目 录

第一篇 入门篇

练就坚实的基础，总有一天……

我们可以草木皆兵！

第1章 新手入门

1 ACM 国际大学生程序设计竞赛简介

1.1 背景与历史

1970 年在美国 TexasA&M 大学举办了首次区域竞赛，从而拉开了国际大学生程序设计竞赛的序幕。1977 年，该项竞赛被分为两个级别，即区域赛和总决赛，这便是现代 ACM 竞赛的开始。在亚洲、美国、欧洲、太平洋地区均设有区域赛点。1995 至 1996 年，来自世界各地的一千多支高校的代表队参加了 ACM 区域竞赛。ACM 大学生程序设计竞赛由美国计算机协会（ACM）举办，旨在向全世界的大学生提供一个展示和锻炼其解决问题和运用计算机能力的机会，现已成为全世界范围内历史最悠久、规模最大的大学生程序设计竞赛。

1.2 竞赛组织

竞赛在由各高等院校派出的 3 人一组的队伍间进行，分两个级别。参赛队应首先参加每年 9 月至 12 月在世界各地举行的“区域竞赛(Regional Contest)”。各区域竞赛得分最高的队伍自动进入第二年 3 月在美国举行的“决赛(Final Contest)”，其它的高分队伍也有可能被邀请参加决赛。每个学校有一名教师主管队伍，称为“领队”(faculty advisor)，他负责选手的资格认定并指定或自己担任该队的教练(coach)。每支队伍最多由三名选手(contestant)组成，每个选手必须是正在主管学校攻读学位的学生。每支队伍最多允许有一名选手具有学士学位，已经参加两次决赛的选手不得再参加区域竞赛。

1.3 竞赛形式与评分办法

竞赛进行 5 个小时，一般有 6~10 道试题，由同队的三名选手使用同一台计算机协作完成。当解决了一道试题之后，将其提交给评委，由评委判断其是否正确。若提交的程序运行不正确，则该程序将被退回给参赛队，参赛队可以进行修改后再一次提交该问题。

程序运行不正确是指出现以下 4 种情况之一：

(1) 运行出错 (run-time error)；

(2) 运行超时 (time-limit exceeded)；

(3) 运行结果错误 (wrong answer)；

(4) 运行结果输出格式错误 (presentation error)。

竞赛结束后，参赛各队以解出问题的多少进行排名，若解出问题数相同，按照总用时的长短排名。总用时为每个解决了的问题所用时间之和。一个解决了的问题所用的时间是竞赛开始到提交被接受的时间加上该问题的罚时（每次提交通不过，罚时 20 分钟）。没有解决的问题不记时。美国英语为竞赛的工作语言。竞赛的所有书面材料（包括试题）将用

美国英语写出，区域竞赛中可以使用其它语言。总决赛可以使用的程序设计语言包括PASCAL，C，C++及 Java，也可以使用其它语言。具体的操作系统及语言版本各年有所不同。

1.4 竞赛奖励情况

总决赛前十名的队伍将得到高额奖学金：第一名奖金为 12000 美元，第二名奖金为6000 美元，第三名奖金为 3000 美元，第四名至第十名将各得到 l500 美元。除此之外还将承认北美冠军、欧洲冠军、南太平洋冠军及亚洲冠军。

2 ACM 竞赛需要的知识

2.1 编程语言是最重要的基本功

无论侧重于什么方面，只要是通过计算机程序去最终实现的竞赛，编程语言都是大家要过的第一道关。亚洲赛区的比赛支持的语言包括 C/C++与 Java。首先说说 Java，众所周知，作为面向对象的王牌语言，Java 在大型工程的组织与安全性方面有着自己独特的优势，但是对于信息学比赛的具体场合，Java 则显得不那么合适，它对于输入输出流的操作相比于 C++要繁杂很多，更为重要的是 Java 程序的运行速度要比 C++慢 10 倍以上，而竞赛中对于 Java 程序的运行时限却往往得不到同等比例的放宽，这无疑对算法设计提出了更高的要求，是相当不利的。其实，并不主张大家在这种场合过多地运用面向对象的程序设计思维，因为对于小程序来说这不旦需要花费更多的时间去编写代码，也会降低程序的执行效率。

接着说 C 和 C++。在赛场上使用纯 C 的选手还是大有人在的，它们主要是看重了纯 C 在效率上的优势，所以这部分同学如果时间有限，并不需要急着去学习新的语言，只要提高了自己在算法设计上的造诣，纯 C 一样能发挥巨大的威力。

而 C++相对于 C，在输入输出流上的封装大大方便了编程操作，同时降低了出错的可能性，并且能够很好地实现标准流与文件流的切换，方便了调试的工作。如果有些同学比较在意这点，可以尝试 C 和 C++的混编，毕竟仅仅学习 C++的流操作还是不花什么时间的。

C++的另一个支持来源于标准模版库(STL)，库中提供的对于基本数据结构的统一接口操作和基本算法的实现可以缩减我们编写代码的长度，这可以节省一些时间。但是，与此相对的，使用 STL 要在效率上做出一些牺牲，对于输入规模很大的题目，有时候必须放弃 STL，这意味着我们不能存在“有了 STL 就可以不去管基本算法的实现”的想法；另外，熟练和恰当地使用 STL 必须经过一定时间的积累，准确地了解各种操作的时间复杂度，切忌对 STL 中不熟悉的部分滥用，因为这其中蕴涵着许多初学者不易发现的陷阱。

通过以上的分析，我们可以看出仅就信息学竞赛而言，对语言的掌握并不要求十分全面，但是对于经常用到的部分，必须十分熟练，不允许有半点不清楚的地方.

2.2 以数学为主的基础知识十分重要

虽然被定性为程序设计竞赛，但是参赛选手所遇到的问题更多的是没有解决问题的思

路，而不是有了思路却死活不能实现，这就是平时积累的基础知识不够。有一年 World Final 的总冠军是波兰华沙大学，其成员出自于数学系而非计算机系，这就是一个鲜活的例子。竞赛中对于基础学科的涉及主要集中于数学，此外对于物理、电路、生物、化学、医学等也可能有一定应用，但是不多。因此，大一的同学也不必为自己还没学数据结构而感到不知从何入手提高，把数学捡起来吧！下面我来谈谈在竞赛中应用的数学的主要分支。

离散数学

离散数学作为计算机学科的基础是竞赛中涉及最多的数学分支，重中之重又在于图论和组合数学，尤其是图论。图论之所以运用最多是因为它的变化最多，而且可以轻易地结合基本数据结构和许多算法的基本思想，较多用到的知识包括连通性判断、DFS 和 BFS，关节点和关键路径、欧拉回路、最小生成树、最短路径、二部图匹配和网络流等等。虽然这部分的比重很大，但是往往也是竞赛中的难题所在，如果有初学者对于这部分的某些具体内容暂时感到力不从心，也不必着急，可以慢慢积累。

组合数学

竞赛中设计的组合计数问题大都需要用组合数学来解决，组合数学中的知识相比于图论要简单一些，很多知识对于小学上过奥校的同学来说已经十分熟悉，但是也有一些部分需要先对代数结构中的群论有初步了解才能进行学习。组合数学在竞赛中很少以难题的形式出现，但是如果积累不够，任何一道这方面的题目却都有可能成为难题。

数论

数论以素数判断和同余为模型构造出来的题目往往需要较多的数论知识来解决，这部分在竞赛中的比重并不大，但只要来上一道，也足以使知识不足的人冥思苦想上一阵时间。素数判断和同余最常见的是在以密码学为背景的题目中出现，在运用密码学常识确定大概的过程之后，核心算法往往要涉及数论的内容。

计算几何

计算几何相比于其它部分来说是比较独立的，就是说它和其它的知识点很少有过多的结合，较常用到的部分包括线段相交的判断、多边形面积的计算、内点外点的判断、凸包等等。计算几何的题目难度不会很大，但也永远不会成为最弱的题。

线性代数

对线性代数的应用都是围绕矩阵展开的，一些表面上是模拟的题目往往可以借助于矩阵来找到更好的算法。

2.3 计算机专业知识

虽然数学十分十分重要，但是如果让三个只会数学的人参加比赛，我相信多数情况下会比三个只会数据结构与算法的人得到更为悲惨的结局。

数据结构

掌握队列、堆栈和图的基本表达与操作是必需的，至于树，我个人觉得需要建树的问题有但是并不多。（但是树往往是很重要的分析工具）除此之外，排序和查找并不需要对所有方式都能很熟练的掌握，但你必须保证自己对于各种情况都有一个在时间复杂度上满足最低要求的解决方案。说到时间复杂度，就又该说说哈希表了，竞赛时对时间的限制远远多于对空间的限制，这要求大家尽快掌握“以空间换时间”的原则策略，能用哈希表来存储的数据一定不要到时候再去查找，如果实在不能建哈希表，再看看能否建二叉查找树等等，这都是争取时间的策略，掌握这些技巧需要大家对数据结构尤其是算法复杂度有比较全面的理性和感性认识。

算法。

算法中最基本和常用的是搜索，主要是回溯和分支限界法的使用。这里要说的是，有些初学者在学习这些搜索基本算法是不太注意剪枝，这是十分不可取的，因为所有搜索的题目给你的测试用例都不会有很大的规模，你往往察觉不出程序运行的时间问题，但是真正的测试数据一定能过滤出那些没有剪枝的算法。实际上参赛选手基本上都会使用常用的搜索算法，题目的区分度往往就是建立在诸如剪枝之类的优化上。

常用算法中的另一类是以“相似或相同子问题”为核心的，包括递推、递归、贪心法和动态规划。这其中比较难于掌握的就是动态规划，如何抽象出重复的子问题是很多题目的难点所在，笔者建议初学者仔细理解图论中一些以动态规划为基本思想所建立起来的基本算法（比如 Floyd-Warshall 算法），并且多阅读一些定理的证明，这虽然不能有什么直接的帮助，但是长期坚持就会对思维很有帮助。

3 团队配合

通过以上的介绍大家也可以看出，信息学竞赛对于知识面覆盖的非常广，想凭一己之力全部消化这些东西实在是相当困难的，这就要求我们尽可能地发挥团队协作的精神。同组成员之间的熟练配合和默契的形成需要时间，具体的情况因成员的组成不同而不同，这里我就不再多说了。

4 练习、练习、再练习

知识的积累固然重要，但是信息学终究不是看出来的，而是练出来的，这是多少前人最深的一点体会，只有通过具体题目的分析和实践，才能真正掌握数学的使用和算法的应用，并在不断的练习中增加编程经验和技巧，提高对时间复杂度的感性认识，优化时间的分配，加强团队的配合。总之，在这里光有纸上谈兵是绝对不行的，必须要通过实战来锻炼自己。

练习站点推荐：

(1) [acm.pku.edu.cn/OnlineJudge] POJ 收集了大量比赛真题，其中不乏简单题；POJ 的服务器性能良好，响应速度快；POJ 经常举办网上练习赛，练习赛是增长比赛经验的最好途径。

(2) [acm.zju.edu.cn] ZOJ 是国内最早出现的 OJ，有一定的权威性。ZOJ 的论坛是最好的资源，提供了“题目分类”，可以进行专题练习。

(3) [acm.sgu.ru] SGU 是俄罗斯的。比较注重算法和数学。OI 的顶尖高手都在这里做题。

学习资料推荐：

1. 算法导论（英文版）Introduction to Alogrithms.
2. 算法艺术与信息学竞赛 刘汝佳 黄亮 著
3. 历年信息学奥赛中国国家队论文
4. ACM 国际大学生程序设计竞赛试题与解析
5. 组合数学的算法与程序设计
6. 图论的算法与程序设计
7. 实用算法的分析与程序设计
8. 计算几何基础知识

5 对新手的一些建议

首先要看一些基础的算法书籍，把基本的算法搞懂。像递归、二分、宽搜、深搜、简单的图论、数论、简单的组合数学。重点根据书上的例题理解算法的实质，思想，能做到有一定领悟。这时需要做一些题目来巩固了。

先可以做搜索题，搜索是博大精深的，诸多细节技巧都需要靠平时的积累领悟，根据自己练习的目的挑一些题练习。然后可以做简单的数学题，对组合数学、数论有个大致的概念。

再然后可以做 DP 类题目了。DP 也是非一日之功，练好 DP 就像练好了内功，这时可以做一些 DP 的基础题，体会一下，然后做一些提高题，如果不会做，一定要自己想通为什么别人这样设定状态数组，他的技巧在哪里。oibh 上很多的国家集训队关于 DP 的论文是必看的。

图论里有很多基础的东西需要学习，先把图论里面基本的定义看懂，然后把经典的算法看懂，比如最短路、生成树、割点、连通分量等等。如果不会做，一定要好好看书。

很多新手会问碰到不会做的题目怎么办。首先应该考察一下为什么不会做这题，如果是书本上的知识点没掌握，那要赶紧把书本找来，仔细理解之后再来想这题。如果知识点基本都掌握了，那么可以利用网络的资源，多搜索一下关于这题的讨论，看看别人是怎么想的，看是否可以给自己提供思路。总之一条，要自己多开动脑子。重在理解这一题的算法，而不是只知道算法，自己把它编程实现了就算了。对待算法和程序要用严谨的态度，没有搞懂的地方要花力气把它搞懂，这样才能不断提高。

看书是必须的，而且也是迅速提高的最好方法，不要等到做题时才去理解书上的知识点，而要对知识点有了充分的理解后再去做题，这样才能事半功倍，否则看到难题，从哪方面下手的思路都没有。

高级的贪心，300 行的宽搜，A*，STL，诸多的剪枝技巧，统计，查找，treap，对 DP 状态的优化，带集合的 DP，平面图，计算几何，数论.....要学的东西很多。但我相信只要努力你们肯定会取得不错的成绩。和别人比赛，其实是和自己比赛，考场上完全是实力的体现，会就是会，不会就是做不出来。训练时间每个人情况不一样，长短不一，但重在看个人悟性，水平达到一定程度之后会发现有质的变化，理解算法简直是小菜。

这里强调的还是个思维能力的问题，不是为了做题而做题，做题其实是为了训练自己的思维能力和编程能力，从训练中能得到的最大收获就是提升了思维，套用比较流行的一个词就是“脑力”。这也是为什么说进省队是个标志，进了省队说明你前期有了一定的积累，和那里的一些高手一接触自然自己的思路就大大开阔了，对于算法会有一个更深层次的理解。就算只参加了省队的选拔赛，对自己的帮助也是很大的。用 gush 的话说，没进过省队就等于没见过世面。

那大家一定达不到那些现今强人的水平吗？当然不是。强人不是天生就强的，也是从菜鸟做起的，成功的原因只有一个——勤奋。他们的思维能力或者脑力不是天生就这样的。但随便提现今的一个牛人，题量都是上千。他们默默地积累和严谨的态度才取得了现在的成绩。有人说上千题，太恐怖了啊，我做一个题都得花几小时，有些想几天还做不出。一开始自然是这样，知识点众多，考查范围广大，对这些现成的知识要慢慢消化，每个知识点都掌握后，做只考这些知识点的题自然就快了。积累到一定程度后就会发现做题的乐趣，以前很崇拜那些说“今天上课太累了，做几道题休息一下”的人，不知不觉，做题对我来说也成了最大的乐趣。有些题只考查单一的知识点，有些题把几个知识点结合起来考查。比如先用平面图里面的几个知识点，然后凸包求一下，然后 DP 一下，或者线性方程组解

一下再搞个匹配等等，这些题看上去很复杂，但若这些知识点都不折不扣地掌握的话，做这些题自然就像切菜了。题目也是人出的，如果只看现成的这些知识点的话，出题者的思维也是有限的。

参加这个比赛对编程能力的提高也是大有好处的，十分钟上百行无错代码，快速实现逻辑较复杂的算法，debug 技巧.....而且对语言的理解也能上好几个台阶。我们还没有迎来下一次的编程技术革命，编程仍旧是有局限的，它强迫我们像计算机一样思考，而不是令计算机像我们的大脑一样思考，这是我们需要花巨大的努力去克服的。很多人对我说的这句话可能不太理解，至少感受不深。如果 Debug 的时间超过写程序时间的 1/2，那就是失败的。一切都要慢慢训练，持之以恒之后，拥有良好的编程习惯和风格应该是每个人所追求的。编程的境界永无止境！

以前人们常说看书可以升级大脑，对于计算机及其相关专业的同学来说，参加 acm 比赛是最好的升级大脑的方法，思维能力的提升可以让我们受益无穷，而编程的能力和技巧则会因为编写大量的代码而大幅提高。或许今后再也没有这样一个机会能让你计算机水平飞速增长。对算法和数据结构理解深入后，研究计算机专业的其他课程有如“会当凌绝顶，一览众山小”。所以低年级的同学全身心的投入进来是绝对有好处的。有一年多的积累就能小有成绩了。只要坚持下来，踏踏实实，努力提升自身水平，一定可以实现自己的目标！

做 acm 看似是枯燥，但一旦入了门，就会发现其中有无穷的乐趣，即使训练了不参加比赛，对自己也是一个很好的提高。

第2章 C++语言介绍

1 C++简介

在 C++之前先有 C 语言，C++是建立在 C 语言之上的，称为“带类的 C 语言”。这个 C 语言基础在当今的 C++程序中仍然很重要。C++并不是取代 C，而是补充和支持 C。本林！章主要介绍 C++中来源于 C 语言的部分。

始使用 C++时，你可能会遇到内存溢出和访问失效等问题，使程序死机。这里用最简短的篇幅介绍 C++语言基础。C++语言本身有专著介绍，这种书还特别厚，所以别指望我能用三言两语说清楚。读者学完本书并使用 C++ Builder 一般时间之后，最后对 C++语言再作更深入的了解。

C++可以最充分地利用面向对象编程(OOP)的优势。OOP 不只是一个新名词，而有它的实际意义，可以生成可复用的对象。新术语 对象(object)，和前面介绍的构件一样，是完成特定编程任务的软件块（构件是对象，但对象不全是构件，稍后会解释这点）。对象只向用户（使用对象的编程人员）显示必须的部分，从而简化对象的使用。用户不必知道的所有内部机制都隐藏在幕后。这一切都包括在面向对象编程的概念中。OOP 可以用模块化方法进行编程，从而避免每次从头开始。C++ Builder 程序是面向 OOP 的，因为 C++ Builder 大量使用构件。生成构件后（你生成的或 C++ Builder 内置的构件），就可以在任何 C++ Builder 程序中重复使用。构件还可以扩展，通过继承生成具有新功能的新构件。最妙的是，构件隐藏了所有内容细节，使编程人员能集中精力充分利用构件。

2 变量

还是从变量讲起来吧。变量(variable)实际上是赋予内存地址的名称。声明变量后，就可以用它操作内存中的数据。下面举几个例子进行说明。下列码段用了两个变量，每条语句末尾用说明语句描述执行该语句时发生的情况：

```
int x;// variable declared as an integer variable
x = 100;// ‘x’ now contains the value 100
x +=50;// ‘x’ now contains the value 150
int y = 150;// ‘y’ declared and initialized to 150
x += y;// ‘x’ now contains the value 300
x++;// ‘x’ now contains the value 301
```

新术语 变量(variable)是留作存放某个数值的计算机内存地址。注意 x 的值在变量操作时会改变，稍后会介绍操作变量的 C++操作符。

警告 声明而未初始化的变量包含随机值。由于变量所指向的内存还没有初始化，所以不知道该内存地址包含什么值。

例如，下列代码

```
int k;
int y;
```

x=y+10; //oops!

本例中变量 y 没有事先初始化，所以 x 可能取得任何值。例外的情况是全局变量和用 static 修饰声明的变量总是初始化为 0。而所有其它变量在初始化或赋值之前包含随机值。变量名可以混合大写、小写字母和数字与下划线（_），但不能包含空格和其它特殊字符。变量名必须以字母或下划线开始。一般来说，变量名以下划线或双下划线开始不好。变量名允许的最大长度随编译器的不同而不同。如果变量名保持在 32 个字符以下，则绝对安全。实际中，任何超过 20 个字符的变量名都是不实用的。

下例是有效变量名的例子：

int aVeryLongVariableName;// a long variable name

int my_variable;// a variable with an underscore

int_ x;// OK,but not advisedint X;// uppercase variable name

int Labe12;// a variable name containing a number

int GetItemsInContainer(); // thanks Pete!

说明 C++中的变量名是考虑大小写的，下列变量是不同的：int XPos;int xpos;如果你原先所用语言不考虑大小写（如 Pascal），则开始接触考虑大小写的语言可能不太适应。

3 C++数据类型

新术语 C++数据类型定义编译器在内存中存放信息的方式。在有些编程语言中，可以向变量赋予任何数值类型。例如，下面是 BASIC 代码的例子：x = 1;x = 1000;x = 3.14;x = 457000;在 BASIC 中，翻译器能考虑根据数字长度和类型分配空间。而在 C++，则必须先声明变量类型再使用变量：int x1 = 1;int x = 1000;float y = 3.14;long z = 457000;这样，编译器就可以进行类型检查，确保程序运行时一切顺利。数据类型使用不当会导致编译错误或警告，以便分析和纠正之后再运行。有些数据类型有带符号和无符号两种。带符号（signed）数据类型可以包含正数和负数，而无符号（unsigned）数据类型只能包含正数。表 2.1 列出了 C++中的数据类型、所要内存量和可能的取值范围。

表 2.1 C++数据类型(32 位程序)

数据类型	字节	数取值范围
char	1	-128 到 126
unsigned char	1	0 到 255
unsigned short	2	0 到 65,535
long	4	-2,147,483,649 到 2,147,483,648
unsigned long	4	0 到 4,294,967,295
int	4	同 long
unsigned int	4	同 unsigned long
float	4	1.2E-38 到 3.4E381
double	8	2.2E-308 到 1.8E3082
bool	1	true 或 false

从上表可以看出，int 与 long 相同。那么，为什么 C++还要区分这两种数据类型呢?实际上这是个遗留问题。在 16 位编程环境中，int 要求 2 个字节而 long 要求 4 个字节。而在 32 位编程环境中，这两种数据都用 4 个字节存放。C++Builder 只生成 32 位程序，所以 int 与 long 相同。说明 在 C++ Builder 和 BorLand C++ 5.0 中，Bool 是个真正的数据类型。

有些 C++编译器有 Bool 关键字，则 Bool 不是个真正的数据类型。有时 Bool 只是个 typedef，使 Bool 等价于 int。typedef 实际上建立别名，使编译器在一个符号与另一符号间划上等号。typedef 的语法如下：typedef int Bool;这就告诉编译器:Bool 是 int 的别名。说明 只有 double 和 float 数据类型使用浮点数（带小数点的数）。其它数据类型只涉及整数值。尽管 integer 数据类型也可以指定带小数点的数值，但小数部分会舍弃，只将整数部分赋予整型变量，例如：int x=3.75;得到的 x 取值为 3。注意，这个整数值并不是四舍五入，而是放弃小数部分。顺便说一句，大多数 Windows 程序很少用到浮点数。C++可以在必要时进行不同数据类型间的换算。例如：short result;long num1 = 200;long num2 = 200;result = num1 * num2;这里我想将两个长整型的积赋予一个短整型。尽管这个公式混用了两种数据类型，但 C++能够进行换算。计算结果会怎样呢?结果会让你大吃一惊，是 25536，这是绕接(wrop)的结果。从表 2.1 可以看出，短整型的最大取值为 32767，在最大值之上加 1 会怎么样呢?得到的是 32768。这实际上与汽车里程计从 99999 回到 00000 的道理一样。为了说明这点，请输入并运行下列清单 1.3 中包含的程序。

清单 1.3Wrapme.cpp

```
#include <iostream.h>
using namespace std;

int main()
{
    short x = 32767;
    cout << “ x = “ << x << endl;
    x++;
    cout << “ x = “ << x << endl;
    return 0;
}
```

分析输出结果为：x=32767 x=32768 如果用 int 数据类型，则不会有这个问题，因为 int 数据类型的取值范围在正向 20 亿之间，一般不会有绕回的问题。但这时程序可能会稍大一些，因为 int 需要 4 字节存储，而 short 只需要 2 字节存储。对于大多数应用程序，这个差别是不显著的。前面介绍了自动类型换算。有时 C++无法进行换算，这时可能在编译器中产生编译错误，说 Cannot convert from x to y(无法从 x 换算到 Y)。编译器也可能警告说 Conversion might lose significant digits（换算可能丢失显著位）。提示 编译器警告应当作编译器错误，因为它表明出了错误。我们应努力生成无警告的编译。有时警告无法避免，但一定要认真检查所有警告。应充分了解警告的原因并尽量予以纠正。

4 C++操作符

操作符(operator)用于操作数据。操作符进行计算、检查等式、进行赋值、操作变量和进行其它更奇怪的工作。C++中有许多操作符，这里不想列出全部，只列出最常用的操作符.常用 C++操作符操作符说明举例

算术运算符：

+	加	x=y+z;
-	减	x=y-z;
*	乘	x=y*z;
/	除	x=y / z;

赋值运算符：

=	赋值	x=10;
+=	赋值与和	x+=10;(等 于 x=x+10;)
-=	赋值与减	x-=10;
*=	赋值与乘	x*=10;
\=	赋值与除	x\=10;
&=	赋值位与	x&=0x02;
|=	赋值位或	x|=0x02;

逻辑操作符：

&&	逻辑与	if(x && 0xFF) {...}
||	逻辑或	if(x || 0xFF) {...}

等式操作符

==	等于	if(x == 10) {...}
!=	不等于	if(x != 10) {...}
<	小于	if(x < 10) {...}
>	大于	if(x > 10) {...}
<=	小于或等于	if(x <= 10) {...}
>=	大于或等于	if(x >= 10) {...}

一元操作符：

*	间接操作符	int x=*y;
&	地址操作符	int* x=&y;
～	位非	x &=～0x02;
!	逻辑非	if(!valid) {...}
++	递增操作符	x++ （ 等 于 x=x+1;）
--	递减操作符	x--;

类和结构操作符：

::	范围解析	MyClass:: SomeFunction();
->	间接成员	MyClass-> SomeFunction();
·	直接成员	MyClass . SomeFunction();

可以看出，这个清单长了些，没法一下子记住。使用 C++时，你会慢慢熟悉这些操作符的。必须指出，递增操作符既可用作前递增(++x)，也可用作后递增(x++)。前递增操作符告诉编译器先递增再使用变量，而后递增操作符则让编译器先使用变量值再递增。例如下列代码：

```
int x = 10;
cout << “x = “ << x++ << end1;
cout << “x = “ << x << end1;
cout << “x = “ x << end1;
cout << “x = “ << ++x << end1;
```

输出结果如下：

```
x=10
x=11
x=12
x=12
```

递减操作符也是这样，这里不想将这些内容讲得太深，但读者可以耐心阅读下去，正如彭兹对奥古斯特所说，“奥古，耐心点，罗马不是一天建成的”。说明 在 C++中操作符可以过载(overload)。编程人员可以通过过载标准操作符让它在特定类中进行特定运行。例如，可以在一个类中过载递增操作符，让它将变量递增 10 而不是递增 1。操作符过载是个高级 C++技术，本书不准备详细介绍。你也许会发现，有些操作符使用了相同的符号。符号的意义随情境的不同而不同。例如，星号(*)可以作为乘号、声明指针或取消指针引用。这初看起来有点乱，事实上，C++编程老手有时也觉得有点乱。多实践，你会慢慢适应的。本书有许多例子介绍这些操作符。读者不必死记每个操作符的作用，而可以在学习中通过程序和码段去理解其作用。

5 数组

任何 C++固有数据类型都可以放进数组中。数组(array)就是数值的集合。例如，假设要保存一个整型数组，放五个整型值。可以声明数组如下：int myArray[5];由于每个 int 要 4 个字节存储，所以整个数组占用 20 字节的内存空间。

声明数组后，就可以用如下脚标操作符([])填入数值：

```
myArray[0] = -200;
myArray[1] = -100;
myArray[2] = 0;
myArray[3] = 100;
myArray[4] = 200;
```

由上可见，C++中的数组是以 0 为基数的。后面程序中可以用脚标操作符访问数组的各个元素：

```
int result=myarray[3]+myArray[4]; // result will be 300
```

还有一次声明和填入整个数组内容的简捷方法如下：

int myArray[5] = {-200, -100,0,100,200};

进一步说，如果知道数组的元素个数，并在声明数组时填充数组，则声明数组时连数组长度都可以省略。例如：int myArray[] = {-200, -100,0,100,200 };这是可行的，因为编译器从赋予的数值表可以判断出数组中元素的个数和分配给数组的内存空间。

数组可以是多维的。为了生成两维整型数组，可用下列代码：

int mdArray[3][5];

这样就分配 15 个 int 空间(共 60 字节)。数组的元素可以和一维数组一样访问，只是要提供两个脚标操作符：int x = mdArray[1][1]+mdArray[2][1];

C++一个强大的特性是能直接访问内存。由于这个特性，C++无法阻止你写入特定内存地址，即使这个地址是程序不让访问的。下列代码是合法的，但会导致程序或 Windows 崩溃：int array[5];array[5]=10;这是常见的错误，因为数组是以 0 为基数的，最大脚标应是 4 而不是 5。如果重载数组末尾，则无法知道哪个内存被改写了，使结果难以预料，甚至会导致程序或 Windows 崩溃。这类问题很难诊断，因为受影响的内存通常要在很久以后才访问，这时才发生崩溃（让你莫名其中之妙）。所以写入数组时一定要小心。

数组规则：

- 数组是以 0 为基数。数组中的第一个元素为 0，第二个元素为 1，第三个元素为 2.
- 数组长度应为编译常量。编译器在编译时必须知道为数组分配多少内存空间。不能用变量指定数组长度。所以下列代码不合法，会导致编译错误： int x = 10;int myArray[x]; // compiler error here·小心不要重载数组末尾。
- 大数组从堆叠（heap）而不是堆栈(stack)中分配。
- 从堆叠分配的数组可以用变量指定数组长度。例如：int x = 10;int* myArray = new int[x];

6 字符数组

奇怪的是，C++不支持字串变量（放置文本的变量），C++程序中的字串是用 char 数据类型的数组表示的。例如，可以将变量赋予 char 数组如下：

char text[] = “This is a string.”;

这就在内存中分配 18 字节的内存空间用于存放字串。根据你的领悟能力，也许你会发现该字串中只有 17 个字符。分配 18 个字节的原因是字串要以终止 null 结尾，C++在分配内存空间时把终止 null 算作一个字符。

新术语 终止 null 是个特殊字符，用|0 表示，等于数值 0。程序遇到字符数组中的 0 时，表示已经到字串末尾。为了说明这点，输入并运行下列控制台应用程序。

清单 1.6Nulltest.cpp

```
1: #include <iostream>
2: using namespace std;
3:
4:
5: int main(int argc,char **argv)
6: {
7:    char str[]=”This is a string.”;
8.    cout << str << end1;
9.    str[7]= ‘\0’;
10.   cout << str << end1
```

11. System (“pause”);
12: }
分析 最初，字符数组包含字符串 This is a string 和一个终止 null，这个字串通过 cout 送到屏幕上。下一行将数组的第 7 个元素赋值为|0，即终止 null。字串再次发送到屏幕上，但这时只显示 This is。原因是计算机认为数组中字串在第 7 个元素上终止，余下字串仍然在内存空间中，但不显示，因为遇到了终止 null。演示将数组的第 7 个元素赋值为|0 的语句前后的字符数组。
之前
This is a string.\0
之后
This is\0a string.\0
清单 1.6 中也可以赋值 0 而不是’|0’，结果相同，因为数字 0 和 char 数据类型’|0’是等值的。例如，下列语句是等价的：
str[7] = ‘|0’;
str[7] = 0;
说明 C++程序中单引号与双引号是有差别的。向数组元素赋值终止 null 和其它字符值时，必须用单引号。单引号的作用是将引号内的字符变成整型值（该字符的 ASCII 值），然后将这个值存放在内存地址中。将字串赋予字符数组时，必须用双引号。如果用错引号，则编译器会发生编译错误。

7 字串操作函数

如果你用过具有 string 数据类型的编程语言，你可能很不习惯，别人也有同感，所以标准 C 语言库中提供了几个字串操作函数。表 1.3 列出了最常用的字串操作函数及其用法说明。关于每个函数的详细说明和实例，见 C++ Builder 联机帮助。

表 1.3 字串操作函数
函数 说明
strcat() 将字串接合到目标字串的末尾
strcmp() 比较两个字串是否相等
strcmpi() 比较两个字串是否相等，不考虑大小写
strcpy() 将字串内容复制到目标字串中
strstr() 扫描字串中第一个出现的字串
strlen() 返回字串长度
strupr() 将字串中的所有字符变成大写
sprintf() 根据几个参数建立字串
说明 这里介绍的字串操作是 C 语言中的字串处理方法。大多数 C++编译器提供了 cstring 类，可以简化字串的处理(C++ Builder 的 Visual 构件库中有个 AnsiString 类，可以处理字串操作。C++ Builder 联机帮助中详细介绍了 AnsiString 类)。尽管 C 语言中的字串处理方法比较麻烦，但并不过时，C++编程人员经常在使用 cstring 类和 AnsiString 类等字串类的同时使用 C 语言中的字串处理方法。这里不想对表中的每个函数进行举例说明，只想举两个最常用的函数。strcpy()函数将一个字串复制到另一字串中，源字串可以是变量或直接字串。例如下列代码：
//set up a string to hold 29 characters

```
char buff[30];
//copy a string literal to the buffer
strcpy (buff,”This is a test.”);//display it
cout << buff << end;
//initialize a second string buffer
char buff2[]=”A second string.”;
//copy the contents of this string to the first buffer
strcpy (buff,buff2);
cout << buff << end1;
```

字符数组中比数字数组中更容易重载数字末尾。例如下列代码：

```
char buff[10]= “A string”;// later....
strcpy(buff,”This is a test.”); //oops!
```

这里建立了放 10 个字符的字符数组，最初指定需要 9 个字节的字符串（记住终止 null）。后来可能忘记了数组长度，将需要 16 个字节的字串复制到了缓冲区，对数组重载了六个字节。这个小小错误就擦去了某个内存位置上的六个字节。所以将数据复制到字符数组中时要特别小心。另一个常用的字串函数是 sprintf()。这个函数可以混合文本和数字建立格式化字串。下面例子将两个数相加，然后用 sprintf()建立字串以报告结果：

```
char buff[20];
int x = 10 * 20;
sprintf(buff,”The result is: %d”,x);
cout << buff;
```

执行这个码段时，程序显示下列结果：The result is:200

本例中%d 告诉 sprintf()函数此处有个整型值，格式字串末尾插入变量 x，告诉 sprintf()在字串的这个位置放上变量 x 的值。sprintf()是个特别的函数，可以取多个变元。你必须提供目标缓冲区和格式字串，但格式字串后面的变元数是个变量。下面的 sprintf()例子用了另外三个变元：

```
int x = 20;
int y = 5;
sprintf(buff, “%d + %d”, x, y, x + y);
cout << buff;
```

执行这个码段时，屏幕上显示的结果如下：20 + 5 = 25

说明 C++字串中的单斜杠表示特殊字符。例如，’\n’表示新行符，’\t’表示跳表符。为了在字串中放上实际的斜杠，要用双斜杠如下：

```
strcpy(fileName, “c:\\windows\\system\\win.ini”);
```

字串数组不仅可以有字符数组，还可以有字符数组的数组（即字串数组）。这听起来有点复杂，其实前面的 Argstest 程序中已经用过。这类数组可以分配如下：

```
    char strings[][20] = {
”This is string 1”,
”This is string 2”,
”This is string 3”,
”This is string 4”};
```

这个代码生成四个字串的数组，每个字串最多放 19 个字符。尽管可以使用这种字串数组，但 C++ Builder 中还有更简单的字串数组处理办法（将在后面介绍 C++ Builder 时介绍）。

说明 如果经常用到字串数组，应当看看标准模板库(STL).STL 提供了比用 C 语言式字符

数组更方便地存放和操作字串数组的方法.STL 中还有个 string 类。

8 过程控制

8.1 if 语句

if 语句用于测试条件并在条件为真时执行一桌或几条语句。说明:if 表达式后面不能带分号，否则它本身表示代码中的空语句，使编译器将空语句解释为在条件为真时执行的语句。

```
if (x == 10); // Warning! Extra semicolon!
DoSomething(x);
```

这里 DoSomething()函数总会执行，因为编译器不把它看成在条件为真时执行的第一条语句。由于这个代码完全合法（但无用），所以编译器无法发出警告。

假设要在条件为真时执行多行语句，则要将这些语句放在大括号内：

```
if (x > 10) {
cout << “The number is greater than 10” << end1;
DoSomethingWithNumber(x);
}
```

条件表达式求值为 false 时，与 if 语句相关联的码段忽略，程序继续执行该码段之后的第一条语句。

C＋＋中包含许多快捷方法，其中一个是用变量名测试 true，例如：

```
if (fileGood) ReadData();
```

这个方法是下列语句的速写方法：

```
if (fileGood == true) ReadData();
```

本例用了 bool 变量，也可以用其它数据类型。只要变量包含非零数值，表达式即求值为 true，对变量名加上逻辑非(!)操作符可以测试 false 值：

```
bool fileGood = OpenSomeFile();
if (!fileGood) ReportError();
```

学会 C＋＋快捷方法有助于写出更精彩的代码。 有时要在条件表达式求值为 true 时进行某个动作，在条件表达式求值为 false 时进行另一动作，这时可以用 else 语句如下：

```
if (x == 20) {DoSomething(x);}
else {DoADifferentThing(x);}
```

新术语 else 语句和 if 语句一起使用，表示 if 语句失败时（即在条件表达式求值为 false 时）执行的码段。

if 语句形式之二：

```
if (cond_expr_1) {
   true_statements_1;
}
else if (cond_expr_2)
{
      true_statements_2;
```

} else false_statements;
如果条件表达式 cond_expr 为 1 真（非零），则执行 true_statements1 码段;如果条件表达式 cond_expr 为 1 为假而如果条件表达式 cond_expr 为 2 真(非零),则执行 true_statements 2 码段;如果两个表达式均为假，执行 false_statements 码段。

8.2 switch 语句

switch 语句是高级 if 语句，可以根据表达式的结果执行几个码段之一。表达式可以是变量、函数调用结果或其它有效 C＋＋表达式。下面举一个 switch 语句例子：

```
    switch (amountOverSpeedLimit) {
case 0 : {fine =0; break;}
case 10 : {fine = 20; break; }
case 15 : {fine =20; break; }
case 20 :
case 25 :
case 30 : {  fine=amountOverSpeedLimit * 10; break;  }
default : {fine =GoToCourt(); jailTime=GetSentence(); }
    }
```

switch 语句分为几个部分。首先有一个表达式，本例中是 amountOverSpeedLimit 变量（够长的变量名!），然后用 case 语句测试表达式，如果 amountOverSpeedLimit 等于 0(case 0:)，则向变量 fine 赋值 0,如果 amountOverSpeedLimit 等于 10，则向变量 fine 赋值 20，等等。在前三个 case 中都有 break 语句。break 语句用于转出 switch 块，即找到了符合表达式的情况，switch 语句的余下部分可以忽略了。最后有个 default 语句，如果没有符合表达式的情况，则程序执行 default 语句。

8.3 for 语句

for 循环是最常用的循环，取三个参数：起始数，测试条件和增量表达式。

for 循环语句：

```
for(initial; cond_expr;adjust)
{
    statements;
}
```

for 循环重复执行 statements 码段，直到条件表达式 cond_expr 不为真。循环状态由 initial 语句初始化，执行 statements 码段后，这个状态用 adjust 语句修改。下面举一个 for 循环的典型例子进行说明：

```
for (int i=0;i<10;i++){
    cout << “This is iteration” << i << end1;
}
```

8.4 while 循环

while 循环与 for 循环的差别在于它只有一个在每次循环开始时检查的测试条件。只要测试条件为 true，循环就继续运行。

```
int x;
while (x < 1000) {
    x = DoSomeCalculation();
}
```

本例中我调用一个函数，假定它最终会返回大于或等于 1000 的值。只要这个函数的返回值小于 1000，while 循环就继续运行。变量 x 包含大于或等于 1000 的值时，测试条件变成 false，程序转入 while 循环闭括号后面的第一条语句。while 循环通常用 bool 变量进行测试。测试变量状态可以在循环体中进行设置：

```
bool done = false;
while (!done) {
  //some code here
  done = SomeFunctionReturningABool();
  //more code
}
```

8.5 do while 语句

while 循环测试发生在循环体开头，而 do while 循环测试则发生在循环结束时：

```
bool done = false;
do {
    // some code
    done =SomeFunctionReturningABool();
    // more code
} while (! done);
```

使用 do while 循环还是 while 循环取决于循环本身的作用。语法中 do while 循环语句：

```
do {
  tatements;
} while (cond_expr);
```

只要条件表达式 cond_expr 为真(非零)，do 循环重复 statements 码段。循环状态必须在 do 语句之前初始化，并在码段中显式修改。条件表达式 cond_expr 为假时，循环终止。

8.6 continue 语句

continue 语句强制程序转入循环底部，跳过 continue 语句之后的任何语句。例如，某个测试为真时，循环的某个部分可能不需要执行。这时可以用 continue 语句跳过 continue 语句之后的任何语句：

```
    bool done = false;
while (!done) {
// some codebool error =SomeFunction();
if (error) continue;
// jumps to the top of the loop
// other code that will execute only if no error occurred
}
```

8.7 break 语句

break 语句用于在循环正常测试条件符合之前终止循环执行。例如，可以在 ints 数组中搜索某个元素，找到数字后可以终止循环执行，取得该数字所在的索引位置：

```
    int index=1
int searchNumber=50;
for (int i=0;i<numElements;i＋＋) {
  if (myArray［i］==searchNumber) {
       index=i;break;
   }
}
if(index !=1)   cout << “Number found at index “ << index << end1;
else   cout <<”Number not found in array.” << end1;
```

continue 和 break 语句在许多情况下有用。和其它要介绍的知识一样，continue 和 break 语句也要在实践中不断熟悉。

8.8 goto 语句

可以将程序转入前面用标号和冒号声明的标号处。

下列代码演示了这个语句：

```
bool done = false;
startPoint:
// do some stuff
if (!done) goto(startPoint);// loop over, moving on...
```

这里不需要大括号，因为 goto 语句与标号之间的所有代码均会执行。goto 语句被认为是 C＋＋程序中的不良语句。用 goto 语句能做的任何工作都可以用 While 和 dowhile 循环进行。一个好的 C＋＋编程人员很少在程序中使用 goto 语句。如果你从别的语言转入 C＋＋，你会发现 C＋＋的基本结构使 goto 语句显得多余。

9 C++中的函数

函数是与主程序分开的码段。这些码段在程序中需要进行特定动作时调用（执行）。例如，函数可能取两个值并对其进行复杂的数学(study888.com)运算。然后返回结果，函数可能取一个字串进行分析，然后返回分析字串的一部分。新术语 函数（function）是与

主程序分开的码段，进行预定的一个服务。函数是各种编程语言的重要部分，C++也不例外。最简单的函数不带参数，返回 void（表示不返回任何东西），其它函数可能带一个或几个参数并可能返回一个值。函数名规则与变量名相同。

新术语 参数(parameter)是传递给函数的值，用于改变操作或指示操作程度。

返回类型 函数名 参数表

```
int     SomeFunction(int x, int y){
   函数体→int z = (x * y);
      return z; ↑返回语句
}
```

函数的构成部分使用函数前，要先进行声明。函数声明或原型(prototype)告诉编译器函数所取的参数个数、每个参数的数据类型和函数返回值的数据类型。清单 1.4 列示了这个概念。新术语 原型(prototype)是函数外观的声明或其定义的说明。

清单 1.4Muttiply.cpp

```
1.#include <iostream>
2.using namespace std;
3.int multiply(int,int)
4.void showResult(int);
      5.int main(int argc,char **argv);
6.{
7.    int x,y,result;
      8.    cout << end1 <<  “Enter the first value:”;
9.    cin >> x;
10.     cout <<  “Enter the second value:  “;
11.     cin >> y;
12.    result=multiply(x,y);
13.    showResult(result);
14.    cout << end1 << end1 <<  “Press any key to continue...”;
15. }

      16.int multiply(int x,int y)
17.{
18.    return x * y;
19. }
20.void showResult(int res)
21.{
22.    cout <<  “The result is:  “  << res <<end1;
23.}
```

这个程序的 9 到 11 行用标准输入流 cin 向用户取两个数字，第 15 行调用 multiply()函数将两个数相乘，第 13 行调用 showResult()函数显示相乘的结果。注意主程序前面第 3 和第 4 行 multiply()和 showResult()函数的原型声明。原型中只列出了返回类型、函数名和函数参数的数据类型。这是函数声明的最基本要求。函数原型中还可以包含用于建档函数功能的变量名。例如，multiply()函数的函数声明可以写成如下：int multiply(int firstNumber,int secondNumber);这里函数 multiply()的作用很明显，但代码既可通过说明也可通过代码本身建档。注意清单 1.4 中函数 multiply()的定义(17 到 19 行)在主函数定义码段(5 到 15 行)之

外。函数定义中包含实际的函数体。这里的函数体是最基本的，因为函数只是将函数的两个参数相乘并返回结果。清单 1.4 中函数 multiply()可以用多种方法调用，可以传递变量、直接数或其它函数调用的结果：

```
result = multiply(2,5);//passing literal values
result = multiply(x,y); //passing variables
showResult(multiply(x,y));
//return value used as a
//parameter for another function
multiply(x,y);//return value ignored
```

注意 最后一例中没有使用返回值。本例中调用函数 multiply()而不用返回值没什么道理，但 C++编程中经常忽略返回值。有许多函数是先进行特定动作再返回一个数值，表示函数调用的状态。有时返回值与程序无关，可以忽略不计。如果将返回值忽略，则只是放弃这个值，而不会有别的危害。例如，前面的样本程序中忽略了 getch()函数的返回值（返回所按键的 ASCII 值）。函数可以调用其它函数，甚至可以调用自己，这种调用称为递归(recursion)。这在 C++编程中是个较复杂的问题，这里先不介绍。新术语 递归(recursion)就是函数调用自己的过程。本节介绍的函数指的是 C 或 C++程序中的独立函数（独立函数不是类的成员）。C++中的独立函数可以和 C 语言中一样使用，但 C++将函数进一步深化，将在稍后介绍 C++时介绍。

10 函数规则

- 函数可以取任意多个参数或不取参数。
- 函数可以返回一个值，但函数不强求返回一个值。
- 如果函数返回 void 类型，则不能返回数值。如果要让返回 void 类型的函数返回数值，则会发生编译错误。返回 void 类型的函数不需包含 return 语句，但也可以包含这个语句。如果没有 return 语句，则函数到达末尾的结束大括号时自动返回。
- 如果函数原型表示函数返回数值，则函数体中应包含返回数值的 return 语句，如果函数不返回数值，则会发生编译错误。
- 函数可以取任意多个参数，但只能返回一个数值。
- 变量可以按数值、指针或引用传递给函数（将在稍后介绍）。

  语法：函数语句的声明（原型）格式如下：

  ret_type function_name(argtype_1 arg_1,argtype_2 arg_2,...,argtype_n arg_n);

函数声明表示代码中要包括的函数，应当显示函数的返回数据类型(ret_type)和函数名(function_name)，表示函数所要数据变元的顺序（arg_1,arg_2,...,arg_n）和类型(argtype_1,argtype_2,...argtype_n)。

函数语句的定义格式如下：

```
ret_type function_name(argtype_1 arg_1,argtype_2 arg_2,...,argtype_narg_n);
{
  statements;
  return ret_type;
}
```

函数定义表示构成函数的代码块(statements),应当显示函数的返回数据类型(ret type)和函数名(function_name)，包括函数所要数据变元（arg_1,arg_2,...,arg_n）和类型(argtype_1,argtype_2,...argtype_n)。

第3章 STL 简介

1 泛型程序设计

我们可以简单的理解为：使用模板的程序设计就是泛型程序设计。就像我们我们可以简单的理解面向对象程序设计就是使用虚函数的程序设计一样。

1.1 STL 是什么

作为一个 C++程序设计者，STL 是一种不可忽视的技术。Sandard Template Library (STL)：

标准模板库,更准确的说是 C++ 程序设计语言标准模板库。学习过 MFC 的人知道，MFC 是微软公司创建的 C++ 类库。而与之类似的是 STL 是模板库，只不过 STL 是 ANSI/ISO 标准的一部分，而 MFC 只不过是微软的一个产品而已。也就是说 STL 是所有 C++编译器和所有操作系统平台都支持的一种库，说它是一种库是因为，虽然 STL 是一种标准，也就是说对所有的编译器来说，提供给 C++程序设计者的接口都是一样的。也就是说同一段 STL 代码在不同编译器和操作系统平台上运行的结果都是相同的，但是底层实现可以是不同的。 令人兴奋的是，STL 的使用者并不需要了解它的底层实现。 试想一下，如果我们有一把能打开所有锁的钥匙，那将是多么令人疯狂啊。

STL 的目的是标准化组件，这样你就不用重新开发它们了。你可以仅仅使用这些现成的组件。STL 现在是 C++的一部分，因此不用额外安装什么。它被内建在你的编译器之内。

1.2 为什么我们需要学习 STL

- STL 是 C++的 ANSI/ISO 标准的一部分,可以用于所有 C++语言编译器和所有平台(Windows/Unix/Linux..)。STL 的同一版本在任意硬件配置下都是可用的；
- STL 提供了大量的可复用软件组织。例如，程序员再也不用自己设计排序，搜索算法了，这些都已经是 STL 的一部分了。嘎嘎，有意思吧；
- 使用 STL 的应用程序保证了得到的实现在处理速度和内存利用方面都是高效的，因为 STL 设计者们已经为我们考虑好了；
- 使用 STL 编写的代码更容易修改和阅读，这是当然的鸟。因为代码更短了，很多基础工作代码已经被组件化了；
- 使用简单，虽然内部实现很复杂；

虽然，STL 的优点甚多，但是 STL 的语法实在令初学者人头疼，许多人望而却步。可是 STL 是每个 C++程序设计者迟早都要啃的一块骨头。

1.3 初识 STL

下面让我们来看几段代码吧：

```
//stl_cpp_1.cpp
#include <iostream>
double mean(double *array, size_t n)
{
    double m=0;
    for(size_t i=0; i<n; ++i){
        m += array[i];
    }
    return m/n;
}
int main(void)
{
    double a[] = {1, 2, 3, 4, 5};
    std::cout<<mean(a, 5)<<std::endl;       // will print 3
    return 0;
}
```

好懂吧，除了那个 std 有点让人不舒服以外？这是一段普通的没有使用 STL 的 C++代码。再看下面一段：

```
//stl_cpp_2.cpp
#include <vector>
#include <iostream>
int main(void)
{
    std::vector<double> a;
    std::vector<double>::const_iterator i;
    a.push_back(1);
    a.push_back(2);
    a.push_back(3);
    a.push_back(4);
    a.push_back(5);
    for(i=a.begin(); i!=a.end(); ++i){
        std::cout<<(*i)<<std::endl;
    }
    return 0;
}
```

如果你真的没有接触过 STL 的话，你会问，呀，vector 是啥呀？这是一段纯种的 STL 代码，看到尖括号了吧，知道那是模板了吧。看到 a.push_back(5),a.begin(),a.end()你不感觉奇怪么？可是我们并没有定义这些函数啊。

```
//stl_cpp_3.cpp
#include <vector>
#include <iostream>
int main(void)
{
    std::vector<int> q;
    q.push_back(10);
    q.push_back(11);
    q.push_back(12);

    std::vector<int> v;
    for(int i=0; i<5; ++i){
        v.push_back(i);
    }
    std::vector<int>::iterator it = v.begin() + 1;
    it = v.insert(it, 33);
    v.insert(it, q.begin(), q.end());
    it = v.begin() + 3;
    v.insert(it, 3, -1);
    it = v.begin() + 4;
    v.erase(it);
    it = v.begin() + 1;
    v.erase(it, it + 4);
    v.clear();
    return 0;
}
```

这一段你又看到了新东西了吧，iterator？？？不罗嗦了，等你看完这篇文章，回头再看就简单了。

关于模板的其他细节，读者可以参阅《C++ Templates 中文版》在这里，简单的介绍一下模板类和函数模板的概念。

模板是 C++中实现代码重用机制的一种工具，可以实现类型参数化，把类型定义为参数。函数模板和类模板允许用户构造模板函数和模板类。

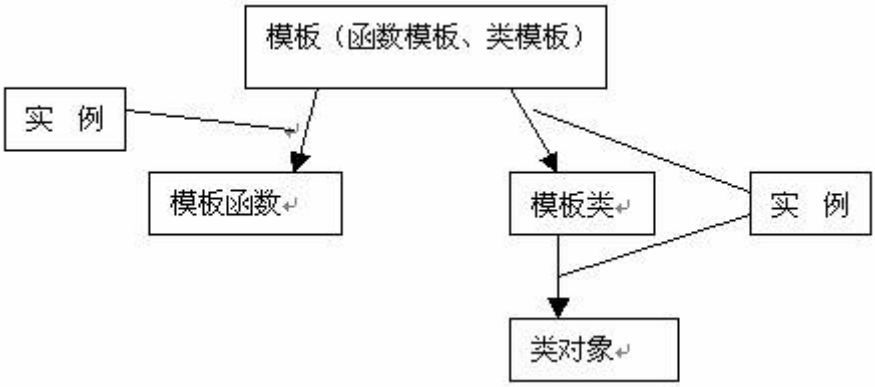


图 1

下面我们来看一段函数模板的例子：

```
//stl_cpp_4.cpp
#include<iostream.h>
#include<string.h>
//定义函数模板
template<class T>   //template 是关键字，T 表示一种待实例化的类型
//template<typename T>   也是对的
T max(T a, T b)//函数模板，函数名为 max，此函数有 2 个 T 类型的参数，返回类型为 T
{
   return (a>b)?a:b;
}
//在此例实例化的时候，T 可以是多种类型的，int,char,string…
int main(void)
{
   int x=2,y=6;
   double x1=9.123,y1=12.6543;
   cout<<"把 T 实例化为 int:"<<max(x,y)<<endl;//实例化函数模板，把 T 实例化为 int
   cout<<"把 T 实例化为 double:"<<max(x1,y1)<<endl;
   //实例化函数模板，把 T 实例化为 double
getchar(); //这一行代码用来在 dos 下查看结果，也可以用 cin.get();
}
```

下面再看看，类模板：

```
//stl_cpp_5.cpp
#include<iostream.h>
//定义名为 ex_class 的类模板
template < typename T>   class ex_class
{
```

```
    T value;
public:
    ex_class(T v) { value=v; }
    void set_value(T v) { value=v; }
    T get_value(void) {return value;}
};
//main()函数中测试 ex_class 类模板
int main(void)
{
    //测试 int 类型数据
    ex_class <int> a(5),b(10);
    cout<<"a.value:"<<a.get_value()<<endl;
    cout<<"b.value:"<<b.get_value()<<endl;
    //测试 char 类型数据
    ex_class <char> ch("A");
    cout<<"ch.value:"<<ch.get_value()<<endl;
    ch.set_value("a");
    cout<<"ch.value:"<<ch.get_value()<<endl;
    //测试 double 类型数据
    ex_class <double> x(5.5);
    cout<<"x.value:"<<x.get_value()<<endl;
    x.set_value(7.5);
    cout<<"x.value:"<<x.get_value()<<endl;
}
```

2 STL 的组成

STL 有三大核心部分：容器（Container）、算法（Algorithms）、迭代器（Iterator），容器适配器（container adaptor），函数对象(functor)，除此之外还有 STL 其他标准组件。通俗的讲：

- 容器：装东西的东西，装水的杯子，装咸水的大海，装人的教室……STL 里的容器是可容纳一些数据的模板类；
- 算法：就是往杯子里倒水，往大海里排污，从教室里撵人……STL 里的算法，就是处理容器里面数据的方法，操作；
- 迭代器：往杯子里倒水的水壶，排污的管道，撵人的那个物业管理人员……STL 里的迭代器：遍历容器中数据的对象；

对存储于容器中的数据进行处理时，迭代器能从一个成员移向另一个成员。他能按预先定义的顺序在某些容器中的成员间移动。对普通的一维数组、向量、双端队列和列表来说，迭代器是一种指针。

下面让我们来看看专家是怎么说的：

- **容器（container）**：容器是数据在内存中组织的方法，例如，数组、堆栈、队列、链表或二叉树（不过这些都不是 STL 标准容器）。STL 中的容器是一种存储 T（Template）类型值的有限集合的数据结构,容器的内部实现一般是类。这些值可以是对象本身，如果数据类型 T 代表的是 Class 的话。
- **算法（algorithm）**：算法是应用在容器上以各种方法处理其内容的行为或功能。例如，有对容器内容排序、复制、检索和合并的算法。在 STL 中，算法是由模板函数表现的。这些函数不是容器类的成员函数。相反，它们是独立的函数。令人吃惊的特点之一就是其算法如此通用。不仅可以将其用于 STL 容器，而且可以用于普通的 C＋＋数组或任何其他应用程序指定的容器。
- **迭代器(iterator)**：一旦选定一种容器类型和数据行为(算法)，那么剩下唯一要他做的就是用迭代器使其相互作用。可以把达代器看作一个指向容器中元素的普通指针。可以如递增一个指针那样递增迭代器，使其依次指向容器中每一个后继的元素。迭代器是 STL 的一个关键部分，因为它将算法和容器连在一起。

下面我将依次介绍 STL 的这三个主要组件。

2.1 容器

STL 中的容器有队列容器和关联容器，容器适配器（congtainer adapters：stack,queue，priority queue），位集（bit_set），串包(string_package)等等。

在本文中，我将介绍 list,vector，deque 等队列容器，和 set 和 multisets,map 和 multimaps 等关联容器，一共 7 种基本容器类。

队列容器（顺序容器）：队列容器按照线性排列来存储 T 类型值的集合，队列的每个成员都有自己的特有的位置。顺序容器有向量类型、双端队列类型、列表类型三种。

2.1.1 基本容器——向量

向量（vector 容器类）：＃include <vector>，vector 是一种动态数组，是基本数组的类模板。其内部定义了很多基本操作。既然这是一个类，那么它就会有自己的构造函数。vector 类中定义了 4 中种构造函数：

- 默认构造函数，构造一个初始长度为 0 的空向量，如：vector<int> v1;
- 带有单个整形参数的构造函数，此参数描述了向量的初始大小。这个构造函数还有一个可选的参数，这是一个类型为 T 的实例，描述了各个向量种各成员的初始值；如：vector<int> v2(init_size,0); 如果预先定义了：int init_size;他的成员值都被初始化为 0；
- 复制构造函数，构造一个新的向量，作为已存在的向量的完全复制，如：vector<int> v3(v2);
- 带两个常量参数的构造函数，产生初始值为一个区间的向量。区间由一个半开区间[first,last](MS word 的显示可能会有问题，first 前是一个左方括号，last 后面是一个右圆括号)来指定。如：vector<int> v4（first,last）

下面一个例子用的是第四种构造方法，其它的方法读者可以自己试试。

```
//stl_cpp_7.cpp
//程序：初始化演示
#include <cstring>
#include <vector>
#include <iostream>
using namespace std;
int ar[10] = {    12, 45, 234, 64, 12, 35, 63, 23, 12, 55    };
char* str = "Hello World";
int main(void)
{
   vector <int> vec1(ar, ar+10);                  //first=ar,last=ar+10,不包括 ar+10
   vector <char> vec2(str, str+strlen(str));    //first=str,last= str+strlen(str),不包括最后一个
   cout<<"vec1:"<<endl;
//打印 vec1 和 vec2，const_iterator 是迭代器，后面会讲到
//当然，也可以用 for (int i=0; i<vec1.size(); i++)cout << vec[i];输出
//size()是 vector 的一个成员函数
   for(vector<int>::const_iterator p=vec1.begin();p!=vec1.end(); ++p)
       cout<<*p;
   cout<<"\n"<<"vec2:"<<endl;
   for(vector<char>::const_iterator p1=vec2.begin();p1!=vec2.end(); ++p1)
        cout<<*p1;
   getchar();
   return 0;
}
```

为了帮助理解向量的概念，这里写了一个小例子，其中用到了 vector 的成员函数：begin()，end()，push_back()，assign()，front()，back()，erase()，empty()，at()，size()。

```
//stl_cpp_8.cpp
#include <iostream>
#include <vector>
using namespace std;
typedef vector<int> INTVECTOR;//自定义类型 INTVECTOR
//测试 vector 容器的功能
void main(void)
{
    //vec1 对象初始为空
    INTVECTOR vec1;
    //vec2 对象最初有 10 个值为 6 的元素
    INTVECTOR vec2(10,6);
    //vec3 对象最初有 3 个值为 6 的元素，拷贝构造
    INTVECTOR vec3(vec2.begin(),vec2.begin()+3);
    //声明一个名为 i 的双向迭代器
```

```
    INTVECTOR::iterator i;
    //从前向后显示 vec1 中的数据
    cout<<"vec1.begin()--vec1.end():"<<endl;
    for (i =vec1.begin(); i !=vec1.end(); ++i)
        cout << *i << " ";
    cout << endl;
    //从前向后显示 vec2 中的数据
    cout<<"vec2.begin()--vec2.end():"<<endl;
    for (i =vec2.begin(); i !=vec2.end(); ++i)
        cout << *i << " ";
    cout << endl;
    //从前向后显示 vec3 中的数据
    cout<<"vec3.begin()--vec3.end():"<<endl;
    for (i =vec3.begin(); i !=vec3.end(); ++i)
        cout << *i << " ";
    cout << endl;
    //测试添加和插入成员函数，vector 不支持从前插入
    vec1.push_back(2);//从后面添加一个成员
    vec1.push_back(4);
vec1.insert(vec1.begin()+1,5);//在 vec1 第一个的位置上插入成员 5
//从 vec1 第一的位置开始插入 vec3 的所有成员
vec1.insert(vec1.begin()+1,vec3.begin(),vec3.end());
cout<<"after push() and insert() now the vec1 is:" <<endl;
    for (i =vec1.begin(); i !=vec1.end(); ++i)
        cout << *i << " ";
    cout << endl;
    //测试赋值成员函数
    vec2.assign(8,1);    // 重新给 vec2 赋值，8 个成员的初始值都为 1
    cout<<"vec2.assign(8,1):" <<endl;
    for (i =vec2.begin(); i !=vec2.end(); ++i)
        cout << *i << " ";
    cout << endl;
    //测试引用类函数
    cout<<"vec1.front()="<<vec1.front()<<endl;//vec1 第零个成员
    cout<<"vec1.back()="<<vec1.back()<<endl;//vec1 的最后一个成员
    cout<<"vec1.at(4)="<<vec1.at(4)<<endl;//vec1 的第五个成员
    cout<<"vec1[4]="<<vec1[4]<<endl;
    //测试移出和删除
    vec1.pop_back();//将最后一个成员移出 vec1
    vec1.erase(vec1.begin()+1,vec1.end()-2);//删除成员
    cout<<"vec1.pop_back() and vec1.erase():" <<endl;
    for (i =vec1.begin(); i !=vec1.end(); ++i)
        cout << *i << " ";
    cout << endl;
```

```
    //显示序列的状态信息
    cout<<"vec1.size(): "<<vec1.size()<<endl;//打印成员个数
    cout<<"vec1.empty(): "<<vec1.empty()<<endl;//清空
}
```

push_back()是将数据放入 vector（向量）或 deque（双端队列）的标准函数。Insert()是一个与之类似的函数，然而它在所有容器中都可以使用，但是用法更加复杂。end()实际上是取末尾加一，以便让循环正确运行--它返回的指针指向最靠近数组界限的数据。

在 Java 里面也有向量的概念。Java 中的向量是对象的集合。其中，各元素可以不必同类型，元素可以增加和删除，不能直接加入原始数据类型。

2.1.2 双端队列（deque 容器类）：#include <deque>

deque（读音：deck，意即：double queue）容器类与 vector 类似，支持随机访问和快速插入删除，它在容器中某一位置上的操作所花费的是线性时间。与 vector 不同的是，deque 还支持从开始端插入数据：
push_front()。此外 deque 也不支持与 vector 的 capacity()、reserve()类似的操作。

```
//stl_cpp_9.cpp
#include <iostream>
#include <deque>
using namespace std;
typedef deque<int> INTDEQUE;//有些人很讨厌这种定义法，呵呵
//从前向后显示 deque 队列的全部元素
void put_deque(INTDEQUE deque, char *name)
{
    INTDEQUE::iterator pdeque;//仍然使用迭代器输出
    cout << "The contents of " << name << " : ";
    for(pdeque = deque.begin(); pdeque != deque.end(); pdeque++)
        cout << *pdeque << " ";//注意有 "*"号哦，没有"*"号的话会报错
    cout<<endl;
}
//测试 deqtor 容器的功能
void main(void)
{
    //deq1 对象初始为空
    INTDEQUE deq1;
    //deq2 对象最初有 10 个值为 6 的元素
    INTDEQUE deq2(10,6);
    //deq3 对象最初有 3 个值为 6 的元素
```

```
//声明一个名为 i 的双向迭代器变量
INTDEQUE::iterator i;
//从前向后显示 deq1 中的数据
put_deque(deq1,"deq1");
//从前向后显示 deq2 中的数据
put_deque(deq2,"deq2");
   //从 deq1 序列后面添加两个元素
   deq1.push_back(2);
   deq1.push_back(4);
   cout<<"deq1.push_back(2) and deq1.push_back(4):"<<endl;
put_deque(deq1,"deq1");
   //从 deq1 序列前面添加两个元素
   deq1.push_front(5);
   deq1.push_front(7);
   cout<<"deq1.push_front(5) and deq1.push_front(7):"<<endl;
put_deque(deq1,"deq1");
   //在 deq1 序列中间插入数据
   deq1.insert(deq1.begin()+1,3,9);
   cout<<"deq1.insert(deq1.begin()+1,3,9):"<<endl;
put_deque(deq1,"deq1");
   //测试引用类函数
   cout<<"deq1.at(4)="<<deq1.at(4)<<endl;
   cout<<"deq1[4]="<<deq1[4]<<endl;
   deq1.at(1)=10;
   deq1[2]=12;
   cout<<"deq1.at(1)=10 and deq1[2]=12 :"<<endl;
put_deque(deq1,"deq1");
   //从 deq1 序列的前后各移去一个元素
   deq1.pop_front();
   deq1.pop_back();
   cout<<"deq1.pop_front() and deq1.pop_back():"<<endl;
put_deque(deq1,"deq1");
   //清除 deq1 中的第 2 个元素
   deq1.erase(deq1.begin()+1);
   cout<<"deq1.erase(deq1.begin()+1):"<<endl;
put_deque(deq1,"deq1");
   //对 deq2 赋值并显示
   deq2.assign(8,1);
```

```
        cout<<"deq2.assign(8,1):"<<endl;
    put_deque(deq2,"deq2");
}
```

上面我们演示了 deque 如何进行插入删除等操作，像 erase(),assign()是大多数容器都有的操作。关于 deque 的其他操作请参阅附录。

2.1.3 表（List 容器类）：#include <list>

List 又叫链表，是一种双线性列表，只能顺序访问（从前向后或者从后向前），图 2 是 list 的数据组织形式。与 前面两种容器类有一个明显的区别就是：它不支持随机访问。要访问表中某个下标处的项需要从表头或表尾处（接近该下标的一端）开始循环。而且缺少下标预算符：operator[]。

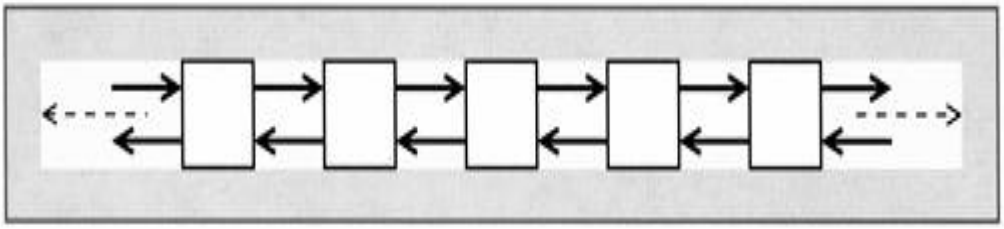


图 2

同时，list 仍然包涵了 erase(),begin(),end(),insert(),push_back(),push_front()这些基本函数，下面我们来演示一下 list 的其他函数功能。

merge()：合并两个排序列表；

splice()：拼接两个列表；

sort()：列表的排序；

```
//stl_cpp_10.cpp
#include <iostream>
#include <string>
#include <list>
using namespace std;
void PrintIt(list<int> n)
{
    for(list<int>::iterator iter=n.begin(); iter!=n.end(); ++iter)
        cout<<*iter<<" ";//用迭代器进行输出循环
    }
int main(void)
{
```

```
    list<int> listn1,listn2;      //给 listn1,listn2 初始化
    listn1.push_back(123);
    listn1.push_back(0);
    listn1.push_back(34);
    listn1.push_back(1123);       //now listn1:123,0,34,1123
    listn2.push_back(100);
    listn2.push_back(12);         //now listn2:12,100
    listn1.sort();
    listn2.sort();       //给 listn1 和 listn2 排序
    //now listn1:0,34,123,1123              listn2:12,100
    PrintIt(listn1);
    cout<<endl;
    PrintIt(listn2);
    listn1.merge(listn2);       //合并两个排序列表后,listn1:0，12，34，100，123，1123
    cout<<endl;
    PrintIt(listn1);
    cin.get();
}
```

上面并没有演示 splice()函数的用法，这是一个拗口的函数。用起来有点麻烦。图 3 所示是 splice 函数的功能。将一个列表插入到另一个列表当中。list 容器类定义了 splice()函数的 3 个版本：

```
splice(position,list_value);
splice(position,list_value,ptr);
splice(position,list_value,first,last);
```

list_value 是一个已存在的列表，它将被插入到源列表中，position 是一个迭代参数，他当前指向的是要进行拼接的列表中的特定位置。

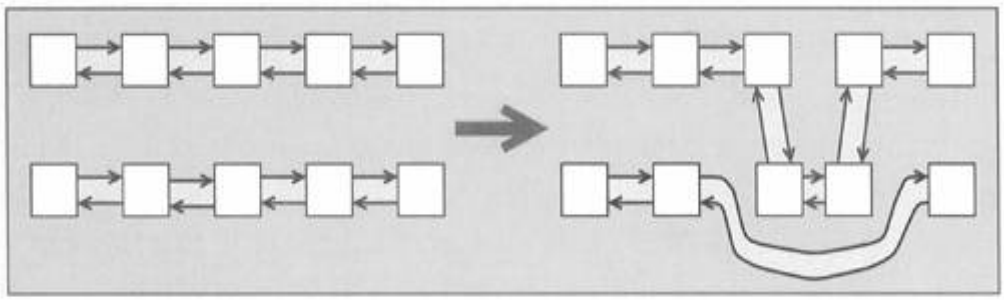


图 3

```
listn1:123,0,34,1123    listn2:12,100
```

执行 listn1.splice(find(listn1.begin(),listn1.end(),0),listn2);之后，listn1 将变为：123，12，100，34，1123。即把 listn2 插入到 listn1 的 0 这个元素之前。其中，find()函数找到 0 这个元素在 listn1 中的位置。值得注意的是，在执行 splice 之后，list_value 将不复存在了。这个例子中是 listn2 将不再存在。

第二个版本当中的 ptr 是一个迭代器参数，执行的结果是把 ptr 所指向的值直接插入到

position 当前指向的位置之前.这将只向源列表中插入一个元素。

第三个版本的 first 和 last 也是迭代器参数，并不等于 list_value.begin(),list_value.end()。First 指的是要插入的列的第一个元素，last 指的是要插入的列的最后一个元素。

如果 listn1:123,0,34,1123 listn2:12,43，87，100 执行完以下函数之后

```
listn1.splice(find(listn1.begin(),listn1.end(),0),++listn2.begin(),--listn2.end());
```

listn1:123,43,87,0,34,1123 listn2:12,100

以上，我们学习了 vector,deque,list 三种基本顺序容器，其他的顺序容器还有：slist,bit_vector 等等。

2.1.4 集和多集（set 和 multiset 容器类）：#include <set>

一个集合（set）是一个容器，它其中所包含的元素的值是唯一的。这在收集一个数据的具体值的时候是有用的。集合中的元素按一定的顺序排列，并被作为集合中的实例。如果你需要一个键/值对（pair）来存储数据，map（也是一个关联容器，后面将马上要讲到）是一个更好的选择。一个集合通过一个链表来组织，在插入操作和删除操作上比向量（vector）快，但查找或添加末尾的元素时会有些慢。

在集中，所有的成员都是排列好的。如果先后往一个集中插入：12，2，3，123，5，65　　则输出该集时为：2，3，5，12，65，123

集和多集的区别是：set 支持唯一键值，set 中的值都是特定的，而且只出现一次；而 multiset 中可以出现副本键，同一值可以出现多次。

Set 和 multiset 的模板参数：

```
template<class key, class compare, class Allocator=allocator>
```

第一个参数 key 是所存储的键的类型，第二个参数是为排序值而定义的比较函数的类型，第三个参数是被实现的存储分配符的类型。在有些编译器的具体实现中，第三个参数可以省略。第二个参数使用了合适形式的迭代器为键定义了特定的关系操作符，并用来在容器中遍历值时建立顺序。集的迭代器是双向，同时也是常量的，所以迭代器在使用的时候不能修改元素的值。

Set 定义了三个构造函数：

默认构造函数：

```
explicit set(const Compare&=compare());
```

如：`set<int,less<int> > set1;`

less<int>是一个标准类，用于形成降序排列函数对象。升序排列是用 greater<int>。通过指定某一预先定义的区间来初始化 set 对象的构造函数：

template<class InputIterator> set(InputIterator, InputIterator,\ const Compare&=compare());

如：set<int ,less<int> >set2(vector1.begin(),vector1.end());

复制构造函数：

set（const set<Key,Compare&>）;

如：set<int ,less<int> >set3(set2);

下面我们来看一个简单的集和多集的插入例程：

```
//stl_cpp_11.cpp
#include <iostream>
#include <set>
using namespace std;
int main(void)
{
      set<int> set1;
     for(int i=0; i<10; ++i)
       set1.insert(i);
      for(set<int>::iterator p=set1.begin();p!=set1.end();++p)
         cout<<*p<<"";
       if(set1.insert(3).second)//把 3 插入到 set1 中
//插入成功则 set1.insert(3).second 返回 1，否则返回 0
//此例中，集中已经有 3 这个元素了，所以插入将失败
           cout<<"set insert success";
        else
           cout<<"set insert failed";
      int a[] = {4, 1, 1, 1, 1, 1, 0, 5, 1, 0};
      multiset<int> A;
A.insert(set1.begin(),set1.end());
      A.insert(a,a+10);
      cout<<endl;
      for(multiset<int>::iterator p=A.begin();p!=A.end();++p)
      cout<<*p<<" ";
     cin.get();
      return 0;
}
```

在集之间可以进行并集（set_union()）、交集(set_intersection())、差集(set_diffrence())d 等操作，功能强大。

2.1.5 映射和多重映射（map 和 multimap）：#include <map>

映射和多重映射基于某一类型 Key 的键集的存在，提供对 T 类型的数据进行快速和高效的检索。对 map 而言，键只是指存储在容器中的某一成员。Map 不支持副本键，multimap 支持副本键。Map 和 multimap 对象包涵了键和各个键有关的值，键和值的数据类型是不相同的，这与 set 不同。set 中的 key 和 value 是 Key 类型的，而 map 中的 key 和 value 是一个 pair 结构中的两个分量。Map 支持下表运算符 operator[],用访问普通数组的方式访问 map，不过下标为 map 的键。在 multimap 中一个键可以对应多个不同的值。

下面的例程说明了 map 中键与值的关系。

```
//stl_cpp_12.cpp
#include <iostream>
#include <map>
using namespace std;
int main(void)
{
    map<char,int,less<char> > map1;
    map<char,int,less<char> >::iterator mapIter;
     //char  是键的类型，int 是值的类型
     //下面是初始化，与数组类似
//也可以用 map1.insert(map<char,int,less<char> >::value_type("c",3));
map1["c"]=3;
      map1["d"]=4;
      map1["a"]=1;
      map1["b"]=2;
      for(mapIter=map1.begin();mapIter!=map1.end();++mapIter)
         cout<<" "<<(*mapIter).first<<": "<<(*mapIter).second;
    //first 对应定义中的 char 键，second 对应定义中的 int 值
    //检索对应于 d 键的值是这样做的：
    map<char,int,less<char> >::const_iterator ptr;
    ptr=map1.find("d");
    cout<<"\n"<<" "<<(*ptr).first<<"  键对应于值："<<(*ptr).second;
    cin.get();
        return 0;
}
```

从以上例程中，我们可以看到 map 对象的行为和一般数组的行为类似。Map 允许两个或多个值使用比较操作符。下面我们再看看 multimap:

```
//stl_cpp_13.cpp
#include <iostream>
#include <map>
#include <string>
using namespace std;
int main(void)
{
     multimap<string,string,less<string> >mulmap;
     multimap<string,string,less<string> >::iterator p;
     //初始化多重映射 mulmap:
     typedef multimap<string,string,less<string> >::value_type vt;
     typedef string s;
```

```
    mulmap.insert(vt(s("Tom "),s("is a student")));
    mulmap.insert(vt(s("Tom "),s("is a boy")));
    mulmap.insert(vt(s("Tom "),s("is a bad boy of blue!")));
    mulmap.insert(vt(s("Jerry "),s("is a student")));
    mulmap.insert(vt(s("Jerry "),s("is a beatutiful girl")));
    mulmap.insert(vt(s("DJ "),s("is a student")));
    //输出初始化以后的多重映射 mulmap:
    for(p=mulmap.begin();p!=mulmap.end();++p)
        cout<<(*p).first<<(*p).second<<endl;
    //检索并输出 Jerry 键所对应的所有的值
    cout<<"find Jerry :"<<endl;
    p=mulmap.find(s("Jerry "));
    while((*p).first=="Jerry ")
    {
        cout<<(*p).first<<(*p).second<<endl;
        ++p;
    }
    cin.get();
    return 0;
}
```

在 map 中是不允许一个键对应多个值的，在 multimap 中，不支持 operator[],也就是说不支持 map 中允许的下标操作。

2.2 算法（algorithm）：#inlcude <algorithm>

STL 中算法的大部分都不作为某些特定容器类的成员函数，他们是泛型的，每个算法都有处理大量不同容器类中数据的使用。值得注意的是，STL 中的算法大多有多种版本，用户可以依照具体的情况选择合适版本。中在 STL 的泛型算法中有 4 类基本的算法：

- 变序型队列算法，可以改变容器内的数据；
- 非变序型队列算法，处理容器内的数据而不改变他们；
- 排序值算法，包涵对容器中的值进行排序和合并的算法，还有二叉搜索算法 $$通用数值算法；

注：STL 的算法并不只是针对 STL 容器，对一般容器也是适用的。

2.2.1 变序型队列算法（mutating algorithms）

又叫可修改的序列算法。这类算法有复制（copy）算法、交换（swap）算法、替代（replace）算法、删除（remove）算法，移动（transfer）算法、翻转（reverse）算法等

等。这些算法可以改变容器中的数据（数据值和值在容器中的位置）。下面介绍 2 个比较常用的算法 reverse()和 copy()。

```
//stl_cpp_14.cpp
#include <iostream>
#include <algorithm>
#include <iterator>//下面用到了输出迭代器 ostream_iterator
using namespace std;
int main(void)
{
    int arr[6]={1,12,3,2,1215,90};
        int arr1[7];
    int arr2[6]={2,5,6,9,0,-56};
    copy(arr,(arr+6),arr1);//将数组 aar 复制到 arr1
    cout<<"arr[6] copy to arr1[7],now arr1: "<<endl;
        for(int i=0;i<7;i++)
                cout<<" "<<arr1[i];
    reverse(arr,arr+6);//将排好序的 arr 翻转
    cout<<"\n"<<"arr reversed ,now arr:"<<endl;
        copy(arr,arr+6,ostream_iterator<int>(cout, " "));//复制到输出迭代器
swap_ranges(arr,arr+6,arr2);//交换 arr 和 arr2 序列
        cout<<"\n"<<"arr swaped to arr2,now arr:"<<endl;
        copy(arr,arr+6,ostream_iterator<int>(cout, " "));
        cout<<"\n"<<"arr2:"<<endl;
        copy(arr2,arr2+6,ostream_iterator<int>(cout, " "));
        cin.get();
    return 0;
}
```

revese()的功能是将一个容器内的数据顺序翻转过来，它的原型是：

```
template<class Bidirectional >
void reverse(Bidirectional first, Bidirectional last);
```

将 first 和 last 之间的元素翻转过来，上例中你也可以只将 arr 中的一部分进行翻转：

reverse(arr+3,arr+6);这也是有效的。First 和 last 需要指定一个操作区间。

Copy()是要将一个容器内的数据复制到另一个容器内，它的原型是：

```
Template<class InputIterator , class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
OutputIterator result);
```

它把[first,last－1]内的队列成员复制到区间[result,result+(last-first)-1]中。泛型交换算法：

Swap()操作的是单值交换，它的原型是：

```
template<class T>
void swap(T& a,T& b);
```

swap_ranges()操作的是两个相等大小区间中的值，它的原型是：

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator2swap_ranges(ForwardIterator1
first1,ForwardIterator1 last1, ForwardIterator1 first2);
```

交换区间[first1,last1-1]和[first2, first2+(last1-first1)-1]之间的值，并假设这两个区间是不重叠的。

2.2.2 非变序型队列算法（Non-mutating algorithm）：

又叫不可修改的序列算法。这一类算法操作不影响其操作的容器的内容，包括搜索队列成员算法，等价性检查算法，计算队列成员个数的算法。我将用下面的例子介绍其中的find(),search(),count()：

```
//stl_cpp_15.cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main(void)
{
    int a[10]={12,31,5,2,23,121,0,89,34,66};
    vector<int> v1(a,a+10);
    vector<int>::iterator result1,result2;//result1 和 result2 是随机访问迭代器
    result1=find(v1.begin(),v1.end(),2);
    //在 v1 中找到 2，result1 指向 v1 中的 2
    result2=find(v1.begin(),v1.end(),8);
    //在 v1 中没有找到 8，result2 指向的是 v1.end()
    cout<<result1-v1.begin()<<endl; //3－0＝3 或 4－1＝3，屏幕结果是 3
    cout<<result2-v1.end()<<endl;
    int b[9]={5,2,23,54,5,5,5,2,2};
    vector<int> v2(a+2,a+8);
    vector<int> v3(b,b+4);
    result1=search(v1.begin(),v1.end(),v2.begin(),v2.end());
    cout<<*result1<<endl;
    //在 v1 中找到了序列 v2，result1 指向 v2 在 v1 中开始的位置
     result1=search(v1.begin(),v1.end(),v3.begin(),v3.end());
     cout<<*(result1-1)<<endl;
```

```
    //在 v1 中没有找到序列 v3，result 指向 v1.end(),屏幕打印出 v1 的最后一个元素 66
      vector<int> v4(b,b+9);
      int i=count(v4.begin(),v4.end(),5);
      int j=count(v4.begin(),v4.end(),2);
      cout<<"there are "<<i<<" members in v4 equel to 5"<<endl;
      cout<<"there are "<<j<<" members in v4 equel to 2"<<endl;
      //计算 v4 中有多少个成员等于 5,2
    cin.get();
    return 0;
}
```

find()的原型是：

```
template<class InputIterator, class EqualityComparable>
InputIterator find(InputIterator first, InputIterator last,
                   const EqualityComparable& value);
```

其功能是在序列[first,last-1]中查找 value 值，如果找到，就返回一个指向 value 在序列中第一次出现的迭代，如果没有找到，就返回一个指向 last 的迭代（last 并不属于序列）。

search()的原型是：

```
template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2);
```

其功能是在源序列[first1,last1-1]查找目标序列[first2，last2-1]如果查找成功，就返回一个指向源序列中目标序列出现的首位置的迭代。查找失败则返回一个指向 last 的迭代。 Count()的原型是：

```
template <class InputIterator, class EqualityComparable>
iterator_traits<InputIterator>::difference_type count(InputIterator first,
InputIterator last, const EqualityComparable& value);
```

其功能是在序列[first,last-1]中查找出等于 value 的成员，返回等于 value 得成员的个数。

2.2.3 排序算法（sort algorithm）：

这一类算法很多，功能强大同时也相对复杂一些。这些算法依赖的是关系运算。在这里我只介绍其中比较简单的几种排序算法：sort(),merge(),includes()

```
//stl_cpp_16.cpp
#include <iostream>
#include <algorithm>
using namespace std;
int main(void)
{
    int a[10]={12,0,5,3,6,8,9,34,32,18};
    int b[5]={5,3,6,8,9};
```

```
    int d[15];
    sort(a,a+10);
    for(int i=0;i<10;i++)
       cout<<" "<<a[i];
    sort(b,b+5);
    if(includes(a,a+10,b,b+5))
        cout<<"\n"<<"sorted b members are included in a."<<endl;
    else
        cout<<"sorted a dosn`t contain sorted b!";
     merge(a,a+10,b,b+5,d);
    for(int j=0;j<15;j++)
        cout<<" "<<d[j];
    cin.get();
    return 0;
}
```

sort()的原型是：

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);
```

功能是对[first,last-1]区间内的元素进行排序操作。与之类似的操作还有：partial_sort(), stable_sort()，partial_sort_copy()等等。 merge()的原型是：

```
template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,InputIterator2  first2, InputIterator2
st2,OutputIterator result);
```

将有序区间[first1,last1-1]和[first2,last2-1]合并到[result, result + (last1 - first1) + (last2 - first2)-1]区间内。

Includes()的原型是：

```
template <class InputIterator1, class InputIterator2>
bool includes(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2);
```

其功能是检查有序区间[first2,last2-1]内元素是否都在[first1,last1-1]区间内，返回一个 bool 值。

2.2.4 通用数值算法（generalized numeric algorithms）

这一类算法还不多，涉及到专业领域中有用的算术操作，独立包涵于头文件<numeric>中（HP 版本的 STL 中是<algo.h>）。这里不作介绍。

STL 中的算法大都有多种版本，常见的版本有以下 4 中：

- 默认版本，假设给出了特定操作符；
- 一般版本，使用了成员提供的操作符；
- 复制版本，对原队列的副本进行操作，常带有 _copy 后缀；
- 谓词版本，只应用于满足给定谓词的队列成员，常带有 _if 后缀；

以上我们学习了 STL 容器和算法的概念，以及一些简单的 STL 容器和算法。在使用算法处理容器内的数据时，需要从一个数据成员移向另一个数据成员，迭代器恰好实现了这一功能。下面我们来学习 STL 迭代器 。

2.3 迭代器（itertor）：#include<iterator>

迭代器实际上是一种泛化指针，如果一个迭代器指向了容器中的某一成员，那么迭代器将可以通过自增自减来遍历容器中的所有成员。迭代器是联系容器和算法的媒介，是算法操作容器的接口。在运用算法操作容器的时候，我们常常在不知不觉中已经使用了迭代器。

STL 中定义了 6 种迭代器：

- 输入迭代器，在容器的连续区间内向前移动，可以读取容器内任意值；
- 输出迭代器，把值写进它所指向的队列成员中；
- 前向迭代器，读取队列中的值，并可以向前移动到下一位置（++p,p++）；
- 双向迭代器，读取队列中的值，并可以向前向后遍历容器；
- 随机访问迭代器, vector<T>::iterator，list<T>::iterator 等都是这种迭代器 ；
- 流迭代器，可以直接输出、输入流中的值；

实际上，在前面的例子中，我们不停的在用迭代器。下面我们用几个例子来帮助理解这些迭代器的用法。

下面的例子用到了输入输出迭代器：

```
// stl_cpp_17.cpp
#include <iostream>
#include <fstream>
#include <iterator>
#include <vector>
#include <string>
using namespace std;
int main(void)
{
    vector<string> v1;
    ifstream file("Text1.txt");
    if(file.fail())
    {
      cout<<"open file Text1.txt failed"<<endl;
      return 1;
    }
  copy(istream_iterator<string>(file),istream_iterator<string>(),inserter(v1,\
v1.begin()));
    copy(v1.begin(),v1.end(),ostream_iterator<string>(cout," "));
    cout<<endl;
```

```
    cin.get();
    return 0;
}
```

这里用到了输入迭代器 istream_iterator，输出迭代器 ostream_iterator。程序完成了将一个文件输出到屏幕的功能，先将文件读入，然后通过输入迭代器把文件内容复制到类型为字符串的向量容器内，最后由输出迭代器输出。Inserter 是一个输入迭代器的一个函数(迭代器适配器)，它的使用方法是：

```
inserter (container ,pos);
```

congtainer 是将要用来存入数据的容器，pos 是容器存入数据的开始位置。上例中，是把文件内容存入（copy()）到向量 v1 中。

2.4 STL 的其他标准组件

2.4.1 函数对象（functor 或者 funtion objects）：#include <function>

函数对象又称之为仿函数。函数对象将函数封装在一个对象中，使得它可作为参数传递给合适的 STL 算法，从而使算法的功能得以扩展。可以把它当作函数来使用。用户也可以定义自己的函数对象。下面让我们来定义一个自己的函数对象.

```
// stl_cpp_18.cpp
#include <iostream>
using namespace std;
struct int_max{
    int operator()(int x,int y){return x>y?x:y; }
    };//operator()  重载了"（）",   (int x,int y)是参数列表
int main(void)
{
    cout<<int_max()(3,4)<<endl;
    cin.get();
    return 0;
}
```

这里的 int_max（）就是一个函数对象，struct 关键字也可以用 class 来代替，只不过 struct 默认情况下是公有访问权限，而 class 定义的是默认私有访问权限。下面我们来定义一个 STL 风格的函数对象：

```
// stl_cpp_19.cpp
```

```
#include <iostream>
#include <vector>
using namespace std;
struct adder : public unary_function<double, void>
{
        adder() : sum(0) {}
        double sum;
        void operator()(double x) { sum += x; }
      };
int main(void)
{
      double a[5]={0.5644,1.1,6.6,8.8,9.9};
      vector<double> V(a,a+5);
      adder result = for_each(V.begin(), V.end(), adder());
      cout << "The sum is " << result.sum << endl;
      cin.get();
      return 0;
}
```

在这里，我们定义了一个函数对象 adder()，这也是一个类，它的基类是 unary_function 函数对象。unary_function 是一个空基类，不包涵任何操作或变量。只是一种格式说明，它有两个参数，第一个参数是函数对象的使用数据类型，第二个参数是它的返回类型。基于它所定义的函数对象是一元函数对象。（注：用关键字 struct 或者 class 定义的类型实际上都是"类"）

STL 内定义了各种函数对象，否定器、约束器、一元谓词、二元谓词都是常用的函数对象。函数对象对于编程来说很重要，因为他如同对象类型的抽象一样作用于操作。

2.4.2 适配器（adapter）

适配器是用来修改其他组件接口的 STL 组件，是带有一个参数的类模板（这个参数是操作的值的数据类型）。STL 定义了 3 种形式的适配器：容器适配器，迭代器适配器，函数适配器。

- 容器适配器：包括栈（stack）、队列(queue)、优先(priority_queue)。使用容器适配器，stack 就可以被实现为基本容器类型（vector,dequeue,list）的适配。可以把 stack 看作是某种特殊的 vctor,deque 或者 list 容器，只是其操作仍然受到 stack 本身属性的限制。queue 和 priority_queue 与之类似。容器适配器的接口更为简单，只是受限比一般容器要多；
- 迭代器适配器：修改为某些基本容器定义的迭代器的接口的一种 STL 组件。反向迭代器和插入迭代器都属于迭代器适配器，迭代器适配器扩展了迭代器的功能；
- 函数适配器：通过转换或者修改其他函数对象使其功能得到扩展。这一类适配器有否定器（相当于"非"操作）、帮定器、函数指针适配器。

第二篇 算法篇

我们用广度优先学习更多的算法!

我们用深度优先学习更难的算法!

第1章　基本算法

1　算法初步

1.1 算法

算法是解决问题方法的精确描述，但是并不是所有问题都有算法，有些问题经研究可行，则相应有算法，但这并不是说问题就有结果。上述的“可行”，是指对算法的研究。

待解问题表述应精确、简练、清楚，使用形式化模型刻划问题是最恰当的。例如，使用数学模型刻划问题是最简明、严格的，一旦问题形式化了，就可依据相应严格的模型对问题求解。

算法设计的任务是对各类具体问题设计良好的算法及研究设计算法的规律和方法。常用的算法有：穷举搜索法、递归法、回溯法、贪心法、分治法等。

算法分析的任务是对设计出的每一个具体的算法，利用数学工具，讨论各种复杂度，以探讨某种具体算法适用于哪类问题，或某类问题宜采用哪种算法。

1.2 算法的复杂度分时间复杂度和空间复杂度。

时间复杂度：在运行算法时所耗费的时间为 f(n)(即 n 的函数)。

空间复杂度：实现算法所占用的空间为 g(n)(也为 n 的函数)。

称 O(f(n))和 O(g(n))为该算法的复杂度。

1.3 程序设计

程序

程序是对所要解决的问题的各个对象和处理规则的描述，或者说是数据结构和算法的描述，因此有人说，数据结构＋算法＝程序。

程序设计

程序设计就是设计、编制和调试程序的过程。

结构化程序设计

结构化程序设计是利用逐步求精的方法，按一套程式化的设计准则进行程序的设计。由这种方法产生的程序是结构良好的。所谓“结构良好”是指：

- 易于保证和验证其正确性；
- 易于阅读、易于理解和易于维护。
- 按照这种方法或准则设计出来的程序称为结构化的程序。
- “逐步求精”是对一个复杂问题，不是一步就编成一个可执行的程序，而是分步进行。

- 第一步编出的程序最为抽象；
- 第二步编出的程序是把第一步所编的程序（如过程、函数等）细化，较为抽象；
- ……
- 第 i 步编出的程序比第 i-1 步抽象级要低；
- 直到最后，第 n 步编出的程序即为可执行的程序。

所谓“抽象程序”是指程序所描述的解决问题的处理规则，是由那些“做什么”操作组成，而不涉及这些操作“怎样做”以及解决问题的对象具有什么结构，不涉及构造的每个局部细节。

这一方法原理就是：对一个问题（或任务），程序人员应立足于全局，考虑如何解决这一问题的总体关系，而不涉及每局部细节。在确保全局的正确性之后，再分别对每一局部进行考虑，每个局部又将是一个问题或任务，因而这一方法是自顶而下的，同时也是逐步求精的。采用逐步求精的优点是：

- 便于构造程序。由这种方法产生的程序，其结构清晰、易读、易写、易理解、易调试、易维护；
- 适用于大任务、多人员设计，也便于软件管理。

[例]求两自然数,其和是 667,最小公倍数与最大公约数之比是 120:1(例如(115,552) 、(232,435))。

[解]两个自然数分别为 m 和 667-m(2≤m≤ 333)。

处理对象：m(自然数)、l(两数的最小公倍数)、g（两数的最大公约数）。

处理步骤：对 m 从 2 到 333 检查 l 与 g 的商为 120,且余数为 0 时,打印 m 与 667-m 。

第一层抽象程序：

```
int main()
{
for(int m=2;m<=333;m++){
        int l=lcm(m,667-m);//求最小公倍数
        int g=gcd(m,667-m);//求最大公约数
        if(l==g*120&&l%g==0)
        cout<<m<<’  ‘<<667-m<<endl;
    }
system(“pause”);
}
```

第二层考虑函数 lcm（最小公倍数）、gcd（最大公约数）的细化。

最大公约数问题是对参数 a、b，找到一个数 i 能整除 a 与 b，i 就是 gcd 的函数值。

```
int   gcd(int a,int b)
{
        int i=a;
        if(i>b)i=b;
        while(i>0){
            if(a%i==0&&b%i==0) break;
            i--;
  }
return i;
}
```

而最小公倍数的计算是：若干个 b 之和，若能被 a 整除，则该和便是 a、b 的最小公

倍数。

```
int lcm(int a,int b)
{
        int x=a>b?b:a;
        int s=0;
        do{
s+=x;
        }while(s%a||s%b);
        return s;
}
```

1.4 编程入门题

1、位数对调：输入一个三位自然数，把这个数的百位与个位数对调，输出对调后的数。例如：Input 3 bit natrue data:234　　n=432

[解]先确定输入数 n 是否三位数，即 n>99 且 n<=999，位数对调:n=abc→cba=x　① 百位数 a=n 整除 100；② 十位数 b=(n-a*100)整除 10；③ 个位数 c=n 除以 10 的余数；得对调后的数：x=c*100+b*10+a

```
        #include<iostream>
        using namespace std;

int main()
{
        int x;
        while(cin>>x){
            int a=x%10;
            x/=10;
            int b=x%10;
            x/=10;
            cout<<a<<b<<x<<endl;
   }
}
```

2、求三角形面积：给出三角形的三个边长为 a，b，c，求三角形的面积。

提示：根据海伦公式来计算三角形的面积：

S＝ ；Area＝

[解]输入的三角形三边长 a,b,c 要满足“任意两边长的和大于第三边长”。按海伦公式计算：s=（a+b+c）/2；x=s*(s-a)*(s-b)*(s-c) 这时若 x>=0，则求面积：area= ，并输出 area 的值。

```
#include<iostream>
using namespace std;

int main()
{
```

```
        double a,b,c,s,area;
        while(cin>>a>>b>>c){
            s=（a+b+c）/2;
            area=s*(s-a)*(s-b)*(s-c);
            if(area>＝0)cout<<area<<endl;
    }
}
```

5、数列找数：数组 A(N)的各下标变量中Ｎ个互不相等的数，键盘输入正整数Ｍ（Ｍ≤Ｎ），要求打印数组中第Ｍ大的下标变量的值。

6、数制转换：编程输入十进制Ｎ（Ｎ：-32767～32767），请输出它对应的二进制、八进制、十六进制数。

综合题

1、寻找数：求所有这样的三位数，这些三位数等于它各位数字的立方和。

例如，153＝13 ＋53 ＋33 。

[解]穷尽三位数，用一个循环语句。其数码可用“模取”运算%完成。

2、最小自然数：求具有下列两个性质的最小自然数 n：

(1)n 的个位数是 6；

(2)若将 n 的个位数移到其余各位数字之前，所得的新数是 n 的 4 倍。

[解]仍用穷举法寻找，当找到一个符合条件者便停止。“找到便停止”的重复，宜采用 do while 循环。

3、找素数：寻找 160 以内的素数，它的倒序数（如 123 的倒序数为 321）、数码和、数码积不是素数便是 1。

4、完全平方数：寻找具有完全平方数，且不超过 7 位数码的回文数。所谓回文数是指这样的数，它的各位数码是左右对称的。例如 121、676、94249 等。

[解]判断一个数是否回文数，可以将其转化成字符判断。也可以分解数，所谓分解就是将数的后半段倒置后再与前半段比较。这里采用分解的方法，其函数为 symm。

5、成等差的素数：寻找 6 个成等差级数且小于 160 的素数。

[解]设级数为：n，n-d，n-2d，n-3d，n-4d，n-5d。若这 6 个数全为素数，则为要求的解。这里 d、n 均是要寻找的。仍用穷尽法，d 最大可为 33。判断素数函数为 isprime。

6、取数列：取｛2m ，3n ｜m>=1，n>=1｝中由小到大排列的前 70 项数。

[解]这个数的集合事实上存在两个数列：一个是 2m ，另一个是 3n 。若依次从两数列中取出小的进入新的数列，该新数列便为所求（这里不用数组，而直接打印输出）。

7、发奖章：运动会连续开了 n 天，一共发了 m 枚奖章，第一天发 1 枚并剩下(m-1)枚的 1/7，第二天发 2 枚并剩下的 1/7，以后每天按此规律发奖章，在最后一天即第 n 天发了剩下的 n 枚奖章。问运动会开了多少天？一共发了几枚奖章？

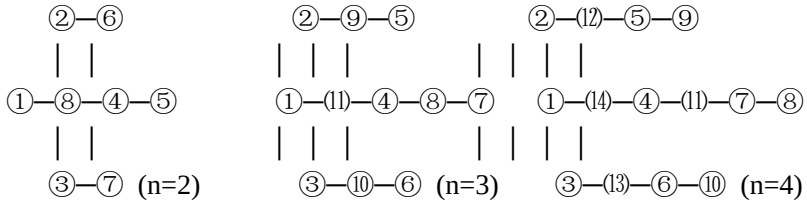
[解]由于题目涉及 m-1 的 1/7，于是 m-1 应是 7 的倍数，即 m=7x+1。递推 x，寻找 m、n。

8、猜名次：五个学生 A、B、C、D、E 参加某一项比赛。甲、乙两人在猜测比赛的结果。甲猜的名次顺序为 A、B、C、D、E，结果没有猜中任何一个学生的名次，也没有猜中任

何一对相邻名次（所谓一对相邻名次，是指其中一对选手在名次上邻接。例如 1 与 2 ，或者 2 与 3 等）。乙猜的名次顺序为 D、A、E、C、B，结果猜中了两个学生的名次，并猜对了两对学生名次是相邻的。问比赛结果如何？答案为：E、D、A、C、B。乙猜对 C、B 为最后两名，两对相邻为（D、A）、（C、B））。

[解]设五名选手 A、B、C、D、E 的编号分别为 1、2、3、4、5。用五个变量 c1、c2、c3、c4、c5 标记第一名至第五名。算法仍用穷尽法。其中处理相邻问题用一个两位数表示，即 DA、AE、EC、CB 分别用 41、15、53、32 表示，并按两位数比较判断相邻问题。

9 、填自然数：设有如图所示的 3n＋2 个球互连，将自然数 1－3n＋2 分别为这些球编号，使如图相连的球编号之差的绝对正好是数列 1，2，……，3n+2 中各数。



[解]填自然数的一种算法是：

(1)先自左向右，第 1 列中间 1 个填数，然后第 2 列上、下 2 个填数，每次 2 列；但若 n 是奇数，最后 1 次只排第 1 列中间 1 个数。

(2)自右向左，先右第 1 列中间填数；若 n 是奇数，再右第 2 列中间填数。然后依次右第 1 列上、下 2 个填数，再右第 2 列中间 1 个填数，直到左第 2 列为止。

2 分治算法

2.1 简介

对于一个规模为 n 的问题，若该问题可以容易地解决（比如说规模 n 较小）则直接解决，否则将其分解为 k 个规模较小的子问题，这些子问题互相独立且与原问题形式相同，递归地解这些子问题，然后将各子问题的解合并得到原问题的解。这种算法设计策略叫做分治法。

2.2 分治法的基本思想

任何一个可以用计算机求解的问题所需的计算时间都与其规模有关。问题的规模越小，越容易直接求解，解题所需的计算时间也越少。例如，对于 n 个元素的排序问题，当 n=1 时，不需任何计算。n=2 时，只要作一次比较即可排好序。n=3 时只要作 3 次比较即可，…。而当 n 较大时，问题就不那么容易处理了。要想直接解决一个规模较大的问题，有时是相当困难的。分治法的设计思想是，将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之如果原问题可分割成 k 个子问题，$1 < k \leqslant n$ ，且这些子问题都可解，并可利用这些子问题的解求出原问题的解，那么这种分治法就是可行的。由分治法产生的子问题往往是原问题的较小模式，这就为使用递归技术提供了方便。在这种情况下，反复应用分治手段，可以使子问题与原问题类型一致而其规模却不断缩小，最终使子问题缩小到很容易直接求出其解。这自然导致递归过程的产生。分治与递归像一对孪生兄弟，经常同时应用在算法设计之中，并由此产生许多高效算法。

2.3 分治法的适用条件

分治法所能解决的问题一般具有以下几个特征：

1。该问题的规模缩小到一定的程度就可以容易地解决；

2。该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质。

3。利用该问题分解出的子问题的解可以合并为该问题的解；

4。该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子子问题。

上述的第一条特征是绝大多数问题都可以满足的，因为问题的计算复杂性一般是随着问题规模的增加而增加；第二条特征是应用分治法的前提，它也是大多数问题可以满足的，此特征反映了递归思想的应用；第三条特征是关键，能否利用分治法完全取决于问题是否具有第三条特征，如果具备了第一条和第二条特征，而不具备第三条特征，则可以考虑贪心法或动态规划法。第四条特征涉及到分治法的效率，如果各子问题是不独立的，则分治法要做许多不必要的工作，重复地解公共的子问题，此时虽然可用分治法，但一般用动态规划法较好。

2.4 分治法的基本步骤

分治法在每一层递归上都有三个步骤：

1。分解：将原问题分解为若干个规模较小，相互独立，与原问题形式相同的子问题；

2。解决：若子问题规模较小而容易被解决则直接解，否则递归地解各个子问题；

3。合并：将各个子问题的解合并为原问题的解。

它的一般的算法设计模式如下：

```
Divide-and-Conquer(P)
1。    if |P|≤n0
2。      then return( ADHOC(P) )
3。    将 P 分解为较小的子问题 P1 ,P2 ,...,Pk
4。    for i←1 to k
5。      do yi ← Divide-and-Conquer(Pi)              △ 递归解决 Pi
6。    T ← MERGE(y1,y2,...,yk)                       △ 合并子问题
7。    return(T)
```

其中|P|表示问题 P 的规模；n0 为一阈值，表示当问题 P 的规模不超过 n0 时，问题已容易直接解出，不必再继续分解。ADHOC(P)是该分治法中的基本子算法，用于直接解小规模的问题 P。因此，当 P 的规模不超过 n0 时，直接用算法 ADHOC(P)求解。算法 MERGE(y1,y2,...,yk)是该分治法中的合并子算法，用于将 P 的子问题 P1 ,P2 ,...,Pk 的相应的解 y1,y2,...,yk 合并为 P 的解。

根据分治法的分割原则，原问题应该分为多少个子问题才较适宜？各个子问题的规模应该怎样才为适当？这些问题很难予以肯定的回答。但人们从大量实践中发现，在用分治法设计算法时，最好使子问题的规模大致相同。换句话说，将一个问题分成大小相等的 k 个子问题的处理方法是行之有效的。许多问题可以取 k=2。这种使子问题规模大致相等的做法是出自一种平衡(balancing)子问题的思想，它几乎总是比子问题规模不等的做法要好。

分治法的合并步骤是算法的关键所在。有些问题的合并方法比较明显，如下面的例 1，例 2；有些问题合并方法比较复杂，或者是有多种合并方案，如例 3，例 4；或者是合并方案不明显，如例 5。究竟应该怎样合并，没有统一的模式，需要具体问题具体分析。

2.5 分治法的复杂性分析

从分治法的一般设计模式可以看出，用它设计出的程序一般是一个递归过程。因此，分治法的计算效率通常可以用递归方程来进行分析。为方便起见，设分解阈值 n0=1，且算法 ADHOC 解规模为 1 的问题耗费 1 个单位时间。又设分治法将规模为 n 的问题分成 k 个规模为 n/m 的子问题去解，而且，将原问题分解为 k 个子问题以及用算法 MERGE 将 k 个子问题的解合并为原问题的解需用 f(n)个单位时间。如果用 T(n)表示该分治法 Divide-and-Conquer(P)解规模为|P|=n 的问题 P 所需的计算时间，则有:

$$T(1) = 1$$

$$T(n) = k*T(n/m)+f(n) \quad \text{..... (1)}$$

用迭代法，可以求得(1)的解:

$$T(n) = n^{(\lg(k)/\lg(m))} + \sum k^j * f(n/m^j) , j = 0..\lg(n-1)/\lg(m)-1 \quad \text{.....(2)}$$

注意，递归方程(1)及其解(2)只给出 n 等于 m 的方幂时 T(n)的值，但是如果认为 T(n)足够平滑，那么由等于 m 的方幂时 T(n)的值可以估计 T(n)的增长速度。通常，我们可以

假定 T(n)是单调上升的，从而当 m^i≤n<m^(i+1)时，T(m^i)≤T(n)<T(m^(i+1))。

另一个需要注意的问题是，在分析分治法的计算效率时，通常得到的是递归不等式：

T(n0) ≤ c

T(n) ≤ k*T(n/m) + f(n), n > n0 ..... (3)

由于我们关心的一般是最坏情况下的计算时间复杂度的上界，所以用等于号(=)还是小于或等于号(≤)是没有本质区别的。

2.6 分治法的几种变形

1。二分法 dichotomy

一种每次将原问题分解为两个子问题的分治法，是一分为二的哲学思想的应用。这种方法很常用，由此法产生了许多经典的算法和数据结构。

2。分解并在解决之前合并法 divide and marriage before conquest

一种分治法的变形，其特点是将分解出的子问题在解决之前合并。

divide and marriage before conquest:

A variant of divide and conquer in which subproblems created in the

“divide” step are merged before the “conquer” step.

3。管道传输分治法 pipelined divide and conquer

一种分治法的变形，它利用某种称为“管道”的数据结构在递归调用结束前将其中的某些结果返回。此方法经常用来减少算法的深度。

pipelined divide and conquer:

A divide and conquer paradigm in which partial results from recursive

calls can be used before the calls complete. The technique is often

useful for reducing the depth of an algorithm.

2.7 分治法的实例分析

以上讨论的是分治法的基本思想和一般原则，下面我们用具体的例子来说明如何针对具体问题用分治法来设计有效解法

例 1 和例 2 是分治法的经典范例，其分解和合并过程都比较简单明显；例 3 和例 4 的合并方法有多种选择，只有选择最好的合并方法才能够改进算法的复杂度；例 5 是一个计算几何学中的问题，它的合并步骤需要较高的技巧。例 6 则是 IOI’95 的试题 Wires and Switches 。

例 1 二分查找

例 2 快速排序

例 3 大整数乘法

例 4 Strassen 矩阵乘法

例 5 最接近点对问题

例 6 导线和开关

更多实例请参阅分治法问题

3 搜索算法

3.1 搜索简介

寻找问题的解的一种可靠的方法是首先列出所有候选解，然后依次检查每一个，在检查完所有或部分候选解后，即可找到所需要的解。理论上，当候选解数量有限并且通过检查所有或部分候选解能够得到所需解时，上述方法是可行的。不过，在实际应用中，很少使用这种方法，因为候选解的数量通常都非常大（比如指数级，甚至是大数阶乘），即便采用最快的计算机也只能解决规模很小的问题。对候选解进行系统检查的方法有多种，其中回溯和分枝定界法是比较常用的两种方法。按照这两种方法对候选解进行系统检查通常会使问题的求解时间大大减少（无论对于最坏情形还是对于一般情形）。事实上，这些方法可以使我们避免对很大的候选解集合进行检查，同时能够保证算法运行结束时可以找到所需要的解。因此，这些方法通常能够用来求解规模很大的问题。

3.2 基本搜索算法

3.2.1 回溯算法

回溯算法是所有搜索算法中最为基本的一种算法，其采用了一种“走不通就掉头”思想作为其控制结构，其相当于采用了先根遍历的方法来构造解答树，可用于找解或所有解以及最优解。具体的算法描述如下：

[非递归算法]

```
struct Node{
Situtation:TSituation（当前节点状态）;
int Way-NO（已使用过的扩展规则的数目）;
}
Node List[MaxNode] (回溯表);
int pos(当前扩展节点编号);
<Init>
List<-0;
pos<-0;
List[0].Situation<-初始状态;
<Main Program>
While( (pos>0(有路可走)) && ([未达到目标]) ){
if( pos>=Max) break; (数据溢出,跳出主程序);
List[pos].Way-NO=List[pos].Way-No+1;
If (List[pos].Way-NO<=TotalExpendMethod)   (如果还有没用过的扩展规则)
   {
If (可以使用当前扩展规则) {
(用第 way 条规则扩展当前节点)
```

```
List[pos+1].Situation=ExpendNode(List[pos].Situation, List[pos].Way-NO);
List[pos+1].Way-NO=0;
pos=pos+1;
      }
   }
else    pos=pos-1;
}

[递归算法]
Type BackTrack(Situation:TSituation;deepth:Integer);
{
if (deepth>Max) then (空间达到极限或找不到目标,跳出本过程);
If (Situation==Target){  和最优解比较，记录    return Type;} (找到目标);
for( int i=0 ;i<TotalExpendMethod;i++){
BackTrack(ExpendNode(Situation,I),deepth+1);
   }
}
```

范例：一个 M*M 的棋盘上某一点上有一个马，要求寻找一条从这一点出发不重复的跳完棋盘上所有的点的路线。

评价：回溯算法对空间的消耗较少，当其与分枝定界法一起使用时，对于所求解在解答树中层次较深的问题有较好的效果。但应避免在后继节点可能与前继节点相同的问题中使用，以免产生循环。

3.2.2 深度搜索与广度搜索

深度搜索与广度搜索的控制结构和产生系统很相似，唯一的区别在于对扩展节点选取上。由于其保留了所有的前继节点，所以在产生后继节点时可以去掉一部分重复的节点，从而提高了搜索效率。这两种算法每次都扩展一个节点的所有子节点，而不同的是，深度搜索下一次扩展的是本次扩展出来的子节点中的一个，而广度搜索扩展的则是本次扩展的节点的兄弟节点。在具体实现上为了提高效率,所以采用了不同的数据结构

[广度搜索]

```
#include<queue>
using namespace std;
struct Node{
TSituation Situtation;（当前节点状态）;
      int Level(当前节点深度);
      int Last; (父节点);
};
int main(){
queue<Node> q;
q.push(init_situation); //  初始状态
bool found=0;                //未找到最解
While (!q.empty()&&!found) {
```

```
for(int i=0;i<ExpendMethod;i++){（扩展一层子节点）
Node old=q.front();q.pop(); //取出一个结点进行扩展
Node New=ExpendNode(List[close].Situation,i);//用第 i 种方法对结点进行扩展
if(New-S is new){   (扩展出的节点从未出现过)
new.level=old.level+1;
new.Last=old;
            }
        }
    }
}
```

范例：迷宫问题，求解最短路径和可通路径。

评价：广度搜索是求解最优解的一种较好的方法，在后面将会对其进行进一步的优化。而深度搜索多用于只要求解，并且解答树中的重复节点较多并且重复较难判断时使用，但往往可以用 A*或回溯算法代替。

3.3 搜索算法的优化

3.3.1 双向广度搜索

广度搜索虽然可以得到最优解，但是其空间消耗增长太快。但如果从正反两个方向进行广度搜索，理想情况下可以减少二分之一的搜索量，从而提高搜索速度。

范例：有 N 个黑白棋子排成一派，中间任意两个位置有两个连续的空格。每次空格可以与序列中的某两个棋子交换位置，且两子的次序不变。要求出入长度为 length 的一个初始状态和一个目标状态，求出最少的转化步数。

问题分析：该题要求求出最少的转化步数，但如果直接使用广度搜索，很容易产生数据溢出。但如果从初始状态和目标状态两个方向同时进行扩展，如果两棵解答树在某个节点第一次发生重合，则该节点所连接的两条路径所拼成的路径就是最优解。

对广度搜索算法的改进：

１．添加一张节点表，作为反向扩展表。

２．在 while 循环体中在正向扩展代码后加入反向扩展代码，其扩展过程不能与正向过程共享一个 for 循环。

３．在正向扩展出一个节点后，需在反向表中查找是否有重合节点。反向扩展时与之相同。

对双向广度搜索算法的改进：

略微修改一下控制结构，每次 while 循环时只扩展正反两个方向中节点数目较少的一个，可以使两边的发展速度保持一定的平衡，从而减少总扩展节点的个数，加快搜索速度。

3.3.2 分支定界

分支定界实际上是 A*算法的一种雏形，其对于每个扩展出来的节点给出一个预期值，如果这个预期值不如当前已经搜索出来的结果好的话，则将这个节点(包括其子节点)从解答树中删去，从而达到加快搜索速度的目的。

范例：在一个商店中购物，设第 I 种商品的价格为 Ci。但商店提供一种折扣，即给出一组商品的组合，如果一次性购买了这一组商品，则可以享受较优惠的价格。现在给出一张购买清单和商店所提供的折扣清单，要求利用这些折扣，使所付款最少。

问题分析：显然，折扣使用的顺序与最终结果无关，所以可以先将所有的折扣按折扣率从大到小排序，然后采用回溯法的控制结构，对每个折扣从其最大可能使用次数向零递减搜索，设 A 为以打完折扣后优惠的价格，C 为当前未打折扣的商品零售价之和，则其预期值为 A+a*C，其中 a 为下一个折扣的折扣率。如当前已是最后一个折扣，则 a=1。

对回溯算法的改进：

１．添加一个全局变量 BestAnswer，记录当前最优解。

２．在每次生成一个节点时，计算其预期值，并与 BestAnswer 比较。如果不好，则调用回溯过程。

3.3.3 A*算法

A*算法中更一般的引入了一个估价函数 f,其定义为 f=g+h。其中 g 为到达当前节点的耗费，而 h 表示对从当前节点到达目标节点的耗费的估计。其必须满足两个条件：

１．h 必须小于等于实际的从当前节点到达目标节点的最小耗费 h*。

２．f 必须保持单调递增。

A*算法的控制结构与广度搜索的十分类似，只是每次扩展的都是当前待扩展节点中 f 值最小的一个，如果扩展出来的节点与已扩展的节点重复，则删去这个节点。如果与待扩展节点重复，如果这个节点的估价函数值较小，则用其代替原待扩展节点.

范例：一个 3*3 的棋盘中有 1-8 八个数字和一个空格，现给出一个初始态和一个标态，要求利用这个空格，用最少的步数，使其到达目标态。

问题分析：预期值定义为 h=|x-dx|+|y-dy|。

估价函数定义为 f=g+h。

```
<Type>
Node(节点类型)＝Record
Situtation:TSituation（当前节点状态）;
g:Integer;(到达当前状态的耗费)
h:Integer;(预计的耗费)
f:Real;(估价函数值)
Last:Integer;(父节点)
End
<Var>
List(节点表):Array[1..Max(最多节点数)] of Node(节点类型);
open(总节点数):Integer;
close(待扩展节点编号):Integer;
New-S:Tsituation;(新节点)
<Init>
List<-0;
open<-1;
close<-0;
List[1].Situation<- 初始状态;
```

```
<Main Program>
While (close<open(还有未扩展节点)) and
(open<Max(空间未用完)) and
(未找到目标节点) do
Begin
Begin
close:=close+1;
For I:=close+1 to open do (寻找估价函数值最小的节点)
Begin
if List[i].f<List[close].f then
Begin
交换 List[i]和 List[close];
End-If;
End-For;
End;
For I:=1 to TotalExpendMethod do（扩展一层子节点）
Begin
New-S:=ExpendNode(List[close].Situation,I)
If Not (New-S in List[1..close]) then
(扩展出的节点未与已扩展的节点重复)
Begin
If Not (New-S in List[close+1..open]) then
(扩展出的节点未与待扩展的节点重复)
Begin
open:=open+1;
List[open].Situation:=New-S;
List[open].Last:=close;
List[open].g:=List[close].g+cost;
List[open].h:=GetH(List[open].Situation);
List[open].f:=List[open].h+List[open].g;
End-If
Else Begin
If List[close].g+cost+GetH(New-S)<List[same].f then
(扩展出来的节点的估价函数值小于与其相同的节点)
Begin
List[same].Situation:= New-S;
List[same].Last:=close;
List[same].g:=List[close].g+cost;
List[same].h:=GetH(List[open].Situation);
List[same].f:=List[open].h+List[open].g;
End-If;
End-Else;
End-If
End-For;
```

End-While;

对 A*算法的改进—分阶段 A*：　　当 A*算法出现数据溢出时，从待扩展节点中取出若干个估价函数值较小的节点，然后放弃其余的待扩展节点，从而可以使搜索进一步的进行下去。

4 贪婪算法

4.1 贪婪算法简介

虽然设计一个好的求解算法更像是一门艺术，而不像是技术，但仍然存在一些行之有效的能够用于解决许多问题的算法设计方法，你可以使用这些方法来设计算法，并观察这些算法是如何工作的。一般情况下，为了获得较好的性能，必须对算法进行细致的调整。但是在某些情况下，算法经过调整之后性能仍无法达到要求，这时就必须寻求另外的方法来求解该问题。

4.2 最优化问题

本章及后续章节中的许多例子都是最优化问题（ optimization problem），每个最优化问题都包含一组限制条件（ c o n s t r a i n t）和一个优化函数（ optimization function），符合限制条件的问题求解方案称为可行解（ feasible solution），使优化函数取得最佳值的可行解称为最优解（optimal solution）。

例 1-1 [渴婴问题] 有一个非常渴的、聪明的小婴儿，她可能得到的东西包括一杯水、一桶牛奶、多罐不同种类的果汁、许多不同的装在瓶子或罐子中的苏打水，即婴儿可得到 n 种不同的饮料。根据以前关于这 n 种饮料的不同体验，此婴儿知道这其中某些饮料更合自己的胃口，因此，婴儿采取如下方法为每一种饮料赋予一个满意度值：饮用 1 盎司第 i 种饮料，对它作出相对评价，将一个数值 si 作为满意度赋予第 i 种饮料。

通常，这个婴儿都会尽量饮用具有最大满意度值的饮料来最大限度地满足她解渴的需要，但是不幸的是：具有最大满意度值的饮料有时并没有足够的量来满足此婴儿解渴的需要。设 ai 是第 i 种饮料的总量（以盎司为单位），而此婴儿需要 t 盎司的饮料来解渴，那么，需要饮用 n 种不同的饮料各多少量才能满足婴儿解渴的需求呢？

设各种饮料的满意度已知。令 xi 为婴儿将要饮用的第 i 种饮料的量，则需要解决的问题是：

找到一组实数 xi（1≤i≤n），使 n ?i = 1si xi 最大，并满足：n ?i=1xi =t 及 0≤xi≤ai 。

需要指出的是：如果 n ?i = 1ai < t，则不可能找到问题的求解方案，因为即使喝光所有的饮料也不能使婴儿解渴。

对上述问题精确的数学描述明确地指出了程序必须完成的工作，根据这些数学公式，可以对输入/ 输出作如下形式的描述：

输入：n，t，si ，ai（其中 1≤i≤n，n 为整数，t、si 、ai 为正实数）。

输出：实数 xi（1≤i≤n），使 n ?i= 1si xi 最大且 n ?i=1xi =t（0≤xi≤ai）。

如果 n ?i = 1ai <t，则输出适当的错误信息。

在这个问题中，限制条件是 n ?i= 1xi =t 且 0≤xi≤ai，1≤i≤n。而优化函数是 n ?i= 1si xi 。任何满足限制条件的一组实数 xi 都是可行解，而使 n ?i= 1si xi 最大的可行解是最优解。

例 1-2 [装载问题] 有一艘大船准备用来装载货物。所有待装货物都装在货箱中且所有货箱的大小都一样，但货箱的重量都各不相同。设第 i 个货箱的重量为 wi（1≤i≤n），而货船的最大载重量为 c，我们的目的是在货船上装入最多的货物。

这个问题可以作为最优化问题进行描述：设存在一组变量 xi ，其可能取值为 0 或 1。如 xi 为 0，则货箱 i 将不被装上船；如 xi 为 1，则货箱 i 将被装上船。我们的目的是找到一组 xi ，使它满足限制条件 n ?i = 1wi xi ≤c 且 x i ? {0, 1}, 1 ≤i≤n。相应的优化函数是 n ?i= 1xi 。

满足限制条件的每一组 xi 都是一个可行解，能使 n ?i= 1xi 取得最大值的方案是最优解。

例 1-3 [最小代价通讯网络] 城市及城市之间所有可能的通信连接可被视作一个无向图，图的每条边都被赋予一个权值，权值表示建成由这条边所表示的通信连接所要付出的代价。包含图中所有顶点（城市）的连通子图都是一个可行解。设所有的权值都非负，则所有可能的可行解都可表示成无向图的一组生成树，而最优解是其中具有最小代价的生成树。在这个问题中，需要选择一个无向图中的边集合的子集，这个子集必须满足如下限制条件：所有的边构成一个生成树。而优化函数是子集中所有边的权值之和。

4.3 算法思想

在贪婪算法（greedy method）中采用逐步构造最优解的方法。在每个阶段，都作出一个看上去最优的决策（在一定的标准下）。决策一旦作出，就不可再更改。作出贪婪决策的依据称为贪婪准则（greedy criterion）。

例 1-4 [找零钱] 一个小孩买了价值少于 1 美元的糖，并将 1 美元的钱交给售货员。售货员希望用数目最少的硬币找给小孩。假设提供了数目不限的面值为 2 5 美分、1 0 美分、5 美分、及 1 美分的硬币。售货员分步骤组成要找的零钱数，每次加入一个硬币。选择硬币时所采用的贪婪准则如下：每一次选择应使零钱数尽量增大。为保证解法的可行性（即：所给的零钱等于要找的零钱数），所选择的硬币不应使零钱总数超过最终所需的数目。

假设需要找给小孩 6 7 美分，首先入选的是两枚 2 5 美分的硬币，第三枚入选的不能是 2 5 美分的硬币，否则硬币的选择将不可行（零钱总数超过 6 7 美分），第三枚应选择 1 0 美分的硬币，然后是 5 美分的，最后加入两个 1 美分的硬币。

贪婪算法有种直觉的倾向，在找零钱时，直觉告诉我们应使找出的硬币数目最少（至少是接近最少的数目）。可以证明采用上述贪婪算法找零钱时所用的硬币数目的确最少（见练习 1）。

例 1-5 [机器调度] 现有 n 件任务和无限多台的机器，任务可以在机器上得到处理。每件任务的开始时间为 si，完成时间为 fi ，si < fi 。[si , fi] 为处理任务 i 的时间范围。两个任务 i，j 重指两个任务的时间范围区间有重叠，而并非是指 i，j 的起点或终点重合。例如：区间[1，4]与区间[2，4]重叠，而与区间[4，7]不重叠。一个可行的任务分配是指在分配中没有两件重叠的任务分配给同一台机器。因此，在可行的分配中每台机器在任何时刻最多只处理一个任务。最优分配是指使用的机器最少的可行分配方案。

一种获得最优分配的贪婪方法是逐步分配任务。每步分配一件任务，且按任务开始时间的非递减次序进行分配。若已经至少有一件任务分配给某台机器，则称这台机器是旧的；若机器非旧，则它是新的。在选择机器时，采用以下贪婪准则：根据欲分配任务的开始时间，若此时有旧的机器可用，则将任务分给旧的机器。否则，将任务分配给一台新的机器。

根据例子中的数据，贪婪算法共分为 n = 7 步，任务分配的顺序为 a、f、b、c、g、e、d。第一步没有旧机器，因此将 a 分配给一台新机器（比如 M1）。这台机器在 0 到 2 时刻处于忙状态。在第二步，考虑任务 f。由于当 f 启动时旧机器仍处于忙状态，因此将 f 分配给一台新机器(设为 M2)。第三步考虑任务 b, 由于旧机器 M1 在 Sb = 3 时刻已处于闲状

态，因此将 b 分配给 M1 执行，M1 下一次可用时刻变成 fb = 7，M2 的可用时刻变成 ff = 5。第四步，考虑任务 c。由于没有旧机器在 Sc = 4 时刻可用，因此将 c 分配给一台新机器（M3），这台机器下一次可用时间为 fc = 7。第五步考虑任务 g，将其分配给机器 M2，第六步将任务 e 分配给机器 M1，最后在第七步，任务 2 分配给机器 M3。（注意：任务 d 也可分配给机器 M2）。

可按如下方式实现一个复杂性为 O (nl o gn)的贪婪算法：首先采用一个复杂性为 O (nl o gn)的排序算法（如堆排序）按 Si 的递增次序排列各个任务，然后使用一个关于旧机器可用间的最小堆。

例 1-6 [最短路径] 给出一个有向网络，路径的长度定义为路径所经过的各边的耗费之和。要求找一条从初始顶点 s 到达目的顶点 d 的最短路径。贪婪算法分步构造这条路径，每一步在路径中加入一个顶点。假设当前路径已到达顶点 q 且顶点 q 并不是目的顶点 d。加入下一个顶点所采用的贪婪准则为：选择离 q 最近且目前不在路径中的顶点。这种贪婪算法并不一定能获得最短路径。例如，假设在图 1 3 - 2 中希望构造从顶点 1 到顶点 5 的最短路径，利用上述贪婪算法，从顶点 1 开始并寻找目前不在路径中的离顶点 1 最近的顶点。到达顶点 3，长度仅为 2 个单位，从顶点 3 可以到达的最近顶点为 4，从顶点 4 到达顶点 2，最后到达目的顶点 5。所建立的路径为 1 , 3 , 4 , 2 , 5，其长度为 1 0。这条路径并不是有向图中从 1 到 5 的最短路径。事实上，有几条更短的路径存在，例如路径 1，4，5 的长度为 6。

根据上面三个例子，回想一下前几章所考察的一些应用，其中有几种算法也是贪婪算法。例如，霍夫曼树算法，利用 n- 1 步来建立最小加权外部路径的二叉树，每一步都将两棵二叉树合并为一棵，算法中所使用的贪婪准则为：从可用的二叉树中选出权重最小的两棵。L P T 调度规则也是一种贪婪算法，它用 n 步来调度 n 个作业。首先将作业按时间长短排序，然后在每一步中为一个任务分配一台机器。选择机器所利用的贪婪准则为：使目前的调度时间最短。将新作业调度到最先完成的机器上（即最先空闲的机器）。

注意到在机器调度问题中，贪婪算法并不能保证最优，然而，那是一种直觉的倾向且一般情况下结果总是非常接近最优值。它利用的规则就是在实际环境中希望人工机器调度所采用的规则。算法并不保证得到最优结果，但通常所得结果与最优解相差无几，这种算法也称为启发式方法（ h e u r i s t i c s ）。因此 L P T 方法是一种启发式机器调度方法。定理 9 - 2 陈述了 L P T 调度的完成时间与最佳调度的完成时间之间的关系，因此 L P T 启发式方法具有限定性能（ bounded performance ）。具有限定性能的启发式方法称为近似算法（ a p p r o x i m a t i o n a l g o r i t h m）。

本章的其余部分将介绍几种贪婪算法的应用。在有些应用中，贪婪算法所产生的结果总是最优的解决方案。但对其他一些应用，生成的算法只是一种启发式方法，可是也可能不是近似算法。

4. 4 应用

4.4.1 货箱装船

这个问题来自例 1 - 2。船可以分步装载，每步装一个货箱，且需要考虑装载哪一个货箱。根据这种思想可利用如下贪婪准则：从剩下的货箱中，选择重量最小的货箱。这种选择次序可以保证所选的货箱总重量最小，从而可以装载更多的货箱。根据这种贪婪策略，首先选择最轻的货箱，然后选次轻的货箱，如此下去直到所有货箱均装上船或船上不能再

容纳其他任何一个货箱。

例 1-7 假设 n =8, [w1 , ... w8]=[100,200,50,90,150,50,20,80], c= 4 0 0。利用贪婪算法时，所考察货箱的顺序为 7 , 3 , 6 , 8 , 4 , 1 , 5 , 2。货箱 7 , 3 , 6 , 8 , 4 , 1 的总重量为 3 9 0 个单位且已被装载，剩下的装载能力为 1 0 个单位，小于剩下的任何一个货箱。在这种贪婪解决算法中得到[x1 , ..., x8]= [1 , 0 , 1 , 1 , 0 , 1 , 1 , 1]且?xi = 6。

定理 1-1 利用贪婪算法能产生最佳装载。

证明可以采用如下方式来证明贪婪算法的最优性：令 x = [x1 , ..., xn]为用贪婪算法获得的解，令 y =[y1 , ..., yn]为任意一个可行解，只需证明 n ?i= 1xi ≥n ?i= 1yi 。不失一般性，可以假设货箱都排好了序：即 wi≤wi + 1（1≤i≤n）。然后分几步将 y 转化为 x，转换过程中每一步都产生一个可行的新 y，且 n ?i = 1yi 大于等于未转化前的值，最后便可证明 n ?i = 1xi ≥n ?j = 1yi 。

根据贪婪算法的工作过程，可知在[0, n] 的范围内有一个 k，使得 xi =1, i≤k 且 xi =0, i>k。寻找[1 ,n]范围内最小的整数 j，使得 xj≠yj 。若没有这样的 j 存在，则 n ?i= 1xi =n ?i = 1yi 。如果有这样的 j 存在，则 j≤k，否则 y 就不是一 yj = 1，若结果得到的 y 不是可行解，则在[j+ 1 ,n]范围内必有一个 l 使得 yl = 1。令 yl = 0，由于 wj≤wl ，则得到的 y 是可行的。而且，得到的新 y 至少与原来的 y 具有相同数目的 1。

经过数次这种转化，可将 y 转化为 x。由于每次转化产生的新 y 至少与前一个 y 具有相同数目的 1，因此 x 至少与初始的 y 具有相同的数目 1。货箱装载算法的 C + +代码实现见程序 1 3 - 1。由于贪婪算法按货箱重量递增的顺序装载，程序 1 3 - 1 首先利用间接寻址排序函数 I n d i r e c t S o r t 对货箱重量进行排序，随后货箱便可按重量递增的顺序装载。由于间接寻址排序所需的时间为 O (nl o gn).

程序 13-1 货箱装船

```
template<class T>
void ContainerLoading(int x[], T w[], T c, int n)
{// 货箱装船问题的贪婪算法
// x[i] = 1 当且仅当货箱 i 被装载，  1<=i<=n
// c 是船的容量, w 是货箱的重量
// 对重量按间接寻址方式排序
// t 是间接寻址表
int *t = new int [n+1];
I n d i r e c t S o r t ( w, t, n);
// 此时, w[t[i]] <= w[t[i+1]], 1<=i<n
// 初始化 x
for (int i = 1; i <= n; i++)
x[i] = 0;
// 按重量次序选择物品
for (i = 1; i <= n && w[t[i]] <= c; i++) {
x[t[i]] = 1;
c -= w[t[i]];} // 剩余容量
delete [] t;
}
```

4.4.2 0/1 背包问题

在 0 / 1 背包问题中，需对容量为 c 的背包进行装载。从 n 个物品中选取装入背包的物品，每件物品 i 的重量为 wi ，价值为 pi 。对于可行的背包装载，背包中物品的总重量不能超过背包的容量，最佳装载是指所装入的物品价值最高，即 n ?i=1pi xi 取得最大值。约束条件为 n ?i =1wi xi≤c 和 xi?[0 , 1] (1≤i≤n)。

在这个表达式中，需求出 xt 的值。xi = 1 表示物品 i 装入背包中，xi =0 表示物品 i 不装入背包。0 / 1 背包问题是一个一般化的货箱装载问题，即每个货箱所获得的价值不同。货箱装载问题转化为背包问题的形式为：船作为背包，货箱作为可装入背包的物品。

例 1-8　在杂货店比赛中你获得了第一名，奖品是一车免费杂货。店中有 n 种不同的货物。规则规定从每种货物中最多只能拿一件，车子的容量为 c，物品 i 需占用 wi 的空间，价值为 pi 。你的目标是使车中装载的物品价值最大。当然，所装货物不能超过车的容量，且同一种物品不得拿走多件。这个问题可仿照 0 / 1 背包问题进行建模，其中车对应于背包，货物对应于物品。

0 / 1 背包问题有好几种贪婪策略，每个贪婪策略都采用多步过程来完成背包的装入。在每一步过程中利用贪婪准则选择一个物品装入背包。一种贪婪准则为：从剩余的物品中，选出可以装入背包的价值最大的物品，利用这种规则，价值最大的物品首先被装入（假设有足够容量），然后是下一个价值最大的物品，如此继续下去。这种策略不能保证得到最优解。例如，考虑 n=2, w=[100,10,10], p =[20,15,15], c = 1 0 5。当利用价值贪婪准则时，获得的解为 x= [1 , 0 , 0]，这种方案的总价值为 2 0。而最优解为[0 , 1 , 1]，其总价值为 3 0。

另一种方案是重量贪婪准则是：从剩下的物品中选择可装入背包的重量最小的物品。虽然这种规则对于前面的例子能产生最优解，但在一般情况下则不一定能得到最优解。考虑 n= 2 ,w=[10,20], p=[5,100], c= 2 5。当利用重量贪婪策略时，获得的解为 x =[1,0], 比最优解[0 , 1]要差。

还可以利用另一方案，价值密度 pi /wi 贪婪算法，这种选择准则为：从剩余物品中选择可装入包的 pi /wi 值最大的物品，这种策略也不能保证得到最优解。利用此策略试解 n= 3 ,w=[20,15,15], p=[40,25,25], c=30 时的最优解.

我们不必因所考察的几个贪婪算法都不能保证得到最优解而沮丧， 0 / 1 背包问题是一个 N P-复杂问题。对于这类问题，也许根本就不可能找到具有多项式时间的算法。虽然按 pi /wi 非递（增）减的次序装入物品不能保证得到最优解，但它是一个直觉上近似的解。我们希望它是一个好的启发式算法，且大多数时候能很好地接近最后算法。在 6 0 0 个随机产生的背包问题中，用这种启发式贪婪算法来解有 2 3 9 题为最优解。有 5 8 3 个例子与最优解相差 1 0 %，所有 6 0 0 个答案与最优解之差全在 2 5 %以内。该算法能在 O (nl o gn) 时间内获得如此好的性能。我们也许会问，是否存在一个 x (x<1 0 0)，使得贪婪启发法的结果与最优值相差在 x%以内。

答案是否定的。为说明这一点，考虑例子 n =2, w = [1 ,y], p= [1 0 , 9y], 和 c= y。贪婪算法结果为 x=[1,0], 这种方案的值为 1 0。对于 y≥1 0 / 9，最优解值为 9 y。因此，贪婪算法的值与最优解的差对最优解的比例为((9y - 10)/9y* 1 0 0) %，对于大的 y，这个值趋近于 1 0 0 %。但是可以建立贪婪启发式方法来提供解，使解的结果与最优解的值之差在最优值的 x% (x<100) 之内。首先将最多 k 件物品放入背包，如果这 k 件物品重量大于 c，则放弃它。否则，剩余的容量用来考虑将剩余物品按 pi /wi 递减的顺序装入。通过考虑由启发法产生的解法中最多为 k 件物品的所有可能的子集来得到最优解。

例 13-9 考虑 n =4, w=[2,4,6,7], p=[6,10,12,13], c = 11。当 k= 0 时，背包按物品价值密度非递减顺序装入，首先将物品 1 放入背包，然后是物品 2，背包剩下的容量为 5 个单元，剩下的物品没有一个合适的，因此解为 x = [1 , 1 , 0 , 0]。此解获得的价值为 1 6。

现在考虑 k = 1 时的贪婪启发法。最初的子集为{ 1 } , { 2 } , { 3 } , { 4 }。子集{ 1 } , { 2 }产生与 k= 0 时相同的结果，考虑子集{ 3 }，置 x3 为 1。此时还剩 5 个单位的容量，按价值密度非递增顺序来考虑如何利用这 5 个单位的容量。首先考虑物品 1，它适合，因此取 x1 为 1，这时仅剩下 3 个单位容量了，且剩余物品没有能够加入背包中的物品。通过子集{ 3 }开始求解得结果为 x = [1 , 0 , 1 , 0]，获得的价值为 1 8。若从子集{ 4 }开始，产生的解为 x = [1 , 0 , 0 , 1]，获得的价值为 1 9。考虑子集大小为 0 和 1 时获得的最优解为[1 , 0 , 0 , 1]。这个解是通过 k= 1 的贪婪启发式算法得到的。

若 k= 2，除了考虑 k< 2 的子集，还必需考虑子集{ 1 , 2 } , { 1 , 3 } , { 1 , 4 } , { 2 , 3 } , { 2 , 4 }和{ 3 , 4 }。首先从最后一个子集开始，它是不可行的，故将其抛弃，剩下的子集经求解分别得到如下结果：[1 , 1 , 0 , 0] , [1 , 0 , 1 , 0] , [1 , 0 , 0 , 1] , [0 , 1 , 1 , 0]和[0 , 1 , 0 , 1]，这些结果中最后一个价值为 2 3，它的值比 k= 0 和 k= 1 时获得的解要高，这个答案即为启发式方法产生的结果。

这种修改后的贪婪启发方法称为 k 阶优化方法（k - o p t i m a l）。也就是，若从答案中取出 k 件物品，并放入另外 k 件，获得的结果不会比原来的好，而且用这种方式获得的值在最优值的(1 0 0 / (k + 1)) %以内。当 k= 1 时，保证最终结果在最佳值的 5 0 %以内；当 k= 2 时，则在 3 3 . 3 3 %以内等等，这种启发式方法的执行时间随 k 的增大而增加，需要测试的子集数目为 O (nk)，每一个子集所需时间为 O (n)，因此当 k >0 时总的时间开销为 O (nk+1)。实际观察到的性能要好得多。

4.4.3 拓扑排序

一个复杂的工程通常可以分解成一组小任务的集合，完成这些小任务意味着整个工程的完成。例如，汽车装配工程可分解为以下任务：将底盘放上装配线，装轴，将座位装在底盘上，上漆，装刹车，装门等等。任务之间具有先后关系，例如在装轴之前必须先将底板放上装配线。任务的先后顺序可用有向图表示—称为顶点活动（ Activity On Vertex, AOV）网络。有向图的顶点代表任务，有向边(i, j) 表示先后关系：任务 j 开始前任务 i 必须完成。图 1 - 4 显示了六个任务的工程，边（ 1 , 4）表示任务 1 在任务 4 开始前完成，同样边（ 4 , 6）表示任务 4 在任务 6 开始前完成，边（1 , 4）与（4 , 6）合起来可知任务 1 在任务 6 开始前完成，即前后关系是传递的。由此可知，边（1 , 4）是多余的，因为边（1 , 3）和（3 , 4）已暗示了这种关系。

在很多条件下，任务的执行是连续进行的，例如汽车装配问题或平时购买的标有“需要装配”的消费品（自行车、小孩的秋千装置，割草机等等）。我们可根据所建议的顺序来装配。在由任务建立的有向图中，边（ i, j）表示在装配序列中任务 i 在任务 j 的前面，具有这种性质的序列称为拓扑序列（topological orders 或 topological sequences）。根据任务的有向图建立拓扑序列的过程称为拓扑排序（topological sorting）。图 1 - 4 的任务有向图有多种拓扑序列，其中的三种为 1 2 3 4 5 6，1 3 2 4 5 6 和 2 1 5 3 4 6，序列 1 4 2 3 5 6 就不是拓扑序列，因为在这个序列中任务 4 在 3 的前面，而任务有向图中的边为（ 3 , 4），这种序列与边（ 3 , 4）及其他边所指示的序列相矛盾。可用贪婪算法来建立拓扑序列。算法按从左到右的步骤构造拓扑序列，每一步在排好的序列中加入一个顶点。利用如下婪准则来选择顶点：从剩下的顶点中，选择顶点 w，使得 w 不存在这样的入边（v,w），其中

顶点 v 不在已排好的序列结构中出现。注意到如果加入的顶点 w 违背了这个准则（即有向图中存在边（ v,w）且 v 不在已构造的序列中），则无法完成拓扑排序，因为顶点 v 必须跟随在顶点 w 之后。贪婪算法的伪代码如图 1 3 - 5 所示。while 循环的每次迭代代表贪婪算法的一个步骤。

现在用贪婪算法来求解图 1 - 4 的有向图。首先从一个空序列 V 开始，第一步选择 V 的第一个顶点。此时，在有向图中有两个候选顶点 1 和 2，若选择顶点 2，则序列 V = 2，第一步完成。第二步选择 V 的第二个顶点，根据贪婪准则可知候选顶点为 1 和 5，若选择 5，则 V = 2 5。下一步，顶点 1 是唯一的候选，因此 V = 2 5 1。第四步，顶点 3 是唯一的候选，因此把顶点 3 加入 V

得到 V = 2 5 1 3。在最后两步分别加入顶点 4 和 6 ，得 V = 2 5 1 3 4 6。

贪婪算法的正确性

为保证贪婪算法算的正确性，需要证明：

1) 当算法失败时，有向图没有拓扑序列；

2) 若算法没有失败，V 即是拓扑序列。

3) 即是用贪婪准则来选取下一个顶点的直接结果

1) 的证明见定理 1 3 - 2，它证明了若算法失败，则有向图中有环路。若有向图中包含环 qj qj + 1.qk qj，则它没有拓扑序列，因为该序列暗示了 qj 一定要在 qj 开始前完成。

定理 1-2 如果图 1 3 - 5 算法失败，则有向图含有环路。

证明注意到当失败时| V |<n, 且没有候选顶点能加入 V 中，因此至少有一个顶点 q1 不在 V 中，有向图中必包含边（ q2 , q1）且 q2 不在 V 中，否则， q1 是可加入 V 的候选顶点。同样，必有边（q3 , q2）使得 q3 不在 V 中，若 q3 = q1 则 q1 q2 q3 是有向图中的一个环路；若 q3 ≠q1，则必存在 q4 使（q4 , q3）是有向图的边且 q4 不在 V 中，否则，q3 便是 V 的一个候选顶点。若 q4 为 q1 , q2 , q3 中的任何一个，则又可知有向图含有环，因为有向图具有有限个顶点数 n，继续利用上述方法，最后总能找到一个环路。

数据结构的选择

为将图 1 - 5 用 C + +代码来实现，必须考虑序列 V 的描述方法，以及如何找出可加入 V 的候选顶点。一种高效的实现方法是将序列 V 用一维数组 v 来描述的，用一个栈来保存可加入 V 的候选顶点。另有一个一维数组 I n D e g r e e，I n D e g re e[j]表示与顶点 j 相连的节点 i 的数目，其中顶点 i 不是 V 中的成员，它们之间的有向图的边表示为（ i, j）。当 I n D e g r e e[j]变为 0 时表示 j 成为一个候选节点。序列 V 初始时为空。I n D e g r e e[j]为顶点 j 的入度。每次向 V 中加入一个顶点时，所有与新加入顶点邻接的顶点 j，其 I n D e g r e e[j]减 1。

对于有向图 1 - 4，开始时 I n D e g r e e [1 : 6] = [0 , 0 , 1 , 3 , 1 , 3]。由于顶点 1 和 2 的 I n D e g r e e 值为 0，因此它们是可加入 V 的候选顶点，由此，顶点 1 和 2 首先入栈。每一步，从栈中取出一个顶点将其加入 V，同时减去与其邻接的顶点的 I n D e g r e e 值。若在第一步时从栈中取出顶点 2 并将其加入 V，便得到了 v [0] = 2，和 I n D e g r e e [1 : 6] = [0 , 0 , 1 , 2 , 0 , 3]。由于 I n D e g r e e [5]刚刚变为 0，因此将顶点 5 入栈。

程序 1 3 - 2 给出了相应的 C + +代码，这个代码被定义为 N e t w o r k 的一个成员函数。而且，它对于有无加权的有向图均适用。但若用于无向图（不论其有无加权）将会得到错误的结果，因为拓扑排序是针对有向图来定义的。为解决这个问题，利用同样的模板来定义成员函数 AdjacencyGraph, AdjacencyWGraph，L i n k e d G r a p h 和 L i n k e d W G r a p h。这些函数可重载 N e t w o r k 中的函数并可输出错误信息。如果找到拓扑序列，则 Topological 函数返回 t r u e；若输入的有向图无拓扑序列则返回 f a l s e。当找到拓扑序列时，将其返回到 v [0 :n- 1]中。

Network:Topological 的复杂性

第一和第三个 f o r 循环的时间开销为(n)。若使用（耗费）邻接矩阵,则第二个 for 循环所用的时间为(n2)；若使用邻接链表,则所用时间为(n+e)。在两个嵌套的 while 循环中，外层循环需执行 n 次，每次将顶点 w 加入到 v 中，并初始化内层 while 循环。使用邻接矩阵时，内层 w h i l e 循环对于每个顶点 w 需花费(n)的时间；若利用邻接链表，则这个循环需花费 dwout 的时间，因此，内层 while 循环的时间开销为(n2)或(n+e)。所以，若利用邻接矩阵，程序 1 3 - 2 的时间复杂性为(n2)，若利用邻接链表则为(n+e)。

程序 13-2 拓扑排序

```
        bool Network::Topological(int v[])
        {// 计算有向图中顶点的拓扑次序
        // 如果找到了一个拓扑次序，则返回 t r u e，此时，在 v [ 0 : n - 1 ]中记录拓扑次序
        // 如果不存在拓扑次序，则返回 f a l s e
        int n = Ve r t i c e s ( ) ;// 计算入度
        int *InDegree = new int [n+1];
        InitializePos(); // 图遍历器数组
        for (int i = 1; i <= n; i++) // 初始化
        InDegree[i] = 0;
        for (i = 1; i <= n; i++) {// 从 i 出发的边
        int u = Begin(i);
        while (u) {
        I n D e g r e e [ u ] + + ;
        u = NextVe r t e x ( i ) ; }
}
        // 把入度为 0 的顶点压入堆栈
        LinkedStack<int> S;
        for (i = 1; i <= n; i++)
        if (!InDegree[i]) S.Add(i);
        // 产生拓扑次序
        i = 0; // 数组 v 的游标
        while (!S.IsEmpty()) {// 从堆栈中选择
        int w; // 下一个顶点
        S . D e l e t e ( w ) ;
        v[i++] = w;
        int u = Begin(w);
        while (u) {// 修改入度
        I n D e g r e e [ u ] - - ;
        if (!InDegree[u]) S.Add(u);
        u = NextVe r t e x ( w ) ; }
}
        D e a c t i v a t e P o s ( ) ;
        delete [] InDegree;
        return (i == n);
}
```

1. 贪心法的简单应用

题目 017 浇草地 来源：ACM Regionals 2002 Warm-up Contest

（时间限制：3 秒 内存限制：64M）

一个长 l 米，宽 w 米的草坪的中线上安装着 n（n<=10000）个不同的喷水器为草坪浇水。怎样开启最少的喷水器，使得每一寸草坪都被润湿？

输入数据：

输入文件包含多组数据。对于每组数据：第一行是 n,l,w，其中 n<=10000。接下来是 n 对整数，整数对中的第一个表示这个喷水器的横坐标 x，第二个数表示喷水器的作用范围 r，即与喷水器距离小于或等于 r 的地方能够被这个喷水器润湿。

输出数据：

对于每组测试数据，输出润湿每一寸草地所需的最少的喷水器的数量，如果无论如

何这块地都不能被完全润湿，则输出-1

样例输入：

8 20 2
5 3
4 1
1 2
7 2
10 2
13 3
16 2
19 4
3 10 1
3 5
9 3
6 1
3 10 1
5 3
1 1
9 1

输出样例：

6
2
1

这道题讨论的范围是一个二维空间，不妨对其进行一下转化，将其压缩到一维空间，以便进行讨论。其实，草坪的上边界或者下边界能够被润湿等价于整个草坪都可以被润湿，以下对这个结论进行说明。如果一个喷水器不能够喷到草坪的上下边界，则这个喷水器无效。对于一个有效的喷水器，不妨把其能 i 喷到的区域形成的圆在某条水平线上截得区间称为喷水器 i 在这条直线上的作用范围。对于喷水器 i 在草坪上下边界上的作用范围[ai,bi]和它在草坪上任意水平线上的作

用范围[ai’,bi’]总有 ai>ai’且 bi<bi’。

因此，如果两个喷水器在草坪上下边界上的作用范围[a1,b1]、[a2,b2] 接壤或部分重叠，即 a2<=b1，则对于草坪中任意一条水平线，均有 a2’<a2<=b1<b1’，也就是说，如果两个

喷水器在草坪上下边界上的的作用范围[a1,b1]、[a2,b2]互相接壤或部分重叠，则这两个喷水器可以将它们之间的草地喷得天衣无缝。

类似地，我们也可以说明，如果两个喷水器可以将其之间的每寸草地喷上水，则这两个喷水器在草坪上下边界的作用范围相互接壤.经过降维处理，现在我们要做的就是，找出一个最小的喷水器集合，使得这些喷水器在上下边界的总作用范围（即各个喷水器的作用范围并起来所得的结果）包含这块草地的上下边界。

先做些与处理，结合勾股定理算出每个喷水器的作用范围，舍去无效（作用范围包含于其他喷水器或浇不到草坪边界）的喷水器，然后按照各区间左端由小到大的顺序对喷水器进行排序。选择一个能喷到草坪上下边界左端点的，作用范围最大的喷水器作为起点，向后扫描，找一个“作用范围与前一个喷水器接壤”且“作用范围最大”的喷水器，然后继续扫描……如此往复，直到当前喷水器的作用范围的右端达到或超过草坪上下边界的右端。在这个算法中，每次取喷水器的时候，除了要顾及“作用范围与前一个喷水器接壤”之外，也要保证“作用范围最大”，这是因为如果每次都使“当前可被浇到的区间”尽量向后伸展，这样以来，更靠后的喷水器就有了“被取用”的机会，换句或说，如果第 k 次不取“作用范围最大”的喷水器，就有可能使得第 m（m>k）次取喷水器时，可选择的最靠右的喷水器比原算法选择能够选择的喷水器靠左，这样一来，得到的结果便不是最优解了。

第2章　进阶算法

1　数论基础

1.1 素数

数学类的基本算法大多数属于初等数论范畴，相当大一部分与素数有直接关系，因此素数是一个很基本又很重要的内容。

我们先来看看怎么判断一个数是否素数。素数的定义为：如果一个数的正因子只有 1 和这个数本身，那么这个数就是素数。根据定义，我们立即能得到判断一个数Ｎ（大于 1 ）是否素数的简单的算法：枚举 2 到Ｎ－ 1 之间的整数，判断是否能整除Ｎ。

如果 n 很大，那么上面的程序就要运行比较长的一段时间，那么有没有更快一点的算法呢？回答是肯定的！因为如果 n 含有不为 1 和自身的因子，那么这些因子中必定有不大于 sqrt(n)的（假设 n 有因子 p，1<p<n，如果 p<=sqrt(n)，那么 p 就不大于 sqrt(n)，如果 p>sqrt(n)，那么 n/p 也是 n 的因子，而且 1<n/p<n，所以 n/p 不大于 sqrt(n)）。算法复杂度为（n^1.5）

有没有更好的算法呢，　更进一步，我们用一个 bool 数组代表筛，它的每一位对应一个整数。首先将数组所能容纳的上述数放入筛中，即将数组的全部位置成 true。从筛中找出最小的数，该数即为质数，然后将该质数的倍

数从筛中去掉，即将在数组中与它们对应的位置成 false。5 倍……等整数。反复上述过程，直至筛为空。程序就能找到指定范围内的全部质数。

下面给出程序：

```
#include <iostream>
#include <assert.h>
using namespace std;
const int MAX_N = 1000;

    bool isPrim[MAX_N];
    int main()
    {
            int i, j, n;
            cin >> n;
            assert( (n > 0) && (n < MAX_N));
            memset(isPrim, true, sizeof(isPrim));
            isPrim[0] = isPrim[1] = false;
            for (i = 2; i < n; i++) {
    if (isPrim[i]) {
    for (j = i; i*j <= n; j++) {    //注意这里的的 J 是从 I 开始的，想一想为什么？？？
    isPrim[i*j] = false;
```

```
                }
    cout << i << endl;
            }
        }
    return 0;
    }
```

该程序的效率接近 O(n);

1.2 因式分解

因式分解的算法很简单，模拟手工分解的过程，我们得到分解 n 的算法：枚举所有不大于 n 的所有素数，判断这些素数能整除 n 多少次。事实上，我们有更好的算法。先看一个显而易见的结论：如果 p 是能整除 n 的所有大于 1 的数中最小的，那么 p 是 n 的一个素因子。

1.3 公因子的数量

问题描述：已知一个正整数 N，问这个数有多少正公因子。

算法分析：最容易想到的算法是：枚举 1..N，看看有多少个数能整除 N，这种算法的复杂度为 O(N)。可以优化一下：如果 N 有小于 SQRT(N)的因子 X，那么 N 必定有大于 SQRT(N)的因子 Y 与 X 对应，而且 XY=N。所以我们只需要枚举 1..SQRT(N)的数即可，还要考虑 N 为完全平方数的特殊情况。程序：Pascal。上面这个算法的复杂度为 O(sqrt(N))。其实我们可以利用因式分解的方法来做。假设我们已经分解 N 得到 N =(p[1]^s[1])*(p[2]^s[2])...*(p[pnum]^s[pnum])，其中 p[i]为互不相同的素数，那么 N 的正因子的数量为（具体怎么推导请参考组合数学教材中的母函数一章）：(s[1]+1)*(s[2]+1)*…*(s[pnum]+1)。

1.4 最大公因式

问题描述：已知两个正整数 a 和 b，求这两个数的最大公因数 GCD(a , b)。

(GCD 是 Greatest Common Divisor 的缩写)

算法分析：不妨设 a<=b，一种十分容易想到的算法是：枚举 1 到 a 的所有整数，在能同时整除 a 和 b 的数中取最大的。这个算法的时间复杂度为 O(min(a,b))，当 min(a,b)较大的时候程序要执行比较长的时间。我们可以利用初等数论中的一个定理：

GCD(a , b) = GCD(a , b-a) = GCD(a , b-2*a) = GCD(a , b-3*a) = … = GCD(a , b mod a)

算法实现：

```
int gcd(int a, int b)
{
while (b) {
        int tmp = b;
        b = a % b;
```

```
            a = tmp;
        }
    return a;
    }
```

1.5 最小公倍数

问题描述：已知两个正整数 a 和 b，求这两个数的最小公倍数 LCM (a , b)。
(LCM 是 Least Common Multiply 的缩写)
算法分析：直接利用公式：LCM (a , b) * GCD(a , b) = a * b 即可

```
int lcm(int a, int b)
{
return a/gcd(a, b)*b;
}
```

1.6 进制转换

我们平常计算都是用十进制数的，但是有的时候我们需要用到 2 进制数、16 进制数等。一个 k 进制的数可以表示为：(As-1 As-2… A0)k = As-1 K^(s-1) + As-2 K^(s-2) + … + A0 K^(0) ，记为<1>式，其中 0<=Ai<K（I=0,1,2..s-1）。对于一个已知的正整数 n，如何得到 n 的 K 进制表示呢？换句话说，我们就是要求出 As-1 As-2… A0 来。具体的求解顺序是：先求出 A0，然后是 A1 ……，最后得到 An-1。将<1>式等号两边同时取模 k 得：n mod K = a0。得到 A0 以后，(n-A0) div K = As-1 K^(s-2) + As-2 K^(s-3) + … + A1 K^(0)，用与求 A0 同样的方法可以得到 A1，然后是 A2……。Pascal 程序。运行这个程序，输入：155 16 就可以得到结果 9B （16 进制的 9B = 9*16+11=155）

怎么进行任意进制间的数的转换呢？已知一个数正整数 n 的 P 进制表示（As-1As-2…A0），求 n 的 Q 进制表示（Bl-1Bl-2…B0）。一种简单的方法是：根据 P 进制表示求出十进制的 n，然后再将 n 转化为 Q 进制表示即可。

前面考虑的都是整数的问题，我们现在来看看怎么处理实有理数。由于实数跟整数的区别仅仅在于小数部分，所以现在只考虑实数 r，r 满足 0 <= r <1 的情况。定义 r 的 K 进制表示为：r =（0.A1 A2 A3 … As）K = A1 K^(-1) + A2 K^(-2) + A3 K^(-3) …. As K^(-s)。求解顺序为：A1、A2……As。解法：r K = A1 + A2 K^(-1) + A3 K^(-2) …. As K^(-(s-1)) = A1 + B。考察一下 B 的范围 0<= B <= (K-1)K^(-1) +(K-1)K^(-2)… (K-1)K^(-(s-1)) = 1-K^(-s) < 1，也就是说 B 是 r K 的小数部分，A1 是整数部分，于是 A1 = [r K]，[x]表示不大于 x 的最大整数。由于 B = r-A1，所以用同样的方法分解 B 就可以得到 A2、A3……As。

1.7 模取幂运算

事实上，m^e mod n 可以直接计算，没有必要先算 m^e，那样肯定不行。m^e mod n 叫做模取幂运算，根据简单的数论知识，很容易设计一个分治算法。具体如下:

设<b[k], b[k-1],...,b[1],b[0]>是整数 b 的二进制表示（即 b 的二进制有 k+1 位，b[k]是最高位），下列过程随着 c 的值从 0 到 b 成倍增加，最终计算出 a^c mod n

```
Modular-Exponentiation(a, b, n)
1.  c ← 0
2.  d ← 1
3.  设<b[k],b[k-1],..b[0]>是 b 的二进制表示
4.  for i←k downto 0
5.     do c ← 2c
6.        d ← (d*d) mod n
7.        if b[i] = 1
8.           then c ← c + 1
9.                d ← (d*a) mod n
10. return d
```

首先说明一下，上述伪代码中用缩紧表示语句之间的层次关系，例如第 5~9 行都是 for 循环体内的语句，第 8~9 行都是 then 里面的语句。这是我比较喜欢的一种表示方法 ；）上述伪代码依次计算出的每个幂或者是前一个幂的两倍，或者比前一个幂大 1。过程依次从右到左逐个读入 b 的二进制表示已控制执行哪一种操作。循环中的每次迭代都用到了下面的

两个恒等式中的一个：

a^(2c) mod n = (a^c mod n)^2

a^(2c+1) mod n = a * (a^c mod n)^2

用哪一个恒等式取决于 b[i]=0 还是 1。由于平方在每次迭代中起着关键作用，所以这种方法叫做“反复平方法(repeated squaring)”。在读入 b[i]位并进行相应处理后，c 的值与 b 的二进制表示<b[k],b[k-1],..b[0]>的前缀的值相同。事实上，算法中并不真正需要变量 c，只是为了说明算法才设置了变量 c：当 c 成倍增加时，算法保持条件 d = a^c mod n 不变，直至 c=b。

如果输入 a,b,n 是 k 位的数，则算法总共需要执行的算术运算次数为 O(k)，总共需要执行的位操作次数为 O(k^3)。

1.8 欧拉 PHI 函数

欧拉函数的定义：小于 N，并与 N 互质的数的总个数（包括 1）。如 F(9)=6。

公式（1）F(n)=n*(1-1/p1)...*(1-1/pk)。其中，pk 是 n 的素因子。

公式（2）如果 m 和 n 互质，且 N=mn，那么 F(N)=F(mn)=F(m)*F(n)。

证明：

设 \phi(n) 表示小于 n 且与 n 互素的正整数数目。

设 n 可以分解为素数乘积形式 n = p_1^a_1 * p_2^a_2 * ... * p_k^a_k 且诸 a_i > 0

设 A_i 表示 1..n 中 p_i 的倍数的集合，i = 1,2,..k 则

|A_i| = n/p_i, i = 1, 2, ..., k

|A_i \cap A_j| = n / (p_i * p_j), i, j = 1, 2, ..., k

...

根据容斥原理，

phi(n) = | (~A_1) \cap (~A_2) \cap ... \cap (~A_k)|

= n - |A_1 \cup A_2 \cup ... \cup A_k|

= n - \sum_{i=1}^k (n/p_i) + \sum_{i,j = 1}^k (n / (p_ip_j)) - ...

= n*(1 - 1/p_1) * (1 - 1/p_2) * ...* (1 - 1/p_k)

注意这其中包含了 1。

这个函数就是著名的欧拉函数

1.9 牛顿迭代

牛顿法是方程求根的一个有力方法，常常能快速求出其他方法求不出或者难以求出的解。

假定有一个函数 y=f（x），方程 f(x)=0 在 x = r 处有一个根，对于此根，我们先估计一个初始值 Xo（可以是猜测的）。我们现在来得到一个更好的估计值 X1。为此　趎=Xo 处作该曲线的切线，并将其延长与 x 轴相交。切线与 x 轴的交点通常很接近 r ，我们用它作为下一个估计值 X1，求出 X1 后，用 X1 代替 Xo。重复上述过程，在 x=X1 处作曲线的另一条切线，并将其延长至与 x 轴相交，用切线的 x 轴截距作为下一个近似值 X2……这样继续下去，所得出的这个 x 轴截距的序列通常迅速接近根 r。

现在再让我们从代数角度看上述过程，我们知道，在初始值 Xo 处，切线的斜率是 f’(x)，切线方程为

y - f(xo) = f’(xo)(x - xo)

在此切线与 x 轴相交处，有 y=0 ,x=x1，因而有

0 - f(xo) = f’(xo)(x1 - xo)

只要 f’(xo)不为 0，可解出 x1，得

$$x1 = x0 - \frac{f(xo)}{f'(xo)}$$

重复该过程，可得下一近似值为

$$x2 = x1 - \frac{f(x1)}{f'(x1)}$$

总结 n = 0，1，2，……的情形得出下述结果

牛顿法：
只要 f’(xn) ≠ 0，则有 $X(n+1) = Xn - \frac{f(Xn)}{f'(Xn)}$

注意：牛顿法也有不成功的时候，若 f （ x ）无根，则，序列不收敛。另外，一些函数图像可能形成随即序列，这就需要其他的辅助条件。

附注：f’(x)表示函数 f(x)的导函数，f’(xo)则表示函数 f(x)在 x = xo 处的导数

【算法】迭代法求方程的根

```
{    x0=初始近似根;
     do {
        x1=x0;
        x0=g(x1);       /*按特定的方程计算新的近似根*/
```

```
} while ( fabs(x0-x1)>Epsilon);
printf(“方程的近似根是%f\n”，x0);
}
```

迭代算法也常用于求方程组的根，令

X=（x0，x1，…，xn-1）

设方程组为：

xi=gi(X)　　(I=0，1，…，n-1)

则求方程组根的迭代算法可描述如下：

【算法】迭代法求方程组的根

```
    {       for (i=0;i<n;i++)
              x[ i]=初始近似根;
          do {
              for (i=0;i<n;i++)
y[ i]=x[ i];
for (i=0;i<n;i++)
x[ i]=gi(X[ i]);
for (delta=0.0,i=0;i<n;i++)
if (fabs(y[ i]-x[ i])>delta)          delta=fabs(y[ i]-x[ i]);
} while (delta>Epsilon);
for (i=0;i<n;i++)
printf(“变量 x[%d]的近似根是 %f”，I，x[ i]);
printf(“\n”);
}
```

2 图论算法

图论是一个应用十分广泛而又极其有趣的数学分支。物理、化学、生物、科学管理，特别是计算机等各个领域都有图论的足迹。信息学竞赛中图论内容相当丰富，出现的题目越来越频繁，因此，有必要对图论方面的一些常见的经典算法以及隐含在题中的图沦模型的构造进行系统的研究，以便很好地掌握它的精髓。

2.1 最小生成树

生成树的概念

设图 G＝(V，E)是一个连通图，当从图中任一顶点出发遍历图 G 时，将边集 E(G)分成两个集合 A(G)和 B(G)。其中 A(G)是遍历图时所经过的边的集合，B(G)是遍历图时未经过的边的集合。显然，G1＝(V，A)是图 G 的子图，则称子图 G1 是连通图 G 的生成树。图的生成树不是惟一的。如对图 1(a)，当按深度和广度优先搜索法进行遍历就可以得到图 1 中(b)和(c)的两棵不同的生成树，并分别称之为深度优先生成树和广度优先生成树

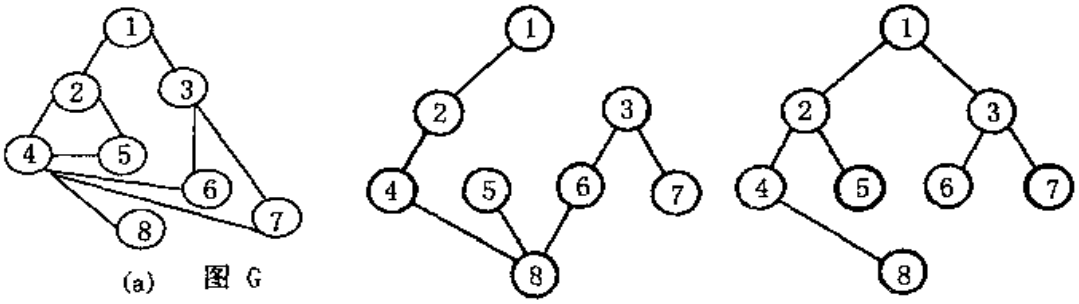


图 1 生成树

对于有 n 个顶点的连通图，至少有 n-1 条边，而生成树中恰好有 n-1 条边，所以连通图的生成树是该图的极小连通子图。若图 G 的生成树中任意加一条边属于边集 B(G)中的边，则必然形成回路。

求解生成树在许多领域有实际意义。例如，对于供电线路或煤气管道的铺设问题，即假设要把 n 个城市联成一个供电或煤气管道网络，则需要铺设 n－1 条线路。任意两城市间可铺设一条线路，n 个城市间最多可能铺设 n(n－1)/2 条线路，各条线路的造价一般是不同的。一个很实际的问题就是如何在这些可能的线路中选择 n-1 条使该网络的建造费用最少，这就是下面要讨论的最小生成树问题。

2.2 网的最小生成树

在前面我们已经给出图的生成树的概念。这里来讨论生成树的应用。

假设，要在 n 个居民点之间敷设煤气管道。由于，在每一个居民点与其余 n－1 个居民点之间都可能敷设煤气管道。因此，在 n 个居民点之间，最多可能敷设 n(n-1)/2 条煤气管道。然而，连通 n 个居民点之间的管道网络，最少需要 n-1 条管道。也就是说，只需要 n-1 条管道线路就可以把 n 个居民点间的煤气管道连通。另外，还需进一步考虑敷设每一

条管道要付出的经济代价。这就提出了一个优选问题。即如何在 n(n-1)/2 条可能的线路中优选 n-1 条线路，构成一个煤气管道网络，从而既能连通 n 个居民点，又能使总的花费代价最小。

解决上述问题的数学模型就是求图中网的最小生成树问题。把居民点看作图的顶点，把居民点之间的煤气管道看作边，而把敷设各条线路的代价当作权赋给相应的边。这样，便构成一个带权的图，即网。对于一个有 n 个顶点的网可以生成许多互不相同的生成树，每一棵生成树都是一个可行的敷设方案。现在的问题是应寻求一棵所有边的权总和为最小的生成树。

如何构造这种网的最小生成树呢?下面给出这样一种解法：

(1) 已知一个网，将网中的边按其权值由小到大的次序顺序选取。

(2) 若选某边后不形成回路，则将其保留作为树的一条边；若选某边后形成回路，则将其舍弃，以后也不再考虑。

(3) 如此依次进行，直到选够(n-1)条边即得到最小生成树。

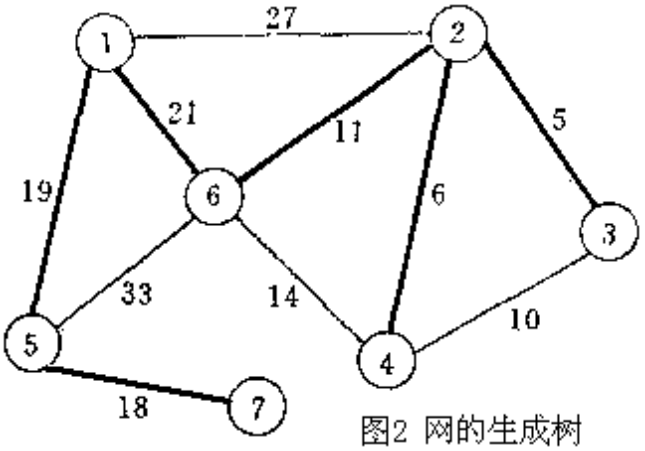


图2 网的生成树

现以图 2 为例说明此算法。设此图是用边集数组 EV 表示的，且数组中各边是按权值由小到大的次序排列，如下表所示。

k	1	2	3	4	5	6	7	8	9	10
EV[k].p1	2	2	4	2	6	5	1	1	1	5
EV[k].p2	3	4	3	6	4	7	5	6	2	6
COST[EV[k].p1,EV[k].p2]	5	6	10	11	14	18	19	21	27	33

按权值由小到大选取各边就是在数组中按下标 k 由 1 到 en(图中边数)的次序选取。选前 2 条边(2，3)，(2，4)时均无问题，保留作为树的边；到第 3 条边(4，3)时将与已保留的边形成回路，将其舍去；同样继续做：保留(2， 6)；舍去(6，4)；保留(5，7)，(1，5)，(1，6)，此时，保留的边数已够(n-1)=6 条边，此时必定将 7 个顶点全部互相连通了，后面剩下的边(1，2)，(5，6)就不必再考虑了。最后得到的最小生成树如图 2a 中深色边所示，其各边权值总和等于 80。由离散数学中的图论可以证明，这就是最小生成树了，其权值最小。当图中有权值相等的边时，其最小生成树可能有不同的选取方案。

实现此算法的关键是，在选取某条边时应判断是否与已保留的边形成回路。

这可用将各顶点划分为集合的办法解决：假设数组 tag(1..en)作为顶点集合划分的标志初值为 0。在算法的执行过程中，当所选顶点 u，v 是连通的，则将相应位置的 tag[u]，tag[v]置以相同的数字，而不连通的点在初期分属不同的集合，置不同的数字；一旦两个不同的

连通分支连通了，则修改 tag 的值，将新的连通分支改为相同的数字。我们以图 2 为例。首先选(2，3)(2，4)边，由于是连通的，并且不出现回路。tag[2]:＝1，tag[3]:=tag[4]:=1 是同一个集合 A；选(6，2)边与 A 集合连通；tag[6]:=1；再选(5，7)与集合 A 不连通，tag[5]:＝ tag[7]:＝2 构成另一集合 B；选(1，5)边与集合 B 连通，tag [1]:=2；此时，集合 A＝ {2，3，4，6}；集合 B＝{5，7，1}；当选(1，6)边时，(1，6)与集合 A、集合 B 都连通，并且两个顶点分别属于两个不同的集合 A、B，这使得集合 A 与集合 B 通过边(1，6)连通。修改集合 B 中 tag 的值，置为 1，即将集合 B 并入集合 A。边为 n-1 条，这就是一棵最小生成树。

根据集合标志数组 tag 的变化过程，很容易判断，选择一条新的边是否构成回路。当新选边的两个顶点 u、v，若 tag[u]和 tag[v]相同并且均不等于 0 时，即 u，v 已在生成树集合中被保留过，加入 u，v 后即形成回路，不能选。而当 tag [u]<>tag[v]或者 tag[u]＝tag[v]＝0 时，可以选并且不形成回路，说明 u，v 中至少有一个顶点未被选过或者被选过的 u、v 分别属于两个不同的集合，此时选择 u，v 可以将含 u 的集合与含 v 的集合连通，修改 tag 数组。如此下去，到所有顶点均已属于一个集合时，此最小生成树就完全构成了。

网的最小生成树算法描述如下：

假设算法中用到的数据结构是经过处理的。

COST(1..n，1..n)是带权数组存放网中顶点之间的权。EV(1..n*(n-1/2))按权从小到大存放排序后的顶点对，即 EV[K].P1 存放一个顶点，边的另一顶点存放在 EV[K].P2 之中。

tag(1..n)：顶点集合划分标志的数组。

Enumb：当前生成树的边数。

SM：当前权累计和。

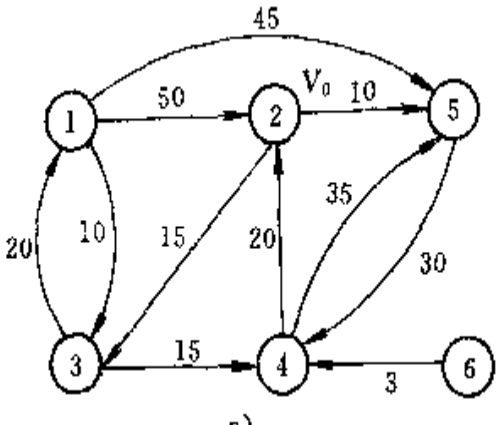
```
PROC minspanningtree(VAR cost；VAR ev)；
 Var tag；
 BEGIN
   CALL INITIAL(tag);
   Enumb：＝0；SM:=0; {诸参量初始化}
   k:=1;    {边数累计}
   WHILE (Enumb<=n-1) AND (k<＝n) DO
    Begin U:=EV[k].P1；V:=EV[k].P2；     {选一对顶点(U，V)}
      CALL FIND(U，T)；   {找到含顶点 U 的集合 T}
      CALL FIND(V，W)；   {找到含顶点 V 的集合 W}
      IF (T<>W) THEN
      Begin
        write(u，v)；Enumb:=Enumb+1；     {最小生成树增加一条边}
        SM:=SM+COST[u，v]；
        MERGE(T，W)；{选 u，v 不会形成环，合并 T，W 集合，并修改 tag}
      end
      K:=K+1;  {找下一条边}
    end
   IF Enumb<n-1 THEN write('There is not a minspanning tree')
   ELSE write(SM)
 END；
```

由算法可知图 2 的最小生成树的结果是(2，3)，(2，4)，(2，6)，(5，7)，(1，5)，(1，6)。

2.3 最短路径

在一个赋权有向图上寻找最短路径问题也是图应用的一个重要课题。

假定图 3 中的有向图 G=(V，E)是一个航空图，V 的每一顶点表示—个城市，正中的每条弧 v—>w 表示从城市 v 到城市 W 的航线，弧 V—>w 上的标号代表从 V 城飞到 w 城所需要的时间。要寻找由该航空图上一给定城市到另一城市所需要的最短飞行时间。可以用求解这个有向图的单源最短路径算法来完成。



a)

	1	2	3	4	5	6
1	0	50	10	∞	45	∞
2	∞	0	15	∞	10	∞
3	20	∞	0	15	∞	∞
4	∞	20	∞	0	35	∞
5	∞	∞	∞	30	0	∞
6	∞	∞	∞	3	∞	0

b)

图3 图单源最短路径示例

下面，我们讨论求解单源最短路径问题的贪心算法，也称 Dijkstra 算法。

设有向图 G=(V，E)，其中，V={1，2，…，n)．cost 是表示 G 的邻接矩阵，cost[i，j]表示有向边(i，j)的权。若不存在有向边(i，j)，则 cost[i，j]的权为无限大(oo)。令 S 是一个集合，其中的每个元素表示一个顶点，从源点到这些顶点的最短距离已经求出。

令顶点 V0 为源点，集合 S 的初态只包含顶点 V0，即 S={V0}。数组 dist 记录从源点到其他各顶点当前的最短距离，其初值为 dist[i]:=cost[v0，i]，(i=2，…，n)。

从 S 之外的顶点集合 V-S 中选出一个顶点 W，使 dist[W]的值最小。于是，从源点到达 W 只通过 S 中的顶点，我们把 W 加入集合 S。

调整 dist 中记录的从源点到 V-S 中每个顶点 V 的距离：从原来的 dist[v]和 dist[w]+cost[w，v]中选择较小的值作为新的 dist[v]。

重复上述过程(2)和（3），直到 S 中包含 V 的全部顶点。

最终数组 dist 记录了从源点到 V 中其余各顶点的最短路径。

对图 3 所示的加权有向图应用 Dijkstra 算法，从源点 V2 出发到达各顶点的最短路径如下表所示。

最短路径

源点	中间顶点	终止顶点	长度
2		5	10
		3	15
	3	4	30
	3	1	35
		6	oo

对图 3 的执行过程：初始时，S＝{2}，dist[1]＝oo，dist[3]＝15，dist[4]＝oo，dist[5]＝10，dist[6]＝oo，第一遍处理时，W＝2 使 dist[5]最小、于是把 5 加入 S。然后，调整 dist 中从源点到其余各顶点的距离：dist[3]＝15，为次小，将 3 加入 S。dist[4]＝cost[2，3]+cost[3，4]=15+15＝30，经中间点 3。S＝{2，5，3，4}，同理，dist[1]＝cost[2，3]+cost[3，1]＝35，S＝{2，5，3，4，1}，由于 2 没有一条到 6 的路径，所以 dist[6]=oo。

由此我们给出最短路径算法如下

```
   PROC shortpath(VAR cost；VAR dist；VAR path；VAR S，V0)；
BEGIN
  FOR W:=1 TO n DO
  Begin dist[W]:=cost[V0，W];{最短路径初始化值}
    IF cost[V0，W]<max
    THEN path[W]:=V0；{path 记载当前最短路径}
  End;
  S:=[V0];Vnum:=1；{到达点集合 S 和到达点 S 个数初值}
  WHILE (Vnum<n-1) DO {最后一点已无选择余地}
  Begin Wm:=max；u:=V0;
    FOR W:=1 TO n DO
      IF(NOT W IN S) AND (dist[W]<Wm)
      THEN Begin U:=W；Wm:=dist[w] End;
      {找最小 dist[w]}
    S:=S+[U];Vnum:=Vnum+1; {U 为找到最短路径的终点}
    FOR W:=1 TO n DO
    IF (NOT W IN S) AND (dist[U]+cost[U，W]<dist[W])
    THEN Begin dist[W]:=dist[U]+cost[U，W]; {调整非 S 集各点最短路径值}
      path[W]:=U； {调整非 S 集各点最短路径}
    End;
    Vnum:=Vnum+1
  End;
END；
PROC PRINTPATH(VAR dist VAR path；VAR S；V0)
BEGIN
  FOR i:=1 TO n DO
  IF(i IN S)
  THEN Begin k:=i;
     WHILE (k<>V0) DO
      Begin write(k)；k:=path[k] End; {通过找前趋点，反向输出最短路径}
     write(k)；writeln(dist[i])
    End;
  ELSE Begin write(i，V0)；writeln('max') End;
END；
```

容易看出，算法 short path 的时间复杂度为 O(n2)，空间复杂度为 O(n)。

2.4 拓扑排序

本节说明了如何用深度优先搜索，对一个有向无回路图进行拓扑排序。有向无回路图又称为 dag。对这种有向无回路图的拓扑排序的结果为该图所有顶点的一个线性序列，满足如果 G 包含(u,v)，则在序列中 u 出现在 v 之前（如果图是有回路的就不可能存在这样的线性序列)。一个图的拓扑排序可以看成是图的所有顶点沿水平线排成的一个序列，使得所有的有向边均从左指向右。因此，拓扑排序不同于通常意义上对于线性表的排序。

有向无回路图经常用于说明事件发生的先后次序，图 1 给出一个实例说明早晨穿衣的过程。必须先穿某一衣物才能再穿其他衣物(如先穿袜子后穿鞋)，也有一些衣物可以按任意次序穿戴(如袜子和短裤)。图 1(a)所示的图中的有向边(u,v)表明衣服 u 必须先于衣服 v 穿戴。因此该图的拓扑排序给出了一个穿衣的顺序。每个顶点旁标的是发现时刻与完成时刻。图 1(b)说明对该图进行拓扑排序后将沿水平线方向形成一个顶点序列，使得图中所有有向边均从左指向右。

下列简单算法可以对一个有向无回路图进行拓扑排序。

```
procedure Topological_Sort(G);
begin
  1.调用 DFS(G)计算每个顶点的完成时间 f[v];
  2.当每个顶点完成后，把它插入链表前端;
  3.返回由顶点组成的链表;
end;
```

图 1(b)说明经拓扑排序的结点以与其完成时刻相反的顺序出现。因为深度优先搜索的运行时间为 θ(V+E)，每一个 v 中结点插入链表需占用的时间为 θ(1)，因此进行拓扑排序的运行时间 θ(V+E)。

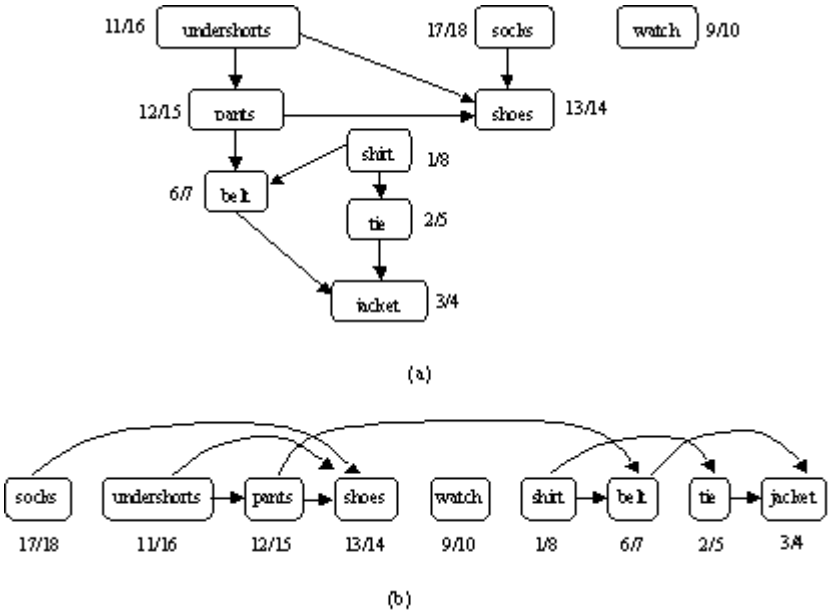


图 1 早晨穿衣的过程

为了证明算法的正确性，我们运用了下面有关有向无回路图的重要引理。

引理 1

有向图 G 无回路当且仅当对 G 进行深度优先搜索没有得到反向边。

证明：

→：假设有一条反向边(u,v)，那么在深度优先森林中结点 v 必为结点 u 的祖先，因此 G 中从 v 到 u 必存在一通路，这一通路和边(u,v)构成一个回路。

←：假设 G 中包含一回路 C，我们证明对 G 的深度优先搜索将产生一条反向边。设 v 是回路 C 中第一个被发现的结点且边(u,v)是 C 中的优先边，在时刻 d[v]从 v 到 u 存在一条由白色结点组成的通路，根据白色路径定理可知在深度优先森林中结点 u 必是结点 v 的后裔，因而(u,v)是一条反向边。(证毕)

定理 1

Topological_Sort(G)算法可产生有向无回路图 G 的拓扑排序。

证明:

假设对一已知有问无回路图 G=(V,E)运行过程 DFS 以确定其结点的完成时刻。那么只要证明对任一对不同结点 u,v∈V，若 G 中存在一条从 u 到 v 的有向边，则 f[v]<f[u]即可。考虑过程 DFS(G)所探寻的任何边(u,v)，当探寻到该边时，结点 v 不可能为灰色，否则 v 将成为 u 的祖先，(u,v)将是一条反向边，和引理 1 矛盾。因此，v 必定是白色或黑色结点。若 v 是白色，它就成为 u 的后裔，因此 f[v]<f[u]。若 v 是黑色，同样 f[v]<f[u]。这样一来对于图中任意边(u,v)，都有 f[v]<f[u]，从而定理得证。（证毕）

另一种拓扑排序的算法基于以下思想：首先选择一个无前驱的顶点（即入度为 0 的顶点，图中至少应有一个这样的顶点，否则肯定存在回路），然后从图中移去该顶点以及由他发出的所有有向边，如果图中还存在无前驱的顶点，则重复上述操作，直到操作无法进行。如果图不为空，说明图中存在回路，无法进行拓扑排序；否则移出的顶点的顺序就是对该图的一个拓扑排序。

下面是该算法的具体实现：

```
procedure Topological_Sort_II(G);
begin
1   for 每个顶点 u∈V[G] do d[u]←0;   //初始化 d[u],d[u]用来记录顶点 u 的入度
2   for 每个顶点 u∈V[G] do
3     for 每个顶点 v∈Adj[u] do d[v]←d[v]+1;   //统计每个顶点的入度
4   CreateStack(s);   //建立一个堆栈 s
5   for 每个顶点 u∈V[G] do
6     if d[u]=0 then push(u,s);   //将度为 0 的顶点压入堆栈
7   count←0;
8   while (not Empty(s)) do
      begin
9       u←top(s);   //取出栈顶元素
10      pop(s);       //弹出一个栈顶元素
11      count←count+1;
12      R[count]←u;    //线性表 R 用来记录拓扑排序的结果
13      for 每个顶点 v∈Adj[u] do //对于每个和 u 相邻的节点 v
       begin
14              d[v]←d[v]-1;
15              if d[v]=0 then push(v,s);   //如果出现入度为 0 的顶点将其压入栈
        end;
      end;
16 if count<>G.size then writeln('Error! The graph has cycle.')
17                      else 按次序输出 R;
```

end;

上面的算法中利用 d[u]来记录顶点 u 的入度，第 2-3 行用来统计所有顶点的入度，第 5-6 行将入度为 0 的顶点压入堆栈，第 8-15 行不断地从栈顶取出顶点，将该顶点输出到拓扑序列中，并将所有与该顶点相邻的顶点的入度减 1，如果某个顶点的入度减至 0，则压入堆栈，重复该过程直到堆栈空了为止。显而易见该算法的复杂度为 O(VE)，因为第 2-3 行的复杂性就是 O(VE)，后面 8-15 行的复杂性也是 O(VE)。这个算法虽然简单，但是没有前面一个算法的效率高

2.5 有向图的强连通分支

在有向图 G 中，如果任意两个不同的顶点相互可达，则称该有向图是*强连通*的。有向图 G 的极大强连通子图称为 G 的*强连通分支*。

把有向图分解为强连通分支是深度优先搜索的一个经典应用实例。下面介绍如何使用两个深度优先搜索过程来进行这种分解，很多有关有向图的算法都从分解步骤开始，这种分解可把原始的问题分成数个子问题，其中每个子子问题对应一个强连通分支。构造强连通分支之间的联系也就把子问题的解决方法联系在一起，我们可以用一种称之为*分支图*的图来表示这种构造。

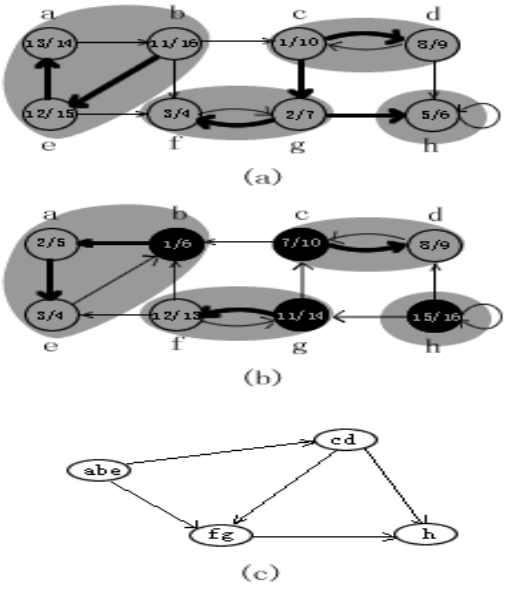
定义有向图 G=(V,E)的分支图为 $G^{SCC}=(V^{SCC},E^{SCC})$,其中 V^{SCC} 包含的每一个结点对应于 G 的每个强连通分支。如果图 G 的对应于节点 u 的强连通分支中，某个结点和对应于 v 的强连通分支中的某个结点之间存在一条有向边，则边$(u,v)\in E^{SCC}$。图 1 给出了一个实例。显而易见对于分支图有以下定理：

定理 1

有向图 G 的分支图 G^{SCC} 是一个有向无回路图。

图 1 展示了一个有向图的强连通分支的实例。(a)为有向图 G，其中的阴影部分是 G 的强连通分支，对每个顶点都标出了其发现时刻与完成时刻，黑色边为树枝；(b)G 的转置图 G^T。图中说明了算法 Strongly_Connected_Components 第 3 行计算出的深度优先树，其中黑色边是树枝。每个强连通子图对应于一棵深度优先树。图中黑色顶点 b,c,g 和 h 是强连通子图中每个顶点的祖先，这些顶点也是对 G^T 进行深度优先搜索所产生的深度优先树的树根。(c)把 G 的每个强连通子图缩减为单个顶点后所得到的有向无回路子图（即 G 的分支图）。

寻找图 G=(V,E)的强连通分支的算法中使用了 G 的转置，其定义为：$G^T=(V,E^T)$，其中 $E^T=\{(v,u)\in V\times V:(u,v)\in E\}$，即 E^T 由 G 中的边改变方向后组成。若已知图 G 的邻接表，则建立 G^T 所需时间为 O(V+E)。注意到下面的结论是很有趣的：G 和 G^T 有着完全相同的强支，即在 G 中 u 和 v 互为可达当且仅当在 G^T 中它们互为可达。图 1(b)即为图 1(a)所示



图的转置，其强连通支上涂了阴影。

图 1 强连通子图的实例

下列运行时间为 θ (V+E)的算法可得出有向图 G=(V,E)的强连通分支，该算法使用了两次深度优先搜索，一次在图 G 上进行，另一次在图 G^T 上进行。

procedure Strongly_Connected_Components(G);

begin

1.调用 DFS(G)以计算出每个结点 u 的完成时刻 f[u];

2.计算出 G^T;

3.调用 DFS(G^T)，但在 DFS 的主循环里按针 f[u]递减的顺序考虑各结点(和第一行中一样计算);

4.输出第 3 步中产生的深度优先森林中每棵树的结点，作为各自独立的强连通支。

end;

这一貌似简单的算法似乎和强连通支无关。下面我们将揭开这一设计思想的秘密并证明算法的证确性。我们将从两个有用的观察资料入手。

引理 1

如果两个结点处于同一强连通支中，那么在它们之间不存在离开该连通支的通路。

证明：

设 u 和 v 是处于同一强连通支的两个结点，根据强连通支的定义，从 u 到 v 和从 v 到 u 皆有通路，设结点 w 在某一通路 u→w→v 上，所以从 u 可达 w、又因为存在通路 v→u，所以可知从 w 通过路径 w→v→u 可达 u，因此 u 和 w 在同一强连通支中。因为 w 是任意选定的，所以引理得证。

定理 2

在任何深度优先搜索中，同一强连通支内的所有顶点均在同一棵深度优先树中。

证明：

在强连通支内的所有结点中，设 r 第一个被发现。因为 r 是第一个被发现，所以发现 r 时强连通支内的其他结点都为白色。在强连通支内从 r 到每一其他结点均有通路，因为这些通路都没有离开该强连通支(据引理 1)，所以其上所有结点均为白色。因此根据白色路径定理，在深度优先树中，强连通支内的每一个结点都是结点 r 的后裔。

在下文中，记号 d[u]和 f[u]分别指由算法 Strongly_Connected_Components 第 1 行的深度优先搜索计算出的发现和完成时刻。类似地，符号 u→v 是指 G 中而不是 G^T 中存在的

一条通路。

为了证明算法 Strongly_Connected_Components 的正确性，我们引进符号Φ(u)表示结点 u 的祖宗 w，它是根据算法第 1 行的深度优先搜索中最后完成的从 u 可达的结点（注意，与深度优先树中祖先的概念不同）。换句话说，Φ(u)是满足 u→w 且 f[w]有最大值的结点 w。

注意有可能Φ(u)=u，因为 u 对其自身当然可达，所以有

f[u]≤f[Φ[u]] (1)

我们同样可以通过以下推理证明Φ(Φ(u))=Φ(u)，对任意结点 u,v∈V，

u→v 说明 f[Φ(v)]≤f[Φ(u)] (2)

因为{w|v→w}包含于{w|u→w}，且祖宗结点是所有可达结点中具有最大完成时刻的结点。又因为从 u 可达结点Φ(u)，所以(2)式表明订 f[Φ(Φ(u))]≤f[Φ(u)]，用Φ(u)代入(1)式中的 u 同样有 f[Φ(u)]≤f[Φ(Φ(u))]，这样 f[Φ(Φ(u))]=f[Φ(u)]成立。所以有Φ(Φ(u))=Φ(u)，因为若两结点有同样的完成时刻，则实际上两结点为同一结点。

我们将发现每个强连通分支中都有一结点是其中每一结点的祖宗，该结点称为相应强连通分支的“代表性结点”。在对图 G 进行的深度优先搜索中，它是强连通支中最先发现且最后完成的结点。在对 G^T 的深度优先搜索中，它是深度优先树的树根，我们现在来证明这一性质。

第一个定理中称Φ(u)是 u 的祖先结点。

定理 3　已知有向图 G=(V,E)，在对 G 的深度优先搜索中，对任意结点 u∈V,其祖宗Φ(u)是 u 的祖先。

证明：

如果Φ(u)=u，定理自然成立。如果Φ(u)≠u，我们来考虑在时刻 d[u]各结点的颜色，若Φ(u)是黑色，则 f[Φ(u)]<f[u]，这与不等式(1)矛盾；若Φ(u)为灰色，则它是结点 u 的祖先，从而定理可以得证。

现在只要证明Φ(u)不是白色即可，在从 u 到Φ(u)的通路中若存在中间结点，则根据其颜色可分两种情况：

若每一中间结点均为白色，那么由白色路径定理知Φ(u)是 u 的后裔，则有 f[Φ(u)]<f[u]，这与不等式(1)相矛盾。

若有某个中间结点不是白色，设 t 是从 u 到Φ(u)的通路上最后一个非非白节点，则由于不可能有从黑色结点到白色结点的边存在，所以 t 必是灰色。这样就存在一条从 t 到Φ(u)且由白色结点组成的通路，因此根据白色路径定理可推知Φ(u)是 t 的后裔，这表明 f[t]>f[Φ(u)]成立，但从 u 到 t 有通路，这巧我们对Φ(u)的选择相矛盾。(证毕)

推论 1 在对有向图 G=(V,E)的任何深度优先搜索中，对所有 u∈V，结点 u 和Φ(u)处于同一个强连通分支内。

证明：

由对祖宗的定义有 u→Φ(u)，同时因为Φ(u)是 u 的祖先，所以又有Φ(u)→u。(证毕)

下面的定理给出了一个关于祖宗和强连通分支之间联系的更强有力的结论。

定理 4　在有向图 G=(V,E)中，两个结点 u,v∈V 处于同一强连通分支当且仅当对 G 进行深度优先搜索时两结点具有同一祖宗。

证明：

→：假设 u 和 v 处于同一强连通分支内，从 u 可达的每个结点也满足从 v 可达，反之亦然。这是由于在 u 和 v 之间存在双向通路，由祖宗的定义我们可以推知Φ(u)=Φ(v)。

←：假设Φ(u)=Φ(v)成立，根据推论 1，u 和Φ(u)在同一强连通分支内且 v 和Φ(v)也处于同一强连通分支内，因此 u 和 v 也在同一强连通支中。(证毕)

有了定理 4，算法 Strongly_Connected_Components 的结构就容易掌握了。强连通分支就是有着同一组总的节点的集合。再根据定理 3 和括号定理可知，在算法第 1 行所述的深度优先搜索中，祖宗是其所在强连通分支中第一个发现最后一个完成的结点。

为了弄清楚为什么在算法 Strongly_Connected_Components 的第 3 行要对 G^T 进行深度优先搜索，我们考察算法的第 1 行的深度优先搜索所计算出的具有最大完成时刻的结点 r。根据祖宗的定义可知结点 r 必为一祖宗结点，这是因为它是自身的祖宗：它可以到达自身且图中其他结点的完成时刻均小于它。在 r 的强连通分支中还有其他哪些节点？它们是那些以 F 为祖宗的结点——指可达 r 但不可达任何完成时刻大于 f[r]的结点的那些结点。但由于在 G 中 r 是完成时刻最大的结点，所以 r 的强连通分支仅由那些可达 r 的结点组成。换句话说，r 的强连通支由那些在 G^T 中从 r 可达的顶点组成。在算法第 3 行的深度优先搜索识别出所有属于 r 强连通支的结点，并把它们置为黑色(宽度优先搜索或任何对可达结点的搜索可以同样容易地做到这一点)。

在执行完第 3 行的深度优先搜索并识别出 r 的强连通分支以后，算法又从不属于 r 强连通分支且有着最大完成时刻的任何结点 r'重新开始搜索。结点，r'必为其自身的祖宗，因为由它不可能达到任何完成时刻大于它的其他结点(否则 r'将包含于 r 的强连通分支中)。根据类似的推理，可达 r'且不为黑色的任何结点必属于 r'的强连通分支，因而在第 3 行的深度优先搜索继续进行时，通过在 G^T 中从 r'开始搜索可以识别出属于 r'的强连通分支的每个结点并将其置为黑色。

因此通过第 3 行的深度优先搜索可以对图“层层剥皮”，逐个取得图的强连通分支。把每个强连通分支的祖宗作为自变量调用 DFS_Visit 过程，我们就可在过程 DFS 的第 7 行识别出每一支。DFS_Visit 过程中的递归调用最终使支内每个结点都成为黑色。当 DFS_Visit 返回到 DFS 中时，整个支的结点都变成黑色且被“剥离”，接着 DFS 在那些非黑色结点中寻找具有最大完成时刻的结点并把该结点作为另一支的祖宗继续上述过程。

下面的定理形式化了以上的论证。

定理 5　过程 Strongly_Connected_Components(G)可正确计算出有向图 G 的强连通分支。

证明：

通过对在 G^T 上进行深度优先搜索中发现的深度优先树的数目进行归纳，可以证明每棵树中的结点都形成一强连通分支。归纳论证的每一步骤都证明对 G^T 进行深度优先搜索形成的树是一强连通分支，假定所有在先生成的树都是强连通分支。归纳的基础是显而易见的，这是因为产生第一棵树之前无其他树，因而假设自然成立。

考察对 G^T 进行深度优先搜索所产生的根为 r 的一棵深度优先树 T，设 C(r)表示 r 为祖宗的所有结点的集合：

$C(r)=\{v\in V|\Phi(v)=r\}$

现往我们来证明结点 u 被放在树 T 中当且仅当 $u\in C(r)$。

→：由定理 2 可知 C(r)中的每一个结点都终止于同一棵深度优先树。因为 $r\in C(r)$且 r 是 T 的根，所以 C(r)中的每个元素皆终止于 T。

←：通过对两种情形 $f[\Phi(w)]>f[r]$或 $f[\Phi(w)]<f[r]$分别考虑，我们来证明这样的结点 w 不在树 T 中。通过对已发现树的数目进行归纳可知满足 $f[\Phi(w)]>f[r]$的任何结点 w 不在树 T 中，这是由于在选择到 r 时，w 已经被放在根为$\Phi(w)$的树中。满足 $f[\Phi(w)]<f[r]$的任意结点 w 也不可能在树 T 中，这是因为若 w 被放入 T 中，则有 w→r：因此根据式(2)和性质 $r=\Phi(r)$，可得 $f[\Phi(w)]\geqslant f[\Phi(r)]=f[r]$，这和 $f[\Phi(w)]<f[r]$相矛盾。

这样树 T 仅包含那些满足$\Phi(u)=r$ 的结点 u，即 T 实际上就是强连通支 C(r)，这样就完成了归纳证明

最小生成树的形成

假设已知一无向连通图 G=(V,E)，其加权函数为 w:E→R，我们希望找到图 G 的最小生成树。后文所讨论的两种算法都运用了贪心方法，但在如何运用贪心法上却有所不同。

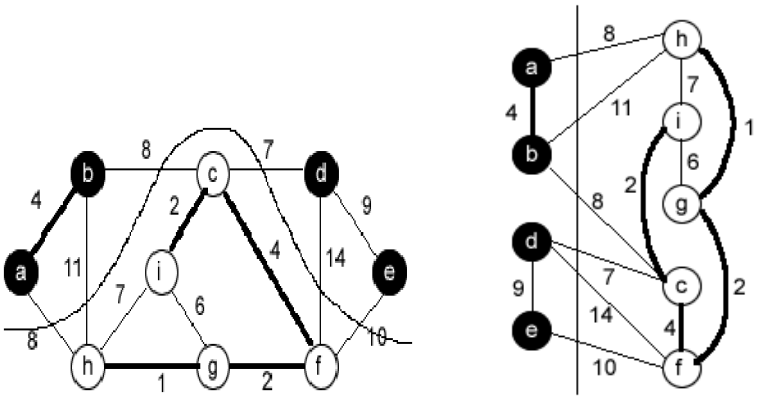
下列的算法 GENERNIC-MIT 正是采用了贪心策略，每步形成最小生成树的一条边。算法设置了集合 A，该集合一直是某最小生成树的子集。在每步决定是否把边(u,v)添加到集合 A 中，其添加条件是 A∪{(u,v)}仍然是最小生成树的子集。我们称这样的边为 A 的安全边，因为可以安全地把它添加到 A 中而不会破坏上述条件。

```
GENERNIC-MIT(G,W)
1. A←
2. while A 没有形成一棵生成树
3       do   找出 A 的一条安全边(u,v);
4.           A←A∪{(u,v)};
5. return A
```

注意从第 1 行以后，A 显然满足最小生成树子集的条件。第 2-4 行的循环中保持着这一条件，当第 5 行中返回集合 A 时，A 就必然是一最小生成树。算法最棘手的部分自然是第 3 行的寻找安全边。必定存在一生成树，因为在执行第 3 行代码时，根据条件要求存在一生成树 T，使 A T，且若存在边(u,v) T 且(u,v) A，则(u，v)是 A 的安全边。

在本节余下部分中，我们将提出一条确认安全边的规则（定理 1），下一节我们将具体讨论运用这一规则寻找安全边的两个有效的算法。

首先我们来定义几个概念。有向图 G=(V,E)的割(S,V-S)是 V 的一个分划。当一条边(u,v) E 的一个端点属于 S 而另一端点属于 V-S，则我们说边(u,v)通过割(S,V-S)。若集合 A 中没有边通过割，则我们说割不妨害边的集合 A。如果某边是通过割的具有最小权值的边，则称该边为通过割的一条轻边。要注意在链路的情况下可能有多条轻边。从更一般意义来讲，如果一条边是满足某一性质的所有边中具有最小权值的边，我们就称该边为满足该性质的一条轻边。图 2 说明了这些概念。(a)集合 S 中的结点为黑色结点，V-S 中的那些结点为白色结点。连接白色和黑色结点的那些边为通过该割的边。边(d,c)为通过该割的唯一一条轻边。子集 A 包含阴影覆盖的那些边，注意，由于 A 中没有边通过割，所以割(S,V-S)不妨害 A。(b)对于同一张图，我们把集合 S 中的结点放在图的左边，集合 V-S 中的结点放在图的有边。如果某条边使左边的顶点与右边的顶点相连则我们说该边通过割。



(a)　　(b)

图 2 从两种途径来观察图 1 所示图的割(S,V-S)

确认安全边的规则由下列定理给出。

定理 1

设图 G(V,E)是一无向连通图，且在 E 上定义了相应的实数值加权函数 w，设 A 是 E

的一个子集且包含于 G 的某个最小生成树中，割(S,V-S)是 G 的不妨害 A 的任意割且边(u,v)是穿过割(S,V-S)的一条轻边，则边(u,v)对集合 A 是安全的。

证明：

设 T 是包含 A 的一棵最小生成树，并假定 T 不包含轻边(u,v)，因为若包含就完成证明了。我们将运用“剪贴技术”(cut-and-paste technique)建立另一棵包含 A∪{(u，v)}的最小生成树 T ，并进而证明(u，v)对 A 是一条安全边；

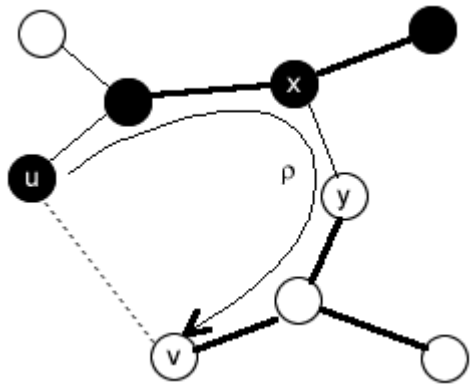


图 3 定理 1 的证明

图 3 演示出了定理 1 的证明。S 中的结点为黑色，V-S 中的结点为白色，边(u,v)与 T 中从 u 到 v 的通路 P 中的边构成一回路。由于 u 和 v 处于割(S,V-S)的相对的边上，因此在 T 中的通路 P 上至少存在一条边也通过割。设(x,y)为满足此条件的边。因为割不妨害 A，所以边(x,y)不属于 A。又因为(x,y)处于 T 中从 u 到 v 的唯一通路上，所以去掉边(x,y)就会把 T 分成两个子图。这时加入边(u,v)以形成一新的生成树 T =T-{(x,y)}∪{(u,v)}。

下一步我们证明 T 是一棵最小生成树。因边(u,v)是通过割(S,V-S)的一条轻边且边(x,y)也通过割，所以有 w(u,v) w(x,y)，因此 w(T)=w(T)-w(x,y)+w(u,v) (T)，但 T 是最小生成树，所以 w(T)=w(T)，因此 T 必定也是最小生成树。

现在还要证明(u,v)实际上是 A 的安全边。由于 A T 且(x,y) A，所以有 A T ，则 A∪{(u,v)} T 。而 T 是最小生成树，因而(u,v)对 A 是安全的。□

定理 1 使我们可以更好地了解算法 GENERNIC-MIT 在连通图 G=(V,E)上的执行流程。在算法执行过程中，集合 A 始终是无回路的，否则包含 A 的最小生成树包含一个环，这是不可能的。在算法执行中的任何一时刻，图 G_A=(V,A)是一森林且 G_A 的每一连通支均为树。其中某些树可能只包含一个结点，例如在算法开始时，A 为空集，森林中包含|V|棵树，每个顶点对应一棵。此外，对 A 安全的任何边(u,v)都连接 G_A 中不同的连通支，这是由于 A∪{(u,v)}必定不包含回路。

随着最小生成树的|V|-1 条边相继被确定，GENERNIC-MIT 中第 2-4 行的循环也随之要执行|V|-1 次。初始状态下，A= ，G_A 中有|V|棵树，每个迭代过程均将减少一棵树，当森林中只包含一棵树时，算法执行终止。

第 2 节中论述的两种算法均使用了下列定理 1 的推论。

推论 2

设 G=(V,E)是一无向连通图，且在 E 上定义了相应的实数值加权函数 w，设 A 是 E 的子集且包含于 G 的某最小生成树中，C 为森林 G_A=(V,A)中的连通支(树)。 若边(u,v)是连接 C 和 G_A 中其他某连通支的一轻边，则边(u,v)对集合 A 是安全的。

证明：

因为割(C,V-C)不妨害 A，因此(u,v)是该割的一条轻边

2.6 Kruskal 算法和 Prim 算法

本节所阐述的两种最小生成树算法是上节所介绍的一般算法的细化。每个算法均采用一特定规则来确定 GENERIC-MST 算法第 3 行所描述的安全边；在 Kruskal 算法中，集合 A 是一森林，加大集合 A 中的安全边总是图中连结两不同连通支的最小权边。在 Prim 算法中，集合 A 仅形成单棵树。添加入集合 A 的安全边总是连结树与一非树结点的最小权边。

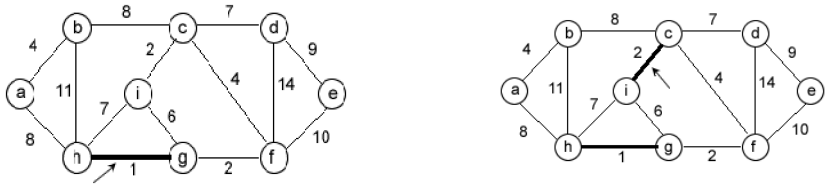
2.6.1 Kruskal 算法

Kruskal 算法是直接基于上一节中给出的一般最小生成树算法的基础之上的。该算法找出森林中连结任意两棵树的所有边中具有最小权值的边(u,v)作为安全边，并把它添加到正在生长的森林中。设 C_1 和 C_2 表示边(u,v)连结的两棵树。因为(u,v)必是连 C_1 和其他某棵树的一条轻边，所以由推论 2 可知(u,v)对 C_1 是安全边。Kruskal 算法同时也是一种贪心算法，因为算法每一步添加到森林中的边的权值都尽可能小。

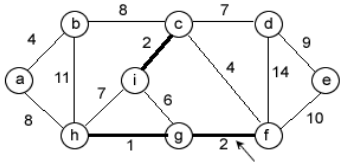
Kruskal 算法的实现类似于计算连通支的算法。它使用了分离集合数据结构以保持数个互相分离的元素集合。每一集合包含当前森林中某个树的结点，操作 FIND-SET(u)返回包含 u 的集合的一个代表元素，因此我们可以通过 FIND-SET(v)来确定两结点 u 和 v 是否属于同一棵树，通过操作 UNION 来完成树与树的联结。

```
MST-KRUSKAL(G,w)
1. A←
2. for  每个结点 v V[G]
3.     do MAKE-SET(v)
4.  根据权 w 的非递减顺序对 E 的边进行排序
5. for  每条边(u,v)    E，按权的非递减次序
6.     do if FIND-SET(u)  ≠  FIND-SET(v)
7.             then A←A∪{(u,v)}
8.                     UNION(u,v)
9    return A
```

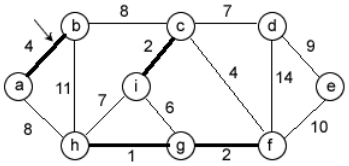
Kruskal 算法的工作流程如图 4 所示。阴影覆盖的边属于正在生成的森林 A。算法按权的大小顺序考察各边。箭头指向算法每一步所考察到的边。第 1-3 行初始化集合 A 为空集并建立|V|棵树，每裸树包含图的一个结点。在第 4 行中根据其权值非递减次序对 E 的边进行排序。在第 5-8 行的 for 循环中首先检查对每条边(u,v)其端点 u 和 v 是否属于同一棵树。如果是，则把(u,v)加入森林就会形成回路，所以这时放弃边(u,v)。如果不是，则两结点分属不同的树，由第 7 行把边加入集合 A 中，第 8 行对两棵树中的结点进行归并。



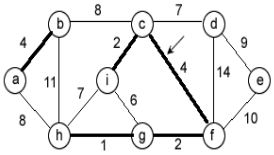
(a)　　(b)



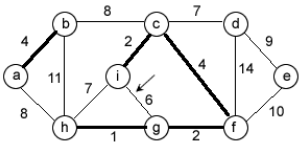
(c)



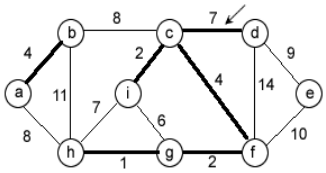
(d)



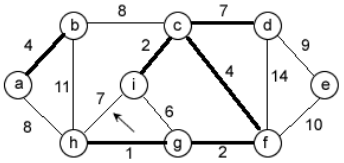
(e)



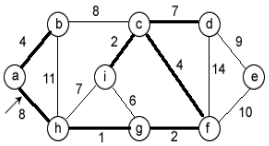
(f)



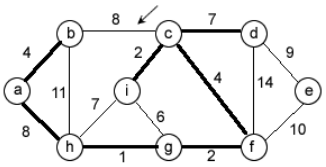
(g)



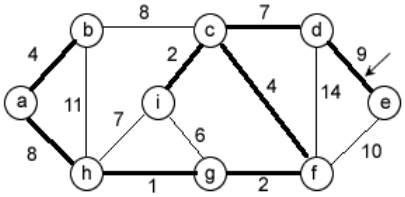
(h)



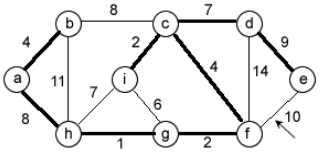
(i)



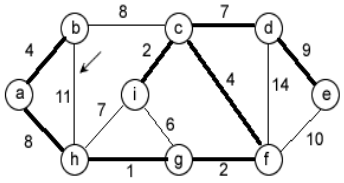
(j)



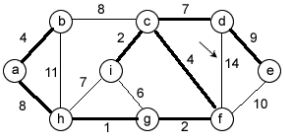
(k)



(l)



(m)



(n)

图 4 Kruskal 算法在图 1 所示的图上的执行流程

Kruskal 算法在图 G=(V,E)上的运行时间取决于分离集合这一数据结构如何实现。我们采用在分离集合中描述的按行结合和通路压缩的启发式方法来实现分离集合森林的结构，这是由于从渐近意义上来说，这是目前所知的最快的实现方法。初始化需占用时间 O(V)，第 4 行中对边进行排序需要的运行时间为 O(E log E)；对分离集的森林要进行 O(E)次操作，总共需要时间为 O(E (E,V))，其中 函数为 Ackerman 函数的反函数。因为 (E,V)=O(log

E)，所以 KruskaI 算法的全部运行时间为 O(E $^{\log}$ E)。

2.6.2 Prim 算法

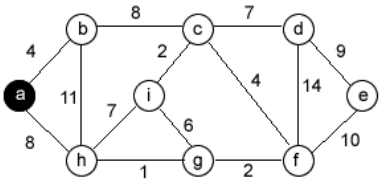
正如 Kruskal 算法一样，Prim 算法也是第上一节中讨论的一般最小生成树算法的特例。Prim 算法的执行非常类似于寻找图的最短通路的 Dijkstra 算法。Prim 算法的特点是集合 A 中的边总是只形成单棵树。如图 5 所示，阴影覆盖的边属于正在生成的树，树中的结点为黑色。在算法的每一步，树中的结点确定了图的一个割，并且通过该割的轻边被加进树中。树从任意根结点 r 开始形成并逐渐生长直至该树跨越了 V 中的所有结点。在每一步，连接 A 中某结点到 V-A 中某结点的轻边被加入到树中，由推论 2，该规则仅加大对 A 安全的边，因此当算法终止时，A 中的边就成为一棵最小生成树。因为每次添加到树中的边都是使树的权尽可能小的边，因此上述策略也是贪心的。

有效实现 Prim 算法的关键是设法较容易地选择一条新的边添加到由 A 的边所形成的树中，在下面的伪代码中，算法的输入是连通图 G 和将生成的最小生成树的根 r。在算法执行过程中，不在树中的所有结点都驻留于优先级基于 key 域的队列 Q 中。对每个结点 v，key[v]是连接 v 到树中结点的边所具有的最小权值；按常规，若不存在这样的边则 key[v]= ∞。域 [v]说明树中 v 的“父母”。在算法执行中，GENERIC-MST 的集合 A 隐含地满足：

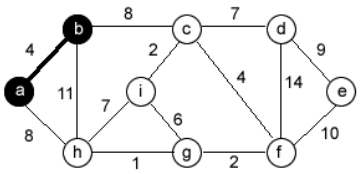
A={(v, [v])|v V-{r}-Q}

当算法终止时，优先队列 Q 为空，因此 G 的最小生成树 A 满足:

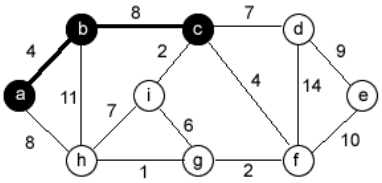
A={(v, [v])|v V-{r}}



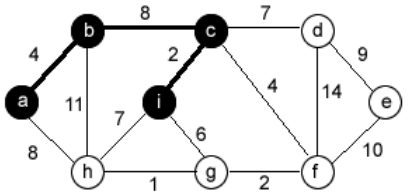
(a)



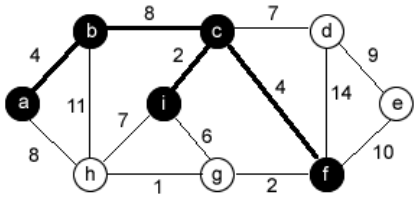
(b)



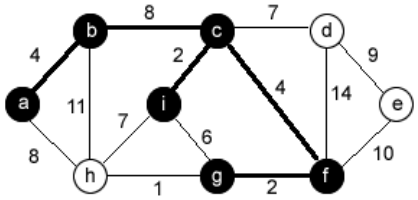
(c)



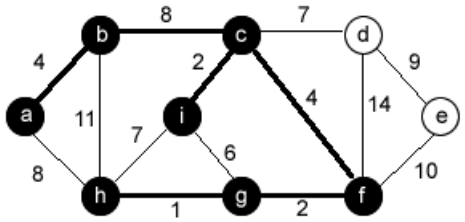
(d)



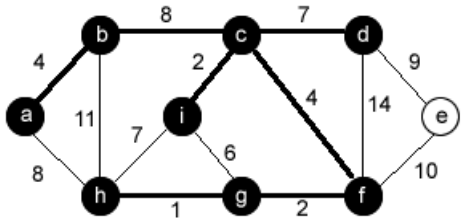
(e)



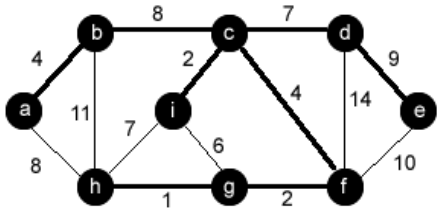
(f)



(g)



(h)



(i)

图 5 Prim 算法在图 1 所示的图上的执行流程

```
 MST-PRIM(G,w,r)
1. Q←V[G]
2. for  每个 u    Q
3.      do key[u]←∞
4. key[r]←0
5.     [r]←NIL
6. while Q≠
7.      do u←EXTRACT-MIN(Q)
8.           for  每个 v    Adj[u]
9.                do if v    Q and w(u,v)<key[v]
10.                        then     [v]←u
11.                                 key[v]←w(u,v)
```

Prim 算法的工作流程如图 5 所示。第 1-4 行初始化优先队列 Q 使其包含所有结点，置每个结点的 key 域为∞（除根 r 以外），r 的 key 域被置为 0。第 5 行初始化 [r]的值为 NIL，这是由于 r 没有父母。在整个算法中，集合 V-Q 包含正在生长的树中的结点。第 7 行识别出与通过割(V-Q，Q)的一条轻边相关联的结点 u Q（第一次迭代例外，根据第 4 行这时 u=r）。从集合 Q 中去掉 u 后把它加入到树的结点集合 V-Q 中。第 8-11 行对与 u 邻接且不在树中的每个结点 v 的 key 域和 域进行更新，这样的更新保证 key[v]=w(v, [v])

且(v, [v])是连接 v 到树中某结点的一条轻边。

Prim 算法的性能取决于我们如何实现优先队列 Q。若用二叉堆来实现 Q,我们可以使用过程 BUILD-HEAP 来实现第 1-4 行的初始化部分，其运行时间为 O(V)。循环需执行|V|次，且由于每次 EXTRACT-MIN 操作需要 O($\log$ V)的时间，所以对 EXTRACT-MIN 的全部调用所占用的时间为 O(V $\log$ V)。第 8-11 行的 for 循环总共要执行 O(E)次，这是因为所有邻接表的长度和为 2|E|。在 for 循环内部，第 9 行对队列 Q 的成员条件进行测试可以在常数时间内完成，这是由于我们可以为每个结点空出 1 位(bit)的空间来记录该结点是否在队列 Q 中，并在该结点被移出队列时随时对该位进行更新。第 11 行的赋值语句隐含一个对堆进行的 DECREASE-KEY 操作，该操作在堆上可用 0($\log$ V)的时间完成。因此，Prim 算法的整个运行时间为 O(V $\log$ V+E $\log$ V)=O(E $\log$ V)，从渐近意义上来说，它和实现 Kruskal 算法的运行时间相同。

通过使用 Fibonacci 堆，Prim 算法的渐近意义上的运行时间可得到改进。在 Fibonacci 堆中我们已经说明，如果|V|个元素被组织成 Fibonacci 堆，可以在 O($\log$ V)的平摊时间内完成 EXTRACT-MIN 操作，在 O(1)的平摊时间里完成 DECREASE-KEY 操作（为实现第 11 行的代码），因此，若我们用 Fibonacci 堆来实现优先队列 Q，Prim 算法的运行时间可以改进为 O(E+V $\log$ V)。

3 计算几何基础

3.1 矢量减法

设二维矢量 P = （x1,y1） ，Q = (x2,y2)
则矢量减法定义为： P - Q = (x1 - x2 , y1 - y2)
显然有性质 P - Q = - (Q - P)
如不加说明，下面所有的点都看作矢量，两点的减法就是矢量相减；

3.2 矢量叉积

设矢量 P = （x1,y1） ，Q = (x2,y2)
则矢量叉积定义为： P × Q = x1*y2 - x2*y1 得到的是一个标量
显然有性质 P × Q = - (Q × P) P × (- Q) = - (P × Q)
如不加说明，下面所有的点都看作矢量，点的乘法看作矢量叉积；
叉乘的重要性质：

- ✧ 若 P × Q > 0， 则 P 在 Q 的顺时针方向
- ✧ 若 P × Q < 0， 则 P 在 Q 的逆时针方向
- ✧ 若 P × Q = 0， 则 P 与 Q 共线，但可能同向也可能反向

3.3 包含关系的判断

判断点在线段上

设点为 Q，线段为 P1P2 ，判断点 Q 在该线段上的依据是：
(Q - P1) × (P2 - P1) = 0　且 Q 在以 P1，P2 为对角顶点的矩形内

判断两线段是否相交

我们分两步确定两条线段是否相交：
(1). 快速排斥试验
设以线段 P1P2 为对角线的矩形为 R， 设以线段 Q1Q2 为对角线的矩形为 T，如果 R 和 T 不相交，显然两线段不会相交；
(2). 跨立试验
如果两线段相交，则两线段必然相互跨立对方，如图 1 所示。在图 1 中，P1P2 跨立 Q1Q2 ，则矢量 (P1 - Q1) 和(P2 - Q1)位于矢量(Q2 - Q1) 的两侧，即
(P1 - Q1) × (Q2 - Q1) * (P2 - Q1) × (Q2 - Q1) < 0
上式可改写成
(P1 - Q1) × (Q2 - Q1) * (Q2 - Q1) × (P2 - Q1) > 0
当(P1 - Q1) × (Q2 - Q1) = 0 时，说明(P1 - Q1) 和 (Q2 - Q1)共线，
但是因为已经通过快速排斥试验，所以 P1 一定在线段 Q1Q2 上；同理，
(Q2 - Q1) ×(P2 - Q1) = 0 说明 P2 一定在线段 Q1Q2 上。
所以判断 P1P2 跨立 Q1Q2 的依据是：
(P1 - Q1) × (Q2 - Q1) * (Q2 - Q1) × (P2 - Q1) ≥ 0
同理判断 Q1Q2 跨立 P1P2 的依据是：
(Q1 - P1) × (P2 - P1) * (P2 - P1) × (Q2 - P1) ≥ 0
至此已经完全解决判断线段是否相交的问题。

判断线段和直线是否相交

如果线段 P1P2 和直线 Q1Q2 相交，则 P1P2 跨立 Q1Q2，即：
(P1 - Q1) × (Q2 - Q1) * (Q2 - Q1) × (P2 - Q1) ≥ 0

判断矩形是否包含点

只要判断该点的横坐标和纵坐标是否夹在矩形的左右边和上下边之间。

判断线段、折线、多边形是否在矩形中

因为矩形是个凸集，所以只要判断所有端点是否都在矩形中就可以了。

判断矩形是否在矩形中

只要比较左右边界和上下边界就可以了。

判断圆是否在矩形中

圆在矩形中的充要条件是：圆心在矩形中且圆的半径小于等于圆心到矩形四边的距离的最小值。

判断点是否在多边形中

以点 P 为端点，向左方作射线 L，由于多边形是有界的，所以射线 L 的左端一定在多边形外，考虑沿着 L 从无穷远处开始自左向右移动，遇到和多边形的第一个交点的时候，进入到了多边形的内部，遇到第二个交点的时候，离开了多边形，……所以很容易看出当 L 和多边形的交点数目 C 是奇数的时候，P 在多边形内，是偶数的话 P 在多边形外。但是有些特殊情况要加以考虑。如果 L 和多边形的顶点相交，有些情况下交点只能计算一个，有些情况下交点不应被计算（自己画个图就明白了）；如果 L 和多边形的一条边重合，这条边应该被忽略不计。为了统一起见，我们在计算射线 L 和多边形的交点的时候，

1。对于多边形的水平边不作考虑；

2。对于多边形的顶点和 L 相交的情况，如果该顶点是其所属的边上纵坐标较大的顶点，则计数，否则忽略；

3。对于 P 在多边形边上的情形，直接可判断 P 属于多边行。

由此得出算法的伪代码

如下：

```
1. count ← 0;
2. 以 P 为端点，作从右向左的射线 L;
3. for 多边形的每条边 s
4.     do if P 在边 s 上
5.             then return true;
6.         if s 不是水平的
7.             then if s 的一个端点在 L 上且该端点是 s 两端点中纵坐标较大的端点
9.                     then count ← count+1
10.                else if s 和 L 相交
11.                    then count ← count+1;
12. if count mod 2 = 1
13.     then return true
14.     else return false;
```

其中做射线 L 的方法是：设 P’的纵坐标和 P 相同，横坐标为正无穷大（很大的一个正

数），则 P 和 P’就确定了射线 L。这个算法的复杂度为 O(n)。

判断线段是否在多边形内

线段在多边形内的一个必要条件是线段的两个端点都在多边形内；

如果线段和多边形的某条边内交（两线段内交是指两线段相交且交点不在两线段的端点），因为多边形的边的左右两侧分属多边形内外不同部分，所以线段一定会有一部分在多边形外。于是我们得到线段在多边形内的第二个必要条件：线段和多边形的所有边都不内交；线段和多边形交于线段的两端点并不会影响线段是否在多边形内；但是如果多边形的某个顶点和线段相交，还必须判断两相邻交点之间的线段是否包含与多边形内部。因此我们可以先求出所有和线段相交的多边形的顶点，然后按照 X-Y 坐标排序，这样相邻的两个点就是在线段上相邻的两交点，如果任意相邻两点的中点也在多边形内，则该线段一定在多边形内。证明如下：

命题 1：

如果线段和多边形的两相邻交点 P1 ，P2 的中点 P’ 也在多边形内，则 P1, P2 之间的所有点都在多边形内。

证明：

假设 P1,P2 之间含有不在多边形内的点，不妨设该点为 Q，在 P1, P’之间，因为多边形是闭合曲线，所以其内外部之间有界，而 P1 属于多边行内部，Q 属于多边性外部，P’属于多边性内部，P1-Q-P’完全连续，所以 P1Q 和 QP’一定跨越多边形的边界，因此在 P1,P’之间至少还有两个该线段和多边形的交点，这和 P1P2 是相邻两交点矛盾，故命题成立。证毕

由命题 1 直接可得出推论：

推论 2：

设多边形和线段 PQ 的交点依次为 P1,P2,……Pn，其中 Pi 和 Pi+1 是相邻两交点，线段 PQ 在多边形内的充要条件是：P，Q 在多边形内且对于 i =1, 2,……, n-1，Pi ,Pi+1 的中点也在多边形内。

在实际编程中，没有必要计算所有的交点，首先应判断线段和多边形的边是否内交，倘若线段和多边形的某条边内交则线段一定在多边形外；如果线段和多边形的每一条边都不内交，则线段和多边形的交点一定是线段的端点或者多边形的顶点，只要判断点是否在线段上就可以了。

至此我们得出算法如下：

```
1. if 线端 PQ 的端点不都在多边形内
2.     then return false;
3.  点集 pointSet 初始化为空;
4. for 多边形的每条边 s
5.     do if 线段的某个端点在 s 上
6.             then 将该端点加入 pointSet;
7.         else if s 的某个端点在线段 PQ 上
8.             then 将该端点加入 pointSet;
9.         else if s 和线段 PQ 相交              // 这时候可以肯定是内交
10.            then return false;
```

11. 将 pointSet 中的点按照 X-Y 坐标排序，X 坐标小的排在前面，
对于 X 坐标相同的点，Y 坐标小的排在前面；
12. for pointSet 中每两个相邻点 pointSet[i] , pointSet[i+1]
13. do if pointSet[i] , pointSet[i+1] 的中点不在多边形中
14. then return false;
15. return true;

这个算法的复杂度也是 O(n)。其中的排序因为交点数目肯定远小于多边形的顶点数目 n，所以最多是常数级的复杂度，几乎可以忽略不计。

判断折线在多边形内

只要判断折线的每条线段是否都在多边形内即可。设折线有 m 条线段，多边形有 n 个
顶点，则复杂度为 O(m*n)。

判断多边形是否在多边形内

只要判断多边形的每条边是否都在多边形内即可。判断一个有 m 个顶点的多边形是否在一个有 n 个顶点的多边形内复杂度为 O(m*n)。

判断矩形是否在多边形内

将矩形转化为多边形，然后再判断是否在多边形内。

判断圆是否在多边形内

只要计算圆心到多边形的每条边的最短距离，如果该距离大于等于圆半径则该圆在多边形内。计算圆心到多边形每条边最短距离的算法在后文阐述。

判断点是否在圆内

计算圆心到该点的距离，如果小于等于半径则该点在圆内。

判断线段、折线、矩形、多边形是否在圆内

因为圆是凸集，所以只要判断是否每个顶点都在圆内即可。

判断圆是否在圆内

设两圆为 O1,O2，半径分别为 r1, r2，要判断 O2 是否在 O1 内。先比较 r1，r2 的大小，如果 r1<r2 则 O2 不可能在 O1 内；否则如果两圆心的距离大于 r1 - r2 ，则 O2 不在 O1 内；否则 O2 在 O1 内。

3.4 距离的计算

计算点到线段的最近点

如果该线段平行于 X 轴（Y 轴），则过点 point 作该线段所在直线的垂线，垂足很容易求得，然后计算出垂足，如果垂足在线段上则返回垂足，否则返回离垂足近的端点；如果该线段不平行于 X 轴也不平行于 Y 轴，则斜率存在且不为 0。设线段的两端点为 pt1 和 pt2，斜率为：

k = (pt2.y - pt1. y) / (pt2.x - pt1.x);

该直线方程为：

y = k* (x - pt1.x) + pt1.y

其垂线的斜率为 - 1 / k，

垂线方程为：

y = (-1/k) * (x - point.x) + point.y

联立两直线方程解得：

x = (k^2 * pt1.x + k * (point.y - pt1.y) + point.x) / (k^2 + 1)

y = k * (x - pt1.x) + pt1.y;

然后再判断垂足是否在线段上，如果在线段上则返回垂足；如果不在则计算两端点到垂足的距离，选择距离垂足较近的端点返回。

计算点到折线、矩形、多边形的最近点

只要分别计算点到每条线段的最近点，记录最近距离，取其中最近距离最小的点即可。

计算点到圆的最近距离

如果该点在圆心，则返回 UNDEFINED

连接点 P 和圆心 O，如果 PO 平行于 X 轴，则根据 P 在 O 的左边还是右边计算出最近点的横坐标为 centerPoint.x - radius 或 centerPoint.x + radius， 如图 4 (a)所示；

如果 PO 平行于 Y 轴，则根据 P 在 O 的上边还是下边计算出最近点的纵坐标为 centerPoint.y + radius 或 centerPoint.y - radius， 如图 4 (b)所示。

如果 PO 不平行于 X 轴和 Y 轴，则 PO 的斜率存在且不为 0，如图 4©所示。这时直线 PO 斜率为 k = （ P.y - O.y ）/ (P.x - O.x)

直线 PO 的方程为：

y = k * (x - P.x) + P.y

设圆方程为:

(x - O.x) ^2 + (y - O.y) ^2 = r ^2，

联立两方程组可以解出直线 PO 和圆的交点，取其中离 P 点较近的交点即可。

计算两条共线的线段的交点

对于两条共线的线段，它们之间的位置关系有图 5 所示的几种情况。

图 5(a)中两条线段没有交点；图 5 (b) 和 (d) 中两条线段有无穷焦点；图 5 (c) 中两条线段有一个交点。设 line1 是两条线段中较长的一条，line2 是较短的一条，如果 line1 包含了 line2 的两个端点，则是图 5(d)的情况，两线段有无穷交点；如果 line1 只包含 line2 的一个端点，那么如果 line1 的某个端点等于被 line1 包含的 line2 的那个端点，则是图 5©的情况，这时两线段只有一个交点，否则就是图 5©的情况，两线段也是有无穷的交点；如果 line1 不包含 line2 的任何端点，则是图 5(a)的情况，这时两线段没有交点。

计算线段或直线与线段的交点

设一条线段为 L0 = P1P2，另一条线段或直线为 L1 = Q1Q2 ，要计算的就是 L0 和 L1 的交点。

1.首先判断 L0 和 L1 是否相交（方法已在前文讨论过），如果不相交则没有交点，否则说明 L0 和 L1 一定有交点，下面就将 L0 和 L1 都看作直线来考虑。

2.如果 P1 和 P2 横坐标相同，即 L0 平行于 Y 轴

a)若 L1 也平行于 Y 轴，

i.若 P1 的纵坐标和 Q1 的纵坐标相同，说明 L0 和 L1 共线，假如 L1 是直线的话他们有无穷的交点，假如 L1 是线段的话可用”计算两条共线线段的交点”的算法求他们 的交点（该方法在前文已讨论过）；

ii.否则说明 L0 和 L1 平行，他们没有交点；

b)若 L1 不平行于 Y 轴，则交点横坐标为 P1 的横坐标，代入到 L1 的直线方程中可以计算出交点纵坐标；

3.如果 P1 和 P2 横坐标不同，但是 Q1 和 Q2 横坐标相同，即 L1 平行于 Y 轴，则交点横坐标为 Q1 的横坐标，代入到 L0 的直线方程中可以计算出交点纵坐标；

4.如果 P1 和 P2 纵坐标相同，即 L0 平行于 X 轴

a)若 L1 也平行于 X 轴，

i.若 P1 的横坐标和 Q1 的横坐标相同，说明 L0 和 L1 共线，假如 L1 是直线的话他们有无穷的交点，假如 L1 是线段的话可用”计算两条共线线段的交点”的算法求 他们的交点（该方法在前文已讨论过）；

ii.否则说明 L0 和 L1 平行，他们没有交点；

b)若 L1 不平行于 X 轴，则交点纵坐标为 P1 的纵坐标，代入到 L1 的直线方程中可以计

算出交点横坐标；

5.如果 P1 和 P2 纵坐标不同，但是 Q1 和 Q2 纵坐标相同，即 L1 平行于 X 轴，则交点纵坐标为 Q1 的纵坐标，代入到 L0 的直线方程中可以计算出交点横坐标；

剩下的情况就是 L1 和 L0 的斜率均存在且不为 0 的情况

a)计算出 L0 的斜率 K0，L1 的斜率 K1 ；

b)如果 K1 = K2

i.如果 Q1 在 L0 上，则说明 L0 和 L1 共线，假如 L1 是直线的话有无穷交点，假如 L1 是线段的话可用”计算两条共线线段的交点”的算法求他们的交点（该方法在 前文已讨论过）；

ii.如果 Q1 不在 L0 上，则说明 L0 和 L1 平行，他们没有交点。

c)联立两直线的方程组可以解出交点来

说明：这个算法并不复杂，但是要分情况讨论清楚，尤其是当两条线段共线的情况需要单独考虑，所以在前文将求两条共线线段的算法单独写出来。另外，一开始就先利用矢量叉乘判断线段与线段（或直线）是否相交，如果结果是相交，那么在后面就可以将线段全部看作直线来考虑。

3.5 求线段或直线与圆的交点

设圆心为 O，圆半径为 r，直线（或线段）L 上的两点为 P1,P2。

1.如果 L 是线段且 P1，P2 都包含在圆 O 内，则没有交点；否则进行下一步

2.如果 L 平行于 Y 轴，

a)计算圆心到 L 的距离 dis

b)如果 dis > r 则 L 和圆没有交点；

c)利用勾股定理，可以求出两交点坐标，如图 6(a)所示；但要注意考虑 L 和圆的相

切情况

3.如果 L 平行于 X 轴，做法与 L 平行于 Y 轴的情况类似；

4.如果 L 既不平行 X 轴也不平行 Y 轴，可以求出 L 的斜率 K，然后列出 L 的点斜式方程，和圆方程联立即可求解出 L 和圆的两个交点；

5.如果 L 是线段，对于 2，3，4 中求出的交点还要分别判断是否属于该线段的范围内。

3.6 算法模板

```
#include <math.h>
#include <stdlib.h>
#include <fstream.h>
#include <algorithm>

struct CPoint
{
double x, y;
};

bool equal(CPoint pt1, CPoint pt2);                    // 相等
double cross(CPoint pt1, CPoint pt2, CPoint pt0);      // 叉积
double dot(CPoint pt1, CPoint pt2, CPoint pt0);        // 点积
```

```
double dissqr(CPoint pt1, CPoint pt2);                    // 距离平方
double dis(CPoint pt1, CPoint pt2);                       // 距离
double area(CPoint a[], int ct);                          // 面积
double angle(CPoint pt1, CPoint pt2, CPoint pt0);         // 角度
        double ptToline(CPoint pt1, CPoint pt2, CPoint pt);   // 点到直线距离
        void convex(CPoint pt[], int ct, CPoint ans[], int &ans_ct); // 凸包
void root(CPoint pt1, CPoint pt2, CPoint pt, CPoint &rt);     // 点在线段上的投影
int center(CPoint a[], int ct, CPoint &pt);               // 物理重心
int dcmp(double a, double b);                             // 比较
int onSegment(CPoint pt1, CPoint pt2, CPoint pt0);        // 点在线段上
int inside(CPoint poly[], int ct, CPoint pt);             // 点在形内
int LineInter(CPoint pt1, CPoint pt2, CPoint pt3, CPoint pt4, CPoint &pt); // 直线交点
int SegInter(CPoint pt1, CPoint pt2, CPoint pt3, CPoint pt4, CPoint &pt);  // 线段交点
int dcmp(double a, double b=0.0)//can be int if all int
{
    if(a-b<1e-9&&b-a<1e-9) return 0;
    if(a>b) return 1;
    return -1;
}

bool equal(CPoint pt1, CPoint pt2)
{
    if(dcmp(pt1.x, pt2.x)!=0) return 0;
    if(dcmp(pt1.y, pt2.y)!=0) return 0;
    return 1;
}

    double cross(CPoint pt1, CPoint pt2, CPoint pt0)//can be int if all int// >0 if pt1 is counter-clock of
 pt2 relative to pt0
{
return (pt1.x-pt0.x)*(pt2.y-pt0.y) - (pt1.y-pt0.y)*(pt2.x-pt0.x);
}

    double dot(CPoint pt1, CPoint pt2, CPoint pt0)//can be int if all int
{
return (pt1.x-pt0.x)*(pt2.x-pt0.x) + (pt1.y-pt0.y)*(pt2.y-pt0.y);
}

        double dissqr(CPoint pt1, CPoint pt2)
        //can be int if all int
{
return (pt1.x-pt2.x)*(pt1.x-pt2.x) + (pt1.y-pt2.y)*(pt1.y-pt2.y);
}
```

```
        double dis(CPoint pt1, CPoint pt2)//must be double
{
return sqrt( (1.0*pt1.x-pt2.x)*(pt1.x-pt2.x) + (1.0*pt1.y-pt2.y)*(pt1.y-pt2.y) );
}

double area(CPoint a[], int ct)
{
        int i;
        double s = 0.0;
        for(i=1; i<ct-1; i++)
 {
s += cross(a[0], a[i], a[i+1]);
 }
        if(dcmp(s,0)<0) s = -s;
        return s/2;
        //if area is only for compare, /2 is not necessary;
}

int center(CPoint a[], int ct, CPoint &pt)
{
        int i;
        pt.x = 0; pt.y = 0;
        double s = 0.0, t;
        for(i=1; i<ct-1; i++)
 {
                t = cross(a[i], a[i+1],a[0]);
                pt.x += a[0].x*t + a[i].x*t + a[i+1].x*t;
                pt.y += a[0].y*t + a[i].y*t + a[i+1].y*t;
                s += t;
 }
        if(dcmp(s)==0) return 0;
        pt.x /= s*3; pt.y /= s*3;
        return 1;
}

double angle(CPoint pt1, CPoint pt2, CPoint pt0)
{
return acos(dot(pt1, pt2, pt0)/dis(pt1, pt0)/dis(pt2, pt0));
}

bool cmp(const CPoint &pt1, const CPoint &pt2)
{
        if(pt1.y==pt2.y) return pt1.x<pt2.x;
        return pt1.y<pt2.y;
```

```
}

    void convex(CPoint pt[], int ct, CPoint ans[], int &ans_ct)// 凸包// 如果需要凸包边上的点,只需要把’=’去掉就行了
{
    int i, hl;
    std::sort(pt, pt+ct, cmp);
    ans_ct = 2;
    ans[0] = pt[0]; ans[1] = pt[1];
    for(i=2; i<ct; i++)
 {
while(cross(pt[i], ans[ans_ct-1], ans[ans_ct-2])>=0 && ans_ct>1) ans_ct--;
ans[ans_ct] = pt[i]; ans_ct++;
 }
    hl = ans_ct;
    ans[ans_ct++] = pt[ct-2];
    for(i=ct-3; i>=0; i--)
 {
while(cross(pt[i], ans[ans_ct-1], ans[ans_ct-2])>=0 && ans_ct>hl) ans_ct--;
ans[ans_ct] = pt[i]; ans_ct++;
 }
ans_ct--;
}

int onSegment(CPoint pt1, CPoint pt2, CPoint pt0)
{
    if(dcmp(cross(pt1, pt2, pt0))!=0) return 0;
    if((pt0.x-pt1.x)*(pt0.x-pt2.x)>0) return 0;
    if((pt0.y-pt1.y)*(pt0.y-pt2.y)>0) return 0;
    return 1;
}

        int inside(CPoint poly[], int ct, CPoint pt)
        // 0 on one of the Segment
        // -1 outside the Poly
        // 1 inside
{
    double ans(0.0);
    int i;
    poly[ct] = poly[0];
    for(i=0; i<ct; i++)
 {
if(onSegment(poly[i], poly[i+1], pt)) return 0;
 }
```

```
for(i=0; i<ct; i++)
 {
ans += angle(poly[i], poly[i+1], pt);
 }
        if(dcmp(ans, acos(-1.0)*2)==0) return 1;
        return -1;
}
        int LineInter(CPoint pt1, CPoint pt2, CPoint pt3, CPoint pt4, CPoint &pt)// 0 paralle// 1 success
{
        double a, b;
        a = cross(pt3, pt2, pt1);
        b = cross(pt2, pt4, pt1);
        if(dcmp(a+b)==0) return 0;
        pt.x = (pt3.x*b + pt4.x*a) / (a+b);
        pt.y = (pt3.y*b + pt4.y*a) / (a+b);
        return 1;
}

        int SegInter(CPoint pt1, CPoint pt2, CPoint pt3, CPoint pt4, CPoint &pt)// -1 parallel// 0    not cross
        // 1    success
{
        double a, b;
        a = cross(pt3, pt2, pt1);
        b = cross(pt2, pt4, pt1);
        if(dcmp(a+b)==0) return -1;
        pt.x = (pt3.x*b + pt4.x*a) / (a+b);
        pt.y = (pt3.y*b + pt4.y*a) / (a+b);
        if(!onSegment(pt3, pt4, pt)) return 0;
        if(!onSegment(pt1, pt2, pt)) return 0;
        return 1;
}

double ptToline(CPoint pt1, CPoint pt2, CPoint pt)
{
        double ans;
        ans = dis(pt1, pt2);
        if(dcmp(ans)==0) return dis(pt1, pt);
        ans = cross(pt1, pt2, pt)/ans;
        if(ans<0) ans = -ans;
        return ans;
}

void root(CPoint pt1, CPoint pt2, CPoint pt, CPoint &rt)
{
```

```
        double t;
        if(equal(pt1,pt2))
 {
rt = pt1; return;
 }
        t = ptToline(pt1, pt2, pt);
        if(dcmp(t)==0)
 {
rt = pt; return;
 }
        t = sqrt(dissqr(pt1, pt) - t*t) / dis(pt1, pt2);
        rt.x = pt2.x * t + pt1.x *(1-t);
        rt.y = pt2.y * t + pt1.y *(1-t);
        if(dot(rt, pt, pt1)>=0) return;
        rt.x = -pt2.x * t + pt1.x *(1+t);
        rt.y = -pt2.y * t + pt1.y *(1+t);
}

int main()
{
CPoint pt1, pt2, pt3, pt0;
        pt1.x = 1; pt1.y = 0;
        pt2.x = 0; pt2.y = 1;
        pt3.x = 3; pt3.y = 0;
        root(pt1, pt2, pt3, pt0);
        cout << pt0.x << “ “ << pt0.y << endl;
        return 0;
}
```

第三篇 实践篇

"横看成岭侧成峰，
远近高低各不同"
学习高手的解题思路

第1章　《多边形》

长沙市长郡中学　金恺

问题描述：

给出 $n,m(n\geqslant m)$，从单位正 n 边形的顶点中选出 m 个，能组成多少种不同的 m 多边形？（若一个 m 边形可由另一个 m 边形通过旋转、翻转、平移得到，则认为两个 m 边形同种）

解决情况：

利用 Polyà定理完全解决。

时间复杂度为 $O(n^2K)$ 1；空间复杂度 $n+K$

算法梗概：

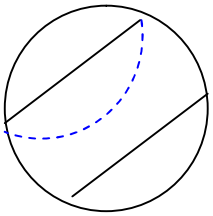
无需考虑平移，原问题转化为对 n 个元素 01 染色使 1 的个数为 m 的本质不同的个数。将旋转和翻转看作一个置换，用 Polyà定理解决。

正文

试题分析：

首先，由于 m 个点都是在同一个圆周上的，所以 m 个点构成的多边形一定为凸多边形。

其次，当 $m\geqslant 3$ 时，平移后不可能发生重合，

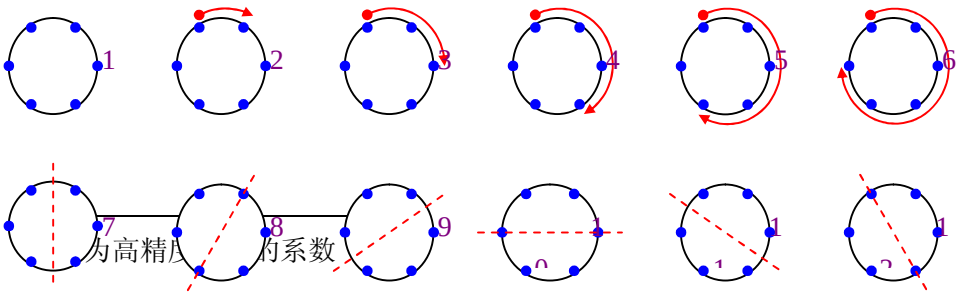


证明如下，当 $m\geqslant 3$ 时，若两种选点方案选出的点平移后重合，则在两种选点方案中肯定存在 3 个互相对应的点，这 3 个点构成一个全等三角形，不妨设为 ABC 和 A’B’C’。作正 n 的外接圆，由于 ⩢ AB　A’B’，A、B、A’、B’都为多边形顶点所以 AB 与 A’B’关于某条直径对称。如果 C 在优弧 AB 上，则 C’必在劣弧 A’B’上，由于△ABC≌△A’B’C’，所以 C 应该在劣弧 AB 沿 AB 翻折后所在的弧上（也就是图中的蓝色虚线上），所以与 C 在优弧 AB 上矛盾。如果 C 在劣弧 AB 上，则 C’必在优弧 A’B’上，同理可证。

因此只要考虑旋转和翻转，问题可以转化为：用 0、1 两种颜色给 n 个均匀分布在圆周上的点进行染色，使 m 个染成 1，$n-m$ 染成 0，绕着圆周的中心旋转或者按圆周的某条直径翻转重叠的方案视作同一种着色方案，求本质不同的方案数。

把每一种旋转或者翻转看作一个置换，如果不要求染色的个数，那么就只要计算每个置换循环的节数，而加上个数的限制，只要进行适当的修改便可以了：对于每个置换，把它看成若干个不相交的循环，它的 V 值设为：将置换中 m 个元素染成黑色、$n-m$ 个元素染成白色，并且使每个循环中的所有元素同色的方案数。那么最终答案就为所有置换的 V 值的平均数。

举个例子来说：$n=6$，$m=4$ 时。



为高精度　的系数

编号	置换	V 值	编号	置换	V 值
1	(0)(1)(2)(3)(4)(5)	C(4,6)	7	(0,1)(2,5)(3,4)	C(2,3)
2	(0,1,2,3,4,5)	0	8	(1)(4) (0,2)(3,5)	C(2,2)+C(1,2)
3	(0,2,4)(1,3,5)	0	9	(0,3)(1,2)(4,5)	C(2,3)
4	(0,3)(1,4)(2,5)	C(2,3)	10	(2)(5) (0,4) (1,3)	C(2,2)+C(1,2)
5	(0,4,2)(1,5,3)	0	11	(0,5)(1,4)(2,3)	C(2,3)
6	(0,5,4,3,2,1)	0	12	(0)(3)(1,5)(2,4)	C(2,2)+C(1,2)

C(*a*,*b*)表示组合数：从 *b* 个元素中选出 *a* 个的方案数。

所以中的方案数为 $\frac{C_6^4+4C_3^2+3C_2^2+3C_2^1}{12}=\frac{15+12+3+6}{12}=3$

下面推广到一般情况：

旋转：

定理：顺时针旋转 *i* 格的置换中，循环的个数为 *Gcd(n,i)*，每个循环的长度为 *n/Gcd(n,i)*。

证明如下：

#x 通过 1 次旋转变到了*#*(*x*+*i*)，通过 2 次选转变为*#*(*x*+2*i*)，……通过 *j* 次旋转后到*#*(*x*+*j*×*i*)(这里是指 mod *n* 意义下的相等)……。当第一次又回到 0 后，旋转的次数显然就是循环的长度。也就说循环的长度 *L* 等于满足 *j*×*i*≡0（mod *n*）的最小正整数 *j*。

设 a=Gcd(n,i)，n=a×b，i=a×c，则(b,c)=1。

∵j×i=n×t(L,i,n,t 都为整数)

∴j×c=b×t

∵Gcd(b,c)=1

∴b | j

∴j≥b

而 *j*=*b* 时，*j*×*i*= *b*×*a*×*c*=*n*×*a*≡0（mod *n*）。

故 *L*=*b*=*n/Gcd(n,i)*时，而由对称性可知每个循环的长度是相等的，所以循环的个数为 *Gcd(n,i)*。

根据定理，如果 *m*≡0（mod *L*），那么 $V=C_{N/L}^{M/L}=C_{Gcd(n,i)}^{M/L}$ 否则为 0；

翻转：这时分 *n*、*m* 的奇偶性考虑

当 *n* 为奇数时，只有一种情况，即：[*n*/2]个循环长度为 2，还有一个长度为 1，所以 $V=C_{[N/2]}^{[m/2]}$；

当 *n* 为偶数时，有两种情况：

有 *n*/2 个置换满足：有 *n*/2 个循环长度为 2，此时

若 *m* 为偶数，则 $V=C_{N/2}^{m/2}$；

若 *m* 为奇数，则 *V*=0。

有 *n*/2 个置换满足：*n*/2-1 个循环长度为 2，两个循环的长度为 1，此时

若 *m* 为偶数，则 $V=C_{n/2-1}^{m/2}+C_{n/2-1}^{m/2-1}=C_{n/2}^{m/2}$；

若 *m* 为奇数 $V=2C_{n/2-1}^{(m-1)/2}$

翻转的总结如下：

n 为偶 m 为奇	n 为奇或 n 为偶 m 为偶
$\sum_{i\text{为翻转}} V = nC_{[n/2]-1}^{[m/2]}$	$\sum_{i\text{为翻转}} V = nC_{[n/2]}^{[m/2]}$

优化：

计算翻转时的公式很简单，但旋转还很复杂。如果枚举旋转的格数，复杂度显然较高。有没有好方法呢？我们可以不枚举 i，反过来枚举 L。

由于 $L|M$、$L|N$ 所以 $L|(M,N)$。枚举了 L，再计算有多少个 i 使得 $0 \leqslant i \leqslant N-1$ 并且 $L=n/Gcd(n,i)$。 即 $Gcd(n,i) = n/L$。

不妨设 a = n / L = Gcd(n,i)

不妨设 $i = a \times t$ 则当且仅当 $Gcd(L, t) = 1$ 时

Gcd(n,i) = Gcd(a×L,a×t) = a

∵0 ⩽i <n ∴0 ⩽ $t< n / a = L$

∴满足这个条件的 t 的个数为 Φ（L）{ $\Phi(x)$表示不超过 x 的与 x 互质的数的正整数个数}

所以旋转的所有置换的 V 值之和为：$\sum_{L|Gcd(M,N)} C_{N/L}^{M/L}\phi(L)$

复杂度分析：而如果不用这个式子，需要计算的次数显然为 $\sum_{L|Gcd(M,N)}\phi(L) = Gcd(M,N)$。因此，我们将计算组合数的次数从 $Gcd(M,N)$降为了 $Gcd(M,N)$的约数个数，这样一来计算的次数将大大降低。当 $Gcd(M,N)$不超过 10000 时，最多只有 64 次，当 $Gcd(M,N)$不超过 32767 时，也不过只有 96 次。(由于高精度的运算占了主要的计算时间，所以做的时候不必要用上面的式子，而可以用一个 Tot 数组，Tot_i 记录下循环节个数为 i 的方案数，这样可以降低编程的难度)

优化前后，程序效率对比：

编号	N	M	优化前	优化后
1	20160	10080	1.10	0.49
2	30000	5000	0.88	0.44
3	32767	12683	0.88	0.88
4	32024	6048	1.26	0.60
5	32024	15120	3.08	1.15

编程环境：*FreePascal* 1.0.6 *Windows*2000 ®4 *CPU* 1700*MHz*

参考文献:

《组合数学》（第 2 版）清华大学出版社 卢开澄

第2章　《灌溉问题》

长沙市长郡中学　金恺

问题描述：

政府将提供资金开凿一条水道，将唯一的水源—山顶一个湖泊中的水引入他们的村落。在开凿水道时，人们却遇到了一个难题：怎样开凿才能得到最大的丰收呢？

要知道，山顶的水是非常有限的，并不可能灌溉到每一个地区。而且，不同地区的土地肥沃程度也是不一样的。虽然并非只有河流经过的地区才能被灌溉，但也不可能将水从很远的地方引过来。你现在必须想办法帮他们解决这个问题。

首先，你可以将这个山区划分成 $N\times N$ 的区域，湖泊的位置(x,y)，以及每个区域的海拔高度（开凿出的河流显然只能从高往低流），还有每块区域的土地价值也是已知的。同时，需要注意的是，你的河流是不允许有分支（支流）的，最多只能灌溉 M 块土地，而且被灌溉的土地只能在离河流的 R 格范围以内（包含对角线）。

注意：河流经过的地区土地依然保持其价值，可以选择是否对其灌溉。

你的目的就是找出一条最好的开凿河流的方法，使得能灌溉的土地价值总和最高。

(N<=20,M<=100,R<=5)

解决情况：

用深度优先搜索，加上动态规划剪枝。

10 个样例数据中有九个利用贪心就可以得出最优解。

讨论收获与感谢：

剪枝方法是从原国家队队员肖洲自出的一道题(《智能武器》)中得到的启发。

正文

试题分析：

很明显，由于河流的形状十分不规则（例如下图所示的两条河流）。可以灌溉的土地的形状就更是奇形怪状了，所以单独用动态规划来解决本题是不太现实的。而贪心和网络流更是无法着手，因此只能考虑用搜索来解决本题。

首先，搜索的对象应为河流，因为只要河流确定了位置，可灌溉区域也就随之确定，所选的区域也可以确定—从可灌溉区域中选出土地价值最高的 m 块区域就行了。但由于数据范围较大，所以必须要有非常高效的剪枝方法。

容易想到下面这条剪枝：搜索时，若河流已经流到了(x,y)这个位置了，有些区域可能永远都不可能被灌溉到了(比如说(x,y)比相邻四格海拔都低时，若$|x-x'|>R$，$|y-y'|>R$ 且(x',y')目前不能被灌溉，则(x',y')将永远不可能被灌溉到)。如果在现在或将来可能被灌溉的区域中选出 m 块价值最大的区域，它们的和仍然比目前的最优值小（或者相等），那么就可以剪枝了，因为继续搜下去不可能找到比当前最优解更优的解来。

这是最常用的卡上界剪枝，但是对于本题来说它的作用似乎不是很大，最根本的问题在于：上述方法估计的上界太大、太不准确、因此剪去的枝太少。为什么会出现这样的问题呢？关键的原因在于：虽然现在或将来可能被灌溉的这些区域都有可能被灌溉到，但是他们中的许多都不能够同时被灌溉到，有时只能两者取其一、甚至 n 里挑一，然而上述剪枝在计算上界的时候却没有能考虑到这一点，从而导致选了多个本不能同时选择的价值较大的区域，使得上界和真正的最优值偏离得太远。所以，要解决这个问题，就应该紧紧围

绕“如何更准确的估计上界”这一个环节入手，尽量改良上界，使剪枝效率得以提高。

稍加联想便会发现：上述方法求上界时，有点像是采用了二维的描述方法：即流到(x,y)后能够灌溉哪些区域。是不是能够适当的增加某些参数，使得状态的描述更精确呢？这时，你大概很快就会想到用 $F_{i,j,k}$ 来描述从(i,j)开始往下流（即将(i,j)看作湖泊的位置）选出 k 块区域的最大值，因为本题刚好就是要求 $F_{x,y,m}$ 呀！至此，一个很棘手的问题摆在面前，如何计算 $F_{i,j,k}$ 之呢？

能不能用动态规划求出它的准确值呢？前面已经说过了：不能！但是，不求准确值，只要求出它的一个近似最优解的上界也行呀，只要这样就可以用来剪枝了嘛。 显然，可以用下列的方法求出 $f_{i,j,k}$ 的上界：

$$f_{i,j,k}= Max\begin{cases} Irrigate_{i-R,j-R,i+R,j+R,k} \\ f_{i-1,j,l}+Irrigate_{i+R,j-R,i+R,j+R,k-l}\,(High_{i,j}>High_{i-1,j}) \\ f_{i+1,j,l}+Irrigate_{i-R,j-R,i-R,j+R,k-l}\,(High_{i,j}>High_{i+1,j}) \\ f_{i,j-1,l}+Irrigate_{i-R,j+R,i+R,j+R,k-l}\,(High_{i,j}>High_{i,j-1}) \\ f_{i,j+1,l}+Irrigate_{i-R,j-R,i+R,j-R,k-l}\,(High_{i,j}>High_{i,j+1}) \end{cases}$$

{其中 $Irrigate_{x1,y1,x2,y2,t}$ 表示在 $(x_1,y_1),(x_2,y_2)$这个矩形中选 t 块区域的最大值。}

方程的意义如下：河流从(i,j) 流出，最多只有五种可能的方式：

不再往后流；所以 $f_{i,j,k}=Irrigate_{i-R,j-R,i+R,j+R,k}$

往北流：

显然：$f_{i,j,k}$ 应该小于等于（$f_{i-1,j,l}$ +从新的可灌溉区域选出 $k-l$ 块区域能具有的最大土地价值）。{更严格的说：这里所说的“新的可灌溉区域” 只是真正的新的可灌溉区域的一个父集（相对于子集而说的），因为在这个集合里面的某些区域有可能从$(i-1,j)$流出来时能够灌溉，但是，由上图可以看出，这个集合以外的所有区域都不可能是新的可灌溉区域。

往南流：与第 2 种情况类似。

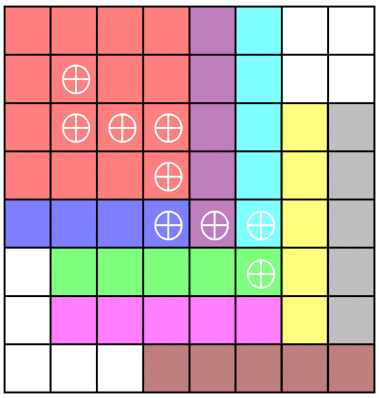
往西流：与第 2 种情况类似。

往东流：与第 2 种情况类似。

计算 $F_{i,j,x}$ 可以用记忆化搜索，既快捷又简单，**时间复杂度只有 $O(N^2MR)$**，空间复杂度也只要 N^2M。求出了 $F_{i,j,x}$ 后，在搜索时，若河流流到了(i,j)，目前可灌溉区域集合为 S，令 $S'\leftarrow S-\{(i,j)$直接可灌溉区域集合$\}$，若对于任意的 t，都满足 S'中价值最大的前 t 块区域的价值之和+$F_{i,j,m-t}$ 都小于当前的最优解，那么就可以剪枝了。

还有没有其他的可以提高剪枝效率的方法呢？剪枝的形式用 *Pascal* 语言的判断语句可以表示为：***if* 目前最大可能达到的价值和 > 目前求出的最大价值和 then *exit***；计算 $F_{i,j,x}$ 的目的就是降低前者的值来提高效率，同样的，如果能够增大后者的值，同样能够起到提高剪枝效率的目的。也就是说：如果能够在搜索进行之前就求到了一个较优的较优解，那么就能够剪掉更多的枝。

有没有办法能够求到一个比较优的较优解呢？容易知道，只要保证新的可灌溉区域与目前的可灌溉区域不重叠，那么就满足无后效性，也就能够通过动态规划来计算这种带有某些限制条件的最优解。如何才能使新的可灌溉区域与目前的可灌溉区域不重叠呢？不难证明：如果规定河流的方向只能向东、南（或东、北；西、南；西、北）两个方向流的话，就能使新的可灌溉区域与目前的不重叠，如下图所示。



1. 此地图中 R=2
2. 左上角的数字表示河流流经的区域的序号；
3. 右下角的数字表明该格是属于哪个河流流经区域的“新的

易得动态规划方程：

$$f'_{i,j,k}=Max\begin{cases} Irrigate_{i-R,j-R,i+R,j+R,k} \\ f'_{i-1,j,l}+Irrigate_{i+R,j-R,i+R,j+R,k-l}\,(High_{i,j}>High_{i-1,j}) \\ f'_{i,j-1,l}+Irrigate_{i-R,j+R,i+R,j+R,k-l}\,(High_{i,j}>High_{i,j-1}) \end{cases}$$

这与上述求上界的动态规划十分类似，唯一的区别仅在于：方向只有两个，而求上界时有四个。其他三个流向（东、北；西、南；西、北）可依此类推。

实践证明，用上述方法求出的带限制条件的最优解很接近真正的最优解，大部分的随机数据都跟标准答案相同，对于所有的除样例以外的标准测试数据，用上述方法求出的下界刚好就等于上界。（因此，不搜索就可以直接输出答案了！）

至于具体如何搜索，本文不再作详细说明，请参考我的程序，有代码说明。

由于标准测试数据中，除了第 0 个点，都满足较优解=上界，因此我另外出了 10 个测试数据，n=20,m=100，并且都不满足这个条件。

程序效率：对于随机产生的数据，运行时间全在 0.1s 以下（测试环境：P4 1.7G），而且手出数据也很没有很慢的。

第3章　《L game》

——长沙雅礼中学　何林

【题目大意】

一个 4*4 的棋盘，放有两个 L 形和两个小圆球；其中一个 L 形被先手控制、另一个被后手控制。

两个人轮流走棋。一个人每一步可以将 L 形拿起然后重新放入棋盘中的任何新位置（不能放在原来的位置，可以翻转、旋转）；然后他可以选择任意一个小圆球移动到新的位置、也可以不移动小圆球，这由他决定。

一旦某人不能走了，就输了。输入一个状态，判断先手是必胜、必败还是平局。（假设两个人都足够聪明）

【解决情况】

O(E)的时间复杂度，E=1632800。

【算法梗概】

经典的博弈模型。将局面抽象成顶点，在可以一步转换的棋局之间连接有向边。

如果一个点必胜，必须满足：他至少有一个后继点是比败点。

如果一个点必败，必须满足：他所有的后继点都是必胜点；或者他根本没有后继点。

除了必胜和比败点就是平局。

本文采取一定的扩展顺序得到必胜、比败、平局的所有状态集合，然后根据输入数据逐个判断之。

【正文】

一、 博弈模型

对任意一个棋盘，我们规定画竖线的 L 形先走。

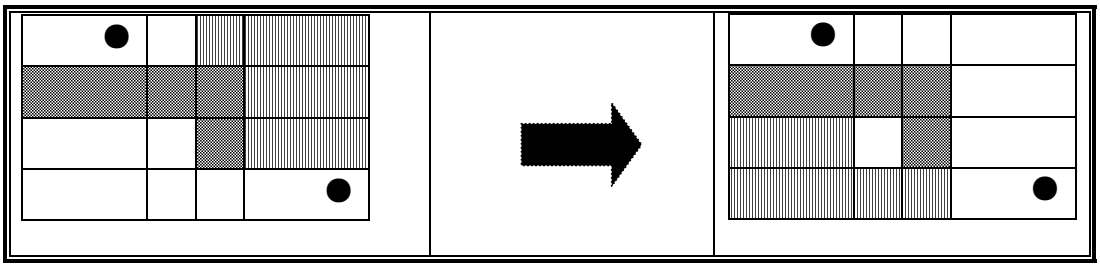


图　　图

图 1 中的状态能通过一步转化到图 2 的状态。但是由于规定画竖线的 L 形必须先走，因此我们略作修改：

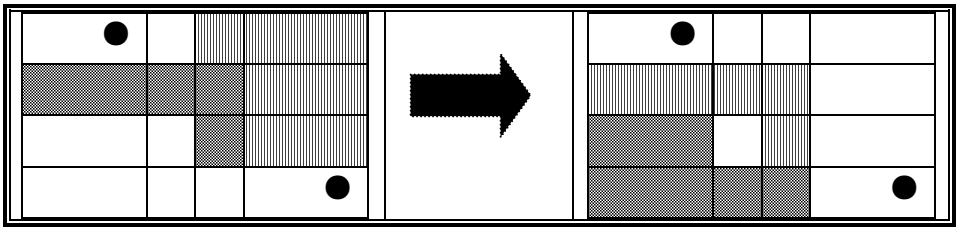


图　　图

称图 3 可“直达”图 4”。

我们把棋盘状态抽象成点，如果 A 棋盘可以直达 B 棋盘，就连一条有向边 A→B。现在考察某个点 V。

如果 V 的出度为 0，也就意味着先手无法做有效的移动，V 是一个先手必败策略。

假设 V 的出度为 d，这 d 个后继节点是 $u_1, u_2, \ldots, u_d$。当且仅当**存在** $u_i(1\leqslant i\leqslant d)$是先手必败节点，V 是一个先手必胜点。因为如果存在这样的点，处于 V 的先手可以把 u_i 这个必败的状态留给对手，他就胜利了。

假设 V 的出度为 d，这 d 个后继节点是 $u_1, u_2, \ldots, u_d$。当且仅当**每一个** $u_i(1\leqslant i\leqslant d)$都是先手必胜节点，V 是一个先手必败点。因为处于 V 的先手无论怎么走都会把一个必胜的状态留给对手，所以他必败。

因此我们设计如下算法：

构图

设置两个队列 Win 和 Lose，分别存储的是必胜状态和必败状态。一开始 Win={}，Lose={所有出度为 0 的点}

如果存在某个既不属于 Win 也不属于 Lose 的点 V，满足：V 的某个后继节点属于 Lose。则令 Win←Win+{V}

如果存在某个既不属于 Win 也不属于 Lose 的点 V，满足：V 的所有后继节点都属于 Win。则令 Lose←Lose+{V}

若第 3、4 步中存在这样的点 V，则转 3；否则转 6

令 Draw = 所有状态-Win-Lose

执行上述算法后，所有的必胜状态都保存在 Win 中；所有的必败状态都保存在 Lose 中；所有的平局都保存在 Draw 中。

Win 和 Lose 好理解，可是为什么除 Win 和 Lose 之外的状态就都是平局呢？

考虑某个点 V 属于 Draw。显然 V 的后继节点不可能是必败点，否则根据规则 V 就会被归到 Win 中；所以 V 的所有后继节点都是必胜或者 Draw 中的点，而且至少有一个 Draw 中的点（否则根据规则就会被归入到 Lose 中）。

如果你是先手，面对 V 这样的局面，你肯定不会把一个必胜的状态留给对手（因为这样你就输了！），所以你肯定会往一个 Draw 中的点走。

类似的，对手也不会把必胜留给你，也往一个 Draw 的点走……如是反复，你们两个就老在 Draw 中的点打转转。这样永远也走不完的棋局不就是平局吗？

所以我们可以肯定 Draw 中的状态都是平局。

根据上述算法求出 Win、Draw、Lose 后，只需根据输入的棋局判断其属于哪个集合即可确定答案。

具体实现

理论算法已经设计出来了，不过具体的实现还需要探讨。

先看程序的第三步：“**如果存在某个既不属于 Win 也不属于 Lose 的点 V，满足：V 的某个后继节点属于 Lose。则令 Win←Win+{V}**”。

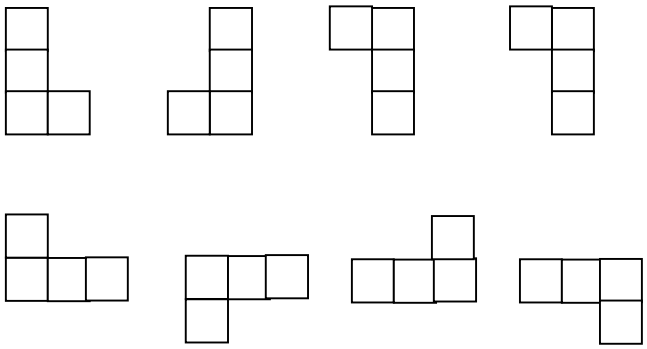
我们当然不需要真的去枚举所有的点 V，然后一个个的判断是否符合要求。只要每次更新 Lose 的时候（包括第二步中的更新），设 V 是即将加入 Lose 的点，我们就把所有指向 V 的点都放到 Win 中去。

再看第四步：“**如果存在某个既不属于 Win 也不属于 Lose 的点 V，满足：V 的所有后继节点都属于 Win。则令 Lose←Lose+{V}**”。

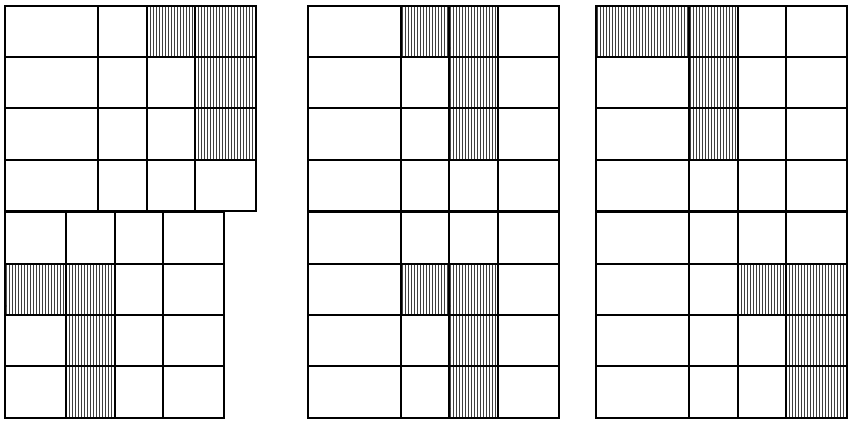
我们不需要也不可能去枚举每个 V 然后逐一判断。可以设 d[i]表示点 i 的出度；然后每次更新 Win 的时候，设 V 是即将加入 Win 的点，将所有指向 V 的点的出度减 1；一旦发现某个点的出度为 0 则将其加入 Lose。

然后还要考虑状态的判重。经过初步的搜索发现只有 18368 种不同的状态，完全可以用 Hash 表保存。我们要设法设计一种既简单、又高效的 Hash 函数。

L 形只有 8 种不同的形状，如下：



对于任意一种形状，在 4*4 的网格中有 6 种放置的方式。以某一个为例说明：



因此一个 L 形在棋盘中的状态就能用一个 1~8*6 的整数来表示，另外两个空格可以用 1~16*15/2 的整数表示。

于是一个棋盘就可以用一个三元组(L_1, L_2, Space)（$1 \le L_1, L_2 \le 48$，$1 \le Space \le 120$），L_1 表示先手的 L 形、L_2 表示后手的 L 形、Space 表示空格的位置。

我们可以开一个 array[1..48,1..48,1..120]的数组作为 Hash 表，这样存储、读取的复杂度为常数级别。空间上仅仅需要 270K 即可。

容易发现一个状态必然对应着一个三元组(L_1, L_2, Space)；但是反过来一个三元组(L_1, L_2, Space)却不见得对应一个合法的状态。因此实际上存在着大量的不可能利用空间。这实际上是一种以空间换时间的策略，在空间不太紧张的状况下是一种常用的策略。

注意以上几点就能较好的完成此题。

每条边都只遍历一次，故而此题复杂度是 O(E)。实践证明 E=1632800，因此运行速度非常快，任何数据都在 0.2s 左右出解。

【进一步研究方向】

本题对于任何一个输入都要算出所有的状态，很有可能实际上只要涉及很少的状态空间即可出解，是不是可以采取经典的博弈树呢？如果答案是肯定的那么如何处理平局的死循环？这都是可以进一步研究的。

第4章 《number》解题报告

长沙雅礼中学 何林

【题目大意】

若数列{ai | 1≤i≤n, 1≤ai≤n}满足：对于任意的 1≤i≤n-2，有 $a_i<a_{i+2}$ 对于任意的 1≤i≤n-3，有 $ai<a_{i+3}$ 则称此数列为“合法数列”。求合法数列的总数。

【解决情况】

时空复杂度均为 $O(n^2)$。所有的数据都非常快【算法梗概】

动态规划。

【正文】

设 f(h, min)表示这样的数列的总数：

数列长度为 h。令数列的元素为 $a_1, a_2, \ldots, a_h$。

$a_1, a_2 \geqslant min$

对于任意的 1≤i≤h-2，有 $a_i<a_{i+2}$

对于任意的 1≤i≤h-3，有 $ai<a_{i+3}$

分三类讨论：（这里假设 n 足够大）

第一类：$a_1, a_2 \geqslant min+1$。

这时的总数是 f(h, min+1)

第二类：$a_2 = min, a_1 \geqslant min$。

可以把 $a_3, a_4, \ldots, a_h$ 看作一个长度为 h-2 的新数列。必须满足：

$a_3, a_4>a1$

$a_4, a_5>a2$

对于任意的 3≤i≤h-2，有 $a_i<a_{i+2}$

对于任意的 3≤i≤h-3，有 $ai<a_{i+3}$

根据(1)、(3)和(4)可以得到：

$a5>a3>a1 \geqslant a2$

$a4>a1 \geqslant a2$

因此(2)可以根据(1), (3), (5)推得。故而可以化简为：

$a_3, a_4>a1$

对于任意的 3≤i≤h-2，有 $a_i<a_{i+2}$

对于任意的 3≤i≤h-3，有 $ai<a_{i+3}$

这样的数列有 $f(h-2, a_1+1)$个。

因此第二类情况中共有 $\sum_{k=\min}^{n} f(h-2,k+1)$ 个符合 f(h,min)定义的数列。

第三类：$a_2>min$，$a_1=min$。必有 $a_2>a_1$。将 $a_2, a_3, \ldots, a_h$ 看作一个长度为 h-1 的数列。必须满足：

$a_2, a_3, a_4 > a1$
对于任意的 $2 \le i \le h-2$，有 $a_i < a_{i+2}$
对于任意的 $2 \le i \le h-3$，有 $ai < a_{i+3}$
根据(2), (3)有：
$a_4 > a_2 > a_1$
于是(1)可化简为：$a_2, a_3 > a_1$，综合起来就是：
$a_2, a_3 > a1$
对于任意的 $2 \le i \le h-2$，有 $a_i < a_{i+2}$
对于任意的 $2 \le i \le h-3$，有 $ai < a_{i+3}$
这样的数列有 $f(h-1,a_1+1)=f(h-1,min+1)$个。

综合上述三类情况：

$$f(h,min)=f(h,min+1)+f(h-1,min+1)+\sum_{k=\min}^{n} f(h-2,k+1)$$

设 $s(h, min) = \sum_{k=\min}^{n} f(h-2,k+1)$，则：

$$\begin{cases} f(i,j) = f(i,j+1) + f(i-1,j+1) + s(i-2,j+1) \\ f(1,n) = 1,\ f(2,n) = 1,\ f(3..n,n) = 0 \\ f(1,i) = n-i+1,\ f(2,i) = (n-i+1)^2 \end{cases}$$

$$\begin{cases} s(i,j) = s(i,j+1) + f(i,j) \\ s(i,n) = f(i,n) \end{cases}$$

时间复杂度为 $O(n^2)$，空间复杂度是 O(n)。输出 f(n,1)即可。

第5章　《Jobs》解题报告

——长沙雅礼中学　何林

【题目大意】

有 n 件原料待加工。必须先经过 A 类机器加工成半成品，然后半成品经过 B 类机器加工成成品。总共有 m_1 台 A 类机器、m_2 台 B 类机器，每台机器加工一个原料（或者半成品）的时间是已知的。

现有两问：

求把所有的原料加工成半成品的最少时间。

求把所有的半成品加工成成品的最少时间。

【解决情况】

很好解决了。时间复杂度为 O(nm)，空间复杂度是 O(n+m)。

【算法梗概】

第一问好做，用简单的贪心。证明采用的是调整法。

第二问比较麻烦，关键是要逆向思维，把加工任务统一往后推，然后利用第一问的有关结论来解决。

【正文】

第一问

约定有 m 台 A 类机器，第 i 台机器的加工时间是 Ai，总共有 n 件原料待加工；并且 $A_1 \leqslant A_2 \leqslant A_3 \leqslant \cdots\cdots \leqslant A_m$。我们的任务就是要把这些原料全部分配到机器上面，使得尽可能早的完成加工。（机器是并行的）

假设已经有 k 件原料被分配到了机器上；第 i 台机器加工完它分配到的原料所需要的时间是 free[i]。接下来考虑如何分配第 k+1 件原料。

如果把第 k+1 件原料分配到第 i 台机器上面，那么第 i 台机器就要多花 Ai 的时间来加工它，令 free’[i]=free[i]+A_i。设{free’[1], free’[2], …, free’[m]}中的最小值是 free’[p]。

我们可以把第 k+1 件原料直接放到第 p 台机器上面，这个可以用反证法证明。

如果说把第 k+1 件原料放在第 p 台机器上面无法得到最优解，由于所有的原料都是等价的，所以以后任何原料都不能放在第 p 台机器。假设最优解中第 k+1 件原料放在第 t 台机器上（t≠p），并且第 t 台机器总共加工的时间为 f[t]、第 p 台机器总加工时间为 f[p]，则：

f[p]=free[p]

f[t]≥free’[t]

我们把第 k+1 件原料从第 t 台机器调整到第 p 台机器，则：

f’[p]=f[p]+A_p=free[p]+A_p=free’[p]

f’[t]=f[t]-A_t≤f[t]

因为 f’[p]=free’[p] ≤free’[t] ≤f[t]且 f’[t]≤f[t]，所以 max{f’[p],f’[t]}≤f[t]。调整后的方案不比最优方案差。

因此把第 k+1 件原料放到第 p 台机器上是可以得到最优解的。

于是我们轻松的得到了一个算法：

令 free[i]=0（i=1,2,…,m）

i←1

令 free'[j]←free[j]+A[j](j=1,2,3,…,m)

求{free'[1], free'[2], …, free'[m]}的最小值 free'[p]

令 free[p]←free'[p]

i←i+1。如果 i≤n 则转 3；否则继续执行 7

输出 max{free[1],free[2],…,free[m]}

算法的正确性前面已经作了说明。

第二问

第二问无疑是在第一问的基础上进行的。

设某个 A 机器加工方案 X 使得 n 件原料的出产时间是：$X=(x_1, x_2, \ldots, x_n)$（$x_1 \leqslant x_2 \leqslant x_3 \leqslant \cdots\cdots \leqslant x_n$）。第一问实际上就是要求一个 X，使得 x_n 最小。

假设 n=4，看两个加工结果：$X_1=(1,3,5)$, $X_2=(1,2,6)$。对第一问而言，无疑 X_1 要更优秀的，因为 X_1 只要 5 就能加工完、而 X_2 则需要 6。

可是如果作为第二问，考虑到还要放到 B 上面加工，我们要考虑的就不仅仅是 x_n 了，前面 $x_1,x_2,\ldots,x_{n-1}$ 的值都有可能对总的结果造成影响。我们定义：

设两个向量 n 维 $X=(x_1,x_2,\ldots,x_n)$，$Y=(y_1,y_2,\ldots,y_n)$，如果对于任意的 i=1,2,…,n 都满足 $x_i \leqslant y_i$，那么就说“X 优于 Y”，或者说 X≤Y。

我们之所以说“优于”，是因为如果按照 Y 能够得到最优方案，那么显然通过 X 也能得到最优方案。我们要证明的是：按照上一节的贪心算法得到的解 R 是所有加工方案中最优的。

证明：把 n 件原料 1~n 编号，我们人为的规定编号为 i 的原料必须是第 i 件加工出来的半成品。也就是说 A 类机器出产的顺序必须是 1,2,3,…,n。

设编号为 1..k-1 的原料均已经安排，用一个 m 维向量 $F=\{f_1, f_2, \ldots, f_m\}$代表这个时刻机器的状态，$f_i$ 表示第 i 台机器已经运转的时间。下面考虑编号为 k 的原料。

令 $f_i'=f_i+A_i$，设 f_p'是$\{f_1',f_2',\ldots,f_m'\}$的最小值。

如果把第 k 件原料分配给第 p 台机器，那么机器的新状态就是 $newF=\{f_1, f_2, \ldots, f_{p-1}, f_p+A_p, f_{p+1}, \ldots, f_m\}$，并且 $X_k=f_p+A_p$。

如果不把第 k 件原料分配给第 p 台机器，由于 f_p'是最小值，因此以后的原料都不能放到 p 上面加工（否则就会比第 k 件原料先出产，违反了我们的规定），我们可以令 f_p←+oo。不妨设第 k 件原料放到了第 t 台机器上 ($t \neq p$)，则机器新状态是 $newF'=(f_1,f_2,\ldots,f_{p-1},+oo,f_{p+1},\ldots,f_t+A_t,\ldots,fm)$，并且 $X_k'=f_t+A_t$。

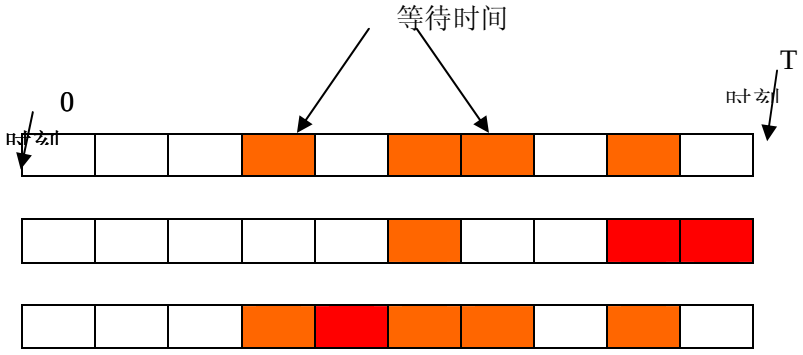
容易发现 newF<newF, $X_k \leqslant X_k'$。因此把第 k 件原料放到第 p 台机器可以使 X_k 最小；并且可以使得 newF（也就是新的机器状态）最小，这样就能保证以后的原料加工出来也是最优的。

证毕。

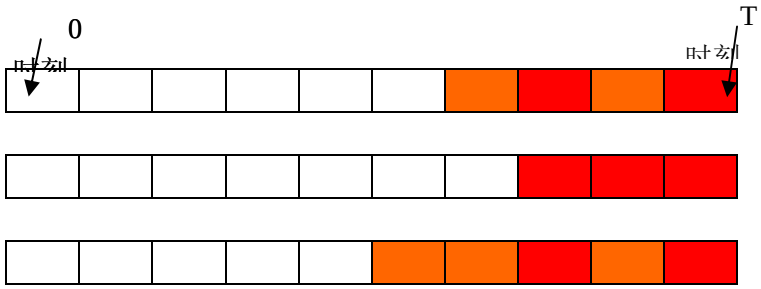
于是把原料加工到半成品应该采取什么策略，我们就可以按照上一节的贪心法完全确定下来了。设 n 件半成品出产的时间是 $X=(x_1, x_2, \ldots, x_n)$。（$x_n$ 就是第一问的答案）

关键是这些半成品还要放到 B 类机器上面加工。第一问中的原料是随叫随到的，也就是说任何时刻一台 A 机器都可以从仓库里面取出一件原料。

可是第二问就不同了。也许某台 B 机器需要半成品了，但 A 机器还没有加工出来。也就是说半成品的提供是断断续续的，这样就不能照搬类似第一问的贪心法。



以上表示 3 台 B 机器的加工情况，其中有很多等待时间；总加工时间是 T。我们可以把所有的加工任务往后提，变成如下形式：



这样调整后总加工时间不变，仍然是一个最优方案。

这样调整有什么好处呢？我们反过来看，把时间轴倒过来：T 时刻看作 0 时刻，0 时刻看作 T 时刻—那么在 B 类机器上的加工情况就可以看作是“反过来的”在 A 类机器上的加工情况！

因此我们可以把 B 类机器完全等价看作 A 类机器，根据上一节的贪心规则得到一组出产时间 $Y=(y_1, y_2, \ldots, y_n)$（$y_i$ 表示在 $T-y_i$ 时刻出产，$y_1 \leqslant y_2 \leqslant \cdots\cdots \leqslant y_n$）。显然这样的 Y 是最优的。

我们剩下的任务就是：找到一个合适的序列$(b_1, b_2, \ldots, b_n)$，$\{b_1, b_2, \ldots, b_n\} = \{1, 2, \ldots, n\}$，使得 $T=\max\{x_1+y_{b1}, x_2+y_{b2}, \ldots, x_n+y_{bn}\}$最小。

如果存在 $i<j$，且 $b_i<b_j$，则：

$x_i+y_{bj} \leqslant x_j+y_{bj}$

$x_j+y_{bi} \leqslant x_j+y_{bj}$

所以 $\max\{ x_i+y_{bj}, x_j+y_{bi}\} \leqslant \max\{ x_i+y_{bi}, x_j+y_{bj}\}$

也就是说调整之后的方案比原方案好。

因此：$b_1>b_2>b_3>\ldots>b_n$

故而第二问的答案就是：

$T=\max\{x_i+y_{n-i+1} \mid i=1,2,\ldots,n\}$

至此问题完全解决了。本题的解决逆向思维的一次巧妙应用。

第6章　《包裹运送》

长沙雅礼学校　何林

【题目大意】

一栋大厦有 10 层楼。现有 k 个包裹，第 k 个包裹在第 s_k 层，要求被运送到第 t_k 层去。

只有一个运货员运送这些包裹，并且他在同一时刻至多只能拿一个包裹。他从 1 层出发并且最后回到 1 层。

问：他最少要上多少层楼才能把所有的包裹运送到位？

【解决情况】

问题已经完全解决，采用的是构造欧拉图、然后求欧拉回路的算法。

算法时间复杂度为 O(k)，空间复杂度为 O(n+k)。

【算法梗概】

此题问的是最少上楼层数，然而我们发现：由于运货员要从第一层出发并且最终返回第一层，所以只要上了一层楼就必定对应的要下一层楼。于是问题可以转化为求总上下楼层数最少。

我们把每层楼看作一个顶点，将上下楼这样的活动看作边，构图。然后进行适当的修改，添加最少的边得到一个欧拉图；最后通过求欧拉回路来构造解。

【正文】

考虑现在如果有一个包裹 X 在 3 楼，要运到 5 楼去。

运货员必然要在某一步把 X 从 3 楼运到 4 楼，然后也许转而去运送其他的包裹；最后再返回 4 楼来把 X 从 4 楼运到 5 楼。

这个运货员拿着 X 经过了 3→4, 4→5 这样两个上楼过程。尽管这两个过程不一定是连在一起进行的，但“拿着 X 从 3 楼上到 4 楼”以及“拿着 X 从 4 楼上到 5 楼”这两个事件迟早都要发生。

我们如果把每一层楼看作一个顶点，上楼、下楼的事件看作边的话，对于上面的例子，就可以连两条有向边：3→4，4→5。更多的例子：

存在包裹要从 6 楼运送到 2 楼，则连边：6→5, 5→4, 4→3, 3→2。

存在包裹要从 1 楼运送到 4 楼，则连边：1→2, 2→3, 3→4。

按照以上规则，根据输入数据我们可以构出一个图 G_1。

G_1 中的每条边代表一个事件，这些都是必须事件，无论如何都要发生的，必不可少。

注意到题目要求：从一楼出发最后回到一楼。也就是运货员每上一层楼，就对应着要下一层楼。比如他某一时刻从 3→4，那么在之后的某个时候他肯定要从 4→3；不然怎么回 1 楼呢！这个规律我们称之为“上下楼平衡原则”。

也就是说 3→4 这样的弧的数目，应该等于 4→3 这样的弧的数目！一般的假设 i→i+1 有 p 条弧，i+1→i 有 q 条弧，则：

1、p>q。增加 p-q 条 i+1→i 的弧。

2、p<q。增加 q-p 条 i→i+1 的弧。

进行如此修改之后的图称之为 G_2。很显然 G_2 中的边也都是必不可少的边。

进一步考察一个例子：一个包裹，s_1=3, t_1=4。

运货员必须从第一层出发，而我们发现 3、4 两个点和 1 实际上是分开的—它们并不

连通！从一楼出发运货员迟早是要通过 3→4 的，为了达到这个目的，必然要先经过 1→2, 2→3；同时为了保持上下楼平衡，我们还要添加 3→2, 2→1。

一般的，我们把 i→i+1, i+1→i 的所有边统称为“i 楼道边”。如果存在“j 楼道边”，而不存在“i 楼道边”（i<j），那么显然就要添加 i→i+1 和 i+1→i 这两条边。（因为从 1 出发要到达 j 必然途径 i，因此 i 楼道至少要有一条边）这样添加之后就能保证所有的边都在一个连通分量里面

通过以上的诸多规则，我们构造出一个全新的图 G=(V,E)，|V|=10，|E|≤2k。根据我们的构造规则，很容易发现 G 中的边都是必须边，缺一不可。因此|E|/2 就是答案的下界。然而这个下界能不能达到呢？

先来看看 G 有什么性质：

G 的所有边在一个连同分量里面。

G 中每个点的出度等和入度相等。（根据上下楼平衡原则可知）

也就是说 G 是一个欧拉图！这样我们就可以把它“一笔画”，也就是不重复的遍历所有的边！这样一种遍历顺序（也就是一条欧拉回路）实际上就是一种送货的顺序，这个下界是可以达到的！

慢着……不要激动得太早。究竟能不能达到我们继续分析。

首先介绍一下欧拉回路的求法。从 s 点出发（s 是出发点），任意求一个回路 C，并且在原图中将 C 中的边删除。如果发现回路上某一点 p 的度不为 0，就再从 p 出发任意求一条回路 C’，并且将 C’从 p 的位置插入 C 中，形成新的 C 回路（也就是 C←C+C’）；删除 C’中的边。

继续检查 C 中的点，如果还存在 p 的度不为 0，则从 p 出发求一个回路……如是反复，直到 C 中所有点的度都为 0。此时的 C 就使一个欧拉回路了。

以上算法要从 p 点出发求一条回路，这条回路上的边是没有任何限制的。然而本题的边却有一定的限制。比如一个包裹 1→2, 2→3。在遍历 2→3 之前，必须先遍历 1→2，否则包裹还在 1 楼，你运什么上 3 楼啊！

正由于边有一定的顺序要求，所以此题又不是简单的欧拉回路。我的解决方法是“一走到底”。也就是说我求“从 p 出发的一条回路”的时候，一旦开始送某个包裹，那么我中途就不去抽空管其他的包裹，先把这个包裹的相关路径全部走完再说。这样就能保证遍历边的顺序性。

也就是说欧拉回路是可以求出来的，下界也是可以达到！问题也就解决了。

我们来回顾一下算法：

根据输入数据构图。

根据“上下楼平衡原则”添边。

为保持“图的连通性”添边。

“一走到底”求欧拉回路。

输出。

以上就是算法的全部内容，时间复杂度 O(k)，空间复杂度 O(n+k)。由于 n=10，k<=50，所以速度非常快。实际上完全可以把 n 和 k 的范围大大的扩大。

【进一步研究的方向】

增加人数。增加人数之后如果要求总上楼数最少，可以发现就是无论多少人结果都等于一个人干活的结果。因此可以考虑几个人并行工作，要求时间最少。

增加货物数。这里运货员只能提一个包裹。如果是两个呢？这样原算法中的边就不要那么多了，很多弧头、弧尾相等的弧都可以互相合并。

以上是两个值得继续研究的方向。

第7章　《桶的摆放》

清华附中　侯启明

【题目大意】

h 行 w 列的仓库中有 n 个位置固定的旧桶，要放进来 m 个可以任意摆放的新桶。所有的桶都在漏出强酸，选择桶泄漏的方向是东西向（腐蚀一整行）还是南北向（腐蚀一整列），使得被腐蚀的格子最少。

【解决情况】

时间复杂度 $O(2^n+h)$（可以优化到 $O(2^n+\min\{n+m,w,h\})$，但似乎价值不大），程序很好写，且没有放置新桶的贪心过程。

【算法梗概】

搜索+枚举。但很难加入剪枝。

【正文】

我曾经尝试从二分+可行性判定的角度入手研究这道题，虽然没有得到多项式级的算法，但得到了很多启示。

首先，我们应该确定二分的对象。为了方便，我选择的对象就是最后要求的解。显然，解必然是由某些行某些列组成的，肯定可以表示成 pw+qh-pq 的形式（其中 0<=p<=h，0<=q<=w）。由于 w、h 都不大，我们可以在多项式时间内列出所有可能解并将它们排序，然后据此二分+判定（二分似乎不太严密，因此可能应该是枚举+判定，但这一点与本文无关）。

那么，判定具体应该怎么做呢？首先，解大于等于 n+m；其次，解中被腐蚀的行和列应该包含所有的旧桶；最后，如果某个腐蚀方案满足上述条件，那么这个腐蚀方案就是一个可行解。这三点都是不难证明的。因此，对解 pw+qh-pq 的可行性判定实质上就是对下面两点的判定：

1．pw+qh-pq 是否大于等于 n+m。

2．能否用 p 行 q 列覆盖所有的旧桶。

第 1 点的判定显然是 O(1)的。但第 2 点的判定就很困难了，我至今没有找到多项式算法，所以我的这个想法夭折了，不过也从思考的过程中得出了一些有益的结论：

如果 p 行 q 列能够覆盖所有的旧桶：

1．如果在 p 行被腐蚀的前提下至少需要腐蚀 best[p]列才能覆盖所有的旧桶，那么 q≥best[p]。

显然成立，不需证明。

2．建立一个 h+w 个节点的二分图 G，如果在 a 行 b 列处有旧桶，那么连边(a,b+h)。则 p+q 大于等于 G 的最大匹配数 t。

如果 p 行 q 列能够覆盖所有的旧桶，那么在 G 中存在一个 p+q 个节点的支配集。因为 G 是二分图，所以 G 的最小支配集包含 t 个节点。故 p+q≥t。

（这条结论有些华而不实，我一直没有找到它的用途）

3．p+best[p]≤n

显然成立，不需证明。

虽然推出了这么多结论，但是我最终得到的算法只用了结论 1。算法如下：

1．搜索旧桶泄漏方向的所有可能情况，求出所有的 best[p]，其中(0≤p≤h)。这一步的复杂度是 $O(2^n)$的。

2．枚举所有的 0≤p≤h，对每一个 p，求出

q(p)=min{q|pw+qh-pq≥n+m 且 q≥best[p]}

这一步可在 O(h)内完成。

3．输出 min{pw+(h-p)q(p)| 0≤p≤h}（在上一步的时候顺便求出来的）。

根据前面的分析，算法的正确性和复杂度不难证明。实测发现对于最大数据可在 0.4 秒左右出解（TP7.0）。

【进一步研究的方向】

本题与二分图最小支配集问题的联系。可行性判定的多项式算法。

测试数据说明：

共 10 个点，有些可能 w、h 过小，有些可能 m 过大。

1．最优解不对应最小支配集。

2．最优解不对应没有新桶的最优解。

3．1 个旧桶，一堆新桶，而且 1 行放不下。

4．1 个旧桶，一堆新桶，w、h 悬殊。

5．贪心（每次选取度最大的点）求最小支配集的反例。

6．12 个旧桶，没有新桶。对任意的 1≤k≤11，只考虑前 k 个旧桶时和只考虑前 k+1 个旧桶时每个桶的最优泄漏方向都不相同。

7．20 个旧桶，小搜索量，没有新桶。

8．20 个旧桶，最大搜索量，没有新桶。

（下面数据中的 m 均超出原题范围，$O(m2^n)$的方法可能会超时）

9．20 个旧桶，最大搜索量，最优解不对应没有新桶的最优解。

10．20 个旧桶，最大搜索量，所有的桶刚好把仓库占满。