

前 言

一些著名的计算机科学家在有关计算机科学教育的论述中认为,计算机科学是一种创造性思维活动,其教育必须面向设计。计算机算法设计与分析正是一门面向设计,且处于计算机学科核心地位的教育课程。通过对计算机算法系统的学习与研究,理解和掌握算法设计的主要方法,培养对算法的计算复杂性进行正确分析的能力,为独立地设计算法和对给定算法进行复杂性分析奠定坚实的理论基础,对从事计算机系统结构、系统软件和应用软件研究与开发的科技工作者是非常重要的。

电子工业出版社出版的《计算机算法设计与分析》是普通高等教育“十一五”国家级规划教材,它是根据教育部高教司主持评审的《中国计算机科学与技术学科教程 2002》以及 ACM 和 IEEE/CS CC2001 组织编写的教材,在内容选材、深度把握、系统性和可用性方面进行了精心设计,力图适合高校本科生教学对学时数和知识结构的要求。本书是与《计算机算法设计与分析》配套的辅助教材,对该书中的全部习题做了解答或给出了解题思路提示。

算法设计与分析是计算学科的 9 个主科目之一,而且在整个学科知识体系中具有学科核心的重要地位,它充分体现了计算机科学方法论的理论、抽象和设计 3 个过程,知识面较宽,且有一定的深度;算法设计与分析课程需要反复再现计算机科学中用到的大问题的复杂性、效率、抽象的层次、重用、折中等带有普遍性的概念。根据作者多年的教学经验,算法设计与分析课程教学有以下 3 个特点,这使许多学生感到学习相当困难。

(1) 按照《中国计算机科学与技术学科教程 2002》以及 ACM 和 IEEE/CS CC2001 的要求,算法设计与分析课程教学包括的知识点多,内容十分丰富,学习量大。

(2) 课程内容理论性很强,对学生的抽象思维能力和逻辑推理能力要求较高。

(3) 课程内容还有很强的实践性,要求学生灵活运用所学到的算法设计策略解决实际问题。

教材中的课后习题能在很大程度上解决上面所说的困难。《计算机算法设计与分析》一书所配备的习题正是为此目的而设计的。教材出版后许多读者纷纷要求给出习题的解答和提示。为了让使用《计算机算法设计与分析》作为教材的教师和学生在广度和深度的各个层面更深刻地理解理论、抽象和设计这3 个过程以及重复出现的12 个基本概念(绑定、大问题的复杂性、概念和形式模型、一致性和完备性、演化、效率、抽象层次、按空间排序、按时间排序、重用、安全性、折中的结论),作者根据多年的教学经验编写了这本辅助教材,旨在让使用该书的教师更容易教,学生更容易学。为了便于对照阅读,本书的章序与《计算机算法设计与分析》一书的章序保持一致,且一一对应。由于篇幅的限制,一些算法及其实现的内容可在作者编写的另一本辅助教材《算法与数据结构学习指导与习题解析》^[1]中找到。

本书的内容是对教材《计算机算法设计与分析》的扩展,一些在教材中无法讲述的较深入的主题通过习题的形式展现出来。为了提高学生灵活运用算法设计策略解决实际问题的能力,本书将原教材中的许多习题改造成算法实现题,要求学生不仅设计出解决具体问题的算法,而且能上机实现。其中很多题目使用了多种不同解法,体现了算法的灵活性和适用性。根据作者多年的教学实践,这类算法实现题的教学效果非常好。

本书内容丰富,理论联系实际,可作为高等学校计算机科学与技术、软件工程、信息与计算科学等专业本科生和研究生学习计算机算法设计的辅助教材,也是工程技术人员和自学者的参考书。

作者还结合精品课程建设,进行了教材的立体化开发,包括主教材、辅助教材、实验与设计、电子课件和教学网站建设。作者的教学网站网址是:<http://www.algorithm.fzu.edu.cn>,作者的 E-mail 地址是:wangxd@fzu.edu.cn。欢迎广大读者访问教学网站并提出宝贵意见。

本书所附光盘中包含各章算法实现题的题目、测试数据和答案。共有 13 个子目录,包括:ch1,ch2,...,ch9,midexam1,midexam2,finalexam1,finalexam2。每章的每个算法实现题都对应一个子目录,其中的 test 子目录中是测试数据,answer 子目录中是相应的答案。midexam1 和 midexam2 目录中是两套期中试卷。finalexam1 和 finalexam2 目录中是两套期末试卷。算法设计实验的实现平台是 Microsoft Visual Studio 6.0 或 Microsoft Visual Studio.NET。

在本书编写过程中,福州大学“211 工程”计算机与信息工程重点学科实验室提供了优良的设备与工作环境。电子工业出版社负责本书编辑出版工作的全体同仁为本书的出版付出了大量辛勤劳动,他们认真细致,一丝不苟的工作精神保证了本书的出版质量。在此,谨向每一位曾经关心和支持本书编写工作的各方面人士表示衷心的感谢!

作 者
2006 年 7 月

目 录

第 1 章 算法概述	(1)
习题 1-1 函数的渐近表达式	(1)
习题 1-2 $O(1)$ 和 $O(2)$ 的区别	(1)
习题 1-4 按渐近阶排列表达式	(1)
习题 1-5 算法效率	(1)
习题 1-6 硬件效率	(2)
习题 1-7 函数渐近阶	(2)
习题 1-8 $n!$ 的阶	(2)
习题 1-9 $3n-1$ 问题	(3)
习题 1-10 平均情况下的计算时间复杂性	(3)
算法实现题 1-1 统计数字问题	(3)
算法实现题 1-2 字典序问题	(4)
算法实现题 1-3 最多约数问题	(5)
算法实现题 1-4 金币阵列问题	(7)
算法实现题 1-5 最大间隙问题	(9)
第 2 章 递归与分治策略	(12)
习题 2-1 Hanoi 塔问题的非递归算法	(12)
习题 2-2 7 个二分搜索算法	(13)
习题 2-3 改写二分搜索算法	(16)
习题 2-4 大整数乘法的 $O(nm^{\log_{3/2}})$ 算法	(17)
习题 2-5 5 次 $n/3$ 位整数的乘法	(17)
习题 2-6 矩阵乘法	(19)
习题 2-7 多项式乘积	(19)
习题 2-8 不动点问题的 $O(\log n)$ 时间算法	(20)
习题 2-9 主元素问题的线性时间算法	(20)
习题 2-10 无序集主元素问题的线性时间算法	(20)
习题 2-11 $O(1)$ 空间子数组换位算法	(20)
习题 2-12 $O(1)$ 空间合并算法	(22)
习题 2-13 \sqrt{n} 段合并排序算法	(29)
习题 2-14 自然合并排序算法	(29)
习题 2-15 最大值和最小值问题的最优算法	(31)
习题 2-16 最大值和次大值问题的最优算法	(32)
习题 2-17 整数集合排序	(32)
习题 2-18 第 k 小元素问题的计算时间下界	(32)
习题 2-19 非增序快速排序算法	(33)

习题 2-20	随机化算法	(34)
习题 2-21	随机化快速排序算法	(34)
习题 2-22	随机排列算法	(34)
习题 2-23	算法 QuickSort 中的尾递归	(34)
习题 2-24	用栈模拟递归	(34)
习题 2-25	算法 Select 中的元素划分	(34)
习题 2-26	$O(n\log n)$ 时间快速排序算法	(34)
习题 2-27	最接近中位数的 k 个数	(34)
习题 2-28	X 和 Y 的中位数	(35)
习题 2-29	网络开关设计	(35)
习题 2-32	带权中位数问题	(35)
习题 2-34	构造 Gray 码的分治算法	(36)
习题 2-35	网球循环赛日程表	(36)
习题 2-36	二叉树 T 的前序、中序和后序序列	(40)
算法实现题 2-1	输油管道问题(习题 2-30)	(41)
算法实现题 2-2	众数问题(习题 2-31)	(42)
算法实现题 2-3	邮局选址问题(习题 2-32)	(43)
算法实现题 2-4	马的 Hamilton 周游路线问题(习题 2-33)	(43)
算法实现题 2-5	半数集问题	(51)
算法实现题 2-6	半数单集问题	(52)
算法实现题 2-7	士兵站队问题	(53)
算法实现题 2-8	有重复元素的排列问题	(54)
算法实现题 2-9	排列的字典序问题	(55)
算法实现题 2-10	集合划分问题	(57)
算法实现题 2-11	集合划分问题	(58)
算法实现题 2-12	双色 Hanoi 塔问题	(60)
算法实现题 2-13	标准二维表问题	(61)
算法实现题 2-14	整数因子分解问题	(61)
第 3 章 动态规划		(63)
习题 3-1	最长单调递增子序列	(63)
习题 3-2	最长单调递增子序列的 $O(n\log n)$ 算法	(63)
习题 3-7	漂亮打印	(65)
习题 3-11	整数线性规划问题	(65)
习题 3-12	二维 0-1 背包问题	(65)
习题 3-14	Ackermann 函数	(66)
习题 3-17	最短行驶路线	(68)
习题 3-19	最优旅行路线	(69)
算法实现题 3-1	独立任务最优调度问题(习题 3-3)	(69)
算法实现题 3-2	最少硬币问题(习题 3-4)	(71)
算法实现题 3-3	序关系计数问题(习题 3-5)	(71)

算法实现题 3-4	多重幂计数问题(习题 3-6)	(72)
算法实现题 3-5	编辑距离问题(习题 3-8)	(72)
算法实现题 3-6	石子合并问题(习题 3-9)	(73)
算法实现题 3-7	数字三角形问题(习题 3-10)	(75)
算法实现题 3-8	乘法表问题(习题 3-13)	(76)
算法实现题 3-9	租用游艇问题(习题 3-15)	(77)
算法实现题 3-10	汽车加油行驶问题(习题 3-16)	(78)
算法实现题 3-11	最小 m 段和问题	(79)
算法实现题 3-12	圈乘运算问题(习题 3-18)	(80)
算法实现题 3-13	最大长方体问题(习题 3-21)	(85)
算法实现题 3-14	正则表达式匹配问题(习题 3-22)	(86)
算法实现题 3-15	双调旅行售货员问题(习题 3-23)	(90)
算法实现题 3-16	最大 k 乘积问题(习题 5-28)	(92)
算法实现题 3-17	最少费用购物问题(习题 3-20)	(94)
算法实现题 3-18	收集样本问题	(96)
算法实现题 3-19	最优时间表问题	(98)
算法实现题 3-20	字符串比较问题	(98)
算法实现题 3-21	有向树 k 中值问题	(100)
算法实现题 3-22	有向树独立 k 中值问题	(104)
算法实现题 3-23	有向直线 m 中值问题	(108)
算法实现题 3-24	有向直线 2 中值问题	(111)
算法实现题 3-25	树的最大连通分支问题	(114)
算法实现题 3-26	直线 k 中值问题	(116)
算法实现题 3-27	直线 k 覆盖问题	(121)
算法实现题 3-28	m 处理器问题	(125)
算法实现题 3-29	红黑树的红色内结点问题	(127)

第 4 章 贪心算法 (136)

习题 4-2	活动安排问题的贪心选择	(136)
习题 4-3	背包问题的贪心选择性质	(136)
习题 4-4	特殊的 0-1 背包问题	(136)
习题 4-10	程序最优存储问题	(136)
习题 4-13	最优装载问题的贪心算法	(137)
习题 4-18	Fibonacci 序列的哈夫曼编码	(137)
习题 4-19	最优前缀码的编码序列	(137)
习题 4-21	任务集独立性问题	(137)
习题 4-22	矩阵拟阵	(137)
习题 4-23	最小权最大独立子集拟阵	(138)
习题 4-27	整数边权 Prim 算法	(138)
习题 4-28	最大权最小生成树	(138)
习题 4-29	最短路径的负边权	(138)

习题 4-30	整数边权 Dijkstra 算法	(138)
算法实现题 4-1	会场安排问题(习题 4-1)	(138)
算法实现题 4-2	最优合并问题(习题 4-5)	(139)
算法实现题 4-3	磁带最优存储问题(习题 4-6)	(140)
算法实现题 4-4	磁盘文件最优存储问题(习题 4-7)	(141)
算法实现题 4-5	程序存储问题(习题 4-8)	(142)
算法实现题 4-6	最优服务次序问题(习题 4-11)	(143)
算法实现题 4-7	多处最优服务次序问题(习题 4-12)	(143)
算法实现题 4-8	d 森林问题(习题 4-14)	(144)
算法实现题 4-9	汽车加油问题(习题 4-16)	(146)
算法实现题 4-10	区间覆盖问题(习题 4-17)	(147)
算法实现题 4-11	硬币找钱问题(习题 4-24)	(148)
算法实现题 4-12	删数问题(习题 4-25)	(148)
算法实现题 4-13	数列极差问题(习题 4-26)	(149)
算法实现题 4-14	嵌套箱问题(习题 4-31)	(149)
算法实现题 4-15	套汇问题(习题 4-32)	(150)
算法实现题 4-16	信号增强装置问题(习题 5-20)	(152)
算法实现题 4-17	磁带最大利用率问题(习题 4-9)	(153)
算法实现题 4-18	非单位时间任务安排问题(习题 4-15)	(154)
算法实现题 4-19	多元 Huffman 编码问题(习题 4-20)	(156)
算法实现题 4-20	多元 Huffman 编码变形	(157)
算法实现题 4-21	区间相交问题	(159)
算法实现题 4-22	任务时间表问题	(159)
算法实现题 4-23	最优分解问题	(160)
算法实现题 4-24	可重复最优分解问题	(161)
算法实现题 4-25	可重复最优组合分解问题	(161)
算法实现题 4-26	旅行规划问题	(163)
算法实现题 4-27	登山机器人问题	(163)
第 5 章	回溯法	(165)
习题 5-1	装载问题改进回溯法 1	(165)
习题 5-2	装载问题改进回溯法 2	(166)
习题 5-4	0-1 背包问题的最优解	(166)
习题 5-5	最大团问题的迭代回溯法	(168)
习题 5-7	旅行售货员问题的费用上界	(169)
习题 5-8	旅行售货员问题的上界函数	(171)
算法实现题 5-1	子集和问题(习题 5-3)	(171)
算法实现题 5-2	最小长度电路板排列问题(习题 5-9)	(173)
算法实现题 5-3	最小重量机器设计问题(习题 5-10)	(175)
算法实现题 5-4	运动员最佳配对问题(习题 5-14)	(176)
算法实现题 5-5	无分隔符字典问题(习题 5-15)	(178)

算法实现题 5-6	无和集问题(习题 5-16)	(180)
算法实现题 5-7	n 色方柱问题(习题 5-17)	(181)
算法实现题 5-8	整数变换问题(习题 5-18)	(186)
算法实现题 5-9	拉丁矩阵问题	(187)
算法实现题 5-10	排列宝石问题(习题 5-19)	(188)
算法实现题 5-11	重复拉丁矩阵问题(习题 5-19)	(190)
算法实现题 5-12	罗密欧与朱丽叶的迷宫问题(习题 5-21)	(192)
算法实现题 5-13	工作分配问题(习题 5-22)	(195)
算法实现题 5-14	独立钻石跳棋问题(习题 5-23)	(196)
算法实现题 5-15	智力拼图问题(习题 5-24)	(202)
算法实现题 5-16	布线问题(习题 5-25)	(208)
算法实现题 5-17	最佳调度问题(习题 5-26)	(209)
算法实现题 5-18	无优先级运算问题(习题 5-27)	(210)
算法实现题 5-19	世界名画陈列馆问题(习题 5-29)	(212)
算法实现题 5-20	世界名画陈列馆问题(不重复监视)(习题 5-30)	(216)
算法实现题 5-21	$2 \times 2 \times 2$ 魔方问题	(218)
算法实现题 5-22	魔方(Rubik's Cube)问题(习题 5-31)	(234)
算法实现题 5-23	算 24 点问题	(250)
算法实现题 5-24	算 m 点问题	(252)
算法实现题 5-25	双轨车皮编序问题	(255)
算法实现题 5-26	多轨车皮编序问题	(259)
算法实现题 5-27	部落卫队问题(习题 5-6)	(262)
算法实现题 5-28	虫蚀算式问题	(263)
算法实现题 5-29	完备环序列问题	(267)
算法实现题 5-30	离散 0-1 串问题	(269)
算法实现题 5-31	喷漆机器人问题	(270)
算法实现题 5-32	子集树问题(习题 5-11)	(273)
算法实现题 5-33	0-1 背包问题(习题 5-11)	(274)
算法实现题 5-34	排列树问题(习题 5-12)	(276)
算法实现题 5-35	一般解空间搜索问题(习题 5-13)	(278)
算法实现题 5-36	最短加法链问题	(280)
算法实现题 5-37	$n^2 - 1$ 谜问题	(286)
第 6 章 分支限界法		(292)
习题 6-1	0-1 背包问题的栈式分支限界法	(292)
习题 6-2	释放结点空间的队列式分支限界法	(294)
习题 6-3	及时删除不用的结点	(295)
习题 6-4	用最大堆存储活结点的优先队列式分支限界法	(297)
习题 6-5	释放结点空间的优先队列式分支限界法	(300)
习题 6-6	团顶点数的上界	(303)
习题 6-7	团顶点数改进的上界	(303)

习题 6-8	修改解旅行售货员问题的分支限界法	(303)
习题 6-9	解旅行售货员问题的分支限界法中保存已产生的排列树	(306)
习题 6-10	电路板排列问题的队列式分支限界法	(308)
算法实现题 6-1	最小长度电路板排列问题(习题 6-11)	(310)
算法实现题 6-2	最小长度电路板排列问题(习题 6-12)	(313)
算法实现题 6-3	最小权顶点覆盖问题(习题 6-13)	(316)
算法实现题 6-4	无向图的最大割问题(习题 6-14)	(319)
算法实现题 6-5	最小重量机器设计问题(习题 6-15)	(322)
算法实现题 6-6	运动员最佳配对问题(习题 6-16)	(325)
算法实现题 6-7	n 皇后问题(习题 6-18)	(327)
算法实现题 6-8	圆排列问题(习题 6-19)	(330)
算法实现题 6-9	布线问题(习题 6-20)	(333)
算法实现题 6-10	最佳调度问题(习题 6-21)	(335)
算法实现题 6-11	无优先级运算问题(习题 6-22)	(338)
算法实现题 6-12	世界名画陈列馆问题(习题 6-24)	(341)
算法实现题 6-13	子集空间树问题(习题 6-25)	(344)
算法实现题 6-14	排列空间树问题(习题 6-26)	(347)
算法实现题 6-15	一般解空间的队列式分支限界法(习题 6-27)	(352)
算法实现题 6-16	子集空间树问题(习题 6-28)	(357)
算法实现题 6-17	排列空间树问题(习题 6-29)	(362)
算法实现题 6-18	一般解空间的优先队列式分支限界法(习题 6-30)	(368)
算法实现题 6-19	骑士征途问题	(373)
算法实现题 6-20	推箱子问题	(374)
算法实现题 6-21	图形变换问题	(379)
算法实现题 6-22	行列变换问题	(381)
算法实现题 6-23	重排 n^2 宫问题	(386)
算法实现题 6-24	最长距离问题	(394)
第 7 章	概率算法	(399)
习题 7-1	模拟正态分布随机变量	(399)
习题 7-2	随机抽样算法	(400)
习题 7-3	随机产生 m 个整数	(400)
习题 7-4	集合大小的概率算法	(401)
习题 7-5	生日问题	(402)
习题 7-6	易验证问题的拉斯维加斯算法	(402)
习题 7-7	用数组模拟有序链表	(403)
习题 7-8	$O(n^{3/2})$ 舍伍德型排序算法	(403)
习题 7-9	n 后问题解的存在性	(403)
习题 7-11	整数因子分解算法	(405)
习题 7-12	非蒙特卡罗算法的例子	(405)
习题 7-13	重复 3 次的蒙特卡罗算法	(406)

习题 7-14	集合随机元素算法	(407)
习题 7-15	由蒙特卡罗算法构造拉斯维加斯算法	(408)
习题 7-16	产生素数算法	(408)
习题 7-19	矩阵方程问题	(409)
算法实现题 7-1	模平方根问题(习题 7-10)	(410)
算法实现题 7-2	素数测试问题(习题 7-17)	(412)
算法实现题 7-3	集合相等问题(习题 7-18)	(412)
算法实现题 7-4	逆矩阵问题(习题 7-20)	(413)
算法实现题 7-5	多项式乘积问题(习题 7-21)	(414)
算法实现题 7-6	皇后控制问题	(414)
算法实现题 7-7	3-SAT 问题	(418)
算法实现题 7-8	战车问题	(419)
算法实现题 7-9	圆排列问题	(421)
算法实现题 7-10	骑士控制问题	(422)
算法实现题 7-11	骑士对攻问题	(423)
第 8 章 线性规划与网络流		(425)
习题 8-1	线性规划可行区域无界的例子	(425)
习题 8-2	单源最短路与线性规划	(425)
习题 8-3	网络最大流与线性规划	(426)
习题 8-4	最小费用流与线性规划	(426)
习题 8-5	运输计划问题	(427)
习题 8-6	单纯形算法	(427)
习题 8-7	边连通度问题	(428)
习题 8-8	有向无环网络的最大流	(428)
习题 8-9	无向网络的最大流	(429)
习题 8-12	最大流更新算法	(429)
习题 8-16	混合图欧拉回路问题	(429)
习题 8-22	单源最短路与最小费用流	(430)
习题 8-23	中国邮路问题	(430)
算法实现题 8-1	飞行员配对方案问题(习题 8-10)	(430)
算法实现题 8-2	太空飞行计划问题(习题 8-11)	(432)
算法实现题 8-3	最小路径覆盖问题(习题 8-13)	(433)
算法实现题 8-4	魔术球问题(习题 8-14)	(434)
算法实现题 8-5	圆桌问题(习题 8-15)	(435)
算法实现题 8-6	最长递增子序列问题(习题 8-17)	(436)
算法实现题 8-7	试题库问题(习题 8-18)	(438)
算法实现题 8-8	机器人路径规划问题(习题 8-19)	(440)
算法实现题 8-9	方格取数问题(习题 8-20)	(443)
算法实现题 8-10	餐巾计划问题(习题 8-21)	(447)
算法实现题 8-11	航空路线问题(习题 8-24)	(448)

算法实现题 8-12	软件补丁问题(习题 8-25)	(449)
算法实现题 8-13	星际转移问题(习题 8-26)	(451)
算法实现题 8-14	孤岛营救问题(习题 8-27)	(452)
算法实现题 8-15	汽车加油行驶问题(习题 8-28)	(453)
算法实现题 8-16	数字梯形问题	(457)
算法实现题 8-17	运输问题	(462)
算法实现题 8-18	分配工作问题	(464)
算法实现题 8-19	负载均衡问题	(466)
算法实现题 8-20	深海机器人问题	(467)
算法实现题 8-21	最长 k 可重区间集问题	(471)
算法实现题 8-22	最长 k 可重线段集问题	(474)
算法实现题 8-23	火星探险问题	(478)
算法实现题 8-24	骑士共存问题	(479)
第 9 章 NP 完全性理论与近似算法		(485)
习题 9-1	RAM 和 RASP 程序	(485)
习题 9-2	RAM 和 RASP 程序的复杂性	(485)
习题 9-3	计算 n^n 的 RAM 程序	(485)
习题 9-4	平面图着色问题的绝对近似算法	(485)
习题 9-5	最优程序存储问题	(486)
习题 9-6	树的最优顶点覆盖	(487)
习题 9-7	顶点覆盖算法的性能比	(488)
习题 9-8	闭的常数性能比近似算法	(488)
习题 9-10	旅行售货员问题的常数性能比近似算法	(489)
习题 9-11	瓶颈旅行售货员问题	(489)
习题 9-12	最优旅行售货员回路不自相交	(490)
习题 9-14	集合覆盖问题的实例	(490)
习题 9-16	多机调度问题的近似算法	(491)
习题 9-17	LPT 算法的最坏情况实例	(493)
习题 9-18	多机调度问题的多项式时间近似算法	(493)
算法实现题 9-1	旅行售货员问题的近似算法(习题 9-9)	(494)
算法实现题 9-2	可满足问题的近似算法(习题 9-19)	(496)
算法实现题 9-3	最大可满足问题的近似算法(习题 9-20)	(497)
算法实现题 9-4	子集和问题的近似算法(习题 9-15)	(499)
算法实现题 9-5	子集和问题的完全多项式时间近似算法	(500)
算法实现题 9-6	2-SAT 问题的线性时间算法	(501)
算法实现题 9-7	实现算法 greedySetCover(习题 9-13)	(504)
参考文献		(508)

第1章 算法概述

习题 1-1 函数的渐近表达式

求下列函数的渐近表达式:

$3n^2 + 10n$; $n^2/10 + 2^n$; $21 + 1/n$; $\log n^3$; $10\log 3^n$ 。

分析与解答:

$$3n^2 + 10n = O(n^2);$$

$$n^2/10 + 2^n = O(2^n);$$

$$21 + 1/n = O(1);$$

$$\log n^3 = O(\log n);$$

$$10\log 3^n = O(n)。$$

习题 1-2 $O(1)$ 和 $O(2)$ 的区别

试论 $O(1)$ 和 $O(2)$ 的区别。

分析与解答:

根据符号 O 的定义易知 $O(1) = O(2)$ 。用 $O(1)$ 或 $O(2)$ 表示同一个函数时,差别仅在于其中的常数因子。

习题 1-4 按渐近阶排列表达式

按照渐近阶从低到高的顺序排列以下表达式: $4n^2$, $\log n$, 3^n , $20n$, $2 \cdot n^{2/3}$ 。又 $n!$ 应该排在哪一位?

分析与解答:

按渐近阶从低到高,函数排列顺序如下: 2 , $\log n$, $n^{2/3}$, $20n$, $4n^2$, 3^n , $n!$ 。

习题 1-5 算法效率

(1) 假设某算法在输入规模为 n 时的计算时间为 $T(n) = 3 \times 2^n$ 。在某台计算机上实现并完成该算法的时间为 t 秒。现有另一台计算机,其运行速度为第一台的 64 倍,那么在这台新机器上用同一算法在 t 秒内能解输入规模为多大的问题?

(2) 若上述算法的计算时间改进为 $T(n) = n^2$, 其余条件不变,则在新机器上用 t 秒时间能解输入规模为多大的问题?

(3) 若上述算法的计算时间进一步改进为 $T(n) = 8$, 其余条件不变,那么在新机器上用 t 秒时间能解输入规模为多大的问题?

分析与解答:

(1) 设新机器用同一算法在 t 秒内能解输入规模为 n_1 的问题。因此有: $t = 3 \times 2^n = 3 \times 2^{n_1}/64$, 解得 $n_1 = n + 6$ 。

(2) $n_1^2 = 64n^2 \Rightarrow n_1 = 8n$ 。

(3) 由于 $T(n) = \text{常数}$, 因此算法可解任意规模的问题。

习题 1-6 硬件效率

硬件厂商 XYZ 公司宣称他们最新研制的微处理器运行速度为其竞争对手 ABC 公司同类产品的 100 倍。对于计算复杂性分别为 n, n^2, n^3 和 $n!$ 的各算法, 若用 ABC 公司的计算机在 1 小时内能解输入规模为 n 的问题, 那么用 XYZ 公司的计算机在 1 小时内分别能解输入规模为多大的问题?

分析与解答:

$$n' = 100n$$

$$n'^2 = 100n^2 \Rightarrow n' = 10n$$

$$n'^3 = 100n^3 \Rightarrow n' = \sqrt[3]{100}n = 4.64n$$

$$n'! = 100n! \Rightarrow n' < n + \log 100 = n + 6.64$$

习题 1-7 函数渐近阶

对于下列各组函数 $f(n)$ 和 $g(n)$, 确定 $f(n) = O(g(n))$ 或 $f(n) = \Omega(g(n))$ 或 $f(n) = \theta(g(n))$, 并简述理由。

(1) $f(n) = \log n^2$; $g(n) = \log n + 5$

(2) $f(n) = \log n^2$; $g(n) = \sqrt{n}$

(3) $f(n) = n$; $g(n) = \log^2 n$

(4) $f(n) = n \log n + n$; $g(n) = \log n$

(5) $f(n) = 10$; $g(n) = \log 10$

(6) $f(n) = \log^2 n$; $g(n) = \log n$

(7) $f(n) = 2^n$; $g(n) = 100n^2$

(8) $f(n) = 2^n$; $g(n) = 3^n$

分析与解答:

(1) $\log n^2 = \theta(\log n + 5)$

(2) $\log n^2 = O(\sqrt{n})$

(3) $n = \Omega(\log^2 n)$

(4) $n \log n + n = \Omega(\log n)$

(5) $10 = \theta(\log 10)$

(6) $\log^2 n = \Omega(\log n)$

(7) $2^n = \Omega(100n^2)$

(8) $2^n = O(3^n)$

习题 1-8 $n!$ 的阶

证明: $n! = o(n^n)$ 。

分析与解答:

Stirling's approximation;

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left[1 + \theta\left(\frac{1}{n}\right)\right]$$

$$\lim_{n \rightarrow \infty} n! / n^n = \frac{\sqrt{2\pi n} \left[1 + \theta\left(\frac{1}{n}\right) \right]}{e^n} = 0$$

习题 1-9 $3n+1$ 问题

下面的算法段用于确定 n 的初始值。试分析该算法段所需计算时间的上界和下界。

```
while(n>1)
    if (odd(n))
        n=3*n+1;
    else
        n=n/2;
```

分析与解答:

该算法表述的是著名的 $3n+1$ 问题。在最坏情况下, 该算法的计算时间下界显然为 $\Omega(\log n)$ 。

算法的计算时间上界至今未知。算法是否在有限时间内结束, 至今还是一个悬而未决的问题。日本学者米田信夫曾对 10^{13} 内的所有自然数验证上述算法均在有限步结束。人们猜测, 对所有自然数, 上述算法均在有限步结束, 但无法给出理论证明, 因此也无法分析上述算法的计算时间上界。这个猜测就成为著名的 $3n+1$ 猜想, 也称为 Collatz 猜想。

习题 1-10 平均情况下的计算时间复杂性

证明: 如果一个算法在平均情况下的计算时间复杂性为 $\theta(f(n))$, 则该算法在最坏情况下所需的计算时间为 $\Omega(f(n))$ 。

分析与解答:

$$\begin{aligned} T_{\text{avg}}(N) &= \sum_{I \in D_N} P(I) T(N, I) \\ &\leq \sum_{I \in D_N} P(I) \max_{I' \in D_N} T(N, I') \\ &= T(N, I^*) \sum_{I \in D_N} P(I) \\ &= T(N, I^*) \\ &= T_{\text{max}}(N) \end{aligned}$$

因此, $T_{\text{max}}(N) = \Omega(T_{\text{avg}}(N)) = \Omega(\theta(f(n))) = \Omega(f(n))$ 。

算法实现题 1-1 统计数字问题

★问题描述:

一本书的页码从自然数 1 开始顺序编码直到自然数 n 。书的页码按照通常的习惯编排, 每个页码都不含多余的前导数字 0。例如, 第 6 页用数字 6 表示, 而不是 06 或 006 等。数字计数问题要求对给定书的总页码 n , 计算出书的全部页码中分别用到多少次数字 0, 1, 2, ..., 9。

★编程任务:

给定表示书的总页码的十进制整数 n ($1 \leq n \leq 10^9$)。编程计算书的全部页码中分别用到

多少次数字 0,1,2,...,9。

★数据输入:

输入数据由文件名为 input.txt 的文本文件提供。每个文件只有 1 行,给出表示书的总页码的整数 n 。

★结果输出:

程序运行结束时,将计算结果输出到文件 output.txt 中。输出文件共有 10 行,在第 k 行输出页码中用到数字 $k-1$ 的次数, $k=1,2,\dots,10$ 。

输入文件示例

input.txt

11

输出文件示例

output.txt

1

4

1

1

1

1

1

1

1

1

分析与解答:

考察由 0,1,2,...,9 组成的所有 n 位数。从 n 个 0 到 n 个 9 共有 10^n 个 n 位数。在这 10^n 个 n 位数中,0,1,2,...,9 每个数字使用次数相同,设为 $f(n)$ 。 $f(n)$ 满足如下递归式:

$$f(n)=\begin{cases} 10f(n-1)+10^{n-1} & n>1 \\ 1 & n=1 \end{cases}$$

由此可知, $f(n)=n10^{n-1}$ 。

据此,可从高位向低位进行统计,再减去多余的 0 的个数即可。

算法实现题 1-2 字典序问题

★问题描述:

在数据加密和数据压缩中常需要对特殊的字符串进行编码。给定的字母表 A 由 26 个小写英文字母组成 $A=\{a,b,\dots,z\}$ 。该字母表产生的升序字符串是指字符串中字母从左到右出现的次序与字母在字母表中出现的次序相同,且每个字符最多出现 1 次。例如, a,b,ab,bc,xyz 等字符串都是升序字符串。现在对字母表 A 产生的所有长度不超过 6 的升序字符串按照字典序排列并编码如下。

1	2	...	26	27	28	...
a	b	...	z	ab	ac	...

对于任意长度不超过 6 的升序字符串,迅速计算出它在上述字典中的编码。

★编程任务:

对于给定的长度不超过 6 的升序字符串,编程计算出它在上述字典中的编码。

★数据输入:

输入数据由文件名为 input.txt 的文本文件提供。文件的第 1 行是一个正整数 k ，表示接下来共有 k 行。在接下来的 k 行中，每行给出一个字符串。

★结果输出：

程序运行结束时，将计算结果输出到文件 output.txt 中。文件共有 k 行，每行对应于一个字符串的编码。

输入文件示例	输出文件示例
input.txt	output.txt
2	1
a	2
b	

分析与解答：

考察一般情况下长度不超过 k 的升序字符串。

设以第 i 个字符打头的长度不超过 k 的升序字符串个数为 $f(i, k)$ ，长度不超过 k 的升序字符串总个数为 $g(k)$ ，则 $g(k) = \sum_{i=1}^{26} f(i, k)$ 。易知

$$f(i, 1) = 1 \quad g(1) = \sum_{i=1}^{26} f(i, 1) = 26$$

$$f(i, 2) = \sum_{j=i+1}^{26} f(j, 1) = 26 - i \quad g(2) = \sum_{i=1}^{26} f(i, 2) = \sum_{i=1}^{26} (26 - i) = 325$$

一般情况下有

$$f(i, k) = \sum_{j=i+1}^{26} f(j, k-1) \quad g(k) = \sum_{i=1}^{26} f(i, k) = \sum_{i=1}^{26} \sum_{j=i+1}^{26} f(j, k-1)$$

据此可计算出每个升序字符串的编码。

算法实现题 1-3 最多约数问题

★问题描述：

正整数 x 的约数是能整除 x 的正整数。正整数 x 的约数个数记为 $\text{div}(x)$ 。例如，1, 2, 5, 10 都是正整数 10 的约数，且 $\text{div}(10)=4$ 。设 a 和 b 是 2 个正整数， $a \leq b$ ，找出 a 和 b 之间约数个数最多的数 x 。

★编程任务：

对于给定的 2 个正整数 $a \leq b$ ，编程计算 a 和 b 之间约数个数最多的数。

★数据输入：

输入数据由文件名为 input.txt 的文本文件提供。文件的第 1 行有 2 个正整数 a 和 b 。

★结果输出：

程序运行结束时，若找到的 a 和 b 之间约数个数最多的数是 x ，则将 $\text{div}(x)$ 输出到文件 output.txt 中。

输入文件示例	输出文件示例
input.txt	output.txt
1 36	9

分析与解答：

设正整数 x 的质因子分解为

$$x = p_1^{N_1} p_2^{N_2} \cdots p_k^{N_k}$$

则

$$\text{div}(x) = (N_1 + 1)(N_2 + 1) \cdots (N_k + 1)$$

搜索区间 $[a, b]$ 中数的质因子分解。

primes 产生质数。

```
void primes()
{
    bool get[MAXP+1];
    for(int i=2; i<=MAXP; i++) get[i]=true;
    for(i=2; i<=MAXP; i++)
        if(get[i]) {
            int j=i+i;
            while(j<=MAXP) {get[j]=false; j+=i;}
        }
    for(int ii=2, j=0; ii<=MAXP; ii++)
        if(get[ii]) prim[++j]=ii;
}
```

search 搜索最多约数。

```
void search(int from, int tot, int num, int low, int up)
{
    if(num>=1)
        if((tot>max) || ((tot==max) && (num<numb))) {
            max=tot; numb=num;
        }
    if((low==up) && (low>num)) search(from, tot*2, num*low, 1, 1);
    for(int i=from; i<=PCOUNT; i++)
        if(prim[i]>up) return;
        else {
            int j=prim[i], x=low-1, y=up, n=num, t=tot, m=1;
            while(true) {
                m++; t+=tot; x/=j; y/=j;
                if(x==y) break;
                n*=j;
                search(i+1, t, n, x+1, y);
            }
            m=1<<m;
            if(tot<max/m) return;
        }
}
```

实现算法的主函数如下。

```

int main()
{
    primes();
    cin>>l>>u;
    if((l==1)&&(u==1)){max=1;numb=1;}
    else{max=2;numb=1;search(1,1,1,1,u);}
    cout<<max<<endl;
    return 0;
}

```

算法实现题 1-4 金币阵列问题

★问题描述:

有 $m \times n$ ($m \leq 100, n \leq 100$) 枚金币在桌面上排成一个 m 行 n 列的金币阵列。每一枚金币或正面朝上,或背面朝上。用数字表示金币状态,0 表示金币正面朝上,1 表示金币背面朝上。

金币阵列游戏的规则是:

- (1) 每次可将任一行金币翻过来放在原来的位置上;
- (2) 每次可任选 2 列,交换这 2 列金币的位置。

★编程任务:

给定金币阵列的初始状态和目标状态,编程计算按金币游戏规则,将金币阵列从初始状态变换到目标状态所需的最少变换次数。

★数据输入:

由文件 input.txt 给出输入数据。文件中有多组数据。文件的第 1 行有 1 个正整数 k , 表示有 k 组数据。每组数据的第 1 行有 2 个正整数 m 和 n 。以下的 m 行是金币阵列的初始状态,每行有 n 个数字表示该行金币的状态,0 表示正面朝上,1 表示背面朝上。接着的 m 行是金币阵列的目标状态。

★结果输出:

将计算出的最少变换次数按照输入数据的次序输出到文件 output.txt。相应数据无解时输出 -1。

输入文件示例

input.txt

2

4 3

1 0 1

0 0 0

1 1 0

1 0 1

1 0 1

1 1 1

0 1 1

输出文件示例

output.txt

2

-1

```

1 0 1
4 3
1 0 1
0 0 0
1 0 0
1 1 1
1 1 0
1 1 1
0 1 1
1 0 1

```

分析与解答：

枚举初始状态每一列变换为目标状态第 1 列的情况。算法描述如下。

```

int k, n, m, count, best;
int b0[Size+1][Size+1], b1[Size+1][Size+1], b[Size+1][Size+1];
bool found;

int main()
{
    cin >> k;
    for(int i=1; i<=k; i++) {
        cin >> n >> m;
        for(int x=1; x<=n; x++)
            for(int y=1; y<=m; y++)
                cin >> b0[x][y];
        for(x=1; x<=n; x++)
            for(int y=1; y<=m; y++)
                cin >> b1[x][y];
        acpy(b, b1); best = m + n + 1;
        for(int j=1; j<=m; j++) {
            acpy(b1, b); count = 0; trans2(1, j);
            for(int p=1; p<=n; p++)
                if(b0[p][1] != b1[p][1]) trans1(p);
            for(p=1; p<=m; p++) {
                found = false;
                for(int q=p; q<=m; q++)
                    if(same(p, q)) {trans2(p, q); found = true; break;}
                if(! found) break;
            }
            if(found && count < best) best = count;
        }
        if(best < m + n + 1) cout << best << endl;
        else cout << -1 << endl;
    }
}

```

```

    }
    return 0;
}

```

其中,trans1 模拟行翻转变换,trans2 模拟列交换变换。

```

void trans1(int x)
{
    for(int i=1;i<=m;i++)b1[x][i]=b1[x][i]^1;
    count++;
}

void trans2(int x,int y)
{
    for(int i=1;i<=n;i++) Swap(b1[i][x],b1[i][y]);
    if(x!=y)count++;
}

bool same(int x,int y)
{
    for(int i=1;i<=n;i++) if(b0[i][x]!=b1[i][y]) return false;
    return true;
}

void acpy(int a[Size+1][Size+1],int b[Size+1][Size+1])
{
    for(int i=1;i<=n;i++)
        for(int j=1;j<=m;j++)
            a[i][j]=b[i][j];
}

```

算法实现题 1-5 最大间隙问题

★问题描述:

最大间隙问题:给定 n 个实数 x_1, x_2, \dots, x_n , 求这 n 个数在实轴上相邻 2 个数之间的最大差值。假设对任何实数的下取整函数耗时 $O(1)$, 设计解最大间隙问题的线性时间算法。

★编程任务:

对于给定的 n 个实数 x_1, x_2, \dots, x_n , 编程计算它们的最大间隙。

★数据输入:

输入数据由文件名为 input.txt 的文本文件提供。文件的第 1 行有 1 个正整数 n 。接下来的 1 行中有 n 个实数 x_1, x_2, \dots, x_n 。

★结果输出:

程序运行结束时, 将找到的最大间隙输出到文件 output.txt 中。

输入文件示例

input.txt

5

2.3 3.1 7.5 1.5 6.3

输出文件示例

output.txt

3.2

分析与解答:

用鸽舍原理设计最大间隙问题的线性时间算法如下。

```
double maxgap(int n, double *x)
{
    double minx=x[mini(n, x)], maxx=x[maxi(n, x)];
    // 用  $n-2$  个等间距点分割区间  $[\text{minx}, \text{maxx}]$ .
    // 产生  $n-1$  个桶, 每个桶  $i$  中用  $\text{high}[i]$  和  $\text{low}[i]$ 
    // 分别存储分配给桶  $i$  的数中的最大数和最小数
    int *count=new int[n+1];
    double *low=new double[n+1];
    double *high=new double[n+1];
    // 桶初始化
    for(int i=1; i<=n-1; i++){
        count[i]=0;
        low[i]=maxx;
        high[i]=minx;
    }
    // 将  $n$  个数置于  $n-1$  个桶中
    for(i=1; i<=n; i++){
        int bucket=int((n-1)*(x[i]-minx)/(maxx-minx))+1;
        count[bucket]++;
        if(x[i]<low[bucket]) low[bucket]=x[i];
        if(x[i]>high[bucket]) high[bucket]=x[i];
    }
    // 此时, 除了 maxx 和 minx 外的  $n-2$  个数被置于  $n-1$  个桶中.
    // 由鸽舍原理即知, 至少有一个桶是空的.
    // 这意味着最大间隙不会出现在同一桶中的两个数之间.
    // 对每一个桶做一次线性扫描即可找出最大间隙.
    double tmp=0, left=high[1];
    for(i=2; i<=n-1; i++){
        if(count[i]){
            double thisgap=low[i]-left;
            if(thisgap>tmp) tmp=thisgap;
            left=high[i];
        }
    }
    return tmp;
}
```

其中,mini 和 maxi 分别计算数组中最小元素和最大元素的下标。

```
template<class T>
int mini(int n,T*x)
{
    T tmp=x[1];
    for(int i=1,k=1;i<=n;i++)
        if(x[i]<tmp){tmp=x[i];k=i;}
    return k;
}

template<class T>
int maxi(int n,T*x)
{
    T tmp=x[1];
    for(int i=1,k=1;i<=n;i++)
        if(x[i]>tmp){tmp=x[i];k=i;}
    return k;
}
```

由于下取整函数耗时 $O(1)$, 故循环体内的运算耗时 $O(1)$ 。因此, 整个算法耗时 $O(n)$ 。即算法 maxgap 是求最大间隙问题的线性时间算法。注意到在代数判定树计算模型下, $\Omega(n \log n)$ 是最大间隙问题的一个计算时间下界。这意味着在代数判定树的计算模型下, 最大间隙问题是不可能有线性时间算法的。在此题中假设下取整函数耗时 $O(1)$, 实际上这可以看作是在代数判定树模型中, 将下取整运算作为基本运算增加到原有的基本运算集中, 从而使代数判定树计算模型的计算能力得到增强。因而可以在线性时间内解最大间隙问题。

第2章 递归与分治策略

习题 2-1 Hanoi 塔问题的非递归算法

证明 Hanoi 塔问题的递归算法与非递归算法实际上是一回事。

分析与解答:

Hanoi 塔问题的递归算法:

```
void hanoi(int n, int A, int B, int C)
{
    if(n>0){
        hanoi(n-1, A, C, B);
        move(n, A, B);
        hanoi(n-1, C, B, A);
    }
}
```

主教材中所述非递归算法的目的塔座不确定。当 n 为奇数时, 目的塔座是 B , 按顺时针方向移动; 而当 n 为偶数时, 目的塔座为 C , 按反时针方向移动。为确定起见, 规定目的塔座为 B 。Hanoi 塔问题的非递归算法可描述如下。

```
void hanoi(int n)
{
    int top[3]={0,0,0};
    int **tower;
    Make2DArray(tower,n+1,3);
    int b,bb,x,y,min=0;
    for(int i=0;i<=n;i++){tower[i][0]=n-i+1;tower[i][1]=n+1;tower[i][2]=n+1;}
    top[0]=n;b=odd(n);bb=1;
    while(top[1]<n){
        if(bb){
            x=min;
            if(b)y=(x+1)%3;else y=(x+2)%3;
            min=y;bb=0;
        }
        else{
            x=(min+1)%3;y=(min+2)%3;bb=1;
            if(tower[top[x]][x]>tower[top[y]][y])Swap(x,y);
        }
        move(tower[top[x]][x],x+1,y+1);
        tower[top[y]+1][y]=tower[top[x]][x];
    }
```

```

        top[x]--;top[y]++;
    }
}

```

下面用数学归纳法证明递归算法和非递归算法产生相同的移动序列。

当 $n=1$ 和 $n=2$ 时容易直接验证。设当 $k \leq n-1$ 时，递归算法和非递归算法产生完全相同的移动序列。考察 $k=n$ 的情形。

将移动分为顺时针移动(C)、逆时针移动(CC)和非最小圆盘塔座间的移动(O)三种情况。

当 n 为奇数时，顺时针非递归算法产生的移动序列为 C,O,C,O,...,C；逆时针非递归算法产生的移动序列为 CC,O,CC,O,...,CC。

当 n 为偶数时，顺时针非递归算法产生的移动序列为 CC,O,CC,O,...,CC；逆时针非递归算法产生的移动序列为 C,O,C,O,...,C。

(1) 当 n 为奇数时，顺时针递归算法 $\text{hanoi}(n,A,B,C)$ 产生的移动序列为

$\text{hanoi}(n-1,A,C,B)$ 产生的移动序列, O, $\text{hanoi}(n-1,C,B,A)$ 产生的移动序列

其中, $\text{hanoi}(n-1,A,C,B)$ 和 $\text{hanoi}(n-1,C,B,A)$ 均为偶数圆盘逆时针移动问题。由数学归纳法知，它们产生的移动序列均为 C,O,C,O,...,C。因此， $\text{hanoi}(n,A,B,C)$ 产生的移动序列为 C,O,C,O,...,C。

(2) 当 n 为偶数时，顺时针递归算法 $\text{hanoi}(n,A,B,C)$ 产生的移动序列为

$\text{hanoi}(n-1,A,C,B)$ 产生的移动序列, O, $\text{hanoi}(n-1,C,B,A)$ 产生的移动序列

其中, $\text{hanoi}(n-1,A,C,B)$ 和 $\text{hanoi}(n-1,C,B,A)$ 均为奇数圆盘逆时针移动问题。由数学归纳法知，它们产生的移动序列均为 CC,O,CC,O,...,CC。因此， $\text{hanoi}(n,A,B,C)$ 产生的移动序列为 CC,O,CC,O,...,CC。

当 n 为奇数和偶数时的逆时针递归算法也类似。

由数学归纳法可知，递归算法和非递归算法产生相同的移动序列。

习题 2-2 7 个二分搜索算法

下面的 7 个算法与主教材第 2 章中的二分搜索算法 BinarySearch 略有不同。请判断这 7 个算法的正确性。如果算法不正确，请说明产生错误的原因。如果算法正确，请给出算法的正确性证明。

```

(1) template<class Type>
int BinarySearch1(Type a[], const Type& x, int n)
{
    int left = 0; int right = n - 1;
    while (left <= right) {
        int middle = (left + right) / 2;
        if (x == a[middle]) return middle;
        if (x > a[middle]) left = middle;
        else right = middle;
    }
    return -1;
}

```

}

```
(2) template<class Type>
int BinarySearch2(Type a[], const Type& x, int n)
{
    int left = 0; int right = n - 1;
    while (left < right - 1) {
        int middle = (left + right) / 2;
        if (x < a[middle]) right = middle;
        else left = middle;
    }
    if (x == a[left]) return left;
    else return - 1;
}
```

```
(3) template<class Type>
int BinarySearch3(Type a[], const Type& x, int n)
{
    int left = 0; int right = n - 1;
    while (left + 1 != right) {
        int middle = (left + right) / 2;
        if (x >= a[middle]) left = middle;
        else right = middle;
    }
    if (x == a[left]) return left;
    else return - 1;
}
```

```
(4) template<class Type>
int BinarySearch4(Type a[], const Type& x, int n)
{
    if (n > 0 && x >= a[0]) {
        int left = 0; int right = n - 1;
        while (left < right) {
            int middle = (left + right) / 2;
            if (x < a[middle]) right = middle - 1;
            else left = middle;
        }
        if (x == a[left]) return left;
    }
    return - 1;
}
```

```

    }

(5) template<class Type>
int BinarySearch5(Type a[], const Type& x, int n)
{
    if (n>0&& x>=a[0]) {
        int left =0; int right=n-1;
        while (left <right) {
            int middle =(left +right+1)/2;
            if (x <a[middle]) right=middle-1;
            else left =middle;
        }
        if (x==a[left]) return left;
    }
    return -1;
}

```

```

(6) template<class Type>
int BinarySearch6(Type a[], const Type& x, int n)
{
    if (n>0&& x>=a[0]) {
        int left =0; int right=n-1;
        while (left <right) {
            int middle =(left +right+1)/2;
            if (x <a[middle]) right=middle-1;
            else left =middle+1;
        }
        if (x==a[left]) return left;
    }
    return -1;
}

```

```

(7) template<class Type>
int BinarySearch7(Type a[], const Type& x, int n)
{
    if (n>0&& x>=a[0]) {
        int left =0; int right=n-1;
        while (left <right) {
            int middle =(left +right+1)/2;
            if (x <a[middle]) right=middle;
            else left =middle;
        }
    }
}

```

```

        if (x == a[left]) return left;
    }
    return -1;
}

```

分析与解答:

(1) 与主教材中的算法 BinarySearch 相比, 数组段左、右游标 left 和 right 的调整不正确, 导致陷入死循环。

(2) 与主教材中的算法 BinarySearch 相比, 数组段左、右游标 left 和 right 的调整不正确, 导致当 $x = a[n-1]$ 时返回错误。

(3) 与正确算法 BinarySearch5 相比, 数组段左、右游标 left 和 right 的调整不正确, 导致当 $x = a[n-1]$ 时返回错误。

(4) 与正确算法 BinarySearch5 相比, 数组段左、右游标 left 和 right 的调整不正确, 导致陷入死循环。

(5) 算法正确, 且当数组中有重复元素时, 返回满足条件的最右元素。

(6) 与正确算法 BinarySearch5 相比, 数组段左、右游标 left 和 right 的调整不正确, 导致当 $x = a[n-1]$ 时返回错误。

(7) 与正确算法 BinarySearch5 相比, 数组段左、右游标 left 和 right 的调整不正确, 导致当 $x = a[0]$ 时陷入死循环。

习题 2-3 改写二分搜索算法

设 $a[0:n-1]$ 是已排好序的数组。请改写二分搜索算法, 使得当搜索元素 x 不在数组中时, 返回小于 x 的最大元素位置 i 和大于 x 的最小元素位置 j 。当搜索元素在数组中时, i 和 j 相同, 均为 x 在数组中的位置。

分析与解答:

改写二分搜索算法如下。

```

template<class T>
int binarySearch(T a[], const T&x, int left, int right, int &i, int &j)
{
    int middle;
    while (left <= right) {
        middle = (left + right) / 2;
        if (x == a[middle]) {i = j = middle; return 1;}
        if (x > a[middle]) left = middle + 1;
        else right = middle - 1;
    }
    i = right; j = left;
    return 0;
}

```

习题 2-4 大整数乘法的 $O(nm^{\log(3/2)})$ 算法

给定 2 个大整数 u 和 v , 它们分别有 m 位和 n 位数字, 且 $m \leq n$ 。用通常的乘法求 uv 的值需要 $O(mn)$ 时间。可以将 u 和 v 均看作是有 n 位数字的大整数, 用主教材第 2 章介绍的分治法, 在 $O(n^{\log 3})$ 时间内计算 uv 的值。当 m 比 n 小得多时, 用这种方法就显得效率不够高。试设计一个算法, 在上述情况下用 $O(nm^{\log(3/2)})$ 时间求出 uv 的值。

分析与解答:

当 m 比 n 小得多时, 将 v 分成 n/m 段, 每段 m 位。计算 uv 需要 n/m 次 m 位乘法运算。每次 m 位乘法可以用主教材中的分治法计算, 耗时 $O(m^{\log 3})$ 。因此, 算法所需的计算时间为 $O((n/m)m^{\log 3}) = O(nm^{\log(3/2)})$ 。

习题 2-5 5 次 $n/3$ 位整数的乘法

在用分治法求 2 个 n 位大整数 u 和 v 的乘积时, 将 u 和 v 都分割为长度为 $n/3$ 位的 3 段。证明可以用 5 次 $n/3$ 位整数的乘法求得 uv 的值。按此思想设计一个求 2 个大整数乘积的分治算法, 并分析算法的计算复杂性(提示: n 位的大整数除以一个常数 k 可以在 $\theta(n)$ 时间内完成。符号 θ 所隐含的常数可能依赖于 k)。

分析与解答:

这个问题有更一般的解。将 2 个 n 位大整数 u 和 v 都分割为长度为 n/m 位的 m 段, 可以用 $2m-1$ 次 n/m 位整数的乘法求得 uv 的值。

事实上, 设 $x=2^{n/m}$, 可以将 u 和 v 及其乘积 $w=uv$ 表示为

$$u = u_0 + u_1x + \cdots + u_{m-1}x^{m-1}$$

$$v = v_0 + v_1x + \cdots + v_{m-1}x^{m-1}$$

$$w = uv = w_0 + w_1x + w_2x^2 + \cdots + w_{2m-2}x^{2m-2}$$

将 u, v 和 w 都看作关于变量 x 的多项式, 并取 $2m-1$ 个不同的数 $x_1, x_2, \dots, x_{2m-1}$ 代入多项式可得

$$u(x_i) = u_0 + u_1x_i + \cdots + u_{m-1}x_i^{m-1}$$

$$v(x_i) = v_0 + v_1x_i + \cdots + v_{m-1}x_i^{m-1}$$

$$w(x_i) = u(x_i)v(x_i) = w_0 + w_1x_i + w_2x_i^2 + \cdots + w_{2m-2}x_i^{2m-2}$$

用矩阵形式表示为

$$\begin{bmatrix} w(x_1) \\ w(x_2) \\ \vdots \\ w(x_{2m-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{2m-2} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{2m-2} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{2m-1} & x_{2m-1}^2 & \cdots & x_{2m-1}^{2m-2} \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{2m-2} \end{bmatrix}$$

设

$$B = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{2m-2} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{2m-2} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{2m-1} & x_{2m-1}^2 & \cdots & x_{2m-1}^{2m-2} \end{bmatrix}$$

则

$$\begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{2m-2} \end{bmatrix} = B^{-1} \begin{bmatrix} w(x_1) \\ w(x_2) \\ \vdots \\ w(x_{2m-1}) \end{bmatrix}$$

式中, $w(x_i) = u(x_i)v(x_i)$ 是 2 个 n/m 位数的乘法运算, 共有 $2m-1$ 个乘法。其他均为加减法或数乘运算。

下面用 $m=3$ 的具体例子来说明。

设 $x=2^{n/3}$, 可以将 u 和 v 及其乘积 $w=uv$ 表示为

$$u = u_0 + u_1x + u_2x^2$$

$$v = v_0 + v_1x + v_2x^2$$

$$w = uv = w_0 + w_1x + w_2x^2 + w_3x^3 + w_4x^4$$

取 5 个数 x_1, x_2, \dots, x_5 如下:

$$x_1=0, x_2=-2, x_3=2, x_4=-1, x_5=1$$

代入多项式得

$$a = w(x_1) = u_0v_0$$

$$b = w(x_2) = (u_0 - 2u_1 + 4u_2)(v_0 - 2v_1 + 4v_2)$$

$$c = w(x_3) = (u_0 + 2u_1 + 4u_2)(v_0 + 2v_1 + 4v_2)$$

$$d = w(x_4) = (u_0 - u_1 + u_2)(v_0 - v_1 + v_2)$$

$$e = w(x_5) = (u_0 + u_1 + u_2)(v_0 + v_1 + v_2)$$

$$\begin{bmatrix} a \\ b \\ c \\ d \\ e \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & -2 & 4 & -8 & 16 \\ 1 & 2 & 4 & 8 & 16 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix}$$

$$\begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & -2 & 4 & -8 & 16 \\ 1 & 2 & 4 & 8 & 16 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}^{-1} \begin{bmatrix} a \\ b \\ c \\ d \\ e \end{bmatrix}$$

解得

$$w_0 = a$$

$$w_1 = \frac{b - c - 8(d - e)}{12}$$

$$w_2 = \frac{-b - c + 16(d + e) - 30a}{24}$$

$$w_3 = \frac{-b+c+2(d-e)}{12}$$

$$w_4 = \frac{b+c-4(d+e)+6a}{24}$$

由 x_1, x_2, \dots, x_5 的不同取法, 可得到不同的分解方法。

按此分解设计的求 2 个 n 位大整数乘积的分治算法需要 5 次 $n/3$ 位整数乘法。分割及合并步所需的加减法和数乘运算时间为 $O(n)$ 。设 $T(n)$ 是算法所需的计算时间, 则

$$T(n) = \begin{cases} O(1) & n=1 \\ 5T(n/3) + O(n) & n>1 \end{cases}$$

由此可得 $T(n) = O(n^{\log_3 5})$ 。

在一般情况下, 将 2 个 n 位大整数 u 和 v 都分割为长度为 n/m 位的 m 段, 可以用 $2m-1$ 次 n/m 位整数的乘法求得 uv 的值。由此设计出的求 2 个 n 位大整数乘积的分治算法需要 $2m-1$ 次 n/m 位整数乘法。分割及合并步所需的加减法和数乘运算时间为 $O(n)$ 。因此其计算时间 $T(n)$ 满足:

$$T(n) = \begin{cases} O(1) & n=1 \\ (2m-1)T(n/m) + O(n) & n>1 \end{cases}$$

解此递归式可得 $T(n) = O(n^{\log_m (2m-1)})$ 。

习题 2-6 矩阵乘法

对任何非零偶数 n , 总可以找到奇数 m 和正整数 k , 使得 $n = m2^k$ 。为了求出 2 个 n 阶矩阵的乘积, 可以把一个 n 阶矩阵分成 $m \times m$ 个子矩阵, 每个子矩阵有 $2^k \times 2^k$ 个元素。当需要求 $2^k \times 2^k$ 的子矩阵的积时, 使用 Strassen 算法。设计一个传统方法与 Strassen 算法相结合的矩阵相乘算法, 对任何偶数 n , 都可以求出 2 个 n 阶矩阵的乘积。并分析算法的计算时间复杂性。

分析与解答:

将 n 阶矩阵分块为 $m \times m$ 的矩阵。用传统方法求 2 个 m 阶矩阵的乘积需要计算 $O(m^3)$ 次 $2^k \times 2^k$ 矩阵的乘积。用 Strassen 矩阵乘法计算 2 个 $2^k \times 2^k$ 矩阵的乘积需要的计算时间为 $O(7^k)$, 因此算法的计算时间为 $O(7^k m^3)$ 。

习题 2-7 多项式乘积

设 $P(x) = a_0 + a_1x + \dots + a_dx^d$ 是一个 d 次多项式。假设已有一个算法能在 $O(i)$ 时间内计算一个 i 次多项式与一个一次多项式的乘积, 以及一个算法能在 $O(i \log i)$ 时间内计算 2 个 i 次多项式的乘积。对于任意给定的 d 个整数 n_1, n_2, \dots, n_d , 用分治法设计一个有效算法, 计算出满足 $P(n_1) = P(n_2) = \dots = P(n_d) = 0$ 且最高次项系数为 1 的 d 次多项式 $P(x)$, 并分析算法的效率。

分析与解答:

$$P(x) = \prod_{i=1}^d (x - n_i) = \prod_{i=1}^{\lfloor d/2 \rfloor} (x - n_i) \prod_{i=\lfloor d/2 \rfloor + 1}^d (x - n_i) = P_1(x)P_2(x)$$

用分治法将 d 次多项式转化为 2 个 $d/2$ 次多项式的乘积。

设用此算法计算 d 次多项式所需计算时间为 $T(d)$, 则 $T(d)$ 满足如下递归式:

$$T(d) = \begin{cases} O(1) & d=1 \\ 2T(d/2) + O(d \log d) & d>1 \end{cases}$$

解此递归式可得 $T(d) = O(d \log^2 d)$ 。

习题 2-8 不动点问题的 $O(\log n)$ 时间算法

设 n 个不同的整数排好序后存于 $T[1:n]$ 中。若存在一个下标 i , $1 \leq i < n$, 使得 $T[i] = i$, 设计一个有效算法找到这个下标。要求算法在最坏情况下的计算时间为 $O(\log n)$ 。

分析与解答:

见参考文献[1], 41。

习题 2-9 主元素问题的线性时间算法

设 $T[0:n-1]$ 是 n 个元素的数组。对任一元素 x , 设 $S(x) = \{i | T[i] = x\}$ 。当 $|S(x)| > n/2$ 时, 称 x 为 T 的主元素。设计一个线性时间算法, 确定 $T[0:n-1]$ 是否有一个主元素。

分析与解答:

见参考文献[1], 41~42。

习题 2-10 无序集主元素问题的线性时间算法

若在习题 2-9 中, 数组 T 中元素不存在序关系, 只能测试任意 2 个元素是否相等, 试设计一个有效算法确定 T 是否有一主元素。算法的计算复杂性应为 $O(n \log n)$ 。更进一步, 能找到一个线性时间算法吗?

分析与解答:

见参考文献[1], 42~45。

习题 2-11 $O(1)$ 空间子数组换位算法

设 $a[0:n-1]$ 是一个有 n 个元素的数组, k ($1 \leq k \leq n-1$) 是一个非负整数。试设计一个算法将子数组 $a[0:k-1]$ 与 $a[k+1:n-1]$ 换位。要求算法在最坏情况下耗时 $O(n)$, 且只用到 $O(1)$ 的辅助空间。

分析与解答:

算法 1: 循环换位算法

(1) 向前循环换位

```
template<class T>
void forward(T a[], int n, int k)
{
    for(int i=0; i<k; i++){
        T tmp=a[0];
        for(int j=1; j<n; j++) a[j-1]=a[j];
        a[n-1]=tmp;
    }
}
```

(2) 向后循环换位

```
template<class T>
void backward(T a[], int n, int k)
{
    for(int i=k; i<n; i++) {
        T tmp=a[n-1];
        for(int j=n-1; j>0; j--) a[j]=a[j-1];
        a[0]=tmp;
    }
}
```

(3) 选择较小的数组块进行循环

```
template<class T>
void exch0(T a[], int n, int k)
{
    if(k>n-k) backward(a, n, k);
    else forward(a, n, k);
}
```

在最坏情况下, 算法所需的元素移动次数为 $\min\{k, n-k\} \cdot (n+1)$ 。算法仅用到一个辅助单元 tmp, 因此算法只用到 $O(1)$ 的辅助空间。当 $k=n/2$ 时, 计算时间非线性。

算法 2:3 次反转算法

将数组块 $a[i:j]$ 反转的算法如下。

```
template<class T>
void reverse(T a[], int i, int j)
{
    while(i<j) {Swap(a[i], a[j]); i++; j--;}
}
```

设 $a[0:k-1]$ 为 U , $a[k:n-1]$ 为 V , 换位算法要求将 UV 变换为 VU 。3 次反转算法先将 U 反转为 U^{-1} , 再将 V 反转为 V^{-1} , 最后将 $U^{-1}V^{-1}$ 反转为 VU 。

```
template<class T>
void exch1(T a[], int n, int k)
{
    reverse(a, 0, k-1);
    reverse(a, k, n-1);
    reverse(a, 0, n-1);
}
```

3 次反转算法用了 $\lfloor k/2 \rfloor + \lfloor (n-k)/2 \rfloor + \lfloor n/2 \rfloor \leq n$ 次数组单元交换运算。每个数组单元交

换运算需要 3 次元素移动。因此在最坏情况下, 3 次反转算法用了 $3n$ 次元素移动。算法显然只用到 $O(1)$ 的辅助空间。

算法 3: 排列循环算法

向后循环换位算法实际上执行了数组元素的一个重新排列。因此向后循环换位对应于 n 个元素的一个置换。这类置换具有如下的特殊性质。

循环置换分解定理: 对于给定数组 $a[0:n-1]$ 向后循环换位 $n-k$ 位运算, 可分解为恰好 $\gcd(k, n-k)$ 个循环置换, 且 $0, \dots, \gcd(k, n-k)-1$ 中每个数恰属于一个循环置换。其中 $\gcd(x, y)$ 表示整数 x 和 y 的最大公因数。

基于循环置换分解定理可设计下面的数组块换位算法。

```
template<class T>
void exch(T a[], int n, int k)
{
    for(int i=0, cyc=gcd(k, n-k); i<cyc; i++){
        T tmp=a[i];
        int p=i, j=(k+i)%n;
        while(j!=i){a[p]=a[j]; p=j; j=(k+p)%n;}
        a[p]=tmp;
    }
}
```

上述算法总共移动元素 $n+\gcd(k, n-k)$ 次, 算法显然只用到 $O(1)$ 的辅助空间。

当换位数组块的长度相等时, 算法更简单。例如, 将数组块 $a[b:b+1-1]$ 和 $a[c:c+1-1]$ 换位的算法可表述如下。

```
template<class T>
void eqexch(T a[], int b, int c, int l)
{
    // equal size block exchange a[b:b+1-1] and a[c:c+1-1].
    for(int i=0; i<l; i++) Swap(a[b+i], a[c+i]);
}
```

上述算法总共移动元素 $3 \times l$ 次。

习题 2-12 $O(1)$ 空间合并算法

设子数组 $a[0:k-1]$ 和 $a[k:n-1]$ 已排好序 ($0 \leq k \leq n-1$)。试设计一个合并这 2 个子数组为排好序的数组 $a[0:n-1]$ 的算法。要求算法在最坏情况下所用的计算时间为 $O(n)$, 且只用到 $O(1)$ 的辅助空间。

分析与解答:

算法 1: 循环换位合并算法

(1) 向右循环换位合并

向右循环换位合并算法首先用二分搜索算法在数组段 $a[k:n-1]$ 中搜索 $a[0]$ 的插入位置, 即找到位置 p 使得 $a[p] < a[0] \leq a[p+1]$ 。数组段 $a[0:p]$ 向右循环换位 $p-k+1$ 个位

置, 使 $a[k-1]$ 移动到 $a[p]$ 的位置。此时, 原数组元素 $a[0]$ 及其左边的所有元素均已排好序。对剩余的数组元素重复上述过程, 直至只剩下一个数组段, 此时整个数组已排好序。

向右循环换位合并算法可描述如下。

```
template<class T>
void mergefor(T a[], int k, int n)
{ // Merge a[0:k-1] and a[k:n-1].
    int i=0, j=k;
    while(i<j && j<n){
        int p=binarySearch(a, a[i], j, n-1);
        shiftright(a, i, p, p-j+1);
        j=p+1; i+=p-j+2;
    }
}
```

其中, 算法 $\text{binarySearch}(a, x, \text{left}, \text{right})$ 用于在数组段 $a[\text{left}:\text{right}]$ 中搜索元素 x 的插入位置。

```
template<class T>
int binarySearch(T a[], const T& x, int left, int right)
{
    int middle;
    while (left <= right) {
        middle = (left + right) / 2;
        if (x == a[middle]) return middle;
        if (x > a[middle]) left = middle + 1;
        else right = middle - 1;
    }
    if (x > a[middle]) return middle;
    else return middle - 1;
}
```

算法 $\text{shiftright}(a, s, t, k)$ 用于将数组段 $a[s:t]$ 中元素循环右移位 k 个位置。

```
template<class T>
void shiftright(T a[], int s, int t, int k)
{
    for(int i=0; i<k; i++){
        T tmp=a[t];
        for(int j=t; j>s; j--) a[j]=a[j-1];
        a[s]=tmp;
    }
}
```

上述算法中,数组段 $a[0:k-1]$ 中元素的移动次数不超过 k 次,数组段 $a[k:n-1]$ 中元素最多移动 1 次。因此,算法的元素移动总次数不超过 $k^2 + (n-k)$ 次。算法的元素比较次数不超过 $k \log(n-k)$ 次。当 $k \leq \sqrt{n}$ 时,算法的计算时间为 $O(n)$ 。而当 $k = O(n)$ 时,算法的计算时间为 $O(n^2)$ 。由于数组段循环右移位算法只用了 $O(1)$ 的辅助空间,所以整个算法所用的辅助空间为 $O(1)$ 。

(2) 类似地,可以设计向左循环换位合并算法 $\text{mergeback}(T\ a[], \text{int } k, \text{int } n)$ 。

算法 2: 内部缓存算法

(1) 算法思想概述

为便于叙述,假定 n 是一个完全平方数,稍后讨论一般情况。图 2-1 (a) 是当 $n=25$, $k=12$ 时的一个示例。其中用大写字母表示数组元素的键值,其下标表示相同键值出现的次序。

首先将待合并数组划分为 \sqrt{n} 个数组块,每块的大小为 \sqrt{n} 。将数组中最大的 \sqrt{n} 个元素置于数组的最左端,作为算法的内部缓存,如图 2-1 (b) 所示。接下来用选择排序算法对除了内部缓存外的 $\sqrt{n}-1$ 个数组块排序,使数组块的最右端元素按非减序排列,每个数组块内部元素的相对次序保持不变。这个排序过程需要用 $O(n)$ 次元素比较和 $O(n)$ 次元素移动。用习题 2-11 中的算法 eqexch 可保证上述排序过程只用 $O(1)$ 的辅助空间。数组块排序后的结果如图 2-1 (c) 所示。

$B_1 B_2 B_3 D_1 D_2 E_1 E_2 F_1 G_1 H_1 H_2 J_1$	$A_1 A_2 A_3 B_4 B_5 C_1 C_2 E_3 G_2 G_3 H_3 I_1 I_2$
0	$k-1$
	k
	$n-1$

(a) 给定的待合并数组

$H_2 H_3 I_1 I_2 J_1$	$B_1 B_2 B_3 D_1 D_2$	$E_1 E_2 F_1 G_1 H_1$	$A_1 A_2 A_3 B_4 B_5$	$C_1 C_2 E_3 G_2 G_3$
内部缓存	数组段 1		数组段 2	

(b) 抽取内部缓存

$H_2 H_3 I_1 I_2 J_1$	$A_1 A_2 A_3 B_4 B_5$	$B_1 B_2 B_3 D_1 D_2$	$C_1 C_2 E_3 G_2 G_3$	$E_1 E_2 F_1 G_1 H_1$
内部缓存	数组块 2	数组块 3	数组块 4	数组块 5

(c) 数组块排序

图 2-1 数组块重排结果

接下来要确定待合并的数组块序列。第 1 个序列是从数组块 2 开始的最长的非减数组块序列。第 2 个序列是接着的那个数组块,如图 2-2 (a) 所示。

现在可以用内部缓存对两个序列进行合并。每次比较两个序列的最小元素,将较小者与内部缓存的最左端元素交换位置。合并过程中内部缓存可能被分为两个不连续的段,如图 2-2 (b) 所示。当第 1 个序列的最后一个元素进入正确位置后,完成这一轮序列合并动作,此时内部缓存形成一个连续的块,且第 2 个序列中至少还有 1 个元素,如图 2-2 (c) 所示。

下一轮的序列合并是类似的。第 1 个序列是从内部缓存右端的下一个元素开始的最长的非减数组块序列。第 2 个序列是接着的那个数组块,如图 2-3 (a) 所示。继续用内部缓存合并这两个序列,直至第 1 个序列完成合并,如图 2-3 (b) 所示。上述过程一直进行到只剩下一个序列时为止。此时只要将剩下的这个序列左移,内部缓存成为最后的一个数组块,如图 2-3 (c) 所示。此时内部缓存左边的所有元素均已排好序,且内部缓存中的元素是整个数组中最大的 \sqrt{n} 个元素,用选择排序算法对内部缓存中元素排序,完成整个合并过程,

如图 2-3(d)所示。

$H_2 H_3 I_1 I_2 J_1$	$A_1 A_2 A_3 B_4 B_5$	$B_1 B_2 B_3 D_1 D_2$	$C_1 C_2 E_3 G_2 G_3$	$E_1 E_2 F_1 G_1 H_1$
内部缓存	待合并序列 1		待合并序列 2	数组块 5

(a) 确定待合并序列

$A_1 A_2 A_3 B_4 B_5 B_1 B_2 B_3 C_1 C_2$	$H_2 H_3 I_1$	$D_1 D_2$	$I_2 J_1$	$E_3 G_2 G_3$	$E_1 E_2 F_1 G_1 H_1$
已合并序列	缓存		缓存		数组块 5

(b) 内部缓存合并过程

$A_1 A_2 A_3 B_4 B_5 B_1 B_2 B_3 C_1 C_2 D_1 D_2$	$I_1 H_2 H_3 I_2 J_1$	$E_3 G_2 G_3$	$E_1 E_2 F_1 G_1 H_1$
已合并序列	内部缓存		数组块 5

(c) 完成序列合并

图 2-2 合并两个序列

$A_1 A_2 A_3 B_4 B_5 B_1 B_2 B_3 C_1 C_2 D_1 D_2$	$I_1 H_2 H_3 I_2 J_1$	$E_3 G_2 G_3$	$E_1 E_2 F_1 G_1 H_1$
已合并序列	内部缓存	待合并序列 1	待合并序列 2

(a) 确定下一对待合并序列

$A_1 A_2 A_3 B_4 B_5 B_1 B_2 B_3 C_1 C_2 D_1 D_2 E_3 E_1 E_2 F_1 G_2 G_3$	$J_1 I_1 H_2 H_3 I_2$	$G_1 H_1$
	内部缓存	

(b) 完成序列合并

$A_1 A_2 A_3 B_4 B_5 B_1 B_2 B_3 C_1 C_2 D_1 D_2 E_3 E_1 E_2 F_1 G_2 G_3 G_1$	$H_1 H_2 H_3 I_2 J_1 I_1$
	内部缓存

(c) 移位完成单序列合并

$A_1 A_2 A_3 B_4 B_5 B_1 B_2 B_3 C_1 C_2 D_1 D_2 E_3 E_1 E_2 F_1 G_2 G_3 G_1 H_1 H_2 H_3 I_2 I_1 J_1$

(d) 对内部缓存排序,完成整个数组合并

图 2-3 整个数组的合并过程

由于合并过程只用到内部缓存,上述整个合并过程只用 $O(1)$ 的辅助空间。数组块排序、序列合并以及内部缓存排序显然只需要 $O(n)$ 计算时间。因此,内部缓存算法需要 $O(n)$ 计算时间和 $O(1)$ 的辅助空间。

(2) 一般情况

在一般情况下,可以用 $O(\sqrt{n})$ 时间将问题转换为前面讨论的特殊情况。

当待合并的 2 个数组段中有一个数组段的长度小于 \sqrt{n} 时,可以用前面讨论过的循环换位合并算法在 $O(n)$ 计算时间内,用 $O(1)$ 的辅助空间完成合并。

接下来的讨论中,设 $s = \lfloor \sqrt{n} \rfloor$, 且 $\min\{k, n-k\} > \sqrt{n}$ 。数组中大小为 s 的块记为 s -block。由于 $(s+1)^2 > n$, 数组中 s -block 的个数不超过 $s+2$ 。

首先,找出数组中组成内部缓存的 s 个最大元素。一般情况下,这 s 个元素由数组中大小分别为 s_1 和 s_2 的 2 部分 A 和 B 组成, $s_1 + s_2 = s$ 。紧挨着 A 的左边的 s_2 个元素记作 C 。 D 是紧挨着 B 的左边的数组块,其大小是使数组块 2 剩余元素为 s 的倍数的最小值。按此划分,可以将数组看作由 s -block 组成的数组块。除了第 1 块的大小 t_1 和最后一块的大小 t_2 外,每个数组块 s -block 的大小均为 s , 而且 $0 < t_1 \leq s$, $0 < t_2 < 2s$, 如图 2-4(a)所示。

接下来交换数组块 C 和 B , 使数组块 B 和 A 构成内部缓存。然后用内部缓存合并数组块 D 和 C , 构成排好序的数组块 E , 如图 2-4 (b) 所示。

当 $t_1 < s$ 时, 需要对最左端的 t_1 -block 进行特殊处理。设 F 是这个特殊的数组块, G 是紧挨着内部缓存的 s -block, 如图 2-4 (c) 所示。用内部缓存的最右 t_1 个单元将 F 和 G 合并得到 H 和 I , 如图 2-4 (d) 所示。将 H 与最左端的内部缓存 t_1 -block 交换, 使 H 已在最终位置, 且内部缓存连成一体, 如图 2-4 (e) 所示。最后, 将内部缓存与第 1 个 s -block 换位, 转化为前面讨论过的规则情形, 如图 2-4 (f) 所示。

以上这些处理只需要 $O(s) = O(\sqrt{n})$ 的计算时间和 $O(1)$ 的辅助空间。



图 2-4 对一般情况的处理

(3) 算法实现

上面讨论的算法可以分为 2 个阶段和 4 个子任务分别实现如下。

当 $k < \sqrt{n}$ 或 $n - k < \sqrt{n}$ 时, 可以用前面讨论过的循环换位合并算法 `mergefor` 和 `mergeback` 在 $O(n)$ 计算时间内, 用 $O(1)$ 的辅助空间完成合并。

```

template<class T>
void merge(T a[], int k, int n)
{
    // Merge a[0:k-1] and c[k:n-1].
    int s = (int) sqrt(n);
    if (k < s) { mergefor(a, k, n); return; }
    if (n - k < s) { mergeback(a, k, n); return; }
    int j = task1(a, k, n, s);
    eqexch(a, k - s, j, n - j);
    int bfs = k - s, bft = k - 1;
    int ds = j - (j - k) % s, dt = j - 1;

```



```

    task2(a, n, ds, dt);
    task3(a, k, n, s, bfs, bft);
    maintask(a, k, n, s, bfs, ds);
}

```

算法的第 1 阶段是初始化阶段, 分别由 task1, task2 和 task3 子任务来完成。第 2 阶段进入主算法, 由子任务 maintask 来完成。下面分别讨论。

算法 task1 完成抽取内部缓存的任务, 相应于图 2-4 (a), 其返回值是数组块 B 的最左元素的下标。

```

template<class T>
int task1(T a[], int k, int n, int s)
{
    int i=k-1, j=n-1;
    for(int t=0; t<s; t++){
        if(a[i]<a[j]) j--;
        else i--;
    }
    return j+1;
}

```

算法 task2 完成对最右数组块 E 的排序, 相应于图 2-4 (b)。

```

template<class T>
void task2(T a[], int n, int ds, int dt)
{
    if(ds>dt) return;
    selectionSort(a, ds, n-1);
}

```

算法 task3 完成对最左数组块的排序, 相应于图 2-4 (c) ~ 图 2-4 (f)。

```

template<class T>
void task3(T a[], int k, int n, int s, int &bfs, int &bft)
{
    int t1=k%s;
    bfs=bft-t1+1;
    bfmerge(a, bfs, bft, 0, t1-1, bft+1, bft+s);
    eqexch(a, 0, bft-t1+1, t1);
    eqexch(a, t1, bft-s+1, s);
    bfs=t1; bft=bfs+s-1;
}

```

其中, 用到的数组块交换算法 eqexch 在习题 2-11 中已有描述。用内部缓存进行合并的

算法 bfmerge 表述如下。

```
template<class T>
void bfmerge(T a[], int &bs, int bt, int s1, int t1, int s2, int t2)
{ // buffer Merge
    int ps1=s1;
    while(s1<=t1) {
        if(s2<=t2 && a[s1]>a[s2]) Swap(a[s2++], a[bs++]);
        else Swap(a[s1++], a[bs++]);
        if(bs==s2) bs=ps1;
    }
}
```

算法 maintask 是主算法，完成最后的合并任务。

```
template<class T>
void maintask(T a[], int k, int n, int s, int bfs, int ds)
{
    sortblock(a, bfs+s, ds-1, s);
    while(bfs<n-s) {
        int s1=bfs+s, t1=s1+(ds-bfs-1)%s;
        while(t1<ds && a[t1]<=a[t1+1]) t1+=s;
        if(t1>n-1) t1=n-1;
        int s2=t1+1, t2=t1+s;
        if(s2>ds) t2=n-1;
        bfmerge(a, bfs, bfs+s-1, s1, t1, s2, t2);
        if(s1>t1 && s2<=t2) {eqexch(a, bfs, s2, t2-s2-1); break;}
    }
    selectionSort(a, bfs, n-1);
}
```

sortblock 是用选择排序对数组块进行排序的算法。

```
template<class T>
int maxblock(T a[], int left, int r, int s)
{
    int pos=1;
    for (int i=2; i<=r; i++)
        if (a[left+pos*s-1] < a[left+i*s-1]) pos=i;
    return pos;
}

template<class T>
void sortblock(T a[], int left, int right, int s)
```

```

{
    int m=(right-left+1)/s;
    for (int r=m; r>1; r--) {
        int j=maxblock(a, left, r, s);
        if(j<r) eqexch(a, left+(j-1)*s, left+(r-1)*s, s);
    }
}

```

习题 2-13 \sqrt{n} 段合并排序算法

如果在合并排序算法的分割步骤中,将数组 $a[0:n-1]$ 划分为 $\lfloor \sqrt{n} \rfloor$ 个子数组,每个子数组中有 $O(\sqrt{n})$ 个元素。然后递归地对分割后的子数组进行排序,最后将所得到的 $\lfloor \sqrt{n} \rfloor$ 个排好序的子数组合并成所要求的排好序的数组 $a[0:n-1]$ 。设计一个实现上述策略的合并排序算法,并分析算法的计算复杂性。

分析与解答:

实现上述策略的合并排序算法如下。

```

template<class T>
void mergesort(T *a, int left, int right)
{
    if(left<right){
        int j=(int)sqrt(right-left+1);
        if(j>1){
            for(int i=0; i<j; i++) mergesort(a, left+i*j, left+(i+1)*j-1);
            mergesort(a, left+j*j, right);
        }
        mergeall(a, left, right);
    }
}

```

其中,算法 mergeall 合并 \sqrt{n} 个排好序的数组段。在最坏情况下,算法 mergeall 需要 $O(n \log n)$ 时间。因此上述算法所需的计算时间 $T(n)$ 满足:

$$T(n) = \begin{cases} O(1) & n \leq 1 \\ \sqrt{n}T(\sqrt{n}) & n > 1 \end{cases}$$

此递归式的解为 $T(n) = O(n \log n)$ 。

习题 2-14 自然合并排序算法

对所给元素存储于数组中和存储于链表中 2 种情形,写出自然合并排序算法。

分析与解答:

对于所给元素存储于数组中的情形,自然合并排序算法如下。

```

template<class T>
void sort(T *a0, int m)

```

```

{
    a=a0;
    n=m;
    b=new int[n];
    naturalmergesort();
}

void naturalmergesort()
{
    while (!mergeruns(a, b) &!mergeruns(b, a));
}

```

由 mergeruns 实际完成自然合并排序算法。

```

template<class T>
bool mergeruns(T *a, T *b)
{
    int i=0, k=0;
    bool asc=true;
    T x;
    while (i<n)
    {
        k=i;
        do x=a[i++]; while (i<n && x<=a[i]);
        while (i<n && x>=a[i]) x=a[i++];
        merge (a, b, k, i-1, asc);
        asc=!asc;
    }
    return k==0;
}

```

```

template<class T>
void merge(T *a, T *b, int lo, int hi, bool asc)
{
    int k=asc ? lo : hi;
    int c=asc ? 1 : -1;
    int i=lo, j=hi;
    while (i<=j) {
        if (a[i]<=a[j]) b[k]=a[i++];
        else b[k]=a[j--];
        k+=c;
    }
}

```

对于所给元素存储于链表中的情形，自然合并排序算法如下。
链表结构如下。

```
typedef struct node *link;
struct node { Item item; link next; };

link mergesort(link t)
{
    link a, b;
    QUEUE<link> Q;
    if (t == 0 || t->next == 0) return t;
    for (link u = t, v; t != 0; t = u) {
        while (u && u->next && u->item <= u->next->item) u = u->next;
        v = u; u = u->next; v->next = 0;
        Q.ENQUEUE(t);
    }
    Q.DEQUEUE(t);
    while (!Q.EMPTY()) {
        Q.ENQUEUE(t);
        Q.DEQUEUE(a);
        Q.DEQUEUE(b);
        t = merge(a, b);
    }
    return t;
}
```

算法 merge 实现已排序链表的合并。

```
link merge(link a, link b)
{
    link c, head;
    c = head = new node;
    while ((a != 0) && (b != 0))
        if (a->item < b->item)
            { c->next = a; c = a; a = a->next; }
        else
            { c->next = b; c = b; b = b->next; }
    c->next = (a == 0) ? b : a;
    return head->next;
}
```

习题 2-15 最大值和最小值问题的最优算法

给定数组 $a[0:n-1]$ ，试设计一个算法，在最坏情况下用 $\lceil 3n/2 - 2 \rceil$ 次比较找出

$a[0:n-1]$ 中元素的最大值和最小值。

分析与解答:

见参考文献[1], 147~148。

习题 2-16 最大值和次大值问题的最优算法

给定数组 $a[0:n-1]$, 试设计一个算法, 在最坏情况下用 $n + \lceil \log n \rceil - 2$ 次比较找出 $a[0:n-1]$ 中元素的最大值和次大值。

分析与解答:

见参考文献[1], 148~149。

习题 2-17 整数集合排序

设 S_1, S_2, \dots, S_k 是整数集合, 其中每个集合 $S_i (1 \leq i \leq k)$ 中整数取值范围是 1 到 n , 且 $\sum_{i=1}^k |S_i| = n$, 试设计一个算法在 $O(n)$ 时间内将 S_1, S_2, \dots, S_k 分别排序。

分析与解答:

用桶排序或基数排序算法思想。

习题 2-18 第 k 小元素问题的计算时间下界

试证明, 在最坏情况下, 求 n 个元素组成的集合 S 中的第 k 小元素至少需要 $n + \min(k, n-k+1) - 2$ 次比较。

分析与解答:

由于要建立下界, 不失一般性, 可设 S 中的 n 个元素互不相同。首先注意到, 要确定第 k 小元素 z , 就要确定 S 中其他元素与 z 的关系。对于 S 中每个元素 x , 必须确定 $x > z$ 或 $x < z$ 。

换句话说, 要建立 S 中元素与 z 的序关系树, 如图 2-5 所示。

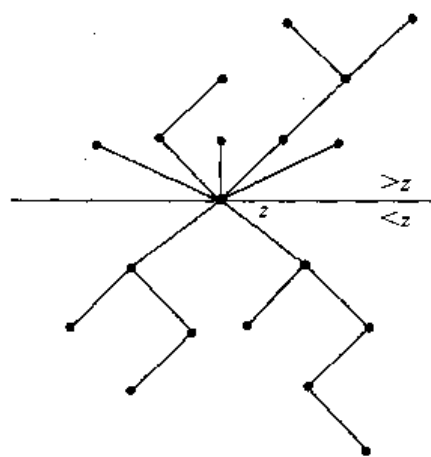


图 2-5 序关系树

图 2-5 中每个顶点表示一个元素, 每一条边表示一次比较。较高的元素的值也较大。如果有一个元素 y 与第 k 小元素 z 的大小关系不确定, 则对手可以改变 y 的值, 使其从 z 的一侧移向另一侧, 而不改变已做过比较的序关系, 从而改变了 z 的第 k 小的地位, 产生矛盾, 如图 2-6 所示。

由于关系树中有 n 个顶点, 因此有 $n-1$ 条边, 从而至少需要 $n-1$ 次比较。

下面进一步证明, 采用对手论证方法, 对手能迫使算法在得到所需的关系树中的 $n-1$ 次比较之前, 做其他“无用”的比较。

当 $y \geq z$ 时, 两元素 x 和 y 之间的第 1 次比较 $x > y$ 对应于关系树中的一条边。同理, 当 $y \leq z$ 时, 两元素 x 和 y 之间的第 1 次比较 $x < y$ 也对应于关系树中的一条边。这类比较称为关键比较。反之, 当 $x > z$ 且 $y < z$ 时, 两元素 x 和 y 之间的比较是非关键比较。

下面的对手策略, 将迫使算法做尽可能多的非关键比较。对手首先选定第 k 小元素 z 的值, 集合中其他元素的值未定, 仅在算法做与该元素有关的比较时确定该元素的值。在算法执行的任何阶段, 集合中的元素有以下 3 种状态:

L——该元素的值已确定，且大于 z ；

S——该元素的值已确定，且小于 z ；

N——该元素的值未确定。

算法做两个元素 x 和 y 的比较时，对手根据元素 x 和 y 的状态，按照表 2-1 中的对手策略确定状态为 N 的元素的值。

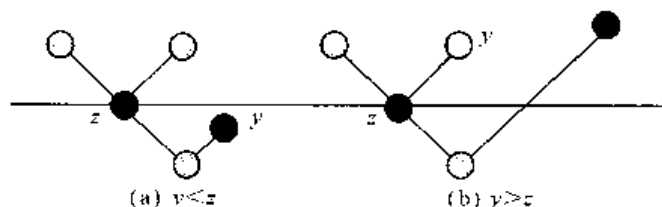


图 2-6 对手策略

表 2-1 对手策略

x	y	对手策略
N	N	x 取值大于 z , y 取值小于 z
N	L	x 取值小于 z
L	N	y 取值小于 z
N	S	x 取值大于 z
S	N	y 取值大于 z

对手策略中的每个比较均为非关键比较。每个这类比较最多产生一个元素状态 L，且最多产生一个元素状态 S。算法最终产生 $k-1$ 个状态为 S 的元素和 $n-k$ 个状态为 L 的元素。因此，上述对手策略至少迫使算法做了 $\min\{k-1, n-k\}$ 次非关键比较。因此任何算法至少做了 $n-1+\min\{k-1, n-k-1\}$ 次比较。由此可见，在最坏情况下，第 k 小元素问题至少需要 $n+\min(k, n-k+1)-2$ 次比较。

当 n 是奇数时，集合 S 的中位数是第 $(n+1)/2$ 小元素。由上述结论可知， $3n/2-3/2$ 是中位数问题的一个计算时间下界。

习题 2-19 非增序快速排序算法

如何修改 QuickSort 才能使其将输入元素按非增序排序？

分析与解答：

将算法 Partition 中的不等号反向即可。

```
template<class T>
int Partition (T a[], int p, int r)
{
    int i=p, j=r+1;
    T x=a[p];
    while (true) {
        // 将>x 的元素交换到左边区域
        while (a[++i]>x && i<r);
        // 将<x 的元素交换到右边区域
        while (a[--j]<x);
        if (i>=j) break;
        Swap(a[i], a[j]);
    }
    Swap(a[j], a[p]);
    return j;
}
```

习题 2-20 随机化算法

对一个随机化算法,为什么我们只需分析其平均情况下的性能,而不分析其最坏情况下的性能?

分析与解答:

在随机化算法中,出现最坏情况的实例往往是小概率事件。

习题 2-21 随机化快速排序算法

在执行 RandomizedQuicksort 时,在最坏情况下,调用 Random 多少次?在最好情况下又怎样?

分析与解答:

见参考文献[1], 90。

习题 2-22 随机排列算法

试设计一个 $O(n)$ 时间算法,使之能产生数组 $a[0:n-1]$ 元素的一个随机排列。

分析与解答:

见参考文献[1], 91。

习题 2-23 算法 QuickSort 中的尾递归

试用 while 循环消去算法 QuickSort 中的尾递归,并比较消去尾递归前后算法的效率。

分析与解答:

见参考文献[1], 88~90。

习题 2-24 用栈模拟递归

试用栈来模拟递归,消去算法 QuickSort 中的递归。并证明所需的栈空间为 $O(\log n)$ 。

分析与解答:

见参考文献[1], 88~90。

习题 2-25 算法 Select 中的元素划分

在算法 Select 中,输入元素被划分为 5 个一组,如果将它们划分为 7 个一组,该算法仍然是线性时间算法吗?划分成 3 个一组又怎样?

分析与解答:

见参考文献[1], 99。

习题 2-26 $O(n \log n)$ 时间快速排序算法

试说明如何修改快速排序算法,使它在最坏情况下的计算时间为 $O(n \log n)$ 。

分析与解答:

见参考文献[1], 99~100。

习题 2-27 最接近中位数的 k 个数

给定由 n 个互不相同数组成的集合 S , 以及正整数 $k \leq n$, 试设计一个 $O(n)$ 时间算法找

出 S 中最接近 S 的中位数的 k 个数。

分析与解答:

见参考文献[1], 100。

习题 2-28 X 和 Y 的中位数

设 $X[0:n-1]$ 和 $Y[0:n-1]$ 为 2 个数组, 每个数组中含有 n 个已排好序的数。试设计一个 $O(\log n)$ 时间的算法, 找出 X 和 Y 的 $2n$ 个数的中位数。

分析与解答:

见参考文献[1], 100~102。

习题 2-29 网络开关设计

考察如图 2-7 所示的有两个输入端和两个输出端的二位置开关。当开关处于位置 1 时, 输入 1 和 2 分别产生输出 1 和 2; 当开关处于位置 2 时, 输入 1 和 2 分别产生输出 2 和 1。使用这种开关设计一个有 n 个输入端和 n 个输出端的开关网络, 实现将输入的 n 个数值以它们的 $n!$ 种不同排列中的任意一种排列输出 (通过开关位置的适当选择)。要求网络中使用的开关个数为 $O(n \log n)$ 。

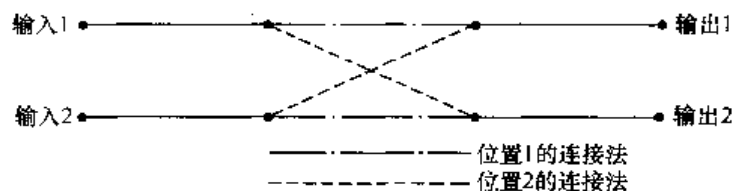


图 2-7 二位置开关

分析与解答:

见参考文献[1], 45~51。

习题 2-32 带权中位数问题

对于 n 个带有正权 w_1, w_2, \dots, w_n , 且 $\sum_{i=1}^n w_i = 1$ 的互不相同的元素 x_1, x_2, \dots, x_n , 其带权中位数 x_k 满足:

$$\begin{cases} \sum_{x_i < x_k} w_i \leq \frac{1}{2} \\ \sum_{x_i > x_k} w_i \leq \frac{1}{2} \end{cases}$$

(1) 试证明 x_1, x_2, \dots, x_n 的不带权中位数是带权 $w_i = 1/n, i=1, 2, \dots, n$ 的带权中位数。

(2) 说明如何通过排序, 在最坏情况下用 $O(n \log n)$ 时间求出 n 个元素的带权中位数。

(3) 说明如何利用一个线性时间选择算法 (如 Select), 在最坏情况下用 $O(n)$ 时间求出 n 个元素的带权中位数。

邮局位置问题定义为: 已知 n 个点 p_1, p_2, \dots, p_n 及与它们相联系的权 w_1, w_2, \dots, w_n , 要求确定一点 p (p 不一定是 n 个输入点之一), 使和式 $\sum_{i=1}^n w_i d(p, p_i)$ 达到最小, 其中 $d(a, b)$ 表示 a 与 b 之间的距离。

(4) 试论证带权中位数是一维邮局问题的最优解。此时 $d(a, b) = |a - b|$ 。

(5) 在二维的情形下如何找最优解?

分析与解答:

见参考文献[1], 104~108。

习题 2-34 构造 Gray 码的分治算法

Gray 码是一个长度为 2^n 的序列。序列中无相同元素, 每个元素都是长度为 n 位的 $(0,1)$ 串, 相邻元素恰好只有一位不同。用分治策略设计一个算法对任意的 n 构造相应的 Gray 码。

分析与解答:

考察 $n=1, 2, 3$ 的简单情形, 如表 2-2 所示。

表 2-2

设 n 位 Gray 码序列为 $G(n)$, $G(n)$ 以相反顺序排列的序列为 $G^{-1}(n)$ 。从上面的简单情形可以看出 $G(n)$ 的构造规律: $G(n+1)=0G(n)1G^{-1}(n)$ 。

长度(位)	(0,1)串			
$n=1$	0	1		
$n=2$	00	01		
	11	10		
$n=3$	000	001	011	010
	110	111	101	100

注意到 $G(n)$ 的最后一个 n 位串与 $G^{-1}(n)$ 的第一个 n 位串相同, 可用数学归纳法证明 $G(n)$ 的上述构造规律。由此规律容易设计构造 $G(n)$ 的分治法如下。

```
void gray(int n)
{
    if (n==1) {a[1]=0;a[2]=1;return;}
    gray(n-1);
    for(int k=1<<(n-1), i=k; i>0; i--) a[2*k-i+1]=a[i]+k;
}
```

上述算法中将 n 位 $(0,1)$ 串看作是二进制整数, 第 i 个串存储在 $a[i]$ 中。完成计算后, 由 out 输出 Gray 码序列。

```
void out(int n)
{
    char str[32];
    int m=1<<n;
    for (int i=1; i<=m; i++){
        _itoa(a[i], str, 2);
        int s=strlen(str);
        for(int j=0; j<n-s; j++) cout<<"0";
        cout<<str<<" ";
    }
    cout<<endl;
}
```

习题 2-35 网球循环赛日程表

设有 n 个运动员要进行网球循环赛。设计一个满足以下要求的比赛日程表:

- (1) 每个选手必须与其他 $n-1$ 个选手各赛一次;
- (2) 每个选手一天只能赛一次;
- (3) 当 n 是偶数时, 循环赛进行 $n-1$ 天; 当 n 是奇数时, 循环赛进行 n 天。

分析与解答:

(1) 分治法

主教材中的分治法应描述如下:

```
void tourna(int n)
{
    if (n==1) {a[1][1]=1;return;}
    tourna(n/2);
    copy(n);
}
```

其中, 算法 copy 将左上角递归计算出的小块中的所有数字按其相对位置抄到右下角, 将右上角小块中的所有数字加 $n/2$ 后按其相对位置抄到左下角, 将左下角小块中的所有数字按其相对位置抄到左上角, 这样就完成了比赛日程表。

```
void copy(int n)
{
    int m=n/2;
    for(int i=1;i<=m;i++)
        for(int j=1;j<=m;j++) {
            a[i][j+m]=a[i][j]+m;
            a[i+m][j]=a[i][j+m];
            a[i+m][j+m]=a[i][j];
        }
}
```

对于一般的正整数 n , 当 n 是奇数时, 增设一个虚拟选手 $n+1$, 将问题转换为 n 是偶数的情形。当选手与虚拟选手比赛时, 表示轮空。因此只要关注 n 为偶数的情形即可。

当 $n/2$ 为偶数时, 与 $n=2^k$ 的情形类似, 可用分治法求解。

当 $n/2$ 为奇数时, 递归返回的轮空的比赛要做进一步处理。其中一种处理方法是在前 $n/2$ 轮比赛中让轮空选手与下一个未参赛选手进行比赛。

一般情况下的分治法 tournament 可描述如下。

```
void tournament(int n)
{
    if (n==1) {a[1][1]=1;return;}
    if (odd(n)) {tournament(n+1);return;}
    tournament(n/2);
    makecopy(n);
}
```

```

bool odd(int n)
{
    return n & 1;
}

```

其中,算法 makecopy 与算法 copy 类似,但要区分 $n/2$ 为奇数或偶数的情形。

```

void makecopy(int n)
{
    if (n/2>1 && odd(n/2)) copyodd(n);
    else copy(n);
}

```

算法 copyodd 实现 $n/2$ 为奇数时的复制。

```

void copyodd(int n)
{
    int m=n/2;
    for(int i=1;i<=m;i++){
        b[i]=m+i;b[m+i]=b[i];
    }
    for(i=1;i<=m;i++){
        for(int j=1;j<=m+1;j++){
            if (a[i][j]>m) {a[i][j]=b[i];a[m+i][j]=(b[i]+m)%n;}
            else a[m+i][j]=a[i][j]+m;
        }
        for(j=2;j<=m;j++){
            a[i][m+j]=b[i+j-1];
            a[b[i+j-1]][m+j]=i;
        }
    }
}

```

用上述算法计算出的 $n=10$ 的比赛日程表如表 2-3 所示。

表 2-3 分治法 $n=10$ 的比赛日程表

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

(2) 多边形方法

n 是偶数的情形: 循环赛进行 $n-1$ 天, 每个选手与其他 $n-1$ 个选手各赛一次。

用一个 $n-1$ 边的正多边形表示一轮比赛。多边形的顶点和中心点表示参赛选手。 $n=8$ 时的循环赛多边形如图 2-8 所示。

用水平线连接循环赛多边形的 $n-2$ 个顶点, 并将剩下的那个顶点与中心点连接起来, 如图 2-9 所示。每一条连线表示一场比赛。

图 2-9 所表示的第 1 轮比赛的场次是 (7, 6), (1, 5), (2, 4) 和 (3, 8)。

将多边形绕中心顺时针旋转 $2\pi/(n-1)$ 弧度得到新的循环赛多边形, 如图 2-10 所示。

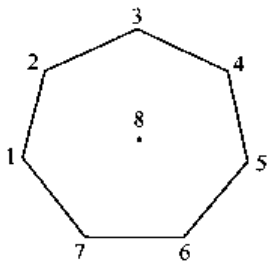


图 2-8 循环赛多边形

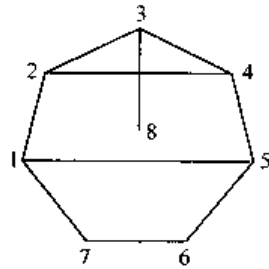


图 2-9 顶点间的连线表示比赛场次

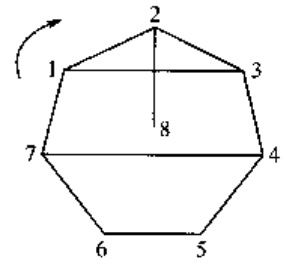


图 2-10 顺时针旋转 $2\pi/(n-1)$ 弧度

由此得到新一轮比赛的场次是 (6, 5), (7, 4), (1, 3) 和 (2, 8)。

按此方式可旋转多边形 $n-2$ 次。后继的旋转如图 2-11 所示。

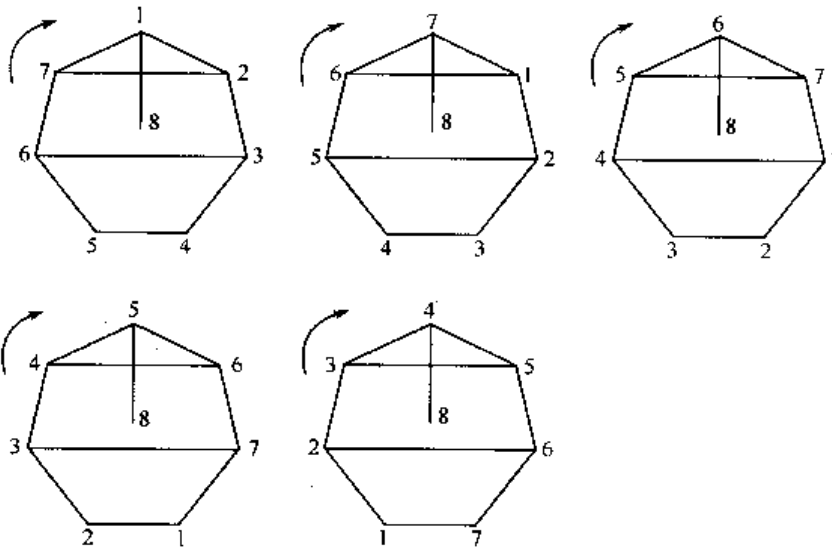


图 2-11 顺时针旋转多边形 $n-2$ 次

n 是奇数的情形同样可转换为 n 是偶数的情形。

按照上述思想实现的构造法如下。

```
void construct(int n)
{
    if (n==1) return;
    int m=odd(n)? n:n-1;
    a[n][1]=n;
    for(int i=1;i<=m;i++){
```

```

        a[i][1]=i;b[i]=i+1;b[m+i]=i+1;
    }
    for(i=1;i<=m;i++){
        a[1][i+1]=b[i];a[b[i]][i+1]=1;
        for(int j=1;j<=m/2;j++){
            int k=b[i+j], r=b[i+m-j];
            a[k][i+1]=r;a[r][i+1]=k;
        }
    }
}
}

```

用上述算法计算出的 $n=10$ 的比赛日程表如表 2-4 所示。

表 2-4 多边形方法 $n=10$ 的比赛日程表

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

用上述算法计算出的 $n=8$ 的比赛日程表完全图如图 2-12 所示。

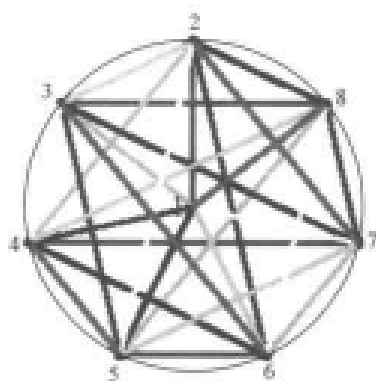


图 2-12 循环赛日程表 $n=8$ 的解

习题 2-36 二叉树 T 的前序、中序和后序序列

设二叉树 T 的前序、中序和后序序列分别为 pre , $inor$ 和 $post$ 。

(1) 给定 pre 和 $inor$, 能惟一确定 $post$ 吗? 如果能, 请写出由 pre 和 $inor$ 确定 $post$ 的算法。如果不能, 请给出一个反例。

(2) 给定 $post$ 和 $inor$, 能惟一确定 pre 吗? 如果能, 请写出由 $post$ 和 $inor$ 确定 pre 的算法。如果不能, 请给出一个反例。

(3) 给定 pre 和 $post$, 能惟一确定 $inor$ 吗? 如果能, 请写出由 pre 和 $post$ 确定 $inor$ 的算法。如果不能, 请给出一个反例。

分析与解答:

(1) 能; (2) 能; (3) 不能。

string $pre, post, inor$;

根据后序和中序序列确定前序序列。

```

void preorder(string a, string b)
{
    if(b.length()==1) {pre+=b;}
    else{
        int k=a.find(b.substr(b.length()-1,1));
        pre+=a[k];
        if(k>0) preorder(a.substr(0,k), b.substr(0,k));
        if(k<a.length()-1) preorder(a.substr(k+1, a.length()-k-1), b.substr(k,
        b.length()-k-1));
    }
}

```

根据前序和中序序列确定后序序列。

```

void postorder(string a, string b)
{
    if(b.length()==1) {post+=b;}
    else{
        int k=a.find(b.substr(0,1));
        if(k>0) postorder(a.substr(0,k), b.substr(1,k));
        if(k<a.length()-1) postorder(a.substr(k+1, a.length()-k-1), b.substr(k+1,
        b.length()-k-1));
        post+=a[k];
    }
}

```

算法实现题 2-1 输油管道问题(习题 2-30)

★问题描述:

某石油公司计划建造一条由东向西的主输油管道。该管道要穿过一个有 n 口油井的油田。从每口油井都要有一条输油管道沿最短路径（或南或北）与主管道相连。如果给定 n 口油井的位置，即它们的 x 坐标（东西向）和 y 坐标（南北向），应如何确定主管道的最优位置，即使各油井到主管道之间的输油管道长度总和最小的位置？证明可在线性时间内确定主管道的最优位置。

★编程任务:

给定 n 口油井的位置，编程计算各油井到主管道之间的输油管道最小长度总和。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行是油井数 n ， $1 \leq n \leq 10000$ 。接下来 n 行是油井的位置，每行 2 个整数 x 和 y ， $-10000 \leq x, y \leq 10000$ 。

★结果输出:

程序运行结束时，将计算结果输出到文件 output.txt 中。文件的第 1 行中的数是各油井到主管道之间的输油管道最小长度总和。

输入文件示例

输出文件示例

input.txt

output.txt

5

6

1 2

2 2

1 3

3 -2

3 3

分析与解答:

见参考文献[1], 108。

算法实现题 2-2 众数问题(习题 2-31)

★问题描述:

给定含有 n 个元素的多重集合 S , 每个元素在 S 中出现的次数称为该元素的重数。多重集 S 中重数最大的元素称为众数。

例如, $S = \{1, 2, 2, 2, 3, 5\}$ 。

多重集 S 的众数是 2, 其重数为 3。

★编程任务:

对于给定的由 n 个自然数组成的多重集 S , 编程计算 S 的众数及其重数。

★数据输入:

输入数据由文件名为 input.txt 的文本文件提供。

文件的第 1 行为多重集 S 中元素个数 n ; 在接下来的 n 行中, 每行有一个自然数。

★结果输出:

程序运行结束时, 将计算结果输出到文件 output.txt 中。输出文件有 2 行, 第 1 行是众数, 第 2 行是重数。

输入文件示例

输出文件示例

input.txt

output.txt

6

2

1

3

2

2

2

3

5

分析与解答:

见参考文献[1], 102~104。

算法具体实现如下。

```
void mode(int l, int r)
```

```
{
```



```

int ll,rl;
int med=median(a,l,r);
split(a,med,l,r,ll,rl);
if(largest<rl-ll+1) largest=rl-ll+1, element=med;
if (ll-1>largest) mode(l,ll-1);
if (r-rl>largest) mode(rl+1,r);
}

```

其中，median 用于找中位数。split 用中位数将数组分割为 2 段。

算法实现题 2-3 邮局选址问题(习题 2-32)

★问题描述:

在一个按照东西和南北方向划分成规整街区的城市里， n 个居民点散乱地分布在不同的街区中。用 x 坐标表示东西向，用 y 坐标表示南北向。各居民点的位置可以由坐标 (x,y) 表示。街区中任意两点 (x_1,y_1) 和 (x_2,y_2) 之间的距离可以用数值 $|x_1-x_2|+|y_1-y_2|$ 度量。

居民们希望在城市中选择建立邮局的最佳位置，使 n 个居民点到邮局的距离总和最小。

★编程任务:

给定 n 个居民点的位置，编程计算 n 个居民点到邮局的距离总和的最小值。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行是居民点数 n , $1 \leq n \leq 10000$ 。接下来 n 行是居民点的位置，每行 2 个整数 x 和 y , $-10000 \leq x,y \leq 10000$ 。

★结果输出:

程序运行结束时，将计算结果输出到文件 output.txt 中。文件的第 1 行中的数是 n 个居民点到邮局的距离总和的最小值。

输入文件示例

输出文件示例

input.txt

output.txt

5

10

1 2

2 2

1 3

3 -2

3 3

分析与解答:

见参考文献[1], 104~108。

算法实现题 2-4 马的 Hamilton 周游路线问题(习题 2-33)

★问题描述:

8×8 的国际象棋棋盘上的一只马，恰好走过除起点外的其他 63 个位置各一次，最后回到起点。这条路线称为马的一条 Hamilton 周游路线。对于给定的 $m \times n$ 的国际象棋棋盘， m 和 n 均为大于 5 的偶数，且 $|m-n| \leq 2$ ，试设计一个分治算法找出马的一条 Hamilton 周游路线。

★编程任务:

对于给定的偶数 $m, n \geq 6$, 且 $|m-n| \leq 2$, 编程计算 $m \times n$ 的国际象棋棋盘上马的一条 Hamilton 周游路线。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 m 和 n , 表示给定的国际象棋棋盘由 m 行, 每行 n 个格子组成。

★结果输出:

程序运行结束时, 将计算出的马的 Hamilton 周游路线用下面的 2 种表达方式输出到文件 output.txt 中。

第 1 种表达方式按照马步的次序给出马的 Hamilton 周游路线。马的每一步用所在的方格坐标 (x, y) 来表示。 x 表示行坐标, 编号为 $0, 1, \dots, m-1$; y 表示列坐标, 编号为 $0, 1, \dots, n-1$ 。起始方格为 $(0, 0)$ 。

第 2 种表达方式在棋盘的方格中标明马到达该方格的步数。 $(0, 0)$ 方格为起跳步, 并标明为第 1 步。

输入文件示例

input.txt

6 6

输出文件示例

output.txt

(0,0) (2,1) (4,0) (5,2) (4,4) (2,3)
(0,4) (2,5) (1,3) (0,5) (2,4) (4,5)
(5,3) (3,2) (5,1) (3,0) (1,1) (0,3)
(1,5) (3,4) (5,5) (4,3) (3,1) (5,0)
(4,2) (5,4) (3,5) (1,4) (0,2) (1,0)
(2,2) (0,1) (2,0) (4,1) (3,3) (1,2)

1 32 29 18 7 10
30 17 36 9 28 19
33 2 31 6 11 8
16 23 14 35 20 27
3 34 25 22 5 12
24 15 4 13 26 21

分析与解答:

(1) 算法思想

在 $n \times n$ 的国际象棋棋盘上的一只马, 可按 8 个不同方向移动。定义 $n \times n$ 的国际象棋棋盘上的马步图为 $G=(V, E)$ 。棋盘上的每个方格对应于图 G 中的一个顶点, $V=\{(i, j) | 0 \leq i, j < n\}$ 。从一个顶点到另一个马步可跳达的顶点之间有一条边 $E=\{(u, v), (s, t) | \{|u-s|, |v-t|\}=\{1, 2\}\}$ 。

图 G 有 n^2 个顶点和 $4n^2-12n+8$ 条边。马的 Hamilton 周游路线问题即是图 G 的 Hamilton 回路问题。容易看出, 当 n 为奇数时该问题无解。事实上, 由于马在棋盘上移动的方格是黑白相间的, 如果有解, 则走到的黑白格子数相同, 因此棋盘格子总数应为偶数, 然而 n^2 为奇数, 此为矛盾。下面给出的算法可以证明, 当 $n \geq 6$ 是偶数时, 问题有解, 而且可以用分治法在线性时间内构造出一个解。

考察稍一般的情况, 即给定的国际象棋棋盘有 m 行和 n 列, 且 $|m-n| \leq 2$ 的情况。因此可能有 $m \times m$, $m \times (m-2)$ 和 $m \times (m+2)$ 共 3 种不同规格的棋盘。为了采用分治策略, 考察一类具有特殊结构的解, 这类解在棋盘的 4 个角都包含 2 条特殊的边, 如图 2-13 所示。我们称具有这类特殊结构的 Hamilton 回路为结构化的 Hamilton 回路。

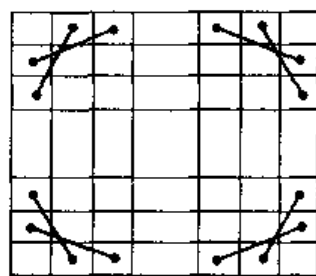


图 2-13 结构化的 Hamilton 回路

用回溯法可在 $O(1)$ 时间内找出 6×6 , 6×8 , 8×8 , 8×10 , 10×10 , 10×12 棋盘上的结构化的 Hamilton 回路, 如图 2-14 所示。

1	30	33	16	3	24
32	17	2	23	34	15
29	36	31	14	25	4
18	9	6	35	22	13
7	28	11	20	5	26
10	19	8	27	12	21

(a) 6×6 棋盘上的结构化 Hamilton 回路

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

(b) 6×8 棋盘上的结构化 Hamilton 回路

1	46	17	50	3	6	31	52
18	49	2	7	30	51	56	5
45	64	47	16	27	4	53	32
48	19	8	29	10	55	26	57
63	44	11	22	15	28	33	54
12	41	20	9	36	23	58	25
43	62	39	14	21	60	37	34
40	13	42	61	38	35	24	59

(c) 8×8 棋盘上的结构化 Hamilton 回路

1	46	37	66	3	48	35	68	5	8
38	65	2	47	36	67	4	7	34	69
45	80	39	24	49	18	31	52	9	6
64	23	44	21	30	15	50	19	70	33
79	40	25	14	17	20	53	32	51	10
26	63	22	43	54	29	16	73	58	71
41	78	61	28	13	76	59	56	11	74
62	27	42	77	60	55	12	75	72	57

(d) 8×10 棋盘上的结构化 Hamilton 回路

1	54	69	66	3	56	39	64	5	8
68	71	2	55	38	65	4	7	88	63
53	100	67	70	57	26	35	40	9	6
72	75	52	27	42	37	58	87	62	89
99	30	73	44	25	34	41	36	59	10
74	51	76	31	28	43	86	81	90	61
77	98	29	24	45	80	33	60	11	92
50	23	48	79	32	85	82	91	14	17
97	78	21	84	95	46	19	16	93	12
22	49	96	47	20	83	94	13	18	15

(e) 10×10 棋盘上的结构化 Hamilton 回路

1	4	119	100	65	6	69	102	71	8	75	104
118	99	2	5	68	101	42	7	28	103	72	9
3	120	97	64	41	66	25	70	39	74	105	76
98	117	48	67	62	43	40	27	60	29	10	73
93	96	63	44	47	26	61	24	33	38	77	106
116	51	94	49	20	23	46	37	30	59	34	11
95	92	115	52	45	54	21	32	35	80	107	78
114	89	50	19	22	85	36	55	58	31	12	81
91	18	87	112	53	16	57	110	83	14	79	108
88	113	90	17	86	111	84	15	56	109	82	13

(f) 10×12 棋盘上的结构化 Hamilton 回路

图 2-14 几种棋盘上的结构化 Hamilton 回路

其中, $6 \times 8, 8 \times 10$ 和 10×12 棋盘上的结构化 Hamilton 回路, 旋转 90° 可以得到 $8 \times 6, 10 \times 8$ 和 12×10 棋盘上的结构化 Hamilton 回路。

在棋盘上画出的结构化 Hamilton 回路, 如图 2-15 所示。

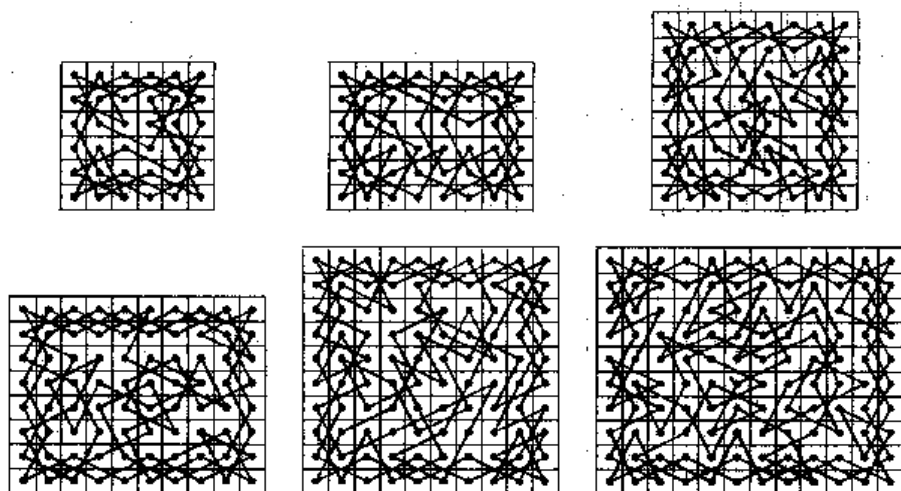


图 2-15 在棋盘上画出的结构化 Hamilton 回路

对于 $m, n \geq 12$ 的情形, 采用分治策略。

分割步:

将棋盘尽可能平均地分割成 4 块。当 $m, n = 4k$ 时, 分割为 2 个 $2k$; 当 $m, n = 4k + 2$ 时, 分割为 1 个 $2k$ 和 1 个 $2k + 2$ 。

合并步:

4 个子棋盘拼接后的结构如图 2-16 所示。

分别删除 4 个子棋盘中的结构化边 A, B, C, D, 添入新边 E, F, G, H 构成整个棋盘的结构化 Hamilton 回路, 如图 2-17 所示。

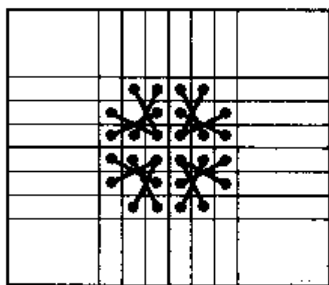


图 2-16 子棋盘的拼接

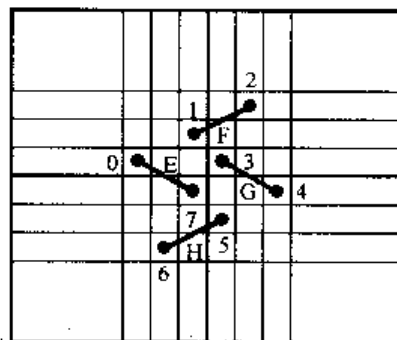
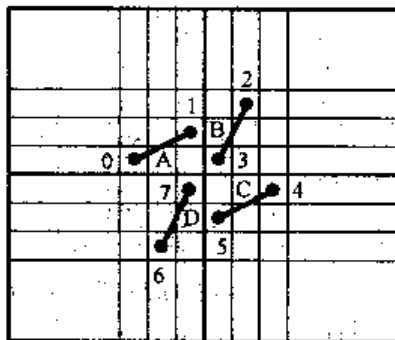


图 2-17 子棋盘中 Hamilton 回路的合并

上述分治法得到的 Hamilton 回路显然仍是结构化 Hamilton 回路。

图 2-18 所示是用上述分治法计算出的 16×16 棋盘上的结构化 Hamilton 回路。

(2) 算法复杂性

设用上述分治法计算 $n \times n$ 棋盘上的 Hamilton 回路所需计算时间为 $T(n)$, 则 $T(n)$ 满足如下递归式:

$$T(n) = \begin{cases} O(1) & n < 12 \\ 4T(n/2) + O(1) & n \geq 12 \end{cases}$$

解此递归式可得 $T(n) = O(n^2)$ 。

1	238	17	242	3	6	31	244	59	40	76	44	61	64	89	46
18	241	2	7	30	243	248	5	76	43	80	65	88	45	50	63
237	256	239	16	27	4	245	32	39	58	41	74	85	62	47	90
240	19	8	29	10	247	26	249	42	77	66	87	68	49	84	51
255	236	11	22	15	28	33	246	57	38	69	80	73	86	91	48
12	233	20	9	228	23	250	25	70	35	78	67	94	81	52	83
235	254	231	14	21	252	229	34	37	56	97	72	79	54	95	92
232	13	234	253	230	227	24	251	98	71	36	55	96	93	82	53
175	220	191	224	177	180	205	226	115	134	99	130	113	110	149	128
192	223	176	181	204	225	166	179	162	131	114	109	150	129	124	111
219	174	221	190	201	178	163	206	135	116	133	100	153	112	127	148
222	193	182	203	184	165	200	167	132	161	108	151	106	125	154	123
173	218	185	196	189	202	207	164	117	136	105	158	101	152	147	126
186	215	194	183	210	197	168	199	104	139	160	107	144	157	122	155
217	172	213	188	195	170	211	208	137	118	141	102	159	120	143	146
214	187	216	171	212	209	198	169	140	103	138	119	142	145	156	121

图 2-18 用分治法计算出的 16×16 棋盘上的结构化 Hamilton 回路

由此可见，上述算法是一个最优算法。

(3) 算法实现

用一个类 Knight 实现算法。

```
class Knight{
public:
    Knight(int m, int n);
    ~Knight(){};
    void out();
private:
    int m, n;
    grid *b66, *b68, *b86, *b88, *b810, *b108, *b1010, *b1012, *b1210, **link;
    int pos(int x, int y, int col);
    void step(int m, int n, int **a, grid *b);
    void build(int m, int n, int offx, int offy, int col, grid *b);
    void base(int mm, int nn, int offx, int offy);
    bool comp(int mm, int nn, int offx, int offy);
};
```

其中，grid 是表示整数对的结构。

```
typedef struct {
    int x;
    int y;
} grid;
```

m 和 n 分别表示棋盘的行数和列数。二维数组 link 用来表示 Hamilton 回路。

b66, b68, b86, b88, b810, b108, b1010, b1012, b1210 分别表示 $6 \times 6, 6 \times 8, 8 \times 6, 8 \times 8, 8 \times 10, 10 \times 8, 10 \times 10, 10 \times 12, 12 \times 10$ 棋盘上的结构化 Hamilton 回路。

构造函数读入基础数据, 初始化各数组。

```
Knight::Knight(int mm, int nn)
{
    int i, j, **a;
    m=mm; n=nn;
    b66=new grid[36];
    b68=new grid[48];
    b86=new grid[48];
    b88=new grid[64];
    b810=new grid[80];
    b108=new grid[80];
    b1010=new grid[100];
    b1012=new grid[120];
    b1210=new grid[120];
    Make2DArray(link, m, n);
    Make2DArray(a, 10, 12);
    for(i=0; i<6; i++)
        for(j=0; j<6; j++) fin0>>a[i][j];
    step(6, 6, a, b66);
    for(i=0; i<6; i++)
        for(j=0; j<8; j++) fin0>>a[i][j];
    step(6, 8, a, b68); step(8, 6, a, b86);
    for(i=0; i<8; i++)
        for(j=0; j<8; j++) fin0>>a[i][j];
    step(8, 8, a, b88);
    for(i=0; i<8; i++)
        for(j=0; j<10; j++) fin0>>a[i][j];
    step(8, 10, a, b810); step(10, 8, a, b108);
    for(i=0; i<10; i++)
        for(j=0; j<10; j++) fin0>>a[i][j];
    step(10, 10, a, b1010);
    for(i=0; i<10; i++)
        for(j=0; j<12; j++) fin0>>a[i][j];
    step(10, 12, a, b1012); step(12, 10, a, b1210);
}
```

其中, step 用于将读入的基础棋盘的 Hamilton 回路转化为网格数据。

```
void Knight::step(int m, int n, int **a, grid *b)
{
    int i, j, k=m*n;
```

```

    if(m<n){
        for (i=0; i<m; i++)
            for (j=0; j<n; j++){
                int p=a[i][j]-1;
                b[p].x=i;b[p].y=j;
            }
    }
    else{
        for (i=0; i<m; i++)
            for (j=0; j<n; j++){
                int p=a[j][i]-1;
                b[p].x=i;b[p].y=j;
            }
    }
}

```

分治法的主体由如下算法 comp 给出。

```

bool Knight::comp(int mm,int nn,int offx,int offy)
{
    int mm1,mm2,nn1,nn2;
    int x[8],y[8],p[8];
    if(odd(mm) || odd(nn) || mm-nn>2 || nn-mm>2 || mm<6 || nn<6)return 1;
    if(mm<12 || nn<12){base(mm,nn,offx,offy);return 0;} // 基础解
    mm1=mm/2;
    if(mm%4>0)mm1--;
    mm2=mm-mm1;
    nn1=nn/2;
    if(nn%4>0)nn1--;
    nn2=nn-nn1;
    // 分割步
    comp(mm1,nn1,offx,offy);
    comp(mm1,nn2,offx,offy+nn1);
    comp(mm2,nn1,offx+mm1,offy);
    comp(mm2,nn2,offx+mm1,offy+nn1);
    // 合并步
    x[0]=offx+mm1-1;y[0]=offy+nn1-3;
    x[1]=x[0]-1;y[1]=y[0]+2;
    x[2]=x[1]-1;y[2]=y[1]+2;
    x[3]=x[2]+2;y[3]=y[2]-1;
    x[4]=x[3]+1;y[4]=y[3]+2;
    x[5]=x[4]+1;y[5]=y[4]-2;
    x[6]=x[5]+1;y[6]=y[5]-2;
    x[7]=x[6]-2;y[7]=y[6]+1;
}

```

```

for(int i=0;i<8;i++)p[i]=pos(x[i],y[i],n);
for(i=1;i<8;i+=2){
    int j1=(i+1)%8,j2=(i+2)%8;
    if(link[x[i]][y[i]].x==p[i-1])link[x[i]][y[i]].x=p[j1];
    else link[x[i]][y[i]].y=p[j1];
    if(link[x[j1]][y[j1]].x==p[j2])link[x[j1]][y[j1]].x=p[i];
    else link[x[j1]][y[j1]].y=p[i];
}
return 0;
}

```

其中,base 根据基础解构造子棋盘的结构化 Hamilton 回路。

```

void Knight::base(int mm,int nn,int offx,int offy)
{
    if(mm==6 && nn==6)build(mm,nn,offx,offy,n,b66);
    if(mm==6 && nn==8)build(mm,nn,offx,offy,n,b68);
    if(mm==8 && nn==6)build(mm,nn,offx,offy,n,b86);
    if(mm==8 && nn==8)build(mm,nn,offx,offy,n,b88);
    if(mm==8 && nn==10)build(mm,nn,offx,offy,n,b810);
    if(mm==10 && nn==8)build(mm,nn,offx,offy,n,b108);
    if(mm==10 && nn==10)build(mm,nn,offx,offy,n,b1010);
    if(mm==10 && nn==12)build(mm,nn,offx,offy,n,b1012);
    if(mm==12 && nn==10)build(mm,nn,offx,offy,n,b1210);
}

```

其实质性的构造由算法 build 完成。

```

void Knight::build(int m,int n,int offx,int offy,int col,grid *b)
{
    int i,p,q,k=m*n;
    for(i=0;i<k;i++){
        int x1=offx+b[i].x,y1=offy+b[i].y,
            x2=offx+b[(i+1)%k].x,y2=offy+b[(i+1)%k].y;
        p=pos(x1,y1,col);q=pos(x2,y2,col);
        link[x1][y1].x=q;link[x2][y2].y=p;
    }
}

```

其中, pos 用于计算棋盘方格的编号。棋盘方格各行从上到下,各列从左到右依次编号为 $0,1,\dots,mn-1$ 。

```

int Knight::pos(int x,int y,int col)
{

```



```

        return col*x + y;
    }

```

最后，由 out 按照要求输出计算出的结构化 Hamilton 回路。

```

void Knight::out()
{
    int i, j, k, x, y, p, **a;
    Make2DArray(a, m, n);
    if (comp(m, n, 0, 0)) return;
    for (i = 0; i < m; i++)
        for (j = 0; j < n; j++) a[i][j] = 0;
    i = 0; j = 0; k = 2; a[0][0] = 1;
    cout << "(0, 0)" << " ";
    for (p = 1; p < m*n; p++) {
        x = link[i][j].x; y = link[i][j].y;
        i = x/n; j = x%n;
        if (a[i][j] > 0) { i = y/n; j = y%n; }
        a[i][j] = k++;
        cout << "(" << i << ", " << j << ") ";
        if ((k - 1) % n == 0) cout << endl;
    }
    cout << endl;
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) cout << a[i][j] << " ";
        cout << endl;
    }
}

```

算法实现题 2-5 半数集问题

★问题描述：

给定一个自然数 n ，由 n 开始可以依次产生半数集 $\text{set}(n)$ 中的数如下：

- (1) $n \in \text{set}(n)$;
- (2) 在 n 的左边加上一个自然数，但该自然数不能超过最近添加的数的一半；
- (3) 按此规则进行处理，直到不能再添加自然数为止。

例如， $\text{set}(6) = \{6, 16, 26, 126, 36, 136\}$ 。半数集 $\text{set}(6)$ 中有 6 个元素。

注意，该半数集是多重集。

★编程任务：

对于给定的自然数 n ，编程计算半数集 $\text{set}(n)$ 中的元素个数。

★数据输入：

输入数据由文件名为 input.txt 的文本文件提供。每个文件只有一行，给出整数 n ($0 < n < 1000$)。

★结果输出:

程序运行结束时, 将计算结果输出到文件 output.txt 中。输出文件只有一行, 给出半数集 set(n) 中的元素个数。

输入文件示例

input.txt

6

输出文件示例

output.txt

6

分析与解答:

设 set(n) 中的元素个数为 $f(n)$, 则显然有

$$f(n) = 1 + \sum_{i=1}^{n/2} f(i)$$

据此可设计求 $f(n)$ 的递归算法如下。

```
long comp(int n)
{
    long ans=1;
    if(n>1) for (int i=1;i<=n/2;i++) ans+=comp(i);
    return ans;
}
```

上述算法中显然有很多的重复子问题计算。用数组存储已计算过的结果, 避免重复计算, 可明显改进算法的效率。改进后的算法如下。

```
long comp(int n)
{
    long ans=1;
    if (a[n]>0) return a[n];
    for (int i=1;i<=n/2;i++) ans+=comp(i);
    a[n]=ans;
    return ans;
}
```

```
int main()
{
    while (cin>>n) {
        memset(a, sizeof(a), 0);
        a[1]=1;
        cout<<comp(n)<<endl;
    }
    return 0;
}
```

算法实现题 2-6 半数单集问题

★问题描述:

给定一个自然数 n , 由 n 开始可以依次产生半数集 set(n) 中的数如下:

- (1) $n \in \text{set}(n)$;
- (2) 在 n 的左边加上一个自然数, 但该自然数不能超过最近添加的数的一半;
- (3) 按此规则进行处理, 直到不能再添加自然数为止。

例如, $\text{set}(6) = \{6, 16, 26, 126, 36, 136\}$ 。半数集 $\text{set}(6)$ 中有 6 个元素。

注意该半数集不是多重集。集合中已经有的元素不再添加到集合中。

★编程任务:

对于给定的自然数 n , 编程计算半数集 $\text{set}(n)$ 中的元素个数。

★数据输入:

输入数据由文件名为 input.txt 的文本文件提供。每个文件只有一行, 给出整数 n ($0 < n < 201$)。

★结果输出:

程序运行结束时, 将计算结果输出到文件 output.txt 中。输出文件只有一行, 给出半数集 $\text{set}(n)$ 中的元素个数。

输入文件示例

input.txt

6

输出文件示例

output.txt

6

分析与解答:

此题与算法实现题 2-5 类似。主要区别在于此题的半数集为单集, 不允许重复元素。因此, 在计算时应剔除重复元素。注意题中条件 $0 < n < 201$, 蕴含 $0 < n/2 \leq 100$ 。因此, 在计算时, 可能产生重复的元素是 2 位数。一个 2 位数 x 重复产生的条件是, 在 1 位数 $y = x \% 10$ 的半数集中已产生 x , 因此 $x/10 \leq y/2$, 或等价地, $2(x/10) \leq x \% 10$ 。在前面的算法中, 加入剔除重复元素的语句即可。

```

long comp(int n)
{
    long ans=1;
    if (a[n]>0)return a[n];
    for (int i=1;i<=n/2;i++){
        ans+=comp(i);
        if ((i>10)&&(2*(i/10)<=i%10)) ans-=a[i/10];
    }
    a[n]=ans;
    return ans;
}

```

如果不利用题中对于 n 的范围限制, 则应考虑一般情况, 即有 $\log_{10} n$ 位数字的情况。此题还可用散列表或数字检索树等数据结构方法来实现。

算法实现题 2-7 士兵站队问题

★问题描述:

在一个划分成网格的操场上, n 个士兵散乱地站在网格点上。网格点由整数坐标 (x, y)

表示。士兵们可以沿网格边上、下、左、右移动一步，但在同一时刻任一网格点上只能有一名士兵。按照军官的命令，士兵们要整齐地列成一个水平队列，即排列成 $(x, y), (x+1, y), \dots, (x+n-1, y)$ 。如何选择 x 和 y 的值才能使士兵们以最少的总移动步数排成一行。

★编程任务：

计算使所有士兵排成一行需要的最少移动步数。

★数据输入：

由文件 input.txt 提供输入数据。文件的第 1 行是士兵数 $n, 1 \leq n \leq 10000$ 。接下来 n 行是士兵的初始位置，每行 2 个整数 x 和 $y, -10000 \leq x, y \leq 10000$ 。

★结果输出：

程序运行结束时，将计算结果输出到文件 output.txt 中。文件的第 1 行中的数是士兵排成一行需要的最少移动步数。

输入文件示例	输出文件示例
input.txt	output.txt
5	8
1 2	
2 2	
1 3	
3 -2	
3 3	

分析与解答：

与习题 2-30 解法相同。见参考文献[1], 108。

算法实现题 2-8 有重复元素的排列问题

★问题描述：

设 $R = \{r_1, r_2, \dots, r_n\}$ 是要进行排列的 n 个元素。其中元素 r_1, r_2, \dots, r_n 可能相同。试设计一个算法，列出 R 的所有不同排列。

★编程任务：

给定 n 以及待排列的 n 个元素。计算出这 n 个元素的所有不同排列。

★数据输入：

由文件 input.txt 提供输入数据。文件的第 1 行是元素个数 $n, 1 \leq n \leq 500$ 。接下来的 1 行是待排列的 n 个元素。

★结果输出：

程序运行结束时，将计算出的 n 个元素的所有不同排列输出到文件 output.txt 中。文件最后 1 行中的数是排列总数。

输入文件示例	输出文件示例
input.txt	output.txt
4	aacc
aacc	acac
	acca
	caac

caca

ccaa

6

分析与解答：

与主教材中的算法 Perm 类似。主要区别是对重复元素的处理。

```
template<class Type>
void Perm(Type list[], int k, int m)
{
    if(k==m){
        ans[1]++;
        for(int i=0;i<=m;i++) cout<<list[i];
        cout<<endl;
    }
    else for (int i=k;i<=m;i++)
        if(ok(list,k,i)){
            Swap(list[k],list[i]);
            Perm(list,k+1,m);
            Swap(list[k],list[i]);
        }
}
```

其中，函数 ok 用于判别重复元素。

```
template<class Type>
int ok(Type list[], int k, int i)
{
    if(i>k) for(int t=k;t<i;t++)if(list[t]==list[i])return 0;
    return 1;
}
```

算法实现题 2-9 排列的字典序问题

★问题描述：

n 个元素 $\{1, 2, \dots, n\}$ 有 $n!$ 个不同的排列。将这 $n!$ 个排列按字典序排列，并编号为 $0, 1, \dots, n!-1$ 。每个排列的编号为其字典序值。例如，当 $n=3$ 时，6 个不同排列的字典序值如下：

字典序值	0	1	2	3	4	5
排列	123	132	213	231	312	321

★编程任务：

给定 n 以及 n 个元素 $\{1, 2, \dots, n\}$ 的一个排列，计算出这个排列的字典序值，以及按字典序排列的下一个排列。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行是元素个数 n 。接下来的 1 行是 n 个元素 $\{1, 2, \dots, n\}$ 的一个排列。

★结果输出:

程序运行结束时, 将计算出的排列的字典序值和按字典序排列的下一个排列输出到文件 output.txt 中。文件的第 1 行是字典序值, 第 2 行是按字典序排列的下一个排列。

输入文件示例	输出文件示例
input.txt	output.txt
8	8227
2 6 4 5 8 1 7 3	2 6 4 5 8 3 1 7

分析与解答:

(1) 由排列计算字典序值

设给定的 $\{1, 2, \dots, n\}$ 的排列为 π , 其字典序值为 $\text{rank}(\pi, n)$ 。按字典序的定义显然有

$$(\pi[1]-1)(n-1)! \leq \text{rank}(\pi, n) \leq \pi[1](n-1)! - 1$$

设 r 是 π 在以 $\pi[1]$ 开头的排列中的序号, 则 r 也是 $[\pi[2], \pi[3], \dots, \pi[n]]$ 作为集合 $\{1, 2, \dots, n\} / \{\pi[1]\}$ 中排列的字典序值。如果将 $[\pi[2], \pi[3], \dots, \pi[n]]$ 中每个大于 $\pi[1]$ 的元素都减 1, 则得到集合 $\{1, 2, \dots, n-1\}$ 的一个排列 π' , 其字典序值也是 r 。由此得到计算 $\text{rank}(\pi, n)$ 的递归式如下:

$$\text{rank}(\pi, n) = (\pi[1]-1)(n-1)! + \text{rank}(\pi', n-1)$$

其中,

$$\pi'[i] = \begin{cases} \pi[i+1]-1 & \text{若 } \pi[i+1] > \pi[1] \\ \pi[i+1] & \text{若 } \pi[i+1] < \pi[1] \end{cases}$$

初始条件为

$$\text{rank}([1], 1) = 0$$

据此可设计计算 $\text{rank}(\pi, n)$ 的算法 permRank 如下。

```

int permRank(int n, int *pi)
{
    for(int j=1, r=0; j<=n; j++) rho[j]=pi[j];
    for(j=1; j<=n; j++){
        r+=(rho[j]-1)*f[n-j];
        for(int i=j+1; i<=n; i++){
            if(rho[i]>rho[j])rho[i]--;
        }
    }
    return r;
}

```

其中, $f[j]$ 存储预先计算出的 $j!$ 的值。

(2) 由字典序值计算排列

对于每个整数 r , $0 \leq r \leq n! - 1$, 都有惟一的阶层分解: $r = \sum_{i=1}^{n-1} d_i \cdot i!$, $0 \leq d_i \leq i$ 。

设 $r = \text{rank}(\pi, n)$, 则显然有 $\pi[1] = d_{n-1} + 1$ 。

进一步, 由 $r' = r - d_{n-1} \cdot (n-1)! = \text{rank}(\pi', n-1)$ 可递归地找到排列 π' 。最后令 $\pi[i] = \pi'[i+1]$ 可得到排列 π 。

据此可设计计算排列 π 使 $r = \text{rank}(\pi, n)$ 的算法 permUnrank 如下。

```
void permUnrank(int n, int r, int *pi)
{
    pi[n] = 1;
    for(int j=1; j<n; j++) {
        int d = (r % f[j-1]) / f[j];
        r -= d * f[j];
        pi[n-j] += d + 1;
        for(int i=n-j+1; i<=n; i++)
            if(pi[i] > d) pi[i]++;
    }
}
```

(3) 由排列计算下一个排列

按字典序的定义可设计从一个排列计算下一个排列的算法。对于给定的排列 π , 首先找到下标 i , 使得 $\pi[i] < \pi[i+1]$, 且 $\pi[i+1] > \pi[i+2] > \dots > \pi[n]$; 其次找到下标 j , 使得 $\pi[i] < \pi[j]$ 且对所有 $j < k \leq n$ 有 $\pi[k] < \pi[i]$; 然后交换 $\pi[i]$ 和 $\pi[j]$; 最后将子排列 $[\pi[i+1], \pi[i+2], \dots, \pi[n]]$ 反转。按此思想设计的算法 permSucc 如下。

```
void permSucc(int n, int *pi, int &flag)
{
    pi[0] = 0;
    int i = n - 1;
    while (pi[i+1] < pi[i]) i--;
    if(i == 0) flag = 0;
    else {
        flag = 1;
        int j = n;
        while(pi[j] < pi[i]) j--;
        Swap(pi[i], pi[j]);
        for(int h=i+1; h<=n; h++) rho[h] = pi[h];
        for(h=i+1; h<=n; h++) pi[h] = rho[n+i+1-h];
    }
}
```

算法实现题 2-10 集合划分问题

★问题描述:

n 个元素的集合 $\{1, 2, \dots, n\}$ 可以划分为若干非空子集。例如, 当 $n=4$ 时, 集合 $\{1, 2, 3,$

4)可以划分为 15 个不同的非空子集如下:

$\{\{1\},\{2\},\{3\},\{4\}\}$	$\{\{1,3\},\{2,4\}\}$
$\{\{1,2\},\{3\},\{4\}\}$	$\{\{1,4\},\{2,3\}\}$
$\{\{1,3\},\{2\},\{4\}\}$	$\{\{1,2,3\},\{4\}\}$
$\{\{1,4\},\{2\},\{3\}\}$	$\{\{1,2,4\},\{3\}\}$
$\{\{2,3\},\{1\},\{4\}\}$	$\{\{1,3,4\},\{2\}\}$
$\{\{2,4\},\{1\},\{3\}\}$	$\{\{2,3,4\},\{1\}\}$
$\{\{3,4\},\{1\},\{2\}\}$	$\{\{1,2,3,4\}\}$
$\{\{1,2\},\{3,4\}\}$	

★编程任务:

给定正整数 n , 计算出 n 个元素的集合 $\{1,2,\dots,n\}$ 可以划分为多少个不同的非空子集。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行是元素个数 n 。

★结果输出:

程序运行结束时, 将计算出的不同的非空子集数输出到文件 output.txt 中。

输入文件示例	输出文件示例
input.txt	output.txt
5	52

分析与解答:

所求的是 Bell 数, 满足如下递归式:

$$B(n) = \sum_{i=0}^{n-1} \binom{n-1}{i} B(i); B(0) = 1$$

算法实现题 2-11 集合划分问题

★问题描述:

n 个元素的集合 $\{1,2,\dots,n\}$ 可以划分为若干非空子集。例如, 当 $n=4$ 时, 集合 $\{1,2,3,4\}$ 可以划分为 15 个不同的非空子集如下:

$\{\{1\},\{2\},\{3\},\{4\}\}$	$\{\{1,3\},\{2,4\}\}$
$\{\{1,2\},\{3\},\{4\}\}$	$\{\{1,4\},\{2,3\}\}$
$\{\{1,3\},\{2\},\{4\}\}$	$\{\{1,2,3\},\{4\}\}$
$\{\{1,4\},\{2\},\{3\}\}$	$\{\{1,2,4\},\{3\}\}$
$\{\{2,3\},\{1\},\{4\}\}$	$\{\{1,3,4\},\{2\}\}$
$\{\{2,4\},\{1\},\{3\}\}$	$\{\{2,3,4\},\{1\}\}$
$\{\{3,4\},\{1\},\{2\}\}$	$\{\{1,2,3,4\}\}$
$\{\{1,2\},\{3,4\}\}$	

其中, 集合 $\{\{1,2,3,4\}\}$ 由 1 个子集组成; 集合 $\{\{1,2\},\{3,4\}\}, \{\{1,3\},\{2,4\}\}, \{\{1,4\},\{2,3\}\}, \{\{1,2,3\},\{4\}\}, \{\{1,2,4\},\{3\}\}, \{\{1,3,4\},\{2\}\}, \{\{2,3,4\},\{1\}\}$ 由 2 个子集组成; 集合 $\{\{1,2\},\{3\},\{4\}\}, \{\{1,3\},\{2\},\{4\}\}, \{\{1,4\},\{2\},\{3\}\}, \{\{2,3\},\{1\},\{4\}\}, \{\{2,4\},\{1\},\{3\}\}, \{\{3,4\},\{1\},\{2\}\}$ 由 3 个子集组成; 集合 $\{\{1\},\{2\},\{3\},\{4\}\}$ 由 4 个子集组成。

★编程任务:

给定正整数 n 和 m , 计算出 n 个元素的集合 $\{1, 2, \dots, n\}$ 可以划分为多少个不同的由 m 个非空子集组成的集合。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行是元素个数 n 和非空子集数 m 。

★结果输出:

程序运行结束时, 将计算出的不同的由 m 个非空子集组成的集合数输出到文件 output.txt 中。

输入文件示例

输出文件示例

input.txt

output.txt

4 3

6

分析与解答:

所求的是第 2 类 Stirling 数 $S(n, m)$, 满足如下递归式:

$$S(n, n+1) = 0$$

$$S(n, 0) = 0$$

$$S(0, 0) = 1$$

$$S(n, m) = mS(n-1, m) + S(n-1, m-1)$$

关于 Bell 数, 显然有

$$B(n) = \sum_{m=1}^n S(n, m)$$

```
void StirlingNumbers2(int m, int n)
{
    int min;
    S[0][0]=1;
    for(int i=1;i<=m;i++) S[i][0]=0;
    for(i=0;i<m;i++) S[i][i-1]=0;
    for(i=1;i<=m;i++){
        if(i<n) min=i;else min=n;
        for(int j=1;j<=min;j++) S[i][j]=j*S[i-1][j]+S[i-1][j-1];
    }
}

int computeB(int m)
{
    StirlingNumbers2(m, m);
    for(int i=0;i<m;i++) B[i]=0;
    for(i=1;i<=m;i++){
        for(int j=0;j<=i;j++)
            B[i-1] += S[i][j];
    }
    return B[m-1];
}
```

算法实现题 2-12 双色 Hanoi 塔问题

★问题描述:

设 A, B, C 是 3 个塔座。开始时, 在塔座 A 上有一叠共 n 个圆盘, 这些圆盘自下而上, 由大到小地叠在一起。各圆盘从小到大编号为 $1, 2, \dots, n$, 奇数号圆盘着红色, 偶数号圆盘着蓝色, 如图 2-19 所示。现要求将塔座 A 上的这一叠圆盘移到塔座 B 上, 并仍按同样顺序叠置。在移动圆盘时应遵守以下移动规则:

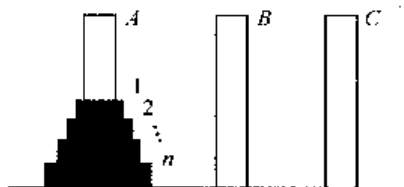


图 2-19 双色 Hanoi 塔

规则 (1): 每次只能移动 1 个圆盘;

规则 (2): 任何时刻都不允许将较大的圆盘压在较小的圆盘之上;

规则 (3): 任何时刻都不允许将同色圆盘叠在一起;

规则 (4): 在满足移动规则 (1) ~ (3) 的前提下, 可将圆盘移至 A, B, C 中任一塔座上。

试设计一个算法, 用最少的移动次数将塔座 A 上的 n 个圆盘移到塔座 B 上, 并仍按同样顺序叠置。

★编程任务:

对于给定的正整数 n , 编程计算最优移动方案。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行是给定的正整数 n 。

★结果输出:

将计算出的最优移动方案输出到文件 output.txt。文件的每一行由一个正整数 k 和 2 个字符 $c1$ 和 $c2$ 组成, 表示将第 k 个圆盘从塔座 $c1$ 移到塔座 $c2$ 上。

输入文件示例

input.txt

3

输出文件示例

output.txt

1 A B

2 A C

1 B C

3 A B

1 C A

2 C B

1 A B

分析与解答:

可用主教材中的标准 Hanoi 塔算法。问题是要证明标准 Hanoi 塔算法不违反规则 (3)。用数学归纳法。

设 $\text{hanoi}(n, A, B, C)$ 是将塔座 A 上的 n 个圆盘, 以塔座 C 为辅助塔座, 移到目的塔座 B 上的标准 Hanoi 塔算法。

归纳假设: 当圆盘个数小于 n 时, $\text{hanoi}(n, A, B, C)$ 不违反规则 (3), 且在移动过程中, 目的塔座 B 上最低圆盘的编号与 n 具有相同奇偶性, 辅助塔座 C 上最低圆盘的编号与 n 具有不同奇偶性。

当圆盘个数为 n 时, 标准 Hanoi 塔算法 $\text{hanoi}(n, A, B, C)$ 由以下 3 个步骤完成:

(1) $\text{hanoi}(n-1, A, C, B)$;

(2) $\text{move}(A, B)$;

(3) $\text{hanoi}(n-1, C, B, A)$ 。

按归纳假设, 步骤 (1) 不违反规则 (3), 且在移动过程中, 塔座 C 上最低圆盘的编号与 $n-1$ 具有相同奇偶性, 塔座 B 上最低圆盘的编号与 $n-1$ 具有不同奇偶性, 从而塔座 B 上最低圆盘的编号与 n 具有相同奇偶性, 塔座 C 上最低圆盘的编号与 n 具有不同奇偶性。

步骤 (2) 也不违反规则 (3), 且塔座 B 上最低圆盘的编号与 n 相同。

按归纳假设, 步骤 (3) 不违反规则 (3), 且在移动过程中, 塔座 B 上倒数第 2 个圆盘的编号与 $n-1$ 具有相同奇偶性, 塔座 A 上最低圆盘的编号与 $n-1$ 具有不同奇偶性, 从而塔座 B 上倒数第 2 个圆盘的编号与 n 具有不同奇偶性, 塔座 A 上最低圆盘的编号与 n 具有相同奇偶性。因此在移动过程中, 塔座 B 上圆盘不违反规则 (3), 而且塔座 B 上最低圆盘的编号与 n 具有相同奇偶性, 塔座 C 上最低圆盘的编号与 n 具有不同奇偶性。

由数学归纳法可知, $\text{hanoi}(n, A, B, C)$ 不违反规则 (3)。

算法实现题 2-13 标准二维表问题

★问题描述:

设 n 是一个正整数。 $2 \times n$ 的标准二维表是由正整数 $1, 2, \dots, 2n$ 组成的 $2 \times n$ 数组, 该数组的每行从左到右递增, 每列从上到下递增。 $2 \times n$ 的标准二维表全体记为 $\text{Tab}(n)$ 。例如, 当 $n=3$ 时, $\text{Tab}(3)$ 二维表如图 2-20 所示。

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

图 2-20 $n=3$ 时 $\text{Tab}(3)$ 二维表

★编程任务:

给定正整数 n , 计算 $\text{Tab}(n)$ 中 $2 \times n$ 的标准二维表的个数。

★数据输入:

由文件 `input.txt` 给出输入数据。第 1 行有 1 个正整数 n 。

★结果输出:

将计算出的 $\text{Tab}(n)$ 中 $2 \times n$ 的标准二维表的个数输出到文件 `output.txt`。

输入文件示例

输出文件示例

`input.txt`

`output.txt`

3

5

分析与解答:

Catalan 数。

算法实现题 2-14 整数因子分解问题

★问题描述:

大于 1 的正整数 n 可以分解为 $n = x_1 \cdot x_2 \cdot \dots \cdot x_m$ 。

例如, 当 $n=12$ 时, 共有 8 种不同的分解式:

$$12=12$$

$12=6\times 2$
 $12=4\times 3$
 $12=3\times 4$
 $12=3\times 2\times 2$
 $12=2\times 6$
 $12=2\times 3\times 2$
 $12=2\times 2\times 3$

★编程任务：

对于给定的正整数 n ，编程计算 n 共有多少种不同的分解式。

★数据输入：

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 $n(1\leq n\leq 2000000000)$ 。

★结果输出：

将计算出的不同的分解式数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
12	8

分析与解答：

对 n 的每个因子递归搜索。

```
void solve(int n)
{
    if (n==1) total++;
    else for (int i=2; i<=n; i++) if (n%i==0) solve(n/i);
}
```

此题可用动态规划求解。

第3章 动态规划

习题 3-1 最长单调递增子序列

设计一个 $O(n^2)$ 时间的算法, 找出由 n 个数组成的序列的最长单调递增子序列。

分析与解答:

用数组 $b[0:n-1]$ 记录以 $a[i]$, $0 \leq i < n$, 为结尾元素的最长递增子序列的长度。序列 a 的最长递增子序列的长度为 $\max_{0 \leq i < n} \{b[i]\}$ 。易知, $b[i]$ 满足最优子结构性质, 可以递归地定义为

$$b[0]=1, b[i]=\max_{\substack{0 \leq k < j \\ a[k] \leq a[i]}} \{b[k]\} + 1$$

据此将计算 $b[i]$ 转化为 i 个规模更小的子问题。

按此思想设计的动态规划算法描述如下。

```
int LISdyna()
{
    int i, j, k;
    for (i=1, b[0]=1; i<n; i++) {
        for (j=0, k=0; j<i; j++) if (a[j]<=a[i] && k<b[j]) k=b[j];
        b[i]=k+1;
    }
    return maxL(n);
}

int maxL(int n)
{
    for (int i=0, temp=0; i<n; i++) if (b[i]>temp) temp=b[i];
    return temp;
}
```

其中, 算法 LISdyna 按照递归式计算出 $b[0:n-1]$ 的值, 然后由 maxL 计算出序列 a 的最长递增子序列的长度。从算法 LISdyna 的二重循环容易看出, 算法所需的计算时间为 $O(n^2)$ 。

习题 3-2 最长单调递增子序列的 $O(n \log n)$ 算法

将习题 3-1 中算法的计算时间减至 $O(n \log n)$ (提示: 一个长度为 i 的候选子序列的最后一个元素至少与一个长度为 $i-1$ 的候选子序列的最后一个元素一样大。通过指向输入序列中元素的指针来维持候选子序列)。

分析与解答:

可用归纳设计策略解此问题。归纳假设是: 已知计算序列 $a[0:i-1]$ ($i < n$) 的最长递增子序列的长度的正确算法。归纳的初始情况是平凡的。对于长度为 n 的序列 $a[0:n-1]$, 应设法转换为长度小于 n 的序列。

用归纳设计策略解题时, 归纳假设对应于算法循环中的循环不变式。本题的循环不变式 P 是:

$P: k$ 是序列 $a[0:i]$ 的最长递增子序列的长度, $0 \leq i < n$ 。

容易看出, 在由 $i-1$ 到 i 的循环中, $a[i]$ 的值起关键作用。如果 $a[i]$ 能扩展序列 $a[0:i-1]$ 的最长递增子序列的长度, 则 $k=k+1$, 否则 k 不变。设 $a[0:i-1]$ 中长度为 k 的最长递增子序列的结尾元素是 $a[j]$ ($0 \leq j \leq i-1$), 则当 $a[i] \geq a[j]$ 时可以扩展, 否则不能扩展。如果序列 $a[0:i-1]$ 中有多个长度为 k 的最长递增子序列, 那么需要存储哪些信息? 容易看出, 只要存储序列 $a[0:i-1]$ 中所有长度为 k 的递增子序列中结尾元素的最小值 $b[k]$ 。因此, 需要将循环不变式 P 增强为:

$P: 0 \leq i < n; k$ 是序列 $a[0:i]$ 的最长递增子序列的长度;

$b[k]$ 是序列 $a[0:i]$ 中所有长度为 k 的递增子序列中的最小结尾元素值。

相应地, 归纳假设也增强为: 已知计算序列 $a[0:i-1]$ ($i < n$) 的最长递增子序列的长度 k 以及序列 $a[0:i-1]$ 中所有长度为 k 的递增子序列中的最小结尾元素值 $b[k]$ 的正确算法。

增强归纳假设后, 在由 $i-1$ 到 i 的循环中, 当 $a[i] \geq b[k]$ 时, $k=k+1, b[k]=a[i]$, 否则 k 值不变。注意到当 $a[i] \geq b[k]$ 时, k 值增加, $b[k]$ 的值为 $a[i]$ 。那么, 当 $a[i] < b[k]$ 时, $b[1:k]$ 的值应该如何改变? 如果 $a[i] < b[1]$, 则显然应该将 $b[1]$ 的值改变为 $a[i]$ 。当 $b[1] \leq a[i] \leq b[k]$ 时, 注意到数组 b 是有序的, 可以用二分搜索算法找到下标 j , 使得 $b[j-1] \leq a[i] < b[j]$ 。此时, $b[1:j-1]$ 和 $b[j+1:k]$ 的值不变, $b[j]$ 的值改变为 $a[i]$ 。

按上述思想设计的算法可实现如下。

```
int LIS()
{
    b[1]=a[0];
    for (int i=1, k=1; i<n; i++){
        if (a[i]>=b[k]) b[++k]=a[i];
        else b[binary(i, k)]=a[i];
    }
    return k;
}

int binary(int i, int k)
{
    if (a[i]<b[1]) return 1;
    for(int h=1, j=k; h!=j-1;){
        if (b[k=(h+j)/2]<=a[i]) h=k;
        else j=k;}
    return j;
}
```

其中, $\text{binary}(i, k)$ 用二分搜索算法在 $b[1:k]$ 中找到下标 j , 使得 $b[j-1] \leq a[i] < b[j]$ 。算法 $\text{binary}(i, k)$ 所需的计算时间显然为 $O(\log k)$ 。由此可见, 在最坏情况下, 上述算法所需的计算时间为 $O(n \log n)$ 。

习题 3-7 漂亮打印

给定由 n 个英文单词组成的一段文章，每个单词的长度（字符个数）依序为 l_1, l_2, \dots, l_n 。要在—台打印机上将这段文章“漂亮”地打印出来。打印机每行最多可打印 M 个字符。这里所说的“漂亮”的定义如下：在打印机所打印的每一行中，行首和行尾可不留空格；行中每两个单词之间留一个空格；如果在一行中打印从单词 i 到单词 j 的字符，则按打印规则，应在一行中恰好打印 $\sum_{k=i}^j l_k + j - i$ 个字符（包括字间空格字符），且不允许将单词打破；多余的空格数为 $M - j + i - \sum_{k=i}^j l_k$ ；除文章的最后一行外，希望每行多余的空格数尽可能少。因此，以各行（最后一行除外）的多余空格数的立方和达到最小作为“漂亮”的标准。试用动态规划算法设计一个“漂亮打印”方案，并分析算法的计算复杂性。

分析与解答：

见参考文献[1], 60~61。

习题 3-11 整数线性规划问题

考虑下面的整数线性规划问题

$$\begin{aligned} \max \quad & \sum_{i=1}^n c_i x_i \\ \text{s.t.} \quad & \begin{cases} \sum_{i=1}^n a_i x_i \leq b \\ x_i \text{ 为非负整数, } 1 \leq i \leq n \end{cases} \end{aligned}$$

试设计一个解此问题的动态规划算法，并分析该算法的计算复杂性。

分析与解答：

该问题是一般情况下的背包问题，具有最优子结构性质。

设所给背包问题的子问题

$$\begin{aligned} \max \quad & \sum_{k=1}^i c_k x_k \\ \text{s.t.} \quad & \sum_{k=1}^i a_k x_k \leq j \end{aligned}$$

的最优值为 $m(i, j)$ ，即 $m(i, j)$ 是背包容量为 j ，可选择物品为 $1, 2, \dots, i$ 时背包问题的最优值。由背包问题的最优子结构性质，可以建立计算 $m(i, j)$ 的递归式如下：

$$m(i, j) = \begin{cases} \max\{m(i-1, j), m(i, j-a_i) + c_i\} & a_i \leq j \\ m(i-1, j) & 0 \leq j < a_i \end{cases}$$

$$m(0, j) = m(i, 0) = 0; m(i, j) = -\infty, j < 0$$

按此递归式计算出的 $m(n, b)$ 为最优值。算法所需的计算时间为 $O(nb)$ 。

习题 3-12 二维 0-1 背包问题

给定 n 种物品和一个背包。物品 i 的重量是 w_i ，体积是 b_i ，其价值为 v_i ，背包的容量为 c ，容积为 d 。问应如何选择装入背包中的物品，使得装入背包中物品的总价值最大？在选择装入背包的物品时，对每种物品 i 只有两种选择，即装入背包或不装入背包。不能将物品

i 装入背包多次, 也不能只装入部分的物品 i 。试设计一个解此问题的动态规划算法, 并分析算法的计算复杂性。

分析与解答:

该问题是二维 0-1 背包问题。问题的形式化描述是: 给定 $c > 0, d > 0, w_i > 0, b_i > 0, v_i > 0, 1 \leq i \leq n$, 要求找出 n 元 0-1 向量 $(x_1, x_2, \dots, x_n), x_i \in \{0, 1\}, 1 \leq i \leq n$, 使得 $\sum_{i=1}^n w_i x_i \leq c, \sum_{i=1}^n b_i x_i \leq d$ 而且 $\sum_{i=1}^n v_i x_i$ 达到最大。

因此, 二维 0-1 背包问题也是一个特殊的整数规划问题。

$$\begin{aligned} \max \quad & \sum_{i=1}^n v_i x_i \\ \text{s.t.} \quad & \begin{cases} \sum_{i=1}^n w_i x_i \leq c \\ \sum_{i=1}^n b_i x_i \leq d \\ x_i \in \{0, 1\}, 1 \leq i \leq n \end{cases} \end{aligned}$$

容易证明该问题具有最优子结构性质。

设所给二维 0-1 背包问题的子问题

$$\begin{aligned} \max \quad & \sum_{i=t}^n v_i x_i \\ \text{s.t.} \quad & \begin{cases} \sum_{i=t}^n w_i x_i \leq j \\ \sum_{i=t}^n b_i x_i \leq k \\ x_i \in \{0, 1\}, i \leq t \leq n \end{cases} \end{aligned}$$

的最优值为 $m(i, j, k)$, 即 $m(i, j, k)$ 是背包容量为 j , 容积为 k , 可选择物品为 $i, i+1, \dots, n$ 时二维 0-1 背包问题的最优值。由二维 0-1 背包问题的最优子结构性质, 可以建立计算 $m(i, j, k)$ 的递归式如下:

$$\begin{aligned} m(i, j, k) &= \begin{cases} \max\{m(i+1, j), m(i+1, j-w_i, k-b_i) + v_i\} & j \geq w_i \text{ and } k \geq b_i \\ m(i+1, j) & 0 \leq j < w_i \text{ or } 0 \leq k < b_i \end{cases} \\ m(n, j, k) &= \begin{cases} v_n & j \geq w_n \text{ and } k \geq b_n \\ 0 & 0 \leq j < w_n \text{ or } 0 \leq k < b_n \end{cases} \end{aligned}$$

按此递归式计算出的 $m(n, c, d)$ 为最优值。算法所需的计算时间为 $O(ncd)$ 。

习题 3-14 Ackermann 函数

Ackermann 函数 $A(m, n)$ 可递归地定义如下:

$$A(m, n) = \begin{cases} n+1 & m=0 \\ A(m-1, 1) & m>0, n=0 \\ A(m-1, A(m, n-1)) & m>0, n>0 \end{cases}$$

试设计一个计算 $A(m, n)$ 的动态规划算法, 该算法只占用 $O(m)$ 空间 (提示: 用 2 个数

组 $val[0:m]$ 和 $ind[0:m]$, 使得对任何 i 有 $val[i] = A(i, ind[i])$ 。

分析与解答:

按定义容易写出递归算法。

```
int ackermann(int m, int n)
{
    if(m == 0) return n+1;
    if(n == 0) return ackermann(m-1, 1);
    else return ackermann(m-1, ackermann(m, n-1));
}
```

按备忘录方法的思想, 可将上述算法修改为备忘录算法。

```
int ack(int m, int n)
{
    if(a[m][n]) return a[m][n];
    if(m==0) return a[0][n]=n+1;
    if(n==0) return a[m][0]=ack(m-1, 1);
    return a[m][n]=ack(m-1, ack(m, n-1));
}
```

另外, 还可用消去递归的方法, 将上述算法非递归化如下。

```
int ackm(int m, int n)
{
    int top=1;
    s[1][1]=m; s[1][2]=n;
    while (top>0) {
        m=s[top][1]; n=s[top][2]; top--;
        if (top==0 && m==0) return n+1;
        if (m==0) s[top][2]=n+1;
        else if (n==0) {s[++top][1]=m-1; s[top][2]=1;}
        else {s[++top][1]=m-1; s[top+top][1]=m; s[top][2]=n-1;}
    }
    return s[0][1];
}
```

实际上, 还有一个稍简单的递归算法。

```
int ack(int m, int n)
{
    for(int i=m; i>0; i--) {
        if (n==0) n=1;
        else n=ack(i, n-1);
    }
```

```

    }
    return n+1;
}

```

同样也可将其改造为备忘录算法。

这些算法都是自顶向下的递归算法。下面考察自底向上的动态规划算法。

首先对于较小的 m 和 n ，考察 Ackermann 函数 $A(m,n)$ 值的变化，如图 3-1 所示。

$m \backslash n$	0	1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9	10	11
1	2	3	4	5	6	7	8	9	10	11	12
2	3	5	7	9	11	13	15	17	19	21	23
3	5	13	29	61	125	253	509	1021	2045	4093	8189
4	13	65533									
5	65533										

图 3-1 Ackermann 函数 $A(m,n)$ 值的变化

从图 3-1 容易看出，当 $m=0$ 时，第 0 行对应于 $A(0,n)$ 的值。当 $n=0$ 时，第 0 列对应于 $A(m,0)$ 的值，其值恰好是第 $m-1$ 行第 1 列的值。 $A(m,n)$ 的值等于第 $m-1$ 行第 $A(m,n-1)$ 列的值。据此，可从第 1 行的值依次递推出各行的值。为此，用 2 个数组 $val[0:m]$ 和 $ind[0:m]$ 分别记录当前第 i 行计算到第 $ind[i]$ 列的值，即已计算到 $A(i, ind[i])$ 的值，这个值存储于 $val[i]$ 中。当计算到 $A(m,n)$ 时，算法结束。按此思想设计的动态规划算法如下。

```

int ack(int m, int n)
{
    int i,*val,*ind;
    if(m==0)return n+1;
    val=new int[m+1];
    ind=new int[m+1];
    for(i=0;i<=m;i++){val[i]=-1;ind[i]=-2;}
    val[0]=1;ind[0]=0;
    while(ind[m]<n){
        val[0]++;ind[0]++;
        for(i=0;i<m;i++){
            if(ind[i]==1 && ind[i+1]<0){val[i+1]=val[0];ind[i+1]=0;}
            if(val[i+1]==ind[i]){val[i+1]=val[0];ind[i+1]++;}
        }
    }
    return val[m];
}

```

算法显然只占用 $O(m)$ 空间。

习题 3-17 最短行驶路线

给定一个 $m \times n$ 的矩形网格，设其左上角为起点 S 。一辆汽车从起点 S 出发驶向右下角

终点 T 。网格边上的数字表示距离。在若干网格点处设置了障碍,表示该网格点不可到达。试设计一个算法,求出汽车从起点 S 出发到达终点 T 的一条行驶路程最短的路线。

分析与解答:

与习题 3-16 的解答类似。

习题 3-19 最优旅行路线

给定一张航空图,图中的顶点表示城市,边表示城市间的直通航线。试设计一个算法,计算出一条满足下述约束条件且含城市最多的旅行路线。

从最西端的城市出发,单方向由西向东到达最东端的城市。然后,再单方向由东向西飞回起点(可途经若干城市)。

除起点城市外,每个城市最多只经过一次。

分析与解答:

参见习题 8-24 的解答。

算法实现题 3-1 独立任务最优调度问题 (习题 3-3)

★问题描述:

用 2 台处理机 A 和 B 处理 n 个作业。设第 i 个作业交给机器 A 处理时需要时间 a_i ,若由机器 B 来处理,则需要时间 b_i 。由于各作业的特点和机器的性能关系,很可能对于某些 i ,有 $a_i \geq b_i$,而对于某些 $j, j \neq i$,有 $a_j < b_j$ 。既不能将一个作业分开由 2 台机器处理,也没有一台机器能同时处理 2 个作业。设计一个动态规划算法,使得这 2 台机器处理完这 n 个作业的时间最短(从任何一台机器开工到最后一台机器停工的总时间)。研究一个实例: $(a_1, a_2, a_3, a_4, a_5, a_6) = (2, 5, 7, 10, 5, 2); (b_1, b_2, b_3, b_4, b_5, b_6) = (3, 8, 4, 11, 3, 4)$ 。

★编程任务:

对于给定的 2 台处理机 A 和 B 处理 n 个作业,找出一个最优调度方案,使 2 台机器处理完这 n 个作业的时间最短。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行是 1 个正整数 n ,表示要处理 n 个作业。在接下来的 2 行中,每行有 n 个正整数,分别表示处理机 A 和 B 处理第 i 个作业需要的处理时间。

★结果输出:

程序运行结束时,将计算出的最短处理时间输出到文件 output.txt 中。

输入文件示例	输出文件示例
input.txt	output.txt
6	15
2 5 7 10 5 2	
3 8 4 11 3 4	

分析与解答:

(1) 首先计算出 $m = \max\{\max_{1 \leq i \leq n}\{a_i\}, \max_{1 \leq i \leq n}\{b_i\}\}$ 。

(2) 设布尔量 $p(i, j, k)$ 表示前 k 个作业可以在处理机 A 用时不超过 i 时间且在处理机 B 用时不超过 j 时间内完成。用动态规划算法计算

$$p(i, j, k) = p(i - a_k, j, k - 1) \vee p(i, j - b_k, k - 1)$$

(3) 由 (2) 的结果可以计算最短处理时间如下:

$$\min_{0 \leq i \leq mn, 0 \leq j \leq mn, p(i, j, n) = \text{true}} \{\max\{i, j\}\}$$

具体算法实现如下。

read 读入初始数据, 并计算 m 的值。

```
void read()
{
    cin >> n; m = 0;
    a = new int[n]; b = new int[n];
    for (int i = 0; i < n; i++) {cin >> a[i]; if (a[i] > m) m = a[i];}
    for (i = 0; i < n; i++) {cin >> b[i]; if (b[i] > m) m = a[i];}
    mn = m * n;
    Make3DArray(p, mn + 1, mn + 1, n + 1);
}
```

dyna 实现动态规划算法。

```
void dyna()
{
    int i, j, k;
    for (i = 0; i <= mn; i++)
        for (j = 0; j <= mn; j++) {
            p[i][j][0] = true;
            for (k = 1; k <= n; k++) p[i][j][k] = false;
        }
    for (k = 1; k <= n; k++)
        for (i = 0; i <= mn; i++)
            for (j = 0; j <= mn; j++) {
                if (i - a[k - 1] >= 0) p[i][j][k] = p[i - a[k - 1]][j][k - 1];
                if (j - b[k - 1] >= 0) p[i][j][k] = (p[i][j][k] || p[i][j - b[k - 1]][k - 1]);
            }
    for (i = 0, opt = mn; i <= mn; i++)
        for (j = 0; j <= mn; j++)
            if (p[i][j][n]) {
                int tmp = (i > j) ? i : j;
                if (tmp < opt) opt = tmp;
            }
    cout << opt << endl;
}
```

实现算法的主函数如下。

```
int main()
{
```

```

    read();
    dyna();
    return 0;
}

```

上述算法所需的计算时间显然为 $O(m^2n^3)$ 。

该问题的另一解法见参考文献[1], 53~54。

算法实现题 3-2 最少硬币问题 (习题 3-4)

★问题描述:

设有 n 种不同面值的硬币, 各硬币的面值存于数组 $T[1:n]$ 中。现要用这些面值的硬币来找钱。可以使用的各种面值的硬币个数存于数组 $Coins[1:n]$ 中。

对任意钱数 $0 \leq m \leq 20001$, 设计一个用最少硬币找钱 m 的方法。

★编程任务:

对于给定的 $1 \leq n \leq 10$, 硬币面值数组 T 和可以使用的各种面值的硬币个数数组 $Coins$, 以及钱数 m , $0 \leq m \leq 20001$, 编程计算找钱 m 的最少硬币数。

★数据输入:

由文件 input.txt 提供输入数据, 文件的第 1 行中只有 1 个整数给出 n 的值, 第 2 行起每行 2 个数, 分别是 $T[j]$ 和 $Coins[j]$ 。最后 1 行是要找的钱数 m 。

★结果输出:

程序运行结束时, 将计算出的最少硬币数输出到文件 output.txt 中。问题无解时输出一 1。

输入文件示例

输出文件示例

input.txt

output.txt

3

5

1 3

2 3

5 3

18

分析与解答:

见参考文献 [1], 54~57。

算法实现题 3-3 序关系计数问题 (习题 3-5)

★问题描述:

用关系 “ $<$ ” 和 “ $=$ ” 将 3 个数 A, B 和 C 依序排列时有 13 种不同的序关系:

$A=B=C, A=B<C, A<B=C, A<B<C, A<C<B, A=C<B$

$B<A=C, B<A<C, B<C<A, B=C<A, C<A=B, C<A<B, C<B<A$

将 n 个数 ($1 \leq n \leq 50$) 依序排列时有多少种序关系。

★编程任务:

编程计算出将 n 个数 ($1 \leq n \leq 50$) 依序排列时有多少种序关系。

★数据输入:

由文件 input.txt 提供输入数据。文件只有一行, 提供一个数 n 。

★结果输出:

程序运行结束时, 将找到的序关系数输出到文件 output.txt 的第 1 行中。

输入文件示例

输出文件示例

input.txt

output.txt

3

13

分析与解答:

见参考文献[1], 57~59。

算法实现题 3-4 多重幂计数问题 (习题 3-6)

★问题描述:

设给定 n 个变量 x_1, x_2, \dots, x_n 。将这些变量依序作为底和各层幂, 可得 n 重幂如下:

$$\begin{matrix} & & & & x_n \\ & & & \cdot & \\ & & & x_3 & \\ & & \cdot & & \\ & & x_2 & & \\ & \cdot & & & \\ x_1 & & & & \end{matrix}$$

这里将上述 n 重幂看作是不确定的, 当在其中加入适当的括号后, 才能成为一个确定的 n 重幂。不同的加括号方式导致不同的 n 重幂。例如, 当 $n=4$ 时, 全部 4 重幂有 5 个。

★编程任务:

对 n 个变量计算出有多少个不同的 n 重幂。

★数据输入:

由文件 input.txt 提供输入数据。文件只有一行, 提供一个数 n 。

★结果输出:

程序运行结束时, 将找到的序关系数输出到文件 output.txt 的第 1 行中。

输入文件示例

输出文件示例

input.txt

output.txt

4

5

分析与解答:

见参考文献[1], 59~60。

算法实现题 3-5 编辑距离问题 (习题 3-8)

★问题描述:

设 A 和 B 是 2 个字符串。要用最少的字符操作将字符串 A 转换为字符串 B 。这里所说的字符操作包括:

- (1) 删除一个字符;
- (2) 插入一个字符;
- (3) 将一个字符改为另一个字符。

将字符串 A 变换为字符串 B 所用的最少字符操作数称为字符串 A 到 B 的编辑距离, 记为 $d(A, B)$ 。试设计一个有效算法, 对任给的 2 个字符串 A 和 B , 计算出它们的编辑距离 $d(A, B)$ 。

★编程任务:

对于给定的字符串 A 和字符串 B , 编程计算其编辑距离 $d(A, B)$ 。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行是字符串 A , 文件的第 2 行是字符串 B 。

★结果输出:

程序运行结束时, 将编辑距离 $d(A, B)$ 输出到文件 output.txt 的第 1 行中。

输入文件示例

输出文件示例

input.txt

output.txt

fxpimu

5

xwrs

分析与解答:

见参考文献[1], 61~63。算法实现如下。

```
int dist()
{
    int m=a.size();
    int n=b.size();
    vector<int>d(n+1,0);
    for (int i = 1; i <= n; i++) d[i]=i;
    for (i = 1; i <= m; i++){
        int y=i-1;
        for (int j=1; j<=n; j++){
            int x=y;
            y=d[j];
            int z=j>1? d[j-1]:i;
            int del= a[i-1]==b[j-1]? 0:1;
            d[j]=min(x+del, y+1, z+1);
        }
    }
    return d[n];
}
```

算法实现题 3-6 石子合并问题 (习题 3-9)

★问题描述:

在一个圆形操场的四周摆放着 n 堆石子。现要将石子有次序地合并成一堆。规定每次只能选相邻的 2 堆石子合并成新的一堆, 并将新的一堆石子数记为该次合并的得分。试设计一个算法, 计算出将 n 堆石子合并成一堆的最小得分和最大得分。

★编程任务:

对于给定 n 堆石子, 编程计算合并成一堆的最小得分和最大得分。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行是正整数 $n, 1 \leq n \leq 100$, 表示有 n 堆石子。第 2 行有 n 个数, 分别表示每堆石子的个数。

★结果输出:

程序运行结束时, 将计算结果输出到文件 output.txt 中。文件第 1 行中的数是最小得分, 第 2 行中的数是最大得分。

输入文件示例

输出文件示例

input.txt

output.txt

4

43

4 4 5 9

54

分析与解答:

见参考文献 [1], 63~66。算法实现如下。

```
int circle (int ss)
{
    for (int i = 1; i <= n-1; i++) a[n+i]=a[i];
    n = 2*n-1;
    if ((ss==1)) minsum(a);
    else maxsum(a);
    n = (n+1)/2;
    int mm = m[1][n];
    for (i = 2; i <= n; i++)
        if ((ss==1) && (m[i][n+i-1]<mm) || (ss>1) && (m[i][n+i-1]>mm))
            mm = m[i][n+i-1];
    return(mm);
}
```

算法 circle (int ss)解圆排列的石子合并问题。当 ss=1 时, 计算最小得分; 当 ss>1 时, 计算最大得分。

算法 minsum 解直线排列的最小得分石子合并问题。

```
void minsum (vector<int> a)
{
    for (int i = 2; i <= n; i++) a[i] = a[i] + a[i-1];
    for (int r = 2; r <= n; r++)
        for (i = 1; i <= n-r+1; i++) {
            int j = i+r-1;
            int il = i+1;
            int jl = j;
            if ((s[i][j-1] > i))
                il = s[i][j-1];
            if ((s[i+1][j] > i))
                jl = s[i+1][j];
            m[i][j] = m[i][il-1] + m[i1][j];
            s[i][j] = il;
            for (int k = jl; k >= il+1; k--) {
```

```

        int q = m[i][k-1] + m[k][j];
        if ((q < m[i][j])) {
            m[i][j] = q;
            s[i][j] = k;
        }
    }
    m[i][j] = m[i][j] + a[j] - a[i-1];
}
}

```

算法 maxsum 解直线排列的最大得分石子合并问题。

```

void maxsum (vector<int> a)
{
    for (int i = 2; i <= n; i++) a[i] = a[i] + a[i-1];
    for (int r = 2; r <= n; r++)
        for (i = 1; i <= n-r+1; i++) {
            int j = i+r-1;
            if (m[i+1][j] > m[i][j-1])
                m[i][j] = m[i+1][j] + a[j] - a[i-1];
            else
                m[i][j] = m[i][j-1] + a[j] - a[i-1];
        }
}

```

算法实现题 3-7 数字三角形问题 (习题 3-10)

★问题描述:

给定一个由 n 行数字组成的数字三角形,如图 3-2 所示。试设计一个算法,计算出从三角形的顶至底的一条路径,使该路径经过的数字总和最大。

★编程任务:

对于给定的由 n 行数字组成的数字三角形,编程计算从三角形的顶至底的路径经过的数字和的最大值。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行是数字三角形的行数 $n, 1 \leq n \leq 100$ 。接下来 n 行是数字三角形各行中的数字。所有数字在 $0 \sim 99$ 之间。

★结果输出:

程序运行结束时,将计算结果输出到文件 output.txt 中。文件第 1 行中的数是计算出的最大值。

```

      7
     3 8
    8 1 0
   2 7 4 4
  4 5 2 6 5

```

图 3-2 数字三角形

输入文件示例

input.txt

5

7

3 8

8 1 0

2 7 4 4

4 5 2 6 5

输出文件示例

output.txt

30

分析与解答:

以自底向上的方式递归计算。

```
for(int row = n - 2; row >= 0; row--)
    for (int col = 0; col <= row; col++)
        if (triangle[row+1][col] > triangle[row+1][col+1])
            triangle[row][col] += triangle[row+1][col];
        else
            triangle[row][col] += triangle[row+1][col+1];
```

最优值在 `triangle[0][0]` 中。

算法实现题 3-8 乘法表问题 (习题 3-13)

★问题描述:

定义于字母表 $\Sigma = \{a, b, c\}$ 上的乘法表如表 3-1 所示。

表 3-1 Σ 乘法表

	a	b	c
a	b	b	a
b	c	b	a
c	a	c	c

依此乘法表, 对任一定义于 Σ 上的字符串, 适当加括号后得到一个表达式。例如, 对于字符串 $x = bbbba$, 它的一个加括号表达式为 $(b(bb))(ba)$ 。依乘法表, 该表达式的值为 a 。试设计一个动态规划算法, 对任一定义于 Σ 上的字符串 $x = x_1x_2 \cdots x_n$, 计算有多少种不同的加括号方式, 使由 x 导出的加括号表达式的值为 a 。

★编程任务:

对于给定的字符串 $x = x_1x_2 \cdots x_n$, 计算有多少种不同的加括号方式, 使由 x 导出的加括号表达式的值为 a 。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行中给出一个字符串。

★结果输出:

程序运行结束时, 将计算结果输出到文件 output.txt 中。文件的第 1 行中的数是计算出的加括号方式数。

输入文件示例

input.txt

bbbba

输出文件示例

output.txt

6

分析与解答:

```

void dyna()
{
    for(int k=1;k<n;k++){
        for(int i=0;i<n-k;i++){
            int j=i+k;
            for(int p=i;p<j;p++){
                f[i][j][0]+=f[i][p][2]*f[p+1][j][0]+(f[i][p][0]+f[i][p][1])*
                f[p+1][j][2];
                f[i][j][1]+=f[i][p][0]*f[p+1][j][0]+(f[i][p][0]+f[i][p][1])*
                f[p+1][j][1];
                f[i][j][2]+=f[i][p][1]*f[p+1][j][0]+f[i][p][2]*(f[p+1][j][1]+
                f[p+1][j][2]);
            }
        }
    }
}

```

最优值在 $f[0][n-1][0]$ 中。

算法实现题 3-9 租用游艇问题 (习题 3-15)

★问题描述:

长江游艇俱乐部在长江上设置了 n 个游艇出租站 $1, 2, \dots, n$ 。游客可在这些游艇出租站租用游艇, 并在下游的任何一个游艇出租站归还游艇。游艇出租站 i 到游艇出租站 j 之间的租金为 $r(i, j)$, $1 \leq i < j \leq n$ 。试设计一个算法, 计算出从游艇出租站 1 到游艇出租站 n 所需的最少租金。

★编程任务:

对于给定的游艇出租站 i 到游艇出租站 j 之间的租金为 $r(i, j)$, $1 \leq i < j \leq n$, 编程计算从游艇出租站 1 到游艇出租站 n 所需的最少租金。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行中有 1 个正整数 n ($n \leq 200$), 表示有 n 个游艇出租站。接下来的 $n-1$ 行是 $r(i, j)$, $1 \leq i < j \leq n$ 。

★结果输出:

程序运行结束时, 将计算出的从游艇出租站 1 到游艇出租站 n 所需的最少租金输出到文件 output.txt 中。

输入文件示例

输出文件示例

input.txt

output.txt

3

12

5 15

7

分析与解答:

```

void dyna()

```

```

{
    for(int k=2;k<n;k++){
        for(int i=0;i<n-k;i++){
            int j=i+k;
            for(int p=i+1;p<j;p++){
                int tmp=f[i][p]+f[p][j];
                if (f[i][j]>tmp) f[i][j]=tmp;
            }
        }
    }
}

```

最优值在 $f[0][n-1]$ 中。

算法实现题 3-10 汽车加油行驶问题 (习题 3-16)

★问题描述:

给定一个 $N \times N$ 的方形网格, 设其左上角为起点 \odot , 坐标为 $(1,1)$, X 轴向右为正, Y 轴向下为正, 每个方格边长为 1。一辆汽车从起点 \odot 出发驶向右下角终点 \blacktriangle , 其坐标为 (N,N) 。在若干网格交叉点处, 设置了油库, 可供汽车在行驶途中加油。汽车在行驶过程中应遵守如下规则。

(1) 汽车只能沿网格边行驶, 装满油后能行驶 K 条网格边。出发时汽车已装满油, 在起点与终点处不设油库。

(2) 当汽车行驶经过一条网格边时, 若其 X 坐标或 Y 坐标减小, 则应付费 B , 否则免付费。

(3) 汽车在行驶过程中遇油库则应加满油并付加油费用 A 。

(4) 在需要时可在网格点处增设油库, 并付增设油库费用 C (不含加油费用 A)。

(5) (1) ~ (4) 中的各数 N, K, A, B, C 均为正整数。

★编程任务:

求汽车从起点出发到达终点的一条所付费用最少的行驶路线。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行是 N, K, A, B, C 的值, $2 \leq N \leq 100$, $2 \leq K \leq 10$ 。第 2 行起是一个 $N \times N$ 的 0-1 方阵, 每行 N 个值, 至 $N+1$ 行结束。方阵的第 i 行第 j 列处的值为 1 表示在网格交叉点 (i,j) 处设置了一个油库, 为 0 时表示未设油库。各行相邻的 2 个数以空格分隔。

★结果输出:

程序运行结束时, 将找到的最优行驶路线所需的费用, 即最小费用输出到文件 output.txt 中。文件的第 1 行中的数是最低费用值。

输入文件示例

input.txt

9 3 2 3 6

0 0 0 0 1 0 0 0 0

0 0 0 1 0 1 1 0 0

输出文件示例

output.txt

12

```

1 0 1 0 0 0 0 1 0
0 0 0 0 0 1 0 0 1
1 0 0 1 0 0 1 0 0
0 1 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0 1
1 0 0 1 0 0 0 1 0
0 1 0 0 0 0 0 0 0

```

分析与解答:

用 $f(x, y, 0)$ 表示汽车从网格点 $(1, 1)$ 行驶至网格点 (x, y) 所需的最少费用, $f(x, y, 1)$ 表示汽车行驶至网格点 (x, y) 后还能行驶的网格边数。

建立计算 $f(x, y, 0)$ 和 $f(x, y, 1)$ 的递归式如下:

$$f(1, 1, 0) = 0, f(1, 1, 1) = K$$

$$f(x, y, 0) = f(x, y, 0) + A, f(x, y, 1) = K, (x, y) \text{ 是油库}$$

$$f(x, y, 0) = f(x, y, 0) + C + A, f(x, y, 1) = K, (x, y) \text{ 非油库且 } f(x, y, 1) = 0$$

$$f(x, y, 0) = \min_{0 \leq i < 4} \{f(x + s[i][0], y + s[i][1], 0) + s[i][2]\}$$

$$f(x, y, 1) = f(x + s[j][0], y + s[j][1], 1) - 1$$

其中, 数组 $s = \{\{-1, 0, 0\}, \{0, -1, 0\}, \{1, 0, B\}, \{0, 1, B\}\}$ 。

用备忘录方法递归计算。 $f(n, n, 0)$ 为最优值。

算法实现题 3-11 最小 m 段和问题

★问题描述:

给定 n 个整数组成的序列, 现在要求将序列分割为 m 段, 每段子序列中的数在原序列中连续排列。如何分割才能使这 m 段子序列的和的最大值达到最小?

★编程任务:

给定 n 个整数组成的序列, 编程计算该序列的最优 m 段分割, 使 m 段子序列的和的最大值达到最小。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行中有 2 个正整数 n 和 m 。正整数 n 是序列的长度; 正整数 m 是分割的段数。接下来的一行中有 n 个整数。

★结果输出:

程序运行结束时, 将计算结果输出到文件 output.txt 中。文件的第 1 行中的数是计算出的 m 段子序列的和的最大值的最小值。

输入文件示例	输出文件示例
input.txt	output.txt
1 1	10
10	

分析与解答:

见主教材 56~57。

```
void solve(int n, int m)
```

```

{
    int i, j, k, temp, maxt;
    for (i=1; i<=n; i++) f[i][1]=f[i-1][1]+t[i];
    for (j=2; j<=m; j++)
        for (i=j; i<=n; i++) {
            for (k=1, temp=INT_MAX; k<i; k++) {
                maxt=max(f[i][1]-f[k][1], f[k][j-1]);
                if (temp>maxt) temp=maxt;
            }
            f[i][j]=temp;
        }
}
}

```

最优值在 $f[n][m]$ 中。

```

void main()
{
    int n, m;
    cin>>n>>m;
    if ((n<m) || (n==0)) {cout<<0<<endl;return;}
    f.resize(n+1,m+1);
    t.resize(n+1);
    for (int i=1; i<=n; i++) cin>>t[i];
    solve(n,m);
    cout<<f[n][m]<<endl;
}

```

算法实现题 3-12 圈乘运算问题 (习题 3-18)

★问题描述:

关于整数的二元圈乘运算 \otimes 定义为:

$(X \otimes Y) = \text{十进制整数 } X \text{ 的各位数字之和} \times \text{十进制整数 } Y \text{ 的最大数字} + Y \text{ 的最小数字}$

例如, $(9 \otimes 30) = 9 \times 3 + 0 = 27$ 。

对于给定的十进制整数 X 和 K , 由 X 和 \otimes 运算可以组成各种不同的表达式。试设计一个算法, 计算出由 X 和 \otimes 运算组成的值为 K 的表达式最少需用多少个 \otimes 运算。

★编程任务:

给定十进制整数 X 和 K ($1 \leq X, K \leq 10^{20}$)。编程计算由 X 和 \otimes 运算组成的值为 K 的表达式最少需用多少个 \otimes 运算。

★数据输入:

输入数据由文件名为 input.txt 的文本文件提供。每一行有 2 个十进制整数 X 和 K 。最后一行是 0 0。

★结果输出:

程序运行结束时, 将找到的最少 \otimes 运算个数输出到文件 output.txt 中。

输入文件示例

input.txt

3 12

0 0

输出文件示例

output.txt

1

分析与解答:

(1) \otimes 运算一般不满足交换律和结合律。

(2) 最优子结构性质: 设 \otimes 运算表达式 $(X \otimes Y) = K$ 用了最少的 \otimes 运算, 则 X 和 Y 也用了最少的 \otimes 运算。

(3) \otimes 运算表达式值的有限性: 对于给定的 N , 其十进制位数为 $m = \lceil \log_{10}(N+1) \rceil$ 。易知, 当 $m > 1$ 时, \otimes 运算表达式最大值不超过 $81 \times m + 9$; 当 $m = 1$ 时, \otimes 运算表达式最大值不超过 171。

设对于给定的 N , \otimes 运算表达式的值域为 $S(N)$, 则 $S(N) \subseteq [1:L] \cup \{N\}$, 其中,

$$L = 81 \times \max\{2, \lceil \log_{10}(N+1) \rceil\} + 9$$

因此, 对于给定的 N , \otimes 运算表达式最大值为 $O(\log N)$ 量级。

(4) 从 N 出发, 反复用 N 和 \otimes 运算可求得 $S(N)$ 中每一个数。在计算过程中, 用动态规划算法求 $S(N)$ 中每一个数所用的最少 \otimes 运算次数。

(5) 数据结构: 用数组 $\text{num}[0:L][0:3]$ 存储 $S(N)$ 中每个数的相关信息。

对于任意正整数 x , 用 $\text{sum}(x)$, $\text{max}(x)$ 和 $\text{min}(x)$ 分别记其各位数字之和、最大数字和最小数字。 $\text{num}[i][0]$ 存储 i 所需的最少 \otimes 运算次数, $\text{num}[i][1]$ 存储 $\text{min}(i)$, $\text{num}[i][2]$ 存储 $\text{max}(i)$, $\text{num}[i][3]$ 存储 $\text{sum}(i)$ 。 $\text{min}(N)$, $\text{max}(N)$ 和 $\text{sum}(N)$ 分别存储于 $\text{num}[0][1]$, $\text{num}[0][2]$ 和 $\text{num}[0][3]$ 中。

(6) 算法实现。

首先由 input 输入 N 和 K 。计算 L , 当 $K > L$ 时, 明显无解。

```
int input()
{
    cout<<"N = "; cin>>s1;
    cout<<"K = "; cin>>s2;
    if (strcmp(s1,s2)==0) {out(0);return 0;}
    len=strlen(s1);
    big=81*len+9;
    if (big<171) big=171;
    int biglen=ceil(log10(big));
    if (strlen(s2)>biglen) {out(-1);return 0;}
    kk=atoi(s2);
    if (kk > big) {out(-1);return 0;}
    return 1;
}
```

然后, 由 init 初始化数组 num。

```
void init()
```

```

{
    Make2DArray(num, ++big, 4);
    for (int i=0; i<big; i++){
        num[i][0]=INT_MAX;
        num[i][1]=INT_MAX;
        num[i][2]=0; num[i][3]=0;
    }
    for(i=0; i<len; i++) count(ctoi(sl[i]), 0);
    num[0][0]=0;
    for (i=1; i<big; i++){
        int t=i;
        while(t>0) {int j=t%10; t/=10; count(j, i);}
    }
}

```

其中用到的函数 ctoi, count 和 out 如下。

```

int ctoi(char x)
{
    return (int)x - 48;
}

void count(int i, int j)
{
    if (i < num[j][1]) num[j][1] = i;
    if (i > num[j][2]) num[j][2] = i;
    num[j][3] += i;
}

void out(int x)
{
    if (x >= 0) cout << "The number of ⊗ is " << x << endl;
    else cout << "No answer!" << endl;
}

```

算法 compute 从 N 出发, 反复用 N 和 \otimes 运算求 $S(N)$ 中每一个数。在计算过程中, 用动态规划算法求 $S(N)$ 中每一个数所用的最少 \otimes 运算次数。

```

void compute()
{
    bool flag=true;
    while(flag){
        flag=false;
        for (int i=0; i<big; i++)

```



```

        if (num[i][0]<INT_MAX)
            for (int k=0;k<big;k++)
                if (num[k][0]<INT_MAX) {
                    int j=num[i][3]*num[k][2]+num[k][1];
                    if (num[i][0]+num[k][0]+1<num[j][0]) {
                        num[j][0]=num[i][0]+num[k][0]+1;
                        flag=true;
                    }
                }
    }
    if (num[kk][0]<INT_MAX) out(num[kk][0]);
    else out(-1);
}

```

完成整个计算的主函数 main 如下。

```

int kk, len, big, **num;
char s1[20], s2[20];

void main()
{
    if(input()) init();
    else return;
    compute();
}

```

(7) 算法的计算复杂性。

算法所需空间显然为 $O(\log N)$ 。

算法所需计算时间为核心算法 compute 的计算时间。算法 compute 的 1 次 while 循环需要 $O(\log^2 N)$ 的计算时间。在最坏情况下需要 $O(\log N)$ 次 while 循环。因此，算法在最坏情况下所需的计算时间为 $O(\log^3 N)$ 。

(8) 算法的改进。

设 x 和 y 是 2 个正整数，当 $\min(x)=\min(y)$, $\max(x)=\max(y)$ 且 $\text{sum}(x)=\text{sum}(y)$ 时，称 x 和 y 是 \otimes 等价的。由 \otimes 运算的定义知，当 x 和 y 是 \otimes 等价时，对于任意正整数 z 有 $(x \otimes z)=(y \otimes z)$ 且 $(z \otimes x)=(z \otimes y)$ 。由此可见，在 $S(N)$ 中只要关注 \otimes 等价类中 \otimes 运算次数最少的数。

\otimes 等价类可按 \min, \max, sum 进行分类。对于 $S(N)$ 中任一数 x 有：

$$\begin{aligned} 0 \leq \min(x), \max(x) \leq 9 \\ 1 \leq \text{sum}(x) \leq \text{LL} \end{aligned}$$

式中， $\text{LL}=9 \times \lceil \log_{10}(L+1) \rceil$ 。

因此可以用 2 个数组 $\text{leftn}[1;\text{LL}]$ 和 $\text{rightn}[0;9][0;]$ 来存储 $S(N)$ 中 \otimes 等价类，并由此构造出 $S(N)$ 中所有数。

输入 N 和 K 后, 数组 `leftn` 和 `rightn` 由 `init` 初始化如下。

```
void init()
{
    biglen*=9;
    Make2DArray(rightn, 10, 10);
    for (int i=0; i<10; i++)
        for (int j=0; j<10; j++)
            rightn[i][j]= INT_MAX;
    leftn = new int[biglen+1];
    for (i=1; i<=biglen; i++) leftn[i]=INT_MAX;
    int a=INT_MAX, b=0; sum=0; leftn[0]=0;
    for(i=0; i<len; i++) count(ctoi(s1[i]), a, b, sum);
    rightn[a][b]=0;
    if (sum<=biglen) leftn[sum]=0;
}
```

其中, 用函数 `count` 和 `trans` 来计算 $\min(i)$, $\max(i)$ 和 $\text{sum}(i)$ 。

```
void count(int i, int &a, int &b, int &c)
{
    if (i < a)  a = i;
    if (i > b)  b = i;
    c+=i;
}

void trans(int t, int &a, int &b, int &c)
{
    a=INT_MAX; b=0; c=0;
    while(t>0) {
        int j = t%10;
        t/=10;
        count(j, a, b, c);
    }
}
```

用⊗等价类思想实现的动态规划算法描述如下。

```
void compute()
{
    int a, b, c, best=INT_MAX;
    bool flag=true;
    while(flag) {
        flag=false;
```

```

        for (int i=0;i<=biglen;i++)
            if (leftn[i]<INT_MAX)
                for (int j=0;j<10;j++)
                    for (int k=0;k<=j;k++)
                        if (rightn[k][j]<INT_MAX){
                            int num=i>0 ? i*j+k:sum*j+k;
                            trans(num, a, b, c);
                            int curr=leftn[i]+rightn[k][j]+1;
                            if (curr<leftn[c]){leftn[c]=curr;flag=true;}
                            if (curr<rightn[a][b]){rightn[a][b]=curr;flag=true;}
                            if (num==kk && curr<best){best=curr;flag=true;}
                        }
            }
        if (best<INT_MAX) out(best);
        else out(-1);
    }
}

```

完成整个计算的主函数 main 如下。

```

int kk, len, biglen, sum, *leftn, **rightn;
char s1[20], s2[20];
void main()
{
    if(input()) init();
    else return;
    compute();
}

```

(9) 改进算法的计算复杂性。

改进算法所需空间显然为 $O(\log\log N)$ 。

改进算法所需计算时间为核心算法 compute 的计算时间。算法 compute 的 1 次 while 循环需要 $O(\log\log N)$ 的计算时间。在最坏情况下需要 $O(\log N)$ 次 while 循环。因此，算法在最坏情况下所需的计算时间为 $O(\log N \log\log N)$ 。

可以看到，新算法的计算复杂性有十分显著的改进。

算法实现题 3-13 最大长方体问题 (习题 3-21)

★问题描述:

一个长、宽、高分别为 m, n, p 的长方体被分割成 $m \times n \times p$ 个小立方体。每个小立方体内有一个整数。试设计一个算法，计算出所给长方体的最大子长方体。子长方体的大小由它所含所有整数之和确定。

★编程任务:

对于给定的长、宽、高分别为 m, n, p 的长方体，编程计算最大子长方体的大小。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行是 3 个正整数 $m, n, p, 1 \leq m, n, p \leq 50$ 。在接下来 $m \times n$ 行中每行 p 个正整数, 表示小立方体中的数。

★结果输出:

程序运行结束时, 将计算结果输出到文件 output.txt 中。文件的第 1 行中的数是计算出的最大子长方体的大小。

输入文件示例	输出文件示例
input.txt	output.txt
3 3 3	14
0 -1 2	
1 2 2	
1 1 -2	
-2 -1 -1	
-3 3 -2	
-2 -3 1	
-2 3 3	
0 1 3	
2 1 -3	

分析与解答:

在最大子矩阵和问题的动态规划算法基础上, 容易设计解此问题的动态规划算法如下。

```
template <class T>
T maxSum3(const vector< matrix<T> > & a)
{
    T max, sum=0;
    int m=a.size(), n=a[0].rows(), p=a[0].cols();
    matrix<int> b(n, p, 0);
    for (int i=0; i<m; i++) {
        for (int k=0; k<n; k++)
            for (int t=0; t<p; t++) b[k][t]=0;
        for (int j=i; j<m; j++) {
            for (int k=0; k<n; k++)
                for (int t=0; t<p; t++)
                    b[k][t] += a[j][k][t];
            max=maxSum2(b);
            if (max>sum) sum=max;
        }
    }
    return sum;
}
```

算法实现题 3-14 正则表达式匹配问题 (习题 3-22)

★问题描述:

许多操作系统采用正则表达式实现文件匹配功能。一种简单的正则表达式由英文字母、数字及通配符“*”和“?”组成。“?”代表任意一个字符,“*”则可以代表任意多个字符。现要用正则表达式对部分文件进行操作。

试设计一个算法,找出一个正则表达式,使其能匹配的待操作文件最多,但不能匹配任何不进行操作的文件。所找出的正则表达式的长度还应是最短的。

★编程任务:

对于给定的待操作文件,找出一个能匹配最多待操作文件的正则表达式。

★数据输入:

由文件 input.txt 提供输入数据。文件由 n ($1 \leq n \leq 250$) 行组成。每行给出一个文件名。文件名由英文字母和数字组成。英文字符要区分大小写,文件名长度不超过 8 个字符。文件名后是一个空格符和一个字符“+”或“-”。“+”表示要对该行给出的文件进行操作,“-”表示不进行操作。

★结果输出:

程序运行结束时,将计算出的最多文件匹配数和最优正则表达式输出到文件 output.txt 中。文件第 1 行中的数是计算出的最多文件匹配数,第 2 行是最优正则表达式。

输入文件示例	输出文件示例
input.txt	output.txt
EXCHANGE +	3
EXTRA +	* A *
HARDWARE +	
MOUSE -	
NETWORK -	

分析与解答:

设当前考察的正则表达式为 s , 当前期考察的文件为 f 。用 $\text{match}(i, j)$ 表示 $s[1..i]$ 与 $f[1..j]$ 的匹配情况。当 $s[1..i]$ 能匹配 $f[1..j]$ 时, $\text{match}(i, j) = 1$, 否则 $\text{match}(i, j) = 0$ 。显然可用下面的递归式计算 $\text{match}(i, j)$ 。

$$\text{match}(i, j) = \begin{cases} 1 & \begin{cases} \text{match}(i-1, j-1) = 1 & s[i] = '?' \\ \text{match}(i-1, j-1) = 1 & s[i] = f[j] \\ \text{match}(i-1, k) = 1 & s[i] = '*' \end{cases} \\ 0 & \end{cases}$$

据此可设计求最优匹配的回溯法如下。

```
void search(int len)
{
    if((currmat > maxmat || currmat == maxmat && len < minlen) && ok(len)) {
        maxmat = currmat; minmat = s; minlen = len;
    }
    len++;
    if(len == -1 || s[len-1] != '*') {
        s[len] = '?';
```

```

        if(check(len)) search(len);
        s[len]='*';
        if(check(len)) search(len);
    }
    for(int i=1;i<=p[len-1];i++) {
        s[len]=cha[len-1][i].c;
        if(check(len)) search(len);
    }
}

```

其中，check 用于计算当前匹配情况，ok 用于判定是否匹配非操作文件。

```

bool check(int len)
{
    int i, j, t, k=0;
    curmat=0;
    for(i=1;i<=n[0];i++) {
        memset(&match[len][i], 0, sizeof(match[len][i]));
        if(len==1 && s[1]=='*') match[len][i][0]=1;
        for(j=1;j<=f[i].length();j++)
            switch(s[len]) {
                case '*':
                    for(t=0;t<=j;t++)
                        if(match[len-1][i][t]==1) {match[len][i][j]=1;break;}
                    break;
                case '?':
                    match[len][i][j]=match[len-1][i][j-1];
                    break;
                default:
                    if(s[len]==f[i][j-1])
                        match[len][i][j]=match[len-1][i][j-1];
                    break;
            }
        for(j=f[i].length();j>=1;j--)
            if(match[len][i][j]==1) {
                k++;
                if(j==f[i].length())curmat++;
                break;
            }
    }
    if (k<maxmat || k==maxmat && len>=minlen) return 0;
    p[len]=0;
    for(i=1;i<=n[0];i++)
        for(j=1;j<=f[i].length()-1;j++)

```

```

        if (match[len][i][j]==1) save(f[i][j],len);
    return 1;
}

bool ok(int len)
{
    int i, j, k, t;
    for(k=1;k<=len;k++)
        for(i=n[0]-1;i<=n[0]+n[1];i++) {
            memset(&match[k][i], 0, sizeof(match[k][i]));
            if(s[1]=='*' && k==1) match[k][i][0]=1;
            for(j=1;j<=f[i].length();j++)
                switch(s[k]) {
                    case '*':
                        for (t=0;t<=j;t++)
                            if (match[k-1][i][t]==1) {match[k][i][j]=1;break;}
                        break;
                    case '?':
                        match[k][i][j]=match[k-1][i][j-1];
                        break;
                    default:
                        if (s[k]==f[i][j-1])match[k][i][j]=match[k-1][i][j-1];
                        break;
                }
        }
    for(i=n[0]+1;i<=n[0]+n[1];i++)
        if(match[len][i][f[i].length()]==1)return 0;
    return 1;
}

```

算法的主函数如下。

```

int main()
{
    readin();
    search(0);
    out();
    return 0;
}

```

readin 读入数据并初始化。

```

void readin()
{

```

```

string k[MAXN+1], str;
char chr;
n[0]=0;n[1]=0;p[0]=0;
while(1){
    fin>>str>>chr;
    if(str.length()==0)break;
    if (chr=='+') {f[++n[0]]=str;save(str[0],0);}
    else k[++n[1]]=str;
}
for(int i=1;i<=n[1];i++) f[n[0]+i]=k[i];
memset(match,0,sizeof(match));
for(i=1;i<=n[0]+n[1];i++) match[0][i][0]=1;
maxmat=0;minlen=255;
}

```

其中, save 对操作文件名中出现的字符按出现频率排序存储, 以加快搜索进程。

```

void save(char c,int len)
{
    for (int i=1;i<=p[len];i++)
        if(cha[len][i].c==c){
            cha[len][i].f++;
            cha[len][0]=cha[len][i];
            int j=i;
            while(cha[len][j-1].f<cha[len][0].f){
                cha[len][j]=cha[len][j-1];j--;
            }
            cha[len][j]=cha[len][0];
            return;
        }
    cha[len][++p[len]].c=c;cha[len][p[len]].f=1;
}

```

算法实现题 3-15 双调旅行售货员问题 (习题 3-23)

★问题描述:

欧氏旅行售货员问题是对给定的平面上 n 个点确定一条连接这 n 个点的长度最短的哈密顿回路。由于欧氏距离满足三角不等式, 所以欧氏旅行售货员问题是一个特殊的具有三角不等式性质的旅行售货员问题。它仍是一个 NP 完全问题。最短双调 TSP 回路是欧氏旅行售货员问题的特殊情况。平面上 n 个点的双调 TSP 回路是从最左点开始, 严格地由左至右直到最右点, 然后严格地由右至左直至最左点, 且连接每一个点恰好一次的一条闭合回路。

★编程任务:

给定平面上 n 个点, 编程计算这 n 个点的最短双调 TSP 回路。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n , 表示给定的平面上的点数。在接下来的 n 行中, 每行 2 个实数, 分别表示点的 x 坐标和 y 坐标。

★结果输出:

将计算的最短双调 TSP 回路的长度 (保留 2 位小数) 输出到文件 output.txt 中。

输入文件示例	输出文件示例
input.txt	output.txt
7	25.58
0 6	
1 0	
2 3	
5 4	
6 1	
7 5	
8 2	

分析与解答:

首先将给定的平面上 n 个点依其 x 坐标排序。设排好序的 n 个点为 $p_i = (x_i, y_i), i = 1, 2, \dots, n$ 。点集 $\{p_1, p_2, \dots, p_i\}$ 的最短双调 TSP 回路的长度记作 $t(i)$ 。容易证明, 该问题具有最优子结构性质。 $t(i)$ 满足如下动态规划递归式:

$$t(i) = \min_{1 \leq k < i} \{l(k) + D(k, i) + d(k-1, i) - d(k-1, k)\}$$
$$t(1) = 0, t(2) = 2d(1, 2)$$

式中

$$d(i, j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

$$D(i, j) = \sum_{k=i+1}^j d(k-1, k)$$

设 $s(i) = \sum_{k=1}^i d(k-1, k)$, 则

$$D(k, i) = s(i) - s(k), d(k-1, k) = s(k) - s(k-1)$$

上述递归式可进一步简化为

$$t(i) = \min_{1 \leq k < i} \{l(k) + s(i) + s(k-1) - 2s(k) + d(k-1, i)\}$$

按此递归式计算出的 $t(n)$ 为最优值。算法所需的计算时间为 $O(n^2)$ 。

```
void init()
{
    fin >> n;
    point.resize(n);
    for (int i = 0; i < n; i++) {
        point[i].resize(2);
        fin >> point[i][0] >> point[i][1];
    }
}
```

```

        sort(point.begin(), point.end(), vless);
    }

    void dynamic()
    {
        s[1]=0;
        for (int i=2; i<=n; i++) s[i]=dist(i-1, i)+s[i-1];
        t[2]=2*s[2];
        for(i=3; i<=n; i++){
            t[i]=t[2]+s[i]-2*s[2]+dist(i, 1);
            for (int j=2; j<=i-2; j++){
                double temp=t[j+1]+s[i]+s[j]-2*s[j+1]+dist(i, j);
                if(t[i]>temp)t[i]=temp;
            }
        }
    }
}

```

其中，dist 计算两点之间的距离。

```

double dist(int p, int q)
{
    return sqrt(pow(point[p-1][0]-point[q-1][0], 2)+
        pow(point[p-1][1]-point[q-1][1], 2));
}

```

算法实现题 3-16 最大 k 乘积问题 (习题 5-28)

★问题描述:

设 I 是一个 n 位十进制整数。如果将 I 划分为 k 段，则可得到 k 个整数。这 k 个整数的乘积称为 I 的一个 k 乘积。试设计一个算法，对于给定的 I 和 k ，求出 I 的最大 k 乘积。

★编程任务:

对于给定的 I 和 k ，编程计算 I 的最大 k 乘积。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行中有 2 个正整数 n 和 k 。正整数 n 是序列的长度，正整数 k 是分割的段数。在接下来的一行中是一个 n 位十进制整数 ($n \leq 10$)。

★结果输出:

程序运行结束时，将计算结果输出到文件 output.txt 中。文件第 1 行中的数是计算出的最大 k 乘积。

输入文件示例

input.txt

2 1

15

输出文件示例

output.txt

15

分析与解答:

设 $I(s, t)$ 是 I 的由从 s 位开始的 t 位数字组成的十进制数。 $f(i, j)$ 表示 $I(0, i)$ 的最大 j 乘积, 则 $f(i, j)$ 具有最优子结构性质。计算 $f(i, j)$ 的动态规划递归式如下:

$$f(i, j) = \max_{1 \leq k < i} \{f(k, j-1) * I(k, i-k)\}$$

据此可设计最大 k 乘积问题的动态规划算法如下。

```
void solve(int n, int m)
{
    int i, j, k;
    int temp, maxt, tk;
    for (i = 1; i <= n; i++) f[i][1] = conv(0, i);
    for (j = 2; j <= m; j++)
        for (i = j; i <= n; i++) {
            for (k = 1, temp = 0; k < i; k++) {
                maxt = f[k][j-1] * conv(k, i-k);
                if (temp < maxt) {temp = maxt; tk = k;}
            }
            f[i][j] = temp; ka[i][j] = tk;
        }
}
```

其中, conv 计算 $I(s, t)$ 的值。

```
int conv(int i, int j)
{
    string str1 = str.substr(i, j);
    return atoi(str1.c_str());
}
```

实现算法的主函数如下。

```
int main()
{
    int n, m;
    ifstream cin1("kmul.in");
    cin1 >> n >> m;
    if ((n < m) || (n == 0)) {cout << 0 << endl; return;}
    cin1 >> str;
    if (n != str.size()) {cout << 0 << endl; return;}
    f.resize(n+1, m+1);
    ka.resize(n+1, m+1);
    solve(n, m);
    out(n, m);
    return 0;
}
```

```
}
```

out 输出计算结果。

```
void out(int n, int m)
{
    ofstream cout("kmul.out");
    cout<<f[n][m]<<endl;
    for(int i=m, k=n; i>=1 && k>0; k=ka[k][i], i--)
        cout<<"f["<<k<<"]["<<i<<"]="<<f[k][i]<<endl;
}
```

算法实现题 3-17 最少费用购物问题 (习题 3-20)

★问题描述:

商店中每种商品都有标价。例如,一朵花的价格是 2 元,一个花瓶的价格是 5 元。为了吸引顾客,商店提供了一组优惠商品价。优惠商品是把一种或多种商品分成一组,并降价销售。例如,3 朵花的价格不是 6 元而是 5 元。2 个花瓶加 1 朵花的优惠价是 10 元。试设计一个算法,计算出某一顾客所购商品应付的最少费用。

★编程任务:

对于给定欲购商品的价格和数量,以及优惠商品价,编程计算所购商品应付的最少费用。

★数据输入:

由文件 input.txt 提供欲购商品数据。文件的第 1 行中有 1 个整数 $B(0 \leq B \leq 5)$, 表示所购商品种类数。在接下来的 B 行中,每行有 3 个数 C, K 和 P 。 C 表示商品的编码(每种商品有惟一编码), $1 \leq C \leq 999$; K 表示购买该种商品总数, $1 \leq K \leq 5$; P 是该种商品的正常单价(每件商品的价格), $1 \leq P \leq 999$ 。请注意,一次最多可购买 $5 \times 5 = 25$ 件商品。

由文件 offer.txt 提供优惠商品价数据。文件的第 1 行中有 1 个整数 $S(0 \leq S \leq 99)$, 表示共有 S 种优惠商品组合。接下来的 S 行,每行的第 1 个数描述优惠商品组合中商品的种类数 j 。接着是 j 个数字对 (C, K) , 其中 C 是商品编码, $1 \leq C \leq 999$; K 表示该种商品在此组合中的数量, $1 \leq K \leq 5$ 。每行最后一个数字 $P(1 \leq P \leq 9999)$ 表示此商品组合的优惠价。

★结果输出:

程序运行结束时,将计算出的所购商品应付的最少费用输出到文件 output.txt 中。

输入文件示例

input.txt

2

7 3 2

8 2 5

输出文件示例

output.txt

14

1 7 3 5

2 7 1 8 2 10

分析与解答:

设 $\text{cost}(a, b, c, d, e)$ 表示购买商品组合 (a, b, c, d, e) 需要的最少费用。 $A[k], B[k], C[k], D[k], E[k]$ 表示第 k 种优惠方案的商品组合。 $\text{offer}(m)$ 是第 m 种优惠方案的价格。如果

$\text{cost}(a,b,c,d,e)$ 使用了第 m 种优惠方案, 则

$$\begin{aligned}\text{cost}(a,b,c,d,e) = & \text{cost}(a-A[m], b-B[m], \\ & c-C[m], d-D[m], e-E[m]) + \text{offer}(m)\end{aligned}$$

即该问题具有最优子结构性质。按此递归式, 容易设计解此问题的动态规划算法如下。

```
void minicost()
{
    int i, j, k, m, n, p, minm;
    minm=0;
    for(i=1; i<=b; i++)
        minm = product[i]*purch[i].price;
    for(p=1; p<=s; p++) {
        i=product[1] - offer[p][1];
        j=product[2] - offer[p][2];
        k=product[3] - offer[p][3];
        m=product[4] - offer[p][4];
        n=product[5] - offer[p][5];
        if(i>=0 && j>=0 && k>=0 && m>=0 && n>=0 &&
            (cost[i][j][k][m][n]+offer[p][0]<minm))
            minm=cost[i][j][k][m][n]+offer[p][0];
    }
    cost[product[1]][product[2]][product[3]][product[4]][product[5]] = minm;
}
```

其中, $\text{product}[i]$ 是购买第 i 种商品的数量, 由 comp 迭代计算, init 进行初始化计算。

```
void comp(int i)
{
    if(i>b) {minicost(); return;}
    for(int j=0; j<=purch[i].piece; j++) {product[i]=j; comp(i+1);}
}

void init()
{
    int i, j, n, p, t, code;
    for(i=0; i<100; i++)
        for(j=0; j<6; j++) offer[i][j]=0;
    for(i=0; i<6; i++) {purch[i].piece=0; purch[i].price=0; product[i]=0;}
    fin>>b;
    for(i=1; i<=b; i++) {
        fin>>code;
        fin>>purch[i].piece>>purch[i].price;
        num[code]=i;
    }
}
```

```

    finl>>s;
    for(i=1;i<=s;i++){
        finl>>t;
        for(j=1;j<=t;j++){
            finl>>n>>p;
            offer[i][num[n]]=p;
        }
        finl>>offer[i][0];
    }
}

```

实现算法的主函数如下。

```

int main()
{
    init();
    comp(1);
    out();
    return 0;
}

```

算法实现题 3-18 收集样本问题

★问题描述:

机器人 Rob 在一个有 $n \times n$ 个方格的方形区域 F 中收集样本。 (i, j) 方格中样本的价值为 $v(i, j)$ ，如图 3-3 所示。Rob 从方形区域 F 的左上角 A 点出发，向下或向右行走，直到右下角的 B 点，在走过的路上，收集方格中的样本。Rob 从 A 点到 B 点共走 2 次，试找出 Rob 的 2 条行走路径，使其取得的样本总价值最大。

A							
		13			6		
				7			
			14				
	21				4		
		15					
	14						
							B

图 3-3 $n \times n$ 个方格的方形区域 F

★编程任务:

给定方形区域 F 中的样本分布，编程计算 Rob 的 2 条行走路径，使其取得的样本总价值最大。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n ，表示方形区域 F 有 $n \times n$ 个方格。接下来每行有 3 个整数，前 2 个数表示方格位置，第 3 个数为该位置样本价值。最后一行是 3 个 0。

★结果输出:

将计算的最大样本总价值输出到文件 output.txt 中。

输入文件示例

input.txt

8

2 3 13

输出文件示例

output.txt

67

```

2 6 6
3 5 7
4 4 14
5 2 21
5 6 4
6 3 15
7 2 14
0 0 0

```

分析与解答:

设两次行走等长距离, 到达 (x_1, y_1) 和 (x_2, y_2) 处取得的最大价值为 $h[x_1][y_1][x_2][y_2]$ 。容易证明其具有最优子结构性质。动态规划算法如下。

```

void dyna()
{
    int x1, y1, x2, y2, s, v;
    for(int i=0; i<=n; i++)
        for(int j=0; j<=n; j++)
            for(int k=0; k<=n; k++)
                for(int l=0; l<=n; l++) h[i][j][k][l]=0;
    h[1][1][1][1]=g[1][1];
    for(s=2; s<=n+n-1; s++) {
        for(x1=1; x1<=s-1; x1++)
            for(x2=1; x2<=s-1; x2++) {
                y1=s-x1; y2=s-x2;
                v=h[x1][y1][x2][y2];
                val(x1+1, y1, x2+1, y2, v);
                val(x1+1, y1, x2, y2+1, v);
                val(x1, y1+1, x2+1, y2, v);
                val(x1, y1+1, x2, y2+1, v);
            }
    }
}

```

其中, val 动态更新最优值。

```

void val(int x1, int y1, int x2, int y2, int v)
{
    if(x1==x2 && y1==y2)
        h[x1][y1][x2][y2]=max(h[x1][y1][x2][y2], v+g[x1][y1]);
    else
        h[x1][y1][x2][y2]=max(h[x1][y1][x2][y2], v+g[x1][y1]+g[x2][y2]);
}

```

$g[i][j]$ 是方格 (i,j) 处样本的价值。

算法实现题 3-19 最优时间表问题

★问题描述:

一台精密仪器的工作时间为 n 个时间单位。与仪器工作时间同步进行若干仪器维修程序。一旦启动维修程序,仪器必须进入维修程序。如果只有一个维修程序启动,则必须进入该维修程序。如果在同一时刻有多个维修程序,可任选进入其中的一个维修程序。维修程序必须从头开始,不能从中间插入。一个维修程序从第 s 个时间单位开始,持续 t 个时间单位,则该维修程序在第 $s+t-1$ 个时间单位结束。为了提高仪器使用率,希望安排尽可能短的维修时间。

★编程任务:

对于给定的维修程序时间表,编程计算最优时间表。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 k 。 n 表示仪器的工作时间单位, k 是维修程序数。在接下来的 k 行中,每行有 2 个表示维修程序的整数 s 和 t ,该维修程序从第 s 个时间单位开始,持续 t 个时间单位。

★结果输出:

将计算出的最短维修时间输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
15 6	11
1 2	
1 6	
4 11	
8 5	
8 1	
11 5	

分析与解答:

设 $mt(i)$ 表示从第 i 个时间单位开始到第 n 个时间单位结束的最短维修时间,则 $mt(i)$ 具有最优子结构性质,且满足如下递归式:

当时间单位 i 有多个维修程序时, $mt(i) = \min_{s_j=i} \{mt(i+t_j)\}$;

当时间单位 i 没有维修程序时, $mt(i) = mt(i+1) - 1$ 。初始值为 $mt(n+1) = n$ 。

算法实现题 3-20 字符串比较问题

★问题描述:

对于长度相同的 2 个字符串 A 和 B ,其距离定义为相应位置字符距离之和。2 个非空格字符的距离是它们的 ASCII 码之差的绝对值。空格与空格的距离为 0,空格与其他字符的距离为一定值 k 。

在一般情况下,字符串 A 和 B 的长度不一定相同。字符串 A 的扩展是在 A 中插入若干空格字符所产生的字符串。在字符串 A 和 B 的所有长度相同的扩展中,有一对距离最小的

扩展, 该距离称为字符串 A 和 B 的扩展距离。

对于给定的字符串 A 和 B , 试设计一个算法, 计算其扩展距离。

★编程任务:

对于给定的字符串 A 和 B , 编程计算其扩展距离。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行是字符串 A , 第 2 行是字符串 B , 第 3 行是空格与其他字符的距离定值 k 。

★结果输出:

将计算出的字符串 A 和 B 的扩展距离输出到文件 output.txt。

输入文件示例

输出文件示例

input.txt

output.txt

cmc

10

snmn

2

分析与解答:

设字符串 A 和 B 的子串 $A[1..i]$ 和 $B[1..j]$ 的扩展距离为 $val(i, j)$, 则 $val(i, j)$ 具有最优子结构性质, 且满足如下递归式:

$$val(i, j) = \min\{val(i-1, j) + k, val(i, j-1) + k, \\ val(i-1, j-1) + dist(a_i, b_j)\}$$

据此容易设计动态规划算法如下。

```
int comp()
{
    int i, j, tmp, len1, len2;
    val[0][0]=0; len1=strlen(str1); len2=strlen(str2);
    for(i=0; i<=len1; i++)
        for(j=0; j<=len2; j++)
            if(i+j){
                val[i][j]=MAXINT;
                if((i*j) &&
                    (tmp=val[i-1][j-1]+dist(str1[i-1], str2[j-1]))<val[i][j])
                    val[i][j]=tmp;
                if(i && (tmp=val[i-1][j]+dist(str1[i-1], ' '))<val[i][j])
                    val[i][j]=tmp;
                if(j && (tmp=val[i][j-1]+dist(str2[j-1], ' '))<val[i][j])
                    val[i][j]=tmp;
            }
    return val[len1][len2];
}

int main()
{
```

```

    readin();
    cout<<comp()<<endl;
    return 0;
}

```

从动态规划递归式可知, 算法的计算时间为 $O(mn)$, m 和 n 分别是字符串 A 和 B 的长度。

算法实现题 3-21 有向树 k 中值问题

★问题描述:

给定一棵有向树 T , 树 T 中每个顶点 u 都有一个权 $w(u)$, 树的每条边 (u, v) 也都有一个非负边长 $d(u, v)$ 。有向树 T 的每个顶点 u 可以看作客户, 其服务需求量为 $w(u)$ 。每条边 (u, v) 的边长 $d(u, v)$ 可以看作运输费用。如果在顶点 u 处未设置服务机构, 则将顶点 u 处的服务需求沿有向树的边 (u, v) 转移到顶点 v 处服务机构需付出的服务转移费用为 $w(u) \cdot d(u, v)$ 。树根处已设置了服务机构, 现在要在树 T 中增设 k 处服务机构, 使得整棵树 T 的服务转移费用最小。

★编程任务:

对于给定的有向树 T , 编程计算在树 T 中增设 k 处服务机构的最小服务转移费用。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 k 。 n 表示有向树 T 的边数, k 是要增设的服务机构数。有向树 T 的顶点编号为 $0, 1, \dots, n$ 。根结点编号为 0。在接下来的 n 行中, 每行有表示有向树 T 的一条有向边的 3 个整数。第 $i+1$ 行的 3 个整数 w_i, v_i, d_i 分别表示编号为 i 的顶点的权为 w_i , 相应的有向边为 (i, v_i) , 其边长为 d_i 。

★结果输出:

将计算的最小服务转移费用输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
4 2	4
1 0 1	
1 1 10	
10 2 5	
1 2 3	

分析与解答:

实际上要求有向树 T 的 k 个顶点组成的集合 F , 使 $\text{cost}(F) = \sum_{x \in T} \min_{u \in F+r} w(x) \cdot d(x, u)$ 达到最小。在一般情况下, 有向树 T 是一棵多叉树。为便于计算, 可以将 T 变换为与之等价的如下二叉树。将每个顶点的第一个儿子顶点作为其父顶点的左儿子顶点, 同时增加一个 0 权 0 边长的附加顶点作为右儿子顶点。然后对于其他儿子顶点以相同方式作为新增附加顶点的左儿子顶点, 一直继续下去, 直至处理完所有顶点。所得到的二叉树与树 T 具有相同的最小耗费。在下面的讨论中, 假设 T 是二叉树。

设 T 的以顶点 x 为根的子树为 T_x , 其左、右儿子顶点分别为 y 和 z 。 F 中距顶点 x 最

近的顶点为 u 。在以顶点 x 为根的子树 T_x 中取 j 个 F 中顶点的最小耗费记为 $\text{cost}(x, u, j)$ 。

这个问题是树型动态规划问题的典型例子。容易证明, $\text{cost}(x, u, j)$ 具有最优子结构性, 它满足如下递归式:

$$\text{cost}(x, u, j) = \min \begin{cases} \min_{p, q \in F} \{ \text{cost}(y, u, p) + \text{cost}(z, u, q) \} + w(x) \cdot d(x, u) \\ \text{cost}(x, x, j-1) \end{cases}$$

式中, $d(x, u)$ 是从顶点 x 到顶点 u 的有向路径的长度。当 x 是叶结点时, $\text{cost}(x, u, j)$ 满足:

$$\text{cost}(x, u, j) = \begin{cases} w(x) \cdot d(x, u) & j=0 \\ 0 & j>0 \end{cases}$$

据此可设计动态规划算法如下。树型动态规划问题通常采用自底向上的方式计算, 与树的后序遍历方式相同。可以将阶段性计算结果记录在树结点处。

二叉树的结点结构如下。

```
struct binode{
    int w, d, dep;
    int **cost, *dis;
    link parent, left, right;
};
```

其中, w, d, dep 分别记录该结点的权值、边长和深度。在结点 x 处, $\text{cost}(x, u, j)$ 的值记录在 $\text{cost}[j][u]$ 中。 $\text{dis}[u]$ 记录 $d(x, u)$ 的值。

```
typedef struct binode *link;
link *subt;
```

link 是指向二叉树的结点的指针。 subt 是指向二叉树的结点的指针数组, 用于构造与给定的树 T 等价的二叉树。读入初始数据并构造二叉树的算法如下。

```
void init()
{
    int a, b, c;
    cin >> n >> k;
    subt = new link[n+1];
    for(int i=0; i<=n; i++) subt[i] = newnode();
    for(i=1; i<=n; i++){
        cin >> a >> b >> c;
        subt[i] -> w = a; subt[i] -> d = c;
        link p = findc(subt[b]);
        link q = newnode();
        p -> left = subt[i]; p -> right = q;
        subt[i] -> parent = p; q -> parent = p;
    }
    sd(subt[0]);
}
```

其中, `subt[0]`是指向二叉树根结点的指针。算法 `sd` 以前序遍历的方式计算各结点的深度及 `dist` 的值。

```
void sd(link t)
{
    PreOrder(adddep, t);
}
void PreOrder(void(*Visit)(link u), link t)
{
    if (t) {
        Visit(t);
        PreOrder(Visit, t->left);
        PreOrder(Visit, t->right);
    }
}
void adddep(link t)
{
    if(t->parent) t->dep=t->parent->dep+1;
    Make2DArray(t->cost, k+1, t->dep+1);
    t->dis=new int[t->dep+1];
    t->dis[0]=0;
    if(t->parent)
        for(int i=1; i<=t->dep; i++) t->dis[i]=t->parent->dis[i-1]+t->d;
}
```

`newnode` 创建一个新的树结点。

```
link newnode()
{
    link p=new binode;
    p->parent=0; p->left=0; p->right=0;
    p->w=0; p->d=0; p->dep=0;
    return p;
}
```

`findc` 找出新结点的插入位置。

```
link findc(link q)
{
    while(q && q->right) q=q->right;
    return q;
}
```

在每个结点处的动态规划计算由 comp 完成。

```
void comp(link t)
{
    if(!t->left){
        for(int j=0;j<=t->dep;j++){t->cost[0][j]=t->w*t->dis[j];
        for(int i=1;i<=k;i++){
            for(int j=0;j<=t->dep;j++){t->cost[i][j]=0;
        }
    }
    else
        for(int i=0;i<=k;i++){
            for(int j=0;j<=t->dep;j++){
                if(i)t->cost[i][j]=t->cost[i-1][0];
                else t->cost[i][j]=INT_MAX;
                for(int a=0;a<=i;a++){
                    int tmp=t->left->cost[a][j+1]+t->right->cost[j-a][j+1]+
                        t->w*t->dis[j];
                    if(t->cost[i][j]>tmp)t->cost[i][j]=tmp;
                }
            }
        }
}
```

solution 用后序遍历的方式完成整棵树的动态规划计算。

```
int solution()
{
    PostOrder(comp, subt[0]);
    return subt[0]->cost[k][0];
}

void PostOrder(void(*Visit)(link u), link t)
{
    if(t){
        PostOrder(Visit, t->left);
        PostOrder(Visit, t->right);
        Visit(t);
    }
}
```

最后整个算法由主函数完成。

```
int main()
{
    init();
    cout<<solution()<<endl;
```

```

    return 0;
}

```

从动态规划递归式可知, 算法的计算时间为 $O(k^2 n^2)$ 。

算法实现题 3-22 有向树独立 k 中值问题

★问题描述:

给定一棵有向树 T , 树 T 中每个顶点 u 都有一个权 $w(u)$; 树的每条边 (u, v) 也都有一个非负边长 $d(u, v)$ 。有向树 T 的每个顶点 u 可以看作客户, 其服务需求量为 $w(u)$ 。每条边 (u, v) 的边长 $d(u, v)$ 可以看作运输费用。如果在顶点 u 处未设置服务机构, 则将顶点 u 处的服务需求沿有向树的边 (u, v) 转移到顶点 v 处服务机构需付出的服务转移费用为 $w(u) \cdot d(u, v)$ 。树根处已设置了服务机构, 现在要在树 T 中增设 k 处独立服务机构, 使得整棵树 T 的服务转移费用最小。服务机构的独立性是指任何 2 个服务机构之间都不存在有向路径。

★编程任务:

对于给定的有向树 T , 编程计算在树 T 中增设 k 处独立服务机构的最小服务转移费用。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 k 。 n 表示有向树 T 的边数; k 是要增设的服务机构数。有向树 T 的顶点编号为 $0, 1, \dots, n$ 。根结点编号为 0。接下来的 n 行中, 每行有表示有向树 T 的一条有向边的 3 个整数。第 $i+1$ 行的 3 个整数 w_i, v_i, d_i 分别表示编号为 i 的顶点的权为 w_i , 相应的有向边为 (i, v_i) , 其边长为 d_i 。

★结果输出:

将计算的最小服务转移费用输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
4 2	12
1 0 1	
1 1 10	
10 2 5	
1 2 3	

分析与解答:

要求有向树 T 的 k 个独立顶点组成的集合 F , 使 $\text{cost}(F) = \sum_{x \in T} \min_{u \in F+r} w(x) \cdot d(x, u)$ 达到最小。与有向树 k 中值问题类似。首先将有向树 T 变换为与之等价的二叉树。设 T 的以顶点 x 为根的子树为 T_x , 其左、右儿子顶点分别为 y 和 z 。在以顶点 x 为根的子树 T_x 中取 j 个 F 中顶点的最小耗费记为 $\text{cost}(x, j)$, 它具有最优子结构性质, 满足如下递归式:

$$\text{cost}(x, j) = \min \begin{cases} \min_{p+q=j} \{ \text{cost}(y, p) + \text{cost}(z, q) \} + w(x) \cdot d(x) \\ \sum_{t \in T_x} w(t) \cdot d(t, x) \end{cases}$$

式中, $d(t, x)$ 是从顶点 t 到顶点 x 的有向路径的长度; $d(x)$ 是从顶点 x 到根结点的有向路径的长度。当 x 是叶结点时 $\text{cost}(x, j)$ 满足:

$$\text{cost}(x, j) = \begin{cases} w(x) \cdot d(x) & j=0 \\ 0 & j>0 \end{cases}$$

据此可设计动态规划算法如下。与树的后序遍历方式相同,采用自底向上的方式计算,将阶段性计算结果记录在树结点处。

二叉树的结点结构如下。

```
struct binode{
    int w,wx,d,wd,dep;
    int *cost;
    link parent,left,right;
};
```

其中,wx,d,dep 分别记录该结点的权值、边长和深度。在结点 x 处 $\text{cost}(x,j)$ 的值记录在 $\text{cost}[j]$ 中。

```
typedef struct binode *link;
link *subt;
```

link 是指向二叉树的结点的指针。subt 是指向二叉树的结点的指针数组,用于构造与给定的树 T 等价的二叉树。读入初始数据并构造二叉树的算法如下。

```
void init()
{
    int a,b,c;
    cin>>n>>k;
    subt=new link[n+1];
    for(int i=0;i<=n;i++) subt[i]=newnode();
    for(i=1;i<=n;i++){
        cin>>a>>b>>c;
        subt[i]->w=a;subt[i]->wx=a;subt[i]->d=c;
        link p=findc(subt[b]);
        link q=newnode();
        p->left=subt[i];p->right=q;
        subt[i]->parent=p;q->parent=p;
    }
    sd(subt[0]);
    sw(subt[0]);
}
```

其中,subt[0]是指向二叉树根结点的指针。算法 sd 用前序遍历的方式计算各结点的深度。

```
void sd(link t)
{
    PreOrder(adddep,t);
}
```

```

void PreOrder(void(*Visit)(link u), link t)
{
    if (t){
        Visit(t);
        PreOrder(Visit, t->left);
        PreOrder(Visit, t->right);
    }
}

void adddep(link t)
{
    if(t->parent) t->dep=t->parent->dep+1;
    t->cost=new int[k+1];
    if(t->parent) t->d+=t->parent->d;
}

```

算法 sd 用后序遍历的方式计算各结点的 wd 的值。

```

void sw(link t)
{
    PostOrder(addw, t);
}

void addw(link t)
{
    t->wd=t->w*t->d;
    if(t->left){
        t->w+=t->left->w; t->w+=t->right->w;
        t->wd+=t->left->wd; t->wd+=t->right->wd;
    }
}

```

newnode 创建一个新的树结点。findc 找出新结点的插入位置。
在每个结点处动态规划计算由 comp 完成。

```

void comp(link t)
{
    if(!t->left){
        t->cost[0]=t->wd;
        for(int i=1; i<=k; i++) t->cost[i]=0;
    }
    else
        for(int i=0; i<=k; i++){
            t->cost[i]=t->wd-t->w*t->d;
            for(int a=0; a<=i; a++){
                int tmp=t->left->cost[a]+t->right->cost[i-a]+t->w*t->d;

```



```

        if(t->cost[i]>tmp) t->cost[i]=tmp;
    }
}
}

```

solution 用后序遍历的方式完成整棵树的动态规划计算。

```

int solution()
{
    PostOrder(comp, subt[0]);
    return subt[0]->cost[k];
}

void PostOrder(void(*Visit)(link u), link t)
{
    if(t){
        PostOrder(Visit, t->left);
        PostOrder(Visit, t->right);
        Visit(t);
    }
}

```

最后整个算法由主函数完成。

```

int main()
{
    init();
    cout<<solution()<<endl;
    return 0;
}

```

从动态规划递归式可知，算法的计算时间为 $O(k^2n)$ 。

上述问题可以变换为与之等价的另一个树型动态规划问题。

设 $gain(F) = \sum_{x \in T} w(x) \cdot d(x) - cost(F)$ ，则 $cost(F)$ 的最小值对应于 $gain(F)$ 的最大值。设 T 的以顶点 x 为根的子树为 T_x ，其左、右儿子顶点分别为 y 和 z 。在以顶点 x 为根的子树 T_x 中取 j 个 F 中顶点的最大值记为 $gain(x, j)$ ，它具有最优子结构性质，满足如下递归式：

$$gain(x, j) = \max \begin{cases} \max_{p+q=j} \{ gain(y, p) + gain(z, q) \} \\ \left(\sum_{t \in T_x} w(t) \right) \cdot d(x) \end{cases}$$

式中， $d(x)$ 是从顶点 x 到根结点的有向路径的长度。当 x 是叶结点时 $gain(x, j)$ 满足：

$$gain(x, j) = \begin{cases} 0 & j=0 \\ w(x) \cdot d(x) & j>0 \end{cases}$$

据此可设计动态规划算法如下。与树的后序遍历方式相同,采用自底向上的方式计算,将阶段性计算结果记录在树结点处。

在每个结点处动态规划计算由 comp 完成。

```
void comp(link t)
{
    if(!t->left){
        t->gain[0]=0;
        for(int i=1;i<=k;i++) t->gain[i]=t->w*t->d;
    }
    else
        for(int i=0;i<=k;i++){
            t->gain[i]=t->w*t->d;
            for(int a=0;a<=i;a++){
                int tmp=t->left->gain[a]+t->right->gain[i-a];
                if(t->gain[i]<tmp)t->gain[i]=tmp;
            }
        }
}
```

solution 用后序遍历的方式完成整棵树的动态规划计算。

```
int solution()
{
    PostOrder(comp, subt[0]);
    return tot - subt[0]->gain[k];
}
```

从动态规划递归式可知,算法的计算时间为 $O(k^2n)$ 。

算法实现题 3-23 有向直线 m 中值问题

★问题描述:

给定一条有向直线 L 及 L 上的 $n+1$ 个点 $x_0 < x_1 < \dots < x_n$ 。有向直线 L 上的每个点 x_i 都有一个权 $w(x_i)$; 每条有向边 (x_i, x_{i+1}) 也都有一个非负边长 $d(x_i, x_{i+1})$ 。有向直线 L 上的每个点 x_i 可以看作客户,其服务需求量为 $w(x_i)$ 。每条边 (x_i, x_{i+1}) 的边长 $d(x_i, x_{i+1})$ 可以看作运输费用。如果在点 x_i 处未设置服务机构,则将点 x_i 处的服务需求沿有向边转移到点 x_j 处服务机构需付出的服务转移费用为 $w(x_i) \cdot d(x_i, x_j)$ 。在点 x_0 处已设置了服务机构,现在要在直线 L 上增设 m 处服务机构,使得整体服务转移费用最小。

★编程任务:

对于给定的有向直线 L ,编程计算在直线 L 上增设 m 处服务机构的最小服务转移费用。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n , 表示有向直线 L 上除了点 x_0

外还有 n 个点 $x_0 < x_1 < \dots < x_n$ 。接下来的 n 行中，每行有 2 个整数。第 $i+1$ 行的 2 个整数分别表示 $w(x_{n-i-1})$ 和 $d(x_{n-i-1}, x_{n-i-2})$ 。

★结果输出：

将计算的最小服务转移费用输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
9 2	26
1 2	
2 1	
3 3	
1 1	
3 2	
1 6	
2 1	
1 2	
1 1	

分析与解答：

此题是有向树 k 中值问题在树 T 退化为有向直线的特殊情形，可以用算法实现题 3-21 的算法求解，或直接求解。

设 $\text{cost}(i, j)$ 是在前 i 个点中设置 j 个服务机构的最小费用，则 $\text{cost}(i, j)$ 具有最优子结构性质，满足如下递归式：

$$\text{cost}(i, j) = \min_{1 \leq k < i} \{ \text{cost}(k-1, j-1) + \sum_{t=k-1}^i w(t) \cdot d(t, k) \}$$

式中， $d(t, k)$ 是从点 x_t 到点 x_k 的有向路径的长度。当 $j \geq i$ 时，显然有 $\text{cost}(i, j) = 0$ 。

为了在 $O(1)$ 时间内计算 $\sum_{t=k-1}^i w(t) \cdot d(t, k)$ ，先用 $O(n)$ 时间预先计算

$$\begin{aligned} w(1, i) &= \sum_{t=1}^i w(t) \\ d(1, i) &= \sum_{t=1}^i d(x_t, x_{t-1}) \\ wd(1, i) &= \sum_{t=1}^i w(1, t) \cdot d(1, t) \end{aligned}$$

readin 读入初始数据并进行预处理。

```

void readin()
{
    int d, w;
    dist[1]=0; wt[0]=0; swt[1]=0;
    cin >> n >> m;
    cin >> wt[1] >> dist[2];
    for (int i=2; i<=n; i++) {
        cin >> w >> d;
    }
}

```

```

        wt[i]=wt[i-1]+w;
        dist[i+1]=dist[i]+d;
        swt[i]=swt[i-1]+w*dist[i];
    }
}

```

$\sum_{t=i}^j w(t) \cdot d(t,i)$ 可在 $O(1)$ 时间内由 `getw` 计算如下。

```

int getw(int i, int j)
{
    if(i>j)return 0;
    else return (wt[j-1]-wt[i])*dist[j]-(swt[j-1]-swt[i]);
}

```

据此可设计动态规划算法如下。

```

void comp()
{
    int i, j, k, tmp;
    for(i=1; i<=n; i++)minco[i][0]=getw(0, i+1);
    for(i=1; i<=n; i++)
        for(j=1; j<=m; j++){
            if(j>=i)minco[i][j]=0;
            else{
                minco[i][j]=getw(1, i+1);
                for(k=2; k<=i; k++){
                    tmp=minco[k-1][j-1]+getw(k, i+1);
                    if(tmp<minco[i][j])minco[i][j]=tmp;
                }
            }
        }
}

```

`minco[n][m]`给出所求最优值。

```

int main()
{
    readin();
    comp();
    cout<<minco[n][m]<<endl;
    return 0;
}

```

算法需要的计算时间为 $O(mn^2)$ ，占用空间 $O(mn)$ 。

注意到算法计算过程中只用到 minco 的一列数据，可以进一步将空间需求减少到 $O(n)$ 。

```
void comp()
{
    int i, j, k, tmp;
    for(i = 1; i <= n; i++) minco[i] = getw(0, i + 1);
    for(j = 1; j <= m; j++) {
        for(i = n; i > j; i--) {
            minco[i] = getw(1, i + 1);
            for(k = 2; k <= i; k++) {
                tmp = minco[k + 1] + getw(k, i + 1);
                if(tmp < minco[i]) minco[i] = tmp;
            }
        }
        for(i = 1; i <= j; i++) minco[i] = 0;
    }
}
```

minco[n] 给出所求最优值。

```
int main()
{
    readin();
    comp();
    cout << minco[n] << endl;
    return 0;
}
```

算法实现题 3-24 有向直线 2 中值问题

★问题描述：

给定一条有向直线 L 及 L 上的 $n+1$ 个点 $x_0 < x_1 < \cdots < x_n$ 。有向直线 L 上的每个点 x_i 都有一个权 $w(x_i)$ ；每条有向边 (x_i, x_{i+1}) 也都有一个非负边长 $d(x_i, x_{i+1})$ 。有向直线 L 上的每个点 x_i 可以看作客户，其服务需求量为 $w(x_i)$ 。每条边 (x_i, x_{i+1}) 的边长 $d(x_i, x_{i+1})$ 可以看作运输费用。如果在点 x_i 处未设置服务机构，则将点 x_i 处的服务需求沿有向边转移到点 x_j 处服务机构需付出的服务转移费用为 $w(x_i) \cdot d(x_i, x_j)$ 。在点 x_0 处已设置了服务机构，现在要在直线 L 上增设 2 处服务机构，使得整体服务转移费用最小。

★编程任务：

对于给定的有向直线 L ，编程计算在直线 L 上增设 2 处服务机构的最小服务转移费用。

★数据输入：

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n ，表示有向直线 L 上除了点 x_0 外还有 n 个点 $x_0 < x_1 < \cdots < x_n$ 。接下来的 n 行中，每行有 2 个整数。第 $i+1$ 行的 2 个整数

分别表示 $w(x_{n-i-1})$ 和 $d(x_{n-i-1}, x_{n-i-2})$ 。

★结果输出:

将计算的最小服务转移费用输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
9	26
1 2	
2 1	
3 3	
1 1	
3 2	
1 6	
2 1	
1 2	
1 1	

分析与解答:

此题是有向直线 m 中值问题当 $m=2$ 的特殊情形, 可以用算法实现题 3-23 的算法求解, 可设计效率更高的算法。

设 $\text{cost}(i, j)$ 是在 x_i 和 x_j 处 ($i < j$) 设置服务机构的费用, 则问题是求 $i < j$ 的最优位置, 使 $\text{cost}(i, j)$ 达到最小。

假设在位置 i 已取定, j 的最优位置为 $\text{opt}(i)$, 则 $\text{opt}(i)$ 具有如下性质: 对任何 $i < j$, 总有 $\text{opt}(i) < \text{opt}(j)$ 。

利用此性质可设计如下分治算法。

首先计算 $a = \text{opt}(n/2)$ 。由上述性质知, 当 $i < n/2$ 时, $\text{opt}(i) < a$; 当 $i > n/2$ 时, $\text{opt}(i) > a$ 。

据此设计分治算法 $\text{comp}(\text{minp1}, \text{maxp1}, \text{minp2}, \text{maxp2})$, 找出

$$x_i \in \{x_{\min p1}, \dots, x_{\max p1}\}, x_j \in \{x_{\min p2}, \dots, x_{\max p2}\}$$

使 $\text{cost}(i, j)$ 达到最小。

readin 读入初始数据并进行预处理计算。

```
void readin()
{
    int d, w;
    cin >> n;
    dist[1] = 0;
    cin >> wt[1] >> dist[2];
    tot = wt[1] * dist[2];
    for (int i = 2; i <= n; i++) {
        cin >> w >> d;
        wt[i] = wt[i - 1] + w;
        dist[i + 1] = dist[i] + d;
```

```

        tot=tot+wt[i]*d;
    }
}

```

getcost 计算 $\text{cost}(i, j)$ 的值。

```

int getcost(int i, int j)
{
    if(i>j)return 0;
    else return tot-wt[i]*(dist[j]-dist[i])-wt[j]*(dist[n+1]-dist[j]);
}

```

comp 递归计算最优值如下。

```

void comp(int minp1, int maxp1, int minp2, int maxp2)
{
    int i, j, cost, opt, optcost;
    if(minp1>=minp2) minp2=minp1-1;
    if(maxp1>=maxp2) maxp1=maxp2-1;
    if((minp1>maxp1) || (minp2>maxp2)) return;
    // 计算 opt((minp2+maxp2)/2)
    optcost=MAXCOST+1;
    j=(minp2+maxp2)/2;
    for (i=minp1; i<=min(maxp1, j-1); i++) {
        cost=getcost(i, j);
        if(cost<optcost) {
            opt=i;
            optcost=cost;
        }
    }
    if(optcost<mincost) mincost=optcost;
    comp(minp1, opt, minp2, (minp2+maxp2)/2-1);
    comp(opt, maxp1, (minp2+maxp2)/2+1, maxp2);
}

```

comp(1, $n-1, 2, n$) 完成整个计算。

```

int main()
{
    readin();
    mincost=MAXCOST+1;
    comp(1, n-1, 2, n);
    cout<<mincost<<endl;
    return 0;
}

```

}
算法所需的计算时间为 $O(n\log n)$ 。

算法实现题 3-25 树的最大连通分支问题

★问题描述:

给定一棵树 T , 树中每个顶点 u 都有一个权 $w(u)$, 权可以是负数。现在要找到树 T 的一个连通子图使该子图的权之和最大。

★编程任务:

对于给定的树 T , 编程计算树 T 的最大连通分支。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n , 表示树 T 有 n 个顶点。树 T 的顶点编号为 $1, 2, \dots, n$ 。第 2 行有 n 个整数, 表示 n 个顶点的权值。接下来的 $n-1$ 行中, 每行有表示树 T 的一条边的 2 个整数 u, v , 表示顶点 u 与顶点 v 相连。

★结果输出:

将计算出的最大连通分支的权值输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5	4
-1 1 3 1 -1	
4 1	
1 3	
1 2	
4 5	

分析与解答:

此题是树型动态规划算法。

设 $\text{sum}(k)$ 是以第 k 个顶点为根的子树中的最大连通分支的权值, 则 $\text{sum}(k)$ 满足如下递归式:

$$\text{sum}(k) = w(k) + \sum_{\text{sum}(i) > 0} \text{sum}(i)$$

顶点 i 是顶点 k 的儿子结点。所求的最大连通分支的权值为 $\max_{1 \leq k \leq n} \{\text{sum}(k)\}$ 。

据此可设计如下树型动态规划算法。

树结点结构如下。

```
typedef struct node *link;
struct node{
    int s;
    struct node *next;
};
```

init 读入初始数据。

```

void init()
{
    int a,b;
    cin>>n;
    for(int i=1;i<=n;i++)cin>>w[i];
    for(i=1;i<=n-1;i++){
        cin>>a>>b;
        link p=new node;
        p->s=a;
        p->next=child[b];
        child[b]=p;
        p=new node;
        p->s=b;
        p->next=child[a];
        child[a]=p;
    }
}

```

build 建立树结构。

```

void build(int k)
{
    link p,q;
    vis[k]=1;
    p=child[k];
    while(p){
        int i=p->s;
        if(vis[i])
            if(p==child[k]){
                child[k]=p->next;
                delete p;
                p=child[k];
            }
            else{
                q->next=p->next;
                delete p;
                p=q->next;
            }
        else{
            build(i);
            if(p==child[k])q=child[k];
            else q=q->next;
            p=p->next;
        }
    }
}

```

```

    }
}
}

```

comp 实现动态规划算法。

```

void comp(int k)
{
    link p=child[k];
    while(p){
        int i=p->s;
        comp(i);
        if(sum[i]>0)sum[k]+=sum[i];
        p=p->next;
    }
    sum[k]+=w[k];
}

```

主函数 main 如下。

```

int main()
{
    init();
    build(1);
    comp(1);
    out();
    return 0;
}

```

out 输出最优值。

```

void out()
{
    for(int i=1;i<=n;i++)if(sum[i]>maxw)maxw=sum[i];
    cout<<maxw<<endl;
}

```

算法需要的计算时间显然为 $O(n)$ 。

算法实现题 3-26 直线 k 中值问题

★问题描述：

在一个按照南北方向划分成规整街区的城市里， n 个居民点分布在一条直线上的 n 个坐标点 $x_1 < x_2 < \dots < x_n$ 处。居民们希望在城市中至少选择一个，但不超过 k 个居民点建立服务机构。在每个居民点 x_i 处，服务需求量为 $w_i \geq 0$ ，在该居民点设置服务机构的费用为 $c_i \geq 0$ 。

假设居民点 x_i 到距其最近的服务机构的距离为 d_i , 则居民点 x_i 的服务费用为 $w_i \cdot d_i$ 。

建立 k 个服务机构的总费用为 $A+B$ 。 A 是在 k 个居民点设置服务机构的费用的总和; B 是 n 个居民点服务费用的总和。

★编程任务:

对于给定直线 L 上的 n 个点 $x_1 < x_2 < \dots < x_n$, 编程计算在直线 L 上最多设置 k 处服务机构的最小总费用。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 k 。 n 表示直线 L 上有 n 个点 $x_1 < x_2 < \dots < x_n$; k 是服务机构总数的上限。接下来的 n 行中, 每行有 3 个整数。第 $i+1$ 行的 3 个整数 x_i, w_i, c_i , 分别表示相应居民点的位置坐标, 服务需求量和在该点设置服务机构的费用。

★结果输出:

将计算的最小服务费用输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
9 3	19
2 1 2	
3 2 1	
6 3 3	
7 1 1	
9 3 2	
15 1 6	
16 2 1	
18 1 2	
19 1 1	

分析与解答:

直线 L 上的每个点 x_i 是一个客户, 其服务需求量为 $w(i)$ 。每个点 x_i 到服务机构 S 的距离定义为 $d(i, s) = \min_{y \in S} \{ |x_i - y| \}$ 。服务机构 S 的总服务费用为

$$\text{cost}(S) = \sum_{x_i \in S} c(i) + \sum_{j=1}^n w(j) \cdot d(j, S)$$

设 $\text{opt}(i, m)$ 是在 x_1, x_2, \dots, x_m 中恰好设置 i 个服务机构的最小费用; $\text{popt}(i, m)$ 是在 x_1, x_2, \dots, x_m 中恰好设置 i 个服务机构且在 x_m 处设置了服务机构的最小费用。

$$\text{opt}(i, m) = \min_{S \subseteq V_m, |S|=i} \left(\sum_{x_i \in S} c(i) + \sum_{j=1}^m w(j) \cdot d(j, S) \right)$$

$$\text{popt}(i, m) = \min_{S \subseteq V_m, |S|=i, x_m \in S} \left(\sum_{x_i \in S} c(i) + \sum_{j=1}^m w(j) \cdot d(j, S) \right)$$

由此可知, 所求的最优值为

$$\text{opt} = \min_{1 \leq k \leq k} (\text{opt}(i, n))$$

$\text{opt}(i, m)$ 和 $\text{popt}(i, m)$ 具有最优子结构性质, 且满足如下递归式:

$$\text{opt}(i, m) = \min_{1 \leq j \leq m} \left(\text{popt}(i, j) + \sum_{t=j+1}^m w(t) \cdot d(j, t) \right)$$

$$\text{popt}(i, m) = \min_{i-1 \leq j \leq m-1} \left(\text{opt}(i-1, j) + \sum_{t=j+1}^{m-1} w(t) \cdot d(t, m) \right) + c(m)$$

式中, $d(j, t) = x_t - x_j$ 是点 x_i 和 x_j 之间的距离。

边界条件是, 当 $i=1$ 时, $S=\{x_m\}$,

$$\text{popt}(1, m) = c(m) + \sum_{t=1}^{m-1} w(t) \cdot d(t, m)$$

在算法预处理时, 用 $O(n)$ 时间计算出 $\text{sw}(m) = \sum_{i=1}^m w(i)$ 和 $\text{wd}(m) = \sum_{i=1}^m w(i) \cdot$

$d(1, i)$, 则可在 $O(1)$ 时间内计算 $\sum_{t=j+1}^m w(t) \cdot d(j, t)$ 和 $\sum_{t=j+1}^{m-1} w(t) \cdot d(t, m)$ 如下。

$$\sum_{t=j+1}^m w(t) \cdot d(j, t) = \text{wd}(m) - \text{wd}(j) - [\text{sw}(m) - \text{sw}(j)] \cdot d(1, j)$$

$$\sum_{t=j+1}^{m-1} w(t) \cdot d(t, m) = [\text{sw}(m-1) - \text{sw}(j)] \cdot d(1, m) - [\text{wd}(m-1) - \text{wd}(j)]$$

据此可设计动态规划算法如下。

readin 读入初始数据并进行预处理。

```
void readin()
{
    int w;
    wt[0]=0; swt[0]=0;
    cin >> n >> m;
    for (int i=1; i<=n; i++) {
        cin >> x[i] >> w >> c[i];
        ww[i]=w;
        wt[i]=wt[i-1]+w;
        dist[i]=x[i]-x[1];
        swt[i]=swt[i-1]+w*dist[i];
    }
}
```

comp 实现动态规划算法。

```
void comp()
{
    int i, j, k, tmp;
    // opt2[1][j]
    for (j=1; j<=n; j++) opt2[1][j]=c[j]+getw2(0, j);
    // opt1[1][j]
    for (j=1; j<=n; j++) {
        opt1[1][j]=opt2[1][1]+getw1(1, j);
```

```

        for(k=2;k<=j;k++){
            tmp=opt2[1][k]+getw1(k,j);
            if(opt1[1][j]>tmp)opt1[1][j]=tmp;
        }
    }

    for(i=2;i<=m;i++){
        // opt2[i][j]
        for(j=i;j<=n;j++){
            opt2[i][j]=opt1[i-1][i-1]+getw2(i-1,j);
            for(k=i;k<=j;k++){
                tmp=opt1[i-1][k]+getw2(k,j);
                if(opt2[i][j]>tmp)opt2[i][j]=tmp;
            }
            opt2[i][j]+=c[j];
        }
        // opt1[i][j]
        for(j=i;j<=n;j++){
            opt1[i][j]=opt2[i][i]+getw1(i,j);
            for(k=i+1;k<=j;k++){
                tmp=opt2[i][k]+getw1(k,j);
                if(opt1[i][j]>tmp)opt1[i][j]=tmp;
            }
        }
    }
}

```

其中, getw1 和 getw2 分别用 $O(1)$ 时间计算 $\sum_{t=j+1}^m w(t) \cdot d(j,t)$ 和 $\sum_{t=j+1}^{m-1} w(t) \cdot d(t,m)$ 的值。

```

int getw1(int j,int m)
{
    return (swt[m]-swt[j])-(wt[m]-wt[j])*dist[j];
}

int getw2(int j,int m)
{
    return (wt[m-1]-wt[j])*dist[m]-(swt[m-1]-swt[j]);
}

```

算法的主函数如下。

```

int main()

```

```

{
    readin();
    comp();
    cout<<solution()<<endl;
    return 0;
}

```

solution 输出最优值。

```

int solution()
{
    int tmp=opt1[1][n];
    for(int i=2;i<=m;i++)
        if(opt1[i][n]<tmp) tmp=opt1[i][n];
    return tmp;
}

```

从动态规划递归式可知，算法的计算时间为 $O(kn^2)$ ，使用空间 $O(kn)$ 。

注意到动态规划计算只用到矩阵 opt1 和 opt2 的一行数据，可进一步将空间需求减少到 $O(n)$ 。

```

void comp()
{
    int i, j, k, tmp;
    // opt2[j]
    for(j=1; j<=n; j++) opt2[j]=c[j]+getw2(0, j);
    // opt1[j]
    for(j=1; j<=n; j++){
        opt1[j]=opt2[1]+getw1(1, j);
        for(k=2; k<=j; k++){
            tmp=opt2[k]+getw1(k, j);
            if(opt1[j]>tmp) opt1[j]=tmp;
        }
    }
    min=opt1[n];
    for(i=2; i<=m; i++){
        // opt2[j]
        for(j=i; j<=n; j++){
            opt2[j]=opt1[i-1]+getw2(i-1, j);
            for(k=i; k<=j; k++){
                tmp=opt1[k]+getw2(k, j);
                if(opt2[j]>tmp) opt2[j]=tmp;
            }
            opt2[j]+=c[j];
        }
    }
}

```

```

// opt1[j]
for(j=i;j<=n;j++){
    opt1[j]=opt2[i]+getw1(i,j);
    for(k=i+1;k<=j;k++){
        tmp=opt2[k]+getw1(k,j);
        if(opt1[j]>tmp)opt1[j]=tmp;
    }
}
if(opt1[n]<min)min=opt1[n];
}
}

```

最优值在全局变量 min 中。

算法实现题 3-27 直线 k 覆盖问题

★问题描述:

给定一条直线 L 上的 n 个点 $x_1 < x_2 < \dots < x_n$, 每个点 x_i 都有一个权 $w(i) \geq 0$, 以及在该点设置服务机构的费用 $c(i) \geq 0$ 。每个服务机构的覆盖半径为 r 。直线 k 覆盖问题是要求找出 $V_n = \{x_1, x_2, \dots, x_n\}$ 的一个子集 $S \subseteq V_n, |S| \leq k$, 在点集 S 处设置服务机构, 使总覆盖费用达到最小。

直线 L 上的每个点 x_i 是一个客户。每个点 x_i 到服务机构 S 的距离定义为 $d(i, S) = \min_{y \in S} \{ |x_i - y| \}$ 。如果客户 x_i 在 S 的服务覆盖范围内, 即 $d(i, S) \leq r$, 则其服务费用为 0, 否则其服务费用为 $w(i)$ 。服务机构 S 的总覆盖费用为

$$\text{cost}(S) = \sum_{x_i \in S} c(i) + \sum_{j=1}^n w(j) \cdot I(j, S)$$

式中, $I(j, S)$ 的定义为

$$I(j, S) = \begin{cases} 0 & d(j, S) \leq r \\ 1 & d(j, S) > r \end{cases}$$

★编程任务:

对于给定直线 L 上的 n 个点 $x_1 < x_2 < \dots < x_n$, 编程计算在直线 L 上最多设置 k 处服务机构的最小覆盖费用。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 3 个正整数 n, k 和 r 。 n 表示直线 L 上有 n 个点 $x_1 < x_2 < \dots < x_n$; k 是服务机构总数的上限; r 是服务机构的覆盖半径。接下来的 n 行中, 每行有 3 个整数。第 $i+1$ 行的 3 个整数 x_i, w_i, c_i 分别表示 $x(i), w(i)$ 和 $c(i)$ 。

★结果输出:

将计算的最小覆盖费用输出到文件 output.txt。

输入文件示例

input.txt

9 3 2

2 1 12

输出文件示例

output.txt

12

3 2 11
6 3 3
7 1 11
9 3 12
15 1 6
16 2 11
18 1 2
19 1 11

分析与解答:

设 $\text{opt}(i, m)$ 是在 x_1, x_2, \dots, x_m 中设置 i 个服务机构的最小费用; $\text{popt}(i, m)$ 是在 x_1, x_2, \dots, x_m 中设置 i 个服务机构且在 x_m 处设置了服务机构的最小费用。

$$\text{opt}(i, m) = \min_{S \subseteq V_m, |S|=i} \left(\sum_{x_i \in S} c(i) + \sum_{j=1}^m w(j) \cdot I(j, S) \right)$$

$$\text{popt}(i, m) = \min_{S \subseteq V_m, |S|=i, x_m \in S} \left(\sum_{x_i \in S} c(i) + \sum_{j=1}^m w(j) \cdot I(j, S) \right)$$

由此可知, 所求的最优值为

$$\text{opt} = \min_{1 \leq i \leq k} [\text{opt}(i, n)]$$

$\text{opt}(i, m)$ 和 $\text{popt}(i, m)$ 具有最优子结构性质, 且满足如下递归式:

$$\text{opt}(i, m) = \min \{ w(m) + \text{opt}(i, m-1), \min_{\text{cov}(m) \leq j \leq m} \text{popt}(i, j) \}$$

$$\text{popt}(i, m) = c(m) + \min_{\text{unc}(m) \leq j \leq m-1} \text{opt}(i-1, j)$$

其中, $\text{cov}(j) = \min \{ i; i \leq j, x_j - x_i \leq r \}$ 是 x_1, x_2, \dots, x_j 中被 x_j 覆盖的最小下标。

$\text{unc}(j) = \max \{ i; i < j, x_j - x_i > r \}$ 是 x_1, x_2, \dots, x_j 中未被 x_j 覆盖的最大下标。

边界条件是, 当 $i=1$ 时, $S=\{x_m\}$,

$$\text{popt}(1, m) = c(m) + \sum_{t=1}^{\text{unc}(m)} w(t)$$

据此可设计动态规划算法如下。

readin 读入初始数据。

```
void readin()
{
    cin >> n >> m >> r;
    for (int i=1; i<=n; i++) cin >> x[i] >> w[i] >> c[i];
}
```

comp 实现动态规划算法。

```
void comp()
{
    int i, j, k, h, tmp;
    // opt2[1][j]
    for (j=1; j<=n; j++) {
```

```

        h=unc(j);
        opt2[1][j]=c[j];
        for(k=1;k<=h;k++)opt2[1][j]+=w[k];
    }
    // opt1[1][j]
    for(j=1;j<=n;j++){
        if(j>1)opt1[1][j]=w[j]+opt1[1][j-1];
        else opt1[1][j]=INT_MAX;
        h=cov(j);tmp=INT_MAX;
        for(k=h;k<=j;k++){
            if(opt2[1][k]<tmp)tmp=opt2[1][k];
            if(opt1[1][j]>tmp)opt1[1][j]=tmp;
        }

    for(i=2;i<=m;i++){
        // opt2[i][j]
        for(j=i;j<=n;j++){
            h=unc(j);if(h<i-1)h=i-1;
            opt2[i][j]=INT_MAX;
            for(k=h;k<=j;k++){
                tmp=opt1[i-1][k];
                if(opt2[i][j]>tmp)opt2[i][j]=tmp;
            }
            opt2[i][j]+=c[j];
        }
        // opt1[i][j]
        for(j=i;j<=n;j++){
            h=cov(j);if(h<i)h=i;
            tmp=INT_MAX;
            if(j>i)opt1[i][j]=w[j]+opt1[i][j-1];
            else opt1[i][j]=INT_MAX;
            for(k=h;k<=j;k++){
                if(opt2[i][k]<tmp)tmp=opt2[i][k];
                if(opt1[i][j]>tmp)opt1[i][j]=tmp;
            }
        }
    }
}

```

cov 和 unc 分别计算 cov(j) 和 unc(j) 的值。

```

int cov(int j)
{
    for(int i=j;i>0;i--)if(x[j]-x[i]>r)break;
    return i+1;
}

```

```

    }

    int unc(int j)
    {
        for(int i=1;i<=j;i++)if(x[j] - x[i]<=r)break;
        return i - 1;
    }

```

算法的主函数如下。

```

int main()
{
    readin();
    comp();
    cout<<solution()<<endl;
    return 0;
}

```

solution 输出最优值。

```

int solution()
{
    int tmp=opt1[1][n];
    for(int i=2;i<=m;i++)
        if(opt1[i][n]<tmp)tmp=opt1[i][n];
    return tmp;
}

```

从动态规划递归式可知，算法的计算时间为 $O(kn^2)$ ，使用空间 $O(kn)$ 。

注意到动态规划计算只用到矩阵 opt1 和 opt2 的一行数据，可进一步将空间需求减小到 $O(n)$ 。

```

void comp()
{
    int i, j, k, h, tmp;
    // opt2[j]
    for(j=1;j<=n;j++){
        h=unc(j);
        opt2[j]=c[j];
        for(k=1;k<=h;k++)opt2[j]+=w[k];
    }
    // opt1[j]
    for(j=1;j<=n;j++){
        if(j>1)opt1[j]=w[j]+opt1[j-1];
    }
}

```

```

        else opt1[j]=INT_MAX;
        h=cov(j);tmp=INT_MAX;
        for(k=h;k<=j;k++){
            if(opt2[k]<tmp)tmp=opt2[k];
            if(opt1[j]>tmp)opt1[j]=tmp;
        }
        min=opt1[n];
        for(i=2;i<=m;i++){
            // opt2[j]
            for(j=i;j<=n;j++){
                h=unc(j);if(h<i-1)h=i-1;
                opt2[j]=INT_MAX;
                for(k=h;k<=j;k++){
                    tmp=opt1[k];
                    if(opt2[j]>tmp)opt2[j]=tmp;
                }
                opt2[j]+=c[j];
            }
            // opt1[j]
            for(j=i;j<=n;j++){
                h=cov(j);if(h<i)h=i;
                tmp=INT_MAX;
                if(j>i)opt1[j]=w[j]+opt1[j-1];
                else opt1[j]=INT_MAX;
                for(k=h;k<=j;k++){
                    if(opt2[k]<tmp)tmp=opt2[k];
                    if(opt1[j]>tmp)opt1[j]=tmp;
                }
                if(opt1[n]<min)min=opt1[n];
            }
        }
    }
}

```

最优值在全局变量 min 中。

算法实现题 3-28 m 处理器问题

★问题描述:

在一个网络通信系统中, 要将 n 个数据包依次分配给 m 个处理器进行数据处理, 并要求处理器负载尽可能均衡。

设给定的数据包序列为 $\{\sigma_0, \sigma_1, \dots, \sigma_{n-1}\}$ 。

m 处理器问题要求的是 $r_0=0 \leq r_1 \leq \dots \leq r_{m-1} \leq n=r_m$, 将数据包序列划分为 m 段: $\{\sigma_0, \dots, \sigma_{r_1-1}\}, \{\sigma_{r_1}, \dots, \sigma_{r_2-1}\}, \dots, \{\sigma_{r_{m-1}}, \dots, \sigma_{n-1}\}$, 使 $\max_{i=0}^{m-1} \{f(r_i, r_{i+1})\}$ 达到最小。

式中, $f(i, j) = \sqrt{\sigma_i^2 + \dots + \sigma_j^2}$ 是序列 $\{\sigma_i, \dots, \sigma_j\}$ 的负载量。

$\max_{i=0}^{m-1} \{f(r_i, r_{i+1})\}$ 的最小值称为数据包序列 $\{\sigma_0, \sigma_1, \dots, \sigma_{n-1}\}$ 的均衡负载量。

★编程任务:

对于给定的数据包序列 $\{\sigma_0, \sigma_1, \dots, \sigma_{n-1}\}$, 编程计算 m 个处理器的均衡负载量。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m 。 n 表示数据包个数, m 表示处理器数。接下来的 1 行中有 n 个整数, 表示 n 个数据包的大小。

★结果输出:

将计算的处理器均衡负载量输出到文件 output.txt, 且保留 2 位小数。

输入文件示例	输出文件示例
input.txt	output.txt
6 3	12.32
2 2 12 3 6 11	

分析与解答:

设 $g(i, k)$ 是将 $\{\sigma_i, \dots, \sigma_{n-1}\}$ 划分为 k 段的均衡负载量, 所求的最优值为 $g(0, m)$ 。

$g(i, k)$ 具有最优子结构性质, 且满足如下递归式:

当 $2 \leq k < n-i$ 时, $g(i, k) = \min_{i \leq j \leq n-k} \max\{f(i, j), g(j+1, k-1)\}$ 。

当 $n-i < k \leq m$ 时, $g(i, k) = g(i, n-i)$ 。

边界条件是: $g(i, 1) = f(i, n-1); g(i, n-i) = \max_{i \leq j < n} f(j, j)$ 。

据此可设计动态规划算法如下。

```

void solve(int n, int m)
{
    int i, j, k, tmp, maxt;
    tmp = f(n-1, n-1);
    for(i=n-1; i>=0; i--) {
        if(f(i, i) > tmp) tmp = f(i, i);
        g[i][1] = f(i, n-1);
        if(n-i <= m) g[i][n-i] = tmp;
    }
    for(i=n-1; i>=0; i--) {
        for(k=2; k<=m; k++) {
            for(j=i, tmp=INT_MAX; j<=n-k; j++) {
                maxt = max(f(i, j), g[j+1][k-1]);
                if(tmp > maxt) tmp = maxt;
            }
            g[i][k] = tmp;
        }
        for(k=n-i+1; k<=m; k++) g[i][k] = g[i][n-i];
    }
}

```

算法所需的计算时间为 $O(mn^2)$ 。

上述动态规划算法适用于一般的函数 f 。本题的函数 f 具有非负性和单调性,即对于 $0 \leq i \leq j \leq n-1$ 有, $f(i, j) \geq 0$; $f(i+1, j) < f(i, j) < f(i, j+1)$ 。

利用函数的单调性可进一步将算法的计算时间降低到 $O(m(n-m))$ 如下。

```
void solve(int n, int m)
{
    int i, j, k;
    for (i = n - 1; i >= 0; i--) g[i][1] = f(i, n - 1);
    for (k = 2; k <= m; k++) {
        g[n - k][k] = max(f(n - k, n - k), g[n - k + 1][k - 1]);
        j = n - k;
        for (i = n - k - 1; i >= m - k; i--) {
            if (f(i, j) <= g[j + 1][k - 1]) g[i][k] = g[j + 1][k - 1];
            else
                if (f(i, i) >= g[i + 1][k - 1]) {g[i][k] = f(i, i); j = i;}
                else {
                    while (f(i, j - 1) >= g[j][k - 1]) j--;
                    g[i][k] = min(f(i, j), g[j][k - 1]);
                    if (g[i][k] == g[j][k - 1]) j--;
                }
        }
    }
}
```

算法实现题 3-29 红黑树的红色内结点问题

★问题描述:

红黑树是一类特殊的二叉搜索树,其中每个结点被“染成”红色或黑色。若将二叉搜索树结点中的空指针看作是指向一个空结点,则称这类空结点为二叉搜索树的前端结点。并规定所有前端结点的高度为-1。

一棵红黑树是满足下面“红黑性质”的染色二叉搜索树:

- (1) 每个结点被染成红色或黑色;
- (2) 每个前端结点为黑色结点;
- (3) 任一红结点的儿子结点均为黑结点;
- (4) 在从任一结点到其子孙前端结点的所有路径上具有相同的黑结点数。

从红黑树中任一结点 x 出发 (不包括结点 x), 到达一个前端结点的任意一条路径上的黑结点个数称为结点 x 的黑高度, 记作 $bh(x)$ 。红黑树的黑高度定义为其根结点的黑高度。

如图 3-4 所示的二叉搜索树是一棵红黑树。标在结点旁边的数字是相应结点的黑高度。

★编程任务:

给定正整数 n , 试设计一个算法, 计算出在所有含有 n 个结点的红黑树中, 红色内结点

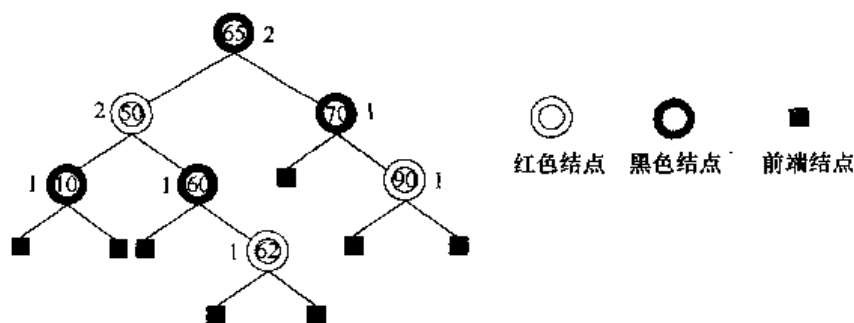


图 3-4 红黑树

个数的最小值和最大值。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行是正整数 n , $1 < n < 5000$ 。

★结果输出:

程序运行结束时, 将红色内结点个数的最小值和最大值输出到文件 output.txt。第 1 行是最小值, 第 2 行是最大值。

输入文件示例

input.txt

8

输出文件示例

output.txt

1

4

分析与解答:

(1) 特殊情况 $n = 2^k - 1$

最低层结点为红结点时, 所含红结点数最多。全为黑结点时, 所含红结点数最少。

设结点数为 n 时, 所含最多红结点数为 $r(n)$ 。

当 $n = 2^k - 1$ 时,

$$\begin{aligned}
 r(n) &= r(2^k - 1) \\
 &= \sum_{i=0}^{\lfloor (k-1)/2 \rfloor} 2^{k-2i-1} \\
 &= 2^{k-1} \sum_{i=0}^{\lfloor (k-1)/2 \rfloor} \frac{1}{4^i} \\
 &= \frac{2^{k-1}}{3} \left(4 - \frac{1}{4^{\lfloor (k-1)/2 \rfloor}} \right) \\
 &= \frac{2^{k+1} - 2^{k-1-2\lfloor (k-1)/2 \rfloor}}{3} \\
 &= \frac{2^{k+1} - 2^{(k-1) \bmod 2}}{3} \\
 &= \frac{2^{k+1} - 2 + k \bmod 2}{3} \\
 &= \frac{2(2^k - 1) + k \bmod 2}{3} \\
 &= \frac{2n + \log(n+1) \bmod 2}{3}
 \end{aligned}$$

此时, 黑结点数 $b(n)$ 为

$$\begin{aligned}
 b(n) &= n - r(n) \\
 &= n - \frac{2n + \log(n+1) \bmod 2}{3} \\
 &= \frac{n - \log(n+1) \bmod 2}{3}
 \end{aligned}$$

由此可得

$$\frac{r(n)}{b(n)} = \frac{2n + \log(n+1) \bmod 2}{n - \log(n+1) \bmod 2}$$

当 $k \bmod 2 = 0$ 时, $\frac{r(n)}{b(n)} = 2$;

当 $k \bmod 2 = 1$ 时, $\frac{r(n)}{b(n)} = \frac{2n+1}{n-1}$ 。

由此可知, 对于任意 k , 当 $n = 2^k - 1$ 时有

$$0 \leq \frac{r(n)}{b(n)} \leq \frac{2n+1}{n-1}$$

稍后将此式推广到任意的 n 。

由于

$$\frac{d}{dx} \left(\frac{2x+1}{x-1} \right) = -\frac{3}{(x-1)^2} < 0, \quad \lim_{n \rightarrow \infty} \frac{2n+1}{n-1} = 2$$

因此 $\frac{2n+1}{n-1}$ 单调递减趋向于 2。

当 $n=1$ 时, $k=1, r(n)=n, b(n)=0$, 这是特殊情况。除此之外, 在 $k=3$, 即 $n=7$ 时

$$\frac{r(n)}{b(n)} = \frac{2n+1}{n-1}$$

达到极值 $5/2 = 2.5$ 。

由此得

$$0 \leq \frac{r(n)}{b(n)} \leq \frac{2n+1}{n-1} \leq 2.5$$

也就是说, 红黑结点的最大比值为 2.5, 最小比值为 0。当 $n=7$ 时, 所有结点均为黑结点时, 达到最小比值 0; 图 3-5 所示的红黑树达到最大比值 2.5。

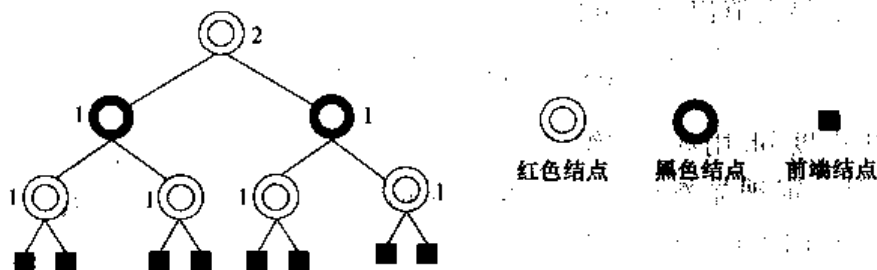


图 3-5 红黑树最大红黑结点比值为 2.5

(2) 一般情况

设红黑树结点数为 n , 根结点为红色结点的红黑树中红结点数的最大值记为 $\text{red}(n, 0)$ 。类似地, 根结点为黑色结点的红黑树中红结点数的最大值记为 $\text{red}(n, 1)$ 。

定理 1: $\text{red}(n,0) \leq \frac{2n+1}{3}$; $\text{red}(n,1) \leq \frac{2n}{3}$ 。

证明: 对结点数 n 用数学归纳法。

当 $n=0,1$ 时, 定理显然成立。

设定理对于结点数 $n-1$ 成立, 则对于结点数 n 有

$$\begin{aligned}\text{red}(n,0) &\leq \max_{0 \leq i \leq n/2} \{ \text{red}(i,1) + \text{red}(n-i-1,1) + 1 \} \\ &\leq \max_{0 \leq i \leq n/2} \left\{ \frac{2i}{3} + \frac{2(n-i-1)}{3} + 1 \right\} \\ &= \frac{2n+1}{3} \\ \text{red}(n,1) &\leq \max_{0 \leq i \leq n/2} \{ \text{red}(i,0) + \text{red}(n-i-1,0) \} \\ &\leq \max_{0 \leq i \leq n/2} \left\{ \frac{2i+1}{3} + \frac{2(n-i-1)+1}{3} \right\} \\ &= \frac{2n}{3}\end{aligned}$$

由数学归纳法即知定理成立。

定理 2: 设在所有结点数为 n 的红黑树中, 所含最多红结点数为 $r(n)$, 则

$$0 \leq \frac{r(n)}{n-r(n)} \leq \frac{2n+1}{n-1} \leq 2.5$$

证明: 由定理 1 知,

$$\begin{aligned}r(n) &= \max \{ \text{red}(n,0), \text{red}(n,1) \} \\ &\leq \max \left\{ \frac{2n}{3}, \frac{2n+1}{3} \right\} = \frac{2n+1}{3}\end{aligned}$$

由此即知

$$0 \leq \frac{r(n)}{n-r(n)} \leq \frac{\frac{2n+1}{3}}{n - \frac{2n+1}{3}} = \frac{2n+1}{n-1} \leq 2.5$$

定理 3: 设红黑树的黑高度为 bh 。在所有黑高度为 bh 且根结点为红色结点的红黑树中红结点数的最大值记为 $\text{red}(bh,0)$ 。类似地, 在所有黑高度为 bh 且根结点为黑色结点的红黑树中红结点数的最大值记为 $\text{red}(bh,1)$, 则有

$$\text{red}(bh,0) = \frac{2^{2bh}-1}{3}; \text{red}(bh,1) = \frac{2^{2bh+1}-2}{3}$$

证明: 对黑高度 bh 用数学归纳法。

当 $bh=0$ 时, 定理显然成立。当 $bh=1$ 时, $\text{red}(1,0)$ 和 $\text{red}(1,1)$ 分别对应于图 3-6 中的 2 棵红黑树。

此时显然有, $\text{red}(1,0)=1, \text{red}(1,1)=2$, 定理成立。

设定理对于黑高度 $bh-1$ 成立, 则对于黑高度 bh 有

$$\begin{aligned}\text{red}(bh,0) &= 2\text{red}(bh-1,1) + 1 \\ &= 2 \frac{2^{2(bh-1)+1}-2}{3} + 1\end{aligned}$$

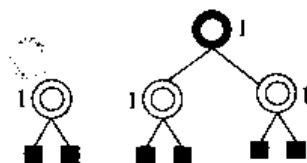


图 3-6 黑高度为 1 的最多红结点红黑树

$$\begin{aligned}
&= \frac{2^{2bh}-4}{3} + 1 \\
&= \frac{2^{2bh}-1}{3} \\
\text{red}(bh, 1) &= 2\text{red}(bh, 0) \\
&= 2 \frac{2^{2bh}-1}{3} \\
&= \frac{2^{2bh+1}-2}{3}
\end{aligned}$$

由数学归纳法即知定理成立。

(3) 最少红结点数

设结点数为 n 的红黑树中红结点数的最小值记为 $\text{minr}(n)$ 。与定理 2 类似地有以下定理。

定理 4: 设在所有结点数为 n 的红黑树中, 所含最少红结点数为 $\text{minr}(n)$, 则

$$0 \leq \text{minr}(n) \leq \frac{n}{3}$$

从而有

$$0 \leq \frac{\text{minr}(n)}{n - \text{minr}(n)} \leq \frac{\frac{n}{3}}{n - \frac{n}{3}} = 0.5$$

证明: 对结点数 n 用数学归纳法。

当 $n=6$ 时, 达到这个最大比值, 如图 3-7 所示。

(4) 动态规划算法

设 $\text{red}(i, j, 0)$ 表示结点数为 i , 黑高度为 j , 且根结点为红色结点的红黑树中红结点最大值;
 $\text{red}(i, j, 1)$ 表示结点数为 i , 黑高度为 j , 且根结点为黑色结点的红黑树中红结点最大值。

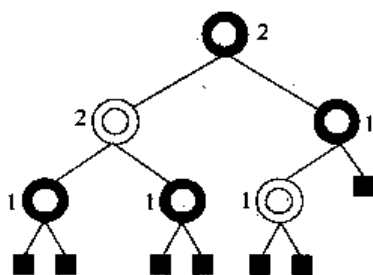


图 3-7 最少红结点最大比值红黑树

$$r_1 = \max_{0 \leq k \leq i/2} \{ \text{red}(k, j-1, 1) + \text{red}(i-k-1, j-1, 1) \}$$

$$r_2 = \max_{0 \leq k \leq i/2} \{ \text{red}(k, j, 0) + \text{red}(i-k-1, j, 0) \}$$

$$r_3 = \max_{0 \leq k \leq i/2} \{ \text{red}(k, j, 0) + \text{red}(i-k-1, j-1, 1) \}$$

$$r_4 = \max_{0 \leq k \leq i/2} \{ \text{red}(k, j-1, 1) + \text{red}(i-k-1, j, 0) \}$$

由前面的讨论易知, $\text{red}(i, j, k)$ 具有最优子结构性质, 且满足如下递归式:

$$\text{red}(i, j, k) = \begin{cases} r_1 + 1 & k=0 \\ \max\{r_1, r_2, r_3, r_4\} & k=1 \end{cases}$$

红结点最小值的计算也类似。

(5) 算法实现

用类 RBtree 表示算法结构。

```

class RBtree {
public:
    RBtree() {};
    ~RBtree() {};
    void readin();
    void output();
private:
    int m, n, ***red;
    void init();
    void dyna();
    void dynamin();
    int rmin(int i);
    int rmax(int i);
    void show();
};

```

readin 和 init 分别读入初始数据和进行初始化计算。

```

void RBtree::readin()
{
    cin >> n;
    m = 2 * ilog(n + 3);
    Make3DArray(red, n + 1, m + 1, 2);
}

void RBtree::init()
{
    for(int i = 0; i <= n; i++)
        for(int j = 0; j <= m; j++)
            for(int k = 0; k < 2; k++) red[i][j][k] = -1;
    red[0][0][0] = 0; red[0][0][1] = 0;
    red[1][1][0] = 1; red[1][1][1] = 0;
    red[2][1][1] = 1;
    red[3][1][1] = 2; red[3][2][0] = 1; red[3][2][1] = 0;
}

```

ilog 和 isum 分别计算以 2 为底的对数和非负数的和。

```

int ilog(int x)
{
    int i = (int) (log(x) / log(2));
    return i;
}

```

```

int isum(int x, int y)
{
    if(x<0||y<0) return -1;
    return x+y;
}

```

dyna 计算红结点最大值。

```

void RBtree::dyna()
{
    int i, j, jj, k;
    init();
    for(i=4; i<=n; i++)
        for(j=ilog(i+1)/2, jj=4*j; j<=jj; j++){
            int imax=-1;
            for(k=0; k<=i/2; k++){
                int temp=isum(red[k][j-1][1], red[i-k-1][j-1][1]);
                if(imax<temp) imax=temp;
            }
            if(imax>=0) red[i][j][0]=imax+1;
            imax=-1;
            for(k=0; k<=i/2; k++){
                int temp=isum(red[k][j][0], red[i-k-1][j][0]);
                if(imax<temp) imax=temp;
                temp=isum(red[k][j][0], red[i-k-1][j+1][1]);
                if(imax<temp) imax=temp;
                temp=isum(red[k][j-1][1], red[i-k-1][j][0]);
                if(imax<temp) imax=temp;
                temp=isum(red[k][j-1][1], red[i-k-1][j-1][1]);
                if(imax<temp) imax=temp;
            }
            if(imax>=0) red[i][j][1]=imax;
        }
}

```

dynamin 计算红结点最小值。

```

void RBtree::dynamin()
{
    int i, j, jj, k;
    init();
    for(i=4; i<=n; i++)
        for(j=ilog(i+1)/2, jj=4*j; j<=jj; j++){
            int imin=i+1;

```

```

        for(k=0;k<=i/2;k++){
            int temp=isum(red[k][j-1][1],red[i-k-1][j-1][1]);
            if(temp>=0 && imin>temp) imin=temp;
        }
        if(imin<i+1) red[i][j][0]=imin+1;
        imin=i+1;
        for(k=0;k<=i/2;k++){
            int temp=isum(red[k][j][0],red[i-k-1][j][0]);
            if(temp>=0 && imin>temp) imin=temp;
            temp=isum(red[k][j][0],red[i-k-1][j-1][1]);
            if(temp>=0 && imin>temp) imin=temp;
            temp=isum(red[k][j-1][1],red[i-k-1][j][0]);
            if(temp>=0 && imin>temp) imin=temp;
            temp=isum(red[k][j-1][1],red[i-k-1][j-1][1]);
            if(temp>=0 && imin>temp) imin=temp;
        }
        if(imin<i+1) red[i][j][1]=imin;
    }
}

```

output 执行动态规划算法并输出计算结果。

```

void RBtree::output()
{
    double ratio1=0,ratio2=0,temp=0;
    int *minr=new int[n+1];
    int *maxr=new int[n+1];
    dynamin();
    for(int i=1;i<=n;i++) minr[i]=rmin(i);
    dyna();
    for(i=1;i<=n;i++) maxr[i]=rmax(i);
    cout<<minr[n]<<endl<<maxr[n]<<endl;
}

```

rmin 和 rmax 分别计算整体最小值和整体最大值。

```

int RBtree::rmin(int i)
{
    int j,jj,imin=i+1;
    for(j=ilog(i+3)/2,jj=4*j;j<=jj;j++){
        if(red[i][j][0]>=0 && imin>red[i][j][0]) imin=red[i][j][0];
        if(red[i][j][1]>=0 && imin>red[i][j][1]) imin=red[i][j][1];
    }
    return imin;
}

```

```

    }

    int RBtree::rmax(int i)
    {
        int j, jj, imax=-1;
        for(j=ilog(i+3)/2, jj=4*j; j<=jj; j++){
            if(imax<red[i][j][0]) imax=red[i][j][0];
            if(imax<red[i][j][1]) imax=red[i][j][1];
        }
        return imax;
    }
}

```

完成全部计算的主函数如下。

```

int main()
{
    RBtree X;
    X.readin();
    X.output();
    return 0;
}

```

算法所需的计算时间显然为 $O(n\log n)$ 。

第4章 贪心算法

习题 4-2 活动安排问题的贪心选择

在活动安排问题中,还可以有其他的贪心选择方案,但并不能保证产生最优解。给出一个例子,说明若选择具有最短时段的相容活动作为贪心选择,得不到最优解。若选择覆盖未选择活动最少的相容活动作为贪心选择,也得不到最优解。

分析与解答:

见参考文献[1],68。

习题 4-3 背包问题的贪心选择性质

证明背包问题具有贪心选择性质。

分析与解答:

见参考文献[1],68~70。

习题 4-4 特殊的 0-1 背包问题

若在 0-1 背包问题中,各物品依重量递增排列时,其价值恰好依递减序排列。对这个特殊的 0-1 背包问题,设计一个有效算法找出最优解,并说明算法的正确性。

分析与解答:

见参考文献[1],72。

习题 4-10 程序最优存储问题

假定要把长为 l_1, l_2, \dots, l_n 的 n 个程序放在磁带 T_1 和 T_2 上,并且希望按照使最大检索时间取最小值的方式存放,即如果存放在 T_1 和 T_2 上的程序集合分别是 A 和 B ,则希望所选择的 A 和 B 使得 $\max\{\sum_{i \in A} l_i, \sum_{i \in B} l_i\}$ 取最小值。贪心算法:开始将 A 和 B 都初始化为空,然后一次考虑一个程序,如果 $\sum_{i \in A} l_i = \min\{\sum_{i \in A} l_i, \sum_{i \in B} l_i\}$,则将当前正在考虑的那个程序分配给 A ,否则分配给 B 。证明无论是按 $l_1 \leq l_2 \leq \dots \leq l_n$ 或是按 $l_1 \geq l_2 \geq \dots \geq l_n$ 的次序来考虑程序,这种方法都不能产生最优解。应当采用什么策略?写出一个完整的算法并证明其正确性。

分析与解答:

设变量 $x_i = 1$ 表示将 l_i 存放在 T_1 上,且 T_1 的检索时间较短,则

$$\sum_{i=1}^n l_i x_i - \sum_{i=1}^n l_i (1 - x_i) \leq 0, \quad 2 \sum_{i=1}^n l_i x_i \leq \sum_{i=1}^n l_i, \quad \sum_{i=1}^n l_i x_i \leq \frac{1}{2} \sum_{i=1}^n l_i$$

T_1 的检索时间应取最大值,因此问题归结为

$$\begin{aligned} & \max \sum_{i=1}^n l_i x_i \\ \text{s. t.} \quad & \sum_{i=1}^n l_i x_i \leq \frac{1}{2} \sum_{i=1}^n l_i \end{aligned}$$

这与主教材中第5章的装载问题等价,是一个特殊的 0-1 背包问题。

习题 4-13 最优装载问题的贪心算法

将最优装载问题的贪心算法推广到 2 艘船的情形, 贪心算法仍能产生最优解吗?

分析与解答:

贪心算法不能产生最优解, 见主教材第 5 章的装载问题。

习题 4-18 Fibonacci 序列的哈夫曼编码

字符 a~h 出现的频率恰好是前 8 个 Fibonacci 数, 它们的哈夫曼编码是什么? 将结果推广到 n 个字符的频率恰好是前 n 个 Fibonacci 数的情形。

分析与解答:

频率恰好是前 8 个 Fibonacci 数的哈夫曼编码树如图 4-1 所示。

在一般情况下, n 个字符的频率恰好是前 n 个 Fibonacci 数, 则相应的哈夫曼编码树深度为 $n-1$, 第 1 个字符的编码长度为 $n-1$ 。

自底向上第 i 个圆结点中的数为 $\sum_{k=0}^i f_k$ 。用数学归纳法容易证明

$\sum_{k=0}^i f_k < f_{i+2}$ 。该性质保证了频率恰好是前 n 个 Fibonacci 数的哈夫曼编码树具有所述形状和性质。

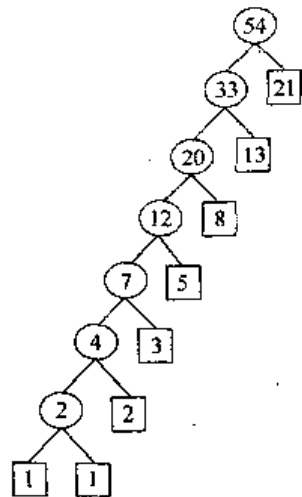


图 4-1 哈夫曼编码树

习题 4-19 最优前缀码的编码序列

设 $C=\{0,1,\dots,n-1\}$ 是 n 个字符的集合。证明关于 C 的任何最优前缀码可以表示为长度为 $2n-1+n\lceil \log n \rceil$ 位的编码序列。(提示用 $2n-1$ 位描述树结构)。

分析与解答:

任何最优前缀码所相应的编码二叉树是一棵完全二叉树, 有 n 个叶结点和 $n-1$ 个内结点。用 1 位表示 1 个结点的类型, 1 表示内结点, 0 表示叶结点, 总共需 $2n-1$ 位。对编码树的前序遍历可以惟一表示该编码树结构。

例如, 当 $n=4$ 时, 如图 4-2 所示编码树结构可以惟一表示为 1101000。

在每个叶结点后, 即每个 0 后面紧跟 $\lceil \log n \rceil$ 位表示该叶结点处的数字, 即可完整表示整棵编码树。例如, 图 4-2 中的编码树可表示为

110111001010000。由此可知, 在一般情况下, 最优前缀码可以表示长

度为 $2n-1+n\lceil \log n \rceil$ 位的编码序列。

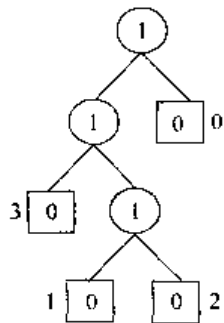


图 4-2 编码树结构

习题 4-21 任务集独立性问题

说明如何用引理 4.6 的性质 (2), 在 $O(|A|)$ 时间里确定给定的任务集 A 是否独立。

分析与解答:

见参考文献[1], 75。

习题 4-22 矩阵拟阵

给定 $n \times n$ 实值矩阵 T , 证明 (S, I) 是拟阵。其中, S 是 T 的列向量的集合, $A \in I$ 当且仅当 A 中的列是线性独立的。

分析与解答:

由线性空间理论中的基交换定理容易证明, I 满足拟阵的交换性质 (3), 从而证明 (S, I) 是拟阵。

习题 4-23 最小权最大独立子集拟阵

说明如何变换带权拟阵的权函数, 使最小权最大独立子集问题变换为等价的标准带权拟阵问题, 并证明变换的正确性。

分析与解答:

设带权拟阵 $M=(S, I)$ 的权函数为 W 。令 $W_0 = \max\{W(x) | x \in S\} + 1$ 。定义 $M=(S, I)$ 的另一权函数为 $W'(x) = W_0 - W(x)$ 。则容易证明, 权函数为 W 的最小权最大独立子集问题等价于权函数为 W' 的标准带权拟阵问题。

习题 4-27 整数边权 Prim 算法

假设具有 n 个顶点的连通带权图中所有边的权值均为从 1 到 n 之间的整数, 能对 Kruskal 算法进行何改进, 时间复性能改进到何程度? 若对某常量 N , 所有边的权值均为从 1 到 N 之间的整数, 在这种情况下又如何? 在上述两种情况下, 对 Prim 算法能进行何改进?

分析与解答:

见参考文献[1], 130。

习题 4-28 最大权最小生成树

试设计一个构造图 G 生成树的算法, 使得构造出的生成树的边的最大权值达到最小。

分析与解答:

见参考文献[1], 130~131。

习题 4-29 最短路径的负边权

试举例说明, 如果允许带权有向图中某些边的权为负实数, 则 Dijkstra 算法不能正确求得从源到所有其他顶点的最短路径长度。

分析与解答:

见参考文献[1], 131~132。

习题 4-30 整数边权 Dijkstra 算法

设 G 是具有 n 个顶点和 e 条边的带权有向图, 各边的权值为 0 到 $N-1$ 之间的整数, N 为一非负整数。修改 Dijkstra 算法使其能在 $O(Nn+e)$ 时间内计算出从源到所有其他顶点之间的最短路径长度。

分析与解答:

见参考文献[1], 133~136。

算法实现题 4-1 会场安排问题 (习题 4-1)

★问题描述:

假设要在足够多的会场里安排一批活动, 并希望使用尽可能少的会场。设计一个有效的贪

心算法进行安排（这个问题实际上是著名的图着色问题。若将每一个活动作为图的一个顶点，不相容活动间用边相连。使相邻顶点着有不同颜色的最小着色数，相应于要找的最小会场数。）

★编程任务：

对于给定的 k 个待安排的活动，编程计算使用最少会场的时间表。

★数据输入：

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 k ，表示有 k 个待安排的活动。接下来的 k 行中，每行有 2 个正整数，分别表示 k 个待安排的活动的开始时间和结束时间。时间以 0 点开始的分钟计。

★结果输出：

将编程计算出的最少会场数输出到文件 output.txt。

输入文件示例

输出文件示例

input.txt

output.txt

5

3

1 23

12 28

25 35

27 80

36 50

分析与解答：

见参考文献[1], 66~68。

具体算法描述如下。

```
int greedy(vector<point> x)
{
    int sum=0, curr=0, n=x.size();
    sort(x.begin(), x.end());
    for(int i=0; i<n; i++){
        if (x[i].leftend() curr++){
            else curr--;
            if((i==n-1 || x[i]<x[i+1]) && curr>sum) sum=curr; //处理 x[i]=x[i+1]
                                                                    的情况
        }
    }
    return sum;
}
```

算法实现题 4-2 最优合并问题（习题 4-5）

★问题描述：

给定 k 个排好序的序列 s_1, s_2, \dots, s_k ，用 2 路合并算法将这 k 个序列合并成一个序列。假设所采用的 2 路合并算法合并 2 个长度分别为 m 和 n 的序列需要 $m+n-1$ 次比较。试设计一个算法确定合并这个序列的最优合并顺序，使所需的总比较次数最少。

为了进行比较, 还需要确定合并这个序列的最差合并顺序, 使所需的总比较次数最多。

★编程任务:

对于给定的 k 个待合并序列, 编程计算最多比较次数和最少比较次数合并方案。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 k , 表示有 k 个待合并序列。接下来的 1 行中, 有 k 个正整数, 表示 k 个待合并序列的长度。

★结果输出:

将编程计算出的最多比较次数和最少比较次数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
4	78 52
5 12 11 2	

分析与解答:

见主教材中的 Huffman 算法。

算法实现题 4-3 磁带最优存储问题 (习题 4-6)

★问题描述:

设有 n 个程序 $\{1, 2, \dots, n\}$ 要存放在长度为 L 的磁带上。程序 i 存放在磁带上的长度是 $l_i, 1 \leq i \leq n$ 。这 n 个程序的读取概率分别是 p_1, p_2, \dots, p_n , 且 $\sum_{i=1}^n p_i = 1$ 。如果将这 n 个程序按 i_1, i_2, \dots, i_n 的次序存放, 则读取程序 i_r 所需的时间 $t_r = c \sum_{k=1}^r p_k l_{i_k}$ 。这 n 个程序的平均读取时间为 $\sum_{r=1}^n t_r$ 。

磁带最优存储问题要求确定这 n 个程序在磁带上的一个存储次序, 使平均读取时间达到最小。试设计一个解此问题的算法, 并分析算法的正确性和计算复杂性。

★编程任务:

对于给定的 n 个程序存放在磁带上的长度和读取概率, 编程计算 n 个程序的最优存储方案。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行是正整数 n , 表示文件个数。接下来的 n 行中, 每行有 2 个正整数 a 和 b , 分别表示程序存放在磁带上的长度和读取概率。实际上第 k 个程序的读取概率为 $a_k / \sum_{i=1}^n a_i$ 。对所有输入均假定 $c=1$ 。

★结果输出:

将编程计算出的最小平均读取时间输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5	85.6193
71 872	
46 452	

9 265

73 120

35 85

分析与解答:

贪心策略: 最短平均读取时间程序优先。

算法实现题 4-4 磁盘文件最优存储问题 (习题 4-7)

★问题描述:

设磁盘上有 n 个文件 f_1, f_2, \dots, f_n , 每个文件占用磁盘上的 1 个磁道。这 n 个文件的检索概率分别是 p_1, p_2, \dots, p_n , 且 $\sum_{i=1}^n p_i = 1$ 。磁头从当前磁道移到被检信息磁道所需的时间可用这 2 个磁道之间的径向距离来度量。如果文件 f_i 存放在第 i 道上, $1 \leq i \leq n$, 则检索这 n 个文件的期望时间是 $\sum_{1 \leq i < j \leq n} p_i p_j d(i, j)$ 。其中 $d(i, j)$ 是第 i 道与第 j 道之间的径向距离 $|i - j|$ 。

磁盘文件的最优存储问题要求确定这 n 个文件在磁盘上的存储位置, 使期望检索时间达到最小。试设计一个解此问题的算法, 并分析算法的正确性与计算复杂性。

★编程任务:

对于给定的文件检索概率, 编程计算磁盘文件的最优存储方案。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行是正整数 n , 表示文件个数。第 2 行有 n 个正整数 a_i , 表示文件的检索概率。实际上第 k 个文件的检索概率应为 $a_k / \sum_{i=1}^n a_i$ 。

★结果输出:

将编程计算出的最小期望检索时间输出到文件 output.txt。

输入文件示例

输出文件示例

input.txt

output.txt

5

0.547396

33 55 22 11 9

分析与解答:

将 n 个文件按其概率排序。设排序后有 $p_1 \geq p_2 \geq \dots \geq p_n$ 。

贪心策略: f_1 占中心磁道, f_2 和 f_3 分居 f_1 的两侧, f_4 在 f_2 的左侧, f_5 在 f_3 的右侧, ……。

具体算法实现如下。

```
double greedy(vector<int> p)
{
    int n=p.size();
    vector<int> x(n,0);
    sort(p.begin(),p.end());
    int k=(n-1)/2;
    x[k]=p[n-1];
    for(int i=k+1;i<n;i++)
```

```

        x[i]=p[n-2*(i-k)];
    for(i=k-1;i>=0;i--)
        x[i]=p[n-2*(k-i)-1];
    double m=0,t=0;
    for (i=0;i<n;i++){
        m+=p[i];
        for(int j=i+1;j<n;j++){
            t+=x[i]*x[j]*(j-i);
        }
        t=t/m/m;
    }
    return t;
}

```

算法实现题 4-5 程序存储问题 (习题 4-8)

★问题描述:

设有 n 个程序 $\{1, 2, \dots, n\}$ 要存放在长度为 L 的磁带上。程序 i 存放在磁带上的长度是 $l_i, 1 \leq i \leq n$ 。

程序存储问题要求确定这 n 个程序在磁带上的一个存储方案,使得能够在磁带上存储尽可能多的程序。

★编程任务:

对于给定的 n 个程序存放在磁带上的长度,编程计算磁带上最多可以存储的程序数。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行是 2 个正整数,分别表示文件个数 n 和磁带的长度 L 。接下来的 1 行中,有 n 个正整数,表示程序存放在磁带上的长度。

★结果输出:

将编程计算出的最多可以存储的程序数输出到文件 output.txt。

输入文件示例

输出文件示例

input.txt

output.txt

6 50

5

2 3 13 8 80 20

分析与解答:

贪心策略:最短程序优先。

```

int greedy(vector<int> x, int m)
{
    int i=0, sum=0, n=x.size();
    sort(x.begin(), x.end());
    while(i<n){
        sum+=x[i];
        if(sum<=m) i++;
        else return i;
    }
}

```

```

    }
    return n;
}

```

算法实现题 4-6 最优服务次序问题 (习题 4-11)

★问题描述:

设有 n 个顾客同时等待一项服务。顾客 i 需要的服务时间为 $t_i, 1 \leq i \leq n$ 。应如何安排 n 个顾客的服务次序才能使平均等待时间达到最小? 平均等待时间是 n 个顾客等待服务时间的总和除以 n 。

★编程任务:

对于给定的 n 个顾客需要的服务时间, 编程计算最优服务次序。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行是正整数 n , 表示有 n 个顾客。接下来的 1 行中, 有 n 个正整数, 表示 n 个顾客需要的服务时间。

★结果输出:

将编程计算出的最小平均等待时间输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
10	532.00
56 12 1 99 1000 234 33 55 99 812	

分析与解答:

贪心策略: 最短服务时间优先。

```

double greedy(vector<int> x)
{
    int n=x.size ();
    sort(x.begin (),x.end ());
    for(int i=1;i<n;i++)
        x[i]+=x[i-1];
    double t=0;
    for(i=0;i<n;i++) t+=x[i];
    t/=n;
    return t;
}

```

算法实现题 4-7 多处最优服务次序问题 (习题 4-12)

★问题描述:

设有 n 个顾客同时等待一项服务。顾客 i 需要的服务时间为 $t_i, 1 \leq i \leq n$ 。共有 s 处可以提供此项服务。应如何安排 n 个顾客的服务次序才能使平均等待时间达到最小? 平均等待时间是 n 个顾客等待服务时间的总和除以 n 。

★编程任务:

对于给定的 n 个顾客需要的服务时间和 s 的值, 编程计算最优服务次序。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 s , 表示有 n 个顾客且有 s 处可以提供顾客需要的服务。接下来的 1 行中, 有 n 个正整数, 表示 n 个顾客需要的服务时间。

★结果输出:

将编程计算出的最小平均等待时间输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
10 2	336
56 12 1 99 1000 234 33 55 99 812	

分析与解答:

贪心策略: 最短服务时间优先。

```
double greedy(vector<int> x, int s)
{
    vector<int> st(s+1, 0);
    vector<int> su(s+1, 0);
    int n=x.size ();
    sort(x.begin (), x.end ());
    int i=0, j=0;
    while(i<n){
        st[j]+=x[i];
        su[j]+=st[j];
        i++;j++;
        if (j==s) j=0;
    }
    double t=0;
    for(i=0;i<s;i++) t+=su[i];
    t/=n;
    return t;
}
```

算法实现题 4-8 d 森林问题 (习题 4-14)

★问题描述:

设 T 是一棵带权树, 树的每一条边带一个正权。又设 S 是 T 的顶点集, T/S 是从树 T 中将 S 中顶点删去后得到的森林。如果 T/S 中所有树的从根到叶的路长都不超过 d , 则称 T/S 是一个 d 森林。

- (1) 设计一个算法求 T 的最小顶点集 S , 使 T/S 是 d 森林 (提示: 从叶向根移动)。
- (2) 分析算法的正确性和计算复杂性。
- (3) 设 T 中有 n 个顶点, 则算法的计算时间复杂性应为 $O(n)$ 。

★编程任务:

对于给定的带权树, 编程计算最小分离集 S 。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n , 表示给定的带权树有 n 个顶点, 编号为 $1, 2, \dots, n$ 。编号为 1 的顶点是树根。接下来的 n 行中, 第 $i+1$ 行描述与 i 个顶点相关联的边的信息。每行的第 1 个正整数 k 表示与该顶点相关联的边数。其后 $2k$ 个数中, 每 2 个数表示 1 条边。第 1 个数是与该顶点相关联的另一个顶点的编号, 第 2 个数是边权值。当 $k=0$ 时表示相应的结点是叶结点。文件的最后一行是正整数 d , 表示森林中所有树的从根到叶的路长都不超过 d 。

★结果输出:

将编程计算出的最小分离集 S 的顶点数输出到文件 output.txt。如果无法得到所要求的 d 森林则输出 “No Solution!”。

输入文件示例

输出文件示例

input.txt

output.txt

4

1

2 2 3 3 1

1 4 2

0

0

4

分析与解答:

用父亲数组 parent 表示树; leaf 存储叶结点编号; readin 读入初始数据。

```
void readin()
{
    fin >> n;
    for (int i=1; i<=n; i++) {
        fin >> deg[i];
        for (int j=0; j<deg[i]; j++) {
            fin >> p >> len;
            parent[p]=i;
            parlen[p]=len;
        }
        if (deg[i]==0) leaf[++leaf[0]]=i;
    }
    fin >> p;
}
```

从叶结点向根结点移动。从根结点到叶结点的路长超过 d 时, 将该子树分离。

```
int count()
{
```

```

for (int i=1, total=0; i<=leaf[0]; i++)
    if (leaf[i]!=1) //非根结点
    {
        int plen=parlen[leaf[i]], par=parent[leaf[i]];
        if (cut[par]<1 && dist[leaf[i]]+plen>p) {
            total++;
            cut[par]=1;
            par=parent[par];
        }
        else if (cut[par]<1 && dist[par]<dist[leaf[i]]+plen) dist[par]=
            dist[leaf[i]]+plen;
        if (--deg[par]==0) leaf[++leaf[0]]=par;
    }
return total;
}

```

算法实现题 4-9 汽车加油问题 (习题 4-16)

★问题描述:

一辆汽车加满油后可行驶 n km。旅途中有若干加油站。设计一个有效算法, 指出应在哪些加油站停靠加油, 使沿途加油次数最少。并证明算法能产生一个最优解。

★编程任务:

对于给定的 n 和 k 个加油站位置, 编程计算最少加油次数。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 k , 表示汽车加满油后可行驶 n km, 且旅途中有 k 个加油站。接下来的 1 行中, 有 $k+1$ 个整数, 表示第 k 个加油站与第 $k-1$ 个加油站之间的距离。第 0 个加油站表示出发地, 汽车已加满油。第 $k+1$ 个加油站表示目的地。

★结果输出:

将编程计算出的最少加油次数输出到文件 output.txt。如果无法到达目的地, 则输出 “No Solution”。

输入文件示例

input.txt

7 7

1 2 3 4 5 1 6 6

输出文件示例

output.txt

4

分析与解答:

贪心策略: 最远加油站优先。

```

int greedy(vector<int> x, int n)
{
    int sum=0, k=x.size();
    for (int j=0; j<k; j++)

```



```

        if(x[j]>n){
            cout<<"No Solution!"<<endl;
            return -1;
        }
        for(int i=0,s=0;i<k;i++){
            s+=x[i];
            if(s>n){sum++;s=x[i];}
        }
        return sum;
    }
}

```

算法实现题 4-10 区间覆盖问题 (习题 4-17)

★问题描述:

设 x_1, x_2, \dots, x_n 是实直线上的 n 个点。用固定长度的闭区间覆盖这 n 个点, 至少需要多少个这样的固定长度闭区间? 设计解此问题的有效算法, 并证明算法的正确性。

★编程任务:

对于给定的实直线上的 n 个点和闭区间的长度 k , 编程计算覆盖点集的最少区间数。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 k , 表示有 n 个点, 且固定长度闭区间的长度为 k 。接下来的 1 行中, 有 n 个整数, 表示 n 个点在实直线上的坐标 (可能相同)。

★结果输出:

将编程计算出的最少区间数输出到文件 output.txt。

输入文件示例

输出文件示例

input.txt

output.txt

7 3

3

1 2 3 4 5 -2 6

分析与解答:

贪心策略: 每次覆盖尽可能多的点。

```

int greedy(vector<int> x, int k)
{
    int sum=1, n=x.size();
    sort(x.begin(), x.end());
    for(int i=1, temp=x[0]; i<n; i++)
        if (x[i] - temp > k) {
            sum++; temp=x[i];
        }
    return sum;
}

```

算法实现题 4-11 硬币找钱问题 (习题 4-24)

★问题描述:

设有 6 种不同面值的硬币,各硬币的面值分别为 5 分、1 角、2 角、5 角、1 元和 2 元。现要用这些面值的硬币来购物和找钱。购物时可以使用的各种面值的硬币个数存于数组 `Coins[1:6]` 中,商店里各面值的硬币有足够多。在一次购物中希望使用最少硬币个数。

例如,一次购物需要付款 0.55 元,没有 5 角的硬币,只好用 $2 \times 20 + 10 + 5$ 共 4 枚硬币来付款。如果付出 1 元,找回 4 角 5 分,同样需要 4 枚硬币。但是如果付出 1.05 元 (1 枚 1 元和 1 枚 5 分),找回 5 角,只需要 3 枚硬币。这个方案用的硬币个数最少。

★编程任务:

对于给定的各种面值的硬币个数和付款金额,编程计算使用硬币个数最少的交易方案。

★数据输入:

由文件 `input.txt` 给出输入数据。每一行有 6 个整数和一个有 2 位小数的实数。分别表示可以使用的各种面值的硬币个数和付款金额。文件以 6 个 0 结束。

★结果输出:

将编程计算出的最少硬币个数输出到文件 `output.txt`。结果应分行输出,每行一个数据。如果不可能完成交易,则输出 “impossible”。

输入文件示例

输出文件示例

`input.txt`

`output.txt`

2 4 2 2 1 0 0.95

2

2 4 2 0 1 0 0.55

3

0 0 0 0 0 0

分析与解答:

见参考文献[1],76~77。

算法实现题 4-12 删数问题 (习题 4-25)

★问题描述:

给定 n 位正整数 a ,去掉其中任意 $k \leq n$ 个数字后,剩下的数字按原次序排列组成一个新的正整数。对于给定的 n 位正整数 a 和正整数 k ,设计一个算法找出剩下数字组成的新数最小的删数方案。

★编程任务:

对于给定的正整数 a ,编程计算删去 k 个数字后得到的最小数。

★数据输入:

由文件 `input.txt` 提供输入数据。文件的第 1 行是 1 个正整数 a 。第 2 行是正整数 k 。

★结果输出:

程序运行结束时,将计算出的最小数输出到文件 `output.txt`。

输入文件示例

输出文件示例

`input.txt`

`output.txt`

178543

13

4

分析与解答：

贪心策略：最近下降点优先。

```
void delele()
{
    int m=a.size();
    if (k>=m){a.erase();return;}
    while (k>0){
        for(int i=0; (i<a.size()-1) && (a[i]<=a[i+1]);i++);
        a.erase(i,1);k--;
    }
    while(a.size()>1 && a[0]=='0') a.erase(0,1);
}
```

算法实现题 4-13 数列极差问题 (习题 4-26)

★问题描述：

在黑板上写了 N 个正数组成的一个数列，进行如下操作：每一次擦去其中 2 个数，设为 a 和 b ，然后在数列中加入一个数 $a \cdot b + 1$ ，如此下去直至黑板上只留下一个数。在所有按这种操作方式最后得到的数中，最大的数记为 \max ，最小的数记为 \min ，则该数列的极差 M 定义为 $M = \max - \min$ 。

★编程任务：

对于给定的数列，编程计算出其极差 M 。

★数据输入：

由文件 input.txt 给出输入的数列，第 1 行是数列的长度 N (不超过 2000)，第 2 行起是数列中的 N 个数，相邻 2 个数由空格分隔。文件名由键盘输入。

★结果输出：

将编程计算出的数列极差 M 写入文件 output.txt。结果应分两行输出，第 1 行是数 M 的位数，第 2 行是数 M 。

输入文件示例

输出文件示例

input.txt

output.txt

3

1

1 1 1

0

分析与解答：

贪心策略：与 Huffman 算法类似。

算法实现题 4-14 嵌套箱问题 (习题 4-31)

★问题描述：

一个 d 维箱 (x_1, x_2, \dots, x_d) 嵌入另一个 d 维箱 (y_1, y_2, \dots, y_d) 是指存在 $1, 2, \dots, d$ 的一个排列 π ，使得 $x_{\pi(1)} < y_1, x_{\pi(2)} < y_2, \dots, x_{\pi(d)} < y_d$ 。

- (1) 证明上述箱嵌套关系具有传递性。
- (2) 试设计一个有效算法, 用于确定一个 d 维箱是否可嵌入另一个 d 维箱。
- (3) 给定由 n 个 d 维箱组成的集合 $\{B_1, B_2, \dots, B_n\}$, 试设计一个有效算法找出这 n 个 d 维箱中的一个最长嵌套箱序列, 并用 n 和 d 描述算法的计算时间复杂性。

★编程任务:

给定由 n 个 d 维箱, 试设计一个有效算法, 找出这 n 个 d 维箱中的一个最长嵌套箱序列。

★数据输入:

由文件 input.txt 提供输入数据。文件含多个测试数据项, 每个测试数据项的第 1 行中有 2 个整数 n 和 d , 分别表示箱的个数和维数。其后 n 行每行有 d 个正整数, 表示箱的各维的长度。

★结果输出:

程序运行结束时, 对每个测试数据项, 输出其最长嵌套箱序列的长度和从小到大排列的最长嵌套箱序列。所有结果输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5 2	5
3 7	3 1 2 4 5
8 10	4
5 2	7 2 5 6
9 11	
21 18	
8 6	
5 2 20 1 30 10	
23 15 7 9 11 3	
40 50 34 24 14 4	
9 10 11 12 13 14	
31 4 18 8 27 17	
44 32 13 19 41 19	
1 2 3 4 5 6	
80 37 47 18 21 9	

分析与解答:

见参考文献[1], 136~138。

算法实现题 4-15 套汇问题 (习题 4-32)

★问题描述:

套汇是指利用货币兑换率的差异将一个单位的某种货币转换为大于一个单位的同种货币。例如, 假定 1 美元可以买 0.7 英镑, 1 英镑可以买 9.5 法郎, 且 1 法郎可以买到 0.16 美元。通过货币兑换, 一个商人可以从 1 美元开始买入, 得到 $0.7 \times 9.5 \times 0.16 = 1.064$ 美元, 从而获得 6.4% 的利润。

★编程任务:

给定 n 种货币 c_1, c_2, \dots, c_n 的有关兑换率, 试设计一个有效算法, 用以确定是否存在套

汇的可能性。

★数据输入:

由文件 input.txt 提供输入数据。文件含多个测试数据项, 每个测试数据项的第 1 行中只有 1 个整数 n ($1 \leq n \leq 30$), 表示货币总数。其后 n 行给出 n 种货币的名称。接下来的一行中有 1 个整数 m , 表示有 m 种不同的货币兑换率。其后 m 行给出 m 种不同的货币兑换率, 每行有 3 个数据项 c_i, r_{ij} 和 c_j , 表示货币 c_i 和 c_j 的兑换率为 r_{ij} 。文件最后以数字 0 结束。

★结果输出:

程序运行结束时, 对每个测试数据项 j , 如果存在套汇的可能性则输出 “case j yes”, 否则输出 “case j no”。所有结果输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3	case 1 yes
USDollar	case 2 no
BritishPound	
FrenchFranc	
3	
USDollar 0.5 BritishPound	
BritishPound 10.0 FrenchFranc	
FrenchFranc 0.21 USDollar	
0	

分析与解答:

通过计算兑换率矩阵的传递闭包进行判断。算法主体如下。

```
while (1) {
    inFile>>n;
    if (n==0) break;
    for (i=0; i<n; i++) inFile>>name[i];
    for (i=0; i<n; i++)
        for (j=0; j<n; j++) r[i][j] = 0.0;
    inFile>>edges;
    for (i=0; i<edges; i++) {
        inFile>>a; inFile>>x; inFile>>b;
        for (j=0; strcmp(a, name[j]); j++);
        for (k=0; strcmp(b, name[k]); k++);
        r[j][k] = x;
    }
    for (i=0; i<n; i++) r[i][i] = max(1.0, r[i][i]);
    for (k=0; k<n; k++)
        for (i=0; i<n; i++)
            for (j=0; j<n; j++) r[i][j] = max(r[i][j], r[i][k]*r[k][j]);
    for (i=0; i<n; i++) if (r[i][i]>1.0) break;
    if (i<n) outFile<<"case "<<(cases)<<" yes"<<endl;
```

```

else outFile<<"case "<<(cases)<<" no"<<endl;
}

```

算法实现题 4-16 信号增强装置问题（习题 5-20）

★问题描述：

各种资源传输网络的功能是将始发地的资源通过网络传输到一个或多个目的地。例如，通过石油或天然气输送管网可以将油田开采的石油和天然气传送给消费者。同样，通过高压传输网络可以将发电厂生产的电力传送给用电消费者。为了使问题更具一般性，用术语信号统称网络中传输的资源（石油、天然气、电力等）。各种资源传输网络统称为信号传输网络。信号经信号传输网络传输时，需要消耗一定的能量，并导致传输能量的衰减（油压、气压、电压等）。当传输能量衰减量（压降）达到某个阈值时，将导致传输故障。为了保证传输畅通，必须在传输网络的适当位置放置信号增强装置，确保传输能量的衰减量不超过其衰减量容许值。

为了简化问题，假定给定的信号传输网络是以信号始发地为根的一棵树 T 。在树 T 的每一个结点处（除根结点外）可以放置一个信号增强装置。树 T 的结点也代表传输网络的消费结点。信号经过树 T 的结点传输到其儿子结点。树的每一边上的正权是流经该边的信号所发生的信号衰减量。信号衰减量是可加的。

信号增强装置问题要求对于一个给定的信号传输网络，计算如何放置最少的信号增强装置来保证网络传输的畅通。

★编程任务：

对于给定的带权树，编程计算放置信号增强装置的最少数量。

★数据输入：

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n ，表示给定的带权树有 n 个顶点，编号为 $1, 2, \dots, n$ ，编号为 1 的顶点是树根。在接下来的 n 行中，第 $i+1$ 行描述与 i 个顶点相关联的边的信息。每行的第 1 个正整数 k 表示与该顶点相关联的边数。其后 $2k$ 个数中，每 2 个数表示 1 条边。第 1 个数是与该顶点相关联的另一个顶点的编号，第 2 个数是边权值。文件的最后一行是正整数 d ，表示衰减量容许值。

★结果输出：

将编程计算出的最小信号增强装置数输出到文件 output.txt。如果无法得到满足要求的网络，则输出 “No Solution!”。

输入文件示例

input.txt

4

2 2 3 3 1

2 1 3 4 2

1 1 1

1 2 2

4

输出文件示例

output.txt

1

分析与解答：

与习题 4-14 解法相同。

算法实现题 4-17 磁带最大利用率问题 (习题 4-9)

★问题描述:

设有 n 个程序 $\{1, 2, \dots, n\}$ 要存放在长度为 L 的磁带上。程序 i 存放在磁带上的长度是 $l_i, 1 \leq i \leq n$ 。

程序存储问题要求确定这 n 个程序在磁带上的一个存储方案, 使得能够在磁带上存储尽可能多的程序。在保证存储最多程序的前提下还要求磁带的利用率达到最大。

★编程任务:

对于给定的 n 个程序存放在磁带上的长度, 编程计算磁带上最多可以存储的程序数和占用磁带的长度。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行是 2 个正整数, 分别表示文件个数 n 和磁带的长度 L 。接下来的 1 行中, 有 n 个正整数, 表示程序存放在磁带上的长度。

★结果输出:

将编程计算出的最多可以存储的程序数和占用磁带的长度以及存放在磁带上的每个程序的长度输出到文件 output.txt。第 1 行输出最多可以存储的程序数和占用磁带的长度; 第 2 行输出存放在磁带上的每个程序的长度。

输入文件示例

输出文件示例

input.txt

output.txt

9 50

5 49

2 3 13 8 80 20 21 22 23 2 3 13 8 23

分析与解答:

贪心策略: 最短程序优先。求得最多可以存储的程序个数 m 后, 再求最大利用率。问题转化为主教材第 5 章中的装载问题, 但 m 已知。对主教材第 5 章中的装载问题的解略做如下修改。

```
template<class T>
void Loading<T>::maxLoading(int i)
{
    if(i>n){
        if(xm==m){
            for(int j=1;j<=n;j++) bestx[j]=x[j];
            bestw=cw;
        }
        return;
    }
    r-=w[i];
    if(cw+w[i]<=c && xm<m){
        x[i]=1; xm++;
        cw+=w[i];
        maxLoading(i+1);
        cw-=w[i]; xm--;
    }
}
```

```

        if(cw+r>bestw){
            x[i]=0;
            maxLoading(i+1);
        }

        r-=w[i];
    }
}

```

算法实现题 4-18 非单位时间任务安排问题 (习题 4-15)

★问题描述:

具有截止时间和误时惩罚的任务安排问题可描述如下。

- (1) 给定 n 个任务的集合 $S = \{1, 2, \dots, n\}$;
- (2) 完成任务 i 需要 t_i 时间, $1 \leq i \leq n$;
- (3) 任务 i 的截止时间 d_i , $1 \leq i \leq n$, 即要求任务 i 在时间 d_i 之前结束;
- (4) 任务 i 的误时惩罚 w_i , $1 \leq i \leq n$, 即任务 i 未在时间 d_i 之前结束将招致 w_i 的惩罚; 若按时完成则无惩罚。

任务安排问题要求确定 S 的一个时间表 (最优时间表) 使得总误时惩罚达到最小。

★编程任务:

对于给定的 n 个任务, 编程计算总误时惩罚最小的最优时间表。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行是 1 个正整数 n , 表示任务数。接下来的 n 行中, 每行有 3 个正整数 a, b, c , 表示完成相应任务需要时间 a , 截止时间为 b , 误时惩罚为 c 。

★结果输出:

将编程计算出的总误时惩罚输出到文件 output.txt。

输入文件示例

input.txt

7

1 4 70

2 2 60

1 4 50

1 3 40

1 1 30

1 4 20

3 6 80

输出文件示例

output.txt

110

6

分析与解答:

首先将任务依其截止时间非减序排列。

设对任务 $1, 2, \dots, i$, 截止时间为 d 的最小误时惩罚为 $p(i, d)$, 则 $p(i, d)$ 具有最优子结构性质且满足如下递归式:

$$p(i,d)=\min\{p(i-1,d)+w(i),p(i-1,\min\{d,d_i\}-t_i)\}$$

$$p(1,d)=\begin{cases} 20 & t_1 \leq d \\ w(1) & t_1 > d \end{cases}$$

据此可设计解此问题的算法如下。

init 读入数据并进行初始化处理。

```

void init()
{
    cin>>n;
    tsk.resize(n);
    for (int i=0;i<n;i++){
        tsk[i].resize(3);
        cin>>tsk[i][0]>>tsk[i][1]>>tsk[i][2];
    }
    sort(tsk.begin(),tsk.end(),vless);
    d=tsk[n-1][1];
    f.resize(n,d+1);
    for(i=0;i<n;i++)
        for(int j=0;j<=d;j++)
            f[i][j]=INT_MAX;
}

```

dyna 进行动态规划计算。

```

void dyna()
{
    for(int i=0;i<=d;i++)
        if(tsk[0][0]<=i)f[0][i]=0;
        else f[0][i]=tsk[0][2];
    for(i=1;i<n;i++){
        for(int j=0;j<=d;j++){
            f[i][j]=f[i-1][j]+tsk[i][2];
            int jj=tsk[i][1]>j? j:tsk[i][1];
            if(jj>=tsk[i][0] && f[i][j]>f[i-1][jj-tsk[i][0]])
                f[i][j]=f[i-1][jj-tsk[i][0]];
        }
    }
}

```

实现算法的主函数如下。

```

void main()
{

```

```

init();
dyna();
cout<<f[n-1][d]<<endl;
}

```

算法所需的计算时间为 $O(n\log n + nd)$ 。其中, $d = \max_{1 \leq i \leq n} \{d_i\}$ 。

算法实现题 4-19 多元 Huffman 编码问题 (习题 4-20)

★问题描述:

在一个操场的四周摆放着 n 堆石子。现要将石子有次序地合并成一堆。规定每次至少选 2 堆最多选 k 堆石子合并成新的一堆, 合并的费用为新的一堆的石子数。试设计一个算法, 计算出将 n 堆石子合并成一堆的最大总费用和最小总费用。

★编程任务:

对于给定的 n 堆石子, 编程计算合并成一堆的最大总费用和最小总费用。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行有 2 个正整数 n 和 k , 表示有 n 堆石子, 每次至少选 2 堆最多选 k 堆石子合并。第 2 行有 n 个数, 分别表示每堆石子的个数。

★结果输出:

程序运行结束时, 将计算出的最大总费用和最小总费用输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
7 3	593 199
45 13 12 16 9 5 22	

分析与解答:

不妨设 $n \bmod (k-1) = 1$, 若不满足, 可增加若干 0。

贪心策略: 每次选最小的 k 个元素进行合并。与二元 Huffman 算法类似, 可证明其满足贪心选择性质。具体算法描述如下。

```

int tuffman(int a[], int n)
{
    MinHeap<int> H(1);
    H.Initialize(a, n, n);
    int i, j, m, x, t, sum;
    m = (k - n % (k - 1)) % (k - 1);
    for (i = 1, t = 0; i <= k - m; i++) {
        H.DeleteMin(x);
        t += x;
    }
    sum = t; n = (n - k + m) / (k - 1);
    H.Insert(t);
    for (i = 1; i <= n; i++) {
        for (j = 1, t = 0; j <= k; j++) {

```

```

        H.DeleteMin(x);t |=x;
    }
    sum+=t;H.Insert(t);
}
H.Deactivate();
return sum;
}

```

算法所需的计算时间为 $O(n \log_k n)$ 。

算法实现题 4-20 多元 Huffman 编码变形

★问题描述:

在一个操场的四周摆放着 n 堆石子。现要将石子有次序地合并成一堆。规定在合并过程中最多可以有 $m(k)$ 次选 k 堆石子合并成新的一堆, $2 \leq k \leq n$, 合并的费用为新的一堆的石子数。试设计一个算法, 计算出将 n 堆石子合并成一堆的最小总费用。

★编程任务:

对于给定 n 堆石子, 编程计算合并成一堆的最小总费用。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行有 1 个正整数 n , 表示有 n 堆石子。第 2 行有 n 个数, 分别表示每堆石子的个数。第 3 行有 $n-1$ 个数, 分别表示 $m(k)$ ($2 \leq k \leq n$) 的值。

★结果输出:

程序运行结束时, 将计算出的最小总费用输出到文件 output.txt。问题无解时输出 “No Solution!”

输入文件示例

输出文件示例

input.txt

output.txt

7

136

45 13 12 16 9 5 22

3 3 0 2 1 0

分析与解答:

首先找到向量 y 使

$$y = \max \left\{ y \leq m \mid \sum_{k=2}^n y(k) * (k-1) = n-1 \right\}$$

如果找不到, 则问题无解, 否则用向量 y 做贪心计算。

贪心策略: 每次选最小的 k , 作 $y(k)$ 次 k 个元素进行合并。与二元 Huffman 算法类似, 可证明其满足贪心选择性质。具体算法描述如下。

search 找向量 y 。

```

void search(int dep, int sum)
{
    if(dep==n-1) {
        if(sum==n-1) found=1;
    }
}

```

```

        return;
    }
    int ii=n/(n-dep-1);
    if(ii>c[n-dep])ii=c[n-dep];
    for(int i=ii;i>=0;i--){
        b[n-dep]=i;
        sum+=i*(n-dep-1);
        search(dep+1,sum);
        if(found) return;
        sum-=i*(n-dep-1);
    }
}

```

guffman 做贪心计算。

```

int guffman(int a[], int b[], int n)
{
    MinHeap<int> H(1);
    H.Initialize(a, n, n);
    int i, j, x, t, sum=0;
    for (i=2; i<=n; i++){
        for(k=1; k<=b[i]; k++){
            for(j=1, t=0; j<=i; j++){
                H.DeleteMin(x); t+=x;
            }
            sum+=t; H.Insert(t);
        }
    }
    H.Deactivate();
    return sum;
}

```

算法的主函数如下。

```

int main()
{
    cin>>n;
    a=new int[n+1];
    b=new int[n+1];
    c=new int[n+1];
    for(int i=1; i<=n; i++)cin>>a[i];
    for(i=2; i<=n; i++)cin>>c[i];
    for(i=0; i<=n; i++)b[i]=0; found=0;
    search(0, 0);
}

```

```

        if(!found)cout<<"No Solution!"<<endl;
        else cout<<guffman(a,b,n)<<endl;
        return 0;
    }

```

算法实现题 4-21 区间相交问题

★问题描述:

给定 x 轴上 n 个闭区间。去掉尽可能少的闭区间, 使剩下的闭区间都不相交。

★编程任务:

给定 n 个闭区间, 编程计算去掉的最少闭区间数。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行是正整数 n , 表示闭区间数。接下来的 n 行中, 每行有 2 个整数, 分别表示闭区间的 2 个整数端点。

★结果输出:

将计算出的去掉的最少闭区间数输出到文件 output.txt。

输入文件示例

输出文件示例

input.txt

output.txt

3

2

10 20

10 15

20 15

分析与解答:

与活动安排问题类似, 每次选取右端点坐标最小的闭区间, 保留该闭区间, 并将与其相交的闭区间删去。

算法实现题 4-22 任务时间表问题

★问题描述:

一个单位时间任务是恰好需要一个单位时间完成的任务。给定一个单位时间任务的有限集 S 。关于 S 的一个时间表用于描述 S 中单位时间任务的执行次序。时间表中第 1 个任务从时间 0 开始执行直至时间 1 结束, 第 2 个任务从时间 1 开始执行至时间 2 结束, ……第 n 个任务从时间 $n-1$ 开始执行直至时间 n 结束。

具有截止时间和误时惩罚的单位时间任务时间表问题可描述如下。

(1) n 个单位时间任务的集合 $S = \{1, 2, \dots, n\}$;

(2) 任务 i 的截止时间 $d_i, 1 \leq i \leq n, 1 \leq d_i \leq n$, 即要求任务 i 在时间 d_i 之前结束;

(3) 任务 i 的误时惩罚 $w_i, 1 \leq i \leq n$, 即任务 i 未在时间 d_i 之前结束将招致 w_i 的惩罚; 若按时完成则无惩罚。

任务时间表问题要求确定 S 的一个时间表 (最优时间表) 使得总误时惩罚达到最小。

★编程任务:

给定 n 个单位时间任务, 各任务的截止时间 d_i , 各任务的误时惩罚 $w_i, 1 \leq i \leq n$, 编程

计算最优时间表。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行是正整数 n , 表示任务数。接下来的 2 行中, 每行有 n 个正整数, 分别表示各任务的截止时间和误时惩罚。

★结果输出:

将计算出的最小总误时惩罚输出到文件 output.txt。

输入文件示例

输出文件示例

input.txt

output.txt

7

50

4 2 4 3 1 4 6

70 60 50 40 30 20 10

分析与解答:

见主教材。

算法实现题 4-23 最优分解问题

★问题描述:

设 n 是一个正整数。现在要求将 n 分解为若干互不相同的自然数的和, 且使这些自然数的乘积最大。

★编程任务:

对于给定的正整数 n , 编程计算最优分解方案。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行是正整数 n 。

★结果输出:

程序运行结束时, 将计算出的最大乘积输出到文件 output.txt。

输入文件示例

输出文件示例

input.txt

output.txt

10

30

分析与解答:

注意到, 若 $a+b=\text{const}$, 则 $|a-b|$ 越小, $a \cdot b$ 越大。

贪心策略: 将 n 分成从 2 开始的连续自然数的和。如果最后剩下一个数, 将此数在后项优先的方式下均匀地分给前面各项。

```
void dicomp()
```

```
{
```

```
    k=1;
```

```
    if(n<3) {a[1]=0;return;}
```

```
    if(n<5) {a[k]=1;a[++k]=n-1;return;}
```

```
    a[1]=2;n-=2;
```

```
    while(n>a[k]){
```

```
        k++;a[k]=a[k-1]+1;n-=a[k];
```

```

    }
    if(n==a[k]) {a[k]++;n--;}
    for(int i=0;i<n;i++)a[k-i]++;
}

```

算法实现题 4-24 可重复最优分解问题

★问题描述:

设 n 是一个正整数。现在要求将 n 分解为若干自然数的和, 且使这些自然数的乘积最大。

★编程任务:

对于给定的正整数 n , 编程计算最优分解方案。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行是正整数 n 。

★结果输出:

程序运行结束时, 将计算出的最大乘积输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
10	36

分析与解答:

注意到, 若 $a+b=\text{const}$, 则 $|a-b|$ 越小, $a \cdot b$ 越大。

贪心策略: 若 $n = \sum_{i=1}^k m_i$, 则 $-1 \leq m_i - m_j \leq 1$ 。

$$\max \prod_{i=1}^k m_i = \begin{cases} 3^{n/3} & n \equiv 0 \pmod{3} \\ 4 \times 3^{(n-4)/3} & n \equiv 1 \pmod{3} \\ 2 \times 3^{(n-2)/3} & n \equiv 2 \pmod{3} \end{cases}$$

```

void compute()
{
    int r=n%3;
    t[1]=1;k=n/3;
    length=1;
    if (r==1) {t[1]=4;k=(n-4)/3;}
    if (r==2) {t[1]=2;k=(n-2)/3;}
    for(int i=1;i<=k;i++) mult(3);
    for(i=length;i>0;i--) cout<<t[i];
    cout<<endl;
}

```

算法实现题 4-25 可重复最优组合分解问题

★问题描述:

对于任意正整数 m , 它的取 2 组合数定义为

$$\binom{m}{2} = m(m-1)/2$$

设 n 是一个正整数, 现在要求将 n 分解为若干自然数的和, 且使这些自然数的取 2 组合数的乘积最大。

★编程任务:

对于给定的正整数 n , 编程计算最优分解方案的最大取 2 组合数乘积的末尾有多少个 0。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行是正整数 n 。

★结果输出:

程序运行结束时, 将计算出的最大取 2 组合数乘积的末尾的 0 的个数输出到文件 output.txt。

输入文件示例

输出文件示例

input.txt

output.txt

5

1

分析与解答:

注意到, 若 $a+b=\text{const}$, 则 $|a-b|$ 越小, $a \cdot b$ 越大。

贪心策略: 若 $n = \sum_{i=1}^k m_i$, 则 $-1 \leq m_i - m_j \leq 1$ 。

设 $f(n) = \max \prod_{i=1}^k \binom{m_i}{2}$, 则

当 $n < 8$ 时, 有

$$f(n) = \binom{n}{2}$$

当 $n \geq 8$ 时, 有

$$f(n) = \begin{cases} 10^{n/5} & n \equiv 0 \pmod{5} \\ 15 \times 10^{(n-6)/5} & n \equiv 1 \pmod{5} \\ 225 \times 10^{(n-12)/5} & n \equiv 2 \pmod{5} \\ 36 \times 10^{(n-8)/5} & n \equiv 3 \pmod{5} \\ 6 \times 10^{(n-4)/5} & n \equiv 4 \pmod{5} \end{cases}$$

```

void compute()
{
    if (n < 8 && n != 5) {cout << 0 << endl; return;}
    int r = n % 5;
    if (r == 0) cout << n/5 << endl;
    if (r == 1) cout << (n-6)/5 << endl;
    if (r == 2) cout << (n-12)/5 << endl;
    if (r == 3) cout << (n-8)/5 << endl;
    if (r == 4) cout << (n-4)/5 << endl;
}

```

算法实现题 4-26 旅行规划问题

★问题描述:

G 先生想独自驾驶汽车从城市 A 到城市 B。从城市 A 到城市 B 的距离为 d_0 km。汽车油箱的容量为 c 公升。每公升汽油能行驶 e km。出发点每公升汽油的价格为 p 元。从城市 A 到城市 B 沿途有 n 个加油站。第 i 个加油站距出发点的距离为 d_i ，油价为每公升 p_i 元。如何规划才能使旅行的费用最省。

★编程任务:

对于给定的 d_0 , c , e , p , 和 n 以及 n 个加油站的距离和油价 d_i 和 p_i ，编程计算最小的旅行费用。如果无法到达目的地，则输出 “No Solution!”。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行是 d_0 , c , e , p 和 n 。接下来的 n 行中每行 2 个数 d_i 和 p_i 。

★结果输出:

程序运行结束时，将计算出的最小的旅行费用输出到文件 output.txt，精确到小数点后 2 位。

输入文件示例	输出文件示例
input.txt	output.txt
275.6 11.9 27.4 2.8 2	26.95
102.0 2.9	
220.0 2.2	

分析与解答:

贪心策略：找出从后向前油价最低的加油站，在此处加油。

算法实现题 4-27 登山机器人问题

★问题描述:

登山机器人是一个极富挑战性的高技术密集型科学研究项目，它为研究发展多智能体系统和多机器人之间的合作与对抗提供了生动的研究模型。

登山机器人可以携带有限的能量。在登山过程中，登山机器人需要消耗一定能量，连续攀登的路程越长，其攀登的速度就越慢。在对 n 种不同类型的机器人进行性能测试时，测定出每个机器人连续攀登 $1, 2, \dots, km$ 所用的时间。现在要对这 n 个机器人进行综合性能测试，举行机器人接力攀登演习。攀登的总高度为 m m。规定每个机器人只能攀登 1 次，每次至少攀登 1 m，最多攀登 km ，而且每个机器人攀登的高度必须是整数，即只能在整米处接力。安排每个机器人攀登适当的高度，使完成接力攀登用的时间最短。

★编程任务:

给定 n 个登山机器人接力攀登的总高度 m ，以及每个机器人连续攀登 $1, 2, \dots, km$ 所用的时间，编程计算最优攀登方案。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行是正整数 n , k 和 m 分别表示机器人的个数、每个机器人最多可以攀登的高度和攀登的总高度。接下来的 n 行中，每行有 k 个正整数，分别表示机器人连续攀登 $1, 2, \dots, km$ 所用的时间。

★结果输出:

将计算出的最短攀登时间输出到文件 output.txt。

输入文件示例

input.txt

5 10 25

24 49 75 102 130 160 192 230 270 320

23 48 75 103 139 181 224 274 344 415

22 49 80 180 280 380 480 580 680 780

25 51 80 120 170 220 270 320 370 420

23 49 79 118 158 200 250 300 350 400

输出文件示例

output.txt

727

分析与解答:

贪心策略: 每次选择所用时间最少的机器人。

第5章 回溯法

习题 5-1 装载问题改进回溯法 1

用主教材中提到的改进策略 1 重写装载问题回溯法, 使改进后算法计算时间复杂性为 $O(2^n)$ 。

分析与解答:

先运行只计算最优值的算法 `maxLoading1`, 计算出最优装载量 `bestw`。由于该算法不记录最优解, 故所需的计算时间为 $O(2^n)$ 。

```
template<class T>
void Loading<T>::maxLoading1(int i)
{
    if(i>n){bestw=cw;return;}
    r-=w[i];
    if(cw+w[i]<=c){cw+=w[i];maxLoading1(i+1);cw-=w[i];}
    if(cw+r>bestw)maxLoading1(i+1);
    r+=w[i];
}
```

然后运行改进后的算法 `maxLoading`, 在首次到达的叶结点处, 即首次遇到 $i>n$ 时终止算法。由此返回的 `bestx` 即为最优解。

```
template<class T>
void Loading<T>::maxLoading(int i)
{
    if(found)return;
    if(i>n){
        for (int j=1;j<=n;j++)bestx[j]=x[j];
        found=true;
        return;
    }
    r-=w[i];
    if(cw+w[i]<=c){
        x[i]=1;cw+=w[i];
        maxLoading(i+1);
        cw-=w[i];
    }
    if(cw+r>=bestw){x[i]=0;maxLoading(i+1);}
    r+=w[i];
}
```

习题 5-2 装载问题改进回溯法 2

用主教材中提到的改进策略 2 重写装载问题回溯法, 使改进后算法计算时间复杂性为 $O(2^n)$ 。

分析与解答:

在算法中动态地更新 bestx。在第 i 层的当前结点处, 当前最优解由 $x[j], 1 \leq j < i$ 和 $bestx[j], i \leq j \leq n$ 组成。每当算法回溯一层, 将 $x[i]$ 存入 $bestx[i]$ 。这样在每个结点处更新 bestx 只需 $O(1)$ 时间, 从而整个算法中更新 bestx 所需的时间为 $O(2^n)$ 。

```
template<class T>
void Loading<T>::maxLoading(int i)
{
    if(i>n) {ii=n;bestw=cw;return;}
    r-=w[i];
    if(cw+w[i]<=c) {
        x[i]=1;cw+=w[i];
        maxLoading(i+1);
        if(ii==i) {bestx[i]=1;ii--;}
        cw-=w[i];
    }
    if(cw+r>bestw) {
        x[i]=0;
        maxLoading(i+1);
        if(ii==i) {bestx[i]=0;ii--;}
    }
    r+=w[i];
}

template<class T>
T MaxLoading(T w[], T c, int n, int bestx[])
{
    Loading<T> X;
    X.x=new int [n+1];
    X.w=w;X.c=c;X.n=n;
    X.bestx=bestx;
    X.bestw=0;X.cw=0;X.ii=0;X.r=0;
    for (int i=1;i<=n;i++)X.r+=w[i];
    X.maxLoading(1);
    delete [] X.x;
    return X.bestw;
}
```

习题 5-4 0-1 背包问题的最优解

重写 0-1 背包问题的回溯法, 使算法运行结束后能输出最优解。

分析与解答:

为了构造最优解, 必须在算法中记录与当前最优值相应的当前最优解。为此, 在类 Knap 中增加 2 个私有数据成员 x 和 bestx。x 用于记录从根至当前结点的路径; bestx 记录当前最优解。算法搜索到达叶结点处, 就修正 bestx 的值。

修改后的算法描述如下。

```
template<class Typew, class Typep>
class Knap {
    friend Typep Knapsack(Typep*, Typew*, Typew, int, int[]);
private:
    Typep Bound(int i);
    void Backtrack(int i);
    Typew c;        // 背包容量
    int n;          // 物品数
    *x,             // 当前解
    *bestx;         // 当前最优解
    Typew *w;       // 物品重量数组
    Typep *p;       // 物品价值数组
    Typew cw;       // 当前重量
    Typep cp;       // 当前价值
    Typep bestp;    // 当前最优价值
};
```

在回溯过程中记录从根至当前结点的路径。

```
template<class Typew, class Typep>
void Knap<Typew, Typep>::Knapsack(int i)
{
    if(i>n) {
        for(int j=1; j<=n; j++) bestx[j]=x[j];
        bestp=cp;
        return;
    }
    if(cw+w[i]<=c) {
        x[i]=1; cw+=w[i]; cp+=p[i];
        Knapsack(i+1);
        cw-=w[i]; cp-=p[i]; x[i]=0;
    }
    if(Bound(i+1)>bestp) Knapsack(i+1);
}
```

Knapsack 作初始化, 并用回溯法求解。

```

template<class Typew, class Typep>
Typep Knapsack (Typep p[], Typew w[], Typew c, int n, int bestx[])
{
    Typew W=0;
    Typep P=0;
    Object *Q=new Object[n];
    for (int i=1; i<=n; i++) {
        Q[i-1].ID=i;
        Q[i-1].d=1.0*p[i]/w[i];
        P+=p[i];
        W+=w[i];
    }
    if (W<=c) return P;
    MergeSort (Q, n);
    Knap<Typew, Typep> K;
    K.p=new Typep[n+1];
    K.w=new Typew[n+1];
    K.x=new int[n+1];
    for (i=1; i<=n; i++) {
        K.p[i]=p[Q[i-1].ID];
        K.w[i]=w[Q[i-1].ID];
        K.x[i]=0;
    }
    K.cp=0; K.cw=0; K.c=c; K.n=n;
    K.bestp=0; K.bestx=bestx;
    K.Knapsack (1);
    for (i=1; i<=n; i++) K.x[i]=K.bestx[i];
    for (i=1; i<=n; i++) K.bestx[Q[i-1].ID]=K.x[i];
    delete []Q;
    delete []K.w;
    delete []K.p;
    delete []K.x;
    return K.bestp;
}

```

习题 5-5 最大团问题的迭代回溯法

试设计一个解最大团问题的迭代回溯法。

分析与解答：

与主教材中装载问题的迭代回溯法类似，最大团问题的迭代回溯法描述如下。

```

void Clique::iterClique()
{

```

```

for(int i=0;i<=n;i++)x[i]=0;
i=1;
while(true){
    while(i<=n && ok(i)) {x[i++]=1;cn++;}
    if(i>=n){
        for (int j=1;j<=n;j++)bestx[j]=x[j];
        bestn=cn;
    }
    else x[i++]=0;
    while(cn+n-i<=bestn){
        i--;
        while(i && !x[i])i--;
        if(i==0) return;
        x[i++]=0;cn--;
    }
}
}

```

其中, ok 用于判断当前顶点是否可加入当前团。

```

bool Clique::ok(int i)
{
    for(int j=1;j<i;j++)
        if(x[j] && a[i][j]==NoEdge) return false;
    return true;
}

```

IterClique 作初始化, 并调用迭代回溯法求解。

```

int Clique::IterClique(int v[])
{
    x=new int[n+1];
    cn=0;
    bestn=0;
    bestx=v;
    IterClique();
    delete [] x;
    return bestn;
}

```

习题 5-7 旅行售货员问题的费用上界

设 G 是有 n 个顶点的有向图, 从顶点 i 发出的边的最大费用记为 $\max(i)$ 。

(1) 证明旅行售货员回路的费用不超过 $\sum_{i=1}^n \max(i) + 1$ 。

(2) 在旅行售货员问题的回溯法中, 用上面的界作为 bestc 的初始值, 重写该算法, 并尽可能地简化代码。

分析与解答:

(1) 任一旅行售货员回路可表示为 n 个顶点的一个排列 $(\pi(1), \pi(2), \dots, \pi(n))$ 。这个回路的费用为

$$h(\pi) = \sum_{i=1}^n a(\pi(i), \pi(i \bmod n + 1))$$

由此可知,

$$\begin{aligned} h(\pi) &= \sum_{i=1}^n a(\pi(i), \pi(i \bmod n + 1)) \\ &\leq \sum_{i=1}^n \max(\pi(i)) \\ &= \sum_{i=1}^n \max(i) \\ &< \sum_{i=1}^n \max(i) + 1 \end{aligned}$$

(2) 对图 G 的简单遍历即可计算出 $\sum_{i=1}^n \max(i) + 1$ 的值。

```
template<class T>
T Traveling<T>::TSP1(int v[])
{
    bestc=1;
    for(int i=1, MaxCost=0; i<=n; i++){
        for(int j=1; j<=n; j++){
            if(a[i][j]!=NoEdge && a[i][j]>MaxCost) MaxCost=a[i][j];
            if(MaxCost==NoEdge) return NoEdge;
            bestc+=MaxCost;
        }
    }
    x=new int[n+1];
    for(i=1; i<=n; i++) x[i]=i;
    bestx=v; cc=0;
    tSP1(2);
    delete [] x;
    return bestc;
}
```

主教材的 TSP 回溯法中语句 bestc==NoEdge 可以删去, 修改如下。

```
template<class T>
void Traveling <T>::tSP1(int i)
{
    if(i==n){
```

```

        if(a[x[n-1]][x[n]]!=NoEdge && a[x[n]][1]!=NoEdge &&
            (cc+a[x[n-1]][x[n]]+a[x[n]][1]<bestc)){
            for(int j=1;j<=n;j++)bestx[j]=x[j];
            bestc=cc+a[x[n-1]][x[n]]+a[x[n]][1];
        }
    }
    else{
        for(int j=i;j<=n;j++){
            if(a[x[i-1]][x[j]]!=NoEdge && (cc+a[x[i-1]][x[j]]<bestc)){
                Swap(x[i], x[j]);
                cc+=a[x[i-1]][x[i]];
                tSP1(i+1);
                cc-=a[x[i-1]][x[i]];
                Swap(x[i], x[j]);
            }
        }
    }
}
}

```

习题 5-8 旅行售货员问题的上界函数

设 G 是有 n 个顶点的有向图, 从顶点 i 发出的边的最小费用记为 $\min(i)$ 。

(1) 证明图 G 的所有前缀为 $x[1:i]$ 的旅行售货员回路的费用至少为

$$\sum_{j=2}^i a(x_{j-1}, x_j) + \sum_{j=i}^n \min(x_j)$$

式中, $a(u, v)$ 是边 (u, v) 的费用。

(2) 利用上述结论设计一个高效的上界函数, 重写旅行售货员问题的回溯法, 并与主教材中的算法进行比较。

分析与解答:

(1) 前缀为 $x[1:i]$ 的旅行售货员回路任一旅行售货员回路可表示为 n 个顶点的一个排列 $(x[1], x[2], \dots, x[i], \pi(i+1), \pi(i+2), \dots, \pi(n))$ 。

这个回路的费用为

$$h(\pi) = \sum_{j=2}^i a(x_{j-1}, x_j) + a(x_i, \pi(i+1)) + \sum_{j=i+1}^n a(\pi(j), \pi(j \bmod n + 1))$$

由此可知

$$\begin{aligned}
 h(\pi) &\geq \sum_{j=2}^i a(x_{j-1}, x_j) + \min(x_i) + \sum_{j=i+1}^n \min(\pi(j)) \\
 &= \sum_{j=2}^i a(x_{j-1}, \pi_j) + \sum_{j=i}^n \min(x_j)
 \end{aligned}$$

(2) 先对图 G 简单遍历, 计算出 $\sum_{i=1}^n \min(i)$ 的值。

算法实现题 5-1 子集和问题 (习题 5-3)

★问题描述:

子集和问题的一个实例为 $\langle S, t \rangle$ 。其中, $S = \{x_1, x_2, \dots, x_n\}$ 是一个正整数的集合,

c 是一个正整数。子集和问题判定是否存在 S 的一个子集 S_1 , 使得 $\sum_{x \in S_1} x = c$ 。

试设计一个解子集和问题的回溯法。

★编程任务:

对于给定的正整数的集合 $S = \{x_1, x_2, \dots, x_n\}$ 和正整数 c , 编程计算 S 的一个子集 S_1 , 使得 $\sum_{x \in S_1} x = c$ 。

★数据输入:

由文件 input.txt 提供输入数据。文件第 1 行有 2 个正整数 n 和 c , n 表示 S 的大小, c 是子集和的目标值。接下来的 1 行中, 有 n 个正整数, 表示集合 S 中的元素。

★结果输出:

程序运行结束时, 将子集和问题的解输出到文件 output.txt。当问题无解时, 输出 “No Solution!”。

输入文件示例

输出文件示例

input.txt

output.txt

5 10

2 2 6

2 2 6 5 4

分析与解答:

与装载问题类似, 可设计解子集和问题的回溯法如下。

```
template<class T>
bool Subsum<T>::backtrack(int i)
{
    if (i > n) {
        for (int j = 1; j <= n; j++) bestx[j] = x[j];
        bestw = cw;
        if (bestw == c) return true;
        else return false;
    }
    r -= w[i];
    if (cw + w[i] <= c) {
        x[i] = 1; cw += w[i];
        if (backtrack(i+1)) return true;
        cw -= w[i];
    }
    if (cw + r > bestw) {
        x[i] = 0;
        if (backtrack(i+1)) return true;
    }
    r += w[i];
    return false;
}
```

算法实现题 5-2 最小长度电路板排列问题 (习题 5-9)

★问题描述:

最小长度电路板排列问题是大规模电子系统设计中提出的实际问题。该问题的提法是, 将 n 块电路板以最佳排列方案插入带有 n 个插槽的机箱中。 n 块电路板的不同的排列方式对应于不同的电路板插入方案。

设 $B = \{1, 2, \dots, n\}$ 是 n 块电路板的集合。集合 $L = \{N_1, N_2, \dots, N_m\}$ 是 n 块电路板的 m 个连接块。其中每个连接块 N_i 是 B 的一个子集, 且 N_i 中的电路板用同一根导线连接在一起。

例如, 设 $n=8, m=5$ 。给定 n 块电路板及其 m 个连接块如下:

$$B = \{1, 2, 3, 4, 5, 6, 7, 8\}; L = \{N_1, N_2, N_3, N_4, N_5\}$$

$$N_1 = \{4, 5, 6\}; N_2 = \{2, 3\}; N_3 = \{1, 3\}; N_4 = \{3, 6\}; N_5 = \{7, 8\}$$

这 8 块电路板的一个可能的排列如图 5-1 所示。

在最小长度电路板排列问题中, 连接块的长度是指该连接块中第 1 块电路板到最后 1 块电路板之间的距离。例如, 在图 5-1 所示的电路板排列中, 连接块 N_4 的第 1 块电路板在插槽 3 中, 它的最后 1 块电路板在插槽 6 中, 因此 N_4 的长度为 3。同理 N_2 的长度为 2。图 5-1 中连接块最大长度为 3。试设计一个回溯法

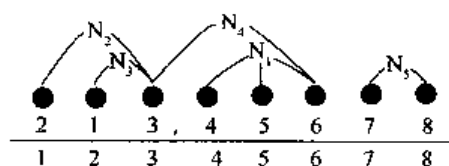


图 5-1 电路板排列

找出所给 n 个电路板的最佳排列, 使得 m 个连接块中最大长度达到最小。

★编程任务:

对于给定的电路板连接块, 设计一个算法, 找出所给 n 个电路板的最佳排列, 使得 m 个连接块中最大长度达到最小。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m ($1 \leq m, n \leq 20$)。接下来的 n 行中, 每行有 m 个数。第 k 行的第 j 个数为 0 表示电路板 k 不在连接块 j 中, 各 1 表示电路板 k 在连接块 j 中。

★结果输出:

将计算出的电路板排列最小长度及其最佳排列输出到文件 output.txt。文件的第一行是最小长度; 接下来的 1 行是最佳排列。

输入文件示例

输出文件示例

input.txt

output.txt

8 5

4

1 1 1 1 1

5 4 3 1 6 2 8 7

0 1 0 1 0

0 1 1 1 0

1 0 1 1 0

1 0 1 0 0

1 1 0 1 0

0 0 0 0 1

0 1 0 0 1

分析与解答:

与主教材中电路板排列问题类似, 可设计解最小长度电路板排列问题的回溯法如下。主要区别是计算连接块的长度, 由算法 len 完成。

```
class Board {
    friend ArrangeBoards(int **, int, int, int []);
private:
    void Backtrack(int i);
    int len(int ii);
    int *x, *bestx, *low, *high, bestd, n, m, **B;
};

int Board::len(int ii)
{
    for (int i=1; i<=m; i++) {high[i]=0;low[i]=n+1;}
    for (i=1; i<=ii; i++)
        for (int k=1; k<=m; k++)
            if(B[x[i]][k]){
                if(i<low[k]) low[k]=i;
                if(i>high[k]) high[k]=i;
            }
    int tmp=0;
    for (int k=1; k<=m; k++)
        if(low[k]<=n && high[k]>0 && tmp<high[k]-low[k]) tmp=high[k]-low[k];
    return tmp;
}
```

回溯法实体是 Backtrack。

```
void Board::Backtrack(int i)
{
    if (i==n) {
        int tmp=len(i);
        if(tmp<bestd){bestd=tmp;for (int j=1;j<=n;j++) bestx[j] = x[j];}
    }
    else
        for (int j = i; j <= n; j++) {
            Swap(x[i], x[j]);
            int ld=len(i);
            if (ld < bestd) Backtrack(i+1);
            Swap(x[i], x[j]);
        }
}
```

最后由 ArrangeBoards 完成计算。

```
int ArrangeBoards(int **B, int n, int m, int bestx[])
{
    Board X;
    X.x=new int[n+1];
    X.low=new int[m+1];
    X.high=new int[m+1];
    X.B=B;
    X.n=n;
    X.m=m;
    X.bestx=bestx;
    X.bestd=n+1;
    for (int i=1; i<=n; i++) X.x[i]=i;
    X.Backtrack(1);
    delete [] X.x;
    delete [] X.low;
    delete [] X.high;
    return X.bestd;
}
```

实现算法的主函数如下。

```
int main()
{
    int n,m,*p;
    fin>>n>>m;
    p=new int[n+1];
    int **B;
    Make2DArray(B,n+1,m+1);
    for (int i=1; i<=n; i++)
        for (int j=1; j<=m; j++) fin>> B[i][j];
    cout<< ArrangeBoards(B, n, m, p)<< endl;
    for (i=1; i<=n; i++) cout<< p[i]<<' '; cout<< endl;
    return 0;
}
```

算法实现题 5-3 最小重量机器设计问题 (习题 5-10)

★问题描述:

设某一机器由 n 个部件组成, 每一种部件都可以从 m 个不同的供应商处购得。设 w_{ij} 是从供应商 j 处购得的部件 i 的重量, c_{ij} 是相应的价格。

试设计一个算法, 给出总价格不超过 c 的最小重量机器设计。

★编程任务:

对于给定的机器部件重量和机器部件价格, 编程计算总价格不超过 d 的最小重量机器设计。

★数据输入:

由文件 input.txt 给出输入数据。第一行有 3 个正整数 n , m 和 d 。接下来的 $2n$ 行, 每行 n 个数。前 n 行是 c , 后 n 行是 w 。

★结果输出:

将计算出的最小重量, 以及每个部件的供应商输出到文件 output.txt。

输入文件示例

输出文件示例

input.txt

output.txt

3 3 4

4

1 2 3

1 3 1

3 2 1

2 2 2

1 2 3

3 2 1

2 2 2

分析与解答:

与背包问题类似, 可设计解最小重量机器设计问题的回溯法如下。

```
template<class Typew, class Typep>
bool Machine<Typew, Typep>::backtrack(int i)
{
    if (i > n) {
        bestw = cw;
        for(int j=1; j<=n; j++) bestx[j]=x[j];
        return true;}
    bool found=false;
    if(bestw<=cc) found=true;
    for(int j=1; j<=m; j++){
        x[i]=j;
        cw+=w[i][j];
        cp+=c[i][j];
        if (cp<= cc && cw<bestw) if(backtrack(i+1)) found=true;
        cw-=w[i][j];
        cp-=c[i][j];
    }
    return found;
}
```

算法实现题 5-4 运动员最佳配对问题 (习题 5-14)

★问题描述:

羽毛球队有男女运动员各 n 人。给定 2 个 $n \times n$ 矩阵 P 和 Q 。 $P[i][j]$ 是男运动员 i 和女

运动员 j 配对组成混合双打的男运动员竞赛优势; $Q[i][j]$ 是女运动员 i 和男运动员 j 配合的女运动员竞赛优势。由于技术配合和心理状态等各种因素影响, $P[i][j]$ 不一定等于 $Q[j][i]$ 。男运动员 i 和女运动员 j 配对组成混合双打的男女双方竞赛优势为 $P[i][j] * Q[j][i]$ 。设计一个算法, 计算男女运动员最佳配对法, 使各组男女双方竞赛优势的总和达到最大。

★编程任务:

设计一个算法, 对于给定的男女运动员竞赛优势, 计算男女运动员最佳配对法, 使各组男女双方竞赛优势的总和达到最大。

★数据输入:

由文件 input.txt 给出输入数据。第一行有 1 个正整数 n ($1 \leq n \leq 20$)。接下来的 $2n$ 行, 每行 n 个数。前 n 行是 p , 后 n 行是 q 。

★结果输出:

将计算出的男女双方竞赛优势的总和的最大值输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3	52
10 2 3	
2 3 4	
3 4 5	
2 2 2	
3 5 3	
4 5 1	

分析与解答:

此题的解空间显然是一棵排列树, 可以套用搜索排列树的回溯法框架。

```
void pref::Backtrack(int t)
{
    if (t > n) Compute();
    else
        for (int j = t; j <= n; j++) {
            swap(r[t], r[j]);
            Backtrack(t+1);
            swap(r[t], r[j]);
        }
}
```

其中, Compute 计算当前配对的竞赛优势的总和。

```
void pref::Compute(void)
{
    for (int i=1, temp=0; i<=n; i++)
        temp += p[i][r[i]] * q[r[i]][i];
    if (temp > best) {
```

```

        best=temp;
        for(int i=1;i<=n;i++) beststr[i]=r[i];
    }
}

```

算法实现题 5-5 无分隔符字典问题 (习题 5-15)

★问题描述:

设 $\Sigma = (\alpha_1, \alpha_2, \dots, \alpha_n)$ 是 n 个互不相同的符号组成的符号集。

$L_k = \{\beta_1\beta_2\cdots\beta_k \mid \beta_i \in \Sigma, 1 \leq i \leq k\}$ 是 Σ 中字符组成的长度为 k 的字符串全体。

$S \subseteq L_k$ 是 L_k 的 1 个无分隔符字典是指对任意 $a_1a_2\cdots a_k \in S$ 和 $b_1b_2\cdots b_k \in S$, 则

$$\{a_2a_3\cdots a_kb_1, a_3a_4\cdots b_1b_2, \dots, a_kb_1b_2\cdots b_{k-1} \cap S = \emptyset\}$$

无分隔符字典问题要求对给定的 n 和 Σ 以及正整数 k , 编程计算 L_k 的最大无分隔符字典。

★编程任务:

设计一个算法, 对于给定的正整数 n 和 k , 编程计算 L_k 的最大无分隔符字典。

★数据输入:

由文件 input.txt 给出输入数据。文件第 1 行有 2 个正整数 n 和 k 。

★结果输出:

将计算出的 L_k 的最大无分隔符字典的元素个数输出到文件 output.txt。

输入文件示例

输出文件示例

input.txt

output.txt

2 2

2

分析与解答:

用逐步加深的回溯法搜索解空间。

```

void search(int dep)
{
    if(dep>lk) {
        if(s.size()>best) {
            best=s.size();
            out();
        }
        return;
    }
    if(oka(dep)){
        s.insert(dep);
        search(dep+1);
        s.erase(dep);
    }
}

```



```

        search(dep+1);
    }

```

其中, `oka(dep)` 判断当前字符串 `dep` 是否可加入字典。本题中将字符串 $a_1a_2\cdots a_k$ 看作 k 位 n 进制数。当前字典中的字符串存储在集合 S 中。

```

bool oka(int b)
{
    for(it=s.begin();it!=s.end();it++)
        int a=*it;
        if (pref(a,b)) return false;
    }
    return true;
}

```

`pref(a,b)` 用于判断字符串 a 和 b 是否互不为前缀。

```

bool pref(int a, int b)
{
    int x=a, y=b/n;
    for(int i=0;i<k-1;i++){ak[k-i-2]=x%n;x/=n;ak[2*k-i-3]=y%n;y/=n;}
    for(i=1;i<k;i++) if(s.find(digi(i))!=s.end()) return true;
    x=b, y=a/n;
    for(i=0;i<k-1;i++){ak[k-i-2]=x%n;x/=n;ak[2*k-i-3]=y%n;y/=n;}
    for(i=1;i<k;i++) if(s.find(digi(i))!=s.end()) return true;
    return false;
}

```

`digi` 将相应字符串转换为 n 进制数。

```

int digi(int i)
{
    int ii=k+i-2;
    int x=ak[ii--];
    for(int j=0;j<k-1;j++){x*=n;x+=ak[ii];ii--;}
    return x;
}

```

`readin` 读入数据。

```

void readin()
{
    fin>>n>>k;
    ak=new int[2*k];
}

```

```

        lk=n;
        for(int i=1;i<=k;i++) lk*=n;
        lk--;best=0;
    }

```

实现算法的主函数如下。

```

int main()
{
    readin();
    if(k<3){cout<<n<<endl;return 0;}
    search(0);
    cout<<best<<endl;
    return 0;
}

```

算法实现题 5-6 无和集问题 (习题 5-16)

★问题描述:

设 S 是正整数集合。 S 是一个无和集当且仅当 $x, y \in S$ 蕴含 $x+y \notin S$ 。

对于任意正整数 k , 如果可将 $\{1, 2, \dots, k\}$ 划分为 n 个无和子集 S_1, S_2, \dots, S_n , 则称正整数 k 是 n 可分的。记 $F(n) = \max\{k | k \text{ 是 } n \text{ 可分的}\}$ 。试设计一个算法, 对任意给定的 n , 计算 $F(n)$ 的值。

★编程任务:

对任意给定的 n , 编程计算 $F(n)$ 的值。

★数据输入:

由文件 input.txt 给出输入数据。第一行有 1 个正整数 n 。

★结果输出:

将计算出的 $F(n)$ 的值以及 $\{1, 2, \dots, F(n)\}$ 的一个 n 划分输出到文件 output.txt。文件的第 1 行是 $F(n)$ 的值。接下来的 n 行, 每行是一个无和子集 S_i 。

输入文件示例

input.txt

2

输出文件示例

output.txt

8

1 2 4 8

3 5 6 7

分析与解答:

此题是子集选取问题, 其解空间显然是一棵子集树, 可以套用搜索子集树的回溯法框架。由于搜索空间很大, 用搜索时间控制搜索深度。

```

bool search(int dep)
{
    ti=clock();

```

```

elapsed += (t1 - t0) / ((double)CLOCKS_PER_SEC);
t0 = t1;
if (elapsed > 15.0) return false;
if (dep > k) {out(); return true;}
for (int i = 1; i <= n; i++) {
    if (sum[i][dep] == 0) {
        t[dep] = i; s[i][dep] = true;
        for (int j = 1; j < dep; j++) if (s[i][j]) sum[i][dep + j]++;
        if (search(dep + 1)) return true;
        s[i][dep] = false; t[dep] = 0;
        for (j = 1; j < dep; j++) if (s[i][j]) sum[i][dep + j]--;
    }
}
return false;
}

```

算法实现题 5-7 n 色方柱问题 (习题 5-17)

★问题描述:

设有 n 个立方体, 每个立方体的每一面用红、黄、蓝、绿等 n 种颜色之一染色。要把这 n 个立方体叠成一个方形柱体, 使得柱体的 4 个侧面的每一侧均有 n 种不同的颜色。试设计一个回溯算法, 计算出 n 个立方体的一种满足要求的叠置方案。

★编程任务:

对于给定的 n 个立方体以及每个立方体各面的颜色, 计算出 n 个立方体的一种叠置方案, 使得柱体的 4 个侧面的每一侧均有 n 种不同的颜色。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n , $0 < n < 27$, 表示给定的立方体个数和颜色数均为 n 。第 2 行是 n 个大写英文字母组成的字符串。该字符串的第 k ($0 \leq k < n$) 个字符代表第 k 种颜色。接下来的 n 行中, 每行有 6 个数, 表示立方体各面的颜色。立方体各面的编号如图 5-2 所示。

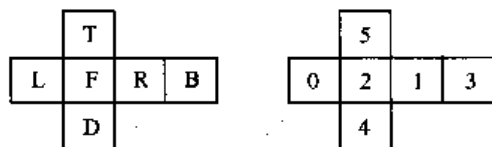


图 5-2 立方体各面的编号

图 5-2 中 F 表示前面, B 表示背面, L 表示左面, R 表示右面, T 表示顶面, D 表示底面。相应地, 2 表示前面, 3 表示背面, 0 表示左面, 1 表示右面, 5 表示顶面, 4 表示底面。

例如, 在示例输出文件中, 第 3 行的 6 个数 0 2 1 3 0 0 分别表示第 1 个立方体的左面的颜色为 R, 右面的颜色为 B, 前面的颜色为 G, 背面的颜色为 Y, 底面的颜色为 R, 顶面的颜色为 R。

★结果输出:

将计算出的 n 个立方体的一种可行的叠置方案输出到文件 output.txt。每行 6 个字符, 表示立方体各面的颜色。如果不存在所要求的叠置方案, 输出 “No Solution!”。

输入文件示例

input.txt

4

RGBY

0 2 1 3 0 0

3 0 2 1 0 1

2 1 0 2 1 3

1 3 3 0 2 2

输出文件示例

output.txt

RBGYRR

YRBGRG

BGRBGY

GYRBBB

分析与解答:

(1) 算法思想

每个立方体可以按 3 个方向旋转, 每个方向有 4 个不同的面, 因此每个立方体可有 64 种不同状态。用回溯法对 n 个立方体的每种状态进行搜索, 可以找到满足要求的叠置方案。然而, 这样做的计算量较大。下面讨论用图论的方法进行简化。在此问题中, 立方体的每对相对的面的颜色是要考察的关键因素。将每个立方体表示为有 n 个顶点的图。图中每个顶点表示一种颜色。在立方体每对相对面的顶点间连一条边。例如, 图 5-3 (b) 是图 5-3 (a) 所示的 4 个立方体所相应的子图。

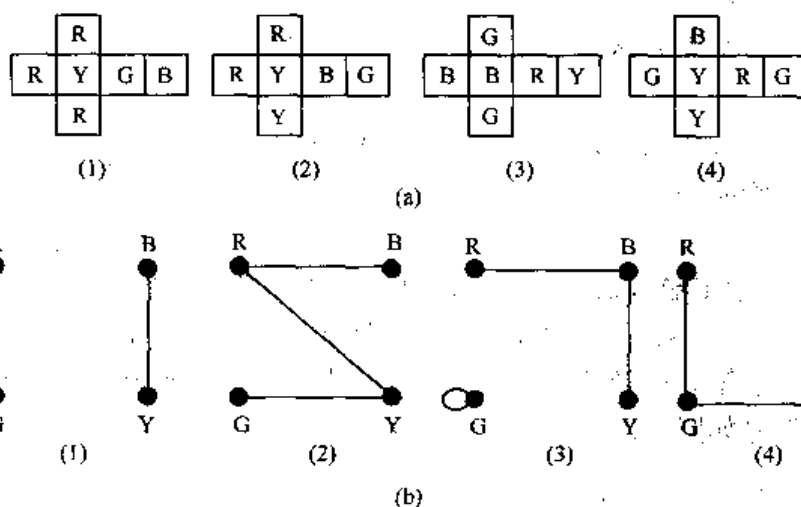


图 5-3 立方体及其相应的子图

将上述子图合并, 并标明每一条边来自哪一个立方体, 如图 5-4 所示。

下一步在构成的图中, 找出 2 个特殊子图。一个子图表示叠置的 n 个立方体的前侧面与背侧面, 另一子图表示叠置的 n 个立方体的左侧面与右侧面。这两个子图应满足下述性质。

- ① 每个子图有 n 条边, 且每个立方体恰好一条边。
- ② 2 个子图没有公共边。
- ③ 子图中每个顶点的度均为 2。

对于图 5-4 中的图, 找出满足要求的两个子图如图 5-5 所示。

给子图的每条边一个方向, 使每个顶点有一条出边和一条入边。有向边的始点对应于前面和左面; 有向边的终点对应于背面和右面。图 5-5 给出的满足要求的解如下。

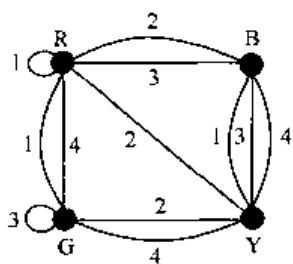


图 5-4 n 个立方体及其相应的子图

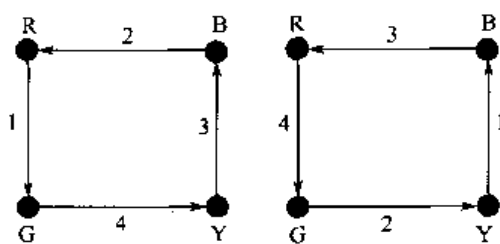


图 5-5 表示 n 个立方体 4 个侧面的子图

	0 (L)	1 (R)	2 (F)	3 (B)
Cube1	Y	B	G	R
Cube2	G	Y	R	B
Cube3	B	R	B	Y
Cube4	R	G	Y	G

上述算法的关键是找满足性质 (1)、(2) 和 (3) 的子图。用回溯法。

(2) 算法实现

二维数组 `board[n][6]` 存储 n 个立方体各面的颜色，`solu[n][6]` 存储解。

找满足性质 (1)、(2) 和 (3) 的子图的回溯法如下。

```
void search()
{
    int i, t, cube, newg, over, ok;
    int *vert = new int[n];
    int *edge = new int[n*2];
    for (i=0; i<n; i++) vert[i]=0;
    t=-1; newg=1;
    while(t>=-2) {
        t++;
        cube=t%n;           // 每个立方体找 2 次
        if(newg) edge[t]=-1;
        over=0; ok=0;
        while(!ok && !over) {
            edge[t]++;
            if(edge[t]>2) over=1; // 每个立方体只有 3 条边
            else ok=(t<n && edge[t]!=edge[cube]); // 是否已用过
        }
        if(!over) {
            if(++vert[board[cube][edge[t]*2]]>2+t/n*2) ok=0;
            if(++vert[board[cube][edge[t]*2+1]]>2+t/n*2) ok=0;
            if(t%n==n-1 && ok) //check that each vertex is order 2
                for (i=0; i<n; i++) if(vert[i]>2+t/n*2) ok=0;
            if(ok) {
                if (t==n*2-1) { // 找到解
                    ans++;
                }
            }
        }
    }
}
```

```

        out(edge);
        return;
    }
    else newg=1;
} //ok
else{ // 取下一条边
    --vert[board[cube][edge[t]*2]];
    --vert[board[cube][edge[t]*2+1]];
    t--;newg=0;
}
} //over
else{ // 回溯
    t--;
    if(t>-1){
        cube=t%n;
        --vert[board[cube][edge[t]*2]];
        --vert[board[cube][edge[t]*2+1]];
    }
    t--;newg=0;
}
}
}
}

```

找到一个解后由 out 输出。

```

void out(int edge[])
{
    int k, a, b, c, d;
    for(int i=0; i<2; i++){
        for (int j=0; j<n; j++) used[j]=0;
        do{
            j=0; // 找下一条未用边
            d=c-1;
            while (j<n && used[j]) j++;
            if(j<n)
                do{
                    a=board[j][edge[i*n+j]*2];
                    b=board[j][edge[i*n+j]*2+1];
                    if (b==d) {k=a; a=b; b=k;}
                    solu[j][i*2]=a;
                    solu[j][i*2+1]=b;
                    used[j]=1;
                    if (c<0) c=a; // 开始顶点
                    d=b;
                }
            }
        }
    }
}

```

```

        for (k=0; k<n; k++) // 找下一个立方体
            if (!used[k] && (board[k][edge[i*n+k]*2]==b ||
                            board[k][edge[i*n+k]*2+1]==b)) j=k;
        }while(b!=c);
    }while(j<n);
}
for (int j=0; j<n; j++){
    k=3-edge[j]-edge[j+n];
    a=board[j][k*2];
    b=board[j][k*2+1];
    solu[j][4]=a;
    solu[j][5]=b;
}
for (i=0; i<n; i++){
    for (j=0; j<6; j++){
        cout<<color[solu[i][j]];
        cout<<endl;
    }
}
}

```

执行算法的主函数如下。

```

int main()
{
    readin();
    search();
    if(ans==0)cout<<"No Solution!"<<endl;
    return 0;
}

```

初始数据由 readin 读入。

```

void readin()
{
    fin>>n;
    Make2DArray(board,n,6);
    Make2DArray(solu,n,6);
    color=new char[n];
    used=new int[n];
    for(int j=0;j<n;j++)fin>>color[j];
    for (int i=0;i<n;i++)
        for (int j=0;j<6;j++)fin>>board[i][j];
}

```

算法实现题 5-8 整数变换问题 (习题 5-18)

★问题描述:

关于整数 i 的变换 f 和 g 定义如下: $f(i)=3i$; $g(i)=\lfloor i/2 \rfloor$ 。

试设计一个算法, 对于给定的 2 个整数 n 和 m , 用最少的 f 和 g 变换次数将 n 变换为 m 。

例如, 可以将整数 15 用 4 次变换将它变换为整数 4: $4=gfgg(15)$ 。当整数 n 不可能变换为整数 m 时, 算法应如何处理?

★编程任务:

对任意给定的整数 n 和 m , 编程计算将整数 n 变换为整数 m 所需要的最少变换次数。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m 。

★结果输出:

将计算出的最少变换次数以及相应的变换序列输出到文件 output.txt。文件的第一行是最少变换次数。文件的第 2 行是相应的变换序列。

输入文件示例

input.txt

15 4

输出文件示例

output.txt

4

gfgg

分析与解答:

此题是 $3n+1$ 问题的变形。为了找最短变换序列, 用逐步加深的回溯法搜索。

```
void compute()
{
    k=1;
    while(! search(1,n)) {
        k++;
        if (k>maxdep) break;
        init();
    }
    if (found) output();
    else cout<<"No Solution!"<<endl;
}
```

search 实现回溯搜索。

```
bool search(int dep, int n)
{
    if(dep>k) return false;
    for(int i=0; i<2; i++){
        int n1=f(n, i); t[dep]=i;
        if(n1==m || search(dep+1, n1)) {found=true; out(); return true;}
    }
}
```



```
    return false;
}
```

算法实现题 5-9 拉丁矩阵问题

★问题描述:

现有 n 种不同形状的宝石, 每种宝石有足够多颗。欲将这些宝石排列成 m 行 n 列的一个矩阵, $m \leq n$, 使矩阵中每一行和每一列的宝石都没有相同形状。试设计一个算法, 计算出对于给定的 m 和 n , 有多少种不同的宝石排列方案。

★编程任务:

对于给定的 m 和 n , 计算出不同的宝石排列方案数。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 m 和 n , $0 < m \leq n < 9$ 。

★结果输出:

将计算出的宝石排列方案数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3 3	12

分析与解答:

(1) 算法思想

设 n 种宝石编号为 $1, 2, \dots, n$ 。宝石矩阵的第 1 行从左到右排列为 $1, 2, \dots, n$, 且第 1 列从上到下排列为 $1, 2, \dots, m$ 的阵列为标准拉丁矩阵。设 m 行 n 列的标准拉丁矩阵个数为 $L(m, n)$ 。一般情况下, m 行 n 列的拉丁矩阵个数为 $R(m, n)$ 。本题要求 $R(m, n)$ 。

容易证明, $R(m, n) = n! (n-1)! L(m, n) / (n-m)!$ 。于是问题可转化为求标准拉丁矩阵个数 $L(m, n)$ 。问题显然与排列有关, 可用主教材中的排列树回溯法框架求解。

(2) 算法实现

二维数组 `board[m][n]` 存储宝石矩阵。每行初始化为单位排列, 第 1 列从上到下排列为 $1, 2, \dots, m$ 。

```
void init()
{
    fin >> m >> n;
    cout << m << " " << n << endl;
    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= n; ++j) board[i][j] = j;
    for (i = 2; i <= n; ++i) Swap(board[i][1], board[i][i]);
    if (m == n) m--;
}
```

对数组 `board` 从上到下、从左到右递归搜索。

```
void backtrack(int r, int c)
```

```

{
    for(int i=c;i<=n;i++)
        if(ok(r,c,board[r][i])){
            Swap(board[r][c],board[r][i]);
            if(c==n){
                if(r==m)count+=1.0;
                else backtrack(r+1,2);
            }
            else backtrack(r,c+1);
            Swap(board[r][c],board[r][i]);
        }
}

```

其中，ok 用于判断在当前列中宝石是否重复。

```

int ok(int r, int c, int k)
{
    for (int i=1;i<r;i++)if(board[i][c]==k)return 0;
    return 1;
}

```

执行算法的主函数如下。

```

int main()
{
    init();
    backtrack(2,2);
    outlong(count);
    return 0;
}

```

其中，outlong 按公式 $R(m,n)$ 计算输出 $R(m,n)=n!(n-1)!L(m,n)/(n-m)!$ 的值。由于输出的值较大，这一步需要高精度计算。

注意到，当 $m=n$ 时，第 $n-1$ 行排定后，第 n 行就已确定，无须回溯。这就是 init 中的语句 “if($m==n$) $m--$ ” 的含义。当然还有其他的优化方法。

算法实现题 5-10 排列宝石问题（习题 5-19）

★问题描述：

现有 n 种不同形状的宝石，每种 n 颗，共 n^2 颗。同一种形状的 n 颗宝石分别具有 n 种不同的颜色 c_1, c_2, \dots, c_n 中的一种颜色。欲将这 n^2 颗宝石排列成 n 行 n 列的一个方阵，使方阵中每一行和每一列的宝石都有 n 种不同形状和 n 种不同颜色。试设计一个算法，计算出对于给定的 n ，有多少种不同的宝石排列方案。

★编程任务：

对于给定的 n ，计算出不同的宝石排列方案数。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 $n, 0 < n < 9$ 。

★结果输出:

将计算出的宝石排列方案数输出到文件 output.txt。

输入文件示例

输出文件示例

input.txt

output.txt

1

1

分析与解答:

(1) 算法思想

此题与上一题类似, 用回溯法时, 需对形状和颜色 2 种因素循环考察。

(2) 算法实现

二维数组 $a[n][n]$, $b[n][n]$ 分别存储宝石形状矩阵和颜色矩阵。每行初始化为单位排列。二维数组 $cc[n][n]$ 的单元 $cc[i][j]$ 用于搜索时记录形状为 i , 颜色为 j 的宝石是否已用过, 初始化为 0。

```
void init()
{
    fin >> n;
    for(int i=1; i<=n; ++i)
        for(int j=1; j<=n; ++j) {a[i][j]=j; b[i][j]=j; cc[i][j]=0;}
}
```

对数组 a 和 b 从上到下、从左到右递归搜索。

```
void backtrack(int r, int c)
{
    for(int i=c; i<=n; i++)
        if(ok(r, c, i, 0)) {
            Swap(a[r][c], a[r][i]);
            for(int j=c; j<=n; j++)
                if(ok(r, c, j, 1)) {
                    Swap(b[r][c], b[r][j]);
                    cc[a[r][c]][b[r][c]]=1;
                    if(c==n) {
                        if(r==n) count += 1.0;
                        else backtrack(r+1, 1);
                    }
                    else backtrack(r, c+1);
                    cc[a[r][c]][b[r][c]]=0;
                    Swap(b[r][c], b[r][j]);
                }
            Swap(a[r][c], a[r][i]);
        }
}
```

其中, ok 用于判断在当前列中宝石的形状和颜色是否重复。

```
int ok(int r, int c, int k, int fla)
{
    if(fla){
        if(cc[a[r][c]][b[r][k]])return 0;
        for(int i=1;i<r;i++)if(b[i][c]==b[r][k])return 0;
    }
    else for(int i=1;i<r;i++)if(a[i][c]==a[r][k])return 0;
    return 1;
}
```

执行算法的主函数如下。

```
int main()
{
    init();
    backtrack(1, 1);
    cout<<(int)count<<endl;
    return 0;
}
```

与上一题一样, 注意到, 第 $n-1$ 行排定后, 第 n 行就已确定, 无须回溯, 但必须判断是否矛盾。这个任务可由 last 完成如下。

```
int last()
{
    for (int j=1;j<=n;j++){
        for (int i=1;i<=n;i++){dd[i][0]=0;dd[i][1]=0;}
        for (i=1;i<=n;i++){dd[a[i][j]][0]=1;dd[b[i][j]][1]=1;}
        for (i=1;i<=n;i++){if(dd[i][0]==0)ee[j][0]=i;if(dd[i][1]==0)ee[j][1]=i;}
    }
    for(int i=1;i<=n;i++)if(cc[ee[i][0]][ee[i][1]])return 0;
    return 1;
}
```

最后, 将回溯法中的语句 `if(r==n)count+=1.0;` 换成 `if(r==n-1){if(last())count+=1.0;}`。

算法实现题 5-11 重复拉丁矩阵问题 (习题 5-19)

★问题描述:

现有 k 种不同价值的宝石, 每种宝石都有足够多颗。欲将这些宝石排列成一个 m 行 n 列的矩阵, $m \leq n$, 使矩阵中每一行和每一列的同一种宝石数都不超过规定的数量。另外还规定, 宝石阵列的第 1 行从左到右和第 1 列从上到下的宝石按宝石的价值最小字典序从小到大

大排列。试设计一个算法,对于给定的 k , m 和 n 以及每种宝石的规定数量,计算出有多少种不同的宝石排列方案。

★编程任务:

对于给定的 m , n 和 k , 以及每种宝石的规定数量,计算出不同的宝石排列方案数。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 3 个正整数 m , n 和 k , $0 < m \leq n < 9$ 。第 2 行有 k 个数,第 j 个数表示第 j 种宝石在矩阵的每行和每列出现的最多次数。这 k 个数按照宝石的价值从小到大排列。设这 k 个数为 $1 \leq v_1 \leq v_2 \leq \dots \leq v_k$, 则 $v_1 + v_2 + \dots + v_k = n$ 。

★结果输出:

将计算出的宝石排列方案数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
4 7 3	84309
2 2 3	

分析与解答:

(1) 算法思想

此题与前两题类似,用回溯法时,需考察相同价值的情况。

(2) 算法实现

二维数组 $\text{board}[m][n]$ 存储宝石矩阵。每行初始化为单位排列,第 1 列从上到下排列为 $1, 2, \dots, m$ 。用数组 mv 记录规定的每种宝石的重复数, mu 记录 n 个宝石中,每个宝石的价值序号。由 init 初始化各数组。

```
void init()
{
    fin >> m >> n >> mm;
    for(int k=1, j=1, t=0; k<=mm; k++){
        fin >> t; mv[k]=t;
        while(t){mu[j++]=k; t--;}
    }
    for(int i=1; i<=n; ++i)
        for(int j=1; j<=m; ++j) board[i][j]=j;
    for(i=2; i<=n; ++i) Swap(board[i][1], board[i][i]);
}
```

对数组 board 从上到下,从左到右递归搜索。

```
void backtrack(int r, int c)
{
    for(int i=c; i<=n; i++){
        if(ok(r, c, i)){
            Swap(board[r][c], board[r][i]);
```

```

        if(c==n){
            if(r==m)count+=1.0;
            else backtrack(r+1,2);
        }
        else backtrack(r,c+1);
        Swap(board[r][c],board[r][i]);
    }
}

```

其中, ok 用于判断在当前列中宝石是否超过规定数。

```

int ok(int r,int c,int s)
{
    int k=board[r][s];
    if(s>c) for(int t=c;t<s;t++)if(mu[board[r][t]]==mu[k])return 0;
    for (int i=1,j=0;i<r;i++)if(mu[board[i][c]]==mu[k])j++;
    if(j>mv[mu[k]]-1)return 0;
    else return 1;
}

```

执行算法的主函数如下。

```

int main()
{
    init();
    backtrack(2,2);
    cout<<(int)count<<endl;
    return 0;
}

```

算法实现题 5-12 罗密欧与朱丽叶的迷宫问题 (习题5-21)

★问题描述:

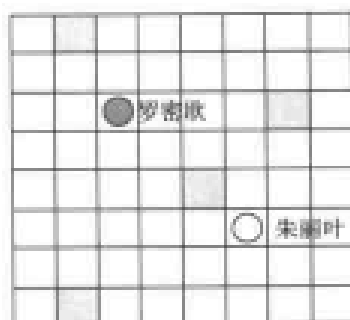


图 5-6 罗密欧与朱丽叶的迷宫

罗密欧与朱丽叶身处一个 $m \times n$ 的迷宫中, 如图 5-6 所示。每一个方格表示迷宫中的一个房间。这 $m \times n$ 个房间中有一些房间是封闭的, 不允许任何人进入。在迷宫中任何位置均可沿 8 个方向进入未封闭的房间。罗密欧位于迷宫的 (p, q) 方格中, 他必须找出一条通向朱丽叶所在的 (r, s) 方格的路。在抵达朱丽叶之前, 他必须走遍所有未封闭的房间各一次, 而且要使到达朱丽叶的转弯次数为最少。每改变一次前进方向算作转弯一次。请设计一个算法帮助罗密欧找出这样一条道路。

★编程任务:

对于给定的罗密欧与朱丽叶的迷宫,编程计算罗密欧通向朱丽叶的所有最少转弯道路。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 3 个正整数 n, m, k , 分别表示迷宫的行数, 列数和封闭的房间数。接下来的 k 行中, 每行 2 个正整数, 表示被封闭的房间所在的行号和列号。最后的 2 行, 每行也有 2 个正整数, 分别表示罗密欧所处的方格 (p, q) 和朱丽叶所处的方格 (r, s) 。

★结果输出:

将计算出的罗密欧通向朱丽叶的最少转弯次数和有多少条不同的最少转弯道路输出到文件 output.txt。文件的第一行是最少转弯次数。文件的第 2 行是不同的最少转弯道路数。接下来的 n 行每行 m 个数, 表示迷宫的一条最少转弯道路。 $A[i][j]=k$ 表示第 k 步到达方格 (i, j) ; $A[i][j]=-1$ 表示方格 (i, j) 是封闭的。

如果罗密欧无法通向朱丽叶则输出 “No Solution!”。

输入文件示例	输出文件示例
input.txt	output.txt
3 4 2	6
1 2	7
3 4	1 -1 9 8
1 1	2 10 6 7
2 2	3 4 5 -1

分析与解答:

在当前位置按照 8 个方向搜索。

```
void search (int dep, int x, int y, int di)
{
    if (dep==m*n-k && x==xl && y==yl && dirs<=best){
        if(dirs<best) {best=dirs;count=1;save();}
        else count++;
        return;
    }
    if (dep==m*n-k || x==xl && y==yl || dirs>best) return;
    else
        for (int i = 1; i <= 8; i++)
            if (stepok(x+dx[i],y+dy[i])) {
                board[x+dx[i]][y+dy[i]]=dep+1;
                if(di!=i) dirs++;
                search(dep+1,x+dx[i],y+dy[i],i);
                if(di!=i) dirs--;
                board[x+dx[i]][y+dy[i]]=0;
            }
}
```

stepok 用于判断是否越界。

```
bool stepok(int x, int y)
{
    return (x>0 && x<=n && y>0 && y<=m && board[x][y]==0);
}
```

save 保存找到的解。

```
void save()
{
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            bestb[i][j]=board[i][j];
}
```

对于当前位置还可加入剪枝函数 live 提早判断无解, 进行剪枝。

```
bool live(int x, int y, int dep)
{
    int nm=n*m;
    if(d[x][y]>1 && endpoint>1) return false;
    for(int j=1;j<=8;j++){
        int p=x+dx[j], q=y+dy[j];
        if(stepok(p, q) && d[p][q]<2 && dep<nm-k-2) return false;
    }
    return true;
}
```

加入剪枝函数后的回溯法如下。

```
void search (int dep, int x, int y, int di)
{
    if (dep==m*n-k && x==x1 && y==y1 && dirs<=best){
        if(dirs<best){best=dirs;count=1;save();}
        else count++;
        return;
    }
    if (dep==m*n-k || x==x1 && y==y1 || dirs>best) return;
    else
        for (int i = 1; i <= 8; i++){
            int p=x+dx[i], q=y+dy[i];
            if (stepok(p, q) && live(p, q, dep)) {
                save(p, q, dep);
            }
        }
}
```

```

        if(di!= i) dirs++;
        search(dep+1,p,q,i);
        if(di!= i) dirs--;
        restore(p,q);
    }
}
}

```

算法实现题 5-13 工作分配问题 (习题 5-22)

★问题描述:

设有 n 件工作分配给 n 个人。将工作 i 分配给第 j 个人所需的费用为 c_{ij} 。试设计一个算法, 为每一个人都分配 1 件不同的工作, 并使总费用达到最小。

★编程任务:

设计一个算法, 对于给定的工作费用, 计算最佳工作分配方案, 使总费用达到最小。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n ($1 \leq n \leq 20$)。接下来的 n 行, 每行 n 个数, 表示工作费用。

★结果输出:

将计算出的最小总费用输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3	9
10 2 3	
2 3 4	
3 4 5	

分析与解答:

此题的解空间显然是一棵排列树, 可以套用搜索排列树的回溯法框架。

```

void job::Backtrack(int t)
{
    if (t>n) Compute();
    else
        for (int j = t; j <= n; j++) {
            swap(r[t], r[j]);
            Backtrack(t+1);
            swap(r[t], r[j]);
        }
}

```

其中, Compute 计算当前方案的费用。

```

void job::Compute(void)
{
    for (int i=1,temp=0;i<=n;i++)
        temp+=p[i][r[i]];
    if (temp<best){
        best=temp;
        for(int i=1;i<=n;i++) bestr[i]=r[i];
    }
}

```

算法实现题 5-14 独立钻石跳棋问题 (习题 5-23)

★问题描述:

独立钻石跳棋的棋盘上有 33 个方格,如图 5-7 所示,每个方格中可放 1 枚棋子,棋盘中最多可摆放 32 枚棋子。下棋的规则是任一棋子可以沿水平或垂直方向跳过与其相邻的棋子进入空着的方格并吃掉被跳过的棋子。试设计一个算法,对于任意给定的棋盘布局,找出一种下棋方法,使得最终棋盘上只剩下一个棋子。

★编程任务:

对于给定的独立钻石跳棋的棋盘初始布局,如图 5-7 所示,和棋盘上最终剩下的棋子所在的位置 (x,y) ,编程计算一种遵循下棋的规则下棋方法,使最终棋盘上仅在位置 (x,y) 处有一枚棋子。当 $(x,y)=(0,0)$ 时,表示不指定棋子的最终位置。棋子位置的坐标定义如图 5-8 所示。

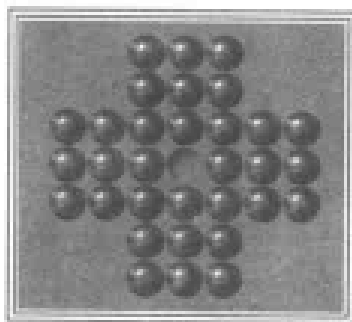


图 5-7 独立钻石跳棋初始布局

		(2,0)	(3,0)	(4,0)		
		(2,1)	(3,1)	(4,1)		
(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)
(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)
(0,4)	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)
		(2,5)	(3,5)	(4,5)		
		(2,6)	(3,6)	(4,6)		

图 5-8 棋子位置的坐标定义

★数据输入:

由文件 input.txt 提供输入数据,文件的第 1 行中有 1 个正整数 n ,表示棋盘的初始布局中有 n 个棋子。第 2 行起每行 2 个数,分别是 n 个棋子的位置。最后 1 行是棋盘上最终剩下的棋子所在的位置。

★结果输出:

程序运行结束时,将计算出的下棋步法依次输出到文件 output.txt 中。每行有 2 对方格坐标 (a,b) 和 (c,d) ,表示从方格 (a,b) 跳到方格 (c,d) 。问题无解时输出 “No Solution!”。

输入文件示例	输出文件示例
input.txt	output.txt
8	(3,4) (5,4)
4 1	(4,6) (4,4)
5 2	(4,4) (6,4)
5 3	(6,4) (6,2)
6 3	(6,2) (4,2)
3 4	(4,1) (4,3)
4 4	(5,3) (3,3)
4 5	
4 6	
0 0	

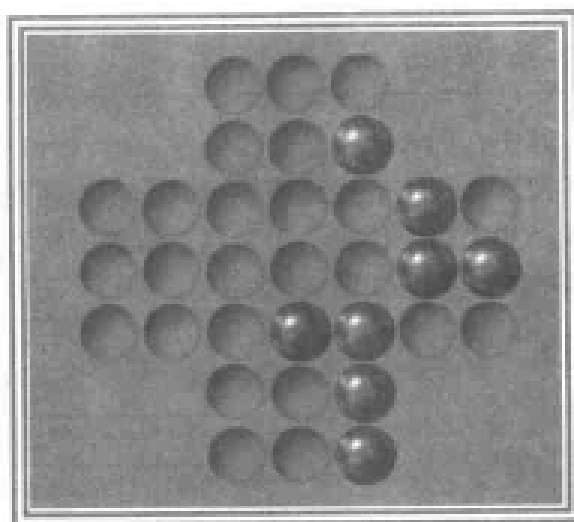


图 5-9 棋盘的一个初始布局

分析与解答:

(1) 算法思想

解独立钻石跳棋问题的回溯算法框架如下。

```
int backtrack(int t)
{
    if(finish(t)) return 1;
    for each step m{
        if(ok(m)){
            next(m);
            if(backtrack(t+1)) return 1;
            restore(m);
        }
    }
    return 0;
}
```

(2) 基本数据结构

用 $board[7][7]$ 表示棋盘, $board[x][y]=0$, 表示空方格; $board[x][y]=1$, 表示棋子占用方格; $board[x][y]=2$, 表示棋盘外方格。初始棋盘为空棋盘, 根据输入数据填入棋子。

```
void init()
{
    for(int i=0;i<7;i++)
        for(int j=0;j<7;j++)
            if((i<2 || i>4) && (j<2 || j>4)) board[i][j]=2;
            else board[i][j]=0;
}
```

用结构 Move 表示棋子的走步。

```
typedef struct{
```

```

int sx, sy, xv, yv;
}Move;

```

其中, (sx,sy) 是起步坐标, (xv,yv) 是走步方向。

搜索过程将产生许多棋盘状态。问题是如何有效地存储这些棋盘状态。棋盘共有 33 个位置。如果用 1 位表示 1 个棋盘位置的状态, 则一个棋盘需用 33 位来表示。每一位与棋盘位置的对应关系如图 5-10 所示。

		(2,0)	(3,0)	(4,0)		
		(2,1)	(3,1)	(4,1)		
(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)
(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)
(0,4)	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)
		(2,5)	(3,5)	(4,5)		
		(2,6)	(3,6)	(4,6)		

图 5-10 用 33 位来表示棋盘

函数 coord2bit 将棋盘坐标转换为相应的位。

```

BoardType coord2bit(int x, int y)
{
    int pos;
    switch(y) {
        case 0:    pos=x-2;break;
        case 1:    pos=x+1;break;
        case 5:    pos=x+25;break;
        case 6:    pos=x+28;break;
        default:   pos=x-8+y*7;
    }
    return ((BoardType)1)<<pos;
}

```

其中, BoardType 是 64 位整型数。typedef __int64 BoardType。

算法的难点在于如何对当前棋局产生所有合法走步。一个可行的方法是, 一次性产生所有可能的走步存储在一个表中, 算法通过对表的扫描来产生下一个合法走步。这个任务由算法 caches 和 cache 来完成。

```

void caches()
{
    for(int i=0;i<7;i++)
        for(int j=0;j<7;j++){
            if(i<6 && i>0){cache(i, j, 1, 0);cache(i, j, -1, 0);}
            if(j<6 && j>0){cache(i, j, 0, 1);cache(i, j, 0, -1);}
        }
}

```

```

}

void cache(int x, int y, int xv, int yv)
{
    if((board[x][y]!=2) && (board[x+xv][y+yv]!=2)&& (board[x-xv][y-yv]!=2)){
        Move *m;
        moves[count++] = m = new Move;
        m->sx = x - xv; m->sy = y - yv;
        m->xv = xv; m->yv = yv;
    }
}

```

所有 76 个合法走步存储在数组 moves 中。

二维数组 gmoves 存储相应的走步状态，用于判定走步的合法性。算法根据数组 moves 产生 gmoves 的值。

```

void genmv()
{
    for(int j=0;j<count;j++){
        Move *m = moves[j];
        gmoves[j][0] = coord2bit(m->sx, m->sy);
        gmoves[j][0] |= coord2bit(m->sx+m->xv, m->sy+m->yv);
        gmoves[j][0] |= coord2bit(m->sx+2*m->xv, m->sy+2*m->yv);
        gmoves[j][1] = coord2bit(m->sx, m->sy) | coord2bit(m->sx+m->xv, m->sy+m->yv);
    }
}

```

算法中用一个全局变量 gboard 表示当前棋盘状态，由算法 initreg 对其进行初始化。

```

void initreg()
{
    gboard = 0;
    for(int i=0;i<7;i++)
        for(int j=0;j<7;j++)
            if(board[i][j]==1)
                gboard |= coord2bit(i, j);
}

```

函数 ok 判定走步 move 的合法性。

```

int ok(BoardType *move)
{
    return ((gboard & move[0]) == move[1]);
}

```

函数 next 将棋盘状态变换为走步 move 后的状态。

```
void next(BoardType *move)
{
    gboard ^= move[0];
}
```

函数 restore 将棋盘状态恢复为走步 move 前的状态。

```
void restore(BoardType *move)
{
    gboard ^= move[0];
}
```

在算法搜索过程中, 不同的搜索路径可到达同一个棋盘状态。为了避免无效搜索, 可将搜索过的棋盘状态存储起来。算法中采用动态申请内存空间的方法, 以避免空间资源的浪费。

用 makesp 申请内存块指针。

```
unsigned char **makesp(unsigned int mm)
{
    unsigned int i, cnt=mm/blksize+1;
    unsigned char **tmp=new unsigned char *[cnt];
    if(tmp==0)return 0;
    for(i=0;i<cnt;i++)tmp[i]=0;
    return tmp;
}
```

在空间不够时由 lookup 申请新的大小为 blksize 的内存块。

```
unsigned char *lookup(unsigned char **table, IndexType index)
{
    IndexType block=index/blksize, offset=index%blksize;
    if(table[block]==0){
        table[block]=new unsigned char[blksize];
        for(int i=0;i<blksize;i++)table[block][i]=0;
    }
    if(table[block]==0) return 0;
    return (table[block])+offset;
}
```

save 将当前棋盘状态 gboard 存储到内存表 sptab 中。

```
void save()
```

```

{
    *lookup(sptab, gboard/8) |= (1<<(gboard&0x07));
}

```

tried 检测当前棋盘状态 gboard 是否已存储在内存表中。

```

int tried()
{
    return (int) (*lookup(sptab, gboard/8) & (1<<(gboard&0x07)));
}

```

(3) 算法实现

在上述讨论的基础上，解独立钻石跳棋问题的回溯法可具体描述如下。

```

int backtrack(int t)
{
    BoardType *m;
    if(finish(t)) return 1;
    if(t<26 && t%2==0 && tried())return 0;
    for(int i=0;i<count;i++){
        m=gmoves[i];
        if(ok(m)){
            next(m);
            if(backtrack(t+1)){
                ans[t]=find(m);
                return 1;
            }
            restore(m);
        }
    }
    save();
    return 0;
}

```

其中，finish 检测算法终止条件。(endx,endy) 是终止方格的坐标；num 是初始棋盘上棋子个数。

```

bool finish(int t)
{
    if (endx>0 || endy>0) return gboard==coord2bit(endx,endy);
    else return t>num-2;
}

```

find 找出与走步状态相应的走步描述。

```

Move *find(BoardType *m)
{
    for(int i=0; i<count; i++)
        if(gmoves[i]==m) return moves[i];
    return 0;
}

```

最后的主函数描述如下。

```

int main()
{
    int x,y;
    init();
    fin>>num;
    for(int i=0;i<num;i++){
        fin>>x>>y;
        board[x][y]=1;
    }
    fin>>endx>>endy;
    caches();
    genmv();
    initreg();
    sptab = makesp(((unsigned int) 1)<<30);
    if(backtrack(0))
        for(int i=0;i<num-1;i++)out(ans[i]);
    else cout<<"No Solution!"<<endl;
    return 0;
}

```

函数 out 输出走步。

```

void out(Move *m)
{
    cout<<"("<<m->sx<<","<<m->sy<<") ("
        <<m->sx+2*m->xv<<","<<m->sy+2*m->yv<<")"<<endl;
}

```

算法实现题 5-15 智力拼图问题 (习题 5-24)

★问题描述:

设有 12 个平面图形如图 5-11 所示。每个图形的形状互不相同,但它们都是由 5 个大小相同的正方形组成。图 5-11 中这 12 个图形拼接成一个 6×10 的矩形。试设计一个算法,计算出用这 12 个图形拼接成给定矩形的拼接方案。

★编程任务:

对于给定矩形, 计算用上述 12 个图形拼接成给定矩形的一个拼接方案。拼接方案中每个图形可以经过旋转或翻转后进行拼接, 但要求使用 12 个图形中每个图形恰好一次。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 m 和 n , 表示给定的矩形是一个 $m \times n$ 矩形, 且 $m \times n = 60$ 。

★结果输出:

将计算出的矩形的拼接方案输出到文件 output.txt。每行 n 个字符, 共 m 行。

给定的 12 个图形的编号如图 5-12 所示。如果不存在所要求的拼接方案, 输出 “No Solution!”。

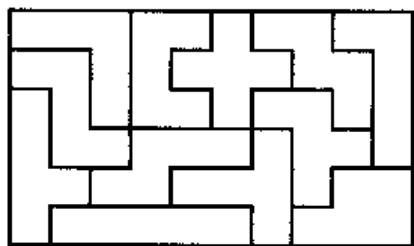


图 5-11 智力拼图

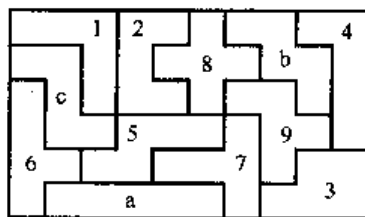


图 5-12 智力拼图图形编号

输入文件示例

input.txt
6 10

输出文件示例

output.txt
111c9aaaaa
1ccc999777
1c3339bb74
22233bb874
26255b8884
6666555844

分析与解答:

(1) 算法思想

将矩形划分成 $m \times n$ 个方格。按从上到下和从左到右的次序, 考察每一个未覆盖方格。每个方格都可能被 12 个图形中的一个图形覆盖。每一个图形经过旋转或翻转, 最多可能产生 8 个不同形态。对每种可能进行无遗漏的回溯搜索, 直至找到一个解。

(2) 基本数据结构

全局变量 brow 和 bcol 分别表示 m 和 n 的值。

用数组 aa[5][5] 表示给定的每个图形。定义结构 pp 如下。

```
typedef unsigned char BYTE;  
typedef struct {  
    BYTE aa[5][5];  
} pp;
```

对于每个图形按其编号初始化如下。

```

BYTE pc00[5][5] = {{1, 1, 1}, {1}, {1}};
BYTE pc01[5][5] = {{2, 2}, {2}, {2, 2}};
BYTE pc02[5][5] = {{3, 3}, {3, 3}, {3}};
BYTE pc03[5][5] = {{4, 4}, {4}, {4}, {4}};
BYTE pc04[5][5] = {{5}, {5, 5}, {0, 5}, {0, 5}};
BYTE pc05[5][5] = {{6}, {6, 6}, {6}, {6}};
BYTE pc06[5][5] = {{7, 7, 7}, {0, 7}, {0, 7}};
BYTE pc07[5][5] = {{0, 8}, {8, 8, 8}, {0, 8}};
BYTE pc08[5][5] = {{0, 9}, {9, 9}, {0, 9, 9}};
BYTE pc09[5][5] = {{10, 10, 10, 10, 10}};
BYTE pc10[5][5] = {{0, 11, 11}, {11, 11}, {11}};
BYTE pc11[5][5] = {{12, 12}, {0, 12}, {0, 12, 12}};

```

将每个图形结构的指针存储于数组 orig 中。

```

pp * orig[] = {
    (pp *) pc00,
    (pp *) pc01,
    (pp *) pc02,
    (pp *) pc03,
    (pp *) pc04,
    (pp *) pc05,
    (pp *) pc06,
    (pp *) pc07,
    (pp *) pc08,
    (pp *) pc09,
    (pp *) pc10,
    (pp *) pc11,
};

```

用结构 fboard 表示给定矩形的状态。

```

typedef struct {
    BYTE pos[60];
    BYTE mark[12];
} fboard;

```

其中, pos 用于表示矩形中 60 个方格被覆盖的情况。在矩形方格 (row, col) 处的覆盖状态存储在 pos[k] 中, 数组下标 k 的值由 addr 计算如下。

```

int addr(int row, int col)
{
    return (row - 1) * bcol + col - 1;
}

```

```
}
```

数组 mark 用于标记使用过的图形。

(3) 算法实现

算法一开始先进行初始化工作，由 init 来完成。

```
void init()
{
    pp tmp1, tmp2;
    for (int b=0; b<12; b++) {
        tmp1=*orig[b];
        for (int d=0; d<5; d++)
            for (int c=0; c<5; c++)
                if (tmp1.aa[d][c]) tmp1.aa[d][c]=0x10;
        for (int a=0, vc=0; a<8; a++) {
            for (int c=0, exist=0; c<vc &&!exist; c++)
                if (memcmp(&var[b][c], &tmp1, sizeof(pp))==0) exist=1;
            if (!exist) var[b][vc++]=tmp1;
            // 旋转
            tmp2=tmp1; rotp(&tmp1, &tmp2);
            // 翻转
            if (a==3) { tmp2=tmp1; flip(&tmp1, &tmp2); }
        }
        nvar[b]=vc;
    }
}
```

其中，全局数组 var[12][8] 用于存储 12 个图形中每个图形的不同形态；nvar[12] 用于存储 12 个图形中每个图形的不同形态数。图形旋转和图形翻转分别由 rotp 和 flip 完成。

```
void rotp(pp *des, pp *src)
{
    memset(des, 0, sizeof(pp));
    for (int x=4, flag=0, xp=0; x>=0; x--) {
        for (int y=0; y<5; y++) {
            des->aa[xp][y]=src->aa[y][x];
            if (src->aa[y][x]) flag=1;
        }
        if (flag) xp++;
    }
}

void flip(pp *des, pp *src)
{
}
```

```

memset(des, 0, sizeof(pp));
for (int x=4; flag=0; xp=0; x>=0; x--) {
    for (int y=0; y<5; y++) {
        des->aa[y][xp]=src->aa[y][x];
        if (src->aa[y][x]) flag=1;
    }
    if (flag) xp++;
}
}

```

算法的主体是对矩形覆盖的回溯搜索。

```

void search(fboard *pre)
{
    int npla=0, frow=0, fcol=0;
    fboard ff=*pre;
    if (ans>0) return;
    // 找最左上的未覆盖方格
    for (int row=1; found=0; row<=brow && !found; row++)
        for (int col=1; col<=bcol && !found; col++)
            if (pre->pos[addr(row, col)]==0) {
                frow=row; fcol=col; found=1;
            }
    // 计算已用过的图形数
    for (int pn=0; pn<12; pn++) if (pre->mark[pn]) npla++;
    for (pn=0; pn<12; pn++) {
        // 是否已用过
        if (pre->mark[pn]) continue;
        // 对每种不同形态
        for (int pv=0; pv<nvar[pn]; pv++) {
            pp tryp=var[pn][pv];
            int py=frow;
            for (int px=1; px<=bcol; px++) {
                if (px>fcol) break;
                if (check(&tryp, &ff, px, py)) {
                    // 用该图形覆盖
                    for (int row=0; row<5; row++) {
                        for (int col=0; col<5; col++) {
                            ff.pos[addr(row+py, col+px)]|=tryp.aa[row][col];
                        }
                    }
                    ff.mark[pn]=1;
                    if (npla>10) {++ans; outf(&ff); return;}
                    else search(&ff);
                }
            }
        }
    }
}

```

```

        // 回溯
        ff=*pre;
    }//if(ok)
} //px
} //pv
} //pn
}

```

其中，算法 check 用于检测图形覆盖的可行性。outf 输出找到的图形覆盖方案。

```

bool check(pp *tryp, fboard *ff, int px, int py)
{
    for (int row=0;row<5;row++)
        for(int col=0;col<5;col++)
            if (tryp->aa[row][col]){
                int r=row+py, c=col+px;
                if(r>brow || c>bcol || ff->pos[addr(r, c)])return 0;
            }
    return 1;
}

void outf(fboard *ff)
{
    if(flag){
        for (int col=1;col<=bcol;col++){
            for (int row=1;row<=brow;row++){
                int x=ff->pos[addr(row, col)]- 16;
                if(x<10)cout<<x;
                if(x==10)cout<<"a";
                if(x==11)cout<<"b";
                if(x==12)cout<<"c";
            }
            cout<<endl;
        }
    }
    else{
        for (int row=1;row<=brow;row++){
            for (int col=1;col<=bcol;col++){
                int x=ff->pos[addr(row, col)]- 16;
                if(x<10)cout<<x;
                if(x==10)cout<<"a";
                if(x==11)cout<<"b";
                if(x==12)cout<<"c";
            }
        }
    }
}

```

```

        cout<<endl;
    }
}
}

```

执行算法的主函数如下。

```

int main()
{
    fin>>brow>>bcol;
    if(brow<bcol){int tmp=brow;brow=bcol;bcol=tmp;flg=1;}
    if(bcol<3 || brow*bcol!=60)cout<<"No Solution!"<<endl;
    else{
        init();
        fboard ff;
        memset(&ff,0,sizeof(fboard));
        search(&ff);
    }
    return 0;
}

```

算法实现题 5-16 布线问题 (习题 5-25)

★问题描述:

假设要将一组元件安装在一块线路板上,为此需要设计一个线路板布线方案。各元件的连线数由连线矩阵 conn 给出。元件 i 和元件 j 之间的连线数为 $\text{conn}(i,j)$ 。如果将元件 i 安装在线路板上位置 r 处,而将元件 j 安装在线路板上位置 s 处,则元件 i 和元件 j 之间的距离为 $\text{dist}(r,s)$ 。确定了所给的 n 个元件的安装位置,就确定了一个布线方案。与此布线方案相应的布线成本为 $\sum_{1 \leq i < j \leq n} \text{conn}(i,j) \cdot \text{dist}(r,s)$ 。试设计一个算法找出所给 n 个元件的布线成本最小的布线方案。

★编程任务:

设计一个算法,对于给定的 n 个元件,计算最佳布线方案,使布线费用达到最小。

★数据输入:

由文件 `input.txt` 给出输入数据。第 1 行有 1 个正整数 n ($1 \leq n \leq 20$)。接下来的 $n-1$ 行,每行 $n-i$ 个数,表示元件 i 和元件 j 之间连线数, $1 \leq i < j \leq 20$ 。

★结果输出:

将计算出的最小布线费用以及相应的最佳布线方案输出到文件 `output.txt`。

输入文件示例

`input.txt`

3

2 3

3

输出文件示例

`output.txt`

10

1 3 2

分析与解答:

与主教材中的电路板排列问题类似。回溯法如下。

```
void Board::Backtrack(int i)
{
    if (i==n) {
        int tmp=len(i);
        if(tmp<bestd){bestd = tmp;for (int j=1;j<=n;j++) bestx[j] = x[j];}
    }
    else
        for (int j = i; j <= n; j++) {
            Swap(x[i], x[j]);
            int ld =len(i);
            if (ld < bestd) Backtrack(i+1);
            Swap(x[i], x[j]);
        }
}
```

其中 len 计算布线费用。

```
int Board::len(int ii)
{
    for (int i=1,sum=0; i<=ii; i++)
        for(int j=i+1;j<=ii;j++){
            int dist=x[i]>x[j]? x[i] - x[j]:x[j] - x[i];
            sum+=conn[i][j]*dist;
        }
    return sum;
}
```

算法实现题 5-17 最佳调度问题 (习题 5-26)

★问题描述:

假设有 n 个任务由 k 个可并行工作的机器来完成。完成任务 i 需要的时间为 t_i 。试设计一个算法找出完成这 n 个任务的最佳调度,使得完成全部任务的时间最早。

★编程任务:

对任意给定的整数 n 和 k ,以及完成任务 i 需要的时间为 $t_i, i=1 \sim n$ 。编程计算完成这 n 个任务的最佳调度。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 k 。第 2 行的 n 个正整数是完成 n 个任务需要的时间。

★结果输出:

将计算出的完成全部任务的最早时间输出到文件 output.txt。

输入文件示例

input.txt

7 3

2 14 4 16 6 5 3

输出文件示例

output.txt

17

分析与解答：

简单回溯搜索。

```
void search(int dep)
{
    if(dep==n) {
        int tmp=comp();
        if(tmp<best) best=tmp;
        return;
    }
    for(int i=0;i<k;i++){
        len[i]+=t[dep];
        if(len[i]<best) search(dep+1);
        len[i]-=t[dep];
    }
}
```

comp 计算完成任务的时间。

```
int comp()
{
    int tmp=0;
    for(int i=0;i<k;i++) if(len[i]>tmp)tmp=len[i];
    return tmp;
}
```

算法实现题 5-18 无优先级运算问题 (习题 5-27)

★问题描述：

给定 n 个正整数和 4 个运算符 $+$, $-$, $*$, $/$, 且运算符无优先级, 如 $2+3\times 5=25$ 。对于任意给定的整数 m , 试设计一个算法, 用以上给出的 n 个数和 4 个运算符, 产生整数 m , 且用的运算次数最少。给出的 n 个数中每个数最多只能用 1 次, 但每种运算符可以任意使用。

★编程任务：

对于给定的 n 个正整数, 设计一个算法, 用最少的无优先级运算次数产生整数 m 。

★数据输入：

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m 。第 2 行是给定的用于运算的 n 个正整数。

★结果输出:

将计算出的产生整数 m 的最少无优先级运算次数以及最优无优先级运算表达式输出到文件 output.txt。

输入文件示例

input.txt

5 25

5 2 3 6 7

输出文件示例

output.txt

2

2+3*5

分析与解答:

readin 读入初始数据。

```
void readin()
{
    fin>>n>>m;
    a=new int[n];
    num=new int[n];
    oper=new int[n];
    flag=new int[n];
    for(int i=0;i<n;i++) {fin>>a[i];flag[i]=0;}
}
```

采用迭代加深的回溯法。

```
bool search(int dep)
{
    if(dep>k) {if(found()) return true; else return false;}
    for(int i=0;i<n;i++)
        if (flag[i]==0) {
            num[dep]=a[i];
            flag[i]=1;
            for(int j=0;j<4;j++){
                oper[dep]=j;
                if(search(dep+1)) return true;
            }
            flag[i]=0;
        }
    return false;
}
```

found 判断是否找到解。

```
bool found()
{
    int x=num[0];
```

```

        for(int i=0;i<k;i++){
            switch (oper[i]){
                case 0: x+=num[i+1]; break;
                case 1: x-=num[i+1]; break;
                case 2: x*=num[i+1]; break;
                case 3: x/=num[i+1]; break;
            }
        }
        return (x==m);
    }
}

```

实现算法的主函数如下。

```

int main()
{
    create();
    readin();
    for(k=0;k<n;k++)
        if(search(0)) {cout<<k<<endl;out();return 0;}
    cout<<"No Solution!"<<endl;
    return 0;
}

```

算法实现题 5-19 世界名画陈列馆问题（习题 5-29）

★问题描述：

世界名画陈列馆由 $m \times n$ 个排列成矩形阵列的陈列室组成。为了防止名画被盗，需要在陈列室中设置警卫机器人哨位。每个警卫机器人除了监视它所在的陈列室外，还可以监视与它所在的陈列室相邻的上、下、左、右 4 个陈列室。试设计一个安排警卫机器人哨位的算法，使得名画陈列馆中每一个陈列室都在警卫机器人的监视之下，且所用的警卫机器人数量最少。

★编程任务：

设计一个算法，计算警卫机器人的最佳哨位安排，使得名画陈列馆中每一个陈列室都在警卫机器人的监视之下，且所用的警卫机器人数量最少。

★数据输入：

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 m 和 n ($1 \leq m, n \leq 20$)。

★结果输出：

将计算出的警卫机器人数量及其最佳哨位安排输出到文件 output.txt。文件的第 1 行是警卫机器人数量；接下来的 m 行中每行 n 个数，0 表示无哨位，1 表示哨位。

输入文件示例

input.txt

4 4

输出文件示例

output.txt

4

0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

分析与解答:

(1) 状态空间树

本题的状态空间树是一棵子集树, 可用对子集树进行搜索的回溯算法框架求解。依从上到下、从左到右的顺序依次考察每一个陈列室设置警卫机器人哨位的情况, 以及该陈列室受警卫机器人监视的情况。用 $x[i, j]$ 表示陈列室 (i, j) 当前设置警卫机器人哨位的状态。当 $x[i, j]=1$ 时, 表示已经在陈列室 (i, j) 设置了警卫机器人哨位。当 $x[i, j]=0$ 时, 表示在陈列室 (i, j) 尚未设置警卫机器人哨位。另一方面, 用 $y[i, j]$ 表示陈列室 (i, j) 当前受警卫机器人监视的状态。当 $y[i, j]=1$ 时, 表示陈列室 (i, j) 已受警卫机器人监视。当 $y[i, j]=0$ 时, 表示在陈列室 (i, j) 尚未受警卫机器人监视。

(2) 采用剪枝技术提高回溯法的效率

① 下界剪枝法

设当前已设置的警卫机器人哨位数为 k , 已受警卫机器人监视的陈列室数为 t , 当前最优警卫机器人哨位数为 best 。在一般情况下, 可根据 k 和 t 的值, 估计出尚需设置的警卫机器人哨位数下界 $f(k, t)$ 。当 $f(k, t) \geq \text{best}$ 时, 可将状态空间树中以当前结点为根的子树剪去。

② 控制剪枝法

设 p 和 q 是状态空间树中 2 个不同的结点。如果按照结点 p 和 q 的某一相关关系, 可以确定以结点 q 为根的子树中的解不优于以结点 p 为根的子树中的解, 则称结点 p 控制了结点 q 。对于本题来说, 可考虑以下的结点控制关系。

a. 已受监视结点的控制关系

设在回溯搜索时当前所关注的是陈列室 (i, j) 。该陈列室已受监视, 即 $y[i, j]=1$ 。与其相邻的其他陈列室的受监视状态如图 5-13 所示。

此时在陈列室 (i, j) 处设置一个警卫机器人哨位, 即取 $x[i, j]=1$, 相应于状态空间树中的一个结点 q 。在陈列室 $(i+1, j+1)$ 处设置一个警卫机器人哨位, 即取 $x[i+1, j+1]=1$, 相应于状态空间树中的另一个结点 p 。容易看出, 此时以结点 q 为根的子树中的解不优于以结点 p 为根的子树中的解, 即结点 p 控制了结点 q 。可将状态空间树中以结点 q 为根的子树剪去。由此总结出在以从上到下、从左到右的顺序依次考察每一个陈列室时, 已受监视的陈列室处不必设置警卫机器人哨位。这样可以避免许多无效搜索。

b. 测试结点的控制关系

设陈列室 (i, j) 是当前以从上到下、从左到右的顺序搜索遇到的第一个未受监视的陈列室。为了使陈列室 (i, j) 受到监视, 可在陈列室 $(i+1, j), (i, j), (i, j+1)$ 处设置警卫机器人哨位。在这 3 处设置哨位所相应的状态空间树中的结点分别为 p, q 和 r , 如图 5-14 所示。

显而易见, 当 $y(i, j+1)=1$ 时, 结点 p 控制结点 q ; 当 $y(i, j+1)=1$ 且 $y(i, j+2)=1$ 时, 结点 p 控制结点 r 。因此, 在搜索时应按 $p \rightarrow q \rightarrow r$ 的顺序来扩展结点, 并检测结点 p 对结点 q 和结点 r 的控制条件, 及时剪去受控结点相应的子树。

(3) 简化边界条件

在陈列室矩形阵列的外圈扩展一层, 即增加 0 行和 0 列, $n+1$ 行和 $m+1$ 列。使算法可以统一地处理边界条件。

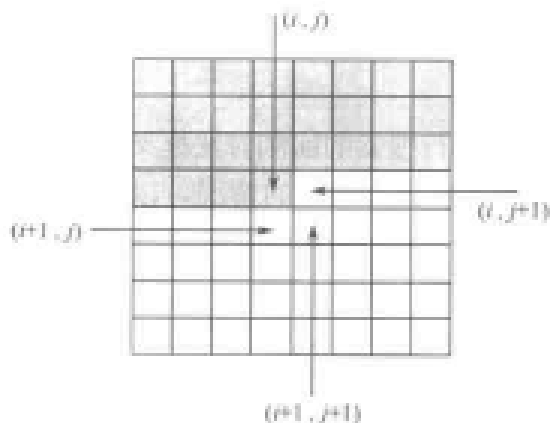


图 5-13 受监视结点的控制关系

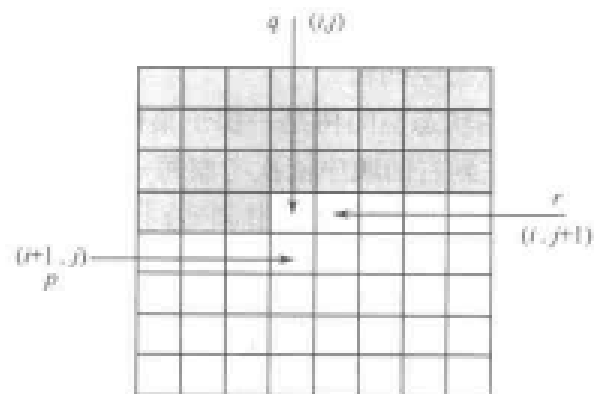


图 5-14 测试结点的控制关系

根据前面的分析, 可以设计安排警卫机器人哨位的回溯法如下。

算法中用到的变量类型说明如下。

```
#define MLEN 50
int d[6][3] = {{0,0,0},{0,0,0},{0,0,-1},{0,-1,0},{0,0,1},{0,1,0}};
int x[MLEN+1][MLEN+1], y[MLEN+1][MLEN+1], bestx[MLEN+1][MLEN+1];
int n, m, best, k=0, t=0, t1, t2, more;
bool p;
```

回溯法的主体由 $\text{search}(i, j)$ 来实现。参数 i 和 j 表示当前搜索位置。

```
void search(int i, int j)
{
    do{
        j++;
        if(j>n){i++;j=1;}
    }while (!((y[i][j]==0)|| (i>n)));
    if(i>n){
        if(k<best){best=k;copy(bestx, x);}
        return;
    }
    if(k+(t1-t)/5>=best) return;
    if((i<n-1) && (k+(t2-t)/5>=best)) return;
    if((i<n)){
        change(i+1, j);
        search(i, j);
        restore(i+1, j);
    }
    if((j<n)&&((y[i][j+1]==0)|| (y[i][j+2]==0))){
```

```

        change(i, j+1);
        search(i, j);
        restore(i, j+1);
    }
    if(((y[i+1][j]==0)&&(y[i][j+1]==0))) {
        change(i, j);
        search(i, j);
        restore(i, j);
    }
}

```

其中, $\text{change}(i, j)$ 用于在 (i, j) 处设置一个警卫机器人哨位, 并相应地改变其相邻陈列室的受监视状况。

```

void change (int i, int j)
{
    x[i][j]=1;k++;
    for (int s=1;s<=5;s++) {
        int p=i+d[s][1];
        int q=j+d[s][2];
        y[p][q]++;
        if((y[p][q]==1)) t++;
    }
}

```

$\text{restore}(i, j)$ 则用于撤销在 (i, j) 处设置的警卫机器人哨位, 并相应地改变其相邻陈列室的受监视状况。

```

void restore (int i, int j)
{
    x[i][j]=0;k--;
    for (int s=1;s<=5;s++) {
        int p=i+d[s][1];
        int q=j+d[s][2];
        y[p][q]--;
        if((y[p][q]==0)) t--;
    }
}

```

最后由 `compute` 调用主体算法, 搜索最优解。

```

void compute()
{
    more=m/4+1;

```

```

        if(m%4==3)more++;
        else if(m%4==2) more+=2;
        t2=m*n+more+4;
        t1=m*n+4;
        best=INT_MAX;
        memset(y,0,sizeof(y));
        memset(x,0,sizeof(x));
        if((n==1) && (m==1)) {
            cout<<1<<endl<<1<<endl;
            return;
        }
        for (int i=0;i<=m+1;i++) {y[0][i]=1;y[n+1][i]=1;}
        for (i=0;i<=n+1;i++) {y[i][0]=1;y[i][m+1]=1;}
        search(1,0);
        output();
    }
}

```

算法实现题 5-20 世界名画陈列馆问题（不重复监视）（习题 5-30）

★问题描述：

世界名画陈列馆由 $m \times n$ 个排列成矩形阵列的陈列室组成。为了防止名画被盗，需要在陈列室中设置警卫机器人哨位。每个警卫机器人除了监视它所在的陈列室外，还可以监视与它所在的陈列室相邻的上、下、左、右 4 个陈列室。试设计一个安排警卫机器人哨位的算法，使得名画陈列馆中每一个陈列室都在警卫机器人的监视之下，并且要求每一个陈列室仅受一个警卫机器人监视，且所用的警卫机器人人数最少。

★编程任务：

设计一个算法，计算警卫机器人的最佳哨位安排，使得名画陈列馆中每一个陈列室都仅受一个警卫机器人监视。且所用的警卫机器人人数最少。

★数据输入：

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 m 和 n ($1 \leq m, n \leq 20$)。

★结果输出：

将计算出的警卫机器人人数及其最佳哨位安排输出到文件 output.txt。文件的第一行是警卫机器人人数；接下来的 m 行中每行 n 个数，0 表示无哨位，1 表示哨位。如果不存在满足要求的哨位安排，则输出 “No Solution!”。

输入文件示例

input.txt

4 4

输出文件示例

output.txt

4

0 0 1 0

1 0 0 0

0 0 0 1

0 1 0 0

分析与解答:

本题要求每一个陈列室仅受一个警卫机器人监视,在许多情况下问题无解。下面分3种情形来讨论。

(1) $1=n \leq m$ 的情形

此时问题恒有解,且容易直接写出其最优解。

当 $m \bmod 3 = 1$ 时,将机器人哨位置于 $(1, 3k+1)$ $k=0, 1, \dots, \lfloor m/3 \rfloor$ 。

当 $m \bmod 3 = 0$ 或 2 时,将机器人哨位置于 $(1, 3k+2)$ $k=0, 1, \dots, \lfloor m/3 \rfloor$ 。

(2) $2=n \leq m$ 的情形

在这种情况下,如果问题有解,则易知必须在2端分别设置2个机器人哨位。这2个机器人哨位各监视3个陈列室。其余的 k 个机器人哨位均监视4个陈列室。由此可见, $2m = 4k+6$, 即 $m = 2k+3$ 为奇数。当 m 为偶数时问题无解。

当 m 为奇数时,容易直接写出其最优解。将机器人哨位分别置于 $(1, 4k+3)$ 和 $(2, 4k+1)$, $k=0, 1, \dots, \lfloor m/4 \rfloor$ 。

(3) $2 < n \leq m$ 的情形

当 $n > 2$ 时,用直接枚举法容易验证 $n=3, m=3$ 和 $n=3, m=4$ 时问题无解; $n=4, m=4$ 时问题有解。当 $n \geq 3$ 且 $m \geq 5$ 时,问题无解。这一结论可证明如下。考虑左上角的 3×5 阵列。下面证明在不重复监视的前提下,无法使这 3×5 阵列中的每一个陈列室都受到监视。为此,考察本问题的一个变形问题 $P(n, m)$ 如下。在设置警卫机器人的哨位时,允许在第 $n+1$ 行和第 $m+1$ 列设置哨位,但不要求第 $n+1$ 行和第 $m+1$ 列的陈列室均受监视。当 $n \geq 3$ 且 $m \geq 5$ 时,在不重复监视的前提下原问题有解,则 $P(3, 5)$ 一定有解。换句话说,如果问题 $P(3, 5)$ 无解,则当 $n \geq 3$ 且 $m \geq 5$ 时,不重复监视问题无解。因此,问题转化为证明问题 $P(3, 5)$ 无解。对习题 5-29 解答的算法 search 做适当修改可用于解问题 $P(n, m)$ 。用修改过的算法 search 对问题 $P(3, 5)$ 求解得知该问题无解。这就证明了上述结论。

具体算法由 compute 实现如下。

```
void compute()
{
    memset(x, 0, sizeof(x));
    bool ok=false;
    if(n==1){
        int k=m/3;
        if(m%3==1) for(int j=0; j<=k; j++) x[1][3*j+1]=1;
        else {
            if(m%3==0) k--;
            for(int j=0; j<=k; j++) x[1][3*j+2]=1;
        }
        best=k+1; ok=true;
    }
    if(m==1){
        int k=n/3;
        if(n%3==1) for(int j=0; j<=k; j++) x[3*j+1][1]=1;
        else {
```

```

        if(n%3==0)k--;
        for(int j=0;j<=k;j++){x[3*j+2][1]=1;
        }
        best=k+1;ok=true;
    }
    if(n==2 && m%2==0){
        int k=m/4;
        if(m%4==0)k--;
        for(int j=0;j<=k;j++){x[1][4*j+3]=1;x[2][4*j+1]=1;}
        best=2*k+2;ok=true;
    }
    if(m==2 && n%2==0){
        int k=n/4;
        if(n%4==0)k--;
        for(int j=0;j<=k;j++){x[4*j+3][1]=1;x[4*j+1][2]=1;}
        best=2*k+2;ok=true;
    }
    if(n==4 && m==4){x[1][1]=1;x[1][4]=1;x[4][1]=1;x[4][4]=1;best=4;ok=
        true;}
    if(ok) output();
    else cout<<"No Solution!"<<endl;
}

```

算法实现题 5-21 $2 \times 2 \times 2$ 魔方问题

★问题描述:

$2 \times 2 \times 2$ 魔方的构造如图 5-15 所示。图中英文字母 U, L, F, R, B, D 分别表示魔方的 6 个面中的上面、左面、前面、右面、后面、底面。魔方的每个面都可以绕其中轴旋转。给定魔方的初始状态, 可以经过若干次旋转将魔方变换成每个面都只有一种颜色的状态。绕中轴将一个面旋转 90° 算作一次旋转。试设计一个算法计算出从初始状态到目标状态 (每个面都只有一种颜色的状态) 所需的最少旋转次数。

★编程任务:

设计一个算法, 对于给定的 $2 \times 2 \times 2$ 魔方的初始状态, 计算从初始状态到目标状态 (每个面都只有一种颜色的状态) 所需的最少旋转次数。

★数据输入:

由文件 input.txt 给出输入数据。 $2 \times 2 \times 2$ 魔方目标状态的每个面都有一种颜色, 分别用大写英文字母 W, O, G, R, Y, B 来表示。将 6 个面展开并编号如图 5-16 所示。文件将给定魔方的初始状态的 6 个面, 按照其编号依次排列, 共有 12 行, 每行有 2 个表示方块颜色的大写英文字母。

★结果输出:

将计算出的最少旋转次数和最优旋转序列输出到文件 output.txt。文件的第 1 行是最少旋转次数; 接下来是最优旋转序列。用表示各面的大写英文字母表示每个面的顺时针旋转;

表示各面的大写英文字母紧接一个“-”表示每个面的逆时针旋转。如果不存在满足要求的旋转序列则输出 “No Solution!”。

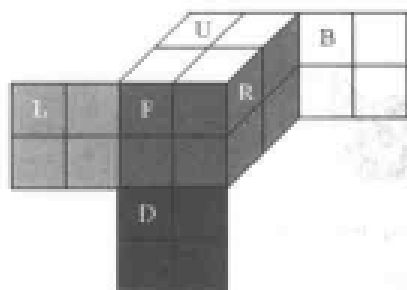


图 5-15 2×2×2 魔方

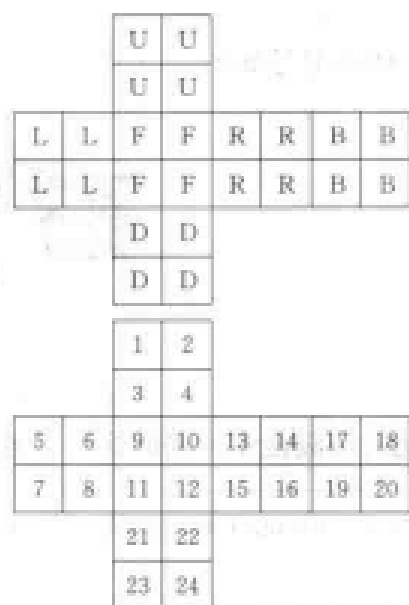


图 5-16 2×2×2 魔方编号

输入文件示例

input.txt

G W

G R

W W

O O

O G

B G

B R

B R

Y R

Y W

Y O

Y B

输出文件示例

output.txt

2

R U

分析与解答：

(1) 算法思想

对于魔方的每个面，有 12 种旋转方式将其变换为另一种状态。这 12 种旋转变换是 F, F-, U, U-, R, R-, L, L-, B, B-, D, D-。对于 2×2×2 魔方而言，后 6 种变换可等价地由前 6 种变换来实现。因此可将回溯法的搜索空间组织成一棵六叉树。

由于需要找最短旋转序列，采用逐步深化（Iterative Deepening）的回溯搜索策略。

(2) 算法实现

2×2×2 魔方由 8 个小立方体组成。每个小立方体都有 3 种不同的定向。用结构 cubT 表示每个小立方体的定向。

```
typedef struct
{ int cube[8]; } cubeT;
```

8 个小立方体的编号如图 5-17 所示。

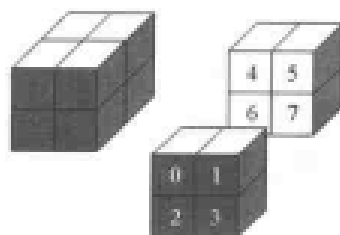


图 5-17 8 个小立方体的编号

用逐步深化的回溯搜索策略的算法表述如下。

```
int *IDsearch(cubeT *x, int maxdepth)
{
    for(int t=0;t<=maxdepth;t++){
        int *res=backtrack(t,x);
        if (res) {minmov=t;return res;}
    }
    return 0;
}
```

其中，maxdepth 是最大搜索深度；minmov 记录最少旋转次数。实现搜索的回溯法由 backtrack 完成。

```
int *backtrack(int depth, cubeT *p)
{
    cubeT newp[6];
    int *res=0;
    if (depth==0){
        if (solved(p)){
            res=new int[15];
            return res;
        }
        else return 0;
    }
    for (int i=0;i<6;i++){
        move(p,&newp[i],i);
        res=backtrack(depth-1,&newp[i]);
        if (res) {res[depth-1]=i;return res;}
    }
    return 0;
}
```

```
}
```

参数 depth 控制递归深度。旋转到目标状态时，每个面的 4 个小立方块颜色相同，其定向均为 0 号定向，由 solved 判定。move 对每种旋转变换产生魔方的一种新状态。这 2 个算法稍后再做进一步说明。

在此基础上，算法主函数表述如下。

```
int main()
{
    cubeT x;
    readCube(&x);
    outsol(IDsearch(&x,maxdepth));
    return 0;
}
```

其中，readCube 读入魔方初始状态，并对整个算法进行初始化运算；outsol 输出计算结果。

```
void readCube(cubeT *x)
{
    char *s=new char[24];
    ifstream fin("cub2.in");
    init();
    for(int i=0;i<24;i++)fin>>s[i];
    setCube(x,s);
}
```

算法 init 和 setCube 稍后进一步说明。

```
void outsol(int *res)
{
    if(!res) cout<<"No Solution! (maxdepth="<<maxdepth<<)"<<endl;
    else{
        cout<<minmov<<endl;
        for(int j=minmov-1;j>=0;j--)
            cout<<moveOut[res[j]]<<" ";cout<<endl;
    }
}
```

moveOut 是表示 6 种旋转变换的字符串数组。

```
char moveOut[6][3]={{ "F "}, {"F -"}, {"U "}, {"U -"}, {"R "}, {"R -"},};
```

(3) 实现算法的数据结构

2×2×2 魔方的结构比较复杂，因而表示算法的数据结构也较复杂，这是算法实现的难

点。关键问题是每个魔方小立方体的定向，以及魔方作为一个整体立方体的定向。
标准状态下，魔方各面的颜色和编号如图 5-18 所示。

面	上 U	左 L	前 F	右 R	后 B	下 D
颜色	白 W	橙 O	绿 G	红 R	黄 Y	蓝 B
编号	0	1	2	3	4	5

图 5-18 魔方各面的颜色和编号

按此编号，魔方的 24 种定向及其编号如图 5-19 所示。

定向	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
左 L	1	2	4	3	1	1	1	5	0	3	2	0	5	4	5	2	4	0	3	2	4	3	0	5
右 R	3	4	2	1	3	3	3	0	5	1	4	5	0	2	0	4	2	5	1	4	2	1	5	0
前 F	2	3	1	4	0	5	4	2	2	2	0	1	3	0	1	5	5	3	0	1	3	5	4	4
后 B	4	1	3	2	5	0	2	4	4	4	5	3	1	5	3	0	0	1	5	3	1	0	2	2
上 U	0	0	0	0	4	2	5	1	3	5	1	2	2	3	4	3	1	4	2	5	5	4	1	3
下 D	5	5	5	5	2	4	0	3	1	0	3	4	4	1	2	1	3	2	4	0	0	2	3	1

图 5-19 魔方的 24 种定向及其编号

根据每个定向的特点，其命名如图 5-20 所示。

#define ID	0	#define CORNER_2_CW	12
#define FACE_U_CW	1	#define CORNER_2_CCW	13
#define FACE_U_CCW	2	#define CORNER_5_CW	14
#define FACE_U_180	3	#define CORNER_5_CCW	15
#define FACE_L_CW	4	#define CORNER_7_CW	16
#define FACE_L_CCW	5	#define CORNER_7_CCW	17
#define FACE_L_180	6	#define EDGE_1	18
#define FACE_F_CW	7	#define EDGE_3	19
#define FACE_F_CCW	8	#define EDGE_4	20
#define FACE_F_180	9	#define EDGE_6	21
#define CORNER_0_CW	10	#define EDGE_8	22
#define CORNER_0_CCW	11	#define EDGE_9	23

图 5-20 魔方的 24 种定向及其命名

其中，每个定向也可以看作从魔方的标准状态到相应定向的一个旋转变换。

用二维数组 mult[24][24]记录变换矩阵。mult[i][j]表示对第 j 个定向进行第 i 个变换得到的定向。

每个旋转变换作用于小立方体，改变了小立方体的定向。实现旋转变换的算法由 smove 完成。

```
static void smove(cubeT *t, int snum)
```

```

{
switch(snum) {
case ID:
break;
case FACE_U_CW:
swap4(&t->cube[0], &t->cube[4], &t->cube[5], &t->cube[1]);
swap4(&t->cube[2], &t->cube[6], &t->cube[7], &t->cube[3]);
break;
case FACE_U_CCW:
swap4(&t->cube[0], &t->cube[1], &t->cube[5], &t->cube[4]);
swap4(&t->cube[2], &t->cube[3], &t->cube[7], &t->cube[6]);
break;
case FACE_U_180:
swap2(&t->cube[0], &t->cube[5]);
swap2(&t->cube[1], &t->cube[4]);
swap2(&t->cube[2], &t->cube[7]);
swap2(&t->cube[3], &t->cube[6]);
break;
case FACE_L_CW:
swap4(&t->cube[0], &t->cube[2], &t->cube[6], &t->cube[4]);
swap4(&t->cube[1], &t->cube[3], &t->cube[7], &t->cube[5]);
break;
case FACE_L_CCW:
swap4(&t->cube[0], &t->cube[4], &t->cube[6], &t->cube[2]);
swap4(&t->cube[1], &t->cube[5], &t->cube[7], &t->cube[3]);
break;
case FACE_L_180:
swap2(&t->cube[0], &t->cube[6]);
swap2(&t->cube[1], &t->cube[7]);
swap2(&t->cube[2], &t->cube[4]);
swap2(&t->cube[3], &t->cube[5]);
break;
case FACE_F_CW:
swap4(&t->cube[0], &t->cube[1], &t->cube[3], &t->cube[2]);
swap4(&t->cube[4], &t->cube[5], &t->cube[7], &t->cube[6]);
break;
case FACE_F_CCW:
swap4(&t->cube[0], &t->cube[2], &t->cube[3], &t->cube[1]);
swap4(&t->cube[4], &t->cube[6], &t->cube[7], &t->cube[5]);
break;
case FACE_F_180:
swap2(&t->cube[0], &t->cube[3]);
swap2(&t->cube[1], &t->cube[2]);
swap2(&t->cube[4], &t->cube[7]);

```

```

    swap2(&t->cube[5], &t->cube[6]);
    break;
case CORNER_0_CW:
    swap3(&t->cube[1], &t->cube[2], &t->cube[4]);
    swap3(&t->cube[3], &t->cube[6], &t->cube[5]);
    break;
case CORNER_0_CCW:
    swap3(&t->cube[1], &t->cube[4], &t->cube[2]);
    swap3(&t->cube[3], &t->cube[5], &t->cube[6]);
    break;
case CORNER_2_CW:
    swap3(&t->cube[0], &t->cube[5], &t->cube[3]);
    swap3(&t->cube[2], &t->cube[4], &t->cube[7]);
    break;
case CORNER_2_CCW:
    swap3(&t->cube[0], &t->cube[3], &t->cube[5]);
    swap3(&t->cube[2], &t->cube[7], &t->cube[4]);
    break;
case CORNER_5_CW:
    swap3(&t->cube[0], &t->cube[3], &t->cube[6]);
    swap3(&t->cube[1], &t->cube[7], &t->cube[4]);
    break;
case CORNER_5_CCW:
    swap3(&t->cube[0], &t->cube[6], &t->cube[3]);
    swap3(&t->cube[1], &t->cube[4], &t->cube[7]);
    break;
case CORNER_7_CW:
    swap3(&t->cube[0], &t->cube[5], &t->cube[6]);
    swap3(&t->cube[1], &t->cube[7], &t->cube[2]);
    break;
case CORNER_7_CCW:
    swap3(&t->cube[0], &t->cube[6], &t->cube[5]);
    swap3(&t->cube[1], &t->cube[2], &t->cube[7]);
    break;
case EDGE_1:
    swap2(&t->cube[0], &t->cube[1]);
    swap2(&t->cube[2], &t->cube[5]);
    swap2(&t->cube[3], &t->cube[4]);
    swap2(&t->cube[6], &t->cube[7]);
    break;
case EDGE_3:
    swap2(&t->cube[0], &t->cube[2]);
    swap2(&t->cube[1], &t->cube[6]);
    swap2(&t->cube[3], &t->cube[4]);

```

```

        swap2(&t->cube[5], &t->cube[7]);
        break;
    case EDGE_4:
        swap2(&t->cube[0], &t->cube[7]);
        swap2(&t->cube[1], &t->cube[3]);
        swap2(&t->cube[2], &t->cube[5]);
        swap2(&t->cube[4], &t->cube[6]);
        break;
    case EDGE_6:
        swap2(&t->cube[0], &t->cube[7]);
        swap2(&t->cube[1], &t->cube[6]);
        swap2(&t->cube[2], &t->cube[3]);
        swap2(&t->cube[4], &t->cube[5]);
        break;
    case EDGE_8:
        swap2(&t->cube[0], &t->cube[4]);
        swap2(&t->cube[1], &t->cube[6]);
        swap2(&t->cube[2], &t->cube[5]);
        swap2(&t->cube[3], &t->cube[7]);
        break;
    case EDGE_9:
        swap2(&t->cube[0], &t->cube[7]);
        swap2(&t->cube[1], &t->cube[5]);
        swap2(&t->cube[2], &t->cube[6]);
        swap2(&t->cube[3], &t->cube[4]);
        break;
    }
}

```

其中, swap2, swap3, swap4 分别循环交换 2, 3, 4 个元素。

```
static void swap2(int *a, int *b)
```

```
{
    int A=*a, B=*b;
    *b=A; *a=B;
}
```

```
static void swap3(int *a, int *b, int *c)
```

```
{
    int A=*a, B=*b, C=*c;
    *b=A; *c=B; *a=C;
}
```

```
static void swap4(int *a, int *b, int *c, int *d)
```

```

{
    int A=*a,B=*b,C=*c,D=*d;
    *b=A;*c=B;*d=C;*a=D;
}

```

旋转变换矩阵 mult[24][24] 由 init 初始化。

```

void init()
{
    cubeT x,y;
    for(int i=0;i<24;i++)
        for(int j=0;j<24;j++){
            for(int k=0;k<8;k++) x.cube[k]=k;
            smove(&x,j);smove(&x,i);
            for(int p=0;p<24;p++){
                for(int k=0;k<8;k++)y.cube[k]=k;
                smove(&y,p);
                if(equal(&x,&y)){mult[i][j]=p;break;}
            }
        }
}

```

其中, equal 用于判断 2 个魔方状态是否相同。

```

static int equal(cubeT *x,cubeT *y)
{
    for(int i=0;i<8;i++)
        if(x->cube[i]!=y->cube[i]) return 0;
    return 1;
}

```

用数组 lc[24], rc[24], fc[24], bc[24], uc[24], dc[24] 表示魔方的 24 种定向的各面颜色号。读入魔方初始数据后, 由 setCube 确定魔方各小立方体的定向。

```

static int lc[24]={1,2,4,3,1,1,1,5,0,3,2,0,5,4,5,2,4,0,3,2,4,3,0,5};
static int rc[24];
static int fc[24];
static int bc[24];
static int uc[24];
static int dc[24];

void setCube(cubeT *x,const char *s)
{
    int cube[6][4];

```



```

int colors[256];
int i, j;
colors['W']=0; colors['O']=1; colors['G']=2; colors['R']=3; colors['Y']=4; colors
    ['B']=5;
for(i=0; i<8; i++) x->cube[i]=0;
for(i=0; i<6; i++)
    for(j=0; j<4; j++)
        cube[i][j]=i;
for(i=0; i<6; i++)
    for(j=0; j<4; j++)
        cube[i][j]=colors[s[i*4+j]];
for(i=0; i<24; i++) {
    rc[i]=lc[mult[FACE_U_180][i]];
    fc[i]=lc[mult[FACE_U_CW][i]];
    bc[i]=lc[mult[FACE_U_CCW][i]];
    uc[i]=lc[mult[FACE_F_CCW][i]];
    dc[i]=lc[mult[FACE_F_CW][i]];
}

for(i=0; i<24; i++) {
    if(uc[i]==cube[0][2] && lc[i]==cube[1][1] && fc[i]==cube[2][0]) x->cube[0]=i;
    if(uc[i]==cube[0][3] && fc[i]==cube[2][1] && rc[i]==cube[3][0]) x->cube[1]=i;
    if(lc[i]==cube[1][3] && fc[i]==cube[2][2] && dc[i]==cube[5][0]) x->cube[2]=i;
    if(fc[i]==cube[2][3] && rc[i]==cube[3][2] && dc[i]==cube[5][1]) x->cube[3]=i;
    if(uc[i]==cube[0][0] && lc[i]==cube[1][0] && bc[i]==cube[4][1]) x->cube[4]=i;
    if(uc[i]==cube[0][1] && rc[i]==cube[3][1] && bc[i]==cube[4][0]) x->cube[5]=i;
    if(lc[i]==cube[1][2] && bc[i]==cube[4][3] && dc[i]==cube[5][2]) x->cube[6]=i;
    if(rc[i]==cube[3][3] && bc[i]==cube[4][2] && dc[i]==cube[5][3]) x->cube[7]=i;
}
}

```

判定每个面是否达到目标状态的算法 solved 实现如下。

```

int solved(cubeT *x)
{
    // 上 U
    char col=lc[s[mult[FACE_F_CCW][x->cube[4]]];
    if(lc[s[mult[FACE_F_CCW][x->cube[5]]]!=col) return 0;
    if(lc[s[mult[FACE_F_CCW][x->cube[0]]]!=col) return 0;
    if(lc[s[mult[FACE_F_CCW][x->cube[1]]]!=col) return 0;

    // 左 L
    col=lc[s[x->cube[4]]];
    if(lc[s[x->cube[0]]!=col) return 0;
}

```

```

    if(lcols[x->cube[6]]!=col)return 0;
    if(lcols[x->cube[2]]!=col)return 0;

    // 前 F
    col=lcols[mult[FACE_U_CW][x->cube[0]]];
    if(lcols[mult[FACE_U_CW][x->cube[1]]]!=col)return 0;
    if(lcols[mult[FACE_U_CW][x->cube[2]]]!=col)return 0;
    if(lcols[mult[FACE_U_CW][x->cube[3]]]!=col)return 0;

    // 右 R
    col=lcols[mult[FACE_U_180][x->cube[1]]];
    if(lcols[mult[FACE_U_180][x->cube[5]]]!=col)return 0;
    if(lcols[mult[FACE_U_180][x->cube[3]]]!=col)return 0;
    if(lcols[mult[FACE_U_180][x->cube[7]]]!=col)return 0;

    // 后 B
    col=lcols[mult[FACE_U_CCW][x->cube[5]]];
    if(lcols[mult[FACE_U_CCW][x->cube[4]]]!=col)return 0;
    if(lcols[mult[FACE_U_CCW][x->cube[7]]]!=col)return 0;
    if(lcols[mult[FACE_U_CCW][x->cube[6]]]!=col)return 0;

    // 下 D
    col=lcols[mult[FACE_F_CW][x->cube[2]]];
    if(lcols[mult[FACE_F_CW][x->cube[3]]]!=col)return 0;
    if(lcols[mult[FACE_F_CW][x->cube[6]]]!=col)return 0;
    if(lcols[mult[FACE_F_CW][x->cube[7]]]!=col)return 0;
    return 1;
}

```

绕每个面的中轴进行旋转的 6 个变换定义如下。

```

# define  FF  0  // 前面顺时针
# define  FR  1  // 前面逆时针
# define  UF  2  // 上面顺时针
# define  UR  3  // 上面逆时针
# define  RF  4  // 右面顺时针
# define  RR  5  // 右面逆时针

```

实现这 6 个变换的算法 move 表述如下。

```

void move(cubeT *f, cubeT *t, int mv)
{
    *t=*f;
    switch(mv){

```

```

case FF :
    t->cube[0]=mult[FACE_F_CW][f->cube[2]];
    t->cube[1]=mult[FACE_F_CW][f->cube[0]];
    t->cube[2]=mult[FACE_F_CW][f->cube[3]];
    t->cube[3]=mult[FACE_F_CW][f->cube[1]];
    break;

case FR :
    t->cube[2]=mult[FACE_F_CCW][f->cube[0]];
    t->cube[0]=mult[FACE_F_CCW][f->cube[1]];
    t->cube[3]=mult[FACE_F_CCW][f->cube[2]];
    t->cube[1]=mult[FACE_F_CCW][f->cube[3]];
    break;

case UF :
    t->cube[4]=mult[FACE_U_CW][f->cube[0]];
    t->cube[5]=mult[FACE_U_CW][f->cube[4]];
    t->cube[0]=mult[FACE_U_CW][f->cube[1]];
    t->cube[1]=mult[FACE_U_CW][f->cube[5]];
    break;

case UR :
    t->cube[0]=mult[FACE_U_CCW][f->cube[4]];
    t->cube[4]=mult[FACE_U_CCW][f->cube[5]];
    t->cube[1]=mult[FACE_U_CCW][f->cube[0]];
    t->cube[5]=mult[FACE_U_CCW][f->cube[1]];
    break;

case RF :
    t->cube[1]=mult[FACE_L_CCW][f->cube[3]];
    t->cube[5]=mult[FACE_L_CCW][f->cube[1]];
    t->cube[3]=mult[FACE_L_CCW][f->cube[7]];
    t->cube[7]=mult[FACE_L_CCW][f->cube[5]];
    break;

case RR :
    t->cube[3]=mult[FACE_L_CW][f->cube[1]];
    t->cube[1]=mult[FACE_L_CW][f->cube[5]];
    t->cube[7]=mult[FACE_L_CW][f->cube[3]];
    t->cube[5]=mult[FACE_L_CW][f->cube[7]];
    break;
}
}

```

(4) $2 \times 2 \times 2$ 魔方的全部解

$2 \times 2 \times 2$ 魔方有 $(8!) \times 3^7 = 88179840$ 种不同状态。待搜索的解空间相对较小, 因此可用回溯法在较短时间内找出每种状态的最少旋转次数。经计算, $2 \times 2 \times 2$ 魔方在任何状态下, 变换为目标状态的最少旋转次数不超过 14。对每个状态, 可用 4 位存储其最少旋转次数。用一个 $88179840/8$ 个元素的整型数组 cub222 可以存储 $2 \times 2 \times 2$ 魔方在所有状态下的最少旋转次数。

下面的算法 create222 用前面已讨论过的逐步深化的回溯搜索策略找出 $2 \times 2 \times 2$ 魔方的全部解, 并存储于文件 cub222.data 中。

```
void create222()
{
    cubeT x;
    FILE *fd;
    for(int i=0;i<88179840/8;i++) cub222[i]=-1;
    for (i=0;i<8;i++) x.cube[i]=0;
    for(maxdepth=0;maxdepth<=14;maxdepth++){
        nodes=0;
        backtrack(maxdepth,&x);
        cout<<"maxdepth="<<maxdepth<<" nodes="<<nodes<<endl;
    }
    fd=fopen("cub222.data","w");
    if (fd<=0) cout<<"couldn't open cub222.data"<<endl;
    fwrite(cub222,1,88179840/2,fd);
    fclose(fd);
}
```

实现回溯搜索的算法 backtrack 表述如下。

```
void backtrack(int depth,cubeT *p)
{
    cubeT newp[12];
    int key,x,*entry,offset;
    key=hash(p);
    entry=getent(cub222,key,2);
    x=*entry;
    offset=getoff(key,2);
    if (depth==0){
        if (((x>>offset)&0xF)!=15) return;
        nodes++;
        x &= ~(15<<offset);
        x |= (maxdepth<<offset);
        *entry=x;
        return;
    }
```

```

    }
    for (int i=0;i<12;i++){
        move(p,&newp[i],i);
        backtrack(depth-1,&newp[i]);
    }
}

```

其中，算法 hash 计算魔方状态的编号；getent 和 getoff 用于计算相应编号在数组 cub222 中的存储位置。

```

static int *getent(int *table, int key, int logbits)
{
    return table+(key>>(5-logbits));
}

static int getoff(int key, int logbits)
{
    return (key&(31>>logbits))<<logbits;
}

```

对魔方的 88179840 种不同状态进行编号。每个小立方体有 3 种不同定向，魔方的 8 个小立方体的定向有 3^7 个组合。将每种组合看作一个三进制数，可对这个组合确定惟一编号。每一种组合又有 $8!$ 种排列分布，由此可以确定魔方的每种状态的编号。假设魔方的一种状态的定向为 $r_i, i=0,1,\dots,7$ ；其相应的排列为 π ； π 的字典序值为 $\text{rank}(\pi)$ 。这个魔方状态编号为

$$8! \left(\sum_{i=0}^6 r_i 3^i \right) + \text{rank}(\pi)$$

因此，为了计算魔方状态的编号，应先确定 8 个小立方体的定向以及相应的排列。这个任务由算法 init22 进行初始化计算。

```

void init22()
{
    int a,b,c,e,i,j;
    for(i=0;i<8;i++)
        for(j=0;j<24;j++)
            memb22[i][j]=100;
    for(i=0;i<8;i++){
        cubeT x,y,z;
        for(b=0;b<8;b++)x.cube[b]=24;
        x.cube[i]=0;
        for(a=0;a<13;a++)
            for(b=0;b<13;b++)
                for(c=0;c<13;c++){
                    y=x;

```

```

        if (a!=12) move(&y,&z,a);y=z;
        if (b!=12) move(&y,&z,b);y=z;
        if (c!=12) move(&y,&z,c);y=z;
        for(e=0;e<8;e++)if(y.cube[e]<24)break;
        if (memb22[e][y.cube[e]]==100) memb22[e][y.cube[e]]=i;
    }
}
for(i=0;i<24;i++) orientclass[i]=4;
for(i=0;i<24;i++){
    c=0;
    orientclass[c]=0;
    orientclass[mult[FACE_U_CW][c]]=0;
    orientclass[mult[FACE_U_CCW][c]]=0;
    orientclass[mult[FACE_U_180][c]]=0;
    orientclass[mult[EDGE_3][c]]=0;
    orientclass[mult[EDGE_4][c]]=0;
    orientclass[mult[FACE_L_180][c]]=0;
    orientclass[mult[FACE_F_180][c]]=0;
    c=mult[FACE_F_CCW][0];
    orientclass[c]=1;
    orientclass[mult[FACE_L_CW][c]]=1;
    orientclass[mult[FACE_L_CCW][c]]=1;
    orientclass[mult[FACE_L_180][c]]=1;
    orientclass[mult[EDGE_1][c]]=1;
    orientclass[mult[EDGE_6][c]]=1;
    orientclass[mult[FACE_U_180][c]]=1;
    orientclass[mult[FACE_F_180][c]]=1;
    c=mult[FACE_L_CW][0];
    orientclass[c]=2;
    orientclass[mult[FACE_F_CW][c]]=2;
    orientclass[mult[FACE_F_CCW][c]]=2;
    orientclass[mult[FACE_F_180][c]]=2;
    orientclass[mult[EDGE_8][c]]=2;
    orientclass[mult[EDGE_9][c]]=2;
    orientclass[mult[FACE_U_180][c]]=2;
    orientclass[mult[FACE_L_180][c]]=2;
}
}

```

其中，数组元素 $\text{memb22}[i][j]$ 用于记录第 i 个小立方体在第 j 种变换下的排列序号；数组 $\text{orientclass}[24]$ 记录 24 种小立方体状态的定向分类。

在此基础上，容易确定每个魔方状态的编号。

```
int hash(cubeT *x)
```

```

{
    int p,h,orient;
    int pi[9];
    for(int e=0;e<8;e++)pi[e+1]=memb22[e][x->cube[e]]+1;
    p=permRank(8,pi);
    orient=orientclass[x->cube[6]]; orient*=3;
    orient+=orientclass[x->cube[5]]; orient*=3;
    orient+=orientclass[x->cube[4]]; orient*=3;
    orient+=orientclass[x->cube[3]]; orient*=3;
    orient+=orientclass[x->cube[2]]; orient*=3;
    orient+=orientclass[x->cube[1]]; orient*=3;
    orient+=orientclass[x->cube[0]];
    h=orient*40320+p;
    return h;
}

```

其中, permRank 计算排列的字典序值。

```

int permRank(int n,int *pi)
{
    int *f=new int[n+1];
    int *rho=new int[n+1];
    int j,r=0;
    f[0]=1;
    for(j=1;j<=n;j++){f[j]=f[j-1]*j;rho[j]=pi[j];}
    for(j=1;j<=n;j++){
        r+=(rho[j]-1)*f[n-j];
        for(int i=j+1;i<=n;i++)
            if(rho[i]>rho[j])rho[i]--;
    }
    delete []rho;delete []f;
    return r;
}

```

计算 $2 \times 2 \times 2$ 魔方任意状态的最少旋转次数的主函数如下。

```

int main()
{
    cubeT x;
    int key,i,*entry,offset;
    cub222=new int[88179840/8];
    init();
    read222();
    readCube(&x);
}

```

```

key=hash(&x);
entry=getent(cub222,key,2);
i=*entry;
offset=getoff(key,2);
cout<<((i>>offset)&0xF)<<endl;
return 0;
}

```

其中，read222 读入由 create222 计算好的全部解。

```

void read222()
{
FILE *fd;
fd=fopen("cub222.data","r");
fread(cub222,1,88179840/2,fd);
fclose(fd);
}

```

计算出的全部解的最少旋转次数分布如表 5-1 所示。

表 5-1 $2 \times 2 \times 2$ 魔方全部解的最少旋转次数分布

最少旋转次数	初始状态数
0	1
1	12
2	114
3	924
4	6539
5	39528
6	199926
7	806136
8	2761740
9	8656152
10	22334112
11	32420448
12	18780864
13	2166720
14	6624

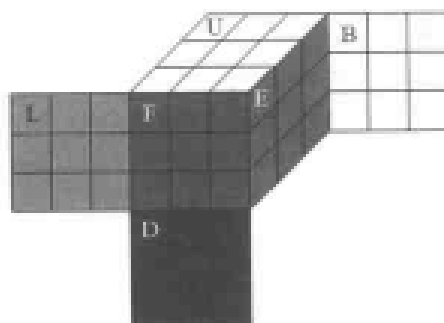


图 5-21 $3 \times 3 \times 3$ 魔方

算法实现题 5-22 魔方 (Rubik's Cube) 问题
(习题 5-31)

★问题描述：

$3 \times 3 \times 3$ 魔方的构造如图 5-21 所示。图中英文字母 U, L, F, R, B, D 分别表示魔方的 6 个面中的上面、左面、前面、右面、后面、底面。魔方的每个面都可以绕其中轴旋转。给定魔方的初始状态，可以经过若干次旋转将魔方变换成每个面都只有一种颜色的状态。绕中轴将一个面旋转 90° 算作一次旋转。试设计一个

算法计算出从初始状态到目标状态（每个面都只有一种颜色的状态）所需的最少旋转次数。

★编程任务：

设计一个算法，对于给定的 $3 \times 3 \times 3$ 魔方的初始状态，计算从初始状态到目标状态（每个面都只有一种颜色的状态）所需的最少旋转次数。

★数据输入：

由文件 input.txt 给出输入数据。 $3 \times 3 \times 3$ 魔方目标状态的每个面都有一种颜色，分别用大写英文字母 W, O, G, R, Y, B 来表示。将 6 个面展开并编号如图 5-22 所示。文件将给定魔方的初始状态的 6 个面，按照其编号依次排列，共有 18 行，每行有 3 个表示方块颜色的大写英文字母。

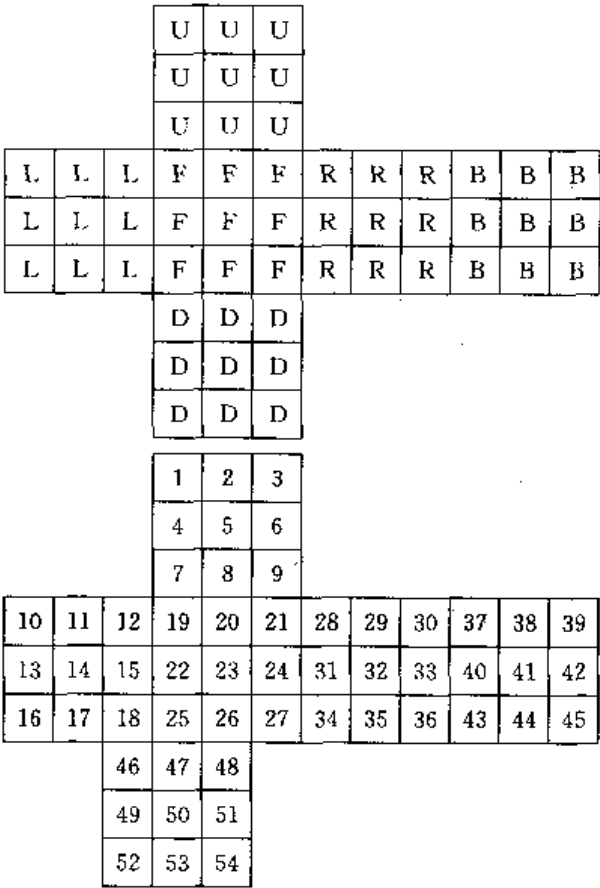


图 5-22 $3 \times 3 \times 3$ 魔方编号

★结果输出：

将计算出的最少旋转次数和最优旋转序列输出到文件 output.txt。文件的第一行是最少旋转次数；接下来是最优旋转序列。用表示各面的大写英文字母表示每个面的顺时针旋转；表示各面的大写英文字母紧接一个“-”表示每个面的逆时针旋转。如果不存在满足要求的旋转序列则输出 “No Solution!”。

输入文件示例

input.txt

G W W

G W W

G R R

W W W

输出文件示例

output.txt

2

L F

```

O O O
O O O
O G G
B G G
B G G
B R R
B R R
B R R
Y Y R
Y Y W
Y Y W
Y O O
Y B B
Y B B

```

分析与解答:

(1) 算法思想

对于魔方的每个面, 有 12 种旋转方式将其变换为另一种状态。这 12 种旋转变换是 F, F⁻, U, U⁻, R, R⁻, L, L⁻, B, B⁻, D, D⁻。因此可将回溯法的搜索空间组织成一棵 12 叉树。

由于需要找最短旋转序列, 采用逐步深化 (Iterative Deepening) 的回溯搜索策略。

(2) 算法实现

3×3×3 魔方由 24 个小立方体组成。8 个角块小立方体有 3 种不同的定向; 12 个边块小立方体有 2 种不同的定向; 每个面的中心小立方体不变定向。用结构 cubeT 表示 20 个可变动向小立方体的定向。

```

typedef struct
{ int cube[20]; } cubeT;

```

20 个小立方体的编号如图 5-23 所示。

用逐步深化的回溯搜索策略的算法表述如下。

```

int *IDsearch(cubeT *x, int maxdepth)
{
    for(int t=0;t<=maxdepth;t++){
        int *res=backtrack(t,x);
        if (res){minmov=t;return res;}
    }
    return 0;
}

```

其中, maxdepth 是最大搜索深度; minmov 记录最少旋转次数。实现搜索的回溯法由 backtrack 完成。

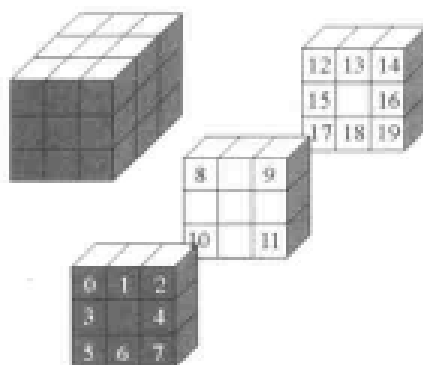


图 5-23 20 个小立方体的编号

```

int *backtrack(int depth, cubeT *p)
{
    cubeT newp[12];
    int *res=0;
    if (depth==0) {
        if (solved(p)) {
            res=new int[24];
            return res;
        }
        else return 0;
    }
    for (int i=0; i<12; i++) {
        move(p, &newp[i], i);
        res=backtrack( depth-1, &newp[i]);
        if (res) {res[depth-1]=i; return res;}
    }
    return 0;
}

```

参数 depth 控制递归深度。旋转到目标状态时，每个小立方体均为 0 号定向，由 solved 判定。move 对每种旋转变换产生魔方的一种新状态。

```

int solved(cubeT *x)
{
    for (int i=0; i<20; i++)
        if (x->cube[i]) return 0;
    return 1;
}

```

在此基础上，算法主函数表述如下：

```

int main()
{

```

```

cubeT x;
readCube(&x);
outsol(IDsearch(&x,maxdepth));
return 0;
}

```

其中, readCube 读入魔方初始状态, 并对整个算法进行初始化运算; outsol 输出计算结果。

```

void readCube(cubeT *x)
{
    char *s=new char[54];
    init();
    for(int i=0;i<54;i++)fin>>s[i];
    setCube(x,s);
}

```

算法 init 和 setCube 稍后进一步说明。

```

void outsol(int *res)
{
    if(!res) cout<<"No Solution! (maxdepth="<<maxdepth<<)"<<endl;
    else{
        cout<<minmov<<endl;
        for(int j=minmov-1;j>=0;j--)
            cout<<moveOut[res[j]]<<" ";cout<<endl;
    }
}

```

moveOut 是表示 12 种旋转变换的字符串数组。

```

char moveOut[12][3]=
{"F", {"F-"}, {"B"}, {"B-"}, {"L"}, {"L-"}, {"R"}, {"R-"}, {"U"}, {"U-"}, {"D"}, {"D-"};

```

(3) 实现算法的数据结构

$3 \times 3 \times 3$ 魔方的结构比较复杂, 因而表示算法的数据结构也较复杂, 这是算法实现的难点。关键问题是每个魔方小立方体的定向, 以及魔方作为一个整体立方体的定向。

标准状态下, 魔方各面的颜色和编号如图 5-24 所示。

面	上 U	左 L	前 F	右 R	后 B	下 D
颜色	白 W	橙 O	绿 G	红 R	黄 Y	蓝 B
编号	0	1	2	3	4	5

图 5-24 魔方各面的颜色和编号

按此编号, 魔方的 24 种定向及其编号如图 5-25 所示。

定向	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
左 L	1	2	4	3	1	1	1	5	0	3	2	0	5	4	5	2	4	0	3	2	4	3	0	5
右 R	3	4	2	1	3	3	3	0	5	1	4	5	0	2	0	4	2	5	1	4	2	1	5	0
前 F	2	3	1	4	0	5	4	2	2	2	0	1	3	0	1	5	5	3	0	1	3	5	4	4
后 B	4	1	3	2	5	0	2	4	4	4	5	3	1	5	3	0	0	1	5	3	1	0	2	2
上 U	0	0	0	0	4	2	5	1	3	5	1	2	2	3	4	3	1	4	2	5	5	4	1	3
下 D	5	5	5	5	2	4	0	3	1	0	3	4	4	1	2	1	3	2	4	0	0	2	3	1

图 5-25 魔方的 24 种定向及其编号

根据每个定向的特点，其命名如图 5-26 所示。

#define ID	0	#define CORNER_2_CW	12
#define FACE_U_CW	1	#define CORNER_2_CCW	13
#define FACE_U_CCW	2	#define CORNER_5_CW	14
#define FACE_U_180	3	#define CORNER_5_CCW	15
#define FACE_L_CW	4	#define CORNER_7_CW	16
#define FACE_L_CCW	5	#define CORNER_7_CCW	17
#define FACE_L_180	6	#define EDGE_1	18
#define FACE_F_CW	7	#define EDGE_3	19
#define FACE_F_CCW	8	#define EDGE_4	20
#define FACE_F_180	9	#define EDGE_6	21
#define CORNER_0_CW	10	#define EDGE_8	22
#define CORNER_0_CCW	11	#define EDGE_9	23

图 5-26 魔方的 24 种定向及其命名

其中，每个定向也可以看作从魔方的标准状态到相应定向的一个旋转变换。

用二维数组 `mult[24][24]` 记录变换矩阵。`mult[i][j]` 表示对第 j 个定向进行第 i 个变换得到的定向。

每个旋转变换作用于小立方体，改变了小立方体的定向。实现旋转变换的算法由 `smove` 完成。

```
static void smove(cubeT *t, int snum)
{
    switch(snum) {
        case ID:
            break;
        case FACE_U_CW:
            swap4(&t->cube[0], &t->cube[12], &t->cube[14], &t->cube[2]);
            swap4(&t->cube[1], &t->cube[8], &t->cube[13], &t->cube[9]);
            swap4(&t->cube[3], &t->cube[15], &t->cube[16], &t->cube[4]);
            swap4(&t->cube[5], &t->cube[17], &t->cube[19], &t->cube[7]);
            swap4(&t->cube[6], &t->cube[10], &t->cube[18], &t->cube[11]);
```

```

    break;
case FACE_U_CCW:
    swap4(&t->cube[0], &t->cube[2], &t->cube[14], &t->cube[12]);
    swap4(&t->cube[1], &t->cube[9], &t->cube[13], &t->cube[8]);
    swap4(&t->cube[3], &t->cube[4], &t->cube[16], &t->cube[15]);
    swap4(&t->cube[5], &t->cube[7], &t->cube[19], &t->cube[17]);
    swap4(&t->cube[6], &t->cube[11], &t->cube[18], &t->cube[10]);
    break;
case FACE_U_180:
    swap2(&t->cube[0], &t->cube[14]);
    swap2(&t->cube[1], &t->cube[13]);
    swap2(&t->cube[2], &t->cube[12]);
    swap2(&t->cube[3], &t->cube[16]);
    swap2(&t->cube[4], &t->cube[15]);
    swap2(&t->cube[5], &t->cube[19]);
    swap2(&t->cube[6], &t->cube[18]);
    swap2(&t->cube[7], &t->cube[17]);
    swap2(&t->cube[8], &t->cube[9]);
    swap2(&t->cube[10], &t->cube[11]);
    break;
case FACE_L_CW:
    swap4(&t->cube[0], &t->cube[5], &t->cube[17], &t->cube[12]);
    swap4(&t->cube[3], &t->cube[10], &t->cube[15], &t->cube[8]);
    swap4(&t->cube[1], &t->cube[6], &t->cube[18], &t->cube[13]);
    swap4(&t->cube[2], &t->cube[7], &t->cube[19], &t->cube[14]);
    swap4(&t->cube[4], &t->cube[11], &t->cube[16], &t->cube[9]);
    break;
case FACE_L_CCW:
    swap4(&t->cube[0], &t->cube[12], &t->cube[17], &t->cube[5]);
    swap4(&t->cube[3], &t->cube[8], &t->cube[15], &t->cube[10]);
    swap4(&t->cube[1], &t->cube[13], &t->cube[18], &t->cube[6]);
    swap4(&t->cube[2], &t->cube[14], &t->cube[19], &t->cube[7]);
    swap4(&t->cube[4], &t->cube[9], &t->cube[16], &t->cube[11]);
    break;
case FACE_L_180:
    swap2(&t->cube[0], &t->cube[17]);
    swap2(&t->cube[1], &t->cube[18]);
    swap2(&t->cube[2], &t->cube[19]);
    swap2(&t->cube[3], &t->cube[15]);
    swap2(&t->cube[4], &t->cube[16]);
    swap2(&t->cube[5], &t->cube[12]);
    swap2(&t->cube[6], &t->cube[13]);
    swap2(&t->cube[7], &t->cube[14]);
    swap2(&t->cube[8], &t->cube[10]);

```

```

    swap2(&t->cube[9], &t->cube[11]);
    break;
case FACE_F_CW:
    swap4(&t->cube[0], &t->cube[2], &t->cube[7], &t->cube[5]);
    swap4(&t->cube[1], &t->cube[4], &t->cube[6], &t->cube[3]);
    swap4(&t->cube[8], &t->cube[9], &t->cube[11], &t->cube[10]);
    swap4(&t->cube[12], &t->cube[14], &t->cube[19], &t->cube[17]);
    swap4(&t->cube[13], &t->cube[16], &t->cube[18], &t->cube[15]);
    break;
case FACE_F_CCW:
    swap4(&t->cube[0], &t->cube[5], &t->cube[7], &t->cube[2]);
    swap4(&t->cube[1], &t->cube[3], &t->cube[6], &t->cube[4]);
    swap4(&t->cube[8], &t->cube[10], &t->cube[11], &t->cube[9]);
    swap4(&t->cube[12], &t->cube[17], &t->cube[19], &t->cube[14]);
    swap4(&t->cube[13], &t->cube[15], &t->cube[18], &t->cube[16]);
    break;
case FACE_F_180:
    swap2(&t->cube[0], &t->cube[7]);
    swap2(&t->cube[1], &t->cube[6]);
    swap2(&t->cube[2], &t->cube[5]);
    swap2(&t->cube[3], &t->cube[4]);
    swap2(&t->cube[8], &t->cube[11]);
    swap2(&t->cube[9], &t->cube[10]);
    swap2(&t->cube[12], &t->cube[19]);
    swap2(&t->cube[13], &t->cube[18]);
    swap2(&t->cube[14], &t->cube[17]);
    swap2(&t->cube[15], &t->cube[16]);
    break;
case CORNER_O_CW:
    swap3(&t->cube[1], &t->cube[3], &t->cube[8]);
    swap3(&t->cube[2], &t->cube[5], &t->cube[12]);
    swap3(&t->cube[4], &t->cube[10], &t->cube[13]);
    swap3(&t->cube[6], &t->cube[15], &t->cube[9]);
    swap3(&t->cube[7], &t->cube[17], &t->cube[14]);
    swap3(&t->cube[11], &t->cube[18], &t->cube[16]);
    break;
case CORNER_O_CCW:
    swap3(&t->cube[1], &t->cube[8], &t->cube[3]);
    swap3(&t->cube[2], &t->cube[12], &t->cube[5]);
    swap3(&t->cube[4], &t->cube[13], &t->cube[10]);
    swap3(&t->cube[6], &t->cube[9], &t->cube[15]);
    swap3(&t->cube[7], &t->cube[14], &t->cube[17]);
    swap3(&t->cube[11], &t->cube[16], &t->cube[18]);
    break;

```

```

case CORNER_2_CW:
    swap3(&t->cube[0], &t->cube[14], &t->cube[7]);
    swap3(&t->cube[1], &t->cube[9], &t->cube[4]);
    swap3(&t->cube[3], &t->cube[13], &t->cube[11]);
    swap3(&t->cube[5], &t->cube[12], &t->cube[19]);
    swap3(&t->cube[6], &t->cube[8], &t->cube[16]);
    swap3(&t->cube[10], &t->cube[15], &t->cube[18]);
    break;

case CORNER_2_CCW:
    swap3(&t->cube[0], &t->cube[7], &t->cube[14]);
    swap3(&t->cube[1], &t->cube[4], &t->cube[9]);
    swap3(&t->cube[3], &t->cube[11], &t->cube[13]);
    swap3(&t->cube[5], &t->cube[19], &t->cube[12]);
    swap3(&t->cube[6], &t->cube[16], &t->cube[8]);
    swap3(&t->cube[10], &t->cube[18], &t->cube[15]);
    break;

case CORNER_5_CW:
    swap3(&t->cube[0], &t->cube[7], &t->cube[17]);
    swap3(&t->cube[1], &t->cube[11], &t->cube[15]);
    swap3(&t->cube[2], &t->cube[19], &t->cube[12]);
    swap3(&t->cube[3], &t->cube[6], &t->cube[10]);
    swap3(&t->cube[4], &t->cube[18], &t->cube[8]);
    swap3(&t->cube[9], &t->cube[16], &t->cube[13]);
    break;

case CORNER_5_CCW:
    swap3(&t->cube[0], &t->cube[17], &t->cube[7]);
    swap3(&t->cube[1], &t->cube[15], &t->cube[11]);
    swap3(&t->cube[2], &t->cube[12], &t->cube[19]);
    swap3(&t->cube[3], &t->cube[10], &t->cube[6]);
    swap3(&t->cube[4], &t->cube[8], &t->cube[18]);
    swap3(&t->cube[9], &t->cube[13], &t->cube[16]);
    break;

case CORNER_7_CW:
    swap3(&t->cube[0], &t->cube[14], &t->cube[17]);
    swap3(&t->cube[1], &t->cube[16], &t->cube[10]);
    swap3(&t->cube[2], &t->cube[19], &t->cube[5]);
    swap3(&t->cube[3], &t->cube[9], &t->cube[18]);
    swap3(&t->cube[4], &t->cube[11], &t->cube[6]);
    swap3(&t->cube[8], &t->cube[13], &t->cube[15]);
    break;

case CORNER_7_CCW:
    swap3(&t->cube[0], &t->cube[17], &t->cube[14]);
    swap3(&t->cube[1], &t->cube[10], &t->cube[16]);
    swap3(&t->cube[2], &t->cube[5], &t->cube[19]);

```



```

    swap3(&t->cube[3], &t->cube[18], &t->cube[9]);
    swap3(&t->cube[4], &t->cube[6], &t->cube[11]);
    swap3(&t->cube[8], &t->cube[15], &t->cube[13]);
    break;
case EDGE_1:
    swap2(&t->cube[0], &t->cube[2]);
    swap2(&t->cube[3], &t->cube[9]);
    swap2(&t->cube[4], &t->cube[8]);
    swap2(&t->cube[5], &t->cube[14]);
    swap2(&t->cube[6], &t->cube[13]);
    swap2(&t->cube[7], &t->cube[12]);
    swap2(&t->cube[10], &t->cube[16]);
    swap2(&t->cube[11], &t->cube[15]);
    swap2(&t->cube[17], &t->cube[19]);
    break;
case EDGE_3:
    swap2(&t->cube[0], &t->cube[5]);
    swap2(&t->cube[1], &t->cube[10]);
    swap2(&t->cube[2], &t->cube[17]);
    swap2(&t->cube[4], &t->cube[15]);
    swap2(&t->cube[6], &t->cube[8]);
    swap2(&t->cube[7], &t->cube[12]);
    swap2(&t->cube[9], &t->cube[18]);
    swap2(&t->cube[11], &t->cube[13]);
    swap2(&t->cube[14], &t->cube[19]);
    break;
case EDGE_4:
    swap2(&t->cube[0], &t->cube[19]);
    swap2(&t->cube[1], &t->cube[11]);
    swap2(&t->cube[2], &t->cube[7]);
    swap2(&t->cube[3], &t->cube[16]);
    swap2(&t->cube[5], &t->cube[14]);
    swap2(&t->cube[6], &t->cube[9]);
    swap2(&t->cube[8], &t->cube[18]);
    swap2(&t->cube[10], &t->cube[13]);
    swap2(&t->cube[12], &t->cube[17]);
    break;
case EDGE_6:
    swap2(&t->cube[0], &t->cube[19]);
    swap2(&t->cube[1], &t->cube[18]);
    swap2(&t->cube[2], &t->cube[17]);
    swap2(&t->cube[3], &t->cube[11]);
    swap2(&t->cube[4], &t->cube[10]);
    swap2(&t->cube[5], &t->cube[7]);

```

```

        swap2(&t->cube[8], &t->cube[16]);
        swap2(&t->cube[9], &t->cube[15]);
        swap2(&t->cube[12], &t->cube[14]);
        break;
    case EDGE_8:
        swap2(&t->cube[0], &t->cube[12]);
        swap2(&t->cube[1], &t->cube[15]);
        swap2(&t->cube[2], &t->cube[17]);
        swap2(&t->cube[3], &t->cube[13]);
        swap2(&t->cube[4], &t->cube[18]);
        swap2(&t->cube[5], &t->cube[14]);
        swap2(&t->cube[6], &t->cube[16]);
        swap2(&t->cube[7], &t->cube[19]);
        swap2(&t->cube[9], &t->cube[10]);
        break;
    case EDGE_9:
        swap2(&t->cube[0], &t->cube[19]);
        swap2(&t->cube[1], &t->cube[16]);
        swap2(&t->cube[2], &t->cube[14]);
        swap2(&t->cube[3], &t->cube[18]);
        swap2(&t->cube[4], &t->cube[13]);
        swap2(&t->cube[5], &t->cube[17]);
        swap2(&t->cube[6], &t->cube[15]);
        swap2(&t->cube[7], &t->cube[12]);
        swap2(&t->cube[8], &t->cube[11]);
        break;
    }
}

```

其中, swap2, swap3, swap4 分别循环交换 2, 3, 4 个元素。

```

static void swap2(int *a, int *b)
{
    int A=*a, B=*b;
    *b=A; *a=B;
}

static void swap3(int *a, int *b, int *c)
{
    int A=*a, B=*b, C=*c;
    *b=A; *c=B; *a=C;
}

static void swap4(int *a, int *b, int *c, int *d)

```

```

{
    int A=*a,B=*b,C=*c,D=*d;
    *b=A;*c=B;*d=C;*a=D;
}

```

旋转变换矩阵 mult[24][24] 由 init 初始化。

```

void init()
{
    cubeT x,y;
    for(int i=0;i<24;i++)
        for(int j=0;j<24;j++){
            for(int k=0;k<20;k++) x.cube[k]=k;
            smove(&x,j);smove(&x,i);
            for(int p=0;p<24;p++){
                for(int k=0;k<20;k++) y.cube[k]=k;
                smove(&y,p);
                if(equal(&x,&y)){mult[i][j]=p;break;}
            }
        }
}

```

其中, equal 用于判断 2 个魔方状态是否相同。

```

static int equal(cubeT *x,cubeT *y)
{
    for(int i=0;i<20;i++)
        if(x->cube[i]!=y->cube[i]) return 0;
    return 1;
}

```

用数组 lc[24],rc[24],fc[24],bc[24],uc[24],dc[24] 表示魔方的 24 种定向的各面颜色号。读入魔方初始数据后, 由 setCube 确定魔方各小立方体的定向。

```

static int lc[24]={1,2,4,3,1,1,1,5,0,3,2,0,5,4,5,2,4,0,3,2,4,3,0,5};
static int rc[24];
static int fc[24];
static int bc[24];
static int uc[24];
static int dc[24];

void setCube(cubeT *x,const char *s)
{
    int cube[6][9];

```

```

int colors[256];
int i, j;
for(i=0; i<20; i++) x->cube[i]=24;
for(i=0; i<6; i++)
    for(j=0; j<9; j++)
        cube[i][j]=s[i*9+j];
for(i=0; i<256; i++) colors[i]=255;
for(i=0; i<6; i++) colors[cube[i][4]]=i;
for(i=0; i<6; i++)
    for(j=0; j<9; j++)
        cube[i][j]=colors[cube[i][j]];
for(i=0; i<6; i++)
    for(j=0; j<9; j++)
        if(cube[i][j]==255) {
            cout<<"bad color in cube!"<<endl;
            return;
        }
for(i=0; i<24; i++) {
    rc[i]=lc[mult[FACE_U_180][i]];
    fc[i]=lc[mult[FACE_U_CW][i]];
    bc[i]=lc[mult[FACE_U_CCW][i]];
    uc[i]=lc[mult[FACE_F_CCW][i]];
    dc[i]=lc[mult[FACE_F_CW][i]];
}

for(i=0; i<24; i++) {
    // 角块
    if(uc[i]==cube[0][6] && lc[i]==cube[1][2] && fc[i]==cube[2][0]) x->cube[0]=i;
    if(uc[i]==cube[0][8] && fc[i]==cube[2][2] && rc[i]==cube[3][0]) x->cube[2]=i;
    if(lc[i]==cube[1][8] && fc[i]==cube[2][6] && dc[i]==cube[5][0]) x->cube[5]=i;
    if(fc[i]==cube[2][8] && rc[i]==cube[3][6] && dc[i]==cube[5][2]) x->cube[7]=i;
    if(uc[i]==cube[0][0] && lc[i]==cube[1][0] && bc[i]==cube[4][2]) x->cube[12]=i;
    if(uc[i]==cube[0][2] && rc[i]==cube[3][2] && bc[i]==cube[4][0]) x->cube[14]=i;
    if(lc[i]==cube[1][6] && bc[i]==cube[4][8] && dc[i]==cube[5][6]) x->cube[17]=i;
    if(rc[i]==cube[3][8] && bc[i]==cube[4][6] && dc[i]==cube[5][8]) x->cube[19]=i;

    // 边块
    if(uc[i]==cube[0][7] && fc[i]==cube[2][1]) x->cube[1]=i;
    if(lc[i]==cube[1][5] && fc[i]==cube[2][3]) x->cube[3]=i;
    if(fc[i]==cube[2][5] && rc[i]==cube[3][3]) x->cube[4]=i;
    if(fc[i]==cube[2][7] && dc[i]==cube[5][1]) x->cube[6]=i;
    if(uc[i]==cube[0][3] && lc[i]==cube[1][1]) x->cube[8]=i;
    if(uc[i]==cube[0][5] && rc[i]==cube[3][1]) x->cube[9]=i;
    if(lc[i]==cube[1][7] && dc[i]==cube[5][3]) x->cube[10]=i;

```

```

        if(rc[i]==cube[3][7] && dc[i]==cube[5][5]) x->cube[11]=i;
        if(uc[i]==cube[0][1] && bc[i]==cube[4][1]) x->cube[13]=i;
        if(lc[i]==cube[1][3] && bc[i]==cube[4][5]) x->cube[15]=i;
        if(rc[i]==cube[3][5] && bc[i]==cube[4][3]) x->cube[16]=i;
        if(bc[i]==cube[4][7] && dc[i]==cube[5][7]) x->cube[18]=i;
    }
    for(i=0;i<20;i++)
        if(x->cube[i]==24){
            cout<<"bad cube description!"<<endl;
            return;
        }
}

```

绕每个面的中轴进行旋转的 12 个变换定义如下。

```

# define FF 0 // 前面顺时针
# define FR 1 // 前面逆时针
# define BF 2 // 后面顺时针
# define BR 3 // 后面逆时针
# define LF 4 // 左面顺时针
# define LR 5 // 左面逆时针
# define RF 6 // 右面顺时针
# define RR 7 // 右面逆时针
# define UF 8 // 上面顺时针
# define UR 9 // 上面逆时针
# define DF 10 // 底面顺时针
# define DR 11 // 底面逆时针

```

实现这 12 个变换的算法 move 表述如下。

```

void move(cube T *f, cubeT *t, int mv)
{
    *t=*f;
    switch(mv){
    case FF :
        t->cube[0]=mult[FACE_F_CW][f->cube[5]];
        t->cube[1]=mult[FACE_F_CW][f->cube[3]];
        t->cube[2]=mult[FACE_F_CW][f->cube[0]];
        t->cube[3]=mult[FACE_F_CW][f->cube[6]];
        t->cube[4]=mult[FACE_F_CW][f->cube[1]];
        t->cube[5]=mult[FACE_F_CW][f->cube[7]];
        t->cube[6]=mult[FACE_F_CW][f->cube[4]];
        t->cube[7]=mult[FACE_F_CW][f->cube[2]];
        break;

```

```

case FR :
    t->cube[5]=mult[FACE_F_CCW][f->cube[0]];
    t->cube[3]=mult[FACE_F_CCW][f->cube[1]];
    t->cube[0]=mult[FACE_F_CCW][f->cube[2]];
    t->cube[6]=mult[FACE_F_CCW][f->cube[3]];
    t->cube[1]=mult[FACE_F_CCW][f->cube[4]];
    t->cube[7]=mult[FACE_F_CCW][f->cube[5]];
    t->cube[4]=mult[FACE_F_CCW][f->cube[6]];
    t->cube[2]=mult[FACE_F_CCW][f->cube[7]];
    break;

case UF :
    t->cube[12]=mult[FACE_U_CW][f->cube[0]];
    t->cube[13]=mult[FACE_U_CW][f->cube[8]];
    t->cube[14]=mult[FACE_U_CW][f->cube[12]];
    t->cube[8]=mult[FACE_U_CW][f->cube[1]];
    t->cube[9]=mult[FACE_U_CW][f->cube[13]];
    t->cube[0]=mult[FACE_U_CW][f->cube[2]];
    t->cube[1]=mult[FACE_U_CW][f->cube[9]];
    t->cube[2]=mult[FACE_U_CW][f->cube[14]];
    break;

case UR :
    t->cube[0]=mult[FACE_U_CCW][f->cube[12]];
    t->cube[8]=mult[FACE_U_CCW][f->cube[13]];
    t->cube[12]=mult[FACE_U_CCW][f->cube[14]];
    t->cube[1]=mult[FACE_U_CCW][f->cube[8]];
    t->cube[13]=mult[FACE_U_CCW][f->cube[9]];
    t->cube[2]=mult[FACE_U_CCW][f->cube[0]];
    t->cube[9]=mult[FACE_U_CCW][f->cube[1]];
    t->cube[14]=mult[FACE_U_CCW][f->cube[2]];
    break;

case RF :
    t->cube[2]=mult[FACE_L_CCW][f->cube[7]];
    t->cube[9]=mult[FACE_L_CCW][f->cube[4]];
    t->cube[14]=mult[FACE_L_CCW][f->cube[2]];
    t->cube[4]=mult[FACE_L_CCW][f->cube[11]];
    t->cube[16]=mult[FACE_L_CCW][f->cube[9]];
    t->cube[7]=mult[FACE_L_CCW][f->cube[19]];
    t->cube[11]=mult[FACE_L_CCW][f->cube[16]];
    t->cube[19]=mult[FACE_L_CCW][f->cube[14]];
    break;

```

case RR :

```
t->cube[7]=mult[FACE_L_CW][f->cube[2]];
t->cube[4]=mult[FACE_L_CW][f->cube[9]];
t->cube[2]=mult[FACE_L_CW][f->cube[14]];
t->cube[11]=mult[FACE_L_CW][f->cube[4]];
t->cube[9]=mult[FACE_L_CW][f->cube[16]];
t->cube[19]=mult[FACE_L_CW][f->cube[7]];
t->cube[16]=mult[FACE_L_CW][f->cube[11]];
t->cube[14]=mult[FACE_L_CW][f->cube[19]];
break;
```

case LF :

```
t->cube[12]=mult[FACE_L_CW][f->cube[17]];
t->cube[8]=mult[FACE_L_CW][f->cube[15]];
t->cube[0]=mult[FACE_L_CW][f->cube[12]];
t->cube[15]=mult[FACE_L_CW][f->cube[10]];
t->cube[3]=mult[FACE_L_CW][f->cube[8]];
t->cube[17]=mult[FACE_L_CW][f->cube[5]];
t->cube[10]=mult[FACE_L_CW][f->cube[3]];
t->cube[5]=mult[FACE_L_CW][f->cube[0]];
break;
```

case LR :

```
t->cube[17]=mult[FACE_L_CCW][f->cube[12]];
t->cube[15]=mult[FACE_L_CCW][f->cube[8]];
t->cube[12]=mult[FACE_L_CCW][f->cube[0]];
t->cube[10]=mult[FACE_L_CCW][f->cube[15]];
t->cube[8]=mult[FACE_L_CCW][f->cube[3]];
t->cube[5]=mult[FACE_L_CCW][f->cube[17]];
t->cube[3]=mult[FACE_L_CCW][f->cube[10]];
t->cube[0]=mult[FACE_L_CCW][f->cube[5]];
break;
```

case DF :

```
t->cube[5]=mult[FACE_U_CCW][f->cube[17]];
t->cube[6]=mult[FACE_U_CCW][f->cube[10]];
t->cube[7]=mult[FACE_U_CCW][f->cube[5]];
t->cube[10]=mult[FACE_U_CCW][f->cube[18]];
t->cube[11]=mult[FACE_U_CCW][f->cube[6]];
t->cube[17]=mult[FACE_U_CCW][f->cube[19]];
t->cube[18]=mult[FACE_U_CCW][f->cube[11]];
t->cube[19]=mult[FACE_U_CCW][f->cube[7]];
break;
```

```

case DR :
    t->cube[17]=mult[FACE_U_CW][f->cube[5]];
    t->cube[10]=mult[FACE_U_CW][f->cube[6]];
    t->cube[5]=mult[FACE_U_CW][f->cube[7]];
    t->cube[18]=mult[FACE_U_CW][f->cube[10]];
    t->cube[6]=mult[FACE_U_CW][f->cube[11]];
    t->cube[19]=mult[FACE_U_CW][f->cube[17]];
    t->cube[11]=mult[FACE_U_CW][f->cube[18]];
    t->cube[7]=mult[FACE_U_CW][f->cube[19]];
    break;

case BF :
    t->cube[14]=mult[FACE_F_CCW][f->cube[19]];
    t->cube[13]=mult[FACE_F_CCW][f->cube[16]];
    t->cube[12]=mult[FACE_F_CCW][f->cube[14]];
    t->cube[16]=mult[FACE_F_CCW][f->cube[18]];
    t->cube[15]=mult[FACE_F_CCW][f->cube[13]];
    t->cube[19]=mult[FACE_F_CCW][f->cube[17]];
    t->cube[18]=mult[FACE_F_CCW][f->cube[15]];
    t->cube[17]=mult[FACE_F_CCW][f->cube[12]];
    break;

case BR :
    t->cube[19]=mult[FACE_F_CW][f->cube[14]];
    t->cube[16]=mult[FACE_F_CW][f->cube[13]];
    t->cube[14]=mult[FACE_F_CW][f->cube[12]];
    t->cube[18]=mult[FACE_F_CW][f->cube[16]];
    t->cube[13]=mult[FACE_F_CW][f->cube[15]];
    t->cube[17]=mult[FACE_F_CW][f->cube[19]];
    t->cube[15]=mult[FACE_F_CW][f->cube[18]];
    t->cube[12]=mult[FACE_F_CW][f->cube[17]];
    break;
}
}

```

算法实现题 5-23 算 24 点问题

★问题描述:

给定 4 个正整数, 用算术运算符 +, -, *, / 将这 4 个正整数连接起来, 使最终的得数恰为 24。

★编程任务:

对于给定的 4 个正整数, 给出计算 24 的算术表达式。

★数据输入:

由文件 input.txt 给出输入数据。第一行有 4 个正整数。

★结果输出:

将计算 24 的算术表达式输出到文件 output.txt。如果有多个满足要求的表达式, 只要输出 1 组, 每一步算式用分号隔开。如果无法得到 24, 则输出 “No Solution!”。

输入文件示例

input.txt

4 24

1 2 3 7

输出文件示例

output.txt

2+1=3; 7*3=21; 21+3=24;

分析与解答:

对输入的 4 个数 a,b,c,d, 与运算符 +, -, *, / 的所有组合用回溯法进行搜索。

```
void search(int k,vector<float>d)
{
    int i,j,m,t;
    float a,b;
    vector<float>e(5,0);
    if(k==1){if(int((d[1]-24)*10000)==0){outanswer();found=true;}}
    else {
        for(i=1;i<=k-1;i++){
            for(j=i+1;j<=k;j++){
                a=d[i];b=d[j];
                if(a<b) swap(a,b);
                for(m=1,t=0;m<=k;m++){
                    if((m!=i)&&(m!=j)) e[t+t]=d[m];
                    r[5-k][1]=a;r[5-k][3]=b;r[5-k][4]=-1;
                    for(m=1;m<=5;m++){
                        switch(m){
                            case 1: r[5-k][4]=a+b;break;
                            case 2: r[5-k][4]=a-b;break;
                            case 3: r[5-k][4]=a*b;break;
                            case 4: if(b!=0) r[5-k][4]=a/b;break;
                            case 5: if(a!=0) r[5-k][4]=b/a;break;
                        }
                    }
                    r[5-k][2]=m;
                    if(r[5-k][4]!=-1){
                        e[t+1]=r[5-k][4];
                        search(k-1,e);
                    }
                }
            }
        }
    }
}
```

其中, outanswer 输出找到的表达式。

```
void outanswer()
{
    ostringstream msg;
    for(int i=1;i<=3;i++) {
        switch (int(r[i][2])){
            case 1: msg<<r[i][1]<<"+"<<r[i][3];break;
            case 2: msg<<r[i][1]<<"-"<<r[i][3];break;
            case 3: msg<<r[i][1]<<"*"<<r[i][3];break;
            case 4: msg<<r[i][1]<<"/"<<r[i][3];break;
            case 5: msg<<r[i][3]<<"/"<<r[i][1];break;
        }
        msg<<"="<<r[i][4]<<" ";
    }
    string ans=msg.str();
    if(answer.find(ans)==answer.end()) answer.insert(ans);
}
```

实现算法的主函数如下。

```
int main()
{
    vector<float>d(5);
    for(int i=1;i<=4;i++)fin>>d[i];
    found=false;
    search(4,d);
    for(set<string>::iterator it=answer.begin();it!=answer.end();it++)
        fout<<*it<<endl;
    if(!found) fout<<"No answer!"<<endl;
    return 0;
}
```

算法实现题 5-24 算 m 点问题

★问题描述:

给定 k 个正整数,用算术运算符 $+$, $-$, $*$, $/$ 将这 k 个正整数连接起来,使最终的得数恰为 m 。

★编程任务:

对于给定的 k 个正整数,给出计算 m 的算术表达式。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 k 和 m ,表示给定 k 个正整数,且最终的得数恰为 m 。接下来的一行中,有 k 个正整数。

★结果输出:

将计算 m 的算术表达式输出到文件 output.txt。如果有多个满足要求的表达式,只要输

出一组,每一步算式用分号隔开。如果无法得到 m 则输出“No Solution!”。

输入文件示例

输出文件示例

input.txt

output.txt

5 125

77 * 3 = 21; 21 * 12 = 252; 252 - 2 = 250; 250 / 2 = 125;

2 2 12 3

分析与解答:

对输入的 k 个正整数与运算符 +, -, *, / 的所有组合用回溯法进行搜索。

```
void search(int k, vector<float> d)
{
    vector<float> e(kk+1, 0);
    if (k==1) {if (int((d[1]-mm)*10000)==0) {outanswer();found=true;}}
    else {
        for (int i=1; i<=k-1; i++)
            for (int j=i+1; j<=k; j++) {
                float a=d[i], b=d[j];
                if (a<b) swap(a, b);
                for (int m=1, t=0; m<=k; m++)
                    if ((m!=i) && (m!=j)) e[t]=d[m];
                r[kk+1-k][1]=a; r[kk+1-k][3]=b; r[kk+1-k][4]=-1;
                for (m=1; m<=6; m++) {
                    switch (m) {
                        case 1: r[kk+1-k][4]=a+b; break;
                        case 2: r[kk+1-k][4]=a-b; break;
                        case 3: r[kk+1-k][4]=a*b; break;
                        case 4: if (b!=0) r[kk+1-k][4]=a/b; break;
                        case 5: if (a!=0) r[kk+1-k][4]=b/a; break;
                        case 6: r[kk+1-k][4]=b-a; break;
                    }
                }
                r[kk+1-k][2]=m;
                if (r[kk+1-k][4]!=-1) {
                    e[t+1]=r[kk+1-k][4];
                    search(k-1, e);
                }
            }
    }
}
```

其中, outanswer 输出找到的表达式。

```

void outanswer()
{
    ostringstream msg;
    for (int i=1; i<kk; i++) {
        switch (int(r[i][2])){
            case 1: msg<<((r[i][1]<0)?("":""))<<r[i][1]<<((r[i][1]<0)?("":""))<<
                    "+"<<((r[i][3]<0)?("":""))<<r[i][3]<<((r[i][3]<0)?("":""))<<
                    break;
            case 2: msg<<((r[i][1]<0)?("":""))<<r[i][1]<<((r[i][1]<0)?("":""))<<
                    "-"<<((r[i][3]<0)?("":""))<<r[i][3]<<((r[i][3]<0)?("":""))<<
                    break;
            case 3: msg<<((r[i][1]<0)?("":""))<<r[i][1]<<((r[i][1]<0)?("":""))<<
                    "*"<<((r[i][3]<0)?("":""))<<r[i][3]<<((r[i][3]<0)?("":""))<<
                    break;
            case 4: msg<<((r[i][1]<0)?("":""))<<r[i][1]<<((r[i][1]<0)?("":""))<<
                    "/"<<((r[i][3]<0)?("":""))<<r[i][3]<<((r[i][3]<0)?("":""))<<
                    break;
            case 5: msg<<((r[i][3]<0)?("":""))<<r[i][3]<<((r[i][3]<0)?("":""))<<
                    "/"<<((r[i][1]<0)?("":""))<<r[i][1]<<((r[i][1]<0)?("":""))<<
                    break;
            case 6: msg<<((r[i][3]<0)?("":""))<<r[i][3]<<((r[i][3]<0)?("":""))<<
                    "-"<<((r[i][1]<0)?("":""))<<r[i][1]<<((r[i][1]<0)?("":""))<<
                    break;
        }
        msg<<"="<<r[i][4]<<" ";
    }
    string ans=msg.str();
    answer.insert(ans);
}

```

readin 读入初始数据并作初始化计算。

```

void readin(vector<float> &d)
{
    fin>>kk>>mm;
    d.resize(kk+1);
    r.resize(kk, kk+1);
    for (int i = 1; i <= kk; i++)fin>>d[i];
    found=false;
}

```

实现算法的主函数如下。

```

void main()
{
    vector<float> d;
    readin(d);
    search(kk, d);
    for(set<string>::iterator it=answer.begin(); it!=answer.end(); it++)

```

```

        fout<<*it<<endl;
    if(! found) fout<<"No Solution!"<<endl;
}

```

算法实现题 5-25 双轨车皮编序问题

★问题描述:

在一个列车调度站中, 2 条轨道连接到 2 条侧轨处, 形成 2 个铁路转轨栈。其中左边轨道为车皮入口, 编号为 A; 右边轨道为出口, 编号为 D, 2 个铁路转轨栈分别编号为 C 和 D, 如图 5-27 所示。编号为 a, b, ..., 的 n 个车皮依序停放在车皮入口处。调度室要安排各车皮进出栈次序, 使得在出口处各车皮按照预先指定的顺序依次出站。车皮移动时只能按照从左到右的方向移动。

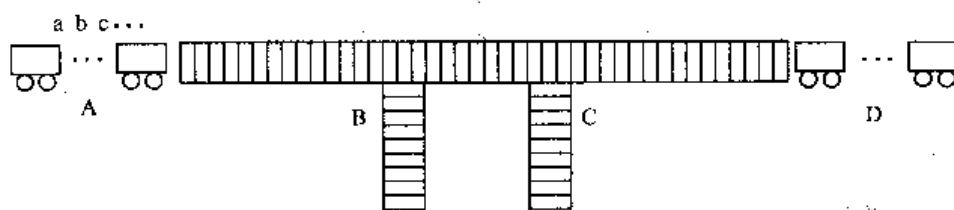


图 5-27 铁路转轨栈

★编程任务:

给定车皮数 n , 以及各车皮的出站顺序, 编程计算最优调度方案, 使得移动车皮的总次数最少。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n , 表示车皮数。接下来的 1 行中, 是预先指定的车皮出站顺序。

★结果输出:

将计算的最优调度方案输出到文件 output.txt。文件的第一行是最少移动次数 m 。接下来的 m 行是对应于最优方案的 m 次移动。每次移动用形如 “c X Y” 的 3 个字符来表示, 其中 c 表示车皮编号, X 表示起始栈轨号, Y 表示目标栈轨号。如果无法调度则输出 “No Solution!”。

输入文件示例

input.txt

3

abc

输出文件示例

output.txt

5

c A B

b A B

a A D

b B D

c B D

分析与解答:

将车皮的移动分为两类: 进入出口和进入侧轨。对于进入侧轨的情形, 递归回溯搜索。

move1 实现第 1 类移动。

```
void move1(short &topa, short &topb, short &topc, short &topd, short &ks)
{
    bool t=true;
    while(t){
        t=false;
        if (topd>n){
            if (best>ks){best=ks;copyans(ks);}
            count++;
            return;
        }
        if (sta[topa] == stdd[topd]) {go('A', topa, topd, ++ks);t=true;}
        if (stb[topb] == stdd[topd]) {go('B', topb, topd, ++ks);t=true;}
        if (stc[topc] == stdd[topd]) {go('C', topc, topd, ++ks);t=true;}
    }
}
```

go 实现具体移动。

```
void go(char from, short &topdrom, short &topto, int ks)
{
    char c=sta[topdrom];
    if (from=='B')c=stb[topdrom];
    if (from=='C')c=stc[topdrom];
    solu[ks].code = c;
    solu[ks].source = from;
    solu[ks].target = 'D';
    topdrom--;topto++;
}
```

copyans 复制搜索到的解。

```
void copyans(int k)
{
    for(int i=1;i<=k;i++){
        opt[i].code=solu[i].code;
        opt[i].source=solu[i].source;
        opt[i].target=solu[i].target;
    }
}
```

move2 实现第 2 类移动。

```

void move2 (short topa2, short topb2, short topc2, short topd2, short ks2)
{
    short tp, ksl;
    short topa, topb, topc, topd;
    char gb2[MAXLEN], gc2[MAXLEN];
    topa=topa2; topb=topb2; topc=topc2; topd=topd2; ks=ks2;
    move1(topa, topb, topc, topd, ks);
    copy(stb, gb2); copy(stc, gc2);
    ksl=ks+1;
    for (tp=1; tp<=3; tp++)
        switch(tp) {
            case 1://a-->b
                if (topa>0) {
                    solu[ksl].code = sta[topa];
                    solu[ksl].source = 'A';
                    solu[ksl].target = 'B';
                    stb[++topb] = sta[topa--];
                    move2(topa, topb, topc, topd, ksl);
                    copy(gb2, stb); copy(gc2, stc);
                    topa++; topb--;
                }
                break;
            case 2://a-->c
                if ((ord(sta[topa])<ord(stc[topc])) && (topa>0)) {
                    solu[ksl].code = sta[topa];
                    solu[ksl].source = 'A';
                    solu[ksl].target = 'C';
                    stc[++topc] = sta[topa--];
                    move2(topa, topb, topc, topd, ksl);
                    copy(gb2, stb); copy(gc2, stc);
                    topa++; topc--;
                }
                break;
            case 3://b-->c
                if ((ord(stb[topb])<ord(stc[topc])) && (topb>0)) {
                    solu[ksl].code = stb[topb];
                    solu[ksl].source = 'B';
                    solu[ksl].target = 'C';
                    stc[++topc] = stb[topb--];
                    move2(topa, topb, topc, topd, ksl);
                    copy(gb2, stb); copy(gc2, stc);
                    topb++; topc--;
                }
        }
}

```

```

        break;
    }
}

```

copy 复制数组。

```

void copy(char *a, char *b)
{
    for(int i=0; i<MAXLEN; i++) b[i]=a[i];
}

```

ord 计算编号。

```

short ord(char c)
{
    if (c=='0') return 0;
    if (c=='1') return 100;
    return order[c-'a'+1];
}

```

实现算法的主函数如下。

```

int main()
{
    short topa, topb, topc, topd;
    fin>>n;
    for (int i = 1; i <= n; i++) fin>>stdd[i];
    for (i=1; i<=n; i++)
        for (char j='a'; j<='a'+n-1; j++)
            if (j==stdd[i]) order[j-'a'+1]=i;
    for (i=1; i<MAXSTEPS; i++) solu[i].code='0';
    for (i=1; i<=n; i++) sta[i]='a'+i-1;
    topa=n; topb=0; topc=0; topd=1;
    sta[0]='0'; stb[0]='0'; stc[0]='1';
    move2(topa, topb, topc, topd, ks);
    if (best==MAXSHORT) cout<<"No Solution!"<<endl;
    else {
        cout<<best<<endl;
        output(opt, best);
    }
    return 0;
}

```


算法实现题 5-26 多轨车皮编序问题

★问题描述:

在一个列车调度站中, k 条轨道连接到 k 条侧轨处, 形成 k 个铁路转轨栈, 从左到右依次编号为 $1, 2, \dots, k$, 其中左边轨道为车皮入口, 编号为 0 ; 右边轨道为出口, 编号为 $k+1$ 。当 $k=2$ 时, 如图 5-28 所示。编号为 $1, 2, \dots, n$ 的 n 个车皮散乱地停放在编号为 $0, 1, 2, \dots, k$ 的栈轨处。调度室要安排各车皮进出站次序, 使得在出口处各车皮按照其编号次序 $1, 2, \dots, n$ 依次出站。车皮移动时只能按照从左到右的方向移动。

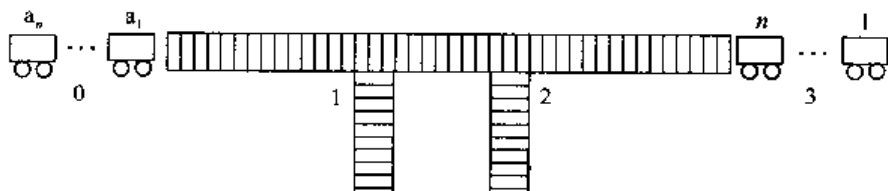


图 5-28 铁路转轨栈

★编程任务:

给定车皮数 n 和侧轨数 k , 以及各车皮的位置, 编程计算最优调度方案, 使得移动车皮的总次数最少。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 k , 表示车皮数为 n 和侧轨数为 k 。接下来的 $k+1$ 行中, 表示编号为 $0, 1, 2, \dots, k$ 的栈轨处按照从下到上的顺序停放的车皮序列。每行的第一个数表示该栈轨处的车皮数, 紧接着是车皮序列。

★结果输出:

将计算的最优调度方案输出到文件 output.txt。文件的第 1 行是最少移动次数 m 。接下来的 m 行是对应于最优方案的 m 次移动。每次移动用形如 “c x y” 的 3 个整数来表示, 其中 c 表示车皮编号, x 表示起始栈轨号, y 表示目标栈轨号。如果无法调度则输出 “No Solution!”。

输入文件示例

输出文件示例

input.txt

output.txt

6 2

9

4 4 1 5 3

3 0 1

2 6 2

5 0 2

0

1 0 3

3 1 2

2 1 3

3 2 3

4 0 3

5 2 3

6 1 3

分析与解答:

与双轨车皮编序问题解法类似, 将车皮的移动分为两类: 进入出口和进入侧轨。对于进入侧轨的情形, 递归回溯搜索。

moves 实现第 1 类移动。

```
bool moves(int &count)
{
    bool t=true;
    while(t){
        t=false;
        if (top[k+1]==n){
            if (best>count){best=count;copyans(count);}
            scount++;
            return true;
        }
        for(int i=0;i<=k;i++){
            if (top[i]>0 && sta[i][top[i]] == sta[k+1][top[k+1]+1]) {
                move(i,k+1,++count);t=true;
            }
        }
    }
    return false;
}
```

move 实现具体移动。

```
void move(int i, int j, int count)
{
    solu[count].code = sta[i][top[i]];
    solu[count].source = i;
    solu[count].target = j;
    sta[j][top[j]+1]=sta[i][top[i]];
    top[j]++;top[i]--;
}
```

backtrack 实现第 2 类移动。

```
void backtrack(int count)
{
    int *stop,**ssta;
    stop=new int[k+2];
    Make2DArray(ssta,k+2,n+1);
    int count1=count;
    if (moves(count1)) return;
    save(ssta,stop);
    count1++;
    for(int i=0;i<k;i++){
        for(int j=i+1;j<=k;j++){
            if(top[i]>0 && (j<k || j==k && sta[i][top[i]]<sta[j][top[j]])){
```

```

        move(i, j, count1);
        backtrack(count1);
        restore(ssta, stop);
    }
    Delete2DArray(ssta, k+2);
    delete []stop;
}

```

save 和 restore 分别完成保存和恢复状态的工作。

```

void save(int **a, int *b)
{
    for(int i=0;i<k+2;i++){
        b[i]=top[i];
        for(int j=0;j<=n;j++) a[i][j]=sta[i][j];
    }
}

void restore(int **a, int *b)
{
    for(int i=0;i<k+2;i++){
        top[i]=b[i];
        for(int j=0;j<=n;j++) sta[i][j]=a[i][j];
    }
}

```

实现算法的主函数如下。

```

int main()
{
    fin>>n>>k;
    for (int i = 0; i <= k; i++){
        fin>>top[i];
        for(int j=1;j<=top[i];j++)fin>>sta[i][j];
    }
    for (i=1;i<=n+1;i++) sta[k+1][i]=i;
    top[k+1]=0;
    sta[k][0]=n+1;
    backtrack(0);
    if (best==MAXSHORT) cout<<"No Solution!"<<endl;
    else {
        cout<<best<<endl;
        output(opt, best);
    }
    return 0;
}

```

算法实现题 5-27 部落卫队问题 (习题 5-6)

★问题描述:

原始部落 byteland 中的居民们为了争夺有限的资源,经常发生冲突。几乎每个居民都有他的仇敌。部落酋长为了组织一支保卫部落的队伍,希望从部落的居民中选出最多的居民入伍,并保证队伍中任何 2 个人都不是仇敌。

★编程任务:

给定 byteland 部落中居民间的仇敌关系,编程计算组成部落卫队的最佳方案。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m ,表示 byteland 部落中有 n 个居民,居民间有 m 个仇敌关系。居民编号为 $1, 2, \dots, n$ 。接下来的 m 行中,每行有 2 个正整数 u 和 v ,表示居民 u 与居民 v 是仇敌。

★结果输出:

程序运行结束时,将计算出的部落卫队的最佳组建方案输出到文件 output.txt 中。文件的第 1 行是部落卫队的人数;文件的第 2 行是卫队组成 $x_i, 1 \leq i \leq n, x_i = 0$ 表示居民 i 不在卫队中, $x_i = 1$ 表示居民 i 在卫队中。

输入文件示例

input.txt

7 10

1 2

1 4

2 4

2 3

2 5

2 6

3 5

3 6

4 5

5 6

输出文件示例

output.txt

3

1 0 1 0 0 0 1

分析与解答:

本题即习题 5-6,设计解最大独立集问题的回溯法。与主教材中最大团问题的解法十分相似。

```
void AdjacencyGraph::maxInde(int i)
{
    if(i > n){
        for(int j=1; j<=n; j++) bestx[j]=x[j];
        bestn=cn;
        return;
    }
```

```

    }
    int OK=1;
    for(int j=1;j<i;j++)
        if (x[j] && a[i][j]!=NoEdge) {OK=0;break;}
    if(OK) {
        x[i]=1;cn++;
        maxInde(i+1);
        x[i]=0;cn--;
    }
    if(cn+n-i>bestn) {x[i]=0;maxInde(i+1);}
}

```

MaxInde 进行初始化，并用回溯法求解。

```

int AdjacencyGraph::MaxInde(int v[])
{
    x=new int[n+1];
    for(int i=0;i<=n;i++)x[i]=0;
    cn=0;
    bestn=0;
    bestx=v;
    maxInde(1);
    delete []x;
    return bestn;
}

```

算法实现题 5-28 虫蚀算式问题

★问题描述：

虫蚀算式是指古书中算式的一部分被虫蛀了。虫蚀算式问题是根据虫蚀算式剩下的数字，逻辑推断被虫蛀了的数字。例如，

$$\begin{array}{r}
 43?98650?45 \\
 + \quad 8468?6633 \\
 \hline
 44445506978
 \end{array}$$

其中，“?”表示虫蛀的数字。根据此虫蚀算式，容易推断出，第1行的2个虫蛀数字分别是5和3，第2行的虫蛀数字是5。

一般情况下，虫蚀算式问题假设算式中所有数字都被虫蛀了，但是知道虫蚀算式中哪些数字相同。另外还知道虫蚀算式是 n 进制加法算式。虫蚀算式中的3个数都是 n 位数，且允许前导0。

★编程任务：

对于给定的虫蚀算式，编程计算算式中的虫蚀数字。

★数据输入：

由文件 input.txt 给出输入数据。文件有4行，第1行有1个正整数 n ($n \leq 26$)，表示所

给的虫蚀算式是 n 进制加法算式。其后 3 行中, 每行有 1 个由 n 个大写英文字母组成的字符串, 分别表示虫蚀算式中的 2 个加数及其和。相同的英文字母代表相同的数字。

★结果输出:

将计算出的虫蚀数字输出到文件 output.txt。在文件的第 1 行输出英文字母 A,B,C,..., 所表示的数字。

输入文件示例	输出文件示例
input.txt	output.txt
5	1 0 3 4 2
ABCED	
BDACE	
EBBAA	

分析与解答:

此题要找的是英文字母 A,B,C,..., 与数字 0,1,2,..., 的对应关系, 其解空间显然是一棵排列树, 可以套用搜索排列树的回溯法框架。

用类 Alpha 表示算法结构。

```
class Alpha{
    friend void Compute(int * *, int);
private:
    bool Backtrack(int i);
    void construct();
    bool constrain(int i);
    bool oka();
    bool vio(link p);
    int *x, n, **B;
    link *cut;
};
```

其中, link 是结点 node 的指针。Node 中的 3 个数 a,b,c, 表示加法竖式对应位的 3 个数。

```
typedef struct node *link;
typedef struct node{
    int a,b,c;
    link next;
}Node;
```

回溯法的主体是 Backtrack。

```
bool Alpha::Backtrack(int i)
{
    if(i==n-1){
        if(oka()){ for(int j=0;j<n;j++) cout<<x[j]<<" ";cout<<endl;return true;}
        else return false;
    }
```

```

    }
    else
        for(int j=i;j<n;j++){
            Swap(x[i],x[j]);
            if(constrain(i) && Backtrack(i+1))return true;
            Swap(x[i],x[j]);
        }
    return false;
}

```

其中, oka 判断最终得到的算式是否成立。

```

bool Alpha::oka()
{
    int carr=0;
    for(int i=n-1;i>=0;i--){
        int sum=x[B[i][0]]+x[B[i][1]]+carr;
        if(sum%n!=x[B[i][2]]) return false;
        carr=sum/n;
    }
    return true;
}

```

constrain 判断部分算式是否成立。

```

bool Alpha::constrain(int i)
{
    for(int j=0;j<=i;j++) if(vio(cut[j]))return false;
    return true;
}

bool Alpha::vio(link p)
{
    while(p){
        if((x[p->a]+x[p->b])%n!=x[p->c]&&(x[p->a]+x[p->b]+1)% n!=x[p->c])
            return true;
        p=p->next;
    }
    return false;
}

```

Compute 完成计算。

```

void Compute(int **B, int n)

```

```

{
    Alpha X;
    X.x=new int[n];
    X.cut=new link[n];
    X.n=n;
    X.B=B;
    X.construct();
    for(int i=0;i<n;i++) X.x[i]=i;
    X.Backtrack(0);
    delete [] X.x;
}

```

construct 进行结构初始化。

```

void Alpha::construct()
{
    link p;
    for(int i=0;i<n;i++)cut[i]=0;
    for(i=0;i<n;i++){
        int maxi=B[i][0];
        for(int j=1;j<3;j++) if(maxi<B[i][j])maxi=B[i][j];
        p=new Node;
        p->a=B[i][0];p->b=B[i][1];p->c=B[i][2];
        p->next =cut[maxi];
        cut[maxi]=p;
    }
}

```

实现算法的主函数如下。

```

int main()
{
    int n;
    string a;
    cin>>n;
    int **B;
    Make2DArray(B,n,3);
    for(int j=0;j<3;j++){
        cin>>a;
        for(int i=0;i<n;i++) B[i][j]=a[i]-'A';
    }
    Compute(B,n);
    return 0;
}

```

算法实现题 5-29 完备环序列问题

★问题描述:

长度为 n 的环序列定义为含有 n 个互不相同的元素且首尾相接的环状序列。如果环序列中连续若干个数的和能形成一个连续的整数序列 $1, 2, \dots, m$, 则称该环序列为一个完备的 (n, m) 序列。对于给定的 n , 计算存在完备 (n, m) 序列的 m 的最大值。同时, 计算出有多少个不同的完备 (n, m) 序列。

★编程任务:

对于给定的正整数 n , 计算存在完备 (n, m) 序列的 m 的最大值; 计算有多少个不同的完备 (n, m) 序列。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 $n, 1 \leq n \leq 10$ 。

★结果输出:

将计算出的最大值 m 和不同的完备 (n, m) 序列的个数 k , 以及所有不同的完备 (n, m) 序列输出到文件 output.txt。文件的第 1 行是 m 和 k ; 接下来的 k 行, 每行是一个完备 (n, m) 序列。

输入文件示例

input.txt

2

输出文件示例

output.txt

3 1

1 2

分析与解答:

长度为 n 的环序列最多可以产生 $r = n(n-1) + 1$ 个不同的数, 因此 m 的最大值不超过 r 。用回溯法搜索所有可能的排列。

```
void backtrack(int m)
{
    int y=r;
    for(int x=1;x<=m-1;x++) y-=a[x];
    for(x=2;x<=y;x++)
        if(b[x]==0){
            a[m]=x;b[x]=1;
            if(m==n){if(oka()) ans++;}
            else backtrack(m+1);
            b[x]=0;
        }
}
```

其中, oka 判断产生的整数是否为连续整数序列。maxi 记录产生的连续整数序列的长度。

```
bool oka()
{
    for(int i=1;i<=r;i++) t[i]=0;
```

```

        for(i=1;i<=n;i++){
            int k=a[i];
            t[k]=1;
            for(int j=1;j<=n-1;j++){
                if(i+j<=n) k+=a[i+j];
                else k+=a[i+j-n];
                t[k]=1;
            }
        }
        for(i=1;i<=r;i++){
            if(t[i]==0){if(maxi<i-1) maxi=i-1;return false;}
        }
        maxi=r;
        return true;
    }

```

ans 记录找到的解。

```

void ans()
{
    for(int i=1;i<=n;i++) f[count][i]=a[i];
    count++;
}

```

init 进行初始化计算。

```

void init()
{
    cin>>n;
    r=n*(n-1)+1;
    a=new int[n+1];
    b=new int[r+1];
    t=new int[r+1];
    f.resize(20,n+1);
    for(int i=0;i<=r;i++)b[i]=0;
    a[1]=1;b[1]=1;
}

```

out 输出所有解。

```

void out()
{
    cout<<r<<" "<<count<<endl;
    for(int i=0;i<count;i++){
        for(int j=1;j<=n;j++) cout<<f[i][j]<<" ";
    }
}

```

```

        cout<<endl;
    }
}

```

实现算法的主函数如下。

```

int main()
{
    init();
    backtrack(2);
    if(maxi<r){r=maxi;backtrack(2);}
    out();
    return 0;
}

```

算法实现题 5-30 离散 0-1 串问题

★问题描述:

(n, k) 0-1 串定义为: 长度为 n 的 0-1 串, 其中不含 k 个连续的相同子串。对于给定的正整数 n 和 k , 计算 (n, k) 0-1 串的个数。

★编程任务:

对于给定的正整数 n 和 k , 计算 (n, k) 0-1 串的个数。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 k , $1 \leq k, n \leq 40$ 。

★结果输出:

将计算出的 (n, k) 0-1 串的个数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
2 3	4

分析与解答:

此题可看作子集选取问题, 并可套用于子集树回溯搜索算法框架。由于对称性, 只要考察首字符为 0 的情况, 最后将找到的符合条件的 0-1 串个数加倍。

```

void backtrack(int lev)
{
    if(lev>n) {tot+=2;return;}
    for(int i=0;i<2;i++){
        bstr[lev]=i;
        if(bstrok(lev))backtrack(lev+1);
    }
}

```

其中, bstrok 判断当前子串是否满足要求。

```

bool bstrok(int lev)
{
    for(int i=0;i<k;i++) x[i]=lev-i;
    while (x[k-1]>0) {
        if(same()) return false;
        for(int i=0;i<k;i++) x[i]-=i+1;
    }
    return true;
}

```

same 判断当前子串是否重复。

```

bool same()
{
    int len=x[0]-x[1];
    for(int i=0;i<len;i++)
        for(int j=1;j<k;j++) if(bstr[x[j]+i]!=bstr[x[j-1]+i]) return false;
    return true;
}

```

实现算法的主函数如下。

```

int main()
{
    cin>>n>>k;
    bstr=new short[n+1];
    x=new int[k];
    bstr[1]=0;
    if(k>1)backtrack(2);
    delete []x;
    delete []bstr;
    cout<<tot<<endl;
    return 0;
}

```

算法实现题 5-31 喷漆机器人问题

★问题描述:

F 大学开发出一种喷漆机器人 Rob, 能用指定颜色给一块矩形材料喷漆。Rob 每次拿起一种颜色的喷枪, 为指定颜色的小矩形区域喷漆。喷漆工艺要求, 一个小矩形区域只能在所有紧靠它上方的矩形区域都喷过漆后, 才能开始喷漆, 且小矩形区域开始喷漆后必须一次性喷完, 不能只喷一部分。为 Rob 编写一个自动喷漆程序, 使 Rob 拿起喷枪的次数最少。

★编程任务:

对于给定的矩形区域和指定的颜色, 计算 Rob 拿起喷枪的最少次数。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n , $1 \leq n \leq 16$, 表示小矩形的个数。大矩形坐标系如图 5-29 所示, 左上角点的坐标为 $(0, 0)$ 。颜色编号为正整数。接下来的 n 行, 每行用 5 个整数 $y1, x1, y2, x2, c$ 来表示一个矩形。 $(x1, y1)$ 和 $(x2, y2)$ 分别表示小矩形的左上角点坐标和右下角点坐标, c 表示小矩形的颜色。

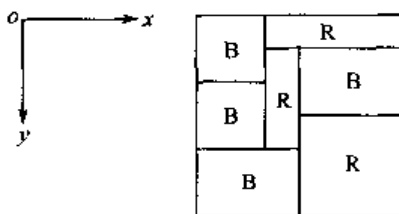


图 5-29 矩形喷漆材料

★结果输出:

将计算出的 Rob 拿起喷枪的最少次数输出到文件 output.txt。

输入文件示例

input.txt

7

0 0 2 2 1

0 2 1 6 2

2 0 4 2 1

1 2 4 3 2

1 3 3 6 1

4 0 6 3 1

3 3 6 6 2

输出文件示例

output.txt

3

分析与解答:

用一个位向量表示每个小矩形的喷漆情况, 对所有状态回溯搜索。

```
void backtrack(int r, int p)
{
    if(m[r][p] >= 0) return;
    for(int i=0; i<n; ++i)
        if(i!=r && (p&po2[i]) && g[i][r]) {
            m[r][p] = MAXx;
            return;
        }
    int np = p - po2[r];
    if(np == 0) m[r][p] = 1;
    else
        for(i=0; i<n; ++i)
            if(np&po2[i]) {
                backtrack(i, np);
                int v = m[i][np] + (color[r] == color[i] ? 0 : 1);
```

```

        if(m[r][p]<0 || m[r][p]>v)m[r][p]=v;
    }
}

```

comp 执行回溯法，并输出最优值。

```

void comp()
{
    memset(m, -1, sizeof(m));
    int opt=-1;
    for(int i=0;i<n;++i){
        backtrack(i, po2[n]-1);
        if(opt<0 || opt>m[i][po2[n]-1])
            opt=m[i][po2[n]-1];
    }
    cout<<opt<<endl;
}

```

init 进行初始化计算。

```

void init()
{
    int board[MAXx][MAXy];
    cin>>n;
    memset(board, -1, sizeof(board));
    for(int i=0;i<n;++i){
        int x1,y1,x2,y2;
        cin>>x1>>y1>>x2>>y2>>color[i];
        for(int j=x1;j<x2;++j)
            for(int k=y1;k<y2;++k)board[j][k]=i;
    }
    memset(g, false, sizeof(g));
    for(i=0;i<MAXx;++i)
        for(int j=0;j<MAXy;++j)
            if(board[i][j]>=0 && board[i+1][j]>=0 && board[i][j]!=board[i+1][j])
                g[board[i][j]][board[i+1][j]]=1;
}

```

实现算法的主函数如下。

```

int main()
{
    po2[0]=1;
    for(int i=1;i<MAXn;++i)po2[i]=po2[i-1]<<1;
}

```

```

    init();
    comp();
    return 0;
}

```

算法实现题 5-32 子集树问题 (习题 5-11)

★问题描述:

试设计一个用回溯法搜索子集空间树的函数。该函数的参数包括结点可行性判定函数和上界函数等必要的函数,并将此函数用于解装载问题。

装载问题描述如下:有一批共 n 个集装箱要装上艘载重量为 c 的轮船,其中集装箱 i 的重量为 w_i 。找出一种最优装载方案,将轮船尽可能装满,即在装载体积不受限制的情况下,将尽可能重的集装箱装上轮船。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 c 。 n 是集装箱数, c 是轮船的载重量。接下来的 1 行中有 n 个正整数,表示集装箱的重量。

★结果输出:

将计算出的最大装载重量输出到文件 output.txt。

输入文件示例

输出文件示例

input.txt

output.txt

5 10

10

7 2 6 5 4

分析与解答:

用主教材中搜索子集树的一般算法。

```

template<class T>
void Loading<T>::backtrack(int t)
{
    if(t>n)Output();
    else for(int i=0;i<=1;i++){
        x[t]=i;
        if(Constraint(t) && Bound(t)){
            Change(t);
            backtrack(t+1);
            Restore(t);
        }
    }
}

```

其中,结点可行性判定函数 Constraint 和上界函数 Bound 等必要的函数通过类 Loading 的私有函数传递。

```

template<class T>
void Loading<T>::Output()
{
    ii=n;bestw=cw;return;
}

template<class T>
bool Loading<T>::Constraint(int t)
{
    if(x[t]==0||x[t]==1&&cw+w[t]<=c)return true;
    else return false;
}

template<class T>
bool Loading<T>::Bound(int t)
{
    if(x[t]==1||x[t]==0&&cw+r-w[t]>bestw)return true;
    else return false;
}

template<class T>
void Loading<T>::Change(int t)
{
    if(x[t]==1)cw+=w[t];
    r-=w[t];
}

template<class T>
void Loading<T>::Restore(int t)
{
    if(x[t]==1)cw-=w[t];
    r+=w[t];
    if(ii==t){bestx[t]=x[t];ii--;}
}

```

算法实现题 5-33 0-1 背包问题 (习题 5-11)

★问题描述:

试设计一个用回溯法搜索子集空间树的函数。该函数的参数包括结点可行性判定函数和上界函数等必要的函数,并将此函数用于解 0-1 背包问题。

0-1 背包问题描述如下:给定 n 种物品和一个背包。物品 i 的重量是 w_i , 其价值为 v_i , 背包的容量为 C 。应如何选择装入背包的物品,使得装入背包中物品的总价值最大?

在选择装入背包的物品时,对每种物品 i 只有 2 种选择,即装入背包或不装入背包。不能将物品 i 装入背包多次,也不能只装入部分的物品 i 。

0-1 背包问题形式化描述如下:给定 $C>0, w_i>0, v_i>0, 1\leq i\leq n$, 要求 n 元 0-1 向量 $(x_1, x_2, \dots, x_n), x_i \in \{0, 1\}, 1\leq i\leq n$, 使得 $\sum_{i=1}^n w_i x_i \leq C$, 而且 $\sum_{i=1}^n v_i x_i$ 达到最大。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 c , n 是物品数, c 是背包的容量。接下来的 1 行中有 n 个正整数,表示物品的价值。第 3 行中有 n 个正整数,表示物品的重量。

★结果输出:

将计算出的装入背包物品的最大价值和最优装入方案输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5 10	15
6 3 5 4 6	1 1 0 0 1
2 2 6 5 4	

分析与解答:

用主教材中搜索子集树的一般算法。

```
template<class Typew, class Typep>
void Knap<Typew, Typep>::backtrack(int t)
{
    if(t>n)Output();
    else for(int i=0;i<=1;i++){
        x[t]=i;
        if(Constraint(t) && Bound(t)){
            Change(t);
            backtrack(t+1);
            Restore(t);
        }
    }
}
```

其中, 结点可行性判定函数 Constraint 和上界函数 Bound 等必要的函数通过类 Knap 的私有函数传递。

```
template<class Typew, class Typep>
void Knap<Typew, Typep>::Output()
{
    bestp=cp;
    for(int j=1;j<=n;j++)bestx[j]=x[j];
}

template<class Typew, class Typep>
```

```

bool Knap<Typew, Typew>::Constraint(int t)
{
    if(x[t]==0||x[t]==1&&cw+w[t]<=c)return true;
    else return false;
}

template<class Typew, class Typew>
bool Knap<Typew, Typew>::Bound(int t)
{
    if(x[t]==1||x[t]==0&&UpBound(t+1)>bestp)return true;
    else return false;
}

template<class Typew, class Typew>
void Knap<Typew, Typew>::Change(int t)
{
    if(x[t]==1){cw+=w[t];cp+=p[t];}
}

template<class Typew, class Typew>
void Knap<Typew, Typew>::Restore(int t)
{
    if(x[t]==1){cw-=w[t];cp-=p[t];}
}

```

算法实现题 5-34 排列树问题 (习题 5-12)

★问题描述:

试设计一个用回溯法搜索排列空间树的函数。该函数的参数包括结点可行性判定函数和上界函数等必要的函数,并将此函数用于解圆排列问题。

圆排列问题描述如下:给定 n 个大小不等的圆 c_1, c_2, \dots, c_n , 现要将这 n 个圆排进一个矩形框中,且要求各圆与矩形框的底边相切。圆排列问题要求从 n 个圆的所有排列中找出有最小长度的圆排列。

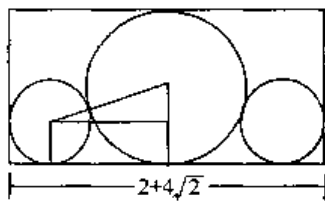


图 5-30 圆排列

例如,当 $n=3$, 且所给的 3 个圆的半径分别为 1, 1, 2 时,这 3 个圆的最小长度的圆排列如图 5-30 所示,其最小长度为 $2+4\sqrt{2}$ 。

★编程任务:

对于给定的 n 个圆,编程计算最小长度圆排列。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行是 1 个正整数 n , 表示有 n 个圆。第 2 行有 n 个正数, 分别表示 n 个圆的半径。

★结果输出:

程序运行结束时,将计算出的最小长度输出到文件 output.txt 中。文件的第 1 行是最小

长度, 保留 5 位小数。

输入文件示例

input.txt

3

1 1 2

输出文件示例

output.txt

7.65685

分析与解答:

用主教材中搜索排列树的一般算法。

```
void Circle::Backtrack(int t)
{
    if(t>n)Output();
    else for(int i=t;i<=n;i++){
        swap(x[t],x[i]);
        if(Constraint(t) && Bound(t)){
            Change(t);
            Backtrack(t+1);
            Restore(t);
        }
        swap(x[t],x[i]);
    }
}
```

其中, 结点可行性判定函数 Constraint 和上界函数 Bound 等必要的函数通过类 Circle 的私有函数传递。

```
void Circle::Output()
{
    float low=0,high=0;
    for(int i=1;i<=n;i++){
        if(xr[i]-x[i]<low) low=xr[i]-x[i];
        if(xr[i]+x[i]>high) high=xr[i]+x[i];
    }
    if(high-low<min) min=high-low;
}

bool Circle::Constraint(int t)
{
    return true;
}

bool Circle::Bound(int t)
{
    centerx=Center(t);
```

```

        if(centerx+x[t]+x[l]<min)return true;
        else return false;
    }

    void Circle::Change(int t)
    {
        xr[t]=centerx;
    }

    void Circle::Restore(int t)
    {}

```

Center 用于计算当前所选择圆的圆心横坐标。

```

float Circle::Center(int t)
{
    float temp=0;
    for (int j=1;j<t;j++) {
        float valuel=xr[j]+2.0*sqrt(x[t]*x[j]);
        if(valuel>temp) temp=valuel;
    }
    return temp;
}

```

算法实现题 5-35 一般解空间搜索问题（习题 5-13）

★问题描述：

试设计一个用回溯法搜索一般解空间的函数。该函数的参数包括：生成解空间中下一扩展结点的函数、结点可行性判定函数和上界函数等必要的函数，并将此函数用于解图的 m 着色问题。

图的 m 着色问题描述如下：给定无向连通图 G 和 m 种不同的颜色。用这些颜色为图 G 的各顶点着色，每个顶点着一种颜色。如果有一种着色法使 G 中每条边的 2 个顶点着不同颜色，则称这个图是 m 可着色的。图的 m 着色问题是对于给定图 G 和 m 种颜色，找出所有不同的着色法。

★编程任务：

对于给定的无向连通图 G 和 m 种不同的颜色，编程计算图的所有不同的着色法。

★数据输入：

由文件 input.txt 给出输入数据。第 1 行有 3 个正整数 n ， k 和 m ，表示给定的图 G 有 n 个顶点和 k 条边， m 种颜色。顶点编号为 $1, 2, \dots, n$ 。接下来的 k 行中，每行有 2 个正整数 u, v ，表示图 G 的一条边 (u, v) 。

★结果输出：

程序运行结束时，将计算出的不同的着色方案数输出到文件 output.txt。

输入文件示例

输出文件示例

input.txt

output.txt

5 8 4

48

1 2

1 3

1 4

2 3

2 4

2 5

3 4

4 5

分析与解答:

用主教材中搜索一般解空间的回溯法框架。

```
void Color::Backtrack(int t)
{
    if(t>n)Output();
    else for(int i=f(n,t);i<=g(n,t);i++){
        x[t]=h(i);
        Change(t);
        if(Constraint(t) && Bound(t))Backtrack(t+1);
        Restore(t);
    }
}
```

其中,生成解空间中下一扩展结点的函数、结点可行性判定函数 Constraint 和上界函数 Bound 等必要的函数通过类 Color 的私有函数传递。

```
void Color::Output()
{
    sum++;
}

bool Color::Constraint(int t)
{
    for (int j=1;j<=n;j++)
        if ((a[t][j]==1)&&(x[j]==x[t])) return false;
    return true;
}

bool Color::Bound(int t)
{
    return true;
}
```

```

    }

    void Color::Change(int t)
    {}

    void Color::Restore(int t)
    {
        x[t]=0;
    }

    int Color::f(int n, int t)
    {
        return 1;
    }

    int Color::g(int n, int t)
    {
        return m;
    }

    int Color::h(int i)
    {
        return i;
    }

```

算法实现题 5-36 最短加法链问题

★问题描述:

最优求幂问题: 给定一个正整数 n 和一个实数 x , 如何用最少的乘法次数计算出 x^n 。例如, 可以用 6 次乘法逐步计算 x^{23} 如下: $x, x^2, x^3, x^5, x^{10}, x^{20}, x^{23}$ 。可以证明, 计算 x^{23} 最少需要 6 次乘法。计算 x^{23} 的幂序列中各幂次 1, 2, 3, 5, 10, 20, 23 组成了一个关于整数 23 的加法链。在一般情况下, 计算 x^n 的幂序列中各幂次组成正整数 n 的一个加法链:

$$1 = a_0 < a_1 < a_2 < \cdots < a_r = n$$

$$a_i = a_j + a_k, k \leq j < i, i = 1, 2, \cdots, r$$

上述最优求幂问题相应于正整数 n 的最短加法链问题, 即求 n 的一个加法链使其长度 r 达到最小。正整数 n 的最短加法链长度记为 $l(n)$ 。

★编程任务:

对于给定的正整数 n , 编程计算相应于正整数 n 的最短加法链。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n 。

★结果输出:

程序运行结束时, 将计算出的最短加法链长度 $l(n)$ 和相应的最短加法链输出到文件

output.txt。

输入文件示例

输出文件示例

input.txt

output.txt

23

6

1 2 3 5 10 20 23

分析与解答：

(1) 标准回溯法

对最短加法链问题的状态空间树进行深度优先搜索的回溯法可描述如下。

```
void backtrack(int step)
{
    if(a[step]==n){
        if(step<best){
            best=step;
            for(int i=1;i<=best;i++)chain[i]=a[i];
        }
        return;
    }
    for(int i=step;i>=1;i--){
        if(2*a[i]>a[step])
            for(int j=i;j>=1;j--){
                int k=a[i]+a[j];
                a[step+1]=k;
                if(k>a[step] && k<=n) backtrack(step+1);
            }
    }
}
```

由于加法链问题的状态空间树的每一个第 k 层结点至少有 $k+1$ 个儿子结点，因此从根结点到第 k 层的任一结点的路径数至少是 $k!$ 。因此状态空间树以指数方式增长。用标准回溯法只能对较小的 n 构造出最短加法链。

(2) 迭代搜索法

用回溯法搜索加法链问题的状态空间树时，由于采用了深度优先的搜索方法，算法所搜索到的第一个加法链不一定是最短加法链。如果利用广度优先的方式搜索加法链问题的状态空间树，则算法找到的第一个加法链就是最短加法链，但这种方法的空间开销太大。逐步深化的迭代搜索算法既能保证算法找到的第一个加法链就是最短加法链，又不需要太大的空间开销。其基本思想是控制回溯法的搜索深度 d ，从 $d=1$ 开始搜索，每次搜索后使 d 增 1，加深搜索深度，直到找到一条加法链为止。

控制搜索深度的回溯法如下。

```
void backtrack(int step)
{
    if(!found)
```

```

    if(a[step]==n){
        best=step;
        for(int i=1;i<=best;i++)chain[i]=a[i];
        found=true;
        return;
    }
    else if(step<lb)
        for(int i=step;i>=1;i--){
            if(2*a[i]>a[step])
                for(int j=i;j>=1;j--){
                    int k=a[i]+a[j];
                    a[step+1]=k;
                    if(k>a[step] && k<=n) backtrack(step+1);
                }
        }
}

```

逐步深化的迭代搜索算法如下。

```

void iterativedeepening()
{
    best=n+1;
    found=false;
    lb=2;
    while(!found){
        a[1]=1;
        backtrack(1);
        lb++;
    }
}

```

(3) 算法优化

算法可进一步进行如下改进。

利用 $l(n)$ 的下界 $lb(n)$ 对迭代深度作精确估计。

采用剪枝函数对问题的状态空间树进行剪枝搜索，加速搜索进程。

用幂树构造 $l(n)$ 的精确上界 $ub(n)$ 。

当 $lb(n)=ub(n)$ 时，幂树给出的加法链已是最短加法链。

当 $lb(n)<ub(n)$ 时，用改进后的逐步深化迭代搜索算法，从深度 $d=lb(n)$ 开始搜索。

关于下界 $lb(n)$ 有：

剪枝 1：设在求正整数 n 的最短加法链的逐步深化迭代搜索算法中，当前搜索深度为 d ，则在状态空间树的第 i 层结点 a_i 处的一个剪枝条件是

$$\begin{cases} \log\left(\frac{n}{3a_i}\right) + i + 2 > d & 0 \leq i \leq d-2 \\ \log\left(\frac{n}{a_i}\right) + i > d & d-1 \leq i \leq d \end{cases}$$

剪枝 2: 设在求正整数 n 的最短加法链的逐步深化迭代搜索算法中, 当前搜索深度为 d 。且正整数 n 可表示为 $n=2^t(2k+1), k \geq 1$, 则在状态空间树的第 i 层结点 a_i 处的一个剪枝条件是

$$\begin{cases} \log\left(\frac{n}{3a_i}\right) + i + 2 > d & 0 \leq i \leq d-t-2 \\ \log\left(\frac{n}{a_i}\right) + i > d & d-t-1 \leq i \leq d \end{cases}$$

与加法链问题密切相关的幂树给出了 $l(n)$ 的精确上界, 如图 5-31 所示。

假设已定义了幂树 T 的第 k 层结点, 则 T 的第 $k+1$ 层结点可定义如下。依从左到右顺序取第 k 层结点 a_k , 定义其按从左到右顺序排列的儿子结点为 $a_k + a_j, 0 \leq j \leq k$ 。其中, a_0, a_1, \dots, a_k 是从 T 的根到结点 a_k 的路径。且 $a_k + a_j$ 在 T 中未出现过。

含正整数 n 的部分幂树 T 容易在线性时间内构造如下。

find 递归构造幂树。

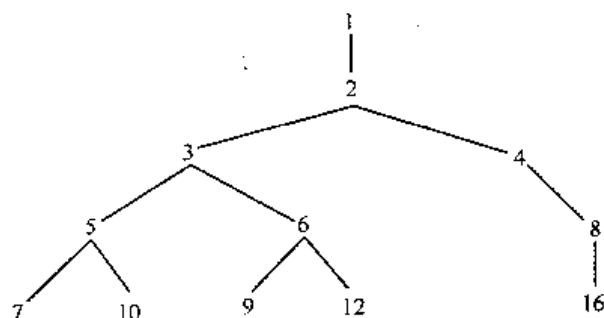


图 5-31 幂树

```

void find(int step)
{
    int i, k;
    if (!found)
        if (a[step] == n) {
            best = step;
            for (i = 1; i <= best; i++) chain[i] = a[i];
            found = true;
            return;
        }
        else if (step <= ub)
            for (i = 1; i <= step; i++) {
                k = a[step] + a[i];
                if (k <= n) {
                    a[step+1] = k;
                    if (parent[k] == 0) parent[k] = a[step];
                    if (parent[k] == a[step]) find(step+1);
                }
            }
}
  
```

powertree 以逐步深化的迭代搜索方式构造幂树。

```

int powertree(int n)
{
  
```

```

        found=false;
        ub=1;
        for(int i=1;i<=MAXN;i++) parent[i]=0;
        while (!found){
            a[1]=1;
            find(1);
            ub++;
        }
        return best;
    }
}

```

改进后的逐步深化迭代搜索算法描述如下。

```

void search()
{
    lb=lowerb(n);
    ub=powertree(n);
    t=gett(n);
    if(lb<ub){
        found=false;
        while(!found){
            cout<<"lb="<<lb<<endl;
            a[1]=1;
            backtrack(1);
            lb++;
            if(lb==ub) found=true;
        }
    }
}

```

其中, lowerb 和 gett 实现如下。

```

int lowerb(int m)
{
    int i=0, j=1;
    while(m>1){i++;if(odd(m)) j++;m=m>>1;}
    i+=log2(j)+1;
    return i;
}

int log2(int m)
{
    int i=0, j=1;
    while(m>1){i++;if(odd(m)) j++;m=m>>1;}
}

```

```

        if(j>1) i++;
        return i;
    }

    int gett(int num)
    {
        int i=0;
        while(!odd(num)) {num=num>>1;i++;}
        return i-1;
    }

```

算法的主体是 backtrack。

```

void backtrack(int step)
{
    if(!found)
        if(a[step]==n) {
            best=step;
            for(int i=1;i<=best;i++)chain[i]=a[i];
            found=true;
            return;
        }
    else if(step<lb)
        for(int i=step;i>=1;i--)
            if(2*a[i]>a[step])
                for(int j=i;j>=1;j--){
                    int k=a[i]+a[j];
                    a[step+1]=k;
                    if(k>a[step] && k<=n)
                        if(!pruned(step+1)) backtrack(step+1);
                }
}

```

pruned 实现剪枝。

```

bool pruned(int step)
{
    if (step<lb-t-1) return (h(3*a[step])+step+2>lb);
    else return (h(a[step])+step>lb);
}

int h(int num)
{
    int i=0;

```

```

while(num<n){num=num<<1;i++;}
return i;
}

```

实现算法的主函数如下。

```

int main()
{
    cin>>n;
    if(n<=MAXN){search();output();}
    else cout<<"Done!"<<endl;
    return 0;
}

```

算法实现题 5-37 n^2-1 谜问题

★问题描述：

重排九宫是一个古老的单人智力游戏。据说重排九宫起源于我国古时由三国演义故事“关羽义释曹操”而设计的智力玩具“华容道”，后来流传到欧洲，将人物变成数字。原始的重排九宫问题是这样的：将数字 1~8 按照任意次序排在 3×3 的方格阵列中，留下一个空格。与空格相邻的数字，允许从上、下、左、右方向移动到空格中。游戏的最终目标是通过合法移动，将数字 1~8 按行排好序。在一般情况下， n^2-1 谜问题是将数字 1~ n^2-1 按照任意次序排在 $n \times n$ 的方格阵列中，留下一个空格。允许与空格相邻的数字从上、下、左、右 4 个方向移动到空格中。游戏的最终目标是通过合法移动，将初始状态变换到目标状态。 n^2-1 谜问题的目标状态是将数字 1~ n^2-1 按从小到大的次序排列，最后一个位置为空格。

1	2	3
4		6
7	5	8

图 5-32 重排九宫

★编程任务：

对于给定的 $n \times n$ 方格阵列中数字 1~ n^2-1 初始排列，编程计算将初始排列通过合法移动变换为目标状态的最少移动次数。

★数据输入：

由文件 input.txt 给出输入数据。文件的第 1 行有 1 个正整数 n 。以下的 n 行是 $n \times n$ 方格阵列的中数字 1~ n^2-1 的初始排列，每行有 n 个数字表示该行方格中的数字，0 表示空格。

★结果输出：

将计算出的最少移动次数和相应的移动序列输出到文件 output.txt。第 1 行是最少移动次数。从第 2 行开始，依次输出移动序列。

输入文件示例

```

input.txt
3
1 2 3
4 0 6
7 5 8

```

输出文件示例

```

output.txt
2
5 8

```

分析与解答:

逐步深化的搜索算法描述如下。

```
#define row(x) (x/rowsz)
#define col(x) (x%rowsz)
int rowsz, boardsz, moves, *start, *board;
int pos[100];

bool solve(int l, int t, int p)
{
    register int d, q, del;
    pos[l]=p, q=pos[l-1];
    if(t==0) {out(l, q); return true;}
    if(t>0)
        for(d=0; d<4; d++) {
            del=trymove(p, q, d);
            if(del && solve(l+1, t-del, p)) return true;
            if(del)p=restore(p, d);
        }
    return false;
}

void idastar()
{
    int p=initstate();
    if(moves==0) {out(0, 0); return;}
    while(!solve(1, moves, p)) moves+=0x101;
}
```

其中, trymove 按照可移动方向移动。

```
int trymove(int &p, int q, int d)
{
    int del=0;
    switch(d) {
        case 0: // right
            if(col(p)<=rowsz-2&&q!=p+1) {
                q=p+1;
                del=(col(board[q])<col(q)? 0x1:0x100);
            }
            break;
        case 1: // up
            if(row(p)>=1&&q!=p-rowsz) {
                q=p-rowsz;
```

```

        del=(row(board[q])>row(q)? 0x1:0x100);
    }
    break;
case 2: // left
    if(col(p)>=1&&q!=p-1){
        q=p-1;
        del=(col(board[q])>col(q)? 0x1:0x100);
    }
    break;
case 3: // down
    if(row(p)<=rowsz-2&&q!=p+rowsz){
        q=p+rowsz;
        del=(row(board[q])<row(q)? 0x1:0x100);
    }
}
if(del){board[p]=board[q];p=q;}
return del;
}

```

restore 恢复移动前的状态。

```

int restore(int p, int d)
{
    int q=p-1;           // right
    if(d==1) q=p+rowsz;  // up
    if(d==2) q=p+1;      // left
    if(d==3) q=p-rowsz;  // down
    board[p]=board[q];
    return q;
}

```

initstate 给出初始状态。

```

int initstate()
{
    register int j, del, p;
    for(j=moves=0; j<boardsz; j++) if(start[j]>=0){
        del=row(start[j])-row(j);
        moves+=(del<0? -del:del);
        del=col(start[j])-col(j);
        moves+=(del<0? -del:del);
    }
    pos[0]=boardsz;
    for(j=0; j<boardsz; j++){

```

```

        board[j]=start[j];
        if(board[j]<0) p=j;
    }
    return p;
}

```

实现算法的主函数如下。

```

int main()
{
    if(init())idastar();
    else cout<<"No Solution!"<<endl;
    return 0;
}

```

init 读入初始数据并判定可解性。

```

bool init()
{
    cin>>rowsz;boardsz=rowsz*rowsz;
    start=new int[boardsz];
    board=new int[boardsz];
    for(int i=0,del=0,t=0;i<boardsz;i++){
        cin>>start[i],start[i]--;
        for(int s=0;s<i;s++)if(start[s]>start[i])del++;
        if(start[i]<0)t=i;
    }
    if(((row(t)+col(t)+del+rowsz)&0x1)==0) return false;
    return true;
}

```

out 输出最优解。

```

void out(int l, int q)
{
    pos[l+1]=q;
    cout<<(moves&0xff)+(moves>>8)<<endl;
    if(moves>0){
        for(int j=0;j<boardsz;j++)board[j]=start[j];
        for(int k=1;k<l;k++){
            cout<<(board[pos[k+1]]+1)<<" ";
            board[pos[k]]=board[pos[k+1]];
        }
    }
}

```

```
}
```

上述算法可非递归化如下，效率明显提高。

```
#define row(x) (x/rowsz)
#define col(x) (x%rowsz)
int rowsz,boardsz,moves,*start,*board;
int pos[100],tim[100],dir[100];

void idastar()
{
    register int j,t,l,d,p,q,del,piece,moves;
    moves=dist0();
    if(moves==0) {out(0,0,moves);return;}
    while(1){
        t=moves;
        for(j=0;j<boardsz;j++){
            board[j]=start[j];
            if(board[j]<0)p=j;
        }
        pos[0]=boardsz,l=1;

newlevel:
        d=0,tim[1]=t,pos[1]=p,q=pos[l-1];

trymove:
        switch(d){
            case 0: // right
                if(col(p)<=rowsz-2&&q!=p+1){
                    q=p+1,piece=board[q];
                    del=(col(piece)<col(q)?0x1:0x100);
                    break;
                }
                d++;
            case 1: // up
                if(row(p)>=1&&q!=p-rowsz){
                    q=p-rowsz,piece=board[q];
                    del=(row(piece)>row(q)?0x1:0x100);
                    break;
                }
                d++;
            case 2: // left
                if(col(p)>=1&&q!=p-1){
                    q=p-1,piece=board[q];
```



```

        del=(col(piece)> col(q)?0x1:0x100);
        break;
    }
    d++;
case 3: // down
    if(row(p)<=rowsz-2&&q!=p+rowsz){
        q=p+rowsz,piece=board[q];
        del=(row(piece)<row(q)? 0x1:0x100);
        break;
    }
    d++;
case 4: goto backtrack;
}
if(t<=del){
    if(t==del){out(l,q,moves);return;}
    d++;goto trymove;
}
dir[l]=d,board[p]=board[q],t-=del,p=q,l++;
goto newlevel;

backtrack:
    if(l>1){
        l--;
        q=pos[l],board[p]=board[q],p=q,q=pos[l-1],t=tim[l],d=dir[l]+1;
        goto trymove;
    }
    moves+=0x101;
}
}

```

其中，dist0 计算初始 Manhattan 距离。

```

int dist0()
{
    register int j,del,moves;
    for(j=moves=0;j<boardsz;j++)if(start[j]>=0){
        del=row(start[j])-row(j);
        moves+=(del<0?-del:del);
        del=col(start[j])-col(j);
        moves+=(del<0?-del:del);
    }
    return moves;
}

```

第 6 章 分支限界法

习题 6-1 0-1 背包问题的栈式分支限界法

栈式分支限界法将活结点表以后进先出(LIFO)的方式存储于一个栈中。试设计一个解 0-1 背包问题的栈式分支限界法,并说明栈式分支限界法与回溯法的区别。

分析与解答:

函数 StackKnapsack 实施对子集树的栈式分支限界法。其中假定各物品依其单位重量价值从大到小排好序。相应的排序过程可在算法的预处理部分完成。

算法中 enode 是当前扩展结点; cw 是该结点所相应的重量; cp 是相应的价值; up 是价值上界。算法的 while 循环不断扩展结点,首先检查当前扩展结点的左儿子结点的可行性。如果该左儿子结点是可行结点,则将它加入到子集树和活结点栈中。当前扩展结点的右儿子结点一定是可行结点,仅当右儿子结点满足上界约束时才将它加入子集树和活结点栈。具体算法描述如下。

```
template<class Typew, class Typep>
Typep Knap<Typew, Typep>::StackKnapsack()
{ // 栈式分支限界法, 返回最大价值
  // 定义栈的容量为 1000
  H=new STACK< HeapNode<Typep, Typew> >(1000);
  // 初始化
  int i=1;
  E=0; cw=cp=0;
  Typep bestp=0;           // 当前最优值
  Typep up=Bound(1);       // 价值上界
  // 搜索子集空间树
  while(true) {
    // 检查当前扩展结点的左儿子结点
    Typew wt=cw+w[i];
    if(wt<=c) { // 左儿子结点为可行结点
      if(cp+p[i] > bestp) bestp=cp+p[i];
      AddLiveNode(up, cp+p[i], cw+w[i], true, i+1);
    }
    up=Bound(i+1);
    // 检查当前扩展结点的右儿子结点
    if(up >= bestp) // 右子树可能含最优解
      AddLiveNode(up, cp, cw, false, i+1);
    // 取下一扩展结点
    if(H->EMPTY()) return bestp;
    HeapNode<Typep, Typew> N;
```

```

        H->POP(N);
        E=N.ptr;
        cw=N.weight;
        cp=N.profit;
        up=N.uprofit;
        i=N.level;
    }
}

```

其中, AddLiveNode 将一个新的活结点插入到子集树和栈中。

```

template<class Typep, class Typew>
void Knap<Typep, Typew>::AddLiveNode(Typep up, Typep cp, Typew cw, bool ch, int lev)
{
    bbnode *b=new bbnode;
    b->parent=E;
    b->LChild=ch;
    HeapNode<Typep, Typew> N;
    N.uprofit=up;
    N.profit=cp;
    N.weight=cw;
    N.level=lev;
    N.ptr=b;
    if(lev<=n)H->PUSH(N);
}

```

下面的算法 Knapsack 完成对输入数据的预处理。主要任务是将各物品依其单位重量价值从大到小排好序。然后调用 StackKnapsack 完成对子集树的栈式分支限界搜索。

```

template<class Typew, class Typep>
Typep Knapsack(Typep *p, Typew *w, Typew c, int n)
{
    // 初始化
    Typew W=0;           // 装包物品重量
    Typep P=0;           // 装包物品价值
    // 定义依单位重量价值排序的物品数组
    Object *Q=new Object [n];
    for(int i=1;i<=n;i++){
        // 单位重量价值数组
        Q[i-1].ID=i;
        Q[i-1].d=1.0*p[i]/w[i];
        P+=p[i];
        W+=w[i];
    }
}

```

```

    if(W<=c) return P; // 所有物品装包
    // 依单位重量价值排序
    MergeSort(Q, n);
    // 创建类 Knap 的数据成员
    Knap<Typew, Typep> K;
    K.p=new Typep [n+1];
    K.w=new Typew [n+1];
    for(i=1;i<=n;i++){
        K.p[i]=p[Q[i-1].ID];
        K.w[i]=w[Q[i-1].ID];
    }
    K.cp=0;K.cw=0;K.c=c;K.n=n;
    // 调用 StackKnapsack 求问题的最优解
    Typep bestp=K.StackKnapsack();
    delete [] Q;
    delete [] K.w;
    delete [] K.p;
    return bestp;
}

```

栈式分支限界法与回溯法的主要区别在于,对当前扩展结点所采用的扩展方式不同。在栈式分支限界法中,每一个活结点只有一次机会成为扩展结点。活结点一旦成为扩展结点,就一次性产生其所有儿子结点。在这些儿子结点中,导致不可行解或导致非最优解的儿子结点被舍弃,其余儿子结点被加入活结点表中。此后,从活结点表中取下一结点成为当前扩展结点,并重复上述结点扩展过程。这个过程一直持续到找到所需的解或活结点表为空时为止。而回溯法中的结点有可能多次成为活结点。

习题 6-2 释放结点空间的队列式分支限界法

修改解装载问题的分支限界算法 MaxLoading,使得算法在结束前释放所有已由 EnQueue 产生的结点。

分析与解答:

用一个队列 Que 收集算法删除的扩展结点。算法结束后释放所有已由 EnQueue 产生的结点。修改后的算法如下。

```

template<class Type>
Type MaxLoading(Type w[], Type c, int n, int bestx[])
{ // 队列式分支限界法,返回最优载重量, bestx 返回最优解
    // 初始化
    Queue<QNode<Type>*> Q, Que; // 活结点队列
    Q.Add(0); // 同层结点尾部标志
    int i=1; // 当前扩展结点所处的层
    Type Ew=0, bestw=0, r=0;
    for(int j=2;j<=n;j++) r+=w[i];
}

```

```

QNode<Type>*E=0,*bestE;
// 搜索子集空间树
while(true){
    // 检查左儿子结点
    Type wt=Ew+w[i];
    if(wt<=c){ // 可行结点
        if(wt>bestw) bestw=wt;
        EnQueue(Q,wt,i,n,bestw,E,bestE,bestx,true);
    }
    // 检查右儿子结点
    if(Ew+r>bestw) EnQueue(Q,Ew,i,n,bestw,E,bestE,bestx,false);
    Q.Delete(E); // 取下一扩展结点
    if(E)Que.Add(E);
    if(!E){ // 同层结点尾部
        if(Q.IsEmpty()) break;
        Q.Add(0); // 同层结点尾部标志
        Q.Delete(E); // 取下一扩展结点
        if(E)Que.Add(E);
        i++; // 进入下一层
        r-=w[i];
    }
    Ew=E->weight; // 新扩展结点所相应的载重量
}
// 构造当前最优解
for(j=n-1;j>0;j--){
    bestx[j]=bestE->LChild;
    bestE=bestE->parent;
}
while(!Que.IsEmpty()){
    QNode<Type>*b;
    Que.Delete(b);
    delete b;
}
return bestw;
}

```

习题 6-3 及时删除不用的结点

解装载问题的分支限界算法中,由 EnQueue 产生的结点可以在算法结束前一次性删除。然而那些没有活儿子结点或没有叶结点的扩展结点可以立即被删除。试设计一个在算法中及时删除不用结点的方案,并讨论其时间与空间之间的折中。

分析与解答:

修改后及时删除不用结点的算法如下。

```

template<class Type>
Type MaxLoading(Type w[], Type c, int n, int bestx[])
{ // 队列式分支限界法, 返回最优载重量, bestx 返回最优解
  // 初始化
  Queue<QNode<Type>*> Q, Que; // 活结点队列
  Q.Add(0); // 同层结点尾部标志
  int i=1; // 当前扩展结点所处的层
  Type Ew=0, bestw=0, r=0;
  for(int j=2; j<=n; j++) r+=w[j];
  QNode<Type> *E=0, *bestE;
  // 搜索子集空间树
  while(true) {
    // 检查左儿子结点
    Type wt=Ew+w[i];
    bool del=true;
    if(wt<=c) { // 可行结点
      if(wt>bestw) bestw=wt;
      EnQueue(Q, wt, i, n, bestw, E, bestE, bestx, true);
      del=false;
    }
    // 检查右儿子结点
    if(Ew+r>bestw) {
      EnQueue(Q, Ew, i, n, bestw, E, bestE, bestx, false);
      del=false;
    }
    if(E) { if(del) delete E; else Que.Add(E); }
    Q.Delete(E); // 取下一扩展结点
    if(!E) { // 同层结点尾部
      if(Q.IsEmpty()) break;
      Q.Add(0); // 同层结点尾部标志
      Q.Delete(E); // 取下一扩展结点
      i++; // 进入下一层
      r-=w[i];
    }
    Ew=E->weight; // 新扩展结点所相应的载重量
  }
  // 构造当前最优解
  for(j=n-1; j>0; j--) {
    bestx[j]=bestE->LChild;
    bestE=bestE->parent;
  }
  while(!Que.IsEmpty()) {
    QNode<Type> *b;

```

```

        Que.Delete(b);
        delete b;
    }
    return bestw;
}

```

习题 6-4 用最大堆存储活结点的优先队列式分支限界法

试修改解装载问题和解 0-1 背包问题的优先队列式分支限界法, 使其仅使用一个最大堆来存储活结点, 而不必存储所产生的解空间树。

分析与解答:

对于解 0-1 背包问题的优先队列式分支限界法, 在堆结点中增加记录当前解的数组 x 。

```

template<class Tw, class Tp>
class HeapNode {
    friend Knap<Tw, Tp>;
public:
    operator Tp () const {return uprofit;}
private:
    Tp uprofit, profit;
    Tw weight;
    int level, *x;
};

```

修改后的算法如下。

```

template<class Typew, class Typep>
class Knap {
    friend Typep Knapsack(Typep *, Typew *, Typew, int, int *);
public:
    Typep MaxKnapsack();
private:
    MaxHeap<HeapNode<Typep, Typew>>*H;
    Typep Bound(int i);
    void AddLiveNode(Typep up, Typep cp, Typew cw, bool ch, int level, int *x);
    Typew c;          // 背包容量
    int n;             // 物品总数
    Typew *w;          // 物品重量数组
    Typep *p;          // 物品价值数组
    Typew cw;          // 当前装包重量
    Typep cp;          // 当前装包价值
    int *bestx;        // 最优解
};

```

上界函数 Bound 计算结点所相应价值的上界。

```
template<class Typew, class Typep>
Typep Knap<Typew, Typep>::Bound(int i)
{
    // 计算结点所相应价值的上界
    Typew cleft=c-cw;    // 剩余容量
    Typep b=cp;          // 价值上界
    // 以物品单位重量价值递减序装填剩余容量
    while(i<=n && w[i]<=cleft){
        cleft-=w[i];
        b+=p[i];
        i++;
    }
    // 装填剩余容量装满背包
    if(i<=n) b+=p[i]/w[i]*cleft;
    return b;
}
```

修改后的函数 AddLiveNode 将一个新结点插入到优先队列中。

```
template<class Typep, class Typew>
void Knap<Typep, Typew>::AddLiveNode(Typep up,
                                     Typew cp, Typew cw, bool ch, int lev, int *x)
{
    // 将一个新结点插入到最大堆 H 中
    HeapNode<Typep, Typew> N;
    N.x=new int[n+1];
    for(int j=0; j<=n; j++) N.x[j]=x[j];
    N.uprofit=up;
    N.profit=cp;
    N.weight=cw;
    N.level=lev;
    N.x[lev-1]=ch;
    H->Insert(N);
}
```

函数 MaxKnapsack 实施对子集空间树的优先队列式分支限界搜索。其中假定各物品依其单位重量价值从大到小排好序。相应的排序过程可在算法的预处理部分完成。

```
template<class Typew, class Typep>
Typep Knap<Typew, Typep>::MaxKnapsack()
{
    // 优先队列式分支限界法, 返回最大价值, bestx 返回最优解
    // 定义最大堆的容量为 10000
    H=new MaxHeap<HeapNode<Typep, Typew>>(10000);
    HeapNode<Typep, Typew> N;
```



```

N.x=new int[n+1];
// 为 bestx 分配存储空间
bestx=new int [n+1];
for(int k=0;k<=n;k++)bestx[k]=0;
// 初始化
int i=1;
cw=cp=0;
Typep bestp=0;    // 当前最优值
Typep up=Bound(1); // 价值上界
// 搜索子集空间树
while(i!=n+1){ // 非叶结点
    // 检查当前扩展结点的左儿子结点
    Typep wt=cw+w[i];
    if(wt<=c){ // 左儿子结点为可行结点
        if(cp+p[i]>bestp) bestp=cp+p[i];
        AddLiveNode(up, cp+p[i], cw+w[i], true, i+1, bestx);
        up=Bound(i+1);
    }
    // 检查当前扩展结点的右儿子结点
    if(up>=bestp) // 右子树可能含最优解
        AddLiveNode(up, cp, cw, false, i+1, bestx);
    // 取下一扩展结点
    H->DeleteMax(N);
    cw=N.weight;
    cp=N.profit;
    up=N.uprofit;
    i=N.level;
    for(int j=0;j<=n;j++)bestx[j]=N.x[j];
}
// 构造当前最优解
for(int j=0;j<=n;j++) bestx[j]=N.x[j];
while(true){ // 释放最小堆中所有结点
    try{H->DeleteMax(N);}
    catch(OutOfBounds){break;}
}
return cp;
}

```

下面的算法 Knapsack 完成对输入数据的预处理。主要任务是将各物品依其单位重量价值从大到小排好序。然后调用 MaxKnapsack 完成对子集空间树的优先队列式分支限界搜索。

```

template<class Typep, class Typew>
Typep Knapsack(Typep *p, Typew *w, Typep c, int n, int *bestx)
{ // 返回最大价值, bestx 返回最优解
    // 初始化

```

```

    Typew W=0;    // 装包物品重量
    Typep P=0;    // 装包物品价值
    // 定义依单位重量价值排序的物品数组
    Object *Q=new Object [n];
    for(int i=1;i<=n;i++){
        // 单位重量价值数组
        Q[i-1].ID=i;
        Q[i-1].d=1.0*p[i]/w[i];
        P+=p[i];
        W+=w[i];
    }
    if(W<=c) return P; // 所有物品装包
    // 依单位重量价值排序
    MergeSort(Q,n);
    // 创建类 Knap 的数据成员
    Knap<Typew, Typep> K;
    K.p=new Typep [n+1];
    K.w=new Typew [n+1];
    for(i=1;i<=n;i++){
        K.p[i]=p[Q[i-1].ID];
        K.w[i]=w[Q[i-1].ID];
    }
    K.cp=0;K.cw=0;K.c=c;K.n=n;
    // 调用 MaxKnapsack 求问题的最优解
    Typep bestp=K.MaxKnapsack();
    for(int j=1;j<=n;j++)bestx[Q[j-1].ID]=K.bestx[j];
    delete [] Q;
    delete [] K.w;
    delete [] K.p;
    delete [] K.bestx;
    return bestp;
}

```

习题 6-5 释放结点空间的优先队列式分支限界法

试修改解装载问题和解 0-1 背包问题的优先队列式分支限界法,使得算法在运行结束时释放所有类型为 bbnode 和 HeapNode 的结点所占用的空间。

分析与解答:

用一个队列 Que 收集算法删除的扩展结点。算法结束后释放所有类型为 bbnode 和 HeapNode 结点所占用的空间。修改后的解 0-1 背包问题的优先队列式分支限界法如下。

```

template<class Typew, class Typep>
Typep Knap<Typew, Typep>::MaxKnapsack()

```

```

{ // 优先队列式分支限界法, 返回最大价值, bestx 返回最优解
    // 定义最大堆的容量为 10000
    H=new MaxHeap<HeapNode<Typep, Typew>> (10000);
    Queue<bnode*> Que;
    // 为 bestx 分配存储空间
    bestx=new int [n+1];
    // 初始化
    int i=1;
    E=0; cw=cp=0;
    Typep bestp=0; // 当前最优值
    Typep up=Bound(1); // 价值上界
    // 搜索子集空间树
    while(i!=n+1) { // 非叶结点
        // 检查当前扩展结点的左儿子结点
        Typew wt=cw+w[i];
        if(wt<=c) { // 左儿子结点为可行结点
            if(cp+p[i]>bestp) bestp=cp+p[i];
            AddLiveNode(up, cp+p[i], cw+w[i], true, i+1);
            up=Bound(i+1);
        }
        // 检查当前扩展结点的右儿子结点
        if(up>=bestp) // 右子树可能含最优解
            AddLiveNode(up, cp, cw, false, i+1);
        // 取下一扩展结点
        HeapNode<Typep, Typew> N;
        H->DeleteMax(N);
        E=N.ptr; Que.Add(E);
        cw=N.weight;
        cp=N.profit;
        up=N.uprofit;
        i=N.level;
    }
    // 构造当前最优解
    for(int j=n; j>0; j--) {
        bestx[j]=E->LChild;
        E=E->parent;
    }
    HeapNode<Typep, Typew> N;
    while(true) { // 释放堆中所有结点
        try {H->DeleteMax(N);}
        catch(OutOfBounds) {break;}
        Que.Add(N.ptr);
    }
    while(!Que.IsEmpty()) {
        bnode*b;

```

```

        Que.Delete(b);
        delete b;
    }
    return cp;
}

```

修改后的解装载问题的优先队列式分支限界法如下。

```

template<class T>
T MaxLoading(T *w, T c, int n, int *bestx)
{
    MaxHeap<HeapNode<T> > H(1000);
    Queue<bbnode *> Que;
    T *r=new T[n+1];
    r[n]=0;
    for(int j=n-1;j>0;j--) r[j]=r[j+1]+w[j+1];
    int i=1;
    bbnode *E=0;
    int Ew=0;
    while(i!=n+1){
        if(Ew+w[i]<=c){
            AddLiveNode(H, E, Ew+w[i]+r[i], true, i+1);
            AddLiveNode(H, E, Ew+r[i], false, i+1);
            HeapNode<T> N;
            H.DeleteMax(N);
            i=N.level;
            E=N.ptr;Que.Add(E);
            Ew=N.uweight-r[i-1];
        }
        for(j=n;j>0;j--){
            bestx[j]=E->LChild;
            E=E->parent;
        }
        HeapNode<T> N;
        while(true){
            try{H.DeleteMax(N);}
            catch(OutOfBounds){break;}
            Que.Add(N.ptr);
        }
        while(!Que.IsEmpty()){
            bbnode *b;
            Que.Delete(b);
            delete b;
        }
    }
}

```

```

return Ew;
}

```

习题 6-6 团顶点数的上界

解最大团问题的优先队列式分支限界法中, 当前扩展结点满足 $cn+n-i \geq \text{bestn}$ 的右儿子结点被插入到优先队列中。如果将这个条件修改为满足 $cn+n-i > \text{bestn}$ 右儿子结点插入优先队列, 仍能保证算法的正确性吗? 为什么?

分析与解答:

如果将条件 $cn+n-i \geq \text{bestn}$ 修改为满足 $cn+n-i > \text{bestn}$ 右儿子结点插入优先队列, 不能保证算法的正确性。因为在当前扩展结点处, 团顶点数的上界为 $cn+n-i+1$ 。

习题 6-7 团顶点数改进的上界

考虑最大团问题的子集空间树中第 i 层的一个结点 x , 设 $\text{MinDegree}(x)$ 是以结点 x 为根的子树中所有结点度数的最小值。

(1) 设 $x.u = \min \{x.cn+n-i+1, \text{MinDegree}(x)+1\}$, 证明以结点 x 为根的子树中任一叶结点所相应的团的大小不超过 $x.u$ 。

(2) 依此 $x.u$ 的定义重写算法 BBMaxClique。

(3) 比较新旧算法所需的计算时间和产生的排列树结点数。

分析与解答:

(1) 在当前扩展结点 x 处, $\text{MinDegree}(x)+1$ 显然是团顶点数的一个上界。主教材中在当前扩展结点 x 处团顶点数的上界为 $x.cn+n-i+1$ 。

由此可见, $\min\{x.cn+n-i+1, \text{MinDegree}(x)+1\}$ 是当前扩展结点 x 处团顶点数的上界。

(2) 在算法预处理时, 先计算出每个顶点的 $\text{MinDegree}(x)$, 然后在算法中修正团顶点数的上界。

(3) 新算法所产生的排列树结点数较少。

习题 6-8 修改解旅行售货员问题的分支限界法

试修改解旅行售货员问题的分支限界法, 使得 $s=n-2$ 的结点不插入优先队列, 而是将当前最优排列存储于 bestp 中。经这样修改后, 算法在下一个扩展结点满足条件 $\text{Lcost} \geq \text{bestc}$ 时结束。

分析与解答:

在类 Traveling 中增加保存当前最优排列的数组 bestp 。

```

template<class Type>
class Traveling{
    friend int main();
    public:
        Type BBTSP(int v[]);
    private:
        int n,*bestp;

```

```

        Type **a, NoEdge, c, bestc;
};

```

最小堆结点类型不变。

```

template<class Type>
class MinHeapNode{
    friend Traveling<Type>;
public:
    operator Type () const{return lcost;}
private:
    Type lcost, cc, rcost;
    int s, *x;
};

```

修改后的算法描述如下。

```

template<class Type>
Type Traveling<Type>::BBTSP(int v[])
{// 解旅行售货员问题的优先队列式分支限界法
    // 定义最小堆的容量为 1000000
    MinHeap<MinHeapNode<Type>> H(1000000);
    Type *MinOut=new Type[n+1];
    // 计算 MinOut[i]=顶点 i 的最小出边费用
    Type MinSum=0; // 最小出边费用和
    for(int i=1;i<=n;i++){
        Type Min=NoEdge;
        for(int j=1;j<=n;j++){
            if(a[i][j]!=NoEdge && (a[i][j]<Min || Min==NoEdge))
                Min=a[i][j];
            if(Min==NoEdge) return NoEdge; // 无回路
            MinOut[i]=Min; MinSum+=Min;
        }
    }
    // 初始化
    MinHeapNode<Type> E;
    E.x=new int [n];
    for(i=0;i<n;i++) E.x[i]=i+1;
    E.s=0; E.cc=0; E.rcost=MinSum;
    Type bestc=NoEdge;
    // 搜索排列空间树
    while (E.s<n-1 && E.lcost<bestc){// 非叶结点
        if(E.s==n-2){// 当前扩展结点是叶结点的父结点
            // 再加 2 条边构成回路
            // 所构成回路是否优于当前最优解

```

```

        if(a[E.x[n-2]][E.x[n-1]]!=NoEdge &&
            a[E.x[n-1]][1]!=NoEdge && (E.cc+
            a[E.x[n-2]][E.x[n-1]]+a[E.x[n-1]][1]
            <bestc || bestc==NoEdge)){
            // 费用更小的回路
            bestc=E.cc+a[E.x[n-2]][E.x[n-1]]+a[E.x[n-1]][1];
            for(int j=0;j<n;j++)bestp[j]=E.x[j];
        }
        else delete [] E.x;          // 舍弃扩展结点
    }
    else{// 产生当前扩展结点的儿子结点
        for(int i=E.s+1;i<n;i++)
            if(a[E.x[E.s]][E.x[i]]!=NoEdge){
                // 可行儿子结点
                Type cc=E.cc+a[E.x[E.s]][E.x[i]];
                Type rcost=E.rcost-MinOut[E.x[E.s]];
                Type b=cc+rcost;      // 下界
                if(b<bestc || bestc==NoEdge){
                    // 子树可能含最优解
                    // 结点插入最小堆
                    MinHeapNode<Type> N;
                    N.x=new int [n];
                    for(int j=0;j<n;j++) N.x[j]=E.x[j];
                    N.x[E.s+1]=E.x[i];
                    N.x[i]=E.x[E.s+1];
                    N.cc=cc;
                    N.s=E.s+1;
                    N.lcost=b;
                    N.rcost=rcost;
                    H.Insert(N);
                }
                delete [] E.x;          // 完成结点扩展
            }
        try(H.DeleteMin(E);)           // 取下一扩展结点
        catch (OutOfBounds){break;}   // 堆已空
    }
    if(bestc==NoEdge) return NoEdge; // 无回路
    // 将最优解复制到 v[1:n]
    for(i=0;i<n;i++) v[i+1]=bestp[i];
    while(true){// 释放最小堆中所有结点
        delete [] E.x;
        try(H.DeleteMin(E);)
        catch (OutOfBounds){break;}
    }
    return bestc;

```

}

习题 6-9 解旅行售货员问题的分支限界法中保存已产生的排列树
试修改解旅行售货员问题的分支限界法,使得算法保存已产生的排列树。

分析与解答:

排列树中结点类型定义为:

```
class bbnode {
public:
    bbnode *parent;
    int s,*x;
};
```

保存已产生的排列树的旅行售货员问题的分支限界法如下。

```
template<class Type>
class Traveling{
    friend int main();
public:
    Type BBTSP(int v[]);
private:
    int n;
    Type **a, NoEdge, cc, bestc;
};

template<class Type>
class MinHeapNode{
    friend Traveling<Type>;
public:
    operator Type () const{return lcost;}
private:
    Type lcost, cc, rcost;
    bbnode *ptr;
};

template<class Type>
Type Traveling<Type>::BBTSP(int v[])
{// 解旅行售货员问题的优先队列式分支限界法
    // 定义最小堆的容量为 100000
    MinHeap<MinHeapNode<Type> > H(100000);
    Type *MinOut=new Type[n+1];
    // 计算 MinOut[i]=顶点 i 的最小出边费用
    Type MinSum=0;
```



```

for(int i=1;i<=n;i++){
    Type Min=NoEdge;
    for(int j=1;j<=n;j++){
        if(a[i][j]!=NoEdge && (a[i][j]<Min || Min==NoEdge))
            Min=a[i][j];
        if(Min==NoEdge) return NoEdge;
        MinOut[i]=Min;
        MinSum+=Min;
    }
}
// 初始化
MinHeapNode<Type> E;
bbnode *bb=new bbnode;
bb->parent=0;
bb->x=new int[n];
bb->s=0;
for(int j=0;j<n;j++)bb->x[j]=j+1;
E.ptr=bb;E.cc=0;E.rcost=MinSum;
Type bestc=NoEdge;
// 搜索排列空间树
while (E.ptr->s<n-1){
    if(E.ptr->s==n-2){
        if(a[E.ptr->x[n-2]][E.ptr->x[n-1]]!=NoEdge &&
            a[E.ptr->x[n-1]][1]!=NoEdge && (E.cc+
            a[E.ptr->x[n-2]][E.ptr->x[n-1]]+a[E.ptr->x[n-1]][1]
            <bestc || bestc==NoEdge)){
            bestc=E.cc+a[E.ptr->x[n-2]][E.ptr->x[n-1]]+a[E.ptr->x[n-1]][1];
            E.cc=bestc;
            E.lcost=bestc;
            E.ptr->s++;
            H.Insert(E);
        }
    }
    else{// 产生当前扩展结点的儿子结点
        for(int i=E.ptr->s+1;i<n;i++){
            if(a[E.ptr->x[E.ptr->s]][E.ptr->x[i]]!=NoEdge){
                Type cc=E.cc+a[E.ptr->x[E.ptr->s]][E.ptr->x[i]];
                Type rcost=E.rcost-MinOut[E.ptr->x[E.ptr->s]];
                Type b=cc+rcost;
                if(b<bestc || bestc==NoEdge){
                    // 子树可能含最优解
                    // 结点插入最小堆
                    bbnode *bb=new bbnode;
                    bb->parent=E.ptr;
                    bb->x=new int[n];

```

```

        bb->s=E.ptr->s+1;
        for(int j=0;j<n;j++)bb->x[j]=E.ptr->x[j];
        bb->x[E.ptr->s+1]=E.ptr->x[i];
        bb->x[i]=E.ptr->x[E.ptr->s+1];
        MinHeapNode<Type> N;
        N.cc=cc;
        N.lcost=b;
        N.rcost=rcost;
        N.ptr=bb;
        H.Insert(N);
    }
}
// 完成结点扩展
try {H.DeleteMin(E);} // 取下一扩展结点
catch (OutOfBounds) {break;}
}
if(bestc==NoEdge) return NoEdge;
// 将最优解复制到 v[1:n]
bb=E.ptr;v[n]=bb->x[n-1];
for(j=n;j>1;j--){
    v[j-1]=bb->x[j-2];
    bb=bb->parent;
}
return bestc;
}

```

习题 6-10 电路板排列问题的队列式分支限界法

试设计解电路板排列问题的队列式分支限界法,并使算法在运行结束时输出最优解和最优值。

分析与解答:

```

class BoardNode{
    friend int FIFOArr(int **,int,int,int * );
public:
    operator int() const{return'cd;'}
private:
    int *x,s,cd,*now;
};

```

解电路板排列问题的队列式分支限界法实现如下。

```

int FIFOArr(int **B,int n,int m,int * bestx)
{// 解电路板排列问题的队列式分支限界法

```

```

Queue<BoardNode> H;                // 活结点队列
// 初始化
BoardNode E;
E.x=new int[n+1];
E.s=0;E.cd=0;
E.now=new int[m+1];
int *total=new int[m+1];
// now[i]=x[1:s]所含连接块 i 中电路板数
// total[i]=连接块 i 中电路板数
for(int i=1;i<=m;i++){total[i]=0;E.now[i]=0;}
for(i=1;i<=n;i++){
    E.x[i]=i;                        // 初始排列为 1 2 3 ... n
    for(int j=1;j<=m;j++) total[j]+=B[i][j]; // 连接块 j 中电路板数
}
int bestd=m+1;                       // 当前最小密度
while(true){// 结点扩展
    if(E.s==n-1){// 仅一个儿子结点
        int ld=0;                    // 最后一块电路板的密度
        for(int j=1;j<=m;j++) ld+=B[E.x[n]][j];
        if(ld<bestd && E.cd<bestd){// 密度更小的电路板排列
            bestd=max(ld,E.cd);
            for(int k=0;k<=n;k++)bestx[k]=E.x[k];
        }
        else delete[] E.x;
        delete[] E.now;
    }
    else{// 产生当前扩展结点的所有儿子结点
        for(int i=E.s+1;i<=n;i++){
            BoardNode N;
            N.now=new int[m+1];
            for(int j=1;j<=m;j++)
                // 新插入的电路板
                N.now[j]=E.now[j]+B[E.x[i]][j];
            int ld=0;                  // 新插入电路板的密度
            for(j=1;j<=m;j++)
                if(N.now[j]>0 && total[j]!=N.now[j])ld++;
            N.cd=max(ld,E.cd);
            if(N.cd<bestd){// 可能产生更好的叶结点
                N.x=new int[n+1];
                N.s=E.s+1;
                for(int j=1;j<=n;j++) N.x[j]=E.x[j];
                N.x[N.s]=E.x[i];
                N.x[i]=E.x[N.s];
                H.Add(N);
            }
        }
    }
}

```

```

    }
    else delete[] N.now;
    }
    delete[] E.x;
} // 完成当前结点扩展
if (H.IsEmpty()) break;
else H.Delete(E); // 取下一扩展结点
}
return bestd;
}

```

算法实现题 6-1 最小长度电路板排列问题 (习题 6-11)

★问题描述:

最小长度电路板排列问题是大规模电子系统设计中提出的实际问题。该问题的提法是, 将 n 块电路板以最佳排列方案插入带有 n 个插槽的机箱中。 n 块电路板的不同的排列方式对应于不同的电路板插入方案。

设 $B = \{1, 2, \dots, n\}$ 是 n 块电路板的集合。集合 $L = \{N_1, N_2, \dots, N_m\}$ 是 n 块电路板的 m 个连接块。其中每个连接块 N_i 是 B 的一个子集, 且 N_i 中的电路板用同一根导线连接在一起。

例如, 设 $n=8, m=5$ 。给定 n 块电路板及其 m 个连接块如下:

$B = \{1, 2, 3, 4, 5, 6, 7, 8\}; L = \{N_1, N_2, N_3, N_4, N_5\};$

$N_1 = \{4, 5, 6\}; N_2 = \{2, 3\}; N_3 = \{1, 3\}; N_4 = \{3, 6\}; N_5 = \{7, 8\}.$

这 8 块电路板的一个可能的排列如图 6-1 所示。

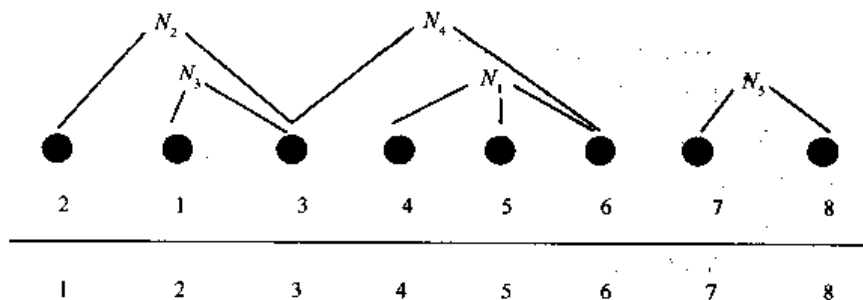


图 6-1 最小长度电路板排列

在最小长度电路板排列问题中, 连接块的长度是指该连接块中第 1 块电路板到最后 1 块电路板之间的距离。例如, 在图 6-1 所示的电路板排列中, 连接块 N_4 的第 1 块电路板在插槽 3 中, 它的最后 1 块电路板在插槽 6 中, 因此 N_4 的长度为 3。同理 N_2 的长度为 2。图 6-1 中连接块最大长度为 3。试设计一个队列式分支限界法找出所给 n 个电路板的最佳排列, 使得 m 个连接块中最大长度达到最小。

★编程任务:

对于给定的电路板连接块, 设计一个队列式分支限界法, 找出所给 n 个电路板的最佳排列, 使得 m 个连接块中最大长度达到最小。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m ($1 \leq m, n \leq 20$)。接下来的 n 行中, 每行有 m 个数。第 k 行的第 j 个数为 0 表示电路板 k 不在连接块 j 中, 为 1 表示电路板 k 在连接块 j 中。

★结果输出:

将计算出的电路板排列最小长度及其最佳排列输出到文件 output.txt。文件的第 1 行是最小长度; 接下来的 1 行是最佳排列。

输入文件示例	输出文件示例
input.txt	output.txt
8 5	4
1 1 1 1 1	5 4 3 1 6 2 8 7
0 1 0 1 0	
0 1 1 1 0	
1 0 1 1 0	
1 0 1 0 0	
1 1 0 1 0	
0 0 0 0 1	
0 1 0 0 1	

分析与解答:

与最小密度电路板排列问题类似, 结点元素类型是 BoardNode。

```
class BoardNode{
    friend int FIF0Boards(int **, int, int, int *&);
public:
    operator int() const{return cd;}
    int len();
private:
    int *x, s, cd, *low, *high;
};
```

其中, len 计算当前排列的最小长度。

```
int BoardNode::len()
{
    int tmp=0;
    for(int k=1;k<=m;k++)
        if(low[k]<=n && high[k]>0 && tmp<high[k]-low[k])tmp=high[k]-low[k];
    return tmp;
}
```

解最小长度电路板排列问题的队列式分支限界法如下。

```

int FIFOBoards(int **B, int n, int m, int *&bestx)
{
    Queue<BoardNode> Q;
    BoardNode E;
    E.x=new int[n+1];
    E.s=0;E.cd=0;E.low=new int[m+1];E.high=new int[m+1];
    for(int i=1;i<=m;i++){E.high[i]=0;E.low[i]=n+1;}
    for(i=1;i<=n;i++) E.x[i]=i;
    int bestd=n+1;
    bestx=0;
    do{
        if(E.s==n-1){
            for(int j=1;j<=m;j++){
                if(B[E.x[n]][j] && n>E.high[j])E.high[j]=n;
            }
            int ld=E.len();
            if(ld<bestd){
                delete[] bestx;
                bestx=E.x;
                bestd=ld;
            }
            else delete[] E.x;
            delete[] E.low;delete[] E.high;
        }
        else{
            int curr=E.s+1;
            for(int i=E.s+1;i<=n;i++){
                BoardNode N;
                N.low=new int[m+1];
                N.high=new int[m+1];
                for(int j=1;j<=m;j++){
                    N.low[j]=E.low[j];N.high[j]=E.high[j];
                    if(B[E.x[i]][j]){
                        if(curr<N.low[j])N.low[j]=curr;
                        if(curr>N.high[j])N.high[j]=curr;
                    }
                }
                N.cd=N.len();
                if(N.cd<bestd){
                    N.x=new int[n+1];
                    N.s=E.s+1;
                    for(int j=1;j<=n;j++) N.x[j]=E.x[j];
                    N.x[N.s]=E.x[i];
                    N.x[i]=E.x[N.s];
                }
            }
        }
    } while(!Q.empty());
    return bestx;
}

```

```

        Q.Add(N);
    }
    else {delete[] N.low;delete[] N.high;}
}
delete[] E.x;
}
try {Q.Delete(E);}
catch(OutOfBounds) {return bestd;}
}while(!Q.IsEmpty());
return bestd;
}

```

算法实现题 6-2 最小长度电路板排列问题 (习题 6-12)

★问题描述:

最小长度电路板排列问题是大规模电子系统设计中提出的实际问题。该问题的提法是, 将 n 块电路板以最佳排列方案插入带有 n 个插槽的机箱中。 n 块电路板的不同的排列方式对应于不同的电路板插入方案。

设 $B = \{1, 2, \dots, n\}$ 是 n 块电路板的集合。集合 $L = \{N_1, N_2, \dots, N_m\}$ 是 n 块电路板的 m 个连接块。其中每个连接块 N_i 是 B 的一个子集, 且 N_i 中的电路板用同一根导线连接在一起。

例如, 设 $n=8, m=5$ 。给定 n 块电路板及其 m 个连接块如下:

$B = \{1, 2, 3, 4, 5, 6, 7, 8\}; L = \{N_1, N_2, N_3, N_4, N_5\};$

$N_1 = \{4, 5, 6\}; N_2 = \{2, 3\}; N_3 = \{1, 3\}; N_4 = \{3, 6\}; N_5 = \{7, 8\}。$

这 8 块电路板的一个可能的排列如图 6-1 所示。

在最小长度电路板排列问题中, 连接块的长度是指该连接块中第 1 块电路板到最后 1 块电路板之间的距离。例如在图 6-1 所示的电路板排列中, 连接块 N_4 的第 1 块电路板在插槽 3 中, 它的最后 1 块电路板在插槽 6 中, 因此 N_4 的长度为 3。同理 N_2 的长度为 2。图 6-1 中连接块最大长度为 3。试设计一个优先队列式分支限界法找出所给 n 个电路板的最佳排列, 使得 m 个连接块中最大长度达到最小。

★编程任务:

对于给定的电路板连接块, 设计一个优先队列式分支限界法, 找出所给 n 个电路板的最佳排列, 使得 m 个连接块中最大长度达到最小。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m ($1 \leq m, n \leq 20$)。接下来的 n 行中, 每行有 m 个数。第 k 行的第 j 个数为 0 表示电路板 k 不在连接块 j 中, 为 1 表示电路板 k 在连接块 j 中。

★结果输出:

将计算出的电路板排列最小长度及其最佳排列输出到文件 output.txt。文件的第 1 行是最小长度; 接下来的 1 行是最佳排列。

输入文件示例

```
input.txt
8 5
1 1 1 1 1
0 1 0 1 0
0 1 1 1 0
1 0 1 1 0
1 0 1 0 0
1 1 0 1 0
0 0 0 0 1
0 1 0 0 1
```

输出文件示例

```
output.txt
4
5 4 3 1 6 2 8 7
```

分析与解答:

与最小密度电路板排列问题类似, 结点元素类型是 BoardNode。

```
class BoardNode{
    friend int BBArrange(int **, int, int, int *&);
public:
    operator int() const{return cd;}
    int len();
private:
    int *x, s, cd, *low, *high;
};
```

其中, len 计算当前排列的最小长度。

```
int BoardNode::len()
{
    int tmp=0;
    for(int k=1;k<=m;k++)
        if(low[k]<=n && high[k]>0 && tmp<high[k]-low[k])tmp=high[k]-low[k];
    return tmp;
}
```

解最小长度电路板排列问题的优先队列式分支限界法如下。

```
int BBArrange(int **B, int n, int m, int *&bestx)
{
    MinHeap<BoardNode> H(2000000);
    BoardNode E;
    E.x=new int[n+1];
    E.s=0;E.cd=0;E.low=new int[m+1];
    E.high=new int[m+1];
    for(int i=1;i<=m;i++){E.high[i]=0;E.low[i]=n+1;}
```



```

for(i=1;i<=n;i++) E.x[i]=i;
int bestd=n+1;
bestx=0;
do{
    if(E.s==n-1){
        for(int j=1;j<=m;j++){
            if(B[E.x[n]][j] && n>E.high[j])E.high[j]=n;
        }
        int ld=E.len();
        if(ld<bestd){
            delete[] bestx;
            bestx=E.x;
            bestd=ld;
        }
        else delete[] E.x;
        delete[] E.low;delete[] E.high;
    }
    else{
        int curr=E.s+1;
        for(int i=E.s+1;i<=n;i++){
            BoardNode N;
            N.low=new int[m+1];
            N.high=new int[m+1];
            for(int j=1;j<=m;j++){
                N.low[j]=E.low[j];N.high[j]=E.high[j];
                if(B[E.x[i]][j]){
                    if(curr<N.low[j])N.low[j]=curr;
                    if(curr>N.high[j])N.high[j]=curr;
                }
            }
            N.cd=N.len();
            if(N.cd<bestd){
                N.x=new int[n+1];
                N.s=E.s+1;
                for(int j=1;j<=n;j++) N.x[j]=E.x[j];
                N.x[N.s]=E.x[i];
                N.x[i]=E.x[N.s];
                H.Insert(N);
            }
            else{delete[] N.low;delete[] N.high;}
        }
        delete[] E.x;
    }
}
try {H.DeleteMin(E);}
catch(OutOfBounds){return bestd;}

```

```

        }while(E.cd<bestd);
    do{
        delete[] E.x;
        delete[] E.low;delete[] E.high;
        try {H.DeleteMin(E);}
        catch(...) {break;}
    }while(true);
    return bestd;
}

```

算法实现题 6-3 最小权顶点覆盖问题 (习题 6-13)

★问题描述:

给定一个赋权无向图 $G=(V,E)$, 每个顶点 $v \in V$ 都有一个权值 $w(v)$ 。如果 $U \subseteq V$, 且对任意 $(u,v) \in E$ 有 $u \in U$ 或 $v \in U$, 就称 U 为图 G 的一个顶点覆盖。 G 的最小权顶点覆盖是指 G 中所含顶点权之和最小的顶点覆盖。

★编程任务:

对于给定的无向图 G , 设计一个优先队列式分支限界法, 计算 G 的最小权顶点覆盖。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m , 表示给定的图 G 有 n 个顶点和 m 条边, 顶点编号为 $1, 2, \dots, n$ 。第 2 行有 n 个正整数表示 n 个顶点的权。接下来的 m 行中, 每行有 2 个正整数 u, v , 表示图 G 的一条边 (u, v) 。

★结果输出:

程序运行结束时, 将计算出的最小权顶点覆盖的顶点权之和以及最优解输出到文件 output.txt 中。文件的第 1 行是最小权顶点覆盖顶点权之和; 文件第 2 行是最优解 $x_i, 1 \leq i \leq n$, $x_i=0$ 表示顶点 i 不在最小权顶点覆盖中, $x_i=1$ 表示顶点 i 在最小权顶点覆盖中。

输入文件示例	输出文件示例
input.txt	output.txt
7 7	13
1 100 1 1 1 100 10	1 0 1 1 0 0 1
1 6	
2 4	
2 5	
3 6	
4 5	
4 6	
6 7	

分析与解答:

最小堆结点元素类型是 HeapNode。

```
class HeapNode{
```

```

friend class VC;
public:
    operator int () const{return cn;}
private:
    int i,cn,*x,*c;
};

```

解最小权顶点覆盖问题的优先队列式分支限界法如下。

```

class VC{
    friend MinCover(int **,int [],int);
private:
    void BBVC();
    bool cover(HeapNode E);
    void AddLiveNode(MinHeap<HeapNode> &H,HeapNode E,int cn,int i,bool ch);
    int **a,n,*w,*bestx,bestn;
};

void VC::BBVC()
{
    MinHeap<HeapNode> H(100000);
    HeapNode E;
    E.x=new int[n+1];
    E.c=new int[n+1];
    for(int j=1;j<=n;j++){E.x[j]=E.c[j]=0;}
    int i=1,cn=0;
    while(true){
        if(i>n){
            if(cover(E)){
                for(int j=1;j<=n;j++) bestx[j]=E.x[j];
                bestn=cn;
                break;
            }
        }
        else{
            if(!cover(E)) AddLiveNode(H,E,cn,i,1);
            AddLiveNode(H,E,cn,i,0);
            if(H.Size()==0) break;
            H.DeleteMin(E);
            cn=E.cn;
            i=E.i+1;
        }
    }
}

```

cover 判定图是否已完全覆盖。

```
bool VC::cover(HeapNode E)
{
    for(int j=1;j<=n;j++) if(E.x[j]==0 && E.c[j]==0) return false;
    return true;
}
```

AddLiveNode 将活结点加入堆中。

```
void VC::AddLiveNode(MinHeap<HeapNode> &H, HeapNode E, int cn, int i, bool ch)
{
    HeapNode N;
    N.x=new int[n+1];
    N.c=new int[n+1];
    for(int j=1;j<=n;j++) {N.x[j]=E.x[j];N.c[j]=E.c[j];}
    N.x[i]=ch;
    if(ch) {
        N.cn=cn+w[i];
        for(int j=1;j<=n;j++) if(a[i][j]) N.c[j]++;
    }
    else N.cn=cn;
    N.i=i;
    H.Insert(N);
}
```

MinCover 完成最小覆盖计算。

```
int MinCover(int **a, int v[], int n)
{
    VC Y;
    Y.w=new int [n+1];
    for(int j=1;j<=n;j++) {Y.w[j]=v[j];}
    Y.a=a;Y.n=n;Y.bestx=v;
    Y.BBVC();
    return Y.bestn;
}
```

算法的主函数如下。

```
int main()
{
    int u,v;
    cin>>n>>e;
```

```

Make2DArray(a, n+1, n+1);
for(int i=0; i<=n; i++)
    for(int j=0; j<=n; j++) a[i][j]=0;
p=new int[n+1];
for(i=1; i<=n; i++) cin>>p[i];
for(i=1; i<=e; i++){
    cin>>u>>v;
    a[u][v]=1; a[v][u]=1;
}
cout<<MinCover(a, p, n)<<endl;
for(i=1; i<=n; i++) cout<<p[i]<<" "; cout<<endl;
return 0;
}

```

算法实现题 6-4 无向图的最大割问题 (习题 6-14)

★问题描述:

给定一个无向图 $G=(V, E)$, 设 $U \subseteq V$ 是 G 的顶点集。对任意 $(u, v) \in E$, 若有 $u \in U$ 且 $v \in V-U$, 就称 (u, v) 为关于顶点集 U 的一条割边。顶点集 U 的所有割边构成图 G 的一个割。 G 的最大割是指 G 中所含边数最多的割。

★编程任务:

对于给定的无向图 G , 设计一个优先队列式分支限界法, 计算 G 的最大割。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m , 表示给定的图 G 有 n 个顶点和 m 条边, 顶点编号为 $1, 2, \dots, n$ 。接下来的 m 行中, 每行有 2 个正整数 u, v , 表示图 G 的一条边 (u, v) 。

★结果输出:

程序运行结束时, 将计算出的最大割的边数和顶点集 U 输出到文件 output.txt 中。文件的第 1 行是最大割的边数; 文件的第 2 行是表示顶点集 U 的向量 $x_i, 1 \leq i \leq n, x_i=0$ 表示顶点 i 不在顶点集 U 中, $x_i=1$ 表示顶点 i 在顶点集 U 中。

输入文件示例

input.txt

7 18

1 4

1 5

1 6

1 7

2 3

2 4

2 5

2 6

输出文件示例

output.txt

12

1 1 1 0 1 0 0

2 7
3 4
3 5
3 6
3 7
4 5
4 6
5 6
5 7
6 7

分析与解答：

最大堆结点元素类型是 HeapNode。

```
class HeapNode{
    friend class MCut;
public:
    operator int () const{return cut+e;}
private:
    int i, cut, e, *x;
};
```

解无向图最大割问题的优先队列式分支限界法如下。

```
class MCut{
    friend MaxCut(int **, int [], int, int);
private:
    void BBCut();
    void AddLiveNode(MaxHeap<HeapNode> &H, HeapNode E, bool ch);
    int **a, n, e, *bestx, bestn;
};
```

```
void MCut::BBCut()
{
    MaxHeap<HeapNode> H(HeapSize);
    HeapNode E;
    E.x=new int[n+1];
    E.cut=0;E.e=e;E.i=1;
    for(int j=1;j<=n;j++)E.x[j]=0;
    while(true){
        if(E.i>n){
            if(E.cut>bestn){
                for(int j=1;j<=n;j++) bestx[j]=E.x[j];
                bestn=E.cut;
            }
        }
    }
```

```

        }
    }
    else{
        AddLiveNode(H, E, 1);
        if(E.cut+E.e>bestn) AddLiveNode(H, E, 0);
    }
    if(H.Size()==0) break;
    H.DeleteMax(E);
}
}

void MCut::AddLiveNode(MaxHeap<HeapNode> &H, HeapNode E, bool ch)
{
    HeapNode N;
    int i=E.i;
    N.x=new int[n+1];
    for(int j=1;j<=n;j++)N.x[j]=E.x[j];
    N.x[i]=ch;N.cut=E.cut;N.e=E.e;
    if(ch){
        for(int j=1;j<=n;j++)
            if(a[i][j]){
                if(N.x[j]==0){N.cut++;N.e--;}
                else N.cut--;
            }
    }
    N.i=i+1;
    H.Insert(N);
}

```

MaxCut 完成最大割计算。

```

int MaxCut(int **a, int v[], int n, int e)
{
    MCut Y;
    Y.a=a;Y.n=n;Y.e=e;Y.bestn=0;Y.bestx=v;
    Y.BBCut();
    return Y.bestn;
}

```

算法的主函数如下。

```

int main()
{

```

```

int e, u, v;
cin >> n >> e;
Make2DArray(a, n+1, n+1);
for(int i=0; i<=n; i++)
    for(int j=0; j<=n; j++) a[i][j]=0;
p=new int[n+1];
for(i=1; i<=e; i++){
    cin >> u >> v;
    a[u][v]=1; a[v][u]=1;
}
cout << MaxCut(a, p, n, e) << endl;
for(i=1; i<=n; i++) cout << p[i] << " "; cout << endl;
return 0;
}

```

算法实现题 6-5 最小重量机器设计问题 (习题 6-15)

★问题描述:

设某一机器由 n 个部件组成, 每一种部件都可以从 m 个不同的供应商处购得。设 w_{ij} 是从供应商 j 处购得的部件 i 的重量, c_{ij} 是相应的价格。

设计一个优先队列式分支限界法, 给出总价格不超过 d 的最小重量机器设计。

★编程任务:

对于给定的机器部件重量和机器部件价格, 设计一个优先队列式分支限界法, 计算总价格不超过 d 的最小重量机器设计。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 3 个正整数 n , m 和 d 。接下来的 $2n$ 行, 每行 n 个数。前 n 行是 c , 后 n 行是 w 。

★结果输出:

将计算出的最小重量, 以及每个部件的供应商输出到文件 output.txt。

输入文件示例

输出文件示例

input.txt

output.txt

3 3 4

4

1 2 3

1 3 1

3 2 1

2 2 2

1 2 3

3 2 1

2 2 2

分析与解答:

状态空间树中结点类型为 bbnode。

```
class bbnode{
```



```

    friend Mach<int, int>;
    friend int Machine(int **, int **, int, int, int, int * );
private:
    bbnode *parent;
    int mj;
};

```

堆结点元素类型是 HeapNode。

```

template<class Typew, class Typep>
class HeapNode{
    friend Mach<Typew, Typep>;
public:
    operator Typew () const{return weight;}
private:
    Typep profit;
    Typew weight;
    int level;
    bbnode *ptr;
};

```

解最小重量机器设计问题的优先队列式分支限界法如下。

```

template<class Typew, class Typep>
class Mach{
    friend Typep Machine(Typep **, Typew **, Typew, int, int, int * );
public:
    Typep MinWeightMachine();
private:
    MinHeap<HeapNode<Typep, Typew>> *H;
    void AddLiveNode(Typep cp, Typew cw, int i, int j);
    bbnode *E;
    int n, m, *bestx;
    Typew cc, cw, **w;
    Typep cp, **c;
};

```

```

template<class Typew, class Typep>
Typep Mach<Typew, Typep>::MinWeightMachine()
{
    H=new MinHeap<HeapNode<Typep, Typew>>(HeapSize);
    bestx=new int[n+1];
    int i=1;
    E=0; cw=cp=0;

```

```

Typep besTypep=0;
while (i!=n+1){
    for(int j=1;j<=m;j++){
        Typew wt=cw+w[i][j];
        Typep ct=cp+c[i][j];
        if (ct<=cc) AddLiveNode(ct,wt,i+1,j);
    }
    HeapNode<Typep, Typew> N;
    try {H->DeleteMin(N);}
    catch (...) {break;}
    E=N.ptr;cw=N.weight;
    cp=N.profit;i=N.level;
}
if(i<=n) return 0;
for(int j=n;j>0;j--){
    bestx[j]=E->mj;
    E=E->parent;
}
return cw;
}

void Mach<int,int>::AddLiveNode(int cp,int cw,int i,int j)
{
    bbnode *b=new bbnode;
    b->parent=E;b->mj=j;
    HeapNode<Typep, Typew> N;
    N.profit=cp;N.weight=cw;
    N.level=i;N.ptr=b;
    H->Insert(N);
}

```

Machine 完成最小重量机器设计。

```

int Machine(int **c,int **w,int cc,int n,int m,int bestx[])
{
    Mach<int,int> K;
    K.c=c;K.w=w;K.cp=0;K.cw=0;K.cc=cc;
    K.n=n;K.m=m;K.bestx=bestx;
    int besTypep=K.MinWeightMachine();
    for (int j=1;j<=n;j++) bestx[j]=K.bestx[j];
    return besTypep;
}

```

算法的主函数如下。

```

int main()
{
    int cc, n, m, **c, **w, *bestx;
    cin >> n >> m >> cc;
    Make2DArray(c, n+1, m+1);
    Make2DArray(w, n+1, m+1);
    bestx = new int[n+1];
    for(int i=1; i<=n; i++)
        for(int j=1; j<=m; j++) cin >> c[i][j];
    for(i=1; i<=n; i++)
        for(int j=1; j<=m; j++) cin >> w[i][j];
    int answer = Machine(c, w, cc, n, m, bestx);
    if(answer > 0) {
        cout << answer << endl;
        for(int i=1; i<=n; i++) cout << bestx[i] << " ";
        cout << endl;
    }
    else cout << "No Solution!" << endl;
    return 0;
}

```

算法实现题 6-6 运动员最佳配对问题 (习题 6-16)

★问题描述:

羽毛球队有男女运动员各 n 人。给定 2 个 $n \times n$ 矩阵 P 和 Q 。 $P[i][j]$ 是男运动员 i 和女运动员 j 配对组成混合双打的男运动员竞赛优势; $Q[i][j]$ 是女运动员 i 和男运动员 j 配合的女运动员竞赛优势。由于技术配合和心理状态等各种因素影响, $P[i][j]$ 不一定等于 $Q[j][i]$ 。男运动员 i 和女运动员 j 配对组成混合双打的男女双方竞赛优势为 $P[i][j] * Q[j][i]$ 。设计一个算法, 计算男女运动员最佳配对法, 使各组男女双方竞赛优势的总和达到最大。

★编程任务:

设计一个优先队列式分支限界法, 对于给定的男女运动员竞赛优势, 计算男女运动员最佳配对法, 使各组男女双方竞赛优势的总和达到最大。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n ($1 \leq n \leq 20$)。接下来的 $2n$ 行, 每行 n 个数。前 n 行是 p , 后 n 行是 q 。

★结果输出:

将计算出的男女双方竞赛优势的总和的最大值输出到文件 output.txt。

输入文件示例

输出文件示例

input.txt

output.txt

3

52

```

10 2 3
2 3 4
3 4 5
2 2 2
3 5 3
4 5 1

```

分析与解答：

堆结点元素类型是 pref。

```

class pref{
public:
    operator int() const{return val;}
    int getbest();
private:
    void init();
    void Compute(int ii);
    int s, val, *r;
};

```

解运动员最佳配对问题的优先队列式分支限界法如下。

```

void pref::init()
{
    cin>>n;
    s=0;
    r=new int[n+1];
    bestr=new int[n+1];
    Make2DArray(p, n+1, n+1);
    Make2DArray(q, n+1, n+1);
    for(int i=1;i<=n;i++) r[i]=i;
    for(i=1;i<=n;i++)
        for(int j=1;j<=n;j++) cin>>p[i][j];
    for(i=1;i<=n;i++)
        for(int j=1;j<=n;j++) cin>>q[i][j];
}

void pref::Compute(int ii)
{
    for(int i=1,temp=0;i<=ii;i++)
        temp+=p[i][r[i]]*q[r[i]][i];
    val=temp;
}

```

```

int pref::getbest()
{
    MaxHeap<pref> H(HeapSize);
    pref E;
    E.init();
    while(true){
        if(E.s==n-1){
            E.Compute(n);
            if(E.val>best){
                delete [] bestr;
                bestr=E.r;best=E.val;
            }
            else delete [] E.r;
        }
        else{
            for(int i=E.s+1;i<=n;i++){
                pref N;
                N.r=new int[n+1];
                N.s=E.s+1;N.val=E.val;
                for(int j=1;j<=n;j++)N.r[j]=E.r[j];
                N.r[N.s]=E.r[i];
                N.r[i]=E.r[N.s];
                N.Compute(N.s);
                H.Insert(N);
            }
            delete [] E.r;
        }
        try {H.DeleteMax(E);}
        catch(OutOfBounds){return best;}
    }
}

```

算法实现题 6-7 n 皇后问题 (习题 6-18)

★问题描述:

在 $n \times n$ 格的棋盘上放置彼此不受攻击的 n 个皇后。按照国际象棋的规则,皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。 n 皇后问题等价于在 $n \times n$ 格的棋盘上放置 n 个皇后,任何 2 个皇后不放在同一行或同一列或同一斜线上。

★编程任务:

设计一个解 n 皇后问题的队列式分支限界法,计算在 $n \times n$ 个方格上放置彼此不受攻击的 n 个皇后的一个放置方案。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n 。

★结果输出:

将计算出的彼此不受攻击的 n 个皇后的一个放置方案输出到文件 output.txt。文件的第 1 行是 n 个皇后的放置方案。

输入文件示例

input.txt

5

输出文件示例

output.txt

1 3 5 2 4

分析与解答:

用习题 6-26 的算法框架设计。排列空间树中结点类型为 QNode。

```
template<class Type>
class QNode{
public:
    operator int() const{return cd;}
    int *x,i;
};
```

用队列式分支限界法搜索排列树的一般算法 fifobb 如下。

```
template<class Type>
void Queen<Type>::fifobb()
{
    Queue<QNode<Type>*> Q;    // 活结点队列
    QNode<Type> *E,* N;      // 当前扩展结点
    Init(E);
    // 搜索排列空间树
    while(true){
        if(Answer(E))Save(E);
        else
            for(int i=E->i+1;i<=n;i++){
                NewNode(N,E,i);
                if(Constrain(N))Q.Add(N);
            }
        // 取下一扩展结点
        if(!Getnext(Q,E))break;
    }
    Output();
}
```

结点可行性判定函数 Constrain 等函数通过类 Queen 的私有函数传递。

```
template<class Type>
class Queen{
    friend int main();
private:
```

```

    void Init(QNode<Type> *&E);
    bool Answer(QNode<Type> *E);
    void Save(QNode<Type> *&E);
    void NewNode(QNode<Type> *&N, QNode<Type> *E, int i);
    bool Constrain(QNode<Type> *E);
    bool Getnext(Queue<QNode<Type>*> &Q, QNode<Type> *&E);
    void Output();
    void fifobb();
    int n, *bestx;
    bool found;
};

```

```

template<class Type>
void Queen<Type>::Init(QNode<Type> *&E)
{
    E=new QNode<Type>;
    E->x=new int[n+1];
    for(int j=1;j<=n;j++)E->x[j]=j;
    E->i=0;
    bestx=new int[n+1];
    found=false;
}

```

```

template<class Type>
bool Queen<Type>::Answer(QNode<Type> *E)
{
    return E->i==n;
}

```

```

template<class Type>
void Queen<Type>::Save(QNode<Type> *&E)
{
    for(int k=1;k<=n;k++)bestx[k]=E->x[k];
    found=true;
}

```

```

template<class Type>
void Queen<Type>::NewNode(QNode<Type> *&N, QNode<Type> *E, int i)
{
    N=new QNode<Type>;
    N->x=new int[n+1];
    N->i=E->i+1;
    for(int j=1;j<=n;j++) N->x[j]=E->x[j];
    N->x[N->i]=E->x[i];
}

```

```

    N->x[i]=E->x[N->i];
}

template<class Type>
bool Queen<Type>::Constrain(QNode<Type> *E)
{
    for(int j=1;j<E->i;j++)
        if((abs(E->i-j)==abs(E->x[j]-E->x[E->i]))||(E->x[j]==E->x[E->i]->j))return false;
    return true;
}

template<class Type>
bool Queen<Type>::Getnext(Queue<QNode<Type>*> &Q,QNode<Type> *&E)
{
    if(found || Q.IsEmpty())return false;
    Q.Delete(E);
    return true;
}

template<class Type>
void Queen<Type>::Output()
{
    for(int i=1;i<=n;i++) cout<<bestx[i]<<" ";cout<<endl;
}

```

算法实现题 6-8 圆排列问题 (习题 6-19)

★问题描述:

给定 n 个大小不等的圆 c_1, c_2, \dots, c_n , 现要将这 n 个圆排进一个矩形框中, 且要求各圆与矩形框的底边相切。圆排列问题要求从 n 个圆的所有排列中找出有最小长度的圆排列。例如, 当 $n=3$, 且所给的 3 个圆的半径分别为 1, 1, 2 时, 这 3 个圆的最小长度的圆排列如图 6-2 所示, 其最小长度为 $2+4\sqrt{2}$ 。

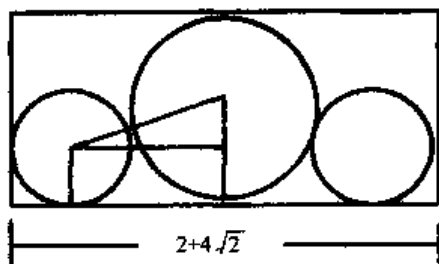


图 6-2 圆排列问题

★编程任务:

对于给定的 n 个圆, 设计一个优先队列式分支限界法, 计算 n 个圆的最佳排列方案, 使其长度达到

最小。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n ($1 \leq n \leq 20$)。接下来的 1 行有 n 个数, 表示 n 个圆的半径。

★结果输出:

将计算出的最小圆排列的长度输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3	7.65685
1 1 2	

分析与解答:

堆结点元素类型是 Circle。

```
class Circle{
    friend float CirclePerm(float *,int,float *&);
public:
    operator float() const{return len;}
private:
    float Center(int t);
    void Compute(int ii);
    float len,          // 当前圆排列的长度
          *x,           // 当前圆排列圆心横坐标
          *r;           // 当前圆排列
    int s;              // 待排列圆的个数
};
```

解圆排列问题的优先队列式分支限界法如下。

```
float Circle::Center(int t)
{
    // 计算当前所选择圆的圆心横坐标
    float temp=0;
    for(int j=1;j<t;j++){
        float valuex=x[j]+2.0*sqrt(r[t]*r[j]);
        if(valuex>temp) temp=valuex;
    }
    return temp;
}

void Circle::Compute(int ii)
{
    // 计算当前圆排列的长度
    float low=0,high=0;
    for(int i=1;i<=ii;i++){
        if(x[i]-r[i]<low) low=x[i]-r[i];
        if(x[i]+r[i]>high) high=x[i]+r[i];
    }
    len=high-low;
}
```

```

float CirclePerm(float *a, int n, float *&bestx)
{
    MinHeap<Circle> H(HeapSize);
    Circle E;
    E.x=new float [n+1];
    E.r=new float [n+1];
    E.r=a;E.s=0;E.len=FLT_MAX;bestx=0;
    float minlen=FLT_MAX;
    do{
        if(E.s==n-1){
            E.x[n]=E.Center(n);
            E.Compute(n);
            if(E.len<minlen){
                delete [] bestx;
                bestx=E.r;minlen=E.len;
            }
            else{delete [] E.x;delete [] E.r;}
        }
        else{
            for(int i=E.s+1;i<=n;i++){
                Circle N;
                N.x=new float [n+1];
                N.r=new float [n+1];
                N.s=E.s+1;N.len=E.len;
                for(int j=1;j<=n;j++){N.x[j]=E.x[j];N.r[j]=E.r[j];}
                N.r[N.s]=E.r[i];N.r[i]=E.r[N.s];
                N.x[N.s]=N.Center(N.s);
                N.Compute(N.s);
                if(N.len<minlen) H.Insert(N);
                else{delete [] N.x;delete [] N.r;}
            }
            delete [] E.x;delete [] E.r;
        }
        try{H.DeleteMin(E);}
        catch(OutOfBounds){return minlen;}
    }while(E.len<minlen);
    while(true){
        delete [] E.x;
        try{H.DeleteMin(E);}
        catch(...) {break;}
    }
    return minlen;
}

```

算法的主函数如下。

```
int main()
{
    cin>>n;
    p=new float[n+1];
    float *B;
    B=new float[n+1];
    for(int i=1;i<=n;i++) cin>>B[i];
    cout<<CirclePerm(B,n,p)<<endl;
    for(i=1;i<=n;i++) cout<<p[i]<<" ";cout<<endl;
    return 0;
}
```

算法实现题 6-9 布线问题 (习题 6-20)

★问题描述:

假设要将一组元件安装在一块线路板上,为此需要设计一个线路板布线方案。各元件的连线数由连线矩阵 conn 给出。元件 i 和元件 j 之间的连线数为 $\text{conn}(i,j)$ 。如果将元件 i 安装在线路板上位置 r 处,而将元件 j 安装在线路板上位置 s 处,则元件 i 和元件 j 之间的距离为 $\text{dist}(r,s)$ 。确定了所给的 n 个元件的安装位置,就确定了一个布线方案。与此布线方案相应的布线成本为 $\sum_{1 \leq i < j \leq n} \text{conn}(i,j) * \text{dist}(r,s)$ 。试设计一个优先队列式分支限界法,找出所给 n 个元件的布线成本最小的布线方案。

★编程任务:

对于给定的 n 个元件,设计一个优先队列式分支限界法,计算最佳布线方案,使布线费用达到最小。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n ($1 \leq n \leq 20$)。接下来的 $n-1$ 行,每行 $n-i$ 个数,表示元件 i 和元件 j 之间连线数, $1 \leq i < j \leq 20$ 。

★结果输出:

将计算出的最小布线费用以及相应的最佳布线方案输出到文件 output.txt。

输入文件示例

输出文件示例

input.txt

output.txt

3

10

2 3

1 3 2

3

分析与解答:

堆结点元素类型是 BoardNode。

```
class BoardNode{
    friend int BBArrangeBoards(int **,int,int *&);
public:
```

```

        operator int() const{return cd;}
        int len(int ** ,int ii);
private:
        int *x, s, cd;
};

```

解布线问题的优先队列式分支限界法如下。

```

int BoardNode::len(int ** conn, int ii)
{
    for(int i=1, sum=0; i<=ii; i++)
        for(int j=i+1; j<=ii; j++){
            int dist=x[i]>x[j]? x[i]-x[j]:x[j]-x[i];
            sum+=conn[i][j]*dist;
        }
    return sum;
}

int BBArrangeBoards(int **conn, int n, int *&bestx)
{
    MinHeap<BoardNode> H(HeapSize);
    BoardNode E;
    E.x=new int[n+1];
    E.s=0; E.cd=0;
    for(int i=1; i<=n; i++) E.x[i]=i;
    int bestd=INT_MAX;
    bestx=0;
    while(E.cd<bestd){
        if(E.s==n-1){
            int ld=E.len(conn, n);
            if(ld<bestd){
                delete[] bestx;
                bestx=E.x; bestd=ld;
            }
            else delete[] E.x;
        }
        else{
            for(int i=E.s+1; i<=n; i++){
                BoardNode N;
                N.x=new int[n+1];
                N.s=E.s+1;
                for(int j=1; j<=n; j++) N.x[j]=E.x[j];
                N.x[N.s]=E.x[i];
                N.x[i]=E.x[N.s];
            }
        }
    }
}

```

```

        N.cd=N.len(conn,N.s);
        if(N.cd<bestd) H.Insert(N);
        else delete[] N.x;
    }
    delete[] E.x;
}
try{H.DeleteMin(E);}
catch(OutOfBounds){return bestd;}
}
while(true){
    delete[] E.x;
    try{H.DeleteMin(E);}
    catch(...) {break;}
}
return bestd;
}

```

算法的主函数如下。

```

int main()
{
    cin>>n;
    p=new int[n+1];
    int **B;
    Make2DArray(B,n+1,n+1);
    for(int i=1;i<=n-1;i++)
        for(int j=i+1;j<=n;j++)cin>>B[i][j];
    cout<<BBArrangeBoards(B,n,p)<<endl;
    for(i=1;i<=n;i++) cout<<p[i]<<" ";cout<<endl;
    return 0;
}

```

算法实现题 6-10 最佳调度问题 (习题 6-21)

★问题描述:

假设有 n 个任务由 k 个可并行工作的机器完成。完成任务 i 需要的时间为 t_i 。试设计一个算法找出完成这 n 个任务的最佳调度,使得完成全部任务的时间最早。

★编程任务:

对任意给定的整数 n 和 k ,以及完成任务 i 需要的时间为 $t_i, i=1\sim n$ 。设计一个优先队列式分支限界法,计算完成这 n 个任务的最佳调度。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 k 。第 2 行的 n 个正整数是完成 n 个任务需要的时间。

★结果输出:

将计算出的完成全部任务的最早时间输出到文件 output.txt。

输入文件示例

input.txt

7 3

2 14 4 16 6 5 3

输出文件示例

output.txt

17

分析与解答:

堆结点元素类型是 Machine。

```
class Machine{
    friend int BBMachine();
public:
    operator int() const{return len[i];}
private:
    int i,dep,*len;
};
```

解最佳调度问题的优先队列式分支限界法如下。

```
int BBMachine()
{
    MinHeap<Machine> H(HeapSize);
    Machine E;
    E.len=new int[k];
    E.i=0;E.dep=0;
    for(int i=0;i<k;i++) E.len[i]=0;
    int dep=0;
    while(true){
        if(dep==n){
            int tmp=comp(E.len);
            if(tmp<best)best=tmp;
            delete[] E.len;
        }
        else{
            for(int i=0;i<k;i++){
                Machine N;
                N.len=new int[k];
                N.i=i; N.dep=dep;
                for(int j=0;j<k;j++) N.len[j]=E.len[j];
                N.len[i]+=t[dep];
                if(N.len[i]<best) H.Insert(N);
                else delete[] N.len;
            }
        }
    }
}
```

```

        delete[] E.len;
    }
    try {H.DeleteMin(E);}
    catch(OutOfBounds) {return best;}
    dep=E.dep+1;
}
while(true){
    delete[] E.len;
    try {H.DeleteMin(E);}
    catch(...) {break;}
}
return best;
}

```

算法的主函数如下。

```

int main()
{
    readin();
    cout<<BBMachine()<<endl;
    return 0;
}

```

其中, readin 读入初始数据并作初始化计算。

```

void readin()
{
    cin>>n>>k;
    len=new int[k];
    t.resize(n);
    for(int i=0;i<n;i++) cin>>t[i];
    for(i=0;i<k;i++) len[i]=0;
    best=bound();
}

```

bound 计算上界值。

```

int bound()
{
    sort(t.begin(),t.end(), greater<int>());
    for(int i=0;i<n;i++) len[ind(len)]+=t[i];
    return comp(len);
}

```

```

int ind(int *len)
{
    int tmp=0;
    for(int i=1;i<k;i++) if(len[i]<len[tmp]) tmp=i;
    return tmp;
}

int comp(int *len)
{
    int tmp=0;
    for(int i=0;i<k;i++) if(len[i]>tmp) tmp=len[i];
    return tmp;
}

```

算法实现题 6-11 无优先级运算问题 (习题 6-22)

★问题描述:

给定 n 个正整数和 4 个运算符 $+$ 、 $-$ 、 $*$ 、 $/$ ，且运算符无优先级，如 $2+3*5=25$ 。对于任意给定的整数 m ，试设计一个算法，用以上给出的 n 个数和 4 个运算符，产生整数 m ，且用的运算次数最少。给出的 n 个数中每个数最多只能用一次，但每种运算符可以任意使用。

★编程任务:

对于给定的 n 个正整数，设计一个优先队列式分支限界法，用最少的无优先级运算次数产生整数 m 。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m 。第 2 行是给定的用于运算的 n 个正整数。

★结果输出:

将计算出的产生整数 m 的最少无优先级运算次数以及最优无优先级运算表达式输出到文件 output.txt。

输入文件示例

input.txt

5 25

5 2 3 6 7

输出文件示例

output.txt

2

2+3*5

分析与解答:

堆结点元素类型是 Arit。

```

class Arit{
public:
    Arit(int n);
    Arit();
    operator int() const{return dep;}
    int dep,*num,*oper,*flag;
}

```



```
};

Arit::Arit(int n)
{
    num=new int[n];
    oper=new int[n];
    flag=new int[n];
    for(int i=0;i<n;i++) {num[i]=oper[i]=flag[i]=0;}
    dep=0;
}
```

解无优先级运算问题的优先队列式分支限界法如下。

```
int BBArity()
{
    MinHeap<Arit> H(HeapSize);
    Arit E(n);
    while(true){
        if(found(E)) {out(E);return 1;}
        else{
            for(int i=0;i<n;i++){
                if (!E.flag[i])
                    for(int j=0;j<4;j++){
                        Arit N(n);
                        newnode(N,E,i,j,dep);
                        H.Insert(N);
                    }
            }
            try {H.DeleteMin(E);}
            catch(OutOfBounds){return 0;}
            dep=E.dep+1;
        }
    }
}
```

其中, found 判定是否找到解; out 输出解; newnode 生成新结点。

```
bool found(Arit E)
{
    int x=E.num[0];
    for(int i=0;i<E.dep;i++){
        switch (E.oper[i]){
            case 0: x+=E.num[i+1]; break;
            case 1: x-=E.num[i+1]; break;
            case 2: x*=E.num[i+1]; break;
```

```

        case 3: x/=E.num[i+1]; break;
    }
}
return (x==m);
}

void out(Arit E)
{
    cout<<E.dep<<endl;
    for(int i=0;i<E.dep;i++){
        cout<<E.num[i];
        switch (E.oper[i]){
            case 0: cout<<"+"; break;
            case 1: cout<<"-"; break;
            case 2: cout<<"* "; break;
            case 3: cout<<"/"; break;
        }
    }
    cout<<E.num[E.dep]<<endl;
}

void newnode(Arit &N,Arit E,int i,int j,int dep)
{
    for(int k=0;k<n;k++){
        N.num[k]=E.num[k];
        N.oper[k]=E.oper[k];
        N.flag[k]=E.flag[k];
    }
    N.dep=dep;
    N.oper[dep]=j;
    N.num[dep]=a[i];
    N.flag[i]=1;
}

```

算法的主函数如下。

```

int main()
{
    readin();
    if(!BBArit())cout<<"No Solution!"<<endl;
    return 0;
}

void readin()

```

```

{
    cin>>n>>m;
    a=new int[n];
    for(int i=0;i<n;i++) cin>>a[i];
}

```

算法实现题 6-12 世界名画陈列馆问题 (习题 6-24)

★问题描述:

世界名画陈列馆由 $m \times n$ 个排列成矩形阵列的陈列室组成。为了防止名画被盗,需要在陈列室中设置警卫机器人哨位。每个警卫机器人除了监视它所在的陈列室外,还可以监视与它所在的陈列室相邻的上、下、左、右 4 个陈列室。试设计一个安排警卫机器人哨位的算法,使得名画陈列馆中每一个陈列室都在警卫机器人的监视之下,且所用的警卫机器人人数最少。

★编程任务:

设计一个优先队列式分支限界法,计算警卫机器人的最佳哨位安排,使得名画陈列馆中每一个陈列室都在警卫机器人的监视之下,且所用的警卫机器人人数最少。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 m 和 n ($1 \leq m, n \leq 20$)。

★结果输出:

将计算出的警卫机器人人数及其最佳哨位安排输出到文件 output.txt。文件的第 1 行是警卫机器人人数;接下来的 m 行中每行 n 个数,0 表示无哨位,1 表示有哨位。

输入文件示例

input.txt

4 4

输出文件示例

output.txt

4

0 0 1 0

1 0 0 0

0 0 0 1

0 1 0 0

分析与解答:

堆结点元素类型是 HeapNode。

```

class HeapNode{
public:
    HeapNode(int n, int m);
    HeapNode() {} ;
    operator int() const{return k;};
    void out(int n, int m);
    int i, j, k, t;
    int **x, **y;
};

```

```

HeapNode::HeapNode(int n, int m)
{
    Make2DArray(x, n+2, m+2);
    Make2DArray(y, n+2, m+2);
    for(int a=0; a<=n+1; a++)
        for(int b=0; b<=m+1; b++) {x[a][b]=0; y[a][b]=0;}
    for(a=0; a<=m+1; a++) {y[0][a]=1; y[n+1][a]=1;}
    for(a=0; a<=n+1; a++) {y[a][0]=1; y[a][m+1]=1;}
    i=1; j=1; k=t=0;
}

```

解世界名画陈列馆问题的优先队列式分支限界法如下（解空间结点控制关系与习题5-26相同）。

```

class Robot{
    friend int main();
private:
    void copy(int **x, int **y);
    void change(MinHeap<HeapNode> &H, HeapNode E, int i, int j);
    void init();
    void output();
    void pqbb();
    void compute();
    int **bestx;
    int n, m, best;
    bool p;
};

void Robot::pqbb()
{
    MinHeap<HeapNode> H(HeapSize);
    HeapNode E(n, m);
    while(true){
        int i=E.i, j=E.j, k=E.k, t=E.t;
        if(t==n*m){best=k; copy(bestx, E.x); return;}
        else{
            if(i<n) change(H, E, i+1, j);
            if((j<m) && ((E.y[i][j+1]==0) || (E.y[i][j+2]==0))) change(H, E, i, j+1);
            if(((E.y[i+1][j]==0) && (E.y[i][j+1]==0))) change(H, E, i, j);
        }
        try {H.DeleteMin(E);}
        catch(OutOfBounds){break;}
    }
}

```

```

void Robot::change(MinHeap<HeapNode> &H, HeapNode E, int i, int j)
{
    HeapNode N(n, m);
    N.i=E.i;N.j=E.j;N.k=E.k+1;N.t=E.t;
    for(int a=0;a<=n+1;a++)
        for(int b=0;b<=m+1;b++){
            N.x[a][b]=E.x[a][b];
            N.y[a][b]=E.y[a][b];
        }
    N.x[i][j]=1;
    for(int s=1;s<=5;s++){
        int p=i+d[s][1],q=j+d[s][2];
        N.y[p][q]++;
        if(N.y[p][q]==1)N.t++;
    }
    while(!(N.y[N.i][N.j]==0 || N.i>n)){
        N.j++;
        if(N.j>m){N.i++;N.j=1;}
    }
    H.Insert(N);
}

void Robot::copy(int **x, int **y)
{
    for(int i=0;i<=n;i++)
        for(int j=0;j<=m;j++)x[i][j]=y[i][j];
}

```

用 compute 完成计算。

```

void Robot::compute()
{
    Make2DArray(bestx, n+2, m+2);
    if(n==1 && m==1){cout<<1<<endl<<1<<endl;return;}
    pqbb();output();
}

```

算法的主函数如下。

```

int main()
{
    Robot X;
    X.init();
}

```

```

        X.compute();
        return 0;
    }

    void Robot::init()
    {
        cin>>n>>m;
        p=false;
        if(n<m){Swap(n,m);p=true;}
    }

```

算法实现题 6-13 子集空间树问题 (习题 6-25)

★问题描述:

试设计一个用队列式分支限界法搜索子集空间树的函数。该函数的参数包括结点可行性判定函数和上界函数等必要的函数,并将此函数用于解装载问题。

装载问题描述如下:有一批共 n 个集装箱要装上一艘载重量为 c 的轮船,其中集装箱 i 的重量为 w_i 。找出一种最优装载方案,将轮船尽可能装满,即在装载体积不受限制的情况下,将尽可能重的集装箱装上轮船。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 c 。 n 是集装箱数, c 是轮船的载重量。接下来的 1 行中有 n 个正整数,表示集装箱的重量。

★结果输出:

将计算出的最大装载重量输出到文件 output.txt。

输入文件示例

input.txt

5 10

7 2 6 5 4

输出文件示例

output.txt

10

分析与解答:

用队列式分支限界法搜索子集空间树的一般算法 fifobb 如下。

```

template<class Type>
void Loading<Type>::fifobb()
{
    Queue<QNode<Type>*> Q;    // 活结点队列
    int i=1;                  // 当前扩展结点所处的层
    QNode<Type> *E=0;          // 当前扩展结点
    QNode<Type> *bestE;         // 当前最优扩展结点
    // 搜索子集空间树
    while(true){
        // 检查左儿子结点
        if(Constraint(E,bestE,i)) EnQueue(Q,E,i,1);
    }
}

```

```

        // 检查右儿子结点
        if (Bound(E, i)) EnQueue(Q, E, i, 0);
        // 取下一扩展结点
        if (!Getnext(Q, E, i)) break;
    }
    Solution(bestE);    // 构造最优解
    Output();
}

```

其中, 子集树的结点类型为 QNode:

```

template<class Type>
class QNode{
public:
    QNode *parent;
    int i, LChild;
    Type weight;
};

```

结点可行性判定函数 Constraint 和上界函数 Bound 等必要的函数通过类 Loading 的私有函数传递。

```

template<class Type>
class Loading{
    friend MaxLoading(Type * , Type, int, int * );
private:
    void maxLoading(int i);
    bool Constraint(QNode<Type> *E, QNode<Type> *&bestE, int i);
    bool Bound(QNode<Type> *E, int i);
    void EnQueue(Queue<QNode<Type>*> &Q, QNode<Type> *E, int i, int ch);
    bool Getnext(Queue<QNode<Type>*> &Q, QNode<Type> *&E, int &i);
    void Solution(QNode<Type> *bestE);
    void Output();
    void fifobb();
    int n, *bestx;
    Type *w, c, r, Ew, wt, bestw;
};

template<class Type>
bool Loading<Type>::Constraint(QNode<Type> *E, QNode<Type> *&bestE, int i)
{
    wt = Ew + w[i];
    if (wt <= c) {
        if (wt > bestw) {bestw = wt; bestE = E;}
    }
}

```

```

        if(i<n) return true;
    }
    return false;
}

```

```

template<class Type>
bool Loading<Type>::Bound(QNode<Type> *E, int i)
{
    return (Ew+r>bestw && i<n);
}

```

```

template<class Type>
void Loading<Type>::EnQueue(Queue<QNode<Type>*> &Q, QNode<Type> *E, int i, int ch)
{ // 将活结点加入到活结点队列 Q 中
    QNode<Type> *b;
    b=new QNode<Type>;
    if(ch)b->weight=wt;
    else b->weight=Ew;
    b->parent=E;
    b->i=i;
    b->LChild=ch;
    Q.Add(b);
}

```

```

template<class Type>
bool Loading<Type>::Getnext(Queue<QNode<Type>*> &Q, QNode<Type> *&E, int &i)
{
    if(Q.IsEmpty())return false;
    Q.Delete(E); // 取下一扩展结点
    Ew=E->weight; // 新扩展结点所相应的载重量
    if(i<E->i)r-=w[i]; // 剩余集装箱重量
    i=E->i+1;
    return true;
}

```

```

template<class Type>
void Loading<Type>::Solution(QNode<Type> *bestE)
{ // 构造最优解
    int besti=bestE->i+1;
    for(int j=n;j>besti;j--) bestx[j]=0;bestx[besti]=1;
    for(j=besti-1;j>0;j--){
        bestx[j]=bestE->LChild;
        bestE=bestE->parent;
    }
}

```



```

}

template<class Type>
void Loading<Type>::Output()
{
    cout<<bestw<<endl;
    for(int i=1;i<=n;i++) cout<<bestx[i]<<" ";cout<<endl;
}

```

MaxLoading 实现装载问题的队列式分支限界法。

```

template<class Type>
Type MaxLoading(Type w[], Type c, int n, int bestx[])
{
    Loading<Type> X;
    X.c=c;X.n=n;X.Ew=0;X.bestw=0;X.r=0;
    X.w=w;X.bestx=bestx;
    for(int j=2;j<=n;j++)X.r+=w[j];
    X.fifobb();
    return X.bestw;
}

```

算法的主函数如下。

```

int main()
{
    int n,c;
    cin>>n>>c;
    int *w=new int[n+1];
    int *x=new int[n+1];
    for(int i=1;i<=n;i++)cin>>w[i];
    cout<<"Max loading is "<<MaxLoading(w, c, n, x)<<endl;
    cout<<"Loading vector is"<<endl;
    for(i=1;i<=n;i++) cout<<x[i]<<" ";cout<<endl;
    return 0;
}

```

算法实现题 6-14 排列空间树问题 (习题 6-26)

★问题描述:

试设计一个用队列式分支限界法搜索排列空间树的函数。该函数的参数包括结点可行性判定函数和上界函数等必要的函数,并将此函数用于解电路板排列问题。

电路板排列问题是大规模电子系统设计中提出的实际问题。该问题的提法是,将 n 块电

电路板以最佳排列方案插入带有 n 个插槽的机箱中。 n 块电路板的不同的排列方式对应于不同的电路板插入方案。

设 $B=\{1,2,\cdots,n\}$ 是 n 块电路板的集合。集合 $L=\{N_1,N_2,\cdots,N_m\}$ 是 n 块电路板的 m 个连接块。其中每个连接块 N_i 是 B 的一个子集,且 N_i 中的电路板用同一根导线连接在一起。在设计机箱时,插槽一侧的布线间隙由电路板排列的密度所确定。因此电路板排列问题要求对于给定电路板连接条件(连接块),确定电路板的最佳排列,使其具有最小密度。

★编程任务:

对于给定电路板连接条件,编程计算电路板的最佳排列,使其具有最小密度。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行是 2 个正整数 n 和 m ,表示有 n 块电路板和 m 个连接块。接下来的 n 行每行有 m 个数,第 i 行的第 j 个数 $a[i][j]=1$ 表示第 i 块电路板在第 j 个连接块中,否则第 i 块电路板不在第 j 个连接块中。

★结果输出:

程序运行结束时,将计算出的最小密度和电路板的最佳排列输出到文件 output.txt。文件的第 1 行是最小密度;文件的第 2 行是电路板的最佳排列。

输入文件示例	输出文件示例
input.txt	output.txt
8 5	4
1 1 1 1 1	2 3 4 5 1 6 7 8
0 1 0 1 0	
0 1 1 1 0	
1 0 1 1 0	
1 0 1 0 0	
1 1 0 1 0	
0 0 0 0 1	
0 1 0 0 1	

分析与解答:

用队列式分支限界法搜索排列空间树的一般算法 fifobb 如下。

```
template<class Type>
void Board<Type>::fifobb()
{
    Queue<QNode<Type>*> Q;    // 活结点队列
    QNode<Type> *E,*N;        // 当前扩展结点
    Init(E);
    // 搜索排列空间树
    while(true) {
        if(Answer(E)) Save(E);
        else
            for(int i=E->i+1;i<=n;i++){
                NewNode(N,E,i);
```

```

        if(Constrain(N) && Bound(N))EnQueue(Q,N,E,i);
        else DelNode(N);
    }
    // 取下一扩展结点
    if(!Getnext(Q,E))break;
}
Output();
}

```

其中,排列树的结点类型为 QNode。

```

template<class Type>
class QNode{
public:
    operator int() const{return cd;}
    int *x,i;
    Type cd,*now;
};

```

结点可行性判定函数 Constrain 和上界函数 Bound 等必要的函数通过类 Board 的私有函数传递。

```

template<class Type>
class Board{
    friend int main();
private:
    void Init(QNode<Type> *&E);
    bool Answer(QNode<Type> *E);
    void Save(QNode<Type> *&E);
    void NewNode(QNode<Type> *&N,QNode<Type> *E,int i);
    bool Constrain(QNode<Type> *E);
    bool Bound(QNode<Type> *E);
    void EnQueue(Queue<QNode<Type>*> &Q,QNode<Type> *N,QNode<Type> *E,int i);
    bool Getnext(Queue<QNode<Type>*> &Q,QNode<Type> *&E);
    void DelNode(QNode<Type> *&E);
    void Output();
    void fifobb();
    int n,m,*bestx;
    Type **B,bestd,*total;
};

template<class Type>
void Board<Type>::Init(QNode<Type> *&E)
{// 结点初始化

```

```

E=new QNode<Type>;
E->x=new int[n+1];
E->i=0;E->cd=0;
E->now=new Type[m+1];
total=new Type[m+1];
for(int i=1;i<=m;i++){total[i]=0;E->now[i]=0;}
for(i=1;i<=n;i++){
    E->x[i]=i;
    for(int j=1;j<=m;j++) total[j]+=B[i][j];
}
bestd=m+1;
}

template<class Type>
bool Board<Type>::Answer(QNode<Type> *E)
{ // 叶结点判定
    return E->i==n-1;
}

template<class Type>
void Board<Type>::Save(QNode<Type> *&E)
{ // 保存最优解
    int ld=0;        // 最后一块电路板的密度
    for(int j=1;j<=m;j++) ld+=B[E->x[n]][j];
    if(ld<bestd && E->cd<bestd){
        bestd=max(ld,E->cd);
        for(int k=0;k<=n;k++)bestx[k]=E->x[k];
    }
    else delete[] E->x;
    delete[] E->now;
}

template<class Type>
void Board<Type>::NewNode(QNode<Type> *&N, QNode<Type> *E, int i)
{ // 产生新结点
    N=new QNode<Type>;
    N->now=new Type[m+1];
    for(int j=1;j<=m;j++)
        N->now[j]=E->now[j]+B[E->x[i]][j];
    int ld=0;        // 新插入电路板的密度
    for(j=1;j<=m;j++)
        if(N->now[j]>0 && total[j]!=N->now[j])ld++;
    N->cd=max(ld,E->cd);
}

```

```

template<class Type>
bool Board<Type>::Constrain(QNode<Type> *E)
{ // 可行性约束
    return true;
}

template<class Type>
bool Board<Type>::Bound(QNode<Type> *E)
{ // 边界约束
    return E->cd<bestd;
}

template<class Type>
void Board<Type>::EnQueue(Queue<QNode<Type>*> &Q,
    QNode<Type> *N, QNode<Type> *E, int i)
{ // 结点入队列
    N->x=new int[n+1];
    N->i=E->i+1;
    for(int j=1;j<=n;j++) N->x[j]=E->x[j];
    N->x[N->i]=E->x[i];
    N->x[i]=E->x[N->i];
    Q.Add(N);
}

template<class Type>
void Board<Type>::DelNode(QNode<Type> *&E)
{ // 删除结点
    delete[] E->now;
}

template<class Type>
bool Board<Type>::Getnext(Queue<QNode<Type>*> &Q, QNode<Type> *&E)
{ // 取队列中下一结点
    if(Q.IsEmpty())return false;
    Q.Delete(E);
    return true;
}

template<class Type>
void Board<Type>::Output()
{ // 输出最优解
    cout<<bestd<<endl;
    for(int i=1;i<=n;i++) cout<<bestx[i]<<" ";cout<<endl;
}

```

算法实现题 6-15 一般解空间的队列式分支限界法 (习题 6-27)

★问题描述:

试设计一个用队列式分支限界法搜索一般解空间的函数。该函数的参数包括结点可行性判定函数和上界函数等必要的函数,并将此函数用于解布线问题。

印制电路板将布线区域划分成 $n \times m$ 个方格阵列如图 6-3 (a) 所示。精确的电路布线问题要求确定连接方格 a 的中点到方格 b 的中点的最短布线方案。在布线时,电路只能沿直线或直角布线,如图 6-3 (b) 所示。为了避免线路相交,已布了线的方格做了封锁标记,其他线路不允许穿过被封锁的方格。

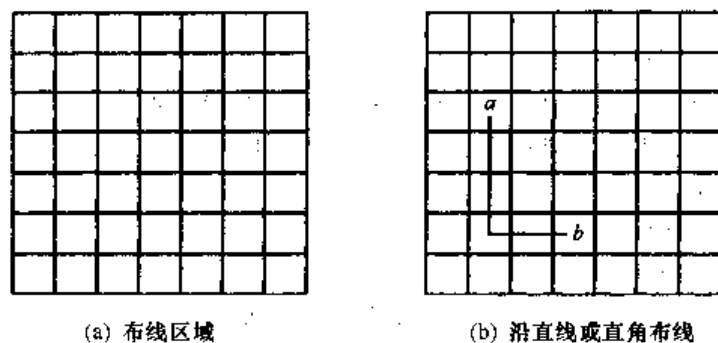


图 6-3 印制电路板问题

★编程任务:

对于给定的布线区域,编程计算最短布线方案。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 3 个正整数 n, m, k , 分别表示布线区域方格阵列的行数、列数和封闭的方格数。接下来的 k 行中, 每行 2 个正整数, 表示被封闭的方格所在的行号和列号。最后的 2 行, 每行也有 2 个正整数, 分别表示开始布线的方格 (p, q) 和结束布线的方格 (r, s) 。

★结果输出:

将计算出的最短布线长度和最短布线方案输出到文件 output.txt。文件的第 1 行是最短布线长度。从文件的第 2 行起, 每行 2 个正整数, 表示布线经过的方格坐标。如果无法布线则输出 “No Solution!”。

输入文件示例

input.txt

8 8 3

3 3

4 5

6 6

2 1

7 7

输出文件示例

output.txt

11

2 1

3 1

4 1

5 1

6 1

7 1

7 2
7 3
7 4
7 5
7 6
7 7

分析与解答:

用队列式分支限界法搜索一般解空间的算法 fifobb 如下。

```
template<class Type>
void Maze<Type>::fifobb()
{
    Queue<QNode<Type>*> Q;    // 活结点队列
    QNode<Type> *E, *N;        // 当前扩展结点
    Init(E);
    // 搜索一般解空间树
    while(true) {
        if (Answer(E)) Save(E);
        else
            for (int i=f(n,E); i<=g(n,E); i++) {
                NewNode(N, E, i);
                if (Constrain(N) && Bound(N)) EnQueue(Q, N, E, i);
                else DelNode(N);
            }
        // 取下一扩展结点
        if (!Getnext(Q, E)) break;
    }
    Output();
}
```

其中, 解空间树的结点类型为 QNode。

```
template<class Type>
class QNode {
public:
    operator int() const {return row;}
    Type row, col;
};
```

结点可行性判定函数 Constraint 和上界函数 Bound 等必要的函数通过类 Maze 的私有函数传递。

```
template<class Type>
class Maze {
```

```

friend int main();
private:
    void Init(QNode<Type>*&E);
    bool Answer(QNode<Type>*&E);
    void Save(QNode<Type>*&E);
    int f(int n, QNode<Type>*&E);
    int g(int n, QNode<Type>*&E);
    void NewNode(QNode<Type>*&N, QNode<Type>*&E, int i);
    bool Constrain(QNode<Type>*&E);
    bool Bound(QNode<Type>*&E);
    void EnQueue(Queue<QNode<Type>*> &Q, QNode<Type>*&N,
                QNode<Type>*&E, int i);
    bool Getnext(Queue<QNode<Type>*> &Q, QNode<Type>*&E);
    void DelNode(QNode<Type>*&E);
    void Output();
    void fifobb();
    int n, m;
    bool found;
    Type **grid, PathLen;
    QNode<Type> start, finish, offset[4], *path;
};

```

```

template<class Type>
void Maze<Type>::Init(QNode<Type> *&E)
{
    // 设置方格阵列“围墙”
    for(int i=0; i<=m+1; i++)
        grid[0][i]=grid[n+1][i]=1; // 顶部和底部
    for(i=0; i<=n+1; i++)
        grid[i][0]=grid[i][m+1]=1; // 左翼和右翼
    // 初始化相对位移
    offset[0].row=0; offset[0].col=1; // 右
    offset[1].row=1; offset[1].col=0; // 下
    offset[2].row=0; offset[2].col=-1; // 左
    offset[3].row=-1; offset[3].col=0; // 上
    E=new QNode<Type>;
    E->row=start.row;
    E->col=start.col;
    grid[start.row][start.col]=2;
}

```

```

template<class Type>
bool Maze<Type>::Answer(QNode<Type>*&E)
{ return false;}

```



```

template<class Type>
void Maze<Type>::Save(QNode<Type>*&E)
{}

template<class Type>
int Maze<Type>::f(int n, QNode<Type>*&E)
{return 1;}

template<class Type>
int Maze<Type>::g(int n, QNode<Type>*&E)
{return 4;}

template<class Type>
void Maze<Type>::NewNode(QNode<Type>*&N, QNode<Type>*&E, int i)
{
    N=new QNode<Type>;
    N->row=E->row+offset[i-1].row;
    N->col=E->col+offset[i-1].col;
}

template<class Type>
bool Maze<Type>::Constrain(QNode<Type>*&E)
{
    return grid[E->row][E->col]==0;
}

template<class Type>
bool Maze<Type>::Bound(QNode<Type>*&E)
{
    if(!found) found=E->row==finish.row && E->col==finish.col;
    return true;
}

template<class Type>
void Maze<Type>::EnQueue(Queue<QNode<Type>*> &Q,
                        QNode<Type>*&N, QNode<Type>*&E, int i)
{
    grid[N->row][N->col]=grid[E->row][E->col]+1;
    if(!found) Q.Add(N);
}

template<class Type>
void Maze<Type>::DelNode(QNode<Type>*&E)

```

```

{}

template<class Type>
bool Maze<Type>::Getnext (Queue<QNode<Type>*> &Q, QNode<Type>*&E)
{
    if(found || Q.IsEmpty())return false;
    Q.Delete(E);
    return true;
}

template<class Type>
void Maze<Type>::Output ()
{
    if(!found) {cout<<"No path!"<<endl;return;}
    // 构造最优解
    PathLen= grid[finish.row][finish.col]-2;
    path=new QNode<Type> [PathLen];
    // 从目标位置 finish 开始向起始位置回溯
    QNode<Type> N,E=finish;
    for (int j=PathLen-1;j>=0;j--){
        path[j]=E;
        // 找前驱位置
        for (int i=0;i<4;i++){
            N.row=E.row+offset[i].row;
            N.col=E.col+offset[i].col;
            if(grid[N.row][N.col]==j+2) break;
        }
        E=N;      // 向前移动
    }
    cout<<PathLen<<endl;
    cout<<start.row<<" "<<start.col<<endl;
    for(j=0;j<PathLen;j++)cout<<path[j].row<<" "<<path[j].col<<endl;
}

```

实现算法的主函数如下。

```

int main()
{
    int n,m,a,b,x;
    Maze<int> X;
    cin>>n>>m>>x;
    X.n=n;X.m=m;X.found=false;
    Make2DArray(X.grid,n+2,m+2);
    for(a=0;a<n+2;a++)
        for(b=0;b<m+2;b++)X.grid[a][b]=0;
}

```

```

for(x=x;x>=1;x--){cin>>a>>b;X.grid[a][b]=1;}
cin>>X.start.row>>X.start.col>>X.finish.row>>X.finish.col;
X.fifobb();
return 0;
}

```

算法实现题 6-16 子集空间树问题 (习题 6-28)

★问题描述:

试设计一个用优先队列式分支限界法搜索子集空间树的函数。该函数的参数包括结点可行性判定函数和上界函数等必要的函数,并将此函数用于解 0-1 背包问题。

0-1 背包问题描述如下:给定 n 种物品和一背包。物品 i 的重量是 w_i , 其价值为 v_i , 背包的容量为 C 。问应选择装入背包的物品,使得装入背包中物品的总价值最大,在选择装入背包的物品时,对每种物品 i 只有两种选择,即装入背包或不装入背包。不能将物品 i 装入背包多次,也不能只装入部分的物品 i 。

问题的形式化描述是,给定 $C > 0, w_i > 0, v_i > 0, 1 \leq i \leq n$, 要求找出 n 元 0-1 向量 (x_1, x_2, \dots, x_n) , $x_i \in \{0, 1\}, 1 \leq i \leq n$, 使得 $\sum_{i=1}^n w_i x_i \leq C$, 而且 $\sum_{i=1}^n v_i x_i$ 达到最大。因此, 0-1 背包问题是一个特殊的整数规划问题。

$$\begin{aligned} & \max \sum_{i=1}^n v_i x_i \\ & \begin{cases} \max \sum_{i=1}^n w_i x_i \leq C \\ x_i \in \{0, 1\}, 1 \leq i \leq n \end{cases} \end{aligned}$$

★数据输入:

由文件 input.txt 提供输入数据。文件第 1 行有 2 个正整数 n 和 C , 分别表示有 n 种物品, 背包的容量为 C 。接下来的 2 行中, 每行有 n 个数, 分别表示各物品的价值和重量。

★结果输出:

程序运行结束时, 将最佳装包方案及其最大价值输出到文件 output.txt。文件的第 1 行是最大价值, 第 2 行是最佳装包方案。

输入文件示例	输出文件示例
input.txt	output.txt
5 10	15
6 3 5 4 6	1 1 0 0 1
2 2 6 5 4	

分析与解答:

用优先队列式分支限界法搜索子集树的一般算法 pqbb 如下。

```

template<class Typew, class Typep>
void Knap<Typew, Typep>::pqbb()
{

```

```

NewHeap(H);
NewHeapNode(N);
int i=1;
Init();
// 搜索子集空间树
while(true){
    // 检查左儿子结点
    if(Constraint(N,i)){ AddLiveNode(N,i,1);}
    // 检查右儿子结点
    if(Bound(N,i)) AddLiveNode(N,i,0);
    // 取下一扩展结点
    if(!Getnext(N,i))break;
}
Solution(N);    // 构造最优解
Output();
}

```

其中,堆结点类型为 HeapNode。

```

template<class Typew, class Typep >
class HeapNode {
    friend Knap< Typew, Typep >;
public:
    operator Tp () const {return uprofit;}
private:
    Typep uprofit,profit;
    Typew weight;
    int level, *x;
};

```

结点可行性判定函数 Constraint 和上界函数 Bound 等必要的函数通过类 Knap 的私有函数传递。

```

template<class Typew, class Typep>
class Knap{
    friend Typep Knapsack(Typep * ,Typew * ,Typew, int, int * );
public:
    void pqbb();
private:
    MaxHeap<HeapNode<Typep, Typew> > *H;
    HeapNode<Typep, Typew> N;
    void NewHeap(MaxHeap<HeapNode<Typep, Typew> > *&H);
    void NewHeapNode(HeapNode<Typep, Typew> &N);
    void Init();

```

```

    Typep Bd(int i);
    bool Constraint(HeapNode<Typep, Typew> N, int i);
    bool Bound(HeapNode<Typep, Typew> N, int i);
    void AddLiveNode(HeapNode<Typep, Typew> &N, int i, bool ch);
    bool Getnext(HeapNode<Typep, Typew> &N, int &i);
    void Solution(HeapNode<Typep, Typew> N);
    void Output();
    Typew c, cw, wt, *w;
    Typep cp, up, bestp, *p;
    int n, * bestx;
};

template<class Typew, class Typep>
void Knap<Typew, Typep>::NewHeap(MaxHeap<HeapNode<Typep, Typew> >*&H)
{
    H=new MaxHeap<HeapNode<Typep, Typew> >(10000);
}

template<class Typew, class Typep>
void Knap<Typew, Typep>::NewHeapNode(HeapNode<Typep, Typew> &N)
{
    N.x=new int[n+1];
}

template<class Typew, class Typep>
void Knap<Typew, Typep>::Init()
{
    // 初始化
    bestx=new int [n+1];
    for(int k=0;k<=n;k++)bestx[k]=0;
    cw=cp=0;
    bestp=0;
    up=Bd(1);
}

template<class Typew, class Typep>
Typep Knap<Typew, Typep>::Bd(int i)
{// 计算结点所相应价值的上界
    Typew cleft=c - cw;        // 剩余容量
    Typep b=cp;                // 价值上界
    // 以物品单位重量价值递减序装填剩余容量
    while(i<=n && w[i]<=cleft){
        cleft-=w[i];
        b+=p[i];
        i++;
    }
}

```

```

    }
    // 装填剩余容量装满背包
    if(i<=n) b+=p[i]*cleft/w[i];
    return b;
}

template<class Typew, class Typep>
bool Knap<Typew, Typep>::Constraint (HeapNode<Typep, Typew> N, int i)
{
    wt=cw-w[i];
    if(wt<=c){
        if(cp+p[i]>bestp) bestp=cp+p[i];
        return true;
    }
    return false;
}

template<class Typew, class Typep>
bool Knap<Typew, Typep>::Bound (HeapNode<Typep, Typew> N, int i)
{
    up=Bd(i+1);
    return up>=bestp;
}

template<class Typep, class Typew>
void Knap<Typep, Typew>::AddLiveNode (HeapNode<Typep, Typew> &N, int i, bool ch)
// 将一个新结点插入到最大堆 H 中
{
    N.x=new int[n+1];
    for(int j=0;j<=n;j++)N.x[j]=bestx[j];
    N.uprofit=up;
    N.profit=cp;
    N.weight=cw;
    if(ch){N.weight=cw+w[i];N.profit=cp+p[i];}
    N.level=i+1;
    N.x[i]=ch;
    H->Insert(N);
}

template<class Typew, class Typep>
bool Knap<Typew, Typep>::Getnext (HeapNode<Typep, Typew> &N, int &i)
{
    H->DeleteMax(N);
    cw=N.weight;
    cp=N.profit;

```

```

        up=N. uprofit;
        i=N. level;
        for(int j=0;j<=n;j++)bestx[j]=N. x[j];
        if(i>n)return false;
        else return true;
    }

template<class Typep, class Typew>
void Knap<Typep, Typew>::Solution(HeapNode<Typep, Typew> N)
{
    // 构造当前最优解
    for(int j=0;j<=n;j++) bestx[j]=N. x[j];
    while(true){// 释放堆中所有结点
        try {H->DeleteMax(N);}
        catch(OutOfBounds){break;}
    }
}

template<class Typep, class Typew>
void Knap<Typep, Typew>::Output()
{
    cout<<cp<<endl;
    for(int j=1;j<=n;j++) cout<<bestx[j]<<" ";cout<<endl;
}

```

Knapsack 实现 0-1 背包问题的优先队列式分支限界法。

```

template<class Typew, class Typep>
Typep Knapsack(Typep *p, Typew *w, Typew c, int n, int *bestx)
{// 返回最大价值,bestx 返回最优解
    // 初始化
    Typew W=0;    // 装包物品重量
    Typep P=0;    // 装包物品价值
    // 定义依单位重量价值排序的物品数组
    Object *Q=new Object [n];
    for(int i=1;i<=n;i++){
        // 单位重量价值数组
        Q[i-1]. ID=i;
        Q[i-1]. d=1.0*p[i]/w[i];
        P+=p[i];
        W+=w[i];
    }
    if(W<=c) return P;    // 所有物品装包
    // 依单位重量价值排序

```

```

MergeSort(Q, n);
// 创建类 Knap 的数据成员
Knap<Typew, Typep> K;
K.p=new Typep [n+1];
K.w=new Typew [n+1];
for(i=1;i<=n;i++){
    K.p[i]=p[Q[i-1].ID];
    K.w[i]=w[Q[i-1].ID];
}
K.cp=0;K.cw=0;K.c=c;K.n=n;
// 调用 pqbb 求问题的最优解
K.pqbb();
bestp=K.bestp;
for(int j=1;j<=n;j++)bestx[Q[j-1].ID]=K.bestx[j];
delete [] Q;
delete [] K.w;
delete [] K.p;
delete [] K.bestx;
return bestp;
}

```

算法的主函数如下。

```

int main()
{
    cin>>n>>c;
    p=new int[n+1];
    w=new int[n+1];
    bestx=new int[n+1];
    for(int i=1;i<=n;i++) cin>>p[i];
    for(i=1;i<=n;i++) cin>>w[i];
    for(i=1;i<=n;i++) bestx[i]=1;
    cout<<Knapsack(p, w, c, n, bestx)<<endl;
    for(i=1;i<=n;i++) cout<<bestx[i]<<" ";cout<<endl;
    return 0;
}

```

算法实现题 6-17 排列空间树问题 (习题 6-29)

★问题描述:

试设计一个用优先队列式分支限界法搜索排列空间树的函数。该函数的参数包括结点可行性判定函数和上界函数等必要的函数,并将此函数用于解批处理作业调度问题。

给定 n 个作业的集合 $J = \{J_1, J_2, \dots, J_n\}$ 。每一个作业 J_i 都有 2 项任务分别在 2 台机器上完成。每个作业必须先由机器 1 处理,然后由机器 2 处理。作业 J_i 需要机器 j 的处

理时间为 t_{ij} , $i=1,2,\dots,n; j=1,2$ 。对于一个确定的作业调度, 设 F_{ij} 是作业 i 在机器 j 上完成处理的时间。所有作业在机器 2 上完成处理的时间和 $f = \sum_{i=1}^n F_{2i}$ 称为该作业调度的完成时间和。

批处理作业调度问题要求对于给定的 n 个作业, 制定最佳作业调度方案, 使其完成时间和达到最小。

★数据输入:

由文件 input.txt 提供输入数据。文件第 1 行有 1 个正整数 n , 表示作业数。接下来的 n 行中, 每行有 2 个正整数 i, j , 分别表示在机器 1 和机器 2 上完成该作业所需的处理时间。

★结果输出:

程序运行结束时, 将最佳作业调度方案及其完成时间和输出到文件 output.txt。文件的第 1 行是完成时间和, 第 2 行是最佳作业调度方案。

输入文件示例	输出文件示例
input.txt	output.txt
3	18
2 1	1 3 2
3 1	
2 3	

分析与解答:

用优先队列式分支限界法搜索排列空间树的一般算法 pqbb 如下。

```

template<class Type>
void Flowshop<Type>::pqbb()
{
    MinHeap<HeapNode<Type>> H(HeapSize);
    HeapNode<Type> E;
    Init(E);
    // 搜索排列空间树
    while(true) {
        if(Answer(E)) Save(E);
        else
            for(int i=E.s; i<n; i++) {
                Swap(E.x[E.s], E.x[i]);
                if(Constrain(E) && Bound(E)) AddLiveNode(H, E);
                Swap(E.x[E.s], E.x[i]);
            }
        // 取下一扩展结点
        if(!Getnext(H, E)) break;
    }
    Output();
}

```

其中,堆结点类型为 HeapNode。

```
template<class Type>
class HeapNode{
    friend Flowshop<Type>;
public:
    operator Type()const{return bb;}
private:
    void Init(int),
        NewNode(HeapNode<Type>, Type, Type, Type, int);
    int s,      // 已安排作业数
        *x;     // 当前作业调度
    Type f1,    // 机器 1 上最后完成时间
        f2,    // 机器 2 上最后完成时间
        sf2,   // 当前机器 2 上的完成时间和
        bb;    // 当前完成时间和下界
};

template<class Type>
void HeapNode<Type>::Init(int n)
{ // 堆结点初始化
    x=new int[n];
    for(int i=0;i<n;i++)x[i]=i;
    s=0;f1=0;f2=0;sf2= 0;bb=0;
}

template<class Type>
void HeapNode<Type>::NewNode(HeapNode<Type> E, Type Ef1, Type Ef2, Type Ebb, int n)
{ // 新结点
    x=new int[n];
    for(int i=0;i<n;i++)x[i]=E.x[i];
    f1=Ef1;f2=Ef2;sf2=E.sf2+f2;
    bb=Ebb;s=E.s+1;
}
```

结点可行性判定函数 Constraint 和上界函数 Bound 等必要的函数通过类 Flowshop 的私有函数传递。

```
template<class Type>
class Flowshop{
    friend Type Flow(Type **M, int n, int bestx[]);
public:
    void pqbb();
private:
```

```

Type Bd(HeapNode<Type>);
void Sort();
void Init(HeapNode<Type> &E);
bool Answer(HeapNode<Type> E);
void Save(HeapNode<Type> &E);
bool Constrain(HeapNode<Type> E);
bool Bound(HeapNode<Type> E);
void AddLiveNode(MinHeap<HeapNode<Type> > &H, HeapNode<Type> E);
bool Getnext(MinHeap<HeapNode<Type> > &H, HeapNode<Type> &E);
void Output();
int n, // 作业数
    *bestx; // 最优解
Type **M, // 各作业所需的处理时间数组
    **b, // 各作业所需的处理时间排序数组
    **a, // 数组 M 和 b 的对应关系数组
    bestc, // 最小完成时间和
    f1, f2, bb;
bool **y; // 工作数组
};

```

```

template<class Type>
void Flowshop<Type>::Sort()
{ // 对各作业在机器 1 和 2 上所需时间排序
    int *c=new int[n];
    for(int j=0;j<2;j++){
        for(int i=0;i<n;i++){b[i][j]=M[i][j];c[i]=i;}
        for(i=0;i<n-1;i++){
            for(int k=n-1;k>i;k--){
                if(b[k][j]<b[k-1][j]){
                    Swap(b[k][j], b[k-1][j]);
                    Swap(c[k], c[k-1]);
                }
            }
            for(i=0;i<n;i++) a[c[i]][j]=i;
        }
        delete[] c;
    }
}

```

```

template<class Type>
Type Flowshop<Type>::Bd(HeapNode<Type> E)
{ // 计算完成时间和下界
    for(int k=0;k<n;k++){
        for(int j=0;j<2;j++){
            y[k][j]=false;
        }
        for(k=0;k<=E.s;k++){

```

```

        for(int j=0;j<2;j++)
            y[a[E.x[k]][j]][j]=true;
f1=E.f1+M[E.x[E.s]][0];
f2=((f1>E.f2)? f1:E.f2)+M[E.x[E.s]][1];
Type sf2=E.sf2+f2;
Type s1=0,s2=0,k1=n-E.s,k2=n-E.s,f3=f2;
// 计算 s1 的值
for(int j=0;j<n;j++)
    if(!y[j][0]){
        k1--;
        if(k1==n-E.s-1) f3=(f2>f1+b[j][0]?f2:f1+b[j][0];
        s1+=f1+k1*b[j][0];
    }
// 计算 s2 的值
for(j=0;j<n;j++)
    if(!y[j][1]){
        k2--;
        s1+=b[j][1];
        s2+=f3+k2*b[j][1];
    }
// 返回完成时间和下界
return sf2+((s1>s2)?s1:s2);
}

```

```

template<class Type>
void Flowshop<Type>::Init(HeapNode<Type> &E)
{
    Sort();    // 对各作业在机器 1 和 2 上所需时间排序
    E.Init(n);
}

```

```

template<class Type>
bool Flowshop<Type>::Answer(HeapNode<Type> E)
{
    return E.s==n;
}

```

```

template<class Type>
void Flowshop<Type>::Save(HeapNode<Type> &E)
{
    if(E.sf2<bestc){
        bestc=E.sf2;
        for(int i=0;i<n;i++)bestx[i]=E.x[i];
    }
}

```

```

        delete[] E.x;
    }

template<class Type>
bool Flowshop<Type>::Constrain(HeapNode<Type> E)
{
    return true;
}

template<class Type>
bool Flowshop<Type>::Bound(HeapNode<Type> E)
{
    bb=Bd(E);
    return bb<bestc;
}

template<class Type>
void Flowshop<Type>::AddLiveNode(MinHeap<HeapNode<Type> > &H,
                                HeapNode<Type> E)
{
    // 将新结点插入堆中
    HeapNode<Type> N;
    N.NewNode(E, f1, f2, bb, n);
    H.Insert(N);
}

template<class Type>
bool Flowshop<Type>::Getnext(MinHeap<HeapNode<Type> > &H, HeapNode<Type> &E)
{
    try {H.DeleteMin(E);} // 取下一扩展结点
    catch(OutOfBounds) {return false;}
    return true;
}

template<class Type>
void Flowshop<Type>::Output()
{
    cout<<bestc<<endl;
    for(int j=0;j<n;j++) cout<<bestx[j]+1<<" ";cout<<endl;
}

```

Flow 实现批处理作业调度问题的优先队列式分支限界法。

```

template<class Type>
Type Flow(Type **M, int n, int bestx[])

```

```

{
    Flowshop<Type> X;
    X.M=M; X.n=n; X.bestx=bestx; X.bestc=INT_MAX;
    Make2DArray(X.a, n, 2);
    Make2DArray(X.b, n, 2);
    Make2DArray(X.y, n, 2);
    X.pqbb();
    return X.bestc;
}

```

算法的主函数如下。

```

int main()
{
    cin>>n;
    Make2DArray(M, n, 2);
    bestx=new int[n+1];
    for(int i=0;i<n;i++)
        for(int j=0;j<2;j++)cin>>M[i][j];
    cout<<Flow(M, n, bestx)<<endl;
    for(i=0;i<n;i++)cout<<bestx[i]+1<<" ";cout<<endl;
    return 0;
}

```

算法实现题 6-18 一般解空间的优先队列式分支限界法 (习题 6-30)

★问题描述:

试设计一个用优先队列式分支限界法搜索一般解空间的函数。该函数的参数包括结点可行性判定函数和上界函数等必要的函数,并将此函数用于解布线问题。

印刷电路板将布线区域划分成 $n \times m$ 个方格阵列如图 6-3 (a) 所示。精确的电路布线问题要求确定连接方格 a 的中点到方格 b 的中点的最短布线方案。在布线时,电路只能沿直线或直角布线,如图 6-3 (b) 所示。为了避免线路相交,已布了线的方格做了封锁标记,其他线路不允许穿过被封锁的方格。

★编程任务:

对于给定的布线区域,编程计算最短布线方案。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 3 个正整数 n, m, k , 分别表示布线区域方格阵列的行数、列数和封闭的方格数。接下来的 k 行中, 每行 2 个正整数, 表示被封闭的方格所在的行号和列号。最后的 2 行, 每行也有 2 个正整数, 分别表示开始布线的方格 (p, q) 和结束布线的方格 (r, s) 。

★结果输出:

将计算出的最短布线长度和最短布线方案输出到文件 output.txt。文件的第 1 行是最短

布线长度。从文件的第 2 行起, 每行 2 个正整数, 表示布线经过的方格坐标。如果无法布线则输出 “No Solution!”。

输入文件示例	输出文件示例
input.txt	output.txt
8 8 3	11
3 3	2 1
4 5	3 1
6 6	4 1
2 1	5 1
7 7	6 1
	7 1
	7 2
	7 3
	7 4
	7 5
	7 6
	7 7

分析与解答:

用优先队列式分支限界法搜索一般解空间的算法 pqbb 如下。

```
template<class Type>
void Maze<Type>::pqbb()
{
    NewHeap(H);
    HeapNode<Type> E, N;
    Init(E);
    // 搜索一般解空间树
    while(true){
        if(Answer(E))Save(E);
        else
            for(int i=f(n,E);i<=g(n,E);i++){
                NewNode(N, E, i);
                if(Constrain(N) && Bound(N))AddLiveNode(N, E, i);
                else DelNode(N);
            }
        // 取下一扩展结点
        if(!Getnext(E))break;
    }
    Output();
}
```

其中, 堆结点类型为 HeapNode。

```

template<class Type>
class HeapNode{
public:
    operator int() const{return len;}
    Type row, col, len;
};

```

结点可行性判定函数 Constraint 和上界函数 Bound 等必要的函数通过类 Maze 的私有函数传递。

```

template<class Type>
class Maze{
    friend int main();
private:
    MinHeap<HeapNode<Type> > *H;
    void NewHeap(MinHeap<HeapNode<Type> > *&H);
    void Init(HeapNode<Type> &E);
    bool Answer(HeapNode<Type> E);
    void Save(HeapNode<Type> &E);
    int f(int n, HeapNode<Type> E);
    int g(int n, HeapNode<Type> E);
    void NewNode(HeapNode<Type> &N, HeapNode<Type> E, int i);
    bool Constrain(HeapNode<Type> E);
    bool Bound(HeapNode<Type> E);
    void AddLiveNode(HeapNode<Type> N, HeapNode<Type> E, int i);
    bool Getnext(HeapNode<Type> &E);
    void DelNode(HeapNode<Type> &E);
    void Output();
    void pqbb();
    int n, m;
    bool found;
    Type **grid, PathLen;
    HeapNode<Type> start, finish, offset[4], *path;
};

```

```

template<class Type>
void Maze<Type>::NewHeap(MinHeap<HeapNode<Type> > *&H)
{
    H=new MinHeap<HeapNode<Type> >(HeapSize);
}

```

```

template<class Type>
void Maze<Type>::Init(HeapNode<Type> &E)

```



```

{
    // 设置方格阵列“围墙”
    for(int i=0;i<=m+1;i++)
        grid[0][i]=grid[n+1][i]=1;    // 顶部和底部
    for(i=0;i<=n+1;i++)
        grid[i][0]=grid[i][m+1]=1;    // 左翼和右翼
    // 初始化相对位移
    offset[0].row=0;offset[0].col=1;    // 右
    offset[1].row=1;offset[1].col=0;    // 下
    offset[2].row=0;offset[2].col=-1;    // 左
    offset[3].row=-1;offset[3].col=0;    // 上
    E.row=start.row;
    E.col=start.col;
    E.len=0;
    grid[start.row][start.col]=2;
}

template<class Type>
bool Maze<Type>::Answer(HeapNode<Type> E)
{ return false;}

template<class Type>
void Maze<Type>::Save(HeapNode<Type> &E)
{}

template<class Type>
int Maze<Type>::f(int n, HeapNode<Type> E)
{return 1;}

template<class Type>
int Maze<Type>::g(int n, HeapNode<Type> E)
{return 4;}

template<class Type>
void Maze<Type>::NewNode(HeapNode<Type> &N, HeapNode<Type> E, int i)
{
    N.row=E.row+offset[i-1].row;
    N.col=E.col+offset[i-1].col;
}

template<class Type>
bool Maze<Type>::Constrain(HeapNode<Type> E)
{
    return grid[E.row][E.col]==0;
}

```

```

}

template<class Type>
bool Maze<Type>::Bound(HeapNode<Type> E)
{
    if(!found) found=E.row==finish.row && E.col==finish.col;
    return true;
}

template<class Type>
void Maze<Type>::AddLiveNode(HeapNode<Type> N, HeapNode<Type> E, int i)
{
    grid[N.row][N.col]=grid[E.row][E.col]+1;
    N.len=grid[N.row][N.col];
    if(!found) H->Insert(N);
}

template<class Type>
void Maze<Type>::DelNode(HeapNode<Type> &E)
{}

template<class Type>
bool Maze<Type>::Getnext(HeapNode<Type> &E)
{
    if(found) return false;
    H->DeleteMin(E);
    return true;
}

template<class Type>
void Maze<Type>::Output()
{
    if(!found) {cout<<"No path!"<<endl;return;}
    // 构造最优解
    PathLen= grid[finish.row][finish.col]-2;
    path=new HeapNode<Type> [PathLen];
    // 从目标位置 finish 开始向起始位置回溯
    HeapNode<Type> N,E=finish;
    for(int j=PathLen-1;j>=0;j--){
        path[j]=E;
        // 找前驱位置
        for(int i=0;i<4;i++){
            N.row=E.row+offset[i].row;
            N.col=E.col+offset[i].col;

```

```

        if(grid[X.row][X.col]==j+2) break;
    }
    E=N;    // 向前移动
}
cout<<PathLen<<endl;
cout<<start.row<<" "<<start.col<<endl;
for(j=0;j<PathLen;j++)cout<<path[j].row<<" "<<path[j].col<<endl;
}

```

实现算法的主函数如下。

```

int main()
{
    int n,m,a,b,x;
    Maze<int> X;
    cin>>n>>m>>x;
    X.n=n;X.m=m;X.found=false;
    Make2DArray(X.grid,n+2,m+2);
    for(a=0;a<n+2;a++)
        for(b=0;b<m+2;b++)X.grid[a][b]=0;
    for(x=x;x==1;x--) {cin>>a>>b;X.grid[a][b]=1;}
    cin>>X.start.row>>X.start.col>>X.finish.row>>X.finish.col;
    X.pqbb();
    return 0;
}

```

算法实现题 6-19 骑士征途问题

★问题描述：

在一个 $n \times n$ 个方格的国际象棋棋盘上，马（骑士）从任意指定方格出发，按照横 1 步竖 2 步，或横 2 步竖 1 步的跳马规则，走遍棋盘的每一个格子，且每个格子只走 1 次。这样的跳马步骤称为 1 个成功的骑士征途。例如，当 $n=5$ 时的 1 个成功的骑士征途如图 6-4 所示。

★编程任务：

对于给定的 n 和 $n \times n$ 方格的起始位置 x 和 y 。用分支限界法找出从指定的方格 (x,y) 出发的一条成功的骑士征途。

★数据输入：

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n ($1 \leq n \leq 10$)；第 2 行有 2 个正整数 x 和 y ，表示骑士的起始位置为 (x,y) 。

★结果输出：

将计算出的成功骑士征途输出到文件 output.txt。如果不存在从 (x,y) 出发的成功的骑士征途则输出 “No Solution!”。

	1	2	3	4	5
1	25	14	1	8	19
2	4	9	18	13	2
3	15	24	3	20	7
4	10	5	22	17	12
5	23	16	11	6	21

图 6-4 骑士征途问题

输入文件示例

input.txt

5

1 3

输出文件示例

output.txt

25 14 1 8 19

4 9 18 13 2

15 24 3 20 7

10 5 22 17 12

23 16 11 6 21

分析与解答:

与旅行售货员问题类似。

算法实现题 6-20 推箱子问题

★问题描述:

码头仓库是划分为 $n \times m$ 个格子的矩形阵列。有公共边的格子是相邻格子。当前仓库中有的格子是空闲的,有的格子则已经堆放了沉重的货物。由于堆放的货物很重,单凭仓库管理员的力量是无法移动的。仓库管理员有一项任务:要将一个小箱子推到指定的格子上去。管理员可以在仓库中移动,但不能跨过已经堆放了货物的格子。管理员站在与箱子相对的空闲格子上时,可以做一次推动,把箱子推到另一相邻的空闲格子。推箱时只能向管理员的对面方向推。由于要推动的箱子很重,仓库管理员想尽量减少推箱子的次数。

★编程任务:

对于给定的仓库布局,以及仓库管理员在仓库中的位置和箱子的开始位置和目标位置,设计一个解推箱子问题的分支限界法,计算出仓库管理员将箱子从开始位置推到目标位置所需的最少推动次数。

★数据输入:

由文件 input.txt 提供输入数据。输入文件第 1 行有 2 个正整数 n 和 m ($1 \leq n, m \leq 100$),表示仓库是 $n \times m$ 个格子的矩形阵列。接下来有 n 行,每行有 m 个字符,表示格子的状态。

S——格子上放了不可移动的沉重货物;

w——格子空闲;

M——仓库管理员的初始位置;

P——箱子的初始位置;

K——箱子的目标位置。

★结果输出:

将计算出的最少推动次数输出到文件 output.txt。如果仓库管理员无法将箱子从开始位置推到目标位置则输出 “No Solution!”。

输入文件示例

input.txt

10 12

SSSSSSSSSSSS

SwwwwwwwSSSS

SwSSSSwwSSSS

SwSSSSwwSKSS

输出文件示例

output.txt

7

```

SwSSSSwwSwSS
SwwwwwPwwwww
SSSSSSSwSwSw
SSSSSSMwSwww
SSSSSSSSSSSS
SSSSSSSSSSSS

```

分析与解答：

与布线问题类似，结点元素类型是 Position。

```

class Position {
public:
    operator int() const {return row;}
    int row,col,dir;
};

```

它的私有成员 row 和 col 分别表示方格所在的行和列；dir 表示推的方向。
算法用到如下全局变量。

```

int op[4]={1,0,3,2};
int m,n,totm,markr;
int **grid,**reach,**mark,**low,**totr,**comp;
long ***ans;
Position start,finish,man;
Position offset[4];

```

Init 实现数据输入及预处理。

```

void Init()
{
    char c;
    cin>>n>>m;
    Make2DArray(grid,n+2,m+2);
    Make2DArray(reach,n+1,m+1);
    Make2DArray(mark,n+1,m+1);
    Make2DArray(low,n+1,m+1);
    Make2DArray(totr,n+1,m+1);
    Make3DArray(ans,n+1,m+1,4);
    Make3DArray(comp,n+1,m+1,10);
    for(int i=0;i<n+2;i++)
        for(int j=0;j<m+2;j++)grid[i][j]=0;
    for(i=1;i<=n;i++)
        for(int j=1;j<=m;j++){
            cin>>c;

```

```

        while(!isupper(c) && !islower(c)) cin>>c;
        if(c=='M') man.row=i,man.col=j;
        if(c=='P') start.row=i,start.col=j;
        if(c=='K') finish.row=i,finish.col=j;
        if(c=='S') grid[i][j]=1;
    }
    // 设置方格阵列"围墙"
    for(i=0;i<=m+1;i++)
        grid[0][i]=grid[n+1][i]=1;    // 顶部和底部
    for(i=0;i<=n+1;i++)
        grid[i][0]=grid[i][m+1]=1;    // 左翼和右翼
    // 初始化相对位移
    offset[0].row=0;offset[0].col=-1;// 左
    offset[1].row=0;offset[1].col=1;  // 右
    offset[2].row=-1;offset[2].col=0;// 上
    offset[3].row=1;offset[3].col=0;  // 下
    Prepro();
    for(i=0;i<=n;i++)
        for(int j=0;j<=m;j++){
            reach[i][j]=0;
            for(int k=0;k<4;k++)ans[i][j][k]=LONG_MAX;
        }
    dfs(man.row,man.col);
}

```

其中, Prepro 对方格连通性进行预处理计算。

```

void Prepro()
{
    totm=0;markr=0;
    for(int i=1;i<=n;i++)
        for(int j=1;j<=m;j++){
            mark[i][j]=-1;
            low[i][j]=INT_MAX;
            totr[i][j]=0;
            reach[i][j]=-1;
        }
    for(i=1;i<=n;i++)
        for(int j=1;j<=m;j++)
            if(grid[i][j]==0 && mark[i][j]==-1)fill(i,j);
}

void put(int x,int y,int a,int b)
{

```

```

    if(x==a && y==b) return;
    if(reach[x][y]==2) return;
    comp[x][y][totr[x][y]]=markr;
    totr[x][y]++;
    reach[x][y]=2;
    for(int i=0;i<4;i++){
        int x1=x+offset[i].row,y1=y+offset[i].col;
        if(grid[x1][y1]==0) put(x1,y1,a,b);
    }
}

void fill(int x,int y)
{
    for(int i=0;i<4;i++){
        int x1=x+offset[i].row,y1=y+offset[i].col;
        if(grid[x1][y1]==0){
            if(mark[x1][y1]==-1){
                mark[x1][y1]=totm;totm++;
                fill(x1,y1);
                low[x][y]=min(low[x][y],low[x1][y1]);
                if(low[x1][y1]>=mark[x][y]){
                    markr++;
                    put(x1,y1,x,y);
                    comp[x][y][totr[x][y]]=markr;
                    totr[x][y]++;
                    reach[x][y]=1;
                }
            }
            else low[x][y]=min(low[x][y],mark[x1][y1]);
        }
    }
}

```

预处理后，由 connect 计算方格连通性。

```

int connect(int x1,int y1,int x2,int y2)
{
    for(int i=0;i<totr[x1][y1];i++)
        for(int j=0;j<totr[x2][y2];j++)
            if(comp[x1][y1][i]==comp[x2][y2][j]) return 1;
    return 0;
}

```

dfs 计算初始可达性。

```

void dfs(int x, int y)
{
    if(reach[x][y]==1) return;
    reach[x][y]=1;
    for(int i=0;i<4;i++){
        int x1=x+offset[i].row, y1=y+offset[i].col;
        if(grid[x1][y1]==0 && (x1!=start.row || y1!=start.col)) dfs(x1, y1);
    }
}

```

解推箱子问题的队列式分支限界法 FIFOBB 如下。

```

void FIFOBB()
{
    Queue<Position> Q;
    Position here, nbr;
    for(int i=0;i<4;i++){
        nbr.row=start.row; nbr.col=start.col; nbr.dir=i;
        if(ok(start.row+offset[i].row, start.col+offset[i].col)) {
            ans[start.row][start.col][i]=0;
            Q.Add(nbr);
        }
    }
    while(!Q.IsEmpty()){
        Q.Delete(here);
        int d=here.dir, x=here.row+offset[op[d]].row, y=here.col+offset[op[d]].col;
        if(grid[x][y]==0 && ans[x][y][d]>ans[here.row][here.col][d]+1) {
            ans[x][y][d]=ans[here.row][here.col][d]+1;
            nbr.row=x; nbr.col=y; nbr.dir=d;
            Q.Add(nbr);
            for(i=0;i<4;i++){
                if(i!=d) {
                    int x1=x+offset[i].row, y1=y+offset[i].col;
                    if(grid[x1][y1]==0 && connect(x1, y1, here.row, here.col) &&
                        ans[x][y][i]>ans[x][y][d]) {
                        ans[x][y][i]=ans[x][y][d];
                        nbr.row=x; nbr.col=y; nbr.dir=i;
                        Q.Add(nbr);
                    }
                }
            }
        }
    }
}

```



```

bool ok(int a,int b)
{
    return grid[a][b]==0 && reach[a][b]==1;
}

```

算法的主函数如下。

```

int main()
{
    Init();
    FIFOBB();
    Out();
    return 0;
}

```

Out 输出计算结果。

```

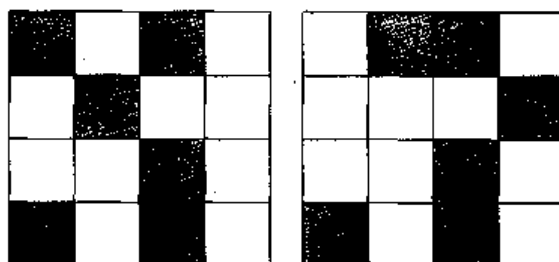
void Out()
{
    for(long min=ans[finish.row][finish.col][0],i=1;i<4;i++)
        if(ans[finish.row][finish.col][i]<min) min=ans[finish.row][finish.col][i];
    if(min==LONG_MAX) cout<<"No Solution!"<<endl;
    else cout<<min<<endl;
}

```

算法实现题 6-21 图形变换问题

★问题描述：

给定 2 个 4×4 方格阵列组成的图形 A 和 B，每个方格的颜色为黑色或白色，如图 6-5 所示。方格阵列中有公共边的方格称为相邻方格。图形变换问题的每一步变换可以交换相邻方格的颜色。试设计一个队列式分支限界算法，计算最少需要多少步变换，才能将图形 A 变换为图形 B。



图形 A

图形 B

图 6-5 图形变换问题

★编程任务：

对于给定的 2 个方格阵列，编程计算将图形 A 变换为图形 B 的最少变换次数。

★数据输入：

由文件 input.txt 给出输入数据。前 4 行是图形 A 的方格阵列，后 4 行是图形 B 的方格

阵列。0 表示白色, 1 表示黑色。

★结果输出:

将计算出的最少变换次数和相应的变换序列输出到文件 output.txt。第 1 行是最少变换次数。从第 2 行开始, 每行用 4 个数表示一次变换。例如, 1112 表示交换方格 (1,1) 和 (1,2) 的颜色。问题无解时输出 “No Solution!”。

输入文件示例	输出文件示例
input.txt	output.txt
1010	3
0100	1112
0010	2223
1010	2324
0110	
0001	
0010	
1010	

分析与解答:

由于图形由 16 个方格组成, 可以用 16 位的整型数表示图形方格的颜色状态。
解图形变换问题的队列式分支限界法 fifobb 如下。

```
const int MaxSize=1<<16;
int sour,dest,a[MaxSize],b[MaxSize];
bool fifobb() {
    Queue<int> Q;
    for(int i=1,E=0;i<MaxSize;i++)a[i]=-1;
    a[sour]=0;Q.Add(sour);
    while(!Q.IsEmpty()){
        Q.Delete(E);
        for(int j=0,mode=3;j<16;j++,mode<<=1)
            if(j%4!=3 &&(((E & mode)>>j)==1||((E & mode)>>j)==2)){
                int N=E^mode;
                if(a[N]==-1) a[N]=a[E]+1,b[N]=j,Q.Add(N);
            }
        for(j=0,mode=17;j<12;j++,mode<<=1)
            if(((E & mode)>>j)==1||((E & mode)>>j)==16){
                int N=E^mode;
                if(a[N]==-1) a[N]=a[E]+1,b[N]=-j-1,Q.Add(N);
            }
        if(a[dest]!=-1) return true;
    }
    return false;
}
```

算法的主函数如下。

```
int main()
{
    init();
    if(fifobb()) {cout<<a[dest]<<endl;output(dest);}
    else cout<<"No Solution!"<<endl;
    return 0;
}

void init()
{
    int i;char c;
    for(sour=i=0;i<16;i++){cin>>c;sour|=(int)(c-'0')<<i;}
    for(dest=i=0;i<16;i++){cin>>c;dest|=(int)(c-'0')<<i;}
}

void output(int E){
    if(E==sour) return;
    int last=E;
    if(b[E]>=0) last^=3<<b[E];
    else last^=17<<(-b[E]-1);
    output(last);
    if(b[E]>=0)
        cout<<b[E]/4+1<<b[E]%4+1<<b[E]/4+1<<b[E]%4+2<<endl;
    else
        cout<<(-b[E]-1)/4+1<<(-b[E]-1)%4+1<<(-b[E]-1)/4+2<<(-b[E]-1)%4+
        1<<endl;
}
```

算法实现题 6-22 行列变换问题

★问题描述:

给定 2 个 $m \times n$ 方格阵列组成的图形 A 和 B, 每个方格的颜色为黑色或白色, 如图 6-6 所示。行列变换问题的每一步变换可以交换任意 2 行或 2 列方格的颜色, 或者将某行或某列颠倒。上述每次变换算做一步。试设计一个队列式分支限界算法, 计算最少需要多少步, 才能将图形 A 变换为图形 B。



图形 A 图形 B
图 6-6 行列变换问题

★编程任务:

对于给定的 2 个方格阵列, 编程计算将图形 A 变换为图形 B 的最少变换次数。

★数据输入:

由文件 input.txt 给出输入数据。文件的第 1 行有 2 个正整数 m 和 n 。以下的 m 行是方格阵列的初始状态 A, 每行有 n 个数字表示该行方格的状态, 0 表示白色, 1 表示黑色。接着的 m 行是方格阵列的目标状态 B。

★结果输出:

将计算出的最少变换次数和相应的变换序列输出到文件 output.txt。第 1 行是最少变换次数。从第 2 行开始, 依次输出变换的图形序列。问题无解时输出 “No Solution!”。

输入文件示例

输出文件示例

input.txt

output.txt

4 4

2

1010

0100

1010

0010

0100

1010

0010

1010

1010

0000

0110

1010

0101

0000

0110

1010

1010

0000

0110

0101

分析与解答:

与图形变换问题类似, 用 32 位的整型数表示图形方格的颜色状态。

解行列变换问题的队列式分支限界法 fifobb 如下。

```
const unsigned MaxSize=1<<25;
int m,n;
unsigned sour,dest,*row,*col,hash[MaxSize],ans[MaxSize];

bool fifobb()
{
    Queue<unsigned> Q;
    for(unsigned i=0,E=0,N=0;i<MaxSize;i++)hash[i]=-1;
    hash[sour]=0;Q.Add(sour);
    while(!Q.IsEmpty()){
```

```

Q.Delete(E);
for(int i=0;i<m-1;i++){
    for(int j=i+1;j<m;j++){
        N=rowswap(E,i,j);
        if(hash[N]==-1) hash[N]=hash[E]+1,ans[N]=E,Q.Add(N);
    }
for(i=0;i<n-1;i++){
    for(int j=i+1;j<n;j++){
        N=colswap(E,i,j);
        if(hash[N]==-1) hash[N]=hash[E]+1,ans[N]=E,Q.Add(N);
    }
for(i=0;i<m;i++){
    N=revrow(E,i);
    if(hash[N]==-1) hash[N]=hash[E]+1,ans[N]=E,Q.Add(N);
}
for(i=0;i<n;i++){
    N=revcol(E,i);
    if(hash[N]==-1) hash[N]=hash[E]+1,ans[N]=E,Q.Add(N);
}
if(hash[dest]!=-1) return true;
}
return false;
}

```

其中, rowswap 实现行交换; colswap 实现列交换; revrow 实现行翻转; revcol 实现列翻转。

```

unsigned rowswap(unsigned x,int i,int j)
{
    rowx(x);
    Swap(row[i],row[j]);
    rowy(x);
    return x;
}

```

```

unsigned colswap(unsigned x,int i,int j)
{
    colx(x);
    Swap(col[i],col[j]);
    coly(x);
    return x;
}

```

```

unsigned revrow(unsigned x,int i)

```

```

    {
        rowx(x);
        reve(row[i], n);
        rowy(x);
        return x;
    }

unsigned revcol(unsigned x, int i)
{
    colx(x);
    reve(col[i], m);
    coly(x);
    return x;
}

void rowx(unsigned x)
{
    for(int i=0; i<m; i++){
        unsigned y=1; row[i]=0;
        for(int j=0; j<n; j++){
            if(x & 1) row[i] |= y;
            y<<=1; x>>=1;
        }
    }
}

void colx(unsigned x)
{
    for(int j=0; j<n; j++) col[j]=0;
    unsigned y=1;
    for(int i=0; i<m; i++){
        for(int j=0; j<n; j++){
            if(x & 1) col[j] |= y;
            x>>=1;
        }
        y<<=1;
    }
}

void rowy(unsigned &x)
{
    x=0;
    for(int i=0, z=1; i<m; i++){
        unsigned y=row[i];

```

```

        for(int j=0;j<n;j++){
            if(y & 1)x|=z;
            z<<=1;y>>=1;
        }
    }

void coly(unsigned &x)
{
    x=0;
    for(int i=0,z=1;i<m;i++){
        for(int j=0;j<n;j++){
            if(col[j] & 1)x|=z;
            col[j]>>=1;z<<=1;
        }
    }
}

void reve(unsigned &x, int k)
{
    unsigned y=0,z=1<<(k-1);
    for(int j=0;j<k;j++){
        if(x & 1)y|=z;
        z>>=1;x>>=1;
    }
    x=y;
}

```

算法的主函数如下。

```

int main()
{
    if(!init()){cout<<"No Solution!"<<endl;return 0;}
    if(fifobb())cout<<hash[dest]<<endl,output(dest);
    return 0;
}

```

init 读入初始数据; output 输出最优解。

```

bool init()
{
    int i,a=0,b=0;char c;
    cin>>m>>n;
    row=new unsigned[m];
}

```

```

        col=new unsigned[n];
        for(sour=i=0;i<n*n;i++){cin>>c;sour|=(int)(c-'0')<<i;a+=(int)(c-'0');}
        for(dest=i=0;i<n*n;i++){cin>>c;dest|=(int)(c-'0')<<i;b+=(int)(c-'0');}
        return a==b;
    }

    void output(unsigned N)
    {
        if(N==sour){cout<<endl;outb(N);return;}
        output(ans[N]);
        cout<<endl;outb(N);
    }

    void outb(unsigned x)
    {
        for(int i=0;i<n;i++){
            for(int j=0;j<n;j++){
                if(x&1)cout<<1;
                else cout<<0;
                x/=2;
            }
            cout<<endl;
        }
    }
}

```

算法实现题 6-23 重排 n^2 宫问题

★问题描述:

重排九宫是一个古老的单人智力游戏。据说重排九宫起源于我国古时由三国演义故事“关羽义释曹操”而设计的智力玩具“华容道”，后来流传到欧洲，将人物变成数字。原始的重排九宫问题是这样的：将数字 1~8 按照任意次序排在 3×3 的方格阵列中，留下一个空格，如图 6-7 所示。与空格相邻的数字，允许从上、下、左、右方向移动到空格中。游戏的最终目标是通过合法移动，将数字 1~8 按行排好序。

在一般情况下，重排 n^2 宫问题是将数字 1~ n^2-1 按照任意次序排在 $n \times n$ 的方格阵列中，留下一个空格。允许与空格相邻的数字从上、下、左、右 4 个方向移动到空格中。游戏的最终目标是通过合法移动，将初始状态变换到目标状态。

1	2	3
4		6
7	5	8

1	2	3
4	5	6
7	8	

图 6-7 重排 n^2 宫问题

★编程任务:

对于给定的 $n \times n$ 方格阵列中数字 1~ n^2-1 初始排列和目标状态，用优先队列式分支限

界法编程计算将初始排列通过合法移动变换为目标状态最少移动次数。

★数据输入:

由文件 input.txt 给出输入数据。文件的第 1 行有 1 个正整数 n 。以下的 n 行是 $n \times n$ 方格阵列的中数字 $1 \sim n^2 - 1$ 的初始排列, 每行有 n 个数字表示该行方格中的数字, 0 表示空格。接着的 n 行是方格阵列中数字 $1 \sim n^2 - 1$ 的目标状态。

★结果输出:

将计算出的最少移动次数和相应的移动序列输出到文件 output.txt。第 1 行是最少移动次数。从第 2 行开始, 依次输出移动序列。用大写英文字母 D,U,L,R 分别表示向下、向上、向左、向右移动。问题无解时输出 “No Solution!”

输入文件示例

输出文件示例

input.txt

output.txt

3

2

1 2 3

DR

4 0 6

7 5 8

1 2 3

4 5 6

7 8 0

分析与解答:

用类 Board 表示 $n \times n$ 方格阵列。

```
int rowsz, boardsz, *sour, *dest, *pos;
```

```
class Board
```

```
{
```

```
    friend class Prio;
```

```
public:
```

```
    vector<int> boardm;
```

```
    void getboard(int *m);
```

```
    bool move(int dir);
```

```
    int heur() {return dist;}
```

```
    int getdist(const int &v, const int &loc);
```

```
    bool reached();
```

```
    void out();
```

```
private:
```

```
    int y, x;          // 空格位置
```

```
    vector<char> path;
```

```
    int dist;          // manhattan 距离
```

```
};
```

```
void Board::getboard(int *m)
```

```
{
```

```

    for(int i=0;i<boardsz;i++){
        boardm.push_back(m[i]);
        if(m[i]==0){y=i/rowsz;x=i%rowsz;}
    }
    for(i=0,dist=0;i<boardsz;i++){
        if(boardm[i]!=0)dist+=getdist(boardm[i],i);
    }

// 按4个方向移动
bool Board::move(int dir)
{
    const int step[4][2]={ {0,-1},{0,1},{-1,0},{1,0}};
    int nx=x+step[dir][0],ny=y+step[dir][1];
    if(nx> -1 && nx<rowsz && ny> -1 && ny<rowsz){
        dist=dist+1-getdist(boardm[ny*rowsz+nx],ny*rowsz+nx)+
            getdist(boardm[ny*rowsz+nx],y*rowsz+x);
        swap(boardm[y*rowsz+x],boardm[ny*rowsz+nx]);
        y=ny;x=nx;
        path.push_back(dir);
        return true;
    }
    else return false;
}

// 计算 manhattan 距离
int Board::getdist(const int &v,const int &loc)
{
    int dis=abs((pos[v]%rowsz)-(loc%rowsz));
    dis+=abs((pos[v]/rowsz)-(loc/rowsz));
    return dis;
}

// 到达目标状态
bool Board::reached()
{
    for(int i=0;i<boardsz;i++){
        if(boardm[i]!=dest[i])return false;
    }
    return true;
}

void Board::out()
{
    const char dir[]="DULR";
    cout<<path.size()<<endl;
}

```

```

for(int i=0;i<path.size();i++){
    cout<<dir[path[i]];
    if(i%20==19) cout<<endl;
}
if(path.size()%20!=0)cout<<endl;
}

```

结点的优先级是当前状态到目标状态的 manhattan 距离。

```

class Prio
{
public:
    int operator() (const Board &x, const Board &y)
    {
        if(x.dist==y.dist) return x.path.size()<y.path.size();
        else return x.dist>y.dist;
    }
};

```

解重排 n^2 宫问题的优先队列式分支限界法 fifobb 如下。

```

// 优先队列式分支限界法(A* 算法)
void pqbb()
{
    Board E,N;
    priority_queue<Board,vector<Board>,Prio> H;
    map<vector<int>,int> hash;
    map<vector<int>,int>::iterator iter;
    E.getboard(sour);
    H.push(E);
    hash.insert(pair<vector<int>,int>(E.boardm,E.heur()));
    while(true){
        E=H.top();H.pop();
        if(E.reached()){E.out();return;}
        for(int i=0;i<4;i++){
            N=E;
            if(N.move(i)){
                iter=hash.find(N.boardm);
                if(iter==hash.end()){
                    H.push(N);
                    hash.insert(pair<vector<int>,int>(N.boardm,N.heur()));
                }
                else if(iter->second>N.heur()){
                    hash.erase(iter);
                }
            }
        }
    }
}

```

```

        H.push(N);
        hash.insert(pair<vector<int>,int>(N.boardm,N.heur()));
    }
}
}
}
}

```

算法的主函数如下。

```

int main()
{
    init();
    if(odd()) pqbb();
    else cout<<"No Solution!"<<endl;
    return 0;
}

```

init 读入初始数据。

```

void init()
{
    cin>>rowsz;boardsz=rowsz*rowsz;
    sour=new int[boardsz];
    dest=new int[boardsz];
    pos=new int[boardsz];
    for(int i=0;i<boardsz;i++)cin>>sour[i];
    for(i=0;i<boardsz;i++){cin>>dest[i];pos[dest[i]]=i;}
}

```

$n \times n$ 方格阵列中的数字按照从上到下、从左到右的顺序排列,对应于数字 $0 \sim n^2 - 1$ 的一个排列。设 $m = n^2 - 1$, 初始排列为 (s_0, s_1, \dots, s_m) , 目标排列为 (t_0, t_1, \dots, t_m) 。初始排列中空格位置为 (x_0, y_0) ; 目标排列中空格位置为 (x_1, y_1) 。 (x_0, y_0) 与 (x_1, y_1) 的 manhattan 距离为 $\text{dist}_0 = |x_1 - x_0| + |y_1 - y_0|$ 。初始排列 (s_0, s_1, \dots, s_m) 到目标排列 (t_0, t_1, \dots, t_m) 的变换对应于置换 $\begin{pmatrix} s_0, s_1, \dots, s_m \\ t_0, t_1, \dots, t_m \end{pmatrix}$ 。该置换的奇偶性与 dist_0 的奇偶性相同时问题有解, 否则问题无解。由此可见, 对于给定的初始排列, 只有 $(n^2)! / 2$ 个排列是从初始排列可达到的。

下面的算法 odd 计算初始状态到目标状态的置换奇偶性, 由此判定问题的可解性。

```

bool odd()
{
    int *c, count=0, count1=0, i1, j1, i2, j2;
    c=new int[boardsz];

```

```

for(int k=0;k<boardsz;k++){
    c[dest[k]]=sour[k];
    if(sour[k]==0){i1=k/rowsz+1;j1=k%rowsz+1;}
    if(dest[k]==0){i2=k/rowsz+1;j2=k%rowsz+1;}
}
int posi=((i1+i2)%2-(j1+j2)%2)%2;
for(int j=0;j<boardsz;j++){
    int k=c[j],k1=j;
    count1=0;
    while(k>=0){k=c[k1];c[k1]=-1;k1=k;count1++;}
    if(count1>0) count+=count1-2;
}
if(count%2==posi)return true;
else return false;
}

```

上述解重排 n^2 宫问题的优先队列式分支限界法 `fifobb` 找到的解使 manhattan 距离最小，但问题要求的是移动次数最少。如何保证这一点？事实上，`fifobb` 用到了当前状态到目标状态的 manhattan 距离为结点的优先级，这是一种启发式的搜索策略。由于 manhattan 距离是单调的，所以算法 `fifobb` 是一个 A^* 算法，从而保证了找到的解是移动次数最少的解。

逐步深化的 A^* 搜索算法 (IDA^*) 与上述算法相比，只用很少的内存，时间效率也相当高，具体算法描述如下。

用类 `Board` 表示 $n \times n$ 方格阵列。

```

int rowsz,boardsz,maxdep,*sour,*dest,*pos;
const int op[4]={1,0,3,2};

class Board
{
    friend bool solve(int dep,Board E);
public:
    vector<int> boardm;
    void getboard(int *m);
    bool move(int dir);
    int heur(){return dist;}
    int getdist(const int &v,const int &loc);
    bool reached();
    int dir(){return (path.back());}
    void out();
private:
    int y,x;          // 空格位置
    vector<char> path;
    int dist;          // manhattan 距离

```

```

);

void Board::getboard(int *m)
{
    for(int i=0;i<boardsz;i++){
        boardm.push_back(m[i]);
        if(m[i]==0){y=i/rowsz;x=i%rowsz;}
    }
    for(i=0,dist=0;i<boardsz;i++){
        if(boardm[i]!=0)dist+=getdist(boardm[i],i);
    }

    // 按 3 个方向移动
    bool Board::move(int dir)
    {
        const int step[4][2]={{0,-1},{0,1},{-1,0},{1,0}};
        int nx=x+step[dir][0],ny=y+step[dir][1];
        if((path.empty() || op[dir]!=(path.back())) &&
            nx>=1 && nx<rowsz && ny>=1 && ny<rowsz){
            dist=dist+1+getdist(boardm[ny*rowsz+nx],ny*rowsz+nx)
                +getdist(boardm[ny*rowsz+nx],y*rowsz+x);
            swap(boardm[y*rowsz+x],boardm[ny*rowsz+nx]);
            y=ny;x=nx;
            path.push_back(dir);
            return true;
        }
        else return false;
    }

    // 计算 manhattan 距离
    int Board::getdist(const int &v,const int &loc)
    {
        int dis=abs((pos[v]%rowsz)-(loc%rowsz));
        dis+=abs((pos[v]/rowsz)-(loc/rowsz));
        return dis;
    }

    // 到达目标状态
    bool Board::reached()
    {
        for(int i=0;i<boardsz;i++){
            if(boardm[i]!=dest[i])return false;
        }
        return true;
    }

```

```

void Board::out()
{
    const char dir[]="DULR";
    cout<<path.size()<<endl;
    for(int i=0;i<path.size();i++){
        cout<<dir[path[i]];
        if(i%20==19) cout<<endl;
    }
    if(path.size()%20!=0)cout<<endl;
}

```

逐步深化的 A * 搜索算法 idastar 描述如下。

```

bool solve(int dep, Board E)
{
    if(dep+E.dist<=maxdep){
        if(E.reached()){E.out();return true;}
        for(int i=0;i<4;i++){
            Board N=E;
            if(N.move(i))
                if(solve(dep+1,N))return true;
        }
    }
    return false;
}

// IDA*算法
void idastar()
{
    Board E;
    E.getboard(sour);
    maxdep=E.heur();
    if(maxdep==0){cout<<"0"<<endl;return;}
    while(!solve(0,E))maxdep+=2;
}

```

算法的主函数如下。

```

int main()
{
    init();
    if(odd())idastar();
    else cout<<"No Solution!"<<endl;
    return 0;
}

```

算法实现题 6-24 最长距离问题

★问题描述:

重排九宫是一个古老的单人智力游戏。据说重排九宫起源于我国古时由三国演义故事“关羽义释曹操”而设计的智力玩具“华容道”，后来流传到欧洲，将人物变成数字。原始的重排九宫问题是这样的：将数字 1~8 按照任意次序排在 3×3 的方格阵列中，留下一个空格，如图 6-8 所示。与空格相邻的数字，允许从上、下、左、右方向移动到空格中。游戏的最终目标是通过合法移动，将数字 1~8 按行排好序。最长距离问题考察的是，从数字 1~8 在 3×3 的方格阵列的初始排列 A 出发，找出与其相应的最长距离目标状态 B。换句话说，从 A 到 B 的最优移动序列的长度最长。

★编程任务:

对于给定的 3×3 方格阵列中数字 1~8 初始排列，用队列式分支限界法编程计算与初始排列相应的最长距离目标状态。

★数据输入:

由文件 input.txt 给出输入数据。文件有 3 行，每行有 3 个数字表示该行方格中的数字，0 表示空格。

★结果输出:

将计算出的最长距离目标状态输出到文件 output.txt。第 1 行有 2 个正整数 x 和 y ， x 是最长距离的值， y 是最长距离目标状态个数。从第 2 行开始，依次输出最长距离目标状态和到达该最长距离目标状态的最优移动序列。用大写英文字母 D,U,L,R 分别表示向下、向上、向左、向右移动。

输入文件示例

input.txt

2 6 4

1 3 7

0 5 8

输出文件示例

output.txt

31 2

8 7 1

0 3 5

4 6 2

UURDDLURRDDLLURDLLUURDLURRDDLLU

8 1 5

7 3 6

4 0 2

UURDDRULLURRDLLDRRULULDDRULDDR

分析与解答:

算法中用到的一些变量如下。

```
#define Bitsint (8*sizeof(int))
```

```
#define Rowsz 3
```

```
#define Permsz (Rowsz*Rowsz)
```



```

struct Node
{
    int code, dep, dir, parent;
};

const char op[4]={'U','D','L','R'};

Node *buff;
int first, last, fact, masksz, *bitset, sour[Permsz], perm[Permsz];

```

由于总共有 $9!/2=181440$ 个可达状态。可用数组 buff 记录所有状态。每个状态对应于数字 0~8 的一个排列，对每个排列编序。

```

int ptoi(int const perm[])
{
    int pcpy[Permsz], invp[Permsz];
    for(int i=0, n=0; i<Permsz; ++i) {
        pcpy[i]=perm[i];
        invp[perm[i]]=i;
    }
    for(i=Permsz-1; i>0; --i) {
        int p=invp[i];
        pcpy[p]=pcpy[i]; pcpy[i]=i;
        invp[pcpy[p]]=p; invp[i]=i;
        n*=i+1; n+=i-p;
    }
    return n;
}

```

反之，给定 0~362879 中的一个数，可惟一确定数字 0~8 的一个排列。

```

void itop(int perm[], int n)
{
    perm[0]=0;
    for(int i=1; i<Permsz; ++i) {
        int p=i-n%(i+1);
        perm[i]=perm[p]; perm[p]=i; n/=i+1;
    }
}

```

用下面的队列式分支限界法 fifobb 解此问题。

```

void fifobb()

```

```

{
    while(first<fact/2){
        int code=buff[first].code;
        itop(perm,code);
        for(int i=0;perm[i]!=Permsz-1&&i<Permsz;++i);
        int brow=i/Rowsz,bcol=i%Rowsz;
        for(int d=0;d<4;d++) trymove(brow,bcol,i,d);
        first++;
    }
}

```

其中, trymove 按照可移动方向扩展结点。

```

void trymove(int brow,int bcol,int i,int d)
{
    switch(d){
        case 0:
            if(brow>0){
                swap(perm[i],perm[i-Rowsz]);
                addperm(d);
                swap(perm[i],perm[i-Rowsz]);
            }
            break;
        case 1:
            if(brow<Rowsz-1){
                swap(perm[i],perm[i+Rowsz]);
                addperm(d);
                swap(perm[i],perm[i+Rowsz]);
            }
            break;
        case 2:
            if(bcol>0){
                swap(perm[i],perm[i-1]);
                addperm(d);
                swap(perm[i],perm[i-1]);
            }
            break;
        case 3:
            if(bcol<Rowsz-1){
                swap(perm[i],perm[i+1]);
                addperm(d);
                swap(perm[i],perm[i+1]);
            }
    }
}

```

```
}
```

Addperm 将新结点存储到数组 buff 中。

```
void addperm(int dir)
{
    int code=ptoi(perm);
    int m=code/Bitsint;
    int n=code%Bitsint;
    if((bitset[m]>>n)&1) return;
    bitset[m]|=1<<n;
    buff[last].parent=first;
    buff[last].dir=dir;
    buff[last].dep=buff[first].dep+1;
    buff[last++].code=code;
}
```

算法的主函数如下。

```
int main()
{
    init();
    fifobb();
    output();
    return 0;
}
```

其中, init 读入初始数据并进行初始化计算。

```
void init()
{
    fact=1;
    for(int i=0,n=0;i<Permsz;++i){
        cin>>n;
        if(n==0)sour[i]=Permsz-1;
        else sour[i]=n-1;
        fact*=i+1;
        perm[i]=sour[i];
    }
    int icode=ptoi(sour);
    masksz=(fact+Bitsint-1)/Bitsint;
    bitset=new int[masksz];
    for(i=0;i<masksz;++i) bitset[i]=0;
    bitset[icode/Bitsint]|=1<<(icode%Bitsint);
}
```

```

    first=0;last=0;
    buff=new Node[fact/2];
    buff[last].parent=-1;
    buff[last].dep=0;
    buff[last++].code=icode;
}

```

output 输出计算结果。

```

void output()
{
    for(int i=0,j=0,best=0;i<fact/2;++i)
        if(buff[i].dep>=best){best=buff[i].dep;j=i;}
    for(i=0,j=0;i<fact/2;++i)
        if(buff[i].dep==best)j++;
    cout<<best<<" "<<j<<endl;
    for(i=0,j=0;i<fact/2;++i)
        if(buff[i].dep==best){
            itop(perm,buff[i].code);
            outperm();outmove(i);cout<<endl;
        }
}

void outmove(int first)
{
    if(buff[first].dep==0)return;
    outmove(buff[first].parent);
    cout<<op[buff[first].dir];
}

void outperm()
{
    for(int i=0;i<Permsz;i++){
        cout<<(perm[i]+1)%Permsz<<" ";
        if((i+1)%Rowsz==0)cout<<endl;
    }
}

```

第7章 概率算法

习题 7-1 模拟正态分布随机变量

在实际应用中,常需模拟服从正态分布的随机变量,其密度函数为

$$\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{(x-a)^2}{2\sigma^2}}$$

式中, a 为均值, σ 为标准差。

如果 s 和 t 是 $(-1,1)$ 中均匀分布的随机变量,且 $s^2+t^2<1$,令

$$p=s^2+t^2$$

$$q=\sqrt{(-2\ln p)/p}$$

$$u=sq$$

$$v=tq$$

则 u 和 v 是服从标准正态分布($a=0, \sigma=1$)的两个相互独立的随机变量。

(1) 利用上述事实,设计一个模拟标准正态分布随机变量的算法。

(2) 将上述算法扩展到一般的正态分布。

分析与解答:

(1) 模拟标准正态分布随机变量的算法如下。

```
double RandomNumber::Norm()
{
    double s, t, p, q;
    while(true) {
        s=2*fRandom()-1;
        t=2*fRandom()-1;
        p=s*s+t*t;
        if(p<1)break;
    }
    q=sqrt((-2*log(p))/p);
    return s*q;
}
```

(2) 扩展到一般的正态分布的算法如下。

```
double RandomNumber::Norm(double a, double b)
{
    double x=Norm();
    return a+b*x;
}
```

习题 7-2 随机抽样算法

设有一个文件含有 n 个记录。

(1) 试设计一个算法随机抽取该文件中 m 个记录。

(2) 如果事先不知道文件中记录的个数, 应如何随机抽取其中的 m 个记录。

分析与解答:

(1) 以概率 m/n 抽取记录。该方法的标准差是 $\sqrt{m(1-m/n)}$, 有时可能不满足要求。假设在前 t 次考察的记录中, 已抽取了 k 个记录, 接下来第 $t+1$ 次考察取得第 $k+1$ 个记录的概率为

$$\frac{\binom{n-t-1}{n-k-1}}{\binom{n-t}{n-k}} = \frac{n-k}{n-t}$$

按此概率抽取记录是无偏的。

抽样算法如下。

```
void samp(int n, int m, int s[])
{
    RandomNumber rnd;
    int x=0, y=0, k;
    while(y<m) {
        double u=rnd.fRandom();
        k=u*n;
        if(u*(n-x)<(n-y)) {s[k]++;y++;}
        x++;
    }
}
```

(2) 如果事先不知道文件中记录的个数, 通常可以先做一次扫描, 确定记录的个数后再抽样。另一个较好的方法是在扫描时预先随机抽取 $p>m$ 个记录, 然后对抽取出的 p 个记录做 2 次抽样, 从中随机抽取 m 个记录。

习题 7-3 随机产生 m 个整数

试设计一个算法随机地产生范围在 $1\sim n$ 中的 m 个随机整数, 且要求这 m 个随机整数互不相同。

分析与解答:

与习题 7-2 类似, 解此问题的 Floyd 算法如下。

```
void Floyd(int n, int m, int s[])
{
    RandomNumber rnd;
    int y=0;
    while(y<m) {
        for(int j=n-m+1; j<=n; j++) {
```

```

double u=rnd. fRandom();
int k=1+j*u;
if (s[k]==0) {s[k]++;y++;}
else if (s[j]==0) {s[j]++;y++;}
}
}
}

```

习题 7-4 集合大小的概率算法

设 X 是含有 n 个元素的集合, 从 X 中均匀地选取元素。设第 k 次选取时首次出现重复。

(1) 试证明当 n 充分大时, k 的期望值为 $\beta\sqrt{n}$, 其中, $\beta=\sqrt{\pi/2}=1.253$ 。

(2) 由此设计一个计算给定集合 X 中元素个数的概率算法。

分析与解答:

(1) 从含有 n 个元素的集合 X 中均匀地选取元素, 第 k 次选取时首次出现重复的概率为

$$\frac{\binom{n}{k-1}(k-1)!(k-1)}{n^k}$$

k 的期望值为

$$E(k) = \sum_{k=1}^n \frac{\binom{n}{k-1}(k-1)!(k-1)k}{n^k}$$

用 Stirling 公式

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + \theta\left(\frac{1}{n^2}\right)\right)$$

代入计算可得

$$E(k) = \sqrt{\frac{\pi n}{2} - \frac{1}{3} + \theta\left(\frac{1}{\sqrt{n}}\right)}$$

(2) 由 (1) 的结果可设计计算给定集合中元素个数的概率算法如下。

```

int count (SET X)
{
    int k=0; SET S;
    a=uniform(X);
    while(true) {
        S.insert(a);
        k++; a=uniform(X);
        if (S.find(a) != S.end()) break;
    }
    return 2*k*k/pi;
}

```

习题 7-5 生日问题

试设计一个近似算法计算 $365!/(340! \times 365^{25})$, 并精确到 4 位有效数字。

分析与解答:

该问题中的数是概率论中著名的生日问题的解答。在 k 个人中, 至少 2 人有相同生日的概率为 $1 - 365! / [(365 - k)! \times 365^k]$ 。

一般情况下, $n! / [(n - k)! n^k]$ 可以用 Stirling 公式化简后做近似计算。

由 Stirling 公式

$$\begin{aligned} n! &= \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left[1 + \theta\left(\frac{1}{n}\right)\right] \\ \ln(x) &= x - x^2/2 + \theta(x^3) \\ n! / [(n - k)! n^k] &= \left[1 + \theta\left(\frac{1}{n}\right)\right] \left[\frac{\sqrt{2\pi n}}{\sqrt{2\pi(n - k)}}\right] \frac{\left(\frac{n}{e}\right)^n}{\left(\frac{n - k}{e}\right)^{n - k} n^k} \\ &= \left[1 + \theta\left(\frac{1}{n}\right)\right] \left(\frac{n}{n - k}\right)^{n - k} e^{-k} \\ &= \left[1 + \theta\left(\frac{1}{n}\right)\right] e^{(n - k) \ln\left(1 + \frac{k}{n - k}\right)} \\ &= \left[1 + \theta\left(\frac{1}{n}\right)\right] e^{(n - k) \left[1 + \frac{k}{n - k} - \frac{k^2}{2(n - k)^2} + \theta\left(\frac{1}{(n - k)^3}\right)\right]} \\ &= \left[1 + \theta\left(\frac{1}{n}\right)\right] e^{-k^2/(2n) + \theta\left(\frac{1}{n^2}\right)} \end{aligned}$$

由此可得

$$n! / [(n - k)! n^k] \approx e^{-k^2/(2n)}$$

用更精确些的估计式

$$\begin{aligned} n! &= \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left[1 + \frac{1}{12n} + \theta\left(\frac{1}{n^2}\right)\right] \\ \ln(x) &= x - x^2/2 + x^3/3 - \theta(x^4) \end{aligned}$$

可得

$$n! / [(n - k)! n^k] = e^{-k(k-1)/(2n-k^3)/(6n^2) \pm O(\max(k^2/n^2, k^4/n^3))}$$

由此可求得

$$\begin{aligned} 365! / (340! \times 365^{25}) &= 0.4311 \\ 1 - 365! / (340! \times 365^{25}) &= 0.5689 \end{aligned}$$

习题 7-6 易验证问题的拉斯维加斯算法

一个问题是易验证的是指对该问题的给定实例的每一个解, 都可以有效地验证其正确性。例如, 求一个整数的非平凡因子问题是易验证的, 而求一个整数的最小非平凡因子就不是易验证的。一般情况下, 易验证问题未必是易解的。

(1) 给定一个解易验证问题 P 的蒙特卡罗方法, 由此设计一个相应的解问题 P 的拉斯维加斯算法。

(2) 给定一个解易验证问题 P 的拉斯维加斯算法, 由此设计一个相应的解问题 P 的蒙特卡罗算法。

分析与解答:

(1) 设给定的解易验证问题 P 的蒙特卡罗算法为 Monte , 验证其解的正确性的算法为 charact , 则相应的解问题 P 的拉斯维加斯算法 Las 如下。

```
void Las(ST x)
{
    bool r=Monte(x);
    while(!charact(x,r))r=Monte(x);
}
```

(2) 设给定的解易验证问题 P 的拉斯维加斯算法为 Las 。在超过时间界限时终止算法 Las 。相应的解问题 P 的蒙特卡罗算法 Monte 如下。

```
bool Monte(ST x)
{
    Las(x);
    if(timeused>maxt)return false;
    else return true;
}
```

习题 7-7 用数组模拟有序链表

用数组模拟有序链表的数据结构, 设计支持下列运算的舍伍德型算法, 并分析各运算所需的计算时间:

- (1) Predecessor 找出一给定元素 x 在有序集 S 中的前驱元素;
- (2) Successor 找出一给定元素 x 在有序集 S 中的后继元素;
- (3) Min 找出有序集 S 中的最小元素;
- (4) Max 找出有序集 S 中的最大元素。

分析与解答:

对主教材中的有序链表类 OrderList 做简单修改。

习题 7-8 $O(n^{3/2})$ 舍伍德型排序算法

采用数组模拟有序链表的数据结构, 设计一个舍伍德型排序算法, 使算法最坏情况下的平均计算时间为 $O(n^{3/2})$ 。

分析与解答:

用主教材中的有序链表类 OrderList , 对有序链表做 n 次插入运算。第 i 次插入运算平均需要 $O(\sqrt{i})$ 计算时间。因此, 相应的排序算法在最坏情况下的平均计算时间为 $O(n^{3/2})$ 。

习题 7-9 n 后问题解的存在性

如果对于某一个 n 的值, n 后问题无解, 则算法将陷入死循环。

- (1) 证明或否定下述论断: 对于 $n \geq 4$, n 后问题有解;
- (2) 是否存在一个正数 δ , 使得对所有 $n \geq 4$ 算法成功的概率至少是 δ 。

分析与解答:

用 x_{ij} 表示在棋盘格子 (i, j) 处放置皇后的状态。当 $x_{ij} = 1$ 时, 表示在棋盘格子 (i, j) 处放置了一个皇后, 否则没有放置皇后。 n 后问题可以表示为如下的 0-1 线性规划问题。

$$\begin{aligned} \max \quad & \sum_{i=1}^n \sum_{j=1}^n x_{ij} \\ \sum_{j=1}^n x_{ij} \leq & 1 & 1 \leq i \leq n \\ \sum_{i=1}^n x_{ij} \leq & 1 & 1 \leq j \leq n \\ \sum_{i+j=k} x_{ij} \leq & 1 & 2 \leq k \leq 2n \\ \sum_{i-j=k} x_{ij} \leq & 1 & 1-n \leq k \leq n-1 \\ x_{ij} \in & \{0, 1\} & 1 \leq i, j \leq n \end{aligned}$$

易知

$$\max \sum_{i=1}^n \sum_{j=1}^n x_{ij} \leq n$$

下面对 $n \geq 4$ 构造上述线性规划问题的一个解, 使 $\max \sum_{i=1}^n \sum_{j=1}^n x_{ij} = n$, 从而证明, 对于 $n \geq 4$, n 后问题有解。分为以下 3 种情况讨论。

(1) 当 $n \geq 4$ 为偶数且 $(n-2) \% 6 > 0$ 时, 对于 $1 \leq j \leq n/2$, 取 $x(j, 2j) = 1$; $x(n/2 + j, 2j - 1) = 1$ 。

(2) 当 $n \geq 4$ 为偶数且 $n \% 6 > 0$ 时, 对于 $1 \leq j \leq n/2$, 取 $x(j, k(j)) = 1$; $x(n+1-j, n-k(j)) = 1$ 。其中, $k(j) = (n/2 + 2(j-1) - 1) \% n$ 。

(3) 当 $n > 4$ 为奇数时, 取 $x(n, n) = 1$, 转换为 $(n-1) \times (n-1)$ 棋盘的问题, 用(1)或(2)的方法构造其余的值。

按照上述方法构造 n 后问题解的算法如下。

```

void construct(int k)
{
    if(k<4)return;
    if (odd(k))x[k]=k--;
    if((k-2)%6>0)build1(k);
    else build2(k);
}

void build1(int k)
{
    k/=2;
    for(int i=1;i<=k;i++){x[i]=2*i;x[k+i]=2*i-1;}
}

void build2(int k)
{

```

```

    for(int i=1;i<=k/2;i++){
        x[i]=1+(2*(i-1)+k/2-1)%k;
        x[k+1-i]=k-(2*(i-1)+k/2-1)%k;
    }
}

```

容易验证,按照上述方法构造出的解是相应于 n 后问题的 0-1 线性规划问题的一个解。由此可见,对于 $n \geq 4$, n 后问题有解。

习题 7-11 整数因子分解算法

假设已有一个算法 $\text{Prime}(n)$ 可用于测试整数 n 是否为一素数。另外还有一个算法 $\text{Split}(n)$ 可实现对合数 n 的因子分割。试利用这 2 个算法设计一个对给定整数 n 进行因子分解的算法。

分析与解答:

因子分解算法如下。

```

void fact(int n)
{
    if(prime(n)) {output(n);return;}
    int i=split(n);
    if(i>1) fact(i);
    if(n>i) fact(n/i);
}

```

习题 7-12 非蒙特卡罗算法的例子

(1) 试证明下面的算法 Primality 能以 80% 以上的正确率判定给定的一个整数 n 是否为素数。另一方面,举出整数 n 的一个例子表明算法对此整数 n 总是给出错误的解答,进而说明该算法不是一个蒙特卡罗算法。

```

bool Primality(int n)
{
    if (gcd(n, 30030) == 1) return true;
    else return false;
}

```

(2) 试找出上述算法 Primality 中可用于替换整数 30030 的另一个整数 (可使用大整数),使得用此整数代替 30030 后,算法的正确率提高到 85% 以上,且允许整数 n 是非常大的整数。

分析与解答:

(1) 30030 是前 6 个素数的乘积, $30030 = 2 \times 3 \times 5 \times 7 \times 11 \times 13$ 。因此,当合数含有素因子 2, 3, 5, 7, 11, 13 时,算法 Primality 给出的解答是正确的。而当合数不含素因子 2, 3, 5,

7, 11, 13 时, 算法 Primality 给出的解答是错误的。例如, 当 $n = 323 = 17 \times 19$ 时, 算法 Primality 给出的解答总是 true, 但总是错误的。可见算法 Primality 不是一个蒙特卡罗算法。

一般情况下, 全体整数集合中含有素因子 p 的整数的比例为 $1/p$ 。设前 m 个素数为 p_1, p_2, \dots, p_m , 由容斥原理知, 全体整数集合中至少含有素因子 p_1, p_2, \dots, p_m 之一的整数的比例为

$$\sum_{i=1}^m \frac{1}{p_i} - \sum_{i=1}^{m-1} \sum_{j=i+1}^m \frac{1}{p_i p_j} + \dots + (-1)^{m-1} \frac{1}{p_1 p_2 \dots p_m} = 1 - \prod_{i=1}^m \left(1 - \frac{1}{p_i}\right)$$

对于本题容易计算出, 全体整数集合中至少含有前 6 个素因子之一的整数的比例为

$$1 - \left(1 - \frac{1}{2}\right) \times \left(1 - \frac{1}{3}\right) \times \left(1 - \frac{1}{5}\right) \times \left(1 - \frac{1}{7}\right) \times \left(1 - \frac{1}{11}\right) \times \left(1 - \frac{1}{13}\right) = 0.8082$$

由此可见, 算法 Primality 能以 80.82% 以上的正确率判定给定的一个整数 n 是否为素数。

(2) 要使算法的正确率提高到 85% 以上, 必须用前 m 个素数的乘积 $p_1 p_2 \dots p_m$ 替代算法中的常数 30030, 使

$$1 - \prod_{i=1}^m \left(1 - \frac{1}{p_i}\right) > 0.85$$

经计算得知:

当 $m = 11$ 时,

$$1 - \prod_{i=1}^{11} \left(1 - \frac{1}{p_i}\right) = 0.8471$$

当 $m = 12$ 时,

$$1 - \prod_{i=1}^{12} \left(1 - \frac{1}{p_i}\right) = 0.8513$$

可见应该用 $p_1 p_2 \dots p_{12} = 7420738134810$ 替代算法中的常数 30030 才能使算法的正确率提高到 85% 以上。

习题 7-13 重复 3 次的蒙特卡罗算法

设 $mc(x)$ 是一个一致的 75% 正确的蒙特卡罗算法, 考虑下面的算法:

```

int mc3(int x)
{
    int t, u, v;
    t = mc(x);
    u = mc(x);
    v = mc(x);
    if ((t == u) || (t == v)) return t;
    return v;
}
    
```

(1) 试证明上述算法 $mc3(x)$ 是一致的 $27/32$ 正确的算法, 因此是 84% 正确的。

(2) 试证明如果 $mc(x)$ 不是一致的, 则 $mc3(x)$ 的正确率有可能低于 71%。

分析与解答:

(1) 重复 3 次的蒙特卡罗算法各次正确的分布有 8 种不同情况:

000, 001, 010, 011, 100, 101, 110, 111

其中 011, 101, 110, 111 这 4 种情况返回正确解。因此返回正确解的概率为

$$\frac{1}{4} \times \frac{3}{4} \times \frac{3}{4} + \frac{3}{4} \times \frac{1}{4} \times \frac{3}{4} + \frac{3}{4} \times \frac{3}{4} \times \frac{1}{4} + \frac{3}{4} \times \frac{3}{4} \times \frac{3}{4} = \frac{27}{32}$$

(2) 如果 $mc(x)$ 不是一致的, 则 110 不能保证返回正确解。因此返回正确解的概率可能低到

$$\frac{1}{4} \times \frac{3}{4} \times \frac{3}{4} + \frac{3}{4} \times \frac{1}{4} \times \frac{3}{4} + \frac{3}{4} \times \frac{3}{4} \times \frac{3}{4} = \frac{45}{64} = 0.7031$$

因此, $mc3(x)$ 的正确率有可能低于 71%。

习题 7-14 集合随机元素算法

设 $I = \{1, 2, \dots, n\}$, $S \subseteq I$ 是 I 的一个子集。MC(x) 是一个偏假 p 正确蒙特卡罗算法。该算法用于判定所给的整数 $1 \leq x \leq n$ 是否为集合 S 中的整数, 即 $x \in S$ 。设 $q = 1 - p$ 。由偏假算法的定义可知, 对任意 $x \in S$ 有 $\text{Prob}\{\text{MC}(x) = \text{true}\} = 1$ 。当 $x \notin S$ 时, $\text{Prob}\{\text{MC}(x) = \text{true}\} \leq q$ 。考虑下面的产生 S 中随机元素的算法 GenRand 如下:

```
bool RepeatMC(int x, int k)
```

```
{
```

```
    int i=0;
```

```
    bool ans=true;
```

```
    while (ans && (i < k)) {
```

```
        i++;
```

```
        ans=MC(x);
```

```
    }
```

```
    return ans;
```

```
}
```

```
int GenRand(int n, int k)
```

```
{
```

```
    RandomNumber rnd;
```

```
    int x=rnd.Random(n)+1;
```

```
    while (!RepeatMC(x, k)) x=rnd.Random(n)+1;
```

```
    return x;
```

```
}
```

假设由语句 $x = \text{rnd.Random}(n) + 1$ 产生的整数 $x \in S$ 的概率为 r , 证明算法 GenRand 返回的整数不在 S 中的概率最多为

$$\frac{1}{1 + \frac{r}{1-r} q^{-k}}$$

分析与解答:

算法 GenRand 返回的整数 x 是第 1 次由语句 $x = \text{rnd.Random}(n) + 1$ 产生的整数, 且 x 不在 S 中的概率为

$$(1-r)q^k$$

算法 GenRand 返回的整数 x 是由语句 $x=\text{rnd. Random}(n)+1$ 第 2 次产生的整数, 且 x 不在 S 中的概率为

$$(1-r)(1-q^k)(1-r)q^k=(1-r)^2(1-q^k)q^k$$

.....

由此可知, 算法 GenRand 返回的整数不在 S 中的概率为

$$\begin{aligned} & (1-r)q^k + (1-r)^2(1-q^k)q^k + (1-r)^3(1-q^k)^2q^k + \cdots \\ &= (1-r)q^k \sum_{i=0}^{\infty} (1-r)(1-q^k)^i \\ &= \frac{(1-r)q^k}{1-(1-r)(1-q^k)} \\ &= \frac{(1-r)q^k}{1-(1-r)(1-q^k)} \\ &= \frac{1}{\frac{1}{(1-r)q^k} - \left(\frac{1}{q^k} - 1\right)} \\ &= \frac{1}{1 + \left(\frac{1}{1-r} - 1\right)q^{-k}} \\ &= \frac{1}{1 + \frac{r}{1-r}q^{-k}} \end{aligned}$$

习题 7-15 由蒙特卡罗算法构造拉斯维加斯算法

设算法 A 和 B 是解同一判定问题的两个有效的蒙特卡罗算法。算法 A 是一个 p 正确偏真算法, 算法 B 则是一个 q 正确偏假算法。试利用这两个算法设计一个解同一问题的拉斯维加斯算法, 并使所得到的算法对任何实例的成功率尽可能高。

分析与解答:

```
bool Las(ST x)
{
    while(true){
        if(A(x))return true;
        if(!B(x))return false;
    }
}
```

习题 7-16 产生素数算法

考虑下面的无限循环算法。

```
void PrintPrimes(void)
{
    cout<<'2' <<endl;
    cout<<'3' <<endl;
```

```

int n=5;
while (true) {
    int m=floor(log(double(n)));
    if (PrimeMC(n,m)) cout<<n<<endl;
    n=n+2;
}
}

```

易知，每一个素数都会被上述算法输出。但是除了所有素数外，算法可能偶尔错误地输出某些合数。说明上述情况不太少可能发生。或更精确地，证明上述算法错误地输出一个（大于 100 的）合数的概率小于 1%。

分析与解答：

$m = \log n$ 。PrimeMC(n, m) 发生错误的概率小于 $\left(\frac{1}{4}\right)^m = \frac{1}{2^{2\log n}} = \frac{1}{n^2}$ 。

习题 7-19 矩阵方程问题

给定 3 个 $n \times n$ 矩阵 A , B 和 C ，下面的偏假 1/2 正确的蒙特卡罗算法用于判定 $AB=C$ 。

```

bool Product(int **A, int **B, int **C, int n)
{
    // 判定 AB=C 的蒙特卡罗算法
    RandomNumber rnd;
    int *x=new int [n+1];
    int *y=new int [n+1];
    int *z=new int [n+1];
    for (int i=1;i<=n;i++) {
        x[i]=rnd.Random(2);
        if (x[i]==0) x[i]=-1;
    }
    Mult(B, x, y, n);
    Mult(A, y, z, n);
    Mult(C, x, y, n);
    for (int i=1;i<=n;i++)
        if (y[i]!=z[i]) return false;
    return true;
}

```

算法所需的计算时间为 $O(n^2)$ 。显然当 $AB=C$ 时，算法 Product(A, B, C, n) 返回 true。试证明当 $AB \neq C$ 时，算法返回值为 false 的概率至少为 1/2（提示：考虑矩阵 $AB-C$ 并证明当 $AB \neq C$ 时，将该矩阵各行相加或相减最终得到的行向量至少有一半是非零向量）。

分析与解答：

设

$$X = \{x \in \mathbb{R}^n \mid x_i = \pm 1, 1 \leq i \leq n\}$$

$$Y = \{x \in X \mid (AB-C)x \neq 0\}$$

$$Z = \{x \in X \mid (AB-C)x = 0\}$$

当 $AB \neq C$ 时, 设 $AB - C$ 的第 i 列非 0, 即 $(AB - C)e_i \neq 0$ 。

对于任意 $z \in Z$, 取 $y \in R^n$ 满足: $y_j = z_j, 1 \leq j \leq n, j \neq i; y_i = -z_i$, 则易知 $y = z - 2z_i e_i$ 。由此可知,

$$(AB - C)y = (AB - C)(z - 2z_i e_i) = -2(AB - C)e_i \neq 0$$

可见如此构造出的 $y \in Y$ 。由不同的 z 构造出的 y 也不同。因此, $|Z| \leq |Y|$ 。

由此可知, 当 $AB \neq C$ 时, 算法 $\text{Product}(A, B, C, n)$ 返回值为 false 的概率至少为 $1/2$ 。

算法实现题 7-1 模平方根问题(习题 7-10)

★问题描述:

设 p 是一个奇素数, $1 \leq x \leq p-1$, 如果存在一个整数 $y, 1 \leq y \leq p-1$, 使得 $x \equiv y^2 \pmod{p}$, 则称 y 是 x 的模 p 平方根。例如 63 是 55 的模 103 平方根。试设计一个求整数 x 的模 p 平方根的拉斯维加斯算法 (算法的计算时间应为 $\log p$ 的多项式)。

★编程任务:

设计一个拉斯维加斯算法, 对于给定的奇素数 p 和整数 x , 计算 x 的模 p 平方根。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 p 和 x 。

★结果输出:

将计算出的 x 的模 p 平方根输出到文件 output.txt。当不存在 x 的模 p 平方根时, 输出 0。

输入文件示例

输出文件示例

input.txt

output.txt

103 55

63

分析与解答:

求整数 x 的模 p 平方根的 Tonelli 算法是一个拉斯维加斯算法, 描述如下。

```
Tonelli (x, p)
{
    (1) 随机选取  $g$ ;
    (2) 设  $p-1=2^s t, t$  奇数;
    (3)  $e=0$ ;
    (4) for ( $i=2; i \leq s; i++$ ) if ( $(xg^{-e})^{(p-1)/2^i} \neq 1$ )  $e=e+2^{i-1}$ ;
    (5)  $h=xg^{-e}$ ;
    (6)  $b=g^{e/2} h^{(t+1)/2}$ ;
    (7) return  $b$ ;
}
```

算法实现如下。

```
uint64 sqrt(uint64 a, uint64 q)
{
    RandomNumber rnd;
    uint64 b=1, d, g, x, y, h, k, s=0, t=q-1;
    if ((q%2)==0) return 0;
```

```

while(t%2==0){t/=2;s++;}
g=rnd.Random(q-1)+1;
execlid(g,q,d,x,y);
if(x<0)x+=q;a%=q;
for(uint64 i=2,j=2,e=0;i<=s;i++,j*=2)
    if(del(a,q,e,x,j)!=1)e+=j;
for(i=1,h=a;i<=e;i++){h*=x;h%=q;}
for(i=1,e/=2;i<=e;i++){b*=g;b%=q;}
for(i=1,k=(t+1)/2;i<=k;i++){b*=h;b%=q;}
return b;
}

```

其中, uint64 是 64 位整型数, uint32 是 32 位整型数。

```
#define uint64 __int64
```

```
#define uint32 unsigned int
```

Exeuclid 是扩展的 euclid 算法, 对于整数 a 和 b , 计算出 d , x 和 y 使

$$d = \gcd(a, b) = ax + by$$

```

void execlid(uint64 a, uint64 b, uint64 &d, uint64 &x, uint64 &y)
{
    uint64 d1, x1, y1;
    if (b==0) {d=a;x=1;y=0;return;}
    execlid(b,a%b,d1,x1,y1);
    d=d1;x=y1;y=x1-a/b*y1;
}

```

算法的计算时间为 $O(\log b)$ 。

del 计算 $(ag^{-e})^{(q-1)/2^j}$ 。

```

uint64 del(uint64 a, uint64 q, uint64 e, uint64 x, uint64 j)
{
    uint64 i, y=1, k=(q-1)/j/2;
    for(i=1;i<=e;i++){a*=x;a%=q;}
    for(i=1;i<=k;i++){y*=a;y%=q;}
    return y;
}

```

算法返回正确解的概率为 $1/2$, 所需计算时间为 $O(\log^4 q)$ 。

可以通过多次调用, 提高算法返回正确解的概率。

```

uint64 sqrtLV(uint64 a, uint64 q)
{
    RandomNumber rnd;
    uint64 k=rnd.Random(100)+1;

```

```

    for(uint64 i=1; i<=k; i++) {
        uint64 r=sqrt(a, q);
        if(r*r%q==a) return r;
    }
    return 0;
}

```

算法实现题 7-2 素数测试问题(习题 7-17)

★问题描述:

试设计一个素数测试的偏真蒙特卡罗算法。要求对于测试的整数 n ，所述算法是一个关于 $\log n$ 的多项式时间算法。

结合教材中素数测试的偏假蒙特卡罗算法，设计一个素数测试的拉斯维加斯算法（参见习题 7-15）。

★编程任务:

设计一个拉斯维加斯算法，对于给定的正整数，判定其是否为素数。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 p 。

★结果输出:

将计算结论输出到文件 output.txt。若正整数 p 是素数则输出 “YES”，否则输出 “NO”。

输入文件示例

input.txt

103

输出文件示例

output.txt

YES

分析与解答:

用 Goldwasser-Kilian 方法或 Adleman-Huang 方法。见如下参考文献:

[1] Shafi Goldwasser and Joe Kilian, *Almost all primes can be quickly certified*, Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing (11 West 42nd St., New York), Association for Computing Machinery, May 1986, pp. 316~329.

[2] Leonard M. Adleman and Ming-Deh A. Huang, *Primality Testing and Abelian Varieties Over Finite Fields*, Lecture Notes in Mathematics, vol. 1512, Springer-Verlag, 1992.

算法实现题 7-3 集合相等问题(习题 7-18)

★问题描述:

给定 2 个集合 S 和 T ，试设计一个判定 S 和 T 是否相等的蒙特卡罗算法。

★编程任务:

设计一个拉斯维加斯算法，对于给定的集合 S 和 T ，判定其是否相等。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n ，表示集合的大小。接下来的 2 行，每行有 n 个正整数，分别表示集合 S 和 T 中的元素。

★结果输出:

将计算结论输出到文件 output.txt。若集合 S 和 T 相等则输出 “YES”，否则输出 “NO”。

输入文件示例

输出文件示例

input.txt

output.txt

3

YES

2 3 7

7 2 3

分析与解答:

类似于主教材中的主元素问题。

算法实现题 7-4 逆矩阵问题(习题 7-20)

★问题描述:

给定 2 个 $n \times n$ 矩阵 A 和 B ，试设计一个判定 A 和 B 是否互逆的蒙特卡罗算法（算法的计算时间应为 $O(n^2)$ ）。

★编程任务:

设计一个蒙特卡罗算法，对于给定的矩阵 A 和 B ，判定其是否互逆。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n ，表示矩阵 A 和 B 为 $n \times n$ 矩阵。接下来的 $2n$ 行，每行有 n 个实数，分别表示矩阵 A 和 B 中的元素。

★结果输出:

将计算结论输出到文件 output.txt。若矩阵 A 和 B 互逆则输出 “YES”，否则输出 “NO”。

输入文件示例

输出文件示例

input.txt

output.txt

3

YES

1 2 3

2 2 3

3 3 3

-1 1 0

1 -2 1

0 1 -0.666667

分析与解答:

与习题 7-19 的算法类似。

```
bool verse(double **A, double **B, int n)
{
    // 判定  $AB=I$  的蒙特卡罗算法
    RandomNumber rnd;
    double *x=new double [n+1];
    double *y=new double [n+1];
    double *z=new double [n+1];
    for (int i=1;i<=n;i++) {
        x[i]=rnd.Random(2);
```

```

        if (x[i]==0.0) x[i]=-1.0;
    }
    Mult(B, x, y, n);
    Mult(A, y, z, n);
    for (i=1; i<=n; i++)
        if (fabs(x[i] - z[i])>1e-4) return false;
    return true;
}

```

算法实现题 7-5 多项式乘积问题(习题 7-21)

★问题描述:

给定阶数分别为 n, m 和 $n+m$ 的多项式 $p(x), q(x)$ 和 $r(x)$ 。试设计一个判定 $p(x)q(x)=r(x)$ 的偏差 $1/2$ 正确的蒙特卡罗算法,并要求算法的计算时间为 $O(n+m)$ 。

★编程任务:

设计一个蒙特卡罗算法,对于给定多项式 $p(x), q(x)$ 和 $r(x)$,判定 $p(x)q(x)=r(x)$ 是否成立。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 3 个正整数 n, m, l , 分别表示多项式 $p(x), q(x)$ 和 $r(x)$ 的阶数。接下来的 3 行, 每行分别有 n, m, l 个实数, 分别表示多项式 $p(x), q(x)$ 和 $r(x)$ 的系数。

★结果输出:

将计算结论输出到文件 output.txt。若 $p(x)q(x)=r(x)$ 成立则输出 “YES”, 否则输出 “NO”。

输入文件示例	输出文件示例
input.txt	output.txt
2 1 3	YES
1 2 3	
2 2	
2 6 10 6	

分析与解答:

多项式的阶为 $k=\max\{mn, l\}$ 。随机选取 $k+1$ 个实数,测试等式是否成立。

算法实现题 7-6 皇后控制问题

★问题描述:

在一个 $n \times n$ 个方格组成的棋盘上的任一方格中放置一个皇后, 该皇后可以控制其所在的行、列及对角线上的所有方格。

对于给定的自然数 n , 在 $n \times n$ 个方格组成的棋盘上最少要放置多少个皇后才能控制棋盘上的所有方格, 且放置的皇后互不攻击?

★编程任务:

设计一个拉斯维加斯算法, 对于给定的自然数 $n(1 \leq n \leq 100)$ 计算在 $n \times n$ 个方格组成的棋盘上最少要放置多少个皇后才能控制棋盘上的所有方格, 且放置的皇后互不攻击。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n 。

★结果输出:

将计算出的最少皇后数及最佳放置方案输出到文件 output.txt。文件的第 1 行是最少皇后数; 接下来的 1 行是皇后的最佳放置方案。

输入文件示例

input.txt

8

输出文件示例

output.txt

5

0 3 6 0 0 2 5 8

分析与解答:

与主教材中 n 后问题的拉斯维加斯算法类似。具体算法描述如下。

类 Queen 的私有成员 n 表示皇后个数; 数组 x 存储 n 后问题的解。

```
class Queen {
    friend bool nQueen(int);
private:
    bool Place(int k);
    int QLV(int m, int stopVegas);
    bool QueensLV(int stopVegas);
    bool ctrl(int n);
    int placed(int k);
    bool backtrack (int t);
    int n,*x,*y,**yy;
};
```

Place(k)用于测试将皇后 k 置于第 $x[k]$ 列的合法性。

```
bool Queen::Place(int k)
{
    if (x[k]>0)
        for (int j=1; j<=k-1; j++)
            if (x[j]>0 && (abs(k-j)==abs(x[j]-x[k]) || x[j]==x[k])) return false;
    return true;
}
```

ctrl 用于测试皇后是否已控制棋盘。

```
bool Queen::ctrl(int nn)
{
    int i, j, t1, t2, cont=0, xmin=0;
    for (i=1; i <=nn; i++)
        for (j=1; j <=nn; j++) yy[i][j]=0;
    for (i=1; i <=nn; i++) {
```

```

        if (x[i]>0) {
            xmin++;
            for (j=1; j <=nn; j++) {yy[i][j]=1;yy[j][x[i]]=1;}
            for(t1=i, t2=x[i]; t1>=1 && t2>=1; t1--, t2--) yy[t1][t2]=1;
            for(t1=i, t2=x[i]; t1<=nn && t2<=nn; t1++, t2++) yy[t1][t2]=1;
            for(t1=i, t2=x[i]; t1>=1 && t2<=nn; t1--, t2++) yy[t1][t2]=1;
            for(t1=i, t2=x[i]; t1<=nn && t2>=1; t1++, t2--) yy[t1][t2]=1;
        }
    }
    for (i=1; i <=nn; i++)
        for (j=1; j <=nn; j++) cont+=yy[i][j];
    return (cont==nn*nn);
}

```

QueensLV (stopVegas) 实现在棋盘上随机放置若干皇后的拉斯维加斯算法。其中 $1 \leq \text{stopVegas} \leq n$ 表示随机放置的皇后数。

```

bool Queen::QueensLV(int stopVegas)
{
    int m=stopVegas, count=0, cont=10000;
    while(true) {
        int ret=QLV(m, stopVegas);
        if(ret>0) {m=ret-1; count=0;}
        else count++;
        if(count>cont) {while(QLV(m+1, stopVegas)==0); break;}
    }
    return true;
}

int Queen::QLV(int m, int stopVegas)
{
    randomNumber rnd;
    while(true) {
        int k=1;
        while (k<=stopVegas) {
            for (int i=0, count=0; i<=n; i++) {
                x[k]=i;
                if (Place(k)) y[count++]=i;
            }
            x[k++] = y[rnd.random(count)];
        }
        int pla=placed(stopVegas);
        if(pla<=m) return pla;
        else return 0;
    }
}

```

```

    }
}

```

与回溯法相结合的解 n 后控制问题的拉斯维加斯算法描述如下。

```

bool nQueen(int n)
{
    Queen X;
    X.n=n;
    int *p=new int [n+1];
    int *q=new int [n+1];
    int **r;
    Make2DArray(r, n+1, n+1);
    for (int i=0; i<=n; i++) p[i]=0;
    X.x=p;X.y=q;X.yy=r;
    int stop=3;
    if(n>15)stop=n-15;
    bool found=false;
    while (!X.QueensLV(stop)) ;
    if(X.backtrack(stop+1)){
        cout<<X.placed(n)<<endl;
        for (i=1;i<=n;i++) cout<<p[i]<<" ";
        cout<<endl;
        found=true;
    }
    delete [] p; delete [] q;
    Delete2DArray(r, n+1);
    return found;
}

```

算法的回溯搜索部分与解 n 后问题的回溯法是类似的，只要找到一个解就返回。

```

bool Queen::backtrack (int t)
{
    if (t>n) {
        if (ctrl(n)) return true;
        else return false;
    }
    for (int i=0; i <=n; i++) {
        x[t]=i;
        if (Place(t) && backtrack(t+1))return true;
    }
    return false;
}

```

placed 计算已放置的皇后数。

```
int Queen::placed(int k)
{
    for (int j=1, num=0; j<=k; j++) if (x[j]>0) num++;
    return num;
}
```

算法实现题 7-7 3-SAT 问题

★问题描述:

SAT 的一个实例是 k 个布尔变量 x_1, x_2, \dots, x_k 的 m 个布尔表达式 A_1, A_2, \dots, A_m 。若存在各布尔变量 $x_i (1 \leq i \leq k)$ 的 0,1 赋值,使每个布尔表达式 $A_i (1 \leq i \leq m)$ 都取值为 1,则称布尔表达式 $A_1 A_2 \dots A_m$ 是可满足的。

(1) 合取范式的可满足性问题 CNF-SAT

如果一个布尔表达式是一些因子和之积,则称为合取范式,简称 CNF (Conjunctive Normal Form)。这里的因子是变量 x 或 \bar{x} 。例如 $(x_1 + x_2)(x_2 + x_3)(\bar{x}_1 + \bar{x}_2 + x_3)$ 就是一个合取范式,而 $x_1 x_2 + x_3$ 就不是合取范式。

(2) k -SAT

如果一个布尔合取范式的每个乘积项最多是 k 个因子的析取式,就称为 k 元合取范式,简记为 k -CNF。一个 k -SAT 问题是判定一个 k -CNF 是否可满足。特别地,当 $k=3$ 时,3-SAT 问题在 NP 完全问题树中具有重要地位。

(3) MAX-SAT

给定 k 个布尔变量 x_1, x_2, \dots, x_k 的 m 个布尔表达式 A_1, A_2, \dots, A_m , 求各布尔变量 $x_i (1 \leq i \leq k)$ 的 0,1 赋值,使尽可能多的布尔表达式 A_i 取值为 1。

(4) Weighted-MAX-SAT

给定 k 个布尔变量 x_1, x_2, \dots, x_k 的 m 个布尔表达式 A_1, A_2, \dots, A_m , 每个布尔表达式 A_i 都有一个权值 w_i , 求各布尔变量 $x_i (1 \leq i \leq k)$ 的 0,1 赋值,使取值 1 的布尔表达式权值之和达到最大。

★编程任务:

对于给定的带权 3-CNF, 设计一个蒙特卡罗算法,使其权值之和尽可能大。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 k 和 m , 分别表示变量数和布尔表达式数。接下来的 m 行中,每行有 5 个整数 $w, i, j, k, 0$, 表示相应表达式的权值为 w , 表达式含的变量下标分别为 i, j, k , 行末以 0 结尾。下标为负数时,表示相应的变量为取反变量。

★结果输出:

将计算出的最大权值输出到文件 output.txt。

输入文件示例

input.txt

5 3

输出文件示例

output.txt

26


```

9 3 1 4 0
9 1 -5 3 0
8 2 -5 1 0

```

分析与解答:

与主教材中 n 后问题的拉斯维加斯算法类似。随机产生布尔变量的真值赋值。

算法实现题 7-8 战车问题

★问题描述:

在 $n \times n$ 格的棋盘上放置彼此不受攻击的车。按照国际象棋规则,车可以攻击与之处在同一行或同一列上的车。在棋盘上的若干个格中设置了堡垒,战车无法穿越堡垒攻击别的战车。对于给定的设置了堡垒的 $n \times n$ 格棋盘,设法放置尽可能多的彼此不受攻击的车。

★编程任务:

对于给定的设置了堡垒的 $n \times n$ 格棋盘,设计一个概率算法,在棋盘上放置尽可能多的彼此不受攻击的车。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n 。接下来的 n 行中,每行有 1 个由字符 “.” 和 “X” 组成的长度为 n 的字符串。

★结果输出:

将计算出的在棋盘上可以放置的彼此不受攻击的战车数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
4	6
....	
..X.	
.X..	
....	

分析与解答:

与主教材中 n 后问题的拉斯维加斯算法类似。在 $n \times n$ 格的棋盘上随机放置彼此不受攻击的车。具体算法实现如下。

init 进行初始化计算。

```

void init(int n)
{
    int i, j, k, x, y;
    state=new int[n*n+2];
    Make2DArray(link, n*n+1, 2*n+1);
    state[0]=-1;
    state[n*n+1]=-1;
    for (int no=1; no <=n*n; no++) {
        i=(no-1)/n; j=(no-1)%n;
        link[no][0]=0; state[no]=-1;
    }
}

```

```

    if (row[i][j]!='.') {
        state[no]=0;k=0;y=j;
        while ((y<n-1) && (row[i][y+1]!='.')) {
            link[no][0]++;y++;k++;
            link[no][k]=i*n+y+1;
        }
        y=j;
        while ((y>0) && (row[i][y-1]!='.')) {
            link[no][0]++;y--;k++;
            link[no][k]=i*n+y+1;
        }
        x=i;
        while ((x<n-1) && (row[x+1][j]!='.')) {
            link[no][0]++;x++;k++;
            link[no][k]=x*n+j+1;
        }
        x=i;
        while ((x>0) && (row[x-1][j]!='.')) {
            link[no][0]++;x--;k++;
            link[no][k]=x*n+j+1;
        }
    }
}
}

```

实现随机算法的主函数如下。

```

int main()
{
    int n;
    randomNumber rnd;
    fin>>n;
    for (int i=0; i<n; i++) fin>>row[i];
    init(n);
    int max=0, rept=0, put=0;
    while(rept<100000) {
        rept++;int count=0;
        while(true) {
            int x=rnd.random(n*n)+1, c=x;
            while ((x<=n*n) && (state[x]!=put))x++;
            if (state[x]!=put) {
                x=c;
                while ((x>0) && (state[x]!=put))x--;
            }
        }
    }
}

```

```

    if (state[x]==put){
        count++;
        for (i=1; i<=link[x][0]; i++)
            if (state[link[x][i]]==put) state[link[x][i]]++;
        state[x]++;
    }
    else break;
}
if (count>max) max=count;
put++;
}
cout<<max<<endl;
return 0;
}

```

算法实现题 7-9 圆排列问题

★问题描述:

给定 n 个大小不等的圆 c_1, c_2, \dots, c_n , 现要将这 n 个圆排进一个矩形框中, 且要求各圆与矩形框的底边相切。圆排列问题要求从 n 个圆的所有排列中找出有最小长度的圆排列。例如, 当 $n=3$, 且所给的 3 个圆的半径分别为 1, 1, 2 时, 这 3 个圆的最小长度的圆排列如图 7-1 所示。其最小长度为 $2+4\sqrt{2}$ 。

解圆排列问题的一个随机化算法如下。

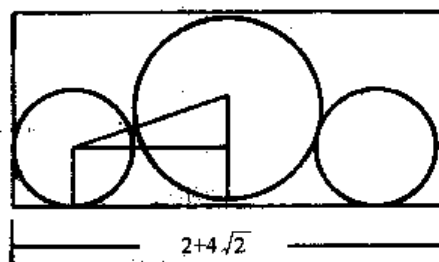


图 7-1 圆排列问题

```

void Circle_search(int *x)
{
    random_perm(x);
    found=true;
    while(found){
        found=false;
        for(int i=1; i<=n; i++)
            for(int j=1; j<=n; j++)
                if (swap(x[i], x[j]) reduces length){
                    swap(x[i], x[j]);
                    found=true;
                }
    }
}

```

其中, random_perm(x) 产生 x 的一个随机排列。

★编程任务:

根据上述算法框架, 设计一个随机化算法, 对于给定的 n 个圆, 计算 n 个圆的最佳排列

方案,使其长度尽可能小。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n ($1 \leq n \leq 20$)。接下来的 1 行有 n 个数,表示 n 个圆的半径。

★结果输出:

将计算出的最小圆排列的长度输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3	7.65685
1 1 2	

分析与解答:

与主教材中 n 后问题的拉斯维加斯算法类似,算法已在题目描述中。

算法实现题 7-10 骑士控制问题

★问题描述:

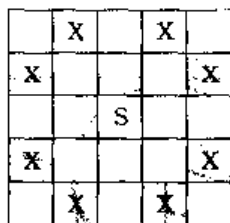


图 7-2 骑士攻击的棋盘方格

在一个 $m \times n$ 个方格的国际象棋棋盘上,骑士(马)可以攻击的棋盘方格如图 7-2 所示。

★编程任务:

对于给定的 $m \times n$ 个方格的国际象棋棋盘,计算棋盘上最少需要放置多少个骑士,使得每个方格至少受到 k 个骑士的攻击。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 3 个正整数 m , n 和 k , 分别表示棋盘的大小为 m 行, n 列, 每个方格至少受到 k 个骑士的攻击。

★结果输出:

将计算出的最少骑士数输出到文件 output.txt。问题无解时输出 “No Solution!”。

输入文件示例	输出文件示例
input.txt	output.txt
4 4 1	6

分析与解答:

用主教材中 n 后问题的拉斯维加斯算法类似的算法求解。

此题也可以用整数线性规划算法求解。设第 i 行第 j 列相应的变量为 x_{ij} 。变量 x_{ij} 的含义是, (i, j) 方格中放置了 x_{ij} 个骑士, $x_{ij} \in \{0, 1\}$ 。

骑士控制问题的求解目标是 $\sum_{i=1}^m \sum_{j=1}^n x_{ij}$ 达到最小。

约束条件是, 每个方格至少受到 k 个骑士的攻击。

设可以攻击 (i, j) 方格的其他方格的集合是 S_{ij} , $k_{ij} = |S_{ij}|$, 则 $2 \leq k_{ij} \leq 8$ 。

相应于每个方格的约束条件可以表述为

$$\sum_{(p,q) \in S_{ij}} x_{pq} \geq k$$

由此可见，骑士控制问题可以变换为如下的整数线性规划问题：

$$\begin{aligned} \min & \sum_{i=1}^m \sum_{j=1}^n x_{ij} \\ \text{s. t.} & \sum_{(p,q) \in S_{ij}} x_{pq} \geq k \\ & x_{ij} \in \{0,1\} \end{aligned}$$

算法实现题 7-11 骑士对攻问题

★问题描述：

在一个 $m \times n$ 个方格的国际象棋棋盘上，骑士（马）可以攻击的棋盘方格如图 7-3 所示。

★编程任务：

对于给定的 $m \times n$ 个方格的国际象棋棋盘，计算棋盘上最多可以放置多少个骑士，使得每个骑士仅受到另一个骑士的攻击，如图 7-4 所示。

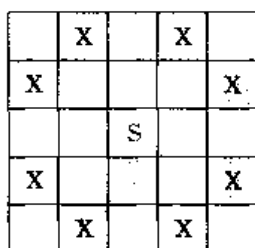


图 7-3 国际象棋棋盘

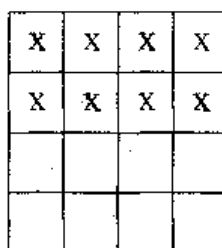


图 7-4 骑士对攻问题

★数据输入：

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 m 和 n ，分别表示棋盘的大小为 m 行、 n 列。

★结果输出：

将计算出的最多骑士数输出到文件 output.txt。问题无解时输出“No Solution!”。

输入文件示例

输出文件示例

input.txt

output.txt

4 4

8

分析与解答：

用主教材中 n 后问题的拉斯维加斯算法类似的算法求解。

此题也可以用整数线性规划算法求解。设第 i 行第 j 列相应的变量为 x_{ij} 。变量 x_{ij} 的含义是， (i,j) 方格中放置了 x_{ij} 个骑士， $x_{ij} \in \{0,1\}$ 。

骑士对攻问题的求解目标是 $\sum_{i=1}^m \sum_{j=1}^n x_{ij}$ 达到最大。

约束条件是，每个骑士仅受到另一个骑士的攻击。

设可以攻击 (i,j) 方格的其他方格的集合是 S_{ij} ， $k_{ij} = |S_{ij}|$ ，则 $2 \leq k_{ij} \leq 8$ 。

相应于每个方格的约束条件可以表述为

$$(k_{ij} - 1)x_{ij} + \sum_{(p,q) \in S_{ij}} x_{pq} \leq k_{ij}$$

$$\sum_{(p,q) \in S_{ij}} x_{pq} - x_{ij} \geq 0$$

由此可见，骑士对攻问题可以变换为如下的整数线性规划问题：

$$\begin{aligned} & \max \sum_{i=1}^m \sum_{j=1}^n x_{ij} \\ \text{s. t. } & (k_{ij} - 1)x_{ij} + \sum_{(p,q) \in S_{ij}} x_{pq} \leq k_{ij} \\ & \sum_{(p,q) \in S_{ij}} x_{pq} - x_{ij} \geq 0 \\ & x_{ij} \in \{0, 1\} \end{aligned}$$

第 8 章 线性规划与网络流

习题 8-1 线性规划可行区域无界的例子

试给出一个线性规划的例子,使其可行区域是无界的,但其最优目标函数值却是有限的。

分析与解答:

$$\begin{aligned} \max z &= -y \\ y - x &\geq 1 \\ x, y &\geq 0 \end{aligned}$$

在 $(x, y) = (0, 1)$ 处达到惟一的最大值 -1 。可行区域显然是无界的。如图 8-1 所示。

习题 8-2 单源最短路与线性规划

试将单源最短路问题表示为一个线性规划问题。

分析与解答:

(1) 单源单终点最短路问题

给定一个赋权有向图 $G = (V, E)$, 其中每条边 (i, j) 的权是一个非负实数 w_{ij} 。另外, 还给定 V 中的两个顶点 s 和 t 。图 G 中以 s 为起点 t 为终点的有向路称为 $s-t$ 有向路, 它经过的所有边权之和称为该 $s-t$ 有向路的长度。单源单终点最短路问题要求最短的 $s-t$ 有向路。

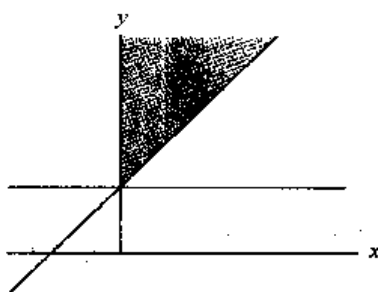


图 8-1 可行区域无界

设变量 x_{ij} 表示边 (i, j) 是否在 $s-t$ 有向路上: 当 $x_{ij} = 1$ 时表示边 (i, j) 在 $s-t$ 有向路上; 当 $x_{ij} = 0$ 时表示边 (i, j) 不在 $s-t$ 有向路上。由此可将单源单终点最短路问题表示为如下线性规划问题:

$$\begin{aligned} \min \quad & \sum_{(i,j) \in E} w_{ij} x_{ij} \\ \text{s. t.} \quad & \sum_{j, (i,j) \in E} x_{ij} - \sum_{j, (i,j) \in E} x_{ji} = \begin{cases} 1 & i = s \\ -1 & i = t \\ 0 & i \neq s, t \end{cases} \\ & x_{ij} \geq 0 \end{aligned}$$

从变量 x_{ij} 的定义可以看出, 变量 x_{ij} 应为 0-1 变量, 上述线性规划问题应表述为整数线性规划问题。但从约束条件可以看出, 约束矩阵为全 1 模阵, 因此 0-1 变量可以松弛为区间 $[0, 1]$ 中的实数, 用单纯形算法求解可得到 0-1 整数解。如果将变量 x_{ij} 松弛为所有非负实数, 则约束条件还不足以保证所有非 0 变量所对应的边构成一条 $s-t$ 有向路, 可能含有向圈。由于每条边的权是一个非负实数, 目标函数是非负的。任何正圈不可能使目标函数最小, 因此找到的结构是一条 $s-t$ 有向路, 最多含有 0 圈。

最短路的最优性条件定理: 对于赋权有向图 G 的每个顶点 $j \in V$, 设 d_j 表示从源 s 到 j 的有向路的长度, 则 d_j 是最短路的长度的充分必要条件是, 对于任何 $(i, j) \in E$ 有

$$d_j \leq d_i + w_{ij}$$

由最短路最优性条件定理, 可将单源单终点最短路问题表述为如下的线性规划问题:

$$\begin{aligned} & \min d_t \\ \text{s. t.} \quad & \begin{cases} d_j \leq d_i + w_{ij} \\ d_s = 0 \end{cases} \end{aligned}$$

(2) 单源最短路问题

一般的单源最短路问题要求从源 s 到所有其他顶点的最短路。类似地, 可将单源最短路问题表示为如下一个线性规划问题:

$$\begin{aligned} & \min \sum_{(i,j) \in E} w_{ij} x_{ij} \\ \text{s. t.} \quad & \sum_{j, (i,j) \in E} x_{ij} - \sum_{j, (j,i) \in E} x_{ji} = \begin{cases} n-1 & i = s \\ -1 & i \neq s \end{cases} \\ & x_{ij} \geq 0 \end{aligned}$$

由最短路最优性条件定理, 也可将单源最短路问题表述为如下的线性规划问题:

$$\begin{aligned} & \min \sum_{i \neq s} d_i \\ \text{s. t.} \quad & \begin{cases} d_j \leq d_i + w_{ij} \\ d_s = 0 \end{cases} \end{aligned}$$

习题 8-3 网络最大流与线性规划

试将网络最大流问题表示为一个线性规划问题。

分析与解答:

对于给定的网络 $G=(V,E)$, 源为 s , 汇为 t 。设边 (i,j) 的容量为 u_{ij} , 边 (i,j) 的流量为 x_{ij} , 可将网络最大流问题表示为如下线性规划问题:

$$\begin{aligned} & \max f \\ \text{s. t.} \quad & \sum_{j, (i,j) \in E} x_{ij} - \sum_{j, (j,i) \in E} x_{ji} = \begin{cases} f & i = s \\ -f & i = t \\ 0 & i \neq s, t \end{cases} \\ & 0 \leq x_{ij} \leq u_{ij} \end{aligned}$$

习题 8-4 最小费用流与线性规划

试将网络最小费用流问题表示为一个线性规划问题。

分析与解答:

对于给定的网络 $G=(V,E)$, 源为 s , 汇为 t 。设边 (i,j) 的容量为 u_{ij} , 边 (i,j) 的流量为 x_{ij} , 边 (i,j) 的单位流量费用为 c_{ij} , 顶点 i 的流供需量为 d_i , 可将网络最小费用流问题表示为如下线性规划问题:

$$\begin{aligned} & \min \sum_{(i,j) \in E} c_{ij} x_{ij} \\ \text{s. t.} \quad & \sum_{j, (i,j) \in E} x_{ij} - \sum_{j, (j,i) \in E} x_{ji} = d_i \\ & 0 \leq x_{ij} \leq u_{ij} \end{aligned}$$

习题 8-5 运输计划问题

某集团公司拥有自己的产品运输网络。该公司现生产 k 种不同的产品。每种产品都需要从生产地运输到销售地。假设第 i 种产品的产地为 s_i ，销售地为 t_i ，需要运送的运输量为 d_i 。集团公司需要规划其运输计划满足各种产品的运输需求。试建立该问题的线性规划模型。

分析与解答：

设公司的产品运输网络为 $G=(V,E)$ 。边 (u,v) 的容量为 $c(u,v)$ 。产品 i 在边 (u,v) 上的运输量为 $f_i(u,v)$ ，则上述问题可以表示为如下线性规划问题：

$$\begin{aligned} & \min 0 \\ \text{s. t.} \quad & \begin{cases} \sum_{i=1}^k f_i(u,v) \leq c(u,v) & u,v \in V \\ f_i(u,v) = -f_i(v,u) & i = 1, 2, \dots, k \\ \sum_{v \in V} f_i(u,v) = 0 & u \in V - \{s_i, t_i\} \\ \sum_{v \in V} f_i(s_i, v) = d_i & i = 1, 2, \dots, k \end{cases} \end{aligned}$$

习题 8-6 单纯形算法

试用单纯形算法解下面的线性规划问题：

$$\begin{aligned} & \min z = x_1 + x_2 + x_3 \\ & 2x_1 + 7.5x_2 + 3x_3 \geq 10000 \\ & 20x_1 + 5x_2 + 10x_3 \geq 30000 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

分析与解答：

用主教材中单纯形算法求解。输入格式为

-1 2 3

0 0 2

2 7.5 3 10000

20 5 10 30000

1 1 1

输出结果如下：

Minimize: 1.00 * x1+1.00 * x2+1.00 * x3

2.00 * x1+7.50 * x2+3.00 * x3 >= 10000.00

20.00 * x1+5.00 * x2+10.00 * x3 >= 30000.00

初始单纯形表：

	0.00	-1.00	-1.00	-1.00	0.00	0.00
6	10000.00	2.00	7.50	3.00	-1.00	0.00
7	30000.00	20.00	5.00	10.00	0.00	-1.00
	0.00	22.00	12.50	13.00	-1.00	-1.00

单纯形算法求解：

	1500.00	0.05	-0.75	-0.50	0.00	-0.05
6	7000.00	-0.10	7.00	2.00	-1.00	0.10
1	1500.00	0.05	0.25	0.50	0.00	-0.05
	33000.00	-1.10	7.00	2.00	-1.00	0.10

	2250.00	0.04	0.11	-0.29	-0.11	-0.04
2	1000.00	0.01	0.14	0.29	-0.14	0.01
1	1250.00	0.05	-0.04	0.43	0.04	-0.05
	-40000.00	-1.00	-1.00	0.00	0.00	0.00

变量：1,2,3；松弛变量：4,5；人工变量：6,7。

最优值：2250.00。

最优解： $x_1=1250.00$ ， $x_2=1000.00$ ， $x_3=0.00$ 。

习题 8-7 边连通度问题

无向图 $G=(V,E)$ 的边连通度为 k 是指最少需要移去 G 的 k 条边才能使 G 成为不连通图。例如，树的边连通度为 1；循环链的边连通度为 2。试用网络最大流算法求给定图 G 的边连通度。

分析与解答：

可以通过解 $|V|$ 个最大流问题求给定图 G 的边连通度，具体做法如下。设 (u,v) 是 G 的一条边。将图 G 看作是等价的有向图，即将图 G 的每条边看作 2 条有向边。设每条边的流容量为 1，并设源 s 为 u ，汇 t 为 v 。该网络的最大流量记为 $f(u,v)$ ，则由最大流和最小割定理可知，穿过任何割的边数等于割的容量，从而等于该网络的最大流量 $f(u,v)$ 。由此可见，图 G 的边连通度为 $\min_{v \in V - \{u\}} \{f(u,v)\}$ 。

习题 8-8 有向无环网络的最大流

试证明有向无环网络的最大流问题等价于标准网络最大流问题。

分析与解答：

有向无环网络的最大流问题是标准网络最大流问题的特殊情形，因此只要证明标准网络最大流问题可以变换为有向无环网络的最大流问题。给定一个标准网络 $G=(V,E)$ ，设 $n=|V|$ ， $m=|E|$ ，下面构造一个与之相应的有 $2n+2$ 个顶点和 $m+3n$ 条边的有向无环网络。

设标准网络 G 的顶点编号为 $0,1,\dots,n-1$ 。每个顶点 i 增加一个新顶点 $n+i$ 构成一个如下二分图：

- (1) 原网络中的边 (i,j) 对应于新网络中的边 $(i,n+j)$ ，容量不变， $0 \leq i,j < n$ 。
- (2) 设 X 是原网络中各边容量之和，新网络中增加边 $(i,n+i)$ ， $0 \leq i < n$ ，容量为 X 。
- (3) 另外增加源 s 和汇 t 。增加边 (s,i) ， $(n+i,t)$ ， $0 \leq i < n$ 。

新网络显然是一个有向无环网络。

进一步还可以证明，对于原网络中每个大小为 c 的 s - t 割都对应于新网络中的一个大小为 $c+X$ 的 s - t 割；新网络中每个大小为 $c+X$ 的最小 s - t 割都对应于原网络中的一个大小为 $c+X$ 的 s - t 割。因此，新网络的最小割对应于原网络的最小割。从而新网络的最大流也对应

于原网络的最大流。

习题 8-9 无向网络的最大流

试将无向网络的最大流问题变换为标准网络最大流问题。

分析与解答：

无向网络的最大流问题的定义为：给定一个赋权无向图 G ，每条边的边权解释为该边上流的容量。一条边上的流可以看作对流的定向，但必须满足容量约束和顶点的流量平衡条件。无向网络的最大流问题要求从 s 到 t 的最大流。

对于给定的无向网络，构造相应的有向网络如下：将原网络的每条边都扩展为 2 条有向边，这 2 条有向边的容量均为原无向边的边权。原无向网络的流显然也是新网络中的流。反之，若新网络中边 (u, v) 上的流为 f ，边 (v, u) 上的流为 g ，则当 $f \geq g$ 时，对应于原网络边流量为 $f - g$ 的 (u, v) 定向，否则对应于原网络边流量为 $g - f$ 的 (v, u) 定向。由此可见，新网络的最大流与原网络最大流对应。

习题 8-12 最大流更新算法

设 $G=(V, E)$ 是源为 s 汇为 t ，且容量均为整数的一个流网络。已知 f 是 G 的一个最大流。

(1) 假设一条边 $(u, v) \in E$ 的容量增 1，试设计一个在 $O(|V| + |E|)$ 时间内更新最大流 f 的算法；

(2) 假设一条边 $(u, v) \in E$ 的容量减 1，试设计一个在 $O(|V| + |E|)$ 时间内更新最大流 f 的算法。

分析与解答：

(1) 将边 $(u, v) \in E$ 的容量增 1 得到新的残留网络。在新的残留网络网络中找一条增广路并增流。更新最大流的算法显然只需 $O(|V| + |E|)$ 时间。如果边 $(u, v) \in E$ 的容量增加 k ，则在最坏情况下更新最大流的算法需要 $O(k(|V| + |E|))$ 时间。

(2) 将边 $(u, v) \in E$ 的容量减 1，如果边 (u, v) 的流量违反容量约束，则将 $\text{flow}(u, v)$ 减 1。此时顶点 u 处的存流为 1，而顶点 v 处的存流为 -1。用增广路算法找顶点 u 到顶点 v 的增广路，消去顶点 u 和顶点 v 处的存流。如果找不到顶点 u 到顶点 v 的增广路，则最大流必须减 1。此时仍可用增广路算法找顶点 u 到顶点 s 的路和顶点 t 到顶点 v 的路，并沿这 2 条路减流得到新的最大流。更新最大流的算法显然只需 $O(|V| + |E|)$ 时间。如果边 $(u, v) \in E$ 的容量减少 k ，则在最坏情况下更新最大流的算法需要 $O(k(|V| + |E|))$ 时间。

习题 8-16 混合图欧拉回路问题

试设计一个找混合图(既有无向边也有有向边的图)的欧拉回路的有效算法。

分析与解答：

问题的关键是给每条无向边定向，使定向后的有向图有欧拉回路。为此构造网络 G 如下。设原图有 n 个顶点和 m 条边。将原图中每个顶点 u 拆为 2 个顶点 u 和 u_0 ，并增加边 (u, u_0) ，其容量为 ∞ 。原图中每条无向边 (u, v) 对应 2 条容量为 1 的有向边 (u_0, v) 和 (v_0, u) 。原图中每条有向边 (u, v) 对应一条容量为 1 的有向边 (u_0, v) 。网络中若存在一个可行流使得每条新增加边 (u, u_0) 上的流量均为顶点 u 在原图中度数的一半，则原图有欧拉回路。为此目的，另外增加源 s 和汇 t 。从源 s 到每个顶点 u_0 有一条有向边 (s, u_0) ，其容量为

$\deg(u)/2$ 。从每个顶点 u 到汇 t 有一条有向边 (u, t) ，其容量为 $\deg(u)/2$ 。求所建立的网络中流量为 m 的可行流。

习题 8-22 单源最短路与最小费用流

试将单源最短路问题表示为一个最小费用流问题。

分析与解答：

对于给定的源顶点为 $source$ 的有向图 G 的单源最短路问题，构造相应网络如下。网络在原有向图 G 的基础上，给每条有向边的容量为 ∞ ，费用为原来的边权。另外增加源 s 和汇 t 。从 s 到源顶点 $source$ 有一条有向边 $(s, source)$ ，其容量为 $n-1$ ，费用为 0。从 $source$ 外的其他 $n-1$ 个顶点到汇 t 有一条有向边，其容量为 1，费用为 0。

求上述网络的最小费用可行流。得到的流支撑树就对应于原图 G 的最短路支撑树。

习题 8-23 中国邮路问题

试用最小费用流算法解中国邮路问题。

分析与解答：

对于给定的有向图 G 的中国邮路问题，构造相应网络如下。网络在原有向图 G 的基础上，给每条有向边的容量为 ∞ ，费用为原来的边权。另外增加每条边的容量下界约束 1，即要求每条边上的流量至少为 1。求上述网络的最小费用循环流。根据流分解定理，得到的循环流可以分解为若干个圈，由此容易构造出原图 G 的中国邮路。

算法实现题 8-1 飞行员配对方案问题（习题 8-10）

★问题描述：

第二次世界大战时期，英国皇家空军从沦陷国征募了大量外籍飞行员。由皇家空军派出的每一架飞机都需要配备在航行技能和语言上能互相配合的 2 名飞行员，其中一名是英国飞行员，另一名是外籍飞行员。在众多的飞行员中，每一名外籍飞行员都可以与其他若干名英国飞行员很好地配合。如何选择配对飞行的飞行员才能使一次派出最多的飞机。对于给定的外籍飞行员与英国飞行员的配合情况，试设计一个算法找出最佳飞行员配对方案，使皇家空军一次能派出最多的飞机。

★编程任务：

对于给定的外籍飞行员与英国飞行员的配合情况，编程找出一个最佳飞行员配对方案，使皇家空军一次能派出最多的飞机。

★数据输入：

由文件 `input.txt` 提供输入数据。文件第 1 行有 2 个正整数 m 和 n 。 n 是皇家空军的飞行员总数 ($n < 100$)； m 是外籍飞行员数。外籍飞行员编号为 $1 \sim m$ ；英国飞行员编号为 $(m+1) \sim n$ 。接下来每行有 2 个正整数 i 和 j ，表示外籍飞行员 i 可以和英国飞行员 j 配合。文件最后以 2 个 -1 结束。

★结果输出：

程序运行结束时，将最佳飞行员配对方案输出到文件 `output.txt`。第 1 行是最佳飞行员配对方案一次能派出的最多的飞机数 M 。接下来 M 行是最佳飞行员配对方案。每行有 2 个正整数 i 和 j ，表示在最佳飞行员配对方案中，飞行员 i 和飞行员 j 配对。

如果所求的最佳飞行员配对方案不存在, 则输出 “No Solution!”。

输入文件示例

输出文件示例

input.txt

output.txt

5 10

4

1 7

1 7

1 8

2 9

2 6

3 8

2 9

5 10

2 10

3 7

3 8

4 7

4 8

5 10

-1 -1

分析与解答:

本题的数学模型是二分图的最大基数匹配问题, 可将其变换为网络最大流问题。

如果图 G 是一个二分图, 则可将图 G 的顶点集合分成两部分 X 和 Y , 使得 G 中任一边所关联的两个顶点中, 有一个顶点属于 X , 而另一个顶点属于 Y 。本题中外籍飞行员与英国飞行员的配合情况图就是一个二分图。外籍飞行员所对应的顶点为 X , 英国飞行员所对应的顶点为 Y 。

二分图的最大匹配问题就是在已知图 G 是一个二分图的前提下, 求 G 的最大匹配。

给定一个二分图 G 和将图 G 的顶点集合 V 分成互不相交的两部分的顶点子集 X 和 Y , 构造与之相应的网络 F 如下。

- (1) 增加一个源 s 和一个汇 t ;
- (2) 从 s 向 X 的每一个顶点都增加一条边; 从 Y 的每一个顶点都向 t 增加一条边;
- (3) 原图 G 中的每一条边都改为相应的由 X 指向 Y 的有向边;
- (4) 置所有边的容量为 1。

求网络 F 的最大流。在从 X 指向 Y 的边集中, 流量为 1 的边对应于二分图中的匹配边。最大流值对应于二分图 G 的最大匹配边数。

主教材中的类 BMATCHING 算法可用于解二分图的最大匹配问题。

实现算法的主函数如下。

```
int main()
{
    int s, t, N, nl;
    char file[11] = "bm.txt";
    fin >> nl >> N;
    fout << nl << " " << N << endl;
    fin >> s >> t;
    while(s >= 0) {
```

```

    fout<<s-1<<" "<<t-1<<endl;
    fin>>s>>t;
}
fout<<-1<<-1<<endl;
GRAPH<EDGE>G(N,1);
G.readbm(file);
BMATCHING<GRAPH<EDGE>, EDGE>(G, n1);
return 0;
}

```

算法实现题 8-2 太空飞行计划问题(习题 8-11)

★问题描述:

W 教授正在为国家航天中心计划一系列的太空飞行。每次太空飞行可进行一系列商业性实验而获取利润。现已确定了一个可供选择的实验集合 $E=\{E_1, E_2, \dots, E_m\}$ 和进行这些实验需要使用的全部仪器的集合 $I=\{I_1, I_2, \dots, I_n\}$ 。实验 E_j 需要用到的仪器是 I 的子集 $R_j \subseteq I$ 。配置仪器 I_k 的费用为 c_k 美元。实验 E_j 的赞助商已同意为该实验结果支付 p_j 美元。W 教授的任务是找出一个有效算法, 确定在一次太空飞行中要进行哪些实验并因此而配置哪些仪器才能使太空飞行的净收益最大。这里净收益是指进行实验所获得的全部收入与配置仪器的全部费用的差额。

★编程任务:

对于给定的实验和仪器配置情况, 编程找出净收益最大的实验计划。

★数据输入:

由文件 input.txt 提供输入数据。文件第 1 行有 2 个正整数 m 和 n 。 m 是实验数, n 是仪器数。接下来的 m 行, 每行是一个实验的有关数据。第一个数是赞助商同意支付该实验的费用; 接着是该实验需要用到的若干仪器的编号。最后一行的 n 个数是配置每个仪器的费用。

★结果输出:

程序运行结束时, 将最佳实验方案输出到文件 output.txt。第 1 行是实验编号; 第 2 行是仪器编号; 最后一行是净收益。

输入文件示例

input.txt

2 3

10 1 2

25 2 3

5 6 7

输出文件示例

output.txt

1 2

1 2 3

17

分析与解答:

建立网络最大流模型解此问题。

(1) 构造网络 G 如下。

网络 G 有 $m+n+2$ 个顶点。全部仪器 I_1, I_2, \dots, I_n 和可供选择的实验 E_1, E_2, \dots, E_m 分别对应于 $m+n$ 个顶点。另外增加源 s 和汇 t 。从源 s 到每个顶点 I_i 有一条有向边 (s, I_i) , 其容量为 c_i 。从每个顶点 E_j 到汇 t 有一条有向边 (E_j, t) , 其容量为 p_j 。若 $I_k \in R_j$, 则从顶点 I_k 到

顶点 E_j 有一条有向边 (I_k, E_j) , 其容量为 ∞ 。

(2) 求网络 G 的最大流。设最大流量为 f , 则所求的最大净收益为 $\sum_{j=1}^m p_j - f$ 。

事实上, 网络 G 的任何一个割 $\text{cut}(S, T)$ 对应于一个实验方案。如果顶点 $E_j \in T$, 则任何 $I_k \in R_j$ 有 $I_k \in T$, 否则割的容量将为 ∞ 。因此, 实验 E_j 和 E_j 所需的仪器均在 T 中, T 中的所有实验构成一个实验方案。该实验方案的净收益为

$$\begin{aligned} \sum_{E_j \in T} p_j - \sum_{I_k \in T} c_k &= \sum_{j=1}^m p_j - \sum_{E_j \in S} p_j - \sum_{I_k \in T} c_k \\ &= \sum_{j=1}^m p_j - \text{cap}(S, T) \end{aligned}$$

由此可见, 要使净收益最大, 必须使 $\text{cap}(S, T)$ 最小, 即最优实验方案构成网络 G 的一个最小割。由最大流最小割定理即知, 若求得网络 G 的最大流量 f , 则所求的最大净收益为

$$\sum_{j=1}^m p_j - f$$

算法实现题 8-3 最小路径覆盖问题 (习题 8-13)

★问题描述:

给定有向图 $G=(V, E)$ 。设 P 是 G 的一个简单路 (顶点不相交) 的集合。如果 V 中每个顶点恰好在 P 的一条路上, 则称 P 是 G 的一个路径覆盖。 P 中路径可以从 V 的任何一个顶点开始, 长度也是任意的, 特别地, 可以为 0。 G 的最小路径覆盖是 G 的所含路径条数最少的路径覆盖。

设计一个有效算法求一个有向无环图 G 的最小路径覆盖。

[提示: 设 $V=\{1, 2, \dots, n\}$, 如下构造网络 $G1=(V1, E1)$

$$V1 = \{x_0, x_1, \dots, x_n\} \cup \{y_0, y_1, \dots, y_n\}$$

$$E1 = \{(x_0, x_i) | i \in V\} \cup \{(y_i, y_0) | i \in V\} \cup \{(x_i, y_j) | (i, j) \in E\}$$

(每条边的容量均为 1。)求网络 $G1$ 的 (x_0, y_0) 最大流。]

★编程任务:

对于给定的有向无环图 G , 编程找出 G 的一个最小路径覆盖。

★数据输入:

由文件 input.txt 提供输入数据。文件第 1 行有 2 个正整数 n 和 m 。 n 是给定有向无环图 G 的顶点数, m 是 G 的边数。接下来的 m 行, 每行有 2 个正整数 i 和 j , 表示一条有向边 (i, j) 。

★结果输出:

程序运行结束时, 将最小路径覆盖输出到文件 output.txt。从第 1 行开始, 每行输出一条路径。文件的最后一行是最少路径数。

输入文件示例

input.txt

11 12

1 2

1 3

1 4

输出文件示例

output.txt

1 4 7 10 11

2 5 8

3 6 9

3

2 5
3 6
4 7
5 8
6 9
7 10
8 11
9 11
10 11

分析与解答:

按照提示, 建立网络最大流模型解此问题。

(1) 将有向无环图 G 的每个顶点 i , 拆成两个顶点 x_i 和 y_i 。另外增加源 x_0 和汇 y_0 。构造网络 $G1=(V1,E1)$ 如下:

$$V1=\{x_0,x_1,\cdots,x_n\}\cup\{y_0,y_1,\cdots,y_n\}$$

$$E1=\{(x_0,x_i)|i\in V\}\cup\{(y_i,y_0)|i\in V\}\cup\{(x_i,y_j)|(i,j)\in E\}$$

每条边的容量均为 1。

(2) 求网络 $G1$ 的 (x_0,y_0) 最大流。

(3) 从 $G1$ 的 (x_0,y_0) 最大流构造出 G 的一个最小路径覆盖如下。

当 $\text{flow}(x_0,x_i)=\text{flow}(x_i,y_i)=\text{flow}(y_i,y_0)=1$ 时, 边 (i,j) 属于最小路径覆盖。

当 $\text{flow}(x_0,x_i)=\text{flow}(y_i,y_0)=0$ 时, 顶点 i 是最小路径覆盖中的一条长度为 0 的路径。

网络 $G1$ 的 (x_0,y_0) 最大流量为 f , 则最少路径数为 $n-f$ 。

算法实现题 8-4 魔术球问题(习题 8-14)

★问题描述:

假设有 n 根柱子, 现要按下述规则在这 n 根柱子中依次放入编号为 $1,2,3,\cdots$ 的球。

(1) 每次只能在某根柱子的最上面放球。

(2) 在同一根柱子中, 任何 2 个相邻球的编号之和为完全平方数。

试设计一个算法, 计算出在 n 根柱子上最多能放多少个球。例如, 在 4 根柱子上最多可放 11 个球。

★编程任务:

对于给定的 n , 计算在 n 根柱子上最多能放多少个球。

★数据输入:

由文件 input.txt 提供输入数据。文件第 1 行有 1 个正整数 n , 表示柱子数。

★结果输出:

程序运行结束时, 将 n 根柱子上最多能放的球数以及相应的放置方案输出到文件 output.txt。文件的第 1 行是球数。接下来的 n 行, 每行是一根柱子上的球的编号。

输入文件示例

input.txt

4

输出文件示例

output.txt

11

1 8

2 7 9
3 6 10
4 5 11

分析与解答:

借助有向无环图的最小路径覆盖问题, 并建立网络最大流模型解此问题。

设给定编号为 $1, 2, 3, \dots, n$ 的 n 个球, 建立一个有 n 个顶点的有向无环图 G 如下。每个球对应于图 G 中一个顶点。当 2 个不同的球 i 和 j 满足 $i < j$ 且 $i + j$ 是一个平方数时, G 中有一条边 (i, j) 。图 G 的最小路径覆盖确定了本题中的最少柱子数。

进一步分析可以证明, n 个柱子能放入的最多球数为 $\lfloor (n^2 + 2n - 1) / 2 \rfloor$ 。

由此可知, 魔术球问题等价于有向无环图的最小路径覆盖问题。

算法实现题 8-5 圆桌问题 (习题 8-15)

★问题描述:

假设有来自 n 个不同单位的代表参加一次国际会议。每个单位的代表数分别为 $r_i, i = 1, 2, \dots, n$ 。会议餐厅共有 m 张餐桌, 每张餐桌可容纳 $c_i (i = 1, 2, \dots, m)$ 个代表就餐。为了使代表们充分交流, 希望从同一个单位来的代表不在同一个餐桌就餐。试设计一个算法, 给出满足要求的代表就餐方案。

★编程任务:

对于给定的代表数和餐桌数以及餐桌容量, 编程计算满足要求的代表就餐方案。

★数据输入:

由文件 input.txt 提供输入数据。文件第 1 行有 2 个正整数 m 和 n , m 表示单位数, n 表示餐桌数, $1 \leq m \leq 150, 1 \leq n \leq 270$ 。文件第 2 行有 m 个正整数, 分别表示每个单位的代表数。文件第 3 行有 n 个正整数, 分别表示每个餐桌的容量。

★结果输出:

程序运行结束时, 将代表就餐方案输出到文件 output.txt。如果问题有解, 在文件第 1 行输出 1, 否则输出 0。接下来的 m 行给出每个单位代表的就餐桌号。如果有多个满足要求的方案, 只要输出一个方案。

输入文件示例

input.txt

4 5

4 5 3 5

3 5 2 6 4

输出文件示例

output.txt

1

1 2 4 5

1 2 3 4 5

2 4 5

1 2 3 4 5

分析与解答:

建立网络最大流模型解此问题。

(1) 构造网络 G 如下。

网络 G 有 $m + n + 2$ 个顶点。 n 个不同的单位和 m 张餐桌分别对应于 $m + n$ 个顶点。另外增加源 s 和汇 t 。从源 s 到每个单位顶点 I_i 有一条有向边 (s, I_i) , 其容量为 r_i 。从每个餐桌顶点 E_j 到汇 t 有一条有向边 (E_j, t) , 其容量为 c_j 。从每个单位顶点 I_i 到每个餐桌顶点 E_j 有

一条有向边 (I_i, E_j) ，其容量为1。

(2) 求网络 G 的最大流。设最大流量为 f ，当 $\sum_{j=1}^n r_j = f$ 时有满足要求的代表就餐方案。

算法实现题 8-6 最长递增子序列问题 (习题 8-17)

★问题描述:

给定正整数序列 x_1, x_2, \dots, x_n 。要求

(1) 计算其最长递增子序列的长度 s 。

(2) 设计一个有效算法，计算从给定的序列中最多可取出多少个长度为 s 的递增子序列。

(3) 如果允许在取出的序列中多次使用 x_1 和 x_n ，则从给定序列中最多可取出多少个长度为 s 的递增子序列。

★编程任务:

设计有效算法完成(1)，(2)，(3)提出的计算任务。

★数据输入:

由文件 input.txt 提供输入数据。文件第1行有1个正整数 n ，表示给定序列的长度。接下来的1行有 n 个正整数 x_1, x_2, \dots, x_n 。

★结果输出:

程序运行结束时，将任务(1)，(2)，(3)的解答输出到文件 output.txt 中。第1行是最长递增子序列的长度 s 。第2行是可取出的长度为 s 的递增子序列个数。第3行是允许在取出的序列中多次使用 x_1 和 x_n 时可取出的长度为 s 的递增子序列个数。

输入文件示例	输出文件示例
input.txt	output.txt
4	2
3 6 2 5	2
	3

分析与解答:

先用动态规划算法计算出以 x_i 开头的最长递增子序列的长度 $b[i]$ 。然后构造网络 G 如下。按照 $b[i]$ 的大小将 x_i 分层， $b[i]=k$ 的数对应于第 k 层顶点。当 x_i 是第 k 层顶点， x_j 是第 $k-1$ 层顶点，且 $x_i < x_j$ 时， x_i 和 x_j 所相应的顶点之间增加一条有向边。由于每个数只能取1次，所以顶点的容量约束为1。如果允许在取出的序列中多次使用 x_1 和 x_n ，则 x_1 和 x_n 所相应的顶点没有顶点的容量约束。求相应网络的最大流可得最多的最长递增子序列。

具体算法用类 LIS 实现如下。

```
class LIS {
public:
    LIS();
    ~LIS();
    void input(char *filename);
    void compute(int tt);
private:
    ofstream outFile;
```

```

        int n,*a,*b,**c;
        int maxL(int n);
        int dynb();
};

```

其中,input 读入初始数据。

```

void LIS::input(char *filename)
{
    ifstream inFile;
    inFile.open(filename);
    inFile>>n;
    a=new int[n];
    b=new int[n+1];
    Make2DArray(c,n+1,n+1);
    for(int i=0;i<n;i++) inFile>>a[i];
    inFile.close();
}

```

dynb 用动态规划算法计算出以 x_i 开头的最长递增子序列的长度 $b[i]$ 。

```

int LIS::dynb()
{
    int i,j,k;
    for(i=n-2,b[n-1]=1;i>=0;i--){
        for(j=n-1,k=0;j>i;j--){
            if(a[j]>=a[i] && k<b[j]) k=b[j];
            b[i]=k+1;
        }
        for(i=0;i<=n;i++) c[i][j]=0;
        for(i=0;i<n;i++){c[b[i]][0]++;c[b[i]][c[b[i]][0]]=i;}
    }
    return maxL(n);
}

```

maxL 计算最长递增子序列的长度 s 。

```

int LIS::maxL(int n)
{
    for(int i=0,temp=0;i<n;i++){
        if(b[i]>temp) temp=b[i];
    }
    return temp;
}

```

compute 构造相应网络 G , 并计算最大流。

```
void LIS::compute(int tt)
{
    GRAPH<EDGE>G(2*n+2, 1);
    int s=0, t=2*n+1, maxflow=0;
    int len=dynb();
    if(tt==0)cout<<len<<endl;
    for(int i=len; i>1; i--)
        for(int j=1, kj=c[i][0]; j<=kj; j++)
            for(int k=1, kk=c[i-1][0]; k<=kk; k++)
                if(c[i][j]<c[i-1][k] && a[c[i][j]]<=a[c[i-1][k]])
                    G.insert(new EDGE(2*c[i][j]+2, 2*c[i-1][k]+1, 1));

    int sf=1, tf=1;
    if(tt>0){sf=n; tf=n;}
    for(i=1; i<=c[len][0]; i++)G.insert(new EDGE(s, 2*c[len][i]+1, sf));
    for(i=1; i<=c[1][0]; i++)G.insert(new EDGE(2*c[1][i]+2, t, 1));
    for(i=2; i<=n; i++)G.insert(new EDGE(2*i-1, 2*i, 1));
    G.insert(new EDGE(1, 2, sf));
    G.insert(new EDGE(2*n-1, 2*n, tf));
    G.insert(new EDGE(2*n, t, tf));
    MAXFLOW<GRAPH<EDGE>, EDGE>(G, s, t, maxflow);
    cout<<maxflow<<endl;
}
```

实现算法的主函数如下。

```
int main()
{
    char *filein;
    filein="alis.in";
    LIS X;
    X.input(filein);
    X.compute(0);
    X.compute(1);
    return 0;
}
```

算法实现题 8-7 试题库问题(习题 8-18)

★问题描述:

假设一个试题库中有 n 道试题。每道试题都标明了所属类别。同一道题可能有多个类别属性。现要从题库中抽取 m (如 $m=100$) 道题组成试卷。并要求试卷包含指定类型 (如 20 类不同类型) 的试题。试设计一个满足要求的组卷算法。

★编程任务:

对于给定的组卷要求, 计算满足要求的组卷方案。

★数据输入:

由文件 input.txt 提供输入数据。文件第 1 行有 2 个正整数 k 和 n ($2 \leq k \leq 20, k \leq n \leq 1000$), k 表示题库中试题类型总数, n 表示题库中试题总数。第 2 行有 k 个正整数, 第 i 个正整数表示要选出的类型 i 的题数。这 k 个数相加就是要选出的总题数 m 。接下来的 n 行给出了题库中每个试题的类型信息。每行的第 1 个正整数 p 表明该题可以属于 p 类, 接着的 p 个数是该题所属的类型号。

★结果输出:

程序运行结束时, 将组卷方案输出到文件 output.txt。文件第 i 行输出 “ i :” 后接类型 i 的题号。如果有多个满足要求的方案, 只要输出 1 个方案。如果问题无解, 则输出 “No Solution!”。

输入文件示例

input.txt

3 15

3 3 4

2 1 2

1 3

1 3

1 3

1 3

3 1 2 3

2 2 3

2 1 3

1 2

1 2

2 1 2

2 1 3

2 1 2

1 1

3 1 2 3

输出文件示例

output.txt

1: 1 6 8

2: 7 9 10

3: 2 3 4 5

分析与解答:

建立网络最大流模型解此问题。

(1) 构造网络 G 如下。

网络 G 有 $k+n+2$ 个顶点。每道试题对应于一个顶点 u_i ; 每个试题类型也对应于一个顶点 v_i 。另外增加源 s 和汇 t 。设要选出类型 i 的题数为 k_i 。从源 s 到每个试题类型顶点 v_i 有一条有向边 (s, v_i) , 其容量为 k_i 。从每个试题顶点 u_i 到汇 t 有一条有向边 (u_i, t) , 其容量为 1。从每个试题类型顶点 v_i 到每个试题顶点 u_j 有一条容量为 1 的有向边 (v_i, u_j) 的前提条件是试题 j 属于试题类型 i 。

(2) 求网络 G 的最大流。设最大流量为 f , 当 $\sum_{j=1}^k k_j = f$ 时有满足要求的组卷方案。

算法实现题 8-8 机器人路径规划问题 (习题 8-19)

★问题描述:

机器人 (Rob) 可在一个树状路径上自由移动。给定 (树状路径 T 上的) 起点 s 和终点 t , 机器人要从 s 运动到 t 。树状路径 (T) 上有若干可移动的障碍物。由于路径狭窄, 任何时刻在路径的任何位置不能同时容纳 2 个物体。每一步可以将障碍物或机器人移到相邻的空顶点上。设计一个有效算法用最少移动次数使机器人从 s 运动到 t 。

★编程任务:

对于给定的树 T , 以及障碍物在树 T 中的分布情况。计算机器人从起点 s 到终点 t 的最少移动次数。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行有 3 个正整数 n , s 和 t , 分别表示树 T 的顶点数, 起点 s 的编号和终点 t 的编号。

接下来的 n 行分别对应于树 T 中编号为 $0, 1, \dots, n-1$ 的顶点。每行的第 1 个整数 h 表示顶点的初始状态, 当 $h=1$ 时表示该顶点为空顶点, 当 $h=0$ 时表示该顶点为满顶点, 其中已有一个障碍物。第 2 个数 k 表示有 k 个顶点与该顶点相连。接下来的 k 个数是与该顶点相连的顶点编号。

★结果输出:

程序运行结束时, 将计算出的机器人最少移动次数输出到文件 output.txt。如果无法将机器人从起点 s 移动到终点 t , 输出 “No Solution!”。

输入文件示例	输出文件示例
input.txt	output.txt
5 0 3	3
1 1 2	
1 1 2	
1 3 0 1 3	
0 2 2 4	
1 1 3	

分析与解答:

用最小费用流算法求解。

根据给定条件构造网络 G 之前, 需要对问题进行较深入的分析。

对于给定的树 T , 设初始时已有障碍物的顶点集合为 O 。机器人移动的起点为 s , 终点为 t 。二元组 $\text{conf} = (O, s)$ 构成机器人路径规划问题在给定树 T 上的一个布局; 三元组 $S = (O, s, t)$ 构成机器人路径规划问题对于给定树 T 的一个实例。

设给定的树 T 中有 n 个顶点, 从 s 到 t 的路径为 $P = (v_0, v_1, \dots, v_d)$, 起点为 $s = v_0$, 终点为 $t = v_d$, 路长为 d 。路径 P 称为机器人路径规划问题中的关键路径。除了起点 s 外, 路径 P 上有分叉的顶点称为路径 P 上的分叉顶点。对于路径 P 上每个顶点 v_j , 将 v_j 的所有分支中的顶点记为 B_j 。如果 B_j 非空, 称 B_j 中与顶点 v_j 相邻的顶点为顶点 v_j 的旁路顶点。

容易看出, P, B_0, B_1, \dots, B_d 构成树 T 中顶点的一个划分。

另外, 对于路径 P 上每个顶点 v_j 和每个分叉顶点 $v_r, 0 < r < j$, 定义顶点集合 $\text{Ch}(r, j)$ 如下:

$$\text{Ch}(r, j) = \{v_h \mid r \leq h < j\}; \text{Ch}(0, j) = B_0 \cup \{v_h \mid r \leq h < j\}$$

对于树 T 中的顶点对 (u, w) , 定义偏序关系 “ $<$ ” 如下:

$$u < w \Leftrightarrow \begin{cases} u = v_i, w = v_j & i < j \\ u = v_i, w \in B_j & i < j \\ u \in B_i, w = v_j & i \leq j \\ u \in B_i, w \in B_j & i < j \\ u, w \in B_0, w \in p(u, s) \end{cases}$$

设 A 是机器人从起点 s 到终点 t 的一个移动方案。依据关键路径上的每个顶点 $v_j, 0 \leq j \leq d$, 将 A 中的移动划分为左半部移动 $L(A, j)$ 和右半部移动 $R(A, j)$ 如下。

(1) 机器人移动 $u \rightarrow w$:

当 $w \leq v_j$ 时, $u \rightarrow w \in L(A, j)$, 否则 $u \rightarrow w \in R(A, j)$ 。

(2) 障碍物外移 $u \rightarrow w, w \in B_i$:

当 $v_i \leq v_j$ 时, $u \rightarrow w \in L(A, j)$;

当 $v_j < v_i$ 且 $v_j < u$ 时, $u \rightarrow w \in R(A, j)$;

当 $u < v_j < v_i$ 时, $u \rightarrow v_j \in L(A, j), v_j \rightarrow w \in R(A, j)$ 。

(3) 障碍物后移 $u \rightarrow w$, 且机器人位于顶点 v_i 的旁路顶点中:

设 v_r 是机器人到达顶点 v_j 之前进入其旁路顶点的分叉顶点。如果机器人到达顶点 v_j 之前未进入任何旁路顶点, 则 $v_r = s$ 。

当 $w < v_r$ 时, $u \rightarrow w \in L(A, j)$; 否则 $u \rightarrow w \in R(A, j)$ 。

(4) 障碍物前移 $u \rightarrow w$:

设 v_r 是机器人到达顶点 v_j 之前进入其旁路顶点的分叉顶点。如果机器人到达顶点 v_j 之前未进入任何旁路顶点, 则 $v_r = s$ 。

设 l 是满足下面条件的最大整数: $j \leq l, v_l \rightarrow z$ 是一个以 v_l 为初始起点的移动, 且 $z < v_r$, 或 $z \in B_h, h \leq j$ 。

当 $w \leq v_l$ (从而 $u \leq v_l$) 时, $u \rightarrow w \in L(A, j)$;

当 $v_l < u$ (从而 $v_l < w$) 时, $u \rightarrow w \in R(A, j)$;

当 $u < v_j$ 且 $v_l < w$ 时, $u \rightarrow v_j \in L(A, j), v_j \rightarrow w \in R(A, j)$ 。

对于机器人从起点 s 到终点 t 的所有移动方案 A , 将左半部移动 $L(A, j)$ 依 5 个参数分类为 $C(j, k, l, f, r)$ 如下。

(1) 参数 j 。 $L(A, j)$ 是关于顶点 v_j 的左半部移动。

(2) 参数 k 。有 k 个障碍物移动到顶点 v_j 。

(3) 参数 l 。 l 是满足下面条件的最大整数: $j \leq l, v_l \rightarrow z$ 是一个以 v_l 为初始起点的移动, 且 $z < v_r$, 或 $z \in B_h, h \leq j$ 。如果没有这样的移动, 则 $l = j$ 。

(4) 参数 f 。 f 是满足下面条件的最大整数: $j < f \leq l, u \rightarrow v_f$ 是 $L(A, j)$ 的一个障碍物前移。如果没有这样的障碍物前移, 则 $f = j$ 。

(5) 参数 r 。 r 是满足下面条件的最大整数: $0 < r < j, v_r$ 是机器人到达顶点 v_j 之前进入其旁路顶点的分叉顶点。如果机器人到达顶点 v_j 之前未进入任何旁路顶点, 则 $r = 0$ 。

$C(j, k, l, f, r)$ 可以看作将机器人从 s 移动到 v_j , 但在顶点 v_j 处还有 k 个障碍物待确定终点。这 k 个障碍物称为悬挂障碍物, 相应的移动称为悬挂移动。若一个悬挂障碍物的起点为 u , 则接着的移动只有两种方式:

- (1) 障碍物外移 $u \rightarrow w, w \in B_i, i > j$ 。
- (2) 先障碍物前移 $u \rightarrow v_z, z > l$, 然后障碍物后移 $v_z \rightarrow w, v_z \leq w$ 。

设 $L_1 \in C(j, k_1, l_1, f_1, r_1), L_2 \in C(j+1, k_2, l_2, f_2, r_2)$, 定义它们的差 $M = \Delta(L_1, L_2)$ 如下:

对于每个移动 $u \rightarrow v \in L_2$, 且移动 $u \rightarrow v$ 分解为 $u \rightarrow v'$ 和 $v' \rightarrow v, u \rightarrow v' \in L_1$, 则 $v' \rightarrow v \in M$; 否则 $u \rightarrow v \in M$ 。 M 也称为 L_1 到 L_2 的一个扩展。

M 中的移动只有下面 4 种类型:

- (1) 机器人移动 $v_j \rightarrow v_{j+1}$;
- (2) 障碍物前移 $v_j \rightarrow v_{j+1}$;
- (3) 障碍物外移 $u \rightarrow w, w \in B_{j+1}$;

(4) 如果 v_j 是一个分叉顶点, 则 M 可能包含机器人从 v_j 到其旁路顶点, 然后从旁路顶点返回的移动。

① 如果发生上述移动, 则 $r_2 = j$, M 包含所有机器人在 v_j 的旁路顶点中时的障碍物后移 $u \rightarrow w, v_{l_1+1} \leq u \leq v_{l_2}, v_r \leq w < v_j$;

② 如果顶点序列 $v_{f_1+1}, \dots, v_{l_1}$ 中没有空顶点, 则对 v_{l_1} 和 v_{f_2} 中的每个空顶点 w , M 中包含移动 $v_j \rightarrow w$ 。

如果 M 中含有 h_1 个障碍物外移, h_2 个障碍物前移和 h_3 个障碍物后移, 则称 M 是 L_1 到 L_2 的一个 (h_1, h_2, h_3) 扩展。

如果 M 是 L_1 到 L_2 的一个 (h_1, h_2, h_3) 扩展, 则有

- (1) 如果存在 $f_1 < m \leq l_1$, 使 v_m 是一个空顶点, 则 $h_2 = 0$;
- (2) 设 v_m 是最靠近 v_{f_1} 的空顶点, 如果 v_{j+1} 是 v_j 和 v_m 间惟一分叉顶点, 则 $h_1 = k$;
- (3) 如果 $j+1 \leq f_1$, 则所有障碍物外移均为悬挂移动。

(4) 如果 $h_2 > 0$ 且 v_f 是 v_{l_1+1}, \dots, v_d 中的第 h_2 个空顶点, 则在关键路径 P 上顶点 v_{l_1} 和 v_f 之间最多有 $h_3 - h_2$ 个 O 中的顶点。

对于机器人路径规划问题的一个实例 S , 构造网络 $G(S)$ 如下。

(1) 每个非空移动类 $C(j, k, l, f, r)$ 对应于网络 $G(S)$ 中一个顶点 $w(j, k, l, f, r)$ 。顶点 $w(j, k, l, f, r)$ 到顶点 $w(j', k', l', f', r')$ 之间有一条有向边, 当且仅当 $C(j', k', l', f', r')$ 是 $C(j, k, l, f, r)$ 的一个 (h_1, h_2, h_3) 扩展, 该有向边的容量为 1, 费用为最小 (h_1, h_2, h_3) 扩展费用。

(2) 另外增加源顶点 w_s 和汇顶点 w_t 。从源顶点 w_s 到顶点 $w(0, 0, 0, 0, 0)$ 之间有一条有向边, 容量为 1, 费用为 0。所有顶点 $w(d, 0, d, d, r)$ 到汇顶点 w_t 之间有一条有向边, 容量为 1, 费用为 0。

网络 $G(S)$ 的最小费用流对应于 S 的一个解。

下面讨论如何计算 $C(j, k, l, f, r)$ 的最小 (h_1, h_2, h_3) 扩展。

设 $\text{out}(j, k, l, f, h)$ 是在 $C(j, k, l, f, r)$ 的扩展中执行 h 个障碍物外移所需的最少移动次数。 $\text{back}(j, k, l, f, r, h_1, h_2)$ 是在 $C(j, k, l, f, r)$ 的扩展中执行 h_1 个障碍物前移和 h_2 个障碍物后移所需的最少移动次数。

上述两个函数除了返回所需的最少移动次数外, 还计算出 k^*, f^* 和 l^* 的值。 k^* 是执

行移动后剩余的悬挂障碍物数； f^* 对应于悬挂障碍物最远前移位置 v_{f^*} ； l^* 是障碍物从 v_{l^*} 移动到 $u < v_{j-1}$ 的最大下标。

计算上述两个函数要用到障碍物在 S 中的位置信息，这可以在 $O(n)$ 时间内预计算。

$\text{out}(j, k, l, f, h)$ 可按照如下方式计算。初始时 $\text{out}(j, k, l, f, 0) = 0$ 。 $\text{out}(j, k, l, f, h+1)$ 的值可以用 $\text{out}(j, k, l, f, h)$ 的值，以及障碍物到 B_{j+1} 中第 $h+1$ 近的空顶点的距离来计算。

由此可见，对所有可能的 h_1 ， $\text{out}(j, k, l, f, h_1)$ 可以在 $O(\min\{d, |B_{j+1}|\})$ 时间内完成。

类似地，对所有可能的 h_2 和 h_3 ， $\text{back}(j, k, l, f, r, h_2, h_3)$ 可以在 $O((\min\{d, |\text{Ch}(r, j)|\})^2)$ 时间内完成。

有了这两个函数，容易计算 $C(j, k, l, f, r)$ 的最小 (h_1, h_2, h_3) 扩展如下。

```
Extend( $j, k, l, f, r, h_1, h_2, h_3$ )
{
    ( $c_{h_1}, k_{h_1}, l_{h_1}, f_{h_1}$ )  $\leftarrow$   $\text{out}(j, k, l, f, h_1)$ ;
    ( $c_{h_2, h_3}, k_{h_2, h_3}, l_{h_2, h_3}, f_{h_2, h_3}$ )  $\leftarrow$   $\text{back}(j, k_{h_1}, l_{h_1}, f_{h_1}, r, h_2, h_3)$ ;
     $c = c_{h_1} + c_{h_2, h_3} + k - k_{h_2, h_3} + 1$ ;
    if ( $h_3 > 0$ )  $r_{h_2, h_3} = j$  else  $r_{h_2, h_3} = r$ ;
    return  $c$ ;
}
```

由此得 $C(j+1, k_{h_2, h_3}, l_{h_2, h_3}, f_{h_2, h_3}, r_{h_2, h_3})$ 。

算法实现题 8-9 方格取数问题 (习题 8-20)

★问题描述:

在一个有 $m \times n$ 个方格的棋盘中，每个方格中有一个正整数。现要从方格中取数，使任意 2 个数所在方格没有公共边，且取出的数的总和最大。试设计一个满足要求的取数算法。

★编程任务:

对于给定的方格棋盘，按照取数要求编程找出总和最大的数。

★数据输入:

由文件 input.txt 提供输入数据。文件第 1 行有 2 个正整数 m 和 n ，分别表示棋盘的行数和列数。接下来的 m 行，每行有 n 个正整数，表示棋盘方格中的数。

★结果输出:

程序运行结束时，将取数的最大总和输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3 3	11
1 2 3	
3 2 3	
2 3 1	

分析与解答:

(1) 解法 1: 用线性规划算法求解

设第 i 行第 j 列中的正整数为 c_{ij} ，相应的变量为 x_{ij} 。变量 x_{ij} 的含义是取了 x_{ij} 个 (i, j) 方格中放置的正整数 c_{ij} ， $x_{ij} \in \{0, 1\}$ 。

方格取数问题的求解目标是 $\sum_{i=1}^m \sum_{j=1}^n c_{ij} x_{ij}$ 达到最大。

约束条件是, 取出的任意 2 个数所在方格没有公共边。

一般情况下, 有 4 个方格与方格 (i, j) 有公共边。因此, 一般情况下方格 (i, j) 对应于下面 4 个不等式约束:

$$x_{ij} + x_{i-1,j} \leq 1; x_{ij} + x_{i+1,j} \leq 1; x_{ij} + x_{i,j-1} \leq 1; x_{ij} + x_{i,j+1} \leq 1$$

由此可见, 方格取数问题可以变换为如下的整数线性规划问题:

$$\max \sum_{i=1}^m \sum_{j=1}^n c_{ij} x_{ij}$$

s. t.

$$x_{ij} + x_{i-1,j} \leq 1$$

$$x_{ij} + x_{i+1,j} \leq 1$$

$$x_{ij} + x_{i,j-1} \leq 1$$

$$x_{ij} + x_{i,j+1} \leq 1$$

$$x_{ij} \in \{0, 1\}$$

如果将相邻方格用黑白 2 种颜色着色, 每个方格对应于图 G 的一个顶点, 相邻方格顶点之间连一条边, 则所得到的图 G 是一个二分图。上述整数线性规划的约束矩阵恰好是二分图 G 的关联矩阵。由已知结论, 任何一个二分图的关联矩阵是一个全 1 模阵, 可知上述整数线性规划的约束矩阵是一个全 1 模阵。因此可以将非线性约束条件 $x_{ij} \in \{0, 1\}$ 松弛为线性约束条件 $x_{ij} \geq 0$ 。从而将整数线性规划问题转化为线性规划问题, 可用单纯形算法求解。

具体算法实现如下。

read 读入初始数据。

```
void read()
{
    cin >> m >> n;
    Make2DArray(c, m, n);
    for (int i=0; i<m; i++)
        for (int j=0; j<n; j++) cin >> c[i][j];
}
```

用线性规划算法的成员函数 lpin 建立相应的初始单纯形表。

```
void LinearProgram::lpin(int mima, int mm, int nn, int**cc)
{
    int i, j, k, t;
    double value;
    minmax = mima; m = (mm - 1) * nn + (nn - 1) * mm, n = mm * nn;
    m1 = m; m2 = 0; m3 = 0; n1 = n; n2 = n + m1;
    Make2DArray(a, m + 2, n1 + 1);
    basic = new int[m + 2];
    nonbasic = new int[n1 + 1];
}
```

```

for (i=0;i<=m+1;i++)
    for (j=0;j<=n1;j++) a[i][j]=0.0;
for (j=0;j<=n1;j++) nonbasic[j]=j;
for (i=1, j=n1+1; i<=m; i++, j++) basic[i]=j;
for (i=m-m3+1, j=n+1; i<=m; i++, j++){
    a[i][j]=-1.0; a[m+1][j]=-1.0;
}
for(k=1,t=1;k<=mm;k++)
    for(i=1;i<=nn;i++,t++){
        j=(k-1)*nn+i; a[t][j]=1.0; a[t][j+1]=1.0;
    }
for(k=1;k<=mm;k++)
    for(i=1;i<=nn;i++,t++){
        j=(k-1)*nn-i; a[t][j]=1.0; a[t][j+nn]=1.0;
    }
for(j=1;j<=m;j++) a[j][0]=1;
for(i=0;i<=mm;i++){
    for(j=0;j<=nn;j++){
        int jj=i*nn+j+1;
        a[0][jj]=minmax*cc[i][j];
    }
for (j=1;j<=n;j++){
    for (i=m1+1, value=0.0; i<=m;i++) value+=a[i][j];
    a[m+1][j]=value;
}
}
}

```

算法的主函数调用单纯形算法求解。

```

int main()
{
    read();
    LinearProgram X;
    X.lpin(1,m,n,c);
    X.solve();
    return 0;
}

```

(2) 解法 2: 用最大流算法求解

将相邻方格用黑白 2 种颜色着色, 每个方格对应于图 G 的一个顶点, 相邻方格顶点之间连一条边, 则所得到的图 G 是一个二分图。接着将每个方格顶点按照方格中的正整数 x 拆成 x 个顶点。原来连接方格顶点 x 和 y 的边增加为 $x \times y$ 条边。由此构成的图 G' 仍是一个二分图。所求的最大取数方案对应于该二分图的一个最大独立集。

设图 $G=(V,E)$ 的最大独立集中顶点数为 $\alpha(G)$;

图 G 的最大匹配中边数为 $\alpha'(G)$;

图 G 的最小顶点覆盖中顶点数为 $\beta(G)$;

图 G 的最小边覆盖中边数为 $\beta'(G)$; 则有

(1) 若 $S \subseteq V$ 是图 G 的独立集, 则 $V-S$ 是图 G 的顶点覆盖。因此, $\alpha(G) + \beta(G) = n$ 。

(2) 若图 G 无孤立顶点, 则 $\alpha'(G) + \beta'(G) = n$ 。

(3) 若图 G 无孤立顶点的二分图, 则 $\alpha'(G) = \beta'(G)$, 从而 $\alpha(G) = n - \alpha'(G)$ 。

由此可见, 所求问题可以转化为二分图的最大匹配问题。

另一方面, 容易看出, 拆点后二分图 G' 的最大匹配对应于未拆点前二分图 G 的最大流。因此方格取数问题又转化为二分图 G 的最大流问题。

具体算法实现如下。

input 读入初始数据。

```
void GRID::input()
{
    cin >> m >> n;
    Make2DArray(c, m+1, n+1);
    Make2DArray(d, m+1, n+1);
    Make2DArray(e, m+1, n+1);
    for (i=1; i<=m; i++)
        for (int j=1; j<=n; j++) cin >> c[i][j];
    d[0][0]=0;
    for (int j=1; j<=m; j++) d[j][0]=!d[j-1][0];
    for (i=1; i<=m; i++)
        for (j=1; j<=n; j++) d[i][j]=!d[i][j-1];
    for (i=1, nn=0; i<=m; i++)
        for (j=1; j<=n; j++) nn+=c[i][j];
}
```

comp 构造相应的二分图 G , 并通过计算网络最大流找出最大取数方案。

```
int GRID::comp()
{
    int i, j, s, t, maxflow=0;
    GRAPH<EDGE> G(m*n+2, 1);
    for (i=1; i<=m; i++)
        for (j=1; j<=n; j++) {
            if (!d[i][j]) adde(G, i, j, i, j+1);
            else adde(G, i, j+1, i, j);
        }
    for (j=1; j<=n; j++)
        for (i=1; i<=m; i++) {
            if (!d[i][j]) adde(G, i, j, i+1, j);
            else adde(G, i+1, j, i, j);
        }
}
```

```

    }
    s=m*n;t=m*n+1;
    for (i=1; i<=m; i++)
        for(j=1;j<=n;j++){
            int x=(i-1)*n+j-1;
            if(d[i][j]) G.insert(new EDGE(x,t,c[i][j]));
            else G.insert(new EDGE(s,x,c[i][j]));
        }
    MAXFLOW<GRAPH<EDGE>, EDGE>(G,s,t,maxflow);
    return nn-maxflow;
}

void GRID::adde(GRAPH<EDGE>&G,int x1,int y1,int x2,int y2)
{
    int i=(x1-1)*n+y1-1,j=(x2-1)*n+y2-1;
    G.insert(new EDGE(i,j,c[x1][y1]));
}

```

算法实现题 8-10 餐巾计划问题(习题 8-21)

★问题描述:

一个餐厅在相继的 N 天里, 每天需用的餐巾数不尽相同。假设第 i 天需要 r_i 块餐巾 ($i=1,2,\dots,N$)。餐厅可以购买新的餐巾, 每块餐巾的费用为 p 分; 或者把旧餐巾送到快洗部, 洗一块需 m 天, 其费用为 f 分; 或者送到慢洗部, 洗一块需 n 天 ($n>m$), 其费用为 $s<f$ 分。每天结束时, 餐厅必须决定将多少块脏的餐巾送到快洗部, 多少块餐巾送到慢洗部, 以及多少块保存起来延期送洗。但是每天洗好的餐巾和购买的新餐巾数之和要满足当天的需求量。试设计一个算法为餐厅合理地安排好 N 天中餐巾使用计划, 使总的花费最小。

★编程任务:

编程找出一个最佳餐巾使用计划。

★数据输入:

由文件 input.txt 提供输入数据。文件第 1 行有 6 个正整数 N, p, m, f, n, s 。 N 是要安排餐巾使用计划的天数; p 是每块新餐巾的费用; m 是快洗部洗一块餐巾需用天数; f 是快洗部洗一块餐巾需要的费用; n 是慢洗部洗一块餐巾需用天数; s 是慢洗部洗一块餐巾需要的费用。接下来的 N 行是餐厅在相继的 N 天里, 每天需用的餐巾数。

★结果输出:

程序运行结束时, 将餐厅在相继的 N 天里使用餐巾的最小总花费输出到文件 output.txt。

输入文件示例

输出文件示例

input.txt

output.txt

3 10 2 3 3 2

145

5

6

7

分析与解答:

用最小费用流算法求解。

对于给定条件构造网络 G 如下。每天对应于图 G 的 2 个顶点 X_i 和 Y_i 。

(1) 顶点 X_i 和 X_{i+1} 之间有一条有向边, 容量为 ∞ , 费用为 0。表示第 i 天可以保存旧餐巾到第 $i+1$ 天, 延期送洗。

(2) 当 $i+m \leq N$ 时, 顶点 X_i 和 Y_{i+m} 之间有一条有向边, 容量为 ∞ , 费用为 f 。表示第 i 天可以把旧餐巾送到快洗部。

(3) 当 $i+n \leq N$ 时, 顶点 X_i 和 Y_{i+n} 之间有一条有向边, 容量为 ∞ , 费用为 s 。表示第 i 天可以把旧餐巾送到慢洗部。

另外增加源 s 和汇 t , 以及附加顶点 k 。

(4) 从源 s 到每个顶点 X_i 有一条有向边, 其容量为第 i 天餐巾需求量 r_i , 费用为 0。

(5) 从附加顶点 k 到每个顶点 Y_i 有一条有向边, 其容量为 ∞ , 费用为 p , 表示第 i 天可以购买新餐巾。

(6) 从每个顶点 Y_i 到汇 t 有一条有向边, 其容量为第 i 天餐巾需求量 r_i , 费用为 0。

求网络 G 的最小费用流。

算法实现题 8-11 航空路线问题 (习题 8-24)

★问题描述:

给定一张航空图, 图中顶点代表城市, 边代表 2 个城市间的直通航线。现要求找出一条满足下述限制条件的且途经城市最多的旅行路线:

(1) 从最西端城市出发, 单向从西向东途经若干城市到达最东端城市, 然后再单向从东向西飞回起点 (可途经若干城市)。

(2) 除起点城市外, 任何城市只能访问 1 次。

★编程任务:

对于给定的航空图, 试设计一个算法找出一条满足要求的最佳航空旅行路线。

★数据输入:

由文件 input.txt 提供输入数据。文件第 1 行有 2 个正整数 N 和 V , N 表示城市数, $N < 100$, V 表示直飞航线数。接下来的 N 行中每一行是一个城市名, 可乘飞机访问这些城市。城市名出现的顺序是从西向东。也就是说, 设 i, j 是城市表列中城市出现的顺序, 当 $i > j$ 时, 表示城市 i 在城市 j 的东边, 而且不会有 2 个城市在同一条经线上。城市名是一个长度不超过 15 的字符串, 串中的字符可以是字母或阿拉伯数字。例如, AGR34 或 BEI4。

再接下来的 V 行中, 每行有 2 个城市名, 中间用空格隔开, 如 city1 city2 表示 city1 到 city2 有一条直通航线, 从 city2 到 city1 也有一条直通航线。

★结果输出:

程序运行结束时, 将最佳航空旅行路线输出到文件 output.txt 中。文件第 1 行是旅行路线中所访问的城市总数 M 。接下来的 $M+1$ 行是旅行路线的城市名, 每行写 1 个城市名。首先是起点城市名, 然后按访问顺序列出其他城市名。注意, 最后一行 (终点城市) 的城市名必然是起点城市名。如果问题无解, 则输出 “No Solution!”。

输入文件示例	输出文件示例
input. txt	output. txt
8 9	7
Vancouver	Vancouver
Yellowknife	Edmonton
Edmonton	Montreal
Calgary	Halifax
Winnipeg	Toronto
Toronto	Winnipeg
Montreal	Calgary
Halifax	Vancouver
Vancouver Edmonton	
Vancouver Calgary	
Calgary Winnipeg	
Winnipeg Toronto	
Toronto Halifax	
Montreal Halifax	
Edmonton Montreal	
Edmonton Yellowknife	
Edmonton Calgary	

分析与解答:

对于给定的航空图构造网络 G 如下。

(1) 每个城市 i 对应于图 G 的 2 个顶点 i 和 i' ，并在 2 个顶点之间连接一条由 i 至 i' 的有向边，边的容量为 1，表示该市最多只能被访问一次。为了使该城市尽可能被访问，单位流量费用设为 0。

(2) 若城市 i 到城市 j 有直通航线 ($i < j$)，则在顶点 i' 与顶点 j 之间连接一条边，方向由顶点 i' 至顶点 j 。边的容量为 1，表示这条航线最多只能通过一次。单位流量费用设为 $j-i+1$ ，即城市 j 位于城市 i 的东面，中间间隔的城市数为 $j-i+1$ 。

(3) 顶点 1 与顶点 2 之间的边容量改为 2，表示最西端的城市 1 被访问 2 次。顶点 n 与顶点 n' 的边容量也改为 2。因为如果将往返航线看作 2 条不同路线，则最东端的城市 n 可以看作被访问 2 次。

(4) 顶点 1 作为源，顶点 n' 作为汇。

然后对上述网络求最小费用最大流，其结果是：一条由顶点 1 至顶点 n ，边容量为 1 且互相连接的若干条边组成的流，表示往程航线；另一条由顶点 n 至顶点 1，边容量为 1 且互相连接的若干条边组成的流，表示返程航线。这两条流上的 (顶点序号+1)/2 即为往返航线上被访问的城市序号。显然最佳航空路线上经过的城市数 $= 2 \times (n-1) - \text{流量费用}$ 。

算法实现题 8-12 软件补丁问题 (习题 8-25)

★问题描述:

T 公司发现其研制的一个软件中有 n 个错误，随即为该软件发放了一批共 m 个补丁程

序。每一个补丁程序都有其特定的适用环境，某个补丁只有在软件中包含某些错误而同时又不包含另一些错误时才可以使用。一个补丁在排除某些错误的同时，往往会加入另一些错误。换句话说，对于每一个补丁 i ，都有 2 个与之相应的错误集合 $B1[i]$ 和 $B2[i]$ ，使得仅当软件包含 $B1[i]$ 中的所有错误，而不包含 $B2[i]$ 中的任何错误时，才可以使用补丁 i 。补丁 i 将修复软件中的某些错误 $F1[i]$ ，而同时加入另一些错误 $F2[i]$ 。另外，每个补丁都耗费一定的时间。

试设计一个算法，利用 T 公司提供的 m 个补丁程序将原软件修复成一个没有错误的软件，并使修复后的软件耗时最少。

★编程任务：

对于给定的 n 个错误和 m 个补丁程序，找到总耗时最少的软件修复方案。

★数据输入：

由文件 input.txt 提供输入数据。文件第 1 行有 2 个正整数 n 和 m ， n 表示错误总数， m 表示补丁总数， $1 \leq n \leq 20$ ， $1 \leq m \leq 100$ 。接下来 m 行给出了 m 个补丁的信息。每行包括一个正整数，表示运行补丁程序 i 所需时间，以及 2 个长度为 n 的字符串，中间用一个空格符隔开。在第 1 个字符串中，如果第 k 个字符 b_k 为“+”，则表示第 k 个错误属于 $B1[i]$ ，若为“-”，则表示第 k 个错误属于 $B2[i]$ ，若为“0”，则第 k 个错误既不属于 $B1[i]$ 也不属于 $B2[i]$ ，即软件中是否包含第 k 个错误并不影响补丁 i 的可用性。在第 2 个字符串中，如果第 k 个字符 b_k 为“+”，则表示第 k 个错误属于 $F1[i]$ ，若为“-”，则表示第 k 个错误属于 $F2[i]$ ，若为“0”，则第 k 个错误既不属于 $F1[i]$ 也不属于 $F2[i]$ ，即软件中是否包含第 k 个错误不会因使用补丁 i 而改变。

★结果输出：

程序运行结束时，将总耗时数输出到文件 output.txt。如果问题无解，则输出 0。

输入文件示例

输出文件示例

input.txt

output.txt

3 3

8

1 000 00-

1 00- 0-+

2 0-- -++

分析与解答：

用 n 位串 a 来表示用补丁软件过程中 n 个错误的状态， $a[i]=1$ 表示软件中存在第 i 个错误， $a[i]=0$ 表示软件中第 i 个错误已修正。对于第 i 个补丁软件 $mend[i]$ ，用 2 个 n 位串 $b1$ 和 $b2$ 来表示错误集合 $B1[i]$ 和 $B2[i]$ ；另外用 2 个 n 位串 $f1$ 和 $f2$ 来表示错误集合 $F1[i]$ 和 $F2[i]$ 。在修复软件的任何状态 a ，补丁软件 $mend[i]$ 可运行的条件是

$$a \& mend[i].b1 = mend[i].b1$$

$$a \& mend[i].b2 = 0$$

运行补丁软件 $mend[i]$ 后，软件状态 a 转换为软件状态 $b = (a | mend[i].f1) \& (\sim mend[i].f2)$ 。

软件修复状态构成网络 $G=(V,E)$ 如下：每个软件状态对应于网络中的一个顶点。从软件状态 $a=(11\cdots 1)$ 出发，当补丁软件 $mend[i]$ 可用于当前顶点 a ，并产生新的状态顶点 b 时，增加一条网络边 (a,b) ，其流量为 1，费用为 $time[i]$ （软件 $mend[i]$ 的耗时）。另外增加源 s 和汇 t 。从源 s 到初始状态 $a=(11\cdots 1)$ 有一条有向边，其容量为 1，费用为 0。从目标状

态 $b=(00\cdots 0)$ 到汇 t 有一条有向边, 其容量为 1, 费用为 0。

求相应网络 G 的最小费用流。

算法实现题 8-13 星际转移问题 (习题 8-26)

★问题描述:

由于人类对自然资源的消耗, 人们意识到大约在 2300 年之后, 地球就不能再居住了。于是在月球上建立了新的绿地, 以便在需要时移民。令人意想不到的是, 2177 年冬由于未知的原因, 地球环境发生了连锁崩溃, 人类必须在最短的时间内迁往月球。现有 n 个太空站位于地球与月球之间, 且有 m 艘公共交通太空船在其间来回穿梭。每个太空站可容纳无限多的人, 而每艘太空船 i 只可容纳 $H[i]$ 个人。每艘太空船将周期性地停靠一系列的太空站, 例如, $(1, 3, 4)$ 表示该太空船将周期性地停靠太空站 134134134…。每一艘太空船从一个太空站驶往任一太空站耗时均为 1。人们只能在太空船停靠太空站 (或月球、地球) 时上、下船。初始时所有人全在地球上, 太空船全在初始站。试设计一个算法, 找出让所有人尽快全部转移到月球上的运输方案。

★编程任务:

对于给定的太空船的信息, 找到让所有人尽快全部转移到月球上的运输方案。

★数据输入:

由文件 input.txt 提供输入数据。文件第 1 行有 3 个正整数 n (太空站个数), m (太空船个数) 和 k (需要运送的地球上的人数)。其中 $1 \leq m \leq 13$, $1 \leq n \leq 20$, $1 \leq k \leq 50$ 。

接下来的 m 行给出太空船的信息。第 $i+1$ 行说明太空船 p_i 。第 1 个数表示 p_i 可容纳的人数 H_{p_i} ; 第 2 个数表示 p_i 一个周期停靠的太空站个数 r , $1 \leq r \leq n+2$; 随后 r 个数是停靠的太空站的编号 $(S_{i1}, S_{i2}, \dots, S_{ir})$, 地球用 0 表示, 月球用 -1 表示。时刻 0 时, 所有太空船都在初始站, 然后开始运行。在时刻 1, 2, 3, ... 等正点时刻各艘太空船停靠相应的太空站。人只有在 0, 1, 2, ... 等正点时刻才能上、下太空船。

★结果输出:

程序运行结束时, 将全部人员安全转移所需的时间输出到文件 output.txt。如果问题无解, 则输出 0。

输入文件示例

输出文件示例

input.txt

output.txt

2 2 1

5

1 3 0 1 2

1 3 1 2 -1

分析与解答:

首先考察下面的问题: 在已知时间 T 内, 能转移到月球上的最多人数 $\max p(T)$ 是多少。如果这个问题得到解决, 则原问题变成这个问题的反问题。

将地球、太空站和月球编号为 $0, 1, \dots, n, n+1$ 。其中 0 表示地球, $n+1$ 表示月球, $1, 2, \dots, n$ 分别表示太空站 $1, 2, \dots, n$ 。

建立网络 $G=(V, E)$ 如下: $V=\{u(i, j) \mid i=0, 1, \dots, T; j=0, 1, \dots, n, n+1\}$ 。

其中, $u(i, j)$ 相应于太空站 j 在时刻 i 的状态。从时刻 i 到时刻 $i+1$ 太空站 j 的人流可流向太空站 $0, 1, \dots, n, n+1$ 。因此, 从顶点 $u(i, j)$ 到顶点 $u(i+1, k)$ 之间有一条边 $(u(i, j),$

$u(i+1,k))$ ，其容量 $r(u(i,j), u(i+1,k))$ 为

$$r(u(i,j), u(i+1,k)) = \begin{cases} \sum t(s) & j \neq k \\ \infty & j = k \end{cases}$$

其中， $t(s)$ 表示太空船 s 的容量；求和是对所有满足 $\text{stay}(s,i)=j$ 且 $\text{stay}(s,i+1)=k$ 的 s 进行。 $\text{stay}(s,i)$ 表示太空船 s 在时刻 i 所处的位置。

设源 $s=u(0,0)$ 和汇 $t=(T,n+1)$ 。网络 G 的从 s 到 t 的最大流量即为 $\text{maxp}(T)$ 。

有了求 $\text{maxp}(T)$ 的算法，就可以逐步增加 T 值找出 $\text{maxp}(T) \geq k$ 的最小 T 值。在逐步增加 T 值的过程中，应充分利用已找出的最大流来计算 T 值增加后的最大流。

算法实现题 8-14 孤岛营救问题（习题 8-27）

★问题描述：

1944 年，特种兵麦克接到国防部的命令，要求立即赶赴太平洋上的一个孤岛，营救被敌军俘虏的大兵瑞恩。瑞恩被关押在一个迷宫里，迷宫地形复杂，但幸好麦克得到了迷宫的地形图。迷宫的外形是一个长方形，其南北方向被划分为 N 行，东西方向被划分为 M 列，于是整个迷宫被划分为 $N \times M$ 个单元。每一个单元的位置可用一个有序数对（单元的行号，单元的列号）来表示。南北或东西方向相邻的 2 个单元之间可能互通，也可能有一扇锁着的门，或者是一堵不可逾越的墙。迷宫中有一些单元存放着钥匙，并且所有的门被分成 P 类，打开同一类的门的钥匙相同，不同类门的钥匙不同。

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

★编程任务：

试设计一个算法，帮助麦克以最快的方式到达瑞恩所在单元，营救大兵瑞恩。

★数据输入：

由文件 input.txt 提供输入数据。第 1 行有 3 个整数，分别表示 N, M, P 的值。第 2 行是 1 个整数 K ，表示迷宫中门和墙的总数。第 $I+2$ 行 ($1 \leq I \leq K$)，有 5 个整数，依次为 $Xi1, Yi1, Xi2, Yi2, Gi$ ；

当 $Gi \geq 1$ 时，表示 $(Xi1, Yi1)$ 单元与 $(Xi2, Yi2)$ 单元之间有一扇第 Gi 类的门，当 $Gi = 0$ 时，表示 $(Xi1, Yi1)$ 单元与 $(Xi2, Yi2)$ 单元之间有一堵不可逾越的墙（其中， $|Xi1 - Xi2| + |Yi1 - Yi2| = 1, 0 \leq Gi \leq P$ ）。

第 $K+3$ 行是一个整数 S ，表示迷宫中存放的钥匙总数。

第 $K+3+J$ 行 ($1 \leq J \leq S$) 有 3 个整数，依次为 $Xi1, Yi1, Qi$ ：表示第 J 把钥匙存放在 $(Xi1, Yi1)$ 单元里，并且第 J 把钥匙是用来开启第 Qi 类门的（其中 $1 \leq Qi \leq P$ ）。

输入数据中同一行各相邻整数之间用一个空格分隔。

★结果输出：

程序运行结束时，将麦克营救到大兵瑞恩的最短时间值输出到文件 output.txt。如果问题无解，则输出 -1。

输入文件示例

input.txt

4 4 9

输出文件示例

output.txt

14

```

9
1 2 1 3 2
1 2 2 2 0
2 1 2 2 0
2 1 3 1 0
2 3 3 3 0
2 4 3 4 1
3 2 3 3 0
3 3 4 3 0
4 3 4 4 0
2
2 1 2
4 2 1

```

分析与解答:

用 p 位串 a 来表示钥匙的状态, $a[i]=1$ 表示存在打开 i 类门的钥匙, $a[i]=0$ 表示不存在打开 i 类门的钥匙。特种兵麦克的状态可以用三元组 (x, y, z) 来表示, (x, y) 是当前位置, z 是当前持有钥匙的状态。从状态 (x, y, z) 变换到状态 $(x1, y1, z1)$ 的条件是, 对于 $i=1, 2, 3, 4$ (东, 南, 西, 北):

$$\begin{aligned}
 x1 &= x + \text{dir}(i, 1) \\
 y1 &= y + \text{dir}(i, 2) \\
 z1 &= z | \text{put}(x1, y1) \\
 z \ \& \ \text{map}(x, y, i) &= \text{map}(x, y, i)
 \end{aligned}$$

(x, y) 与 $(x1, y1)$ 之间没有墙。

其中, $\text{dir}(i, 1)$, $\text{dir}(i, 2)$ 表示第 i 个方向上的坐标增量; $\text{put}(x, y)$ 表示在位置 (x, y) 处存放钥匙情况的 p 位串; $\text{map}(x, y, i)$ 也是 p 位串, 表示在位置 (x, y) 的第 i 个方向上门的类型, 仅当门的类型为 j 时, $\text{map}(x, y, i)$ 的第 j 位为 1。

特种兵状态构成网络 $G=(V, E)$ 如下: 每个状态对应于网络中的一个顶点。从软件状态 $a=(1, 1, \text{put}(1, 1))$ 出发, 从当前顶点 (x, y, z) 可以变换到状态 $(x1, y1, z1)$ 时, 增加一条网络边 $((x, y, z), (x1, y1, z1))$, 其流量为 1, 费用为 1。另外增加源 s 和汇 t 。从源 s 到初始状态 $a=(1, 1, \text{put}(1, 1))$ 有一条有向边, 其容量为 1, 费用为 0。从目标状态 $b=(m, n, s)$ 到汇 t 有一条有向边, 其容量为 1, 费用为 0。

求相应网络 G 的最小费用流。

算法实现题 8-15 汽车加油行驶问题 (习题 8-28)

★问题描述:

给定一个 $N \times N$ 的交通方形网格, 设其左上角为起点 \odot , 坐标为 $(1, 1)$, X 轴向右为正, Y 轴向下为正, 每个方格边长为 1, 如图 8-2 所示。一辆汽车从起点 \odot 出发驶向右下角终点 \blacktriangle , 其坐标为 (N, N) 。在若干个网格交叉点处, 设置了油库, 可供汽车在行驶途中加油。汽车在行驶过程中应遵守如下规则:

(1) 汽车只能沿网格边行驶, 装满油后能行驶 K 条网格边。出发时汽车已装满油, 在

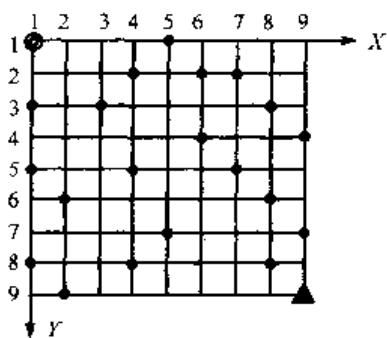


图 8-2 汽车加油行驶问题的交通方形网格

起点与终点处不设油库。

(2) 汽车经过一条网格边时, 若其 X 坐标或 Y 坐标减小, 则应付费 B , 否则免付费。

(3) 汽车在行驶过程中遇油库, 应加满油并付加油费用 A 。

(4) 在需要时可在网格点处增设油库, 并付增设油库费用 C (不含加油费用 A)。

(5) (1) ~ (4) 中的各数 N, K, A, B, C 均为正整数, 且满足约束: $2 \leq N \leq 100, 2 \leq K \leq 10$ 。

设计一个算法, 求出汽车从起点出发到达终点的一条所付费用最少的行驶路线。

★编程任务:

对于给定的交通网格, 计算汽车从起点出发到达终点的一条所付费用最少的行驶路线。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行是 N, K, A, B, C 的值。第 2 行起是一个 $N \times N$ 的 0-1 方阵, 每行 N 个值, 至 $N+1$ 行结束。方阵的第 i 行第 j 列处的值为 1 表示在网格交叉点 (i, j) 处设置了一个油库, 为 0 时表示未设油库。各行相邻两个数以空格分隔。

★结果输出:

程序运行结束时, 将最小费用输出到文件 output.txt 中。

输入文件示例

input.txt

9 3 2 3 6

0 0 0 0 1 0 0 0 0

0 0 0 1 0 1 1 0 0

1 0 1 0 0 0 0 1 0

0 0 0 0 0 1 0 0 1

1 0 0 1 0 0 1 0 0

0 1 0 0 0 0 0 1 0

0 0 0 0 1 0 0 0 1

1 0 0 1 0 0 0 1 0

0 1 0 0 0 0 0 0 0

输出文件示例

output.txt

12

分析与解答:

汽车行驶的状态可以用三元组 (x, y, z) 来表示, (x, y) 是当前位置, z 表示汽车还可行驶 z 条网格边。对于 $i=1, 2, 3, 4$ (东, 南, 西, 北), $x1=x+\text{dir}(i, 1)$, $y1=y+\text{dir}(i, 2)$, 从状态 (x, y, z) 可以变换到状态 $(x1, y1, z-1)$ 。其中, $\text{dir}(i, 1)$, $\text{dir}(i, 2)$ 表示第 i 个方向上的坐标增量。

汽车行驶状态构成网络 $G=(V, E)$ 如下:

每个网格点 (x, y) 对应于网络中的 $K+2$ 个顶点 (x, y, z) , $0 \leq z \leq K+1$ 。当 $0 \leq z \leq K$ 时, 网络顶点 (x, y, z) 对应于汽车行驶的状态; 当 $z=K+1$ 时, 网络顶点 $(x, y, K+1)$ 对应于加油站。

设 $x1 = x + \text{dir}(i, 1)$, $y1 = y + \text{dir}(i, 2)$, $i = 1, 2, 3, 4$ 。

(1) 当 $1 \leq z \leq K$, 且 $(x1, y1)$ 不是加油站时, 从顶点 (x, y, z) 到顶点 $(x1, y1, z-1)$ 有一条有向边, 容量为 1; 当 $x1 < x$ 或 $y1 < y$ 时费用为 B , 否则费用为 0。

(2) 当 $1 \leq z \leq K$, 且 $(x1, y1)$ 是加油站时, 从顶点 (x, y, z) 到顶点 $(x1, y1, K+1)$ 有一条有向边, 容量为 1, 费用为 A 。

(3) 当 $z=0$, 且 (x, y) 不是加油站时, 从顶点 $(x, y, 0)$ 到顶点 $(x, y, K+1)$ 有一条有向边, 容量为 1, 费用为 $A+C$ 。

(4) 从顶点 $(x, y, K+1)$ 到顶点 (x, y, K) 有一条有向边, 容量为 ∞ , 费用为 0。

另外增加源 s 和汇 t 。

(5) 从源 s 到初始状态 $(1, 1, K)$ 有一条有向边, 其容量为 1, 费用为 0。

(6) 从目标状态 (n, n, z) , $0 \leq z < K$ 到汇 t 有一条有向边, 其容量为 1, 费用为 0。

求相应网络 G 的最小费用流。

具体算法实现如下。

read 读入初始数据。

```
int n, k, a, b, c, s, t, f, nn, **map;
int dir[4][2];
void read()
{
    dir[0][0]=0;dir[0][1]=1;
    dir[1][0]=1;dir[1][1]=0;
    dir[2][0]=0;dir[2][1]=-1;
    dir[3][0]=-1;dir[3][1]=0;
    cin>>n>>k>>a>>b>>c;
    Make2DArray(map, n+1, n+1);
    for(int i=1;i<=n;i++)
        for(int j=1;j<=n;j++)
            cin>>map[i][j];
}
```

constructG 构造相应的网络。

```
int num(int i, int j)
{
    return((i-1)*n+j-1)*(k+2)+2;
}

void constructG(GRAPH<EDGE>&G)
{
    s=0;t=1;f=1;
    int i, j, d, e, cost, il, jl, kl, k2;
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++){
```

```

        k1=num(i, j);
        for(e=1;e<=k;e++){
            for(d=0;d<4;d++){
                i1=i+dir[d][0];j1=j+dir[d][1];
                if(i1>0 && i1<=n && j1>0 && j1<=n){
                    cost=(d>1)? b:0;
                    k2=num(i1, j1);
                    if(map[i1][j1])G.insert(new EDGE(k1+e,k2+k-1,1,cost+a));
                    else G.insert(new EDGE(k1+e,k2+e-1,1,cost));
                }
            }
        }
        if(! map[i][j])G.insert(new EDGE(k1,k1+k+1,1,a+c));
        G.insert(new EDGE(k1+k+1,k1+k,5*(k+2),0));
    }
    G.insert(new EDGE(s,k+2,1,0));
    k1=num(n,n);
    for(e=0;e<=k;e++)G.insert(new EDGE(k1+e,t,1,0));
}

```

实现算法的主函数如下。

```

int main()
{
    read();
    nn=n*n*(k+2)+1;
    GRAPH<EDGE>G(nn,1);
    constructG(G);
    MINCOST<GRAPH<EDGE>,EDGE>(G,s,t,f);
    output(G);
    return 0;
}

```

output 输出最小费用。

```

void output (GRAPH<EDGE>&G)
{
    int sum=0;
    for(int i=2;i<=nn;i++){
        adjIterator<EDGE>A(G,i);
        for(EDGE*e=A.beg();! A.end();e=A.nxt())
            if(e->from(i)&&e->flow()>0&&e->cost()>0)
                sum+=e->cost()*e->flow();
    }
}

```

```

    cout<<sum<<endl;
}

```

算法实现题 8-16 数字梯形问题

★问题描述:

给定一个由 n 行数字组成的数字梯形如图 8-3 所示。梯形的第 1 行有 m 个数字。从梯形的顶部的 m 个数字开始,在每个数字处可以沿左下或右下方向移动,形成一条从梯形的顶至底的路径。

规则 1: 从梯形的顶至底的 m 条路径互不相交。

规则 2: 从梯形的顶至底的 m 条路径仅在数字结点处相交。

规则 3: 从梯形的顶至底的 m 条路径允许在数字结点处相交或在边处相交。

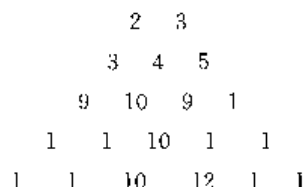


图 8-3 数字梯形

★编程任务:

对于给定的数字梯形,分别按照规则 1、规则 2 和规则 3 计算出从梯形的顶至底的 m 条路径,使这 m 条路径经过的数字总和最大。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行中有 2 个正整数 m 和 n ($m, n \leq 20$), 分别表示数字梯形的第 1 行有 m 个数字,共有 n 行。接下来的 n 行是数字梯形中各行的数字。第 1 行有 m 个数字,第 2 行有 $m+1$ 个数字,……。

★结果输出:

程序运行结束时,将按照规则 1、规则 2 和规则 3 计算出的最大数字总和输出到文件 output.txt 中。每行一个最大总和。

输入文件示例

input.txt

2 5

2 3

3 4 5

9 10 9 1

1 1 10 1 1

1 1 10 12 1 1

输出文件示例

output.txt

66

75

77

分析与解答:

数字梯形构成网络 $G=(V,E)$ 如下: 数字梯形中的每个数字对应于网络 G 的一个顶点。在每个顶点处沿左下和右下方向有一条有向边。另外增加源 s 和汇 t 。从源 s 到数字梯形每个顶层顶点有一条有向边,其容量为 1,费用为 0。从数字梯形每个底层顶点到汇 t 有一条有向边,其容量为 1,费用为 0。

按照规则 1,在每个顶点处有一个顶点容量约束,即经过数字 x 所在顶点的容量为 1,费用为 x 。

按照规则 2,在每条边处有边容量约束,即以数字 x 所在顶点为始发边的容量为 1,费用为 x 。

按照规则 3, 在每条边处没有边容量约束, 即以数字 x 所在顶点为始发边的容量为 ∞ , 费用为 x 。

按照以上 3 种规则, 构造 3 种不同的网络, 求相应网络的最大费用流。

具体算法用类 DIGIT 实现如下。

```

class DIGIT
{
    int n, mm, nn, f, s, t, maxc, **a;
    int num(int i, int j)
    {
        return ((i - 1) * mm + (i - 1) * (i - 2) / 2 + j) * 2;
    }

    int num1(int i, int j)
    {
        return (i - 1) * mm + (i - 1) * (i - 2) / 2 + j + 1;
    }

    void constructG (GRAPH<EDGE>&G)
    {
        f = mm; s = 0; t = 1;
        for (int i = 1, kk = mm; i < nn; i++, kk++)
            for (int j = 1; j <= kk; j++)
            {
                int k = num(i, j), k1 = num(i + 1, j), k2 = num(i + 1, j + 1);
                G.insert(new EDGE(k, k + 1, 1, a[i][j]));
                G.insert(new EDGE(k + 1, k1, 1, 0));
                G.insert(new EDGE(k + 1, k2, 1, 0));
                if (i == 1) G.insert(new EDGE(s, k, 1, 0));
            }
        for (int j = 1; j < mm + nn; j++) {
            int k = num(nn, j);
            G.insert(new EDGE(k, k + 1, 1, a[nn][j]));
            G.insert(new EDGE(k + 1, t, 1, 0));
        }
    }

    void constructG1 (GRAPH<EDGE>&G)
    {
        f = mm; s = 0; t = 1;
        for (int i = 1, kk = mm; i < nn; i++, kk++)
            for (int j = 1; j <= kk; j++)
            {
                int k = num1(i, j), k1 = num1(i + 1, j), k2 = num1(i + 1, j + 1);

```



```

        G.insert(new EDGE(k, k1, 1, a[i+1][j]));
        G.insert(new EDGE(k, k2, 1, a[i+1][j+1]));
        if (i==1) G.insert(new EDGE(s, k, 1, a[i][j]));
    }
    for(int j=1; j<=mm+nn; j++){
        int k=num1(nn, j);
        G.insert(new EDGE(k, t, mm, 0));
    }
}

void constructG2(GRAPH<EDGE>&G)
{
    f=mm; s=0; t=1;
    for (int i=1, kk=mm; i<=nn; i++, kk++)
        for(int j=1; j<=kk; j++)
        {
            int k=num1(i, j), k1=num1(i+1, j), k2=num1(i+1, j+1);
            G.insert(new EDGE(k, k1, mm, a[i+1][j]));
            G.insert(new EDGE(k, k2, mm, a[i+1][j+1]));
            if (i==1) G.insert(new EDGE(s, k, 1, a[i][j]));
        }
    for(int j=1; j<=mm+nn; j++){
        int k=num1(nn, j);
        G.insert(new EDGE(k, t, mm, 0));
    }
    Delete2DArray(a, nn+mm+1);
}

void read(char *filename)
{
    int i, j, k;
    ifstream inFile;
    inFile.open(filename);
    inFile>>mm>>nn;
    Make2DArray(a, nn+mm+1, nn+mm+1);
    for (i=1, maxc=0, k=mm; i<=nn; i++, k++)
        for(j=1; j<=k; j++){
            inFile>>a[i][j];
            if(maxc<a[i][j])maxc=a[i][j];
        }
    inFile.close();
    for (i=1, k=mm; i<=nn; i++, k++)
        for(j=1; j<=k; j++)a[i][j]=maxc-a[i][j];
}

```

```

void trans(int i, int &u, int &v)
{
    u=0;v=i;
    if(i<2) return;
    int k=i/2;
    for(int j=1;j<=nn;j++){
        int ij=j*mm+j*(j-1)/2;
        if(k<=ij){u=j;v=k-(j-1)*mm-(j-1)*(j-2)/2;break;}
    }
}

```

```

void transl(int i, int &u, int &v)
{
    u=0;v=i;
    if(i<2) return;
    i--;
    for(int j=1;j<=nn;j++){
        int ij=j*mm+j*(j-1)/2;
        if(i<=ij){u=j;v=i-(j-1)*mm-(j-1)*(j-2)/2;break;}
    }
}

```

```

void output (GRAPH<EDGE>&G)
{
    int u1,v1,u2,v2, sum=0;
    adjIterator<EDGE>A(G, s);
    for (EDGE*e=A.beg(); !A.end(); e=A.nxt())
        if (e->from(s)&&e->flow()>0) sum++;
    if (sum<f) {cout<<"No Solution!"<<sum<<endl;return;}
    sum=0;
    for (int i=2;i<=n;i++){
        adjIterator<EDGE>A(G, i);
        for (EDGE*e=A.beg(); !A.end(); e=A.nxt())
            if (e->from(i)&&e->flow()>0){
                trans(i, u1, v1);
                trans(e->w(), u2, v2);
                sum+=e->cost()*e->flow();
            }
    }
    cout<<mm*nn*maxc - sum<<endl;
}

```

```

void output1 (GRAPH<EDGE>&G)

```

```

{
    int u1, v1, u2, v2, sum=0;
    adjIterator<EDGE>A(G, s);
    for (EDGE*e=A.beg(); !A.end(); e=A.nxt())
        if (e->from(s)&&e->flow()>0) sum++;
    if (sum<f) {cout<<"No Solution!"<<sum<<endl;return;}
    sum=0;
    for (int i=0;i<=n;i++){
        adjIterator<EDGE>A(G, i);
        for (EDGE*e=A.beg(); !A.end(); e=A.nxt())
            if (e->from(i)&&e->flow()>0){
                trans1(i, u1, v1);
                trans1(e->w(), u2, v2);
                sum+=e->cost()*e->flow();
            }
    }
    cout<<mm*nn*maxc - sum<<endl;
}

public:
    DIGIT(char *filename)
    {
        read(filename);
        n=2*mm*nn+(nn-1)*nn+1;
        GRAPH<EDGE>G(n, 1);
        constructG(G);
        MINCOST<GRAPH<EDGE>, EDGE>(G, s, t, f);
        output(G);
        n=mm*nn+(nn-1)*nn/2+3;
        GRAPH<EDGE>G1(n, 1);
        constructG1(G1);
        MINCOST<GRAPH<EDGE>, EDGE>(G1, s, t, f);
        output1(G1);
        constructG2(G1);
        MINCOST<GRAPH<EDGE>, EDGE>(G1, s, t, f);
        output1(G1);
    }
};

```

其中, constructG 按照规则 1 构造相应网络; constructG1 按照规则 2 构造相应网络; constructG2 按照规则 3 构造相应网络。

实现具体计算的主函数如下。

```
int main()
```

```

{
    create();
    DIGIT("digit.in");
    return 0;
}

```

算法实现题 8-17 运输问题

★问题描述:

W 公司有 m 个仓库和 n 个零售商店。第 i 个仓库有 a_i 个单位的货物；第 j 个零售商店需要 b_j 个单位的货物。货物供需平衡，即 $\sum_{i=1}^m a_i = \sum_{j=1}^n b_j$ 。从第 i 个仓库运送每单位货物到第 j 个零售商店的费用为 c_{ij} 。试分别设计一个将仓库中所有货物运送到零售商店的最优和最差运输方案，即使总运输费用最少或最多。

★编程任务:

对于给定的 m 个仓库和 n 个零售商店间运送货物的费用，计算最优运输方案和最差运输方案。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行有 2 个正整数 m 和 n ，分别表示仓库数和零售商店数。接下来的一行中有 m 个正整数 $a_i, 1 \leq i \leq m$ ，表示第 i 个仓库有 a_i 个单位的货物。再接下来的一行中有 n 个正整数 $b_j, 1 \leq j \leq n$ ，表示第 j 个零售商店需要 b_j 个单位的货物。接下来的 m 行，每行有 n 个整数，表示从第 i 个仓库运送每单位货物到第 j 个零售商店的费用 c_{ij} 。

★结果输出:

程序运行结束时，将计算出的最少运输费用和最多运输费用输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
2 3	48500
220 280	69140
170 120 210	
77 39 105	
150 186 122	

分析与解答:

设最优运输方案从第 i 个仓库运送 x_{ij} 单位货物到第 j 个零售商店。运输问题可以表述为如下的线性规划问题。

$$\begin{aligned}
 & \min \quad \sum_{i=1}^m \sum_{j=1}^n c_{ij} x_{ij} \\
 \text{s. t.} \quad & \sum_{i=1}^m x_{ij} = b_j \quad j = 1, 2, \dots, n \\
 & \sum_{j=1}^n x_{ij} = a_i \quad i = 1, 2, \dots, m
 \end{aligned}$$

$$x_{ij} \geq 0, i = 1, 2, \dots, m, j = 1, 2, \dots, n$$

用单纯形算法求解。最差运输方案相应于求最大值的同一线性规划问题。

具体算法实现如下。

read 读入初始数据。

```
int m, n, *a, *b, **c;
void read()
{
    cin >> m >> n;
    a = new int[m];
    b = new int[n];
    Make2DArray(c, m, n);
    for (int i = 0; i < m; i++) cin >> a[i];
    for (int j = 0; j < n; j++) cin >> b[j];
    for (i = 0; i < m; i++)
        for (j = 0; j < n; j++) cin >> c[i][j];
}
```

用线性规划算法的成员函数 lpin 建立相应的初始单纯形表。

```
void LinearProgram::lpin(int mima, int mm, int nn, int *aa, int *bb, int **cc)
{
    ifstream inFile;
    int i, j, k;
    double value;
    minmax = mima; m = mm + nn, n = mm * nn;
    m1 = 0; m2 = m; m3 = 0; n1 = n; n2 = n;
    Make2DArray(a, m + 2, n1 + 1);
    basic = new int[m + 2];
    nonbasic = new int[n1 + 1];
    for (i = 0; i <= m + 1; i++)
        for (j = 0; j <= n1; j++) a[i][j] = 0.0;
    for (j = 0; j <= n1; j++) nonbasic[j] = j;
    for (i = 1, j = n1 + 1; i <= m; i++, j++) basic[i] = j;
    for (i = m + m3 + 1, j = n + 1; i <= m; i++, j++) {
        a[i][j] = -1.0;
        a[m + 1][j] = -1.0;
    }
    for (k = 0; k < mm; k++)
        for (i = 0, j = k * nn; i < nn; i++) a[i + 1][j + i + 1] = 1;
    for (k = 0; k < mm; k++)
        for (i = 0, j = k * nn; i < nn; i++) a[nn + k + 1][j + i + 1] = 1;
    for (j = 0; j < nn; j++) a[j + 1][0] = bb[j];
    for (i = 0; i < mm; i++) a[nn + i + 1][0] = aa[i];
}
```

```

        for(i=0;i<mm;i++){
            for(j=0;j<nn;j++){
                int jj=i*nn+j+1;
                a[0][jj]=minmax*cc[i][j];
            }
        }
        for(j=1;j<=n;j++){
            for(i=m1+1,value=0.0;i<=m;i++) value+=a[i][j];
            a[m+1][j]=value;
        }
    }
}

```

算法的主函数调用单纯形算法求解。

```

int main()
{
    read();
    LinearProgram X;
    X.lpin(-1,m,n,a,b,c);
    X.solve();
    X.lpin(1,m,n,a,b,c);
    X.solve();
    return 0;
}

```

此题显然也可以用最小费用流模型求解。

算法实现题 8-18 分配工作问题

★问题描述:

有 n 件工作要分配给 n 个人做。第 i 个人做第 j 件工作产生的效益为 c_{ij} 。试分别设计一个将 n 件工作分配给 n 个人做的最优和最差分配方案,使产生的总效益最大或最小。

★编程任务:

对于给定的 n 件工作和 n 个人,计算最优分配方案和最差分配方案。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行有 1 个正整数 n ,表示有 n 件工作要分配给 n 个人做。接下来的 n 行中,每行有 n 个整数 c_{ij} , $1 \leq i \leq n$, $1 \leq j \leq n$,表示第 i 个人做第 j 件工作产生的效益为 c_{ij} 。

★结果输出:

程序运行结束时,将计算出的最小总效益和最大总效益输出到文件 output.txt。

输入文件示例

input.txt

5

2 2 2 1 2

输出文件示例

output.txt

5

14

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

分析与解答:

设变量 x_{ij} 表示将第 j 件工作分配给第 i 个人做。分配问题可以表述为如下的线性规划问题。

$$\begin{aligned} \max \quad & \sum_{i=1}^n \sum_{j=1}^n c_{ij} x_{ij} \\ \text{s. t.} \quad & \sum_{i=1}^n x_{ij} = 1 \quad j = 1, 2, \dots, n \\ & \sum_{j=1}^n x_{ij} = 1 \quad i = 1, 2, \dots, n \\ & x_{ij} \in \{0, 1\}, i = 1, 2, \dots, m, j = 1, 2, \dots, n \end{aligned}$$

上述线性规划问题实际上是整数线性规划问题。但从约束条件可以看出, 约束矩阵为全 1 模阵, 因此 0-1 变量可以松弛为区间 $[0, 1]$ 中的实数, 用单纯形算法求解可得到 0-1 整数解。进一步还可将变量 x_{ij} 松弛为所有非负实数。因此, 上述问题转化为一个特殊的运输问题。

具体算法实现如下。

read 读入初始数据。

```
int n,*a,*b,**c;
void read()
{
    cin>>n;
    a=new int[n];
    b=new int[n];
    Make2DArray(c,n,n);
    for(int i=0;i<n;i++){a[i]=1;b[i]=1;}
    for(i=0;i<n;i++)
        for(int j=0;j<n;j++)cin>>c[i][j];
}
```

用线性规划算法的成员函数 lpin 建立相应的初始单纯形表。算法的主函数调用单纯形法求解。

```
int main()
{
    read();
    LinearProgram X;
    X.lpin(-1,n,n,a,b,c);
    X.solve();
    X.lpin(1,n,n,a,b,c);
    X.solve();
}
```

```
    return 0;
}
```

此题显然也可以用最小费用流模型求解。

算法实现题 8-19 负载均衡问题

★问题描述:

G 公司有 n 个沿铁路运输线环形排列的仓库, 每个仓库存储的货物数量不等。如何用最少搬运量可以使 n 个仓库的库存数量相同。搬运货物时, 只能在相邻的仓库之间搬运。

★编程任务:

对于给定的 n 个环形排列的仓库的库存量, 编程计算使 n 个仓库的库存数量相同的最少搬运量。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行中有 1 个正整数 n , 表示有 n 个仓库。第 2 行中有 n 个正整数, 表示 n 个仓库的库存量。

★结果输出:

程序运行结束时, 将计算出的最少搬运量输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5	11
17 9 14 16 4	

分析与解答:

设每个仓库的货物数量为 x_i , 最终要使 n 个仓库的库存数量相同, 即每个仓库的库存数量变成 $\sum_{i=1}^n x_i / n$ 。因此有些仓库要搬出货物, 有些仓库要搬入货物。容易看出, 此题实际上是算法实现题 8-17 运输问题的特殊情形。要搬出货物的仓库对应于运输问题中的仓库, 要搬入货物的仓库对应于运输问题中的零售商店。从第 i 个仓库搬运单位数量货物到第 j 个仓库的费用为 $\min \{ |j-i|, n-|j-i| \}$ 。

具体算法实现如下。

read 读入初始数据并转化为相应的运输问题。

```
bool read()
{
    int i, k=0, ii=0, jj=0, nn, *data, *ia, *ib;
    cin >> nn;
    data = new int[nn+1];
    for(i=1; i<=nn; i++){
        cin >> data[i];
        k += data[i];
    }
    if(k%nn!=0){
        cout << "No Solution!" << endl;
    }
}
```



```

        return false;
    }
    k/=nn;m=0;n=0;
    for(i=1;i<=nn;i++){if(data[i]>k)m++;if(data[i]<k)n++;}
    a=new int[m];b=new int[n];ia=new int[m];ib=new int[n];
    Make2DArray(c,m,n);
    for(i=1;i<=nn;i++){
        if(data[i]>k){ia[ii]=i;a[ii++]=data[i]-k;}
        if(data[i]<k){ib[jj]=i;b[jj++]=k-data[i];}
    }
    for(i=0;i<m;i++){
        for(int j=0;j<n;j++){
            c[i][j]=abs(ia[i]-ib[j]);
            if(nn-c[i][j]<c[i][j])c[i][j]=nn-c[i][j];
        }
    }
    return true;
}

```

用线性规划算法的成员函数 `lpin` 建立相应的初始单纯形表。算法的主函数调用单纯形算法求解。

```

int main()
{
    if(! read())return 1;
    LinearProgram X;
    X.lpin(-1,m,n,a,b,c);
    X.solve();
    return 0;
}

```

此题也可以直接用最小费用流模型求解。

算法实现题 8-20 深海机器人问题

★问题描述:

深海资源考察探险队的潜艇将到达深海的海底进行科学考察。潜艇内有多个深海机器人。潜艇到达深海海底后,深海机器人将离开潜艇向预定目标移动。深海机器人在移动中还必须沿途采集海底生物标本。沿途生物标本由最先遇到它的深海机器人完成采集。每条预定路径上的生物标本的价值是已知的,而且生物标本只能被采集一次。本题限定深海机器人只能从其出发位置沿着向北或向东的方向移动,而且多个深海机器人可以在同一时间占据同一位置。

★编程任务:

用一个 $P \times Q$ 网格表示深海机器人的可移动位置。西南角的坐标为 $(0,0)$, 东北角的坐标为 (Q,P) , 如图 8-4 所示。

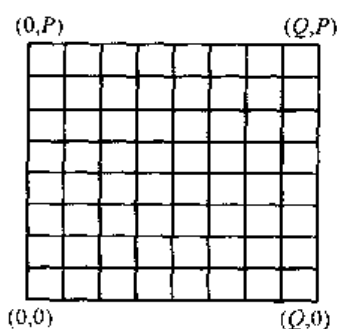


图 8-4 深海机器人的移动网格

给定每个深海机器人的出发位置和目标位置，以及每条网格边上生物标本的价值。计算深海机器人的最优移动方案，使深海机器人到达目的地后，采集到的生物标本的总价值最高。

★数据输入：

由文件 input.txt 提供输入数据。文件的第 1 行为深海机器人的出发位置数 a ，和目的地数 b ，第 2 行为 P 和 Q 的值。接下来的 $P+1$ 行，每行有 Q 个正整数，表示向东移动路径上生物标本的价值，行数据依从南到北方向排列。再接下来的 $Q+1$ 行，每行有 P 个正整数，表示向北移动路径上生物标本的价值，行数据依从西到东方向排列。接下来的 a 行，每行有 3 个正整数 k, x, y ，表示有 k 个深海机器人从 (x, y) 位置坐标出发。再接下来的 b 行，每行有 3 个正整数 r, x, y ，表示有 r 个深海机器人可选择 (x, y) 位置坐标作为目的地。

★结果输出：

程序运行结束时，将采集到的生物标本的最高总价值输出到文件 output.txt。

输入文件示例

input.txt

1 1

2 2

1 2

3 4

5 6

7 2

8 10

9 3

2 0 0

2 2 2

输出文件示例

output.txt

42

分析与解答：

与算法实现题 8-16 数字梯形问题的解法类似。不同之处是本题的容量约束是边容量约束，而且每条边的费用只能计算一次。为此目的，将每个网格顶点拆成 3 个顶点，构成网络 $G=(V, E)$ ，如图 8-5 所示。

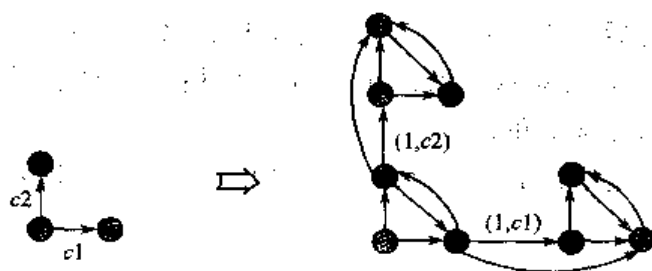


图 8-5 深海机器人的网络变换

图 8-5 中连接原顶点的边的容量为 1, 费用为原来边的价值。其余边的容量为 ∞ , 费用为 0。另外增加源 s 和汇 t 。从源 s 到每个出发位置相应的顶点有一条有向边, 其容量为 1, 费用为 0。从每个目的位置相应的顶点到汇 t 有一条有向边, 其容量为 1, 费用为 0。

求上述网络的最大费用流。

具体算法用类 ROBO 实现如下。

```

class ROBO
{
    int n, p, q, f, s, t, a1, b1, **a, **b, **c, **d;
    void read()
    {
        int i, j;
        cin >> a1 >> b1;
        cin >> p >> q;
        Make2DArray(a, p+1, q);
        Make2DArray(b, q+1, p);
        Make2DArray(c, a1, 3);
        Make2DArray(d, b1, 3);
        for(i=0; i<=p; i++)
            for(j=0; j<q; j++) {
                cin >> a[i][j];
            }
        for(i=0; i<=q; i++)
            for(j=0; j<p; j++) {
                cin >> b[i][j];
            }
        for(i=0, f=0; i<a1; i++) {cin >> c[i][0] >> c[i][1] >> c[i][2]; f+=c[i][0];}
        for(i=0; i<b1; i++) cin >> d[i][0] >> d[i][1] >> d[i][2];
    }

    int num(int i, int j)
    {
        return(i*q+i+j)*3+2;
    }

    void constructG(GRAPH<EDGE>&G)
    {
        s=0; t=1;
        int i, j, k1, k2;
        for(i=0; i<=p; i++)
            for(j=0; j<q; j++) {
                k1=num(i, j); k2=num(i, j+1);
                G.insert(new EDGE(k1, k1+1, f, 0));
            }
    }
}

```

```

        G.insert(new EDGE(k1+1, k2, 1, -a[i][j]));
        G.insert(new EDGE(k1+1, k2+1, f, 0));
        G.insert(new EDGE(k1+1, k1+2, f, 0));
    }
    for(i=0; i<=q; i++)
        for(j=0; j<p; j++) {
            k1=num(j, i); k2=num(j+1, i);
            G.insert(new EDGE(k1, k1+2, f, 0));
            G.insert(new EDGE(k1+2, k2, 1, -b[i][j]));
            G.insert(new EDGE(k1+2, k2+2, f, 0));
            G.insert(new EDGE(k1+2, k1+1, f, 0));
        }
    for(j=0; j<=q; j++) {
        k1=num(p, j);
        G.insert(new EDGE(k1, k1+2, f, 0));
        G.insert(new EDGE(k1+2, k1+1, f, 0));
    }
    for(j=0; j<=p; j++) {
        k1=num(j, q);
        G.insert(new EDGE(k1, k1+1, f, 0));
        G.insert(new EDGE(k1+1, k1+2, f, 0));
    }
    for(j=0; j<a1; j++) {
        k1=num(c[j][1], c[j][2]);
        G.insert(new EDGE(s, k1, c[j][0], 0));
    }
    for(j=0; j<b1; j++) {
        k1=num(d[j][1], d[j][2]);
        G.insert(new EDGE(k1+2, t, d[j][0], 0));
    }
}

void output (GRAPH<EDGE>&G)
{
    int sum=0;
    adjIterator<EDGE>A(G, s);
    for(EDGE*e=A.beg(); !A.end(); e=A.nxt())
        if(e->from(s)&&e->flow()>0) sum+=e->flow();
    if(sum<f) {cout<<"No Maxflow!"<<sum<<endl;}
    sum=0;
    for(int i=2; i<=n; i++) {
        adjIterator<EDGE>A(G, i);
        for(EDGE*e=A.beg(); !A.end(); e=A.nxt())
            if(e->from(i)&&e->flow()>0&&e->cost()<0)

```

```

        sum -= e -> cost() * e -> flow();
    }
    cout << sum << endl;
}

public:
    ROBO()
    {
        read();
        n = (p + 1) * (q + 1) * 3 + 2;
        GRAPH<EDGE> G(n, 1);
        constructG(G);
        MINCOST<GRAPH<EDGE>, EDGE>(G, s, t, f);
        output(G);
    }
};

```

其中, constructG 构造相应网络, 并将最大费用流问题转换为最小费用流问题, 由 MINCOST 求相应的最小费用流。

算法实现题 8-21 最长 k 可重区间集问题

★问题描述:

给定实直线 L 上 n 个开区间组成的集合 I 和一个正整数 k , 试设计一个算法, 从开区间集合 I 中选取开区间集合 $S \subseteq I$, 使得在实直线 L 的任何一点 x , S 中包含点 x 的开区间个数不超过 k , 且 $\sum_{z \in S} |z|$ 达到最大。这样的集合 S 称为开区间集合 I 的最长 k 可重区间集。

$\sum_{z \in S} |z|$ 称为最长 k 可重区间集的长度。

★编程任务:

对于给定的开区间集合 I 和正整数 k , 计算开区间集合 I 的最长 k 可重区间集的长度。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行有 2 个正整数 n 和 k , 分别表示开区间的个数和开区间的可重叠数。接下来的 n 行, 每行有 2 个整数, 表示开区间的左、右端点坐标。

★结果输出:

程序运行结束时, 将计算出的最长 k 可重区间集的长度输出到文件 output.txt。

输入文件示例

输出文件示例

input.txt

output.txt

4 2

15

1 7

6 8

7 10

9 13

分析与解答:

设给定的开区间集合共有 m 个不同的端点。将这 m 个端点从小到大排序为: x_1, x_2, \dots, x_m 。构造网络 $G=(V, E)$ 如下。

(1) 每个端点 x_i 对应于网络中的一个顶点。另外增加源顶点 x_0 和汇顶点 x_{m+1} 。

(2) 设给定的开区间集合中与顶点对 $x_i < x_j$ 相应的开区间数为 c_{ij} 。每对顶点 (x_i, x_j) 之间有一条有向边, 容量为 c_{ij} , 费用为 $x_j - x_i$ 。

(3) 每对顶点 (x_i, x_{i+1}) , $1 \leq i < m$, 之间如果没有边, 则增加一条有向边, 容量为 k , 费用为 0。

顶点对 (x_0, x_1) 和 (x_m, x_{m+1}) 之间分别有一条有向边, 容量为 k , 费用为 0。

求网络 G 的最大费用流, 最大费用流的总费用即为所求最长 k 可重区间集的长度。

具体算法用类 INTERV 实现如下。

```
class INTERV
{
    int m, n, k, f, s, t, **a, *b, **c;
    void read()
    {
        int i, j;
        cin >> n >> k;
        Make2DArray(a, n+1, 2);
        b = new int[2*n];
        b[0] = 0;
        for(i=1; i<=n; i++){
            cin >> a[i][0] >> a[i][1];
            if(a[i][0] > a[i][1]) Swap(a[i][0], a[i][1]);
            b[2*i-2] = a[i][0]; b[2*i-1] = a[i][1];
        }
        MergeSort(b, 2*n);
        m = dis(b, 2*n);
        Make2DArray(c, m, m);
        for(i=0; i<m; i++)
            for(j=0; j<m; j++) c[i][j] = 0;
        for(i=1; i<=n; i++){
            int p = BinarySearch(b, a[i][0], m+1), q = BinarySearch(b, a[i][1], m+1);
            c[p][q]++;
        }
    }

    void constructG (GRAPH<EDGE> &G)
    {
        int i, j;
        s = 0; t = m+1;
        for(i=0; i<m; i++)
```

```

        for(j=i+1;j<=m;j++)
            if(c[i][j])G.insert(new EDGE(i+1,j+1,c[i][j],b[i]-b[j]));
    for(i=0;i<=m-1;i++)
        if(c[i][i+1]==0)G.insert(new EDGE(i+1,i+2,k,0));
    G.insert(new EDGE(0,1,k,0));
    G.insert(new EDGE(m,m+1,k,0));
}

void output (GRAPH<EDGE>&G)
{
    int sum=0;
    adjIterator<EDGE>A(G,s);
    for(EDGE*e=A.beg();!A.end();e=A.nxt())
        if(e->from(s)&&e->flow()>0) sum+=e->flow();
    if(sum<k) {cout<<"No Maxflow! "<<sum<<endl;}
    sum=0;
    for(int i=1;i<=m;i++){
        adjIterator<EDGE>A(G,i);
        for(EDGE*e=A.beg();!A.end();e=A.nxt())
            if(e->from(i)&&e->flow()>0&&e->cost()<0)
                sum-=e->cost()*e->flow();
    }
    cout<<sum<<endl;
}

public:
    INTERV()
    {
        read();
        GRAPH<EDGE>G(m+1,1);
        constructG(G);
        MINCOST<GRAPH<EDGE>,EDGE>(G,s,t,k);
        output(G);
    }
};

```

其中, constructG 构造相应网络, 并将最大费用流问题转换为最小费用流问题, 由 MINCOST 求相应的最小费用流。

函数 dis 删去数组 b 中重复端点。

```

int dis(int *b,int sz)
{
    int *e=new int[sz+1];
    for(int i=1,j=0;i<=sz;i++)e[i]=b[i];
}

```

```

for(i=1, j=1; i<=sz; i++)
    if(e[i]!=b[j-1]) b[j++] = e[i];
delete [] e;
return j-1;
}

```

算法实现题 8-22 最长 k 可重线段集问题

★问题描述:

给定平面 XOY 上 n 个开线段组成的集合 I 和一个正整数 k , 试设计一个算法, 从开线段集合 I 中选取开线段集合 $S \subseteq I$, 使得在 X 轴上的任何一点 p , S 中与直线 $x=p$ 相交的开线段个数不超过 k , 且 $\sum_{z \in S} |z|$ 达到最大。这样的集合 S 称为开线段集合 I 的最长 k 可重线段集。 $\sum_{z \in S} |z|$ 称为最长 k 可重线段集的长度。对于任何开线段 z , 设其端点坐标为 (x_0, y_0) 和 (x_1, y_1) , 则开线段 z 的长度定义为 $|z| = \lfloor \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2} \rfloor$ 。

★编程任务:

对于给定的开线段集合 I 和正整数 k , 计算开线段集合 I 的最长 k 可重线段集的长度。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行有 2 个正整数 n 和 k , 分别表示开线段的个数和开线段的可重叠数。接下来的 n 行, 每行有 4 个整数, 表示开线段的 2 个端点坐标。

★结果输出:

程序运行结束时, 将计算出的最长 k 可重线段集的长度输出到文件 output.txt。

输入文件示例

输出文件示例

input.txt

output.txt

4 2

17

1 2 7 3

6 5 8 3

7 8 10 5

9 6 13 9

分析与解答:

设给定的开线段集合共有 m 个不同的端点。将这 m 个端点按照字典序从小到大排序为 p_1, p_2, \dots, p_m 。构造网络 $G=(V, E)$ 如下。

(1) 每个端点 p_i 对应于网络中的一个顶点。另外增加源顶点 p_0 和汇顶点 p_{m+1} 。

(2) 设给定的开线段集合中与顶点对 $p_i < p_j$ 相应的开线段数为 c_{ij} 。每对顶点 (p_i, p_j) 之间有一条有向边, 容量为 c_{ij} , 费用为 $|(p_i, p_j)|$ 。

(3) 每对顶点 (p_i, p_{i+1}) , $1 \leq i < m$, 之间如果没有边, 则增加一条有向边, 容量为 k , 费用为 0。

顶点对 (p_0, p_1) 和 (p_m, p_{m+1}) 之间分别有一条有向边, 容量为 k , 费用为 0。

求网络 G 的最大费用流, 最大费用流的总费用即为所求最长 k 可重线段集的长度。

与算法实现题 8-21 最长 k 可重区间集问题不同的是, 线段的端点是二维的, 用类 ppair 表示二维端点如下。

```
class ppair{
public:
    bool operator<=(ppair a) const
    {
        if(x==a.x) return (y<=a.y);
        else return(x<a.x);
    }

    bool operator<(ppair a) const
    {
        if (x<a.x || (x==a.x && y<a.y)) return true;
        return false;
    }

    bool operator>(ppair a) const
    {
        if (x>a.x || (x==a.x && y>a.y)) return true;
        return false;
    }

    bool operator==(ppair a) const
    {
        return(x==a.x && y==a.y);
    }

    bool operator!=(ppair a) const
    {
        return !(*this==a);
    }

    int x,y;
};

ostream& operator<<(ostream& ostr, const ppair& a)
{
    return ostr<<a.x<<" "<<a.y;
}

istream& operator>>(istream& istr, ppair& a)
{
    istr>>a.x>>a.y;
```

```

        return istr;
    }

```

具体算法用类 LINE 实现如下。

```

class LINE
{
    int m,n,k,f,s,t,**c;
    ppair **a,*b;
    void read()
    {
        int i,j;
        cin>>n>>k;
        Make2DArray(a,n+1,2);
        b=new ppair[2*n];
        b[0].x=b[0].y=0;
        for(i=1;i<=n;i++){
            cin>>a[i][0]>>a[i][1];
            if(a[i][1]<=a[i][0])Swap(a[i][0],a[i][1]);
            b[2*i-2]=a[i][0];b[2*i-1]=a[i][1];
        }
        MergeSort(b,2*n);
        m=dis(b,2*n);
        Make2DArray(c,m,m);
        for(i=0;i<m;i++){
            for(j=0;j<m;j++)c[i][j]=0;
            for(i=1;i<=n;i++){
                int p=BinarySearch(b,a[i][0],m+1),q=BinarySearch(b,a[i][1],m+1);
                c[p][q]++;
            }
        }

    void constructG(GRAPH<EDGE>&G)
    {
        int i,j;
        s=0;t=m+1;
        for(i=0;i<m;i++){
            for(j=i+1;j<m;j++)
                if(c[i][j])
                    G.insert(new EDGE(i+1,j+1,c[i][j],-len(b[j],b[i])));
            for(i=0;i<m-1;i++)
                if(c[i][i+1]==0)G.insert(new EDGE(i+1,i+2,k,0));
            G.insert(new EDGE(0,1,k,0));
            G.insert(new EDGE(m,m+1,k,0));
        }
    }
}

```

```

    }

    void output (GRAPH<EDGE>&G)
    {
        int sum=0;
        adjIterator<EDGE>A(G, s);
        for(EDGE*e=A.beg();!A.end();e=A.nxt())
            if(e->from(s)&&e->flow()>0) sum+=e->flow();
        if(sum<k) {cout<<"No Maxflow! "<<sum<<endl;}
        sum=0;
        for(int i=1;i<=m;i++){
            adjIterator<EDGE>A(G, i);
            for(EDGE*e=A.beg();!A.end();e=A.nxt())
                if(e->from(i)&&e->flow()>0&&e->cost()<0)
                    sum-=e->cost()*e->flow();
        }
        cout<<sum<<endl;
    }

public:
    LINE()
    {
        read();
        GRAPH<EDGE>G(m+1, 1);
        constructG(G);
        MINCOST<GRAPH<EDGE>, EDGE>(G, s, t, k);
        output(G);
    }
};

```

其中, constructG 构造相应网络, 并将最大费用流问题转换为最小费用流问题, 由 MINCOST 求相应的最小费用流。

函数 dis 删去数组 b 中重复端点。len 计算线段的长度。

```

int dis(ppair *b, int sz)
{
    ppair *e=new ppair[sz+1];
    for(int i=1, j=0; i<=sz; i++) e[i]=b[i];
    for(i=1, j=1; i<=sz; i++)
        if(e[i]!=b[j-1]) b[j++]=e[i];
    delete []e;
    return j-1;
}

```

```

int len(ppair u,ppair v)
{
    double z=sqrt((u.x-v.x)*(u.x-v.x)+(u.y-v.y)*(u.y-v.y));
    return int(z);
}

```

算法实现题 8-23 火星探险问题

★问题描述:

火星探险队的登陆舱将在火星表面着陆,登陆舱内有多部障碍物探测车。登陆舱着陆后,探测车将离开登陆舱向先期到达的传送器方向移动。探测车在移动中还必须采集岩石标本。每一块岩石标本由最先遇到它的探测车完成采集。每块岩石标本只能被采集一次。岩石标本被采集后,其他探测车可以从原来岩石标本所在处通过。探测车不能通过有障碍的地面。本题限定探测车只能从登陆处沿着向南或向东的方向朝传送器移动,而且多个探测车可以在同一时间占据同一位置。如果某个探测车在到达传送器以前不能继续前进,则该车所采集的岩石标本将全部损失。

★编程任务:

用一个 $P \times Q$ 网格表示登陆舱与传送器之间的位置。登陆舱的位置在 (X_1, Y_1) 处,传送器的位置在 (X_P, Y_Q) 处,如图 8-6 所示。

X_1, Y_1	X_2, Y_1	X_3, Y_1	...	X_{P-1}, Y_1	X_P, Y_1
X_1, Y_2	X_2, Y_2	X_3, Y_2	...	X_{P-1}, Y_2	X_P, Y_2
X_1, Y_3	X_2, Y_3	X_3, Y_3	...	X_{P-1}, Y_3	X_P, Y_3
...			...		
X_1, Y_{Q-1}	X_2, Y_{Q-1}	X_3, Y_{Q-1}	...	X_{P-1}, Y_{Q-1}	X_P, Y_{Q-1}
X_1, Y_Q	X_2, Y_Q	X_3, Y_Q	...	X_{P-1}, Y_Q	X_P, Y_Q

图 8-6 火星探测车的移动网格

给定每个位置的状态,计算探测车的最优移动方案,使到达传送器的探测车的数量最多,而且探测车采集到的岩石标本的数量最多。

★数据输入:

由文件 input.txt 提供输入数据。文件的第 1 行为探测车数,第 2 行为 P 的值,第 3 行为 Q 的值。接下来的 Q 行是表示登陆舱与传送器之间的位置状态的 $P \times Q$ 网格。用 3 个数字表示火星表面位置的状态:0 表示平坦无障碍,1 表示障碍,2 表示石块。

★结果输出:

程序运行结束时,将每个探测车向传送器移动的序列输出到文件 output.txt。每行包含探测车号和一个移动方向,0 表示向南移动,1 表示向东移动。

输入文件示例

输出文件示例

input.txt

output.txt

2

1 1

10

1 1

8

1 1

0000000000	11
0000011000	10
0001020000	10
1101200001	11
0100201100	10
0101001100	10
0120000100	10
0000000000	21
	21
	21
	21
	20
	20
	20
	20
	21
	20
	20
	21
	20
	21
	21
	21

分析与解答:

对于给定的 $P \times Q$ 网格, 构造网络 G 如下。每个方格对应于图 G 中的一个顶点。如果当前方格顶点 (i, j) 到方格顶点 $(i+1, j+1)$ 可达, 则在该顶点对之间增加一条边, 容量为 ∞ , 费用为 2。在每个有标本石块的方格顶点处增设顶点容量约束为 1, 费用为 1。为达到此目的, 将该顶点 $v(i, j)$ 拆为 2 个顶点 $v(i, j)$ 和 $v'(i, j)$, 并增设 3 条边:

(1) 顶点 $v(i, j)$ 到顶点 $v'(i, j)$ 间增设一条边, 容量为 1, 费用为 0, 表示标本石块只能收集一次。

(2) 顶点 $v'(i, j)$ 到顶点 $v(i+1, j)$ 和顶点 $v(i, j+1)$ 间分别增设一条边, 容量为 1, 费用为 1。这样一来, 每收集一次标本石块将使流的费用减 1。

最后, 增加源 s 和汇 t 。从源 s 到出发顶点 $v(1, 1)$ 有一条有向边, 其容量为探测车数 n , 费用为 0。从每个目的顶点 $v(P, Q)$ 到汇 t 有一条有向边, 容量为 ∞ , 费用为 2。如果方格 (P, Q) 处有标本石块, 则再增加一条从 $v'(P, Q)$ 到汇 t 的边, 容量为 1, 费用为 1。

求上述网络的最小费用流。

算法实现题 8-24 骑士共存问题

★问题描述:

在一个 $n \times n$ 个方格的国际象棋棋盘上, 骑士(马)可以攻击的棋盘方格如图 8-7 所示。

	X		X	
X				X
		S		
X				X
	X		X	

图 8-7 马攻击的棋盘方格

棋盘上某些方格设置了障碍，骑士不得进入。

★编程任务：

对于给定的 $n \times n$ 个方格的国际象棋棋盘和障碍标志，计算棋盘上最多可以放置多少个骑士，使得它们彼此互不攻击。

★数据输入：

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m ($1 \leq n \leq 200, 0 \leq m < n^2$)，分别表示棋盘的大小和障碍数。接下来的 m 行给出障碍的位置。每行 2 个正整数，表示障碍的方格坐标。

★结果输出：

将计算出的共存骑士数输出到文件 output.txt。

输入文件示例

input.txt

3 2

1 1

3 3

输出文件示例

output.txt

5

分析与解答：

(1) 解法 1: 用最大独立集法求解

将国际象棋棋盘看作黑白相间的方格阵列，除了障碍方格外，每个方格对应于图 G 中的一个顶点，并在马的攻击范围内的顶点对间增加一条边。由此构成的图 G 是一个二分图。所求的马的最优放置方案对应于图 G 的一个最大独立集。

设图 $G=(V, E)$ 的最大独立集中顶点数为 $\alpha(G)$ ；

图 G 的最大匹配中边数为 $\alpha'(G)$ ；

图 G 的最小顶点覆盖中顶点数为 $\beta(G)$ ；

图 G 的最小边覆盖中边数为 $\beta'(G)$ ；则有

① 若 $S \subseteq V$ 是图 G 的独立集，则 $V-S$ 是图 G 的顶点覆盖。因此， $\alpha(G) + \beta(G) = n$ 。

② 若图 G 无孤立顶点，则 $\alpha'(G) + \beta(G) = n$ 。

③ 若图 G 是无孤立顶点的二分图，则 $\alpha(G) = \beta(G)$ ，从而 $\alpha(G) = n - \alpha'(G)$ 。

由此可见，所求问题可以转化为二分图 G 的最大匹配问题。

具体算法实现如下。

read 读入初始数据。

```
void read()
{
    int i, j, k;
    cin >> n >> m;
    Make2DArray(c, n+1, n+1);
    Make2DArray(d, n+1, n+1);
    d[0][0] = 0;
    for(j=1; j<=n; j++) d[j][0] = !d[j-1][0];
    for(i=1; i<=n; i++)
```

```

        for(j=1;j<=n;j++) d[i][j]!=d[i][j-1];
for(k=0;k<m;k++) {
    cin>>i>>j;d[i][j]=2;
}
for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
        if(d[i][j]==0)c[i][j]=t+1;
        else c[i][j]=0;
nl=t-1;
for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
        if(d[i][j]==1)c[i][j]=t++;
t--;
}

```

construct 构造相应的二分图 G 。

```

void construct()
{
    int i, j, k;
    fout<<nl<<" "<<t<<endl;
    for(i=1,k=1;i<=n;i++)
        for(j=1;j<=n;j++)
            if(d[i][j]==0) {
                if(ok(i-2,j-1))fout<<c[i][j]-1<<" "<<c[i-2][j-1]-1<<endl;
                if(ok(i-2,j+1))fout<<c[i][j]-1<<" "<<c[i-2][j+1]-1<<endl;
                if(ok(i-1,j-2))fout<<c[i][j]-1<<" "<<c[i-1][j-2]-1<<endl;
                if(ok(i-1,j+2))fout<<c[i][j]-1<<" "<<c[i-1][j+2]-1<<endl;
                if(ok(i+2,j-1))fout<<c[i][j]-1<<" "<<c[i+2][j-1]-1<<endl;
                if(ok(i+2,j+1))fout<<c[i][j]-1<<" "<<c[i+2][j+1]-1<<endl;
                if(ok(i+1,j-2))fout<<c[i][j]-1<<" "<<c[i+1][j-2]-1<<endl;
                if(ok(i+1,j+2))fout<<c[i][j]-1<<" "<<c[i+1][j+2]-1<<endl;
            }
        fout<<"-1<<"-1<<endl;
}
bool ok(int i, int j)
{
    if(i<1||i>n||j<1||j>n||d[i][j]>1)return false;
    else return true;
}

```

comp 计算最大匹配，并输出计算结果。

```

void comp()

```

```

{
    char file[11]="bm.txt";
    GRAPH<EDGE>G(t,1);
    G.readbm(file);
    BMATCHING<GRAPH<EDGE>,EDGE>(G,n1,ans);
    cout<<t-ans<<endl;
}

```

(2) 解法 2: 用线性规划算法求解

设棋盘的第 i 行、第 j 列相应的变量为 x_{ij} 。变量 x_{ij} 的含义是, 棋盘的 (i, j) 方格中放置了 x_{ij} 个骑士, $x_{ij} \in \{0, 1\}$ 。

骑士共存问题的求解目标是 $\sum_{i=1}^m \sum_{j=1}^n x_{ij}$ 达到最大。

约束条件是, 每个骑士不受其他骑士的攻击。

设可以攻击 (i, j) 方格的其他方格的集合是 S_{ij} , 则对任何 $x_{pq} \in S_{ij}$, 有 $x_{pq} + x_{ij} \leq 1$ 。

由此可见, 骑士共存问题可以变换为如下的整数线性规划问题:

$$\begin{aligned}
 & \max \sum_{i=1}^m \sum_{j=1}^n x_{ij} \\
 \text{s. t.} \quad & x_{ij} + x_{pq} \leq 1, x_{pq} \in S_{ij} \\
 & x_{ij} \in \{0, 1\}
 \end{aligned}$$

G 是一个二分图。上述整数线性规划的约束矩阵恰好是二分图 G 的关联矩阵。由已知结论, 任何一个二分图的关联矩阵是一个全 1 模阵, 可知上述整数线性规划的约束矩阵是一个全 1 模阵。因此可以将非线性约束条件 $x_{ij} \in \{0, 1\}$ 松弛为线性约束条件 $x_{ij} \leq 1$ 。从而将整数线性规划问题转化为线性规划问题。用单纯形算法求解。

具体算法实现如下。

read 读入初始数据。

```

void read()
{
    int i, j, k;
    cin >> n >> m;
    Make2DArray(c, n+1, n+1);
    Make2DArray(d, n+1, n+1);
    d[0][0] = 0;
    for(j=1; j<=n; j++) d[j][0] = !d[j-1][0];
    for(i=1; i<=n; i++)
        for(j=1; j<=n; j++) d[i][j] = !d[i][j-1];
    for(k=0; k<m; k++) {
        cin >> i >> j; d[i][j] = 2;
    }
    for(i=1; i<=n; i++)

```



```

        for(j=1;j<=n;j++)
            if(d[i][j]<2)c[i][j]=t++;
            else c[i][j]=0;
t--;
for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
        if(d[i][j]==0){
            if(ok(i-2,j-1))s++;
            if(ok(i-2,j+1))s++;
            if(ok(i-1,j-2))s++;
            if(ok(i-1,j+2))s++;
            if(ok(i+2,j-1))s++;
            if(ok(i+2,j+1))s++;
            if(ok(i+1,j-2))s++;
            if(ok(i+1,j+2))s++;
        }
Make2DArray(e,s+1,2);
for(i=1,k=1;i<=n;i++)
    for(j=1;j<=n;j++)
        if(d[i][j]==0){
            if(ok(i-2,j-1)){e[k][0]=c[i][j];e[k++][1]=c[i-2][j-1];}
            if(ok(i-2,j+1)){e[k][0]=c[i][j];e[k++][1]=c[i-2][j+1];}
            if(ok(i-1,j-2)){e[k][0]=c[i][j];e[k++][1]=c[i-1][j-2];}
            if(ok(i-1,j+2)){e[k][0]=c[i][j];e[k++][1]=c[i-1][j+2];}
            if(ok(i+2,j-1)){e[k][0]=c[i][j];e[k++][1]=c[i+2][j-1];}
            if(ok(i+2,j+1)){e[k][0]=c[i][j];e[k++][1]=c[i+2][j+1];}
            if(ok(i+1,j-2)){e[k][0]=c[i][j];e[k++][1]=c[i+1][j-2];}
            if(ok(i+1,j+2)){e[k][0]=c[i][j];e[k++][1]=c[i+1][j+2];}
        }
}
bool ok(int i,int j)
{
    if(i<1||i>n||j<1||j>n||d[i][j]>1)return false;
    else return true;
}

```

用线性规划算法的成员函数 lpin 建立相应的初始单纯形表。

```

void LinearProgram::lpin(int mima,int mm,int nn,int **e)
{
    int i,j,k,t;
    double value;
    minmax=mima;m=mm+nn,n=nn;
    m1=m;m2=0;m3=0;n1=n;n2=n+m1;

```

```

Make2DArray(a,m+2,n1+1);
basic=new int[m+2];
nonbasic=new int[n1+1];
for (i=0;i<=m+1;i++)
    for (j=0;j<=n1;j++) a[i][j]=0.0;
for (j=0;j<=n1;j++) nonbasic[j]=j;
for (i=1, j=n1+1; i<=m; i++,j++) basic[i]=j;
for (i=m-m3+1, j=n+1; i<=m; i++,j++){
    a[i][j]=-1.0;
    a[m+1][j]=-1.0;
}
for(k=1, t=1; k<=mm; k++, t++){
    a[t][e[k][0]]=1.0; a[t][e[k][1]]=1.0;
}
for(k=1; k<=n; k++, t++) a[t][k]=1.0;
for(j=1; j<=m; j++) a[j][0]=1.0;
for(i=1; i<=n; i++) a[0][i]=minmax;
for (j=1; j<=n; j++){
    for (i=m1+1, value=0.0; i<=m; i++) value+=a[i][j];
    a[m+1][j]=value;
}
}

```

算法的主函数调用单纯形算法求解。

```

int main()
{
    read();
    LinearProgram X;
    X.lpin(l, s, t, e);
    X.solve();
    return 0;
}

```

第9章 NP完全性理论与近似算法

习题9-1 RAM和RASP程序

试写出完成下面计算的RAM和RASP程序:

- (1) 给定输入 n , 计算 $n!$ 。
- (2) 读入 n 个正整数 (用 0 做结束标志), 然后, 按从大到小的顺序输出这 n 个数。

分析与解答:

见参考文献[1], 151~154。

习题9-2 RAM和RASP程序的复杂性

用对数耗费和均匀耗费两种标准分析习题9-1中程序的时间和空间复杂性。

分析与解答:

见参考文献[1], 151~154。

习题9-3 计算 n^n 的RAM程序

写出一个计算 n^n 的RAM程序, 要求该程序在均匀耗费标准下的时间复杂性为 $O(\log n)$, 并证明程序的正确性。

分析与解答:

见参考文献 [1], 151~154。

习题9-4 平面图着色问题的绝对近似算法

设问题P关于实例 I 的精确解为 $c^*(I)$, 解问题P的近似算法A对于实例 I 得到的近似解为 $c(I)$ 。如果存在一常数 k , 使得对于P的任何实例 I 均有 $|c^*(I) - c(I)| \leq k$, 则称算法A是解问题P的绝对近似格式。

平面图的色数问题是对于给定的平面图 $G=(V, E)$, 确定对其顶点着色的最小色数。试设计解平面图着色问题的一个多项式时间绝对近似算法A使得 $|c^*(I) - c(I)| \leq 1$ 。

分析与解答:

对于给定的平面图 $G=(V, E)$, 求 G 的色数的绝对近似算法描述如下。

```
int acolor
{
    if(G的顶点集V为空) return 0;
    if(G的边集E为空) return 1;
    if(G是二分图) return 2;
    return 4;
}
```

由平面图的4色定理, 上述算法只在图 G 是可3着色时, 近似地返回4, 其余情况下均

返回正确解答。因此, 对于上述近似算法有 $|c^*(I) - c(I)| \leq 1$ 。

习题 9-5 最优程序存储问题

设有 n 个程序, $1, 2, \dots, n$, 要存入 2 张容量为 Max 的磁盘中。第 i 个程序需要的存储空间为 $m_i, i=1, 2, \dots, n$ 。设计一个算法计算出这 2 张磁盘能存放的最多程序个数。

- (1) 证明上述问题是 NP 难的。
- (2) 下面的算法 pStore 是解上述问题的一个绝对近似算法。

```
int pStore(int n, int Max, int *m)
{
    sort(m, n); // 将 m 从小到大排序
    int i=1;
    for (int j=1; j<=2; j++) {
        int sum=0;
        while (sum+m[i]<=Max) {
            cout<<" Store program "<< i<< " on disk  "<< j<<endl;
            sum+=m[i];
            if (i==n) return i;
            else i++;
        }
    }
    return i-1;
}
```

试证明对于上述算法 pStore 有 $|c^*(I) - c(I)| \leq 1$ 。

分析与解答:

(1) 将划分问题变换为最优程序存储问题。对于划分问题的任一实例 $\{a_1, a_2, \dots, a_n\}$, 不妨设 $\sum_{i=1}^n a_i = 2T$ 。定义相应的最优程序存储问题的实例如下: $\text{Max}=T; m_i=a_i, 1 \leq i \leq n$ 。显而易见, $\{a_1, a_2, \dots, a_n\}$ 有一个划分, 当且仅当所有 n 个程序能存入 2 张磁盘中。

上述变换只需要线性时间。由划分问题的 NP 完全性可知, 最优程序存储问题是 NP 难的。

(2) 假设算法 pStore 的解答为 c , 而最优值为 c^* 。不失一般性, 设 $m_1 \leq m_2 \leq \dots \leq m_n$ 。由习题 4-8 的结论可知, 如果只有一张容量为 2Max 的磁盘, 按照贪心策略可以得到最优解。假设用贪心策略最多可存放 p 个程序到一张容量为 2Max 的磁盘上。显然, $c^* \leq p$, 且

$$\sum_{i=1}^p m_i \leq 2\text{Max}。$$

设 $j = \max\{k \mid \sum_{i=1}^k m_i \leq \text{Max}\}$, 则 $j \leq p$, $\sum_{i=1}^{j+1} m_i > \text{Max}$ 。算法 pStore 将前 j 个程序存放到第 1 张磁盘上。由 $\sum_{i=j+1}^{p-1} m_i \leq \sum_{i=j+2}^p m_i \leq \text{Max}$ 可知, 算法 pStore 至少将程序 $j+1, \dots, p-1$ 存放到第 2 张磁盘上。由此可见, $c \geq p-1$ 。因此, $|c^* - c| \leq 1$ 。

习题 9-6 树的最优顶点覆盖

设计一个有效的贪心算法,使其能在线性时间内找到一棵树的最优顶点覆盖。

分析与解答:

设给定的树 T 有 n 个顶点。首先对树 T 作前序标号如下。

- (1) 任选一个顶点 r 作为树 T 的根结点;
- (2) 对以 r 为根的树作前序遍历,并且在遍历过程中对访问的顶点依次编号;
- (3) 用数组 parent 记录每个结点的父结点编号。

树 T 的这个表示过程显然可在 $O(n)$ 时间内完成。前序标号表示法实际上是树在前序标号意义下的父亲数组表示法,它具有以下性质:

- ① 对于 $i=2,3,\dots,n$, 有 $i > \text{parent}[i]$, 当 $i=1$ 时, $\text{parent}[i]=1$;
- ② 若将树 T 看作是一般的图 $G=(V,E)$, 则有

$$V=\{1,2,\dots,n\}$$

$$E=\{(i,\text{parent}[i]), i=2,3,\dots,n\}$$

- ③ 对于任意 $j, 2 \leq j \leq n$, 定义树 $T_j=(V_j, E_j)$ 为

$$V_j=\{1,2,\dots,j\}$$

$$E_j=\{(i,\text{parent}[i]), i=2,3,\dots,j\}$$

则 $\text{parent}[1..j]$ 是子树 T_j 的前序标号表示。特别地, $T_n=T$;

- ④ 标号为 j 的结点是 T_j 的叶结点, $j=2,3,\dots,n$ 。特别地, 标号为 n 的结点是 T 的叶结点。

基于树的前序标号表示法,可设计树 T 的最小顶点覆盖的贪心算法 `treecover` 如下。

```
void treecover()
{
    for(int i=1; i<=n; i++) {cover[i]=0; s[i]=0;}
    for(i=n; i>1; i--)
        if((!cover[i]) && (!cover[parent[i]])) s[parent[i]]=1;
}
```

算法 `treecover` 是一个贪心算法。算法中用数组 `cover` 来标记选入覆盖点集的树结点,即当结点 i 被选入覆盖点集,则 $\text{cover}[i]=1$, 否则 $\text{cover}[i]=0$ 。

为了说明算法的正确性,必须证明关于树的最小顶点覆盖问题满足贪心选择性质并具有最优子结构性质。

- (1) 贪心选择性质。对于树 T , 存在一个 T 的最小顶点覆盖 S , 使 $\text{parent}[n] \in S$ 。

事实上,对于 T 的任意一个最小顶点覆盖 S , 若 $\text{parent}[n] \notin S$, 则 $n \in S$, 否则 S 就不是 T 的一个顶点覆盖。在这种情况下,令 $S'=S \cup \{\text{parent}[n]\} - \{n\}$, 则 S' 仍为 T 的一个顶点覆盖,且 $|S'|=|S|$ 。因此 S' 是 T 的一个最小顶点覆盖,且 $\text{parent}[n] \in S'$ 。

- (2) 最优子结构性质。对于 T 的任一最小顶点覆盖 S , 当 $n \in S$ 时, $S - \{n\}$ 是 T_{n-1} 的最小顶点覆盖。

事实上, $S - \{n\}$ 显然是 T_{n-1} 的一个顶点覆盖。若它不是 T_{n-1} 的最小顶点覆盖,则存在 T_{n-1} 的一个更小的顶点覆盖 S' , 且 $|S'| < |S| - 1$ 。 $S' \cup \{n\}$ 显然是 T 的一个顶点覆盖,且 $|S' \cup \{n\}| \leq |S'| + 1 < |S|$, 这与 S 是 T 的一个最小顶点覆盖矛盾。

当 $n \notin S$ 时, 设 $i = \text{parent}[n]$, 则必有 $i \in S$ 。

令 $S_1 = \{j \mid i \leq j \leq n\} \cap S$, 则 $S - S_1$ 是 T_{i-1} 的一个最小顶点覆盖。事实上, 由于 S 是 T 的一个顶点覆盖, 故 $S - S_1$ 是 T_{i-1} 的一个顶点覆盖。若在 T_{i-1} 中有一个比 $S - S_1$ 更小的顶点覆盖 S_{i-1} , 则 $|S_{i-1}| < |S| - |S_1|$ 。显而易见, $S_{i-1} \cup S_1$ 是 T 的一个顶点覆盖, 且 $|S_{i-1} \cup S_1| \leq |S_{i-1}| + |S_1| < |S|$, 这与 S 是 T 的一个最小顶点覆盖相矛盾。

根据上述的贪心选择性质和最优子结构性质, 容易用数学归纳法证明算法 `treecover` 的正确性。

算法的 `for` 循环显然只需要 $O(n)$ 时间, 从而整个算法所需的时间为 $O(n)$ 。

习题 9-7 顶点覆盖算法的性能比

解顶点覆盖问题的一个启发式算法如下: 每次选择具有最高度数的顶点, 然后将与其关联的所有边删去。举例说明该算法的性能比将大于 2。

分析与解答:

对于 $n \geq 3, i \leq n$, 定义 $A(n, i) = \sum_{j=2}^i \left\lfloor \frac{n}{j} \right\rfloor$ 。构造图 $G_n = (V, E)$ 如下:

$$V = \{a_1, \dots, a_{A(n, n-1)}, b_1, \dots, b_n, c_1, \dots, c_n\}$$

$$E = \{(b_i, c_i) \mid i = 1, 2, \dots, n\} \cup \bigcup_{i=2}^{n-1} \bigcup_{j=A(n, i-1)+1}^{A(n, i)} \{(a_j, b_k) \mid (j - A(n, i-1) - 1)i + 1 \leq k \leq (j - A(n, i-1))i\}$$

易见, $|V| = A(n, n-1) + 2n, nH(n-1) - n - (n-2) \leq A(n, n-1) \leq nH(n-1) - n$ 。

将顶点覆盖问题的启发式算法应用于图 $G_n = (V, E)$, 选择顶点的序列为 $a_{A(n, n-1)}, a_{A(n, n-1)-1}, \dots, a_1$ 。此后, 剩下 n 条不相交的边, 必须再选 n 个顶点。因此, 算法选出的覆盖顶点集中顶点数为 $A(n, n-1) + n$ 。而顶点集 $\{b_1, b_2, \dots, b_n\}$ 显然是一个最优顶点覆盖, 其顶点数为 n 。图 9-1 是 G_6 示意图。

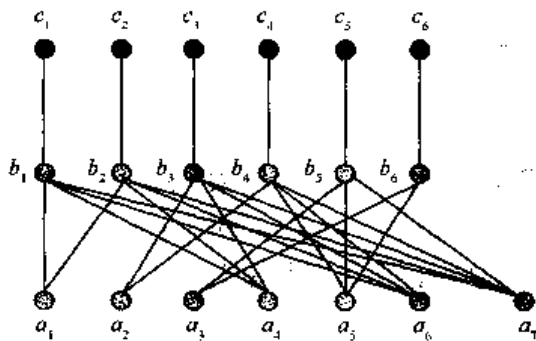


图 9-1 G_6 示意图

由此可见, 启发式算法的性能比 η 满足:

$$\eta \geq (nH(n-1) - n + 2) / n \geq H(n-1) - 1$$

$\lim_{n \rightarrow \infty} H(n-1) = \infty$, 可见启发式算法的性能比 η 可任意大。

习题 9-8 团的常数性能比近似算法

图 G 的最优顶点覆盖是其补图中最大团集的补集。这个关系是否暗示对于团问题也有一个常数性能比的近似算法?

分析与解答:

见参考文献[1], 176。

习题 9-10 旅行售货员问题的常数性能比近似算法

证明旅行售货员问题的一个实例可在多项式时间内变换为该问题的另一个实例, 使得其费用函数满足三角不等式, 且两实例具有相同的最优解。说明是否可以通过这个变换使得一般的旅行售货员问题具有一个常数性能比的近似算法。

分析与解答:

见参考文献[1], 176~177。

习题 9-11 瓶颈旅行售货员问题

瓶颈旅行售货员问题是要找出图 $G=(V,E)$ 的一条哈密顿回路, 且使回路中最长边的长度最小。若费用函数满足三角不等式, 给出解此问题的性能比为 3 的近似算法 (提示: 递归证明中, 可以通过对 G 的最小生成树进行完全遍历并跳过某些顶点, 但不能跳过多于 2 个连续的中间顶点, 以此方式访问最小生成树中每个顶点恰好一次)。

分析与解答:

解瓶颈旅行售货员问题的性能比为 3 的近似算法描述如下。

```
void approxTSP (Graph G)
```

```
{
```

```
    (1) 选择任一顶点  $r \in V$ ;
```

```
    (2) 找出  $G$  的一棵以  $r$  为根的最小瓶颈生成树  $T$ ;
```

```
    (3) 选取  $T$  的一条边  $(p, q)$  作为哈密顿回路  $H$  的一条边;
```

```
    (4) 遍历树  $T$ , 跳过不多于 2 个连续的重复顶点, 递归构造  $H$  的其他边;
```

```
    (5) 将所得到的哈密顿回路  $H$  作为计算结果返回。
```

```
}
```

设 $V=\{1, 2, \dots, n\}$ 。由习题 4-28 的结论可知, 任何一棵最小生成树都是最小瓶颈生成树。因此可用 Prim 算法或 Kruskal 算法构造 (2) 的最小瓶颈生成树 T 。事实上, 可以在线性时间内构造最小瓶颈生成树。

下面讨论步骤 (4) 中, 以最小瓶颈生成树 T 为基础, 递归构造满足要求的哈密顿回路 H 的算法。

当 $n \leq 3$ 时, 容易构造满足要求的哈密顿回路 H 。当 $n > 3$ 时,

(1) 将步骤 (3) 中选取的边 (p, q) 删去, 树 T 分裂成 2 棵树 T_p 和 T_q 。其中, T_p 包含顶点 p , 而 T_q 包含顶点 q 。

(2) 设 (p, p') 是树 T_p 中的一条边 (如果存在); 在树 T_p 中递归地构造以 (p, p') 为一条边的哈密顿回路 H_p 。

(3) 设 (q, q') 是树 T_q 中的一条边 (如果存在)。在树 T_q 中递归地构造以 (q, q') 为一条边的哈密顿回路 H_q 。

(4) 删去 H_p 中的边 (p, p') 得到一条 $(p \rightarrow p')$ 哈密顿路 P_p ; 删去 H_q 中的边 (q, q') 得到一条 $(q' \rightarrow q)$ 哈密顿路 P_q ; 由此可得, q, p, P_p, q', P_q 是满足要求的以 (p, q) 为一条边的哈密

顿回路, 如图 9-2 所示。

由归纳假设知, H_p 和 H_q 中的边满足要求, 即跳过不多于 2 个连续的 T 中顶点。按照上面的构造法, $(p, q), (p, p')$ 和 (q, q') 均为 T 的边。因此, 如果边 (p', q') 不是 T 的边, 它最多也只跳过 p 和 q 这 2 个顶点。由此可见, 所构造的哈密顿回路是以 (p, q) 为一条边的满足要求的哈密顿回路。

下面讨论上述近似算法 approxTSP 的性能比。

设 G 的最小瓶颈生成树 T 中的最长边的长度为 c ; G 最优瓶颈旅行售货员回路 H_{opt} 中的最长边的长度为 c_{opt} ; 由算法 approxTSP 构造出的哈密顿回路为 H , 其最长边的长度为 c_H 。从中任意删去一条边都构成 G 的一棵生成树, 因此有 $c \leq c_{\text{opt}}$ 。设 (p, q) 为 H 的一条边。如果 (p, q) 不是 T 的边, 由 H 的性质可知, (p, q) 与 T 中 2 条边构成一个三角形, 或与 T 中 3 条边构成四边形, 如图 9-3 所示。

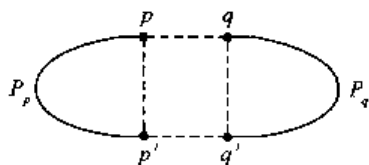


图 9-2 构造哈密顿回路

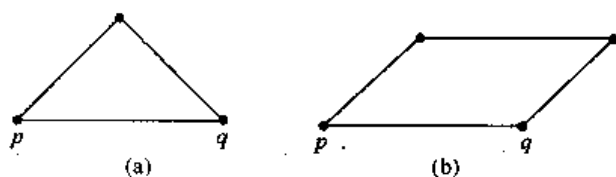


图 9-3 哈密顿回路 H 中的边

由费用函数满足三角不等式可知, 在图 9-3 (a) 的情形有, $\text{dist}(p, q) \leq 2c$; 在图 9-3 (b) 的情形有, $\text{dist}(p, q) \leq 3c$ 。由此可推知, $c_H \leq 3c \leq 3c_{\text{opt}}$ 。也就是说, 算法 approxTSP 的性能比为 3。

习题 9-12 最优旅行售货员回路不自相交

若旅行售货员问题中, 图 G 的各顶点均为平面上的点, 且费用函数 $c(u, v)$ 定义为点 u 和 v 之间的欧氏距离, 证明 G 的最优旅行售货员回路不会自相交。

分析与解答:

见参考文献[1], 177~178。

习题 9-14 集合覆盖问题的实例

试给出一族集合覆盖问题的实例, 用以说明算法 greedySetCover 可以产生的不同解的个数随实例中元素个数 n 的指数增长。这里所说的不同解是指算法 greedySetCover 在作贪心选择时可以有多种选择, 即使 $|S \cap U|$ 最大的子集可有多个时, 不同的选择导致算法的不同的解。

分析与解答:

先看一个简单实例。设 $X = \{1, 2, 3, 4\}$, $F = \bigcup S(i, j)$, 其中,

$S(1, 1) = \{1\}, S(1, 2) = \{2\}, S(1, 3) = \{3\}, S(1, 4) = \{4\};$

$S(2, 1) = \{1, 2\}, S(2, 2) = \{1, 3\}, S(2, 3) = \{1, 4\}, S(2, 4) = \{2, 3\}, S(2, 5) = \{2, 4\}, S(2, 6) = \{3, 4\};$

$S(3, 1) = \{1, 2, 3\}, S(3, 2) = \{1, 2, 4\}, S(3, 3) = \{1, 3, 4\}, S(3, 4) = \{2, 3, 4\}。$

算法 greedySetCover 第 1 次可选择 $S(3, 1), S(3, 2), S(3, 3), S(3, 4)$ 这 4 个集合中的任意一集合。例如, 选择集合 $S(3, 1)$ 后, F 中剩余的集合为

$S(1,4), S(2,3), S(2,5), S(2,6), S(3,2), S(3,3), S(3,4)$

算法 greedySetCover 接下来可选这 7 个集合中的任一集合。由此可见, 对于这个实例, 算法 greedySetCover 可产生 28 个不同的解。

在一般情况下, 设 $X = \{1, 2, \dots, n\}$, $F = \bigcup_{i=1}^n \bigcup_{j=1}^i S(i, j)$, 其中,

$$S(1,1) = \{1\}, S(1,2) = \{2\}, \dots, S(1,n) = \{n\}$$

$$S(2,1) = \{1, 2\}, S(2,2) = \{1, 3\}, \dots, S\left(2, \binom{n}{2}\right) = \{n-1, n\}$$

...

$$S(n-1,1) = \{1, 2, \dots, n-1\}, S(n-1,2) = \{1, 2, \dots, n-1, n\}, \dots, S(n-1,n) = \{2, 3, \dots, n\}$$

算法 greedySetCover 第 1 次可选择 $S(n-1,1), S(n-1,2), \dots, S(n-1,n)$ 这 n 个集合中的任一集合。例如, 选择集合 $S(n-1,1)$ 后, F 中剩余的集合为

$S(1)$ 中 $\binom{n-1}{0}$ 个集合, $S(2)$ 中 $\binom{n-1}{1}$ 个集合, $\dots, S(n-1)$ 中 $\binom{n-1}{n-2}$ 个集合。

总共有 $\sum_{i=0}^{n-2} \binom{n-1}{i} = 2^{n-1} - 1$ 个集合。

算法 greedySetCover 接下来可选这 $2^{n-1} - 1$ 个集合中的任一集合。由此可见, 对于这类实例, 算法 greedySetCover 可产生 $n(2^{n-1} - 1)$ 个不同的解。。

习题 9-16 多机调度问题的近似算法

多机调度问题。设有 m 台完全相同的机器来完成 n 个彼此独立的任务, 第 i 个任务所需的机器时间为 $t_i, i=1, 2, \dots, n$ 。我们需要确定一个时间表, 使全部 n 个任务都结束的时间最短。

解上述问题的最长处理时间算法 LPT 每次从待安排任务中选择最长处理时间的任务, 并安排给一台完全空闲机器。试在 $O(n \log n)$ 时间内实现算法 LPT, 并证明该算法所得到的解的相对误差

$$\lambda = \left| \frac{c^* - c}{c^*} \right| \leq \frac{1}{3} - \frac{1}{3m}$$

分析与解答:

算法描述见主教材 4.7 节多机调度问题的解法。算法先将 n 个任务按照所需的机器时间从大到小排序。不失一般性, 设 $t_1 \leq t_2 \leq \dots \leq t_n$ 。对于多机调度问题的一个具体实例 x , 设算法 LPT 计算出的完成时间为 $T_{LPT}(x)$, 最优解的完成时间为 $T_{opt}(x)$ 。用 $s(i) = j$ 记录 LPT 算法将第 i 个任务分配给第 j 台机器。

下面讨论解的相对误差。

用反证法。设解的相对误差

$$\lambda = \left| \frac{c^* - c}{c^*} \right| > \frac{1}{3} - \frac{1}{3m}$$

则存在多机调度问题的最小实例 I , 使得

$$\left| \frac{T_{LPT}(I) - T_{opt}(I)}{T_{LPT}(I)} \right| > \frac{1}{3} - \frac{1}{3m}$$

由 I 是多机调度问题的最小实例, 可推知任务 n 的完成时间是 $T_{\text{LPT}}(I)$ 。事实上, 若任务 n 的完成时间不是 $T_{\text{LPT}}(I)$, 则算法对任务集 $I' = \{t_1, t_2, \dots, t_{n-1}\}$ 计算得到的完成时间也是 $T_{\text{LPT}}(I)$ 。又由于 I' 是 I 的子集, 故 $T_{\text{opt}}(I') \leq T_{\text{opt}}(I)$ 。由此可得

$$\begin{aligned} T_{\text{LPT}}(I') - T_{\text{opt}}(I') &\geq T_{\text{LPT}}(I) - T_{\text{opt}}(I) \\ &> \left(\frac{1}{3} - \frac{1}{3m}\right) T_{\text{opt}}(I) \\ &\geq \left(\frac{1}{3} - \frac{1}{3m}\right) T_{\text{opt}}(I') \end{aligned}$$

这与 I 是多机调度问题的满足

$$\left| \frac{T_{\text{LPT}}(I) - T_{\text{opt}}(I)}{T_{\text{LPT}}(I)} \right| > \frac{1}{3} - \frac{1}{3m}$$

的最小实例矛盾。

下面证明 $T_{\text{opt}}(I) < 3t_n$ 。

设安排第 n 个任务前, m 台机器的状态为 $\{T_1, T_2, \dots, T_m\}$, $T_k = \min\{T_1, T_2, \dots, T_m\}$ 。算法选择 $s(n) = k$ 。从前面的讨论可知, 任务 n 的完成时间是 $T_{\text{LPT}}(I)$ 。因此, $T_k + t_n = T_{\text{LPT}}(I)$ 。

另一方面, $T_k = \min\{T_1, T_2, \dots, T_m\} \leq \sum_{i=1}^m T_i / m$ 。由此可得

$$\begin{aligned} \sum_{i=1}^{n-1} \frac{t_i}{m} &= \sum_{i=1}^m \frac{T_i}{m} \\ &\geq T_k \\ &= T_{\text{LPT}}(I) - t_n \end{aligned}$$

因此

$$\sum_{i=1}^n t_i \geq mT_{\text{LPT}}(I) - (m-1)t_n$$

由于每个任务都要在一台机器上完成, 所以 $T_{\text{opt}}(I) \geq \sum_{i=1}^n \frac{t_i}{m}$ 。由此可得

$$\begin{aligned} \left(\frac{m-1}{m}\right)t_n &\geq T_{\text{LPT}}(I) - T_{\text{opt}}(I) > \left(\frac{m-1}{3m}\right)T_{\text{opt}}(I) \\ T_{\text{opt}}(I) &< 3t_n \end{aligned}$$

而当 $T_{\text{opt}}(I) < 3t_n$ 时, LPT 算法得到的是最优解。事实上, 由 $T_{\text{opt}}(I) < 3t_n$ 知, 每台机器上的任务数不超过 2, 否则 $T_{\text{opt}}(I) \geq 3t_n$ 。因此, $n \leq 2m$ 。不失一般性, 可以假设 $n = 2m$ 。因为如果 $n < 2m$, 可以加入 $2m - n$ 个任务使 $t_{n+1} = \dots = t_{2m} = 0$ 。

由算法 LPT 可知, 任务 j 和任务 $2m - j + 1$ 被安排在机器 j 上, $1 \leq j \leq m$ 。设 $j = \max\{i \mid t_i + t_{2m-i+1} = T_{\text{LPT}}(I)\}$ 。另外设最优解将任务 i 分配给机器 $s_{\text{opt}}(i)$ 。构造顶点集为 $V = \{1, 2, \dots, n\}$ 的图 G 如下。当 $s(i) = s(k)$, 即 $i + k = 2m + 1$ 时, 加入蓝边 (i, k) ; 当 $s_{\text{opt}}(i) = s_{\text{opt}}(k)$ 时, 加入红边 (i, k) 。不论是最优解还是算法 LPT 的解, 都在每台机器上恰好安排两个任务。因此红边和蓝边组成的图的每个顶点的度数恰为 2, 从而图的每个连通分支都是一个简单圈。考察图中包含顶点 j 的连通分支。设它包含的顶点为 $\{j_1, \dots, j_l, 2m - j_1 + 1, \dots, 2m - j_l + 1\}$, $1 \leq j_1, \dots, j_l \leq m$ 。由于红边匹配了该圈中所有顶点, 因此存在红边 (i, k) 满足 $i \leq j, k \leq 2m - j + 1$ 。由 j 的定义知

$$T_{\text{opt}}(I) \geq t_i + t_k \geq t_j + t_{2m-j+1} = T_{\text{LPT}}(I)$$

可见

$$\begin{aligned} T_{\text{opt}}(I) &= T_{\text{LPT}}(I) \\ \left| \frac{T_{\text{LPT}}(I) - T_{\text{opt}}(I)}{T_{\text{LPT}}(I)} \right| &= 0 \end{aligned}$$

这与 I 的定义矛盾。从而证明了不存在多机调度问题的实例 I ，使得

$$\left| \frac{T_{\text{LPT}}(I) - T_{\text{opt}}(I)}{T_{\text{LPT}}(I)} \right| > \frac{1}{3} - \frac{1}{3m}$$

换句话说，对于多机调度问题的任何实例 I ，均有

$$\left| \frac{T_{\text{LPT}}(I) - T_{\text{opt}}(I)}{T_{\text{LPT}}(I)} \right| \leq \frac{1}{3} - \frac{1}{3m}$$

即算法所得到的解的相对误差

$$\lambda = \left| \frac{c^* - c}{c^*} \right| \leq \frac{1}{3} - \frac{1}{3m}$$

习题 9-17 LPT 算法的最坏情况实例

设 $n = 2m + 1$ 且 $t_i = 2m - \lfloor i + 1/2 \rfloor, 1 \leq i \leq 2m, t_{2m+1} = m$ 。试构造多机调度问题关于该实例的最优解 c^* 和用算法 LPT 求出的解 c ，并计算近似算法 LPT 的性能比

$$\eta = \left| \frac{c^* - c}{c^*} \right|$$

分析与解答：

对于所给实例，用近似算法 LPT 得到的解如下：

任务 i 和 $2m - i + 1$ 安排在第 i 台机器， $1 \leq i < m$ ；安排在第 m 台机器上的是任务 $m, m + 1$ 和 $2m + 1$ 。完成时间是

$$\begin{aligned} t_m + t_{m+1} + t_{2m+1} &= 2m - \left\lfloor \frac{m+1}{2} \right\rfloor + 2m - \left\lfloor \frac{m+2}{2} \right\rfloor + m \\ &= 4m - 1 \end{aligned}$$

所给实例的最优解是：

任务 i 和 $2m - i - 1$ 安排在第 i 台机器， $1 \leq i < m$ ；安排在第 m 台机器上的是任务 $2m - 1, 2m$ 和 $2m + 1$ 。完成时间是 $3m$ 。

由此可知

$$\begin{aligned} \eta &= \left| \frac{c^* - c}{c^*} \right| \\ &= \frac{4m - 1 - 3m}{3m} \\ &= \frac{m - 1}{3m} \\ &= \frac{1}{3} - \frac{1}{3m} \end{aligned}$$

可见该实例使算法 LPT 达到解的相对误差的上界 $\frac{1}{3} - \frac{1}{3m}$ 。

习题 9-18 多机调度问题的多项式时间近似算法

设在多机调度问题中，要在所给 m 台机器上安排的 n 个任务已按各自所需处理时间的

递减序列排列 $t_1 \geq t_2 \geq \dots \geq t_n$ 。解此问题的算法 LPT2 先确定一个正整数 k ，对前 k 个任务求最优安排，然后对后 $n-k$ 个任务用算法 LPT（习题 9-16）求解。

(1) 试证明算法 LPT2 的解的相对误差

$$\lambda \leq \frac{1-1/m}{1+\lfloor k/m \rfloor}$$

(2) 根据 (1) 的结论，设计一个解多机调度问题的多项式时间近似算法，对于给定的 $\epsilon > 0$ ，算法所需的计算时间为 $O(n \log n + m^{m/\epsilon})$ 。

分析与解答：

(1) 设算法 LPT2 计算出的时间表长度为 T_{LPT2} ，最优时间表长度为 T_{opt} ，前 k 个任务的最优时间表长度为 t 。如果 $T_{LPT2} = t$ ，则命题成立。因此可设 $T_{LPT2} > t$ 。设任务 j 的完成时间为 $T_{LPT2}, j > k$ 。在时间 $T_{LPT2} - t_j$ 处，所有机器非空闲。因此， $\sum_{i=1}^{j-1} t_i/m \geq T_{LPT2} - t_j$ 。结合 $T_{opt} \geq \sum_{i=1}^j t_i/m$ 可知

$$T_{LPT2} - T_{opt} \leq \frac{m-1}{m} t_j \leq \frac{m-1}{m} t_{k+1}$$

由于 $t_i \geq t_{k+1}, 1 \leq i \leq k+1$ ，由鸽舍原理知， m 台机器中，至少有一台机器上安排的任务数不少于 $1 + \lfloor k/m \rfloor$ 。因此， $T_{opt} \geq (1 + \lfloor k/m \rfloor) t_{k+1}$ 。

结合前面的讨论即知

$$\frac{T_{LPT2} - T_{opt}}{T_{opt}} \leq \frac{(m-1)/m}{1 + \lfloor k/m \rfloor} = \frac{1-1/m}{1 + \lfloor k/m \rfloor}$$

(2) 对于给定的 $\epsilon > 0$ ，取 $k = \lfloor (m-1)/\epsilon \rfloor$ ，可使

$$\frac{1-1/m}{1 + \lfloor k/m \rfloor} < \epsilon$$

算法 LPT2 用 $O(m^k)$ 时间求前 k 个任务的最优时间表，算法其余部分所需的计算时间不超过 $O(n \log n)$ 。由此可见，算法 LPT2 是解多机调度问题的 ϵ 近似算法，所需的计算时间为 $O(n \log n + m^{m/\epsilon})$ 。

算法实现题 9-1 旅行售货员问题的近似算法（习题 9-9）

★问题描述：

主教材中解旅行售货员问题的近似算法 approxTSP 可以进一步得到改进。由近似算法 $\eta=2$ 的证明过程容易看出，如果将 G 的最小生成树 T 的边看作是 G 的双重边，则回路 W 就是 T 的一个欧拉回路。而近似最优哈密顿回路是在这条欧拉回路中删除第 2 次经过的顶点得到的。如果基于 T 找出一条更短的欧拉回路，则可以得到一条更短的哈密顿回路。下面的算法框架就是基于这个思想来设计的。

```
void approxTSP (Graph G)
```

```
{
```

- (1) 选择任一顶点 $r \in V$;
- (2) 用 PRIM 算法找出 G 的一棵以 r 为根的最小生成树 T ;
- (3) 找出 T 的奇数度顶点集 S ;

- (4) 在以 S 为顶点集的 G 的完全子图中, 找出一个最小完全匹配 M ;
 - (5) 在以 T 和 M 中所有边集组成的多重图中, 找出一条欧拉回路;
 - (6) 将找到的欧拉回路, 除根 r 外第 2 次经过的顶点删去, 得到一条哈密顿回路 H ;
 - (7) 将所得到的哈密顿回路 H 作为计算结果返回。
- }

上述算法是解 TSP 问题的 $O(n^3)$ 时间近似算法, 且其性能比达到 1.5。

★编程任务:

设计并实现上述近似算法。

★数据输入:

由文件 input.txt 提供输入数据。文件第 1 行有 2 个正整数 n 和 e , n 表示 G 的顶点数; e 是 G 的边数。接下来的 e 行中, 每行有 3 个正整数 i, j, c , 表示边 (i, j) 的费用为 c 。

★结果输出:

程序运行结束时, 将近似最优哈密顿回路及其长度输出到文件 output.txt。文件的第 1 行是近似最优哈密顿回路的长度, 第 2 行是近似最优哈密顿回路。

输入文件示例	输出文件示例
input.txt	output.txt
7 8	31
1 4 5	1 4 2 6 5 3 7
4 2 8	
2 6 3	
6 5 1	
5 3 3	
3 7 2	
7 1 9	
1 5 10	

分析与解答:

(1) 算法正确性

算法的关键步骤是第(4)步, 在 G 的奇数度顶点集 S 的完全子图中, 找出一个最小完全匹配 M 。设 $\deg(v)$ 是顶点 v 在树 T 中的度数, 则

$$2|T| = \sum_{v \in V} \deg(v) = \sum_{v \in S} \deg(v) + \sum_{v \in V-S} \deg(v)$$

对任意 $v \in V-S$, 有 $\deg(v)$ 为偶数, 故 $\sum_{v \in V-S} \deg(v)$ 为偶数, 从而 $\sum_{v \in S} \deg(v)$ 也是偶数。

又由于对任意 $v \in S$, $\deg(v)$ 为奇数, 故 $|S|$ 为偶数。因此, 第(4)步中的完全匹配 M 存在。图 $T+M$ 中各边的度数均为偶数, 因此, $T+M$ 的欧拉回路也是存在的。由此可见算法的第(5)步可找到 $T+M$ 的欧拉回路, 并在第(6)步中根据所找出的欧拉回路构造出 G 的一条哈密顿回路。

(2) 算法的计算复杂性

算法的步骤(2)中 PRIM 算法需要 $O(n^2)$ 时间。步骤(3)显然只需要 $O(n)$ 时间。步骤(4)可在 $O(n^3)$ 时间内找出最小完全匹配 M 。算法的步骤(5)和步骤(6)均可在 $O(n)$

时间内完成。因此,整个算法所需的计算时间为 $O(n^3)$ 。

(3) 算法的精度

设 G 的一个最优旅行售货员回路为 $H^* : v_1, v_2, \dots, v_n, v_1$, 并设 S 中的顶点为 $v_{i_1}, v_{i_2}, \dots, v_{i_k}, i_1 < i_2 < \dots < i_k, k = |S|$ 。

由此可知, $M_1 = \{(v_{i_{2j-1}}, v_{i_{2j}}) | 1 \leq j \leq k/2\}$ 和 $M_2 = \{(v_{i_{2j}}, v_{i_{2j+1}}) | 1 \leq j \leq k/2\}$ 是以 S 为顶点集的完全图的 2 个完全匹配。因此

$$C(M_1) + C(M_2) = \sum_{j=1}^{k/2} C(v_{i_{2j-1}}, v_{i_{2j}}) + C(v_{i_{2k}}, v_{i_1}) \leq \sum_{j=1}^{n-1} C(v_j, v_{j+1}) + C(v_n, v_1)$$

其中用到三角不等式性质

$$C(v_{i_j}, v_{i_{j+1}}) \leq C(v_j, v_{i_{j+1}}) + \dots + C(v_{i_{j+1}-1}, v_{i_{j+1}})$$

因此

$$C(M) \leq \min\{C(M_1), C(M_2)\} \leq C(H^*)/2$$

又由于 $C(T) \leq C(H^*)$, 故

$$C(T+M) = C(T) + C(M) \leq 1.5C(H^*)$$

从而 $C(H) \leq C(T+M) \leq 1.5C(H^*)$ 。由此即知, 所述算法的 η 值为 1.5。

存在 TSP 问题的实例, 使上述算法的近似最优值与最优值的比任意接近 1.5。

算法实现题 9-2 可满足问题的近似算法 (习题 9-19)

★问题描述:

设 α 是一个含有 n 个变量和 m 个合取项的合取范式。关于 α 的最大可满足性问题要求确定 α 的最多个数的合取式, 并使这些合取式可同时满足。设 k 是 α 的所有合取式中因子个数的最小值。证明下面的解最大可满足问题的近似算法 mSAT 的相对误差为 $\frac{1}{k+1}$ 。

Set mSAT(α)

{// $x_i, 1 \leq i \leq n$, 是 α 中 n 个变量; $C_i, 1 \leq i \leq m$, 是 α 的 m 个合取项。

cl = \emptyset ;

left = $\{C_i | 1 \leq i \leq m\}$;

lit = $\{x_i, \bar{x}_i | 1 \leq i \leq n\}$;

while (lit 含有在 left 的合取式中出现的因子) {

 设 y 是 lit 在 left 的合取式中出现次数最多的因子;

 设 r 是 left 中含有因子 y 的所有合取式的集合;

 cl = cl \cup r ;

 left = left - r ;

 lit = lit - $\{y, \bar{y}\}$;

}

return (cl);

}

★编程任务:

设计并实现上述近似算法。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 k 和 m , 分别表示变量数和布尔表达式数。接下来的 m 行中, 每行有若干整数 $i, j, k, \dots, 0$, 表示表达式所含的变量下标分别为 i, j, k, \dots , 行末以 0 结尾。下标为负数时, 表示相应的变量为取反变量。

★结果输出:

程序运行结束时, 将计算出的最大可满足合取式数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5 3	3
3 1 4 0	
1 -5 3 0	
2 -5 1 0	

分析与解答:

算法在题中已描述。算法所需的时间为 $O(n \log n)$ 。下面证明近似算法 mSAT 的相对误差为 $\frac{1}{k+1}$ 。

设算法在 while 循环体内选中因子 y , r 是 left 中含有因子 y 的所有合取式的集合。left 中仅含因子 \bar{y} 的合取项 b 未选中, 但其所含的因子 \bar{y} 已从 lit 中删去, 此时称合取项 b 被击中 1 次。留在 left 中的被击中的合取式的个数不超过选入 r 中合取式的个数。算法结束时仍留在 left 中的合取式的所有因子均已从 lit 中删除。这意味着算法结束时, 至少有 $k | \text{left} |$ 次击中。因此算法结束时返回的可满足合取式的个数 $| \text{cl} | \geq k | \text{left} |$ 。因此, 若最优值为 opt , 则有

$$\begin{aligned} \text{opt} &\leq m = | \text{cl} | + | \text{left} | \leq (1 + 1/k) | \text{cl} | \\ \frac{\text{opt} - | \text{cl} |}{\text{opt}} &= 1 - \frac{| \text{cl} |}{\text{opt}} \leq 1 - \frac{k}{k+1} = \frac{1}{k+1} \end{aligned}$$

这个相对误差是可达的。例如, 当 $k=3$ 时, 设给定的合取范式为

$$\alpha = (\bar{x}_1 + x_4 + x_5)(\bar{x}_2 + x_6 + x_7)(\bar{x}_3 + x_8 + x_9)(x_1 + x_2 + x_3)$$

取 $x_1 = x_4 = x_6 = x_8 = \text{true}$, 可使 4 个合取式都满足, 因此, $\text{opt} = 4$ 。而算法 mSAT 在选取 $\bar{x}_1 = \bar{x}_2 = \bar{x}_3 = \text{true}$ 后, 使合取式 $(x_1 + x_2 + x_3)$ 不满足。可见

$$\frac{\text{opt} - | \text{cl} |}{\text{opt}} = \frac{4 - 3}{4} = \frac{1}{4}$$

算法实现题 9-3 最大可满足问题的近似算法 (习题 9-20)

★问题描述:

证明下面的解最大可满足问题的近似算法 mSAT2 的相对误差为 $1/2^k$, k 是 α 的所有合取式中因子个数的最小值。

Set mSAT2(α)

 // $x_i, 1 \leq i \leq n$, 是 α 中 n 个变量; $c_i, 1 \leq i \leq m$, 是 α 的 m 个合取项。

 for (int $i=1; i \leq m; i++$) $w[i] = 2^{-|c_i|}$;

$\text{cl} = \emptyset$;

```

left = {ci | 1 ≤ i ≤ m};
lit = {xi,  $\bar{x}_i$  | 1 ≤ i ≤ n};
while (lit 含有在 left 的合取式中出现的因子) {
    设 y 是 lit 在 left 的合取式中出现的因子;
    设 r 是 left 中含有因子 y 的所有合取式的集合;
    设 s 是 left 中含有因子  $\bar{y}$  的所有合取式的集合;
    if (  $\sum_{c_i \in r} w[i] \geq \sum_{c_i \in s} w[i]$  ) {
        cl = cl ∪ r; left = left - r;
        对所有 ci ∈ s, w[i] = 2 * w[i];
    }
    else { cl = cl ∪ s; left = left - s;
        对所有 ci ∈ r, w[i] = 2 * w[i];
    }
    lit = lit - {y,  $\bar{y}$ };
}
return (cl);
}

```

★编程任务:

设计并实现上述近似算法。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 k 和 m , 分别表示变量数和布尔表达式数。接下来的 m 行中, 每行有若干整数 $i, j, k, \dots, 0$, 表示表达式所含的变量下标分别为 i, j, k, \dots , 行末以 0 结尾。下标为负数时, 表示相应的变量为取反变量。

★结果输出:

程序运行结束时, 将计算出的最大可满足合取式数输出到文件 output.txt。

输入文件示例

输出文件示例

input.txt

output.txt

5 3

3

3 1 4 0

1 -5 3 0

2 -5 1 0

分析与解答:

算法在题中已描述过。算法所需的时间为 $O(n \log n)$ 。下面证明近似算法 mSAT2 的相对误差为 $1/2^k$ 。

初始时, $\sum_{i=1}^m w[i] \leq m/2^k$ 。在算法的 while 循环体内, 每次迭代留在 left 中的被击中的合取式增加的权值不超过选入 r 中合取式的权值。因此 left 中合取式的总权值不增加, 从而在算法结束时, left 中合取式的总权值不超过 $m/2^k$ 。另一方面, 在算法结束时, 留在 left 中的每个合取式的权值为 1。由此可知, 在算法结束时, $|\text{left}| \leq m/2^k$ 。因此, 算法结束时返回的可满足合取式的个数 $|\text{cl}| = m - |\text{left}| \geq m(1 - 1/2^k)$ 。

若最优值为 opt, 则有

$$\text{opt} \leq m \leq \frac{2^k}{2^k - 1} |\text{cl}|$$

$$\frac{\text{opt}-|\text{cl}|}{\text{opt}} = 1 - \frac{|\text{cl}|}{\text{opt}} \leq 1 - \frac{2^k-1}{2^k} = \frac{1}{2^k}$$

这个相对误差是可达的。例如，当 $k=3$ 时，设给定的合取范式为

$$\alpha = (\bar{x}_1 + x_4 + x_5)(\bar{x}_1 + x_6 + x_7)(\bar{x}_1 + x_8 + x_9)(\bar{x}_1 + x_{10} + x_{11}) \\ (x_1 + x_2 + x_3)(x_1 + \bar{x}_2 + x_3)(x_1 + x_2 + \bar{x}_3)(x_1 + \bar{x}_2 + \bar{x}_3)$$

取 $x_1 = x_4 = x_6 = x_8 = x_{10} = \text{true}$ ，可使 8 个合取式都满足，因此， $\text{opt} = 8$ 。而算法 mSAT2 在选取 $\bar{x}_1 = \text{true}$ 后，使含有 x_1 的 4 个合取式之一不满足。可见

$$\frac{\text{opt}-|\text{cl}|}{\text{opt}} = \frac{8-7}{8} = \frac{1}{8}$$

算法实现题 9-4 子集和问题的近似算法（习题 9-15）

★问题描述：

子集和问题的一个实例为 $\langle S, t \rangle$ 。其中， $S = \{x_1, x_2, \dots, x_n\}$ 是一个正整数的集合， t 是一个正整数。子集和问题判定是否存在 S 的一个子集 S_1 ，使得 $\sum_{x \in S_1} x = t$ 。

在实际应用中，常遇到最优化形式的子集和问题。在这种情况下，要找出 S 的一个子集 S_1 ，使得其和不超过 t ，又尽可能地接近 t 。

★编程任务：

对于给定的子集和问题的一个实例 $\langle S, t \rangle$ ，设计一个算法找出 S 的一个子集 S_1 ，使得其和不超过 t ，又尽可能地接近 t 。

★数据输入：

由文件 input.txt 提供输入数据。文件第 1 行有 2 个正整数 n 和 t ， n 表示 S 的大小， t 是子集和的目标值。接下来的 1 行中，有 n 个正整数，表示集合 S 中的元素。

★结果输出：

程序运行结束时，将子集和的最优值输出到文件 output.txt。

输入文件示例

输出文件示例

input.txt

output.txt

3 7

6

1 4 5

分析与解答：

对主教材中算法做适当修改如下。

```
int maxsum()
{
    int max=0;
    list<int> t1;
    list<int>::iterator p,q;
    t1.push_back(0);
    for(int i=0;i<n;i++){
        list<int> t2(t1);
        for(p=t2.begin();p!=t2.end();p++){*p+=s[i];
```

```

        t1.merge(t2);
        p=t1.begin();
        while(p!=t1.end()){
            if(*p>t) p=t1.erase(p);
            else p++;
        }
    }
    for(p=t1.begin();p!=t1.end();p++) if(*p>max) max=*p;
    return max;
}

```

算法实现题 9-5 子集和问题的完全多项式时间近似算法

★问题描述:

子集和问题的一个实例为 $\langle S, t \rangle$ 。其中, $S = \{x_1, x_2, \dots, x_n\}$ 是一个正整数的集合, t 是一个正整数。子集和问题判定是否存在 S 的一个子集 S_1 , 使得 $\sum_{x \in S_1} x = t$ 。

在实际应用中, 常遇到最优化形式的子集和问题。在这种情况下, 要找出 S 的一个子集 S_1 , 使得其和不超过 t , 又尽可能地接近 t 。

★编程任务:

对于给定的子集和问题的一个实例 $\langle S, t \rangle$, 设计一个完全多项式时间近似算法找出 S 的一个子集 S_1 , 使得其和不超过 t , 又尽可能地接近 t 。

★数据输入:

由文件 input.txt 提供输入数据。文件第 1 行有 2 个正整数 n 和 t , n 表示 S 的大小, t 是子集和的目标值。接下来的 1 行中, 有 n 个正整数, 表示集合 S 中的元素。

★结果输出:

程序运行结束时, 将子集和的最优值输出到文件 output.txt。文件的第 1 行是 n 和 t 的值。第 2 行是计算出的近似最优值。

输入文件示例

输出文件示例

input.txt

output.txt

17 100

17 100

10 8 8 5 5 6 3 6 2 9 2 10 100

10 4 9 3 6

分析与解答:

对主教材中算法做适当修改如下。

```

int maxsum(double delta)
{
    int max=0;
    list<int> t1;
    list<int>::iterator p;
    t1.push_back(0);

```

```

for(int i=0;i<n;i++){
    list<int> t2(t1);
    for(p=t1.begin();p!=t1.end();p++){*p+=s[i];
    t2.merge(t1);
    t1.resize(0);
    p=t2.begin();
    int last=*p;
    t1.push_back(last);
    while(p!=t2.end()){
        if(*p<=t && (double)last<(1-delta)*(double)(*p)){last=*p;t1.push_back(last);} *p++;
    }
}
for(p=t1.begin();p!=t1.end();p++) if(*p>max) max=*p;
return max;
}

```

算法实现题 9-6 2-SAT 问题的线性时间算法

★问题描述:

SAT 的一个实例是 k 个布尔变量 x_1, x_2, \dots, x_k 的 m 个布尔表达式 A_1, A_2, \dots, A_m 。若存在各布尔变量 $x_i (1 \leq i \leq k)$ 的 0,1 赋值, 使每个布尔表达式 $A_i (1 \leq i \leq m)$ 都取值 1, 则称布尔表达式 $A_1 A_2 \dots A_m$ 是可满足的。

(1) 合取范式的可满足性问题 CNF-SAT

如果一个布尔表达式是一些因子和之积, 则称为合取范式, 简称 CNF (Conjunctive Normal Form)。这里的因子是变量 x 或 \bar{x} 。例如 $(x_1 + x_2)(x_2 + x_3)(\bar{x}_1 + \bar{x}_2 + x_3)$ 就是一个合取范式, 而 $x_1 x_2 + x_3$ 就不是合取范式。

(2) 2-SAT

如果一个布尔合取范式的每个乘积项最多是 2 个因子的析取式, 就称为二元合取范式, 简记为 2-CNF。一个 2-SAT 问题是判定一个 2-CNF 是否可满足。

★编程任务:

对于给定的 2-CNF, 设计一个线性时间算法, 判定其是否可满足。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 k 和 m , 分别表示变量数和布尔表达式数。接下来的 m 行中, 每行有 2 个整数 i 和 j , 表示相应表达式含的变量下标分别为 i 和 j 。下标为负数时, 表示相应的变量为取反变量。

★结果输出:

将判定结果输出到文件 output.txt。若给定的 2-CNF 可满足则输出 “Yes”, 否则输出 “No”。

输入文件示例

输出文件示例

input.txt

output.txt

4 6

Yes

1 3

-1 3

1 -2

-3 4

1 -4

2 -3

分析与解答:

设二元合取范式为 $\theta = \prod_{i=1}^m (\alpha_i + \beta_i)$, 其中, $\alpha_i, \beta_i \in \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}, i=1, 2, \dots, m$ 。

对于所给的二元合取范式 θ , 构造相应的有向图 $G=(V, E)$ 如下。

变量 x_i 对应于图 G 的顶点 v_{2i-1} ; 变量 \bar{x}_i 对应于图 G 的顶点 v_{2i} , $i=1, 2, \dots, n$, 因此, $|V|=2n$ 。

每一个合取项 $\alpha_i + \beta_i$ 对应于 E 中 2 条有向边 (u, v) 和 (s, t) 如下。

$$(u, v) = \begin{cases} (v_{2j}, v_{2k-1}) & (\alpha_i, \beta_i) = (x_j, x_k) \\ (v_{2j}, v_{2k}) & (\alpha_i, \beta_i) = (x_j, \bar{x}_k) \\ (v_{2j-1}, v_{2k-1}) & (\alpha_i, \beta_i) = (\bar{x}_j, x_k) \\ (v_{2j-1}, v_{2k}) & (\alpha_i, \beta_i) = (\bar{x}_j, \bar{x}_k) \end{cases}$$

$$(s, t) = \begin{cases} (v_{2k}, v_{2j-1}) & (\alpha_i, \beta_i) = (x_j, x_k) \\ (v_{2k-1}, v_{2j-1}) & (\alpha_i, \beta_i) = (x_j, \bar{x}_k) \\ (v_{2k}, v_{2j}) & (\alpha_i, \beta_i) = (\bar{x}_j, x_k) \\ (v_{2k-1}, v_{2j}) & (\alpha_i, \beta_i) = (\bar{x}_j, \bar{x}_k) \end{cases}$$

显而易见, $|E|=2m$ 。边 $(v_i, v_j) \in E$ 表示顶点 v_i 相应的变量取值为 1, 则顶点 v_j 相应的变量也应取值为 1, 否则 $\theta=0$ 。

对于给定的二元合取范式 θ , 可以在 $O(m+n)$ 时间内构造出上述有向图 G 。可以证明, θ 是可满足的当且仅当图 G 中不存在形如 $x_i \rightarrow \dots \rightarrow \bar{x}_i \rightarrow \dots \rightarrow x_i$ 这样的圈。而图 G 中的这种圈可以通过求 G 的强连通分支来检测。用求图的强连通分支的算法, 可在 $O(m+n)$ 时间内找出图 G 的所有强连通分支。对于每对变量 x_i 和 \bar{x}_i 判定是否属于 G 的同一个强连通分支。若属于 G 的同一个强连通分支, 则说明 G 中存在 $x_i \rightarrow \dots \rightarrow \bar{x}_i \rightarrow \dots \rightarrow x_i$ 的圈, 因此 θ 不可满足。若 G 的每一个强连通分支都不存在这样的圈, 则 θ 是可满足的。上述判定显然只需 $O(n)$ 时间。

综上所述, 二元合取范式 θ 的可满足性可以在 $O(m+n)$ 时间内判定。

具体算法实现如下。

```
int main()
{
    int n, m, x, y;
    cin >> n >> m;
    GRAPH G(2*n, 1);
    for(int i=0; i<m; i++){
```

```

        cin>>x>>y;
        G.insert(Edge(f1(x),f2(y)));
        G.insert(Edge(f1(y),f2(x)));
    }
    SC<GRAPH>comp(G);
    for(i=0;i<n;i++) if(comp.stronglyreachable(2*i,2*i+1)) {cout<<"No"<<endl;
    return 0;}
    cout<<"Yes"<<endl;
    return 0;
}

```

其中,函数 f1 和 f2 定义如下。

```

int f1(int x)
{
    if(x<0) return -2*x-2;
    else return 2*x-1;
}

int f2(int x)
{
    if(x<0) return -2*x-1;
    else return 2*x-2;
}

```

SC<GRAPH>是关于图 G 的强连通分支类。

```

template <class inGraph, class outGraph>
void reverse(const inGraph &G, outGraph &R)
{
    for(int v=0;v<G.V();v++){
        typename inGraph::adjIterator A(G,v);
        for(int w=A.beg();!A.end();w=A.nxt())
            R.insert(Edge(w,v));
    }
}

```

```

template <class Graph>class SC
{ const Graph &G;
    int ent, sent;
    vector<int> postI, postR, id;
    void dfsR(const Graph &G, int w)
    {
        id[w]=sent;

```

```

        typename Graph::adjIterator A(G, w);
        for(int t=A.beg();!A.end();t=A.nxt())
            if(id[t]==-1) dfsR(G, t);
        postI[cnt++] = w;
    }
public:
    SC(const Graph &G) : G(G), cnt(0), scnt(0),
        postI(G.V()), postR(G.V()), id(G.V(), -1)
    {
        Graph R(G.V(), 1);
        reverse(G, R);
        for(int v=0;v<R.V();v++)
            if(id[v]==-1) dfsR(R, v);
        postR=postI;cnt=scnt=0;
        id.assign(G.V(), -1);
        for(v=G.V()-1;v>=0;v--)
            if(id[postR[v]]==-1)
                { dfsR(G, postR[v]);scnt++;}
    }
    int count() const{return scnt;}
    void out() {for(int v=0;v<G.V();v++) cout<<id[v]<<" ";cout<<endl;}
    bool stronglyreachable(int v, int w) const
    {return id[v]==id[w];}
};

```

算法实现题 9-7 实现算法 greedySetCover (习题 9-13)

★问题描述:

集合覆盖问题的一个实例 $\langle X, F \rangle$ 由一个有限集 X 及 X 的一个子集族 F 组成。子集族 F 覆盖了有限集 X 。也就是说, X 中每一元素至少属于 F 中的一个子集, 即 $X = \bigcup_{S \in F} S$ 。对于 F 中的一个子集 $C \subseteq F$, 若 C 中的 X 的子集覆盖了 X , 即 $X = \bigcup_{S \in C} S$, 则称 C 覆盖了 X 。集合覆盖问题就是要找出 F 中覆盖 X 的最小子集 C^* , 使得 $|C^*| = \min\{|C| \mid C \subseteq F \text{ 且 } C \text{ 覆盖 } X\}$ 。

设计并实现算法 greedySetCover, 使其计算时间为 $O\left(\sum_{S \in F} |S|\right)$ 。

★编程任务:

实现集合覆盖问题的近似算法 greedySetCover。

★数据输入:

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m , 分别表示有限集 X 中元素个数和子集族 F 中子集个数。 $X = \{0, 1, \dots, n-1\}$, $F = \{f_0, f_1, \dots, f_{m-1}\}$ 。接下来的 m 行中, 每行对应于 F 中一个子集 f_i 。第一个数是子集 f_i 中元素个数 k_i , 接着的 k_i 个数表示 f_i 中的元素。

★结果输出:

程序运行结束时, 将计算出的最小覆盖子集输出到文件 output.txt。第 2 行是最小覆盖

子集数。第 2 行是最小覆盖子集。

输入文件示例

输出文件示例

input.txt

output.txt

12 6

4

6 0 1 2 3 4 5

0 4 2 1

4 0 3 6 9

4 1 4 7 10

4 4 5 7 8

4 2 5 8 11

2 9 10

分析与解答：

设给定的有限集为 $X=\{0,1,\cdots,n-1\}$ ； X 的子集族为 $F=\{f_0,f_1,\cdots,f_{m-1}\}$ 。

建立集合表 F ，顶点表 V 和集合秩表 R 如下。

$F[i]$ 存储集合 f_i 中的元素， $0\leq i<m$ ； $V[i]$ 存储包含元素 i 的集合， $0\leq i<n$ ； $R[i]$ 存储当前集合大小为 i 的集合， $0\leq i\leq n$ 。用双链表存储 $R[i]$ 中的集合。

(1) 算法从 $R[n]$ 开始，扫描集合秩表 R ，取出当前剩余元素最多的集合 $F[i]$ 。

(2) 对于 $F[i]$ 中每个元素 j ，根据 $V[j]$ 存储的每个集合 k ，将 $F[k]$ 中的元素 j 删去，并修改集合 $F[k]$ 在 R 中的位置。

具体算法描述如下。

用结构 `fType` 表示 F 中元素信息。

```
struct fType
{
    int E[MAXELE];
    int size, rank;
    DNode<int>*p;
};
```

其中，`size` 表示相应集合中元素个数；`rank` 表示当前集合中元素个数；`p` 是指向集合在 R 的双链表中元素的指针，用于实现 R 的双链表中元素的 $O(1)$ 时间删除运算。数组 `E` 存放集合中的元素。

用结构 `vType` 表示 V 中元素信息。

```
struct vType
{
    int E[MAXELE];
    int size;
};
```

其中，`size` 表示包含相应元素的集合个数；数组 `E` 存放相应集合。

下面是算法中用到的变量。

```

int n,m,cn=0,C[MAXSET],U[MAXELE],FU[MAXSET];
Double<int>*R;
vType V[MAXELE];
fType F[MAXSET];

```

以上变量 n 表示集合 X 中元素个数。变量 m 表示子集族 F 中集合个数。 $C[\text{MAXSET}]$ 存储解集, cn 是解集 C 中集合个数。 $U[\text{MAXELE}]$ 存储当前未删除元素。 $FU[\text{MAXSET}]$ 存储当前未选择集合。数组 R 存储集合秩表, $R[i]$ 存储当前集合大小为 i 的集合组成的双链表。数组 F 存储集合中的元素。数组 V 存储包含元素的集合信息。

init 读入初始数据, 并建立表 F , V 和 R , 初始化 U 和 FU 。

```

void init()
{
    cin>>n>>m;
    R=new Double<int>[n+1];
    for(int i=0;i<n;i++){U[i]=1;V[i].size=0;}
    for(i=0;i<m;i++){
        FU[i]=1;
        cin>>P[i].size;F[i].rank=F[i].size;
        for(int k=0,j=0;k<F[i].size;k++){
            cin>>j;F[i].E[k]=j;
            V[j].E[V[j].size++]=i;
        }
    }
    for(j=0;j<m;j++){
        R[F[j].rank].INSERT(0,i);
        F[j].p=R[F[j].rank].First();
    }
}

```

算法主体 greedySetCover 描述如下。

```

bool greedySetCover()
{
    int i,j,t,ei,si,ti,fi,vi,k=n,total=n;
    while(total>0 && k>0){
        if(! R[k].EMPTY()){
            R[k].DELETE(1,si);FU[si]=0;
            total-=k;
            C[cn++]=si;        // 将集合 si 加入解集 C
            // 将 F[si]中的每个元素从 F 中其他集合中删除。
            for(i=0,t=F[si].size;i<t;i++){
                vi=F[si].E[i];        // F[si]中的元素
                if(U[vi]){

```



```

        for(j=0,ei=V[vi].size;j<ei;j++){
            ti=V[vi].E[j];    //包含元素 vi 的集合
            if(FU[ti]){
                fi=F[ti].rank;
                R[fi].DELETE(F[ti].p);    // 用 O(1)时间删除 F[ti]
                if(fi>1){
                    R[fi-1].INSERT(0,ti);    // 用 O(1)时间插入 F[ti]
                    F[ti].p=R[fi-1].First();    // 保存指向 F[ti]的指针
                    F[ti].rank--;
                }
            }
            U[vi]=0;
        }
    }
    else k--;
}
return total==0;
}

```

实现算法的主函数如下。

```

int main()
{
    init();
    if(greedySetCover())out();
    else cout<<"No Solution!"<<endl;
    return 0;
}

```

out 输出计算结果。

```

void out()
{
    cout<<cn<<endl;
    for(int i=0;i<cn;i++)cout<<C[i]<<" ";
    cout<<endl;
}

```

算法主体 greedySetCover 的关键之处是可以用 $O(1)$ 时间将集合从 R 的双链表中删除；将集合插入 R 的双链表中也只要 $O(1)$ 时间。在最坏情况下，算法访问 F 的集合中每个元素一次，每次耗费 $O(1)$ 时间。因此，在最坏情况下算法计算时间为 $O(\sum_{S \in F} |S|)$ 。

参 考 文 献

- 1 王晓东, 傅清祥, 叶东毅. 算法与数据结构学习指导与习题解析. 北京: 电子工业出版社, 2000
- 2 王晓东. 数据结构与算法设计. 北京: 电子工业出版社, 2001
- 3 王晓东. 计算机算法设计与分析 (第 2 版). 北京: 电子工业出版社, 2004

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可,复制、销售或通过信息网络传播本作品的行为;歪曲、篡改、剽窃本作品的行为,均违反《中华人民共和国著作权法》,其行为人应承担相应的民事责任和行政责任,构成犯罪的,将被依法追究刑事责任。

为了维护市场秩序,保护权利人的合法权益,我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为,本社将奖励举报有功人员,并保证举报人的信息不被泄露。

举报电话:(010)88254396;(010)88258888

传 真: (010)88254397

E-mail: dbqq@phei.com.cn

通信地址:北京市万寿路173信箱

电子工业出版社总编办公室

邮 编:100036

