本题就与“任务安排”非常类似。饲养员就是“机器”，猫就是“任务”，每个饲养员带走连续的一段猫。设 $F[i,j]$ 表示前 i 个饲养员带走前 j 只猫，猫等待时间的总和最小是多少。

假设第 i 个饲养员带走第 $k+1 \sim j$ 只猫，那么该饲养员的最早出发时间就是 A_j。这些猫的等待时间之和就是 $\sum_{p=k+1}^{j}(A_j - A_p) = A_j * (j-k) - (S_j - S_k)$。写出状态转移方程：

$$F[i,j] = \min_{0 \le k < j}\{F[i-1,k] + A_j * (j-k) - (S_j - S_k))\}$$



直接枚举 k，求出 $F[i,j]$ 的最小值，时间复杂度为 $O(PM^2)$。

把外层循环 i 看作定值，j 是状态变量，k 是决策变量。方程中存在乘积项 $A_j * k$，应考虑使用斜率优化。去掉 min 函数，对方程进行移项，等号左边放仅与 k 有关的项，等号右边放 j,k 乘积项以及仅与 j 有关的项：

$$F[i-1,k]+S_k = A_j * k + F[i,j] - A_j * j + S_j$$

以 k 为横坐标，$F[i-1,k]+S_k$ 为纵坐标建立平面直角坐标系。上式是一条以 A_j 为斜率、$F[i,j]-A_j*j+S_j$ 为截距的直线，当截距最小化时，$F[i,j]$ 取到最小值。

在最小化截距的线性规划问题中，应维护一个下凸壳。建立一个单调队列，队列中相邻两个决策 k_1 和 k_2 应满足 $k_1 < k_2$ 并且“斜率”$((F[i-1,k_2]+S_{k_2})-(F[i-1,k_1]+S_{k_1}))/(k_2-k_1)$ 单调递增。因为直线斜率 A_j 也已经从小到大排过序，所以在程序实现中，采用与“任务安排 2”类似的单调队列的三个基本操作即可。

0x5B 四边形不等式

设 $w(x,y)$ 是定义在整数集合上的二元函数。若对于定义域上的任意整数 a,b,c,d，其中 $a \le b \le c \le d$，都有 $w(a,d)+w(b,c) \ge w(a,c)+w(b,d)$ 成立，则称函数 w 满足**四边形不等式**。

定理（四边形不等式的另一种定义）

设 $w(x,y)$ 是定义在整数集合上的二元函数。若对于定义域上的任意整数 a,b，其中$a<b$，都有 $w(a,b+1)+w(a+1,b) \ge w(a,b)+w(a+1,b+1)$ 成立，则函数 w 满足**四边形不等式**。

证明：

对于 $a<c$，有 $w(a,c+1)+w(a+1,c) \ge w(a,c)+w(a+1,c+1)$。

对于 $a+1<c$，有 $w(a+1,c+1)+w(a+2,c) \ge w(a+1,c)+w(a+2,c+1)$

两式相加，得到 $w(a,c+1)+w(a+2,c) \ge w(a,c)+w(a+2,c+1)$。

依此类推，对任意的 $a \le b \le c$，有 $w(a,c+1)+w(b,c) \ge w(a,c)+w(b,c+1)$。

同理，对任意的 $a \le b \le c \le d$，有 $w(a,d)+w(b,c) \ge w(a,c)+w(b,d)$。

一维线性 DP 的四边形不等式优化

对于形如 $F[i]=\min_{0\le j<i}\{F[j]+val(j,i)\}$ 的状态转移方程，记 $p[i]$ 为令 $F[i]$ 取到最小值的 j 的值，即 $p[i]$ 是 $F[i]$ 的最优决策。若 p 在 $[1,N]$ 上单调不减（非严格单调递增），则称 F 具有决策单调性。

定理（决策单调性）

在状态转移方程 $F[i]=\min_{0\le j<i}\{F[j]+val(j,i)\}$ 中，若函数 val 满足四边形不等式，则 F 具有决策单调性。

证明：

$\forall i \in [1,N]$，$\forall j \in [0,p[i]-1]$，根据 $p[i]$ 的最优性，有：

$$F[p[i]]+val(p[i],i) \le F[j]+val(j,i)$$

$\forall i' \in [i+1, N]$，因为 val 满足四边形不等式，有：

$$val(j, i') + val(p[i], i) \geq val(j, i) + val(p[i], i')$$

移项得：

$$val(p[i], i') - val(p[i], i) \leq val(j, i') - val(j, i)$$

与第一个不等式相加，有：

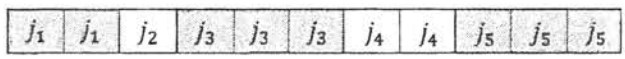
$$F[p[i]] + val(p[i], i') \leq F[j] + val(j, i')$$

这个不等式的含义为，以 $p[i]$ 作为 $F[i']$ 的决策，比以 $j < p[i]$ 作为 $F[i']$ 的决策更优。换言之，$F[i']$ 的最优决策不可能小于 $p[i]$，即 $p[i'] \geq p[i]$。所以 F 有决策单调性。

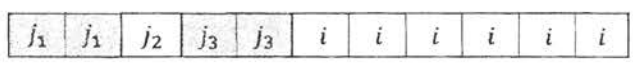
证毕。

当 F 有决策单调性时，我们可以把 $F[i] = \min_{0 \leq j < i}\{F[j] + val(j, i)\}$ 的计算时间从 $O(N^2)$ 优化到 $O(N \log N)$。

考虑对 p 数组进行维护。最初 p 数组全部为 0。在 i 循环进行的任意时刻，根据 $p[i]$ 的单调性，p 的情况如下图所示。



当求出一个新的 $F[i]$ 时，我们应该考虑 i 可以作为哪些 $F[i']$ $(i' > i)$ 的最优决策。根据决策单调性，最终我们会找到一个位置，在该位置之前，p 数组目前存储的决策都比 i 好，在该位置之后，p 数组目前存储的决策都比 i 差。我们要做的就是快速找到上述位置，在 p 数组中把该位置之后的部分都变为 i。



直接修改一个数组效率当然很低。事实上，我们可以建立一个队列，代替 p 数组。队列中保存若干个三元组 (j, l, r)，j 表示决策，l, r 表示目前 $p[l \sim r]$ 的值都是 j。

例如，第一幅图就可以用 5 个三元组 $(j_1, 1, 2), (j_2, 3, 3), (j_3, 4, 6), (j_4, 7, 8), (j_5, 9, 11)$ 来表示。我们从队尾开始检查，判断出整个 $(j_4, 7, 8), (j_5, 9, 11)$ 都不如 i 优，直接从队尾删除。而 $(j_3, 4, 6)$ 左端比 i 优，右端比 i 差。因此，我们在 $(j_3, 4, 6)$ 里二分查找，即可确定所求的位置。最后我们把 $(j_3, 4, 6)$ 变为 $(j_3, 4, 5)$，把 $(i, 6, 11)$ 入队，即可得到第二幅图所示的情景。

另外，队列中没有必要保存小于 $p[1 \sim i-1]$ 的部分，我们可以通过检查队头来排除掉过时的决策。这样就可以像许多单调队列问题一样，直接取队头为最优决策了。

总而言之，对于每个 $i \in [1, N]$，我们执行以下操作。

1. 检查队头：设队头为 (j_0, l_0, r_0)，若 $r_0 = i - 1$，删除队头。否则令 $l_0 = i$。

2. 取队头保存的 j 为最优决策，执行状态转移，计算出 $F[i]$。

3. 尝试插入新决策 i，步骤如下：

(1) 取出队尾，记为 (j_t, l_t, r_t)；

(2) 若对于 $F[l_t]$ 来说，i 是比 j_t 更优的决策，即 $F[i]+val(i,l_t)\le F[j_t]+val(j_t,l_t)$，记 $pos=l_t$，删除队尾，回到步骤(1)；

(3) 若对于 $F[r_t]$ 来说，j_t 是比 i 更优的决策，即 $F[j_t]+val(j_t,r_t)\le F[i]+val(i,r_t)$，去往步骤(5)；

(4) 否则，在 $[l_t,r_t]$ 上二分查找，求出位置 pos，在此之前决策 j_t 更优，在此之后决策 i 更优，去往步骤(5)；

(5) 把三元组 (i,pos,N) 插入队尾。

请读者结合以下例题尝试实现上述过程。

*** 例题 诗人小 G** AcWing304

小 G 是一个出色的诗人，经常作诗自娱自乐。但是，他一直被一件事情所困扰，那就是诗的排版问题。

一首诗包含了若干个句子，对于一些连续的短句，可以将它们用空格隔开并放在一行中，注意一行中可以放的句子数目是没有限制的。小 G 给每首诗定义了一个行标准长度（行的长度为一行中符号的总个数），他希望排版后每行的长度都和行标准长度相差不远。显然排版时，不应改变原有的句子顺序，并且小 G 不允许把一个句子分在两行或者更多的行内。在满足上面两个条件的情况下，小 G 对于排版中的每行定义了一个不协调度，为这行的实际长度与行标准长度差值绝对值的 P 次方，而一个排版的不协调度为所有行不协调度的总和。

小 G 最近又作了几首诗，现在请你对这几首诗进行排版，使得排版后的诗尽量协调（即不协调度尽量小），并把排版的结果告诉他。

诗中的句子数 N 不超过 10^5，每句的字符数不超过 30，行标准长度 L 不超过 $3*10^6$，P 不超过 10。

设 $F[i]$ 表示对前 i 句诗排版的最小不协调度，记 $a[i]$ 为第 i 句诗的长度，$sum[i]$ 为前 i 句诗的总长度。

$$F[i]=\min_{0\le j<i}\{F[j]+|(sum[i]-sum[j])+(i-j-1)-L|^P\}$$

这里 $val(j,i)=|(sum[i]-sum[j])+(i-j-1)-L|^P$，存在大量 i,j 的高次乘积项，不适合用单调队列或斜率优化。于是，我们尝试判断 $val(j,i)$ 是否满足四边形不等式。

要证明对于任意 $j<i$，$val(j,i+1)+val(j+1,i)\ge val(j,i)+val(j+1,i+1)$，

只需证明 $val(j+1,i)-val(j+1,i+1)\geq val(j,i)-val(j,i+1)$。

记 $u=(sum[i]+i)-(sum[j]+j)-(L+1)$。

记 $v=(sum[i]+i)-(sum[j+1]+j+1)-(L+1)$。

只需证明 $|v|^P-|v+(a[i+1]+1)|^P\geq|u|^P-|u+(a[i+1]+1)|^P$。

显然 $u>v$。故只需证明对任意 $c>0$，函数 $y=|x|^P-|x+c|^P$ 单调递减。

当 P 为奇数且 $x\in[-c,0]$ 时，函数写作 $y=-x^P-(x+c)^P$。对 x 求导，得到 $y'=-P*x^{P-1}-P*(x+c)^{P-1}$，因为 $P-1$ 是偶数，P 是正整数，所以 $y'\leq 0$。从而 y 关于 x 单调递减。对于 P 为偶数、$x\in[-\infty,-c]$ 或 $x\in[0,+\infty]$ 的情况，利用导数，同理可证。

综上所述，$val(j,i)$ 满足四边形不等式。因此，F 满足决策单调性。按照我们之前讲解的方法，用队列维护决策三元组，即可在 $O(N\log N)$ 的时间内解决本题。

二维区间 DP 的四边形不等式优化

在区间 DP 问题，例如“石子合并”中，我们经常遇到下面这样的状态转移方程：

$$F[i,j]=\min_{i\leq k<j}\{F[i,k]+F[k+1,j]+w(i,j)\}$$

利用四边形不等式，也可以对该方程的状态转移进行优化。

定理

在状态转移方程 $F[i,j]=\min_{i\leq k<j}\{F[i,k]+F[k+1,j]+w(i,j)\}$ 中（特别地，$F[i,i]=w(i,i)=0$），如果下面两个条件成立：

1. w 满足四边形不等式；
2. 对于任意的 $a\leq b\leq c\leq d$，有 $w(a,d)\geq w(b,c)$。

那么 F 也满足四边形不等式。

证明：

当 $i+1=j$ 时，$F[i,j+1]+F[i+1,j]=F[i,i+2]+F[i+1,i+1]=F[i,i+2]$

若 $F[i,i+2]$ 的最优决策是 $i+1$，则 $F[i,i+2]=F[i,i+1]+F[i+2,i+2]+w(i,i+2)=w(i,i+1)+w(i,i+2)\geq w(i,i+1)+w(i+1,i+2)=F[i,i+1]+F[i+1,i+2]=F[i,j]+F[i+1,j+1]$。

若 $F[i,i+2]$ 的最优决策是 i，则 $F[i,i+2]=F[i,i]+F[i+1,i+2]+w(i,i+2)=w(i+1,i+2)+w(i,i+2)\geq w(i+1,i+2)+w(i,i+1)=F[i+1,i+2]+F[i,i+1]=F[i+1,j+1]+F[i,j]$。

故当 $j-i=1$ 时，四边形不等式对 $F[i,j]$ 成立。

接下来，用数学归纳法，假设当 $j-i<k$ 时，四边形不等式对 $F(i,j)$ 成立。考虑 $j-i=k$ 的情况，设 $F[i,j+1]$ 以 x 为最优决策，$F[i+1,j]$ 以 y 为最优决策。不妨设 $i+1\le x\le y$。

根据 x 和 y 的最优性，有：

$$F[i,j+1]+F[i+1,j]=F[i,x]+F[x+1,j+1]+w(i,j+1)$$
$$+F[i+1,y]+F[y+1,j]+w(i+1,j) \quad ①$$

对于 $F[i,j]$ 和 $F[i+1,j+1]$，x 和 y 是任意决策（不一定最优），故：

$$F[i,j]+F[i+1,j+1]\le F[i,x]+F[x+1,j]+w(i,j)$$
$$+F[i+1,y]+F[y+1,j+1]+w(i+1,j+1) \quad ②$$

因为 w 满足四边形不等式，所以：

$$w(i,j+1)+w(i+1,j)\ge w(i,j)+w(i+1,j+1)$$

根据归纳假设，有：

$$F[x+1,j+1]+F[y+1,j]\ge F[x+1,j]+F[y+1,j+1]$$

结合这两个不等式，比较①②两式，得到：

$$F[i,j+1]+F[i+1,j]\ge F[i,j]+F[i+1,j+1]$$

证毕。

定理（二维决策单调性）

在状态转移方程 $F[i,j]=\min_{i\le k<j}\{F[i,k]+F[k+1,j]+w(i,j)\}$ 中（特别地，$F[i,i]=w(i,i)=0$），记 $P[i,j]$ 为令 $F[i,j]$ 取到最小值的 k 值。

如果 F 满足四边形不等式，那么对于任意 $i<j$，有 $P[i,j-1]\le P[i,j]\le P[i+1,j]$。

证明：

为方便起见，记 $p=P[i,j]$，对于任意的 $i<k\le p$，因为 F 满足四边形不等式：

$$F[i,p]+F[i+1,k]\ge F[i,k]+F[i+1,p]$$

移项可得：

$$F[i+1,k]-F[i+1,p]\ge F[i,k]-F[i,p]$$

根据 p 的最优性，有：

$$F[i,k]+F[k+1,j]\ge F[i,p]+F[p+1,j]$$

因此：

$$\begin{aligned}&\big(F[i+1,k]+F[k+1,j]+w(i+1,j)\big)-\big(F[i+1,p]+F[p+1,j]+w(i+1,j)\big)\\&=(F[i+1,k]-F[i+1,p])+(F[k+1,j]-F[p+1,j])\\&\ge(F[i,k]-F[i,p])+(F[k+1,j]-F[p+1,j])\\&=(F[i,k]+F[k+1,j])-(F[i,p]+F[p+1,j])\end{aligned}$$

≥ 0

这意味着对于 $F[i+1,j]$，p 比任意的 $k \leq p$ 更优。因此 $P[i+1,j] \geq P[i,j]$。

类似地，同理可证 $P[i,j-1] \leq P[i,j]$。

证毕。

例题 再探石子合并 AcWing305

题意与 0x53 节的“石子合并”相同，数据范围 $N \leq 5000$。

石子合并问题的 DP 状态转移方程为：

$$F[l,r] = \min_{l \leq k < r} \{F[l,k] + F[k+1,r]\} + \sum_{i=l}^{r} A_i$$

$\sum_{i=l}^{r} A_i$ 是区间和，显然满足四边形不等式，且取到等号。

根据上面两条定理，我们只需要在 $P[l,r-1] \leq k \leq P[l+1,r]$ 的范围内对 k 进行枚举，求出 $F[l,r]$ 和 $P[l,r]$。算法的时间复杂度为：$O\left(\sum_{l=1}^{N-1} \sum_{r=l+1}^{N} (P[l+1,r] - P[l,r-1] + 1)\right) = O\left(\sum_{l=1}^{N-1} (P[l+1,N] - P[1,N-l] + N - l)\right) = O(N^2)$。

事实上，石子合并问题还有 $O(N \log N)$ 复杂度的非动态规划算法，感兴趣的读者可以自行查阅相关资料，并完成题目 POJ1738/AcWing305。

到此为止，我们对动态规划优化策略的探讨就告一段落。请读者再次回顾 0x56~0x5B 节的各个模型和例题，理解每一种优化的本质，尝试做出一份总结，做到对何种情况下适合使用何种优化方法有个初步的认识。

0x5C 计数类 DP

在本章的最后两节，我们通过几道例题来介绍动态规划中的计数问题和数位统计问题。这两类问题都特别强调“不重不漏”，统计对象的可能情况一般比较多，通常需要精确的划分和整体性的计算。因此，使用动态规划抽象出问题中的“子结构”和推导的“阶段”，将有助于我们准确而高效地进行求解。

例题 **Gerald and Giant Chess** AcWing306

给定一个 $H*W$ 的棋盘，棋盘上只有 N 个格子是黑色的，其他格子都是白色的。

在棋盘左上角有一个卒，每一步可以向右或向下移动一格，并且不能移动到黑色格子中。求这个卒从左上角移动到右下角，一共有多少种可能的路线。

$1 \le H,W \le 10^5, 1 \le N \le 2000$。输出对 10^9+7 取模后的结果即可。

棋盘上的白色格子数量巨大，而黑色格子最多只有 2000 个。我们应该考虑如何依靠黑色格子进行计数。如果这个卒能走进黑色格子，那么从左上角走到右下角的路线总数就是 C_{H+W-2}^{H-1}，我们在 0x37 节“组合计数”中提到过这个公式。若我们再求出从左上角到右下角经过了至少一个黑色格子的路线数量，二者相减就得到了只经过白色格子的路线数。

把所有黑色格子按照行、列坐标递增的顺序排序。特别地，我们假设左上角是第 0 个黑色格子，右下角是第 $N+1$ 个黑色格子。设第 i 个黑色格子在第 x_i 行第 y_i 列。设 $F[i]$ 表示从左上角走到排序后的第 i 个黑色格子，并且途中不经过其他黑色格子的路线数。

$$F[i] = C_{x_i-1+y_i-1}^{x_i-1} - \sum_{j=0}^{i-1} F[j] * C_{x_i-x_j+y_i-y_j}^{x_i-x_j}，\text{其中 } x_i \ge x_j, y_i \ge y_j$$

上式的含义是，枚举 j 作为从左上角到 (x_i,y_i) 经过的**第一个**黑色格子。也就是从左上角到 (x_j,y_j) 不能经过其他黑色格子，路线数为 $F[j]$。从 (x_j,y_j) 到 (x_i,y_i) 随便走，路线数就是一个组合数。等式中求和符号的部分求出了从左上角到 (x_i,y_i)，途中经过至少一个黑色格子的路线数，用总数减掉，就得到了 $F[i]$。

我们为什么这样设计状态转移方程？为什么限制 j 作为经过的第一个黑色格子



呢？请读者回顾 0x53 节“区间 DP”中的“金字塔”一题，在该题中我们提到，一个状态划分成的若干个子问题之间应该具有互斥性，才不会造成重复统计。实际上，我们在求解计数类动态规划问题时，通常要找到一个“基准点”，围绕这个基准点构造一个不可划分的“整体”，以避免子问题之间的重叠。既然需要求出从左上角到 (x_i, y_i) 途中经过至少一个黑色格子的路线数，就枚举第一个经过的黑色格子 j，使左上角到 j 构成一个整体，让这段路程只能经过白色格子，不能再进行拆分。因为第一个经过的黑色格子不同，所以不同的 j 对应的路线肯定不会重复。而 j 又取遍了 i 之前的所有黑色格子，所以路线也不会遗漏。结合乘法原理和加法原理，我们就得到了状态转移方程中求和符号的部分。

以 $F[0] = 1$ 为初值，计算上面的状态转移方程，最终 $F[N+1]$ 就是本题的答案。对于组合数，可以预处理阶乘关于 $10^9 + 7$ 的乘法逆元来计算。

```
#define x first
#define y second
const int SIZE = 2010;
pair<int, int> a[SIZE];
int h, w, n, f[SIZE], mod = 1000000007;
long long jc[200010], jcinv[200010];
int C(int n, int m) {
    return jc[n] * jcinv[m] % mod * jcinv[n - m] % mod;
}

long long power(long long a, int b) {
    long long c = 1;
    for (; b; b >>= 1) {
        if (b & 1) c = c*a%mod;
        a = a*a%mod;
    }
    return c;
}

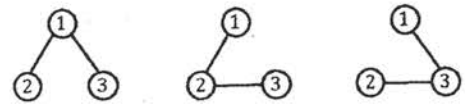
int main() {
    jc[0] = 1, jcinv[0] = 1;
    for (int i = 1; i <= 200000; i++) {
        jc[i] = jc[i - 1] * i % mod;
        jcinv[i] = power(jc[i], mod - 2);
    }
    cin >> h >> w >> n;
```

```
    for (int i = 1; i <= n; i++)
        scanf("%d%d", &a[i].x, &a[i].y);
    sort(a + 1, a + n + 1);
    a[n + 1].x = h, a[n + 1].y = w;
    for (int i = 1; i <= n + 1; i++) {
        f[i] = C(a[i].x + a[i].y - 2, a[i].x - 1);
        for (int j = 1; j < i; j++) {
            if (a[j].x > a[i].x || a[j].y > a[i].y) continue;
            f[i] = (f[i] - (long long)f[j] *
                C(a[i].x + a[i].y - a[j].x - a[j].y,
                a[i].x - a[j].x)) % mod;
        }
    }
    cout << (f[n + 1] + mod) % mod << endl;
}
```

例题 **Connected Graph** AcWing307

求 N 个节点的无向连通图有多少个，节点有标号，编号为 1~N，$1 \le N \le 50$。

注：节点有标号，意味着下列三种是不同的连通图，都应该算作一次计数。



在计数类动态规划中，通常要把一个问题划分成若干个子问题，以便于执行递推。一个连通图不容易进行划分，而一个不连通的无向图则很容易划分成节点更少的两部分。所以我们可以考虑求出 N 个点的无向图总数，减去 N 个点的不连通无向图的数量，就是 N 个点的连通无向图个数。

N 个点的无向图总数显然是 $2^{N*(N-1)/2}$。含义是 N 个点的无向图至多有 $N*(N-1)/2$ 条边，每条边可有可无，所以共有 $2^{N*(N-1)/2}$ 张不同的无向图。

接下来计算 N 个点的不连通无向图的数量。一张不连通的无向图必定由若干个连通块构成。根据上一题提到的“围绕基准点构造一个整体”思想，我们可以枚举**标号为 1 的节点**所在的连通块包含的节点个数 k，从 2~N 这 $N-1$ 个节点中选出 $k-1$ 个与 1 号节点一起构成大小为 k 的连通块。显然，我们有 C_{N-1}^{k-1} 种选法。剩余 $N-k$ 个节点构成任意无向图，有 $2^{(N-k)*(N-k-1)/2}$ 种方法。

综上所述，设 $F[i]$ 表示 i 个节点的无向连通图个数，状态转移方程为：

$$F[i] = 2^{i*(i-1)/2} - \sum_{j=1}^{i-1} F[j] * C_{i-1}^{j-1} * 2^{(i-j)*(i-j-1)/2}$$

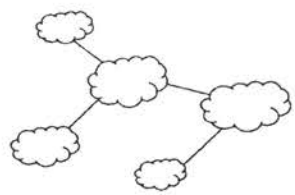
初值：$F[1]=1$。目标：$F[N]$。

例题 How Many of Them? AcWing308

在无向连通图中，若一条边被删除后，图会分成不连通的两部分，则称该边为割边。求满足如下条件的无向连通图的数量：

1. 由 N 个节点构成，节点有标号，编号为 1~N；
2. 割边恰好有 M 条；
3. 没有自环和重边。

数据范围：$2 \le N \le 50$，$0 \le M \le N*(N-1)/2$。答案可能很大，你只需要输出对 10^9+7 取模后的结果。

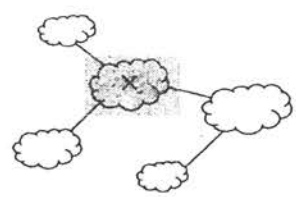


在 0x66 节，我们会详细介绍有关无向图连通性的知识，不熟悉的读者可以简单浏览，或者先记住以下定义和结论。

在无向图中，不含割边的极大连通子图称为双连通分量。若把每个双连通分量看作一个节点，则所有双连通分量和割边构成一个树形结构。如右图所示。

设 $F[i,j]$ 表示 i 个点构成的包含 j 条割边的无向连通图数量。先来考虑 $j>0$ 时的计算方法。

我们多次提到计数类 DP 中“围绕基准点构造一个整体”的思想。在上面的结构中，就以编号为 1 的节点为“基准点”，枚举 1 号节点所在的双连通分量的大小 k。从 2~i 这 $i-1$ 个节点中选出 $k-1$ 个，与 1 号节点共同构成双连通分量，方案数为 $F[k,0]*C_{i-1}^{k-1}$。



接下来考虑图中其他部分。如果去掉 1 号节点所在的双连通分量，那么无向图会分成若干个连通块，从每个连通块出发，都有一条边连接到 1 号节点所在的双连通分量上。例如右图中有 3 个连通块。

设 $G[i,j,k]$ 表示 i 个点构成的、包含 j 条割边的、有 k 个连通块的无向图数量。

回到 $F[i,j]$ 的计算。去掉 1 号节点所在的双连通分量后，图中剩余部分的方案数就是 $G[i-k,j-x,x]$，其中 x 是一个正整数，表示图中剩余部分分为 x 个连通块。每个连通块与 1 号节点所在的双连通分量之间的边都是一条割边，所以剩余部分还需要贡献 $j-x$ 条割边。因为 1 号节点所在的双连通分量包含 k 个节点，每个连通块可以连接到这 k 个节点中的任意一个上，所以连接方案共有 k^x 种。

综上所述，我们得到 $F[i,j]$ $(0<j<i)$ 的状态转移方程：

$$F[i,j]=\sum_{k=1}^{i-1}\left(F[k,0]*C_{i-1}^{k-1}*\sum_{x=1}^{\min(i-k,j)}G[i-k,j-x,x]*k^x\right)$$

特别地，设 $H[i]$ 表示 i 个节点的无向连通图数量，在上一题中我们求出了 $H[i]$，于是：

$$F[i,0]=H[i]-\sum_{j=1}^{i-1}F[i,j]$$

最后，我们考虑 $G[i,j,k]$ 的计算方法。以编号最小的节点所在的连通块为基准，枚举该连通块的大小 p 和割边数量 q。当然，需要从 $i-1$ 个节点中再选出 $p-1$ 个节点构成这个连通块，所以该连通块的方案就有 $F[p,q]*C_{i-1}^{p-1}$ 种。另外，我们需要从该连通块中选出一个节点，用于与刚才删掉的双连通分量相连，显然有 p 种选法。除此之外，图中其他部分的方案数就是 $G[i-p,j-q,k-1]$。于是，我们得到方程：

$$G[i,j,k]=\sum_{p=1}^{i}\sum_{q=0}^{k}F[p,q]*C_{i-1}^{p-1}*p*G[i-p,j-q,k-1]$$

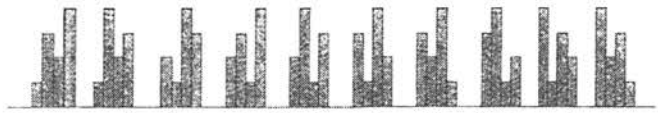
注意：这里算出的 $G[i,j,k]$ 与刚才的定义稍有差别——每个连通块多乘了一个 p，用于与计算 $F[i,j]$ 时删掉的双连通分量相连，因此我们称这些连通块为“有根”连通块更加贴切。这在本题中是必要的，但若抛开本题而言，单纯计算“i 个点构成的、包含 j 条割边的、有 k 个连通块的无向图数量”，则不需要乘 p。

因为 n 的点的无向图至多有 $n-1$ 条割边，所以当 $m\geq n$ 时可以直接输出 0。整个算法的时间复杂度为 $O(n^5)$。在具体实现时，还要注意一些初值、边界、计算顺序等问题，这里就不再赘述，请读者参阅配套资源中的参考程序。本题较为复杂，供学有余力的读者思考。

例题 **A Decorative Fence** AcWing309

有 N 块长方形的木板，长度分别为 $1,2,\cdots,N$，宽度都是 1。现在要用这 N 块木板组成一个宽度为 N 的栅栏，满足在栅栏中，每块木板两侧的木板要么都比它高，要么都比它低。也就是说，栅栏中的木板是高低交错的。我们称“两侧比它低的木板”处于高位，“两侧比它高的木板”处于低位。

显然，有很多种构建栅栏的方案。每个方案可以写作一个长度为 N 的序列，序列中的各元素是木板的长度。把这些序列按照字典序排序，如下页图所示，就是 $N=4$ 时所有满足条件的栅栏按照木板长度的字典序排序后的结果。



All cute fences made of $N = 4$ planks, ordered by their catalogue numbers.

现在给定整数 C，求排名为 C 的栅栏中，各木板的长度从左到右依次是多少。$1 \le N \le 20$，$0 < C < 2^{63}$。

类似于倍增优化 DP 中的“拼凑”思想，我们可以用“试填法”来确定排名为 C 的栅栏中各木板的长度。具体来说，我们可以从小到大枚举第一块木板的长度，当第一块木板长度为 h 时，设后面 $N-1$ 块木板构成栅栏的方案数为 T，若 $T \ge C$，则说明第一块木板的长度就应该是 h，可以继续尝试确定第二块木板的长度。否则，说明第一块木板应该更长，此时令 C 减去 T，h 增加 1，重复上述判断。为了让“试填法”更加高效，我们需要预处理 T 的值。

设 $F[i,j,k]$ 表示用 i 块长度不同的木板构成栅栏，其中最左边的木板的长度从小到大排在第 j 位，并且位置状况为 k（$k=0$ 表示处于低位，$k=1$ 表示处于高位）的方案总数。状态转移方程为：

$$F[i,j,0] = \sum_{p=j}^{i-1} F[i-1,p,1]$$

$$F[i,j,1] = \sum_{p=1}^{j-1} F[i-1,p,0]$$

注意，在动态规划的状态表示中，我们没有采取像“用长度为 $1\sim i$ 的木板、最左边的长度为 j”这种长度绝对确定的表述，而是把 i 和 j 依据相对大小顺序进行定义，这是因为以下两个子问题其实是等价的。

1. 用长度为 $1\sim i$ 的木板构成栅栏，其中最左边的长度为 j。

2. 用 i 块长度不同的木板构成栅栏，其中最左边的长度排名为 j。

实际上，我们之前多次使用的“离散化”思想，就与上面这种“等效代换”思想的本质是相同的。利用上面两个子问题的等价性，我们在进行状态转移、数量统计时会非常容易。

在完成动态规划预处理后，就可以执行“试填法”了。

假设已经确定了从左到右前 $i-1$ 块木板的长度，现在正在考虑第 i 块木板。记录上一块木板的长度 $last$、上一块木板的高低位置情况 k（$k=0$ 表示低位，$k=1$ 表示高位）。

把 k 赋值为 $k \text{ xor } 1$，即可得到第 i 块木板应该处于高位还是低位。

特别地，第 1 块木板既可能处于高位，也可能处于低位。所以要注意单独处理，先

确定下来第 1 块木板的高度和位置情况，作为 $last$ 和 k 的初始值。

从小到大枚举第 i 块木板的长度，设它的真实长度为 len，该长度在剩余木板中从小到大排名为 j。若 k 为 0，需要满足 $len < last$，否则应满足 $len > last$。

若 $F[N-i+1,j,k] < C$，则令 C 减掉 $F[N-i+1,j,k]$，继续尝试更大的 j。

否则，第 i 块木板的长度就应该是 len，可以把它输出。然后，令 i 增加 1，考虑下一块木板。

最后，我们就得到了所求的栅栏方案。动态规划的时间复杂度为 $O(N^3)$，试填过程的时间复杂度为 $O(N^2)$。

```
int t, n; long long m, f[21][21][2];
void prework() {
    f[1][1][0] = f[1][1][1] = 1;
    for (int i = 2; i <= 20; i++)
        for (int j = 1; j <= i; j++) {
            for (int p = j; p <= i - 1; p++)
                f[i][j][0] += f[i - 1][p][1];
            for (int p = 1; p <= j - 1; p++)
                f[i][j][1] += f[i - 1][p][0];
        }
}

int main() {
    prework();
    cin >> t;
    while (t--) {
        cin >> n >> m;
        bool used[21];
        memset(used, 0, sizeof(used));
        int last, k;
        // 第 1 块木板，既可能处于高位，也可能处于低位，单独处理
        for (int j = 1; j <= n; j++) {
            if (f[n][j][1] >= m) {
                last = j, k = 1;
                break;
            }
            else m -= f[n][j][1];
            if (f[n][j][0] >= m) {
                last = j, k = 0;
```

```
                break;
            }
            else m -= f[n][j][0];
        }
        used[last] = 1;
        printf("%d", last);
        // 第 2~n 块木板，高低位置、合法的长度范围与上一块木板有关
        for (int i = 2; i <= n; i++) {
            k ^= 1;
            // 真实长度为 len，在剩余木板中排名为 j
            int j = 0;
            for (int len = 1; len <= n; len++) {
                if (used[len]) continue;
                j++;
                if (k == 0 && len < last || k == 1 && len > last) {
                    if (f[n - i + 1][j][k] >= m) {
                        last = len;
                        break;
                    }
                    else m -= f[n - i + 1][j][k];
                }
            }
            used[last] = 1;
            printf(" %d", last);
        }
        puts("");
    }
}
```

0x5D 数位统计 DP

数位统计 DP 是与数字相关的一类计数问题。在这类题目中，一般给定一些限制条件，求满足限制条件的第 K 小的数是多少，或者求在区间 $[L,R]$ 内有多少个满足限制条件的数。解决方法与上一节的最后一道例题“A Decorative Fence”类似，先用动态规划进行预处理，再基于“拼凑”思想，用“试填法”求出最终的答案。

例题 启示录 AcWing310

古代人认为 666 是属于魔鬼的数。不但如此，只要某数字的十进制表示中有三个连续的 6，古代人也认为这个是魔鬼的数，比如 666, 1666, 2666, 3666, 6663, 16666, 6660666 等，统统是魔鬼的数。古代典籍中经常用“第 X 小的魔鬼的数”来指代这些数，这给研究人员带来了极大的不便。为了帮助他们，你需要写一个程序，输入 X，输出对应的魔鬼数。$X \le 5 * 10^7$，每个测试点包含不超过 1000 组数据。

设 $F[i,3]$ 表示由 i 位数字构成的魔鬼数有多少个，$F[i,j]\ (0 \le j \le 2)$ 表示由 i 位数字构成的、开头已经有连续 j 个 6 的非魔鬼数有多少个。注意，在计算 F 时，我们允许前导 0 的存在。

考虑第 i 位（最高位）是什么数字，容易得到状态转移方程：

$$F[i,0] = 9 * (F[i-1,0] + F[i-1,1] + F[i-1,2])$$

$$F[i,1] = F[i-1,0]$$

$$F[i,2] = F[i-1,1]$$

$$F[i,3] = F[i-1,2] + 10 * F[i-1,3]$$

动态规划完成后，我们先通过 $F[i,3]$ 确定第 X 小的魔鬼数的位数。然后按照“试填法”的思想，从左到右依次考虑每个数位，同时记录当前末尾已经有连续的几个 6。

从小到大枚举当前数位填什么数字，通过预处理的 F 数组可以直接计算出后边的数位有多少种填法可以得到魔鬼数，与 X 比较即可。详细步骤请看参考程序：

```
long long f[21][4];
int t, n, m;
```



```
void prework() {
    f[0][0] = 1;
    for (int i = 0; i < 20; i++) {
        for (int j = 0; j < 3; j++) {
            f[i + 1][j + 1] += f[i][j];
            f[i + 1][0] += f[i][j] * 9;
        }
        f[i + 1][3] += f[i][3] * 10;
    }
}

int main() {
    prework();
    cin >> t; // 数据组数
    while (t--) {
        scanf("%d", &n); // 题目中的X
        // 第n个魔鬼数有m位
        for (m = 3; f[m][3] < n; m++);
        // 试填第i位，末尾已有k个6（k=3也表示已经是魔鬼数）
        for (int i = m, k = 0; i; i--) {
            // 从小到大枚举第i位填的数字j
            for (int j = 0; j <= 9; j++) {
                // 求出后边的i-1位有多少种填法能让整个数是魔鬼数
                long long cnt = f[i - 1][3];
                if (j == 6 || k == 3)
                    for (int l = max(3 - k - (j == 6), 0); l<3; l++)
                        cnt += f[i - 1][l];
                // 如果cnt比n小，说明第n个魔鬼数的第i位应该比j更大
                if (cnt < n) {
                    n -= cnt;
                }
                // 否则，第i位就应该是j
                else {
                    if (k < 3) {
                        if (j == 6) k++; else k = 0;
                    }
                    printf("%d", j);
                    break;
```

```
                }
            }
        }
        cout << endl;
    }
}
```

例题 月之谜 AcWing311

如果一个十进制数能够被它的各位数字之和整除，则称这个数为“月之数”。给定整数 L 和 R，你需要计算闭区间 $[L,R]$ 中有多少个月之数。$1 \le L,R < 2^{31}$，每个测试点包含不超过 3000 组数据。

设 $F[i,j,k,l]$ 表示由 i 位数字构成、各位数字之和是 j、对 k 取模余数是 l 的数有多少个。在计算 F 时，允许前导 0 的存在。枚举第 i 位的数字 p，得到状态转移方程：

$$F[i,j,k,l] = \sum_{p=0}^{9} F\left[i-1, j-p, k, \left(l - p * 10^{i-1}\right) \bmod k\right]$$

闭区间 $[L,R]$ 中月之数的个数，等于 $[1,R]$ 中月之数的个数减去 $[1,L-1]$ 中月之数的个数。接下来以 $[1,R]$ 为例进行说明。

我们还是采取“试填法”的思想，从高位到低位给每一位填数，只要填了一个比上限 R 小的数位，那么后边的数位无论是多少，整个数值都不会超过 R。此时就可以立即把动态规划预处理出的结果累加到答案中。只有在每一位上始终填写与上限 R 相同的数字时，才需要继续向后扫描，所以最终的计算量是数值的“位数”级别的非常小。

具体来说，我们枚举最终的各位数字之和 sum，然后从左到右扫描每个数位，设当前正在处理第 i 位（最高位为第 N 位，最低位为第 1 位），当前已经填写的数字之和是 t，当前数值对 sum 取模余数是 q。我们从小到大枚举第 i 位要填的数字 p。

1. 若 p 小于上限 R 在第 i 位上的数字，则后边 $i-1$ 位可以随便填。因为最终的数值能被 sum 整除，所以第 $1 \sim i$ 位构成的数值对 sum 取模余数应该是 $sum-q$。因此答案直接累加 $F[i-1, sum-t-p, sum, \left(sum-q-p*10^{i-1}\right) \bmod sum]$。

2. 否则，令 $t = t+p$，$q = \left(q + p * 10^{i-1}\right) \bmod sum$，开始处理第 $i-1$ 位。

动态规划的计算量约为 $10 * 9^7$，每组数据查询的计算量约为 $90 * 10 * 9$。



0x5E 总结与练习

本章内容主要涉及两个方面：一是动态规划的模型构建方法，包括线性模型（一般的线性 DP、背包、区间 DP）和树形模型（一般的树形 DP、背包类树形 DP）；二是动态规划的优化，包括状态优化（状态压缩、倍增）和转移优化（数据结构优化、单调队列优化、斜率优化、四边形不等式）。另外，本章也涉及了环形 DP 和有后效性 DP 的处理手段、计数类 DP 与数位统计问题的求解等。动态规划算法的综合性较高，对问题模型抽象、关键信息提炼、算法框架设计和程序细节实现的训练都有很大帮助。

本章知识点

线性 DP

理解动态规划的“阶段”“状态”“决策”三要素

应用动态规划的三个条件：“子问题重叠性”“最优子结构”“无后效性”

能抽象出题目关键点作为状态，并选择覆盖整个状态空间的最小维度集合

简单状态转移方程的设计与实现

通过记录转移来源、递归输出的方法，求出动态规划算法的具体方案

DP 的初步优化：离散化、贪心、变量维护决策集合、前缀和预处理等方法

背包

了解 0/1 背包、完全背包、多重背包、分组背包模型

在传统线性 DP 基础上省略“阶段”维度，控制循环顺序进行背包 DP 的手段

多重背包的二进制拆分优化

区间 DP

区间 DP 的状态设计

区间 DP 的一般转移方式：枚举划分点

DP 的两种等价实现方式：递推（循环）与递归（记忆化搜索）

线性 DP 中“区间”与树形结构中“子树”的联系

树形 DP

树形 DP 的状态设计

树形 DP 的实现方式：深度优先遍历
背包类树形 DP 的实现方式：分组背包转移
不定根的树形 DP：二次扫描与换根法

环形与后效性处理

环形 DP 两种手段，两次 DP（一次断开、一次强制连接）、环拆成链复制一倍
高斯消元求解有后效性的状态转移方程

状态压缩 DP

状态压缩 DP 的状态表示：集合型状态压缩为整数状态的方法，相关的位运算
状态压缩 DP 的状态转移：一般转移、DFS 转移、合法性判定

倍增优化 DP

倍增优化 DP 的状态表示与转移方程设计
倍增优化 DP 的实现：先用动态规划预处理，再用二进制拆分思想进行拼接

数据结构优化 DP

用线段树、树状数组、二叉堆等，维护决策候选集合，优化 DP 的转移

单调队列优化 DP

能够确定决策的取值范围并挖掘其单调性
状态转移方程的变形，分开“只含状态变量”和“只含决策变量”的部分
单调队列优化 DP 的程序实现框架
单调队列的三种通用操作：检查队头合法性、取最优、队尾插入并维护单调性
多重背包的单调队列优化算法

斜率优化

斜率优化与线性规划的联系，建立坐标系，使用斜率和截距分析凸壳形状
单调队列维护凸壳的三种通用操作
插入的坐标或待查询的斜率不具有单调性时的解决方法：二分、平衡树
费用提前计算思想

四边形不等式

四边形不等式的定义和相关定理
一维线性 DP、二维区间 DP 的四边形不等式优化
决策单调性证明、维护与实现

计数类 DP

计数类 DP“不重不漏”的基本原则，加法原理与乘法原理

子问题互斥性，寻找“基准点”的思想，围绕基准点构造一个不可划分的整体

数位统计 DP

数位统计问题的常见模型与求解方法：动态规划预处理、试填法

练习

1. 完成本章中所有例题的代码实现（标有*的题目选做）
2. 乌龟棋

题目地址：AcWing312，提示：线性 DP

3. 花店橱窗

题目地址：AcWing313，提示：线性 DP/输出方案

4. Buy Low Buy Lower

题目地址：AcWing314，提示：线性 DP/统计 LIS 方案数

5. Trip

题目地址：AcWing315，提示：线性 DP/统计 LCS 方案并输出

6. Substract

题目地址：AcWing316，提示：线性 DP

7. 划分大理石

题目地址：AcWing318，提示：背包/多重背包

8. Folding

题目地址：AcWing319，提示：区间 DP

9. 能量项链

题目地址：AcWing320，提示：区间 DP/环拆链并复制一倍

10. 棋盘分割

题目地址：AcWing321，提示：区间 DP/二维平面上的区间 DP

11. *Blocks

题目地址：AcWing322，提示：区间 DP

12. Strategic game

题目地址：AcWing323，提示：树形 DP

13. Bribing FIPA

题目地址：AcWing324，提示：树形 DP/背包类树形 DP

14. Computer

题目地址：AcWing325，提示：树形 DP/二次扫描与换根法

15. XOR 和路径

题目地址：AcWing326，提示：有后效性 DP/高斯消元/数学期望

16. Islands and Bridges

题目地址：AcWing1194，提示：状态压缩 DP

17. Corn Fields

题目地址：AcWing327，提示：状态压缩 DP/填充网格图形

18. *Bugs Integrated Inc

题目地址：AcWing328，提示：状态压缩 DP/填充网格图形

19. Fence Obstacle Course

题目地址：AcWing329，提示：线段树优化 DP

20. Estimation

题目地址：AcWing330，提示：堆优化 DP/中位数

21. 干草堆

题目地址：AcWing331，提示：单调队列优化 DP/贪心

22. 股票交易

题目地址：AcWing332，提示：单调队列优化 DP

23. Largest Submatrix

题目地址：AcWing333，提示：单调队列优化 DP

24. K-Anonymous Sequence

题目地址：AcWing334，提示：斜率优化

25. *特别行动队

题目地址：AcWing335，提示：斜率优化

26. Post Office

题目地址：AcWing336，提示：四边形不等式

27. 陨石的秘密

题目地址：AcWing317，提示：计数类 DP

28. *扑克牌

题目地址：AcWing337，提示：计数类 DP

29. The Counting Problem

题目地址：AcWing338，提示：数位统计 DP

30. Round Numbers

题目地址：AcWing339，提示：数位统计 DP

0x60 图 论

在本章中，我们将探讨图论的相关算法与题目模型。0x61 节和 0x62 节介绍最短路、最小生成树这两个经典模型及其扩展。0x63 节和 0x64 节探讨一些树上的问题，同时扩展到基环树等变体结构。0x65 节介绍最短路中负环的判定方法以及差分约束系统。0x66 节和 0x67 节介绍图连通性的 Tarjan 算法与相关模型，包括割点、割边、双连通分量、强连通分量、2-SAT、欧拉路等。0x68 节和 0x69 节主要介绍二分图，包括二分图最大匹配、最小点覆盖、最大独立集、最小路径覆盖等。0x6A 节初步涉及网络流的内容，使读者开阔眼界，对网络流的概念与简单应用有基本的了解。图论涉及的内容广泛、形象、实际，知识体系的综合性较强，还融合了之前学习的各类算法与数据结构。在数学与信息科学中，图论既有悠久的研究历史，也不失新兴的应用价值，一直是程序设计竞赛的重要关注点。

本章有 30 小时详细视频讲解，读者可以扫描右侧二维码或访问 https://www.acwing.com/activity/content/41/ 了解相关信息。



0x61 最短路

对于一张有向图[①]，我们一般有邻接矩阵和邻接表两种存储方式。对于无向图，可以**把无向边看作两条方向相反的有向边**，从而采用与有向图一样的存储方式。因此，在讨论最短路问题时，我们都以有向图为例。

设有向图 $G=(V,E)$，V 是点集，E 是边集，(x,y) 表示一条从 x 到 y 的有向边，其边权（或称长度）为 $w(x,y)$。设 $n=|V|$，$m=|E|$，**邻接矩阵** A 是一个 $n*n$ 的矩阵，在本书中，我们把它定义为：

$$A[i,j]=\begin{cases}0 & i=j\\ w(i,j) & (i,j)\in E\\ +\infty & (i,j)\notin E\end{cases}$$

邻接矩阵的空间复杂度为 $O(n^2)$。

我们在 0x13 节中已经讲解过了“**邻接表**”，并用数组模拟链表的形式进行了代码实现。长度为 n 的表头数组 head 记录了从每个节点出发的第一条边在 ver 和 edge 数组中的存储位置，长度为 m 的边集数组 ver 和 edge 记录了每条边的终点和边权，长度为 m 的数组 next 模拟了链表指针，表示从相同节点出发的下一条边在 ver 和 edge 数组中的存储位置。邻接表的空间复杂度为 $O(n+m)$。

```
// 加入有向边(x, y)，权值为 z
void add(int x, int y, int z) {
    ver[++tot] = y, edge[tot] = z; // 真实数据
    next[tot] = head[x], head[x] = tot; // 在表头 x 处插入
}

// 访问从 x 出发的所有边
for (int i = head[x]; i; i = next[i]) {
    int y = ver[i], z = edge[i];
    // 找到了一条有向边(x, y)，权值为 z
}
```

① 有向图：图的概念读者应该有所了解，不熟悉的读者可参阅配套资源提供的参考资料。

单源最短路径

单源最短路径问题（Single Source Shortest Path，SSSP 问题）是说，给定一张有向图 $G=(V,E)$，V 是点集，E 是边集，$|V|=n$，$|E|=m$，节点以 $[1,n]$ 之间的连续整数编号，(x,y,z) 描述一条从 x 出发，到达 y，长度为 z 的有向边。设 1 号点为起点，求长度为 n 的数组 $dist$，其中 $dist[i]$ 表示从起点 1 到节点 i 的最短路径的长度。

Dijkstra 算法

Dijkstra 算法的流程如下。

1. 初始化 $dist[1]=0$，其余节点的 $dist$ 值为正无穷大。

2. 找出一个未被标记的、$dist[x]$ 最小的节点 x，然后标记节点 x。

3. 扫描节点 x 的所有出边 (x,y,z)，若 $dist[y]>dist[x]+z$，则使用 $dist[x]+z$ 更新 $dist[y]$。

4. 重复上述 2~3 两个步骤，直到所有节点都被标记。

Dijkstra 算法基于贪心思想，它只适用于所有边的长度都是非负数的图。当边长 z 都是非负数时，全局最小值不可能再被其他节点更新，故在第 1 步中选出的节点 x 必然满足：$dist[x]$ 已经是起点到 x 的最短路径。我们不断选择全局最小值进行标记和扩展，最终可得到起点 1 到每个节点的最短路径的长度。

```
int a[3010][3010], d[3010], n, m;
bool v[3010];
void dijkstra() {
    memset(d, 0x3f, sizeof(d)); // dist 数组
    memset(v, 0, sizeof(v)); // 节点标记
    d[1] = 0;
    for (int i = 1; i < n; i++) { // 重复进行 n-1 次
        int x = 0;
        // 找到未标记节点中 dist 最小的
        for (int j = 1; j <= n; j++)
            if (!v[j] && (x == 0 || d[j] < d[x])) x = j;
        v[x] = 1;
        // 用全局最小值点 x 更新其他节点
        for (int y = 1; y <= n; y++)
            d[y] = min(d[y], d[x] + a[x][y]);
    }
}
```

```
int main() {
    cin >> n >> m;
    // 构建邻接矩阵
    memset(a, 0x3f, sizeof(a));
    for (int i = 1; i <= n; i++) a[i][i] = 0;
    for (int i = 1; i <= m; i++) {
        int x, y, z;
        scanf("%d%d%d", &x, &y, &z);
        a[x][y] = min(a[x][y], z);
    }
    // 求单源最短路径
    dijkstra();
    for (int i = 1; i <= n; i++)
        printf("%d\n", d[i]);
}
```

上面程序的时间复杂度为 $O(n^2)$，主要瓶颈在于第1步的寻找全局最小值的过程。可以用二叉堆（C++ STL priority_queue，0x71 节）对 *dist* 数组进行维护，用 $O(\log n)$ 的时间获取最小值并从堆中删除，用 $O(\log n)$ 的时间执行一条边的扩展和更新，最终可在 $O(m \log n)$ 的时间[①]内实现 Dijkstra 算法。

```
const int N = 100010, M = 1000010;
int head[N], ver[M], edge[M], Next[M], d[N];
bool v[N];
int n, m, tot;
// 大根堆（优先队列），pair 的第二维为节点编号
// pair 的第一维为 dist 的相反数（利用相反数变成小根堆，参见 0x71 节）
priority_queue< pair<int, int> > q;

void add(int x, int y, int z) {
    ver[++tot]=y, edge[tot]=z, Next[tot]=head[x], head[x]=tot;
}

void dijkstra() {
    memset(d, 0x3f, sizeof(d)); // dist 数组
    memset(v, 0, sizeof(v)); // 节点标记
```

① 严格来说，时间复杂度为 $O((m+n)\log n)$。为方便起见，我们假设 m 的规模不小于 n 的规模，因此简写为 $O(m \log n)$，下略。



```
    d[1] = 0;
    q.push(make_pair(0, 1));
    while (q.size()) {
        // 取出堆顶
        int x = q.top().second; q.pop();
        if (v[x]) continue;
        v[x] = 1;
        // 扫描所有出边
        for (int i = head[x]; i; i = Next[i]) {
            int y = ver[i], z = edge[i];
            if (d[y] > d[x] + z) {
                // 更新，把新的二元组插入堆
                d[y] = d[x] + z;
                q.push(make_pair(-d[y], y));
            }
        }
    }
}

int main() {
    cin >> n >> m;
    // 构建邻接表
    for (int i = 1; i <= m; i++) {
        int x, y, z;
        scanf("%d%d%d", &x, &y, &z);
        add(x, y, z);
    }
    // 求单源最短路径
    dijkstra();
    for (int i = 1; i <= n; i++)
        printf("%d\n", d[i]);
}
```

Bellman-Ford 算法和 SPFA[①]算法

给定一张有向图，若对于图中的某一条边 (x, y, z)，有 $dist[y] \leq dist[x] + z$ 成立，则称该边满足三角形不等式。若所有边都满足三角形不等式，则 $dist$ 数组就是所求最

① SPFA 是 Shortest Path Fast Algorithm 的缩写。

短路。

我们先介绍基于迭代思想的 Bellman-Ford 算法。它的流程如下。

1. 扫描所有边 (x,y,z)，若 $dist[y] > dist[x]+z$，则用 $dist[x]+z$ 更新 $dist[y]$。

2. 重复上述步骤，直到没有更新操作发生。

Bellman-Ford 算法的时间复杂度为 $O(nm)$。

实际上，SPFA 算法在国际上通称为“队列优化的 Bellman-Ford 算法”，仅在中国大陆流行“SPFA 算法”的称谓。请读者回顾 0x26 节对“广搜变形”的讨论与总结。SPFA 算法的流程如下。

1. 建立一个队列，最初队列中只含有起点 1。

2. 取出队头节点 x，扫描它的所有出边 (x,y,z)，若 $dist[y] > dist[x]+z$，则使用 $dist[x]+z$ 更新 $dist[y]$。同时，若 y 不在队列中，则把 y 入队。

3. 重复上述步骤，直到队列为空。

在任意时刻，该算法的队列都保存了待扩展的节点。每次入队相当于完成一次 $dist$ 数组的更新操作，使其满足三角形不等式。一个节点可能会入队、出队多次。最终，图中节点收敛到全部满足三角形不等式的状态。这个队列避免了 Bellman-Ford 算法中对不需要扩展的节点的冗余扫描，在随机图上运行效率为 $O(km)$ 级别，其中 k 是一个较小的常数。但在特殊构造的图上，该算法很可能退化为 $O(nm)$，必须谨慎使用。

```
const int N = 100010, M = 1000010;
int head[N], ver[M], edge[M], Next[M], d[N];
int n, m, tot;
queue<int> q;
bool v[N];

void add(int x, int y, int z) {
   ver[++tot]=y, edge[tot]=z, Next[tot]=head[x], head[x]=tot;
}

void spfa() {
   memset(d, 0x3f, sizeof(d)); // dist 数组
   memset(v, 0, sizeof(v)); // 是否在队列中
   d[1] = 0; v[1] = 1;
   q.push(1);
   while (q.size()) {
      // 取出队头
      int x = q.front(); q.pop();
      v[x] = 0;
```

```
        // 扫描所有出边
        for (int i = head[x]; i; i = Next[i]) {
            int y = ver[i], z = edge[i];
            if (d[y] > d[x] + z) {
                // 更新，把新的二元组插入堆
                d[y] = d[x] + z;
                if (!v[y]) q.push(y), v[y] = 1;
            }
        }
    }
}

int main() {
    cin >> n >> m;
    // 构建邻接表
    for (int i = 1; i <= m; i++) {
        int x, y, z;
        scanf("%d%d%d", &x, &y, &z);
        add(x, y, z);
    }
    // 求单源最短路径
    spfa();
    for (int i = 1; i <= n; i++)
        printf("%d\n", d[i]);
}
```

值得一提的是，即使图中存在长度为负数的边，Bellman-Ford 和 SPFA 算法也能够正常工作。

另外，如果图中不存在长度为负数的边，那么类似于优先队列 BFS，我们也可以使用二叉堆（C++ STL priority_queue）对 SPFA 算法进行优化，堆代替了一般的队列，用于保存待扩展的节点，每次取出“当前距离最小”的节点（堆顶）进行扩展，节点第一次从堆中被取出时，就得到了该点的最短路径。这与堆优化 Dijkstra 算法的流程完全一致。“二叉堆优化基于贪心的 Dijkstra 算法”和“优先队列优化基于 BFS 的 SPFA 算法”两种思想殊途同归，都得到了非负权图上 $O(m \log n)$ 的单源最短路径算法。

例题 **Telephone Lines** AcWing340

在郊区有 N 座通信基站，P 条双向电缆，第 i 条电缆连接基站 A_i 和 B_i。特别地，1 号基站是通信公司的总站，N 号基站位于一座农场中。现在，农场主希望对通信线路

进行升级，其中升级第 i 条电缆需要花费 L_i。

电话公司正在举行优惠活动。农场主可以指定一条从1号基站到N号基站的路径，并指定路径上不超过 K 条电缆，由电话公司免费提供升级服务。农场主只需要支付在该路径上剩余的电缆中，升级价格最贵的那条电缆的花费即可。求至少用多少钱能完成升级。

$0 \le K < N \le 1000,\ 1 \le P \le 10000$。

简单来说，本题是在无向图上求出一条从 1 到 N 的路径，使路径上第 $K+1$ 大的边权尽量小。

解法一：分层图最短路径

可以仿照动态规划的思想，用 $D[x,p]$ 表示从 1 号节点到达基站 x，途中已经指定了 p 条电缆免费时，经过的路径上最贵的电缆的花费最小是多少（也就是选择一条从 1 到 x 的路径，使路径上第 $p+1$ 大的边权尽量小）。若有一条从 x 到 y 长度为 z 的无向边，则应该用 $\max(D[x,p],z)$ 更新 $D[y,p]$ 的最小值，用 $D[x,p]$ 更新 $D[y,p+1]$ 的最小值。前者表示不在电缆 (x,y,z) 上使用免费升级服务，后者表示使用。

0x50 章“动态规划”的开头曾指出，所谓“无后效性”其实在告诉我们，**动态规划对状态空间的遍历构成一张有向无环图，遍历顺序就是该有向无环图的一个拓扑序**。有向无环图中的节点对应问题中的“状态”，图中的边则对应状态之间的“转移”，转移的选取就是动态规划中的“决策”。显然，我们刚才设计的状态转移是有后效性的。在有后效性时，一种解决方案就是**利用迭代思想，借助 SPFA 算法进行动态规划**，直至所有状态收敛（不能再更新）。

从最短路径问题的角度去理解，图中的节点也不仅限于“整数编号”，可以扩展到二维，用二元组 (x,p) 代表一个节点，从 (x,p) 到 (y,p) 有长度为 z 的边，从 (x,p) 到 $(y,p+1)$ 有长度为 0 的边。$D[x,p]$ 表示从起点 $(1,0)$ 到节点 (x,p)，路径上最长的边最短是多少。这是 $N*K$ 个点，$P*K$ 条边的广义最短路径问题，被称为**分层图**最短路径。对于非特殊构造的数据，SPFA 算法的时间复杂度为 $O(tNP)$，其中 t 为常数，实际测试可以 AC。本题也让我们进一步领会了动态规划与最短路问题的共通性。

解法二：二分答案，双端队列 BFS

本题的答案显然具有单调性，因为支付的钱更多时，合法的升级方案一定包含了花费更少的升级方案。所以，我们可以二分答案，把问题转化为：是否存在一种合法的升级方法，使花费不超过 mid。

转化后的判定问题非常容易。只需要把升级价格大于 mid 的电缆看作长度为 1 的边，把升级价格不超过 mid 的电缆看作长度为 0 的边，然后求从 1 到 N 的最短路是

否不超过 K 即可。在 0x26 节我们已经学习过，可以用双端队列 BFS 求解这种边权只有 0 和 1 的最短路问题。整个算法时间复杂度为 $O((N+P)\log MAX_L)$。

例题 最优贸易 AcWing341

C 国有 n 个大城市和 m 条道路，每条道路连接这 n 个城市中的某两个城市。任意两个城市之间最多只有一条道路直接相连。这 m 条道路中有一部分为单向通行的道路，一部分为双向通行的道路，双向通行的道路在统计条数时也记为 1 条。

C 国幅员辽阔，各地的资源分布情况各不相同，这就导致了同一种商品在不同城市的价格不一定相同。但是，同一种商品在同一个城市的买入价和卖出价始终是相同的。

商人阿龙来到 C 国旅游。当他得知“同一种商品在不同城市的价格可能会不同”这一信息之后，便决定在旅游的同时，利用商品在不同城市中的差价赚一点旅费。设 C 国 n 个城市的标号从 1~n，阿龙决定从 1 号城市出发，并最终在 n 号城市结束自己的旅行。在旅游的过程中，任何城市可以被重复经过多次，但不要求经过所有城市。

阿龙通过这样的贸易方式赚取旅费：他会选择一个经过的城市买入他最喜欢的商品——水晶球，并在之后经过的另一个城市卖出这个水晶球，用赚取的差价当作旅费。因为阿龙主要是来 C 国旅游，他决定这个贸易只进行最多一次，当然，在赚不到差价的情况下他就无须进行贸易。

现在给出 n 个城市的水晶球价格，m 条道路的信息（每条道路所连接的两个城市的编号以及该条道路的通行情况）。请你告诉阿龙，他最多能赚取多少旅费。

数据范围：$1 \le n \le 10^5, 1 \le m \le 5 * 10^5$，道路是随机生成的。

本题是让我们在一张节点带有权值的图上找出一条从 1 到 n 的路径，使路径上能选出两个点 p,q（先经过 p 后经过 q），并且“节点 q 的权值减去节点 p 的权值”最大。

图中双向通行的道路可看作两条方向相反的单向通行道路。在下面的讨论中，我们均把这张图视为有向图。除此之外，我们再建立一张反图（在原图基础上把所有边的方向取反后得到的图）保存在另外一个邻接表中。

先以 1 为起点，在原图上使用 SPFA 算法，求出一个数组 D，其中 $D[x]$ 表示从节点 1 到节点 x 的所有路径中，能够经过的权值最小的节点的权值。D 数组的计算过程与单源最短路径的计算过程类似，只需要把最短路中的“用 $D[x]+w(x,y)$ 更新 $D[y]$”改为“用 $\min(D[x], price[y])$ 更新 $D[y]$”即可，其中 $price[y]$ 表示点 y 的商品价格。注意上面的更新不满足 Dijkstra 算法的贪心性质，故我们需要选择 SPFA 算法。

再以 n 为起点，在反图上用 SPFA 算法，求出一个数组 F，其中 $F[x]$ 表示在原图上从节点 x 到节点 n（在反图上从 n 到 x）的所有路径中，能够经过的权值最大的节点的权值。F 的计算方法与 D 类似。

最后，枚举每个节点 x，用 $F[x]-D[x]$ 更新答案即可。

因为本题数据为随机生成，所以 SPFA 可以在规定时间内求出结果。更快的做法是求出强连通分量，缩点再使用拓扑序 DP，读者可在学习 0x67 节后再重新回来思考。

例题 道路与航线 AcWing342

Farmer John 正在一个新的销售区域对他的牛奶销售方案进行调查。他想把牛奶送到 $T(1\leq T\leq 25000)$ 个城镇，编号为 1~T。城镇之间通过 $R(1\leq R\leq 50000)$ 条道路和 $P(1\leq P\leq 50000)$ 条航线连接。每条道路或者航线连接城镇 A_i 和 B_i $(1\leq A_i,B_i\leq T)$，花费为 C_i。对于道路，$0\leq C_i\leq 10000$，然而航线的花费很神奇，航线的 C_i 可能是负数，$-10000\leq C_i\leq 10000$。另外，道路都是双向的，航线都是单向的。

事实上，因为最近恐怖主义太嚣张，为了社会和谐，政府出台了一些政策保证：如果有一条航线可以从 A_i 到 B_i，那么不可能通过一些道路和航线从 B_i 回到 A_i。因为 FJ 的牛奶世界公认十分给力，他需要运送牛奶到每一个城镇。他想找到从中心城镇 $S(1\leq S\leq T)$ 把牛奶送到每个城镇的最便宜的方案，或者知道这是不可能的。

本题是一道明显的单源最短路问题，但图中带有负权边，不能使用 Dijkstra 算法。若直接用 SPFA 算法求解，因为测试数据经过了特殊构造，所以程序无法在规定时限内输出答案。题目中有一个特殊条件——双向边都是非负的，只有单向边可能是负的，并且单向边不构成环。我们应该利用这个性质来解答本题。

如果只把双向边（也就是题目中的“道路”）添加到图中，那么会形成若干个连通块。若把每个连通块整体看作一个“点”，再把单向边（也就是题目中的“航线”）添加到图中，会得到一张有向无环图。在有向无环图上，无论边权正负，都可以按照拓扑序进行扫描，在线性时间内求出单源最短路。这启发我们用拓扑序的框架处理整个图，但在双向边构成的每个连通块内部使用堆优化的 Dijkstra 算法快速计算该块内的最短路信息。

详细的算法流程如下。

1. 只把双向边加入到图中，用深度优先遍历划分图中的连通块（第 0x21 节），记 $c[x]$ 为节点 x 所属的连通块的编号。

2. 统计每个连通块的总入度（有多少条边从连通块之外指向连通块之内），记 $\deg[i]$ 为第 i 个连通块的总入度。

3. 建立一个队列（存储连通块编号，用于拓扑排序），最初队列中包含所有总入度为零的连通块编号（当然其中也包括 $c[S]$）。设 $d[S]=0$，其余点的 d 值为 $+\infty$。

4. 取出队头的连通块 i，对该连通块执行堆优化的 Dijkstra 算法，具体步骤为：

(1) 建立一个堆，把第 i 个连通块的所有节点加入堆中；

(2) 从堆中取出 $d[x]$ 最小的节点 x；

(3) 若 x 被扩展过，回到步骤(2)，否则进入下一步；

(4) 扫描从 x 出发的所有边 (x,y,z)，用 $d[x]+z$ 更新 $d[y]$；

(5) 若 $c[x]=c[y]$（仍在连通块内）并且 $d[y]$ 被更新，则把 y 插入堆；

(6) 若 $c[x]\neq c[y]$，则令 deg[c[y]] 减去 1。若减到了 0，则把 $c[y]$ 插入拓扑排序的队列末尾；

(7) 重复步骤(2)~(6)，直至堆为空。

5. 重复步骤 4，直至队列为空，拓扑排序完成。

数组 d 即为所求。

整个算法的时间复杂度为 $O(T+P+R\log T)$。

任意两点间最短路径

为了求出图中任意两点间的最短路径，当然可以把每个点作为起点，求解 N 次单源最短路径问题。不过，在任意两点间最短路问题中，图一般比较稠密。使用 Floyd 算法可以在 $O(N^3)$ 的时间内完成求解，并且程序实现非常简单。

Floyd 算法

设 $D[k,i,j]$ 表示“**经过若干个编号不超过 k 的节点**”从 i 到 j 的最短路长度。该问题可划分为两个子问题，经过编号不超过 $k-1$ 的节点从 i 到 j，或者从 i 先到 k 再到 j。于是：

$$D[k,i,j]=\min(D[k-1,i,j],D[k-1,i,k]+D[k-1,k,j])$$

初值为 $D[0,i,j]=A[i,j]$，其中 A 为本节开头定义的邻接矩阵。

可以看到，Floyd 算法的本质是动态规划。**k 是阶段，所以必须置于最外层循环中。** i 和 j 是附加状态，所以应该置于内层循环。这也解释了为何很多初学者按照 i,j,k 的顺序执行循环，会得到错误的结果。

与背包问题的状态转移方程类似，k 这一维可被省略。最初，我们可以直接用 D 保存邻接矩阵，然后执行动态规划的过程。当最外层循环到 k 时，内层有状态转移：

$$D[i,j]=\min(D[i,j],D[i,k]+D[k,j])$$

最终 $D[i,j]$ 就保存了 i 到 j 的最短路长度。

```
int d[310][310], n, m;
int main() {
    cin >> n >> m;
    // 把 d 数组初始化为邻接矩阵
    memset(d, 0x3f, sizeof(d));
```

```
    for (int i = 1; i <= n; i++) d[i][i] = 0;
    for (int i = 1; i <= m; i++) {
        int x, y, z;
        scanf("%d%d%d", &x, &y, &z);
        d[x][y] = min(d[x][y], z);
    }
    // floyd 求任意两点间最短路径
    for (int k = 1; k <= n; k++)
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
    // 输出
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++)
            printf("%d ", d[i][j]);
        puts("");
    }
}
```

传递闭包

在交际网络中，给定若干个元素和若干对二元关系，且关系具有传递性[①]。“通过传递性推导出尽量多的元素之间的关系”的问题被称为传递闭包。

建立邻接矩阵 d，其中 $d[i,j]=1$ 表示 i 与 j 有关系，$d[i,j]=0$ 表示 i 与 j 没有关系。特别地，$d[i,i]$ 始终为 1。使用 Floyd 算法可以解决传递闭包问题。

```
bool d[310][310];
int n, m;
int main() {
    cin >> n >> m;
    for (int i = 1; i <= n; i++) d[i][i] = 1;
    for (int i = 1; i <= m; i++) {
        int x, y;
        scanf("%d%d", &x, &y);
        d[x][y] = d[y][x] = 1;
    }
```

① 传递性：设 $\odot$ 是定义在集合 S 上的二元关系，若对于 $\forall a,b,c\in S$，只要有 $a\odot b$ 且 $b\odot c$，就必然有 $a\odot c$，则称关系 $\odot$ 具有传递性。

```
    for (int k = 1; k <= n; k++)
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                d[i][j] |= d[i][k] & d[k][j];
}
```

例题 **Sorting It All Out** AcWing343

给定 n 个变量，m 个不等式。不等式之间具有传递性，即若 $A>B$ 且 $B>C$，则 $A>C$。判断这 m 个不等式是否有矛盾。若无矛盾，则判断这 m 个不等式是否能确定每一对变量之间的关系。若能，则求出 t 的最小值，满足仅用前 t 个不等式就能确定每一对变量之间的大小关系。$2 \le n \le 26$。

这是一道“有向”的传递闭包问题，需要在刚才思路的基础上稍作修改。对于每个形如 $i<j$ 的不等式，令 $d[i,j]=1$。对于形如 $i>j$ 的不等式，看作 $j<i$ 进行处理。除了 $i<j$ 之外的情况，均有 $d[i,j]=0$。

使用 Floyd 算法对 d 进行传递闭包。传递闭包完成后，若存在变量 i,j，使得 $d[i,j]$ 和 $d[j,i]$ 均为 1，则说明题目中给出的 m 个不等式矛盾。若存在变量 i,j，使得 $d[i,j]$ 和 $d[j,i]$ 均为 0，则说明题目中给出的 m 个不等式不能确定每一对变量之间的大小关系。

若对于每对变量 i,j，$d[i,j]$ 和 $d[j,i]$ 有且仅有一个为 1，则说明能确定每对变量之间的大小关系。此时，可以再用二分法，二分 t 的值，判断仅用前 t 个不等式能否达到上述条件。

例题 **Sightseeing trip** AcWing344

给定一张无向图，求图中一个至少包含 3 个点的环，环上的节点不重复，并且环上的边的长度之和最小。该问题称为无向图的最小环问题。在本题中，你需要输出最小环的方案，若最小环不唯一，输出任意一个均可。若无解，输出“No solution.”。图的节点数不超过 100。

考虑 Floyd 算法的过程。当外层循环 k 刚开始时，$d[i,j]$ 保存着“经过编号不超过 $k-1$ 的节点”从 i 到 j 的最短路长度。

于是，$\min\limits_{1\le i<j<k}\{d[i,j]+a[j,k]+a[k,i]\}$ 就是满足以下两个条件的最小环长度。

1. 由编号不超过 k 的节点构成。

2. 经过节点 k。

上式中的 i,j 相当于枚举了环上与 k 相邻的两个点。故以上结论显然成立。

$\forall k \in [1,n]$，都对上式进行计算，取最小值，即可得到整张图的最小环。

在该算法中，我们对每个 k 只考虑了由编号不超过 k 的节点构成的最小环，没有考虑编号大于 k 的节点。事实上，由对称性可知，这样做并不会影响结果。

```
int a[310][310], d[310][310], pos[310][310];
int n, m, ans = 0x3f3f3f3f;
vector<int> path; //具体方案

void get_path(int x, int y) {
    if (pos[x][y] == 0) return;
    get_path(x, pos[x][y]);
    path.push_back(pos[x][y]);
    get_path(pos[x][y], y);
}

int main() {
    cin >> n >> m;
    memset(a, 0x3f, sizeof(a));
    for (int i = 1; i <= n; i++) a[i][i] = 0;
    for (int i = 1; i <= m; i++) {
        int x, y, z;
        scanf("%d%d%d", &x, &y, &z);
        a[y][x] = a[x][y] = min(a[x][y], z);
    }
    memcpy(d, a, sizeof(a));
    for (int k = 1; k <= n; k++) {
        for (int i = 1; i < k; i++)
            for (int j = i + 1; j < k; j++)
                if ((long long)d[i][j] + a[j][k] + a[k][i] < ans) {
                    ans = d[i][j] + a[j][k] + a[k][i];
                    path.clear();
                    path.push_back(i);
                    get_path(i, j);
                    path.push_back(j);
                    path.push_back(k);
                }
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                if (d[i][j] > d[i][k] + d[k][j]) {
                    d[i][j] = d[i][k] + d[k][j];
```

```
                pos[i][j] = k;
            }
        }
        if (ans == 0x3f3f3f3f) {
            puts("No solution.");
            return 0;
        }
        for (int i = 0; i < path.size(); i++)
            printf("%d ", path[i]);
        puts("");
    }
```

对于有向图的最小环问题，可枚举起点 $s=1\sim n$，执行堆优化的 Dijkstra 算法求解单源最短路径。s 一定是第一个被从堆中取出的节点，我们扫描 s 的所有出边，当扩展、更新完成后，令 $d[s]=+\infty$，然后继续求解。当 s 第二次被从堆中取出时，$d[s]$ 就是经过点 s 的最小环长度。

例题 **Cow Relays** AcWing345

给定一张由 $T(2\le T\le 100)$ 条边构成的无向图，点的编号为 1~1000 之间的整数。求从起点 S 到终点 E 恰好经过 $N(2\le N\le 10^6)$ 条边（可以重复经过）的最短路。

虽然点的编号在 1~1000 之间，但边最多只有 100 条。我们可以先对点的编号进行离散化，把点的编号映射为 $1\sim P$ 的整数，其中 $P\le 2T$。

在离散化后，设 $P*P$ 的邻接矩阵 A 存储了图中的边。考虑以下方程：

$$\forall i,j\in[1,P],\qquad B[i,j]=\min_{1\le k\le P}\{A[i,k]+A[k,j]\}$$

我们可以认为 $A[i,j]$ 表示从 i 到 j 经过一条边的最短路。在上面的方程中，对于每对二元组 (i,j)，我们枚举了中转点 k，从 i 先到 k，然后再到 j。因此，$B[i,j]$ 表示从 i 到 j 经过两条边的最短路。

一般地，若矩阵 A^m 保存任意两点之间恰好经过 m 条边的最短路，则：

$$\forall i,j\in[1,P],\qquad (A^{r+m})[i,j]=\min_{1\le k\le P}\{(A^r)[i,k]+(A^m)[k,j]\}$$

回顾 0x34 节“矩阵乘法”，我们发现，上式其实等价于一个关于 min 与加法运算的广义“矩阵乘法”。显然，这个广义的“矩阵乘法”也满足结合律。因此，我们可以用快速幂计算出 A^N，具体的实现方法就是：在计算一般的矩阵的幂的基础上，把原来的乘法用加法代替，把原来的加法用 min 运算代替。

最后，$(A^N)[S,E]$ 即为本题的答案。整个算法的时间复杂度为 $O(T^3\log N)$。

0x62 最小生成树

给定一张边带权的无向图 $G=(V,E)$，$n=|V|$，$m=|E|$。由 V 中全部 n 个顶点和 E 中 $n-1$ 条边构成的无向连通子图被称为 G 的一棵生成树。边的权值之和最小的生成树被称为无向图 G 的最小生成树（Minimum Spanning Tree, MST）。

定理

任意一棵最小生成树一定包含无向图中权值最小的边。

证明：

反证法。假设无向图 $G=(V,E)$ 存在一棵最小生成树不包含权值最小的边。设 $e=(x,y,z)$ 是无向图中权值最小的边。把 e 添加到树中，e 会和树上从 x 到 y 的路径一起构成一个环，并且环上其他边的权值都比 z 大。因此，用 e 代替环上的其他任意一条边，会形成一棵权值和更小的生成树，与假设矛盾。故假设不成立，原命题成立。

证毕。

推论

给定一张无向图 $G=(V,E)$，$n=|V|$，$m=|E|$。从 E 中选出 $k<n-1$ 条边构成 G 的一个生成森林。若再从剩余的 $m-k$ 条边中选 $n-1-k$ 条添加到生成森林中，使其成为 G 的生成树，并且选出的边的权值之和最小，则该生成树一定包含这 $m-k$ 条边中连接生成森林的两个不连通节点的权值最小的边。

Kruskal 算法

Kruskal算法就是基于上述推论的。Kruskal算法总是维护无向图的最小生成森林。最初，可认为生成森林由零条边构成，每个节点各自构成一棵仅包含一个点的树。

在任意时刻，Kruskal 算法从剩余的边中选出一条权值最小的，并且这条边的两个端点属于生成森林中两棵不同的树（不连通），把该边加入生成森林。图中节点的连通情况可以用并查集维护。

详细来说，Kruskal 算法的流程如下。

1. 建立并查集，每个点各自构成一个集合。
2. 把所有边按照权值从小到大排序，依次扫描每条边 (x,y,z)。

3. 若 x,y 属于同一集合（连通），则忽略这条边，继续扫描下一条。
4. 否则，合并 x,y 所在的集合，并把 z 累加到答案中。
5. 所有边扫描完成后，第 4 步中处理过的边就构成最小生成树。

时间复杂度为 $O(m\log m)$。

```
struct rec { int x, y, z; } edge[500010];
int fa[100010], n, m, ans;
bool operator <(rec a, rec b) {
    return a.z < b.z;
}
int get(int x) {
    if (x == fa[x]) return x;
    return fa[x] = get(fa[x]);
}

int main() {
    cin >> n >> m;
    for (int i = 1; i <= m; i++)
        scanf("%d%d%d", &edge[i].x, &edge[i].y, &edge[i].z);
    // 按照边权排序
    sort(edge + 1, edge + m + 1);
    // 并查集初始化
    for (int i = 1; i <= n; i++) fa[i] = i;
    // 求最小生成树
    for (int i = 1; i <= m; i++) {
        int x = get(edge[i].x);
        int y = get(edge[i].y);
        if (x == y) continue;
        fa[x] = y;
        ans += edge[i].z;
    }
    cout << ans << endl;
}
```

Prim 算法

Prim 算法同样基于上述推论，但思路略有改变。Prim 算法总是维护最小生成树的一部分。最初，Prim 算法仅确定 1 号节点属于最小生成树。

在任意时刻，设已经确定属于最小生成树的节点集合为 T，剩余节点集合为 S。

Prim 算法找到 $\min_{x\in S,y\in T}\{z\}$，即两个端点分别属于集合 S,T 的权值最小的边，然后把点 x 从集合 S 中删除，加入到集合 T，并把 z 累加到答案中。

具体来说，可以维护数组 d：若 $x\in S$，则 $d[x]$ 表示节点 x 与集合 T 中的节点之间权值最小的边的权值。若 $x\in T$，则 $d[x]$ 就等于 x 被加入 T 时选出的最小边的权值。

可以类比 Dijkstra 算法，用一个数组标记节点是否属于 T。每次从未标记的节点中选出 d 值最小的，把它标记（新加入 T），同时扫描所有出边，更新另一个端点的 d 值。最后，最小生成树的权值总和就是 $\sum_{x=2}^{n} d[x]$。

Prim 算法的时间复杂度为 $O(n^2)$，可以用二叉堆优化到 $O(m\log n)$。但用二叉堆优化不如直接使用 Kruskal 算法更加方便。因此，Prim 主要用于稠密图，尤其是完全图的最小生成树的求解。

```
int a[3010][3010], d[3010], n, m, ans;
bool v[3010];

void prim() {
    memset(d, 0x3f, sizeof(d));
    memset(v, 0, sizeof(v));
    d[1] = 0;
    for (int i = 1; i < n; i++) {
        int x = 0;
        for (int j = 1; j <= n; j++)
            if (!v[j] && (x == 0 || d[j] < d[x])) x = j;
        v[x] = 1;
        for (int y = 1; y <= n; y++)
            if (!v[y]) d[y] = min(d[y], a[x][y]);
    }
}

int main() {
    cin >> n >> m;
    // 构建邻接矩阵
    memset(a, 0x3f, sizeof(a));
    for (int i = 1; i <= n; i++) a[i][i] = 0;
    for (int i = 1; i <= m; i++) {
        int x, y, z;
        scanf("%d%d%d", &x, &y, &z);
```

```
            a[y][x] = a[x][y] = min(a[x][y], z);
        }
        // 求最小生成树
        prim();
        for (int i = 2; i <= n; i++) ans += d[i];
        cout << ans << endl;
    }
```

例题 走廊泼水节 AcWing346

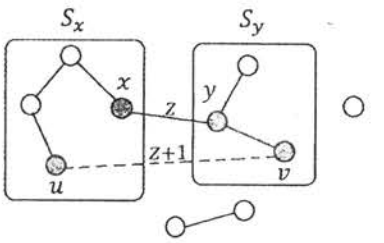
给定一棵 N 个节点的树，要求增加若干条边，把这棵树扩充为完全图，并满足图的唯一最小生成树仍然是这棵树。求增加的边的权值总和最小是多少。

数据范围：$N \le 6000$，原有的边权均为非负整数。

N 个节点的树有 $N-1$ 条边。把这 $N-1$ 条边按照权值从小到大排序，依次扫描每条边，执行一个类似于 Kruskal 算法的过程。

设当前扫描到边 (x,y,z)，x 所在的并查集为 S_x，y 所在的并查集为 S_y，此时应该合并 S_x 与 S_y。合并后的集合 $S_x \cup S_y$ 构成一棵树的结构。

$\forall u \in S_x, v \in S_y$，若 $(u,v) \neq (x,y)$，则在最终的完全图中，我们肯定需要在 (u,v) 之间增加一条边。于是，无向边 (u,v)、S_x 中从 u 到 x 的路径、无向边 (x,y) 以及 S_y 中从 v 到 y 的路径共同构成一个环。如右图所示。



根据本节开头提到的推论，为了保证 (x,y) 一定在最小生成树中，就必须让 (x,y) 是连接集合 S_x 与 S_y 的权值最小的边[否则就有“用 (u,v) 替换 (x,y)”的方案]。因此，(u,v) 的边权应该定为 $z+1$。

S_x 与 S_y 之间一共会增加 $|S_x| * |S_y| - 1$ 条边，所以我们把 $(z+1) * (|S_x| * |S_y| - 1)$ 累加到答案中即可。算法的时间复杂度为 $O(N \log N)$。

```
struct rec { int x, y, z; } edge[6010];
int fa[6010], s[6010], n, T;
long long ans;
bool operator <(rec a, rec b) {
    return a.z < b.z;
}
int get(int x) {
    if (x == fa[x]) return x;
    return fa[x] = get(fa[x]);
```

```
}

int main() {
    cin >> T;
    while (T--) {
        cin >> n;
        for (int i = 1; i < n; i++)
            scanf("%d%d%d", &edge[i].x, &edge[i].y, &edge[i].z);
        sort(edge + 1, edge + n);
        for (int i = 1; i <= n; i++)
            fa[i] = i, s[i] = 1;
        ans = 0;
        for (int i = 1; i < n; i++) {
            int x = get(edge[i].x);
            int y = get(edge[i].y);
            if (x == y) continue;
            ans += (long long)(edge[i].z + 1) * (s[x] * s[y] - 1);
            fa[x] = y;
            s[y] += s[x];
        }
        cout << ans << endl;
    }
}
```

例题 **Picnic Planning** AcWing347

给定一张 N 个点 M 条边的无向图，求出无向图的一棵最小生成树，满足 1 号节点的度数不超过给定的整数 S。$N \leq 30$。

首先，去掉 1 号节点之后，无向图可能会分成若干个连通块。可以用深度优先遍历划分出图中的每个连通块。设连通块共有 T 个，若 $T > S$，则本题无解。

对于每个连通块，在这个连通块内部求出它的最小生成树，然后从连通块中选出一个节点 p 与 1 号节点相连，其中无向边 $(1,p)$ 的权值尽量小。

此时，我们已经得到了原无向图的一棵生成树，1 号节点的度数为 T。我们还可以尝试改动 $S-T$ 条边，让答案更优。

考虑无向图中从节点 1 出发的每条边 $(1,x)$，边权为 z。如果 $(1,x)$ 还不在当前的生成树中，那么继续找到当前生成树中从 x 到 1 的路径上权值最大的边 (u,v)，边权为 w。求出使得 $w-z$ 最大的点 x_0。若 x_0 对应的 $w_0 - z_0 > 0$，则从树中删掉边

(u_0, v_0)，加入边 $(1, x_0)$，答案就会变小 $w_0 - z_0$。

重复上一步 $S-T$ 次，或者直到 $w_0 - z_0 \le 0$，就得到了题目所求的最小生成树。

例题 最优比率生成树 AcWing348

给定一张 N 个点、M 条边的无向图 $(1 \le N, M \le 10000)$，图中每条边 e 都有一个收益 C_e 和一个成本 R_e。求该图的一棵生成树 T，使树中各边的收益之和除以成本之和，即 $\sum_{e \in T} C_e / \sum_{e \in T} R_e$ 最大。

这是一道 0/1 分数规划问题，请读者回顾 0x39 节的内容。我们可以用二分答案来求解本题。设当前二分的值为 mid。

根据 0/1 分数规划模型，我们只需构建一张新的无向图，图的结构不变，但每条边只有一个权值 $C_e - mid * R_e$。在新的无向图中求最大生成树，若最大生成树上的边权之和非负，则令 $l = mid$，否则令 $r = mid$。

当二分结束时，$(l + r)/2$ 就是本题的答案。

例题 黑暗城堡 AcWing349

在顺利攻破 Lord lsp 的防线之后，lqr 一行人来到了 Lord lsp 的城堡下方。Lord lsp 黑化之后虽然拥有了强大的超能力，能够用意念力制造建筑物，但是智商水平却没怎么增加。现在 lqr 已经搞清楚黑暗城堡有 N 个房间 $(1 \le N \le 1000)$，M 条可以制造的双向通道，以及每条通道的长度。

lqr 深知 Lord lsp 的想法，为了避免每次都要琢磨两个房间之间的最短路径，Lord lsp 一定会把城堡修建成树形的。但是，为了尽量提高自己的移动效率，Lord lsp 一定会使得城堡满足下面的条件：设 $D[i]$ 为如果所有的通道都被修建，第 i 号房间与第 1 号房间的最短路径长度；而 $S[i]$ 为实际修建的树形城堡中第 i 号房间与第 1 号房间的路径长度；要求对于所有整数 $i(1 \le i \le N)$，有 $S[i] = D[i]$ 成立。

为了打败 Lord lsp，lqr 想知道有多少种不同的城堡修建方案。于是 lqr 向 applepi 提出了这个问题。因为 applepi 还要忙着出模拟赛，所以这个任务就交给你了。当然，你只需要输出答案对 $2^{31}-1$ 取模之后的结果就行了。

我们可以用 Dijkstra 算法求出 1 号房间到每个房间的单源最短路。把树形城堡看作以 1 为根的有根树。根据题目要求，若 x 是 y 的父节点，x, y 之间的通道长度为 z，则应该有：$dist[y] = dist[x] + z$。

事实上，我们把满足题目要求的树结构，即对任意一对父子节点 x, y 都有上式成立的树结构，称为图的一棵**最短路径生成树**。

我们可以把所有节点按照 $dist$ 值排序，从小到大依次考虑把每个节点 p 加入树形城堡有多少种方法。与 Prim 算法类似，我们维护“最短路径生成树”的一部分，记

为 T。统计有多少个节点 x 满足 $x \in T$ 并且 $dist[p] = dist[x] + \text{edge}(x,p)$，其中 edge 表示边的长度。让 p 与其中任意一个 x 相连，都符合题目的要求。

根据乘法原理，我们把每一步统计出的数量乘起来，就得到了整道题目的答案。

在有向图中，有一个与无向图最小生成树类似的问题，称为最小树形图问题。这超出了我们的讨论范围，感兴趣的读者可以阅读最小树形图的朱刘算法及相关资料。本章的视频讲解中拓展了这一部分内容。

0x63 树的直径与最近公共祖先

树的直径

给定一棵树，树中每条边都有一个权值，树中两点之间的距离定义为连接两点的路径上的边权之和。树中最远的两个节点之间的距离被称为**树的直径**，连接这两点的路径被称为**树的最长链**。后者通常也可称为直径，即直径既是一个数值概念，也可代指一条路径。

树的直径一般有两种求法，时间复杂度都是 $O(N)$。我们假设树以 N 个点 $N-1$ 条边的无向图的形式给出，并存储在邻接表中。

树形 DP 求树的直径

不妨设 1 号节点为根，“N 个点 $N-1$ 条边的无向图”就可以看作“有根树”。

设 $D[x]$ 表示从节点 x 出发走向以 x 为根的子树，能够到达的最远节点的距离。设 x 的子节点为 $y_1, y_2, \cdots, y_t$，$\text{edge}(x,y)$ 表示边权，显然有：

$$D[x] = \max_{1 \le i \le t} \{D[y_i] + \text{edge}(x, y_i)\}$$

接下来，我们可以考虑对每个节点 x 求出“经过节点 x 的最长链的长度” $F[x]$，整棵树的直径就是 $\max_{1 \le x \le n}\{F[x]\}$。

那么如何求出 $F[x]$ 呢？对于 x 的任意两个节点 y_i 和 y_j，“经过节点 x 的最长链的长度”可以通过四个部分构成：从 y_i 到 y_i 子树中的最远距离，边 (x, y_i)，边 (x, y_j)，从 y_j 到 y_j 子树中的最远距离。不妨设 $j < i$，因此：

$$F[x] = \max_{1 \le j < i \le t} \{D[y_i] + D[y_j] + \text{edge}(x, y_i) + \text{edge}(x, y_j)\}$$

我们没有必要使用两层循环来枚举 i, j。请读者思考计算 $D[x]$ 的过程。在子节点的循环将要枚举到 i 时，$D[x]$ 恰好就保存了从节点 x 出发走向“以 $y_j (j < i)$ 为根的子树”，能够到达的最远节点的距离，这个距离就是 $\max_{1 \le j < i}\{D[y_j] + \text{edge}(x, y_j)\}$。所以，此时我们先用 $D[x] + D[y_i] + \text{edge}(x, y_i)$ 更新 $F[x]$，再用 $D[y_i] + \text{edge}(x, y_i)$ 更新 $D[x]$ 即可。

```
void dp(int x) {
    v[x] = 1;
    for (int i = head[x]; i; i = Next[i]) {
        int y = ver[i];
        if (v[y]) continue;
        dp(y);
        ans = max(ans, d[x] + d[y] + edge[i]);
        d[x] = max(d[x], d[y] + edge[i]);
    }
}
```

两次 BFS 求树的直径

通过两次 BFS 或者两次 DFS 也可以求出树的直径，并且更容易计算出直径上的具体节点。详细地说，这种做法包括两步。

1. 从任意一个节点出发，通过 BFS 或 DFS 对树进行一次遍历，求出与出发点距离最远的节点，记为 p。

2. 从节点 p 出发，通过 BFS 或 DFS 再进行一次遍历，求出与 p 距离最远的节点，记为 q。

从 p 到 q 的路径就是树的一条直径。这是因为 p 一定是直径的一端，否则总能找到一条更长的链，与直径的定义矛盾。请读者尝试详细证明这个结论，此处就不再赘述。既然 p 是直径的一端，那么与 p 距离最远的 q 当然就是直径的另一端了。

在第 2 步的遍历过程中，可以记录下来每个点第一次被访问时的前驱节点。最后从 q 递归回到 p，即可得到直径的具体方案。

值得注意的是，两次 BFS 求直径的方法**只适用于边权均为非负的情况**。如果树上存在负权边，那么刚才的证明就不再成立。根据第一次 BFS 选择的起点的不同，可能会找到不同的“端点 p”，进一步得到不同的直径，可能会得到错误解。读者可以尝试举出一个反例。此时只能使用树形 DP 算法来求树的直径。

树的直径可以作为很多树上问题的突破口。接下来我们给出两道例题。除此之外，读者还将在 0x6B 节的练习中解决树的直径的必须边等问题。

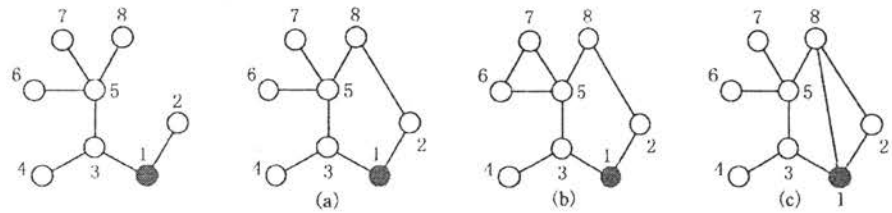
例题 巡逻 AcWing350

在一个地区中有 n 个村庄，编号为 $1,2,\cdots,n$。有 $n-1$ 条道路连接着这些村庄，每条道路刚好连接两个村庄，从任何一个村庄，都可以通过这些道路到达其他任一个村庄。每条道路的长度均为 1 个单位。

为保证该地区的安全，巡警车每天要到所有的道路上巡逻。警察局设在编号为 1 的村庄里，每天巡警车总是从警察局出发，最终又回到警察局。

下图表示一个有 8 个村庄的地区，其中村庄用圆表示（村庄 1 用黑色的圆表示），道路是连接这些圆的线段。为了遍历所有的道路，巡警车需要走的距离为 14 个单位，每条道路都需要经过两次。

为了减少总的巡逻距离，该地区准备在这些村庄之间建立 K 条新的道路，每条新道路可以连接任意两个村庄。两条新道路可以在同一个村庄会合或结束，见下面的图例(c)。一条新道路甚至可以是一个环，即其两端连接到同一个村庄。因为资金有限，K 只能是 1 或 2。同时，为了不浪费资金，每天巡警车必须经过新建的道路正好一次。下图给出了一些建立新道路的例子。

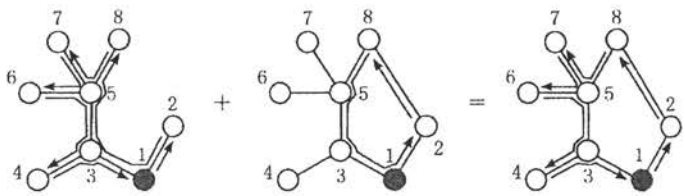


在(a)中，新建了一条道路，总的距离是 11。在(b)中，新建了两条道路，总的巡逻距离是 10。在(c)中，新建了两条道路，但因为巡警车要经过每条新道路正好一次，总的距离变为了 15。

试编写一个程序，读取村庄间道路的信息和需要新建的道路数，计算出最佳的新建道路的方案，使得总的巡逻距离最小，并输出这个最小的巡逻距离。

数据范围：$3 \le n \le 10^5$，$1 \le K \le 2$。

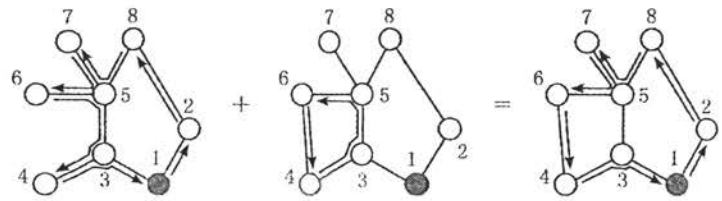
不建立新的道路时，从 1 号节点出发，把整棵树上的每条边遍历至少一次，再回到 1 号节点，会恰好经过每条边两次，路线总长度为 $2(n-1)$，如下图最左边的部分所示。根据树的深度优先遍历思想，很容易证明这个结论，因为每条边必然被递归一次、回溯一次。



建立 1 条新道路之后，因为新道路必须经过恰好一次（0 次、2 次都不可以），所以在沿着新道路 (x,y) 巡逻之后，要返回 x，就必须沿着树上从 y 到 x 的路径巡逻一遍，最终形成一个环。与不建立新道路的情况相结合，相当于树上 x 与 y 之间的路径就只需经过一次了，如上图所示。

因此，当 $K=1$ 时，我们找到树的最长链，在两个端点之间加一条新道路，就能让总的巡逻距离最小。若树的直径为 L，答案就是 $2(n-1)-L+1$。

建立第 2 条新道路 (u,v) 之后，又会形成一个环。若两条新道路形成的环不重叠，则树上 u,v 之间的路径只需经过一次，答案继续减小。否则，在两个环重叠的情况下，如果我们还按照刚才的方法把第 2 个环与建立 1 条新道路的情况相结合，两个环重叠的部分就不会被巡逻到，如下图所示。因为题目要求每条道路必须被巡逻，我们不得不让巡逻车在适当的时候重新巡逻这些边，并且返回。最终的结果是两个环重叠的部分由“只需经过一次”变回了“需要经过两次”。



综上所述，我们得到了如下算法。

1. 在最初的树上求直径，设直径为 L_1。然后把直径上的边权取反(从 1 改为 -1)。

2. 在最长链边权取反之后的树上再次求直径，设直径为 L_2。

答案就是 $2(n-1)-(L_1-1)-(L_2-1)=2n-L_1-L_2$。如果 L_2 这条直径包含 L_1 取反的部分，就相当于两个环重叠。减掉 (L_1-1) 后，重叠的部分变成了“只需经过一次”，减掉 (L_2-1) 后，相当于把重叠的部分加回来，变回“需要经过两次”，与我们之前的讨论相符。时间复杂度为 $O(n)$。

本题第一次求直径时，边权均为非负并且需要求出具体路径，适合使用两次 BFS。第二次求直径时，存在负权边并且只需要求出直径的长度，应该使用树形 DP。

例题 树网的核 AcWing351

设 $T=(V,E,W)$ 是一个无环且连通的无向图（也称为无根树），每条边带有正整数的权值，我们称 T 为树网（Tree Network），其中 V,E 分别表示节点与边的集合，W 表示各边长度的集合，并设 T 有 n 个节点。

路径：树网中任何两个节点 a,b 都存在唯一的一条简单路径，用 $d(a,b)$ 表示以 a,b 为端点的路径的长度，它是该路径上各边长度之和。我们称 $d(a,b)$ 为 a,b 两节点之间的距离。

一点 v 到一条路径 P 的距离为该点与 P 上的最近的节点的距离：

$$d(v,P)=\min\{d(v,u),u \text{ is on } P\}$$

树网的直径：树网中最长的路径称为树网的直径。对于给定的树网 T，直径不一定是唯一的，但可以证明：各直径的中点(不一定恰好是某个节点，可能在某条边的内部)是唯一的，我们称该点为树网的中心。

偏心距 $ECC(F)$：树网 T 中距路径 F 最远的节点到路径 F 的距离，即：

$$ECC(F)=\max\{d(v,F),v\in V\}$$

任务：对于给定的树网 $T=(V,E,W)$ 和非负整数 s，求一个路径 F，它是某直径上的一段路径（该路径两端均为树网中的节点），其长度不超过 s（可以等于 s），使偏心距 $ECC(F)$ 最小。我们称这个路径为树网 $T=(V,E,W)$ 的核（Core）。必要时，F 可以退化为某个节点。一般来说，在上述定义下，核不一定只有一个，但最小偏心距是唯一的。

NOIP 原题数据范围：$5\leq n\leq 300$，$0\leq s\leq 1000$。加强版：$n\leq 5*10^5$。

题目已经告诉我们，树的直径不唯一，但所有直径必定相交，并且各直径的中点汇聚于同一处。进一步可以得到一个推论：在任意一条直径上求出的最小偏心距都相等。请读者思考这两条结论为什么成立。

解法一：枚举，$O(n^3)$

我们很容易想到一个朴素解法。首先，通过两次 BFS 求出任意一条直径（最长链）。然后，在直径上枚举距离不超过 s 的两个点 p 和 q，p,q 之间的路径就作为“树网的核”，同时，我们把核上的每个节点标记为“已访问”。接下来，从核上的每个节点出发，执行深度优先遍历（不经过有“已访问”标记的节点），求出核以外的每个节点到核的距离，取最大值就得到了核的偏心距。在所有枚举的核中取最小值，就是最小偏心距，即本题答案。整个算法的时间复杂度为 $O(n^3)$。

解法二：枚举+贪心，$O(n^2)$

根据贪心策略，在树网的核的一端 p 固定后，另一端 q 在距离不超过 s 的前提下，显然越远越好。因此，我们只需在直径上枚举 p，然后直接确定 q 的位置，再按照上述算法执行深度优先遍历，时间复杂度可以降低到 $O(n^2)$。

解法三：二分答案，$O(n\log SUM)$

为了通过加强版数据，我们还需进一步优化。容易发现，本题的答案具有单调性，可以二分答案，把问题转化为“验证是否存在一个核，其偏心距不超过二分的值 mid”。

设直径的两个端点为 u 和 v。在直径上找到与 u 的距离不超过 mid 的前提下，距离 u 最远的节点，作为节点 p。类似地，在直径上找到与 v 的距离不超过 mid 的前提下，距离 v 最远的节点，作为节点 q。

根据直径的**最长性**，任何从 u,p 之间分叉离开直径的子树，其最远点与 p 的距离都不会比 u 更远。所以 p,q 就是在满足直径两侧的那部分节点偏心距不超过 mid 的前提下，尽量靠近树网中心的节点。

接下来，我们还需检查 p,q 的距离是否不超过 s，同时用深度优先遍历检查把 p,q 之间的路径作为树网的核时，离核最远的点的距离是否不超过 mid。如果两个条件都满足，就判定验证成功，p,q 之间的路径就是偏心距不超过 mid 的一个合法的核。

该二分算法的时间复杂度为 $O(n\log SUM)$，其中 SUM 表示树网所有边的长度之和。

解法四：分析性质，直接扫描，$O(n)$

我们还可以不用二分法，而是用单调队列，把时间复杂度进一步优化到 $O(n)$。

设直径上的节点为 $u_1,u_2,\cdots,u_t$。与前面几个解法类似，先把这 t 个节点标记为“已访问”，然后通过深度优先遍历，求出 $d[u_i]$，表示从 u_i 出发，不经过直径上的其他节点，能够到达的最远点的距离。

以 $u_i,u_j(i\le j)$ 为端点的树网的核的偏心距就是：

$$\max\left(\max_{i\le k\le j}\{d[u_k]\},\mathrm{dist}(u_1,u_i),\mathrm{dist}(u_j,u_t)\right)$$

此时用单调队列维护 $\max_{i\le k\le j}\{d[u_k]\}$，已经能做到 $O(n)$ 了。但这还不够完美，根据解法三中提到的直径的最长性，上式实际上可简化为：

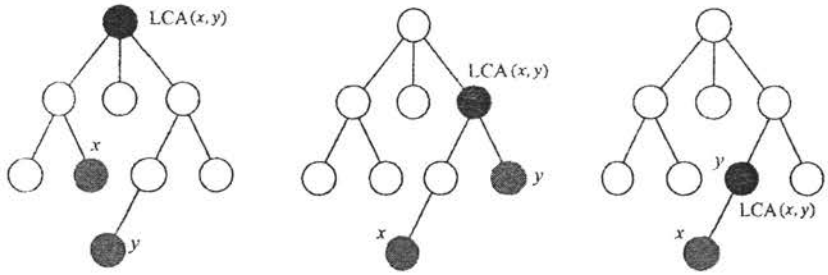
$$\max\left(\max_{1\le k\le t}\{d[u_k]\},\mathrm{dist}(u_1,u_i),\mathrm{dist}(u_j,u_t)\right)$$

$\max_{1\le k\le t}\{d[u_k]\}$ 对于 u_i,u_j 来说是一个定值。所以，我们只需枚举直径上的每个点 u_i，同时用一个指针，在距离不超过 s 的前提下，每次沿着直径向后移动，得到 u_j，然后更新答案即可。对于每个 i，指针 j 的位置是单调递增的，故时间复杂度为 $O(n)$。

经过不断的思考、分析，最终我们没有采用任何高级的算法，仅靠遍历、扫描，就非常简洁、高效地解决了本题。虽然很多问题的时间复杂度是有下界的，但从某种程度上说，算法的设计、优化是永无止境的。读者不应被已有的思维束缚，或只满足于得到AC，而是应该尽可能地挖掘、总结一个模型的相关性质，探究其本质。截至作者完稿，使用搜索引擎都会发现一个有趣的现象，网络上的多数题解均采用了 floyd、单调队列等不必要的解法，提出不同思路者甚少，由此可见，很多算法学习者抄题解而欠思考的现象极其严重，这是每位读者应该避免的行为。

最近公共祖先(LCA)

给定一棵有根树，若节点 z 既是节点 x 的祖先，也是节点 y 的祖先，则称 z 是 x,y 的公共祖先。在 x,y 的所有公共祖先中，深度最大的一个称为 x,y 的最近公共祖先，记为 $\mathrm{LCA}(x,y)$。



$\mathrm{LCA}(x,y)$ 是 x 到根的路径与 y 到根的路径的交会点。它也是 x 与 y 之间的路径上深度最小的节点。求最近公共祖先的方法通常有以下三种。

向上标记法

从 x 向上走到根节点，并标记所有经过的节点。

从 y 向上走到根节点，当第一次遇到已标记的节点时，就找到了 $\mathrm{LCA}(x,y)$。

对于每个询问，向上标记法的时间复杂度最坏为 $O(n)$。

树上倍增法

树上倍增法是一个很重要的算法。除了求 LCA 之外，它在很多问题中都有广泛应用。设 $F[x,k]$ 表示 x 的 2^k 辈祖先，即从 x 向根节点走 2^k 步到达的节点。特别地，若该节点不存在，则令 $F[x,k]=0$。$F[x,0]$ 就是 x 的父节点。除此之外，$\forall k \in [1,\log n]$，$F[x,k]=F[F[x,k-1],k-1]$。

这类似于一个动态规划的过程，“阶段”就是节点的深度。因此，我们可以对树进行广度优先遍历，按照层次顺序，在节点入队之前，计算它在 F 数组中相应的值。

以上部分是预处理，时间复杂度为 $O(n\log n)$，之后可以多次对不同的 x,y 计算 LCA，每次询问的时间复杂度为 $O(\log n)$。

基于 F 数组计算 $\mathrm{LCA}(x,y)$ 分为以下几步。

1. 设 $d[x]$ 表示 x 的深度。不妨设 $d[x]\geq d[y]$（否则可交换 x,y）。

2. 用二进制拆分思想，把 x 向上调整到与 y 同一深度。

具体来说，就是依次尝试从 x 向上走 $k=2^{\log n},\cdots,2^1,2^0$ 步，检查到达的节点是否比 y 深。在每次检查中，若是，则令 $x=F[x,k]$。

3. 若此时 $x=y$，说明已经找到了 LCA，LCA 就等于 y。

这就是上面的图中的第三种情况。

4. 用二进制拆分思想，把 x,y 同时向上调整，并保持深度一致且二者不相会。

具体来说，就是依次尝试把 x,y 同时向上走 $k=2^{\log n},\cdots,2^1,2^0$ 步，在每次尝试中，若 $F[x,k]\neq F[y,k]$（即仍未相会），则令 $x=F[x,k],y=F[y,k]$。

5. 此时 x,y 必定只差一步就相会了，它们的父节点 $F[x,0]$ 就是 LCA。

请读者画图理解，并参阅下面的程序。程序以模板题 HDOJ2586 “How far away?”

为例，多次查询树上两点之间的距离，时间复杂度为 $O((n+m)\log n)$。

```
const int SIZE = 50010;
int f[SIZE][20], d[SIZE], dist[SIZE];
int ver[2 * SIZE], Next[2 * SIZE], edge[2 * SIZE], head[SIZE];
int T, n, m, tot, t;
queue<int> q;

void add(int x, int y, int z) {
    ver[++tot]=y; edge[tot]=z; Next[tot]=head[x]; head[x]=tot;
}

void bfs() {  // 预处理
    q.push(1); d[1] = 1;
    while (q.size()) {
        int x = q.front(); q.pop();
        for (int i = head[x]; i; i = Next[i]) {
            int y = ver[i];
            if (d[y]) continue;
            d[y] = d[x] + 1;
            dist[y] = dist[x] + edge[i];
            f[y][0] = x;
            for (int j = 1; j <= t; j++)
                f[y][j] = f[f[y][j - 1]][j - 1];
            q.push(y);
        }
    }
}

int lca(int x, int y) {  // 回答一个询问
    if (d[x] > d[y]) swap(x, y);
    for (int i = t; i >= 0; i--)
        if (d[f[y][i]] >= d[x]) y = f[y][i];
    if (x == y) return x;
    for (int i = t; i >= 0; i--)
        if (f[x][i] != f[y][i]) x = f[x][i], y = f[y][i];
    return f[x][0];
}
```

```
int main() {
    cin >> T;
    while (T--) {
        cin >> n >> m;
        t = (int)(log(n) / log(2)) + 1;
        // 清空
        for (int i = 1; i <= n; i++) head[i] = d[i] = 0;
        tot = 0;
        // 读入一棵树
        for (int i = 1; i < n; i++) {
            int x, y, z;

            scanf("%d%d%d", &x, &y, &z);
            add(x, y, z), add(y, x, z);
        }
        bfs();
        // 回答问题
        for (int i = 1; i <= m; i++) {
            int x, y;
            scanf("%d%d", &x, &y);
            printf("%d\n", dist[x] + dist[y] - 2 * dist[lca(x, y)]);
        }
    }
}
```

LCA 的 Tarjan 算法

Tarjan 算法本质上是使用并查集对“向上标记法”的优化。它是一个离线算法，需要把 m 个询问一次性读入，统一计算，最后统一输出。时间复杂度为 $O(n+m)$。

在深度优先遍历的任意时刻，树中节点分为三类。

1. 已经访问完毕并且回溯的节点。在这些节点上标记一个整数 2。

2. 已经开始递归，但尚未回溯的节点。这些节点就是当前正在访问的节点 x 以及 x 的祖先。在这些节点上标记一个整数 1。

3. 尚未访问的节点。这些节点没有标记。

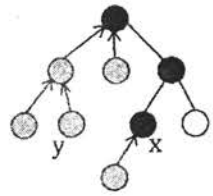
对于正在访问的节点 x，它到根节点的路径已经标记为 1。若 y 是已经访问完毕并且回溯的节点，则 $\mathrm{LCA}(x,y)$ 就是从 y 向上走到根，第一个遇到的标记为 1 的节点。

可以利用并查集进行优化，当一个节点获得整数 2 的标记时，把它所在的集合合

并到它的父节点所在的集合中（合并时它的父节点标记一定为 1，且单独构成一个集合）。

这相当于每个完成回溯的节点都有一个指针指向它的父节点，只需查询 y 所在集合的代表元素（并查集的 get 操作），就等价于从 y 向上一直走到一个开始递归但尚未回溯的节点（具有标记 1），即 $\mathrm{LCA}(x,y)$。

在 x 回溯之前，标记情况与合并情况如下图所示。黑色表示标记为 1，灰色表示标记为 2，白色表示没有标记，箭头表示执行了合并操作。



此时扫描与 x 相关的所有询问，若询问当中的另一个点 y 的标记为 2，就知道了该询问的回答应该是 y 在并查集中的代表元素（并查集中 get(y)函数的结果）。

下面的参考程序以模板题 HDOJ2586 “How far away?” 为例，多次查询树上两点之间的距离，时间复杂度为 $\mathrm{O}(n+m)$。

```
const int SIZE = 50010;
int ver[2 * SIZE], Next[2 * SIZE], edge[2 * SIZE], head[SIZE];
int fa[SIZE], d[SIZE], v[SIZE], lca[SIZE], ans[SIZE];
vector<int> query[SIZE], query_id[SIZE];
int T, n, m, tot, t;

void add(int x, int y, int z) {
    ver[++tot]=y; edge[tot]=z; Next[tot]=head[x]; head[x]=tot;
}
void add_query(int x, int y, int id) {
    query[x].push_back(y), query_id[x].push_back(id);
    query[y].push_back(x), query_id[y].push_back(id);
}

int get(int x) {
    if (x == fa[x]) return x;
    return fa[x] = get(fa[x]);
}

void tarjan(int x) {
    v[x] = 1;
```

```
    for (int i = head[x]; i; i = Next[i]) {
        int y = ver[i];
        if (v[y]) continue;
        d[y] = d[x] + edge[i];
        tarjan(y);
        fa[y] = x;
    }
    for (int i = 0; i < query[x].size(); i++) {
        int y = query[x][i], id = query_id[x][i];
        if (v[y] == 2) {
            int lca = get(y);
            ans[id] = min(ans[id], d[x] + d[y] - 2 * d[lca]);
        }
    }
    v[x] = 2;
}

int main() {
    cin >> T;
    while (T--) {
        cin >> n >> m;
        for (int i = 1; i <= n; i++) {
            head[i] = 0; fa[i] = i, v[i] = 0;
            query[i].clear(), query_id[i].clear();
        }
        tot = 0;
        for (int i = 1; i < n; i++) {
            int x, y, z;
            scanf("%d%d%d", &x, &y, &z);
            add(x, y, z), add(y, x, z);
        }
        for (int i = 1; i <= m; i++) {
            int x, y;
            scanf("%d%d", &x, &y);
            if (x == y) ans[i] = 0;
            else {
                add_query(x, y, i);
```

```
                ans[i] = 1 << 30;
            }
        }
        tarjan(1);
        for (int i = 1; i <= m; i++) printf("%d\n", ans[i]);
    }
}
```

树上差分

在 0x03 节“前缀和与差分”中，我们定义了一个序列的前缀和与差分序列，并通过差分技巧，把“区间”的增减转化为“左端点加 1，右端点减 1”。根据“差分序列的前缀和是原序列”这一原理，在树上可以进行类似的简化，其中“区间操作”对应为“路径操作”，“前缀和”对应为“子树和”。我们通过具体的例子来详细探讨。

例题 闇の連鎖 AcWing352

传说中的“闇の連鎖”被人们称为 Dark。Dark 是人类内心的黑暗的产物，古今中外的勇者们都试图打倒它。经过研究，你发现 Dark 呈现无向图的结构，图中有 N 个节点和两类边，一类边被称为主要边，而另一类边被称为附加边。Dark 有 $N-1$ 条主要边，并且 Dark 的任意两个节点之间都存在一条只由主要边构成的路径。另外，Dark 还有 M 条附加边。

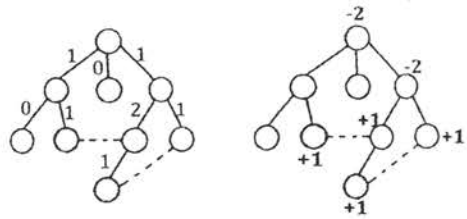
你的任务是把 Dark 斩为不连通的两部分。一开始 Dark 的附加边都处于无敌状态，你只能选择一条主要边切断。一旦你切断了一条主要边，Dark 就会进入防御模式，主要边会变为无敌的，而附加边可以被切断。但是你的能力只能再切断 Dark 的一条附加边。现在你想要知道，一共有多少种方案可以击败 Dark。注意，就算你第一步切断主要边之后就已经把 Dark 斩为两截，你也需要切断一条附加边才算击败了 Dark。

$N \le 10^5$，$M \le 2 * 10^5$。数据保证答案不超过 $2^{31}-1$。

根据题意，“主要边”构成一棵树，“附加边”则是“非树边”。把一条附加边 (x,y) 添加到主要边构成的树中，会与树上 x,y 之间的路径一起形成一个环。如果第一步选择切断 x,y 之间路径上的某条边，那么第二步就必须切断附加边 (x,y)，才能令 Dark 被斩为不连通的两部分。

因此，我们称每条附加边 (x,y) 都把树上 x,y 之间的路径上的每条边“覆盖了一次”。我们只需统计出每条“主要边”被覆盖了多少次。若第一步把被覆盖 0 次的主要边切断，则第二步可任意切断一条附加边。若第一步把被覆盖 1 次的主要边切断，则第二步方法唯一。若第一步把被覆盖 2 次及 2 次以上的主要边切断，则第二步无论如何操作都不能击败 Dark。这样我们就得到了击败 Dark 的方案数。

综上所述，下面我们要解决的问题模型是：**给定一张无向图和一棵生成树，求每条“树边”被“非树边”覆盖了多少次。**下图左侧的例子中标记了覆盖次数，虚线为“非树边”。



解决此问题的经典做法就是“**树上差分**”。我们给树上每个节点一个初始为 0 的权值，然后对每条非树边 (x,y)，令节点 x 的权值加 1，节点 y 的权值加 1，节点 $\mathrm{LCA}(x,y)$ 的权值减 2，如上图右侧的例子所示。最后对这棵生成树进行一次深度优先遍历，求出 $F[x]$ 表示以 x 为根的子树中各节点的权值之和。$F[x]$ 就是 x 与它的父节点之间的“树边”被覆盖的次数。时间复杂度 $\mathrm{O}(N+M)$。

*** 例题** 雨天的尾巴 AcWing353

有 $N\,(N\le 10^5)$ 个点，形成一个树状结构。

有 $M\,(M\le 10^5)$ 次发放操作，每次选择两个点 x,y，对 x 到 y 的路径上（包括 x,y）的每个点发放一袋 $z\,(z\le 10^9)$ 类型的物品。

求完成所有发放操作后，每个点存放最多的是哪种类型的物品。

为了求出每个点存放最多的是哪种类型的物品，我们需要求出每个点上存放的每种物品的数量。一个朴素的做法是：首先对物品的类型进行离散化（至多 M 种不同的类型），然后对每个点 x 建立一个大小为 M 的计数数组 $c[x][1\sim M]$（整个 c 是一个二维数组）。依次执行每次发放操作，对 x 到 y 的路径上的每个点 p，令 $c[p][z]$ 加 1。最终扫描计数数组即可得到答案。时间和空间复杂度均为 $\mathrm{O}(NM)$。

为了避免遍历从 x 到 y 的路径，我们可以应用树上差分技巧。上一道例题是“x 到 y 路径上的所有边被覆盖了一次”，本题相当于是“x 到 y 路径上的所有点被类型 z 覆盖了一次”，略有不同。设 b 是差分以后的计数数组，每次发放操作转化为：点 x 处产生 z，点 y 处产生 z，点 $\mathrm{lca}(x,y)$ 处 z 消失，点 $\mathrm{father}\big(\mathrm{lca}(x,y)\big)$ 处 z 消失。即 $b[x][z]$ 加 1，$b[y][z]$ 加 1，$b[\mathrm{lca}(x,y)][z]$ 减 1，$b\big[\mathrm{father}\big(\mathrm{lca}(x,y)\big)\big][z]$ 减 1。原来的计数数组 c 等于 b 的子树和。详细地说，若 x 的子节点为 $s_1,s_2,\cdots,s_k$，则 $c[x]$ 是 $c[s_1],c[s_2],\cdots,c[s_k]$ 和 $b[x]$ 这些数组对应位置相加以后得到的数组。

为了节省空间并快速地把两个计数数组相加，我们可以使用 0x43 节讲过的“线段树合并”算法。对每个点 x，建立一棵动态开点的权值线段树代替 $b[x]$，支持“修改某个位置”，维护区间最大值以及最大值所在的位置。执行完 M 次发放操作后，对题

目中的树状结构执行一次深度优先遍历，采用“线段树合并”的方式计算子树和，即可解决本题。时间和空间复杂度均为 $O((N+M)\log(N+M))$。

例题 天天爱跑步 AcWing354

小C同学认为跑步非常有趣，于是决定制作一款叫作《天天爱跑步》的游戏。《天天爱跑步》是一款养成类游戏，需要玩家每天按时上线，完成打卡任务。

这个游戏的地图可以看作一棵包含 $n\ (n \le 3*10^5)$ 个节点和 $n-1$ 条边的树，任意两个节点存在一条路径互相可达。树上节点的编号是 $1\sim n$ 之间的连续正整数。

现在有 $m\ (m \le 3*10^5)$ 个玩家，第 i 个玩家的起点为 S_i，终点为 T_i。每天打卡任务开始时，所有玩家在第 0 秒同时从自己的起点出发，以每秒跑一条边的速度，不间断地沿着最短路径向着自己的终点跑去，跑到终点后该玩家就算完成了打卡任务。因为地图是一棵树，所以每个人的路径是唯一的。

小C想知道游戏的活跃度，所以在每个节点上都放置了一个观察员。在节点 j 的观察员会选择在第 W_j 秒观察玩家，一个玩家能被这个观察员观察到，当且仅当该玩家在第 W_j 秒也正好到达了节点 j。小C想知道每个观察员会观察到多少人。

注意：我们认为一个玩家到达自己的终点后，该玩家就会结束游戏，他不能等待一段时间后再被观察员观察到。即对于把节点 j 作为终点的玩家：若他在第 W_j 秒前到达终点，则在节点 j 的观察员不能观察到该玩家；若他正好在第 W_j 秒到达终点，则在节点 j 的观察员可以观察到这个玩家。

每个玩家跑步的路线可以拆成两段：从 S_i 到 $\text{lca}(S_i,T_i)$，从 $\text{lca}(S_i,T_i)$ 到 T_i，其中后者不包括 $\text{lca}(S_i,T_i)$ 这个端点。不难发现，位于节点 x 的观察员能观察到第 i 个玩家，当且仅当满足下列两个条件之一。

1. 点 x 处于 S_i 到 $\text{lca}(S_i,T_i)$ 的路径上，并且满足 $d[S_i]-d[x]=W[x]$，其中数组 d 表示节点在树中的深度。它的意义是玩家从 S_i 跑到 x 所用的时间 $d[S_i]-d[x]$ 等于观察员出现的时间 $W[x]$。

2. 点 x 处于 $\text{lca}(S_i,T_i)$ 到 T_i 的路径上（不含 $\text{lca}(S_i,T_i)$ 这个端点），并且满足 $d[S_i]+d[x]-2*d[lca(S_i,T_i)]=W[x]$。

因为两个条件都包含对 x 所在路径的限制且互不重叠，所以可以分开计算满足每个条件的玩家数量再相加。下面我们以第一个条件为例进行计算。

把关于 i 的变量和关于 x 的变量分别移到等式的两侧，变形为：$d[S_i]=W[x]+d[x]$。类比上一道例题“雨天的尾巴”，不难转化为以下模型。

有 m 个玩家，其中第 i 个玩家给 S_i 到 $\text{lca}(S_i,T_i)$ 的路径上的每个点增加一个类型为 $d[S_i]$ 的物品。最终求每个点 x 处类型为 $W[x]+d[x]$ 的物品有多少个。

通过树上差分，第 i 个玩家的操作可以转化为：在点 S_i 处产生物品 $d[S_i]$，在点

$\text{father}(\text{lca}(S_i,T_i))$ 处物品 $d[S_i]$ 消失。与上一道例题的做法一样，使用动态开点的权值线段树维护计数数组，并用线段树合并算法计算子树和，即可得到答案。

不过本题的数据范围较大，空间不易承受，并且本题只需要求每个点 x 处特定类型的物品数量，不需要求最值。于是我们可以采用更加简单高效的做法。

对树上的每个点建立一个 STL vector（变长数组）。扫描 m 个玩家，把每个玩家的“产生”和“消失”的物品类型记录在对应节点的 vector 中。

建立一个全局数组 c，对每种类型的物品进行计数，起初为全零。

对整棵树执行深度优先遍历。在递归进入每个点 x 时，用一个局部变量 cnt 记录 $c[W[x]+d[x]]$。然后扫描点 x 的 vector，在计数数组里执行修改（类型为 z 的物品产生就令 $c[z]$ 加 1，消失就令 $c[z]$ 减 1）。继续递归遍历 x 的所有子树。从 x 回溯之前，用新的 $c[W[x]+d[x]]$ 减掉 cnt 就是“子树和”，即点 x 处类型为 $W[x]+d[x]$ 的物品数量。

上述做法的依据是：本题统计的信息满足**区间减法**性质。区间减法性质原本指“区间和”可以由“前缀和相减”得到，引申含义是“一段区间的信息”可以由“对应的两段前缀维护的信息”直接推出。在本题中，“遍历以 x 为根的子树过程中累加的值”等于“遍历完成时的值”与“遍历开始时的值”之差。因此，我们可以只维护一个全局数据结构，利用区间减法得到“子树和”。而上一道例题“雨天的尾巴”需要求最值，不满足区间减法性质。

对于第二个条件，只需改为物品 $d[S_i]-2*d[lca(S_i,T_i)]$ 在点 T_i 处产生，在点 $\text{lca}(S_i,T_i)$ 处消失，最后求每个点 x 处类型为 $W[x]-d[x]$ 的物品数量。注意此时的物品类型可能是负数，需要对计数数组的下标范围进行平移或离散化。与第一个条件得到的结果相加，就是点 x 的观察员能观察到的玩家总数。

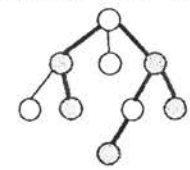
LCA 的综合应用

例题 异象石 AcWing355

Adera 是 Microsoft 应用商店中的一款解谜游戏。

异象石是进入 Adera 中异时空的引导物，在 Adera 的异时空中有一张地图。这张地图上有 N 个点，有 $N-1$ 条双向边把它们连通起来。起初地图上没有任何异象石，在接下来的 M 个时刻中（$1 \le N, M \le 10^5$），每个时刻会发生以下三种类型的事件之一。

1. 地图的某个点上出现了异象石（已经出现的不会再次出现）。
2. 地图某个点上的异象石被摧毁（不会摧毁没有异象石的点）。
3. 向玩家询问使所有异象石所在的点连通的边集的总长度最小是多少。

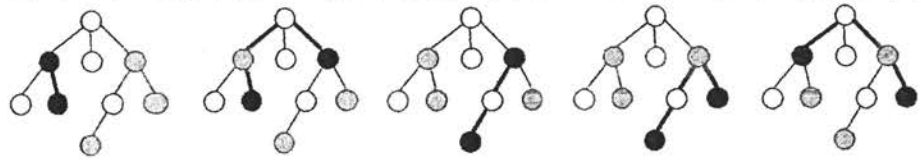


请你作为玩家回答这些问题。右上图是一个例子，灰色节点表示出现了异象石，加

粗的边表示被选为连通异象石的边集。

根据题意，这张地图显然构成一棵树。先对这棵树进行深度优先遍历，求出每个点的时间戳（0x21 节已经介绍了时间戳的概念）。

仔细思考可以发现，如果我们按照时间戳从小到大的顺序，把出现异象石的节点排成一圈（首尾相接），并且累加相邻两个节点之间的路径长度，最后得到的结果恰好是所求答案的两倍。如下图所示，黑色节点表示按照时间戳顺序依次选定的两个节点，加粗的边表示二者之间的路径。五幅图合起来，恰好把上页图中加粗的边集覆盖了两次。



因此，我们可以用一个数据结构（例如 C++ STL set），按照时间戳递增的顺序，维护出现异象石的节点序列，并用一个变量 ans 记录序列中相邻两个节点之间的路径长度之和（序列首尾也看作是相邻的）。

设 $\text{path}(x,y)$ 表示树上 x,y 之间的路径长度，设 $d[x]$ 表示 x 到根节点的路径长度。我们有 $\text{path}(x,y)=d[x]+d[y]-2*d[\text{LCA}(x,y)]$。$d$ 数组可通过一次深度优先遍历全部求出，$\text{path}(x,y)$ 可用树上倍增法求出 LCA 计算。

若一个节点出现了异象石，就依据时间戳，把它插入上述节点序列中适当的位置。设插入的节点为 x，它在序列中前后分别是节点 l 和 r，我们就令 ans 减去 $\text{path}(l,r)$，加上 $\text{path}(l,x)+\text{path}(x,r)$。若一个节点的异象石被摧毁，则类似地更新异象石节点序列以及变量 ans 的值。对于每个询问，直接输出 ans 即可。

整个算法的时间复杂度为 $O\big((N+M)\log N\big)$。

例题 次小生成树 AcWing356

给定一张 N 个点 M 条边的无向图，求无向图的严格次小生成树。设最小生成树的边权之和为 sum，严格次小生成树就是指边权之和大于 sum 的生成树中最小的一个。

数据范围：$N\le 10^5$，$M\le 3*10^5$。

先求出任意一棵最小生成树，设边权之和为 sum。我们称在这棵最小生成树中的 $N-1$ 条边为“树边”，其他 $M-N+1$ 条边为“非树边”。

把一条非树边 (x,y,z) 添加到最小生成树中，会与树上 x,y 之间的路径一起形成一个环。设树上 x,y 之间的路径上的最大边权为 val_1，严格次大边权为 val_2 $(val_1>val_2)$。

若 $z>val_1$，则把 val_1 对应的那条边替换成 (x,y,z) 这条边，就得到了严格次小生成树的一个候选答案，边权之和为 $sum-val_1+z$。

若 $z = val_1$，则把 val_2 对应的那条边替换成 (x, y, z) 这条边，就得到了严格次小生成树的一个候选答案，边权之和为 $sum - val_2 + z$。

枚举每条非树边，添加到最小生成树中，计算出上述所有“候选答案”。在候选答案中取最小值就得到了整张无向图的严格次小生成树。因此，我们要解决的主要问题是：如何快速求出一条路径上的最大边权与严格次大边权。

可以用树上倍增算法来进行预处理。设 $F[x,k]$ 表示 x 的 2^k 辈祖先，$G[x,k,0]$ 与 $G[x,k,1]$ 分别表示从 x 到 $F[x,k]$ 的路径上的最大边权和严格次大边权（最大边权不等于次大边权）。于是 $\forall k \in [1, \log N]$ 有：

$$F[x,k] = F[F[x,k-1],k-1]$$

$$G[x,k,0] = \max(G[x,k-1,0], G[F[x,k-1],k-1,0])$$

$$G[x,k,1] = \begin{cases} \max(G[x,k-1,1], G[F[x,k-1],k-1,1]) & G[x,k-1,0] = G[F[x,k-1],k-1,0] \\ \max(G[x,k-1,0], G[F[x,k-1],k-1,1]) & G[x,k-1,0] < G[F[x,k-1],k-1,0] \\ \max(G[x,k-1,1], G[F[x,k-1],k-1,0]) & G[x,k-1,0] > G[F[x,k-1],k-1,0] \end{cases}$$

当 $k = 0$ 时，有初值：

$$F[x,0] = \text{father}(x)$$

$$G[x,0,0] = \text{edge}(x, \text{father}(x))$$

$$G[x,0,1] = -\infty \text{ (不存在次大值)}$$

接下来，我们考虑每条非树边 (x, y, z)。采用倍增计算 $\text{LCA}(x, y)$ 的框架，x, y 每向上移动一段路径，就把该段路径对应的最大边权、严格次大边权按照与求 G 数组类似的方法合并到答案中，最后即可得到树上 x, y 之间的路径上的最大边权、严格次大边权。整个算法的时间复杂度为 $O(M \log N)$。

例题 疫情控制 AcWing357

H 国有 n $(n \le 5 * 10^4)$ 个城市，这 n 个城市用 $n-1$ 条双向道路相互连通构成一棵树，1 号城市是首都，也是树中的根节点。

H 国的首都暴发了一种危害性极高的传染病。当局为了控制疫情，不让疫情扩散到边境城市（叶子节点所表示的城市），决定动用军队在一些城市建立检查点，使得从首都到边境城市的每一条路径上都至少有一个检查点，边境城市也可以建立检查点。但要注意的是，首都是不能建立检查点的。

现在，在 H 国的一些城市中已经驻扎有军队，且一个城市可以驻扎多支军队。军队总数为 $m(2 \le m \le n)$ 支。一支军队可以在有道路连接的城市间移动，并在除首都以外的任意一个城市建立检查点，且只能在一个城市建立检查点。一支军队经过一条道路从一个城市移动到另一个城市所需要的时间等于道路的长度（单位：小时）。

请问：最少需要多少个小时才能控制疫情？注意：不同的军队可以同时移动。

为了叙述方便，若从首都（根节点）到边境城市（叶子节点）x 的途中经过了军队 i，我们就称军队 i 管辖了节点 x。本题要求我们保证每个叶子节点都有军队管辖。显然，在不移动到根节点的前提下，军队所在的节点深度越浅，能管辖的叶子节点越多。

本题的答案也满足单调性——若 ans 个小时能控制疫情，则对于任意 $k \geq ans$，k 个小时当然也能控制疫情。因此可考虑二分答案，把问题转化为：判定二分的值 mid 小时内能否控制疫情。

军队可分为两类。第一类是在 mid 小时内无法到达根节点的军队。对于这些军队，就让他们尽量往根节点走，mid 小时内能走到哪里，最终就驻扎在哪里即可。

记根节点的子节点集合为 $\text{son}(root)$。处理完第一类军队后，对每个节点 $s \in \text{son}(root)$，统计以 s 为根的子树中是否还有叶子节点尚未被管辖。设 H 是 $\text{son}(root)$ 中还有叶子节点尚未被管辖的节点组成的集合。

第二类是在 mid 小时内能够到达根节点的军队。我们先让这些军队移动到根节点的子节点上（差一步到根节点），用三元组 $(i,s,rest)$ 代表。其中 i 表示军队编号，s 表示军队来自根节点的哪一个子节点，$rest$ 表示军队 i 移动到 s 之后还剩余多少时间。这些军队有两种使用方法：仍然驻扎在 s，或者跨过根节点去管辖其他的子树。

引理：

若存在一支军队 $(i,s,rest)$，满足 $s \in H$ 并且军队 i 在 $rest$ 时间内无法从 s 移动到根再返回 s，则在最优解中，s 一定被自己子树内部的一支军队驻扎，不可能由其他子树的军队跨过根节点过来驻扎。

证明：

因为 $s \in H$，所以需要安排一支军队驻扎在 s。假设 s 由来自另一个子树 s' 的军队 i' 跨过根节点来驻扎，从 s' 到 s 所需时间为 $\text{dist}(root,s') + \text{dist}(root,s)$。

因为存在一支军队 $(i,s,rest)$ 无法从 s 移动到根再返回 s，所以这支军队要么闲置，要么只能跨过根节点驻扎在比 s 离根更近的节点 s''。故对任意的 s''，根据 $\text{dist}(root,s) > \text{dist}(root,s'')$ 可知，i' 一定也有足够的时间去管辖子树 s''。

综上所述，让 i 驻扎在自己的子树 s，把 i' 留作他用，未来的可能性更多，答案不会更差。由贪心的决策包容性可知引理成立。

证毕。

根据这条引理，对于 $\forall s \in H$，若 s 上有军队，并且 s 上 $rest$ 值最小的军队不足以移动到根再返回 s，我们就让 s 上这支 $rest$ 值最小的军队驻扎在 s，管辖以 s 为根的子树。同时，从集合 H 中删除 s。

现在，仍未确定驻扎地点的军队都可以使用第二类方法——跨过根节点去管辖其他子树。集合 H 中剩余的节点就是需要被驻扎的位置。我们把集合 H 中的节点按照

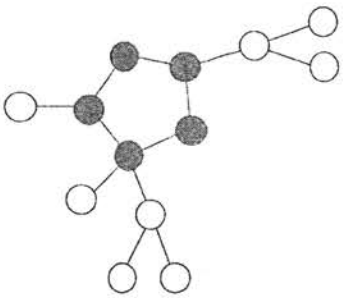
到根的距离从小到大排序，把剩余的军队按照“$rest$ 减去 s 到根的距离”从小到大排序，执行一个贪心扫描算法，让剩余时间短的军队优先驻扎到离根近的节点，让剩余时间长的军队优先驻扎到离根远的节点，判断最后能否把 H 中的节点全部管辖，就知道二分的值是否可行了。整个算法的时间复杂度为 $O((n+m)\log n \log SUM)$，其中 SUM 表示所有边的长度之和。

0x64 基环树

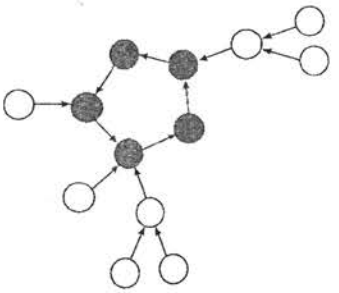
众所周知，N 个点的树有 $N-1$ 条边。若在树上任意添加一条边，则会形成一个环。除了环之外，其余部分由若干棵子树构成。

我们把这种 N 个点 N 条边的连通无向图，即在树上加一条边构成的恰好包含一个环的图，称为“基环树”。如果不保证连通，那么 N 个点 N 条边的无向图也可能是若干棵基环树组成的森林，简称为“基环树森林”。

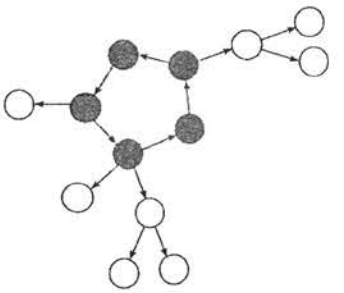
在有向图中，我们也有类似的概念。N 个点、N 条边、每个节点有且仅有一条入边的有向图就好像以“基环”为中心，有向外扩展的趋势，故称为“外向树”。N 个点、N 条边、每个节点有且仅有一条出边的有向连通图就好像以“基环”为中心，有向内收缩的趋势，故称为“内向树”。外向树和内向树也经常统称为“基环树”。如果不保证连通，那么 N 个点、N 条边、每个节点有且仅有一条出（入）边的有向图也可能是“内（外）向树森林”的形态。



基环树
（N 个点 N 条边的连通无向图）



内向树
（每个点有且仅有 1 条出边）



外向树
（每个点有且仅有 1 条入边）

基环树的结构仍然很简单，但比树要复杂些，因此常作为一些经典模型的扩展，以适当增加题目的难度。例如基环树的直径、基环树两点之间的距离、基环树动态规划等。无论哪种模型，求解基环树相关问题的方法一般都是先找出图中唯一的环，把环作为基环树的广义“根节点”，把除了环之外的部分按照若干棵树处理，再考虑与环一起计算。

例题 岛屿 AcWing358

你准备游览一个公园，该公园由 N 个岛屿组成，当地管理部门从每个岛屿出发向另外一个岛屿建了一座桥，不过桥是可以双向行走的。同时，每对岛屿之间都有一艘专用的往来两岛之间的渡船。相对于乘船而言，你更喜欢步行。你希望经过的桥的总长度尽可能长，但受到以下的限制。

1. 可以自行挑选一个岛开始游览。

2. 任何一个岛都不能游览一次以上。

3. 无论任何时间，你都可以由当前所在的岛 S 去另一个从未到过的岛 D。从 S 到 D 有以下方法。

(1) 步行。仅当两个岛之间有一座桥时才有可能。对于这种情况，桥的长度会累加到你步行的总距离中。

(2) 渡船。你可以选择这种方法，仅当没有任何桥和以前使用过的渡船的组合可以由 S 走到 D（当检查是否可到达时，你应该考虑所有的路径，包括经过你曾游览过的那些岛）。

注意，你不必游览所有的岛，也可能无法走完所有的桥。

请你编写一个程序，给定 N 座桥以及它们的长度，按照上述的规则，计算你可以走过的桥的长度之和的最大值。数据范围：$2 \le N \le 10^6$。

先来定义基环树的直径。基环树中最长的简单路径被称为基环树的最长链，其长度被称为基环树的直径。这里，简单路径指的是不自交（不重复经过任何点或边）的路径。

根据题意，整个公园是一个基环树森林的形态。按照题目中给出的乘坐渡船的规则，一旦离开某棵基环树，就不能再通过渡船回来。因此，这道题目的答案就是各棵基环树的直径之和。

如何求出基环树的最长链呢？显然，基环树的最长链可能有两种情况。

1. 在去掉“环”之后的某棵子树中。

2. 经过“环”，其两端分别在去掉“环”上所有边之后的两棵不同子树中。

我们先用一次深度优先遍历找出基环树中的“环”，把“环”上的节点做上标记。设环上的节点为 $s_1, s_2, \cdots, s_t$。

从每个 s_i 出发，在不经过环上其他节点的前提下，再次执行深度优先遍历，即可访问去掉“环”之后以 s_i 为根的子树。在这样的每棵子树中，按照求树的直径的方法进行 DP 并更新答案，即可处理第一种情况。同时，还可以计算出 $D[s_i]$，表示从节点 s_i 出发走向以 s_i 为根的子树，能够到达的最远节点的距离。

最后，我们考虑第二种情况。这相当于找到环上两个不同的节点 s_i, s_j，使得 $D[s_i] + D[s_j] + \text{dist}(s_i, s_j)$ 最大。其中 $\text{dist}(s_i, s_j)$ 表示 s_i, s_j 在环上的距离，有逆时针、顺时针两种走法，取较长的那一种。这等价于我们之前在 0x55 节探讨过的例题“环路

运输”，把环断开成链再复制一倍，用单调队列即可在 $O(N)$ 的时间内解决。

例题 创世纪 AcWing359

上帝手中有 $N(N \le 10^6)$ 种世界元素，每种元素可以限制另外 1 种元素，把第 i 种世界元素能够限制的那种世界元素记为 $A[i]$。现在，上帝要把它们中的一部分投放到一个新的空间中去建造世界。为了世界的和平与安宁，上帝希望所有被投放的世界元素都有至少一个没有被投放的世界元素限制它。上帝希望知道：在此前提下，他最多可以投放多少种世界元素？

把每种世界元素作为图中的节点，从每种元素 i 向它限制的元素 $A[i]$ 连一条有向边。显然，每种世界元素对应的节点都恰好有 1 条出边，所以这张有向图构成内向树森林。

内向树森林中各棵内向树是独立的，可以分别考虑。对于每棵内向树，可以用一次深度优先遍历找出树中唯一的环。任取环上一点 p，断开 p 到 $A[p]$ 的有向边（令 $A[p]=0$），内向树就变为了一棵以 p 为根的树，每个 $A[i]$ 就是 i 的父节点。

在一棵树上求解本题，类似于我们之前在 0x54 节探讨过的例题“没有上司的舞会”。设 $F[x,0]$ 表示在不投放元素 x 的情况下，以 x 为根的子树中最多能投放多少种元素。$F[x,1]$ 表示在投放元素 x 的情况下，以 x 为根的子树中最多能投放多少种元素。

x 不投放时，它的子节点随意，可投放、可不投放：

$$F[x,0]=\sum_{A[y]=x}\max(F[y,0],F[y,1])$$

x 被投放时，至少有一个子节点 y 限制 x，不能投放，其余子节点随意：

$$F[x,1]=\max_{A[y]=x}\left\{1+F[y,0]+\sum_{A[z]=x,\ z\neq y}\max(F[z,0],F[z,1])\right\}$$

断开 p 到 $A[p]$ 的有向边，在以 p 为根的树上求出的结果就是 $\max(F[p,0],F[p,1])$。该结果与原来内向树上的答案只有一点区别——没有利用“p 可以限制 $A[p]$”这条关系。为了弥补这个差别，我们就让 p 强制限制 $A[p]$，在以 p 为根的树上重新进行一次树形 DP，DP 过程中计算 $F[A[p],1]$ 时，让 $A[p]$ 的子节点随意选择投放或不投放，不需要有至少一个子节点限制它（因为它已经被 p 限制）。DP 完成后，只用 $F[p,0]$ 更新答案（p 已经用于限制 $A[p]$，不能被投放）。

这种用两次树形 DP 代替基环树 DP 的方法，就是我们之前在 0x55 节提到的“环形 DP”的解决方案之一，“两次 DP，一次断开，一次强制连接（通过适当的条件和赋值实现）”。读者可以顺便回顾 0x55 节的“Naptime”这道例题。

例题 Freda 的传呼机 AcWing360

为了随时与 Rainbow 快速交流，Freda 制造了两部传呼机。Freda 和 Rainbow 所在的地方有 N 座房屋、M 条双向光缆。每条光缆连接两座房屋，传呼机发出的信号只能沿着光缆传递，并且传呼机的信号从第 i 条光缆的其中一端传递到另一端需要花费 t_i 单位时间。

现在 Freda 要进行 Q 次试验，每次选取两座房屋，并想知道传呼机的信号在这两座房屋之间传递至少需要多长时间。请你帮帮他们吧。

N 座房屋通过光缆一定是连通的，并且这 M 条光缆有以下三类连接情况。

1. 光缆不形成环，也就是光缆仅有 $N-1$ 条。
2. 光缆只形成一个环，也就是光缆仅有 N 条。
3. 每条光缆仅在一个环中（这种情况超出我们的讨论范围，不要求解答）。

数据范围：$2 \le N, M, Q \le 12000$。

第一类数据中的房屋和光缆构成一棵树的结构，光缆的传递时间 t_i 就是边的长度任选一个节点为根，用一次广度优先遍历预处理出倍增 LCA 算法的 F 数组，同时计算出根节点到每个节点 x 的距离 $D[x]$。于是，对于树中任意两点 x, y，在二者之间传递信号的时间就是 $D[x]+D[y]-2*D[\mathrm{LCA}(x,y)]$。

第二类数据中的房屋和光缆构成一棵基环树。在前面两道例题中，我们已经积累了足够的处理基环树的经验，读者应该能够从容应对。先找出“环”，设环上节点为 $s_1, s_2, \cdots, s_t$，然后对去掉“环”之后剩下的每棵子树，以子树与环的交点 s_i 为根，执行倍增 LCA 的预处理。

对于每次试验选取的房屋 x, y，用类似于倍增 LCA 的查询方法，把 x, y 向上移动。若 x, y 在移动到环之前就相会了，则说明 x, y 来自同一棵子树，相会点就是 $\mathrm{LCA}(x,y)$，按照树上的公式来计算 x, y 的距离即可。

若 x, y 分别移动到了环上的 s_i, s_j 两点，则 x, y 之间的最短路径就由子树中 x 到 s_i、环上 s_i 到 s_j、子树中 s_j 到 y 三部分组成。x 到 s_i 的长度就是 $D[x]$，s_j 到 y 的长度就是 $D[y]$。s_i 到 s_j 有顺时针和逆时针两种走法，应该取较短的一边。

在考虑每对 x, y 之前，可以预处理出从 s_1 沿着固定方向依次走到环上各点 $s_2, \cdots, s_t$ 的距离 $G[s_i]$（相当于一个前缀和），即可直接计算出环上 s_i 到 s_j 的两种走法哪一边比较短：$\min\left(\left|G[s_i]-G[s_j]\right|, L-\left|G[s_i]-G[s_j]\right|\right)$，其中 L 表示环的总长度。

题目中第三类连接情况，即每条边至多仅在一个环中的无向图，被称为“边仙人掌”。类似地，每个点至多仅在一个环中的无向图被称为“点仙人掌”。“点仙人掌”一定是“边仙人掌”，故我们一般把“点仙人掌”简称为“仙人掌图”。“仙人掌图”超出了我们的讨论范围，学有余力的读者可以尝试继续解决本题的第三类数据。

0x65 负环与差分约束

负环

给定一张有向图（无向图的每条边可以看作两条方向相反的有向边，从而按照有向图处理），每条边都有一个权值（长度）。若一条边的权值是负数，则称它是负权边。若图中存在一个环，环上各边的权值之和是负数，则称这个环为“负环”。

在 0x61 节中，我们介绍了多种求解单源最短路径问题的算法。这里，我们再次回顾它们的适用条件：

算法名称	能否处理负权边	时间复杂度
Dijkstra	不能，负权可能造成当前最小的 $dist[x]$ 以后不一定最小	$O(n^2)$
Heap-Dijkstra	不能，该算法仅当每个点第一次从堆中取出时执行扩展，等价于选最小的 $dist[x]$	$O(m \log n)$
Bellman-Ford	能，无负环时，最短路包含的边数 $< n$，$n-1$ 轮迭代后一定收敛	$O(nm)$
SPFA	能，本质是队列优化的 Bellman-Ford，负环只是会增加出入队次数	$O(km)$~$O(nm)$

如果图中存在负环，那么直观表现为：无论经过多少轮迭代，总存在有向边 (x,y,z) 使得 $dist[y] > dist[x] + z$，Bellman-Ford 与 SPFA 算法永远不能结束。

根据抽屉原理，若存在一个 $dist[x]$，从起点 1 到节点 x 的最短路径包含 $\geq n$ 条边，则这条路径必然重复经过了某个节点 p。换言之，这条最短路径上存在一个环，环上各点都能更新下一个点的 $dist$ 值。p 绕这个环一圈，最终能更新它自己。因此，这个环的总长度是负数。每绕一圈，最短路径长度只会越来越小，不可能收敛到每条边都满足三角形不等式的状态。基于这个结论，我们有以下的判定法则。

Bellman-Ford 判定负环

若经过 n 轮迭代，算法仍未结束（仍有能产生更新的边），则图中存在负环。

若 $n-1$ 轮迭代之内，算法结束（所有边满足三角形不等式），则图中无负环。

SPFA 判定负环

设 $cnt[x]$ 表示从 1 到 x 的最短路径包含的边数，$cnt[1]=0$。当执行更新 $dist[y]=dist[x]+z$ 时，同样更新 $cnt[y]=cnt[x]+1$。此时若发现 $cnt[y]\geq n$，则图中有负环。若算法正常结束，则图中没有负环。

Bellman-Ford 判定负环的时间复杂度为 $O(nm)$。SPFA 判定负环稍快一些，但最坏情况下仍可能达到 $O(nm)$。

SPFA 判定负环的另一种方法是记录每个点入队的次数，次数达到 n 时说明有负环。这种判定方法的效率不如我们上面提到的判定方法高。例如，在 n 个点构成一个负环的图中，上面提到的判定方法只要绕环一次，就能发现负环，而判定入队次数的做法需要绕环 n 次。在某些题目中，我们甚至可以根据运行时间限制，给队列的总长度（所有点入队总次数）设置一个上界，超出时直接判定存在负环。这样能保证算法不会超时，但可能会产生错误的判定（也就是我们常说的“卡时”做法）。

另外，还有其他优化手段，例如把 SPFA 的队列换成栈、把 SPFA 基于 BFS 改为基于 DFS 等。这些手段确实能提高负环存在时程序运行的效率，但会严重降低负环不存在时计算最短路的效率。本书极其不推荐使用对 SPFA 的任何未经严格证明的优化（包括对时间复杂度上界、时间复杂度期望的证明）。

例题 **Sightseeing Cows** AcWing361

给定一张有向图，每个点都有一个权值 $fun[i]$，每条边有一个权值 $time[i]$。求图中的一个环，使“环上各点的权值之和”除以“环上各边的权值之和”最大。输出这个最大值。

这是一道 0/1 分数规划问题，请读者回顾 0x39 节的内容。

二分答案。设二分的值为实数 mid。

对于一个由节点 $\{v_1,v_2,\cdots,v_t\}$ 以及它们之间的有向边 $\{e_1=(v_1,v_2),\cdots,e_t=(v_t,v_1)\}$构成的环，为了叙述简便，我们把它记为 $S=(\{v_i\},\{e_i\})$。

1. 如果图中存在一个环 S，使得 $\sum_{i=1}^{t}(mid*time[e_i]-fun[v_i])<0$，那么可知：

$$\exists S=(\{v_i\},\{e_i\}),\text{使得 } mid<\frac{\sum_{i=1}^{t}fun[v_i]}{\sum_{i=1}^{t}time[e_i]}$$

也就是说，本题所求的最大值一定大于 mid。

2. 如果对图中任意的环 S，都有 $\sum_{i=1}^{t}(mid*time[e_i]-fun[v_i])\geq 0$，同理可知：

$$\forall S=(\{v_i\},\{e_i\}),\text{都有 } mid\geq\frac{\sum_{i=1}^{t}fun[v_i]}{\sum_{i=1}^{t}time[e_i]}$$

也就是说，本题所求的最大值不超过 mid。

综上所述，对于每轮二分，我们建立一张新图，结构与原图相同，但是没有点权，有向边 $e=(x,y)$ 的权值是 $mid*time[e]-fun[x]$，即本来的边权乘上 mid 再减去入点的权值。

在这张新图上，$\sum_{i=1}^{t}(mid*time[e_i]-fun[v_i])<0$ 的含义就是图中存在负环。因此，我们用 SPFA 算法在新图上求最短路，若有负环，说明 mid 比答案小，应该令 $l=mid$。若最短路的求解正常结束，则令 $r=mid$。二分结束时就求出了本题的答案。

差分约束系统

差分约束系统是一种特殊的 N 元一次不等式组。它包含 N 个变量 X_1~X_N 以及 M 个约束条件，每个约束条件都是由两个变量作差构成的，形如 $X_i-X_j\le c_k$，其中 c_k 是常数（可以是非负数，也可以是负数），$1\le i,j\le N,1\le k\le M$。我们要解决的问题就是：求一组解 $X_1=a_1,X_2=a_2,\cdots,X_N=a_N$，使所有约束条件都得到满足。

差分约束系统的每个约束条件 $X_i-X_j\le c_k$ 可以变形为 $X_i\le X_j+c_k$。这与单源最短路径问题中的三角形不等式 $dist[y]\le dist[x]+z$ 非常相似。因此，可以把每个变量 X_i 看作有向图中的一个节点 i，对于每个约束条件 $X_i-X_j\le c_k$，从节点 j 向节点 i 连一条长度为 c_k 的有向边。

注意到如果 $\{a_1,a_2,\cdots,a_N\}$ 是一组解，那么对任意的常数 Δ，$\{a_1+\Delta,\cdots,a_N+\Delta\}$ 显然也是一组解（作差后 Δ 恰好被消掉）。所以，不妨先求一组负数解，即假设 $\forall i,\ X_i\le 0$，然后再增加一个 0 号节点，令 $X_0=0$。这样一来，就多了 N 个形如 $X_i-X_0\le 0$ 的约束条件，应该从节点 0 向每个节点 i 连一条长度为 0 的有向边。

设 $dist[0]=0$，以 0 为起点求单源最短路。若图中存在负环，则给定的差分约束系统无解。否则，$X_i=dist[i]$ 就是差分约束系统的一组解。

在某些题目中，约束条件形如 $X_i-X_j\ge c_k$。我们仍然可以从 j 到 i 连长度为 c_k 的有向边，只是改为计算单源最长路，若图中有正环则无解。当然，我们也可以把约束条件转化成 $X_j-X_i\le -c_k$，再按照单源最短路进行计算。

例题 **Intervals** AcWing362

给定 n 个闭区间 $[a_i,b_i]$ $(1\le i\le n,\ 0\le a_i,b_i\le 50000)$ 和 n 个整数 c_i $(1\le i\le n)$。你需要构造一个整数集合 Z，使得 $\forall i\in[1,n]$，Z 中满足 $a_i\le x\le b_i$ 的整数 x 不少于 c_i 个。求这样的整数集合 Z 最少包含多少个数。

本题的意思就是从 0~50000 中选出尽量少的整数，使每个区间 $[a_i,b_i]$ 内都有至少 c_i 个数被选。

设 $s[k]$ 表示 0~k 之间最少选出多少个整数。根据题意，有 $s[b_i]-s[a_i-1]\ge$

c_i。这很明显是一个差分约束系统的模型。

不过，我们还要增加一些隐含的条件，才能保证求出的解是有意义的。

1. $s[k]-s[k-1]\geq 0$。0~k 之间选出的数肯定比 0~$k-1$ 多。

2. $s[k]-s[k-1]\leq 1$。每个数只能被选一次。可变形为 $s[k-1]-s[k]\geq -1$。

因此，我们把 -1~50000 这 50002 个整数分别作为图中的节点，从每个 $k-1$ 到 k 连长度为 0 的有向边，k 到 $k-1$ 连长度为 -1 的有向边，从每个 a_i-1 到 b_i 连长度为 c_i 的有向边。

最后，令 $s[-1]=0$，以 -1 为起点求单源最长路。因为本题保证了 $c_i\leq b_i-a_i+1$，所以图中没有正环，差分约束系统一定有解。求完最长路后，$s[50000]=dist[50000]$ 就是本题的答案。

本题也可以用贪心求解，并用数据结构进行优化。读者可以尝试思考一下。

0x66 Tarjan 算法与无向图连通性

无向图的割点与桥

给定无向连通图 $G=(V,E)$：

若对于 $x \in V$，从图中删去节点 x 以及所有与 x 关联的边之后，G 分裂成两个或两个以上不相连的子图，则称 x 为 G 的**割点**。

若对于 $e \in E$，从图中删去边 e 之后，G 分裂成两个不相连的子图，则称 e 为 G 的**桥**或**割边**。

一般无向图（不一定连通）的“割点”和“桥”就是它的各个连通块的“割点”和“桥”。

根据著名计算机科学家 Robert Tarjan 的名字命名的 Tarjan 算法能够在线性时间内求出无向图的割点与桥，进一步可求出无向图的双连通分量（本节后半部分将会介绍）。在有向图方面，Tarjan 算法能够求出有向图的强连通分量、必经点与必经边（下一节将会介绍）。Robert Tarjan 在数据结构方面也做出了很多卓有成效的工作，包括证明并查集的时间复杂度，提出斐波那契堆、Splay Tree 和 Lint-Cut Tree 等。

Tarjan 算法基于无向图的深度优先遍历，在 0x21 节中，我们已经介绍了树与图的深度优先遍历，并引入了“时间戳”的概念。

时间戳

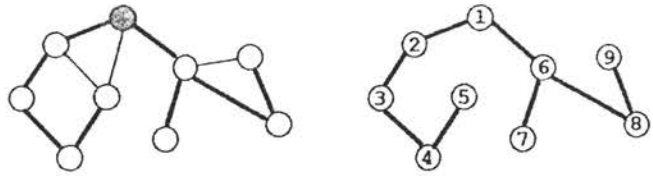
在图的深度优先遍历过程中，按照每个节点第一次被访问的时间顺序，依次给予 N 个节点 $1\sim N$ 的整数标记，该标记就被称为“时间戳”，记为 $dfn[x]$。

搜索树

在无向连通图中任选一个节点出发进行深度优先遍历，每个点只访问一次。所有发生递归的边 (x,y)（换言之，从 x 到 y 是对 y 的第一次访问）构成一棵树，我们把它称为“无向连通图的搜索树”。当然，一般无向图（不一定连通）的各个连通块的搜索树构成无向图的“搜索森林”。

下图左侧展示了一张无向连通图，灰色节点是深度优先遍历的起点，加粗的边是

“发生递归”的边（假设我们在遇到多个分支时，总是优先访问最靠左的一条）。右侧展示了深度优先遍历的搜索树，并标注了节点的时间戳。



追溯值

除了时间戳之外，Tarjan 算法还引入了一个“追溯值” $low[x]$。设 subtree(x) 表示搜索树中以 x 为根的子树。$low[x]$ 定义为以下节点的时间戳的最小值。

1. subtree(x) 中的节点。

2. 通过 1 条不在搜索树上的边，能够到达 subtree(x) 的节点。

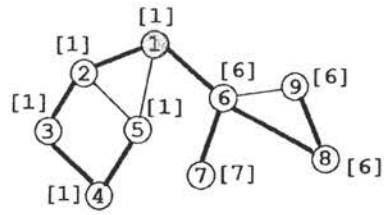
以上图为例。为了叙述简便，我们用时间戳代替节点编号。subtree(2) = {2,3,4,5}。另外，节点 1 通过不在搜索树上的边 (1,5) 能够到达 subtree(2)。所以 $low[2] = 1$。

根据定义，为了计算 $low[x]$，应该先令 $low[x] = dfn[x]$，然后考虑从 x 出发的每条边 (x, y)。

若在搜索树上 x 是 y 的父节点，则令 $low[x] = \min(low[x], low[y])$。

若无向边 (x, y) 不是搜索树上的边，则令 $low[x] = \min(low[x], dfn[y])$。

下图的中括号[]里的数值标注了每个节点的“追溯值” low。



割边判定法则

无向边 (x, y) 是桥，当且仅当搜索树上存在 x 的一个子节点 y，满足：

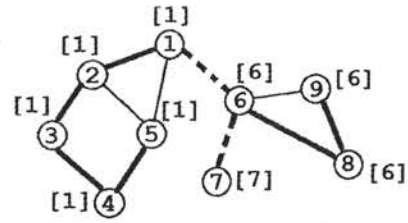
$$dfn[x] < low[y]$$

根据定义，$dfn[x] < low[y]$ 说明从 subtree(y) 出发，在不经过 (x, y) 的前提下，不管走哪条边，都无法到达 x 或比 x 更早访问的节点。若把 (x, y) 删除，则 subtree(y) 就好像形成了一个封闭的环境，与节点 x 没有边相连，图断开成了两部分，因此 (x, y) 是割边。

反之，若不存在这样的子节点 y，使得 $dfn[x] < low[y]$，则说明每个 subtree(y)

都能绕行其他边到达 x 或比 x 更早访问的节点，(x,y) 自然就不是割边。

下面的无向图中有两条“桥”边，用虚线标识。读者不难发现，**桥一定是搜索树中的边，并且一个简单环中的边一定都不是桥。**



下面的程序求出一张无向图中所有的桥。特别需要注意，因为我们遍历的是无向图，所以从每个点 x 出发，总能访问到它的父节点 fa。根据 low 的计算方法，(x,fa) 属于搜索树上的边，且 fa 不是 x 的子节点，故不能用 fa 的时间戳来更新 $low[x]$。

但是，如果仅记录每个节点的父节点，会无法处理重边的情况——当 x 与 fa 之间有多条边时，(x,fa) 一定不是桥。在这些重复的边中，只有一条算是“搜索树上的边”，其他的几条都不算。故有重边时，$dfn[fa]$ 能用来更新 $low[x]$。

一个好的解决方案是：改为记录“递归进入每个节点的边的编号”。编号可认为是边在邻接表中存储的下标位置。在 0x01 和 0x13 节，我们提到过“成对变换”技巧。把无向图的每一条边看作双向边，成对存储在下标“2 和 3”“4 和 5”“6 和 7”…处。若沿着编号为 i 的边递归进入了节点 x，则忽略从 x 出发的编号为 $i \text{ xor } 1$ 的边，通过其他边计算 $low[x]$ 即可。

```
const int SIZE = 100010;
int head[SIZE], ver[SIZE * 2], Next[SIZE * 2];
int dfn[SIZE], low[SIZE], n, m, tot, num;
bool bridge[SIZE * 2];
void add(int x, int y) {
    ver[++tot] = y, Next[tot] = head[x], head[x] = tot;
}

void tarjan(int x, int in_edge) {
    dfn[x] = low[x] = ++num;
    for (int i = head[x]; i; i = Next[i]) {
        int y = ver[i];
        if (!dfn[y]) {
            tarjan(y, i);
            low[x] = min(low[x], low[y]);
            if (low[y] > dfn[x])
```

```
                bridge[i] = bridge[i ^ 1] = true;
        }
        else if (i != (in_edge ^ 1))
            low[x] = min(low[x], dfn[y]);
    }
}

int main() {
    cin >> n >> m;
    tot = 1;
    for (int i = 1; i <= m; i++) {
        int x, y;
        scanf("%d%d", &x, &y);
        add(x, y), add(y, x);
    }
    for (int i = 1; i <= n; i++)
        if (!dfn[i]) tarjan(i, 0);
    for (int i = 2; i < tot; i += 2)
        if (bridge[i])
            printf("%d %d\n", ver[i ^ 1], ver[i]);
}
```

割点判定法则

若 x 不是搜索树的根节点（深度优先遍历的起点），则 x 是割点当且仅当搜索树上存在 x 的一个子节点 y，满足：

$$dfn[x] \le low[y]$$

特别地，若 x 是搜索树的根节点，则 x 是割点当且仅当搜索树上存在至少两个子节点 y_1, y_2 满足上述条件。

证明方法与割边的情形类似，这里就不再赘述。在“割边判定法则”画出的例子中，共有 2 个割点，分别是时间戳为 1 和 6 的两个点。

下面的程序求出一张无向图中所有的割点。因为割点判定法则是小于等于号，所以在求割点时，不必考虑父节点和重边的问题，从 x 出发能访问到的所有点的时间戳都可以用来更新 $low[x]$。

```
const int SIZE = 100010;
int head[SIZE], ver[SIZE * 2], Next[SIZE * 2];
int dfn[SIZE], low[SIZE], stack[SIZE];
```

```
int n, m, tot, num, root;
bool cut[SIZE];
void add(int x, int y) {
    ver[++tot] = y, Next[tot] = head[x], head[x] = tot;
}

void tarjan(int x) {
    dfn[x] = low[x] = ++num;
    int flag = 0;
    for (int i = head[x]; i; i = Next[i]) {
        int y = ver[i];
        if (!dfn[y]) {
            tarjan(y);
            low[x] = min(low[x], low[y]);
            if (low[y] >= dfn[x]) {
                flag++;
                if (x != root || flag > 1) cut[x] = true;
            }
        }
        else low[x] = min(low[x], dfn[y]);
    }
}

int main() {
    cin >> n >> m;
    tot = 1;
    for (int i = 1; i <= m; i++) {
        int x, y;
        scanf("%d%d", &x, &y);
        if (x == y) continue;
        add(x, y), add(y, x);
    }
    for (int i = 1; i <= n; i++)
        if (!dfn[i]) root = i, tarjan(i);
    for (int i = 1; i <= n; i++)
        if (cut[i]) printf("%d ", i);
    puts("are cut-vertexes");
}
```

例题 BLO AcWing363

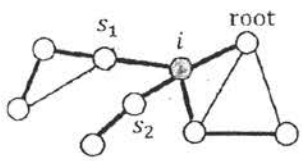
Byteotia 城市有 n 个城镇，m 条双向道路，其中 $n \le 10^5$，$m \le 5 * 10^5$。每条道路连接两个不同的城镇，没有重复的道路，所有城镇连通。把城镇看作节点，把道路看作边。容易发现，整个城市构成一张无向图。

你需要输出 n 个整数，其中第 i 个整数表示把与节点 i 关联的所有边去掉之后（不去掉节点 i 本身），无向图中有多少个有序点对 (x,y)，满足 x 和 y 不连通。

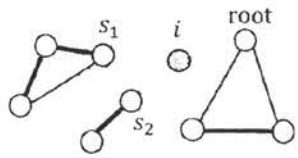
根据割点的定义，若节点 i 不是割点，则把节点 i 关联的所有边去掉之后，只有 i 与其他 $n-1$ 个节点不连通，而其他 $n-1$ 个节点之间还是连通的。注意，题目求的是有序点对，即 (x,y) 和 (y,x) 算不同的点对，故此时答案为 $2*(n-1)$。

若节点 i 是割点，则把节点 i 关联的所有边去掉之后，图会分成若干个连通块。我们应该求出这些连通块的大小，两两相乘再相加。在图论的连通性问题中，我们经常要从搜索树的角度来进行分析。设在搜索树上，节点 i 的子节点集合中，有 t 个点 $s_1, s_2, \cdots, s_t$ 满足割点判定法则 $dfn[i] \le low[s_k]$。于是，删除 i 关联的所有边后，无向图分成至多 $t+2$ 个连通块，每个连通块的节点构成情况为：

1. 节点 i 自身单独构成一个连通块；
2. 有 t 个连通块，分别由搜索树上以 $s_k (1 \le k \le t)$ 为根的子树中的节点构成；
3. 还可能有一个连通块，由除了上述节点之外的所有点构成。



无向连通图和它的搜索树（粗边）
在搜索树上 i 有 3 个子节点
其中 s_1, s_2 的 low 值 $\ge dfn[i]$



删除节点 i 关联的所有边，无向图分成 4 个连通块：节点 i 自身、以 s_1 或 s_2 为根的子树中的节点、图中剩余部分

不妨在 Tarjan 算法执行深度优先遍历的过程中，顺便求出搜索树每棵“子树”的大小。设 $size[x]$ 表示以 x 为根的子树大小。综上所述，删掉一个割点 i 之后，不连通的有序对数量为：

$$size[s_1]*(n-size[s_1])+size[s_2]*(n-size[s_2])+\cdots+size[s_t]*(n-size[s_t])$$

$$+1*(n-1)+\left(n-1-\sum_{k=1}^{t} size[s_k]\right)*\left(1+\sum_{k=1}^{t} size[s_k]\right)$$

```
const int N = 100010, M = 500010;
```

```
int head[N], ver[M * 2], Next[M * 2];
int dfn[N], low[N], Size[N];
long long ans[N];
bool cut[N];
int n, m, tot, num;
void add(int x, int y) {
    ver[++tot] = y, Next[tot] = head[x], head[x] = tot;
}

void tarjan(int x) {
    dfn[x] = low[x] = ++num; Size[x] = 1;
    int flag = 0, sum = 0;
    for (int i = head[x]; i; i = Next[i]) {
        int y = ver[i];
        if (!dfn[y]) {
            tarjan(y);
            Size[x] += Size[y];
            low[x] = min(low[x], low[y]);
            if (low[y] >= dfn[x]) {
                flag++;
                ans[x] += (long long)Size[y] * (n - Size[y]);
                sum += Size[y];
                if (x != 1 || flag > 1) cut[x] = true;
            }
        }
        else low[x] = min(low[x], dfn[y]);
    }
    if (cut[x])
        ans[x] += (long long)(n - sum - 1)*(sum + 1) + (n - 1);
    else
        ans[x] = 2 * (n - 1);
}
int main() {
    cin >> n >> m; tot = 1;
    for (int i = 1; i <= m; i++) {
        int x, y;
        scanf("%d%d", &x, &y);
        if (x == y) continue;
```

```
        add(x, y), add(y, x);
    }
    tarjan(1);
    for (int i = 1; i <= n; i++)
        printf("%lld\n", ans[i]);
}
```

无向图的双连通分量

若一张无向连通图不存在割点，则称它为“点双连通图”。若一张无向连通图不存在桥，则称它为“边双连通图”。

无向图的极大点双连通子图被称为“点双连通分量”，简记为“v-DCC”[①]。无向连通图的极大边双连通子图被称为“边双连通分量”，简记为“e-DCC”[②]。二者统称为“双连通分量”，简记为“DCC”。

在上面的定义中，我们称一个双连通子图 $G'=(V',E')$“极大”（其中 $V'\subseteq V$, $E'\subseteq E$），是指不存在包含 G' 的更大的子图 $G''=(V'',E'')$，满足 $V'\subseteq V''\subseteq V$，$E'\subseteq E''\subseteq E$ 并且 G'' 也是双连通子图。

定理

一张无向连通图是“点双连通图”，当且仅当满足下列两个条件之一。

1. 图的顶点数不超过 2。

2. 图中任意两点都同时包含在至少一个简单环中。其中“简单环”指的是不自交的环，也就是我们通常画出的环。

一张无向连通图是“边双连通图”，当且仅当任意一条边都包含在至少一个简单环中。

证明：

该定理给出了无向连通图是“点双连通图”或“边双连通图”的充要条件。我们以“点双连通图”为例进行证明，对于“边双连通图”的证明，请读者自己完成。

对于顶点数不超过 2 的情况，定理显然成立，下面假设图中顶点数不小于 3。

先证充分性。若任意两点 x,y 都同时包含在至少一个简单环中，则 x,y 之间至少有两条不相交的路径。无论从图中删除哪个节点，x,y 均能通过两条路径之一相连。故图中不存在割点，是点双连通图。

再证必要性。反证法，假设一张无向连通图是“点双连通图”，并且存在两点 x,y，

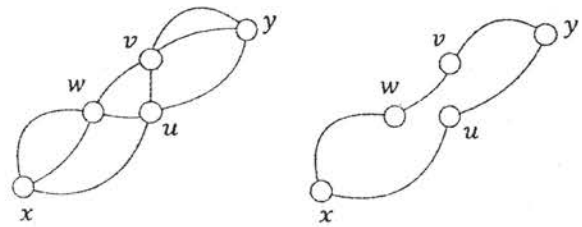
① v-DCC 是 vertex double connected component 的缩写。

② e-DCC 是 edge double connected component 的缩写。

它们不同时处于任何一个简单环中。

如果 x,y 之间仅存在1条简单路径，那么路径上至少有一个割点，与“点双连通”矛盾。

如果 x,y 之间存在2条或2条以上的简单路径，那么容易发现，任意两条都至少有一个除 x,y 之外的交点；进一步可推导出，x,y 之间的所有路径必定同时交于除 x,y 之外的某一点 p（不然就会存在两条没有交点的路径，形成一个简单环，如下图所示）。



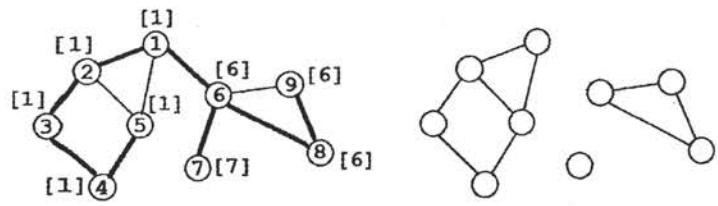
根据定义，p 是一个割点，与“点双连通”矛盾。故假设不成立。

证毕。

边双连通分量(e-DCC)的求法

边双连通分量的计算非常容易。只需求出无向图中所有的桥，把桥都删除后，无向图会分成若干个连通块，每一个连通块就是一个“边双连通分量”。

下面的无向图共有2条“桥”边，3个“边双连通分量”：



在具体的程序实现中，一般先用 Tarjan 算法标记出所有的桥边。然后，再对整个无向图执行一次深度优先遍历（遍历的过程中不访问桥边），划分出每个连通块。下面的代码在 Tarjan 求桥的参考程序基础上，计算出数组 c，$c[x]$ 表示节点 x 所属的“边双连通分量”的编号。

```
int c[SIZE], dcc;
void dfs(int x) {
    c[x] = dcc;
    for (int i = head[x]; i; i = Next[i]) {
        int y = ver[i];
```



```
            if (c[y] || bridge[i]) continue;
            dfs(y);
        }
    }

    // 以下代码片段加在 main 函数中
    for (int i = 1; i <= n; i++)
        if (!c[i]) {
            ++dcc;
            dfs(i);
        }
    printf("There are %d e-DCCs.\n", dcc);
    for (int i = 1; i <= n; i++)
        printf("%d belongs to DCC %d.\n", i, c[i]);
```

e-DCC 的缩点

把每个 e-DCC 看作一个节点，把桥边 (x,y) 看作连接编号为 $c[x]$ 和 $c[y]$ 的 e-DCC 对应节点的无向边，会产生一棵树（若原来的无向图不连通，则产生森林）。这种把 e-DCC 收缩为一个节点的方法就称为“缩点”。下面的代码在 Tarjan 求桥、求 e-DCC 的参考程序基础上，把 e-DCC 缩点，构成一棵新的树（或森林），存储在另一个邻接表中。

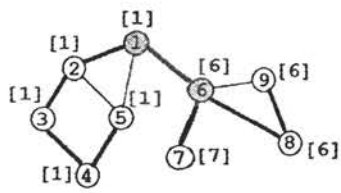
```
    int hc[SIZE], vc[SIZE * 2], nc[SIZE * 2], tc;
    void add_c(int x, int y) {
        vc[++tc] = y, nc[tc] = hc[x], hc[x] = tc;
    }

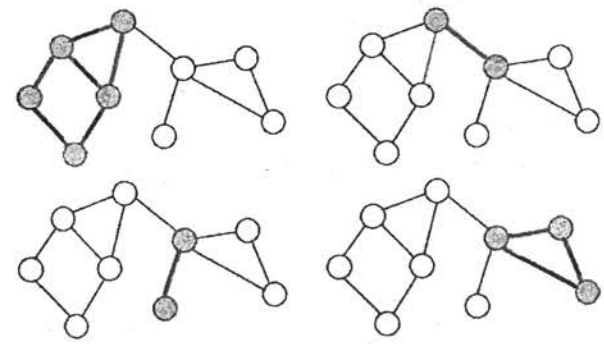
    // 以下代码片段加在 main 函数中
    tc = 1;
    for (int i = 2; i <= tot; i++) {
        int x = ver[i ^ 1], y = ver[i];
        if (c[x] == c[y]) continue;
        add_c(c[x], c[y]);
    }
    printf("缩点之后的森林，点数%d，边数%d\n", dcc, tc / 2);
    for (int i = 2; i < tc; i += 2)
        printf("%d %d\n", vc[i ^ 1], vc[i]);
```

点双连通分量(v-DCC)的求法

“点双连通分量”是一个极其容易误解的概念。它与前面的例题“BLO”中“删除割点后图中剩余的连通块”是不一样的，请读者格外留意。

若某个节点为孤立点，则它自己单独构成一个 v-DCC。除了孤立点之外，点双连通分量的大小至少为 2。根据 v-DCC 定义中的“极大”性，虽然桥不属于任何 e-DCC，但是割点可能属于多个 v-DCC。还是来看本节一直使用的例子，下面的无向图共有 2 个割点，4 个“点双连通分量”。





为了求出“点双连通分量”，需要在 Tarjan 算法的过程中维护一个栈，并按照如下方法维护栈中的元素。

1. 当一个节点第一次被访问时，把该节点入栈。

2. 当割点判定法则中的条件 $dfn[x] \le low[y]$ 成立时，无论 x 是否为根，都要：

(1) 从栈顶不断弹出节点，直至节点 y 被弹出；

(2) 刚才弹出的所有节点与节点 x 一起构成一个 v-DCC。

下面的程序在求出割点的同时，计算出 vector 数组 dcc，$dcc[i]$ 保存编号为 i 的 v-DCC 中的所有节点。

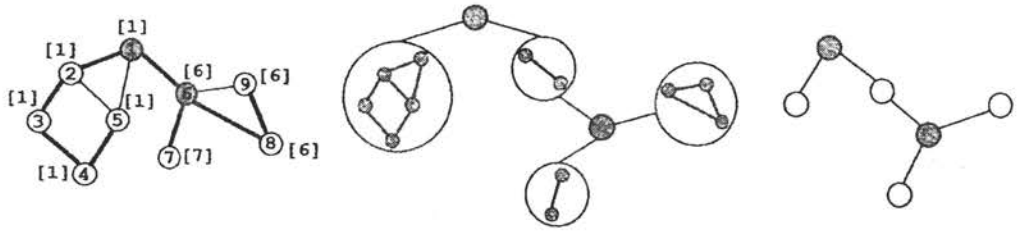
```
void tarjan(int x) {
    dfn[x] = low[x] = ++num;
    stack[++top] = x;
    if (x == root && head[x] == 0) { // 孤立点
        dcc[++cnt].push_back(x);
        return;
    }
    int flag = 0;
    for (int i = head[x]; i; i = Next[i]) {
        int y = ver[i];
        if (!dfn[y]) {
```

```
            tarjan(y);
            low[x] = min(low[x], low[y]);
            if (low[y] >= dfn[x]) {
                flag++;
                if (x != root || flag > 1) cut[x] = true;
                cnt++;
                int z;
                do {
                    z = stack[top--];
                    dcc[cnt].push_back(z);
                } while (z != y);
                dcc[cnt].push_back(x);
            }
        }
        else low[x] = min(low[x], dfn[y]);
    }
}

// 以下代码片段加在求割点程序的 main 函数中
for (int i = 1; i <= cnt; i++) {
    printf("v-DCC #%d:", i);
    for (int j = 0; j < dcc[i].size(); j++)
        printf(" %d", dcc[i][j]);
    puts("");
}
```

v-DCC 的缩点

v-DCC 的缩点比 e-DCC 要复杂一些——因为一个割点可能属于多个 v-DCC。设图中共有 p 个割点和 t 个 v-DCC。我们建立一张包含 $p+t$ 个节点的新图，把每个 v-DCC 和每个割点都作为新图中的节点，并在每个割点与包含它的所有 v-DCC 之间连边。容易发现，这张新图其实是一棵树（或森林），如下图所示。



下面的代码在 Tarjan 求割点和 v-DCC 的参考程序的 main 函数基础上，对 v-DCC 缩点，构成一棵新的树（或森林），存储在另一个邻接表中。

```
// 给每个割点一个新的编号(编号从 cnt+1 开始)
num = cnt;
for (int i = 1; i <= n; i++)
    if (cut[i]) new_id[i] = ++num;
// 建新图，从每个 v-DCC 到它包含的所有割点连边
tc = 1;
for (int i = 1; i <= cnt; i++)
    for (int j = 0; j < dcc[i].size(); j++) {
        int x = dcc[i][j];
        if (cut[x]) {
            add_c(i, new_id[x]);
            add_c(new_id[x], i);
        }
        else c[x] = i; // 除割点外，其他点仅属于 1 个 v-DCC
    }
printf("缩点之后的森林，点数 %d，边数 %d\n", num, tc / 2);
printf("编号 1~%d 的为原图的 v-DCC，编号 >%d 的为原图割点\n",
        cnt, cnt);
for (int i = 2; i < tc; i += 2)
    printf("%d %d\n", vc[i ^ 1], vc[i]);
```

读者将在 0x6B 节的习题中运用 e-DCC 与 v-DCC 的缩点，结合倍增 LCA 算法，多次回答无向图上两点之间的必经边、必经点的询问。

例题 **Network** AcWing364

给定一张 N 个点 M 条边的无向连通图，然后执行 Q 次操作，每次向图中添加一条边，并且询问当前无向图中“桥”的数量。$N \le 10^5, M \le 2 * 10^5, Q \le 1000$。

先用 Tarjan 算法求出无向图中所有的“边双连通分量”(e-DCC)，并对每个 e-DCC 缩点，得到一棵树，树上的每个节点都是原图中的一个 e-DCC。设 $c[x]$ 表示节点 x 所在的 e-DCC 的编号。最初，“桥”的总数就是缩点之后树的边数。

依次考虑每个添加边 (x,y) 的操作。若 x,y 属于同一个 e-DCC ($c[x]=c[y]$)，则加入该边后，无向图中“桥”的数量不变。

若 x,y 属于不同的 e-DCC，则在缩点后得到的树上，$c[x]$ 与 $c[y]$ 之间的路径上的每条边都不再是“桥”，因为它们都处在一个环内。我们可以求出 $p=$

$\mathrm{LCA}(c[x],c[y])$，从 $c[x]$ 不断走向父节点，直至点 p，把经过的边都标记成“不再是桥”。从 $c[y]$ 开始也执行一样的标记过程。若有 cnt 条边新获得了标记，则把图中“桥”的总数减掉 cnt 即可。

上述算法的时间复杂度为 $O(M+Q*N)$，但本题数据比较宽松，可以得到满分。我们也可以用并查集进一步优化，当树上每条边获得“不再是桥”的标记后，就把这条边的子节点所在的集合合并到父节点所在的集合中。这样一来，从 $c[x]$ 走向 $\mathrm{LCA}(c[x],c[y])$ 时，每一步可以直接走到并查集的 get 函数返回的节点上，本质上是利用并查集对“不再是桥”的边进行了路径压缩。时间复杂度可以降低到 $O(M+Q\log N)$。

例题 **Knights of the Round Table** AcWing365

国王要在圆桌上召开骑士会议，但是有若干对骑士之间互相憎恨。出于各种各样奇怪的原因，每次开会都必须有如下要求。

1. 相互憎恨的两个骑士不能坐在相邻的两个位置。

2. 为了让投票表决议题时都能有结果（不平票），出席会议的骑士数必须是奇数。

如果有某个骑士无法出席任何会议，则国王会为了世界和平把他踢出骑士团。

现在给定骑士总数 $n(n\le 1000)$，以及 $m(m\le 10^6)$ 对相互憎恨的关系，求至少要踢掉多少个骑士。

我们可以建出原图的“补图”—— n 个节点代表 n 个骑士，若两名骑士**没有**憎恨关系，则在二者之间连一条无向边。

根据题意，若干名骑士可以召开圆桌会议的条件是：他们对应的节点组成一个长度为奇数的简单环（简称奇环）。因此，本题就是求有多少个点**不被任何奇环包含**，这些点就是应该被踢掉的骑士。

引理 1：

若两个骑士属于两个不同的“点双连通分量”，则他们不可能一起出席会议。

证明：

反证法。假设 x,y 分别属于编号为 $c[x]$ 和 $c[y]$ 的 v-DCC，其中 $c[x]\neq c[y]$，并且 x,y 处于某一个奇环中（能出席会议）。因为一个 v-DCC 不包含割点，所以无论删掉图中哪个点，x 与编号为 $c[x]$ 的 v-DCC 中的其他点仍然连通，y 与编号为 $c[y]$ 的 v-DCC 中的其他点也仍然连通。而 x,y 处于一个环中，环不存在割点，无论删除哪个点，x,y 之间也是连通的。

综上所述，无论删除哪个点，编号为 $c[x]$ 和 $c[y]$ 的两个 v-DCC 中的点都仍然是连通的。换言之，这两个 v-DCC 合在一起构成的图中不存在割点，也是 v-DCC。这与 v-

DCC 的“极大性”矛盾。故假设不成立，原命题成立。

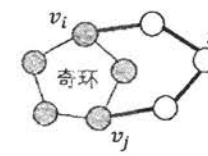
证毕。

引理 2：

若某个“点双连通分量”中存在奇环，则这个“点双连通分量”中的所有点都被至少一个奇环包含。

证明：

设 v-DCC 中的奇环依次由节点 $v_1, v_2, \cdots, v_t$ 构成，t 为奇数。容易发现，对于奇环以外的任意节点 x，奇环上必然存在两点 v_i, v_j，满足 x, v_i, v_j 三个点都在某个简单环上。否则我们就有办法删除环上的一个点，让 x 与奇环不连通。



进一步，我们一定可以构造出如右图所示的包含 x, v_i, v_j 三个点的简单环。在这个环上，无论 x 到 v_i 的路径长度加上 x 到 v_j 的路径长度是奇数还是偶数，因为奇环上从 v_i 到 v_j 有两条长度的奇偶性不同的路径，所以总能拼出包含 x 的奇环。

证毕。

综上所述，用 Tarjan 算法求出“补图”中所有的 v-DCC，判定每个 v-DCC 中是否存在奇环即可。若存在奇环，则该 v-DCC 中的所有骑士都可以参加会议（都在某个奇环中）。

奇环可以用深度优先遍历的染色法进行判定。一张无向图没有奇环，等价于它是一张二分图。我们将在 0x68 节讲解二分图（奇环）的判定方法，不熟悉的读者可提前阅读 0x68 节开头部分的内容。

欧拉路问题

在本节最后，我们抛开 Tarjan 算法，探讨一个关于无向图连通性的小问题——欧拉路问题，俗称“一笔画”问题。

给定一张无向图，若存在一条从节点 S 到节点 T 的路径，恰好不重不漏地经过每条边一次（可以重复经过图中的节点），则称该路径为 S 到 T 的**欧拉路**。

特别地，若存在一条从节点 S 出发的路径，恰好不重不漏地经过每条边一次（可以重复经过图中的节点），最终回到起点 S，则称该路径为**欧拉回路**。存在欧拉回路的无向图被称为欧拉图。

欧拉图的判定

一张无向图为欧拉图，当且仅当无向图连通，并且每个点的度数都是偶数。

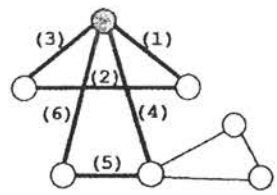
欧拉路的存在性判定

一张无向图中存在欧拉路，当且仅当无向图连通，并且图中恰好有两个节点的度数为奇数，其他节点的度数都是偶数。这两个度数为奇数的节点就是欧拉路的起点 S 和终点 T。

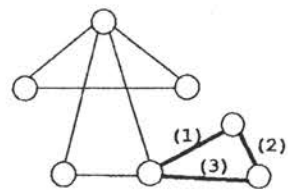
在判定无向图存在欧拉回路后，我们可以借助深度优先遍历和栈，求出欧拉回路的一种具体方案。流程如下：

```
void dfs(int x)
    对于从 x 出发的每条边(x,y)
        如果该边没有被访问过
            把这条边标记为已访问
            dfs(y)
            把 y 入栈
在主函数中
    dfs(1)
    倒序输出栈中所有节点
```

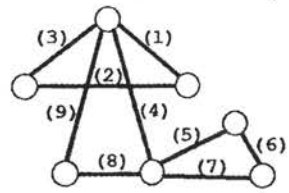
欧拉图每个节点度数为偶数说明：只要到达一个节点，就必定有一条尚未走过的边可以离开该点。故在上面的伪代码中，调用 $dfs(1)$，不断递归，每次都走到“从 x 出发的第一条未访问的边”的另一端点 y，最终一定能回到节点 1，产生一条回路。如下图所示。



当然，这条回路不能保证经过了图中的每条边。所以 dfs 函数会继续考虑从 x 出发的其他未访问的边，找到第二条回路。如下图所示。



上述伪代码实际上找出了若干条回路。我们需要把这些回路按适当的方法拼接在一起，形成整张图的欧拉回路。一个很显然的拼接方法就是：把第二条回路嵌入第一条回路中间。如下图所示。



上面伪代码中维护的栈，恰好替我们完成了这个拼接工作。最后，把栈里的所有节点倒序输出，就得到了一条欧拉回路。

值得提醒的是，上述算法的时间复杂度为 $O(NM)$。因为一个点会被重复遍历多次，每次都会扫描与它相连的所有的边——虽然我们明明知道大部分边已经访问过了。

假设我们采用邻接表存储无向图，我们可以在访问一条边 (x,y) 后，及时修改表头 $head[x]$，令它指向下一条边。这样我们每次只需取出 $head[x]$，就自然跳过了所有已经访问过的边。

另外，因为欧拉回路 DFS 的递归层数是 $O(M)$ 级别，容易造成系统栈溢出，我们可以用另一个栈，模拟机器的递归过程，把代码转化为非递归实现。

加入上述优化后的完整程序如下，其时间复杂度为 $O(N+M)$。

```
int head[100010], ver[1000010], Next[1000010], tot; // 邻接表
int stack[1000010], ans[1000010]; // 模拟系统栈，答案栈
bool vis[1000010];
int n, m, top, t;

void add(int x, int y) {
    ver[++tot] = y, Next[tot] = head[x], head[x] = tot;
}

void euler() {
    stack[++top] = 1;
    while (top > 0) {
        int x = stack[top], i = head[x];
        while (i && vis[i]) i = Next[i];  // 找到一条尚未访问的边
        if (i) {  // 沿着这条边模拟递归过程，标记该边，并更新表头
            stack[++top] = ver[i];
            vis[i] = vis[i ^ 1] = true;
            head[x] = Next[i];
```

```
        }
        else {  // x 相连的所有边均已访问，模拟回溯过程，并记在答案栈中
            top--;
            ans[++t] = x;
        }
    }
}
int main() {
    cin >> n >> m;
    tot = 1;
    for (int i = 1; i <= m; i++) {
        int x, y;  scanf("%d%d", &x, &y);
        add(x, y), add(y, x);
    }
    euler();
    for (int i = t; i; i--) printf("%d\n", ans[i]);
}
```

例题 **Watchcow** AcWing366

给定 N 个点 M 条边的无向图 $(1 \le N \le 10^4,\ 1 \le M \le 5 * 10^4)$，求一条路径，从节点 1 出发，最后回到节点 1，并且满足每条边恰好被沿着正、反两个方向分别经过一次。若有多种方案，输出任意一种即可。

我们只需在求欧拉回路的参考程序中，去掉对每条边的 *vis* 标记，即可得到本题的解答。这是因为，按照一般的存储方式，每条无向边在邻接表中会以正、反两个方向分别保存一次。若没有 *vis* 标记，则根据表头数组 *head* 的更新方法，每条无向边会被正、反两个方向各经过一次，恰好符合题目要求。

0x67 Tarjan 算法与有向图连通性

给定有向图 $G=(V,E)$，若存在 $r\in V$，满足从 r 出发能够到达 V 中所有的点，则称 G 是一个“流图”(Flow Graph)，记为 (G,r)，其中 r 称为流图的源点。

与无向图的深度优先遍历类似，我们也可以定义“流图”的搜索树和时间戳的概念。

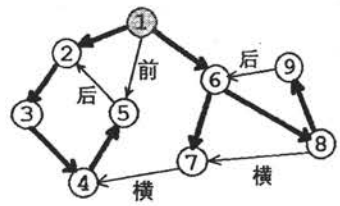
在一个流图 (G,r) 上从 r 出发进行深度优先遍历，每个点只访问一次。所有发生递归的边 (x,y)（换言之，从 x 到 y 是对 y 的第一次访问）构成一棵以 r 为根的树，我们把它称为**流图 (G,r) 的搜索树**。

同时，在深度优先遍历的过程中，按照每个节点第一次被访问的时间顺序，依次给予流图中 N 个节点 $1\sim N$ 的整数标记，该标记被称为**时间戳**，记为 $dfn[x]$。

流图中的每条有向边 (x,y) 必然是以下四种之一。

1. 树枝边，指搜索树中的边，即 x 是 y 的父节点。
2. 前向边，指搜索树中 x 是 y 的祖先节点。
3. 后向边，指搜索树中 y 是 x 的祖先节点。
4. 横叉边，指除了以上三种情况之外的边，它一定满足 $dfn[y]<dfn[x]$。

下图画出了一个“流图”以及它的搜索树、时间戳、边的分类。圆圈中的数字是时间戳。粗边是树枝边，构成一棵搜索树。前向边、后向边与横叉边用第一个汉字标注。



有向图的强连通分量

给定一张有向图。若对于图中任意两个节点 x,y，既存在从 x 到 y 的路径，也存在从 y 到 x 的路径，则称该有向图是“强连通图”。

有向图的极大强连通子图被称为“强连通分量”，简记为SCC[①]。此处“极大”的含义与双连通分量“极大”的含义类似。

Tarjan 算法基于有向图的深度优先遍历，能够在线性时间内求出一张有向图的各个强连通分量。

一个“环”一定是强连通图。如果既存在从 x 到 y 的路径，也存在从 y 到 x 的路径，那么 x, y 显然在一个环中。因此，Tarjan 算法的基本思路就是对于每个点，尽量找到与它一起能构成环的所有节点。

容易发现，“前向边” (x, y) 没有什么用处，因为搜索树上本来就存在从 x 到 y 的路径。“后向边” (x, y) 非常有用，因为它可以和搜索树上从 y 到 x 的路径一起构成环。“横叉边” (x, y) 视情况而定，如果从 y 出发能找到一条路径回到 x 的祖先节点，那么 (x, y) 就是有用的。

为了找到通过“后向边”和“横叉边”构成的环，Tarjan 算法在深度优先遍历的同时维护了一个栈。当访问到节点 x 时，栈中需要保存以下两类节点。

1. 搜索树上 x 的祖先节点，记为集合 $anc(x)$。

设 $y \in anc(x)$。若存在后向边 (x, y)，则 (x, y) 与 y 到 x 的路径一起形成环。

2. 已经访问过，并且存在一条路径到达 $anc(x)$ 的节点。

设 z 是一个这样的点，从 z 出发存在一条路径到达 $y \in anc(x)$。若存在横叉边 (x, z)，则 (x, z)、z 到 y 的路径、y 到 x 的路径形成一个环。

综上所述，栈中的节点就是能与从 x 出发的“后向边”和“横叉边”形成环的节点。进而可以引入“追溯值”的概念。

追溯值

设 subtree(x) 表示流图的搜索树中以 x 为根的子树。x 的追溯值 $low[x]$ 定义为满足以下条件的节点的最小时间戳。

1. 该点在栈中。
2. 存在一条从 subtree(x) 出发的有向边，以该点为终点。

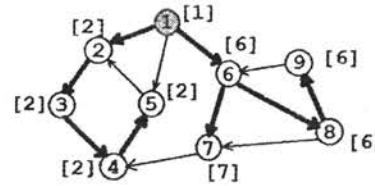
根据定义，Tarjan 算法按照以下步骤计算“追溯值”。

1. 当节点 x 第一次被访问时，把 x 入栈，初始化 $low[x] = dfn[x]$。
2. 扫描从 x 出发的每条边 (x, y)。
   (1) 若 y 没被访问过，则说明 (x, y) 是树枝边，递归访问 y，从 y 回溯之后，令 $low[x] = \min(low[x], low[y])$。
   (2) 若 y 被访问过并且 y 在栈中，则令 $low[x] = \min(low[x], dfn[y])$。
3. 从 x 回溯之前，判断是否有 $low[x] = dfn[x]$。若成立，则不断从栈中弹出节

① SCC 是 Strongly Connected Component 的缩写。

点，直至 x 出栈。

下图中的中括号[]里的数值标注了每个节点的“追溯值” low。读者可以尝试在图中模拟 low 的计算过程。



强连通分量判定法则

在追溯值的计算过程中，若从 x 回溯前，有 $low[x] = dfn[x]$ 成立，则栈中从 x 到栈顶的所有节点构成一个强连通分量。

我们不再详细证明。大致来说，在计算追溯值的第 3 步，如果 $low[x] = dfn[x]$，那么说明 subtree(x) 中的节点不能与栈中其他节点一起构成环。另外，因为横叉边的终点时间戳必定小于起点时间戳，所以 subtree(x) 中的节点也不可能直接到达尚未访问的节点（时间戳更大）。综上所述，栈中从 x 到栈顶的所有节点不能与其他节点一起构成环。

又因为我们及时进行了判定和出栈操作，所以从 x 到栈顶的所有节点独立构成一个强连通分量。

下面的程序实现了 Tarjan 算法，求出数组 c，其中 $c[x]$ 表示 x 所在的强连通分量的编号。另外，它还求出了 vector 数组 scc，$scc[i]$ 记录了编号为 i 的强连通分量中的所有节点。整张图共有 cnt 个强连通分量。

```
const int N = 100010, M = 1000010;
int ver[M], Next[M], head[N], dfn[N], low[N];
int stack[N], ins[N], c[N];
vector<int> scc[N];
int n, m, tot, num, top, cnt;

void add(int x, int y) {
    ver[++tot] = y, Next[tot] = head[x], head[x] = tot;
}

void tarjan(int x) {
    dfn[x] = low[x] = ++num;
    stack[++top] = x, ins[x] = 1;
    for (int i = head[x]; i; i = Next[i])
```

```
        if (!dfn[ver[i]]) {
            tarjan(ver[i]);
            low[x] = min(low[x], low[ver[i]]);
        }
        else if (ins[ver[i]])
            low[x] = min(low[x], dfn[ver[i]]);
    if (dfn[x] == low[x]) {
        cnt++; int y;
        do {
            y = stack[top--], ins[y] = 0;
            c[y] = cnt, scc[cnt].push_back(y);
        } while (x != y);
    }
}
int main() {
    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        int x, y;
        scanf("%d%d", &x, &y);
        add(x, y);
    }
    for (int i = 1; i <= n; i++)
        if (!dfn[i]) tarjan(i);
}
```

与无向图 e-DCC 的“缩点”类似，我们也可以把每个 SCC 缩成一个点。对原图中的每条有向边 (x,y)，若 $c[x] \neq c[y]$，则在编号为 $c[x]$ 与编号为 $c[y]$ 的 SCC 之间连边。最后，我们会得到一张**有向无环图**。下面的程序对 SCC 执行缩点过程，并把新得到的有向无环图保存在另一个邻接表中。

```
void add_c(int x, int y) {
    vc[++tc] = y, nc[tc] = hc[x], hc[x] = tc;
}
// 在 main 函数中
for (int x = 1; x <= n; x++)
    for (int i = head[x]; i; i = Next[i]) {
        int y = ver[i];
        if (c[x] == c[y]) continue;
```

```
        add_c(c[x], c[y]);
    }
```

例题 Network of Schools AcWing367

一些学校连接在一个计算机网络上，学校之间存在软件支援协议，每个学校都有它应支援的学校名单（学校 A 支援学校 B，并不表示学校 B 一定支援学校 A）。当某校获得一个新软件时，无论是直接获得还是通过网络获得，该校都应立即将这个软件通过网络传送给它应支援的学校。因此，一个新软件若想让所有学校都能使用，只需将其提供给一些学校即可。

1. 最少需要将一个新软件直接提供给多少个学校，才能使软件能够通过网络被传送到所有学校？

2. 最少需要添加几条新的支援关系，使得将一个新软件提供给任何一个学校，其他所有学校就都可以通过网络获得该软件？

把学校看作节点，若学校 A 能支援学校 B，则从 A 到 B 连一条有向边，得到一张有向图。在有向图的一个强连通分量内，任意两个点都是互相可达的。因此，只要其中任何一个学校获得新软件，该强连通分量内的其他所有学校都可以通过网络获得这个新软件。

可以用 Tarjan 算法求出所有强连通分量，并执行“缩点”过程，得到一张有向无环图。首先，“零入度点”无法被其他学校支援。其次，若同时向所有“零入度点”提供新软件，则新软件从这些点出发沿着网络显然能遍历到整个有向无环图。综上所述，第一问的答案就是有向无环图中“零入度点”的个数。

第二问就是问最少添加几条有向边，可以把一张任意有向图变成强连通图。设缩点后的有向无环图中有 p 个“零入度点”，q 个“零出度点”，那么答案就是 $\max(p,q)$。请读者思考该结论的正确性。特别地，如果整张图本身就是一个强连通图（缩点后仅剩一个点），那么答案为 0。

例题 银河 AcWing368

银河中的恒星浩如烟海，但是我们只关注那些最亮的恒星。我们用一个正整数来表示恒星的亮度，数值越大则恒星就越亮，恒星的亮度最暗是 1。现在对于 N 颗我们关注的恒星，有 M 对亮度之间的相对关系已经判明。你的任务就是求出这 N 颗恒星的亮度值总和至少有多大。

输入数据的第一行给出两个整数 N 和 M。$N,M \leq 10^5$。之后 M 行，每行三个整数 T,A,B，表示一对恒星 (A,B) 之间的亮度关系。恒星的编号从 1 开始。

1. 如果 $T=1$，说明 A 和 B 亮度相等。

2. 如果 $T=2$，说明 A 的亮度小于 B 的亮度。
3. 如果 $T=3$，说明 A 的亮度不小于 B 的亮度。
4. 如果 $T=4$，说明 A 的亮度大于 B 的亮度。
5. 如果 $T=5$，说明 A 的亮度不大于 B 的亮度。

输出一个整数表示答案。若无解，则输出 -1。

设 $d[A]$ 表示 A 的亮度 $(1 \le A \le N)$。把五种关系的形式进行统一。

1. 关系 $d[A]=d[B]$ 可以转化为 $d[A]-d[B] \ge 0$ 并且 $d[B]-d[A] \ge 0$。
2. 关系 $d[A]<d[B]$ 可以转化成 $d[B]-d[A] \ge 1$。
3. 关系 $d[A] \ge d[B]$ 可以转化成 $d[A]-d[B] \ge 0$。
4. 关系 $d[A]>d[B]$ 可以转化成 $d[A]-d[B] \ge 1$。
5. 关系 $d[A] \le d[B]$ 可以转化成 $d[B]-d[A] \ge 0$。

另外，题目明确指出恒星的亮度最暗是 1，可以设 $d[0]=0$，然后把这个条件写作 $\forall A,\ d[A]-d[0] \ge 1$。

根据 0x65 节“差分约束”的知识，对于每个形如 $d[B]-d[A] \ge k$ 的不等式，从节点 A 到节点 B 连长度为 k 的有向边，然后从 0 号节点出发，求单源最长路。若图中有正环，则无解。否则，从 0 到每个节点的最长路的长度就是对应恒星的最小合法亮度。

然而，本题数据范围达到了 10^5 级别，并且数据较强，直接用 SPFA 算法求最长路并判断是否存在正环会超时。我们需要一个效率更高的算法。

仔细观察，我们根据差分约束系统建立的有向图中，边权只有 0 和 1 两种。如果图中存在一个环，那么环上的边长度必须都是 0，不然一定无解。因此，我们可以用 Tarjan 算法求出有向图中所有的强连通分量，只要强连通分量内部存在长度为 1 的边，就直接判定无解。

如果有解，那么每个强连通分量内部各个恒星的亮度是相等的，可以执行“缩点”，得到一张边权为 0 或 1 的有向无环图。从节点 0 所在的 SCC 出发，按照拓扑序执行动态规划，即可在 $O(N+M)$ 的时间内求出到达每个 SCC 的最长路，即该 SCC 内所有恒星的最小亮度。

有向图的必经点与必经边

给定一张有向图，起点为 S，终点为 T。若从 S 到 T 的每条路径都经过一个点 x，则称点 x 是有向图中从 S 到 T 的必经点。

若从 S 到 T 的每条路径都经过一条边 (x,y)，则称这条边是有向图中从 S 到 T 的必经边或“桥”。

有向图的必经点与必经边是一个较难的问题。因为环上的点也可能是必经点，所以不能简单地把强连通分量缩点后按照有向无环图来处理。Lenguar-Tarjan 算法通过计算支配树(Dominator-Tree)，能够在 $O(N \log N)$ 的时间内求出从有向图的指定起点出发，走到每个点的必经点集。这超出了我们的讨论范围，感兴趣的读者可以阅读本书作者在全国信息学奥林匹克竞赛冬令营上的讲义《图连通性若干拓展问题探讨》，本书配套资源提供了该讲义的最新修订版本。

不过，值得一提的是，我们有很简单的方法计算**有向无环图**的必经点与必经边。

1. 在原图中按照拓扑序进行动态规划，求出起点 S 到图中每个点 x 的路径条数 $fs[x]$。

2. 在反图上再次按照拓扑序进行动态规划，求出每个点 x 到终点 T 的路径条数 $ft[x]$。

显然，$fs[T]$ 表示从 S 到 T 的路径总条数。根据乘法原理：

1. 对于一条有向边 (x,y)，若 $fs[x]*ft[y]=fs[T]$，则 (x,y) 是有向无环图从 S 到 T 的必经边；

2. 对于一个点 x，若 $fs[x]*ft[x]=fs[T]$，则 x 是有向无环图从 S 到 T 的必经点。

路径条数是一个指数级别的整数，通常超过了 32 位或 64 位整数的表示范围。受 Hash 思想的启发，我们可以把路径条数对一个大质数取模后再保存到 fs 与 ft 数组中。这样带来的后果是有较小的概率会产生误判。为了保险起见，若题目时限宽松，我们可以多选取几个质数，分别作为模数进行计算。

例题 **PKU ACM Team's Excursion** AcWing369

给定一张 N 个点 M 条边的有向无环图，每条边都有一个长度。给定一个起点 S 和一个终点 T。若从 S 到 T 的每条路径都经过某条边，则称这条边是有向图的必经边或桥。

PKU ACM 队要从 S 点到 T 点。他们在路上可以搭乘两次车。每次可以从任意位置（甚至是一条边上的任意位置）上车，从任意位置下车，但连续乘坐的长度不能超过 q 米。除去这两次乘车外，剩下的路段步行。定义从 S 到 T 的路径的危险程度等于步行经过的桥上路段的长度之和。求从 S 到 T 的最小危险程度是多少。

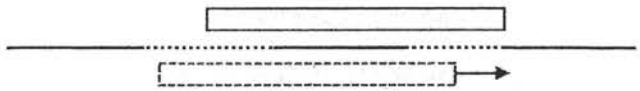
数据范围：$1 \le N,M \le 2*10^5$，$1 \le q \le 10^9$。

首先，用前面提到的计算路径条数（并取模）的方法，求出从 S 到 T 的所有“桥”。根据贪心策略，因为“桥”是从 S 到 T 的任意一条路径都包含的边，所以我们应该让路径上除了“桥”以外的部分尽量短。并且我们只关心长度，不关心“桥”以外的部分到底是哪些边。所以我们可以求出从 S 到 T 的任意一条最短路，考虑在这条最短路径

的什么地方搭车，能使危险程度最小。

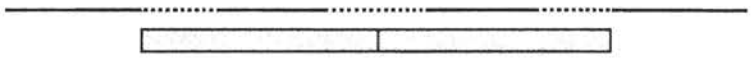
一条最短路其实就是一条“链”，我们要用两个区间覆盖链上两段长度不超过 q 的部分，使未被覆盖的桥的长度之和最小。

如果只有一个区间，那么是非常好做的。当下车位置在“边”的中间而不在“点”上时，把乘车区间向右平移到在“点”上下车，对答案没有影响，如下图所示（虚线表示桥）。因此，可以用双指针（又称尺取法）扫描一遍这条最短路，求出 $ds[i]$，表示从 S 到最短路上的第 i 个节点只搭一次车的最小危险程度。请读者自己写出详细过程。



类似地，还可以倒着扫描一遍，求出 $dt[i]$，表示从最短路上的第 i 个节点到 T，只搭一次车的最小危险程度。最后，我们枚举“切点” i，用 $ds[i]+dt[i]$ 更新答案。

此时我们还遗漏了一种情况——当两个乘车区间相连成为一个长度为 $2q$ 的区间时，它可能无法拆分为两个下（上）车位置在“点”上的乘车区间，如下图所示。这也不难处理，我们只需要用一个长度为 $2q$ 的区间再扫描一次，把答案更新即可。



2-SAT 问题

有 N 个变量，每个变量只有两种可能的取值。再给定 M 个条件，每个条件都是对两个变量的取值限制。求是否存在对 N 个变量的合法赋值，使 M 个条件均得到满足。这个问题被称为 2-SAT 问题。SAT 是英文 satisfiability 的缩写，所以 2-SAT 也可翻译为“2-可满足性”问题。

设一个变量 $A_i(1 \le i \le N)$ 的两种可能取值分别是 $A_{i,0}$ 和 $A_{i,1}$。在 2-SAT 问题中，M 个条件都可以转化为统一的形式——“若变量 A_i 赋值为 $A_{i,p}$，则变量 A_j 必须赋值为 $A_{j,q}$”，其中 $p,q \in \{0,1\}$。

2-SAT 问题的判定方法如下。

1. 建立 $2N$ 个节点的有向图，每个变量 A_i 对应 2 个节点，一般设为 i 和 $i+N$。

2. 考虑每个条件，形如“若变量 A_i 赋值为 $A_{i,p}$，则变量 A_j 必须赋值为 $A_{j,q}$”，$p,q \in \{0,1\}$。从 $i+p*N$ 到 $j+q*N$ 连一条有向边。

注意，上述条件蕴含着它的**逆否命题**“若变量 A_j 必须赋值为 $A_{j,1-q}$，则变量 A_i 必须赋值为 $A_{i,1-p}$”。如果在给出的 M 个限制条件中，原命题和逆否命题不一定成对出现，应该从 $j+(1-q)*N$ 到 $i+(1-p)*N$ 也连一条有向边。

总而言之，根据原命题和逆否命题的对称性，2-SAT 建出的有向图一定能画成“一侧是节点 $1\sim N$，另一侧是节点 $N+1\sim 2N$”。当把图中的边看作无向边时，这两侧连边的情况是对称的。

3. 用 Tarjan 算法求出有向图中所有的强连通分量。

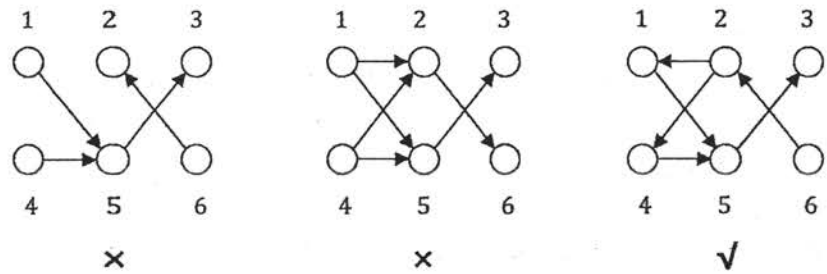
4. 若存在 $i\in[1,N]$，满足 i 和 $i+N$ 属于同一个强连通分量，则表明：若变量 A_i 赋值为 $A_{i,p}$，则变量 A_i 必须赋值为 $A_{i,1-p}$ $(\forall p\in\{0,1\})$。这显然是矛盾的，说明问题无解。若不存在这样的 i，则问题一定有解。我们后面会探讨如何给出一种合法的赋值方案。

时间复杂度为 $O(N+M)$。

我们在 0x41 节讲解了利用并查集处理二元关系的一类模型，前提是关系具有传递性，并且关系是“无向”的。无向关系可以理解为：从原命题“若 P 则 Q”能推出逆命题“若 Q 则 P”、逆否命题“若非 Q 则非 P”、否命题“若非 P 则非 Q”。在扩展了一倍域的并查集中，对于每条关系我们会添加两条无向边，其实就等价于上述 2-SAT 的四个命题对应的四条有向边。

而在 2-SAT 模型中，关系是“有向”的。有向关系就是一般的关系，从原命题“若 P 则 Q”一定能推出逆否命题“若非 Q 则非 P”，但不一定能推出逆命题“若 Q 则 P”、否命题“若非 P 则非 Q”。我们也不用强调关系的传递性，因为“若 P 则 Q”和“若 Q 则 R”两个命题自然能推出命题“若 P 则 R”。总而言之，2-SAT 模型符合二元关系的一般逻辑，能处理更多、更复杂的问题。

如下图所示，设变量数 $N=3$，1~3 三个点代表三个变量的第一种赋值，4~6 三个点代表变量的第二种赋值。前两种情况都不可能是 2-SAT 问题对应的有向图，因为命题“若 1 则 5”的逆否命题“若 2 则 4”未在图中体现。第三种情况可能是某个 2-SAT 问题对应的有向图，其中一组解是选择 1，3，5 三个点代表的赋值。



例题 **Katu Puzzle** AcWing370

有 N 个变量 $X_1\sim X_N$，每个变量的可能取值是 0 或 1。给定 M 个算式，每个算式形如 X_a op $X_b=c$，其中 a,b 是两个变量的编号，c 是数字 0 或 1，op 是 and, or, xor 三

个位运算之一。求是否存在对每个变量的合法赋值，使所有算式都成立。

数据范围：$1 \le N \le 1000, 1 \le M \le 10^6$。

如果只有 xor 运算，那么可以用扩展域的并查集来解决本题。现在有 and, or, xor 三种运算，我们先尝试把它们转化成 2-SAT 能接受的统一的形式。

设节点 $a(1 \le a \le N)$ 表示变量 X_a 赋值为 0，节点 $a+N$ 表示 X_a 赋值为 1。

1. a and $b = 0$

这表示 X_a, X_b 其中一个赋值为 1 时，另一个必须赋值为 0。

若 $X_a = 1$，则必须 $X_b = 0$。从 $a+N$ 到 b 连有向边。

若 $X_b = 1$，则必须 $X_a = 0$。从 $b+N$ 到 a 连有向边。

2. a and $b = 1$

这表示 X_a, X_b 两个变量都必须赋值为 1。该关系可由 2 条有向边描述——若赋值为 0，让它直接产生矛盾即可。

若 $X_a = 0$，则必须 $X_a = 1$。从 a 到 $a+N$ 连有向边。

若 $X_b = 0$，则必须 $X_b = 1$。从 b 到 $b+N$ 连有向边。

3. a or $b = 1$

与第 1 种情况类似，连边 $(a, b+N)$，$(b, a+N)$。

4. a or $b = 0$

与第 2 种情况类似，连边 $(a+N, a)$，$(b+N, b)$。

5. a xor $b = 0$

连边 (a, b)，(b, a)，$(a+N, b+N)$，$(b+N, a+N)$。

6. a xor $b = 1$

连边 $(a, b+N)$，$(b, a+N)$，$(a+N, b)$，$(b+N, a)$。

可以发现，在我们添加的有向边中，“原命题”和“逆否命题”是成对出现的。用 Tarjan 算法求出有向图中所有的强连通分量，检查是否存在 a 和 $a+N$ 属于同一个 SCC 即可。

例题 **Priest John's Busiest Day** AcWing371

在一个城镇流传着一个古老的传说，在 9 月 1 日那天结婚的情侣将收到上帝的祝福和保佑。因此，城镇的牧师 John 在每年的 9 月 1 日都很忙碌。今年的 9 月 1 日，有 N 对情侣准备结婚 $(1 \le N \le 1000)$，每对情侣都预先计划好了婚礼举办的时间，其中第 i 对情侣的婚礼从时刻 S_i 开始，到时刻 T_i 结束（精确到分钟）。

婚礼有一个必需的仪式：站在牧师面前聆听上帝的祝福。这个仪式要么在婚礼开始时举行，要么在结束前举行。第 i 对情侣需要 D_i 分钟完成这个仪式，即必须选择 $S_i \sim S_i + D_i$ 和 $T_i - D_i \sim T_i$ 两个时间段之一。牧师想知道他能否满足每场婚礼的要求，

即给每对情侣安排 $S_i \sim S_i + D_i$ 或 $T_i - D_i \sim T_i$，使得这些仪式的时间段不重叠。若能满足，还需要帮牧师求出任意一种具体方案。

在本题中，每场婚礼是一个变量，有“开始时举行仪式”和“结束前举行仪式”两种取值，分别记为 $X_{i,0}$ 和 $X_{i,1}$，并看作节点 i 和 $i+N$。

$O(N^2)$ 枚举每两场婚礼 i,j，若婚礼 i 的 $X_{i,p}(p \in \{0,1\})$ 时间段与婚礼 j 的 $X_{j,q}(q \in \{0,1\})$ 时间段重叠，则说明二者不能同时被选为最终赋值。转化成 2-SAT 接受的形式，应该连 $(i + p * N, j + (1-q) * N)$ 和 $(j + q * N, i + (1-p) * N)$ 两条有向边二者互为逆否命题。

用 Tarjan 算法求 SCC，检查是否存在 i 和 $i+N$ 在同一 SCC 即可。本题还需要输出方案。下面我们给出 2-SAT 合法方案的两种构造方法。

首先，在一个 SCC 中，只要确定了一个变量的赋值，该 SCC 内其他变量的赋值也就直接确定了，这启发我们考虑缩点。其次，因为互为“逆否命题”的有向边在图中成对出现，所以一个“零出度点”对面的点一定有出边（例如 456 页图中第三种情况的点 3 和点 6）。选择一个“有出边的点”会使得该边指向的点必须也被选择，而选择一个“零出度点”不会对其他任何点造成影响。

根据上述讨论，第一种构造方法的基本思想就是：**自底向上执行拓扑排序，不断尝试选择“零出度点”**。详细流程如下。

1. 把 SCC 缩点。因为一般的拓扑排序是“自顶向下”根据“入度”进行的，所以我们建立一张缩点后的“反图”。具体来说：

(1) 图上每个点都对应原图的 SCC；

(2) 原图中的边 (x,y) 转化为新图中的边 $(c[y], c[x])$，其中 $c[x] \neq c[y]$，$c[x]$ 表示节点 x 所在的 SCC 的编号；

(3) 对于原图中每个点 x，$c[x]$ 和 $c[x+N]$ 就是新图中两个对称的节点。

为了简便起见，记 $opp[c[x]] = c[x+N]$，$opp[c[x+N]] = c[x]$。

2. 在上述“反图”上统计每个点的入度，执行拓扑排序。

设 $val[k]$ 表示原图 k 号 SCC 的赋值标记，初始值为 -1。

从队头每取出一个节点 k（k 相当于原图中一个 SCC 的编号），就检查 k 的赋值标记。若 $val[k] = -1$（尚未确定赋值），就令 $val[k] = 0$，$val[opp[k]] = 1$。随后把它能到达的点的入度减 1（拓扑排序的正常过程）。

3. 拓扑排序结束之后，就得到了最终的答案。对于原图每个节点 $i \in [1,N]$：

若 $val[c[i]] = 0$，则变量 X_i 应赋值为 $X_{i,0}$；

若 $val[c[i]] = 1$，则变量 X_i 应赋值为 $X_{i,1}$。

读者在本书配套资源中可以找到第一种构造方法的参考程序。



第二种构造方法在第一种构造方法的基础上，进一步利用了本节给出的“Tarjan 求 SCC 的程序”对 SCC 编号的特殊性质，使得构造过程十分简洁。注意到 Tarjan 算法的本质是一次 DFS，它在回溯时会先取出有向图“底部”的 SCC 进行标记。故 Tarjan 算法得到的 SCC 编号本身就已经满足缩点后的有向无环图中“自底向上”的顺序——序列 $1,2,\cdots,cnt$ 就是缩点后的反图的拓扑序，无须进行拓扑排序。

因此，直接比较节点所在的 SCC 的编号大小，即可确定对应变量的赋值 $val[i]$。$c[i]$ 和 $c[n+i]$ 哪个小，就代表哪个会在拓扑序列中先被遍历到，从而决定该 SCC 的变量取值。

```
for (int i = 1; i <= 2 * n; i++)
    if (!dfn[i]) tarjan(i);
for (int i = 1; i <= n; i++)
    if (c[i] == c[n + i]) { puts("NO"); return 0; }
puts("YES");
for (int i = 1; i <= n; i++) val[i] = c[i] > c[n + i];
```

若 $val[i]=0$，则变量 X_i 应赋值为 $X_{i,0}$。若 $val[i]=1$，则变量 X_i 应赋值为 $X_{i,1}$。本书配套资源提供了该方法的完整参考程序。

0x68 二分图的匹配

如果一张无向图的 N 个节点 $(N \geq 2)$ 可以分成 A,B 两个非空集合，其中 $A \cap B = \emptyset$，并且在同一集合内的点之间都没有边相连，那么称这张无向图为一张**二分图**。A,B 分别称为二分图的左部和右部。

二分图判定

定理

一张无向图是二分图，当且仅当图中不存在奇环（长度为奇数的环）。

证明略。

根据该定理，我们可以用染色法进行二分图的判定。大致思想为：尝试用黑白两种颜色标记图中的节点，当一个节点被标记后，它的所有相邻节点应该被标记与它相反的颜色。若标记过程中产生冲突，则说明图中存在奇环。二分图染色一般基于深度优先遍历实现，时间复杂度为 $O(N+M)$，其流程如下：

```
void dfs(int x, int color)
    赋值 v[x] ← color
    对于与 x 相连的每条无向边(x,y)
        if v[y] = 0 then
            dfs(y, 3-color)
        else if v[y] = color then
            判定无向图不是二分图，算法结束
```

```
在主函数中
    for i ← 1 to N
        if v[i] = 0 then dfs(i, 1)
    判定无向图是二分图
```

例题 关押罪犯 AcWing257

S 城现有两座监狱，一共关押着 N 名罪犯，编号分别为 1~N。罪犯之间的关系自

然也极不和谐。很多罪犯之间甚至积怨已久，如果客观条件具备则随时可能爆发冲突。我们用“怨气值”（一个正整数值）来表示某两名罪犯之间的仇恨程度，怨气值越大，则这两名罪犯之间的积怨越多。如果两名怨气值为 c 的罪犯被关押在同一监狱，他们之间会发生摩擦，并造成影响力为 c 的冲突事件。

每年年末，警察局会将本年内监狱中的所有冲突事件按影响力从大到小排成一个列表，然后上报到 S 城 Z 市长那里。公务繁忙的 Z 市长只会去看列表中的第一个事件的影响力，如果影响很坏，他就会考虑撤换警察局局长。

在详细考察了 N 名罪犯间的矛盾关系后，警察局长觉得压力巨大。他准备将罪犯们在两座监狱内重新分配，以求产生的冲突事件影响力都较小，从而保住自己的乌纱帽。假设只要处于同一监狱内的某两个罪犯间有仇恨，那么他们一定会在每年的某个时候发生摩擦。

那么，应如何分配罪犯，才能使 Z 市长看到的那个冲突事件的影响力最小？这个最小值是多少？罪犯数量 $N \leq 2 * 10^4$，仇恨关系的数量 $M \leq 10^5$。

在 0x49 节“数据结构进阶”的练习中，我们要求读者用并查集解决了本题。此处我们换用二分图判定的方法再次求解。

考虑这样一个判定问题：是否存在一种分配罪犯的方案，使 Z 市长看到的那个冲突事件的影响力不超过 mid。显然，当 mid 较小时可行的方案，对于更大的 mid 仍然可行。换言之，本题的答案具有单调性，可以通过二分法，把求最值的问题转化为判定问题。

二分答案，设当前二分的值为 mid。此时，任意两个仇恨程度大于 mid 的罪犯都必须被安排在不同的监狱。我们把罪犯作为节点，在仇恨程度大于 mid 的两个罪犯之间连边，得到一张无向图。这张无向图的节点需要被分成两个集合（两个监狱），并且每个集合内部没有边（同一个监狱内没有仇恨程度大于 mid 的罪犯）。所以，我们用染色法判定该无向图是否为二分图即可。若是二分图，则令二分上界 $r = mid$，否则令下界 $l = mid + 1$。

二分图最大匹配

“任意两条边都没有公共端点”的边的集合被称为图的一组**匹配**。在二分图中，包含边数最多的一组匹配被称为二分图的**最大匹配**。

对于任意一组匹配 S（S 是一个边集），属于 S 的边被称为“匹配边”，不属于 S 的边被称为“非匹配边”。匹配边的端点被称为“匹配点”，其他节点被称为“非匹配点”。如果在二分图中存在一条连接两个非匹配点的路径 $path$，使得非匹配边与匹配边在 $path$ 上交替出现，那么称 $path$ 是匹配 S 的**增广路**，也称交错路。

增广路显然具有以下性质。

1. 长度 len 是奇数。

2. 路径上第 $1,3,5,\cdots,len$ 条边是非匹配边，第 $2,4,6,\cdots,len-1$ 条边是匹配边。

正因为以上性质，如果我们把路径上所有边的状态取反，原来的匹配边变成非匹配的，原来的非匹配边变成匹配的，那么得到的新的边集 S' 仍然是一组匹配，并且匹配边数增加了 1。进一步可以得到推论：

二分图的一组匹配 S 是最大匹配，当且仅当图中不存在 S 的增广路。

匈牙利算法（增广路算法）

匈牙利算法，又称增广路算法，用于计算二分图最大匹配。它的主要过程为：

1. 设 $S=\emptyset$，即所有边都是非匹配边；

2. 寻找增广路 $path$，把路径上所有边的匹配状态取反，得到一个更大的匹配 S'；

3. 重复第 2 步，直至图中不存在增广路。

该算法的关键在于如何找到一条增广路。匈牙利算法依次尝试给每一个左部节点 x 寻找一个匹配的右部节点 y。右部点 y 能与左部点 x 匹配，需要满足以下两个条件之一。

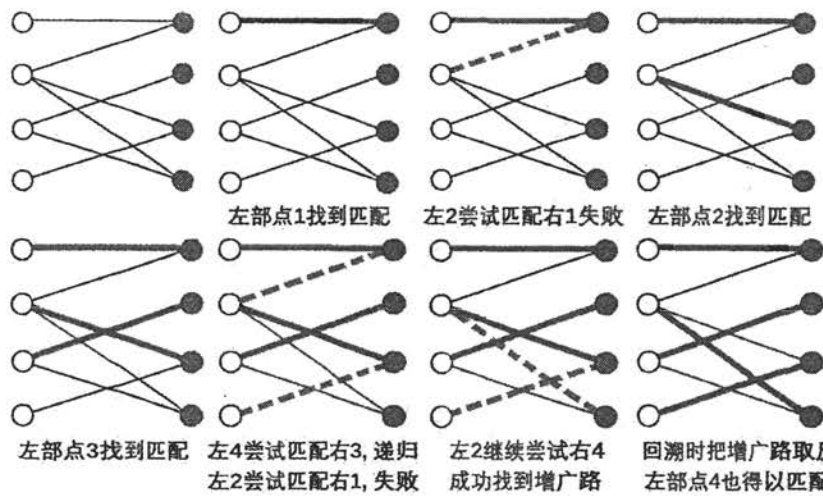
1. y 本身就是非匹配点。

此时无向边 (x,y) 本身就是非匹配边，自己构成一条长度为 1 的增广路。

2. y 已经与左部点 x' 匹配，但从 x' 出发能找到另一个右部点 y' 与之匹配。

此时路径 $x\sim y\sim x'\sim y'$ 为一条增广路。

在实际的程序实现中，我们采用深度优先搜索的框架，递归地从 x 出发寻找增广路。若找到，则在深搜回溯时，正好把路径上的匹配状态取反。另外，可以用全局 bool 数组标记节点的访问情况，避免重复搜索。



匈牙利算法的正确性基于贪心策略，它的一个重要特点是：当一个节点成为匹配点后，至多因为找到增广路而更换匹配对象，但是绝不会再变回非匹配点。

对于每个左部节点，寻找增广路最多遍历整张二分图一次。因此，该算法的时间复杂度为 $O(NM)$。

```
bool dfs(int x) {
    for (int i = head[x], y; i; i = next[i])
        if (!visit[y = ver[i]]) {
            visit[y] = 1;
            if (!match[y] || dfs(match[y])) {
                match[y] = x; return true;
            }
        }
    return false;
}

for (int i = 1; i <= n; i++) {  // 在 main 函数中
    memset(visit, 0, sizeof(visit));
    if (dfs(i)) ans++;
}
```

例题 棋盘覆盖 AcWing372

给定一个 N 行 M 列的棋盘，已知某些格子禁止放置。求最多能往棋盘上放多少块长度为 2、宽度为 1 的骨牌。骨牌的边界与格线重合（骨牌占用两个格子），并且任意两张骨牌都不重叠。$N, M \leq 100$。

在 0x56 节“状态压缩 DP”的例题“Mondriaan's Dream”中，我们解决了用 1*2 的长方形铺满 $N * M$ 网格的方案数问题。与之不同的是，本题的棋盘有禁止放置的格子，并且只需求出能放置的 1*2 骨牌的最大数量。当 N, M 较小时，依然可以用状态压缩 DP 解决，更高效的方法则是构建出二分图匹配的模型。

二分图匹配的模型有两个要素：

1. 节点能分成独立的两个集合，每个集合内部有 0 条边；

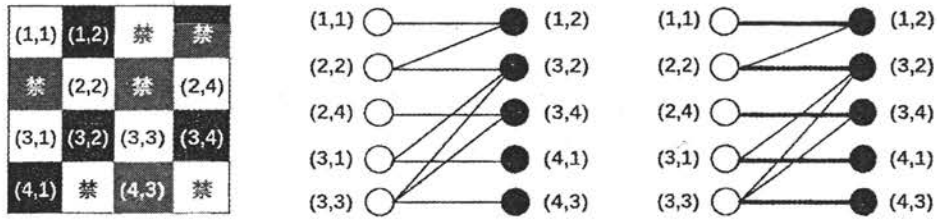
2. 每个节点只能与 1 条匹配边相连。

我们把它简称为“0 要素”和“1 要素”。在把实际问题抽象成二分图匹配时，我们就要寻找题目中具有这种“0”和“1”性质的对象，从而发现模型构建的突破口。

在本题中，任意两张骨牌都不重叠，也就是每个格子只能被 1 张骨牌覆盖。而骨牌的大小为 1*2，覆盖 2 个相邻的格子。这恰好与“1 要素”相对应。于是，我们可以

把棋盘上没有被禁止的格子作为节点，把骨牌作为无向边（两个相邻的格子对应的节点之间连边）。

如果把棋盘黑白染色（行号加列号为偶数的格子染成白色，行号加列号为奇数的格子染成黑色），那么两个相同颜色的格子不可能被同一骨牌覆盖，也就是同色格子之间没有边相连。这恰好与“0 要素”对应。于是，刚才建立的无向图是一张二分图，可把白色格子作为左部节点，黑色格子作为右部节点。



要让骨牌不重叠的情况下尽量多放，就是求上述二分图的最大匹配。使用匈牙利算法，时间复杂度为 $O(N^2M^2)$。

例题 車的放置 AcWing373

给定一个 N 行 M 列的棋盘，已知某些格子禁止放置。问棋盘上最多能放多少个不能互相攻击的車。車放在格子里，攻击范围与中国象棋的“車”一致。$N,M \le 200$。

“1 要素”：每行、每列只能放 1 个車（不然就能互相攻击到）。某个格子 (i,j) 如果放了車，就等于占用了第 i 行与第 j 列放車的“名额”。

因此，我们可以把行、列看作节点，一共 $N+M$ 个节点。如果格子 (i,j) 没有被禁止，就在第 i 行对应的节点与第 j 行对应的节点之间连无向边。

“0 要素”：每个車显然不能既在第 i_1 行又在第 i_2 行，所以两个“行”对应的节点之间没有边。对于“列”也同理。

因此，刚才构建的无向图是二分图，我们可以把 N 个“行节点”作为左部，M 个“列节点”作为右部。

要在不能互相攻击的前提下放置的車最多，就是求上述二分图的最大匹配。时间复杂度为 $O\big((N+M)*N*M\big)$。

完备匹配

给定一张二分图，其左部、右部节点数相同，均为 N 个节点。如果该二分图的最大匹配包含 N 条匹配边，则称该二分图具有完备匹配。

多重匹配

给定一张包含 N 个左部节点、M 个右部节点的二分图。从中选出尽量多的边，使第 $i(1 \le i \le N)$ 个左部节点至多与 kl_i 条选出的边相连，第 $j(1 \le j \le M)$ 个右部

节点至多与 kr_j 条选出的边相连。该问题被称为二分图的多重匹配。

当 $kl_i = kr_j = 1$ 时，上述问题就简化为二分图最大匹配。因此，多重匹配是一个广义的“匹配”问题，每个节点可以与不止一条“匹配”边相连，但不能超过一个给定的限制。

多重匹配一般有四种解决方案。

1. 拆点。把第 i 个左部节点拆成 kl_i 个不同的左部节点，第 j 个右部节点拆成 kr_j 个右部节点。对于原图中的每条边 (i,j)，在 i 拆成的所有节点与 j 拆成的所有节点之间连边。然后求解二分图最大匹配。

2. 如果所有的 $kl_i = 1$，或者所有的 $kr_j = 1$，即只有一侧是“多重”匹配，不妨设左部是“多重”的，那么可以直接在匈牙利算法中让每个左部节点执行 kl_i 次 dfs。

3. 在第 2 种方案中，当然也可以交换左、右两部，设“右部”是多重的，修改匈牙利算法的实现，让右部节点可以匹配 kr_j 次，超过匹配次数后，就要依次尝试递归当前匹配的 kr_j 个左部节点，看能否找到增广路。

4. 网络流。这是最一般也是最高效的解决方法，我们将在 0x6A 节讲解。

例题 导弹防御塔 AcWing374

Freda 的城堡遭受了 M 个入侵者的攻击！Freda 控制着 N 座导弹防御塔，每座塔都有足够数量的导弹，但是每次只能发射一枚。在发射导弹时，导弹需要 T_1 秒才能从防御塔中射出，而在发射导弹后，发射这枚导弹的防御塔需要 T_2 分钟来冷却。

所有导弹都有相同的匀速飞行速度 V，并且会沿着距离最短的路径去打击目标。计算防御塔到目标的距离 Distance 时，你只需要计算水平距离，而忽略导弹飞行的高度。导弹在空中飞行的时间就是 (Distance/V) 分钟，导弹到达目标后可以立即将它击毁。

现在，给出 N 座导弹防御塔的坐标，M 个入侵者的坐标，T_1，T_2 和 V。因为 Freda 的小伙伴 Rainbow 就要来拜访城堡了，你需要求出至少多少分钟才能击退所有的入侵者。

数据范围：$1 \le N, M \le 50$，$T_1, T_2, V <= 2000$，坐标绝对值不超过 10000。

时间较短时如果能击退所有入侵者，那么对于更长的时间，显然也能击退入侵者，因此本题的答案具有单调性。二分答案，设当前二分的值为 mid，问题转化为判定“能否在 mid 秒之内击退所有入侵者”。

“1 要素”：每个入侵者被攻击 1 次后就会被击退。每座塔虽然可以发射很多枚导弹，但是每枚导弹只能攻击 1 个入侵者。

已知发射预热时间 T_1、冷却时间 T_2，我们容易计算出每座塔在 mid 分钟内至多能发射出多少枚导弹，记为 P。容易发现，本题是一个多重匹配问题，可以把入侵者作为二分图左部节点，每座防御塔拆成 P 个导弹，作为二分图右部节点。最终共有 M 个

左部节点、$N*P$ 个右部节点。

注意，因为防御塔、入侵者的坐标各不相同，所以从塔飞到入侵者的时间也可能不同。我们在连边时，还需要检查导弹是否有足够的时间飞到入侵者处。因此，一座塔的 P 个导弹是不等价的，这个多重匹配问题必须用拆点法解决。如果在 mid 时间内，第 $i(1 \le i \le M)$ 个入侵者能被第 $j(1 \le j \le N)$ 座塔的第 $k(1 \le k \le P)$ 个导弹击中，那么就在第 i 个左部点和第 $(j-1)*P+k$ 个右部点之间连一条无向边。

用匈牙利算法计算二分图的最大匹配。若每个左部点都能找到匹配，则说明 mid 分钟内能击退所有入侵者，可令二分上界 $r = mid$，否则令下界 $l = mid + 1$。

二分图带权匹配

给定一张二分图，二分图的每条边都带有一个权值。求出该二分图的一组最大匹配，使得匹配边的权值总和最大。这个问题称为二分图的带权最大匹配，也称二分图最优匹配。注意，二分图带权最大匹配的前提是匹配数最大，然后再最大化匹配边的权值总和。

二分图带权最大匹配有两种解法：费用流和 KM 算法。本节先来介绍 KM 算法，费用流将在 0x6A 节进行讲解。KM 算法程序实现简单，在稠密图上的效率一般高于费用流。不过，KM 算法有很大的局限性，**只能在满足“带权最大匹配一定是完备匹配”的图中正确求解**。所以一般情况下，本书鼓励读者采用费用流来计算二分图带权最大匹配。

在接下来的讨论中，我们设二分图左、右两部的节点数均为 N。

交错树

在匈牙利算法中，如果从某个左部节点出发寻找匹配失败，那么在 DFS 的过程中，所有访问过的节点，以及为了访问这些节点而经过的边，共同构成一棵树。

这棵树的根是一个左部节点，所有叶子节点也都是左部节点（因为最终匹配失败了），并且树上第 1,3,5,… 层的边都是非匹配边，第 2,4,6,… 层的边都是匹配边。因此，这棵树被称为**交错树**。

顶标

全称“顶点标记值”。在二分图中，给第 $i(1 \le i \le N)$个左部节点一个整数值 A_i，给第 $j(1 \le j \le N)$ 个右部节点一个整数值 B_j。同时，必须满足 $\forall i,j, A_i + B_j \ge w(i,j)$，其中 $w(i,j)$ 表示第 i 个左部点与第 j 个右部点之间的边权（没有边时设为负无穷）。这些整数值 A_i, B_j 称为节点的**顶标**。

相等子图

二分图中所有节点和满足 $A_i + B_j = w(i,j)$ 的边构成的子图，称为二分图的**相等**

子图。

定理

若相等子图中存在完备匹配，则这个完备匹配就是二分图的带权最大匹配。

证明：

在相等子图中，完备匹配的边权之和等于 $\sum_{i=1}^{N}(A_i + B_i)$，即所有顶标之和。

因为顶标满足 $\forall i,j,\ A_i + B_j \geq w(i,j)$，所以在整个二分图中，任何一组匹配的边权之和都不可能大于所有顶标的和。

证毕。

KM 算法的基本思想就是，先在满足 $\forall i,j,\ A_i + B_j \geq w(i,j)$ 的前提下，给每个节点随意赋值一个顶标，然后采取适当的策略不断扩大相等子图的规模，直至相等子图存在完备匹配。例如，我们可以赋值 $A_i = \max_{1\leq j\leq N}\{w(i,j)\}$，$B_j = 0$。

对于一个相等子图，我们用匈牙利算法求它的最大匹配。若最大匹配不完备，则说明一定有一个左部节点匹配失败。该节点匹配失败的那次 DFS 形成了一棵交错树，记为 T。

考虑匈牙利算法的流程，容易发现以下两条结论。

1. 除了根节点以外，T 中其他的左部点都是**从右部点沿着匹配边**访问到的，即在程序中调用了 dfs($match[y]$)，其中 y 是一个右部节点。

2. T 中所有的右部点都是**从左部点沿着非匹配边**访问到的。

在寻找到增广路以前，我们不会改变已有的匹配，所以一个右部点沿着匹配边能访问到的左部点是固定的。为了让匹配数增加，我们只能从第 2 条结论入手，考虑“怎样能让左部点沿着非匹配边访问到更多的右部点”。

假如我们把交错树 T 中的所有左部节点顶标 $A_i(i \in T)$ 减小一个整数值 Δ，把 T 中的所有右部节点顶标 $B_j(j \in T)$ 增大一个整数值 Δ，节点的访问情况会有哪些变化？我们分两方面进行讨论。

1. 右部点 j 沿着匹配边，递归访问 $i = match[j]$ 的情形。

对于一条匹配边，显然要么 $i,j \in T$（被访问到），要么 $i,j \notin T$（没被访问到）。故 $A_i + B_j$ 不变，匹配边仍然属于相等子图。

2. 左部点 i 沿着非匹配边，访问右部点 j，尝试与之匹配的情形。

因为左部点的访问是被动的（被右部点沿着匹配边递归），所以只需考虑 $i \in T$。

(1) 若 $i,j \in T$，则 $A_i + B_j$ 不变。

即以前能从 i 访问到的点 j，现在仍能访问。

(2) 若 $i \in T, j \notin T$，则 $A_i + B_j$ 减小。

即以前从 i 访问不到的点 j，现在有可能访问到了。

为了保证顶标符合前提条件 $\forall i,j,\ A_i+B_j\geq w(i,j)$，我们就在所有 $i\in T, j\notin T$ 的边 (i,j) 之中，找出最小的 $A_i+B_j-w(i,j)$，作为 Δ 的值。只要原图存在完备匹配，这样的边一定存在。上述方法既不会破坏前提条件，又能保证至少有一条新的边会加入相等子图，使交错树中至少一个左部点能访问到的右部点增多。

不断重复以上过程，直到每一个左部点都匹配成功，就得到了相等子图的完备匹配，即原图的带权最大匹配。具体实现时，可以在 DFS 的过程中维护一个数组记录可能更新 Δ 的值，以便快速求出新的 Δ。时间复杂度为 $O(N^2M)$。

```
const int N = 105;
int w[N][N]; // 边权
int la[N], lb[N]; // 左、右部点的顶标
bool va[N], vb[N]; // 访问标记：是否在交错树中
int match[N]; // 右部点匹配了哪一个左部点
int n, delta, upd[N];
bool dfs(int x) {
    va[x] = 1; // 访问标记：x 在交错树中
    for (int y = 1; y <= n; y++)
        if (!vb[y])
            if (la[x] + lb[y] - w[x][y] == 0) { // 相等子图
                vb[y] = 1; // 访问标记：y 在交错树中
                if (!match[y] || dfs(match[y])) {
                    match[y] = x;
                    return true;
                }
            }
            else upd[y] = min(upd[y], la[x] + lb[y] - w[x][y]);
    return false;
}
int KM() {
    for (int i = 1; i <= n; i++) {
        la[i] = -(1 << 30); // -inf
        lb[i] = 0;
        for (int j = 1; j <= n; j++)
            la[i] = max(la[i], w[i][j]);
    }
    for (int i = 1; i <= n; i++)
        while (true) { // 直到左部点找到匹配
```

```
            memset(va, 0, sizeof(va));
            memset(vb, 0, sizeof(vb));
            for (int j = 1; j <= n; j++) upd[j] = 1 << 30;
            if (dfs(i)) break;
            for (int j = 1; j <= n; j++)
                if (!vb[j]) delta = min(delta, upd[j]);
            for (int j = 1; j <= n; j++) { // 修改顶标
                if (va[j]) la[j] -= delta;
                if (vb[j]) lb[j] += delta;
            }
        }
    int ans = 0;
    for (int i = 1; i <= n; i++) ans += w[match[i]][i];
    return ans;
}
```

可以进一步优化，把时间复杂度降低到 $O(N^3)$。注意到每次修改顶标后，相等子图只会扩大，原来相等子图中的边并不会消失。这启发我们每次 DFS 中的很多访问都是冗余的——已经遍历过的交错树没有必要再次遍历。实际上，我们只需要从 $A_i + B_j - w(i,j)$ 最小的边，也就是新加入相等子图的边开始继续向下搜索，在已有的交错树基础上继续扩展，而没有必要每次都从交错树的根开始。具体实现中，我们可以记录数组 $upd[j]$ 表示每个右部点 j 对应的最小 Δ，当一个点无法在当前的相等子图中找到匹配时，我们更新顶标，然后直接从最小边开始下一次 DFS。匹配成功后，因为 DFS 不一定从根开始，无法利用递归的回溯来更新增广路，我们还需要记录 $last$ 数组表示每一个右部点 j 在交错树中的上一个右部点，沿着 $last$ 倒推回去即可找出增广路。细节请参阅参考程序中的注释。另外，本程序还有非递归的版本。为了方便读者理解，这里给出递归版，非递归版可以参阅配套资源中的代码。

```
bool dfs(int x, int fa) {
    va[x] = 1;
    for (int y = 1; y <= n; y++)
        if (!vb[y])
            if (la[x] + lb[y] - w[x][y] == 0) { // 相等子图
                vb[y] = 1; last[y] = fa;
                if (!match[y] || dfs(match[y], y)) {
                    match[y] = x;
                    return true;
                }
```

```
            }
            else if (upd[y] > la[x] + lb[y] - w[x][y]) {
                upd[y] = la[x] + lb[y] - w[x][y];
                last[y] = fa;
            }
    return false;
}

long long KM() {
    for (int i = 1; i <= n; i++) {
        la[i] = -(1 << 30); lb[i] = 0;
        for (int j = 1; j <= n; j++)
            la[i] = max(la[i], 1ll * w[i][j]);
    }
    for (int i = 1; i <= n; i++) {
        memset(va, 0, sizeof(va));
        memset(vb, 0, sizeof(vb));
        memset(last, 0, sizeof(last));
        memset(upd, 0x7f, sizeof(upd));
        // 从右部点 st 匹配的左部点开始 dfs，一开始假设有一条 0-i 的匹配
        int st = 0; match[0] = i;
        while (match[st]) { // 当到达一个非匹配点 st 时停止
            long long delta = 1ll << 60;
            if (dfs(match[st], st)) break;
            for (int j = 1; j <= n; j++)
                if (!vb[j] && upd[j] < delta) {
                    delta = upd[j];
                    st = j; // 下一次直接从最小边开始 DFS
                }
            for (int j = 1; j <= n; j++) { // 修改顶标
                if (va[j]) la[j] -= delta;
                if (vb[j]) lb[j] += delta; else upd[j] -= delta;
            }
            vb[st] = 1;
        }
        while (st) { // 倒推更新增广路
            match[st] = match[last[st]];
            st = last[st];
```

```
            }
        }
        long long ans = 0;
        for (int i = 1; i <= n; i++) ans += w[match[i]][i];
        return ans;
    }
```

例题 **Ants** AcWing375

平面上共有 $2*N$ 个点，N 个是白点，N 个是黑点。对于每个白点，找到一个黑点，把二者用线段连起来，要求最后所有线段都不相交，求一种方案。数据保证有解，$N \leq 100$。

题目要求最后所有线段都不相交，等价于让每条线段的长度之和最小。

试想，如果第 i_1 个白点连第 j_1 个黑点，第 i_2 个白点连第 j_2 个黑点，并且线段 (i_1,j_1) 和 (i_2,j_2) 相交，那么就交换一下，让 i_1 连 j_2，i_2 连 j_1，这样就不会交叉，并且由三角形两边之和大于第三边可知，两条线段的长度之和一定变小。如下图所示。



把 N 个白点看作左部点，N 个黑点看作右部点。$\forall i,j$，让第 i 个白点和第 j 个黑点连边，边权 $w(i,j)$ 等于它们在平面上的距离，求二分图带权最小匹配，匹配边就是线段连接的方案（数据保证一定有解）。

注意，当用 KM 算法求二分图带权最小匹配时，我们把所有边权 $w(i,j)$ 取为相反数，转化为求二分图带权最大匹配，最后把 $-ans$ 作为答案即可。

0x69 二分图的覆盖与独立集

二分图最小点覆盖

给定一张二分图，求出一个最小的点集 S，使得图中任意一条边都有至少一个端点属于 S。这个问题被称为二分图的**最小点覆盖**（vertex cover），简称最小覆盖。

König 定理

二分图最小点覆盖包含的点数等于二分图最大匹配包含的边数。

证明：

首先，因为最大匹配是原二分图边集的一个子集，并且所有边都不相交，所以至少需要从每条匹配边中选出一个端点。因此，最小点覆盖包含的点数不可能小于最大匹配包含的边数。如果能对任意二分图构造出一组点覆盖，其包含的点数等于最大匹配包含的边数，那么定理就能得证。构造方法如下：

1. 求出二分图最大匹配；

2. 从左部每个非匹配点出发，再执行一次 DFS 寻找增广路的过程（一定会失败），标记访问过的节点；

3. 取**左部未被标记的点、右部被标记的点**，就得到了二分图最小点覆盖。

下面证明该构造的正确性。经过上述构造方法后：

1. 左部非匹配点一定都被标记——因为它们是出发点；

2. 右部非匹配点一定都没被标记——否则就找到了增广路；

3. 一对匹配点要么都被标记，要么都没被标记——因为在寻找增广路的过程中，左部匹配点只能通过右部到达。

在构造中，我们取了左部未被标记的点、右部被标记的点。根据上面的讨论可以发现，恰好是每条匹配边取了一个点，所以选出的点数等于最大匹配包含的边数。

再来讨论这种取法是否覆盖了所有的边：

1. 匹配边一定被覆盖——因为恰好有一个端点被取走；

2. 不存在连接两个非匹配点的边——否则就有长度为 1 的增广路了；

3. 连接左部非匹配点 i、右部匹配点 j 的边——因为 i 是出发点，所以 j 一定被访问。而我们取了右部所有被标记的点，因此这样的边也被覆盖；

4. 连接左部匹配点 i、右部非匹配点 j 的边—— i 一定没有被访问，否则再走到 j 就找到了增广路。而我们取了左部所有未被标记的点，因此这样的边也被覆盖。

证毕。

例题 **Machine Schedule** AcWing376

有两台机器 A,B 及 N 个任务。每台机器有 M 种不同的模式。

对于每个任务 $i(1 \le i \le N)$，给定两个整数 $a[i]$ 和 $b[i]$，表示如果该任务在 A 上执行，需要设置模式为 $a[i]$，如果在 B 上执行，需要模式为 $b[i]$。

任务可以以任意顺序被执行，但每台机器转换一次模式就要重启一次。求怎样分配任务并合理安排顺序，能使机器重启次数最少。

数据范围：$1 \le N,M \le 100$，$1 \le a[i],b[i] \le M$。

在二分图最大匹配的例题中，我们经常寻找题目里的“0 要素”和“1 要素”，作为解答的突破口。二分图最小覆盖的模型特点则是：**每条边有 2 个端点，二者至少选择一个**。我们不妨称之为“2 要素”。如果一个题目具有“2 要素”的特点，那么可以尝试抽象成二分图最小覆盖模型求解。

在这道题目中，每个“任务”要么在机器 A 上以模式 $a[i]$ 执行，要么在机器 B 上以模式 $b[i]$ 执行，二者必选其一。因此，我们可以把机器 A 的 M 种模式作为 M 个左部点，机器 B 的 M 种模式作为 M 个右部点，每个任务作为无向边，连接左部的第 $a[i]$ 个节点与右部的第 $b[i]$ 个节点。

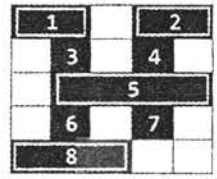
这张图显然是一张二分图，求出它的最小覆盖，就等价于用尽量少的模式（启动次数）来执行所有的任务。时间复杂度为 $O(NM)$。

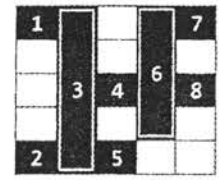
例题 **Muddy Fields** AcWing377

在一块 $N*M$ 的网格状地面上，有一些格子是泥泞的，其他格子是干净的。现在需要用一些宽度为 1、长度任意的木板把泥地盖住，同时不能盖住干净的地面。每块木板必须覆盖若干个完整的格子，木板可以重叠。求最少需要多少块木板。$N,M \le 50$。

每块泥地 (i,j) 要么被第 i 行的一块横着的木板盖住，要么被第 j 列的一块竖着的木板盖住，二者至少选择一个，这是本题的“2 要素”。

因为木板不能盖住干净的地面，所以横着的木板只能放在同一行若干个连续的泥地上，竖着的木板只能放在同一列若干个连续的泥地上。我们把这些连续的泥地分别称为“行泥泞块”和“列泥泞块”。如下页图所示。





把“行泥泞块”作为二分图左部节点，“列泥泞块”作为二分图右部节点。对于每块泥地，在它所属的“行泥泞块”与“列泥泞块”之间连边。

求出二分图的最小覆盖，就是用最少的木板（放在泥泞块里）覆盖所有的泥地。

二分图最大独立集

给定一张无向图 $G=(V,E)$，满足下列条件的点集 S 被称为**图的独立集**。

1. $S \subseteq V$
2. $\forall x,y \in S,\ (x,y) \notin E$

通俗地讲，图的独立集就是“任意两点之间都没有边相连”的点集。包含点数最多的一个就是图的最大独立集。

对应地，“任意两点之间都有一条边相连”的子图被称为无向图的“**团**”。点数最多的团被称为图的最大团。

定理

无向图 G 的最大团等于其补图 G' 的最大独立集。

注：$G'=(V,E')$ 被称为 $G=(V,E)$ 的补图，其中 $E'=\{(x,y) \notin E\}$。

正确性显然。值得提醒的是，在一些题目中，**补图转化思想**能成为解答的突破口。

对于一般无向图，最大团、最大独立集是 NPC 问题[①]。

定理

设 G 是有 n 个节点的二分图，G 的最大独立集的大小等于 n 减去最大匹配数。

证明：

选出最多的点构成独立集

$\Leftrightarrow$ 在图中去掉最少的点，使剩下的点之间没有边

$\Leftrightarrow$ 用最少的点覆盖所有的边

因此，去掉二分图的最小点覆盖，剩余的点就构成二分图的最大独立集。而最小点覆盖数等于最大匹配数。故最大独立集大小等于 n 减去最大匹配数。

① NPC 问题，即 NP 完全问题，是理论计算机科学中有关计算复杂性的一个概念。到目前为止，尚未找到任何 NPC 问题的多项式时间解法。换言之，此类问题“难解”，目前只能使用暴力搜索等方法计算。

证毕。

例题 骑士放置 AcWing378

给定一个 $N*M$ 的棋盘，有一些格子禁止放棋子。问棋盘上最多能放多少个不能互相攻击的骑士（国际象棋的“骑士”，类似于中国象棋的“马”，按照“日”字攻击，但没有中国象棋“别马腿”的规则）。$N,M \leq 100$。

与例题“棋盘覆盖”类似，对棋盘进行黑白染色，黑、白色的格子分别作为二分图左、右部节点。

若两个格子是“日”字的对角（能互相攻击到），则在它们对应的节点之间连边。容易发现，“日”字的两个对角格子的颜色一定不同。因此，我们建出的图确实是一张二分图。

求上述二分图的最大独立集即可。

有向无环图的最小路径点覆盖

给定一张**有向无环图**，要求用尽量少的不相交的简单路径，覆盖有向无环图的所有顶点（也就是每个顶点恰好被覆盖一次）。这个问题被称为有向无环图的**最小路径点覆盖**，简称“最小路径覆盖”。

设原来的有向无环图为 $G=(V,E)$，$n=|V|$。把 G 中的每个点 x 拆成编号为 x 和 $x+n$ 的两个点。建立一张新的二分图，$1\sim n$ 作为二分图左部点，$n+1\sim 2n$ 作为二分图右部点，对于原图的每条有向边 (x,y)，在二分图的左部点 x 与右部点 $y+n$ 之间连边。最终得到的二分图称为 G 的拆点二分图，记为 G_2。

定理

有向无环图 G 的最小路径点覆盖包含的路径条数，等于 n 减去拆点二分图 G_2 的最大匹配数。

证明：

在有向无环图 $G=(V,E)$ 的最小路径覆盖中，对于任意的 $x \in V$，因为路径不相交，所以 x 的入度和出度都不超过 1。因为每个节点都被覆盖，所以 x 的入度和出度至少有一个是 1。

因此，最小路径覆盖中的所有边，在拆点二分图 G_2 中构成一组匹配。最小路径覆盖中每条边 (x,y) 的起点 x 与二分图每条匹配边 $(x,y+n)$ 的左部点 x 是一一对应的。

特别地，对于每条路径的终点 t，因为 t 没有出边，所以在二分图中，t 匹配失

败。即路径的终点和二分图左部的非匹配点是一一对应的。

路径覆盖包含的路径条数最少

⇔ 路径的终点数（出度为 0 的点数）最少

⇔ 二分图左部非匹配点最少

故 G 的最小路径覆盖的路径数等于 n 减去拆点二分图 G_2 的最大匹配数。

证毕。

给定一张有向无环图，要求用尽量少的**可相交**的简单路径，覆盖有向无环图的所有顶点（也就是一个节点可以被覆盖多次）。这个问题被称为有向无环图的**最小路径可重复点覆盖**。

在最小路径可重复点覆盖中，若两条路径 $\cdots \to u \to p \to v \to \cdots$ 和 $\cdots \to x \to p \to y \to \cdots$ 在点 p 相交，则我们在原图中添加一条边 (x,y)，让第二条路径直接走 $x \to y$，就可以避免重复覆盖点 p。

进一步地，如果我们把原图中所有间接连通的点对 x,y 直接连上有向边 (x,y)，那么任何“有路径相交的点覆盖”一定都能转化成“没有路径相交的点覆盖”。

综上所述，有向无环图 G 的最小路径可重复点覆盖，等价于先对有向图**传递闭包**，得到有向无环图 G'，再在 G' 上求一般的（路径不可相交的）最小路径点覆盖。

例题 **Vani 和 Cl2 捉迷藏** AcWing379

Vani 和 cl2 在一片树林里捉迷藏。这片树林里有 N 座房子，M 条有向道路，组成了一张有向无环图。$N \le 200$，$M \le 30000$。

树林里的树非常茂密，足以遮挡视线，但是沿着道路望去，却是视野开阔。如果从房子 A 沿着路走下去能够到达 B，那么在 A 和 B 里的人是能够相互望见的。

现在 cl2 要在这 N 座房子里选择 K 座作为藏身点，同时 Vani 也专挑 cl2 作为藏身点的房子进去寻找，为了避免被 Vani 看见，cl2 要求这 K 个藏身点的任意两个之间都没有路径相连。

为了让 Vani 更难找到自己，cl2 想知道最多能选出多少个藏身点。

输入数据的第一行是两个整数 N 和 M。接下来 M 行，每行两个整数 x,y，表示一条从 x 到 y 的有向道路。输出一个整数 K，表示最多能选取的藏身点个数。

题目所求的最多能选取的藏身点个数，就等于最小路径可重复点覆盖包含的路径条数。我们对题目中给出的有向无环图 G 进行传递闭包，得到一张新的有向无环图 G'，然后求出拆点二分图 G_2' 的最大匹配，输出 N 减 G_2' 的最大匹配数即可。

接下来我们证明上述结论。只需证明：在 G' 中最多能选取的藏身点数等于 G' 的



最小路径点覆盖（路径不可相交）包含的路径条数。

设藏身点集合为 $hide$，最小路径点覆盖的路径集合为 $path$。显然，$path$ 的每条路径上至多选出 1 个点。而 $path$ 覆盖了所有节点，故 $|hide| \leq |path|$。若我们能构造出一种使得 $|hide| = |path|$ 的方案，即可完成证明。

先求出 $path$ 的具体方案。设节点 $x \in G'$ 在拆点二分图 G_2' 中对应左部点 x 和右部点 x'。

1. 依次考虑左部每一个非匹配点 x_0。

2. 从 x_0 出发，不断访问 $x_0, match[x_0'], match[match[x_0']'], \cdots$ 直至到达一个左部点 y_0，满足它对应的右部点 y_0' 是非匹配点。

3. 在 G' 中，节点 x_0, y_0 以及刚才经过的点构成一条路径，y_0 是起点，x_0 是终点。

上述找到的所有路径就是覆盖 G' 中所有点的一个方案，且路径互不相交。

下面给出从 $path$ 的每条路径上选出一个藏身点的方法。

1. 选出 $path$ 中每条路径的终点 x_0，构成一个集合 E。

2. 求出从 E 中节点出发，走一条边，到达的节点集合 $next(E)$。

3. 根据传递闭包的性质，G 中任意两个藏身点没有路径相连，等价于 G' 中任意两个藏身点都没有边相连。因此，若 $E \cap next(E) = \emptyset$，则 $hide = E$ 即为所求。

4. 否则，考虑 $E \cap next(E)$ 中的每个节点 e，沿着 e 所在的路径向上走，直到一个节点 $e' \notin next(E)$。从 E 中删除 e，加入 e'。

5. 对于修改后的集合 E，重复执行步骤 3~4，直至步骤 3 中的条件满足。

可以证明，任何时刻在步骤 4 中，我们一定能找到合法的 e'。这是因为如果找不到这样的 e'，就说明 e 所在路径（记为 pe）上的所有点都能够被其他路径的终点到达。设 pe 的起点是 y_0，我们可以延长那条到达 y_0 的路径，代替 pe 去覆盖 pe 上的所有节点，使集合 $path$ 中的路径减少一条，与 $path$ 的最小性矛盾。

以下程序求出本题的解，同时实现了藏身点方案的构造过程：

```
bool cl[222][222]; // 邻接矩阵
int match[222], n, m;
bool vis[222], succ[222];
int hide[222]; // 藏身点集合

bool dfs(int x) {
    for (int i = 1; i <= n; i++)
        if (cl[x][i] && !vis[i]) {
            vis[i] = true;
```

```
            if (!match[i] || dfs(match[i])) {
                match[i] = x;
                return true;
            }
        }
    return false;
}

int main() {
    // 读入
    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        int x, y; scanf("%d%d", &x, &y);
        cl[x][y] = 1;
    }
    // Floyd 传递闭包
    for (int i = 1; i <= n; i++) cl[i][i] = 1;
    for (int k = 1; k <= n; k++)
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                cl[i][j] |= cl[i][k] && cl[k][j];
    for (int i = 1; i <= n; i++) cl[i][i] = 0;
    // 在拆点二分图上求最大匹配
    int ans = n;
    for (int i = 1; i <= n; i++) {
        memset(vis, 0, sizeof(vis));
        ans -= dfs(i);
    }
    cout << ans <<endl;
    // 构造方案，先把所有路径终点（左部非匹配点）作为藏身点
    for (int i = 1; i <= n; i++) succ[match[i]] = true;
    for (int i = 1, k = 0; i <= n; i++)
        if (!succ[i]) hide[++k] = i;
    memset(vis, 0, sizeof(vis));
    bool modify = true;
    while (modify) {
        modify = false;
        // 求出 next(hide)
```

```
        for (int i = 1; i <= ans; i++)
            for (int j = 1; j <= n; j++)
                if (cl[hide[i]][j]) vis[j] = true;
        for (int i = 1; i <= ans; i++)
            if (vis[hide[i]]) {
                modify = true;
                // 不断向上移动
                while (vis[hide[i]]) hide[i] = match[hide[i]];
            }
    }
    for (int i = 1; i <= ans; i++) printf("%d ", hide[i]);
    cout << endl;
}
```

0x6A 网络流初步

网络流是图论中一个博大精深的分支。若要详细讲解网络流的各种模型与应用，甚至可以单独占用一章的篇幅。本节主要希望读者建立对网络流的初步认识，内容会比较浅显，少部分算法和证明的细节会被省略。

一个**网络** $G=(V,E)$ 是一张有向图，图中每条有向边 $(x,y)\in E$ 都有一个给定的权值 $c(x,y)$，称为边的**容量**。特别地，若 $(x,y)\notin E$，则 $c(x,y)=0$。图中还有两个指定的特殊节点 $S\in V$ 和 $T\in V\ (S\neq T)$，分别称为**源点**和**汇点**。

设 $f(x,y)$ 是定义在节点二元组 $(x\in V,y\in V)$ 上的实数函数，且满足：

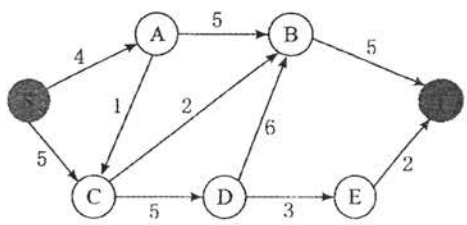
1. $f(x,y)\leq c(x,y)$
2. $f(x,y)=-f(y,x)$
3. $\forall x\neq S,x\neq T,\ \sum_{(u,x)\in E}f(u,x)=\sum_{(x,v)\in E}f(x,v)$

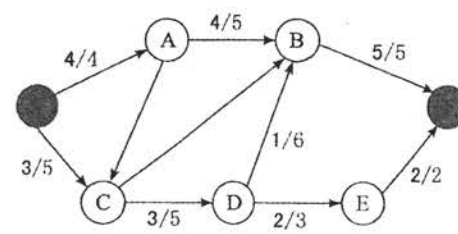
f 称为网络的**流函数**。对于 $(x,y)\in E$，$f(x,y)$ 称为边的**流量**，$c(x,y)-f(x,y)$ 称为边的剩余容量。

$\sum_{(S,v)\in E}f(S,v)$ 称为整个网络的流量（其中 S 表示源点）。

如下图所示。左边是一个网络，有源点 S、汇点 T 以及 A~E 共 7 个点。有向边上的数值标识了边的容量，例如 $c(A,B)=5$。

右边画出了该网络的一个流，在有向边上用“$f(x,y)/c(x,y)$”的形式标注流量，例如 $f(A,B)=4$。流量为 0 的边省略了标注。值得一提的是，按照流函数的定义，图中标注的每条边的**反向边**其实都有一个**负的流量**，例如 $c(B,A)=0$，$f(B,A)=-4$，这些没有标出，请读者留意。





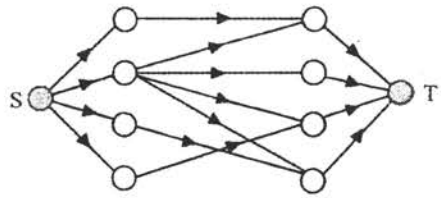
上文给出的流函数 $f(x,y)$ 的三条性质分别称为**容量限制**、**斜对称**和**流量守恒**。尤其是流量守恒定律，它告诉我们网络中除源点、汇点以外，任何节点不储存流，其流入

总量等于流出总量。网络流模型可以形象地描述为：在不超过容量限制的前提下，“流从源点源源不断地产生，流经整个网络，最终全部归于汇点。

最大流

对于一个给定的网络，合法的流函数 f 有很多。其中使得整个网络的流量 $\sum_{(S,v)\in E} f(S,v)$ 最大（其中 S 表示源点）的流函数被称为网络的**最大流**，此时的流量被称为网络的最大流量。

最大流能解决许多实际的问题。就拿我们上两节讨论的二分图为例。对于一张 n 个点 m 条边的二分图，我们可以新增一个源点 S 和一个汇点 T，从 S 到每个左部点连有向边，从每个右部点到 T 连有向边，把原二分图的每条边看作从左部点到右部点的有向边，形成一张 $n+2$ 个点 $n+m$ 条边的网络，网络中每条边的容量都设为 1，如下图所示。容易发现，二分图的最大匹配数就等于网络的最大流量，求出最大流后，所有有“流”经过的点、边就是匹配点、匹配边。



进一步地，若要求二分图多重匹配，则只需把 S 到左部点的有向边容量设为左部点的匹配数量上限 kl_i，右部点到 T 的有向边容量设为右部点的匹配数量上限 kr_i，原二分图每条边的容量设为 1，再求最大流即可。

计算最大流的算法很多，本节简单涉及 Edmonds-Karp 增广路算法和 Dinic 算法。

Edmonds-Karp 增广路算法

若一条从源点 S 到汇点 T 的路径上各条边的剩余容量都大于 0，则称这条路径为一条增广路。显然，可以让一股流沿着增广路从 S 流到 T，使网络的流量增大。Edmonds-Karp 算法的思想就是不断用 BFS 寻找增广路，直至网络上不存在增广路为止。

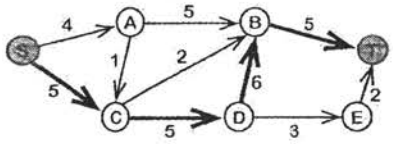
在每轮寻找增广路的过程中，Edmonds-Karp 算法只考虑图中所有 $f(x,y)<c(x,y)$ 的边，用 BFS 找到一条从 S 到 T 包含边数最少（长度最短）的路径，同时计算出路径上各边的剩余容量的最小值 $minf$，则网络的流量就可以增加 $minf$。

需要注意的是，当一条边的流量 $f(x,y)>0$ 时，根据斜对称性质，它的反向边流量 $f(y,x)<0$，此时必定有 $f(y,x)<c(y,x)$。故 Edmonds-Karp 算法在 BFS 时除了原图的边集 E 之外，还应该考虑遍历 E 中每条边的反向边。

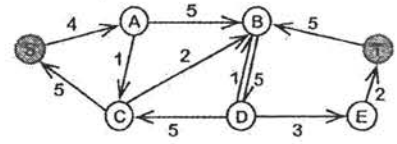
在具体实现时，我们可以按照邻接表“成对存储”技巧（在之前的章节曾多次提

及），把网络的每条有向边及其反向边存在邻接表下标为“2 和 3”“4 和 5”“6 和 7”…的位置上，每条边只记录剩余容量 $c-f$ 即可。当一条边 (x,y) 流过大小为 e 的流时，令 (x,y) 的剩余容量减小 e，(y,x) 的剩余容量增大 e。

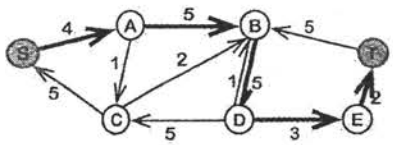
如下图所示。第一幅图是网络中每条边的原始容量，Edmonds-Karp 算法找到了流量为 5 的增广路 $S\to C\to D\to B\to T$。于是在第二幅图中，剩余容量对应发生了变化。紧接着，在第三幅图中，算法又找到了流量为 2 的增广路 $S\to A\to B\to D\to E\to T$。最终在第四幅图中，网络不存在增广路，最大流的大小就是 7。图中省略了剩余容量为 0 的边。



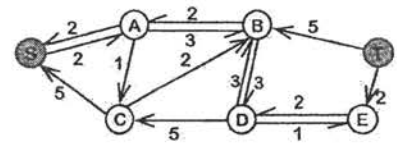
图一



图二



图三



图四

Edmonds-Karp 增广路算法的时间复杂度为 $O(nm^2)$。然而在实际运用中则远远达不到这个上界，效率较高，一般能够处理 $10^3\sim10^4$ 规模的网络。

```
const int inf = 1 << 29, N = 2010, M = 20010;
int head[N], ver[M], edge[M], Next[M], v[N], incf[N], pre[N];
int n, m, s, t, tot, maxflow;
void add(int x, int y, int z) {
    ver[++tot]=y, edge[tot]=z, Next[tot]=head[x], head[x]=tot;
    ver[++tot]=x, edge[tot]=0, Next[tot]=head[y], head[y]=tot;
}

bool bfs() {
    memset(v, 0, sizeof(v));
    queue<int> q;
    q.push(s); v[s] = 1;
    incf[s] = inf; // 增广路上各边的最小剩余容量
    while (q.size()) {
```

```
        int x = q.front(); q.pop();
        for (int i = head[x]; i; i = Next[i])
            if (edge[i]) {
                int y = ver[i];
                if (v[y]) continue;
                incf[y] = min(incf[x], edge[i]);
                pre[y] = i; // 记录前驱，便于找到最长路的实际方案
                q.push(y), v[y] = 1;
                if (y == t) return 1;
            }
    }
    return 0;
}

void update() { // 更新增广路及其反向边的剩余容量
    int x = t;
    while (x != s) {
        int i = pre[x];
        edge[i] -= incf[t];
        edge[i ^ 1] += incf[t]; // 利用“成对存储”的 xor 1 技巧
        x = ver[i ^ 1];
    }
    maxflow += incf[t];
}

int main() {
    while (cin >> m >> n) {
        memset(head, 0, sizeof(head));
        s = 1, t = n; tot = 1; maxflow = 0;
        for (int i = 1; i <= m; i++) {
            int x, y, c; scanf("%d%d%d", &x, &y, &c);
            add(x, y, c);
        }
        while (bfs()) update();
        cout << maxflow << endl;
    }
}
```

Dinic 算法

在任意时刻，网络中所有节点以及剩余容量大于 0 的边构成的子图被称为**残量网络**。Edmonds-Karp 算法每轮可能会遍历整个残量网络，但只找出 1 条增广路，还有进一步优化的空间。

在 0x21 节介绍宽度优先遍历时，我们就提到了节点的层次 $d[x]$，它表示 S 到 x 最少需要经过的边数。在残量网络中，满足 $d[y]=d[x]+1$ 的边 (x,y) 构成的子图被称为**分层图**。分层图显然是一张有向无环图。

Dinic 算法不断重复以下步骤，直到在残量网络中 S 不能到达 T。

1. 在残量网络上 BFS 求出节点的层次，构造分层图。

2. 在分层图上 DFS 寻找增广路，在回溯时实时更新剩余容量。另外，每个点可以流向多条出边，同时还加入了若干剪枝，请参阅参考程序及其注释。

Dinic 算法的时间复杂度为 $O(n^2m)$。实际运用中远远达不到这个上界，可以说是比较容易实现的效率最高的网络流算法之一，一般能够处理 $10^4\sim10^5$ 规模的网络。特别地，Dinic 算法求解二分图最大匹配的时间复杂度为 $O(m\sqrt{n})$，实际表现则更快。

```
const int inf = 1 << 29, N = 50010, M = 300010;
int head[N], ver[M], edge[M], Next[M], d[N], /*当前弧优化*/now[M];
int n, m, s, t, tot, maxflow;
queue<int> q;
void add(int x, int y, int z) {
    ver[++tot]=y, edge[tot]=z, Next[tot]=head[x], head[x]=tot;
    ver[++tot]=x, edge[tot]=0, Next[tot]=head[y], head[y]=tot;
}

bool bfs() { // 在残量网络上构造分层图
    memset(d, 0, sizeof(d));
    while (q.size()) q.pop();
    q.push(s); d[s] = 1; now[s] = head[s];
    while (q.size()) {
        int x = q.front(); q.pop();
        for (int i = head[x]; i; i = Next[i])
            if (edge[i] && !d[ver[i]]) {
                q.push(ver[i]);
                now[ver[i]] = head[ver[i]];
                d[ver[i]] = d[x] + 1;
                if (ver[i] == t) return 1;
```

```
            }
        }
        return 0;
    }

    int dinic(int x, int flow) { // 在当前分层图上增广
        if (x == t) return flow;
        int rest = flow, k, i;
        for (i = now[x]; i && rest; i = Next[i]) {
            now[x] = i; // 当前弧优化（避免重复遍历从 x 出发不可扩展的边）
            if (edge[i] && d[ver[i]] == d[x] + 1) {
                k = dinic(ver[i], min(rest, edge[i]));
                if (!k) d[ver[i]] = 0; // 剪枝，去掉增广完毕的点
                edge[i] -= k;
                edge[i ^ 1] += k;
                rest -= k;
            }
        }
        return flow - rest;
    }

    int main() {
        cin >> n >> m;
        cin >> s >> t; // 源点、汇点
        tot = 1;
        for (int i = 1; i <= m; i++) {
            int x, y, c; scanf("%d%d%d", &x, &y, &c);
            add(x, y, c);
        }
        int flow = 0;
        while (bfs())
            while (flow = dinic(s, inf)) maxflow += flow;
        cout << maxflow << endl;
    }
```

二分图最大匹配的必须边与可行边

给定一张二分图，其最大匹配的方案不一定是唯一的。若任何一个最大匹配方案的匹配边都包括 (x,y)，则称 (x,y) 为二分图最大匹配的必须边。若 (x,y) 至少属于一

个最大匹配的方案，则称 (x,y) 为二分图最大匹配的可行边。

先来考虑一种简化情况——二分图的最大匹配是一组完备匹配。我们先任意求出一组完备匹配的方案，此时左、右部所有的节点都是匹配点。

根据定义，(x,y) 是必须边，当且仅当以下**两个条件都满足**。

1. (x,y) 当前是匹配边。

2. 删除边 (x,y) 后，不能找到另一条从 x 到 y 的增广路。

(x,y) 是可行边，当且仅当满足以下**两个条件之一**。

1. (x,y) 当前是匹配边。

2. (x,y) 是非匹配边，设当前 x 与 v 匹配，y 与 u 匹配，连接边 (x,y) 后，节点 u,v 失去匹配。必须能找到一条从 u 到 v 的增广路。

如果我们把二分图 G 中的“非匹配边”看作从左部到右部的有向边，把“匹配边看作从右部到左部的有向边，构成一张新的有向图，记为 G_1，那么 G 中从 x 到 y 有增广路，等价于 G_1 中存在从 x 到 y 的路径。

考虑必须边的条件，(x,y) 是匹配边，对应 G_1 中有一条从 y 到 x 的有向边。在此前提下，若 x 到 y 还存在一条路径，则 x 和 y 互相可达，必处于同一个强连通分量中。如下图所示。x 和 y 在 G_1 中互相可达，构成一个环，把环上的边的匹配状态取反，得到另一组完备匹配。



因此，必须边的判定条件可改写为：(x,y) 当前是二分图 G 的匹配边，并且 x,y 两点在有向图 G_1 中属于不同的强连通分量。

类似地，可行边的判定条件可改写为：(x,y) 当前是二分图 G 的匹配边，或者 x,y 两点在有向图 G_1 中属于同一个强连通分量。

例题 **舞动的夜晚** AcWing380

L公司和H公司举办了一次联谊晚会。晚会上，L公司的 N 位员工和H公司的 M 位员工打算进行一场交际舞会。在这些领导中，一些L公司的员工和H公司的员工之间是互相认识的，这样的认识关系一共有 E 对。

舞会上，每位员工会尝试选择一名他认识的对方公司的员工作为舞伴，并且每位员工至多跳一支舞。完成的交际舞的数量越多，晚会的气氛就越热烈。顾及晚会的气氛，员工们希望知道，哪些员工之间如果进行了交际舞，就会使整场晚会能够完成的交际舞的最大数量减小。



数据范围：$1 \le N, M \le 10000$，$1 \le E \le 10^5$。

本题是让我们求二分图的不可行边（可行边的补集）。不过，本题不保证二分图的最大匹配是一组完备匹配。

在这种情况下，必须边、可行边的第二个条件会出现问题。设 (x,y) 是非匹配边：

1. x,y 二者之一可能本来就是非匹配点。不妨设 y 是非匹配点，此时直接断开 x 原来的匹配边，连接 (x,y)，得到的匹配集仍然是最大匹配；

2. 即使当前 x 与 v 匹配，y 与 u 匹配，连接边 (x,y) 后，节点 u,v 失去匹配，我们也不一定非要找到从 u 到 v 的增广路。因为从 u 出发找到到达另一个非匹配点 z 的增广路，同样可以得到一组包含 (x,y) 的最大匹配。

所以我们不能像完备匹配中一样，直接用强连通分量进行判定。然而，我们可以借助网络流的源点和汇点。回想我们根据二分图 G 建立有向图 G_1 的方法，G_1 其实就是用网络流计算二分图最大匹配时，残量网络中除去源点、汇点之外的部分——非匹配边（左部到右部）的剩余容量为 1，匹配边的反向边（右部到左部）的剩余容量为 1。

若 z 当前是非匹配点，则 (z,T) 的剩余容量必定为 1。若 v 当前是匹配点，则 (v,T) 的剩余容量必定为 0，(T,v) 的剩余容量必定为 1。换言之，残量网络中存在路径 $z \to T \to v$。若二分图中 u 到 z 有增广路，则残量网络上 u 能到达 z，进而到达 v。故 (u,v) 仍在同一个强连通分量中，它们借助汇点 T 实现了连通。

综上所述，在一般的二分图中（最大匹配不一定是完备匹配），可以用最大流计算出任意一组最大匹配。此时**必须边的判定条件为：(x,y) 流量为 1，并且在残量网络上属于不同的强连通分量。可行边的判定条件为：(x,y) 流量为 1，或者在残量网络上属于同一个强连通分量。**

使用 Dinic 算法求最大流，Tarjan 算法求强连通分量，最后逐一对边进行判定，整个算法的时间复杂度为 $O(E * \sqrt{N+M})$。

最小割

给定一个网络 $G=(V,E)$，源点为 S，汇点为 T。若一个边集 $E' \subseteq E$ 被删去之后，源点 S 和汇点 T 不再连通，则称该边集为网络的**割**。边的容量之和最小的割称为网络的**最小割**。

最大流最小割定理

任何一个网络的最大流量等于最小割中边的容量之和，简记为“最大流 = 最小割”。

假设“最小割 < 最大流”，那么割去这些边之后，因为网络流量尚未最大化，所

以仍然可以找到一条 S 到 T 的增广路，与 S,T 不连通矛盾，故“最小割 $\geq$ 最大流”。如果我们能给出一个“最小割 $=$ 最大流”的构造方案，即可得到上述定理。

求出最大流后，从源点开始沿残量网络 BFS，标记能够到达的点。E 中所有连接“已标记点 x”和“未标记点 y”的边 (x,y) 构成该网络的最小割。

证明：

设在残量网络中，源点 S 可达的点集为 A，不可达的点集为 B，显然 $A \cup B = V$。根据定义，最大流 $=$ 从 S 流出的流量之和 $=$ 从 A 流出的流量之和 $-$ 流入 A 的流量之和。

考虑从 A 流出的流量。对于原图中的任意边 $(x,y) \in E,\ x \in A,\ y \in B$，这些边**一定满流**。若不然，则说明某条边剩余容量大于零，在残量网络中，此时 S 到达 x 后可以沿着这条边到 y，与 $y \in B$ 矛盾。

考虑流入 A 的流量。对于原图中的任意边 $(u,v) \in E,\ u \in B,\ v \in A$，这些边流量**一定为零**。若不然，说明有流经过某条边，则它对应的反向边 (v,u) 具有大于零的剩余容量。在残量网络中，S 到达 v 后可以沿着这条反向边到 u，与 $u \in B$ 矛盾。

综上所述，我们构造的割集，即原图中 A,B 之间边的容量之和，等于从 A 流出的流量之和，等于最大流。

证毕。

例题 **Cable TV Network** AcWing381

给定一张无向图，求最少去掉多少个点，可以使图不连通。点数 $N \leq 50$。

若无向图不连通，则图中必有两个点不连通，但这两个点是未知的。可以枚举这两个点 S 和 T，然后求在剩余的 $N-2$ 个节点中最少去掉多少个，可以使 S 和 T 不连通。当然，我们要求 S 和 T 不是直接相连的，否则不可能达到要求。在每次枚举的结果中取最小值就是本题的答案。

“去掉最少的点使 S,T 不连通”的问题与最小割十分相似，只不过要割的是“点”而不是“边”。我们可以按照下面的方法构建一个网络。

1. 把原来无向图中的每个点 x，拆成 x 和 $x' = x + N$ 两个点。

2. 对 $\forall x \neq S, x \neq T$，连有向边 (x,x')，容量为 1。

3. 对于原无向图的每条边 (x,y)，在网络中连有向边 $(x',y),(y',x)$，容量为 $+\infty$。

以 S' 为网络的源点，T 为网络的汇点，求最小割（最大流），即可得到最少需要去掉的点数。

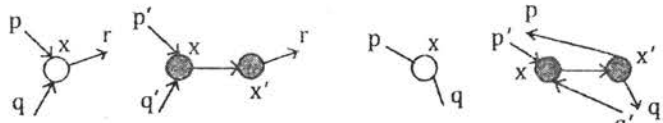
在上面的网络中，(x,x') 这条有向边对应原无向图的节点 x。无向图中任意经过点 x 的路径，对应在网络中必须先到达 x，然后通过 (x,x')，再从 x' 离开。我们一般称 x 为“入点”，x' 为“出点”。容易发现，在无向图中删去一个点，就等价于在

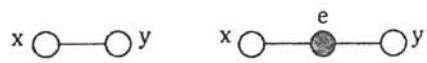
网络中断开 (x,x')。

另外，我们只能删掉无向图的节点，不能删掉无向图的边，所以网络中其他边的容量都设为 $+\infty$（实际可设为大于 N 的整数）。最小割必定不会包含这些容量为 $+\infty$ 的边——因为去掉所有容量为 1 的边（容量之和为 $N-2$），就足以使 S 和 T 不连通了。

综上所述，网络的最小割就是由最少数量的形如 (x,x') 的边构成的边集，也就是原图最少需要去掉的点数。

本题的解法包含两个重要的技巧。一是“**点边转化**”，图的节点可以拆为“入点”和“出点”两个，把点的属性体现在“入点”与“出点”之间的边上；图的边也可以分成两截，在中间加一个节点，把边的属性体现在中间的点上，如下图所示。二是在最小割中，容量为 $+\infty$ 的边具有“防止割断”的含义。





费用流

给定一个网络 $G=(V,E)$，每条边 (x,y) 除了有容量限制 $c(x,y)$，还有一个给定的“单位费用” $w(x,y)$。当边 (x,y) 的流量为 $f(x,y)$ 时，就要花费 $f(x,y)*w(x,y)$。该网络中总花费最小的最大流被称为“**最小费用最大流**”，总花费最大的最大流被称为“**最大费用最大流**”，二者合称为“费用流”模型。注意：费用流的前提是最大流，然后才考虑费用的最值。

类似于“二分图最大匹配”与最大流的关系，“二分图带权最大匹配”可直接用最大费用最大流求解，每条边的权值就是它的单位费用。

Edmonds-Karp 增广路算法

在 Edmonds-Karp 求解最大流的基础上，把“用 BFS 寻找一条包含边数最少的增广路”改为“用 SPFA 寻找一条单位费用之和最小的增广路”（也就是把 $w(x,y)$ 当作边权，在残量网络上求最短路）即可求出最小费用最大流。注意：一条反向边 (y,x) 的费用应设为 $-w(x,y)$。参考程序将以下面的例题为背景给出。

例题 K取方格数 AcWing382

在0x51节“线性DP”的例题“传纸条”中，我们解决了“二取方格数”问题。K取方格数是它的扩展。在一个$N*N$的矩形网格中，每个格子里都写着一个整数。可以从左上角到右下角安排K条路线，每一步只能往下或往右，沿途经过的格子中的整数会被取走。若多条路线重复经过一个格子，只取一次。求能取得的整数的和最大是多少。$N \leq 50$, $K \leq 10$。

当$K=1$时，若把第i行第j列的格子$(1 \leq i,j \leq N)$看作一个节点，从格子(i,j)对应的节点向格子$(i,j+1)$ $(i+1,j)$对应的节点连有向边，则能取得的整数之和的最大值就相当于从$(1,1)$到(N,N)的最长路（只不过权值在节点上）。

当$K>1$时，我们需要限制“每个节点的权值只能被算一次”。这种点或边同时带有权值和上界限制的最优化问题，恰好与费用流的模型非常相似。我们可以按照如下规则构造一个网络。

1. 应用“点边转化”技巧，把每个格子(i,j)对应的节点拆成一个“入点”和一个“出点”，整个网络共有$2N^2$个点。

2. 从每个(i,j)的入点向出点连两条有向边。第一条有向边容量为1，费用为格子(i,j)中的数。第二条有向边容量为$K-1$，费用为0。这表示第一次经过该点时，可以把数取走，之后再经过时就不再计算。

3. 从(i,j)的出点到$(i,j+1)$和$(i+1,j)$的入点连有向边，容量为K，费用为0。

以$(1,1)$的入点为源点，(N,N)的出点为汇点，求最大费用最大流。因为$(1,1)$的入点和出点之间的两条有向边一共只有K的容量，并且“费用流”先保证“最大流”，再最优化“费用”，所以最终会恰好找到K条路线。求出的总费用就是本题的答案。

```
const int N = 5010, M = 200010;
int ver[M], edge[M], cost[M], Next[M], head[N];
int d[N], incf[N], pre[N], v[N];
int n, k, tot, s, t, maxflow, ans;

void add(int x, int y, int z, int c) {
    // 正向边，初始容量 z，单位费用 c
    ver[++tot] = y, edge[tot] = z, cost[tot] = c;
    Next[tot] = head[x], head[x] = tot;
    // 反向边，初始容量 0，单位费用-c，与正向边“成对存储”
    ver[++tot] = x, edge[tot] = 0, cost[tot] = -c;
```

```
    Next[tot] = head[y], head[y] = tot;
}

int num(int i, int j, int k) {
    return (i - 1)*n + j + k*n*n;
}

bool spfa() {
    queue<int> q;
    memset(d, 0xcf, sizeof(d)); // -INF
    memset(v, 0, sizeof(v));
    q.push(s); d[s] = 0; v[s] = 1; // SPFA 求最长路
    incf[s] = 1 << 30; // 增广路上各边的最小剩余容量
    while (q.size()) {
        int x = q.front(); v[x] = 0; q.pop();
        for (int i = head[x]; i; i = Next[i]) {
            if (!edge[i])
            continue; // 剩余容量为0，不在残量网络中，不遍历
            int y = ver[i];
            if (d[y]<d[x] + cost[i]) {
                d[y] = d[x] + cost[i];
                incf[y] = min(incf[x], edge[i]);
                pre[y] = i; // 记录前驱，便于找到最长路的实际方案
                if (!v[y]) v[y] = 1, q.push(y);
            }
        }
    }
    if (d[t]==0xcfcfcfcf) return false; // 汇点不可达，已求出最大流
    return true;
}

// 更新最长增广路及其反向边的剩余容量
void update() {
    int x = t;
    while (x != s) {
        int i = pre[x];
        edge[i] -= incf[t];
        edge[i ^ 1] += incf[t]; // 利用“成对存储”的 xor 1 技巧
```

```
            x = ver[i ^ 1];
        }
        maxflow += incf[t];
        ans += d[t] * incf[t];
    }

    int main() {
        cin >> n >> k;
        s = 1, t = 2 * n * n;
        tot = 1; // 从2开始“成对存储”，2和3是一对，4和5是一对
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++) {
                int c; scanf("%d", &c);
                add(num(i, j, 0), num(i, j, 1), 1, c);
                add(num(i, j, 0), num(i, j, 1), k - 1, 0);
                if (j<n) add(num(i, j, 1), num(i, j + 1, 0), k, 0);
                if (i<n) add(num(i, j, 1), num(i + 1, j, 0), k, 0);
            }
        while (spfa()) update(); // 计算最大费用最大流
        cout << ans << endl;
    }
```



0x6B 总结与练习

本章内容把图论的知识点划分为 6 个板块进行了讲解：最短路、最小生成树、树结构相关的图论问题、图连通性、二分图、网络流。前 3 个板块是图论中的基础内容，后 3 个板块则在难度与综合性上都有所提升。图论的知识脉络不像 DP 那样逐层递进，而是具有广泛、分散的特点。在各式各样的图论算法和模型基础上，还存在许多变种。因为篇幅有限，无法逐一展开，读者在回顾本章知识点时，不妨尝试主动把问题进行变形，争取能够举一反三，独立思考出解决方案。

本章知识点

最短路

图的基本概念，图的邻接矩阵与邻接表存储

单源最短路径问题，Dijkstra 算法及堆优化，Bellman-Ford，SPFA 算法

单源最短路径各算法的适用范围，单源最短路径问题的变形与扩展

任意两点间最短路径问题，Floyd 算法的本质及应用

传递闭包，无向图最小环问题

最小生成树

最小生成树的定义与基本性质，Kruskal 算法，Prim 算法

最小生成树问题的变形与扩展

树的直径与最近公共祖先

树的直径的定义与计算：动态规划或两次 BFS

树的直径的性质与应用，尤其是直径的“最长性”

LCA 的定义与计算，向上标记法，树上倍增法，LCA 的 Tarjan 算法

LCA 的扩展与应用：“树上差分算法”等

树上倍增法的应用：“次小生成树”等

基环树

基环树、外向树、内向树的定义，基环树的处理方法

负环与差分约束

最短路中负环的判定方法

差分约束系统的求解，差分约束到最短路的转化

Tarjan 算法与无向图连通性

无向图的搜索树、时间戳与追溯值，割点与割边判定法则

Tarjan 算法求割点、割边、点双连通分量、边双连通分量

双连通分量的性质与应用，缩点

欧拉图的判定，欧拉路的计算

Tarjan 算法与有向图连通性

有向图的搜索树、边的分类，Tarjan 算法求强连通分量

强连通分量的性质与应用，缩点

有向无环图的必经点与必经边，路径条数取模法

2-SAT 问题的判定，自底向上拓扑排序构造方案

二分图的匹配

二分图的判定：染色法判断是否存在奇环

二分图最大匹配：匈牙利（增广路）算法

二分图匹配的模型构建方法，“0 要素”与“1 要素”，二分图多重匹配

二分图带权最大匹配：KM 算法

二分图的覆盖与独立集

二分图最小点覆盖，模型构建的“2 要素”

二分图最大独立集，“团”与独立集的对应关系，补图转化思想

有向无环图的最小路径点覆盖、最小路径可重复点覆盖

网络流初步

网络流的定义，“容量限制”“斜对称”“流量守恒”三条基本定律

最大流的 Edmonds-Karp 增广路算法与 Dinic 算法

网络流求解二分图匹配的方法，二分图最大匹配的必须边、可行边判定

最小割的定义，最大流最小割定理，“点边转化”技巧

费用流的 Edmonds-Karp 增广路算法及其应用

练习

1. 完成本章中所有例题的代码实现（标有*的题目选做）

2. Sightseeing
题目地址：AcWing383，提示：最短路/单源次短路及其条数
3. 升降梯上
题目地址：AcWing384，提示：最短路/节点扩展到二维
4. GF 和猫咪的玩具
题目地址：AcWing385，提示：最短路/任意两点间最短路
5. 社交网络
题目地址：AcWing386，提示：最短路/任意两点间最短路及其条数
6. Arctic Network
题目地址：AcWing387，提示：最小生成树
7. *四叶草魔杖
题目地址：AcWing388，提示：最小生成树/状态压缩动态规划
8. 直径
题目地址：AcWing389，提示：树的直径的必须边
9. 逃学的小孩
题目地址：AcWing390，提示：树的直径
10. 聚会
题目地址：AcWing391，提示：最近公共祖先
11. Rendezvous
题目地址：AcWing392，提示：基环树/最近公共祖先
12. Cashier Employment
题目地址：AcWing393，提示：差分约束
13. *最优高铁环
题目地址：AcWing394，提示：负环判定/01 分数规划
14. Redundant Paths
题目地址：AcWing395，提示：Tarjan 算法/桥
15. *矿场搭建
题目地址：AcWing396，提示：Tarjan 算法/割点
16. 逃不掉的路
题目地址：AcWing397，提示：边双连通分量/缩点/无向图必经边
17. Traffic Real Time Query System
题目地址：AcWing398，提示：点双连通分量/缩点/无向图必经点
18. John's trip
题目地址：AcWing399，提示：欧拉路

19. 太鼓达人

题目地址：AcWing400，提示：欧拉路

20. Going from u to v or from v to u

题目地址：AcWing401，提示：强连通分量/缩点

21. *杀人游戏

题目地址：AcWing402，提示：强连通分量/缩点

22. Planar

题目地址：AcWing403，提示：2-SAT

23. Wedding

题目地址：AcWing404，提示：2-SAT/输出方案

24. Team Them Up!

题目地址：AcWing405，提示：二分图判定/动态规划

25. Place the Robots

题目地址：AcWing406，提示：二分图最大匹配

26. Steady Cow Assignment

题目地址：AcWing407，提示：二分图多重匹配

27. Going Home

题目地址：AcWing408，提示：二分图带权匹配

28. Air Raid

题目地址：AcWing409，提示：二分图最小路径点覆盖

29. Sorting Slide

题目地址：AcWing410，提示：二分图最大匹配的必须边

30. King's Quest

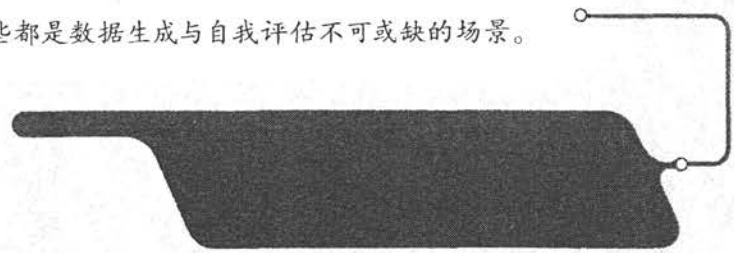
题目地址：AcWing411，提示：二分图最大匹配的可行边

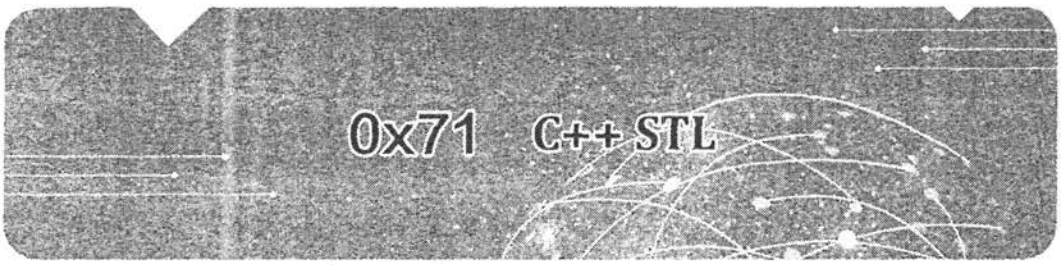
31. Drainage Ditches

题目地址：AcWing412，提示：最大流

0x70 综合技巧与实践

算法竞赛不仅是一门动脑的学问，也是一门动手的艺术。除了算法、数据结构和题目模型的理论分析之外，如何简洁、高效地编写程序，并通过恰当的调试、检测保证其正确性，也是值得探究的要点。本章的主题是“综合技巧与实践”。0x71 节介绍 C++ STL，以及在使用 STL 时需要注意的问题，主要侧重与前面七章知识点的配合，不会从 C++ 语言的专业视角讲解其实现细节。0x72 节介绍典型测试数据的随机生成，以及“对拍”等对程序正确性进行自我检测的方法。在平时解题中，读者可能长时间无法 AC，且下载不到测试数据；在一些比赛规则下，选手无法获得实时的评分反馈；又或是想到有趣的构思，希望自己出一道题，必然需要对标准程序进行检验——这些都是数据生成与自我评估不可或缺的场景。





本节介绍 STL 中 vector, queue, priority_queue, deque, set, multiset, map, bitset 八种算法竞赛中比较常用的容器。另外，我们也会介绍 algorithm 头文件中包含的部分函数。建议读者按顺序依次阅读本节的每部分内容，以免错过大部分 STL 容器共通的背景知识，造成理解困难。对于更详尽的资源，请参见 *C++ Primer* 等语言类书籍，或 cplusplus.com。

#include<vector>

vector 可理解为变长数组，它的内部实现基于倍增思想。按照下列思路可以大致实现一个 vector：设 n,m 为 vector 的实际长度和最大长度。向 vector 加入元素前，若 $n=m$，则在内存中申请 $2m$ 的连续空间，并把内容转移到新的地址上（同时释放旧的空间），再执行插入。从 vector 中删除元素后，若 $n \le m/4$，则释放一半的空间。

vector 支持**随机访问**，即对于任意的下标 $0 \le i < n$，可以像数组一样用 $[i]$ 取值。但它不是链表，不支持在任意位置 O(1) 插入。为了保证效率，元素的增删一般应该在末尾进行。

声明

```
#include<vector>          头文件
vector<int> a;            相当于一个长度动态变化的 int 数组
vector<int> b[233];       相当于第一维长 233，第二维长度动态变化的 int 数组
struct rec{…};
vector<rec> c;            自定义的结构体类型也可以保存在 vector 中
```

size/empty

size 函数返回 vector 的实际长度（包含的元素个数），empty 函数返回一个 bool 类型，表明 vector 是否为空。二者的时间复杂度都是 O(1)。

所有的 STL 容器都支持这两个方法，含义也都相同，之后我们就不再重复给出。

clear

clear 函数把 vector 清空。



迭代器

迭代器就像 STL 容器的“指针”，可以用星号“*”操作符解除引用。

一个保存 int 的 vector 的迭代器声明方法为：

```
vector<int>::iterator it;
```

vector 的迭代器是“随机访问迭代器”，可以把 vector 的迭代器与一个整数相加减，其行为和指针的移动类似。可以把 vector 的两个迭代器相减，其结果也和指针相减类似，得到两个迭代器对应下标之间的距离。

begin/end

begin 函数返回指向 vector 中第一个元素的迭代器。例如 a 是一个非空的 vector，则 *a.begin() 与 a[0] 的作用相同。

所有的容器都可以视作一个“前闭后开”的结构，end 函数返回 vector 的尾部，即第 n 个元素再往后的“边界”。*a.end() 与 a[n] 都是越界访问，其中 n = a.size()

下面两份代码都遍历了 vector<int> a，并输出它的所有元素。

```
for (int i = 0; i < a.size(); i++)
    cout << a[i] << endl;
for (vector<int>::iterator it = a.begin(); it != a.end(); it++)
    cout << *it <<endl;
```

front/back

front 函数返回 vector 的第一个元素，等价于 *a.begin() 和 a[0]。

back 函数返回 vector 的最后一个元素，等价于 *--a.end() 和 a[a.size()-1]。

push_back/pop_back

a.push_back(x) 把元素 x 插入到 vector a 的尾部。

a.pop_back() 删除 vector a 的最后一个元素。

【实例】用 vector 代替邻接表保存有向图

```
const int MAX_EDGES = 100010;
vector<int> ver[MAX_EDGES], edge[MAX_EDGES];

// 保存从 x 到 y 权值为 z 的有向边
    void add(int x, int y, int z) {
        ver[x].push_back(y);
        edge[x].push_back(z);
}
```

```
// 遍历从 x 出发的所有边
for (int i = 0; i < ver[x].size(); i++) {
    int y = ver[x][i], z = edge[x][i];
    // 有向边 (x,y,z)
}
```

#include<queue>

头文件 queue 主要包括循环队列 queue 和优先队列 priority_queue 两个容器。

声明

```
queue<int> q;
struct rec{…}; queue<rec> q;
priority_queue<int> q;
priority_queue< pair<int,int> > q;①
```

循环队列 **queue**

方法	描述	示例	时间复杂度
push	入队（从队尾）	`q.push(element);`	O(1)
pop	出队（从队头）	`q.pop();`	O(1)
front	队头元素	`int x = q.front();`	O(1)
back	队尾元素	`int y = q.back();`	O(1)

读者可以回顾 0x21 节“树与图的遍历”的末尾“图的广度优先遍历”部分，或者 0x61 节“最短路”的“SPFA 算法”部分，找到 queue 的运用实例。

优先队列 **priority_queue**

priority_queue 可理解为一个**大根二叉堆**。

方法	描述	示例	时间复杂度
push	把元素插入堆	`q.push(x);`	$O(\log n)$
pop	删除堆顶元素	`q.pop();`	$O(\log n)$
top	查询堆顶元素（最大值）	`int x = q.top();`	O(1)

① pair<>：C++内置的二元组，尖括号中分别指定二元组的第一元、第二元的类型。可以用 make_pair 函数创建二元组，用成员变量 first 访问第一元、second 访问第二元。在比较大小时，以第一元为第一关键字、第二元为第二关键字。



重载“<”运算符

priority_queue 中存储的元素类型必须定义“小于号”，较大的元素会被放在堆顶。内置的 int, string 等类型本身就可以比较大小。若使用自定义的结构体类型，则需要重载“<”运算符。

例如下面的 poi 结构体保存了二维平面上点的编号和坐标，比较大小时，先比横坐标，再比纵坐标，并且考虑了精度误差：

```
struct poi { int id; double x, y; };
const double eps = 1e-8;

bool operator <(const poi &a, const poi &b) {
    return a.x + eps < b.x || a.x < b.x + eps && a.y < b.y;
}
```

priority_queue 实现小根堆

priority_queue 实现小根二叉堆的方式一般有两种。

对于 int 等内置数值类型，可以把要插入的元素的**相反数**放入堆中。等从堆中取出元素时，再把它取相反数变回原来的元素。这样就相当于把小的放在堆顶。

例如要插入 1,2,3 三个数时，改为插入 −1,−2,−3。此时 −1 最大，处于堆顶，把 −1 取出后再变回 −(−1)，相当于取出了最小的 1。

更为通用的解决方案是，建立自定义结构体类型，重载“小于号”，但是当作“大于号”来编写函数，例如：

```
struct rec { int id; double value; };
bool operator <(const rec &a, const rec &b) {
    return a.value > b.value;
}
```

这样 priority_queue 会认为“大”的更“小”，“小”的更“大”，从而实现大根堆时，value 较小的 rec 元素会被放在堆顶。

请读者回顾 0x61 节“最短路”中的“堆优化的 Dijkstra 算法”部分，参考程序使用 priority_queue 实现了一个小根二叉堆，用于在计算单源最短路径的过程中保存二元组：与起点距离的相反数、节点编号。

懒惰删除法

读者可能已经发现，0x17 节讲解的“二叉堆”支持删除操作，但 STL 的“优先队列”却不支持删除堆中任意元素，这给我们带来了很多不便。

懒惰删除法（又称延迟删除法）就是一种应对策略。当遇到删除操作时，仅在优先

队列之外做一些特殊的记录（例如记录元素的最新值），用于辨别那些堆中尚未清除的“过时”元素。当从堆顶取出最值时，再检查“最值元素”是不是“过时”的，若是，则重新取出下一个最值。换言之，元素的“删除”被延迟到堆顶进行。

我们仍然以 0x61 节“最短路”中“堆优化的 Dijkstra 算法”的参考程序为例。

当一个节点的距离值 $dist[y]$ 被更新时，暂时不从优先队列中删除 y 对应的元素而是直接把新的二元组 $(-dist[y], y)$ 插入优先队列。

当从堆顶取出二元组 (val, x) 准备扩展时，再检查 $dist[x]$ 是否等于 val。如果不相等，那么说明 (val, x) 这个元素是被删除过的，$(dist[x], x)$ 才是最新插入的。此时直接继续循环，取出下一个堆顶。

当然，因为在无负权的图上，每个节点第一次被取出时，就已经得到了它的最短路，所以每个节点只需要扩展一次。因此，我们没有执行上述检查，而是用一个 bool 数组进行了“是否扩展过”的标记，其本质还是懒惰删除法的应用。

#include<deque>

双端队列 deque 是一个支持在两端高效插入或删除元素的连续线性存储空间。它就像是 vector 和 queue 的结合。与 vector 相比，deque 在头部增删元素仅需要 O(1) 的时间；与 queue 相比，deque 像数组一样支持随机访问。

方法	描述	示例	时间复杂度
[]	随机访问	与 vector 类似	O(1)
begin/end	deque 的头/尾迭代器	与 vector 迭代器类似	O(1)
front/back	队头/队尾元素	与 queue 类似	O(1)
push_back	从队尾入队	`q.push_back(x);`	O(1)
push_front	从队头入队	`q.push_front(y);`	O(1)
pop_front	从队头出队	`q.pop_front();`	O(1)
pop_back	从队尾出队	`q.pop_back();`	O(1)
clear	清空队列	`q.clear();`	O(n)

#include<set>

头文件 set 主要包括 set 和 multiset 两个容器，分别是“有序集合”和“有序多重集”，即前者的元素不能重复，而后者可以包含若干个相等的元素。set 和 multiset 的内部实现是一棵红黑树（平衡树的一种），它们支持的函数基本相同。



声明

```
set<int> s;
struct rec{…}; set<rec> s;
multiset<double> s;
```

与优先队列一样，set 和 multiset 存储的元素**必须定义“小于号”运算符**。

size/empty/clear

与 vector 类似，分别为元素个数、是否为空、清空。前两者的时间复杂度为 O(1)。

迭代器

set 和 multiset 的迭代器称为“双向访问迭代器”，不支持“随机访问”，支持星号(*)解除引用，仅支持“++”和“--”两个与算术相关的操作。

设 it 是一个迭代器，例如 `set<int>::iterator it;`

若把 it++，则 it 将会指向“下一个”元素。这里的“下一个”是指在元素从小到大排序的结果中，排在 it 下一名的元素。同理，若把 it--，则 it 将会指向排在“上一个”的元素。

请注意，执行“++”和“--”操作的时间复杂度都是 $O(\log n)$。执行操作前后，务必仔细检查，避免迭代器指向的位置超出首、尾迭代器之间的范围。

begin/end

返回集合的首、尾迭代器，时间复杂度为 O(1)。

s.begin() 是指向集合中最小元素的迭代器。

s.end() 是指向集合中最大元素的下一个位置的迭代器。换言之，就像 vector 一样，是一个“前闭后开”的形式。因此 --s.end() 是指向集合中最大元素的迭代器。

insert

s.insert(x) 把一个元素 x 插入到集合 s 中，时间复杂度为 $O(\log n)$。

在 set 中，若元素已存在，则不会重复插入该元素，对集合的状态无影响。

下面的代码把 n 个整数插入有序多重集 multiset，并从小到大输出，时间复杂度为 $O(n \log n)$，相当于进行了一次排序。假设 n 个整数目前存储在数组 $a[1\sim n]$ 中。

```
multiset<int> s;
for (int i = 1; i <= n; i++) s.insert(a[i]);
for (multiset<int>::iterator it = s.begin(); it != s.end(); it++)
cout << *it << endl;
```

find

s.find(x) 在集合 s 中查找等于 x 的元素，并返回指向该元素的迭代器。若不存在，则返回 s.end()。时间复杂度为 $O(\log n)$。

lower_bound/upper_bound

这两个函数的用法与 find 类似，但查找的条件略有不同，时间复杂度为 $O(\log n)$。

s.lower_bound(x) 查找 $\geq x$ 的元素中最小的一个，并返回指向该元素的迭代器。

s.upper_bound(x) 查找 $> x$ 的元素中最小的一个，并返回指向该元素的迭代器。

erase

设 it 是一个迭代器，s.erase(it) 从 s 中删除迭代器 it 指向的元素，时间复杂度为 $O(\log n)$。

设 x 是一个元素，s.erase(x) 从 s 中删除所有等于 x 的元素，时间复杂度[①]为 $O(k + \log n)$，其中 k 为被删除的元素个数。

如果想从 multiset 中删掉至多 1 个等于 x 的元素，可以执行：

```
if ((it = s.find(x)) != s.end()) s.erase(it);
```

count

s.count(x) 返回集合 s 中等于 x 的元素个数，时间复杂度[②]为 $O(k + \log n)$，其中 k 为元素 x 的个数。

#include<map>

map 容器是一个键值对 key-value 的映射。其内部实现是一棵以 key 为关键码的红黑树。map 的 key 和 value 可以是任意类型，其中 key **必须定义“小于号”运算符**，声明方法为：

```
map<key_type, value_type> name;
```

例如：

```
map<long long, bool> vis;
map<string, int> hash;
map< pair<int, int>, vector<int> > test;
```

在很多时候，map 容器被当作 Hash 表使用，建立从复杂信息 key（如字符串）到简单信息 value（如一定范围内的整数）的映射。

---

① multiset 的 erase 操作的时间复杂度由“www.cplusplus.com/reference/set/multiset/erase”给出。本书配套资源提供了该网页的快照。

② multiset 的 count 操作的时间复杂度由“www.cplusplus.com/reference/set/multiset/count”给出。本书配套资源提供了该网页的快照。



因为 map 基于平衡树实现，所以它的大部分操作的时间复杂度都在 $O(\log n)$ 级别，略慢于使用 Hash 函数实现的传统 Hash 表。从 C++11 开始，STL 中新增了 unordered_map 等基于 Hash 的容器，但部分算法竞赛并不支持 C++11 标准，这里就不再介绍这些新容器。

size/empty/clear/begin/end

与 set 类似，分别为元素个数、是否为空、清空、首迭代器、尾迭代器。

迭代器

map 的迭代器与 set 一样，也是“双向访问迭代器”。对 map 的迭代器解除引用后，将得到一个二元组 pair<key_type, value_type>。

insert/erase

与 set 类似，分别为插入、删除。insert 的参数是 pair<key_type, value_type>，erase 的参数可以是 key 或者迭代器。

```
map<int,int> h;
h.insert(make_pair(1,2)), h.insert(make_pair(2,3));
map<int,int>::iterator it = h.begin();
pair<int,int> p = *it;
h.erase(it), h.erase(2);
cout << p.first << ' ' << p.second << endl;
```

find

h.find(x) 在变量名为 h 的 map 中查找 key 为 x 的二元组，并返回指向该二元组的迭代器。若不存在，返回 h.end()。时间复杂度为 $O(\log n)$。

[] 操作符

h[key] 返回 key 映射到的 value 的引用，时间复杂度为 $O(\log n)$。

[] 操作符是 map 最吸引人的地方。我们可以很方便地通过 h[key]来得到 key 对应的 value，还可以对 h[key]进行赋值操作，改变 key 对应的 value。

需要特别注意的是，若查找的 key 不存在，则执行 h[key]后，h 会自动新建一个二元组(key,zero)，并返回 zero 的引用。这里 zero 表示一个广义“零值”，如整数 0、空字符串等。如果查找之后不对 h[key]进行赋值，那么时间一长，h 会包含很多无用的“零值二元组”，白白地占用了空间，降低了程序运行效率。强烈建议读者在用 [] 操作符查询之前，先用 find 方法检查 key 的存在性。

【实例】用 map 统计字符串出现的次数

给定 n 个字符串，m 个问题，每个问题询问一个字符串出现的次数。

$n \leq 20000$，$m \leq 20000$，每个字符串的长度都不超过 20。

```
map<string, int> h;
char str[25];
for (int i = 1; i <= n; i++) {
    scanf("%s", str); h[str]++;
}
for (int i = 1; i <= m; i++) {
    scanf("%s", str);
    if (h.find(str) == h.end()) puts("0");
    else printf("%d\n", h[str]);
}
```

#include<bitset>

bitset 可看作一个多位二进制数，每 8 位占用 1 个字节，相当于采用了状态压缩的二进制数组，并支持基本的位运算。在估算程序运行的时间时，我们一般以 32 位整数的运算次数为基准，因此 n 位 bitset 执行一次位运算的复杂度可视为 $n/32$，效率较高。

声明

```
bitset<10000> s;
```

表示一个 10000 位二进制数，<>中填写位数。下面把位数记为 n。

位运算操作符

~s：返回对 bitset s 按位取反的结果。

&, |, ^：返回对两个位数相同的 bitset 执行按位与、或、异或运算的结果。

>>, <<：返回把一个 bitset 右移、左移若干位的结果。

==, !=：比较两个 bitset 代表的二进制数是否相等。

[] 操作符

s[k] 表示 s 的第 k 位，既可以取值，也可以赋值。

在 10000 位二进制数中，最低位为 s[0]，最高位为 s[9999]。



count

s.count()[1] 返回有多少位为 1。

any/none

若 s 所有位都为 0，则 s.any()返回 false，s.none()返回 true。

若 s 至少一位为 1，则 s.any()返回 true，s.none()返回 false。

set/reset/flip

s.set() 把 s 所有位变为 1。

s.set(k, v) 把 s 的第 k 位改为 v，即 s[k] = v。

s.reset() 把 s 所有位变为 0。

s.reset(k) 把 s 的第 k 位改为 0，即 s[k] = 0。

s.flip() 把 s 的所有位取反，即 s=~s。

s.flip(k) 把 s 的第 k 位取反，即 s[k]^=1。

#include<algorithm>

下面介绍的几个函数都作用在序列上，接受两个迭代器（或指针）l, r，对下标处于前闭后开区间 $[l, r)$ 中的元素执行一系列操作。

reverse 翻转

翻转一个 vector：

```
reverse(a.begin(), a.end());
```

翻转一个数组，元素存放在下标 1~n：

```
reverse(a+1, a+n+1);
```

unique 去重

返回去重之后的尾迭代器（或指针），仍然为前闭后开，即这个尾迭代器是去重之后末尾元素的下一个位置。该函数常用于离散化，利用迭代器（或指针）的减法，可计算出去重后的元素个数 m。

把一个 vector 去重：

```
int m = unique(a.begin(), a.end()) - a.begin();
```

把一个数组去重，元素存放在下标 1~n：

```
int m = unique(a+1, a+n+1) - (a+1);
```

[1] bitset 的 count 操作的复杂度未在 C++ Reference 中标注。一般来说，其复杂度为关于 bitset 位数的线性复杂度，但实际效率可能稍快一些。

random_shuffle 随机打乱

用法与 reverse 相同。

next_permutation 下一个排列

把两个迭代器（或指针）指定的部分看作一个排列，求出这些元素构成的全排列中，字典序排在下一个的排列，并直接在序列上更新。另外，若不存在排名更靠后的排列，则返回 false，否则返回 true。同理，也有 prev_permutation 函数。

下面的程序按字典序输出 $1\sim n$ 的 $n!$ 种全排列：

```
for (int i = 1; i <= n; i++) a[i] = i;
do {
   for (int i = 1; i < n; i++) cout << a[i] << ' ';
   cout << a[n] << endl;
} while (next_permutation(a+1, a+n+1));
```

sort 快速排序

对两个迭代器（或指针）指定的部分进行快速排序。可以在第三个参数传入定义大小比较的函数，或者重载“小于号”运算符。

把一个 int 数组（元素存放在下标 $1\sim n$）从大到小排序，传入比较函数：

```
int a[MAX_SIZE];
bool cmp(int a, int b) { return a > b; }
sort(a+1, a+n+1, cmp);
```

把自定义的结构体 vector 排序，重载“小于号”运算符：

```
struct rec{ int id, x, y; };
vector<rec> a;
bool operator <(const rec &a, const rec &b) {
   return a.x < b.x || a.x == b.x && a.y < b.y;
}
sort(a.begin(), a.end());
```

lower_bound/upper_bound 二分

lower_bound 的第三个参数传入一个元素 x，在两个迭代器（或指针）指定的部分上执行二分查找，返回指向第一个**大于等于** x 的元素的位置的迭代器（或指针）。

upper_bound 的用法和 lower_bound 大致相同，唯一的区别是查找第一个**大于** x 的元素。当然，两个迭代器（或指针）指定的部分应该是**提前排好序**的。



在有序 int 数组（元素存放在下标 1~n）中查找大于等于 x 的最小整数的下标：

```
int i = lower_bound(a+1, a+n+1, x) - a;
```

在有序 vector<int>中查找小于等于 x 的最大整数（假设一定存在）：

```
int y = *--upper_bound(a.begin(), a.end(), x);
```

0x72 随机数据生成与对拍

本节介绍随机数据的生成方法与对拍测试方法。读者将学习使用 C++ 随机数产生器，根据题目要求构造各种规模的输入数据，用于对自己编写的程序进行检测。同时，读者也将学习编写简单的脚本，自动化、批量化运行“数据生成程序”和两份不同的“问题求解程序”，并对两份程序的输出结果进行比对——我们把这种过程称为“对拍”。

随机数据生成与对拍可用于以下场景。

1. 在无法获得实时评测反馈的比赛中，思考并实现了一个“高分解法”，但实在不会证明自己的结论，或者不能确保自己编写的程序是否完全正确。

这种情况下，建议读者酌情分配一些时间，额外编写一份随机数据生成程序、一份用朴素算法求解的程序（通常朴素解法时间复杂度高，但实现简单，不易出错）。然后把“高分解法”与“朴素解法”进行对拍，看二者的输出结果是否始终保持一致。

2. 在平时解题时，自己编写的程序无法在 Online Judge 上取得 Accepted 结果，调试很久仍未发现错误，并且不能下载到题目的测试数据，或者虽然能下载到测试数据，但发生错误的数据规模过大，不容易进行调试。

这时，可以编写一个随机数据生成程序，再编写一个使用朴素算法的程序（或者直接在网络上搜索其他人的 AC 程序），与自己的“错误解法”对拍。我们可以适当调整随机数据的规模，控制在易于人工演算和调试的范围内。虽然数据越小，出错概率越低，但是“对拍”脚本能够批量化执行，在成千上万次检测中，一般总能找到一个造成错误的小规模数据。

3. 有一个不错的构思，自己出了一道题目。

此时当然需要生成一些测试数据，并且需要用“对拍”来检测自己编写的“标准程序”的正确性。不过，除了随机数据外，通常还需要增加一些特殊构造的数据，保证测试数据的全面性。

随机数据生成

头文件 cstdlib (stdlib.h) 包含 rand 和 srand 两个函数，以及 RAND_MAX 常量。RAND_MAX 是一个不小于 32767 的整数常量，它的值与操作系统环境、编译器环

境有关。一般来说，在 Windows 系统中为 32767，在类 Unix 系统中为 2147483647。

rand() 函数返回一个 0 ~ RAND_MAX 之间的随机整数 int。

srand(seed) 函数接受 unsigned int 类型的参数 seed，以 seed 为“随机种子”。rand 函数基于线性同余递推式生成随机数，“随机种子”相当于计算线性同余时的一个初始参数，感兴趣的读者可以查阅相关资料。若不执行 srand 函数，则种子默认为 1。

当种子确定后，接下来产生的随机数列就是固定的，所以这种随机方法也被称为“伪随机”。因此，一般在随机数据生成程序 main 函数的开头，用当前系统时间作为随机种子。

头文件 ctime (time.h) 包含 time 函数，调用 time(0) 可以返回从 1970 年 1 月 1 日 0 时 0 分 0 秒（Unix 纪元）到现在的秒数。执行 srand((unsigned)time(0)) 即可初始化随机种子。

下面的程序可作为随机数据生成器的模板，函数 random(n) 返回 $0 \sim n-1$ 之间的随机整数，并综合考虑了操作系统和编译器环境的差异，对 int 范围内的 n 均能正常工作。

```
#include<cstdlib>
#include<ctime>
int random(int n) {
    return (long long)rand() * rand() % n;
}
int main() {
    srand((unsigned)time(0));
    // …具体内容…
}
```

若要产生随机实数，则可以先产生一个较大的随机整数，再用它除以 10 的次幂。若要同时产生负数，则可以先产生一个 $0 \sim 2n$ 之间的随机整数，再减去 n，就得到了 $-n \sim n$ 之间的随机整数。

【实例】随机生成整数序列

在本节的模板基础上，随机生成 $n \leq 10^5$ 个绝对值在 10^9 之内的整数。

```
int n = random(100000) + 1;
int m = 1000000000;
for (int i = 1; i <= n; i++) {
    a[i] = random(2 * m + 1) - m;
}
```

【实例】随机生成区间列

在本节的模板基础上，随机生成 m 个 $[1,n]$ 的子区间，这些区间可作为数据结构题目的操作序列。

```
for (int i = 1; i <= m; i++) {
    int l = random(n) + 1;
    int r = random(n) + 1;
    if (l > r) swap(l, r);
    printf("%d %d\n", l, r);
}
```

【实例】随机生成树

在本节的模板基础上，随机生成一棵 n 个点的树，用 n 个点 $n-1$ 条边的无向图的形式输出，每条边附带一个 10^9 以内的正整数权值。

```
for (int i = 2; i <= n; i++) {
    // 从 2~n 之间的每个点 i 向 1~i-1 之间的点随机连一条边
    int fa = random(i - 1) + 1;
    int val = random(1000000000) + 1;
    printf("%d %d %d\n", fa, i, val);
}
```

【实例】随机生成图

在本节的模板基础上，随机生成一张 n 个点 m 条边的无向图，图中不存在重边、自环，且必须连通，保证 $5 \le n \le m \le n*(n-1)/4 \le 10^6$。[①]

```
pair<int, int> e[1000005]; // 保存数据
map< pair<int, int>, bool > h; // 防止重边
// 先生成一棵树，保证连通
for (int i = 1; i < n; i++) {
    int fa = random(i) + 1;
    e[i] = make_pair(fa, i + 1);
    h[e[i]] = h[make_pair(i + 1, fa)] = 1;
}
```

① 当 $n<5$ 时，可能存在边界情况，需要在参考程序中增加特判。当 $m>n*(n-1)/4$ 时，可能造成参考程序的随机效率降低，此时应当考虑随机生成补图，再转化到原图。

```
// 再生成剩余的 m-n+1 条边
for (int i = n; i <= m; i++) {
    int x, y;
    do {
        x = random(n) + 1, y = random(n) + 1;
    } while (x == y || h[make_pair(x, y)]);
    e[i] = make_pair(x, y);
    h[e[i]] = h[make_pair(y, x)] = 1;
}
// 随机打乱，输出
random_shuffle(e + 1, e + m + 1);
for (int i = 1; i <= m; i++)
    printf("%d %d\n", e[i].first, e[i].second);
```

在树、图结构中，有三类特殊数据常用于对程序进行极端状况下的效率测试。

1. 链形数据——有很长的直径。

就是把 N 个节点用 $N-1$ 条边连成一条长度为 $N-1$ 的“链”。这种数据会造成很大的递归深度，也是点分治等算法需要特别注意的数据。

2. 菊花形数据——有度数很大的节点。

以 1 号节点为中心，2~N 号节点都用一条边与 1 号节点相连，最终 1 号节点的度数为 $N-1$。这种数据画出来形似一朵“菊花”，缩点等图论算法若处理不当，复杂度容易在这种数据上退化。

3. 蒲公英形数据。

即链形和菊花形数据的综合。令树的一部分构成链，一部分构成菊花，再把两部分连接。

在以上三种数据的基础上，再添加少量随机的边，即可得到一张既包含局部特殊结构、又不失一般性和多样性的图。请读者自己尝试生成相关的数据。

对拍

假设我们已经编写好了三个程序。

1. 自己编写的“正解”，即准备提交评测的程序，名为 sol.cpp[①]。

该程序从文件 data.in 中读取输入数据，并输出答案到文件 data.out 中。

2. 自己编写的“朴素解法”程序，名为 bf.cpp[②]。

① sol: solution 的缩写。

② bf: brute force 的缩写，直译为“暴力”，即朴素解法。

该程序从文件 data.in 中读取输入数据，并输出答案到文件 data.ans 中。

3. 自己编写的随机数据生成程序，名为 random.cpp。

该程序输出随机数据到文件 data.in 中。

依次编译这三个程序，得到三个可执行文件，例如在 Windows 系统下得到 sol.exe, bf.exe 和 random.exe。

现在，我们需要编写一个脚本，循环执行以下过程。

1. 运行随机数据生成器 random。

2. 运行“正解”程序 sol。

3. 运行“朴素解法”程序 bf。

4. 比对文件 data.out 和 data.ans 的内容是否一致。

Windows 和类 Unix 系统分别有 bat 批处理脚本和 bash 脚本。不过，为了避免介绍一门新的脚本语言，这里就用读者都熟悉的 C++语言来编写“对拍”程序。

头文件 cstdlib (stdlib.h)中提供了一个函数 system，它接受一个字符串参数，并把该字符串作为系统命令执行。例如代码 `system("C:\\random.exe")` 执行 C 盘根目录下的 random.exe 文件。

Windows 系统命令 fc 或类 Unix 系统命令 diff 可以执行文件比对的工作，它们接受两个文件路径，比较二者是否一致。若一致，返回 0，否则返回非零值。

Windows 系统对拍程序

```
#include<cstdlib>
#include<cstdio>
#include<ctime>
int main() {
    for (int T = 1; T <= 10000; T++) {
        // 自行设定适当的路径
        system("C:\\random.exe");
        // 当前程序已经运行的 CPU 时间，Windows 下单位 ms，Unix 下单位 s
        double st = clock();
        system("C:\\sol.exe");
        double ed = clock();
        system("C:\\bf.exe");
        if (system("fc C:\\data.out C:\\data.ans")) {
            puts("Wrong Answer");
            // 程序立即退出，此时 data.in 即为发生错误的数据
            return 0;
        }
```



```
        else {
            printf("Accepted, 测试点 #%d, 用时 %.0lfms\n",T,ed-st)
        }
    }
}
```

编译运行该 C++程序即可开始“对拍”过程。

类 Unix 系统对拍程序

在 Windows 系统对拍程序的基础上，更改 system 中的路径格式，并把 fc 命令改为 diff 命令，用时单位改为“秒”。

到此为止，全书计划讲解的内容就告一段落了。希望读者能够理论联系实践，在充分理解所学知识的同时，认真完成习题，注重思维模型的构建与代码能力的提升，善用数据生成与对拍技巧，独立克服解题困难，并发掘适合自身的比赛策略，最终在算法竞赛带来的挑战中收获乐趣和成就。



附录一 在线资源索引

资源名称	资源概述
GitHub Tedukuri	https://github.com/lydrainbowcat/tedukuri 本书勘误发布、数据下载，相关内容随时更新 http://www.dxeke.com/jiaofucat/sfjj/ 配套资源的 RAR5 格式压缩包，根据 tedukuri 累积性更新
AcWing	https://www.acwing.com/problem/ 检索“算法竞赛进阶指南” 翻译并收录了本书全部题目，补充了缺失的测试数据，包含题解和讨论
CH / Contest Hunter	http://www.contest-hunter.org 或 http://noi-test.zzstep.com Contest Hunter 信息学自助比赛平台 http://noi-test.zzstep.com/contest?type=1 《算法竞赛进阶指南》专用题库
POJ	http://poj.org 北京大学 ACM-ICPC 在线评测系统
CF / Codeforces	http://www.codeforces.com Codeforces 比赛平台
HDOJ	http://acm.hdu.edu.cn 杭州电子科技大学 ACM-ICPC 在线评测系统



附录二 全书例题与习题索引[①]

题目名称	章节	提交地址	来源或版权所有者	标程	数据
a^b	0x01	CH0101		√	√
Raising Modulo Numbers	0x01	POJ1995	CTU Open 1999	√	√
64 位整数乘法	0x01	CH0102		√	√
最短 Hamilton 路径	0x01	CH0103		√	√
起床困难综合症	0x01	CH0104	CCF NOI 2014	√	√
递归实现指数型枚举	0x02	CH0301		√	√
递归实现组合型枚举	0x02	CH0302		√	√
递归实现排列型枚举	0x02	CH0303		√	√
费解的开关	0x02	CH0201	Matrix67	√	√
Strange Towers of Hanoi	0x02	POJ1958 AcWing96	TUD 2002, Germany	√	√
Sumdiv	0x02	POJ1845	Romania OI 2002	√	√
Fractal Streets	0x02	POJ3889 AcWing98	杜宇飞，Nescafé 13 杯 NOIP 模拟赛	√	√
非递归实现组合型枚举	0x02	CH0302		√	√
激光炸弹	0x03	AcWing99	NOI 2003 湖南省选	√	√
IncDec Sequence	0x03	CH0304	毕克，Nescafé 22 杯 NOIP 模拟赛	√	√
Tallest Cow	0x03	POJ3263	USACO 2007 Jan.	√	√
Best Cow Fences	0x04	POJ2018	USACO 2003 Mar.	√	√
Innovative Business	0x04	AcWing113	SPb NRU ITMO	√	√
Cinema	0x05	CF670C AcWing103	Codeforces Round #350 (Div. 2)	√	√
货仓选址	0x05	CH0501		√	√

① 本书大量的例题、习题可视作参考文献，引用自网络资源。一般情况下，以本书所采用的题面的出处为准，标注直接来源。对于部分题目，本书已明确其原始出处，则标注原作个人、单位或竞赛。另外一些题目为经典题目或本书撰稿过程中的直接创作，未进行标注。本书无法保证标注完全正确，若您对标注有异议，且能提供相关资料证明题目的著作权属，请联系作者于再版时更正。

续表

题目名称	章节	提交地址	来源或版权所有者	标程	数据
七夕祭	0x05	CH0502	Nescafé 18 杯 NOIP 模拟赛	√	√
Running Median	0x05	POJ3784	ICPC New York 2009	√	√
Ultra-QuickSort	0x05	POJ2299	ICPC Waterloo 2005	√	√
奇数码问题	0x05	CH0503	李煜东，北京大学数据结构与算法考试	√	√
Genius ACM	0x06	CH0601	沈洋，ACM-ICPC Beijing 2016	√	√
Sunscreen	0x07	POJ3614	USACO 2007 Nov.	√	√
Stall Reservations	0x07	POJ3190	USACO 2006 Feb.	√	√
Radar Installation	0x07	POJ1328	ICPC Beijing 2002	√	√
国王游戏	0x07	CH0701	CCF NOIP 2012	√	√
Color a Tree	0x07	POJ2054	ICPC Beijing 2004	√	√
The Pilots Brothers' Refrigerator	0x08	POJ2965 AcWing116	ICPC NEERC 2004 West Subregion	√	√
占卜 DIY	0x08	CH0802	lydrainbowcat	√	√
Fractal	0x08	POJ2083	ICPC Shanghai 2004	√	√
Raid	0x08	POJ3714	北京大学	√	√
防线	0x08	CH0805	北京大学	√	√
Corral the Cows	0x08	POJ3179	USACO 2006 Jan.	√	√
糖果传递	0x08	CH0807	NOI 2008 河南省选	√	√
Soldiers	0x08	POJ1723	ICPC CERC 1998	√	√
Number Base Conversion	0x08	POJ1220	ICPC New York 2002	√	√
Cow Acrobats	0x08	POJ3045	USACO 2005 Nov.	√	√
To the Max	0x08	POJ1050	ICPC New York 2001	√	√
Task	0x08	HDOJ4864	Multi-Univ. Training	√	√
Push, Pop, GetMin	0x11	AcWing41			
Editor	0x11	HDOJ4699	Multi-Univ. Training	√	√
进出栈序列问题	0x11	CH1101 CH1102		√	√

续表

题目名称	章节	提交地址	来源或版权所有者	标程	数据
Largest Rectangle in a Histogram	0x11	POJ2559 AcWing131	ACM Ulm Local 2003	√	√
Team Queue	0x12	POJ2259	ACM Ulm Local 1998	√	√
蚯蚓	0x12	CH1202	CCF NOIP 2016	√	√
* *双端队列*	0x12	AcWing134	NOI 2011 北京市选	√	√
最大子序和	0x12	CH1201		√	√
邻值查找	0x13	CH1301		√	√
Snowflake Snow Snowflakes	0x14	POJ3349 AcWing137	Canada CCC 2007	√	√
兔子与兔子	0x14	CH1401	罗翔宇，北京大学数据结构与算法考试	√	√
Palindrome	0x14	POJ3974	ACM Egyptian 7th	√	√
后缀数组	0x14	CH1402		√	√
Period	0x15	POJ1961	ICPC SEERC 2004	√	√
前缀统计	0x16	CH1601		√	√
The XOR Largest Pair	0x16	CH1602		√	√
The XOR Longest Path	0x16	POJ3764 AcWing144	北京大学	√	√
Supermarket	0x17	POJ1456	ICPC SEERC 2003	√	√
Sequence	0x17	POJ2442	北京大学	√	√
数据备份	0x17	AcWing147	CCF CTSC 2007	√	√
合并果子	0x17	CH1701	CCF NOIP 2004	√	√
荷马史诗	0x17	AcWing149	CCF NOI 2015	√	√
括号画家	0x18	CH1801 AcWing150	李煜东，北京大学数据结构与算法考试	√	√
表达式计算 4	0x18	CH1802		√	√
City Game	0x18	CH1803	ICPC SEERC 2004	√	√
双栈排序	0x18	CH1813	CCF NOIP 2008	√	√
Sliding Window	0x18	POJ2823	北京大学	√	√
内存分配	0x18	POJ1193	CCF NOI 1999	√	√
Matrix	0x18	CH1806	NOI 2011 北京市选	√	√

续表

题目名称	章节	提交地址	来源或版权所有者	标程	数据
Subway Tree Systems	0x18	POJ1635 AcWing157	ICPC NWERC 2003	√	√
Necklace	0x18	CH1807	郭华阳，NOIP2008 模拟试题	√	√
Milking Grid	0x18	POJ2185	USACO 2003 Fall	√	√
匹配统计	0x18	CH1809		√	√
Phone List	0x18	POJ3630	ACM Nordic 2007	√	√
Black Box	0x18	POJ1442	ICPC NEERC 1996	√	√
生日礼物	0x18	CH1812	北京大学	√	√
可达性统计	0x21	CH2101		√	√
小猫爬山	0x22	CH2201	Nescafé 26 杯 NOIP 模拟赛	√	√
Sudoku	0x22	POJ2676 POJ3074	ICPC SEERC 2005 Stanford Local 2006	√	√
Sticks	0x23	POJ1011	ICPC CERC 1995	√	√
生日蛋糕	0x23	POJ1190	CCF NOI 1999	√	√
* *Sudoku*	0x23	POJ3076	ICPC SEERC 2006	√	√
Addition Chains	0x24	POJ2248	ACM Ulm Local 1997	√	√
送礼物	0x24	CH2401		√	√
Bloxorz	0x25	POJ3322	北京大学	√	√
矩阵距离	0x25	CH2501		√	√
Pushing Boxes	0x25	POJ1475	ICPC SWERC 1997	√	√
电路维修	0x26	CH2601	杜宇飞，Nescafé 5 杯 NOIP 模拟赛	√	√
Full Tank	0x26	POJ3635	ACM Nordic 2007	√	√
Nightmare II	0x26	HDOJ3085 AcWing177	HDOJ	√	√
第 K 短路 Remmarguts' Date	0x27	POJ2449	北京大学	√	√
八数码 Eight	0x27	POJ1077	ACM South Central USA 1998	√	√

续表

题目名称	章节	提交地址	来源或版权所有者	标程	数据
Booksort	0x28	POJ3460	ACM BAPC 2006 QI.	√	√
The Rotation Game	0x28	POJ2286	ICPC Shanghai 2004	√	√
* *Square Destroyer*	0x28	POJ1084	ICPC Taejon 2001	√	√
靶形数独	0x29	CH2901	CCF NOIP 2009	√	√
虫食算	0x29	CH2902	CCF NOIP 2004	√	√
Mayan 游戏	0x29	CH2903	CCF NOIP 2011	√	√
* *The Buses*	0x29	POJ1167	IOI 1994	√	√
* *Missile Defending System*	0x29	POJ3700	北京大学	√	√
武士风度的牛	0x29	CH2906	USACO 2009 Nov.	√	√
乳草的入侵	0x29	CH2907	USACO 2009 Oct.	√	√
字串变换	0x29	CH2908	CCF NOIP 2002	√	√
Weather Forecast	0x29	POJ2044	ICPC Japan 2003	√	√
* *Bloxorz II*	0x29	POJ3323	杜宇飞，Nescafé 15 杯 NOIP 模拟赛	√	√
Power Hungry Cow	0x29	POJ1945	USACO 2002 Feb.	√	√
Flood-it!	0x29	POJ4007 CH2912	ICPC Fuzhou 2011, Nescafé 21 杯模拟赛	√	√
* *骑士精神*	0x29	AcWing185	NOI 2005 四川省选	√	√
Prime Distance	0x31	POJ2689	ICPC Waterloo 1998	√	√
阶乘分解	0x31	CH3101		√	√
反素数	0x32	AcWing198	NOI 2007 河南省选	√	√
余数之和	0x32	AcWing199	NOI 2007 重庆市选	√	√
Hankson 的趣味题	0x32	CH3201	CCF NOIP 2009	√	√
Visible Lattice Points	0x32	POJ3090	ICPC New York 2006	√	√
The Luckiest Number	0x33	POJ3696	ICPC Hefei 2008	√	√
同余方程	0x33	CH3301	CCF NOIP 2012	√	√
Strange Way to Express Integers	0x33	POJ2891	北京大学	√	√
Fibonacci	0x34	POJ3070	Stanford Local 2006	√	√
石头游戏	0x34	CH3401		√	√

续表

题目名称	章节	提交地址	来源或版权所有者	标程	数据
球形空间产生器	0x35	AcWing207	NOI 2008 江苏省选	√	√
开关问题	0x35	POJ1830	北京大学	√	√
装备购买	0x35	AcWing209	王悦铭，NOI 2015 吉林省选	√	√
XOR	0x35	HDOJ3949 AcWing210	Multi-Univ. Training	√	√
计算系数	0x36	CH3601	CCF NOIP 2011	√	√
Counting Swaps	0x36	CH3602	Internet Problem Solving Contest 2016	√	√
古代猪文	0x36	AcWing213	NOI 2010 山东省选	√	√
Devu and Flowers	0x37	CF451E AcWing214	Codeforces Round #258 (Div. 2)	√	√
Zap	0x37	AcWing215	Poland OI 2007	√	√
Rainbow 的信号	0x38	CH3801	Nescafé 26 杯 NOIP 模拟赛	√	√
绿豆蛙的归宿	0x38	CH3802	Nescafé 19 杯 NOIP 模拟赛	√	√
扑克牌	0x38	CH3803	李煜东，NOIP2012 模拟试题	√	√
Cutting Game	0x3A	POJ2311	北京大学	√	√
GCD	0x3B	AcWing220	NOI 湖北省队互测	√	√
Longge's problem	0x3B	POJ2480	北京大学	√	√
青蛙的约会	0x3B	POJ1061		√	√
Xiao 9*大战朱最学	0x3B	CH3B04		√	√
计算器	0x3B	AcWing224	NOI 2011 山东省选	√	√
Matrix Power Series	0x3B	POJ3233	北京大学	√	√
233 Matrix	0x3B	HDOJ5015 AcWing226	ICPC Xi'an 2014 Pre.	√	√
Widget Factory	0x3B	POJ2947	ICPC CERC 2005	√	√
XOR	0x3B	AcWing228	NOI 2011 冬令营	√	√
新 NIM 游戏	0x3B	AcWing229	NOI 2013 重庆市选	√	√

续表

题目名称	章节	提交地址	来源或版权所有者	标程	数据
排列计数	0x3B	AcWing230	NOI 2016 山东省选	√	√
Sky Code	0x3B	POJ3904	ICPC SEERC 2008	√	√
守卫者的挑战	0x3B	CH3B13	Nescafé 17 杯 NOIP 模拟赛	√	√
换教室	0x3B	CH3B14	CCF NOIP 2016	√	√
Dropping tests	0x3B	POJ2976	Stanford Local 2005	√	√
魔法珠	0x3B	CH3B16	李煜东，CoderSpace Challenge #2	√	√
Georgia and Bob	0x3B	POJ1704	北京大学	√	√
程序自动分析	0x41	AcWing237	CCF NOI 2015	√	√
银河英雄传说	0x41	CH4101	CCF NOI 2002	√	√
Parity game	0x41	POJ1733	Central Eu. OI 1999	√	√
食物链	0x41	POJ1182	CCF NOI 2001	√	√
楼兰图腾	0x42	CH4201	张浩千	√	√
A Tiny Problem with Integers	0x42	AcWing242			
A Simple Problem with Integers	0x42	POJ3468	北京大学	√	√
Lost Cows	0x42	POJ2182	USACO 2003 Open	√	√
Can you answer on these queries III	0x43	CH4301	金斌	√	√
Interval GCD	0x43	CH4302		√	√
Atlantis	0x43	POJ1151	ICPC Mid-Central European 2000	√	√
Stars in Your Window	0x43	POJ2482	北京大学	√	√
蒲公英	0x44	CH4401	罗翔宇、杜宇飞，Violet 6 杯省选模拟赛	√	√
磁力块	0x44	CH#46A	李煜东，Adera 8.5 杯 NOI 模拟赛	√	√
小 Z 的袜子	0x44	AcWing251	莫涛	√	√
Tree	0x45	POJ1741	楼天城	√	√

续表

题目名称	章节	提交地址	来源或版权所有者	标程	数据
普通平衡树	0x46	CH4601	张浩千	√	√
天使玩偶	0x47	CH4701	罗翔宇、杜宇飞，Violet 3 杯省选模拟赛	√	√
K-th Number	0x47	POJ2104	ICPC NEERC 2004 North Subregion	√	√
最大异或和	0x48	AcWing256	大视野在线测评	√	√
关押罪犯	0x49	CH4901	CCF NOIP 2010	√	√
Rochambeau	0x49	POJ2912	百度之星 2006 预赛	√	√
True Liars	0x49	POJ1417	ICPC Japan 2002	√	√
Buy Tickets	0x49	POJ2828	北京大学	√	√
Hotel	0x49	POJ3667	USACO 2008 Feb.	√	√
Picture	0x49	POJ1177	IOI 1998	√	√
* *作诗*	0x49	CH4907	李煜东	√	√
Race	0x49	CH4908	IOI 2011	√	√
营业额统计	0x49	AcWing265	NOI 2002 湖南省选	√	√
* *SuperMemo*	0x49	POJ3580	姚金宇，北京大学	√	√
Mokia	0x49	AcWing267	Balkan OI 2007	√	√
* *Meteors*	0x49	CH4912	Poland OI 2011	√	√
* *Fotile 模拟赛 L*	0x49	AcWing269	黄嘉泰	√	√
* *可持久化并查集加强版*	0x49	AcWing270	大视野在线测评	√	√
Mr. Young's Picture Permutations	0x51	POJ2279	ICPC New York 2004	√	√
LCIS 最长公共上升子序列	0x51	CH5101		√	√
Making the Grade	0x51	POJ3666	USACO 2008 Feb.	√	√
Mobile Service	0x51	CH5102		√	√
传纸条	0x51	CH5103	CCF NOIP 2008	√	√
* *I-country*	0x51	CH5104	Saratov State Univ.	√	√

续表

题目名称	章节	提交地址	来源或版权所有者	标程	数据
Cookies	0x51	CH5105	SPb NRU ITMO 圣彼得堡国立信息技术机械与光学大学	√	√
数字组合	0x52	CH5201		√	√
自然数拆分 Lunatic 版	0x52	CH5202		√	√
Jury Compromise	0x52	POJ1015	ICPC SWERC 1996	√	√
Coins	0x52	POJ1742	楼天城	√	√
石子合并	0x53	CH5301	CCF NOI 1995	√	√
Polygon	0x53	POJ1179	IOI 1998	√	√
金字塔	0x53	CH5302	杜宇飞，Nescafé 12 杯 NOIP 模拟赛	√	√
没有上司的舞会	0x54	CH5401		√	√
选课	0x54	CH5402		√	√
Accumulation Degree	0x54	POJ3585	AIPC South Central China 2008	√	√
Naptime	0x55	POJ2228	USACO 2005 Jan., Nescafé 22 杯模拟赛	√	√
环路运输	0x55	CH5501		√	√
* *Broken Robot*	0x55	CF24D AcWing290	Codeforces Beta Round #24	√	√
Mondriaan's Dream	0x56	POJ2411	ACM Ulm Local 2000	√	√
* *炮兵阵地*	0x56	POJ1185	CCF NOI 2001	√	√
宝藏	0x56	CH5603	CCF NOIP 2017	√	√
开车旅行	0x57	CH5701	CCF NOIP 2012	√	√
Count The Repetitions	0x57	CH5702	LeetCode	√	√
Cleaning Shifts	0x58	POJ2376 POJ3171	USACO 2004 Dec. USACO 2005 Dec.	√	√
The Battle of Chibi	0x58	HDOJ5542 AcWing297	CCPC 2015	√	√
Fence	0x59	POJ1821	Romania OI 2002	√	√

续表

题目名称	章节	提交地址	来源或版权所有者	标程	数据
* *Cut the Sequence*	0x59	POJ3017	北京大学	√	√
任务安排 1	0x5A	CH5A01		√	√
任务安排 2	0x5A	POJ1180	IOI 2002	√	√
* *任务安排 3*	0x5A	AcWing302	NOI 2012 山东省选	√	√
* *任务安排 4*	0x5A				
Cats Transport	0x5A	CF311B AcWing303	Codeforces Round #185 (Div. 1)	√	√
* *诗人小 G*	0x5B	AcWing304	CCF NOI 2009	√	√
再探石子合并	0x5B	POJ1738	楼天城	√	√
Gerald and Giant Chess	0x5C	CF559C AcWing306	Codeforces Round #313 (Div. 1)	√	√
Connected Graph	0x5C	POJ1737	楼天城	√	√
* *How many of them?*	0x5C	CH5C01	Gennady Korotkevich		
A Decorative Fence	0x5C	POJ1037	Central Eu. OI 2002	√	√
启示录	0x5D	POJ3208	北京大学， Nescafé 12 杯模拟赛	√	√
月之谜	0x5D	AcWing311	NOI 2009 安徽省选， Nescafé 2 杯模拟赛	√	√
乌龟棋	0x5E	CH5E01	CCF NOIP 2010	√	√
花店橱窗	0x5E	CH5E02	IOI 1999	√	√
Buy Low Buy Lower	0x5E	POJ1952	USACO 2002 Feb.	√	√
Trip	0x5E	POJ1934	Central Eu. OI 2003	√	√
Substract	0x5E	POJ1722	Central Eu. OI 1998	√	√
划分大理石	0x5E	CH5E07		√	√
Folding	0x5E	POJ2176	ICPC NEERC 2002	√	√
能量项链	0x5E	CH5E09	CCF NOIP 2006	√	√
棋盘分割	0x5E	POJ1191	CCF NOI 1999	√	√
* *Blocks*	0x5E	POJ1390	刘汝佳	√	√
Strategic game	0x5E	POJ1463	ICPC SEERC 2000	√	√
Bribing FIPA	0x5E	POJ3345	ICPC Tehran 2006	√	√

续表

题目名称	章节	提交地址	来源或版权所有者	标程	数据
Computer	0x5E	HDOJ2196 AcWing325		√	√
XOR 和路径	0x5E	AcWing326	NOI 2011 湖南省选	√	√
Island and Bridges	0x5E	POJ2288	ICPC Shanghai 2004	√	
Corn Fields	0x5E	POJ3254	USACO 2006 Nov.	√	√
* *Bugs Integrated Inc*	0x5E	POJ1038	Central Eu. OI 2002	√	√
Fence Obstacle Course	0x5E	POJ2374	USACO 2004 Dec.	√	√
Estimation	0x5E	HDOJ4261 AcWing330	The Univ. of Chicago	√	√
干草堆	0x5E	AcWing331	USACO 2009 Open	√	√
股票交易	0x5E	AcWing332	NOI 2010 四川省选	√	√
Largest Submatrix	0x5E	HDOJ2870 AcWing333	Multi-Univ. Training	√	√
K-Anonymous Sequence	0x5E	POJ3709	北京大学	√	√
* *特别行动队*	0x5E	AcWing335	APIO 2010	√	√
Post Office	0x5E	POJ1160	IOI 2000	√	√
陨石的秘密	0x5E	POJ1187	CCF NOI 2001	√	√
* *扑克牌*	0x5E	CH5E26	微软编程之美 2015 初赛第二场	√	√
The Counting Problem	0x5E	POJ2282	ICPC Shanghai 2004	√	√
Round Numbers	0x5E	POJ3252	USACO 2006 Nov.	√	√
Telephone Lines	0x61	POJ3662	USACO 2008 Jan.	√	√
最优贸易	0x61	CH6101	CCF NOIP 2009	√	√
道路与航线	0x61	AcWing342	USACO 2011 Jan.	√	√
Sorting It All Out	0x61	POJ1094	ICPC East Central North America 2001	√	√
Sightseeing trip	0x61	POJ1734	Central Eu. OI 1999	√	√
Cow Relays	0x61	POJ3613	USACO 2007 Nov.	√	√

续表

题目名称	章节	提交地址	来源或版权所有者	标程	数据
走廊泼水节	0x62	CH6201		√	√
Picnic Planning	0x62	POJ1639	ICPC East Central North America 2000	√	√
最优比率生成树	0x62	POJ2728	ICPC Beijing 2005	√	√
黑暗城堡	0x62	CH6202	杜宇飞，Nescafé 2 杯 NOIP 模拟赛	√	√
巡逻	0x63	AcWing350	APIO 2010	√	√
树网的核	0x63	AcWing351	CCF NOIP 2007	√	√
闇の連鎖	0x63	POJ3417	杜宇飞，Nescafé 14 杯 NOIP 模拟赛	√	√
* *雨天的尾巴*	0x63	CH6302	罗翔宇，Vani 有约会杯邀请赛	√	√
天天爱跑步	0x63	CH6303	CCF NOIP 2016	√	√
异象石	0x63	CH#56C	李煜东，Adera 9 杯 NOI 模拟赛	√	√
次小生成树	0x63	AcWing356	Beijing 2010	√	√
疫情控制	0x63	CH6301	CCF NOIP 2012	√	√
岛屿	0x64	AcWing358	IOI 2008	√	√
创世纪	0x64	CH6401	杜宇飞，Nescafé 8 杯 NOIP 模拟赛	√	√
Freda 的传呼机	0x64	CH6402	Nescafé 26 杯 NOIP 模拟赛	√	√
Sightseeing Cows	0x65	POJ3621	USACO 2007 Dec.	√	√
Intervals	0x65	POJ1201	ICPC SWERC 2002	√	√
BLO	0x66	AcWing363	Poland OI 2008	√	√
Network	0x66	POJ3694	ICPC Hefei 2008 Pre.	√	√
Knights of the Round Table	0x66	POJ2942	ICPC CERC 2005	√	√
Watchcow	0x66	POJ2230	USACO 2005 Jan.	√	√
Network of Schools	0x67	POJ1236	IOI 1996	√	√

续表

题目名称	章节	提交地址	来源或版权所有者	标程	数据
银河	0x67	AcWing368	杜宇飞，Nescafé 16 杯 NOIP 模拟赛	√	√
PKU ACM Team's Excursion	0x67	CH6703	唐浩，北京大学	√	
Katu Puzzle	0x67	POJ3678	北京大学	√	√
Priest John's Busiest Day	0x67	POJ3683	北京大学	√	√
关押罪犯	0x68	CH4901	CCF NOIP 2010	√	√
棋盘覆盖	0x68	CH6801		√	√
車的放置	0x68	CH6802		√	√
导弹防御塔	0x68	CH6803	Nescafé 19 杯 NOIP 模拟赛	√	√
Ants	0x68	POJ3565	ICPC NEERC 2007	√	√
Machine Schedule	0x69	POJ1325	ICPC Beijing 2002	√	√
Muddy Fields	0x69	POJ2226	USACO 2005 Jan.	√	√
骑士放置	0x69	CH6901		√	√
Vani 和 Cl2 捉迷藏	0x69	CH6902 AcWing379	Violet 4 杯省选模拟赛 Nescafé 21 杯模拟赛 CCF CTSC 2008	√	√
舞动的夜晚	0x6A	CH#17C	Contest Hunter Round #17 - C	√	√
Cable TV Network	0x6A	POJ1966	ICPC SEERC 2004	√	√
K 取方格数	0x6A	POJ3422	北京大学	√	√
Sightseeing	0x6B	POJ3463	Benelux Algorithm Programming Contest 2006 QI.	√	√
升降梯上	0x6B	CH6B02	Nescafé 28 杯 NOIP 模拟赛	√	√
GF 和猫咪的玩具	0x6B	CH6B03	潘雪峰	√	√
社交网络	0x6B	AcWing386	CCF NOI 2007	√	√
Arctic Network	0x6B	POJ2349	ICPC Waterloo 2002	√	√

续表

题目名称	章节	提交地址	来源或版权所有者	标程	数据
* *四叶草魔杖*	0x6B	CH6B06	Nescafé 29 杯 NOIP 模拟赛	√	√
直径	0x6B	AcWing389	NOI 2013 山东省选	√	√
逃学的小孩	0x6B	AcWing390	CCF NOI 2003	√	√
聚会	0x6B	AcWing391	NOI 2008 安徽省选	√	√
Rendezvous	0x6B	AcWing392	Poland OI 2012	√	√
Cashier Employment	0x6B	POJ1275	ICPC Tehran 2000	√	√
* *最优高铁环*	0x6B	CH6B12	李煜东	√	√
Redundant Paths	0x6B	POJ3177	USACO 2006 Jan.	√	√
* *矿场搭建*	0x6B	AcWing396	NOI 2012 湖南省选	√	√
逃不掉的路	0x6B	CH#24C	Contest Hunter Round #24 - C	√	√
Traffic Real Time Query System	0x6B	HDOJ3686 AcWing398	ICPC Hefei 2010	√	√
John's trip	0x6B	POJ1041	ICPC CERC 1995	√	√
太鼓达人	0x6B	CH6B18	毕克，Nescafé 18 杯 NOIP 模拟赛	√	√
Going from u to v or from v to u	0x6B	POJ2762	北京大学	√	√
* *杀人游戏*	0x6B	AcWing402	NOI 2011 中山市选	√	√
Planar	0x6B	AcWing403	NOI 2010 湖南省选	√	√
Wedding	0x6B	POJ3648	ICPC Waterloo 2007	√	√
Team Them Up!	0x6B	POJ1112	ICPC NEERC 2001	√	√
Place the Robots	0x6B	CH6B24	浙江大学	√	√
Steady Cow Assignment	0x6B	POJ3189	USACO 2006 Feb.	√	√
Going Home	0x6B	POJ2195	ICPC Pacific Northwest 2004	√	√
Air Raid	0x6B	POJ1422	ICPC Asia Dhaka 2002	√	√
Sorting Slide	0x6B	POJ1486	ICPC SWERC 1998	√	√
King's Quest	0x6B	POJ1904	ICPC NEERC 2003	√	√

续表

题目名称	章节	提交地址	来源或版权所有者	标程	数据
Drainage Ditches	0x6B	POJ1273	USACO 1993	√	√

OI和ICPC竞赛"提高阶段"首选之作——

对接竞赛实际难度，以高分段为标杆。精研此书，水平提升不止一个档次！

覆盖NOIP全部考点和ICPC中等题目范围，并向更高级别竞赛延伸。

算法模型逐级扩展，精选国内外数百道题目，巧妙构思思维体系。

融汇作者多年授课与实战经验，提供长期有效的训练指南。

ISBN 978-7-5711-1651-4

9 787571 116514

定价：86.00元