ACM 国际大学生程序设计竞赛
试题与解析

（一）

吴文虎　主编
倪兆中　王　帆　编著

清华大学出版社

(京)新登字 158 号

内 容 简 介

ACM 国际大学生程序设计竞赛是目前国际上历史最长、水平最高、影响最广泛的大学生计算机竞赛。本书精选了近年 ACM 总决赛中的 20 余道难题,对之加以分析,理出思路与解法,并给出作者编写的参考程序。这些试题具有实际背景,所考查的知识范围比较全面,题意新颖,给解题者留有广阔的思维空间和创新的余地,是高等学校大学生和研究生的很好的课外读物,从中可以学习如何用计算机编写程序解决难题的思路与算法。

版权所有,翻印必究。
本书封面贴有清华大学出版社激光防伪标签,无标签者不得销售。

图书在版编目(CIP)数据

ACM 国际大学生程序设计竞赛试题与解析 (1)/ 吴文虎主编;倪兆中,王帆编著 . - 北京:清华大学出版社,1998.12
ISBN 7-302-03212-2

Ⅰ.A… Ⅱ.①吴… ②倪… ③王… Ⅲ. 程序设计-解题 Ⅳ.TP311

中国版本图书馆 CIP 数据核字(98)第 33078 号

出版者: 清华大学出版社(北京清华大学校内,邮编 100084)
http:// www .tup .tsinghua .edu .cn
印刷者: 昌平环球印刷厂
发行者: 新华书店总店北京发行所
开 本: 787×1092 1/ 16 印张: 10 .5 字数: 245 千字
版 次: 1998 年 12 月第 1 版 1998 年 12 月第 1 次印刷
书 号: ISBN 7-302-03212-2/ TP·1716
印 数: 0001～5000
定 价: 12.00 元

序

ACM国际大学生程序设计竞赛是目前世界上规模最大的计算机学科赛事。该项赛事发起于1977年,要比国际信息学奥林匹克(IOI)的首届比赛早12年。ACM(Association of Computing Machinery,美国计算机协会)是国际计算机界的权威学术机构。由ACM发起并组织的这项赛事的宗旨是:为高等学校的大学生们提供一个展示自己在计算机编程解题方面才能的机会。通过竞赛使各国学生相互交流学习经验,为信息学科的发展不断注入新的活力,增进友谊,促进合作。因此,这项一年一届的赛事吸引了几乎所有的知名高等学府。譬如,1998年的第22届大赛就有1250支代表队参赛,其规模与影响之大可见一斑。

该项赛事的题目涉及计算机在各种应用领域中提炼出来的一些理论问题、方法问题,特别是有些问题没有固定算法,需要选手在比赛现场运用所学基础知识,经过认真分析研究归纳整理,才能得出解决问题的办法,此后还要经过编程调试、提交通过等严格的步骤加以检验,因此有相当难度,要想获奖是十分不易的。参加这样的竞赛要具备高等数学、图论集合论、组合数学、高级编程语言、人工智能和算法等方面的基本知识,要有熟练的编程解题功夫,特别是要有将实际问题抽象为数学模型的能力;在心态方面要有不怕困难、不畏强手、敢于拚搏的精神;特别强调要有集体协同作战的团队精神。比赛是三人一组解若干道难题,比谁做得快做得好,可以说是争分夺秒,讲究配合默契,强调协作攻关。

这种比赛形式有其独到的特点,在培养学生的创新能力方面能够起到促进作用。我们在准备参加ACM计算机竞赛的过程中收集了国内外有关的竞赛试题,是历年ACM世界大学生竞赛的题目,对这些题目进行了较详细的分析并做出解答,由参加过ACM大赛的两位选手将其汇集成册,正式出版,目的是将我们的经验体会还有一些技巧介绍给对竞赛有兴趣的同学。当然,也许你从来也没想去参加世界大赛,但我们相信,你看了这套试题解析也会有所收获,起码可以学到使用计算机编程分析问题到解决问题的全过程。因此,本书又可以作为学习高级编程语言和算法的参考书。

吴文虎

1998年9月

前　言

ACM 国际大学生程序设计竞赛(以下简称 ACM/ ICPC)是目前国际上历史最悠久、水平最高、影响最为广泛的大学生计算机竞赛,每年都吸引了成千上万来自世界各地的大学生参加。

ACM/ ICPC 的试题具有以下特点:

1. 试题具有丰富的实际背景,趣味性和实用性较强。许多试题都是从计算机软硬件的实际开发过程中演变而来的。

2. 试题所考查的知识范围较为全面。不仅要求解题者具备高级语言程序设计、数据结构和算法设计等计算机基础知识,还需要熟练掌握人工智能、计算几何等计算机科学的深层次内容,特别是要求具有较强的数学功底和广博的知识面。

3. 试题的层次性较好。不同水平的解题者都能找到适合于自己的试题。

4. 试题灵活、新颖。绝大部分试题没有定解,给解题者留出了广阔的思维空间,对创造性的培养十分有益。

参加 ACM/ ICPC 本身就是一个增长知识、培养能力的绝好机会,这也正是其魅力所在。而在 ACM 竞赛过程中所体现出的团队精神和合作精神,也正是当代大学生的必修课。

国内大学参加 ACM/ ICPC 的历史相对较短,但在各大学师生的努力下,近年来还是取得了长足的进步,一些起步较早的大学也先后达到了世界级的水平。然而必须看到的是,就整体而言,国内大学的水平与世界一流相比,还存在着很大的差距。同时,国内有关这方面的专著和资料也极为缺乏,这同大学生计算机竞赛正在国内蓬勃兴起的现状是不相适应的。本书正是基于这样一种考虑来组织编写的。

本书首先用两章的篇幅,分别向读者介绍了 Borland Pascal 高级程序设计技术和 ACM/ ICPC 的基本概况,作为读者阅读本书的基础。在之后的六章中,精选了近年来 ACM/ ICPC 总决赛的 20 余道试题,将其翻译为中文,并编写了相应的解题思路和参考程序,供读者参考。

本书的编写是在中国计算机学会普及工作委员会主任、清华大学计算机系吴文虎教授的指导下完成的。他亲自参与了本书的选题与规划,并仔细审阅和修改了本书的初稿,确保了本书编写任务高质量地完成。ACM 在国内的分支机构——Tsinghua University ACM Student Chapter——对本书的编写给予了极大的支持和帮助。

希望本书的出版能起到抛砖引玉的作用,促进全国范围内大学生计算机竞赛水平的提高。随着 ACM 国际大学生程序设计竞赛的发展,我们还将继续编写《ACM 国际大学生程序设计竞赛试题与解析(二)》。必须指出的是,虽然编著者在本书的编写过程中努力工作,力图奉献给读者高质量的精品,但由于水平有限,书中的不足和误值之处在所难免,恳请广大读者提出宝贵意见。

编著者

1998 年 5 月于清华园

目　　录

第 1 章　专题讲座——Borland Pascal 高级程序设计技术

1.1　引　　言

首先来明确一下程序设计竞赛对选手的要求:在有限的时间内,用自己所掌握的知识与具备的能力,凭借可能的工具,通过编写程序,去有效地解决问题。

解决程序设计竞赛试题的过程应当是顺着这样一条路线行进的:

理解题意并建立相应的数学模型
↓
根据数学模型的特点选用或构造适当的数据结构及算法
↓
将算法实现为程序并调试正确

在这样一个过程中,算法是首要的,占有战略性的地位。算法的好坏从根本上区分了选手的优劣。

但是,在实际竞赛中,往往会看到这样的情况:一个选手在比赛时找出或构造出了正确的算法,但由于时间不够,造成程序没有编完或没有编正确。测试之后,同不会做的选手一样,没有分数;两个选手,想到的算法大致一样,而编程实现所用的时间却有快有慢,测试时,程序的效率也相去甚远。以上都是非常普遍的现象。

为什么会出现这样的情况呢?一般地,这是由于选手没有熟练、扎实、深入的基本功所致。具体体现在平时对编程环境不够了解,运用不够熟练,忽视了将算法实现到程序这一环节的必要性和重要性。

我们说算法是首要的。但是,一个算法再好,如果不能在限定的时间内转化为正确的程序,那么对于竞赛来说,其效果就是零。这就是算法实现的必要性。与此同时,算法实现的快慢,决定了选手的时间利用率,决定了选手能否有时间去解决其它试题,因而也就影响了选手的成绩;算法实现的好坏,决定了算法的表现,对选手的成绩也有不小的影响,这主要体现在程序的运行时间上。在有多人想到了正确算法的情况下,算法实现的质量就尤其显得重要了。这就是算法实现的重要性。

算法与算法实现的关系,是战略与战术的关系。算法起决定性作用,从根本上决定了算法实现质量的数量级,但算法实现反过来也影响着算法,并且在一定条件下,这种影响有可能超过算法本身的作用。

在算法实现上,“时—空”这一对对立统一的矛盾起着很大作用。算法可以有多种不同的实现,这些实现有的侧重于时间,有的侧重于空间。一般地,程序设计竞赛对程序的速度要求较高,而空间占用只要不溢出,就对成绩没什么影响,因此选手应多注意程序的

时间效率。当然这里有一个前提,那就是空间占用不能超过允许的范围。

Borland Pascal 是标准 Pascal 的超集,它向程序员提供了数以千计的数据类型、语法结构、标准函数与子程序等语言成分,其中绝大多数是标准 Pascal 所没有的。这些语言扩展向程序员提供了强大有力的开发支持,运用得当的话,可以起到事半功倍的效果。正是基于此,无论国内或国际的程序设计竞赛,都提供 Borland Pascal 的最新版本 7.0 ,并建议选手采用。然而目前很少有资料详细、全面、系统地介绍了这些高级技巧及其应用。我们根据自己多年来运用 Borland Pascal 语言工具解决大量竞赛题的体会和经验,对Borland Pascal 7 .0 版本作了较深入的剖析,并从功能、效率、可读性等方面对其语言概念作了大致的总结、归纳与分析,从中选择出了“ 数据类型 ”、“ 常用函数 ”、“ 指针及堆 ”、“ 文件缓冲 ”、“ 快速操作 ”、“ 位操作 ”、“ 编译开关 ”、“ 保护模式 ”等一些对程序设计竞赛较为有用的语言扩展,供读者参考。

1.2 数 据 类 型

Borland Pascal 实现了标准 Pascal 所有的预定义数据类型,并扩充了不少十分有用的数据类型。在 Borland Pascal 中,这些数据类型得到了高效的实现。对数据类型的深入了解将有助于程序设计水平的提高。

1. 整数

标准 Pascal 里定义的整数类型是 Integer, Borland Pascal 还提供了 Shortint、Byte、Word、Longint 这四种整数类型。表 1.1 是它们的详细说明。

表 1.1

类型	大小	范围	存储格式	预定义最大值常量
Shortint	1 字节	- 128 . .127	带符号 8 位	无
Integer	2 字节	- 32768 . .32767	带符号 16 位	MaxInt
Longint	4 字节	- 2147483648 . .2147483647	带符号 32 位	MaxLongInt
Byte	1 字节	0 . .255	无符号 8 位	无
Word	2 字节	0 . .65535	无符号 16 位	无

2. 实数

标准 Pascal 里定义的实数类型是 Real, Borland Pascal 另外定义了 Single, Double, Extended, Comp 四种实数类型。表 1.2 是它们的详细说明。

Borland Pascal 7 .0 的集成环境有两个。turbo .exe 只能在实模式下编译,通常又称为 Turbo Pascal 7 .0; bp .exe 可以在三个目标模式下编译,是真正完备的 Borland Pascal 7 .0。两者的用户界面、语言环境都是一致的。

表 1.2

类型	范围	精度	大小	是否要求协处理器
Real	2.9e-39..1.7e38	1112 位	6 字节	否
Single	1.5e-45..3.4e38	78 位	4 字节	是
Double	5.0e-324..1.7e308	1516 位	8 字节	是
Extended	3.4e-4932..1.1e4932	1920 位	10 字节	是
Comp	-9.2e18..9.2e18	1920 位	8 字节	是

3. 字符

Borland Pascal 中字符的大小是一个字节,字节的值是该字符的 ASCII 码值。在程序中除了可以用' * '的形式表示可显示字符外,还可以用 # 号后跟上该字符的 ASCII 码值来表示任意一个字符。例如, # 0 表示第一个 ASCII 字符, # 9 表示制表符等。

4. 布尔类型

布尔类型在 Borland Pascal 中也用一个字节存储,但实际上只有字节的最低位被用来表示信息。最低位为 0 时表示 false,最低位为 1 时表示 true。

5. 集合

Borland Pascal 的集合类型最多允许有 256 个元素。在实现时,采用了一个字节表示 8 个元素的存储方式。故集合类型存储较布尔数组存储在空间上要节省一些。

6. 字符串

Borland Pascal 定义了字符串类型 string。它的存储结构如图 1.1 所示。

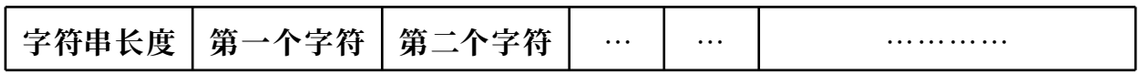

图 1.1

string 类型实际上是用 array[0..255] of char 来定义的。数组的第 0 位表示字符串的长度。因此,字符串的长度最大为 255。

1.3 常用函数

1. ReadKey 和 KeyPressed

Crt 单元提供了函数 ReadKey,其作用是从键盘读入一个字符。ReadKey 没有参数,返回值是一个键盘输入的字符。在读入字符的同时,ReadKey 不像 Read/ ReadLn 那样在屏幕上显示读入的字符。

Crt 单元还提供了函数 KeyPressed,其作用是确定键盘上有无按键动作。KeyPressed

也没有参数。返回值是一个布尔量,返回值为真时表示有按键,反之则无。KeyPressed 不会像 Read/ ReadLn/ ReadKey 那样等待键盘的输入,所以在程序中可以随时用 KeyPressed 来判断有无键盘输入。

结合使用 ReadKey 和 KeyPressed,可以做到在键盘输入的同时"并行"其它操作。

【例程】

下面的程序不断地向屏幕输出"Press any key . . .'信息,直到用户按下任一键后输出用户所按的键。

```
program Demo_of_ReadKey_and_KeyPressed;

  uses
    Crt;

  begin
    repeat
      writeln('Press any key . . .');
    until keypressed;

    writeln('You have pressed the key of', readkey);
  end.
```

2. SizeOf

SizeOf 是一个函数,由其可以得到类型的一个实例的大小。SizeOf 有一个参数,可以是任何类型、变量。SizeOf 返回参数所占内存的字节数。

【例程】

下面的程序输出了 People 类型的大小以及其实例 P 的大小。

```
program Demo_of_SizeOf;

  type
    People =
      record
        Name:string[20];
        Age:byte;
      end;

  var
    P:People;

  begin
    writeln('Size of People = ',sizeof(People));
```

```
  writeln('Size of P = ',sizeof(P));
end.
```

1.4 程序结构

在这一节里将介绍 Borland Pascal 相对于标准 Pascal 在程序结构控制上的几个主要扩展,通过合理使用这些扩展,能够使程序结构简洁、高效、易读。

1. Break 和 Continue

系统提供了 Break 过程,其作用是跳出当前循环。Break 没有参数,调用它以后将立即跳出当前的 for,while,repeat 三种循环结构,转而执行紧接着此循环体的下一个语句。如果对 Break 语句的调用不在一个循环体内,编译器将报告错误。

系统还提供了 Continue 过程,其作用是进入下一次循环。Continue 也没有参数。调用它以后将直接跳转至当前的 for,while,repeat 三种循环结构的入口,进行下一次循环。如果对 Continue 语句的调用不在一个循环体内,编译器将报告错误。

读者可以看出,Break 和 Continue 这两个标准过程,都是同循环结构的控制有关的。Break 是跳至循环结束后的下一个语句;Continue 是跳至循环入口的第一个语句。用 goto 语句也可以实现 Break 和 Continue 的功能。但读者如果试一下,就会发现用 goto 语句实现 Break/Continue 功能编程繁琐,极易引起混乱,至于可读性就更无法同 Break 和 Continue相比了。

【例程】

下面的程序从键盘输入一个结点范围在 1 到 100 之间的有向图的弧。用户每次输入弧的始、末结点。输入一个 0 表示输入结束,其它情况将被忽略。之后程序将输出图中所有的弧。

```
program Demo_of_Break_and_Continue;

  var
    G: array[1..100,1..100] of boolean;
    A, B:byte;

  begin
    fillchar(G,sizeof(G),0);

    repeat
      write('Please input a number: ');
      read(A);

      case A of
        0:
```

```
            begin
              readln;
              break;
            end;

          1..100:
            begin
              readln(B);
              if not(B in [1..100]) then
                continue;

              G[A,B]:= true;
            end;

          else
            begin
              readln;
              continue;
            end;
        end;
      until false;

      for A:= 1 to 100 do
        for B:= 1 to 100 do
          if G[A,B] then
            writeln(A,'-',B);
    end.
```

2. Exit 和 Halt

Exit 是一个过程,其作用是退出当前模块。Exit 没有参数,调用它以后将退出当前子程序的运行。如果当前正在主程序里运行,则将终止整个程序的运行。

Halt 也是一个过程,其作用是终止整个程序的运行。Halt 的格式声明如下:

```
procedure Halt([ExitCode:Word]);
```

这里 ExitCode 表示程序退出至操作系统的返回代码,是可选的,其缺省值为 0(正常退出)。在程序中的任何地方调用 Halt 都将引起整个程序的终止并返回操作系统。

Exit 和 Halt 都具有中断运行的功能,但有很大差异。在子程序里使用时,Exit 会引起返回上一级程序,而 Halt 则会引起整个程序的终止。在主程序里使用时,虽然两者都会引起整个程序的终止,但是用 Exit 终止程序运行只能返回正常退出代码,而使用 Halt 则可以返回其它情况下的退出代码。

【例程】

下面的程序演示了 Exit 和 Halt 的作用。

```
program Demo_of_Exit_and_Halt;

  procedure Proc;

    begin
      writeln('Now in Proc.');
      exit;
      writeln('You will not see this.');
    end;

  begin
    writeln('Now in Main.');
    Proc;
    writeln('Now in Main Again.');
    halt(100);
    writeln('You will not see this.');
  end.
```

1.5 指针及堆

1. Borland Pascal 程序的存储机制

首先来介绍一下 Borland Pascal 程序运行时的存储机制。一个用 Borland Pascal 编译运行的程序在运行时将可用内存划分为以下几个部分:

代码段——存放程序代码;

堆栈段——存放程序运行过程中子程序的参数、局部变量;

常量段——存放程序中出现的常量;

数据段——存放在程序中已声明的变量;

堆——存放程序运行过程中动态声明的变量(通过指针引用)。

堆栈段、常量段、数据段的大小都不得超过 64K,因此静态变量的大小受到很大限制。而动态数据操作灵活,并且通过堆可以用到所有的可用内存,因而有着广泛的应用。程序运行时,通过系统所提供的过程,可以在堆中动态地分配和释放内存块。

必须指出的是,动态数据的操作速度比静态数据大约要慢三分之一左右。

2. Pointer

Borland Pascal 的预定义类型中有一个叫做无类型指针的类型,名称是 Pointer。定义这样一种类型的原因是因为在标准 Pascal 中,指针是依附于它所指向的具体类型的,不同

类型的指针之间是不兼容的。无类型指针则可以指向任何变量,它可以同任何类型指针互相赋值。

3. GetMem 和 FreeMem

系统提供了过程 GetMem 和 FreeMem,它们的作用是分配和释放动态存储空间(堆空间)。GetMem 和 FreeMem 的格式声明如下:

```
procedure GetMem(var P:Pointer;Size:Word);
procedure FreeMem(var P:Pointer;Size:Word);
```

这里 P 是要进行操作的指针,Size 是要分配或释放的存储空间的字节数。GetMem 为指针 P 分配一块大小为 Size 的堆空间,而 FreeMem 正好是 GetMem 的逆操作:它将指针 P 所指向的大小为 Size 的堆空间释放以供其它指针使用。

GetMem 和 FreeMem 的例程见下一节。

4. Mark 和 Release

系统还提供了另一对关于动态存储空间分配的过程 Mark 和 Release,它们的作用是记录和恢复堆的使用情况。Mark 和 Release 的格式声明如下:

```
procedure Mark(var p:pointer);
procedure Release(var p:pointer);
```

指针 p 被用来保存堆使用情况。Mark 将堆顶指针保存到 p 中,从而记录堆空间的使用情况。Release 将堆顶指针设置为 p 的值,从而使堆空间的使用情况恢复到上一次的 Mark(p)。结合使用 Mark 和 Release,可以方便、彻底、安全地释放堆空间。

【例程】

下面的两个程序是等价的,后一个程序演示了 Mark 和 Release 的用法。

```
program Prog_for_Comparation;

  var
    p1,p2,p3:^string;

  begin
    new(p1);
    new(p2);
    new(p3);
    dispose(p2);
    dispose(p3);
  end.

program Demo_of_Mark_and_Release;
```

```
var
  p:pointer;
  p1,p2,p3:^string;

begin
  new(p1);
  mark(p);
  new(p2);
  new(p3);
  release(p);
end.
```

1.6 文件缓冲

1. 缓冲的原理

程序对磁盘的读写过程是:

程序\ 操作系统\ 磁盘

磁盘尤其是硬磁盘的数据传输速度是很快的,但磁头寻道的时间则相对要慢得多,因此,对磁盘的读写时间主要是花在磁头定位上,真正用来传输所需要的数据的时间不到百分之一。尤其是在读入文本文件时,往往是以几个字节大小的数字或字符为单位进行读写,一个几十 K 的文件,磁头往往要做上万次的寻道动作,所以时间浪费极大。

读者也许知道 DOS 里的 Smartdrv 这个应用程序。它的作用是提高磁盘读写的速度。其原理是在操作系统和磁盘之间开辟一个数据缓冲区。每次读磁盘时先查看数据是否在缓冲区中,如果在就直接将其复制至指定内存区域,仅当要读入的数据不在缓冲区里时才去读磁盘,并且一次要读入尽可能多的数据,以便提高以后的命中率。每次写磁盘时并不是将数据直接写入磁盘,而是先送往缓冲区累积起来,当缓冲区满后才一次将缓冲区信息全部写入磁盘。通过缓冲技术,Smartdrv 大大减少了磁盘的读写次数,从而节省了大量的空间。

有系统缓冲的磁盘读写过程是:

程序\ 操作系统\ 缓冲区\ 磁盘

2. SetTextBuf

在竞赛中,有一些试题需要对磁盘进行大量的读写,这时如果能有高速缓存,可以极大地提高程序的运行效率。然而,选手无权要求系统安装 Smartdrv,那么应怎么办呢?Borland Pascal 向程序员提供了设置文本文件自动内部高速缓存的过程——SetTextBuf。

有文本文件缓冲的磁盘读写过程是:

程序\ 缓冲区\ 操作系统\ 磁盘

SetTextBuf 的格式声明如下:

SetTextBuf(var F: Text; var Buf; Size: Word);

其中 F 是要被高速缓存的文件,Buf 是缓冲区变量,Size 是缓冲区的大小。

【例程】

下面的程序利用文本文件缓冲读入文本文件 Input .Txt 到屏幕。

```
program Demo_of_SetTextBuf;

  const
    BufSize = 16384;

  var
    P: pointer;
    F: text;
    S: string;

  begin
    assign(F, 'Input .Txt');
    getmem(P,BufSize);
    settextbuf(F,P^,BufSize);
    reset(F);

    while not eof(F) do
      begin
        readln(F,S);
        writeln(S);
      end;

    close(F);
    freemem(P,BufSize);
  end.
```

1.7 快速操作

为了提高程序的效率,Borland Pascal 提供了一些快速操作的过程。由于这些过程在实现中充分利用了汇编语言的特性,因此效率较普通的实现方法要高出不少。合理使用这些过程可以使程序的效率提高不少,编程也有一定的简化。

1. Fillchar

系统提供了过程 Fillchar,其作用是将一个给定的值写入给定长度的连续内存区域中。FillChar 的格式声明如下:

```
procedure FillChar(var X;Count: Word; value);
```

这里 X 是要写入的内存区域;Count 是要写入的字节数(如果 X 是变量,建议在这里使用 SizeOf(X));value 是写入的值,可以是 byte 类型或 Char 类型。FillChar 在实际应用中主要用来为数组或结构变量初始化清零或统一赋值等。

【例程】

下面的程序从键盘读入一个数值,然后将数组全部初始化为该值。

```
program Demo_of_FillChar;

  var
    X:array[1..1024] of byte;
    Value:byte;

  begin
    readln(Value);
    fillchar(X,sizeof(X), Value);
  end.
```

2. Inc 和 Dec

系统提供了过程 Inc 和 Dec。它们作用是对一个整型或有序类型变量作增量和减量操作。Inc 和 Dec 的格式声明如下:

```
procedure Inc(var X[;N:Longint]);
procedure Dec(var X[;N:Longint]);
```

Inc 是将变量 X 加上 N;Dec 是将变量 X 减去 N。N 的缺省值为 1。

从功能来看,如果 X 是整数,则 Inc(X,N)等价于 X:= X + N;Dec(X,N)等价于 X:= X - N。如果 X 是有序类型,则 Inc(X,N)等价于 X:= Succ(X);Dec(X,N)等价于 X:= Pred(X)。

之所以要有 Inc 和 Dec,是由于它们所生成的代码比较高效。这是因为系统在汇编表达式时,对于形如 A:= B+C 的表达式所生成的典型的代码是:

载入 B 的值到寄存器中;

将 C 的值加到寄存器中;

将寄存器的值保存到 A 中。

而对于 A:= A+B 类型的表达式,代码可以简化为:

载入 B 的值到寄存器中

将寄存器的值加到 A 中

Inc 和 Dec 的参数 N 都可以是负数。

这里提醒读者一点:Inc 和 Dec 不会对操作进行溢出检查。

【例程】

下面的程序从键盘读入一个整数 n,计算 n + (n - 1) + (n - 2) + … + 1 的值。

```
program Demo_of_Inc_and_Dec;

  var
    n,s:integer;

  begin
    readln(n);
    s:= 0;

    while 0 < n do
      begin
        inc(s,n);
        dec(n);
      end;

    writeln(n);
  end.
```

3. Include 和 Exclude

系统提供了过程 Include 和 Exlcude。它们的作用是对一个集合加入或取出一个元素。Include 和 Exclude 的格式声明如下:

```
procedure Include(var S: set of T;I:T);
procedure Exclude(var S: set of T;I:T);
```

这里 T 是集合的基类型,S 是要操作的集合,I 是要加入或取出的元素。Include 是将元素 I 加入集合 S;Exclude 是将元素 I 从集合 S 中取出。

从功能来看,Include(S,I)等价于 S:= S + [I], Exclude(S,I)等价于 S:= S - [I]。

由于集合较大,一般无法在寄存器中存下所有的元素,所以作集合加减法时一般要反复在内存和寄存器来回传送数据,而 Include 和 Exclude 只需要将 S 集合中表示元素 I 的那一位置为 0 或 1 即可。因此 Include 和 Exclude 较之一般的集合加减法要高效得多。

【例程】

下面的程序从键盘读入两行字符,输出在第一行中出现而未在第二行中出现的所有大写英文字母。

```
program Demo_of_Include_and_Exclude;

  var
    S:set of 'A'..'Z';
    I:char;
```

```
begin
  S:= [];

  while not eoln do
    begin
      read(I);
      include(S,I);
    end;
  readln;
  while not eoln do
    begin
      read(I);
      exclude(S,I);
    end;
  readln;

  for I:= 'A' to 'Z' do
    if I in S then
      write(I);
  writeln;
end.
```

1.8 位 操 作

1. 位操作

在标准 Pascal 中没有提供二进制操作。Borland Pascal 弥补了这一不足,提供了位操作运算符 not, and, or, xor。表 1.3 解释了它们的意义。

表 1.3

运算符	运算
not	按位取反
and	按位与
or	按位或
xor	按位异或

应用位操作运算可以实现对变量整体的二进制操作,但无法直接访问单个的二进制位。一种常用的解决办法是预定义每一个二进制位的掩码。例如,要操作的数据单位是字节,就应定义掩码如下:

```
const
  Bits: array[0..7] of byte = ($01, $02, $04, $08,
  $10, $20, $40, $80);
```

记待操作变量为 A, A_i 表示 A 的第 i 位。

如果要得到 A_i 的值,可以用表达式 A and Bits[i] > 0 来进行判断。表达式为真时 A_i 为 1,否则为 0。

如果要将 A_i 置为 0,可以用赋值语句 A:= A and ($ FF - Bits[i])。

如果要将 A_i 置为 1,可以用赋值语句 A:= A or Bits[i]。

如果要将 A_i 取反,可以用赋值语句 A:= A xor Bits[i]。

2. 压缩存储

位操作运算的一个很大的用处是数据的“压缩存储”。

在许多问题中,常常只需保存 0 或 1 两种状态信息。但用普通的内存分配办法时,内存的最小单位是字节。这时如果数据量较大而无法在内存中存放,可以考虑采用一个字节存放 8 个数据单位。这样就可以使内存“扩大”8 倍,从而使程序的求解能力得到提高而无须修改算法。在实战中,适当地使用“压缩存储”的意义是显而易见的。

“压缩存储”也有其一定的局限性。由于访问一个二进制位就要进行至少一次运算,所以其速度比较慢。尤其是在需要频繁访问单个二进制位的情况下,“压缩存储”的效率还是需要注意一下。不过从另一方面来看,如果大量的操作是多个连续的二进制位的操作(例如集合的交、并操作),则“压缩存储”的效率较普通的存储方法反而要高出不少。

【例程】

下面的程序用“压缩存储”实现了自定义“大集合”数据类型。

```
program Demo_of_Storage_Compress;

  const
    ElementCount = 1023;
    StorageCount = ElementCount div 16;

    Bits:array[0 ..15] of word = ( $0001, $0002, $0004, $0008,
                                   $0010, $0020, $0040, $0080,
                                   $0100, $0200, $0400, $0800,
                                   $1000, $2000, $4000, $8000);
  type
    Tset = array[0 .. StorageCount] of word;

  function MyIn(E:word;S:TSet):boolean;
    begin
      MyIn:= S[E div 16] and Bits[E mod 16] > 0;
    end;

  procedure MyAssign(Sour:TSet;var Dest:TSet);
    begin
      Dest:= Sour;
    end;

  procedure MyInclude(var S:TSet; E:word);
```

```
  begin
    S[E div 16] := S[E div 16] or Bits[E mod 16];
  end;

procedure MyExclude(var S:TSet; E:word);
  begin
    S[E div 16] := S[E div 16] and ($FFFF - Bits[E mod 16]);
  end;

procedure MyUnion(A:TSet;var B:TSet);
  var
    I:word;

  begin
    for I := 0 to StorageCount do
      B[I] := B[I] or A[I];
  end;

procedure MyIntersection(A:TSet;var B:TSet);

  var
    I:word;

  begin
    for I := 0 to StorageCount do
      B[I] := B[I] and A[I];
  end;

procedure MyDifference(A:TSet;var B:TSet);

  var
    I:word;

  begin
    for I := 0 to StorageCount do
      B[I] := B[I] and ($FFFF - A[I]);
  end;

var
  A,B:TSet;

procedure PrintSet;
```

```
    var
      I:word;

    begin
      write('A = {');
      for I:= 0 to MaxCount do
        if MyIn(I,A) then
          write(I,' ');
      writeln('}');

      write('B = {');
      for I:= 0 to MaxCount do
        if MyIn(I,B) then
          write(I,' ');
      writeln('}');
      writeln;
    end;

  begin
    fillchar(A,sizeof(A),0);
    fillchar(B,sizeof(B),0);
    printSet;

    MyInclude(A,1);
    MyInclude(B,2);
    PrintSet;

    MyUnion(A,B);
    PrintSet;

    MyInterSection(A,B);
    PrintSet

    MyDifference(A,B);
    PrintSet;

    MyExclude(A,1);
    PrintSet;
  end.
```

1.9 编 译 开 关

1. 编译开关

编译开关是指 Borland Pascal 程序编译时的一些参数。每一个编译开关用一个字母表示,有打开和关闭两种状态。编译指示给出了编译开关的开闭情况。不同的编译指示会引起程序可执行代码的不同表现。

一般将编译指示放在程序正文之前。用法是在一对大括号之间(亦即注释中)以$开头,其后紧跟着代表相应编译开关的字母以及表示其开闭的'+'号或'-'号。不同的编译开关之间用一个逗号隔开,与其次序无关。

例如:

编译指示{$A+}表示编译开关 A 打开。

编译指示{$C-,B+}表示编译开关 B 打开,编译开关 C 关闭。

对于在编译指示中没有显式给出的编译开关,系统将按照当前的默认值处理。

注意对编译指示的修改只有在下次编译时才会起作用。

2. 调试

$D、$L、$Y 这几个编译开关都是同程序的调试相联系的,一般同时开闭。当这三项都打开时,选手可以进行单步、断点、跟踪等调试。当这三项都关闭时,程序将不能进行调试,但运行速度会有所提高。

3. 自动检错

当$R,$S,$I,$Q 这几个编译开关打开时,在程序运行时将自动检查范围超界、堆栈溢出、I/O 操作失败、算术运算溢出等错误。如果发生此类错误,程序将报告错误代码,并停止运行。当这些编译开关关闭时,程序的速度可提高一倍左右,但这时如果程序发生错误,则很可能引起'死机'等情况。

上面所介绍的两类编译开关对程序员调试程序十分有用,但会影响程序的运行速度。因此建议读者在程序的编制、调试过程中打开这些编译开关,调试通过后应立即将其关闭并重新编译,以提高最终可执行文件的运行速度。

4. 其它

$G 是 286 代码生成编译开关。打开它时程序会编译出基于 286 指令集的可执行代码,否则则只能生成基本的 8086 代码。286 代码不能在 8086/8088 的机器上运行。由于现在已很少有人使用 286 以下的机器了,所以这一项一般应设置为打开。需要提醒读者的是,该编译开关的默认值是关闭的。

$X 是扩展语法编译开关。当它被打开时,可以使用一些扩展的 Pascal 语法。其中最为有用的当是将函数作过程使用。函数同过程中唯一的区别是函数有一个返回值。在

许多情况下我们只对函数的运行过程感兴趣,而并不需要得到函数的返回值。这时我们可以将函数当作过程来使用。例如在程序中如要等待用户按任一键,就可以直接使用如下语句:

```
...
readkey;
...
```

1.10 保护模式

1. 什么是保护模式?

在 1.5 节中曾经提到,程序的堆可以用到所有可用的空间。这里的空间是指操作系统所能管理和提供的空间。由于众所周知的原因,DOS 操作系统的内存空间局限于 640K 之中。而当今个人计算机的内存普遍达到 8M32M,因引,即使一个程序使用了所有的 DOS 内存,仍然有大量的内存闲置而无法利用。

如果要利用 640K 之外的内存空间,需要使个人计算机工作在保护模式下。而 DOS 本身是一个实模式操作系统,不能管理保护模式的程序。为了弥补 DOS 在这方面的缺陷,各大计算机公司联合制定了一个叫做"DOS 保护模式介面"的协议(DPMI),以规范保护模式下程序的运行。DPMI 是对 DOS 的扩展,在 DPMI 协议之下,DOS 应用程序可以访问计算机的所有内存。

2. 保护模式的使用

Borland Pascal 7 .0 也提供了同 DPMI 协议相兼容的保护模式编译选项。在 Compile 菜单的 Target 菜单项中有实模式(Real Mode,就是我们通常使用的目标模式)、保护模式(Protect Mode)、Windows 模式这三种目标模式。源程序可以被编译成不同模式下的可执行代码。

在保护模式下,程序的堆可以用到机器所有的内存空间。也就是说,对于某些对空间需求极大的题目,采用保护模式,可以在不改变算法的条件下,使程序的解题能力提高几倍、十几倍。

在保护模式下不能进行程序的调试,并且所有的错误都将被报告为 216 号保护模式错误,因此保护模式的程序调试较为麻烦。一般的做法是先在实模式下对小数据量调试通过,然后改用保护模式编译。

同实模式相比,使用保护模式还有一个需要考虑的问题:在保护模式下程序的运行速度会比实模式要慢一些。

第 2 章　ACM 国际大学生程序设计竞赛简介

2.1　背 景 与 历 史

1970 年在美国 Texas A&M 大学举办了首次区域竞赛,从而拉开了 ACM 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——区域赛和总决赛。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代表队参加了 ACM 区域竞赛。ACM 大学生程序设计竞赛由美国 ACM 国际计算机协会举办,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。

2.2　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 11 月在世界各地举行的“区域竞赛”。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,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并已读完至少一半时间的学生。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。

2.3　竞赛形式与评分办法

竞赛大约进行 5 个小时,一般有 7~8 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。

竞赛的程序设计语言包括 PASCAL,C 及 C++,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。

2.4 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 12 000 美元,第二名奖金为 6 000美元,第三名奖金为 3 000 美元,第四名至第十名将各得到 1 500 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。

2.5 历届竞赛获奖情况

本节讲述 ACM 国际大学生程序设计竞赛总决赛历届冠军的学校和决赛的年份,见表 2.1。

表 2.1

年 份	冠 军 学 校
1977	Michigan State University
1978	M.I.T
1979	Washington U., St. Louis
1980	Washington U., St. Louis
1981	U. of Missouri-Rolla
1982	Baylor University
1983	U. of Nebraska-Lincoln
1984	Johns Hopkins University
1985	Stanford University
1986	Cal Tech
1987	Stanford University
1988	Cal Tech
1989	U.C.L.A
1990	University of Otago (New Zealand)
1991	Stanford University
1992	Melbourne University (Australia)
1993	Harvard University
1994	University of Waterloo
1995	Albert-Ludwigs-University Freiburg
1996	U.C.Berkeley
1997	Harvey Mudd College
1998	Charles U-Prague

1998 年 3 月代表亚洲参加 ACM 总决赛(亚特兰大)的队有:

1. 清华大学
2. 上海交通大学
3. 上海大学
4. 台湾大学
5. 台湾师范大学
6. 南洋理工大学
7. 日本早稻田大学
8. 印尼 Binus 大学
9. 孟加拉理工大学

比赛结果,清华大学获亚洲第一名,世界第七名。1998 年亚洲赛区除上海和高雄两个考点外,还将增加东京和达卡两个预选赛考点。1999 年 ACM 总决赛将在荷兰举行。

第 3 章　试　题　I

3.1　消　防　车

中央城消防部门与交通部门合作维护反映当前城市街道状况的地图。在规定的某天里,一些街道将由于修复或施工而被关闭。消防队员需要能够选择从消防站到火警地点的不经过被关闭街道的路线。

中央城被划分成为不重叠的消防区,每区设有一个消防站。当火警发生时,一个中央调度站向火警地点所在的消防站发出警报并向其提供一个从该消防站到火警地点的可能路线的列表。你必须写一个程序供中央调度站使用以生成从地区消防站到火警地点的路线。

1. 输入

城市的每个消防区有一个独立的地图。每张地图的街区用小于 21 的正数标识,消防站总是在街区 # 1。输入文件包括一些测试项,代表在不同街区发生的不同的火警。测试项的第一行均表示离火警最近街区的编号。接下来若干行由用空格隔开的表示开放的街道所邻接的街区的正整数对组成。(例如,如果数对 4　7 是文件中的一行,那么街区 4 和 7 之间的街道是开放的。在街区 4 和 7 之间的路段上没有其他街区。)每个测试项的最后一行由一个 0 数对组成。

2. 输出

对于每个测试项,你的输出必须用数字来标识测试项(case # 1, case # 2, 等等)。输出必须将每条路线列在不同的行上,街区按路线顺序依次写出。输出必须给出从消防站到火警地点的所有路线的总数。其中只包括那些不经过重复街区的路线(由于显而易见的理由,消防部门不希望他们的车子兜圈子)。不同测试项的输出必须出现在不同的行上。接下来的样例输入和相应的输出代表了两个测试项。

3. 样例输入

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

```
5 6
2 3
2 4
0 0
4
2 3
3 4
5 1
1 6
7 8
8 9
2 5
5 7
3 1
1 8
4 6
6 9
0 0
```

4. 样例输出

```
CASE # 1:
1 2 3 4 6
1 2 3 5 6
1 2 4 3 5 6
1 2 4 6
1 3 2 4 6
1 3 4 6
1 3 5 6
There are 7 routes from the firestation to streetcorner 6 .

CASE # 2:
1 3 2 5 7 8 9 6 4
1 3 4
1 5 2 3 4
1 5 7 8 9 6 4
1 6 4
1 6 9 8 7 5 2 3 4
1 8 7 5 2 3 4
1 8 9 6 4
There are 8 routes from the firestation to streetcorner 4 .
```

3.2 数字三角形

考虑如图 3.1 所示的无限等边三角形网格上的点:

注意到如果我们用数字将这些点从上至下、从左至右标记,某些点构成了一些特定几何图形的顶点。例如,{1,2,3}和{7,9,18}是三角形的顶点集合,{11,13,24,26}和{2,7,9,18}是平行四边形的顶点集合,{4,5,9,13,12,7}和{8,10,17,21,32,34}是六边形的顶点集合。

写一个程序,不断读入这个三角形网格的一个点集合,判断这些点是否是以下这些“可接受的”图形之一的顶点:三角形(triangle)、平行四边形(parallelogram)、六边形(hexagon)。一个图形要成为可接受的,必须满足如下两个条件:

图 3.1

(1) 图形的每一条边必须和网格中的某条边重合;

(2) 图形的所有的边必须长度相同。

1. 输入

输入将由总数未知的点的集合组成。每个点集将出现在文件的一个不同的行上。一个集合中最多有 6 个点,每个点的范围被限制在 1 . .32767。

2. 输出

对于输入文件中的每个点集,你的程序应当根据集合中的点的数目来判断这个集合代表了什么样的几何图形,例如 6 个点只能够代表一个六边形,等等。输出必须是一系列的列出了每个点集及你的分析结果的行。

3. 样例输入

```
1  2  3
11  13  29  31
26  11  13  24
4  5  9  13  12  7
1  2  3  4  5
47
11  13  23  25
```

4. 样例输出

```
1 2 3 are the vertices of a triangle
11 12 29 31 are not the vertices of an acceptable figure
26 11 13 24 are the vertices of a parallelogram
4 5 9 13 12 7 are the vertices of a hexagon
1 2 3 4 5 are not the vertices of an acceptable figure
47 are not the vertices of an acceptable figure
11 13 23 25 are not the vertices of an acceptable figure
```

3.3 透 视 仪

物体透视仪通过对物体的连续水平切片进行透视来工作;每次成像一个切片。成像切片可以被重组以构成物体的三维模型。写一个程序以使用透视过程中得到的数据构造一个二维成像切片。

透视仪由放置在一个 10×15 的矩阵四周的四组传感器组成,见图 3.2。第一组由 10 个指向右方的传感器组成,第二组由 24 个沿对角线方向指向右上方的传感器组成,第三组由 15 个指向上方的传感器组成,第四组由 24 个沿对角线指向左上方的传感器组成。每个传感器记录了在其前面的物体沿直线穿过部分的厚度(方格数)。

图 3.2

各组传感器的读数被顺序记录。在同一组传感器中,数据同样被顺序记录。一个完整的透视由 73 个读数组成。

1. 输入

输入文件的第一行是一个整数,它表示其后成像切片的数目。对于每个成像切片,其后各行分别包括了 10,24,15,24 个代表第一到第四组相应的传感器读数的整数。读数的

顺序在上图中给出。尽管透视的结果有可能不止一个,但提供给你的数据不会有多个解释。

2. 输出

对于每个成像切片,你的程序应当打印 10 行,每行包括 15 个元素。用一个 # 号(#)表示该元素是物体的一部分;用一个点号(.)表示该元素不是物体的一部分。在相继成像切片的输出之间打印一个空行。

3. 样例输入(描述了上图中的物体)

```
1
10  10  6  4  6  8  13  15  11  6
0  1  2  2  2  2  4  5  5  6  7  6  5  6  6  5  5  6  6  3  2  2  1  0
2  4  5  5  7  6  7  10  10  10  7  3  3  5  5
0  0  1  3  4  4  4  4  3  4  5  7  8  8  9  9  6  4  4  2  0  0  0  0
```

4. 样例输出

```
. # # # # # # # # # # . . . .
. # # # # # # # # # # . . . .
. . . . # # # # # # . . . . .
. . . . . . # # # # . . . . .
. . . . . . . # # # # . . # #
. . . . . . . # # # # # # # #
# # # # # . . # # # # # # # #
# # # # # # # # # # # # # # #
. . # # # # # # # # # . . # #
. . . . # # # # # # . . . . .
```

3.4 多米诺效应

一副标准的“双六”多米诺骨牌共有 28 张,每张用类似于骰子的点数表示两个从 0 到 6 的数字的骨牌。这 28 张不同的骨牌由以下点数的组合构成:

骨牌 #	点数	骨牌 #	点数	骨牌 #	点数	骨牌 #	点数
1	0|0	8	1|1	15	2|3	22	3|6
2	0|1	9	1|2	16	2|4	23	4|4
3	0|2	10	1|3	17	2|5	24	4|5
4	0|3	11	1|4	18	2|6	25	4|6
5	0|4	12	1|5	19	3|3	26	5|5
6	0|5	13	1|6	20	3|4	27	5|6
7	0|6	14	2|2	21	3|5	28	6|6

用一副“双六”多米诺骨牌的所有牌可以被拼成一个 7×8 的点数网格。每种拼法至少对应一幅多米诺骨牌的“图”。一幅图由一个用合适的骨牌号替换在该骨牌上的点数而确定 7×8 的网格组成。下面是一个 7×8 的点数网格以及一个相应的骨牌号图。

7×8 点数网格

6	6	2	6	5	2	4	1
1	3	2	0	1	0	3	4
1	3	2	4	6	6	5	4
1	0	4	3	2	1	1	2
5	1	3	6	0	4	5	5
5	5	4	0	2	6	0	3
6	0	5	3	4	2	0	3

骨牌号图

28	28	14	7	17	17	11	11
10	10	14	7	2	2	21	23
8	4	16	25	25	13	21	23
8	4	16	15	15	13	9	9
12	12	22	22	5	5	26	26
27	24	24	3	3	18	1	19
27	6	6	20	20	18	1	19

1. 输入

输入文件包括一些测试项。每项由 7 行每行 8 个在 0 到 6 之间的整数组成,代表所看到的点数网格。每个测试项对应一个合法的骨牌放置方案(每个测试项至少有一个可能的图)。在各测试项之间没有分隔数据。

2. 输出

正确的输出包括一个测试项标号(从 # 1 开始),接下来是测试本身。在这之后是这个测试项的图的标号及对应于测试项的图(若多于一个图可以按任意次序输出)。当一个测试项的所有图都输出完后,一个总计行应当给出所有可能的图的总数。各测试项的输出之间至少应有 3 个空行,同一测试项的标号、测试项本身和图之间至少应有一个空行隔开。下面是一个包含两个测试项的样例输入文件及其正确的输出。

3. 样例输入

```
5 4 3 6 5 3 4 6
0 6 0 1 2 3 1 1
3 2 6 5 0 4 2 0
5 3 6 2 3 2 0 6
4 0 4 1 0 0 4 1
5 2 2 4 4 1 6 5
5 5 3 6 1 2 3 1
4 2 5 2 6 3 5 4
5 0 4 3 1 4 1 1
1 2 3 0 2 2 2 2
1 4 0 1 3 5 6 5
4 0 6 0 3 6 6 5
4 0 1 6 4 0 3 0
6 5 3 6 2 1 5 3
```

4. 样例输出

Layout # 1:

```
5 4 3 6 5 3 4 6
0 6 0 1 2 3 1 1
3 2 6 5 0 4 2 0
5 3 6 2 3 2 0 6
4 0 4 1 0 0 4 1
5 2 2 4 4 1 6 5
5 5 3 6 1 2 3 1
```

Maps resulting from layout # 1 are:

```
6  20 20 27 27 19 25 25
6  18 2  2  3  19 8  8
21 18 28 17 3  16 16 7
21 4  28 17 15 15 5  7
24 4  11 11 1  1  5  12
24 14 14 23 23 13 13 12
26 26 22 22 9  9  10 10
```

There are 1 solution(s) for layout # 1.

Layout # 2:

```
4 2 5 2 6 3 5 4
5 0 4 3 1 4 1 1
1 2 3 0 2 2 2 2
1 4 0 1 3 5 6 5
4 0 1 6 4 0 3 0
4 0 6 0 3 6 6 5
6 5 3 6 2 1 5 3
```

Map Resulting from layout # 2 are:

```
16 16 24 18 18 20 12 11
6  6  24 10 10 20 12 11
8  15 15 3  3  17 14 14
8  5  5  2  19 17 28 26
23 1  13 2  19 7  28 26
23 1  13 25 25 7  21 4
27 27 22 22 9  9  21 4
```

```
16  16  24  18  18  20  12  11
6   6   24  10  10  20  12  11
8   15  15  3   3   17  14  14
8   5   5   2   19  17  28  26
23  1   13  2   19  7   28  26
23  1   13  25  25  7   4   4
27  27  22  22  9   9   21  21
There are 2 solution(s) for layout #2 .
```

3.5 医院设备利用

县中心医院试图在经济疲软和人口增长加快的情况下改进其服务。为了支持医院的计划,你被要求设计一个模拟程序以供医院评估可选择的手术室、恢复室的配置方案。你的程序将监视手术室和恢复室床位在一天中的使用情况。

县中心医院有一些手术室和恢复室床位。每个手术病人被分配到一个可用的手术室,手术之后病人被分配到恢复室的一个床位。将一个病人从一间手术室移到一间恢复室所需的时间是固定的,与病人无关。相似的,为下一个病人准备一间手术室所需的时间和为一个新病人准备一个恢复室所需的时间都是固定的。

医院将在同一时间为所有病人正式编排手术,但他们实际进入手术室的顺序取决于他们的登记编号。一个进入手术的病人进入编号最小的可用的手术室。例如:如果手术室 2 和 4 同时成为可用的,登记表上的下一个不在手术室的病人进入手术室 2,再下一个病人同时进入手术室 4。手术之后,病人被送往编号最小的可用的恢复室床位,如果两个病人同时结束手术,较早进入手术的病人被首先分配到一个恢复室床位。如果两个病人同时进入手术,在登记表中位置较前的那个病人被首先分配到一个床位。

1. 输入

输入文件包括一次模拟运行的数据,输入文件中所有的数字数据都是整数,一行中相继的数字之间用空格隔开。文件的第一行是这个运行所要使用的医院配置参数,这些参数依次是:

手术室数目(最大允许 10);

恢复室床位数目(最大允计 10);

一天中第一次手术的开始时间(基于 24 小时制);

将病人从手术室送至恢复室所需的分钟数;

为下一个病人准备手术室所需的分钟数;

为下一个病人准备恢复床位所需的分钟数;

这天要动手术的病人总数(最大允许 100)。

这些初始配置数据之后是如下成对的病人数据的行:

第一行: 病人的姓(最多 8 个字符);

第二行: 手术需要的分钟数　在恢复室中所需要的分钟数。

输入文件中病人记录的顺序是根据决定病人手术顺序的病人登记表排序的。在任何配置中恢复室床位的数目都足够供从手术室中出来的病人使用(病人不会为恢复室床位排队)。计算到的时间不会超出 24:00。

2. 输出

正确的输出给出每个病人使用的手术室、恢复室床位、病人使用房间和床位的时间段以及这天医院设备的利用情况汇总。输出文件由两个描述模拟运行的结果的表格组成。第一个表格是列式表,包括相应的列项以给出每个病人的编号,病人的姓,手术室编号,手术起始时间,恢复室床位编号,以及病人进入和离开恢复室床位的时间。

第二个表格也是列式表,相应的列项汇总了手术室和恢复室床位的使用情况。这个汇总指出了设施类型(房间/ 床位),设施编号,使用的分钟数和可用时间利用率。可用时间被定义为从这天的第一次手术的开始时间到最后一个病人离开恢复室床位的时间。下面是一个样例输入文件及相应的正确的输出。

3. 样例输入

```
5　12　07　5　15　10　16
Jones
28　140
Smith
120　200
Thompson
23　75
Albright
19　82
Poucher
133　209
Comer
74　101
Perry
93　188
Page
111　223
Roggio
69　122
Brigham
42　79
Nute
22　71
Young
```

38 140

Bush

26 121

Cates

120 248

Johnson

86 181

White

92 140

4. 样例输出

Patient		Operating Room			Recovery Room		
#	Name	Room #	Begin	End	Bed #	Begin	End
1	Jones	1	7:00	7:28	3	7:33	9:53
2	Smith	2	7:00	9:00	1	9:05	12:25
3	Thompson	3	7:00	7:23	2	7:28	8:43
4	Albright	4	7:00	7:19	1	7:24	8:46
5	Poucher	5	7:00	9:13	5	9:18	12:47
6	Comer	4	7:34	8:48	2	8:53	10:34
7	Perry	3	7:38	9:11	4	9:16	12:24
8	Page	1	7:43	9:34	6	9:39	13:22
9	Roggio	4	9:03	10:12	9	10:17	12:19
10	Brigham	2	9:15	9:57	8	10:02	11:21
11	Nute	3	9:26	9:48	7	9:53	11:04
12	Young	5	9:28	10:06	3	10:11	12:31
13	Bush	1	9:49	10:15	10	10:20	12:21
14	Gates	3	10:03	12:03	8	12:08	16:16
15	Johnson	2	10:12	11:38	2	11:43	14:44
16	White	5	10:21	11:53	7	11:58	14:18

Facility Utilization

Type	#	Minutes	% Used
Room	1	165	29.68
Room	2	248	44.60
Room	3	258	46.40
Room	4	162	29.14
Room	5	263	47.30
Bed	1	282	50.72
Bed	2	357	64.21
Bed	3	280	50.36
Bed	4	188	33.81
Bed	5	209	37.59

Bed	6	223	40 .11
Bed	7	211	37 .95
Bed	8	327	58 .81
Bed	9	122	21 .94
Bed	10	121	21 .76
Bed	11	0	0 .00
Bed	12	0	0 .00

3.6 信息解码

一些信息编码方案要求一个编码信息被分为两部分来传输:第一部分叫做编码头,包括了信息的特征;第二部分是一个代表信息的编码,你必须写一个程序以如下的方案给信息解码。

程序所用编码方案的核心是如下的一个由‘0’和‘1’组成的‘关键字’的序列:

0,00,01,10,000,001,010,011,100,101,110,0000,0001,…,1011,1110,00000,…

这个序列中的第一个关键字的长度是1,接下来3个的长度是2,接下来7个的长度是3,接下来15个的长度是4,等等。如果两个相邻的关键字长度相同,第二个关键字可以由第一个关键字加1得到(二进制)。注意在序列中没有只包含‘1’的关键字。

这些关键字依次映射编码头中的字符。这就是说,第一个关键字(0)映射编码头中的第一个字符,第二个关键字映射编码头中的第二个字符,第k个关键字映射编码头中的第k个字符。例如,如果编码头是:

AB# TANCnrtXc

那么0就映射A,00映射B,01映射#,10映射T,000映射A,…,110映射X,0000映射c。

编码信息只包括‘0’和‘1’以及可以被省略的可能的换行符。信息被划分成段,每段的头3个数字以二进制方式表示该段中的每个关键字的长度。例如,如果头3个数字是010,那么该段余下的数字就可由长度为2的关键字(00,01或10)组成。每段的结尾是一个与段中关键字长度相同的全部为‘1’的串,因此一个关键字长度为2的段由11终止。所有的编码信息由000终止(表示一个关键字长度为0的段)。通过将段中的关键字一个一个地转换成编码头中相映射的字符,信息得到了解码。

1. 输入

输入文件包括一些测试数据。每个测试数据由一个独占一行的编码头和占据若干行的编码信息组成。事实上编码头的长度仅受关键字的最大长度7(二进制111)限制。如果编码头中有某个字符重复出现,几个关键字将映射这一个字符。编码信息只包括‘0’和‘1’,并且是根据上述编码方案合法编码的。也就是说,信息段由一个3位长的数字序列开始,以一个适当长度的全‘1’的序列结束。同一个段中的关键字长度都是一样的,并且都与编码头中的字符相对应,信息由000终止。换行符可以出现在编码信息中的任何部

分,它们不被认为是信息的一部分。

2. 输出

对于每个测试数据,你的程序必须在不同的行上写出解码后的信息,在各组信息之间不应有空行。下面是样例输入及相应的正确输出。

3. 样例输入

```
TNM AEIOU
0010101100011
1010001001110110011
11000
$ # * * \
0100000101101100011100101000
```

4. 样例输出

```
TAN ME
# # * \ $
```

3.7 代码生成

你的雇主需要一个简易可操作计算机的编译程序的模拟器。编译程序的输入是后缀表示的算术表达式,输出是汇编语言代码。

目标机器有一个寄存器以及以下操作,操作数或者是一个标记符或者是一个存储器地址。

L——将操作数载入寄存器;

A——将寄存器的值加上操作数;

S——将寄存器的值减去操作数;

M——将寄存器的值乘以操作数;

D——将寄存器的值除以操作数;

N——将寄存器的值取反;

ST——将寄存器的值保在操作数位置。

一个算术操作用运算结果替换寄存器的值。临时存储地址可通过形如“$n”(n是一个数字)的操作数由汇编程序分配。

1. 输入

输入文件由一些合法的后缀表达式组成。每个占一行。表达式算子是单个字母,算符是通常的算术运算符(+,-,*,\)以及一元取反运算符(@)。

2．输出

必须是满足以下要求的汇编语言代码:

(1) 每行一个操作,操作助记符与操作数(如果有的话)由一个空格隔开;

(2) 必须用一个空行隔开相继的表达式的汇编代码;

(3) 在汇编代码中必须保持原来算子的顺序;

(4) 一遇到算符就必须产生汇编代码;

(5) 使用尽可能少的临时存储空间(在满足上述条件的情况下);

(6) 对于表达式中的每一个算符,必须产生数目尽可能少的操作(在满足上述条件的情况下)。

3．样例输入

AB + CD + EF + + GH + + +

AB + DC + -

4．样例输出

L A
A B
ST $1
L C
A D
ST $2
L E
A F
A $2
ST $2
L G
A H
A $2
A $1

L A
A B
ST $1
L C
A D
N
A $1

第 4 章　试　题　II

4.1　电子表格计算器

电子表格是一个二维的元素数组。元素包括数据或可由给定的数据计算出结果的表达式。一个简单电子表格的数据是整数或由整数和引用的元素之间和与差的混合组成的表达式。对于任意表达式,如果它所引用的所有元素都是整数,那么这个表达式可以用其计算出的值来替换。你要写一个程序以计算简单电子表格。

1．输入

输入由简单电子表格的序列组成。每个电子表格由给出其行、列数目的一行开始。电子表格不会超过 20 行或 10 列。行用大写的字母 A 到 T 标识。列用十进制数字 0 到 9 标识。因此,第一行第一列的元素被标识为 A0;第二十行第五列的元素被标识为 T4。

在给出行列数之后是所有的元素,每个元素占一行,按行优先顺序排列(这就是说,第一行的所有元素先出现,之后是第二行的所有元素,等等)。每个元素初始时是一个有符号的整数或一个由无符号整数常量、引用元素标识和算符 + (加法)、- (减法)组成的表达式。如果一个元素初始时是一个有符号整数,相应的输入行由一个可选的负号开始,接下来是一个或多个十进制数字。如果一个元素初始时是一个表达式,其输入行将包括一个或多个被引用元素标号或无符号整数常量,由 + 和 - 号相互隔开。这样的行必从一个被引用元素标号开始。表达式不超过 75 个字符。输入行行首不会是空格。表达式中不包含空格,任意行尾都可能包含任意多的空格。

若电子表格为 0 行 0 列则表示输入文件的结束。

2．输出

对于输入中的每一个电子表格,你要计算每个表达式的值,并用一个有行列标记的矩形数字数组显示计算后的电子表格。一列中的所有数字必须向右对齐列标记,每个表达式中的算符从左向右计算,元素中的数值的绝对值总是小于 10000。由于表达式可以引用同样包含表达式的元素,元素计算的顺序由这些表达式自身所决定。

如果一个电子表格中的一个或多个元素存在相互循环引用的表达式,这个电子表格的输出应只包含一个按行优先顺序排列的未计算元素的列表,每个一行,每行包括元素标号、一个冒号、一个空格和这个元素初始时的表达式。

每个电子表格的输出之后应当是一个空行。下面是样例输入和样例输出。

3. 样例输入

```
2 2
A1 + B1
5
3
B0 - A1
3 2
A0
5
C1
7
A1 + B1
B0 + A1
0 0
```

4. 样例输出

```
        0          1
A       3          5
B       3         - 2

A0:A0
B0:C1
C1:B0 + A1
```

4.2 布　　线

计算机网络要求网络中的计算机被连接起来。本问题考虑一个“线性”的网络,在这一网络中计算机被连接到一起,并且除了首尾的两台计算机只分别连接着一台计算机外,其他任意一台计算机恰连接着两台计算机。图 4.1 中用黑点表示计算机,它们的位置用直角坐标表示(相对于一个在图中未画出的坐标系)。网络中连接的计算机之间的距离单位为英尺。

由于很多原因,我们希望使用的电缆长度应尽可能地短。你的问题是去决定计算机应如何被连接以使你所用的电缆长度最短。在设计方案施工时,电缆将埋在地下,因此连接两台计算机所要用的电缆总长度等于计算机之间的距离加上额外的 16 英尺电缆,以从地下连接到计算机,并为施工留一些余量。

图 4.2 给出了上图中计算机的最优连接方案,这样一个方案所用电缆的总长度是 (4 + 16) + (5 + 16) + (5 .38 + 16) + (11 .18 + 16) = 90 .01 英尺。

图　4.1

图　4.2

1．输入

输入文件由数据项的序列组成，每个数据项的第一行为网络中的计算机总数。每个网络包括的计算机台数至少为 2，至多为 8。如果给出的计算机的数量为 0，则意味着输入的结束。在每个数据项的初始行指出了网络中的计算机总数之后，数据项的随后各行将给出网络中各台计算机的坐标。坐标值是 0 到 150 之间的整数。没有两台计算机在相同坐标位置并且一台计算机只被给出一次坐标。

2．输出

每个网络的输出结果的第一行为该网络的编号(根据其在输入数据中的位置先后决定)，以下各行每行表示一根连接两台计算机的电缆。最后一行应当用一个句子给出使用电缆的总长度。在给出每根电缆的长度时，将网络链从一台计算机连接到另一台计算机(从哪一端开始无关紧要)。使用与样例输出相似的格式，用一个打满星号的行隔开不同的网络，距离以英尺为单位，打印时保留二位小数。

3．样例输入

```
6
5 19
55 28
38 101
```

```
28 62
111 84
43 116
5
11 27
84 99
142 81
88 30
95 38
3
132 73
49 86
72 111
0
```

4. 样例输出

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
Network # 1
Cable requirement to connect (5,19) to (55,28) is 66 .80 feet .
Cable requirement to connect (55,28) to (28,62) is 59 .42 feet .
Cable requirement to connect (28,62) to (38,101) is 56 .26 feet .
Cable requirement to connect (38,101) to (43,116) is 31 .81 feet .
Cable requirement to connect (43,116) to (111,84) is 91 .15 feet .
Number of feet of cable required is 305 .45 .
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
Network # 2
Cable requirement to connect (11,27) to (88,30) is 93 .06 feet .
Cable requirement to connect (88,30) to (95,38) is 26 .63 feet .
Cable requirement to connect (95,38) to (84,99) is 77 .98 feet .
Cable requirement to connect (84,99) to (142,81) is 76 .73 feet .
Number of feet of cable required is 274 .40 .
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
Network # 3
Cable requirement to connect (132,73) to (72,111) is 87 .02 feet .
Cable requirement to connect (72,111) to (49,86) is 49 .97 feet .
Number of feet of cable required is 136 .99 .
```

4.3 无线电定向

船舶可以利用定向天线通过对当地无线电波发射站进行定向测量所得到的数据来确

定其当前位置。每个发射站都坐落在已知的位置并发射特定的信号。当船舶探测到某个信号,就旋转自己的天线直到该信号达到最大强度。这就得到了一个相对于该发射站位置的偏移角。给定了前次定向测量的数据(时间、偏移角、发射站的位置),一般说来,再有一次定向测量就足以确定船舶的当前位置。你要写一个程序以根据两次定向数据在可能的情况下去确定船舶的位置。

对于这个问题,发射站和船舶的位置是基于一个直角坐标系的。X 轴正方向指向正东;Y 轴正方向指向正北。航线指船舶航行的方向,从正北方向顺时针用度计量。也就是说,正北方向是 0°,正东方向是 90°,正南方向是 180°,正西方向是 270°。定向测量的偏移角相对于船舶的航线以度的形式顺时针给出。船舶的天线无法指出发射站是在其哪一侧。一个 90°的偏移角意味着发射站在 90°或 270°方向。

图 4.3 画出了一艘船的航线,图中 Y 轴是正北方向。

图 4.3

船的航线方向为 75°,这个发射站相对于航线的偏移角是 45°。

1. 输入

输入的第一行是一个给出信号发射站数目的整数(最多为 30),接下来每个发射站有一行数据。每行由发射站的名字(不超过 20 个字母的字符串)开始,接下来是其横坐标、纵坐标。这些项之间用一个空格隔开。

发射站数据之后是一个整数(占一行),表示下面将给出的船舶数据的数目。每个船舶数据由三行数据组成,一行是速度数据,另二行是定向测量数据。

输入行中的数据			数据的意义
航线	航速		船舶的航线,船舶的航速
时间 1	名字 1	角度 1	第一次定向测量的时间,信号发射站的名字,偏移角
时间 2	名字 2	角度 2	第二次定向测量的时间,信号发射站的名字,偏移角

所有的时间都用分钟给出,从午夜开始计时,时间不超过 24 小时。速度是单位时间内走过的距离(距离的单位同直角坐标系的单位一致)。每个船舶数据的第二行给出了第

一次定向测量的时间(整数值)、信号发射站的名字、定向测量到的相对于船舶航线的偏移角。这三项之间用一个空格隔开。第三行给出了第二次定向测量的数据。此次测量的时间不小于第一次测量的时间。

2. 输出

对于每个船舶数据,你的程序应首先打印船舶数据编号(Scenario 1, Scenario 2, 等等),再给出第二次定向测量时船舶的位置(精确到小数点后二位)的信息。如果无法确定船舶的位置,应给出"Position cannot be determined"信息。下面是样例输入以及相应的输出。

3. 样例输入

```
4
First 2.0 4.0
Second 6.0 2.0
Third 6.0 7.0
Fourth 10.0 5.0
2
0.0 1.0
1 First 270.0
2 Fourth 90.0
116.5651 2.2361
4 Third 126.8699
5 First 319.3987
```

4. 样例输出

```
Scenario 1: Position cannot be determined
Scenario 2: Position is(6.00,5.00)
```

4.4 扑灭飞蛾

东北地区的昆虫学家设置了试验点以确定该地区 Jolliet 飞蛾的汇集地。他们计划研究控制飞蛾数量增长的扑灭方案。

研究要求将试验点组织起来,使飞蛾能在这些试验点所在区域中被捕捉以便试验每个扑灭方案。一个区域是指能够围住所有试验点且周长最小的多边形。例如,图 4.4 是某个区域的试验点(用黑点表示)及其相应的多边形。

你应当写一个程序对于输入的试验点的位置,输出求得的区域边界上的试验点以及该区域边界的周长。

1. 输入

输入文件将包括多个区域的数据记录。每个记录的第一行是该区域中试验点的个数

(一个整数)。接下来每一行用两个实数分别表示一个试验点所在位置的横、纵坐标。同一记录中的数据不会重复。若区域中试验点个数为 0 则表示输入的结束。

2. 输出

对于一个区域的输出至少包括 3 行:

第一行:

区域的编号(第一个记录对应于区域 # 1,第二个对应于区域 # 2,等等)。

以下各行:

在区域边界上的试验点的列表。试验点必须用标准格式“(横坐标,纵坐标)”表示,精确到小数点后 1 位。该列表的起始点是无关紧要的,但列表中的点必须是顺时针排列的并且起始点和终了点应当是同一个点。对于同在一条直线上的点,任何描述了最小周长的情况都是可接受的。

最后一行:

区域的周长,精确到小数点后 2 位。

输入中相继的记录的输出之间应有一个空行隔开。

下面是一个包含 3 个区域记录的样例输入,其后是正确的样例输出。

3. 样例输入

```
3
1 2
4 10
5 12 3
6
0 0
1 1
3 1 1 3
3 4 5
6 2 1
2 -3 2
7
1 0 5
5 0
4 1 5
3 -0 2
2 5 -1 5
0 0
2 2
0
```

图 4.4

4. 样例输出

```
Region #1:
(1.0,2.0) - (4.0,10.0) - (5.0,12.3) - (1.0,2.0)
Perimeter length = 22.10

Region #2:
(0.0,0.0) - (3.0,4.5) - (6.0,2.1) - (2.0, - 3.2) - (0.0,0.0)
Perimeter length = 19.66

Region #3:
(0.0,0.0) - (2.0,2.0) - (4.0,1.5) - (5.0,0.0) - (2.5, - 1.5) - (0.0,0.0)
Perimeter length = 12.52
```

4.5 寻找冗余

在设计关系数据库的表格时,术语“函数依赖”(FD)被用来表示不同域之间的关系。函数依赖是描述一个集合中的域的值与另一个集合中的域的值之间的关系。记号 X - > Y 被用来表示当集合 X 中的域被赋值后,集合 Y 中的域就可以确定相应的值。例如,如果一个数据表格包含“社会治安编号”(S)、“姓名”(N)、“地址”(A)、“电话”(P)的域,并且每个人都与一个特定的互不相同的 S 值相对应,根据域 S 就可以确定域 N、A、P 的值。这就被记作 S - > NAP。

写一个程序以找出一组依赖中所有的冗余依赖。一个依赖是冗余的是指如果它可以通过组里的其它依赖得到。例如,如果组里包括依赖 A - > B、B - > C 和 A - > C,那么第三个依赖是冗余的,因为域 C 可以用前两个依赖得到(域 A 确定了域 B 的值,同样域 B 确定了域 C 的值)。在 A - > B、B - > C、C - > A、A - > C、C - > B 和 B - > A 中,所有的依赖都是冗余的。

1. 输入

输入文件可以包含任意多组依赖。每组的第一行是一个不超过 100 的整数,它表示该组中函数依赖的个数。一个包含 0 个依赖的组表示输入的结束。组里的每个依赖占一行且不重复,每行包含两个用字符“ - ”和“ > ”隔开的非空域列表。域列表只包含大写的字母,函数依赖的数据行中不包括空格和制表符,不会出现“平凡”冗余依赖(例如 A - > A)。为了标识之用,组被顺序编号,从 1 开始;依赖也顺序编号,每组均从 1 开始。

2. 输出

对于每一组函数依赖,你的程序必须顺序给出组号、组中的每一个冗余依赖,以及组中其它依赖的一个序列以说明该依赖是冗余的。如果许多函数依赖的序列都能被用来说

明一个依赖是冗余的,则其中任一序列,即使它不是最短的证明序列,都是可接受的。在可接受的证明序列中的所有依赖都应当是必需的。如果某一组中不包含冗余依赖,显示“No redundant FDs.”信息。

3. 样例输入

```
3
A->BD
BD->C
A->C
6
P->RST
VRT->SQP
PS->T
Q->TR
QS->P
SR->V
5
A->B
A->C
B->D
C->D
A->D
3
A->B
B->C
A->D
0
```

4. 样例输出

```
Set number 1
    FD 3 is redundant using FDs: 1 2

Set number 2
    FD 3 is redundant using FDs: 1
    FD 5 is redundant using FDs: 4 6 2

Set number 3
    FD 5 is redundant using FDs: 1 3
        ---或---
    FD 5 is redundant using FDs: 2 4
```

Set number 4

No redundant FDs .

4.6 奥 赛 罗

奥赛罗是由两个人在 8×8 的棋盘上进行的一种游戏,所用的棋子一面是白色,另一面是黑色。一个游戏者使用白色面朝上的棋子,另一个游戏者使用黑色面朝上的棋子。游戏者轮流在棋盘上的空格子里放入一个棋子。在放置棋子时,游戏者必须夹住至少一个另一种颜色的棋子。当棋子沿横、纵、对角线方向排在一条直线上并且两端的棋子都是当前游戏者的颜色时,称棋子被夹住了,见图 4.5。当游戏者走了一步之后,所有被夹住的棋子都变到了走这步的游戏者所用棋子的颜色(有可能在一步中夹住多于一行的棋子)。

图 4.5

写一个程序,读入若干奥赛罗游戏。输入的第一行是要处理的游戏个数,每个游戏由棋盘局面及其后所跟的一些命令组成。棋盘局面由 9 行组成,前 8 行给出了棋盘的当前状态。这 8 行每行包括 8 个字符,所有这些字符是如下之一:

'-' 表示空格子;

'B' 表示放有黑色棋子的格子;

'W' 表示放有白色棋子的格子。

第 9 行是'B'或'W'之一,以指出当前游戏者。你可以假定数据是合法格式的。

命令可以是为当前游戏者列出所有可能的走步、走一步或退出当前游戏。每条命令占一行且无空格。命令的格式如下:

为当前游戏者列出所有可能的走步:

命令是'L',在行的第 1 列。程序应检查棋盘并用格式(x,y)打印当前游戏者的所有合法走步,这里 x 代表合法走步的行号,y 代表列号。这些走步应当被按行优先顺序打印,也就是说:

(1) 如果 j 大于 i,则所有行号是 i 的合法走步将在所有行号是 j 的合法走步之前打

印;

(2) 如果有多于一个的合法走步的行号是 i,则这些走步将按列号升序打印。

所有的合法走步应当输出在一行上。如果因为当前游戏者不可能夹住任何棋子而不存在合法走步,程序应当打印“ No legal move .”信息。

走一步:

命令是'M',在行的第 1 列,其后是在第 2、3 列的两个数字。这两个数字是放置当前游戏者颜色的棋子的格子的行、列号,除非当前游戏者没有合法走步。如果当前游戏者没有合法走步,当前游戏者将首先换成另一个游戏者,此走步将是新游戏者的走步。你可以假定此时走步一定是合法的。你应该将变更记录至棋盘,包括加入新棋子和改变所有被夹住的棋子的颜色。走完了此步之后,按“ Black-xx White-yy ”的格式打印棋盘上每种颜色的棋子数目,这里 xx 是棋盘上黑色棋子的数目,yy 是棋盘上白色棋子的数目。走完一步后,当前游戏者将换成没有走步的游戏者。

退出当前游戏:

命令是'Q',在行的第一列。在这里,用同输入相同的格式打印最后的棋盘局面。这终止了当前游戏的命令输入。

你可以假定命令在语法上是正确的。用一个空行隔开不同游戏的输出,输出中的其它地方不得出现空行。

1. 样例输入

```
2
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - WB - - -
- - - BW - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
W
L
M35
L
Q
WWWWB - - -
WWWB - - - -
WWB - - - - -
WB - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
```

- - - - - - - -
B
L
M25
L
Q

2. 样例输出

(3,5) (4,6) (5,3) (6,4)
Black - 1 White - 4
(3,4) (3,6) (5,6)
- - - - - - - -
- - - - - - - -
- - - - W - - -
- - - WW - - -
- - - BW - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -

No legal move .
Black - 3 White - 12
(3,5)
WWWWB - - -
WWWWW - - -
WWB - - - - -
WB - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -

4.7 城市正视图

建筑物的正视图是其在竖直平面上的垂直投影。城市的外部正视图给出了从城市之外以特定角度观察城市所能看到的轮廓和外观。南正视图给出了建筑物的完整的矩形外观或建筑物没有被南面更高的建筑物遮挡的部分的外观,它是没有边界的。对于这个问题,你必须写一个程序以判断在南正视图中城市中的哪些建筑物是可见的。

为了简单起见,假定正视图中所有的建筑物都是完整的矩形固体,每个都有东西向和南北向各两个面。你的程序将在给出了城市中的每座建筑物的位置和高度的基础上,去

寻找在南正视图中出现的建筑物。图 4.6(a)是一个城市的地图(粗体数字(在每座建筑物的左上角)标记了建筑物。常规体数字(右下)是建筑物的高度。被阴影覆盖的建筑在南正视图中是可见的)。图 4.6(b)是这个城市的南正视图。

图 4.6

1. 输入

你的程序的输入由若干城市的用数字描述的地图组成。每个地图的第一行是城市中建筑物的数目(小于 101 的非负整数)。其后每行是一个建筑物的数据——如下顺序的 5 个用空格隔开的实数:

西南角的横坐标;

西南角的纵坐标;

建筑物的宽度(南墙的长度);

建筑物的长度(西墙的长度);

建筑物的高度。

地图是基于 x 轴指向东方、y 轴指向北方的直角坐标系的。保证输入数据是合法的(建筑物的数目同输入文件中的后继行的数目相同;建筑物在地图上不会重叠)。输入由表示一个无建筑物的地图的数字 0 终止。

2. 输出

建筑物按照其数据行在输入数据中出现的位置编号——建筑物 1 对应建筑物数据的第一行,建筑物 2 对应下一行,建筑物 n 对应建筑物数据的第 n 行(后继的地图中的建筑物编号仍然是从 1 开始)。

对于每个地图,输出由标识了地图的行开始(地图 # 1,地图 # 2,等等)。接下来的一行给出了南正视图中从南至北、自西向东可见的建筑物的编号。也就是说,如果建筑物 n 和建筑物 m 都是可见的并且建筑物 n 的西南角在建筑物 m 的西南角之西,那么编号 n 在编号 m 之前打印。如果建筑物 n 和建筑物 m 的西南角有着相同的横坐标并且建筑物 n 在建筑物 m 之南,那么编号 n 在编号 m 之前打印。只要一个建筑物的南墙的一部分在正视图中出现的面积是正的,这个建筑物就被认为是可见的。用一个空行隔开输入中相继的地图的输出。

3. 样例输入

```
14
160  0  30  60  30
125  0  32  28  60
95  0  27  28  40
70  35  19  55  90
0  0  60  35  80
0  40  29  20  60
35  40  25  45  80
0  67  25  20  50
0  92  90  20  80
95  38  55  12  50
95  60  60  13  30
95  80  45  25  50
165  65  15  15  25
165  85  10  15  35
0
```

4. 样例输出

```
For map # 1, the visible building are numbered as follow:
5  9  4  3  10  2  1  14
```

第 5 章　试　题　III

5.1　旅 行 预 算

一个美国旅行代理商经常被要求去估计开车从一城市旅行至另一城市的最小费用。他有一个在通常路线上的大多数加油站的列表。列表包括了所有加油站的位置及当前每加仑汽油的价格。

为了简化估计费用的过程,代理商使用了以下的简化汽车驾驶员行为的规则:

(1) 除非汽车无法用油箱里的汽油达到下一个加油站(如果有的话)或目的地,在油箱里还有不少于最大容量一半的汽油时,驾驶员从不在加油站停下来。

(2) 在每一个停下的加油站驾驶员总是将油箱加满。

(3) 在一个加油站停下之后,驾驶员将为旅程在快餐和糖果上花去 2.00 元。

(4) 在驶向加油站或目的地时,驾驶员不需要超过必需量的汽油。不需要“安全余量”。

(5) 驾驶员开始旅行时油箱总是满的。

(6) 在每个加油站付款时四舍五入到分(1 元等于 100 分)。

你必须写一个程序以估计驾驶员在旅程上至少要为汽油和食品付多少钱。

1. 输入

程序的输入将由若干个对应不同的旅程的数据项组成,每个数据项由若干信息行组成。开始的 2 行给出了出发地和目的地的信息,数据项的后继行代表了路线上的加油站,每个加油站用一行表示。下面是输入数据中数据项的精确格式及其含义。

第一行: 一个实数——从出发地到目的地的距离(英里);

第二行: 三个实数及一个整数。

(1) 第一个实数是汽车油箱的最大的容量(加仑);

(2) 第二个实数是汽车每加仑汽油可以行驶的英里数;

(3) 第三个实数是汽车在出发地城市加满油箱的费用(单元: 元);

(4) 整数(小于 51)是路线上加油站的数目。

接下来的每一行:两个实数:

(1) 第一个实数是从出发地到加油站的距离(单位: 英里);

(2) 第二个实数是该加油站出售的汽油每加仑的价格(单位: 分)。

数据项中的所有数据都是正的。一条路线上的加油站根据其到出发地的距离递增排列。路线上不存在这样的加油站,它到出发点的距离大于从出发点到目的地的距离。每

条路线上的加油站都被适当地安排以使得任何汽车都能够从出发地开到目的地。

输入数据中若某行为一个负数则表示输入数据结束。

2. 输出

对于输入中的每个数据项,你的程序必须打印数据项编号以及一个给出了精确到分的汽油和食品的最小总费用。总费用必须包括开始时在出发地加满油箱的费用。下面是两个不同的数据项的样例输入以及相应的正确的输出。

3. 样例输入

```
475 .6
11 .9   27 .4   14 .98   6
102 .0   99 .9
220 .0   132 .9
256 .3   147 .9
275 .0   102 .9
277 .6   112 .9
381 .8   100 .9
516 .3
15 .7   22 .1   20 .87   3
125 .4   125 .9
297 .9   112 .9
345 .2   99 .9
-1
```

4. 样例输出

```
Data Set #1
   minimum cost = $27 .31
Data Set #2
   minimum cost = $38 .09
```

5.2 分划中土地的划分

一个分划是由许多块具有多边形边界的土地组成的。一个测量者已经测量了这些土地,并给出了所有分界线的位置。这是唯一可以利用的信息,而关于分划中土地的更多的信息是需要求解的。具体来说,计划编制者想要根据边界线段的数目(B = 3,4,5…)将土地划分。

写一个程序,将测量者的数据作为输入,生成所需的关于分划中土地属性信息的输出。

1．输入

输入文件由若干个数据项组成。每个数据项由包含测量数据中边界线段数目(4≤N≤200)的一行开始。接下来的 N 行每行包括四个整数,代表了分界线段的 2 个端点的(x,y)坐标对。输入文件由一个 0 结束。

2．输出

对于每个数据项,给出列出了属于每种边界线段数目(B = 3,4,5…)分类的土地数目的输出。在你的输出中不要包括那些没有任何土地从属的分类的情况。每个数据项的输出由包含了一个适当标记的数据项编号的行开始。相继的数据项的输出之间应用一个空行隔开。

假定:

(1) 每个数据项对应于一个矩形的分划(例如图 5.1(a)和图 5.1(b))。矩形分划的边界平行于 x 轴和 y 轴。

(2) 输入文件中所有的坐标都是从 1 到 10000 的正整数。

(3) 输入文件中的分界线段不会超出某块土地的拐角。例如,在图 5.1(a)中测量者必须从点(10,41)测量到点(15,41),然后从点(15,41)测量到点(20,41),而不是测量从(10,41)到(20,41)的整条直线。

(4) 每块土地至少有一条边界线段在矩形分划的边界上。

图 5.1(a)和图 5.1(b)给出了两个假想的分划。在图 5.1(a)中有 12 条分界线段,图 5.1(b)有 27 条。其后的样例输入文件包括了这两个测试项的数据。图 5.1(b)中左上角的土地有一条从(16,16)到(17,18)的边界线以及另一条从(17,18)到(19,22)的边界线。因此这块土地的边界由 5 条边界线段组成,尽管从几何上说它是一块有四条边的土地。与之类似地,图 5.1(a)中左上角的土地的边界是由 6 条边界线段组成的,尽管从形状上来看它是一个五边形。

图　5.1

3．样例输入

```
12
10 41 15 41
15 41 20 41
10 36 15 36
15 36 17 36
10 31 15 31
15 31 20 31
10 41 10 36
10 36 10 31
15 41 17 38
17 38 17 36
15 36 15 31
20 41 20 31
27
10 22 19 22
19 22 23 22
23 22 28 22
28 22 37 22
10 16 16 16
17 16 23 16
23 16 24 16
24 15 28 15
28 15 31 15
10 10 17 10
17 10 24 10
24 10 31 10
31 10 37 10
10 22 10 16
10 16 10 10
17 18 17 16
17 16 17 10
24 16 24 15
24 15 24 10
23 22 23 16
28 22 28 15
31 15 31 10
37 22 37 17
37 17 37 10
16 16 17 18
```

17 18 19 22

31 15 37 17

0

4. 样例输出

Case 1

Number of lots with perimeter consisting of 4 surveyor 's lines = 1

Number of lots with perimeter consisting of 6 surveyor 's lines = 1

Number of lots with perimeter consisting of 7 surveyor 's lines = 1

Total number of lots = 4

Case 2

Number of lots with perimeter consisting of 4 surveyor 's lines = 1

Number of lots with perimeter consisting of 5 surveyor 's lines = 4

Number of lots with perimeter consisting of 6 surveyor 's lines = 3

Total number of lots = 8

5.3 寻找堂亲

牛津英语词典对堂亲(cousin)的解释如下:

堂亲(又作一代堂亲),是我叔叔或阿姨的子女;我的二(三…)代堂亲,是我父母的一(二…)代堂亲的子女;我的一代隔一(二…)代堂亲,是我的一代堂亲的子女(孙子女…),亦表示我(祖)父母的一代堂亲。

更精确地,任何两个人,如果其最近的公共祖先与其中一人相距(m + 1)代,与另一人相距(m + 1) + n 代,就称这两个人为 m 代隔 n 堂亲。通常 m≥1 且 n≥0,但是由于计算机从 0 开始计数的习惯,在这个问题中我们只要求 m≥0 且 n≥0 即可。这就扩展了通常的定义,因此兄弟姐妹之间是零代堂亲关系。我们把这样的关系记作 m-n-堂亲(cousin-m-n)。

如果其中一人是另一人的 p 代祖先,这里 p≥1,他们之间的关系就是 p-后裔(descendant-p)。

如果 $m_1 < m_2$ 或($m_1 = m_2$ 且 $n_1 < n_2$),则 m_1-n_1-堂亲比 m_2-n_2-堂亲来得近。如果 $p_1 < p_2$,则 p_1-后裔比 p_2-后裔来得近。p-后裔总比 m-n-堂亲来得近。

写一个程序,读入各人之间的简单关系,并在可能的情况下,为任意二人找出他们之间最近的堂亲或后裔关系。

1. 输入

输入的每一行由字符“ # ”、“ R ”、“ F ”、“ E ”之一开始。

“ # ”打头的行是注释。把它们忽略掉。

“R”打头的行指示你的程序记录两个人之间的关系。“R”之后的前5个字符组成了第一个人的名字;接下来的5个字符组成了第二个人的名字。大小写要区分。名字之后是用空格隔开的一个非负整数k,它定义了这样的关系。如果k是0,则这两个人是兄弟姐妹。如果k是1,则前一个人是后一个人的子女。如果k是2,则前一个人是后一个人的孙子女,依此类推。忽略该行上这个整数之后的所有内容。

“F”打头的行是问题;如果“F”之后的5个字符组成名字的人同其后5个字符组成名字的人之间有亲戚关系的话,你的程序要找出最近的关系。忽略该行第二个名字之后的所有内容。问题只能根据在其之前的所有“R”打头的行所提供的信息来回答。

输入数据的末尾会有“E”打头的一行。忽略“E”打头的行及其后面的所有内容。

2. 输出

对于每个“F”打头的行,你的程序应以下面两种格式之一报告名字为aaaaa和bbbbb的二人之间所存在的最近的关系:

aaaaa and bbbbb are descendant-p .

aaaaa and bbbbb are cousin-m-n .

这里m、n、p应用根据前述定义计算出的整数替换。如果这二人之间没有任何关系,你的程序应输出如下信息:

aaaaa and bbbbb are not related .

假定:

一个人不能是其自身的祖先。

3. 样例输入

```
# A Comment !
RFred Joe 1 Fred is Joe's son
RFran Fred2
RJake Fred1
RBill Joe 1
RBill Sue 1
RJean Sue 1
RJean Don 1
RPhil Jean 3
RStan Jean 1
RJohn Jean 1
RMary Don 1
RSusanMary 4
RPeg Mary 2
FFred Joe
FJean Jake
```

FPhil Bill

FPhil Susan

FJake Bill

FDon Sue

FStan John

FPeg John

FJean Susan

FFran Peg

FJohn Avram

RAvramStan 99

FJohn Avram

FAvramPhil

E

4. 样例输出

Fred and Joe are descendant-1 .

Jean and Jake are not related .

Phil and Bill are cousin-0-3 .

Phil and Susan are cousin-3-1 .

Jake and Bill are cousin-0-1 .

Don and Sue are not related .

Stan and John are cousin-0-0 .

Peg and John are cousin-1-1 .

图 5.2

Jean and Susan are cousin-0-4 .

Fran and Peg are not related .

John and Avram are not related .

John and Avram are cousin-0-99 .

Avram and Phil are cousin-2-97 .

样例输入的图解见图 5.2。

5.4 黄 金 图 形

想象有这样一个国家,它的城市中的街道都是有规则的矩形网格。现在假定一个对几何十分着迷的游客打算在若干座这样的城市里旅行。每次旅行从城市的中心十字路口开始,该交叉点的标号为(0,0),我们这位爱好数学的旅游者想要朝北、南、东、西四个方向之一出发,走过一个街区,在向北走后观赏十字路口(0,1)的街景,向南走后观赏(0, - 1),向东走后观赏(1,0),向西走后观赏(- 1,0)。出于对城市规则性的极大热情,我们的数学爱好者想在下一次停下来之前走更长的一段路,也就是走两个街区。不仅如此,我们的旅游者既不想走同前次相同的方向,也不想折返,所以将向左或右转 90 度。下一段路将是三个街区,又一次紧接着一个转弯,然后是四个街区、五个街区,如此不断增长下去,直到最后,在一天的结束时,我们疲劳的旅行者回到出发点(0,0)。

由这些几何上的旅行所描述的自交图形被称做黄金图形。

不走运的是,我们的旅行者将在盛夏作上述旅行,这时的修路工作将破坏城市街道完整的规律性。在某些十字路口将会有无法通行的路障。然而幸运的是,这个国家有限的预算使得在任何一个城市都不会有超过 50 个路口被堵塞。为了获得市民的信任,城市会预先通告修路计划。我们的旅行者已经得到了一份这样的计划,并且保证不会使黄金图形的旅程被融化的沥青所阻挡。

写一个程序以构造一个城市中所有可能的黄金图形。

1. 输入

由于我们的旅游者想要访问若干个城市,输入文件的第一行将是一个表示要访问的城市数目的整数。

对于每个城市将由包含了一个不大于 20 的正整数的一行开始,正整数给出了黄金图形的最长的边的长度。也就是使旅行者回到(0,0)的最后一条边的长度。在这之后的新的一行上将有一个在 0 到 50 之间(包括 0 和 50)的整数 M,给出了被封锁的街区的数目。接下来是 M 个整数对,一对占一行,分别指出了每一个路障的横、纵坐标。

2. 输出

对于输入中的每个城市,构造所有可能的黄金图形。每个黄金图形必须用一个由集合{n,s,e,w}中的字符组成的序列表示(占一行)。在黄金图形的列表之后应有一行给出了所找出的解答数目。这一行的格式应当同样例输出一致。每个城市的输出之后应当有

一个空行。下面是样例输入及输出。

3. 样例输入

```
2
8
2
 -2   0
6   -2
8
2
2   1
 -2   0
```

4. 样例输出

```
wsenenws
Found 1 golygon(s) .
Found 0 golygon(s) .
```

图 5.3

第一个城市见图 5.3。

5.5 MIDI 预处理

MIDI(乐器数字接口)是一个在计算机和音乐合成器之间进行通讯的标准。标准的一部分定义了这样两个指令,当它们传送到合成器之后能打开或关闭一个特定音符发音。在这个问题里,我们将考虑处理简单的 MIDI“程序”。在下面的例子里,三个同步的音符(弦乐,音符数字 60、70、80)被演奏 10 个时间单位,之后紧接着一个音符(数字 62)被演奏 2 个时间单位。

```
0 ON 60
0 ON 70
0 ON 80
10 OFF 60
10 OFF 80
10 OFF 70
10 ON 62
12 OFF 62
```

现有的许多音乐无法被直接转换到这样的程序格式。有些时候一个音符在已经“打开”的情况下被乐谱要求再次奏响。例如:

```
0   ON   60
```

10 ON 60

12 OFF 60

20 OFF 60

一个合成器会将这个程序解释为将音符 60 演奏 12 个时间单元,而不是所给出的 20。因为打开一个已经在发音的音符将被忽略,我们将无法听到时刻 10 时音符演奏的断续。现分析一下开关一盏灯的情况,若其本来是亮的,再次将其打开是无效果的。同样,一盏灯被第一次关闭之后,它就不亮了。当一个已打开的音符被要求再次奏响时,可以通过在第二个打开指令之前一个时间单元插入一个关闭指令来"修正"程序。由于在此种情况下至少已经有两个关闭指令,只有最后一个应当被保留;其它都应被从程序中除去。经过"修正"的程序将使得合成器的表现如同一个音符被极快地连续演奏两次。

程序中存在的另一个问题是在同一个时刻打开和关闭一个音符。根据程序中事件的顺序,或者音符将被过早地关闭(如果关闭指令在打开指令之后出现),或者音符的第二次演奏将无法听出来。例如:

0 ON 60	0 ON 60
10 ON 60	10 OFF 60
10 OFF 60	10 ON 60
20 OFF 60	20 OFF 60

在左边的例子中,音符将在时刻 10 被关闭。在右边的例子中,音符没有被关闭足够长的时间,使得听众无法觉察到声音的"停顿"。这两种情况的解决办法是一样的:移动关闭指令以使得其由合成器在相应的打开指令之前一个时间单位执行。

如果"修正"之后在同一个音符的打开指令 A 之前一个时间单位插入的关闭指令 B 与此前的一个打开指令 C 同时发生,则应当从程序中去除打开指令 A 及与其同时发生的关闭指令,关闭指令 B 也应除去。

写一个程序读入任意数目的 MIDI 程序,并用上述方法"修正"之。

1. 输入

每个程序的行数是任意的。每行依次包括指令被送到合成器的时间(一个非负整数)、一个指令(ON 或 OFF)以及一个音符(一个在 1 到 127 之间的整数)。这些项之间用一个或多个空格隔开。除了最后一个程序之外,每个程序都由仅包含 - 1 的一行结束。最后一个程序由仅包含 - 2 的一行结束。

2. 输出

输出是经过"修正"的程序,格式同输入相同。

假定

(1) ON 和 OFF 指令总是用大写字符给出。

(2) 程序中的时间按非递减顺序排列。

(3) 初始时所有的音符都是关闭的。

(4) 如果不同的音符被同时打开或关闭,相应的指令出现的顺序是无关紧要的。

(5) 程序中每个打开指令之后都有相应的关闭指令。

3. 样例输入

```
0 ON 60
10 ON 60
12 OFF 60
20 OFF 60
-1
0 ON 60
5 ON 70
10 ON 60
10 OFF 60
15 OFF 70
15 ON 70
20 OFF 60
20 OFF 70
-1
0 ON 60
1 OFF 60
1 ON 60
10 OFF 60
-2
```

4. 样例输出

```
0 ON 60
9 OFF 60
10 ON 60
20 OFF 60
-1
0 ON 60
5 ON 70
9 OFF 60
10 ON 60
14 OFF 70
15 ON 70
20 OFF 60
20 OFF 70
-1
0 ON 60
10 OFF 60
-2
```

5.6 魔　　板

30 年前流行的一种儿童玩的魔板游戏由一个包括 24 个大小一样的小正方形板的 5×5底板组成。在每一个小正方形板上印有一个不同的字母。由于底板内只有 24 个方板,底板还包括了一个同小正方形板大小相同的空位。如果一个方板紧挨着空位的右面、左面、上面或下面,则其可以移动至空位。魔板游戏的目的是通过将方板移入空位置以使得底板上的字母按字母表顺序排列。

图 5.4(a)代表了一个魔板初始的局面,图 5.4(b)是顺序移动 6 步之后的局面。

图　5.4

6 步移动的顺序是:

(1) 空位上方的方板移动;

(2) 空位右方的方板移动;

(3) 空位右方的方板移动;

(4) 空位下方的方板移动;

(5) 空位下方的方板移动;

(6) 空位左方的方板移动。

写一个程序,在给出了初始局面和移动序列之后,显示最终的底板局面。

1. 输入

你的程序的输入由若干个魔板组成,每一个魔板均由初始局面和魔板移动序列来描述。每个魔板描述的开始 5 行是初始局面,后继行给出了移动的序列。

底板显示的第一行对应魔板顶行的方板,其余行也依次对应。底板中的空位用一个空格表示。每行恰包含 5 个字符,由最左面的方板上的字母(如果最左面恰好是空位,则用一个空格)开始。底板显示将对应于一个合法的魔板。

移动的序列由表示移动至空位的方板的 A、B、L、R 所组成的序列来表示。A 表示空位上方的方板移动;B 表示空位下方的方板移动;L 表示空位左方的方板移动;R 表示空位右方的方板移动。即使用 4 个移动字符之一来表示,移动仍可能会是非法的。如果发

生了一次非法移动,魔板被认为没有最终局面。这个移动序列可以延至若干行,但总是用数字 0 结束。数据的结束用字符 Z 表示。

2. 输出

每个魔板的输出由一个适当标记的编号开始(Puzzel # 1, Puzzel # 2,等等)。如果魔板没有最终局面,接下来应该给出一个相应的信息。否则应该显示最终局面。

最终局面的每一行的格式应使得在相邻的两个字母之间有一个空格,把空位也看作是一个字母。例如,如果空格在底板里面,它将以 3 个空格的序列的形式出现——一个将其同左边的方板隔开,一个是空位自身,还有一个将其同右边的方板隔开。

至少用一个空行隔开不同的魔板记录。

下面是样例输入以及相应的正确输出。第一个记录对应于前页图中的魔板。

3. 样例输入

```
TRGSJ
XDOKI
M VLN
WPABE
UQHCF
ARRBBL0
ABCDE
FGHIJ
KLMNO
PQRS
TUVWX
AAA
LLLL0
ABCDE
FGHIJ
KLMNO
PQRS
TUVWX
AAAAABBRRRLL0
Z
```

4. 样例输出

```
Puzzle # 1:
 T  R  G  S  J
 X  O  K  L  T
 M  D  V  B  N
 W  P     A  E
```

U Q H C F

Puzzle # 2:

A B C D
F G H I E
K L M N J
P Q R S O
T U V W X

Puzzle # 3:

This puzzle has no final configuration .

5.7 资源分配

为了增加编程小组的生产能力,一家软件开发公司愿意雇用新的程序员以及在软硬件系统上投入更多的资金。由于找不出更好的衡量标准,管理层定义小组生产能力的增量为该小组生产的“代码行数的增量”。公司需要一个资源分配模型以决定如何在各小组之间分配资金及新程序员,从而使得生产能力的总的增量最大。

每一个编程小组在使用新资源的效率上有一定限制。例如,某个小组可以有效地使用 0、3、5、6 个新程序员(该小组内部的人事组织关系使得它无法使用 1、2、4、7 个或更多的新程序员。)这就为将新程序员分配至该小组提供了 4 种选择。对于该小组只有 3 种不同的分配追加资金投入的选择。因此,在这个例子中共有 12 种可能的分配方案。对于每个方案,公司已经预计了该小组生产的代码行数的增量。

你必须写一个程序以精确地建议如何在小组间分配资源。对于每个小组,你的程序必须确定要分配的新程序员和资金的数目。新程序员和资金的分配必须要使得总生产能力增量——所有小组代码行数的增量之和达到最大。分配的程序员的总数不得超过公司愿意雇佣的程序员的数目。分配的资金总额不得超过整个公司的预算总额。在有多种最优方案时,你的程序只需建议其中任意一种即可。

1. 输入

你的程序的输入由若干分配问题组成。所有的输入数据都是非负整数。每个分配问题的输入的开头 3 行由下列组成:

d——编程小组的数目($0 < d \leq 20$,除非 d 是文件结束标志);

p——新程序员的总数;

b——新的计算机资源的资金预算总额。

在这 3 行之后是每个编程小组的输入记录。第一个记录对应于小组 1,第二个记录对应于小组 2,等等。每个小组记录按如下格式组织:

n——新程序员选择的数目($0 \leq n \leq 10$);

$x_1\ x_2 \cdots x_n$——新程序员选择的列表(数字之间用空格隔开);

k——新预算选择的数目($0 < k < 10$);

$b_1\ b_2 \cdots b_n$——每个新预算选择的费用(用空格隔开);

$n \times k$ 的整数列表——列表的第(i, j)个元素是分配 x_i 个新程序员和 b_j 的追加预算之后代码行数的增量。

允许为任何小组分配 0 个新程序员和在新的软硬件上追加投入 0 元——结果是,该小组的生产能力没有任何提高。这样的无效分配要被明确地说明。

每个分配问题从新的一行开始。一个包含 0 个小组的分配‘ 问题 ”表示了输入的结束。在这行之后没有其它输入行。

2. 输出

每个问题的输出由标识所解决问题编号(问题 # 1,问题 # 2,等等)的一行开始。其后是一个空行,之后有 3 行分别给出了所用资金的总额、雇佣的新程序员总数,以及最优资源分配方案所预期创造的新生产能力的总和。

接下来是每个小组的输出。第一行用数字标识了该小组,下面的 3 行给出了小组的预算、小组的新程序员数,以及预期的代码行数的增量。在下一个小组的输出之前应有一个空行,相继问题的输出之间应有两个空行。输出的格式不一定要非常精确,但所有的输出必须是易读和清晰的。

下面是一个包括一个完整的分配问题的样例输入文件。在这个问题里有 3 个编程小组。公司愿意雇佣最多 10 个新程序员并在新计算机资源上最多花 90 000 元。对于小组 1,在新计算机资源上花费 50 000 元并分配 6 个新程序员将引起生产能力上 40 000 行代码的增量。

3. 样例输入

```
3
10
90000
4
0   2   5   6
4
0   20000   50000   70000
0   10000   20000   50000
60000   20000   10000   40000
20000   10000   30000   40000
30000   10000   40000   30000
5
0   1   3   4   8
3
0   40000   80000
```

```
0   50000   30000
50000   40000   60000
20000   30000   50000
80000   90000   50000
30000   40000   70000
3
0   4   6
5
0   50000   30000   40000   50000
0   30000   50000   60000   30000
10000   20000   30000   40000   50000
20000   30000   40000   50000   60000
0
```

4. 样例输出

```
Optimal resource allocation problem #1

Total budget: $80000
Total new programmers: 6
Total productivity increase: 210000

Division #1 resource allocation:
Budget: $0
Programmers: 2
Incremental lines of code: 60000

Division #2 resource allocaiton:
Budget: $40000
Programmers: 4
Incremental lines of code: 90000

Division #3 resource allocation:
Budget: $40000
Programmers: 0
Incremental lines of code: 60000
```

第 6 章　解　答　I

6.1　消　防　车

1. 解题思路

将问题抽象为图论中求两点间所有不同的路径。可以用一个邻接矩阵来表示街区之间的邻接信息。

由于问题中要求所有可能的救火路径,所以宜采用回溯算法求解。在实现中,由于街区的数目比较小(不超过 21),所以采用递归实现比较方便简洁。

2. 程序清单

```
{$A+,B-,D-,E-,F-,G+,I-,L-,N-,O-,P-,Q-,R-,S-,T-,V-,X-,Y-}
{$M 65520,0,655360}
program Firetruck;                                  {消防车}

  type
    TStreetCornerSet = set of 1..20;                {街区集合类型}

  var
    CaseNumber,                                     {数据项编号}
    FirePlace:byte;                                 {火警地点}
    City: array[1..20,1..20] of boolean;            {城市街区连通矩阵}

procedure InputData;                                {读入数据项}

  var
    StreetCornerl,StreetCorner2: byte;

  begin
    readln(FirePlace);
    fillchar(City,sizeof(City),0);

    readln(StreetCornerl,StreetCorner2);
    while(StreetCornerl<>0) or (StreetCorner2<>0) do
      begin
```

```
            City[StreetCorner1,StreetCorner2]:= true;
            City[StreetCorner2,StreetCorner1]:= true;
            readln(StreetCorner1,StreetCorner2);
          end;
        inc(CaseNumber);
      end;

  procedure SearchRoute;                          {寻找救火路线}

    var
      RouteCount:longint;                         {路线数目}

      Route:array[1..20] of byte;                 {路线}

    procedure Go (Depth: byte; StreetCornerSet: TStreetCornerSet);
                                                  {递归搜索路线}
                                                  {Depth 当前深度}
                                                  {StreetCornerSet}
                                                  {已走街区集合}

    var
      I:byte;

    begin
      if Route[Depth - 1] = FirePlace then        {如果已到达火警地点}
        begin                                     {则又找到一条救火路线}
          inc(RouteCount);
          for I:= 1 to Depth - 1 do
            write(Route[I],' ');
          writeln;
          exit;
        end;

      for I:= 1 to 20 do                          {对于所有的街区}
        if not (I in StreetCornerSet) and         {如果尚未走过}
          City[Route[Depth - 1],I] then           {且从当前街区可到达之}
          begin                                   {则走下去}
            Route[Depth]:= I;
            Go(Depth + 1,StreetCornerSet + [I]);
          end;
      end;
```

```
begin {SearchRoute}
    writeln('Case',CaseNumber);
    RouteCount:=0;
    Route[1]:=1;                                    {从街区 1 出发}
    Go(2,[1]);                                      {递归搜索路线}
    writeln('There are',RouteCount,'routes from the firestation to streetcorner',
            FirePlace,'.');
  end;{SearchRoute}

begin {Main}

  assign(input,'ACM91FA .IN');
  assign(output,'ACM91FA .OUT');
  reset(input);
  rewrite(output);

  CaseNumber:=0;

  while not eof do                                  {当还有数据项时}
    begin
      InputData;                                    {读入数据项}
      SearchRoute;                                  {寻找救火路线}
    end;

  close(input);
  close(output);

end .{Main}
```

6.2 数字三角形

1. 解题思路

可以证明可接受的图形只有 6 种基本情况,见图 6.1。

所有的可接受的图形都是上述 6 种基本图形之一。

令这些基本图形的边长为 1,称之为单位图形。

定义一个网格中的点的行列坐标:行坐标是该点的行号,列坐标是该点在其所在行上自左向右数的位置。

图 6.1

例如点 9 的坐标是(4,3)。对于点 N,令其坐标为(X,Y),则 X*(X-1)/2<N<=X*(X+1)/2,因此 X=round(sqrt(8*(N-1)+1)+1) div2,而 Y=N-X*(X-1)/2。

所有可接受的图形的顶点相对于其左上角顶点(即图形中数值最小的顶点)坐标的偏

移量应当是某个与其具有相同顶点数的单位图形相应顶点之间偏移量的整数倍。

因此可以将这 6 种单位图形用常量定义在程序里,对于输入的每个点集所代表的图形,将其同这 6 种单位图形进行匹配,如果顶点数相同且相应顶点相对于各自左上角顶点坐标的偏移量成比例,则该点集代表了一个可接受的图形。

2. 程序清单

```
{$A+,B-,D-,E-,F-,G+,I-,L-,N-,O-,P-,Q-,R-,S-,T-,V-,X-,Y-}
{$M 65520,0,655360}
program Triangular_Vertices;                    {数字三角形}

  const
    FigureCount = 6;                            {图形数目}

  type
    TOutLine = array[1..6,1..2] of byte;        {图形轮廓类型}

    TFigure =                                   {图形类型}
      record
        OutLine:TOutLine;                       {图形轮廓}
        VerticeCount:byte;                      {图形顶点数目}
        Name:string[20];                        {图形名称}
      end;

    TVertice =                                  {顶点类型}
      record
        Point,                                  {顶点数字}
        X,Y:word;                               {顶点的行、列坐标}
      end;

  const
    Figure:array[1..FigureCount] of Tfigure     {6 种单位图形:}
                                                {其中各点坐标是相对于}
                                                {左上角顶点的偏移量}
     =((OutLine:((0,0),(0,1),(1,0),(0,0),(0,0),(0,0));VerticeCount:3;Name:'triangle'),
     (OutLine:((0,0),(1,0),(1,1),(0,0),(0,0),(0,0));VerticeCount:3;Name:'triangle'),
     (OutLine:((0,0),(1,0),(1,1),(2,1),(0,0),(0,0));VerticeCount:4;Name:'parallelogram'),
     (OutLine:((0,0),(0,1),(1,1),(1,2),(0,0),(0,0));VerticeCount:4;Name:'parallelogram'),
     (OutLine:((0,0),(0,1),(1,0),(1,1),(0,0),(0,0));VerticeCount:4;Name:'parallelogram'),
     (OutLine:((0,0),(0,1),(1,0),(1,2),(2,1),(2,2));VerticeCount:6;Name:'hexagon'));

  var
```

```
  VerticeCount:integer;                          {顶点数目}
  Vertice: array[0..20] of TVertice;             {顶点}

procedure InputData;                             {读入顶点集合}

  begin
    VerticeCount:= 0;
    while not eoln do
      begin
        inc(VerticeCount);
        with Vertice[VerticeCount] do
          begin
            read(Point);
            X:= trunc(sqrt(1 + 8* (Point - 1)) + 1) div 2;
            Y:= Point - X* (X - 1) div 2;
          end;
      end;
    readln;
  end;

procedure CheckFigure;                           {匹配图形}

  var
    I,J,K:byte;
    Ratio:word;                                  {边长比例}
    Flag:boolean;                                {匹配成功标志}

  begin
    for K:= 1 to Vertice Count do
      write(Vertice[K].Point,' ');

    for I:= 1 to VerticeCount - 1 do             {将顶点从小到大排序}
      for J:= I + 1 to VerticeCount do
        if Vertice[I].Point > Vertice[J].Point then
          begin
            Vertice[0]:= Vertice[I];
            Vertice[I]:= Vertice[J];
            Vertice[J]:= Vertice[0];
          end;

    for K:= VerticeCount downto 1 do             {计算各点相对于}
                                                 {左上角顶点的偏移量}
```

```
        begin
          dec(Vertice[K].X,Vertice[1].X);
          dec(Vertice[K].Y,Vertice[1].Y);
        end;

      Flag:= false;
      for K:= 1 to FigureCount do                    {对于所有单位图形进行匹配}
        if Figure[K].VerticeCount = VerticeCount then
                                                     {首先应顶点数相同}
          with Figure[K] do
            begin
              I:= 1;
              while OutLine[I,1] = 0 do
                inc(I);
              Ratio:= Vertice[I].X div OutLine[I,1];
                                                     {计算边长比例}
              Flag:= true;
              for I:= 2 to VerticeCount do           {逐点检查}
                if (Vertice[I].X < > OutLine[I,1]* Ratio) or
                  (Vertice[I].Y < > OutLine[I,2]* Ratio) then
                  begin
                    Flag:= false;
                    break;
                  end;
              if Flag then                           {如果匹配成功则跳出循环}
                break;
            end;

      if Flag
        then
          writeln('are the vertices of a',Figure[K].Name)
        else
          writeln('are not the vertices of an acceptable figure');
    end;

  begin {Main}
    assign(input,'ACM91FB.IN');
    assign(output,'ACM91FB.OUT');
    reset(input);
    rewrite(output);

    while not eof do                                 {当还有顶点集合时}
```

```
    begin
      InputData;                                {读入顶点集合}
      CheckFigure;                              {匹配图形}
    end;

  close(input);
  close(output);
end.{Main}
```

6.3 透 视 仪

1. 解题思路

这类题目最自然的想法是,用回溯法从上至下、从左至右依次搜索切片中每个元素是否是实心的。但这种算法的时间效率又如何呢?可能有人会认为,由于矩阵有 10×15 共 150 个元素,即搜索层次有 150 层,因此该算法不可取。其实不然,搜索层次虽多,但由于剪枝条件很强(每个传感器的读数),实际运行效果是完全可以接受的。

本题还有从理论上可以保证更优的算法,感兴趣的读者可参考本丛书的另一册《美国青少年信息学奥林匹克 USACO》。

2. 程序清单

```
{$A+,B-,D-,E-,F-,G+,I-,L-,N-,O-,P-,Q-,R-,S-,T-,V+,X-,Y-}
{$M 65520,0,655360}

program Scanner;                                {透视仪}

  var
    SliceCount,                                 {成像切片数目}
    SliceNumber:byte;                           {成像切片编号}
    Array1:array[1..10] of byte;                {第一组传感器}
    Array2:array[1..24] of byte;                {第二组传感器}
    Array3:array[1..15] of byte;                {第三组传感器}
    Array4:array[1..24] of byte;                {第四组传感器}
    Slice:array[1..10,1..15] of char;           {切片}

  function Min(A,B:integer):integer;            {最小值函数}

    begin
      if A<B
        then
```

```
            Min:= A
        else
            Min:= B;
    end;

function Max(A,B:integer):integer;                    {最大值函数}

    begin
      if A > B
        then
          Max:= A
        else
          Max:= B;
    end;

procedure InputData;                                  {读入传感器信息}
    var
      I:byte;

    begin
      for I:= 1 to 10 do
        read(Array1[I]);
      readln;
      for I:= 1 to 24 do
        read(Array2[I]);
      readln;
      for I:= 1 to 15 do
        read(Array3[I]);
      readln;
      for I:= 1 to 24 do
        read(Array4[I]);
      readln;
    end;

procedure ConstructSlice;                             {构造成像切片}

    var
      NowArray1:array[1..10] of byte;
      NowArray2:array[1..24] of byte;
      NowArray3:array[1..15] of byte;
      NowArray4:array[1..24] of byte;
      Flag:boolean;                                   {构造成像切片成功标志}
```

```
procedure Try(X,Y:byte);                    {按行优先顺序递归构造}
                                            {成像切片的每一个元素}
                                            {X,Y 当前要决定的元素}

  begin
    if Y > 15 then                          {如果已构造到一行的末尾了}
      begin                                 {则从下一行开始构造}
        inc(X);
        Y:= 1;
      end;
    if X > 10 then                          {如果已全部构造}
    begin                                   {则返回}
      Flag:= false;
      exit;
    end;

  if (NowArray1[X] < Array1[X] and (NowArray3[Y] < Array3[Y])and
    (NowArray2[X + Y - 1] < Array2[X + Y - 1]) and
    (NowArray4[10 - X + Y] < Array4[10 - X + Y]) then
                                            {如果还未构造出成像切片}
                                            {且当前元素可以是实心的}
                                            {则搜索下去}
  begin
    Slice[X,Y]:= '# ';
    inc(NowArray1[X]);
    inc(NowArray3[Y]);
    inc(NowArray2[X + Y - 1]);
    inc(NowArray4[10 - X + Y]);

    Try(X,Y + 1);

    if Flag then
      begin
        Slice[X,Y]:= '.';
        dec(NowArray1[X]);
        dec(NowArray3[Y]);
        dec(NowArray2[X + Y - 1]);
        dec(NowArray4[10 - X + Y]);
      end;
  end;
```

```
    if Flag and (NowArray1[X] + 15 - Y > = Array 1[X]) and
       (NowArray3[Y] + 10 - X > = Array3[Y]) and
       (NowArray2[X + Y - 1] + Min(10 - X, Y - 1) > = Array2[X + Y - 1]) and
       (NowArray4[10 - X + Y] + Min(10 - X,15 - Y) > = Array4[10 - X + Y]) then
                                                    {如果还未构造出成像切片}
                                                    {且当前元素可以是空心的}
       Try(X, Y + 1)                                {则搜索下去}
    end;

  begin {ConstructSlice}
    fillchar(Slice, sizeof(Slice), '.');
    fillchar(NowArray1, sizeof(NowArray1), 0);
    fillchar(NowArray2, sizeof(NowArray2), 0);
    fillchar(NowArray3, sizeof(NowArray3), 0);
    fillchar(NowArray4, sizeof(NowArray4), 0);

    flag:= true;
    Try(1, 1);                                      {从第一行第一列的元素}
                                                    {开始构造成像切片}
    end;{ConstructSlice}

  procedure Print;                                  {打印输出成像切片}

    var
      I, J:byte;

    begin
      for I:= 1 to 10 do
        begin
          for J:= 1 to 15 do
            write(Slice[I, J]);
          writeln;
        end;
      writeln;
    end;

  begin {Main}
    assign(input, 'ACM93FH .IN');
    assign(output, 'ACM93FH .OUT');
    reset(input);
    rewrite(output);
```

```
  readln(SliceCount);

  for SliceNumber:= 1 to SliceCount do          {对于每一个成像切片}
    begin
      InputData;                                {读入传感器信息}
      ConstructSlice;                           {构造成像切片}
      Print;                                    {打印输出成像切片}
    end;

  close(input);
  close(output);
end.{Main}
```

6.4 多米诺效应

1. 解题思路

由于题目要求所有可能解,所以这道题目应用回溯算法求解。由于题中要求每张骨牌用且仅用一次,故利用回溯算法求解的效率是可以接受的。搜索时按行优先顺序依次决定每一个位置上的骨牌号及骨牌摆放方向。

2. 程序清单

```
{$A+,B-,D-,E-,F-,G+,I-,L-,N-,O-,P-,Q-,R-,S-,T-,V-,X-,Y-}
{$M 65520,0,655360}
program The_Domino_Effect;                      {多米诺效应}

  const
    Dx:array[1..4] of shortint=(1,-1,0,0);
    Dy:array[1..4] of shortint=(0,0,1,-1);

    Bone:array[0..6,0..6] of byte               {骨牌对应表}
      =((1,2,3,4,5,6,7),
        (2,8,9,10,11,12,13),
        (3,9,14,15,16,17,18),
        (4,10,15,19,20,21,22),
        (5,11,16,20,23,24,25),
        (6,12,17,21,24,26,27),
        (7,13,18,22,25,27,28));

  type
    TBoneSet=set of 1..28;                      {骨牌集合类型}
```

```
  var
    Layout:array[1..7,1..8] of byte;          {点数网格}
    LayoutNumber:byte;                        {数据项编号}

procedure InputData;                          {读入数据项}

  var
    I,J:byte;

  begin
    for I:= 1 to 7 do
      begin
        for J:= 1 to 8 do
          read(Layout[I,J]);
        readln;
      end;

    inc(LayoutNumber);
  end;

procedure PrintLayout;                        {打印点数网格}

  var
    I,J:byte;

  begin
    writeln('Layout #',LayoutNumber);
    writeln;

    for I:= 1 to 7 do
      begin
        for J:= 1 to 8 do
          write(Layout[I,J]:4);
        writeln;
      end;

    writeln;
  end;

procedure SearchMap;                          {寻找骨牌号图}
  var
```

```
  SolutionCount:longint;                          {图的总数}
  Map:array[1..7,1..8] of byte;                   {骨牌号图}

procedure Search (Position:byte;BoneSet;TBoneSet);
                                                  {递归搜索骨牌号图}
                                                  {Position 待确定网格位置}
                                                  {BoneSet 待放置骨牌集合}

var
  I,J:byte;

begin
  if BoneSet = [ ]then                            {如果骨牌已全部放置}
    begin                                         {则找到新的骨牌号图}
      inc(SolutionCount);
      if SolutionCount = 1 then
        begin
          writeln('Map resulting from layout #',LayoutNumber,'are:');
          writeln;
        end;
      for I:= 1 to 7 do
        begin
          for J:= 1 to 8 do
            write(Map[I,J]:4);
          writeln;
        end;
      writeln;
      exit;
    end;

  while (Position<=56) and
        (Map[(Position-1) div 8+1,(Position-1) mod 8+1]<>0) do
                                                  {找到第一个待确定网格位置}
                                                  {(按行优先顺序)}
    inc(Position);

  I:=(Position-1) div 8+1;                        {计算行坐标}
  J:=(Position-1) mod 8+1;                        {计算列坐标}
  if(J<8) and (Map[I,J+1]=0) and
    (Bone[Layout[I,J],Layout[I,J+1]] in BoneSet) then
                                                  {如果可以横向放置骨牌}
    begin                                         {则尝试之}
```

```
            Map[I,J]:= Bone[Layout[I,J],Layout[I,J+1]];
            Map[I,J+1]:= Map[I,J];
            Search(Position+1,BoneSet-[Map[I,J]]);
            Map[I,J]:= 0;
            Map[I,J+1]:= 0;
          end;

        if(I<7) and (Map[I+1,J]=0) and
          (Bone[Layout[I,J],Layout[I+1,J]] in BoneSet) then
                                                {如果可以纵向放置骨牌}
          begin                                 {则尝试之}
            Map[I,J]:= Bone[Layout[I,J],Layout[I+1,J]];
            Map[I+1,J]:= Map[I,J];
            Search(Position+1,BoneSet-[Map[I,J]]);
            Map[I,J]:= 0;
            Map[I+1,J]:= 0;
          end;
        end;

      begin{SearchMap}
        SolutionCount:= 0;
        fillchar(Map,sizeof(Map),0);
        Search(1,[1..28]);                      {递归搜索骨牌号图}
        writeln('There are',SolutionCount,'solution(s) for layout #',LayoutNumber,'.');
        writeln;
        writeln;
        writeln;
      end;{SearchMap}

    begin{Main}
      assign(input,'ACM91FD.IN');
      assign(output,'ACM91FD.OUT');
      reset(input);
      rewrite(output);
      LayoutNumber:= 0;
      while not eof do                          {当还有数据项时}
        begin
          InputData;                            {读入数据项}
          PrintLayout;                          {打印点数网格}
          SearchMap;                            {寻找骨牌号图}
        end;
```

```
  close(input);
  close(output);
end.{Main}
```

6.5 医院设备的利用

1. 解题思路

这是一道模拟类的试题。如果对问题不作深入分析,很可能会立即动手编程,将病人进入手术室和恢复室的模拟合在一次完成。但这样处理比较繁琐,并不是最好的方法。

仔细分析题意之后,可以将问题分为两个阶段:病人进入手术室和病人进入恢复室,后一个阶段对前一个阶段没有影响。因此应先模拟病人进入手术室的情况,并得到病人进入恢复室的顺序,然后再类似地模拟病人进入恢复室的情况,这样就完成了一天运行情况的模拟。

病人进入手术室和恢复室的顺序可以用队列来实现,但在具体实现上,由于操作较为简单,所以用两个一维数组来存放,操作也很方便。

病人进入手术室和恢复室的过程分别用两个循环实现。

2. 程序清单

```
{$A+,B-,D-,E-,F-,G+,I-,L-,N-,O-,P-,Q-,R-,S-,T-,V-,X-,Y-}
{$M 65520,0,655360}
program Use_of_Hospital_Facilities;                {医院设备利用}
                                                   {程序中所有的时间和时}
                                                   {刻都以分钟计}
type
  TPlace =                                         {地点类型}
                                                   {(手术室,恢复室床位)}
    record
      TotalTime,                                   {总利用时间}
      FreeTime:word;                               {开始空闲时刻}
    end;

  TPatient =                                       {病人类型}
    record
      Name:string[8];                              {病人姓名}
      RoomNumber,                                  {病人进入的手术室编号}
      BedNumber:byte;                              {病人进入的恢复室床位编号}
      RoomTime,                                    {病人占用手术室的时间}
      BedTime:word;                                {病人占用恢复室床位的时间}
      RoomEnter,                                   {病人进入手术室的时刻}
```

```
    BedEnter:word;                          {病人进入恢复室床位的时刻}
  end;

var
  RoomCount,BedCount,                       {手术室、恢复室床位的数目}
  PatientCount:byte;                        {病人的数目}
  StartTime,                                {第一次手术开始时刻}
  TransferTime,                             {将病人从手术室}
                                            {移至恢复室所需的时间}
  PrepareRoomTime,                          {准备手术室所需的时间}
  PrepareBedTime,                           {准备恢复室床位所需的时间}
  TimeDuration:word;                        {一天中的可用时间}
  Patient:array[1..100] of TPatient;        {病人}
  Room:array[1..10] of TPlace;              {手术室}
  Bed:array[1..30] of TPlace;               {恢复室床位}

procedure InputData;                        {读入数据}

  var
    I:byte;

  begin
    fillchar(Patient,sizeof(Patient),0);
    fillchar(Room,sizeof(Room),0);
    fillchar(Bed,sizeof(Bed),0);
    readln(RoomCount,BedCount,StartTime,TransferTime,PrepareRoomTime,
           PrepareBedTime,PatientCount);

    for I:=1 to PatientCount do
      with Patient[I] do
        begin
          readln(Name);
          readln(RoomTime,BedTime);
        end;

    StartTime:=StartTime*60;

    for I:=1 to RoomCount do
      Room[I].FreeTime:=StartTime;
    for I:=1 to BedCount do
      Bed[I].FreeTime:=StartTime;
  end;
```

```
procedure Simulate;                                    {模拟运行}

  var
    PatientNumber,I,J,Select:byte;
    PatientList:array[1..100] of byte;                 {病人进入恢复室的顺序列表}

  begin
    fillchar(PatientList,sizeof(PatientList),0);
    for PatientNumber:=1 to PatientCount do            {模拟病人进入手术室的情况}
      with Patient[PatientNumber] do
        begin
          Select:=1;                                   {寻找最早空闲的手术室}
          for I:=2 to RoomCount do
            if Room[I].FreeTime<Room[Select].FreeTime then
              Select:=I;

          RoomNumber:=Select;
          with Room[RoomNumber] do                     {将病人送入手术室}
            begin
            RoomEnter:=FreeTime;
            inc(TotalTime,RoomTime);
            BedEnter:=FreeTime+RoomTime+TransferTime;
            FreeTime:=RoomEnter+RoomTime+PrepareRoomTime;
          end;

        I:=1;                                          {将病人按序插入恢复室列表}
        while (I<PatientNumber) and (Patient[PatientList[I]].BedEnter<=
              Patient[PatientNumber].BedEnter) do
          inc(I);
        for J:=PatientNumber downto I+1 do
          PatientList[J]:=PatientList[J-1];
        PatientList[I]:=PatientNumber;
      end;

  for PatientNumber:=1 to PatientCount do              {模拟病人进入恢复室的情况}
    with Patient[PatientList[PatientNumber]] do
      begin
        for I:=1 to BedCount do                        {寻找病人进入恢复室时}
                                                       {空闲的第一个床位}
          if Bed[I].FreeTime<=BedEnter then
            begin
```

```
                Select:= I;
                break;
              end;

          BedNumber:= Select;
          with Bed[BedNumber] do                    {将病人送入恢复室床位}
            begin
              inc(TotalTime,BedTime);
              FreeTime:= BedEnter + BedTime + PrepareBedTime;
            end;
        end;

      with Patient[PatientList[PatientCount]] do    {计算这天的可用时间}
        TimeDuration:= BedEnter + BedTime - StartTime;
    end;

  procedure Print;                                  {输出模拟运行情况的汇总}
  var
    I:byte;

  procedure WriteTime(Time:word);                   {打印时刻 Time}

    begin
      write(Time div 60:5,':');
      if Time mod 60 < 10 then
        write('0');
      write(Time mod 60);
    end;

  begin {Print}
    writeln('Patient      Operating Room          Recovery Room');
    writeln('#    Name        Room #    Begin     End     Bed #    Begin     End');

    for I:= 1 to PatientCount do                    {打印病人情况列表}
      with Patient[I] do
        begin
          if I < 10
            then
              write(I,'  ')
            else
              write(I,' ');
```

```
            while Name[0] < #8 do
              Name:= Name + ' ';
            write(Name);

            write(RoomNumber:5);
            WriteTime(RoomEnter);
            WriteTime(RoomEnter + RoomTime);

            write(BedNumber:7);
            WriteTime(BedEnter);
            WriteTime(BedEnter + BedTime);

            writeln;
          end;

        writeln;
        writeln('Facility Utilization');
        writeln('Type  # Minutes  % Used');
        writeln('------------');

        for I:= 1 to RoomCount do                    {打印手术室情况列表}
          with Room[I] do
            writeln('Room ',I:2,TotalTime:8,TotalTime* 100/ TimeDuration:8:2);

        for I:= 1 to BedCount do                     {打印恢复室床位情况列表}
          with Bed[I] do
            writeln('Bed  ',I:2,TotalTime:8,TotalTime* 100/ TimeDuration:8:2);
      end;{Print}

    begin{Main}
      assign(input,'ACM91FE.IN');
      assign(output,'ACM91FE.OUT');
      reset(input);
      rewrite(output);

      InputData;                                     {读入数据}
      Simulate;                                      {模拟运行}
      Print;                                         {输出模拟运行情况的汇总}

      close(input);
      close(output);
    end.{Main}
```

6.6 信息解码

1. 解题思路

信息段的结构简图如下:

关键字位数 n(3 位)	关键字 1(n 位)	关键字 2(n 位)	……	段结束标志(n 位)

对于每一个信息段,应首先读入该段中关键字的位数 n,它本身是一个 3 位的二进制数。然后不断读入 n 位的关键字,将其所对应的编码头中的字符输出,直到关键字为段结束标志(全 '1')。

2. 程序清单

```
{$A+,B-,D-,E-,F-,G+,I-,L-,N-,O-,P-,Q-,R-,S-,T-,V-,X-,Y-}
{$M 65520,0,655360}
program Message_Decoding;                    {信息解码}

  const
    MaxCode:array[1..8] of byte              {每种长度的关键字数目}
      =(1,3,7,15,31,63,127,255);
    StartPosition:array[1..8] of byte        {每种长度的关键字映射的第}
      =(1,2,5,12,27,58,121,248);             {一个字符在编码头中的位置}

  var
    CharMap:string;                          {编码头}
    BitLength,                               {关键字长度}
    Code:byte;                               {关键字}

  function InputCode(Count:byte):byte;       {返回一个 Count 位的关键字}

    var
      Result:byte;
      Bit:char;

    begin
      Result:=0;
      while Count>0 do
        begin
          read(Bit);
          while eoln and not eof do
```

```
            readln;
            Result:= Result * 2 + ord(Bit) - ord('0');
            dec(Count);
          end;
        InputCode:= Result;
      end;

  begin {Main}
    assign(input,'ACM91FF.IN');
    assign(output,'ACM91FF.OUT');
    reset(input);
    rewrite(output);

    while not eof do                          {当输入还未结束时}
      begin
        readln(CharMap);                      {读入编码头}

        BitLength:= InputCode(3);             {读入第一段之关键字长度}
        while BitLength > 0 do                {当有待解码信息时}
          begin
            Code:= InputCode(BitLength);      {读入第一个关键字}
            while Code < MaxCode[BitLength] do {当关键字不是段结束标志时}
              begin
                write(CharMap[StartPosition[BitLength] + Code]);
                                              {输出关键字所映射的字符}
                Code:= InputCode(BitLength);  {读入下一个关键字}
              end;
            BitLength:= InputCode(3);         {读入下一段关键字长度}
          end;
        writeln;
      end;

    close(input);
    close(output);
  end.{Main}
```

6.7 代码生成

1. 解题思路

先不考虑代码的优化问题。由于输入给出的表达式是波兰式,所以程序无须考虑表

达式的运算顺序问题。可以用一个栈来存放操作数,计算时从左至右扫描表达式,如果是操作数则入栈,如果是运算符则从栈中取出操作数做相应运算,将运算结果保存在临时存储地址中。这样就可以完成表达式的汇编工作。

现在加入代码的优化。最根本的优化措施是:在载入一个操作数的时候,如果寄存器里存放的正是这个操作数的值,则无须再次载入这个操作数,并且如果这个操作数是一个临时存储地址,则无须保存和使用这个临时存储地址。为了便于实现这一点,将上述算法略作修改,每次运算完后不立刻将计算结果保存在临时存储地址中,而是在下一次载入操作数时,仅当要载入的操作数不同于寄存器所保存的操作数时,才将寄存器的值保存到临时存储地址。

其它的优化措施同运算符紧密相关,但都是基于上述之基本优化的。下面分述之:

对于加法和乘法,由于其满足交换律,操作数 1 和操作数 2 的地位是等价的。因此如果寄存器里存放的是其中任意一个操作数的值,就可以直接将另一个操作数与之进行操作。

对于减法,由于 A - B = A + (- B),所以如果操作数 2 已在寄存器中,则可以将寄存器取反后再加上操作数 1。

2. 程序清单

```
{$A+,B-,D-,E-,F-,G+,I-,L-,N-,O-,P-,Q-,R-,S-,T-,V-,X-,Y-}
{$M 65520,0,655360}
program Code_Generation;                          {代码生成}

  var
    Expression:string;                            {表达式}

  procedure Process (Expression:string);          {生成 Expression 的汇编代码}
                                                  {存储地址用一个字符表示}
                                                  {英文字母表示变量地址}
                                                  {空格表示空地址}
                                                  {其它情况则表示临时存储地}
                                                  {址" $ord(Ch) "}
                                                  {(Ch 是表示存储地址的字符)}

  var
    Stack:array[1..100] of char;                  {操作数栈}
    StackCount:byte;                              {栈顶指针}

  procedure Push (Ch:char);                       {入栈操作}

    begin
      inc(StackCount);
```

```
    Stack[StackCount]:= Ch;
  end;

function Pop:char;                          {返回栈顶元素并退栈}
  begin
    Pop:= Stack[StackCount];
    dec(StackCount);
  end;

function GetName(V:char):string;            {返回存储地址的引用标识}

  var
    S:string;

  begin
    if V< 'A'
      then
        begin
          str(ord(V),S);
          GetName:= '$'+S;
        end
      else
        GetName:= V
  end;

var
  TempStorage,                              {临时存储空间地址指针}
  Operator1,Operator2,                      {操作数 1、2}
  Now:char;                                 {当前寄存器对应的存储地址}
  I:byte;

procedure LoadOperator (Opr:char);          {载入操作数 Opr}
  begin
    if Now< > Opr then                      {如果 Opr 不在寄存器中}
      begin                                 {则进行载入}
        if Now < > ' ' then                 {如果寄存器存放着操作数}
          writeln('ST',GetName(Now));       {则先保存之}
        writeln('L',GetName(Operator));     {载入操作数}
      end;
  end;

procedure ApplyTemp;                        {申请临时存储地址存放结果}
```

```
    begin
      inc(TempStorage);
      Now:= TempStorage;
    end;

  procedure FreeTemp(Opr:char);                    {释放操作数 Opr}
                                                   {所占用的临时存储地址}
    begin
      if Opr < 'A' then
        dec(TempStorage);
    end;

  begin {Process}
    StackCount:= 0;
    Now:= '';
    TempStorage:= #0;

    for I:= 1 to length(Expression) do             {逐一分析表达式的每一字符}
      case Expression[I] of
        '/':                                       {如果是除法运算符}
          begin
            Operator2:= Pop;                       {取得操作数}
            Operator1:= Pop;

            LoadOperator(Operator1);               {载入操作数 1}

            writeln('D', GetName(Operator2));      {除以操作数 2}
            FreeTemp(Operator1);                   {释放临时存储地址}
            FreeTemp(Operator2);

            ApplyTemp;                             {申请临时存储地址存放结果}

            Push(Now);                             {操作数入栈}
          end;

        '+','*':                                   {如果是加法、乘法运算符}
          begin
            Operator2:= Pop;
            Operator1:= Pop;

            if Now = Operator2                     {如果操作数 2 已在寄存器中}
              then                                 {则直接与操作数 1 作运算}
                begin
                  if Expression[I] = '+'
                    then
                      writeln('A', GetName(Operator1))
```

```
            else
              writeln('M',GetName(Operator1))
          end
        else                                    {否则载入操作数 1 后再运算}
          begin
            LoadOperator(Operator1);
            if Expression[I] = '+'              {与操作数 2 作运算}
              then
                writeln('A',GetName(Operator2))
              else
                writeln('M',GetName(Operator2))
          end;

      FreeTemp(Operator1);
      FreeTemp(Operator2);

      ApplyTemp;

      Push(Now);
    end;
  '-':                                          {如果是减法运算符}
    begin
      Operator2:=Pop;
      Operator1:=Pop;

      if Now=Operator2                          {如果操作数 2 已在寄存器中}
        then                                    {则取反后与操作数 1 相加}
          begin
            writeln('N');
            writeln('A',GetName(Operator1));
          end
        else                                    {否则载入操作数 1 后}
                                                {再作减法}
          begin
            LoadOperator(Operator1);
            writeln('S',GetName(Operator2));
          end;

      FreeTemp(Operator1);
      FreeTemp(Operator2);

      ApplyTemp;

      Push(Now);
    end;
```

```
            '@':                                    {如果是取反运算符}
              begin
                Operator1:= Pop;

                LoadOperator(Operator1);

                writeln('N');                       {取反}

                FreeTemp(Operator1);
                ApplyTemp;

                Push(Now);
              end;

            else                                    {否则则是操作数}
              Push(Expression[I]);                  {操作数入栈}
          end;

        writeln;
    end;{Process}

begin {Main}
    assign(input,'ACM91FG.IN');
    assign(output,'ACM91FG.OUT');
    reset(input);
    rewrite(output);

    while not eof do                                {当还有表达式时}
        begin
            readln(Expression);                     {读入表达式}
            Process(Expression);                    {生成表达式汇编代码}
        end;

    close(input);
    close(output);
end.{Main}
```

第 7 章　解　答　II

7.1　电子表格计算器

1. 解题思路

本题有两个难点:

(1) 如何将一个表达式串转换为一个按因子存储的表达式;

(2) 如何计算多个有互相引用关系的表达式的值。

对于第一个问题,由题意可知,每一个表达式串可分离为若干个因子的和。如图 7.1 所示。

图　7.1

对于第二个问题,我们可采用一个常用的方法,即反复计算每一个可以计算出结果的表达式(即其每一个元素引用的值均已求出),直至全部表达式均计算完(输出表格结果),或所剩下的表达式均不能计算(即存在循环引用,输出无解信息)为止。

2. 程序清单

```
{$A+,B-,D-,E-,F-,G+,I-,L-,N-,O-,P-,Q-,R-,S-,T-,V+,X-,Y-}
{$M 65520,0,655360}
program Spreadsheet_Calculator;                    {电子表格计算器}

  type
    TReference =                                   {元素引用标识类型}
      record
        Row, Column, Sign: shortint;               {行、列、符号}
      end;

    TCell =                                        {元素类型}
```

```
    record
      Value: integer;                          {元素的值}
      Reference: array[1..30] of TReference;   {元素引用}
      RefCount: byte;                          {元素引用数目}
      OriginExp: string[75];                   {初始表达式}
    end;

var
  M, N: byte;                                  {行、列数}
  Cell: array[1..20,1..10] of TCell;           {电子表格}

function InputData: boolean;                   {读入电子表格}
                                               {并返回输入结束信息}

  var
    Temp, Code: integer;
    I, J: byte;
    S, T: string;

  function GetFirst (var S: string):string;    {返回 S 串的第一项}

    var
      I: byte;

    begin
      I:= 2;
      while(I<=length(S))and(S[I]<>'+')and(S[I]<>'-')do
        inc(I);
      GetFirst:=copy(S,1,I-1);
      delete(S,1,I-1);
    end;

  begin {InputData}
    readln(M,N);

    if(M=0)and(N=0)then
      begin
        InputData:=false;
        exit;
      end;

    fillchar(Cell,sizeof(Cell),0);
```

```
    for I:= 1 to M do
      for J:= 1 to N do
        begin
          readln(S);

          while S[length(S)] = ''do
            dec(S[0]);
          Cell[I,J].OriginExp:= S;

          T:= GetFirst(S);
          with Cell[I,J]do
            while T<>''do
              begin
                val(T,Temp,Code);
                if Code<>0
                  then
                    begin
                      inc(RefCount);
                      if pos('-',T)>0
                        then
                          Reference[RefCount].Sign:= -1
                        else
                          Reference[RefCount].Sign:=1;

                      Reference[RefCount].Row:= ord(T[length(T)-1])-ord('A')+1;
                      Reference[RefCount].Column:= ord(T[length(T)])-ord('0')+1;
                    end
                  else
                    inc(Value,Temp);

                T:= GetFirst(S);
              end
        end;

    InputData:= true;
  end;{InputData}

procedure Calc;                                    {计算电子表格}

  var
    I,J,K,L:byte;
    Flag:boolean;                                  {可继续计算标志}
```

```
  begin
    repeat
      Flag:= false;

      for I:= 1 to M do                          {对于每一个元素}
        for J:= 1 to N do
          with Cell[I,J]do
            begin
              K := 1;
              while K < = RefCount do            {对其每一个元素引用}
                if Cell[Reference[K].Row,Reference[K].Column].RefCount = 0
                                                 {如果其引用的元素已计算完}
                  then                           {则累加引用的元素}
                    begin
                      inc(Value,Reference[K].Sign *
                          Cell[Reference[K].Row,Reference[K].Column].Value);

                      for L := K to RefCount - 1 do
                        Reference[L]:= Reference[L + 1];
                      dec(RefCount);

                      Flag := true;
                    end
                  else
                    inc(K);
            end;
    until not Flag;
  end;

function AllDone:boolean;                        {返回电子表格是否已计算完}
  var
    I,J:byte;

  begin
    AllDone := false;

    for I := 1 to M do
      for J := 1 to N do
        if Cell[I,J].RefCount > 0 then
          exit;

    AllDone := true;
```

```
  end;

procedure PrintDone;                                    {打印计算后的电子表格}

  var
    I,J: byte;

  begin
    write('');
    for I := 1 to N do
      write(I - 1: 6);
    writeln;

    for I := 1 to M do
      begin
        write(char(I + ord('A') - 1));
        for J := 1 to N do
          write(Cell[I,J].Value: 6);
        writeln;
      end;
    writeln;
  end;

procedure PrintUnDone;                                  {打印无法计算的元素}

  var
    I,J: byte;
  begin
    for I := 1 to M do
      for J := 1 to N do
        if Cell[I,J].RefCount > 0 then
          writeln(char(I + ord('A') - 1),J - 1,': ',Cell[I,J].OriginExp);
    writeln;
  end;

begin{Main}
  assign(input,'ACM92FA.IN');
  assign(output,'ACM92FA.OUT');
  reset(input);
  rewrite(output);

  while InputData do                                    {当输入还未结束时}
```

```
    begin
      Calc;                                  {计算电子表格}
      if AllDone                             {如果电子表格已计算完毕}
        then
          PrintDone                          {则打印计算后的电子表格}
        else
          PrintUnDone;                       {否则打印无法计算的元素}
    end;

  close(input);
  close(output);
end .{Main}
```

7.2 布　　线

1. 解题思路

本题由于输入规模小(计算机台数最多为 8),可直接采用回溯法解决。算法核心如下:

```
PROC GO(K, CS, NowCost);                 {递归搜索连接方案}
                                         {其中 K 为递归深度}
                                         {CS 为现在还没有连接的}
                                         {计算机的集合}
                                         {NowCost 为当前电缆长度}
BEGIN
  IF NowCost > = MinCost                 {MinCost 为当前最优解的值}
    THEN RETURN
    ELSE IF K > N                        {全部计算机都连入网络中}
           THEN 【保存当前连接方案;       {找到了一个更优的解}
                 MinCost := NowCost】
           ELSE  FOR I := 1 TO 8 DO      {继续构造网络}
                 IF I 在 CS 中
                   THEN 【将 I 记录到当前连接方案中;
                          IF K > 1
                            THEN GO(K + 1, Cs - [I], NowCost + 计算机 I 与
                                 连接方案中前一台计算机之间的电缆长度)
                            ELSE GO(K + 1, Cs - [I], NowCost)】
                                         {连入第一台计算机}
ENDP;
```

2．程序清单

```
{$A+,B-,D-,E-,F-,G+,I-,L-,N-,O-,P-,Q-,R-,S-,T-,V+,X-,Y-}
{$M 65520,0,655360}
program Getting_in_Line;                          {布线}

  type
    TComputer =                                   {计算机类型}
      record
        X, Y: byte;                               {横、纵坐标}
      end;

    TComputerSet = set of 1..8;                   {计算机集合类型}

  var
    CaseNumber: byte;                             {数据项编号}
    N: byte;                                      {网络中计算机数目}
    MinCost: real;                                {最小电缆长度}
    BestLink,                                     {最优连接方案}
    NowLink: array[1..8]of byte;                  {当前连接方案}
    Computer: array[1..8]of TComputer;            {计算机}
    Distance: array[1..8, 1..8]of real;           {计算机之间的距离}

  function InputData: boolean;                    {读入数据项}
                                                  {并返回输入结束信息}

    var
      I, J: byte;

    begin
      readln(N);

      if N=0 then
        begin
          InputData:=false;
          exit;
        end;

      for I:=1 to N do
        readln(Computer[I].X,Computer[I].Y);
```

```
    for I := 1 to N do
      for J := 1 to N do
        Distance[I,J] := sqrt(sqr(Computer[I].X - Computer[J].X) +
                              sqr(Computer[I].Y - Computer[J].Y)) + 16.0;

    inc(CaseNumber);
    InputData := true;
  end;

procedure FindBest;                                    {寻找最优连接方案}

  procedure Go(K: byte; CS: TComputerSet; NowCost: real);
                                                       {递归搜索连接方案}
                                                       {K        递归深度}
                                                       {CS       待连接计算机集合}
                                                       {NowCost 当前电缆长度}

    var
      I: byte;

    begin
      if NowCost >= MinCost                            {如果当前代价}
                                                       {已大于或等于最小代价}
        then                                           {则剪枝}
          exit
        else                                           {否则则有可能找到更优的解}
          if K > N                                     {如果计算机已全部连入网络}
            then                                       {则找到了一个更优的解}
              begin
                BestLink := NowLink;
                MinCost := NowCost;
              end
            else                                       {否则继续构造网络}
              for I := 1 to 8 do                       {对于所有的计算机}
                if I in CS then                        {如果其尚未连入网络}
                  begin
                    NowLink[K] := I;

                    if K > 1                           {连接第一台计算机无距离}
                      then
                        Go(K + 1,CS - [I],NowCost + Distance[I,NowLink[K - 1]])
                      else
```

```
                    Go(K+1,CS-[I],NowCost);
                end;
        end;

    begin {FindBest}
        MinCost:=1e18;
        Go(1,[1..N],0);                          {递归搜索最优连接方案}
    end;{FindBest}

procedure Print;                                 {打印输出最优连接方案}

    var
        I: byte;

    begin
        for I:=1 to 58 do
            write('*');
        writeln;

        writeln('Network # ',CaseNumber);
        for I:=1 to N-1 do
            writeln('Cable reqiurement to connect(',Computer[BestLink[I]].X,',',
                    Computer[BestLink[I]].Y,')to(',Computer[BestLink[I+1]].X,',',
                    Computer[BestLink[I+1]].Y,')is',Distance[BestLink[I],
                    BestLink[I+1]]:1:2,'feet.');

        writeln('Number of feet of cable required is',MinCost:1:2);
    end;

begin{Main}
    assign(input,'ACM92FB.IN');
    assign(output,'ACM92FB.OUT');
    reset(input);
    rewrite(output);

    CaseNumber:=0;
    while InputData do                           {当输入还未结束时}
        begin
            FindBest;                            {寻找最优连接方案}
            Print;                               {打印输出最优连接方案}
        end;

    close(input);
    close(output);
end.{Main}
```

7.3 无线电定向

1. 解题思路

先来分析一下定向测量过程。假定船舶的航线相对于图 7.2Y 轴的偏移角为 θ_0，由此可以得到航线的倾角 ϕ_0。在 t_1 时刻信号发射站 $B_1(X_1, Y_1)$ 相对于船舶航线的偏移角为 θ_1，则船舶 t_1 时刻的位置 P_1 一定在过 (X_1, Y_1) 点、相对于 Y 轴的偏移角为 $\theta_0 + \theta_1$ 的直线 L_1 上，记直线 L_1 的倾角为 ϕ_1，ϕ_1 可以由 θ_0、θ_1 求出。同样，在 t_2 时刻信号发射站 $B_2(X_2, Y_2)$ 相对于船舶航线的偏移角为 θ_2，则船舶 t_2 时刻的位置 P_2 一定在过 (X_2, Y_2) 点、相对于 Y 轴的偏移角为 $\theta_0 + \theta_2$ 的直线 L_2 上，记直线 L_2 的倾角为 ϕ_2，ϕ_2 可以由 θ_0、θ_2 求出。如果船舶的航速为 v，则 $|P_2P_1| = v(t_2 - t_1)$。

图 7.2

现在的问题是求 P_2。设 P_2 的坐标为 (X, Y)，则 P_1 的坐标为 $(X - v(t_2 - t_1)\cos\phi_1, Y - v(t_2 - t_1)\sin\phi_1)$。将 P_1、P_2 分别代入直线 L_1、L_2 方程，就可以得到关于 X、Y 的二元一次方程组。解这个方程组，就得到了船舶的位置。方程组有无穷组解的情况意味着无法确定船舶的方向。

2. 程序清单

```
{$A+,B-,D-,E-,F-,G+,I-,L-,N-,O-,P-,Q-,R-,S-,T-,V+,X-,Y-}
{$M 65520,0,655360}
program Radio_Direction_Finder;                    {无线电定向}

const
  Eps = 1E-6;                                      {最小误差}
```

```
type
TPosition =                                          {位置类型}
  record
    X, Y: real;
  end;

TBeacon =                                            {信号发射站类型}
  record
    Name: string[20];                                {信号发射站的名字}
    Position: TPosition;                             {信号发射站的位置}
  end;

TBeaconReading =                                     {定向测量类型}
  record
    Time: longint;                                   {定向测量的时间}
    Name: string;                                    {信号发射站的名字}
    Angle: real;                                     {偏移角}
    Number: byte:                                    {信号发射站的编号}
  end;

var
  ScenarioCount,                                     {船舶数据数目}
  ScenarioNumber,                                    {船舶数据编号}
  BeaconCount: byte;                                 {信号发射站数目}
  BoatPosition: TPosition;                           {船舶位置}
  Beacon: array[1 .30]of TBeacon;                    {信号发射站}
  Course, Speed: real;                               {航线、速度}
  FirstReading, SecondReading: TBeaconReading;       {第一、二次定向测量信息}

procedure InputBeaconData;                           {读入信号发射站信息}

  var
    S: string;
    Code: integer;
    I: byte;

  begin
    readln(BeaconCount);

    for I := 1 to BeaconCount do
      with Beacon[I]do
        begin
```

```
            readln(S);

            Name := copy(S,1,pos(' ',S) - 1);
            delete(S,1,pos(' ',S));

            val(copy(S,1,pos(' ',S) - 1),Position.X,Code);
            delete(S,1,pos(' ',S));

            val(S,Position.Y,Code);
          end;
    end;

  procedure InputScenarioData;                    {读入船舶数据信息}

    procedure GetBeaconReading(var BeaconReading: TBeaconReading);
                                                  {读入定向测量信息}

      var
        S: string;
        Code: integer;
        I: byte;

      begin
        readln(S);

        with BeaconReading do
          begin
            val(copy(S,1,pos(' ',S) - 1),Time,Code);
            delete(S,1,pos(' ',S));

            Name := copy(S,1,pos(' ',S) - 1);
            delete(S,1,pos(' ',S));

            val(S,Angle,Code);
            Angle := Course + Angle;              {计算相对 Y 轴的偏移角}
            Angle := Angle * PI/180;              {角度转弧度}
            Angle := (PI/2 - Angle);              {转换成倾角}
            for I := 1 to BeaconCount do          {寻找信号发射站}
              if Beacon[I].Name = Name then
                begin
                  Number := I;
                  break;
```

```
                end;
            end;
        end;

    begin {InputScenarioData}
        readln(Course, Speed);

        GetBeaconReading(FirstReading);                    {读入第一次定向测量信息}
        GetBeaconReading(SecondReading);                   {读入第二次定向测量信息}

        Course := Course * PI/ 180;                        {角度转弧度}
        Course := PI/ 2 - Course;                          {转换成倾角}
    end; {InputScenarioData}

function FindBoatPosition: boolean;                        {返回是否能确定船舶位置}

    var
        X1, X2, Y1, Y2, C1, C2, D0, D1, D2: real;

    function Tan(X: real): real;                           {正切函数}

        begin
            Tan := Sin(X)/ Cos(X);
        end;

    begin {FindBoatPosition}
        if abs(cos(FirstReading.Angle)) < Eps               {如果第一次定向测量}
                                                           {垂直于 X 轴}
            then
                if abs(cos(SecondReading.Angle)) < Eps      {如果第二次定向测量}
                                                           {也垂直于 X 轴}
                    then                                   {则无法确定船舶位置}
                        FindBoatPosition := false
                    else                                   {否则可根据方程组的解法}
                                                           {计算出船舶位置}

                        begin
                            FindBoatPosition := true;
                            BoatPosition.X := Beacon[FirstReading.Number].Position.X +
                                    Speed * (SecondReading.Time - FirstReading.Time) * cos(Course);
                            BoatPosition.Y := Beacon[SecondReading.Number].Position.Y +
                                    Tan(SecondReading.Angle) *
```

```
                    (BoatPosition.X - Beacon[SecondReading.Number].Position.X) +

                  Speed * (SecondReading.Time - FirstReading.Time) * sin(Course);
        end
    else
      if abs(cos(SecondReading.Angle)) < Eps        {如果第二次定向测量}
                                                    {垂直于 X 轴}
        then                                        {则可根据方程组的解法}
                                                    {计算出船舶位置}
          begin
            FindBoatPosition := true;
            BoatPosition.X := Beacon[SecondReading.Number].Position.X;
            BoatPosition.Y := Beacon[FirstReading.Number].Position.Y +
                    Speed * (SecondReading.Time - FirstReading.Time) * sin(Course) +
                    Tan(FirstReading.Angle) *
                            (BoatPosition.X -
                            Speed * (SecondReading.Time - FirstReading.Time) *
                              cos(Course) -
                            Beacon[FirstReading.Number].Position.X);
          end
        else                                        {否则则需进一步讨论}
          begin
            X1 := Tan(SecondReading.Angle);
            X2 := Tan(FirstReading.Angle);
            D0 := X2 - X1;

            FindBoatPosition := abs(D0) < = Eps;
                                                    {能否确定船舶位置取决于两}
                                                    {次定向测量的方向是否平行}

            if abs(D0) < = Eps then                 {如果平行}
              exit;                                 {则无法确定船舶位置}

                                                    {以下根据方程组的解法}
                                                    {计算船舶位置}

            C1 := X1 * Beacon[SecondReading.Number].Position.X -
                      Beacon[SecondReading.Number].Position.Y;
            C2 := X2 * Beacon[FirstReading.Number].Position.X -
                      Beacon[FirstReading.Number].Position.Y +
                      Speed * (SecondReading.Time - FirstReading.Time) *
                        (X2 * cos(Course) - sin(Course));

            D1 := C2 - C1;
```

```
                D2 := X1 * C2 - X2 * C1;

                BoatPosition.X := D1/D0;
                BoatPosition.Y := D2/D0;
              end;
  end;{FindBoatPosition}

begin {Main}
  assign(input,'ACM92FC.IN');
  assign(output,'ACM92FC.OUT');
  reset(input);
  rewrite(output);

  InputBeaconData;                                  {读入信号发射站信息}
  readln(ScenarioCount);                            {读入船舶数据数目}

  for ScenarioNumber := 1 to ScenarioCount do       {对于所有的船舶数据}
    begin
      InputScenarioData;                            {读入船舶数据信息}

      if FindBoatPosition                           {如果可以确定船舶位置}
        then                                        {则输出船舶位置}
          writeln('Scenario',ScenarioNumber,':Position is(',BoatPosition.X:1:2,',',
                  BoatPosition.Y:1:2,')')
        else                                        {否则输出无解信息}
          writeln('Scenario',ScenarioNumber,':Position cannot be determined');
    end;

  close(input);
  close(output);
end.{Main}
```

7.4 扑灭飞蛾

1.解题思路

题中所需的试验区域在几何中称为由试验点所组成的点集的凸包。凸包有以下几个性质:

(1) 一定是一个凸多边形;

(2) 围住点集中所有的点;

(3) 周长最短。

求平面内 N 个点的凸包是一个基础的几何算法。求凸包的算法有很多,这里介绍一种最常用且较易于理解的算法——卷包裹法。

从一个肯定在凸包边界上的点 A_0 出发(例如 X 坐标最小的点)画一条垂直的直线,将该直线按逆时针方向旋转(以 A_0 为中心),直到遇到点集中一点 A_1,则 A_1 必在凸包边界上;以 A_1 为中心继续逆时针旋转该直线,又可遇到点集中的另一点 A_2,则 A_2 也必在凸包边界上;……这样继续下去,直至该直线转过了 360°后再次遇到出发点 A_0。上述过程类似于卷包裹,直至完全把点集中所有的点卷在内部为止,如图 7.3 所示。

图 7.3

2.程序清单

```
{$A+,B-,D-,E-,F-,G+,I-,L-,N-,O-,P-,Q-,R-,S-,T-,V-,X-,Y-}
{$M 65520,0,655360}
program Moth_Eradication;                          {扑灭飞蛾}

  type
    TPoint =                                       {试验点类型}
      record
        X,Y: real;                                 {横、纵坐标}
      end;

  var
    RegionNumber,                                  {区域编号}
    PointCount,                                    {区域中试验点的个数}
    PerimeterPointCount: byte;                     {区域边界上的试验点个数}
    PerimeterLength: real;                         {区域周长}
    Point: array[1..255]of TPoint;                 {试验点}
    PerimeterPoint: array[1..255]of byte;          {区域边界上的试验点}

  function InputData: boolean;                     {读入区域的数据记录}
                                                   {并返回输入结束信息}

    var
```

```
    I: byte;

  begin
    readln(PointCount);

    if PointCount = 0 then
      begin
        InputData := false;
        exit;
      end;

    for I := 1 to PointCount do
      readln(Point[I].X, Point[I].Y);

    inc(RegionNumber);
    InputData := true;
  end;

procedure Work;                                  {找出区域边界上的试验点}
                                                 {并计算区域周长}

  var
    SelectPoint, I, NowPoint: byte;
    SelectAngle, NewAngle, LastAngle: real;

  procedure Adjust(var Angle: real);             {换算 Angle 到 0～2π 之间}

    begin
      while Angle < 0 do
        Angle := Angle + PI + PI;
      while Angle >= PI + PI do
        Angle := Angle - PI - PI;
    end;

  function CalcAngle(A, B: byte): real;          {计算射线 BA 相对于}
                                                 {前一条边的倾角}

    var
      X, Y, Angle: real;

    begin
      if A = B then
```

```
        begin
          CalcAngle := -1;
            exit;
          end;

      X := Point[A].X - Point[B].X;
      Y := Point[A].Y - Point[B].Y;
      if X = 0
        then
          if Y > 0
            then
              Angle := PI/2
            else
              Angle := PI + PI/2
        else
          if X > 0
            then
              Angle := arctan(Y/X)
            else
              Angle := arctan(Y/X) + PI;

      Angle := Angle - LastAngle;                    {计算倾角}
      Adjust(Angle);

      CalcAngle := Angle;
    end;

  begin{Work}
    PerimeterLength := 0;

    SelectPoint := 1;                                {找到最左边的点作为出发点}
    for I := 2 to PointCount do
      if Point[SelectPoint].X > Point[I].X then
        SelectPoint := I;
    NowPoint := SelectPoint;

    PerimeterPoint[I] := NowPoint;
    PerimeterPointCount := 1;
    LastAngle := PI/2;                               {从射线 OY 开始顺时针扫描}

    repeat
      SelectAngle := CalcAngle(1, NowPoint);
```

```
        SelectPoint := 1;
        for I := 2 to PointCount do                    {找出下一个边界实验点}
          begin
            NewAngle := CalcAngle(I, NowPoint);
            if NewAngle > SelectAngle then
              begin
                SelectPoint := I;
                SelectAngle := NewAngle;
              end;
          end;

        inc(PerimeterPointCount);
        PerimeterPoint[PerimeterPointCount] := SelectPoint;

        PerimeterLength := PerimeterLength + sqrt(sqr(Point[NowPoint].X -
                    Point[SelectPoint].X) + sqr(Point[NowPoint].Y - Point[SelectPoint].Y));
                                                       {累加边界长度}

        LastAngle := LastAngle + SelectAngle;          {记录边界线倾角}
        Adjust(LastAngle);

        NowPoint := SelectPoint;
      until NowPoint = PerimeterPoint[1];              {直到回到出发点}
    end;{Work}

procedure Print;                                       {输出区域边界上的试验点}
                                                       {及区域周长}

    var
      I: byte;

    begin
      writeln('Region # ', RegionNumber, ':');

      for I := 1 to PerimeterPointCount - 1 do
        write('(',Point[PerimeterPoint[I]].X:1:1,',',Point[PerimeterPoint[I]].Y:1:1,') - ');
      writeln('(', Point[PerimeterPoint[1]].X:1:1, ',', Point[PerimeterPoint[1]].Y:1:1, ')');

      writeln('Perimeter length = ', PerimeterLength:1:2);
      writeln;
    end;

begin {Main}
```

```
  assign(input, 'ACM92FD .IN');
  assign(output, 'ACM92FD .OUT');
  reset(input);
  rewrite(output);

  RegionNumber := 0;
  while InputData do                          {当输入还未结束时}
    begin
      Work;                                   {找出区域边界上的试验点}
                                              {并计算区域周长}
      Print;                                  {输出区域边界上的试验点}
                                              {及区域周长}
    end;

  close(input);
  close(output);
end .{Main}
```

7.5 寻找冗余

1. 解题思路

一个依赖冗余,就是说它可以由其它依赖推导出来。那么如何判断一个依赖能否被其它依赖推导出来呢?设待判断的依赖为"α→β",先寻找形式为"α→*"的依赖,由它开始搜索,直到出现"#→β"为止(这里*、#表示任意一个域名)。搜索可以用简单的回溯法完成。由于题中要求所求得的证明序列中所有的依赖都应当是必需的,因此搜索出的证明序列不一定都是符合要求的。在参考程序中,为使程序简洁起见,我们总是求出长度最短的证明序列,因为最短证明序列中一定不存在多余的依赖(这一点留给读者自己证明)。

2. 程序清单

```
{$A+,B-,D-,E-,F-,G+,I-,L-,N-,O-,P-,Q-,R-,S-,T-,V+,X-,Y-}
{$M 65520,0,655360}
program Department_of_Redundancy_Department;      {寻找冗余}

  type
    TFieldSet = set of 'A'..'Z';                  {域集合类型}
    TFD =                                         {函数依赖类型}
      record
        Set1,Set2:TFieldSet;
      end;

    var
```

```
    GroupNumber,                                    {依赖组的编号}
  N: byte;                                          {依赖个数}
  RedundantFDFound: boolean;                        {存在冗余依赖标志}
  FD: array[1..100]of TFD;                          {函数依赖}

function InputData: boolean;                        {读入一组函数依赖}
                                                    {并返回输入结束信息}

  var
    I: byte;
    S: string;

  procedure GetFieldSet(T: string; var S: TFieldSet);
                                                    {从串 T 中分离出集合 S}

    begin
      S:=[];
      while T<>''do
        begin
          Include(S, T[length(T)]);
          dec(T[0]);
        end;
    end;

  begin {InputData}
    readln(N);

    if N=0 then
      begin
        InputData:=false;
        exit;
      end;

    for I:=1 to N do
      begin
        readln(S);

        GetFieldSet(copy(S,1,pos('->',S)-1),FD[I].Set1);
        delete(S,1,Pos('->',S)+1);
        GetFieldSet(S,FD[I].Set2);
      end;
```

```
    inc(GroupNumber);
    InputData := true;
  end;{InputData}

procedure FindRedundantFD;                          {寻找冗余依赖}

  var
    K: byte;
    S: string;

  function SearchRedundant(P: byte): string;
                                                    {通过搜索判断函数依赖 P 是}
                                                    {否是冗余依赖,如果是则返}
                                                    {回用来证明 P 是冗余依赖的}
                                                    {依赖序列,否则返回空串}

    var

      TrueFieldSet: array[1..100]of TFieldSet;      {回溯所用的栈:}
                                                    {栈中的元素为由当前}
                                                    {可确定的域所组成的集合}
      FDPath: array[1..100]of byte;                 {用来证明的依赖序列}
      Depth,I,MinDepth: byte;
      S,T: string;

    begin
                                                    {下面用回溯法搜索用来证明}
                                                    {P 是冗余依赖的依赖序列}

      S := '';
      TrueFieldSet[1] := FD[P].Set1;
      FDPath[1] := 0;
      Depth := 1;
      MinDepth := 255;

      while Depth > 0 do
        begin
          if FD[P].Set2 < = TrueFieldSet[Depth]
            then
              if Depth < MinDepth
                then
                  begin
```

```
                    S := '';
                    for I := 1 to Depth - 1 do
                      begin
                        str(FDPath[I], T);
                        S := S + ' ' + T;
                      end;
                    MinDepth := Depth;
                  end
                else
                  dec(Depth)
              else
                begin
                  repeat
                    inc(FDPath[Depth]);
                  until (FDPath[Depth] > N)or
                        ((FDPath[Depth] < > P)and
                          (FD[FDPath[Depth]].Set1 < = TrueFieldSet[Depth])and
                          not(FD[FDPath[Depth]].Set2 < = TrueFieldSet[Depth]));

                  if FDPath[Depth] > N
                    then
                      dec(Depth)
                    else
                      begin
                        FDPath[Depth + 1] := 0;
                        TrueFieldSet[Depth + 1] := TrueFieldSet[Depth] +
                                                   FD[FDPath[Depth]].Set2;
                        inc(Depth);
                      end;
                end;
            end;

          SearchRedundant := S;
        end;

      begin{FindRedundantFD}
        RedundantFDFound := false;
        writeln('Set number', GroupNumber);

        for K := 1 to N do                          {对于每个依赖}
          begin
            S := SearchRedundant(K);                {搜索判断其是否冗余}

            if S < > ''then
              begin
                writeln('     FD', K, 'is redundant using FDs: ', S);
```

```
                RedundantFDFound := true;
              end;
          end;

        if not RedundantFDFound then
          writeln('      No redundant FDs .');

        writeln;
      end;{FindRedundantFD}

    begin {Main}
      assign(input ,'ACM92FE .IN');
      assign(output ,'ACM92FE .OUT');
      reset(input);
      rewrite(output);

      GroupNumber := 0;
      while InputData do                          {当输入还未结束时}
        FindRedundantFD;                          {寻找冗余依赖}

      close(input);
      close(output);
    end .{Main}
```

7.6 奥 赛 罗

1. 解题思路

本题的关键在于“列出所有可能的走步”,即对于棋盘上给定的位置(Row, Column),判断在该位置放入本方棋子后,是否能够夹住对方棋子。算法描述如下:

```
FUNC CanBracket(Row,Column):Boolean;          {判断可否在位置}
                                              {(Row,Column)上放入棋子}
BEGIN
  FOR Dir := 1 TO 8 DO                        {依次试探八个方向}
    【 以位置(Row,Column)为出发点;
       REPEAT
         沿当前方向走一格;
       UNTIL 走出棋盘 OR 当前位置空 OR 当前位置上为自己的棋子;
       IF 当前位置上为自己的棋子 AND 至少走了两格
```

```
                THEN RETURN(True);
    RETURN(False);
  ENDF;
```

2. 程序清单

```
{$A+,B-,D-,E-,F-,G+,I-,L-,N-,O-,P-,Q-,R-,S-,T-,V+,X-,Y-}
{$M 65520,0,655360}
program Othello;                                    {奥赛罗}

  const
    stWhitePlay = 1;                                {白方走棋状态}
    stBlackPlay = 2;                                {黑方走棋状态}
    stGameOver = 3;                                 {游戏结束状态}

    cmList = 1;                                     {"列出所有可能走步"命令}
    cmMove = 2;                                     {"走一步"命令}
    cmQuit = 3;                                     {"退出当前游戏"命令}

    Piece: array[1..2] of char = 'WB';              {棋子的代表字符}

    DX: array[1..8] of shortint = (0,0,1,1,1,-1,-1,-1);
    DY: array[1..8] of shortint = (1,-1,0,1,-1,0,1,-1);

  var
    GameCount,                                      {游戏数目}
    GameNumber,                                     {游戏编号}
    GameState,                                      {游戏状态}
    Command,                                        {命令}
    Row,Column: byte;                               {行、列号}
    Board,                                          {棋盘}
    Possible: array[1..8,1..8] of char;             {放入棋子的可能性数组}

  procedure InputBoardConfiguration;                {读入棋盘局面}

    var
      I,J: byte;
      Player: char;

    begin
      for I := 1 to 8 do
```

```
        begin
          for J := 1 to 8 do
            read(Board[I,J]);
          readln;
        end;

      readln(Player);

      if Player = 'W'
        then
          GameState := stWhitePlay
        else
          GameState := stBlackPlay
    end;

  procedure InputCommand;                          {读入命令}

    var
      Ch: char;

    begin
      read(Ch);

      case Ch of
        'L':Command := cmList;
        'M':
          begin
            Command := cmMove;
            read(Ch);
            Row := ord(Ch) - ord('0');
            read(Ch);
            Column := ord(Ch) - ord('0');
          end;
        'Q':Command := cmQuit;
      end;

      readln;
    end;

  procedure FindPossible;                          {寻找可能的走步}

    var
```

```
    Row,Column: byte;

  function CanBracket(Row,Column: byte):boolean;
                                                    {返回在指定位置放入棋子后}
                                                    {能否夹住对方棋子}
                                                    {Row      待放入位置的行号}
                                                    {Column   待放入位置的列号}

    var
      Dir,X,Y: byte;

    begin
      for Dir := 1 to 8 do
        begin
          X := Row;
          Y := Column;
          repeat
            inc(X,DX[Dir]);
            inc(Y,DY[Dir]);
          until not((X in[1..8])and(Y in[1..8]))or
                    (Board[X,Y] <> Piece[3 - GameState]);

          if(X in[1..8])and(Y in[1..8])and(Board[X,Y] = Piece[GameState])
            and((Row + DX[Dir] <> X)or(Column + DY[Dir] <> Y))then
            begin
              CanBracket := true;
              exit;
            end;
        end;

      CanBracket := false;
    end;

  begin {FindPossible}
    fillchar(Possible,sizeof(Possible),'-');

    for Row := 1 to 8 do
      for Column := 1 to 8 do
        if(Board[Row,Column] = '-')and CanBracket(Row,Column)then
            Possible[Row,Column] := 'P';
  end;{FindPossible}

procedure DoList;                                   {执行}
```

```
                                                    {"列出所有可能走步"命令}
  var
    I,J: byte;
    Flag: boolean;

  begin
    FindPossible;

    Flag := true;
    for I := 1 to 8 do
      for J := 1 to 8 do
        if Possible[I,J] = 'P' then
          begin
            write('(',I,',',J,') ');
            Flag := false;
          end;

    if Flag
      then
        writeln('No legal move .')
      else
        writeln;
  end;

procedure DoMove;                                   {执行"走一步"命令}

  var
    Dir,X,Y,I,J,
    WhitePieceCount,BlackPieceCount: byte;          {棋盘上白、黑棋子总数}

  begin
    FindPossible;

    if Possible[Row,Column] < > 'P' then            {如果当前游戏者无合法走步}
      GameState := 3 - GameState;                   {则由另一名游戏者接着走棋}

    Board[Row,Column] := Piece[GameState];

    for Dir := 1 to 8 do                            {处理所有被夹住的棋子}
      begin
        X := Row;
        Y := Column;
```

```
            repeat
              inc(X,DX[Dir]);
              inc(Y,DY[Dir]);
            until not ((X in [1..8]) and (Y in [1..8])) or (Board[X, Y] <> Piece[3 -
GameState]);

            if (X in[1..8])and(Y in[1..8])and(Board[X,Y] = Piece[GameState])then
              repeat
                Board[X,Y]:= Piece[GameState];
                dec(X,DX[Dir]);
                dec(Y,DY[Dir]);
              until Board[X,Y] = Piece[GameState];
          end;

        WhitePieceCount := 0;
        BlackPieceCount := 0;
        for I := 1 to 8 do                                    {统计黑白棋子数}
          for J := 1 to 8 do
            if Board[I,J] = 'W'
              then
                inc(WhitePieceCount)
              else
                if Board[I,J] = 'B'then
                  inc(BlackPieceCount);

        writeln('Black - ',BlackPieceCount:2,'White - ',WhitePieceCount:2);
        GameState := 3 - GameState;                           {更替当前游戏者}
      end;

    procedure DoQuit;                                         {执行"退出当前游戏"命令}

      var
        I,J: byte;

      begin
        for I := 1 to 8 do                                    {打印棋盘}
          begin
            for J := 1 to 8 do
              write(Board[I,J]);
            writeln;
          end;

        GameState := stGameOver;
```

```
    end;

begin {Main}
  assign(input, 'ACM92FF .IN');
  assign(output, 'ACM92FF .OUT');
  reset(input);
  rewrite(output);
  readln(GameCount);

  for GameNumber := 1 to GameCount do          {对于所有的游戏}
    begin
      InputBoardConfiguration;                 {读入棋盘局面}

      repeat
        InputCommand;                          {读入命令}
        case Command of                        {执行相应指令}
          cmList:DoList;
          cmMove:DoMove;
          cmQuit:DoQuit;
        end;
      until GameState = stGameOver;            {直到当前游戏结束}

      writeln;
    end;

  close(input);
  close(output);
end .{Main}
```

7.7 城 市 正 视 图

1. 解题思路

建筑物的可见性等价于其南墙的可见性。建筑物之间的互相阻挡实质上是其南墙正投影之间的相互重叠。

对于一座建筑物的南墙矩形,如果没有任何建筑物挡住它,则是可见的。如果有建筑物完全将其挡住,则是不可见的。否则的话,任取一个挡住该矩形一部分的建筑物的南墙,对于没有被挡住的部分,将其划分成几个南墙上的子矩形(如图 7.4),递归判断其是否可见,只要其中有一个可见,南墙矩形就可见。

图 7.4

2. 程序清单

```
{$A+,B-,D-,E-,F-,G+,I-,L-,N-,O-,P-,Q-,R-,S-,T-,V+,X-,Y-}
{$M 65520,0,655360}
program Urban_Elevations;                          {城市正视图}

  var
    XCord,YCord,                                   {建筑物西南角横、纵坐标}
    Width,Depth,Height: array[0..100]of real;      {建筑物长度、宽度、高度}
    VisibleBuilding: array[0..100]of byte;         {可见建筑物}
    MapNumber,                                     {地图编号}
    N,                                             {建筑物总数}
    VisibleBuildingCount: byte;                    {可见建筑物数目}

  function InputData: boolean;                     {读入城市地图}
                                                   {并返回输入结束信息}

    var
      I: byte;

    begin
      readln(N);

      if N=0 then
        begin
          InputData:=false;
          exit;
        end;

      for I:=1 to N do
        readln(XCord[I],YCord[I],Width[I],Depth[I],Height[I]);

      inc(MapNumber);
      InputData:=true;
```

```
  end;

procedure FindVisible;                                    {找出可见的建筑物}

  function Visible(X1,X2,Height1,Height2:real;K: byte):boolean;
                                                          {返回建筑物 K 南墙上}
                                                          {一矩形是否可见}
                                                          {X1,X2}
                                                          {矩形左右两边的横坐标}
                                                          {Height1, Height2}
                                                          {矩形下、上两边的高度}

    var
      I: byte;

    begin
      Visible := false;

      if(X1 >= X2)or(Height1 >= Height2)then              {若矩形不存在}
        exit;                                             {则矩形显然不可见}

      for I := 1 to N do                                  {对于所有其余建筑物}
        if (I <> K)and not(YCord[I] >= YCord[K] + Depth[K]) and
              (X2 <= XCord[I] + Width[I]) and(X1 >= XCord[I])and
              (Height2 <= Height[I])then                  {如果其南墙包含了矩形}
              exit;                                       {则矩形不可见}

      for I := 1 to N do                                  {对于所有其余建筑物}
        if(I <> K)and not((YCord[I] >= YCord[K] + Depth[K])or
          (X1 >= XCord[I] + Width[I])or(X2 <= XCord[I])or
          (Height1 >= Height[I]))then                     {如果其可以部分阻挡矩形}
          begin
            Visible := true;

            if Visible(X1,XCord[I],Height1,Height2,K)then
                                                          {若矩形被截后左部区域可见}
              exit;                                       {则矩形可见}

            if Visible(XCord[I] + Width[I],X2,Height1,Height2,K)then
                                                          {若矩形被截后右部区域可见}
```

```
                exit;                                    {则矩形可见}

              ifX1 < XCord[I]then                        {计算矩形被截后}
                X1 := XCord[I];                          {上部区域的左边界}
              ifX2 > XCord[I] + Width[I]then             {计算矩形被截后}
                X2 := XCord[I] + Width[I]                {上部区域的右边界}

              if Visible(X1, X2, Height[I], Height2, K)then
                                                         {若矩形被截后上部区域可见}
                exit;                                    {则矩形可见}

              Visible := false;                          {否则矩形不可见}
              exit;
            end;

      Visible := true;
    end;

  var
    I: byte;

  begin {FindVisible}
    VisibleBuildingCount := 0;

    for I := 1 to N do                                   {找出所有南墙可见的建筑物}
      if Visible(XCord[I], XCord[I] + Width[I], 0, Height[I], I)then
        begin
          inc(VisibleBuildingCount);
          VisibleBuilding[VisibleBuildingCount]:= I;
        end;
  end;{FindVisible}

procedure SortVisible;                                   {将可见建筑物从南至北、}
                                                         {自西向东排序}

  procedure Exchange(I, J: byte);                        {交换建筑物 I 和 J}

    begin
      VisibleBuilding[0]:= VisibleBuilding[I];
      VisibleBuilding[I]:= VisibleBuilding[J];
```

```
        VisibleBuilding[J]:= VisibleBuilding[0];
      end;

    var
      I,J: byte;

    begin {SortVisible}
      for I := 1 to VisibleBuildingCount - 1 do
        for J := I + 1 to VisibleBuildingCount do
          if XCord[VisibleBuilding[I]] > XCord[VisibleBuilding[J]]
            then
              ExChange(I,J)
            else
              if(XCord[VisibleBuilding[I]] = XCord[VisibleBuilding[J]])
                and(YCord[VisibleBuilding[I]] > YCord[VisibleBuilding[I]]) then
                ExChange(I,J);
    end;{SortVisible}

  procedure Print;                                    {输出可见建筑物信息}

    var
      I: byte;

    begin
      writeln('For map #',MapNumber,', the visible buildings are numbered as follows:');

      for I := 1 to VisibleBuildingCount do
        write(VisibleBuilding[I],' ');
      writeln;

      writeln;
    end;

  begin {Main}
    assign(input,'ACM92FG.IN');
    assign(output,'ACM92FG.OUT');
    reset(input);
    rewrite(output);

    MapNumber := 0;
```

```
  while InputData do                    {当输入还未结束时}
    begin
      FindVisible;                      {找出可见的建筑物}
      SortVisible;                      {将可见建筑物排序}
      Print;                            {输出可见建筑物信息}
    end;

  close(input);
  close(output);
end .{Main}
```

第 8 章　解　答　III

8.1　旅 行 预 算

1 . 解题思路

这是一道采用回溯算法求解的问题。从第一个加油站开始,依次选择所要停下的下一个加油站。从而找出总费用最小的旅行路线。由于加油站的数目不是很大,保证了回溯不会进行得很深,为了方便编程,采用递归来实现。

本题的难点在于,汽车驾驶员的行为规则较为复杂。因此在选择下一个要停下的加油站时比较麻烦。参考程序里所采用的方法是:首先找出第一个可停靠的加油站,判断其后面的加油站是否可以到达,如果不可到达就必须在这里停下来加油;否则就找出可以到达但如果只用一半汽油则无法到达的所有加油站,依次进行停靠。

2 . 程序清单

```
{$A+,B-,D-,E-,F-,G+,I-,L-,N-,O-,P-,Q-,R-,S-,T-,V+,X-,Y-}
{$M 65520,0,655360}
program Budget_Travel;                          {计划旅行}
                                                {程序中所有费用均精确到分}

  type
    TStation =                                  {加油站类型}
      record
        Distance,                               {从出发地到加油站的距离}
        Price: real;                            {每加仑汽油的价格}
      end;

    var
      DataSetNumber: byte;                      {数据项编号}
      Distance,                                 {出发地到目的地的距离}
      Capacity,                                 {油箱容量}
      MilesPerGallon: real;                     {每加仑汽油可行驶的公里数}
      OriginalCost,                             {出发时费用}
      MinimumCost: longint;                     {最小费用}
      StationCount: byte;                       {加油站数目}
      Station: array[0..50] of TStation;        {加油站}

    function InputData: boolean;                {读入数据项}
```

```
                                                    {并返回输入结束信息}
  var
    I: byte;
    C: real;

  begin
    readln(Distance);

    if Distance < 0 then
      begin
        InputData := false;
        exit;
      end;

    readln(Capacity,MilesPerGallon,C,StationCount);
    OriginalCost := round(C * 100);

    fillchar(Station,sizeof(Station),0);
    for I := 1 to StationCount do
      readln(Station[I].Distance,Station[I].Price);

    inc(DataSetNumber);
    InputData := true;
  end;

procedure SearchMinCost;                            {寻找最小费用的旅行计划}

  procedure Drive(Stop: byte;Cost:longint);         {递归搜索最小费用旅行计划}
                                                    {Stop     前次停靠的加油站}
                                                    {Cost     当前费用}

    var
      StationNumber: byte;

    begin
      if Station[Stop].Distance + Capacity * MilesPerGallon > = Distance then
                                                    {如果从前次停靠的加油站}
                                                    {可以直接驶达目的地}
        begin                                       {则找到了一种新的旅行计划}
          if Cost < MinimumCost then                {如果其费用小于最小费用}
            MinimumCost := Cost;                    {则找到了更优的旅行计划}
          exit;
```

```
            end;

          StationNumber := Stop + 1;
          while (StationNumber < StationCount)and(Station[StationNumber + 1].Distance < =
                  Station[Stop].Distance + Capacity * MilesPerGallon)and
                 (Station[StationNumber].Distance < = Station[Stop].Distance +
                  Capacity * MilesPerGallon/ 2)do        {找出第一个可停靠的加油站}
            inc(StationNumber);

          if(StationNumber = StationCount)or(Station[StationNumber + 1].Distance >
            Station[Stop].Distance + Capacity * MilesPerGallon)then
                                                         {如果无法驶达下一个加油站}
                                                         {则停下加油}
            Drive(StationNumber,Cost + 200 + round((Station[StationNumber].Distance -
                    Station[Stop].Distance)/ MilesPerGallon * Station[StationNumber].Price));

          while(StationNumber < = StationCount)and(Station[StationNumber].Distance < =
                  Station[Stop].Distance + Capacity * MilesPerGallon/ 2)do
                                                         {找出用一半汽油无法驶达的}
                                                         {加油站}
            inc(StationNumber);

          while(StationNumber < = StationCount)and(Station[StationNumber].Distance < =
                  Station[Stop].Distance + Capacity * MilesPerGallon)do
                                                         {逐一停靠各个加油站}
            begin
              Drive (StationNumber,Cost + 200 + round((Station[StationNumber].Distance -
                      Station[Stop].Distance)/ MilesPerGallon * Station[StationNumber].Price));
              inc(StationNumber);
            end;
        end;

      begin {SearchMinCost}
        MinimumCost := maxlongint;
        Drive(0,OriginalCost);                           {递归搜索最小费用旅行计划}
      end;{SearchMinCost}

    procedure Print;                                     {打印输出最小费用}

      begin
        writeln('Data Set # ',DataSetNumber);
        writeln('  minimum cost = $ ',MinimumCost/ 100:1:2);
```

```
    end;

begin {Main}
  assign(input, 'ACM93FA.IN');
  assign(output, 'ACM93FA.OUT');
  reset(input);
  rewrite(output);

  DataSetNumber := 0;
  while InputData do                          {当输入还未结束时}
    begin
      SearchMinCost;                          {寻找最小费用的旅行计划}
      Print;                                  {打印输出最小费用}
    end;

  close(input);
  close(output);
end.{Main}
```

8.2 分划中土地的划分

1. 解题思路

由于问题要求确定土地的边界线段的数目,而这与线段的长短无关,因此很容易想到把分划抽象为一个图,分划中的线段就对应于图中的边,线段的端点对应于图中的点。但这个图不同于一般的图论意义上的图,必须考虑到点与边的几何属性(例如点与线的位置关系等)。因此,处理起来比较麻烦。

好在题目中给出了一个很重要的假定:每块土地至少有一条边界线段在矩形分划的边界上。这就意味着矩形分划内部不会有封闭的区域。由此我们可以得到,对于矩形分划边界上的线段,分别从其两端出发寻找封闭线段链,可以找到所有的土地。

这里有一个问题:从边界线段出发,如果走到了属于其它土地的边界线段,则会将多于一块的土地当成是一块。因此寻找时必须限制每一块土地的边界上包括且只能包括一段连续的矩形分划的边界线段链。

这里还有一个问题:一块土地可能包括不止一条边界线段(例如拐角上的土地),如果对于每一条边界线段,都从其出发找出相应的土地,就有可能引起重复。考虑到对于度为2的点来说,其在寻找土地边界的过程中只是起到一个标记土地边界线段数目的作用,我们可以对前述图中的边引入长度概念,令直接对应于分划中线段边的长度为1,将所有中间结点度均为2的路径合并为一条边,其长度为原来各条边长度之和,图8.1是题目中样例输入中的图经过合并、抽象后得到的图。这样处理之后,我们还得到了一个额外的好

处,就是可以进一步限制每一块土地的边界上恰有一条线段在矩形分划的边界上。亦即土地同由矩形分划边界上的线段组成的边一一对应。从每一条这样的边的一端出发,通过矩形分划内部的线段所组成的边,找到回到出发边另一端的回路,就唯一确定了一块土地,回路的长度就是这块土地边界线段的数目。

图 8.1

2. 程序清单

```
{$A+,B-,D-,E-,F-,G+,I-,L-,N-,O-,P-,Q-,R-,S-,T-,V+,X-,Y-}
{$M 65520,0,655360}
program Classifying_Lots_in_a_Subdivision;                {分划中土地的分类}

  type
    TPoint =                                              {点类型}
      record
        X,Y: word;
      end;

    TLine =                                               {线段类型}
      record
        A,B,Length: byte;                                 {端点、中间线段条数}
        Bounding: boolean;                                {边界标志}
      end;

    TPointSet = set of 1..200;                            {点集类型}

  var
    CaseNumber,                                           {数据项编号}
    LineCount,PointCount: byte;                           {线段、点数目}
    Line: array[1..200]of TLine;                          {线段}
    PlotCount: array[1..100]of byte;                      {各种土地的数目}

  function InputData: boolean;                            {读入、简化测试项}
                                                          {并返回输入结束信息}

    var
      LineNumber: byte;
      MaxX,MinX,MaxY,MinY,AX,AY,BX,BY: word;
```

```
  Point: array[1..200]of TPoint;

function Location(X,Y: word):byte;                    {返回点(X,Y)的编号}

  var
    PointNumber: byte;

  begin
    for PointNumber:=1 to PointCount do
      if(Point[PointNumber].X = X)and(Point[PointNumber].Y = Y)then
        begin
          Location:=PointNumber;
          exit;
        end;

    inc(PointCount);
    Point[PointCount].X:=X;
    Point[PointCount].Y:=Y;
    Location:=PointCount;
  end;

procedure UpDate(New: word;var Min,Max: word);
                                                      {根据 New 更新 Min 和 Max}

  begin
    if New<Min
      then
        Min:=New
      else
        if New>Max then
          Max:=New;
  end;

begin {Location}
  readln(LineCount);

  if LineCount=0 then
    begin
      InputData:=false;
      exit;
    end;
```

```
    MinX := 10001;
    MinY := 10001;
    MaxX := 0
    MaxY := 0;
    fillchar(Line,sizeof(Line),0);

    for LineNumber := 1 to LineCount do                    {读入并处理所有线段}
      with Line[LineNumber]do
        begin
          readln(AX,AY,BX,BY);

          A := Location(AX,AY);
          B := Location(BX,BY);
          Length := 1;

          UpDate(AX,MinX,MaxX);
          UpDate(AY,MinY,MaxY);
          UpDate(BX,MinX,MaxX);
          UpDate(BY,MinY,MaxY);
        end;

    for LineNumber := 1 to LineCount do                    {计算线段的边界标志}
      with Line[LineNumber]do
        Bounding := ((Point[A].X = Point[B].X)and   ((Point[A].X = MinX)or
                    (Point[A].X = MaxX)))or((Point[A].Y = Point[B].Y)and
                    ((Point[A].Y = MinY)or(Point[A].Y = MaxY)));

    inc(CaseNumber);
    InputData := true;
  end{Location}

procedure PreProcess;                                      {预处理}

  var
    Degree: array[1..200]of byte;
    PointNumber,Line1,Line2,LineNumber: byte;

  begin
    if LineCount = 4 then                                  {只有 4 个点时无须简化}
      exit;

    fillchar(Degree,sizeof(Degree),0);
    for LineNumber := 1 to LineCount do
      begin
        inc(Degree[Line[LineNumber].A]);
```

```
        inc(Degree[Line[LineNumber].B]);
      end;

    for PointNumber := 1 to PointCount do          {合并所有中间线段}
      if Degree[PointNumber] = 2 then
        begin
          Line1 := 0;
          Line2 := 0;
          for LineNumber := 1 to LineCount do
            if (Line[LineNumber].A = PointNumber)or
                (Line[LineNumber].B = PointNumber)then
                ifLine1 = 0
              then
                Line1 := LineNumber
              else
                Line2 := LineNumber;
          if Line[Line1].A = PointNumber then
              Line[Line1].A := Line[Line1].B;
          if Line[Line2].B = PointNumber
            then
              Line[Line1].B := Line[Line2].A
            else
              Line[Line1].B := Line[Line2].B;
          inc(Line[Line1].Length,Line[Line2].Length);

          for LineNumber := Line2 to LineCount - 1 do
            Line[LineNumber]:= Line[LineNumber + 1];
          dec(LineCount);
        end;
  end;

procedure FindAll;                                 {找出所有土地}

  var
    LineNumber: byte;
    PlotFound: boolean;

  procedure SearchPath(Now,Dest:byte;PSet:TPointSet;NowLength: byte);
                                                   {递归搜索土地边界}
                                                   {Now        当前点}
                                                   {Dest       目标点}
                                                   {PSet       已搜索点集合}
```

```
                                                    {NowLength  当前线段条数}

  var
    LineNumber,New: byte;

  begin
    if Now = Dest then                              {如果返回到了目标点}
      begin                                         {则找到了土地所有边界}
        inc(PlotCount[NowLength]);
        PlotFound := true;
        exit
      end;

    for LineNumber := 1 to LineCount do             {尝试所有的线段}
      with Line[LineNumber]do
        if not Bounding and(Now in [A,B])then
                                                    {线段必须在划分区域内}
                                                    {且同当前点相连}

          begin
            if A = Now
              then
                New := B
              else
                New := A;

            if not(New in PSet)then
              begin
                SearchPath(New,Dest,PSet + [New],NowLength + Length);

                if PlotFound then
                  exit;
              end;
          end;
  end;

begin {SearchPath}
  fillchar(PlotCount,sizeof(PlotCount),0);

  if LineCount = 4 then                             {只有 4 个点时一定是四边形}
    begin
      PlotCount[4] := 1;
```

```
            exit;
          end;

        for LineNumber := 1 to LineCount do          {对于所有的线段}
          with Line[LineNumber]do
            if Bounding then                         {如果其是边界线段}
              begin                                  {则其对应着一片土地}
                PlotFound := false;
                SearchPath(A,B,[A],Length);
              end;

    end;{SearchPath}

procedure Print;                                     {打印输出}

    var
      PlotNumber,TotalPlotCount: byte;

    begin
      writeln('Case',CaseNumber);

      TotalPlotCount := 0;
      for PlotNumber := 1 to 100 do
        if PlotCount[PlotNumber] > 0 then
          begin
            writeln('Number of lots with perimeter consisting of',PlotNumber,
                    'surveyor''s lines = ',PlotCount[PlotNumber]);
            inc(TotalPlotCount,PlotCount[PlotNumber]);
          end;

      writeln('Total number of lots = ',TotalPlotCount);
      writeln;
    end;

begin {Main}
    assign(input,'ACM93FB.IN');
    assign(output,'ACM93FB.OUT');
    reset(input);
    rewrite(output);

    CaseNumber := 0;
    while InputData do                               {当输入还未结束时}
```

```
  begin
    PreProcess;                                  {预处理}
    FindAll;                                     {找出所有土地}
    Print;                                       {打印输出}
  end;

 close(input);
 close(output);
end .{Main}
```

8.3 寻找堂亲

1. 解题思路

这道题目的关键在于正确地理解题意。从题末的图解可以看出,所有的亲戚关系构成了一棵关系树。题目描述中的 p-后裔关系用树的语言来说就是一人是另一人的 p 代直接祖先;m-n-堂亲关系用树的语言来说就是某人的 m 代直接祖先与另一人的 n 代直接祖先是同一个人。

现在来求 A 与 B 之间的最近关系。可以对 A、B 分别构造两张祖先列表,将其所有直接祖先按与其的辈分差距从小到大顺序列出(这里广义地定义一个人是其自身的 0 代祖先)。

例如,Phil 的祖先列表为:

Phil -> Jean -> Sue -> Don(这里 Sue 和 Don 的位置可以互换)

构造好这两张列表之后,我们应先分别检查 A、B 是否在另一个人的祖先列表中出现,如果出现,则他就是另一个人的祖先,A、B 之间是后裔关系。

如果没有找到后裔关系,则用两重循环找出在 A、B 祖先列表中均出现、且距 A、B 的距离最近的人。如果存在这样的共同祖先,则 A、B 之间是堂亲关系,否则便没有关系。

2. 程序清单

```
{$A+,B-,D-,E-,F-,G+,I-,L-,N-,O-,P-,Q-,R-,S-,T-,V+,X-,Y-}
{$M 65520,0,655360}
program Kissin_Cousins;                          {寻找堂亲}

 type
   TName=string[5];                              {姓名类型}

 var
   Line: string;                                 {输入行}
   Name: array[1..255]of TName;                  {姓名列表}
```

```
  NameCount, RelationCount: byte;                    {姓名总数、关系总数}
  Relation: array[1..1000,1..3] of byte;             {关系:}
                                                     {第一维是关系编号}
                                                     {第二维依次是第一、二}
                                                     {个人、他们之间的关系}

function InputData: boolean;                         {读入一个输入行}
                                                     {并返回输入结束信息}

  begin
    readln(Line);
    while Line[0] < #11 do
      Line := Line + '';
    InputData := Line[1] <> 'E';
  end;

function Location(Nam: TName):byte;                  {返回姓名编号}

  var
    I: byte;

  begin
    for I := 1 to NameCount do
      if Name[I] = Nam then
        begin
          Location := I;
          exit;
        end;

    inc(NameCount);
    Name[NameCount] := Nam;
    Location := NameCount;
  end;

function GetValue(S: string):integer;                {返回 S 串包含的第一个整数}

  var
    D,Code,Cod: integer;

  begin
    val(S,D,Code);
    if Code > 0 then
```

```
        val(copy(S,1,Code - 1),D,Cod);
      GetValue := D;
    end;

  procedure AddRelation(First,Second,Dist: byte);        {添加两人之间的最近关系}

    begin
      inc(RelationCount);
      Relation[RelationCount,1] := First;
      Relation[RelationCount,2] := Second;
      Relation[RelationCount,3] := Dist;
    end;

  procedure FindCousin(First,Second: byte);              {寻找两人之间的最近关系}

    type

      TList = array[1..255,1..2]of byte;                 {列表类型}

    function MakeList(var List:TList;Man:byte):byte;
                                                         {构造 Man 的祖先列表 List}

      var
        I,J,Ins,Open,Closed: byte;

      begin
        Open := 1;
        List[1,1] := Man;
        List[1,2] := 0;
        Closed := 0;

        while Closed < Open do
          begin
            inc(Closed);
            for I := 1 to RelationCount do
              if(Relation[I,1] = List[Closed,1])then
                begin
                  inc(Open);
                  List[Open,2] := List[Closed,2] + Relation[I,3];

                  Ins := Closed;
                  while(Ins < Open)and(List[Ins,2] <= List[Closed,2])do
```

```
                inc(Ins);

              for J := Open - 1 downto Ins do
                List[J+1] := List[J];
              List[Ins,1] := Relation[I,2];
              List[Ins,2] := List[Closed,2] + Relation[I,3];
            end;
        end;

      MakeList := Open;
    end;

var
  FirstList,SecondList: TList;                    {第一、二个人的祖先列表}
  FirstCount,SecondCount,                         {第一、二个人的祖先数目}
  P,I,J,M,N,NewM,NewN: byte;

begin {FindCousin}
  FirstCount := MakeList(FirstList,First);        {构造第一个人的祖先列表}
  SecondCount := MakeList(SecondList,Second);     {构造第二个人的祖先列表}
  P := 0;
  for I := 1 to FirstCount do                     {检查第一个人是不是}
                                                  {第二个人的祖先}

    if FirstList[I,1] = Second then
      begin
        P := FirstList[1,2];
        break;
      end;

  if P = 0 then
    for I := 1 to SecondCount do                  {检查第二个人是不是}
                                                  {第一个人的祖先}
      if SecondList[I,1] = First then
        begin
          P := SecondList[I,2];
          break;
        end;

  if P > 0 then                                   {如果一人是另一人的祖先}
    begin                                         {则他们之间是后裔关系}
      writeln(Name[First],'and',Name[Second],'are descendant - ',P,'.');
```

```
            exit;
          end;

        M := 255;
        N := 255;

        for I := 1 to FirstCount do                    {寻找两人的最近公共祖先}
          for J := 1 to SecondCount do
            if FirstList[I,1] = SecondList[J,1]
              then
                begin
                  if FirstList[I,2] < = SecondList[J,2]
                    then
                      begin
                        NewM := FirstList[I,2] - 1;
                        NewN := SecondList[J,2] - FirstList[I,2];
                      end;
                    else
                      begin
                        NewM := SecondList[J,2] - 1;
                        NewN := FirstList[I,2] - SecondList[J,2];
                      end;

                  if(NewM < M)or((NewM = M)and(NewN < N))then
                    begin
                      M := NewM;
                      N := NewN;
                    end;
                end;

        if M< > 255                                    {如果两人有公共祖先}
          then                                         {则他们是堂亲关系}
            writeln(Name[First], 'and', Name[Second], 'are cousin - ', m, '- ', n, '.')
          else                                         {否则他们之间没有任何关系}
            writeln(Name[First] .'and', Name[Second], 'are not related .');
      end;{FindCousin}

    begin {Main}
      assign(input, 'ACM93FC .IN');
      assign(output, 'ACM93FC .OUT');
      reset(input);
      rewrite(output);
```

```
    NameCount := 0;
    RelationCount := 0;

    while InputData do                                    {当输入还未结束时}
      begin
        case Line[1] of                                   {根据输入行做相应处理}
          '# ': ;                                         {注释行}
          'R': AddRelation(Location(copy(Line,2,5)),
                    Location(copy(Line,7,5)),GetValue(copy(Line,12,length(Line) - 11)));
                                                          {添加关系}
          'F': FindCousin(Location(copy(Line,2,5)),Location(copy(Line,7,5)));
                                                          {寻找两人之间的最近关系}
        end;
      end;

    close(input);
    close(output);
  end .{Main}
```

8.4 黄 金 图 形

1. 解题思路

由于题目需要求出所有解,因此采用回溯算法是比较适合的。从(0,0)出发,每次比前次多走一个街区的长度。直至回到出发点,就找到了一个黄金图形。

这里有两个要点:

(1) 如何表述拐弯?

用四个常数代表四个方向,这些常数的集合就表示当前可走方向的集合;

(2) 如何判断路线上是否有路障存在?

对于所有的路障,依次检查其是否在当前路线上即可。

2. 程序清单

```
{$A+,B-,D-,E-,F-,G+,I-,L-,N-,O-,P-,Q-,R-,S-,T-,V+,X-,Y-}
{$M 65520,0,655360}
program Golygons;                                         {黄金图形}

  const
    DX: array[1..4] of shortint = (0,0,1,-1);
    DY: array[1..4] of shortint = (1,-1,0,0);
```

```
  DirName: array[1..4] of char = 'nsew';                    {方向名称}

var
  CityCount,                                                {城市数目}
  CityNumber,                                               {城市编号}
  LastLength,                                               {最后一条边的长度}
  BlockCount: byte;                                         {被封锁的街区数目}
  Block: array[1..50,1..2] of integer;                      {被封锁的街区}

procedure InputData;                                        {读入城市信息}

  var
    I: byte;

  begin
    readln(LastLength);
    readln(BlockCount);
    for I := 1 to BlockCount do
      readln(Block[I,1],Block[I,2]);
  end;

procedure FindGolygons;                                     {寻找黄金图形}

  var
    GolygonCount: byte;                                     {找到的黄金图形数目}
    Path: array[1..20] of byte;                             {走的路线}

  function NoBlock(X,Y: integer;Length,Dir: byte):boolean;
                                                            {返回能否从(X,Y)}
                                                            {向 Dir 方向前进 Length 长度}

    var
      I: byte;

    begin
      NoBlock := false;

      if DX[Dir] = 0
        then
          if DY[Dir] = 1
            then
              begin
```

```
                for I := 1 to BlockCount do
                  if(Block[I,1] = X)and(Block[I,2] > = Y)
                        and(Block[I,2] < = Y + Length)then
                        exit;
              end
            else
            begin
              for I := 1 to BlockCount do
                if(Block[I,1] = X)and(Block[I,2] < = Y)
                      and(Block[I,2] > = Y - Length)then
                      exit;
            end
        else
          if DX[Dir] = 1
            then
              begin
                for I := 1 to BlockCount do
                  if(Block[1,2] = Y)and(Block[I,1] > = X)
                    and(Block[I,1] < = X + Length)then
                    exit;
              end
            else
              begin
                for I := 1 to BlockCount do
                  if(Block[I,2] = Y)and(Block[I,1] < = X)
                    and(Block[I,1] > = X - Length)then
                    exit;
              end;

      NoBlock := true;
    end;

procedure Go(X,Y:integer;Length,LastDir:byte);
                                                    {递归搜索构造黄金图形}
                                                    {X,Y      当前坐标}
                                                    {Length   上一次的步长}
                                                    {LastDir  上一次的方向}

  var
    Dir,I: byte;
    DirSet: set of 1..4;
```

```
  begin
    if Length = LastLength + 1 then
      begin
        if(X = 0)and(Y = 0)then                    {如果回到了出发点}
          begin                                    {则找到了一个新的黄金图形}
            inc(GolygonCount);
            for I := 1 to LastLength do
              write(DirName[Path[I]]);
            writeln;
          end;
        exit;
      end;

    if LastDir in[1,2]                             {确定当前可走方向}
      then
        DirSet := [3,4]
      else
        if LastDir in[3,4]
          then
            DirSet := [1,2]
          else
            DirSet := [1 .4];

    for Dir := 1 to 4 do                           {对于每一个方向}
      if(Dir in DirSet)and NoBlock(X,Y,Length,Dir)then
                                                   {如果可以前进}
        begin                                      {则尝试前进}
          Path[Length]:= Dir;
          Go(X + Length * DX[Dir],Y + Length * DY[Dir],Length + 1,Dir);
        end;
  end;

begin {FindGolygons}
  GolygonCount := 0;

  Go(0,0,1,0);                                     {从(0,0)出发}
                                                   {向四个方向构造黄金图形}

  writeln('Found',GolygonCount,'golygon(s) .');
  writeln;
end;{FindGolygons}
```

```
begin {Main}
  assign(input, 'ACM93FD .IN');
  assign(output, 'ACM93FD .OUT');
  reset(input);
  rewrite(output);

  readln(CityCount);

  for CityNumber := 1 to CityCount do                    {对于每个城市}
    begin
      InputData;                                         {读入城市信息}
      FindGolygons;                                      {寻找黄金图形}
    end;

  close(input);
  close(output);
end .{Main}
```

8.5 MIDI 预处理

1. 解题思路

对于这道题目,注意到各音符之间的开闭情况是相互独立的,所以应对每个音符独立保存各种信息并独立地进行处理。对于新的指令,如果构成了题目中所描述的几种情况,则应根据要求对程序进行修正。详细说明可见程序注释。

这里有一个问题。如果我们可以将所有的指令都存放在内存中,处理起来自然不成问题。然而题目中并没有限制指令的数目,从实用的角度来考虑,应当使程序能够处理的指令数目与内存大小无关。

考虑到音符的数目是有限的,并且对于每个音符,预处理时只会涉及最近若干条指令,所以可以在内存中开一个指令缓冲区,每次输出、删除、插入指令都是对缓冲区进行操作。仅当缓冲区满的时候,才将以后肯定不会再修改的指令按时刻顺序写入输出。一个程序处理完后,将缓冲区中的指令全部写入输出。这样就可以处理任意长度的 MIDI 程序了。

2. 程序清单

```
{$A+,B-,D-,E-,F-,G+,I-,L-,N-,O-,P-,Q-,R-,S-,T-,V+,X-,Y-}
{$M 65520,0,655360}
program MIDI_Preprocessing;                              {MIDI 预处理}
```

```
const
  CommandString: array[boolean]of string[3] = ('OFF', 'ON');
  BufferSize = 254;                                    {缓冲区大小}

type
  TCommand =                                           {指令类型}
    record
      Time: integer;                                   {时刻}
      Play: boolean;                                   {开关}
      Note: byte;                                      {音符}
    end;

var
  Sound: array[1..127]of boolean;                      {音符的开闭情况}
  DoubleOnFlag: array[1..127]of boolean;               {音符接连两次打开标志}
  LastOn,LastOff: array[1..127]of integer;             {前次打开、关闭音符的时刻}
  Buffer: array[1..BufferSize]of TCommand;             {输出指令缓冲区}
  NowCommand: TCommand;                                {当前指令}
  BufferCount: byte;

procedure InputLine;                                   {读入一行指令}

  var
    Code: integer;
    S: string;

  begin
    readln(S);
    S := S + ' ';
    val(copy(S,1,pos(' ',S) - 1),NowCommand.Time,Code);

    if NowCommand.Time < 0 then
      exit;

    dec(S[0]);
    delete(S,1,pos(' ',S));
    NowCommand.Play := copy(S,1,pos(' ',S) - 1) = CommandString[true];

    delete(S,1,pos(' ',S));
    val(S,NowCommand.Note,Code);
  end;
```

```
procedure AppendBuffer(Time:integer; Play:boolean; Note:byte);
                                                    {向缓冲区加入一条指令}

  var
    I,K: byte;
  begin
    K := BufferCount;
    while(K > 0)and(Buffer[K].Time > Time)do
      dec(K);
    inc(K);

    for I := BufferCount downto K do
      Buffer[I + 1] := Buffer[I];

    Buffer[K].Time := Time;
    Buffer[K].Play := Play;
    Buffer[K].Note := Note;

    inc(BufferCount);
    if BufferCount = BufferSize then
      begin
        for I := 1 to 127 do
          with Buffer[I]do
            writeln(Time, '',CommandString[Play], '',Note);

        dec(BufferCount,127);
        for I := 1 to BufferCount do
          Buffer[I] := Buffer[I + 127];
      end;
  end;

procedure RemoveBuffer(Time:integer;Play:boolean;Note:byte);
                                                    {从缓冲区中移去一条指令}

  var
    I,K: byte;

  begin
    K := BufferCount;
    while (K > 0)and not((Buffer[K].Time = Time)and
                (Buffer[K].Play = Play)and(Buffer[K].Note = Note))do
      dec(K);
```

```
    for I := K to BufferCount - 1 do
      Buffer[I] := Buffer[I + 1];
    dec(BufferCount);
  end;

function FindBuffer(Time:integer;Play;boolean;Note:byte):boolean;
                                                    {在缓冲区中寻找一条指令}

  var
    I: byte;

  begin
    for I := 1 to BufferCount do
      if(Buffer[I].Time = Time)and(Buffer[I].Play = Play)and(Buffer[I].Note = Note)
then
        begin
          FindBuffer := true;
          exit;
        end;
    FindBuffer := false;
  end;

procedure CleanUp;                                  {将缓冲区信息全部写入输出}
                                                    {并为下一程序作初始化工作}

  var
    I: byte;

  begin
    if BufferCount > 0 then
      begin
        for I := 1 to BufferCount do
          with Buffer[I]do
            writeln(Time, '',CommandString[Play], '', Note);
        writeln( - 1);
      end;

    fillchar(Sound,sizeof(Sound),0);
    fillchar(DoubleOnFlag, sizeof(DoubleOnFlag),0);
    fillchar(Buffer, sizeof(Buffer),0);
    for I := 1 to 127 do
      LastOn[I] := - 1;
```

```
    for I := 1 to 127 do
      LastOff[I] := - 1;
    BufferCount := 0;
  end;

procedure Process;                                    {处理指令}

  begin
    with NowCommand do
      if Play
        then                                          {如果是打开指令}
          if Sound[Note]
            then                                      {如果音符已打开}
              begin                                   {则在前面插入一个关闭指令}
                AppendBuffer(Time - 1,false,Note);
                AppendBuffer(Time,true,Note);
                LastOn[Note] := Time;
                DoubleOnFlag[Note] := true;
              end
            else                                      {如果音符未打开}
              if LastOff[Note] = Time
                then                                  {如果前次关闭时刻}
                                                      {同当前时刻相同}
                  if LastOn[Note] = Time - 1
                    then                              {如果相邻两次打开紧接着}
                      begin                           {则删去前次关闭指令}
                        RemoveBuffer(Time,false,Note);
                        LastOff[Note] := - 1;
                        Sound[Note] := true;
                      end
                    else                              {否则移动前次关闭指令}
                      begin
                        RemoveBuffer(Time,false,Note);
                        AppendBuffer(Time - 1,false,Note);
                        AppendBuffer(Time,true,Note);
                        LastOff[Note] := Time - 1;
                        LastOn[Note] := Time;
                        Sound[Note] := true;
                      end
                else                                  {如果前次关闭时刻}
                                                      {同当前时刻不同}
                  begin                               {则作正常处理}
                    Sound[Note] := true;
```

```
                    AppendBuffer(Time, true, Note);
                    LastOn[Note] := Time;
                  end
            else                                    {否则是关闭指令}
              if Sound[Note]
                then                                {如果音符已打开}
                  if DoubleOnFlag[Note]
                    then                            {如果音符是二次打开}
                      DoubleOnFlag[Note] := false   {则不输出第二条关闭指令}
                    else
                      if LastOn[Note] = Time
                        then                        {如果前次打开时刻}
                                                    {同当前时刻相同}
                          if FindBuffer(Time - 1, true, Note)
                            then                    {如果在这之前的一次}
                                                    {打开指令与之紧接着}
                              begin                 {则删去前次打开指令}
                                RemoveBuffer(Time, true, Note);
                                LastOn[Note] := Time - 1;
                              end
                            else                    {否则加入一个关闭指令}
                              begin
                                AppendBuffer(Time - 1, false, Note);
                                LastOff[Note] := Time - 1;
                              end
                        else                        {如果前次打开时刻}
                                                    {同当前时刻不同}
                          begin                     {则作正常处理}
                            Sound[Note] := false;
                            AppendBuffer(Time, false, Note);
                            LastOff[Note] := Time;
                          end;
    end;

  begin {Main}
    assign(input, 'ACM93FE.IN');
    assign(output, 'ACM93FE.OUT');
    reset(input);
    rewrite(output);

    BufferCount := 0;
    CleanUp;
```

```
    repeat
      InputLine;                                {读入指令}
      if NowCommand.Time<0                      {如果是结束标志}
        then                                    {则将缓冲区的信息全部输出}
                                                {并为下一程序作初始化工作}
          CleanUp
        else                                    {否则则处理指令}
          Process;

    until NowCommand.Time=-2;                   {直到输入结束}

    writeln(-2);

    close(input);
    close(output);
end.{Main}
```

8.6 魔　　板

1. 解题思路

本题不算很难,只要读入魔板局面,依次进行移动即可。但需要注意当初始局面的空位正好处在最后一列上时,空格所在的行将只有4个字符,这时需要程序能够作相应的处理。

2. 程序清单

```
{$A+,B-,D-,E-,F-,G+,I-,L-,N-,O-,P-,Q-,R-,S-,T-,V+,X-,Y-}
{$M 65520,0,655360}
program Puzzle;                                 {魔板}

  var
    PuzzleNumber: byte;                         {魔板编号}
    Frame: array[1..5,1..5]of char;             {底板}
    Move: char;                                 {当前移动}
    Illegal: boolean;                           {合法性标志}

  function InputConfig: boolean;                {读入初始局面}
                                                {并返回输入结束信息}

    var
      Ch: char;
```

```
    I,J: byte;

  begin
    read(Ch);

    if Ch = 'Z' then
      begin
        InputConfig := false;
        exit;
      end;

    for I := 1 to 5 do
      begin
        for J := 1 to 5 do
          begin
            if (I < > 1)or( J< > 1) then
              if eoln
                then
                  Ch := ''
                else
                  read( Ch) ;

              Frame[I,J] := Ch;
          end;
        readln;
      end;

    Illegal := false;
    inc(PuzzleNumber) ;
    InputConfig := true;
  end;

functioin InputMove: boolean;                    {读入移动信息}
                                                 {并返回移动序列结束信息}

  begin
    read(Move) ;
    while eoln do
      readln;
    InputMove := Move< > '0';
  end;
```

```
procedure MakeMove;                                    {作相应移动}

  procedure LocateEmpty(var X,Y:byte);                 {寻找空位位置}

    var
      I,J: byte;

    begin
      for I := 1 to 5 do
        for J := 1 to 5 do
          if Frame[I,J] = ''then
            begin
              X := I;
              Y := J;
              exit;
            end;
    end;

  var
    X,Y,NewX,NewY: byte;

  begin {LocateEmpty}
    if Illegal then                                    {如果已有不合法移动}
      exit;                                            {则不再移动空位}

    LocateEmpty(X,Y);                                  {寻找空位位置}

    NewX := X;
    NewY := Y;
    case Move of                                       {移动}
      'A':NewX := X - 1;
      'B':NewX := X + 1;
      'R':NewY := Y + 1;
      'L':NewY := Y - 1;
    end;

    if(NewX in[1..5])and(NewY in[1..5])
      then
        begin
          Frame[X,Y] := Frame[NewX,NewY];
          Frame[NewX,NewY] := '';
        end
```

```
        else
          Illegal := true;

    end;{LocateEmpty}

  procedure Print;                                    {打印输出魔板局面}

    var
      I,J: byte;

    begin
      writeln('Puzzle # ',PuzzleNumber,':');
      if Illegal
        then
          writeln('      This puzzle has no final configuration .')
        else
          for I := 1 to 5 do
            begin
              write('   ');
              for J := 1 to 5 do
                write(Frame[I,J]:2);
              writeln;
            end;
      writeln;
    end;

  begin {Main}
    assign(input,'ACM93FF .IN');
    assign(output,'ACM93FF .OUT');
    reset(input);
    rewrite(output);

    PuzzleNumber := 0;
    while InputConfig do                              {当初始局面输入还未结束时}
      begin
        while InputMove do                            {当移动序列还未结束时}
          MakeMove;                                   {作相应的移动}

        Print;                                        {打印输出魔板局面}
      end;

    close(input);
    close(output);
  end .{Main}
```

8.7 资 源 分 配

1. 解题思路

这是一道关于规划的试题,可以考虑用动态规划算法求解。但由于资源有两种,所以这里的动态规划将是二维的。考虑到预算资金有可能取很大的值,因此在这道题目里使用动态规划算法并不是最好的办法。事实上,由于题目将编程小组和资源分配方案的数量限制在一个较小的范围内,如果采用搜索算法求解,效率并不会很低。参考程序就是采用了回溯算法求解的。算法本身很简单,但如果描述编程小组和资源分配方案的数据结构设计得不好,程序就很难写。具体可见程序注释。

2. 程序清单

```
{$A+,B-,D-,E-,F-,G+,I-,L-,N-,O-,P-,Q-,R-,S-,T-,V+,X-,Y-}
{$M 65520,0,655360}
program Resource_Allocation;                          {资源分配}

  type
    TDivision =                                       {编程小组类型}
      record
        ProgrammerOptionCount: byte;                  {新程序员选择数目}
        ProgrammerOption: array[1..10]of word;        {新程序员选择的列表}
        BudgetOptionCount: byte;                      {新预算选择数目}
        BudgetOption: array[1..10]of longint;         {新预算选择的列表}
        CodeProduced: array[1..10,1..10]of longint;   {各种选择下的代码增量}
      end;

  var
    ProblemNumber: byte;                              {问题编号}
    DivisionCount: byte;                              {编程小组数目}
    ProgrammerCount: word;                            {新程序员的总数}
    BudgetLimit: longint;                             {新预算的总额}
    TotalProgrammer: word;                            {雇佣的新程序员总数}
    TotalBudget: longint;                             {所用资金总额}
    TotalProductivityIncrement: longint;              {创造的新生产能力的总和}
    Division: array[1..20]of TDivision;               {编程小组}
    Allocation,NowAllocation: array[1..20,1..2]of byte;
                                                      {最优及当前资源分配计划}

  function InputData: boolean;                        {读入问题}
```

{并返回输入结束信息}

```
var
  DivisionNumber, ProgrammerOptionNumber, BudgetOptionNumber: byte;

begin
  readln(DivisionCount);

  if DivisionCount = 0 then
    begin
      InputData := false;
      exit;
    end;

  readln(ProgrammerCount);
  readln(BudgetLimit);

  fillchar(Division, sizeof(Division), 0);
  for DivisionNumber := 1 to DivisionCount do
    with Division[DivisionNumber]do
      begin
        readln(ProgrammerOptionCount);
        for ProgrammerOptionNumber := 1 to ProgrammerOptionCount do
          read(ProgrammerOption[ProgrammerOptionNumber]);
        readln;

        readln(BudgetOptionCount);
        for BudgetOptionNumber := 1 to BudgetOptionCount do
          read(BudgetOption[BudgetOptionNumber]);
        readln;

        for ProgrammerOptionNumber := 1 to ProgrammerOptionCount do
          begin
            for BudgetOptionNumber := 1 to BudgetOptionCount do
              read(CodeProduced[ProgrammerOptionNumber, BudgetOptionNumber]);
            readln;
          end;
      end;

  inc(ProblemNumber);
```

```
    InputData := true;
  end;

procedure FindOptimal;                                  {寻找最优资源分配方案}

  procedure Allocate                                    {递归搜索最优资源分配方案}
      (DivisionNumber: byte;                            {要决定资源分配的编程小组}
      NowProgrammer: word;                              {已雇佣的新程序员总数}
      NowBudget,                                        {已用的资金总额}
      NowProductivityIncrement: longint);               {已得到的生产力增量}

    var
      ProgrammerOptionNumber,BudgetOptionNumber: byte;

    begin
      if DivisionNumber > DivisionCount then
        begin
          if NowProductivityIncrement > TotalProductivityIncrement then
            begin
              TotalProductivityIncrement := NowProductivityIncrement;
              TotalBudget := NowBudget;
              TotalProgrammer := NowProgrammer;
              Allocation := NowAllocation;
            end;
          exit;
        end;

      with Division[DivisionNumber]do
        for ProgrammerOptionNumber := 1 to ProgrammerOptionCount do
          if NowProgrammer + ProgrammerOption[ProgrammerOptionNumber] < =
            ProgrammerCount then
            for BudgetOptionNumber := 1 to BudgetOptionCount do
              if NowBudget + BudgetOption[BudgetOptionNumber] < = BudgetLimit then
                begin
                  NowAllocation[DivisionNumber,1]:= ProgrammerOptionNumber;
                  NowAllocation[DivisionNumber,2]:= BudgetOptionNumber;

                  Allocate(DivisionNumber + 1,
                    NowProgrammer + ProgrammerOption[ProgrammerOptionNumber],
                    NowBudget + BudgetOption[BudgetOptionNumber],
```

```
                        NowProductivityIncrement +
                        CodeProduced[ProgrammerOptionNumber,BudgetOptionNumber]);
                    end;
        end;

    begin {FindOptimal}
        TotalProductivityIncrement := 0;
        Allocate(1,0,0,0);                          {递归搜索最优资源分配方案}
    end;{FindOptimal}

procedure Print;                                    {打印输出最优资源分配方案}

    var
        DivisionNumber: byte;

    begin
        writeln('Optimal resource allocation problem #',ProblemNumber);
        writeln;
        writeln('Total budget: $',TotalBudget);
        writeln('Total new programmers:',TotalProgrammer);
        writeln('Total productivity increase:',TotalProductivityIncrement);

        for DivisionNumber := 1 to DivisionCount do
            with Division[DivisionNumber]do
                begin
                    writeln;
                    writeln('Division #',DivisionNumber,'resource allocation:');
                    writeln('Budget: $',BudgetOption[Allocation[DivisionNumber,2]]);
                    writeln('Programmers:',ProgrammerOption[Allocation[DivisionNumber,1]]);
                    writeln('Incremental lines of code:',
                    CodeProduced[Allocation[DivisionNumber, 1], Allocation[DivisionNumber,
2]];
                end;

        writeln;
        writeln;
    end;

begin {Main}
    assign(input,'ACM93FG .IN');
```

```
  assign(output, 'ACM93FG.OUT');
  reset(input);
  rewrite(output);

  ProblemNumber := 0;
  while InputData do                          {当输入还未结束时}
    begin
      FindOptimal;                            {寻找最优资源分配方案}
      Print;                                  {打印输出最优资源分配方案}
    end;

  close(input);
  close(output);
end.{Main}
```