全国青少年信息学奥林匹克竞赛教程 第五版

（C++ 版）

信息学奥赛一本通

XINXIXUE AOSAI YIBENTONG

◆ 董永建 宋新波 李 建
张新钢 舒春平 邹 毅 著
黄新军 徐先友 叶国平



科学技术文献出版社
SCIENTIFIC AND TECHNICAL DOCUMENTATION PRESS

前　言

C++是一种混合型的面向对象程序设计语言。它既有面向对象的特征，又具有对传统C语言的向后兼容性，具备结构化程序设计的特征。由于C++涉及的概念多、内容广、语法比较复杂，不少初学者觉得学习难度大、入门难。

本书针对这些问题，合理安排内容，精心设计实例，引导读者一步一步地走入程序设计的知识殿堂，使读者不再感觉学习C++很累、很难，达到入门的目的。相信通过本书的讲解，您将觉得C++要比许多人想象的容易理解。

本书有三大特点：

一是详略得当，主次分明。

C++中包含的内容较多，由于篇幅的限制，不可能面面俱到，必须有所舍弃。书中对于非重点或较复杂的内容略讲，如数组部分重点是一维、二维数组和字符串，三维数组只是提一下，在结构与联合这部分，重点讲解结构，而联合的内容基本没介绍。

二是例题生动，实用性强。

本书针对C++特点，精选重点，强化主要概念，图文并茂地讲解每个重要知识点，并配以较多容易理解的程序实例，以例题释含义、总结出规律，便于理解和应用。同时在每一章的主要内容讲解之后，充分利用前面的知识，将多个知识点有机地结合起来，设计了有一定难度的综合实例，以加强对所学知识的理解和运用。

三是配有练习，学练结合。

本书的每一章节都有适当的练习题，习题典型多样，覆盖本章知识点。读者在学完一章节后，通过练习达到巩固强化的作用。

本书使用了Noip的原题和从网络收集的部分资料，由于时间跨度较长，许多资料难以找到原创作者。在此，向这些素材的作者表示衷心感谢。

本书既可以作为中学生信息学竞赛活动的培训教材,亦可作为普通高等院校电子信息类专业程序设计基础的教材,还可供算法爱好者自学参考使用。

参与本书编写工作的,除了在一线教学的信息学奥赛教师,还有16位在校本科生和研究生,他们在中学时代至少获得全国青少年信息学奥林匹克联赛一等奖。参与第一部分《C++语言》编写的是北京师范大学李贤恩、武汉大学林有我、游明伟、同济大学蔡乐文、北京航空航天大学金胜莺;参与第二部分《基础算法》编写的是浙江大学林锦、林升、同济大学刘剑豪、刘心楠、林鑫、陈文韬、许晨星;参与第三部分《数据结构》编写的是清华大学张伟奇、浙江大学阮思绮、中国科技大学陈箫翰、朱佳凡,他们很好的分工合作,圆满完成了编写工作。

尽管本书主编是多年讲授C++课程的教师,有着比较丰富的教学经验,但是,由于受到篇幅和写作时间的限制,书中仍然可能存在一些不足之处,恳请使用本书的教师、学生和其他读者批评指正,以便再版时修正。

我们的E-mail地址是:fjclyz2006@163.com

目　录

第一部分　C++语言

第一章　C++语言入门 …… 3
第一节　C++语言简介 …… 3
第二节　C++语言程序结构 …… 12
第二章　顺序结构程序设计 …… 16
第一节　赋值语句 …… 16
第二节　运算符和表达式 …… 18
第三节　常量和变量 …… 23
第四节　标准数据类型 …… 25
第五节　数据输入输出 …… 30
第六节　顺序结构实例 …… 39
第三章　程序的控制结构 …… 43
第一节　概　述 …… 43
第二节　if 选择结构 …… 44
第三节　switch 语句 …… 51
第四章　循环结构 …… 56
第一节　for 语句 …… 56
第二节　while 语句 …… 61
第三节　do－while 语句 …… 67
第四节　循环嵌套 …… 70
第五章　数　组 …… 79
第一节　一维数组 …… 79
第二节　二维数组 …… 90
第三节　字符类型和字符数组 …… 99
第六章　函　数 …… 114
第一节　函　数 …… 114
第二节　递归算法 …… 132
第七章　文件和结构体 …… 139
第一节　文件操作 …… 139
第二节　结构体 …… 143
第八章　指针与链表 …… 149
第一节　指针变量 …… 149

第二节　指针与数组 …… 154
第三节　指针与字符串 …… 158
第四节　指针与函数 …… 160
第五节　结构体指针 …… 165
第六节　链表结构 …… 168

第二部分　基础算法

第一章　高精度计算 …… 181
第二章　数据排序 …… 193
第三章　递推算法 …… 213
第四章　递归算法 …… 228
第五章　搜索与回溯算法 …… 245
第六章　贪心算法 …… 265
第七章　分治算法 …… 281
第八章　广度优先搜索算法 …… 301
第九章　动态规划 …… 320
第一节　动态规划的基本模型 …… 320
第二节　背包问题 …… 349
第三节　动态规划经典题 …… 371

第三部分　数据结构

第一章　栈 …… 395
第二章　队　列 …… 404
第三章　树 …… 414
第一节　树的概念 …… 414
第二节　二叉树 …… 421
第三节　堆及其应用 …… 441
第四章　图论算法 …… 461
第一节　基本概念 …… 461
第二节　图的遍历 …… 464
第三节　最短路径算法 …… 471
第四节　图的连通性问题 …… 493
第五节　并查集 …… 496
第六节　最小生成树 …… 508
第七节　拓扑排序与关键路径 …… 519
附录　C++常用库函数 …… 530

5. 生成的代码质量高

C++语言在代码效率方面可以和汇编语言相媲美,一般只比汇编程序生成的目标代码效率低10%～20%。

6. 可移植性强

C++语言编写的程序很容易进行移植,在一个环境下运行的程序不加修改或少许修改就可以在完全不同的环境下运行。

C++语言也存在不足,例如,运算符多,使用灵活,但难记、难用,有些运算符在某种情况下会产生二义性;类型转换较为随便,容易造成数据混乱;数组的定义使用方便,但不作越界检查,容易导致数据出错。对学员要求也高,用C++语言编写程序会感到限制少、灵活性大、功能强,较其他高级语言在学习上要困难一些。

二、简单的C++语言程序实例

在介绍C++语言程序的结构之前,我们先来看一个简单的例子:

例1.1 在屏幕上输出"Hello World!"。

```
#include<iostream>    //使用 cin,cout,须调用 iostream 库,否则编译出错
#include<cstdlib>     // 使用 system()调用 cstdlib 库
using namespace std;  //在 C 语言中要省略,例如在 VC++和 TC++中
int main()            //有的 C 语言可用 void main(),例如 TC++和 VC++
{
  cout<<" Hello  World! "<<endl;  //输出" Hello World! "
  system(" pause ");   //暂停作用,使用 system(" pause ")调用 cstdlib 库
  return 0;            //结束整个程序,在 TC++和VC++中也要保留
}
```

运行结果:

```
Hello World!
```

【说明】

1. 以"//"开头为注释行,"//"后的内容用以对语句进行说明,输入程序时可以不输入。

2. #include<iostream>

告诉编译器的预处理器将输入输出流的标准头文件(iostream)包括在本程序中。这个头文件包括了C++中定义的基本标准输入输出程序库的声明。

3. using namespace std

使用std(标准)名字空间的意思。所谓的名字空间是标准C++中的一种机制,用来控制不同类库的冲突问题。使用它可以在不同的空间内使用相同名字的类或者函数。

4. int main()

这一行为主函数(main function)的起始声明。main()是所有C++程序的运行的起始点。不管它是在代码的开头,结尾还是中间,此函数中的代码总是在程序开始运行时第一个被执行。所有C++程序都必须有一个main(),int main()前int在Dev C++中可省略,在TC++和VC++中最好保留。

main后面跟了一对圆括号(),表示它是一个函数。C++中所有函数都跟有一对圆括

号()，括号中可以有一些输入参数。注意，圆括号中即使什么都没有也不能省略。如例题中显示，主函数 main()中的内容，由一对花括号{ }括起来。

5. cout << " Hello World! "<<endl

这个语句在本程序中最重要。cout 是一个输出语句，告诉计算机把引号之间的字符串送到标准的输出设备(屏幕)上。cout 的声明在头文件 iostream 中，所以要想使用 cout 必须将头文件 iostream 包括在程序开始处。endl 是 C++语言的换行控制符，表示内容输出后换行显示后续的内容。

6. system(" pause ")

在 4.99 以前版本的 Dev C++环境下，为了查看程序运行结果，需要在 main 函数的 return 0 语句前加上：system(" pause ")；这样程序运行到该语句时，结果显示屏幕将会停留，让大家有时间看程序的输出结果，否则结果显示屏幕将会一闪而过。在 4.99 之后的版本可不必加该语句，运行结束后结果显示屏幕将自动停留。

7. return 0

主函数 main()的返回语句，一般是函数的最后一条可执行语句。main()函数末尾使用 return 语句时，数值 0 表示程序顺利结束，其他数表示有异常。在后面的例子中你会看到 C++程序都以类似的语句结束。

在 C++中，语句的分隔是以分号“;”为分隔符的，分行写代码只是为了更方便人阅读。

三、C++语言系统的使用

Dev C++是一个可视化集成开发环境，可以用此软件实现 C/C++程序的编辑、预处理/编译/链接、运行和调试。现在介绍 Dev C++常用的一些基本操作，每一位同学都要掌握。

1. 启动 Dev C++

(1)鼠标点击任务栏中的“开始”按钮，选“程序”菜单项，然后选“程序”下的子菜单项“Bloodshed Dev C++”项，显示该项下的子菜单。

(2)单击“Dev C++”菜单项，即可启动 Dev C++集成开发工具(如图 1-1 所示)。

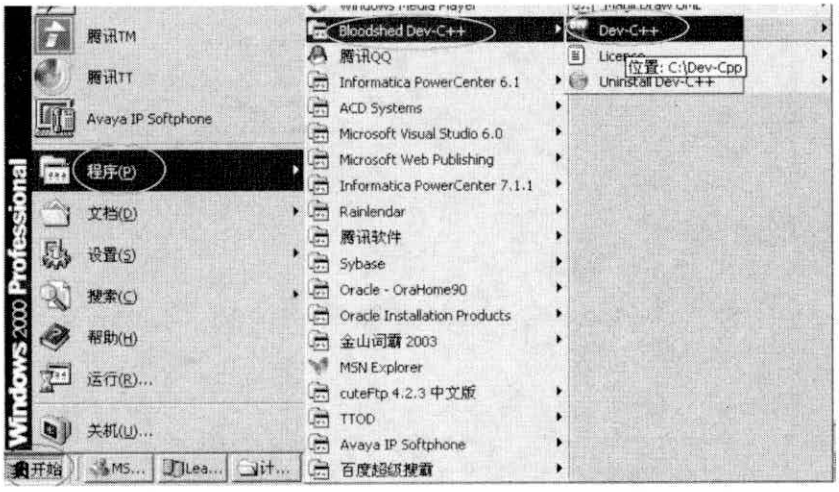


图 1-1

(3)方法二:直接双击桌面上的 Dev C++的图标。

2. 新建源程序

(1)从主菜单选择“文件”—〉“新建”—〉“源代码”即可(如图 1—2 所示)。

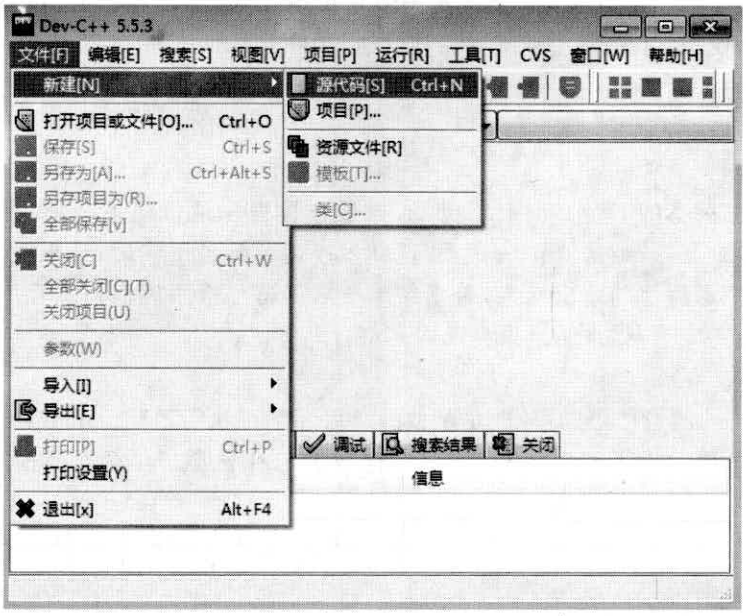


图 1—2

如果大家看到界面上的字是英文的,则可以点击主菜单“Tools > Environment Options”(如图 1—3),在弹出的对话框中选择第二个标签页“interface”(如图 1—4),在 Language 下拉列表中选择 Chinese 即可,将操作界面改为中文的。

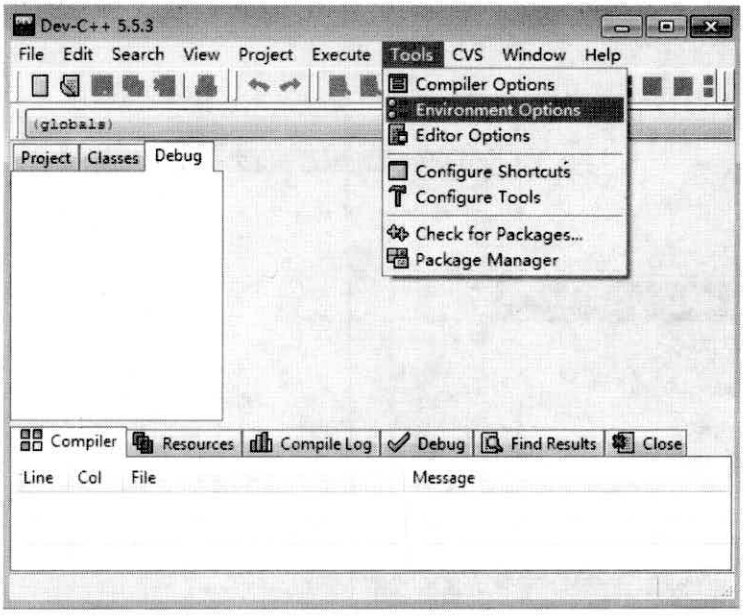
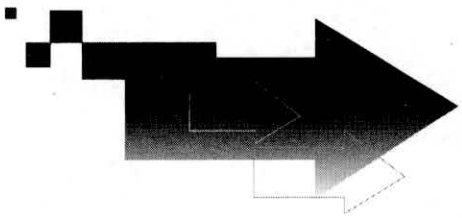


图 1—3

第一部分

C++语言



第一章　C＋＋语言入门

第一节　C＋＋语言简介

信息学奥林匹克竞赛是一项益智性的竞赛活动，核心是考查选手的智力和使用计算机解题的能力，选手首先应针对竞赛题目的要求构建数学模型，进而构造出计算机可以接受的算法，之后编写出计算机能够执行的程序。程序设计是信息学竞赛的基本功，选手参与竞赛活动的第一步是熟练掌握一门程序设计语言，目前竞赛中允许使用的程序设计语言有Pascal、C语言、C＋＋语言。

本书将介绍C＋＋语言的基本语法，但由于篇幅所限，不涉及面向对象的内容。本书中的所有程序，都没有使用面向对象的编程方法，但都应保存为.cpp文件，按C＋＋语法进行编译。实际上，如果不涉及面向对象的部分，那么C＋＋语言和C语言的语法90%以上是一样的，只不过略有扩充，用起来更为方便而已。本书提到的C＋＋语言特性，以目前流行的32位计算机及其操作系统上的情况为准。

一、C＋＋语言的特点

1. 语言简洁紧凑，使用灵活方便

C＋＋语言一共只有32个关键字和9种控制语句，程序书写自由，主要用小写字母表示。它把高级语言的基本结构和语句与低级语言的实用性结合起来，既具有高级语言的功能，又具有低级语言的很多特性。C＋＋语言可以像汇编语言一样对位、字节和地址进行操作，而这三者是计算机最基本的工作单元。

2. 运算符丰富

C＋＋语言的运算符包含的范围很广泛，共有34个运算符。运用这些运算符可构成简洁而功能强大的表达式，表达式的类型灵活多样，可以实现在其他高级语言中难以实现的运算。

3. 数据结构丰富

C＋＋语言的数据类型有：整型、实型、字符型、数组类型、指针类型、结构体类型、共用体类型等，能用来实现各种复杂的数据类型的运算，并引入了指针概念，使程序效率更高。

4. 结构化语言

结构化语言的显著特点是代码及数据的分隔化，即程序的各个部分除了必要的信息交流外彼此独立。这种结构化方式可使程序层次清晰，便于使用、维护以及调试。C＋＋语言是以函数形式提供给用户的，这些函数可方便地调用，并具有多种循环、条件语句控制程序流向，从而使程序完全结构化。

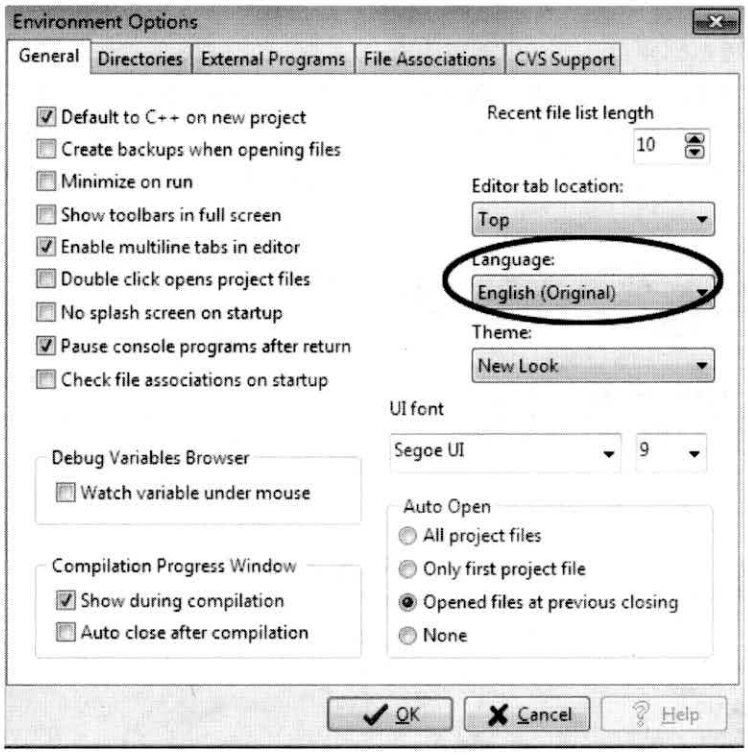


图 1－4

(2)此时屏幕右下侧出现一片白色区域,称为“源程序编辑区域”;可以在此输入程序。(如图 1－5 所示)



图 1－5

注意:

必须在英文输入环境下编辑程序(如果你当前能在程序编辑区输入中文,说明你是在中

文输入环境下。为了输入程序,你必须切换到英文输入环境下)。

3. 保存源程序到硬盘

一个好的习惯是创建了一个新程序后,在还未输入代码之前先将该程序保存到硬盘某个目录下,然后在程序的编辑过程中经常性地保存程序,以防止机器突然断电或者死机。要保存程序,只需从主菜单选择"文件"—〉"保存"就可以将文件保存到指定的硬盘目录(如图1—6所示)。



图1—6

此时会弹出一个对话框。在此你需要指定文件要存放的磁盘目录(例如:D:\temp),自定文件名称(例如:ex1)以及保存类型。在点击右下角的保存按钮后,在D盘上的temp目录下将会出现一个名为ex1.cpp的源文件。

提示:在输入程序的过程中记得要随时对程序进行保存(使用菜单"文件"—〉"保存",或者用组合键Ctrl+S),此时会将程序重新保存到之前指定的目录下,如D:\temp。如果想将程序保存到其他的硬盘路径下,可以选择"文件"—〉"另存为…",你可以重新指定程序的名称和保存的目录。

4. 编译、运行

编译:从主菜单选"运行"—〉"编译运行"或快捷键"F9"(如图1—7)。如果程序中存在词法、语法等错误,则编译过程失败。

编译器会在屏幕右下角的"Compile Log"标签页中显示错误信息,如图1—8所示,并且将源程序相应的错误行标成红色底色。

编译器标签页中显示的错误信息是寻找错误原因的重要信息来源,每一位同学都要学会看这些错误信息,并且每一次你碰到错误并且最终解决了错误时,要记录错误信息以及相应的解决方法。这样以后看到类似的错误提示信息,能熟练反映出是源程序哪里有问题,从

而提高程序调试效率。在排除了程序中存在的词法、语法等错误后，编译成功。此时在源文件所在目录下将会出现一个同名的.exe可执行文件(如ex1.exe)。



图1-7

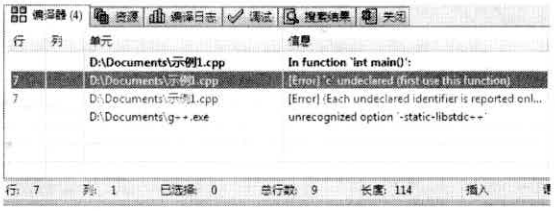


图1-8

5. 调试程序

通过预处理、编译和链接的程序仅仅是该程序中没有词法和语法等错误，而无法发现程序深层次的问题(譬如算法不对导致结果不正确)。当程序运行出错时，需要找出错误原因。仔细读程序来寻找错误固然是一种方法，但是有时光靠读程序已经解决不了问题，此时需要借助于程序调试(Debug)手段。这是一种有效的排错手段，每一位同学都需要掌握。

(1)设置程序断点

调试的基本思想是让程序运行到你认为可能有错误的代码前，然后停下来，在人的控制下逐条语句地运行，通过在运行过程中查看相关变量的值，来判断错误产生的原因。如果想让程序运行到某一行前能暂停下来，就需要将该行设成断点。具体方法是在代码所在行行首单击，该行将被加亮。默认的加亮颜色是红色。如图1－9所示，将c＝a＋b语句设成断点，则程序运行完cin语句后，将会暂停。需要说明的是，你可以在程序中根据需要设置多个断点。

如果想取消不让某行代码成为断点，则在代码行首再次点击即可。

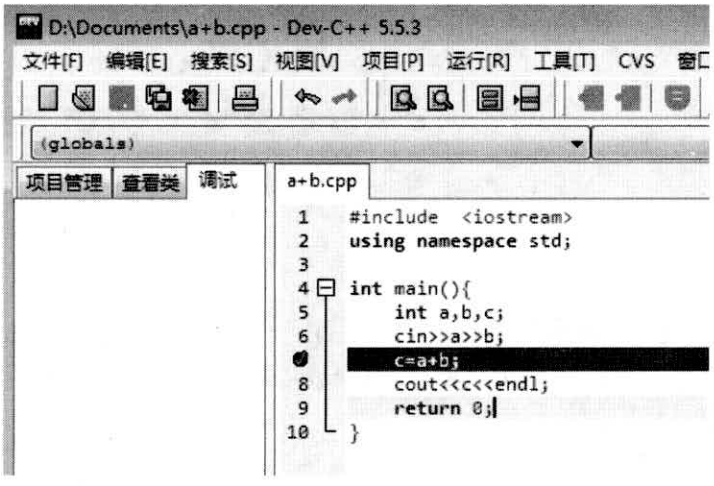


图1－9

(2)运行程序

设置断点后,此时程序运行进入 Debug 状态。要想运行程序,就不能使用主菜单"运行[R]"—〉"运行[R]",而是需要用主菜单"运行[R]"—〉"调试[D]"(或者按快捷键 F5),如图 1—10 所示。



图 1—10

程序将运行到第一个断点处,此时断点处加亮色由红色变成蓝色,表示接下去将运行蓝色底色的代码,如图 1—11 所示。



图 1—11

注意:有时你会发现即使设置了断点,点击了主菜单"运行[R]"—〉"调试[D]",程序还是不在断点处停留。解决方法:取消断点,重新编译程序,然后再设置断点,点击主菜单"运行[R]"—〉"调试[D]"即可。

(3)设置 watch 窗口

在调试程序时,可能要看程序运行过程中变量的值,以检测程序对变量的处理是否正

确，可以在调试时通过调试菜单下的添加变量(Add Watch)窗口来增加变量 watch，新增的变量将会显示在最左边 Explore 的 Debug 页中，如图 1－12 所示。如果左边 Explore 中的当前页不是 Debug 页，则可以点击 Debug 标签使之成为当前页。

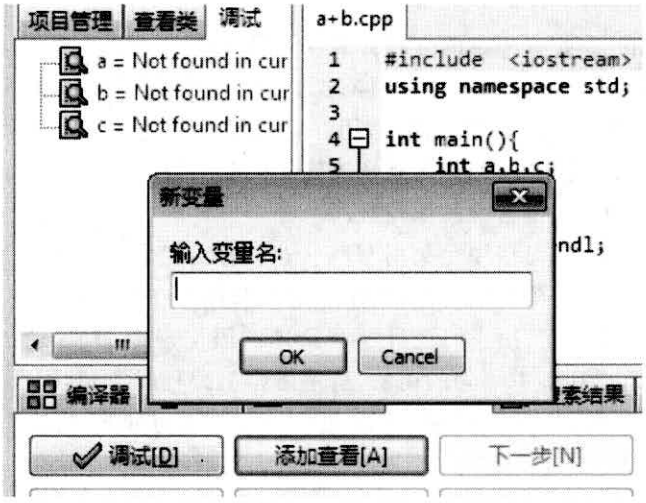


图 1－12

第二节　C++语言程序结构

无论做任何事情,都要有一定的方式方法与处理步骤,所谓"无规矩无以成方圆"。计算机程序设计比日常生活中的事务处理更具有严谨性、规范性和可行性。为了使计算机有效地解决实际问题,必须将处理步骤编排好,用计算机能理解的计算机语言编写成"序列",让计算机自动识别并执行这个"序列",达到解决实际问题的目的。将处理问题的步骤编排好,用计算机语言组成序列,就是常说的编写程序。在C++语言中,执行每条语句都是由计算机完成相应的具体操作,编写程序是利用C++语句的功能来实现预定的处理要求。"千里之行,始于足下",我们从简单程序学起,逐步了解和掌握怎样编写程序。

在学习C++语言之前,让我们绕过那些繁琐的语法规则细节,通过一些简单的例题,来熟悉程序的基本组成和基本语句的用法,选手刚接触编程时,多动手模仿是一条捷径。

例1.2　如图1-13,在梯形中阴影部分面积是150平方厘米,求梯形面积。

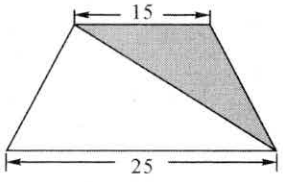


图1-13

【分析】　已知梯形上、下底长为15和25。令梯形的高为h,则由已知三角形面积为150平方厘米,有150=(15*h)/2,得h为20,然后根据梯形面积公式算出梯形面积。

程序如下:

```
#include<cstdio>        //printf 和 scanf 调用 cstdio 库,在 C 语言中可调用 stdio.h 库
#include<cstdlib>       // 使用 system()调用 cstdlib 库
using namespace std;    //在 C 语言中要省略,例如在 VC++和 TC++中
int main()              //有的 C 语言可用 void main(),例如 TC++和 VC++
{                       //整个程序开始
  float s,h,up,down;    // float 定义 s,h,up,down 为单精度实型变量
  up=15;                //输入上底,或使用 scanf("%f",&up);
  down=25;              //输入下底,或使用 scanf("%f",&down);
  h=2*150/up;           //根据上底求出梯形的高
  s=(up+down)*h/2;      //求出梯形的面积
  printf("s=%0.2f\n",s);
                        // \n 是换行控制符,0.2f 按实际位数输出,保留 2 位小数
  system("pause");//暂停作用,TC++和 VC++中使用 system()调用 stdlib.h 库
  return 0;             //结束程序,在 TC++和 VC++中也要保留
}
```

运行结果:

```
s=400.00
```

【说明】

所谓namespace,是指标识符的各种可见范围。C++标准程序库中的所有标识符都被定义于一个名为std的namespace中。

当使用<iostream.h>时,相当于在C中调用库函数,使用的是全局命名空间,也就是

早期的C++实现；当使用<iostream>时，该头文件没有定义全局命名空间，必须使用namespace std；这样才能正确使用cout、cin和endl。

例1.3　已知一位小朋友的电影票价是10元，计算x位小朋友的总票价是多少？

【分析】　假设总票价用y来表示，则这个问题可以用以下几个步骤来实现：

①输入小朋友的数目x；

②用公式y=10*x计算总票价；

③输出总票价y的值。

程序如下：

```
#include<cstdio>                    //使用printf和scanf,须调用cstdio库
#include<iostream>                  //使用cin,cout,须调用iostream库
using namespace std;
int main()
{
    int x,y;                        //定义整型变量
    cin>>x;                         //或使用scanf("%d",&x);
    y=10*x;                         //计算总票价
    cout<<x<<"  "<<y<<endl;
                    //或使用printf("%d  %d\n",x,y);注意%d %d之间空格
    return 0;                       //结束程序
}
```

【说明】

1. int定义x、y为整型变量，有关整型变量将在第二章详细介绍。

2. cin>>x的作用是输入一个数，赋予变量x。

3. cout<<x<<" "<<y<<endl的作用是输出x和y的值，x和y的值中间有空格，endl是换行的意思。

通过以上例1.2和例1.3两个例子，可以总结出C++语言程序的结构如下：

(1)C++语言变量在使用之前必须先定义其数据类型，未经定义的变量不能使用。定义变量类型的语句必须放在可执行语句前面。

(2)程序由一个或多个函数组成，一个程序中必须有且只有一个主函数，主函数的名字为main。不论main()函数在程序中什么位置，程序都是从main()函数开始执行，main()函数执行完毕，程序也就结束了。

(3)在程序中可以调用系统提供的库函数。在调用库函数之前，必须将相应头文件包含在程序中。例如，例1.2和例1.3中分别使用了输出函数cout<<和输入函数cin>>，需要#include<iostream>，将头文件iostream包含在程序中，编译预处理命令通常放在源程序或源文件的最前面。

(4)程序中可以有注释行。注释是为了使程序更易于理解和提示。在程序编译时，注释部分自动忽略。“//”表示行注释，在“//”之后的一行字符都是注释内容。C++语言还有一种注释方式为“/*”和“*/”，一对“/*”和“*/”中间的内容都是注释内容。

(5)程序的语句以分号结束。分号是C++语言不可缺少的组成部分。从上述两个实例中可以看出，每一条语句都以分号结束。但预处理命令、函数头和花括号“}”之后不加分号。

(6)程序的书写要注意适当的缩进,一般采用"逐层缩进"形式,以便使程序更加清晰易读。

把处理问题的步骤编成能从上到下顺序执行的程序,是简单程序的基本特征。再来分析下面一道例题的程序结构,同时继续学习基本语句。

例 1.4 有一个牧场,牧场上的牧草每天都在匀速生长,这片牧场可供 15 头牛吃 20 天,或可供 20 头牛吃 10 天,那么,这片牧场每天新生的草量可供几头牛吃 1 天?

【分析】 解决这类问题的关键是利用牛吃的草量,最终求出这片牧场每天新生长的草量,我们设 1 单位的草量为 1 头牛 1 天所需的草量,于是 15 头牛 20 天所食的草量为 300 单位(包括这 20 天内的新生草量),20 头牛 10 天所食的草量为 200 单位(包括这 10 天内的新生草量),两者的差值即为 10 天内的新生草量。程序如下:

```
#include<iostream>                    //使用 cin,cout,须调用 iostream 库
using namespace std;
int main()
{
    int s1,s2,s3;                     //变量定义
    s1=15*20;                         //15 头牛 20 天所食的草量
    s2=20*10;                         //20 头牛 10 天所食的草量
    s3=(s1-s2)/(20-10);               //每天新生的草量单位数
    cout<<" s="<<s3<<endl;            //1 单位为 1 头牛 1 天的食量
    return 0;                         //结束程序,在 Dev C++中可省略
}
```

运行结果:

```
s=10
```

例 1.5 给定一个字符,用它构造一个底边长 5 个字符,高 3 个字符的等腰字符三角形。

```
  #
 ###
#####
```

程序如下:

```
#include<iostream>                                  //使用 cin,cout,须调用 iostream 库
using namespace std;
int main()
{
    char a;                                        //定义字符变量,
    cin>>a;                                        //输入给定一个字符
    cout<<"  "<<a<<endl;                          //输出 1 个字符,先输出 2 个空格
    cout<<" "<<a<<a<<a<<endl;                      //输出 3 个字符,先输出 1 个空格
    cout<<a<<a<<a<<a<<a<<endl;                     //输出 5 个字符
    return 0;                                      //结束程序
}
```

【上机练习】

1. Hello,World!【1.1 编程基础之输入输出 01】

编写一个能够输出"Hello,World!"的程序，这个程序常常作为一个初学者接触一门新的编程语言所写的第一个程序，也经常用来测试开发、编译环境是否能够正常工作。

2. 输出第二个整数【1.1 编程基础之输入输出 02】

输入三个整数，整数之间由一个空格分隔，整数是 32 位有符号整数。把第二个输入的整数输出。

3. 对齐输出【1.1 编程基础之输入输出 03】

读入三个整数，按每个整数占 8 个字符的宽度，右对齐输出它们，按照格式要求依次输出三个整数，之间以一个空格分开。

4. 字符三角形【1.1 编程基础之输入输出 08】

给定一个字符，用它构造一个底边长 5 个字符，高 3 个字符的等腰字符三角形。

输入：

输入只有一行，包含一个字符。

输出：

该字符构成的等腰三角形，底边长 5 个字符，高 3 个字符。

样例输入：

```
  *
```

样例输出：

```
    *
  * * *
* * * * *
```

5. 地球人口承载力估计【小学奥数 7653】

假设地球上的新生资源按恒定速度增长。照此测算，地球上现有资源加上新生资源可供 x 亿人生活 a 年，或供 y 亿人生活 b 年。

为了能够实现可持续发展，避免资源枯竭，地球最多能够养活多少亿人？

输入：

一行，包括四个正整数 x,a,y,b，两个整数之间用单个空格隔开。x＞y,a＜b,ax＜by，各整数均不大于 10000

输出：

一个实数 z，表示地球最多养活 z 亿人，舍入到小数点后两位。

说明：

为了适应信息学联赛考纲要求，将陆续更新《信息学奥赛一本通 C++版》的【上机练习】，后继版本进一步从【NOI 题库】中精选更新，第一部分【上机练习】含样例题目的 pdf 文件在配套光盘中，学生程序请在相关网站提交评测。

第二章　顺序结构程序设计

第一章的简单程序已体现出处理问题的步骤的顺序关系，每条语句按自上而下的顺序依次执行一次，这种自上而下依次执行的程序称为顺序结构程序。

在一个程序中，所有的操作都由执行部分来完成，而执行部分又都是由一条条语句组成的。因此，先要学习C++语言的基本语句，并且在学习过程中逐步学会程序设计的基本方法。我们还是绕过那些繁琐的语法规则细节，先看一些例子，然后给出语法以供选手们参考。

第一节　赋值语句

在C++语言中，“=”作为赋值运算符，而不表示“等于”判断。赋值语句是由赋值表达式再加上分号构成的表达式语句，它是程序中使用最多的语句之一。

其一般形式为：

变量=表达式；

在赋值语句的使用中，需要注意以下几点：

(1)由于赋值运算符“=”右边的表达式也可以是赋值表达式，因此，下述形式

变量=(变量=表达式)；

是成立的，从而形成嵌套的情形。其展开之后的一般形式为：

变量=变量=…=表达式；

例如，“a=b=c=d=e=5;”，它实际上等价于：e=5;d=e;c=d;b=c;a=b;

(2)在进行赋值运算时，如果赋值运算符两边的数据类型不同，系统将会自动进行类型转换，即将赋值运算符右边的数据类型转换成左边的变量类型。当左边是整型而右边是实型时，将去掉小数部分并截取该整型对应的有效位数。

例2.1　输入两个正整数a和b，试交换a、b的值(使a的值等于b，b的值等于a)。

【分析】　交换两个变量的值方法很多，一般我们采用引入第三个变量的算法，两个变量交换，可以想像成一瓶酱油和一瓶醋进行交换，这时容易想到拿一个空瓶子过来：

① 将酱油倒到空瓶中；② 将醋倒到酱油瓶中；③ 将原空瓶中的酱油倒到醋瓶中。

程序如下：

```
#include<iostream>                    //使用 cin,cout,须调用 iostream 库
using namespace std;
int main()
{   int a,b,c;                        //定义三个变量
```

```
    cout<<" Input a,b=";          //输入提示 Input a,b=
    cin>>a>>b;                    //输入 a、b 的值
    c=a;  a=b;  b=c;              //交换 a、b 的值
    cout<<" a="<<a<<"  b="<<b<<endl;     //输出结果
}
```

例 2.2　圆柱体的表面积

输入底面半径 r 和高 h，输出圆柱体的表面积，保留 3 位小数，格式见样例。

样例输入：3.5 9　　样例输出：274.889

【分析】　圆柱体的表面积由 3 部分组成：上底面积、下底面积和侧面积。由于上下底面积相等，完整的公式可以写成：表面积＝底面积＊2＋侧面积。根据平面几何知识，底面积＝πR^2，侧面积＝$2\pi rh$。参考程序：

```
#include<cstdio>                  //使用 printf 和 scanf，须调用 cstdio 库
using namespace std;
int main()
{ const double pi=3.1415926;      //定义 pi 为双精度实型变量
  double r,h,s1,s2,s;             //定义双精度实型，float 为单精度实型
  scanf("%lf%lf",&r,&h);          //r 和 h 前的 & 符号不能漏掉，double 型用%lf
  s1=pi*r*r;                      //计算底面积
  s2=2*pi*r*h;                    //计算侧面积
  s=2*s1+s2;                      //计算总的表面积
  printf(" Area=%0.3lf\n",s);     //输出结果保留 3 位小数，注意用 lf 格式
  return 0;
}
```

例 2.3　数学中经典的“鸡兔同笼”问题，已知头共 30 个，脚共 90 只，问笼中的鸡和兔各有多少只？

【分析】　设鸡为 j 只，兔为 t 只，头为 h，脚为 f，那么有：

$$j+t=30 \quad ①$$

$$2*j+4*t=90 \quad ②$$

假设笼中 30 个头全都是兔，那么都按每头 4 只脚计算，总脚数为(4＊h)，与实际脚数(f)之差为(4＊h－f)，如果这个差＝0，则笼中全是兔(即鸡为 0 只)；如果这个差值 ＞0，说明多计算了脚数，凡是鸡都多计算了两只脚，用它除以 2 就能得到鸡的只数，算法为：

```
①j=(4*h-f)/2                     //先用脚数差值除以 2 算出鸡的只数
②t=h-j                           //再用总头数减鸡数算出兔的只数
```

注意这两步运算的先后顺序。程序如下：

```
#include<cstdio>                  //使用 getchar()语句，须调用 cstdio 库
#include<iostream>                //使用 cin,cout,须调用 iostream 库
using namespace std;
int main()
{   int h,f,j,t;                  //定个变量
    h=30;f=90;                    //赋初始值
```

```
    j=(4*h-f)/2;                    //计算鸡的只数
    t=h-j;                          //计算兔的只数
    cout<<"j="<<j<<"  t="<<t<<endl;     //输出结果
    getchar();              //和 system("pause")作用相似,用于显示暂停
}
```

程序中语句的自上而下的顺序很关键,这类程序严格自上而下每条语句都被执行一次就称为顺序程序。

第二节　运算符和表达式

C++语言中运算符和表达式数量之多,在高级语言中是少见的。正是丰富的运算符和表达式使C++语言功能十分完善。这也是C++语言的主要特点之一。

C++语言的运算符不仅具有不同的优先级,而且还有一个特点,就是它的结合性。在表达式中,各运算符参与运算的先后顺序不仅要遵守运算符优先级别的规定,还要受运算符结合性的制约,以便确定是自左向右进行运算还是自右向左进行运算。这种结合性是其他高级语言的运算符所没有的,因此也增加了C++语言的复杂性。

C++语言的运算符可分为以下几类:

1. 算术运算符

用于各类数值运算。包括加(+)、减(-)、乘(*)、除(/)、求余(或称模运算,%)、自增(++)、自减(--)共七种。

2. 关系运算符

用于比较运算。包括大于(>)、小于(<)、等于(==)、大于等于(>=)、小于等于(<=)和不等于(!=)六种。

3. 逻辑运算符

用于逻辑运算。包括与(&&)、或(||)、非(!)三种。

4. 位操作运算符

参与运算的量,按二进制位进行运算。包括位与(&)、位或(|)、位非(~)、位异或(^)、左移(<<)、右移(>>)六种。

5. 赋值运算符

用于赋值运算,分为简单赋值(=)、复合算术赋值(+=,-=,*=,/=,%=)和复合位运算赋值(&=,|=,^=,>>=,<<=)三类共十一种。

6. 条件运算符

这是一个三目运算符,用于条件求值(?:)。

7. 逗号运算符

用于把若干表达式组合成一个表达式(,)。

8. 指针运算符

用于取内容(*)和取地址(&)两种运算。

9. 求字节数运算符

用于计算数据类型所占的字节数(sizeof)。

10. 特殊运算符

有括号()、下标[]、成员(→,.)等几种。

一、算术运算符

1. 模运算符

求余的运算符“%”也称为模运算符，是双目运算符，两个操作数都是整型数。a%b 的值就是 a 除以 b 的余数，5%2 余数为 1。其操作对象只能是整型数，而其他四种运算符对 int、float、double、char 都适用。

2. 除法运算符

C++语言的除法运算符有一些特殊之处，即如果 a、b 是两个整数类型的变量或常量，那么 a/b 的值是 a 除以 b 的商。例如，5/2 的值是 2，而不是 2.5，而 5.0/2 或 5/2.0 的值是 2.5。

3. 自增自减运算符

自增、自减运算符用来对一个操作数进行加 1 或减 1 运算，其结果仍然赋予该操作数，而且参加运算的操作数必须是变量，而不能是常量或表达式。

(1)自增运算符。例如，x++表示在使用 x 之后，使 x 的值加 1，即 x=x+1；++x 表示使用 x 之前，先使 x 的值加 1，即 x=x+1。

(2)自减运算符。例如，x--表示在使用 x 之后，使 x 的值减 1，即 x=x-1；--x 表示使用 x 之前，先使 x 的值减 1，即 x=x-1。

4. 复合算术赋值

例如，a+=1，相当于 a=a+1；a+=b，相当于 a=a+b。

例 2.4　变量自加运算

```
#include<iostream>
using namespace std;
int main()
{
    int x=7,y=8;                    //相当于 int x,y;  x=7;  y=8;
    int z1=y-(x++);                 //计算 z1=1,计算后 x=8
    int z2=y-(++x);                 //计算前 x 的值自加 1,x 的值为 9,再与 y 求差
    cout<<" z1="<<z1<<endl<<" z2="<<z2;     //分别输出 z1 和 z2 的值
}
```

运行结果：

```
z1=1
z2=-1
```

二、关系运算符

关系运算符用于数值的大小比较。包括大于(>)、小于(<)、等于(==)、大于等于(>=)、小于等于(<=)和不等于(!=)六种，它们都是双目运算符。

关系运算符运算的结果是整型,值只有两种:0 或 1,0 代表关系不成立,1 代表关系成立。

请看下面的例子:

```
int main()
{    int n1=4,n2=5,n3;
     n3=n1>n2;              //n3 的值为 0
     n3=n1<n2;              //n3 的值变为 1
     n3=n1==4;              //n3 的值变为 1
     n3=n1!=4;              //n3 的值变为 0
     n3=n1==1+3;            //n3 的值变为 1
}
```

三、逻辑运算符

C++语言中提供了三种逻辑运算符:与运算(&&)、或运算(||)、非运算(!)。与运算符(&&)和或运算符(||)均为双目运算符,具有左结合性。非运算符(!)为单目运算符,具有右结合性。逻辑运算符和其他运算符优先级的关系可表示如下:

按照运算符的优先顺序可以得出:

a>b && c>d 等价于(a>b) && (c>d)

!b==c||d<a 等价于((!b)==c)||(d<a)

a+b>c && x+y<b 等价于((a+b)>c) && ((x+y)<b)

逻辑运算的值也为"真"和"假"两种,用"1"和"0 "来表示。其求值规则如下:

(1)与运算 && 参与运算的两个量都为真时,结果才为真,否则为假。例如:5>0 && 4>2,由于 5>0 为真,4>2 也为真,相与的结果也为真。

(2)或运算||参与运算的两个量只要有一个为真,结果就为真;两个量都为假时,结果为假。例如:5>0||5>8,由于 5>0 为真,相或的结果也就为真。

(3)非运算!参与运算量为真时,结果为假;参与运算量为假时,结果为真。例如:!(5>0)的结果为假。

虽然 C++编译在给出逻辑运算值时,以"1"代表"真","0 "代表"假"。但反过来在判断一个量是为"真"还是为"假"时,以"0"代表"假",以非"0"的数值作为"真"。例如:由于 5 和 3 均为非"0",因此 5&&3 的值为"真",即为 1。又如:5||0 的值为"真",即为 1。

四、位运算符

C++语言提供了 6 种位运算符,如下表所示。

运算符	含义	说明	例子
&	按位与	把参与运算的两个数对应的二进制位相与，只有对应的二进制均为1时，结果的对应位才为1，否则为0。	9&5中9可以写成(00001001)，5可以写成(00000101)，那么9&5的运算结果为0000 0001，输出结果是1。
|	按位或	把参与运算的两个数对应的二进制位相或，也就是只要对应的两个二进制位有一个为1时，其结果就为1。	9|5相当于00001001|00000101，运算结果是00001101，输出结果是13。
^	按位异或	把参与运算的两个数对应的二进制位相异或，当对应的二进制位上的数据字不相同时，结果对应为1时，否则为0。	1^1=0，1^0=1，0^0=0，0^1=1 9^5相当于00001001^00000101，运算结果是00001100，输出结果是12。
~	取反	把运算数的各个二进制位按位求反。	~9相当于~(0000 1001)，运算结果为1111 0110。
<<	左移	把"<<"左边的运算数的各二进制位向左移若干位，"<<"右边的数是指定移动的位数，高位丢弃，低位补0。	a<<4指把a的各二进位向左移动4位，如a=00000011(十进制为3)，左移4位后为00110000(十进制为48)。
>>	右移	把">>"左边的运算数的各二进制位全部右移若干位，">>"右边的数是指定移动的位数。	设a=15，a>>2表示把00001111右移为0000 0011(十进制为3)。

需要说明的是，对于有符号数，在右移时，符号位将随同移动。当操作数为正数时，最高位为0，而为负数时，最高位为1。最高位是补0或补1取决于编译系统的规定。

五、运算的简写

在C++语言中，有一些运算可以简写，如下表所示。

简写	含义	简写	含义
a+=b	a=a+b	a&=b	a=a&b
a-=b	a=a-b	a|=b	a=a|b
a*=b	a=a*b	a^=b	a=a^b
a/=b	a=a/b	a<<=b	a=a<<b
a%=b	a=a%b	a>>=b	a=a>>b

算术运算符、关系运算符、逻辑运算符和赋值运算符的优先级如下：

赋值运算符　　逻辑运算符　　关系运算符　　算术运算符

低　　高

关系运算符的结合性为：自左至右。

根据以上优先级和结合性，计算出以下表达式的结果(假设a=3，b=2，c=1)

a>b	表达式为真,所以表达式的值为1
(a>b)==c	表达式为真,所以表达式的值为1
b+c<a	表达式为假,所以表达式的值为0
d=a>b	a>b为真,所以d的值为1
f=a>b>c	a>b为真,结果为1,1>c为假,所以f的值为0

六、常用库函数

函数名	格　式	功能说明	例　子
绝对值函数	abs(x)	求一个数x的绝对值	abs(−5)=5
自然数指数函数	exp(x)	求实数x的自然指数e^x	exp(1)=2.718282
向下取整	floor(x)	求不大于实数x的最大整数	floor(3.14)=3
向上取整	ceil(x)	求不小于实数x的最小整数	ceil(3.14)=4
自然对数函数	log(x)	求实数x的自然数对数	log(1)=0
指数函数	pow(x,y)	计算x^y,结果为双精度实数	pow(2,3)=8
随机函数	rand()	产生0到RAND−MAX之间的随机整数	
平方根值函数	sqrt(x)	求实数x的平方根	sqrt(25)=5

【上机练习】

1. A+B问题【1.3 编程基础之算术表达式与顺序执行01】

大部分的在线题库,都会将A+B问题作为第一题,以帮助新手熟悉平台的使用方法。

A+B问题的题目描述如下:给定两个整数A和B,输出A+B的值。保证A、B及结果均在整型范围内。现在请你解决这一问题。

2. 计算(a+b)*c的值【1.3 编程基础之算术表达式与顺序执行02】

给定3个整数a、b、c,计算表达式(a+b)*c的值。

3. 计算(a+b)/c的值【1.3 编程基础之算术表达式与顺序执行03】

给定3个整数a、b、c,计算表达式(a+b)/c的值,/是整除运算。

4. 带余除法【1.3 编程基础之算术表达式与顺序执行04】

给定被除数和除数,求整数商及余数。此题中请使用默认的整除和取余运算,无需对结果进行任何特殊处理。

5. 计算分数的浮点数值【1.3 编程基础之算术表达式与顺序执行05】

两个整数a和b分别作为分子和分母,既分数a/b,求它的浮点数值(双精度浮点数,保留小数点后9位)。

第三节　常量和变量

一、常量

常量是指在程序中使用的一些具体的数、字符。在程序运行过程中，其值不能被更改。如123、145.88、'm'、TRUE等。

1. 整型常量：如3、−5、0等。

整型常量是表示整数的常量。有三种表示形式：

(1)十进制形式。如99、−1。

(2)八进制形式。以数字0打头，由0～7构成，如012，表示八进制整数12，即$(12)_8$。

(3)十六进制形式。以0x打头，如0x12A，表示十六进制整数12A，即$(12A)_{16}$。

2. 实型常量：如3.1、−6.1E+2(科学记数法)。

3. 字符常量：是用单引号括起来的字符，如'k'、'5'、'%'。

注意：'a'与"a"表示的含义是不同的，'a'表示一个字符常量，"a"表示一个字符串。

二、常量的定义

一个常量可以直接调用(如124，'A')，也可以给常量取个名字用一个标识符代表它，这就是符号常量。其语法格式为：

```
const  符号常量=常量字串;
```

例如：const double PI=3.1415926;

例2.5　输入半径r，求圆的周长及面积。

```
#include<cstdio>          //调用iostream库，否则使用printf和scanf语句编译出错
using namespace std;
const double PI=3.1415926;     //PI是符号常量，代表3.1415926
int main()
{
  float r,c,s;                 //定义实型变量
  printf("r=");                //显示提示符r=
  scanf("%f",&r);              //输入r的值，& 符号不能漏掉
  c=2*PI*r;                    //计算圆的周长
  s=PI*r*r;                    //计算圆的面积
  printf("c=%.2f s=%.2f\n",c,s);   //显示计算结果，结果保留2位小数
}
```

程序中定义的PI代表常量3.1415926，在编译源程序时，遇到PI就用常量3.1415926代替，PI可以和常量一样进行运算。C++语言规定，每个符号常量的定义占据一个书写行，而且符号常量不能被再赋值。如果在例2.5中使用以下赋值语句是错误的。

```
PI=3.1415926;
```

习惯上,符号常量名用大写,而变量名用小写,以便于区别。

使用符号常量的好处:

(1)增加了程序的可读性。如例 2.5 程序中,见到 PI 就可知道它代表圆周率,定义符号常量名时应该尽量使用见名知义的常量名。

(2)增加了程序的易改性。如例 2.5 程序中,只需改动一处,程序中的所有 PI 都会自动全部代换,做到"一改全改"。

三、变量定义

变量代表了一个存储单元,其中的值是可以改变的,因此称为变量。如游戏中玩家命的条数最初为 3,当你死了一次,命减少一条,这里命的条数就是一个变量(或者说命的条数存储在一个存储单元中)。

一个程序中可能要使用到若干个变量,为了区别不同的变量,必须给每个变量(存储单元)取一个名(称为变量名),该变量(存贮单元)存储的值称为变量的值,变量中能够存储值的类型为变量的类型。例如游戏中用于存储"命"的变量,在游戏程序中的存储命的变量名可取为 life,它的类型为整型,游戏初始时这个变量的值为 3。

1. 变量名

用一个合法的标识符代表一个变量。如 n,m,rot,total 等都是合法变量名。在程序中用到的变量要"先定义后使用",变量名应遵循自定义标识符的命名规则,并建议使用"见名知义"的原则,即用一些有意义的单词作为变量名。在 C++语言,变量名大小写有区别。

定义变量的语法格式为:

数据类型　变量表

例如:

```
int  i=5,j,k(5);          //定义 i,j,k 为整型变量,i,k 赋初值为 5,j 的初值未知。
char a,b,c;               //定义 a,b,c 为字符变量
float x,y,z;              //定义 x,y,z 为实型变量
```

C++语言允许在定义变量的同时为变量赋初值,并且可以在任意位置定义变量,在变量第一次使用时定义变量可以提高程序的可读性。读者不需要返回到代码的开始位置去寻找某一特殊变量的定义,而且,在此处定义变量,更容易给它赋以有意义的初始值。

用来标识变量名、符号常量名、函数名、数组名、类型名、文件名的有效字符序列称为标识符。C++语言规定,标识符只能由字母(包含下划线"_")开头,后面的字符可以是字母或数字。对于标识符的长度,不同的 C++语言编译器有不同的规定,考虑到系统的可移植性,建议变量名的长度不要超过 8 个字符。例如:month、_age、s2 为合法的标识符;m.k.jack、a<=b、9y 为不合法的标识符。

2. 变量的类型

常量是有类型的数据,变量在某一固定时刻用来存储一个常量,因此也应有相应的类型。如整型变量用来存储整数,实型变量用来存储实数。变量的类型,可以是标准数据类型 int、short、long、float、double 和 char 等,也可以是用户自定义的各种类型。

变量一经定义,系统就在计算机内存中为其分配一个存储空间。在程序中使用到变量时,就在相应的内存中存入数据或取出数据,这种操作称为变量的访问。

【上机练习】

1. 甲流疫情死亡率【1.3 编程基础之算术表达式与顺序执行 06】

甲流并不可怕，在中国，它的死亡率并不是很高。请根据截止 2009 年 12 月 22 日各省报告的甲流确诊数和死亡数，计算甲流在各省的死亡率。

2. 计算多项式的值【1.3 编程基础之算术表达式与顺序执行 07】

对于多项式 f(x) = ax^3+bx^2+cx+d 和给定的 a,b,c,d,x，计算 f(x)的值，保留到小数点后 7 位。

3. 温度表达转化【1.3 编程基础之算术表达式与顺序执行 08】

利用公式 C = 5*(F−32)/9(其中 C 表示摄氏温度，F 表示华氏温度)进行计算转化，输入华氏温度 f，输出摄氏温度 c，要求精确到小数点后 5 位。

4. 与圆相关的计算【1.3 编程基础之算术表达式与顺序执行 09】

给出圆的半径，求圆的直径、周长和面积。输入圆的半径实数 r，输出圆的直径、周长、面积，每个数保留小数点后 4 位。

5. 计算并联电阻的阻值【1.3 编程基础之算术表达式与顺序执行 10】

对于阻值为 r1 和 r2 的电阻，其并联电阻阻值公式计算如下：R = 1/(1/r1 + 1/r2)。输入两个电阻阻抗大小，浮点型。输出并联之后的阻抗大小，结果保留小数点后 2 位。

第四节　标准数据类型

C++语言提供了丰富的数据类型，本节介绍几种基本的数据类型：整型、实型、字符型。它们都是系统定义的简单数据类型，称为标准数据类型。

一、整型

在 C++语言中，整型类型标识符为 int。根据整型变量的取值范围又可将整型变量定义为以下 8 种整型类型：

数据类型	定义标识符	占字节数	数值范围	数值范围
短整型	short [int]	2(16 位)	−32768～32767	$-2^{15}\sim 2^{15}-1$
整型	[long] int	4(32 位)	−2147483648～2147483647	$-2^{31}\sim 2^{31}-1$
长整型	long [int]	4(32 位)	−2147483648～2147483647	$-2^{31}\sim 2^{31}-1$
超长整型	long long [int]	8(64 位)	−9223372036854775808～9223372036854775807	$-2^{63}\sim 2^{63}-1$
无符号整型	unsigned [int]	2(16 位)	0～65535	$0\sim 2^{16}-1$
无符号短整型	unsigned short [int]	2(16 位)	0～65535	$0\sim 2^{16}-1$
无符号长整型	unsigned long [int]	4(32 位)	0～4294967295	$0\sim 2^{32}-1$
无符号超长整型	unsigned long long	8(64 位)	0～18446744073709551615	$0\sim 2^{64}-1$

二、实型

一个实型数据用来存储实数,实型包括正实数、负实数和零。C++语言中表示实型常量的形式有两种。

① 十进制表示法

这是人们日常使用的带小数点的表示方法。

如 3,0.0,+5.61,-8.0,-6.050 等都是实型常量。

② 科学记数法

科学记数法是采用指数形式的表示方法,如 1.25×10^5 可表示成 1.25E+05。在科学记数法中,字母"E"表示 10 这个"底数",而 E 之前为一个十进制表示的小数,称为尾数,E 之后必须是一个整数,称为"指数"。如-1234.56E+26,+0.268E-50,1E5 是合法形式,而 E5,E,1.2E+0.5 都不是合法的实数。

C++语言支持三种实型,它们是 float(单精度实型)、double(双精度实型)、long double(长双精度实型)。每一种类型规定了相应的实数取值范围、所使用的存储空间(字节数)以及能达到的精度(有效位数)。float 在空间允许的情况下没有必要使用,都应使用 double 提高精度。

数据类型	定义标识符	数值范围	占字节数	有效位数
单精度实型	float	-3.4E-38~3.4E+38	4(32 位)	6~7 位
双精度实型	double	-1.7E+308~1.7E+308	8(64 位)	15~16 位
长双精度实型	long double	-3.4E+4932~1.1E+4932	16(128 位)	18~19 位
布尔变量	bool	真 true 或假 false 之一	1(8 位)	

三、字符型

字符常量有以下两种表示法:

1. 普通表示形式

字符常量是由单个字符组成,所有字符采用 ASCII 编码,ASCII 编码共有 128 个字符(如下表)。在程序中,通常用一对单引号将单个字符括起来表示一个字符常量。如:'a','A','0'等。如字符 A 的序号是 65,字符 a 的序号是 97,字符 0 的序号是 48。

序号	字符	序号	字符	序号	字符	序号	字符	序号	字符	序号	字符
32	空格	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v

39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	deL

2. 转义字符表示形式

转义字符有三种用法：表示控制字符、表示特殊字符、表示所有字符。常用的转义字符如下表所示。

转义字符	含　义
'\n'	换行
'\t'	水平制表
'\b'	退格
'\r'	回车(不换行)
'\0'	空字符
'\"'	单引号
'\"'	双引号
'\\'	一个反斜杠字符
'\ddd'	1～3 位八进制数所代表的字符
'\xhh'	1～2 位十六进制数所代表的字符

例 2.6　整型数据类型存储空间大小

分别定义 int,short 类型的变量各一个，并依次输出它们的存储空间大小(单位：字节)。

【参考程序】

```
#include<iostream>
using namespace std;
int main()
{
    int x;
    short y;                    //sizeof 返回一个对象或者类型所占的内存字节数
    cout<<sizeof(x)<<" "<<sizeof(y)<<endl;
    return 0;
}
```

例 2.7　大小字母的转换

```
#include<iostream>
using namespace std;
int main()
```

```
{
  char c1='a';
  char c2='A';
  cout<<c1<<"  "<<c2<<endl;
  c1=c1-32;              //小写字母转换大写字母
  c2=c2+32;              //大写字母转换小写字母
  cout<<c1<<"  "<<c2<<endl;
}
```

运行结果:

```
a  A
A  a
```

因为所有小写字母的 ASCII 值要比对应大写字母的 ASCII 值大 32,所以 c1 减去 32 后便得到原来字母的大写形式。反之,c2 加上 32 后便得到原来字母的小写形式。

四、数据类型转换

C++语言中,不同数据类型的运算对象进行混合运算,或者需要将一个表达式的结果转换成期望的类型时,就需要依据数据类型转换规则进行转换。

1. 混合运算时的类型转换规则

整型、实型、字符型数据间可以混合运算。在这种情况下,需要将不一致的数据类型转换成一致的数据类型,然后进行运算。为了保证运算精度,系统在运算时的转换规则是将存储长度较短的运算对象转成存储长度较长的类型,然后再进行处理。这种转换是系统自动进行的,具体见图 2-1 所示。

混合运算时的类型转换规则

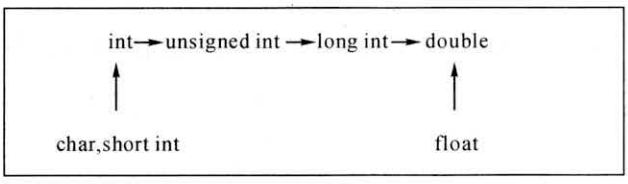


图 2-1

【说明】

(1)纵向箭头表示必定会进行的转换,如 float 型数据必先转换为 double 型数据,然后与其他操作数进行运算。与此类似,char 型或 short 型数据必先转换为 int 型数据,然后进行运算。

(2)横向箭头表示当运算对象为不同类型数据时的转换方向,如 int 型数据与 unsigned 型数据进行运算,int 型转换为 unsigned 型后方可进行运算。int 型与 double 型进行运算,int 型直接转换为 double 型后进行运算,不能理解为先转换为 unsigned int 型,然后转换为 long int 型,最后再转换为 double 型。

2. 赋值时的类型转换规则

当赋值运算符两侧的数据类型不同时,需进行类型转换,这种转换是系统自动进行的,转换规则如下:

(1)float、double 型赋值给 int 型:直接截断小数。

例如,"int i=f+0.6;",f 的值为 4.0,右边算术表达式运算后的结果为 4.6 的 double 型数据,根据上述转换原则,直接舍弃小数,所以 i 的值为 4。

(2)int、char 型赋值给 float、double 型:补足有效位以进行数据类型转换。

例如:"float f=4;",float 为 7 位有效数字,所以 f 的值为 4.0000000。

(3)char 型(1 字节)赋给 int 型(4 字节):数值赋给 int 型的低 8 位,其他位补 0。

(4)long int 型赋值给 int 型:long int 型截断低字节给 int 型。

(5)int 型赋值给 long int 型:赋给 long int 型的低 16 位,如果 int 型的最高位是 0,则 long int 的高 16 位全为 0;如果 int 型的最高位是 1,则 long int 型的高 16 位全为 1(称为"符号扩展")。

(6)unsigned int 型赋值给 int 型:直接传送数值。

(7)非 unsigned int 型赋值给位数相同的 unsigned int 型:直接传送数值。

3. 强制类型转换

在 C++语言中,还允许强制类型转换,即将某一数据的数据类型转换为指定的另一种数据类型,强制类型转换只是临时转换。强制转换运算符组成的运算表达式的一般形式为:(类型名)(表达式)

例如:已知有变量定义"int b=7;float a=2.5,c=4.7;",求下面算术表达式的值。

a+(int)(b/3*(int)(a+c)/2.0)%4

根据运算符的结合性规则,表达式要自左至右执行,b/3 为 2,2*(int)(a+c)为 14,14/2.0 为 7.0,强制类型转换后为 7,7%4 为 3,a 的值 2.5 与 3 相加,最终结果为 5.5。

【上机练习】

1. 整型数据类型存储空间大小【1.2 编程基础之变量定义、赋值及转换 01】

2. 浮点型数据类型存储空间大小【1.2 编程基础之变量定义、赋值及转换 02】

3. 其他数据类型存储空间大小【1.2 编程基础之变量定义、赋值及转换 03】

4. 浮点数向零舍入【1.2 编程基础之变量定义、赋值及转换 06】

5. 打印 ASCII 码【1.2 编程基础之变量定义、赋值及转换 07】

6. 打印字符【1.2 编程基础之变量定义、赋值及转换 08】

7. 整型与布尔型的转换【1.2 编程基础之变量定义、赋值及转换 09】

8. Hello,World! 的大小【1.2 编程基础之变量定义、赋值及转换 10】

第五节　数据输入输出

C++语言中没有提供专门的输入输出语句,所有的输入输出都是调用标准库函数中的输入输出函数来实现的。在使用时,应在源程序的开头使用如下语句:

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

C++语言标准函数库提供了许多标准输入、输出函数,本节将介绍6个最基本的输入、输出函数:字符输入getchar,字符输出putchar,格式化输入scanf,格式化输出prinf,流输入cin,流输出cout。

一、字符输入函数 getchar

getchar函数是接收从键盘输入的单个字符数据。它是一个无参函数,其语法格式为:

```
getchar();
```

【说明】

(1)通常把输入的字符赋予一个字符变量,构成赋值语句。例如:

```
char  ch;
ch=getchar();
```

(2) getchar函数只能接受单个字符,输入数字也按字符处理。

(3)输入多于一个字符时,只接收第一个字符。

(4) getchar函数等待用户输入,直到按回车键才结束,可用于暂停程序的运行,直到输入一个回车键。

(5)如果在程序中连续有两个以上getchar()函数,应该一次性输入所需字符,最后再按回车键,否则会把回车作为一个字符传给后面的getchar()函数。

例2.8　利用getchar函数接收键盘输入。

```
#include<cstdio>
#include<iostream>
using namespace std;
int main()
{
  char ch=getchar();     //读入字符
  cout<<" input="<<ch<<endl;
}
```

二、字符输出函数 putchar

putchar 函数是字符输出函数，功能是向标准输出设备（如显示器）输出单个字符数据，其语法格式为：

```
putchar(ch);          //其中，ch 为一个字符变量或常量。
```

例 2.9　利用 putchar 函数输出字符。

```
#include<cstdio>
#include<iostream>
using namespace std;
int main()
{
  char c='B';          //定义字符变量 c 并赋值'B'
  putchar(c);          //输出该字符
  putchar('\x42');     //用转义字符输出字母'B'
  putchar(0x42);       //用 16 进制 ASCII 码值输出字母'B'
  putchar(66);         //用 10 进制 ASCII 码值输出字母'B'
}
```

运行结果：BBBB

三、通过 cout 流输出数据

流插入运算符<<和 cout 结合在一起使用，可向显示器屏幕输出数据。

格式 1：

```
cout<<表达式;
```

功能：它把表达式的值输出到屏幕上，该表达式可以是各种基本类型的常量、变量或者由它们组成的表达式。输出时，程序根据表达式的类型和数值大小，采用不同的默认格式输出，大多数情况下可满足要求。若要输出多个数据，可以连续使用流插入运算符。

格式 2：

```
cout<<表达式 1<<表达式 2<<表达式 3…;
```

功能：将表达式的内容一项接一项的输出到屏幕上。

1. 输出字符串和输出变量的区别

每当我们输出字符串常量的时候，必须用双引号把字符串引起来，以便将它和变量名明显的区分开来。

例 2.10

下面两个语句是不同的：

```
cout << "Hello";      //打印字符串 Hello 到屏幕上
cout << Hello;        //把变量 Hello 存储的内容打印到屏幕上
```

2. 如何增强信息的可读性

为了增强输出信息的可读性,在输出多个数据时可以通过插入空格符、换行符或其他提示信息将数据进行组织,以获得更好的效果。

例 2.11

```
x=12;
cout << " tom is my friend,he is ";
cout << x;
cout <<" years old ";
```

输出结果为 tom is my friend,he is 12 years old

3. 换行符的使用

必须注意,除非我们明确指定,cout 并不会自动在其输出内容的末尾加换行符,因此下面的语句:

例 2.12

```
cout << " This is a sentence. " ;
cout << " This is another sentence. " ;
```

将会有如下内容输出到屏幕:

This is a sentence. This is another sentence.

虽然我们分别调用了两次 cout,两个句子还是被输出在同一行。所以,为了在输出中换行,我们必须插入一个换行符来明确表达这一要求,在 C++中换行符可以写作\n。

```
cout << " First sentence.\n ";
cout << " Second sentence.\nThird sentence. ";
```

将会产生如下输出:

First sentence.

Second sentence.

Third sentence.

另外,你也可以用操作符 endl 来换行,例如:

```
cout << " First sentence. " << endl;
cout << " Second sentence. " << endl;
```

将会输出:

First sentence.

Second sentence.

例 2.13

在屏幕上输出 2 3

4

```
cout<<2<<"  "<<3<<endl;
cout<<4;
```

或 cout<<" 2 3\n4 ";

除了以上两种写法外，还可以有其他的写法，请试试看。你可以使用\n 或 endl 来指定 cout 输出换行，注意两者的不同用法。

四、通过 cin 流读入数据

流读取运算符>>和 cin 结合在一起使用，可从键盘输入数据。

格式 1：

```
cin>>变量;
```

功能：是从键盘读取一个数据并将其赋给“变量”。

说明：在使用 cin 输入的时候必须考虑后面的变量类型。如果你要求输入一个整数，在>>后面必须跟一个整型变量，如果要求一个字符，后面必须跟一个字符型变量。

例 2.14　声明一个整型变量 age 然后等待用户从键盘输入到 cin 并将输入值存储在这个变量中。

```
int age;
cin >> age;
```

也可以连续使用>>，实现从键盘对多个变量输入数据。

格式 2：

```
cin>>变量 1>>变量 2>>变量 3…;
```

这要求从键盘输入的数据的个数、类型与变量相一致。从键盘读取数据时，各数据之间要有分隔符，分隔符可以是一个或多个空格键、回车键等。

例 2.15　用 cin 让用户输入多个数据 。

```
cin >> a >> b;
```

等同于：

```
cin >> a;
cin >> b;
```

例 2.16　流读取运算符>>和 cin 的使用

```
#include<iostream>
using namespace std;
int main()
{   char c;
    int i;
    float x,y;
    cout<<" enter:\n ";
    cin >>i>>x>>y;
    c=i;
    cout<<" c="<<c<<"\ti="<<i<<"\n ";
    cout<<" x="<<x<<"\ty="<<y<<"\n ";
    return 0;
}
```

程序运行时屏幕先显示：

enter:

这时从键盘输入一个整数和两个实数,中间用一个或多个空格键作分隔符。如输入

65 2.3 3.5

最后屏幕显示:

c=A i=65

x=2.3 y=3.5

程序中"\n"和"\t"都是转义符。'\n'和"\n"效果相同,都表示换行符。'\t'是制表符,可以理解为连续输出几个空格。字符变量和整型变量i的值都是65,但输出的形式不同。

五、格式化输入函数scanf(建议:初学者对格式初步了解,不必深究)

scanf函数的功能是格式化输入任意数据列表,其一般调用格式为:

scanf(格式控制符,地址列表)

【说明】

(1)地址列表中给出各变量的地址,可以是变量的地址,也可以是字符串的首地址。

(2)格式控制符由%和格式符组成,作用是将要输入的字符按指定的格式输入,如%d,%c等。

scanf函数的格式符

格式符	说 明
d,i	用于输入十进制整数
u	以无符号十进制形式输入十进制整数
o(字母)	用于输入八进制整数
x	用于输入十六进制整数
c	用于输入单个字符
s	用于输入字符串(非空格开始,空格结束,字符串变量以'\0'结尾)
f	用于输入实数(小数或指数均可)
e	与f相同(可与f互换)

scanf函数的附加格式说明符

附加格式	说 明
l(字母)	用于长整型数(%ld、%lo、%lx)或double型实数(%lf、%le)
h	用于短整型数(%hd、%ho、%hx)
域宽(一个整数)	指定输入所占列宽
*	表示对应输入量不赋给一个变量

例 1　*格式符的使用

```
#include<cstdio>
int main()
{
  int a,b;
  scanf("%d%*d%d",&a,&b);
  printf("a=%d,b=%d\n",a,b);
}
```

输入:1 2 3 回车

输出:a=1,b=3

例 2　域宽格式符的使用

```
#include<cstdio>
int main()
{
  int a,b;
  scanf("%4d%4d",&a,&b);
  printf("a=%d,b=%d\n",a,b);
}
```

输入:1234567 回车

输出:a=1234,b=567

例 3　%s 格式符的使用

```
#include<cstdio>
int main()
{
  char st[40];
  scanf("%s",st);
  printf("Your input is:%s\n",st);
}
```

输入:abc hello

输出:abc

例 4　非格式符的使用

```
#include<cstdio>
int main()
{
  int a,b,c;
  scanf("%d,%d,%d",&a,&b,&c);
  printf("a=%d,b=%d,c=%d\n",a,b,c);
} //例中 scanf 用非格式字符',作间隔符
```

故输入应为:5,6,7(本来输入:5 6 7)

输出:a=5,b=6,c=7

六、格式化输出函数 printf(建议:初学者对格式初步了解,不必深究)

printf 函数的功能是格式化输出任意数据列表,其一般调用格式为:

printf(格式控制符,输出列表)

【说明】

(1)格式控制由输入格式说明和普通字符组成,必须用双引号括起来。

①格式说明由%和格式字符组成,作用是将要输出的字符转换为指定的格式,如%d,%c等。

②普通字符是在输出时原样输出的字符,一般在显示时起提示作用。

(2)输出列表是需要输出的一组数据(可以为表达式和变量),各参数之间用“,”分开。要求格式说明和各输出项在数量和类型上要一一对应,否则将会出现意想不到的错误。

printf 函数的格式符

格式符	说　明
d(或 i)	以带符号的十进制形式输出整数,正数的(+)号省略不输出
u	以无符号十进制形式输出整数
x(或 X)	以十六进制无符号形式输出整数(不输出前导符 0x)
o(字母)	以八进制无符号形式输出整数(不输出前导符数字 0)
c	输出一个字符
s	输出字符串
f	以小数形式输出单、双精度,隐含输出 6 位小数
e(或 E)	以指数形式输出单、双精度,隐含输出 6 位小数
g(或 G)	自动选用%f、%e 或%E 格式中输出宽度较小的一种使用

d 格式符

参 数	说　明
%d	输出数字长为变量数值的实际长度
%md	输出 m 位(不足补空格,大于 m 位时按实际长度输出)
%-md	m 含义同上。左对齐输出
%ld	l(小写字母)表示输出"长整型"数据
%mld	指定长整型输出宽度 m 位,左边补空格;否则,按实际位数输出
%0md,%0mld	0(数字 0)表示位数不足 m 时补 0

例如,对如下程序段

```
int i=1;
long  j=123;
printf("%d,%2d,%03d,%1d,%-4ld,%05ld ",i,i,i,j,j,j);
```

输出:1,__1,001,123,123__,00123

f 格式符

参　数	说　明
%f	按实数格式输出,整数部分按实际位数输出,6 位小数
%m.nf	总位数 m(含小数点),其中有 n 位小数
%-m.nf	同上,左对齐

s格式符

参　数	说　明
%s	按实际宽度输出一个字符串
%ms	m指定宽度(不足时左补空格,大于时按实际宽度输出)
%-ms	左对齐,不足时右补空格
%m.ns	输出占m个字符位置,其中字符数最多n个,左补空格
%-m.ns	同上,右补空格

例2.17　某幼儿园里,有5个小朋友编号为1、2、3、4、5,他们按自己的编号顺序围坐在一张圆桌旁。他们身上都有若干个糖果(键盘输入),现在他们做一个分糖果游戏。从1号小朋友开始,将自己的糖果均分三份(如果有多余的糖果,则立即吃掉),自己留一份,其余两份分给他的相邻的两个小朋友。接着2号、3号、4号、5号小朋友同样这么做。问一轮后,每个小朋友手上分别有多少糖果?

【分析】　题目中有5位小朋友,他们初始时糖果的数目不确定,用a、b、c、d、e分别存储5个小朋友的糖果数,初始值由键盘输入。

程序如下:

```
#include<cstdio>
using namespace std;
int main()
{
  int a,b,c,d,e;
  scanf("%d%d%d%d%d",&a,&b,&c,&d,&e);
  a=a/3;b=b+a;e=e+a;                          //1号小朋友分糖
  b=b/3;c=c+b;a=a+b;                          //2号小朋友分糖
  c=c/3;d=d+c;b=b+c;                          //3号小朋友分糖
  d=d/3;e=e+d;c=c+d;                          //4号小朋友分糖
  e=e/3;a=a+e;d=d+e;                          //5号小朋友分糖
  printf("%5d%5d%5d%5d%5d\n",a,b,c,d,e);      //%5d按5位宽度输出
  return 0;
}
```

运行结果:

输入:8 9 10 11 12

输出:11 7 9 11 6

七、几种输入输出格式的几点说明

1. cin和cout在Dev C++中只能调用<iostream>库,而其他输入输出格式要调用<stdio.h>库或<cstdio>库。

2. cin 和 cout 属于 C++的概念,调用时涉及输入输出流,而 scanf 和 printf 属于 C 的概念,是 C 语言的标准输入/输出库中的函数,所以在时效上,scanf 和 printf 优于 cin、cout,对于大数据的输入输出,通常情况下应该用 scanf、printf。

3. 对于普通数据的输入输出,cin 和 cout 比较方便,而在格式化方面,scanf 和 printf 比较容易。cin 效率比 scanf 低很多,尤其输入数据达到 20 万以上时非常明显,在 100 万时 cin 读入就会超时。

4. scanf 和 printf 也有缺点,cin 和 cout 能够自动识别变量的数据类型,因此,在进行输入输出时,不需要指定数据类型,printf 和 scanf 函数在输入输出时需指定数据类型。

【上机练习】

1. 输出保留 3 位小数的浮点数【1.1 编程基础之输入输出 04】

读入一个单精度浮点数,保留 3 位小数输出这个浮点数。

2. 输出保留 12 位小数的浮点数【1.1 编程基础之输入输出 05】

读入一个双精度浮点数,保留 12 位小数,输出这个浮点数。

3. 空格分隔输出【1.1 编程基础之输入输出 06】

读入一个字符,一个整数,一个单精度浮点数,一个双精度浮点数,然后按顺序输出它们,并且要求在他们之间用一个空格分隔。输出浮点数时保留 6 位小数。

4. 输出浮点数【1.1 编程基础之输入输出 07】

读入一个双精度浮点数,分别按输出格式"%f","%f"保留 5 位小数,"%e"和"%g"的形式输出这个整数,每次在单独一行上输出。

5. 字符菱形【1.1 编程基础之输入输出 09】

给定一个字符,用它构造一个对角线长 5 个字符,倾斜放置的菱形。

```
  *
 * * *
* * * * *
 * * *
  *
```

第六节　顺序结构实例

我们已经学习了数据输入输出、赋值语句以及基本的数据类型。下面举一些实例，通过阅读和模仿这些程序，让选手逐步熟悉程序的编写和巩固知识点，为以后各章的学习打好基础。

例 2.18　输入一个三位数，要求把这个数的百位数与个位数对调，输出对调后的数。

【分析】　先求出自然数的个位数、十位数、百位数，然后个位数与百位数对调。

程序如下：

```
#include<iostream>
using namespace std;
int main()
{   int m;
  cin>>m;                          //输入一个三位数
  int a=m/100;                     //百位数
  int b=(m/10)%10;                 //十位数
  int c=m%10;                      //个位数
  int n=c*100+b*10+a;              //重新组合对调后的数
  cout<<" n="<<n<<endl;            //输出结果
}
```

运行结果：

输入：234　　输出：n=432

例 2.19　已知某班有男同学 x 位，女同学 y 位，x 位男生平均分是 87 分，y 位女生的平均分是 85，问全体同学平均分是多少？

【分析】　男女生的人数需要用户输入，然后根据题意(x*87+y*85)/(x+y)求出全体同学的平均分。

程序如下：

```
#include<iostream>
using namespace std;
int main()
{
  int x,y;
  cin>>x>>y;                       //输入男女人数
  cout<<float(x*87+y*85)/(x+y)<<endl;
                                   //数据类型强制转换，按实数格式输出，4 位小数
}
```

例 2.20　歌手大奖赛上 6 名评委给一位参赛者打分，6 个人打分的平均分为 9.6 分；如果去掉一个最高分，这名参赛者的平均分为 9.4 分；如果去掉一个最低分，这名参赛者的平

均分为9.8分;如果去掉一个最高分和一个最低分,这名参赛者的平均是多少?

【分析】 首先求出6名评委的总分,然后根据去掉最高分的总分和最低分的总分,求出最高分的分值和最低分的分值,最后总分减去最高分和最低分除以4即是答案。

程序如下:

```
#include<cstdio>
int main()
{
  float sc_all=6*9.6;                  //求6名评委的总分
  float sc_high=5*9.4;                 //求去掉最高分后的总分
  float sc_low=5*9.8;                  //求去掉最低分后的总分
  float high=sc_all-sc_high;           //求最高分
  float low=sc_all-sc_low;             //求最低分
  float ans=(sc_all-high-low)/4;       //求平均分
  printf("%5.2f\n",ans);               //%5.2f按实数格式输出,保留2位小数
}
```

运行结果:9.60

例2.21 传说古代的叙拉古国王海伦二世发现的公式,利用三角形的三条边长来求取三角形面积。已知△ABC中的三边长分别为a,b,c,求△ABC的面积。[提示:海伦公式 $s=\sqrt{p(p-a)(p-b)(p-c)}$,其中p=(a+b+c)/2)]

【分析】 公式中p是三角形周长的一半,求出p后直接代入海伦公式中求得面积。

```
#include<cstdio>                       //Dev C++可调用<iostream>和<stdio.h>库
#include<math.h>                       //在Dev C++中可调用数学函数库cmath
int main()
{
  float a,b,c;
  scanf("%f%f%f",&a,&b,&c);//输入三角形的三边
  float p=(a+b+c)/2;                   //求出p的值
  float s=sqrt(p*(p-a)*(p-b)*(p-c));//根据p求面积,sqrt是开方函数
  printf("%0.3f\n",s);                 //输出面积,0.3f按实际位数输出,保留3位小数
}
```

运行结果:

输入:3 4 5　　输出:6.000

例2.22 分钱游戏。甲、乙、丙三人共有24元钱,先由甲分钱给乙、丙两人,所分给的数与各人已有数相同;接着由乙分给甲、丙,分法同前;再由丙分钱给甲、乙,分法亦同前。经上述三次分钱之后,每个人的钱数恰好一样多。求原先各人的钱数分别是多少?

【分析】 设甲、乙、丙三人的钱数分别为a,b,c。用倒推(逆序)算法,从最后结果入手,按反相顺序,分步骤推算出每次各人当时的钱数 :(在每个步骤中,各人钱数分别存在a、b、c中)

```
#include<cstdio>                       //Dev C++可调用<iostream>和<stdio.h>库
int main()                             //调用iostream库,使用printf语句在编译通不过
```

```
{
  int a=8,b=8,c=8;          //对应于步骤①
  a/=2;b/=2;c=a+b+c;        //对应于步骤②
  a/=2;c/=2;b=a+b+c;        //对应于步骤③
  b/=2;c/=2;a=a+b+c;        //对应于步骤④
  printf(" a=%-5db=%-5dc=%-5d\n ",a,b,c);
                            //%-5d 按 5 位宽度输出,左对齐
}
```

运行结果：

a=13　b=7　c=4

例 2.23　求一元二次方程 $x^2+3x+2=0$ 的两个实数根。

【分析】　方程的系数是常量，分别用 A、B、C 表示，运用数学求方程的根，采取如下方法：

① 先求出 $d=B^2-4AC$；(求根公式中需用开方运算的那部分)

② 再用求根公式算出 x1，x2 的值。(x1，x2=?)

③ 输出 x1，x2。

```
#include<cstdio>                    //Dev C++可调用<stdio.h>库
#include<math.h>                    //在 Dev C++中可调用数学函数库 cmath
#define A  1
#define B  3                        //常量说明,A,B,C 表示方程系数
#define C  2
int main()
{
    int d=B*B-4*A*C;                //d 为整型变量
    float x1,x2;                    //x1,x2 为实型变量
    x1=(-B+sqrt(d))/(2*A);          //求方程的根
    x2=(-B-sqrt(d))/(2*A);
    printf(" x1=%-8.3fx2=%-8.3f\n ",x1,x2);
                                    //%-8.3f 按 8 位宽度,3 位小数,左对齐
}
```

运行结果：

x1=-1.000　x2=-2.000

【Dev C++版本注意事项】

在 5.0 版本 Dev C++中，<iostream>不是万能库，不包括 cstdio、cstring、cstdlib，其他常用的 min、max 函数也不保证包括，需要注意如下几个细节：

①getchar()，stdin，stdout，freopen()，fclose()，scanf()在 C++中需要调用<cstdio>。

②memset()需要包含 C 头文件<string.h>，在 C++中需要调用<cstring>。

③qsort()，exit()需要包含 C 头文件<stdlib.h>，在 C++中需要调用<cstdlib>。

在 5 版本 Dev C++中，根据标准去除了 iostream 对 cstdio、cstring、algorithm 的依赖

性,需要显式包括。你现在还可能不透彻理解以上注意事项,就慢慢消化吧,在信息学奥赛中一定要注意以上细节,请记住:“细节决定成败”。

【上机练习】

1. 计算浮点数相除的余【1.3 编程基础之算术表达式与顺序执行 11】

计算两个双精度浮点数 a 和 b 的相除的余数,a 和 b 都是双精度浮点数。这里余数(r)的定义是:a = k * b + r,其中 k 是整数,0<=r<b。

2. 计算球的体积【1.3 编程基础之算术表达式与顺序执行 12】

对于半径为 r 的球,其体积的计算公式为 V=4/3 * πr3,这里取 π=3.14。现给定 r,即球半径,类型为 double,求球的体积 V,保留到小数点后 2 位。

3. 反向输出一个三位数【1.3 编程基础之算术表达式与顺序执行 13】

将一个三位数反向输出,例如输入 358,反向输出 853。

4. 大象喝水【1.3 编程基础之算术表达式与顺序执行 14】

一只大象口渴了,要喝 20 升水才能解渴,但现在只有一个深 h 厘米,底面半径为 r 厘米的小圆桶(h 和 r 都是整数)。问大象至少要喝多少桶水才会解渴。

5. 计算线段长度【1.3 编程基础之算术表达式与顺序执行 16】

已知线段的两个端点的坐标 A(Xa,Ya),B(Xb,Yb),求线段 AB 的长度,保留到小数点后 3 位。

6. 计算三角形面积【1.3 编程基础之算术表达式与顺序执行 17】

平面上有一个三角形,它的三个顶点坐标分别为(x1, y1),(x2, y2),(x3, y3),那么请问这个三角形的面积是多少,精确到小数点后两位。

7. A * B 问题【1.3 编程基础之算术表达式与顺序执行 19】

输入两个正整数 A 和 B,求 A * B 的值。注意乘积的范围和数据类型的选择。

8. 计算 2 的幂【1.3 编程基础之算术表达式与顺序执行 20】

给定非负整数 n,求 2^n 的值,即 2 的 n 次方。

9. 苹果和虫子【1.3 编程基础之算术表达式与顺序执行 15】

你买了一箱 n 个苹果,很不幸的是买完时箱子里混进了一条虫子。虫子每 x 小时能吃掉一个苹果,假设虫子在吃完一个苹果之前不会吃另一个,那么经过 y 小时你还有多少个完整的苹果?

10. 求三角形面积【1.3 等差数列末项计算 18】

给定三条线段的长度,判断这三条线段是否能够构成三角形。如果能够构成,则计算其面积。

第三章　程序的控制结构

第一节　概　述

程序由若干条语句组成，各语句按照顺序一条一条地执行，这种顺序结构是简洁的。但在现实世界中，在解决问题的过程中，不可避免地遇到需要进行选择或需要循环工作的情况。这时，程序执行的顺序需要发生变化，而非从前向后逐一执行。因此，程序中除了顺序结构以外，通常还有选择结构、循环结构以及转移机制。

C++为了支持这些控制结构，提供了丰富、灵活的控制语句。从结构化程序设计的观点看，所有程序都可用3种控制结构即顺序结构、选择结构和循环结构实现。C++在默认的情况下采取顺序结构，除非特别指明，计算机总是按语句顺序一条一条地执行。为使程序更清晰、更易调试与修改，并且不容易出错，结构化编程要尽量少用或不用goto等跳转语句。

选择类语句包括if语句和switch语句，用它们来解决实际应用中按不同的情况进行不同处理的问题。如根据学生的成绩，对学生做出不同的等第评价。if选择结构称为单分支选择结构，选择或忽略一个分支的操作。if-else选择结构称为双分支选择结构，在两个不同分支中选择。switch选择结构称为多分支(或多项)选择结构，以多种不同的情况选择多个不同的操作。

循环类语句包括for循环语句、while循环语句和do循环语句三种，用它们来解决实际应用中需要重复处理的问题。如当统计全班同学总分时，就需要重复地做加法，依次把每个人的分数累加起来。

if、else、switch、while、do和for等都是C++关键字。这些关键字是该语言保留的，用于实现C++控制结构的不同特性。关键字不能作为变量名等一些标识符。注意，将关键字while的拼写变为“While”是个语法错误，因为C++是区分大小写的语言。while、if和else等所有C++保留关键字只能包含小写字母。

第二节 if选择结构

C++提供三种选择结构,即if选择结构、if-else选择结构和switch选择结构。

一、if语句(单分支结构)

格式1:

```
if(条件表达式)
  语句1;
```

功能:如果条件表达式的值为真,即条件成立,语句1将被执行。否则,语句1将被忽略(不被执行),程序将按顺序从整个选择结构之后的下一条语句继续执行。执行流程如图3-1所示:

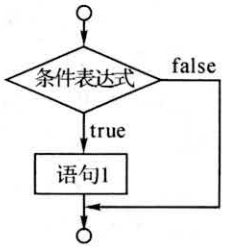


图3-1

说明:格式中的"条件表达式"必须用圆括号括起来。

例3.1 读入一个整数a,如果a为偶数在屏幕上输出yes。

```
#include<iostream>
using namespace std;
int main()
{
  int a;
  cin>>a;
  if (a%2==0)
    cout<<" yes "<<endl;
  return 0;
}
```

注意:关系运算符==用来表达该符号的左右两边是否相等,不要写成赋值号=。

试一试

若题目改为"读入一个整数a,如果a为奇数在屏幕上输出no",该如何修改程序?

例3.2 读入一个数,若这个数大于1并且小于100,则输出yes。

```
# include<iostream>
using namespace std;
int main()
{
  int a;
  cin>>a;
  if ((a>1)&&(a<100))
    cout<<" yes "<<endl;
  return 0;
}
```

注意：此程序中的条件表达式为(a>1)&&(a<100)，根据要求“条件表达式”必须用圆括号括起来，否则编译会出错。

格式2：

```
if (条件表达式)
  {
    语句1;
    语句2;
    …
  }
```

若条件成立时，要执行的操作由多个句子构成，我们必须把这些句子括在一对花括号{}内，我们称这种形式为语句块或复合语句。

程序设计风格提示：书写语句块(也称为复合语句)时，左右花括号要对齐，组成语句块的各语句要相对花括号缩进一层并对齐。

例3.3　输入三个整数，按从大到小的顺序输出。

【分析】输入的三个数存放在a、b、c中，设想让a为三数中最大数，怎么办呢？如果a<b，那么让a和b的值交换，保证了a>=b；如果a<c，那么让a和c的值交换，就保证了a>=c；设想让b为第二大的数，c为第三大的数，怎么做呢？如果b<c，那么让b和c的值交换，就保证了b>=c，最后输出a,b,c的值。

程序如下：

```
#include<iostream>
using namespace std;
int main()
{
    int a,b,c,temp;
    cin>>a>>b>>c;
    if (a<b)                          //保证a大于等于b
    {
        temp=a;a=b;b=temp;
    }
    if (a<c)                          //保证a大于等于c,则a为最大数
    {
        temp=a;a=c;c=temp;
    }
    if (b<c)                          //保证b大于等于c
    {
        temp=b;b=c;c=temp;
    }
    cout<<a<<" "<<b<<" "<<c<<endl;
    return 0;
}
```

二、if-else 语句(双分支结构)

if 单分支选择结构只在条件为 true 时采取操作,条件为 false 时则忽略这个操作。利用 if-else 双分支选择结构则可以在条件为 true 时和条件为 false 时采取不同操作。

格式 1:

```
if (条件表达式)
    语句 1;
else
    语句 2;
```

功能:如果(条件表达式)的值为"真",即条件成立,则执行语句 1,执行完"语句 1"后继续执行整个 if-else 语句的后继语句;如果(条件表达式)的值为"假",即条件不成立,那么跳过语句 1 选择执行"语句 2",执行完语句 2 后继续执行整个 if-else 语句的后继语句;也就是说 if-else 语句总是根据(条件表达式)的结果,选择"语句 1"和"语句 2"中的一个执行,执行完以后,整个 if-else 就算执行完了。执行流程如图 3-2 所示

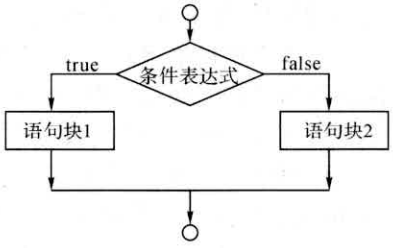


图 3-2

程序设计风格提示:书写 if-else 语句时,if 和 else 要对齐,而分支的语句部分要缩进两格。

例 3.4 输入温度 t 的值,判断是否适合晨练。(25<=t<=30,则适合晨练 ok,否则不适合 no)

```
#include<iostream>
using namespace std;
int main()
{
  int t;
  cin >> t;
  if ((t>=25) &&(t<=30))
    cout<<" ok! "<<endl;
  else
    cout<<" no! "<<endl;
  return 0;
}
```

格式 2:

```
if(条件表达式)
  { 语句 1;
    语句 2;
    …
  }
```

```
    else { 语句 1;
           语句 2;
           …
         }
```

例 3.5　输入三个数，输出其中最大的数。

【方法 1】设 maxn 用于存放三个数中最大的数，输入的三个数存放在 a、b、c 中，那么如果 a 比 b 和 c 大，则最大数是 a，否则，如果 b 比 a 和 c 大，则最大数是 b，否则，最大数是 c。

程序如下：

```
#include<iostream>
using namespace std;
int main()
{
    float a,b,c,maxn;
    cin>>a>>b>>c;
    if (a>b&&a>c) maxn=a;          //判断 a 是否最大
    else if (b>a&&b>c) maxn=b;     //判断 b 是否最大
    else maxn=c;
    cout<<maxn<<endl;
    return 0;
}
```

【方法 2】设 maxn 用于存放三个数中最大的数，输入的三个数存放在 a、b、c 中，初值 maxn=a，即假设 a 为最大，那么如果 b>maxn，则此时的最大数应该是 b 即 maxn=b，如果 c>maxn，则最大数应该是 c，即 maxn=c。

程序如下：

```
#include<iostream>
using namespace std;
int main()
{
    float a,b,c,maxn;
    cin>>a>>b>>c;
    maxn=a;
    if (b>maxn) maxn=b;            //maxn 为 a,b 中的最大值
    if (c>maxn) maxn=c;            //maxn 为 a,b,c 中的最大值
    cout<<maxn<<endl;
    return 0;
}
```

格式 2：

```
if (条件表达式)
  { 语句 1;
    语句 2;
    ……
  }
```

```
else
  {语句 1;
   语句 2;
   ……
  }
```

若分支语句由多个句子构成,我们必须把这些句子括在一对花括号{ }内。

例 3.6 乘坐飞机时,当乘客行李小于等于 20 公斤时,按每公斤 1.68 元收费,大于 20 公斤时,按每公斤 1.98 元收费,编程计算收费(保留 2 位小数)。

```
#include<cstdio>
using namespace std;
int main()
{
  float  w,s;
  scanf("%f",&w);
  if (w<=20)
  {
    s=w*1.68;
    printf("%.2f\n",s);          //printf("%.2f\n",w*1.68)等价于以上 2 行
  }
  else
  {
    s=w*1.98
    printf("%.2f\n",s);          //printf("%.2f\n",w*1.98)等价于以上 2 行
  }
  return 0;
}
```

if 语句允许嵌套,即语句 1 和语句 2 还可以是 if 语句,当 if 语句嵌套时,约定 else 总是和最近的一个 if 语句配对。

例 3.7

```
if (a>b)
        if (b>c) y=a;
              else y=c;
```

else 部分否定的是条件 b>c,即它与第二个 if 语句配对;若想让 else 部分与第一个 if 语句配对,则要引入一个复合语句,将上述语句写成如下形式:

```
if (a>b)
  {
    if (b>c) y=a;
  }
else  y=c;
```

三、三目运算符

C++有一个常用来代替 if-else 语句的操作符，这个操作符被称为三目运算符(?:)，它是 C++中唯一一个需要 3 个操作数的操作符。该操作符的通用格式如下：

b? a:c

如果 b 为 true，则整个表达式的值为 a；否则，整个表达式的值为 c。下面两个语句演示了该操作符是如何工作的：

```
x=5>3? 10:12                    // 5>3 为 true,所以 x=10;
x=3==9? 25:18                   // 3==9 为 false,所以 x=18;
```

例 3.8

```
#include<iostream>
using namespace std;
int main()
{
  int a,b;
  cin>>a>>b;
  int c=a>b? a:b;               // 相当于 int c; if (a>b) c=a; else c=b;
  cout<<c<<endl;
  return 0;
}
```

该例使用三目运算符来确定两值中较大的一个。

与 if-else 序列相比，三目运算符更简洁，但第一次使用时不那么容易理解。这两种方法之间的区别是，三目运算符返回一个值，可以将其赋给变量或者将其放到一个更大的表达式中。例如：x=a>b? (c>d? e:f):g;

相当于

```
if (a>b)
    {
        if (c>d) x=e;
        else x=f;
    }
    else x=g;
```

从可读性来说，条件操作符最适合于简单关系和简单表达式的值：

x=(a>b)? a:b;

当代码变得更复杂时，使用 if-else 语句表达更为清晰。

【上机练习】

1. 判断数正负【1.4 编程基础之逻辑表达式与条件分支 01】

给定一个整数 N,判断其正负。如果 N>0,输出 positive;如果 N=0,输出 zero;如果 N<0,输出 negative。

2. 输出绝对值【1.4 编程基础之逻辑表达式与条件分支 02】

输入一个浮点数,输出这个浮点数的绝对值,保留到小数点后两位。

3. 奇偶数判断【1.4 编程基础之逻辑表达式与条件分支 03】

给定一个整数,判断该数是奇数还是偶数。如果 n 是奇数,输出 odd;如果 n 是偶数,输出 even。

4. 奇偶 ASCII 值判断【1.4 编程基础之逻辑表达式与条件分支 04】

任意输入一个字符,判断其 ASCII 是否是奇数,若是,输出 YES,否则,输出 NO。例如,字符 A 的 ASCII 值是 65,则输出 YES,若输入字符 B(ASCII 值是 66),则输出 NO。

5. 整数大小比较【1.4 编程基础之逻辑表达式与条件分支 05】

输入两个整数,比较它们的大小。若 x>y,输出>;若 x=y,输出=;若 x<y,输出<。

6. 判断是否为两位数【1.4 编程基础之逻辑表达式与条件分支 06】

判断一个正整数是否是两位数(即大于等于 10 且小于等于 99)。若该正整数是两位数,输出 1,否则输出 0。

7. 收集瓶盖赢大奖【1.4 编程基础之逻辑表达式与条件分支 07】

某饮料公司最近推出了一个“收集瓶盖赢大奖”的活动:如果你拥有 10 个印有“幸运”、或 20 个印有“鼓励”的瓶盖,就可以兑换一个神秘大奖。现分别给出你拥有的印有“幸运”和“鼓励”的瓶盖数,判断是否可以去兑换大奖。若可以兑换大奖,输出 1,否则输出 0。

8. 判断一个数能否同时被 3 和 5 整除【1.4 编程基础之逻辑表达式与条件分支 08】

判断一个数 n 能否同时被 3 和 5 整除,如果能同时被 3 和 5 整除输出 YES,否则输出 NO。

9. 判断能否被 3,5,7 整除【1.4 编程基础之逻辑表达式与条件分支 09】

给定一个整数,判断它能否被 3,5,7 整除,并输出以下信息:

①能同时被 3,5,7 整除(直接输出 3 5 7,每个数中间一个空格);

②只能被其中两个数整除(输出两个数,小的在前,大的在后。例如:3 5 或者 3 7 或者 5 7,中间用空格分隔);

③只能被其中一个数整除(输出这个除数);

④不能被任何数整除,输出小写字符‘n’,不包括单引号。

10. 有一门课不及格的学生【1.4 编程基础之逻辑表达式与条件分支 10】

给出一名学生的语文和数学成绩,判断他是否恰好有一门课不及格(成绩小于 60 分)。若该生恰好有一门课不及格,输出 1;否则输出 0。

第三节　switch 语句

应用条件语句可以很方便地使程序实现分支，但是出现分支比较多的时候，虽然可以用嵌套的 if 语句来解决，但是程序结构会显得复杂，其至凌乱。为方便实现多情况选择，C++提供了一种 switch 开关语句。

1. 语句格式

```
switch(表达式)
{
    case  常量表达式 1:
          语句序列 1;
          break;
    case  常量表达式 2:
          语句序列 2;
          break;
          …
    case  常量表达式 n:
          语句序列 n;
          break;
    default:
           语句序列 n+1;
}
```

该语句中可以使用一次或多次 case 标号，但只能使用一次 default 标号，或者省略整个 default 部分；多个 case 标号也允许使用在同一个语句序列的前面；每个语句标号由保留字 case 和后面的常量表达式及冒号组成，每个常量表达式通常为字面常量，如常数或字符。

2. 语句执行过程

switch 语句执行过程分为以下 3 步描述。

(1)计算出 switch 后面圆括号内表达式的值，假定为 M，若它不是整型，系统将自动舍去其小数部分，只取其整数部分作为结果值。

(2)依次计算出每个 case 后常量表达式的值，假定它们为 M1、M2、…，同样，若它们的值不是整型，则自动转换为整型。

(3)让 M 依次同 M1、M2、…进行比较，一旦遇到 M 与某个值相等，则就从对应标号的语句开始执行；在碰不到相等的情况下，若存在 default 子句，则就执行其冒号后面的语句序列，否则不执行任何操作；当执行到复合语句最后的右花括号时就结束整个 switch 语句的执行。

在实际使用 switch 语句时，通常要求当执行完某个 case 后的一组语句序列后，就结束整个语句的执行，而不让它继续执行下一个 case 语句后面的语句序列，为此，可通过使用 break 语句来实现。该语句只有保留字 break，而没有其他任何成分。它是一条跳转语句，在 switch 中执行到它时，将结束该 switch 语句，系统接着向下执行其他语句。

在使用 switch 语句时，还应注意以下几点：

①case 语句后的各常量表达式的值不能相同,否则会出现错误码。

②每个 case 或 default 后,可以包含多条语句,不需要使用"{"和"}"括起来。

③各 case 子句的先后顺序可以变动,这不会影响程序执行结果。

④default 子句可以省略,default 后面的语句末尾可以不必写 break。

程序设计风格提示:写 switch 语句时,switch(表达式)单独一行,各 case 分支和 default 分支要缩进两格并对齐,分支处理语句要相对再缩进两格,以体现不同层次的结构。

3. 语句格式举例

(1)左右两边的书写格式是等价的

```
switch(a)
  {
    case 1:x++;break;
    case 2:y++;break;
    case 3:z++;break;
    default:cout<<"error";
  }
```

```
switch(a)
   { case 1:
       x++; break;
     case 2:
       y++; break;
     case 3:
       z++; break;
     default:
       cout<<"error";
   }
```

(2)

```
switch(ch)
  {
    case 'a':
    case 'A':
         d1=(x+y)/2;
         d2=x*y-2;
         break;
    case 'b':
    case 'B':
         d1=(a+b)/2;
         d2=a*b-2;
         break;
    default:
         cout<<" input error!";
  }
```

说明:1. 每个 case 后面的语句可以写在冒号后的同一行或换到新行写。

2. <语句序列 1>…<语句序列 n+1>都是一组语句,有时可为空。如(2)。

例 3.9 根据从键盘上输入的表示星期几的数字,对应输出它的英文名称。

```
#include<iostream>
using namespace std;
int main()
{
```

```
    int weekday;
    cin>>weekday;
    switch(weekday)
    {
      case 1: cout<<" Monday "<<endl; break;
      case 2: cout<<" Tuesday "<<endl; break;
      case 3: cout<<" Wednesday "<<endl; break;
      case 4: cout<<" Thursday "<<endl; break;
      case 5: cout<<" Friday "<<endl; break;
      case 6: cout<<" Saturday "<<endl; break;
      case 7: cout<<" Sunday "<<endl; break;
      default: cout<<" input  error! "<<endl;
    }
    return 0;
}
```

例 3.10　一个最简单的计算器支持+,-,*,/四种运算。输入只有一行:两个参加运算的数和一个操作符(+,-,*,/)。输出运算表达式的结果。考虑下面两种情况:

(1)如果出现除数为 0 的情况,则输出:Divided by zero!

(2)如果出现无效的操作符(即不为+,-,*,/之一),则输出:Invalid operator!

输入样例:

34　56

输出样例:

90

【分析】设 num1、num2 存放两个参加运算的操作数,op 存放操作符。

①当 op 为"+"号时,实现加法操作。

②当 op 为"-"号时,实现减法操作。

③当 op 为"*"号时,实现乘法操作。

④当 op 为"/"号时,判断 b 值,如果不为 0,则实现除法操作,如果为 0,则输出:Divided by zero!

⑤当 op 不是上面四种操作符时,输出:"Invalid operator!"

程序如下:

```
#include<iostream>
using namespace std;
int main()
{
  float num1,num2;
  char op;
  cin>>num1>>num2>>op;
  switch(op)
  {
     case '+': cout<<num1+num2<<endl;break;
```

```
        case '-': cout<<num1-num2<<endl;break;
        case '*': cout<<num1*num2<<endl;break;
        case '/': if(num2! =0) { cout<<num1/num2<<endl;break; }
        else cout<<" Divided by zero! "<<endl;break;
        default: cout<<" Invalid operrator! "<<endl;
    }
    return 0;
}
```

例 3.11 期末来临了,班长小 Q 决定将剩余班费 x 元钱,用于购买若干支钢笔奖励给一些学习好、表现好的同学。已知商店里有三种钢笔,它们的单价分别为 6 元、5 元和 4 元。小 Q 想买尽量多的笔(鼓励尽量多的同学),同时他又不想有剩余钱。请你编一程序,帮小 Q 制订出一种买笔的方案。

【分析】 对于以上的实际问题,要买尽量多的笔,易知都买 4 元的笔肯定可以买最多支笔。因此最多可买的笔为 x/4 支。由于小 Q 要把钱用完,故我们可以按以下方法将钱用完:

若买完 x/4 支 4 元钱的笔,还剩 1 元,则 4 元钱的笔少买 1 支,换成一支 5 元笔即可;若买完 x/4 支 4 元钱的笔,还剩 2 元,则 4 元钱的笔少买 1 支,换成一支 6 元笔即可;若买完x/4支 4 元钱的笔,还剩 3 元,则 4 元钱的笔少买 2 支,换成一支 5 元笔和一支 6 元笔即可。

从以上对买笔方案的调整,可以看出笔的数目都是 x/4,因此该方案的确为最优方案。

程序如下:

```
#include<iostream>
using namespace std;
int main()
{
    int a,b,c,x,y;          //a,b,c 分别表示在买笔方案中,6 元、5 元和 4 元钱笔的数目
                            //x,y 分别表示剩余班费和买完最多的 4 元笔后剩的钱
    cin>>x;                 //输入 x
    c=x/4;                  //4 元笔最多买的数目
    y=x%4;                  //求买完 c 支 4 元笔后剩余的钱数 y
    switch (y)              //判断购买方案
    {
        case 0: a=0; b=0;           break;
        case 1: a=0; b=1; c--;      break;
        case 2: a=1; b=0; c--;      break;
        case 3: a=1; b=1; c--=2;    break;
    }
    cout<<a<<' '<<b<<' '<<c<<endl;      //三个数间以空格隔开
    return 0;
}
```

【上机练习】

1. 晶晶赴约会【1.4 编程基础之逻辑表达式与条件分支 11】

晶晶的朋友贝贝约晶晶下周一起去看展览，但晶晶每周的 1、3、5 有课必须上课，请帮晶晶判断她能否接受贝贝的邀请，如果能输出 YES；如果不能则输出 NO。注意 YES 和 NO 都是大写字母！

2. 骑车与走路【1.4 编程基础之逻辑表达式与条件分支 12】

在清华校园里，没有自行车，上课办事会很不方便。但实际上。并非去办任何事情都是骑车快，因为骑车总要找车、开锁、停车、锁车等，这要耽误一些时间。假设找到自行车，开锁并车上自行车的时间为 27 秒；停车锁车的时间为 23 秒；步行每秒行走 1.2 米，骑车每秒行走 3.0 米。请判断走不同的距离去办事，是骑车快还是走路快。如果骑车快，输出一行“Bike”；如果走路快，输出一行“Walk”；如果一样快，输出一行“All”。

3. 分段函数【1.4 编程基础之逻辑表达式与条件分支 13】

编写程序，计算下列分段函数 y=f(x)的值。结果保留到小数点后三位。

y=-x+2.5;　　0<=x<5

y=2-1.5(x-3)(x-3);　　5<=x<10

y=x/2-1.5;　　10<=x<20

4. 计算邮资【1.4 编程基础之逻辑表达式与条件分支 14】

根据邮件的重量和用户是否选择加急计算邮费。计算规则：重量在 1000 克以内(包括 1000 克)，基本费 8 元。超过 1000 克的部分，每 500 克加收超重费 4 元，不足 500 克部分按 500 克计算；如果用户选择加急，多收 5 元。

5. 最大数输出【1.4 编程基础之逻辑表达式与条件分支 15】

输入三个整数，数与数之间以一个空格分开。输出一个整数，即最大的整数。

6. 三角形判断【1.4 编程基础之逻辑表达式与条件分支 16】

给定三个正整数，分别表示三条线段的长度，判断这三条线段能否构成一个三角形。如果能构成三角形，则输出“yes”，否则输出“no”。

7. 判断闰年【1.4 编程基础之逻辑表达式与条件分支 17】

判断某年是否是闰年。如果公元 a 年是闰年输出 Y，否则输出 N。

8. 点和正方形的关系【1.4 编程基础之逻辑表达式与条件分支 18】

有一个正方形，四个角的坐标(x,y)分别是(1,-1),(1,1),(-1,-1),(-1,1)，x 是横轴，y 是纵轴。写一个程序，判断一个给定的点是否在这个正方形内(包括正方形边界)。如果点在正方形内，则输出 yes，否则输出 no。

9. 简单计算器【1.4 编程基础之逻辑表达式与条件分支 19】

一个最简单的计算器，支持+，-，*，/ 四种运算。仅需考虑输入输出为整数的情况，数据和运算结果不会超过 int 表示的范围。然而：

1. 如果出现除数为 0 的情况，则输出：Divided by zero!

2. 如果出现无效的操作符(即不为 +，-，*，/ 之一)，则输出：Invalid operator!

10. 求一元二次方程【1.4 编程基础之逻辑表达式与条件分支 20】

利用公式 x1 = (-b + sqrt(b*b-4*a*c))/(2*a)，x2 = (-b - sqrt(b*b-4*a*c))/(2*a)，求一元二次方程 $ax^2+bx+c=0$ 的根，其中 a 不等于 0。结果要求精确到小数点后 5 位。

第四章　循环结构

前面我们学习了顺序结构与分支结构的程序设计。在实际应用中,会经常遇到许多有规律性的重复运算,这就需要掌握循环结构程序设计。C++语言提供三种循环结构 for、while 和 do—while。

第一节　for 语句

一、语句格式

格式 1:

```
for(控制变量初始化表达式;条件表达式;增量表达式)
    语句 1;
```

说明:语句 1 是 for 循环语句的循环体,它将在满足条件的情况下被重复执行。

格式 2:

```
for(控制变量初始化表达式;条件表达式;增量表达式)
  { 语句 1;
    语句 2;
    …
  }
```

说明:循环体部分由多个语句构成,应由一对花括号括起来,构成一个语句块的形式。

程序风格提示:写 for 循环语句时,循环体的语句相对于 for 缩进两格。

二、语句执行过程

for 语句的执行过程可由以下 4 步来描述。

(1)执行"控制变量初始化语句",使控制变量获得一个初值。

(2)判断控制变量是否满足"条件表达式",若满足条件则执行一遍循环体,否则结束整个 for 语句,继续执行 for 循环下面的句子。

(3)根据增量表达式,计算出控制变量所得到的新值。

(4)自动转到第(2)步。

三、语句格式举例

(1)将控制变量从 1 变到 100,增量为 1

```
for(i=1;i<=100;++i)
```

(2)将控制变量从 100 变到 1,增量为-1

```
for(i=100;i>=1;--i)
```

(3)控制变量从 7 变到 77,增量为 7

```
for(i=7;i<=77;i+=7)
```

(4)控制变量从 20 变到 2,增量为-2

```
for(int i=20;i>=2;i-=2)
```

(5)按所示数列改变控制变量值:99、88、77、66、55、44、33、22、11、0,增量为-11

```
for(int j=99;j>=0;j-=11)
```

(6)控制变量 i 和 j 共同进行循环控制,i 从 1 变到 99,j 从 2 变到 100,增量均为 2

```
for(int i=1,j=2;i<=99&&j<=100;i+=2,j+=2)
```

需要说明的是:可以在 for 循环"控制变量初始化语句"中声明变量(如上面最后 3 个例子),这些变量只在 for 循环结构中有效,离开了该 for 结构,变量就无效了。

例 4.1　利用 for 循环,计算输出 1+2+…+100 的和。

```
#include <iostream>
using namespace std;
int main ()
{
  int sum=0;
  for (int i=1; i<=100 ;++i)    //i 初始值为 1,终值为 100,每次增量为 1
      sum+=i;
  cout << sum;
  return 0;
}
```

例 4.2　输出 1～100 之间所有偶数。

【方法 1】可以想到对于 1～100 之间的 100 个数字 i,直接重复进行判断,如果 i 是偶数,则输出 i 的值。

程序如下:

```
#include <iostream>
using namespace std;
int main()
{
  for (int i=1;i<=100;i++)  //对于 i 取 1 至 100 之间的每一个整数,都重复操作
    if (i%2==0) cout<<i<<"  ";  //如果 i 为偶数,则输出 i 的值
  return 0;
}
```

程序中,将 1～100 之间的所有数字都列举出来,然后一一判断,符合偶数条件的,就输出。这种思想,本质上是穷举。穷举法保证在求解的过程中,所有可能解都会判断到,不会丢解。当然缺点就是有时候效率不高。

【方法2】在上述分析的基础上,再进一步分析:我们都知道,相邻偶数之间的差值为2,所以,我们还可以设置变量的初值为2,增量为2的for循环,使得循环次数减少为50次。

程序如下:

```
#include <iostream>
using namespace std;
int main()
{
  for (int i=2; i<=100 ; i+=2) //i初始值为2,终值为100,每次增量为2
    cout << i << "  ";          //输出i的值
  return 0;
}
```

例4.3 利用for循环,分别计算1—100中奇数的和、偶数的和。

【方法1】根据例4.2的分析,很容易找到所有的偶数和奇数,继而计算其和。假设用变量sum1和sum2分别存放偶数与奇数和,累加就是在sum1或sum2的基础上,加上一个数字,改变累加变量的值;再加上一个数字,改变累加变量的值;……;如此重复下去。

程序如下:

```
#include <iostream>
using namespace std;
int main()
{
  int i,sum1=0,sum2=0;      //sum1、sum2分别存放偶数和、奇数和,均初始化为0
  for (i=1; i<=100 ; i++) //对于i取1至100之间的每一个整数,都重复操作
    if (i%2==0) sum1+=i;                    //偶数累加到sum1中
    else sum2+=i;                           //奇数累加到sum2中
  cout<<sum1<<"  "<<sum2;                   //输出偶数和、奇数和
  return 0;
}
```

【方法2】偶数从2开始每次递增2,奇数从1开始每次递增2,for语句的循环变量初始化和循环变量增量两部分都可以使用逗号语句序列。

程序如下:

```
#include <iostream>
using namespace std;
int main()
{
  int i,j,sum1=0,sum2=0;
  for (i=2,j=1; i<=100; i+=2,j+=2)     //同时生成偶数和奇数的初始值
  {
    sum1+=i;                            //偶数i累加到sum1中
    sum2+=j;                            //奇数j累加到sum2中
}
```

```
    cout<<sum1<<"  "<<sum2;                    //输出偶数和、奇数和
    return 0;
}
```

例 4.4　利用 for 循环计算 n! 的值。

分析：n! =1 * 2 * 3 * … * n

```
#include <cstdio>
using namespace std;
int main ()
{
    long long s=1;                    //Noip2010 开始 C++语言 long long 允许使用
    int n;                            //n 不能定义为 long long,否则 for 语句死循环
    scanf("%d",&n);
    for (int i=1; i<=n ;++i) //若 s 定义为 int,当 n=13 时 s 的值就溢出了
      s * =i;
    printf("%lld\n",s);               //低版本也可使用 printf("%I64d\n",s)
    return 0;
}
```

【说明】

当 n>=13 时，s 值超过了 int 类型的表示范围。还有一种比 int 更大的类型，称为 long long，它的表示范围是 $-2^{63}\sim2^{63}-1$，比 $-10^{19}\sim10^{19}$ 略窄，而我们一直使用的 int 范围是 $-2^{31}\sim2^{31}-1$，只比 $-2*10^{9}\sim2*10^{9}$ 略宽。

输入输出 long long 也可以借助于 printf 和 scanf 语句，但对应的占位符却是和平台与编译器相关的：在 linux 中，gcc 很统一的用%lld；在 windows 中，MinGW 的 gcc 和 VC6 可用%I64d；高版本编译器下 windows 可以使用%lld。

【上机练习】

1. 求平均年龄【1.5 编程基础之循环控制 01】

2. 均值【1.5 编程基础之循环控制 03】

3. 求整数的和与均值【1.5 编程基础之循环控制 04】

4. 最高的分数【1.5 编程基础之循环控制 05】

5. 最大跨度值【1.5 编程基础之循环控制 06】

6. 奥运奖牌计数【1.5 编程基础之循环控制 07】

7. 奇数求和【1.5 编程基础之循环控制 09】

8. 满足条件的数累加【1.5 编程基础之循环控制 10】

9. 整数的个数【1.5 编程基础之循环控制 11】

10. 与指定数字相同的数的个数【1.5 编程基础之循环控制 12】

11. 乘方计算【1.5 编程基础之循环控制 13】

12. 人口增长【1.5 编程基础之循环控制 14】

13. 菲波那契数【1.5 编程基础之循环控制 17】

14. 鸡尾酒疗法【1.5 编程基础之循环控制 18】

15. 救援【1.5 编程基础之循环控制 19】

16. 津津的储蓄计划【1.5 编程基础之循环控制 22】Noip2012 提高组第 1 题

17. 药房管理【1.5 编程基础之循环控制 23】

18. 正常血压【1.5 编程基础之循环控制 24】

19. 统计满足条件的 4 位数【1.5 编程基础之循环控制 26】

20. 求分数序列和【1.5 编程基础之循环控制 32】

21. 计算分数加减表达式的值【1.5 编程基础之循环控制 33】

22. 余数相同问题【小学奥数 7647】

23. 分苹果【小学奥数 7826】

24. 求小数的某一位【小学奥数 7830】

25. 计算星期几【小学奥数 7831】

26. 幂的末尾【小学奥数 7833】

第二节　while 语句

一、语句格式

格式 1：

```
while (条件表达式)
  语句 1;
```

说明：语句 1 是 while 循环语句的循环体，它将在满足条件的情况下被重复执行。

格式 2：

```
while (条件表达式)
  {语句 1;
   语句 2;
   …
  }
```

说明：循环体部分由多个语句构成，应由一对花括号括起来，构成一个语句块的形式。

程序风格提示：写 while 循环语句时，循环体的语句相对于 while 缩进两格。

二、语句执行过程

(1)计算作为循环控制条件表达式的值，得到逻辑真或假，假定用 M 表示。

(2)若 M 为真，则执行了一遍循环体，否则离开循环，结束整个 while 语句的执行。

(3)循环体的所有语句执行结束后，自动转向第(1)步执行。

三、格式举例

```
(1)i=0;
   while (i<10)
     ++i;
```

功能：当 i 的值小于 10，重复执行++i 语句

```
(2)while(cin>>x,x<0)        //相当于 cin>>x; while (x<0) cin>>x;
```

功能：当输入的数据小于 0 时，重复读数据。

注：while 的括号中可以包含多个语句(中间用逗号隔开)，但是只判读最后一个语句是否为真，如：while (x>10,x==10,x<10)　如果 x<10，则继续执行循环，否则退出循环。

例 4.5　求 s=1+2+3+…+n，当加到第几项时，s 的值会超过 1000?

程序如下：

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

```
int main ()
{
   int n=0,s=0;
   while (s<=1000)
     {
       ++n;
       s+=n;
     }
   cout<<n;
   return 0;
}
```

例 4.6 求两个正整数 m、n 的最大公约数。

【方法 1】:求任意两个自然数 m 和 n 的最大公约数,可以想到其最大的可能就是两个数中的较小者 min,最小可能是 1。所以,可以设最大公约数 gcd 从 min 开始进行判断,若 gcd>1并且没有同时整除 m 和 n,那么就 gcd−1,重复判断是否整除。

【参考程序】

```
#include<iostream>
using namespace std;
int main()
{
    int m,n,gcd;
    cin>>m>>n;
    gcd=m>n? n : m;          //注意此处的特殊写法
    while (gcd>1&&(m%gcd! =0||n%gcd! =0))
    gcd--;                   //每次减 1 寻找最大公约数
    cout<<gcd<<endl;         //输出最大公约数
    return 0;
}
```

【方法 2】:求两个整数的最大公约数可以采用辗转相除法即欧几里德算法。对于任意两个自然数 m 和 n,用 m,n,r 分别表示被除数、除数、余数,那么 m 和 n 的最大公约数等于 n 和 r 的最大公约数。以下是辗转相除法的算法:

①求 m 除以 n 的余数 r;

②当 r! =0,执行第③步;若 r==0,则 n 为最大公约数,算法结束。

③将 n 的值赋给 m,将 r 的值赋给 n;再求 m 除以 n 的余数 r。

④转到第②步。

【参考程序】

```
#include <iostream>
using namespace std;
int main ()
{
```

```
    int m,n;
    cin>>m>>n;
    int r =m % n;
    while (r! =0)                    //也可以使用 while (r),c++中非 0 即真
    {
      m=n;
      n=r;
      r=m % n;
    }
    cout<<"最大公约数="<<n<<endl;
    return 0;
}
```

例 4.7　编一程序求满足不等式 1+1/2+1/3+…+1/n >=5 的最小 n 值。

【分析】　此题不等式的左边是一个求和的算式，该和式中的数据项个数是未知的，也正是要求出的。对于和式中的每个数据项，对应的通式为 1/i，i=1，2，…，n。

所以可采用循环累加的方法来计算出它的值。设循环变量为 i，它应从 1 开始取值，每次增加 1，直到和式的值不小于 5 为止，此时的 i 值就是所求的 n。设累加变量为 s，在循环体内把 1/i 的值累加到 s 上。

根据以上分析，采用 while 循环编写出程序如下：

```
#include <iostream>
using namespace std;
int main ()
{
  int i=0;
  float s=0;
  while(s<5)                         //当 s 的值还未超过 5 时
    {
      ++i;
      s+=1.0/i;
    }
  cout<<i;
  return 0;
}
```

若采用 for 循环来写，则如下所示：

```
#include <iostream>
using namespace std;
int main ()
{
  int i;
  float s=0;
```

```
    for(i=1;s<5;++i)
      s+=1.0/i;
    cout<<i-1;
    return 0;
}
```

例 4.8 数据统计

输入一些整数,求出它们的最小值、最大值和平均值(保留 3 位小数)。输入保证这些数都是不超过 1000 的整数。

输入样例:2 8 3 5 1 7 3 6

输出样例:1 8 4.375

【参考程序】

```
#include<cstdio>
int main()
{
    int x,n=0,min,max,s=0;
    while (scanf("%d",&x)==1)
    {
      s+=x;
      min=x<min? x: min;
      max=x>max? x: max;
      ++n;
    }
    printf("%d %d %.3lf\n",min,max,(double)s/n);
                          //将 s 转换成 double 类型后进行运算
    return 0;
}
```

【分析】

如果是先输入整数 n,然后输入 n 个整数,相信读者能够写出程序来。关键在于:整数的个数是不确定的。scanf 函数有返回值?它返回的是成功输入的变量个数,当输入结束时,scanf 无法再次读取 x,将返回 EOF。

在 Windows 下,输入完毕后先按 Enter 键,再按 Ctrl+Z 键,最后再按 Enter 键,即可结束输入。在 Linux 下,输入完毕后按 Ctrl+D 键即可结束输入。

好了,输入终于结束了,可是输出让我们大失所望:1 2630976 4.375。这个 2630976 是从哪里来的?这个程序的运行结果每次是不确定的,不难验证下面的结论:变量 max 在一开始就等于 2630976(或其他值),自然无法更新为比它小的 8。变量在未赋值之前的值是不确定的,特别地,它不一定等于 0。

解决的方法就很清楚了:在使用之前赋初值。由于 min 保存的是最小值,它的初值应该是一个很大的数;反过来,max 的初值应该是一个很小的数。一种方法是定义一个很大的常数,如 INF=1000000000,然后让 max=-INF,而 min=INF,而另一种方法是先读取第一个整数 x,然后 max=min=x。这样的好处是避免了人为的"假想无穷大"值,程序更

加优美；而 INF 这样的常数有时还会引起其他问题，如“无限大不够大”，或者“运算溢出”。

【优化程序】

```
#include<cstdio>
#define INF 100000000
int main()
{
    int x,n=0,min=INF,max=-INF,s=0;
    while (scanf("%d",&x)==1)
                    //scanf("%d",&x)!=EOF,如果没数据可读,scanf 返回 EOF
    {
      s+=x;
      min=x<min? x: min;
      max=x>max? x: max;
      ++n;
  }
  printf("%d %d %.3lf\n",min,max,(double)s/n);
  return 0;
}
```

最后，我们来更仔细地研究一下输入输出。研究对象就是经典的“A+B”问题：输入若干对整数，输出每对之和。假设每个整数不超过 10^9，一共不超过 10^6 个数对。

第 1 种方法是：

```
#include<cstdio>
int main()
{
    int a,b;
    while(scanf("%d%d",&a,&b)==2) printf("%d\n",a+b);
    return 0;
}
```

第 2 种方法也许更加常用(你再也不用记住%d、%lf 等恼人的占位符了)：

```
#include<iostream>
using namespace std;
int main()
{
    int a,b;
    while(cin >> a >> b) cout << a+b <<endl;
    return 0;
}
```

【上机练习】

1. 球弹跳高度的计算【1.5 编程基础之循环控制 20】

一球从某一高度 h 落下(单位米),每次落地后反跳回原来高度的一半,再落下。编程计算气球在第 10 次落地时,共经过多少米?第 10 次反弹多高?

输出包含两行,第 1 行:到球第 10 次落地时,一共经过的米数。第 2 行:第 10 次弹跳的高度。

2. 角谷猜想【1.5 编程基础之循环控制 21】

所谓角谷猜想,是指对于任意一个正整数,如果是奇数,则乘 3 加 1,如果是偶数,则除以 2,得到的结果再按照上述规则重复处理,最终总能够得到 1。如,假定初始整数为 5,计算过程分别为 16、8、4、2、1。程序要求输入一个整数,将经过处理得到 1 的过程输出来。最后一行输出" End ",如果输入为 1,直接输出" End "。

3. 级数求和【1.5 编程基础之循环控制 27】Noip2002 普及组第 1 题

已知:Sn= 1+1/2+1/3+…+1/n。显然对于任意一个整数 K,当 n 足够大的时候,Sn 大于 K。现给出一个整数 K(1<=k<=15),要求计算出一个最小的 n,使得 Sn>K。

4. 分离整数的各个数【1.5 编程基础之循环控制 28】

给定一个整数 n(1<=n<=100000000),要求从个位开始分离出它的每一位数字。从个位开始按照从低位到高位的顺序依次输出每一位数字。

5. 数字反转【1.5 编程基础之循环控制 29】Noip2011 普及组第 1 题

给定一个整数,请将该数各个位上数字反转得到一个新数。新数也应满足整数的常见形式,即除非给定的原数为零,否则反转后得到的新数的最高位数字不应为零,例如输入-380,反转后得到的新数为-83。

6. 含 k 个 3 的数【1.5 编程基础之循环控制 30】

输入两个正整数 m 和 k,其中 1<m<100000,1<k<5 ,判断 m 能否被 19 整除,且恰好含有 k 个 3,如果满足条件,则输出 YES,否则,输出 NO。例如,输入:43833 3,满足条件,输出 YES。如果输入:39331 3,尽管有 3 个 3,但不能被 19 整除,也不满足条件,应输出 NO。

第三节　do—while 语句

一、语句格式

格式 1：

```
do
  语句 1;
while (条件表达式);
```

说明：语句 1 是 do—while 的循环体。

格式 2：

```
do
  {
    语句 1;
    语句 2;
    …
  }
while (条件表达式);
```

说明：循环体部分由多个语句构成，应由一对花括号括起来，构成一个语句块的形式。

二、语句执行过程

(1)执行一遍循环体。

(2)求出作为循环条件的“条件表达式”的值，若为逻辑值真，则自动转向第(1)步，否则结束 do 循环的执行过程，继续执行其后面的语句。

在 do 语句的循环体中也可以使用 break 语句，用它来非正常结束循环的执行。

三、实例

例 4.9　对于求两个正整数 m，n 的最大公约数可以用 do—while 实现。

代码如下，请完善：

```
#include <iostream>
using namespace std;
int main ()
{
    int m,n,r;
    cin>>m>>n;
    do                          //辗转相除法
```

```
    {
      r=m % n;
      m=____;
      n=____;
    }
    while( _____);
  cout<<" the greatest common divisor is:"<<_____;
  return 0;
  }
```

例 4.10 求 1992 个 1992 的乘积的末两位数是多少?

【分析】 积的个位与十位数只与被乘数和乘数的个位与十位数字有关,所以本题相当于求 1992 个 92 相乘,而且本次的乘积是下一次相乘的被乘数,因此也只需取末两位参与运算就可以了。

```
#include<iostream>
using namespace std;
int main()
{
    int a=1,t=0;
    do
    {
        ++t;
        a=(a*92)%100;
    }while (t!=1992);
    cout<<a<<endl;
    return 0;
}
```

例 4.11 校体操队到操场集合,排成每行 2 人,最后多出 1 人;排成每行 3 人,也多出 1 人;分别按每行排 4、5、6 人,都多出 1 人;当排成每行 7 人时,正好不多。求校体操队至少多少人?

【分析】 ①设校体操队为 x 人,根据题意 x 应是 7 的倍数,因此 x 的初值为 7,以后用 x+=7改变 x 值;

②为了控制循环,用逻辑变量 yes 为真(true) 使循环结束;

③如果诸条件中有一个不满足,yes 的值就会为假(false),就继续循环。

```
#include<iostream>
using namespace std;
int main()
{
    int x=0;
    do
    {
```

```
        bool yes=true;
        x+=7;
        if (x%2!=1) yes=false;
        if (x%3!=1) yes=false;
        if (x%4!=1) yes=false;
        if (x%5!=1) yes=false;
        if (x%6!=1) yes=false;
    }while (yes==false);              //直到 yes 的值为真
    cout<<" All="<<x;
    return 0;
}
```

程序中对每个 x 值，都先给 yes 赋真值，只有在循环体各句对 x 进行判断时，都得到“通过”（此处不赋假值）才能保持真值。

【上机练习】

1. 球弹跳高度的计算【1.5 编程基础之循环控制 20】

一球从某一高度 h 落下（单位米），每次落地后反跳回原来高度的一半，再落下。编程计算气球在第 10 次落地时，共经过多少米？第 10 次反弹多高？

输出包含两行，第 1 行：到球第 10 次落地时，一共经过的米数。第 2 行：第 10 次弹跳的高度。

2. 角谷猜想【1.5 编程基础之循环控制 21】

所谓角谷猜想，是指对于任意一个正整数，如果是奇数，则乘 3 加 1，如果是偶数，则除以 2，得到的结果再按照上述规则重复处理，最终总能够得到 1。如，假定初始整数为 5，计算过程分别为 16、8、4、2、1。程序要求输入一个整数，将经过处理得到 1 的过程输出来。最后一行输出“End”，如果输入为 1，直接输出“End”。

3. 级数求和【1.5 编程基础之循环控制 27】Noip2002 普及组第 1 题

已知：Sn= 1+1/2+1/3+…+1/n。显然对于任意一个整数 K，当 n 足够大的时候，Sn 大于 K。现给出一个整数 K(1<=k<=15)，要求计算出一个最小的 n，使得 Sn>K。

4. 分离整数的各个数【1.5 编程基础之循环控制 28】

给定一个整数 n(1<=n<=100000000)，要求从个位开始分离出它的每一位数字。从个位开始按照从低位到高位的顺序依次输出每一位数字。

5. 数字反转【1.5 编程基础之循环控制 29】Noip2011 普及组第 1 题

给定一个整数，请将该数各个位上数字反转得到一个新数。新数也应满足整数的常见形式，即除非给定的原数为零，否则反转后得到的新数的最高位数字不应为零，例如输入－380，反转后得到的新数为－83。

6. 含 k 个 3 的数【1.5 编程基础之循环控制 30】

输入两个正整数 m 和 k，其中 1<m<100000，1<k<5 ，判断 m 能否被 19 整除，且恰好含有 k 个 3，如果满足条件，则输出 YES，否则，输出 NO。例如，输入：43833 3，满足条件，输出 YES。如果输入：39331 3，尽管有 3 个 3，但不能被 19 整除，也不满足条件，应输出 NO。

第四节　循环嵌套

例 4.12　求 s=1!+2!+3!+…+10!

【分析】　这个问题是求 10 以内自然数的阶乘之和,可以用 for 循环来实现。程序结构如下:

```
for(int i=1;i<=10;++i)
{  (1)i 阶乘的值存到 t;        //t=i!
   (2)累加 t 到 s 中;          //s+=t
}
```

显然根据以上结构,通过 10 次的循环可以求出 1!,2!,…,10!,并不断累加起来,求出 s。而求 t=i!,又可以用一个 for 循环来实现:

```
t=1;
for (int j=1;j<=i;++j)
  t*=j;
```

因此整个程序为:

```
#include <iostream>
using namespace std;
int main ()
{  int s=0;
   for(int i=1;i<=10;++i)
   {  int t=1;
      for (int j=1;j<=i;++j)          //求 i!
          t*=j;
      s+=t;                           //累加 i!
   }
   cout<<s;
   return 0;
}
```

以上程序是一个 for 循环的嵌套。这种方法是比较容易想到的,但实际上对于求 i!,我们可以根据求出的(i-1)! 乘上 i 即可得到,而无需重新从 1 再累乘到 i。

因此程序可改为:

```
#include <iostream>
using namespace std;
int main ()
{  int t=1,s=0;
   for(int i=1;i<=10;++i)
   {
```

```
        t*=i;                    //t为上一个数的i-1的阶乘值,再乘以i即为i!
        s+=t;                    //累加i!
    }
    cout<<s;
    return 0;
}
```

显然第二个程序的效率要比第一个高得多。第一个程序要进行1+2+3+…+10=55次循环,而第二程序进行10次循环。若题目中求的是1!+2!+…+1000!,则两个程序的效率区别更明显。

例4.13　对于给定的自然数n(n<20),在屏幕上输出仅由"*"构成的n行的直角三角形。

例如:当n=5时,输出:

```
*
* *
* * *
* * * *
* * * * *
```

【分析】打印图形总是逐行进行的,本题要重复n行操作,对于每一行,又重复若干次输出"*"操作。于是,构成了一个两层循环:外层循环是1至n行的处理,而内层循环,则是输出同一行上的每一列。分析样例,不难发现,每一行上"*"的个数恰好是行数。因此对于第i行,内层循环可以设置重复i次。

程序如下:

```
#include <iostream>
using namespace std;
int main ()
{
  int i,j,n;
  cin>>n;
  for (i=1; i<=n; ++i)      //外层循环,控制行数
  {
    for (j=1; j<=i; ++j)    //内层循环,输出一行中的*数
    cout<<"*";
    cout<<endl;             //每行最后要换行
  }
  return 0;
}
```

例4.14　百钱买百鸡问题。鸡翁一,值钱五,鸡母一,值钱三,鸡雏三,值钱一,百钱买百鸡,问鸡翁、鸡母、鸡雏各几何?

【方法1】在数学中解决这个问题,我们通常会列出一个方程组,设鸡翁x,鸡母y,鸡雏z,则:

x+y+z=100

5*x+3*y+z/3=100

同时满足上述两个方程的x、y、z值就是所求。

根据这个思路,问题就转化为求解方程组,我们列举x、y、z的所有可能解,然后判断这些可能解是否能使方程组成立。能使方程组成立的,就是直正的解。

再进一步分析,x的取值范围是1～100/5,y的取值范围是1～100/3,z的取值范围是1～3*100。

程序如下:

```
#include <iostream>
using namespace std;
int main()
{
  int x,y,z;
  for (x=0;x<=100/5;x++)              //列举鸡翁数的所有可能
    for (y=0;y<=100/3;y++)            //列举鸡母数的所有可能
      for (z=0;z<=3*100;z++)          //列举鸡雏数的所有可能
        if(5*x+3*y+z/3==100 && x+y+z==100)     //满足两个方程组
          cout<<x<<" "<<y<<" "<<z<<endl;       //输出x、y、z值
  return 0;
}
```

运行结果:

```
0 25 75
3 20 77
4 18 78
7 13 80
8 11 81
11 6 83
12 4 84
```

【说明】这里用了一个三层循环的程序解决问题。当x取得一个数值时,for的y循环体都要执行遍y的所有取值;当y取得一个数值时,for的z循环体都要执行遍z的所有取值;对于z的每一个取值,if语句都要执行一次。

不难算出,在程序的执行过程中,作为最内层循环体的if语句,将被执行:(1+100/5)*(1+100/3)*(1+3*100)=214914次。而观察程序的运行结果,问题的解远远小于这个数字,只有7组解。如何减少循环次数呢?

【方法2】由于题目的特殊性,鸡翁、鸡母、鸡雏共100只,一旦确定鸡翁x和鸡母y的数量,鸡雏便只能购买100-x-y只。这样,我们可以尝试写出一个两层循环的程序,解决这个问题。

程序如下:

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

```
int main()
{
  int x,y,z;
  for (x=0;x<=100/5;x++)          //列举鸡翁数的所有可能
  for (y=0;y<=100/3;y++)          //列举鸡母数的所有可能
  {
    z=100-x-y;                    //根据 x,y 计算鸡雏的数量
    if(5 * x+3 * y+z/3==100)      //判断总钱数是否符合条件
    cout<<x<<" "<<y<<" "<<z<<endl;        //输出 x、y、z 值
    }
  return 0;
}
```

【说明】对于与本题类似的求解不定方程的问题，都可以用循环来求解。为了提高效率，可以在程序中进行适当优化，减少循环体的执行次数。

例 4.15　求 100～999 中的水仙花数。若三位数 ABC，ABC＝$A^3+B^3+C^3$，则称 ABC 为水仙花数。例如 153，$1^3+5^3+3^3=1+125+27=153$，则 153 是水仙花数。

【分析】　根据题意，采用三重循环来求解。由于循环次数一定，用 for 循环最为简单。程序如下：

```
#include<iostream>
#include<iomanip>              //调用 setw 函数需注明使用该库
using namespace std;
int main()
{
  for (int a=1; a<=9;++a)
    for (int b=0; b<=9;++b)
        for (int c=0; c<=9;++c)
          {
            if (a * a * a+b * b * b+c * c * c==a * 100+b * 10+c)
            cout<<setw(6)<<a * 100+b * 10+c;   //setw 函数控制输出场宽
          }
  return 0;
}
```

运行结果：

```
153 370 371 407
```

同时也可以采用一个 for 循环来求解，表面上看好像优于三重循环，实际上却比上面的程序效率低，请同学们自己分析。

程序如下：

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

```
int main()
{
  for (int m=100; m<=999;++m)
  {
      int a=m/100;                          //m 的百位
      int b=(m%100)/10;                     //m 的十位
      int c=m%10;                           //m 的个位
      if (a*a*a+b*b*b+c*c*c==m)
        cout<<setw(6)<<m;
  }
return 0;
}
```

例 4.16 输出 100～200 中所有的素数。

【分析】 我们可对 100～200 之间的每一个整数进行判断,若它是素数,则输出。而对于任意整数 i,根据素数定义,我们从 2 开始,到 sqrt(i),找 i 的第一个约数,若找到第一个约数,则 i 必然不是素数。

程序如下:

```
#include <iostream>
#include<cmath>              //在 Dev C++中可调用数学函数库 cmath
using namespace std;
int main ()
{
  for (int i=100;i<=200;++i)
  {  int x=2;
     while(x<=floor(sqrt(i))&&(i%x!=0))   //floor 为取整函数,需调用 cmath 库
       ++x;                  //在枚举的范围内并且没有出现约数则继续枚举
     if(x>floor(sqrt(i)))
       cout<<i<<"\t ";
  }
  return 0;
}
```

例 4.17 输出所有形如 aabb 的四位完全平方数(即前两位数字相等,后两位数字也相等)。

【分析】 分支和循环结合在一起时威力特别强大:我们枚举所有可能的 aabb,然后判断它们是否为完全平方数。注意:a 的范围是 1～9,b 可以是 0。主程序如下:

```
for (int a=1; a<=9; a++)
  for (int b=0; b<=9; b++)
    if (aabb 是完全平方数) printf("%d\n ",aabb);
```

另一个思路是枚举平方根 x,参考程序如下:

```
#include<cstdio>
```

```
int main()
{
    for (int x=1 ;  ;++x)          //可以直接从 x=32 开始枚举
    {
      int n=x * x;
      if (n<1000) continue;
      if (n>9999) break;
      int hi=n/100,lo=n%100;
      if (hi/10==hi%10 && lo/10==lo%10) printf("%d\n ",n);
    }
    return 0;
}
```

此程序中的新东西是 continue 和 break 语句。continue 是指跳回 for 循环的开始，执行调整语句并判断循环条件，就是“直接进行下一次循环”，而 break 是指直接跳出循环。

另外，注意到这里的 for 语句是“残缺”的：没有指定循环条件。事实上，3 个部分都是可以省略的。没错，for(;;)就是一个死循环——如果不采取措施(如 break)，它就永远不会结束。

例 4.18　把一个合数分解成若干个质因数乘积的形式(即求质因数的过程)叫做分解质因数。分解质因数(也称分解素因数)只针对合数。

输入一个正整数 n，将 n 分解成质因数乘积的形式。

输入样例：

36

输出样例：

36=2 * 2 * 3 * 3

【分析】将任意的 n 分解为质因数的乘积，要从最小的质数开始，那么，我们就不妨从 2 开始试除，能整除就输出 2，再对商继续试除，直到不再含有因子 2；然后用下一个质数反复试除，……，再用下一个质数试除，……，一直到商为 1，停止操作。

这里，质因数的递增，是一层循环，每一个质因数的反复试除，又是一层循环。因此，本题使用两层循环来解决。

程序如下：

```
#include <iostream>
using namespace std;
int main()
{
  int n,i=2;
  cin>>n;
  cout<<n<<"=";
  do
  {
    while(n%i==0)          //n 能被 i 整除，就重复做除法操作
```

```
    {
      cout<<i;
      n/=i;
      if(n! =1) cout<<"*";
    }
    i++;
  }
  while(n! =1);                    //n没有除尽,就重复操作
  return 0;
}
```

例 4.19 阶乘之和

输入 n,计算 s=1!+2!+3!+…+n! 的末 6 位(不含前导 0)。n<=10^6, n! 表示前 n 个正整数之积。

输入样例:10

输出样例:37913

【分析】 这个任务并不难,引入累加变量 s 之后,核心算法只有一句话:for (i=1;i<=n;i++) s+=i!。不过 C++语言并没有阶乘运算符,所以这句话只是伪代码,而不是真正的代码。事实上,我们还需要一次循环来计算 i!:for (j=1;j<=i;++j) factorial*=j;。代码如下:

```
#include<cstdio>
int main()
{
    int n,s=0;
    scanf("%d",&n);
    for (int i=1;i<=n;++i)
    {
        int factorial=1;
        for (int j=1;j<=i;++j)
          factorial*=j;
        s+=factorial;
    }
    printf("%d\n",s%1000000);
    return 0;
}
```

注意累乘器 factorial(英文"阶乘"的意思)定义在循环里面。换句话说,每执行一次循环体,都要重新声明一次 factorial,并初始化为 1(想一想,为什么不是 0)。因为只要末 6 位,所以输出时对 10^6 取模。

当 n=100 时,输出-961703,直觉告诉我们:乘法溢出了。这个直觉很容易通过"输出中间变量"法得到验证,但若要解决这个问题,还需要一点数学知识。试一下 n=10^6 时输出什么?更会溢出,但是重点不在这里。事实上,它的速度太慢!让我们把程序改成"每步取

模"的形式,然后加一个"计时器",看看它到底有多慢。

```
#include<cstdio>
#include<ctime>
int main()
{
    const int MOD=1000000;
    int n,s=0;
    scanf("%d",&n);
    for (int i=1;i<=n;++i)
    {
        int factorial=1;
        for (int j=1;j<=i;++j)
          factorial=(factorial*j%MOD);
        s=(s+factorial)%MOD;
    }
    printf("%d\n",s);
    printf("Time used=%.2lf\n",(double)clock()/CLOCKS_PER_SEC);
    return 0;  //时间包含键盘输入的时间,建议用文件输入输出,后面章节介绍文件
}
```

这个程序真正的特别之处在于计时函数 clock()的使用。该函数返回程序目前为止运行的时间。这样,在程序结束之前调用它,便可获得整个程序的运行时间。这个时间除以常数 CLOCKS_PER_SEC 之后得到的值以"秒"为单位。

输入 100000,按 Enter 键,系统迟迟不输出答案,原因在于程序中重复进行了多次阶乘运算,浪费了大量时间,具体优化方法请参考例 4.12 。

【上机练习】

1. 求阶乘的和【1.5 编程基础之循环控制 34】

给定正整数 n,求不大于 n 的正整数的阶乘的和(即求 1! +2! +3! +…+n!),输出阶乘的和。

2. 求出 e 的值【1.5 编程基础之循环控制 35】

利用公式 e = 1 + 1/1! + 1/2! + 1/3! + … + 1/n! ,求 e 的值,要求保留小数点后 10 位。

3. 计算多项式的值【1.5 编程基础之循环控制 36】

假定多项式的形式为 x^n+x^(n-1)+…+x^2+x+1,请计算给定单精度浮点数 x 和正整数 n 值的情况下这个多项式的值。x 在 float 范围内,n<=1000000。多项式的值精确到小数点后两位,保证最终结果在 float 范围内。

4. 与 7 无关的数【1.5 编程基础之循环控制 39】

一个正整数,如果它能被 7 整除,或者它的十进制表示法中某一位上的数字为 7,则称其为与 7 相关的数。现求所有小于等于 n(n<100)与 7 无关的正整数的平方和。

5. 数 1 的个数【1.5 编程基础之循环控制 40】

给定一个十进制正整数 n(1<=n<=10000),写下从 1 到 n 的所有整数,然后数一下其中出现的数字"1"的个数。

例如当 n=2 时,写下 1,2。这样只出现了 1 个"1";当 n=12 时,写下 1,2,3,4,5,6,7,8,9,10,11,12。这样出现了 5 个"1"。

6. 数字统计【1.5 编程基础之循环控制 41】Noip2010 普及组第 1 题

请统计某个给定范围[L, R]的所有整数中,数字 2 出现的次数。

比如给定范围[2, 22],数字 2 在数 2 中出现了 1 次,在数 12 中出现 1 次,在数 20 中出现 1 次,在数 21 中出现 1 次,在数 22 中出现 2 次,所以数字 2 在该范围内一共出现了 6 次。

7. 画矩形【1.5 编程基础之循环控制 42】

根据参数,画出矩形。输入四个参数:前两个参数为整数,依次代表矩形的高和宽(高不少于 3 行不多于 10 行,宽不少于 5 列不多于 10 列);第三个参数是一个字符,表示用来画图的矩形符号;第四个参数为 1 或 0,0 代表空心,1 代表实心。

样例输入

7 7 @ 0

样例输出

```
@@@@@@@
@     @
@     @
@     @
@     @
@     @
@@@@@@@
```

8. 质因数分解【1.5 编程基础之循环控制 43】Noip2012 普及组第 1 题

已知正整数 n 是两个不同的质数的乘积,试求出较大的那个质数。

9. 第 n 小的质数【1.5 编程基础之循环控制 44】

输入一个正整数 n,求第 n 小的质数。输入一个不超过 10000 的正整数 n,输出第 n 小的质数。

10. 金币【1.5 编程基础之循环控制 45】

国王将金币作为工资,发放给忠诚的骑士。第 1 天,骑士收到一枚金币;之后两天(第 2 天和第 3 天)里,每天收到两枚金币;之后三天(第 4、5、6 天)里,每天收到三枚金币;之后四天(第 7、8、9、10 天)里,每天收到四枚金币……这种工资发放模式会一直这样延续下去:当连续 n 天每天收到 n 枚金币后,骑士会在之后的连续 n+1 天里,每天收到 n+1 枚金币(n 为任意正整数)。

你需要编写一个程序,确定从第一天开始的给定天数内,骑士一共获得了多少金币。

11. 不定方程求解【小学奥数 7650】

给定正整数 a,b,c。求不定方程 ax+by=c 关于未知数 x 和 y 的所有非负整数解组数。

输入:

一行,包含三个正整数 a,b,c,两个整数之间用单个空格隔开。每个数均不大于 1000。

输出:

一个整数,即不定方程的非负整数解组数。

第五章　数　组

第一节　一维数组

一、为什么要使用数组

通过前面几章的学习，我们已经可以编写程序来解决各种相当复杂的问题了，但是当需要处理的数据比较多时，仅依靠前面的知识是不够的，即使简单的问题也可能需要比较复杂的程序来处理。请看下面的例子：

例题：输入50个学生的某门课程的成绩，打印出低于平均分的学生序号与成绩。

【分析】 在解决这个问题时，虽然可以通过一个变量来累加读入的50个成绩求出学生的总分，进而求出平均分。但因为只有读入最后一个学生的分数后才能求得平均分，并且要求打印出低于平均分的学生序号和成绩，故必须把50个学生的成绩都保留起来，然后逐个和平均分比较，把低于平均分的成绩打印出来。如果，用简单变量a1，a2，…，a50存储这些数据，要用50个变量保存输入的数据，程序片断如下：

```
cin>>a1>>a2>>…>>a10;
…
cin>>a41>>a42>>…>>a50;
```

注意，如果真正要像上面这样编写程序，则上面的所有省略号必须用完整的语句写出来。可以看出，这样的程序是多么繁琐。如果说处理的数据规模达到成千上万，上面的例子单单读入就会异常复杂，电脑的优势没有得到体现。

从以上的讨论可以看出，如果只使用简单变量处理大量数据，就必须使用大量只能单独处理的变量，即使是简单问题也需要编写冗长的程序。

选手们可能已经看出，我们需要把一大批具有相同性质的数据组合成一个新类型的变量，可以用简单的程序（比如循环50次）对这个新变量的各个分量进行相同的处理，每个分量仍然保留单个变量的所有性质（在上面的例子中，各分量是整型变量或实型变量的性质）。

如果能像数学中使用下标变量 a_i 形式表示这50个数，则问题就容易实现。在C++语言中，具有下标性质的数据类型是数组。如果使用数组，上面的问题就变得十分简单、清晰。例如，读入50个学生的成绩，只需写如下语句即可：

```
for (int i=1;i<=50;++i)
    cin>>a[i];
```

在这里引用了带下标的变量（分量变量称为数组元素）a[i]来代替 a_1，a_2，…，a_{50}，方括号中的i称为下标，当循环变量i=1时a[i]就是a[1]；当i=2时a[i]就是a[2]；…；当i=50时

a[i]就是a[50]。输入的时候,让i从1变化到50,循环体内输入语句中的a[i]也就分别代表了a_1,a_2…,a_{50}这50个带下标的变量。这样上述问题的程序可写为:

```
int tot=0;                          // tot存储50个学生的总分
for (int i=1;i<=50;++i)             // 循环读入每一个学生的成绩,并把它累加到总分中
{
  cin>>a[i];
  tot+=a[i];
}
int ave=tot/50;                     //计算平均分
for (int i=1;i<=50;++i)
  if (a[i]<ave) cout<<" No. "<<i<<" "<<a[i];
                    //如果第i个同学成绩小于平均分,则将输出这个学生的序号和成绩
```

要在程序中使用下标变量,必须先说明这些下标变量的整体为数组,即数组是若干个同名(如上面的下标变量的名字都为a)下标变量的集合,这些变量的类型全部一致。

二、一维数组的定义

当数组中每个元素只带有一个下标时,我们称这样的数组为一维数组。

数组的定义格式如下:

类型标识符　数组名[常量表达式]

说明:

①数组名的命名规则与变量名的命名规则一致。

②常量表达式表示数组元素的个数。可以是常量和符号常量,但不能是变量。

例如:

```
int a[10];                          //数组a定义是合法的
int b[n];                           //数组b定义是非法的
```

其中,a是一维数组的数组名,该数组有10个元素,依次表示为:a[0],a[1],a[2],a[3],a[4],a[5],a[6],a[7],a[8],a[9]。需要注意的是:a[10]不属于该数组的空间范围。当在说明部分定义了一个数组变量之后,C++编译程序为所定义的数组在内存空间开辟一串连续的存储单元。例如:上例中的a数组在内存的存储如下表所示:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

a数组共有10个元素组成,在内存中10个数组元素共占10个连续的存储单元。a数组最小下标为0,最大下标9。按定义a数组所有元素都是整型变量。

三、一组数组的引用

通过给出的数组名称和这个元素在数组中的位置编号(即下标),程序可以引用这个数组中的任何一个元素。

一维数组元素的引用格式:

数组名[下标]

例如:若i,j都是int型变量,则:

a[5]

a[i+j]

a[i++]

都是合法的元素。

说明：

(1)下标可以是任意值为整型的表达式，该表达式里可以包含变量和函数调用。引用时，下标值应在数组定义的下标值范围内。

(2)数组的精妙在于下标可以是变量，通过对下标变量值的灵活控制，达到灵活处理数组元素的目的。

(3)C++语言只能逐个引用数组元素，而不能一次引用整个数组。

例如：int a[100],b[100];a=b;这样的写法是非法的。

(4)数组元素可以像同类型的普通变量那样使用，对其进行赋值和运算的操作，和普通变量完全相同。

例如：c[10]=34;实现了给 c[10]赋值为 34。

四、一维数组的初始化

数组的初始化可以在定义时一并完成。格式：

类型标识符　数组名[常量表达式]={值 1,值 2,…}

例如：

int a[5]={1,2,3,4,5}

说明：

(1)在初值列表中可以写出全部数组元素的值，也可以写出部分。例如，以下方式可以对数组进行初始化：

int x[10]={0,1,2,3,4};

该方法一次仅对数组的前 5 个元素依次进行初始化。

(2)对数组元素全部初始化为 0，可以简写为：{0}。

例如：

int a[5]={0}；将数组 a 的 5 个元素都初始化为 0。

下面两程序没有初始化数组，观察程序默认的数组变量初值。

程序 1：

```
#include<iostream>
using namespace std;
int a[5];
int main()
{
  for (int i=0;i<5;i++)
    cout<<a[i]<<" ";
  return 0;
}
```

运行结果：

0 0 0 0 0

程序 2：

```
#include<iostream>
using namespace std;
int main()
{
  int a[5];
  for (int i=0;i<5;i++)
    cout<<a[i]<<" ";
  return 0;
}
```

运行结果：

2293544 4252834 42 99 59

程序 3：

```
#include<iostream>
using namespace std;
int main()
{
  int a[5]={1,2};
  for (int i=0;i<5;i++)
    cout<<a[i]<<" ";
  return 0;
}
```

运行结果：

1 2 0 0 0

【说明】

程序1、程序2和程序3的区别在于数组定义在int main()之外与之内,程序1中数组定义放在int main()之外,其初始值是0值。程序2中数组定义放在int main()之内,其初始值是随机的。程序3中数组定义放在int main()之内,只给a[0]、a[1]赋初值,但后面的a[2]~a[4]元素自动赋0值。

五、数组越界

C++语言规定,使用数组时,要注意:

(1)数组元素的下标值为正整数。

(2)在定义元素个数的下标范围内使用。

然而,当在程序中把下标写成负数、大于数组元素的个数时,程序编译的时候是不会出错的。例如:

```
int a[10];
a[-3]=5;
a[20]=15;
a[10]=20;
int k=a[30]
```

这些语句的语法是正确的,能够通过程序的编译。然而,它们要访问的数组元素并不在数组的存储空间的,这种现象叫数组越界。例如下面程序:

```
#include<iostream>
using namespace std;
int main()
{
  int a[5];
  for (int i=0;i<=10;i++)
  {
    a[i]=i;
    cout<<a[i]<<" ";
  }
  return 0;
}
```

【说明】

该程序能够通过编译,也能运行出结果,程序的问题是定义a[5],使用时数组下标超过了4。C++语言中,数组越界访问系统时不一定会给出任何的提示,也就是说,程序可以超出数组边界进行读/写,从而造成内存的混乱。

数组越界是实际编程中常见的错误,而且这类错误往往难以捕捉。因为越界语句本身并不一定导致程序立即出错,可能在遇到某些数据时才导致错误,有时由于越界,意外地改变了变量或指令,导致在调试器里调试的时候,程序不按照应当的次序运行的怪现象。

六、一维数组的应用

例 5.1　输入 n 个数，要求程序按输入时的逆序把这 n 个数打印出来，已知整数不超过 100 个。也就是说，按输入相反顺序打印这 n 个数。

【分析】　我们可定义一个数组 a 用以存放输入的 n 个数，然后将数组 a 中的内容逆序输出。

```
#include<cstdio>
int a[100];
int main()
{
    int x,n=0;
    while(scanf("%d ",&x)==1)
      a[n++]=x;              //相当{a[n]=x;n++;}
    for (int i=n-1;i>=1;--i)
      printf("%d ",a[i]);    //注意%d后面有一个空格,保证行首行尾均无空格
    printf("%d\n ",a[0]);
    return 0;
}
```

【说明】

语句 int a[100]声明了一个包含 100 个整型变量的数组，它们是：a[0]，a[1]，a[2]，…，a[99]。注意，没有 a[100]。在上述程序中，数组 a 被声明在 main 函数的外面。只有放在外面时，数组 a 才可以开得很大；放在 main 函数内时，数组稍大就会异常退出。它的道理将在后面讨论，只需要记住规则即可。

数组不能够进行赋值操作：如果声明的是 int a[MAXN]，b[MAXN]，是不能赋值 b=a 的(Pascal 语言可以的)。如果要从数组 a 复制 k 个元素到数组 b，可以这样做：memcpy(b，a，sizeof(int) * k)。当然了，如果数组 a 和 b 都是浮点型的，复制时要写成 memcpy(b，a，sizeof(double) * k)。如果需要把数组 a 全部复制到数组 b 中，可以写得简单一些：memcpy(b，a，sizeof(a))。使用 memcpy 函数要包含头文件 cstring。

例 5.2　将 a 数组中第一个元素移到数组末尾，其余数据依次往前平移一个位置。

【分析】　为完成题目所要求的操作，其算法应该包括以下几个主要步骤：

①把第一个元素的值取出放在一个临时单元 temp 中；

②通过 a[2]→a[1]，a[3]→a[2]，a[4]→a[3]，…，a[n]→a[n−1]，实现其余元素前移；

③将 temp 值送入 a[n]。

```
#include<iostream>
#include<iomanip>              //调用 setw 函数需注明使用该库
const int n=10;
using namespace std;
int a[n];
```

```
int main()
{
    cout<<" read "<<n<<" datas "<<endl;
    for (int i=0; i<n;++i) cin>>a[i];
    int temp=a[0];
    for (int i=0; i<n-1;++i) a[i]=a[i+1];
    a[n-1]=temp;
    cout<<" Result:"<<endl;
    for (int i=0; i<n;++i) cout<<setw(3)<<a[i];  //setw 函数控制输出场宽
    return 0;
}
```

运行结果：

```
read 10 datas:
1 2 3 4 5 6 7 8 9 10
Result:
2 3 4 5 6 7 8 9 10 1
```

例 5.3 宾馆里有 100 个房间，从 1－100 编了号。第一个服务员把所有的房间门都打开了，第二个服务员把所有编号是 2 的倍数的房间作“相反处理”，第三个服务员把所有编号是 3 的倍数的房间作“相反处理”，…，以后每个服务员都是如此。当第 100 个服务员来过后，哪几扇门是打开的？（所谓“相反处理”是：原来开着的门关上，原来关上的门打开。）

【分析】 此题较简单，用 a[1]，a[2]，…，a[n]表示编号为 1，2，3，…，n 的门是否开着。模拟这些操作即可，参考程序如下：

```
#include<cstdio>
#include<cstring>
#define MAXN 100+10
int a[MAXN];
int main()
{
    int n,first=1;
    memset(a,0,sizeof(a));
    for (int i=1;i<=100;++i)
      for (int j=1;j<=100;++j)
        if (j%i==0) a[j]=!a[j];
    for (int i=1;i<=100;++i)
      if (a[i])
      {
        if(first) first=0; else printf(" ");
        printf("%d ",i);  }
    printf("\n ");
    return 0;
```

```
}
```

运行结果：

```
1 4 9 16 25 36 49 64 81 100
```

【说明】

memset(a,0,sizeof(a))的作用是把数组 a 清零，它在 cstring 中定义。虽然也能用 for 循环完成相同的任务，但是用 memset 又方便又快捷。另一个技巧在输出：为了避免输出多余空格，设置了一个标志变量 first，可以表示当前要输出是否为第一个。第一个变量前不应该有空格，但其他都有。

例 5.4　约瑟夫问题：N 个人围成一圈，从第一个人开始报数，数到 M 的人出圈；再由下一个人开始报数，数到 M 的人出圈；…输出依次出圈的人的编号。N，M 由键盘输入。

【分析】（1）由于对于每个人只有出圈和没有圈两种状态，因此可以用布尔型标志数组存储游戏过程中每个人的状态。不妨用 true 表示出圈，false 表示没有出圈。

（2）开始的时候，给标志数组赋初值为 false，即全部在圈内。

（3）模拟报数游戏的过程，直到所有的人出圈为止。

程序如下：

```
#include<iostream>
using namespace std;
bool a[101];                          //根据题意开出数组大小
int main()
{
  int n,m;
  cin>>n>>m;                          //共 n 人，报到 m 出圈
  cout<<endl;
  for (int i=1;i<=n;++i) a[i]=false; //等同于 memset(a,0,sizeof(a))
  int f=0,t=0,s=0;
  do
  {
      ++t;                            //逐个枚举圈中的所有位置
      if (t==n+1) t=1;                //数组模拟环状，最后一个与第一个相连
      if (a[t]==false)++s;            //第 t 个位置上有人则报数
      if (s==m)                       //当前报的数是 m
        {
            s=0;                      //计数器清零
            cout<<t<<" ";             //输出出圈人的编号
            a[t]=true;                //此处的人已出圈，设置为空
            ++f;                      //出圈的人数增加一个
        }
  }while(f!=n);                       //直到所有的人都出圈为止
  return 0;
}
```

运行结果：

输入：8 5

输出：5 2 8 7 1 4 6 3

这是一个在算法设计上很有名气的经典约瑟夫(Josephu)问题，它有很多变例。如猴子选大王、持密码报数、狐狸追兔子等(见上机练习)。

例 5.5 输入 n 个整数，存放在数组 a[1]至 a[n]中，输出最大数所在位置(n<=10000)。

输入样例：

67 43 90 78 32

输出样例：

3

【分析】设 maxa 存放最大值，k 存放对应最大值所在的数组位置，maxa 的初值为 a[1]，k 的初值对应为 1，枚举数组元素，找到比当前 maxa 大的数成为 maxa 的新值，k 值为对应位置，输出最后的 k 值。

程序如下：

```
#include <iostream>
using namespace std;
const int MAXN=10001;
int main()
{
    int a[MAXN];                    //定义 10001 个数组
    int i,n,maxa,k;
    cin>>n;
    for (i=1;i<=n;i++)
      cin>>a[i];                    //读入 n 个数存到 a[1]~a[n]中
    maxa=a[1];k=1;                  //赋最大值初值和初始位置
    for(i=2;i<=n;i++)
      if (a[i]>maxa)                //枚举数组，找到最大数和位置
      {
        maxa=a[i];
        k=i;
      }
    cout<<k;                        //输出最大数所在数组中的位置
    return 0;
}
```

例 5.6 学校推出 10 名歌手，校学生会想知道这 10 名歌手受欢迎的程度，设计一个投票箱，让每一个同学给自己喜欢的歌手投票，为了方便，学生会把 10 名歌手用 1~10 进行编号，这样，同学们只要用编号进行投票。现在，学生会找到你，帮助统计一下每个歌手获得的票数。

输入样例：

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

输出样例：

1 2 3 4 5 6 7 8 9 10

1 3 2 1 2 1 0 1 1 1

【分析】最直观的想法是，投谁，谁的票数加1。如：投3号，则3号的票数加1，投5号，5号的票数加1。

设变量num[i]表示第i个歌手的票数，相当于用名称num表示票数，然后再对名称进行编号，编号为i的num表示第i个歌手的票数。

这样，投3号，则3号的票数加1，就可以表达为：

i=3;num[i]=num[i]+1;

投5号，则5号的票数加1，就可以表达为：

i=6; num[i]=num[i]+1;

归纳上述分析，具体实现步骤如下：

(1)开辟num[1]～num[10]变量分别存放10个歌手的票数。

(2)读入选票赋给i。

(3)对应选票i的歌手票数加1，即num[i]=num[i]+1。

(4)重复(2)和(3)的操作，直到读完选票为止。

(5)输出每位歌手的票数。

程序如下：

```
#include<iostream>
#include<cstring>
using namespace std;
int main()
{
    int i,num[11];
    memset(num,0,sizeof(num));          //数组初始化为0
    while(cin>>i)                       //读入选票
    num[i]=num[i]+1;                    //num[i]表示i号歌手的选票数
    for (i=1;i<=10;i++)
    cout<<i<<" ";                       //输出1～10选手编号
    cout<<endl;
    for (i=1;i<=10;i++)
    cout<<num[i]<<" ";                  //输出各位歌手的最终选票数
    return 0;
}
```

例5.7　用筛法求出100以内的全部素数，并按每行五个数显示。

【分析】

(1) 把2到100的自然数放入a[2]到a[100]中(所放入的数与下标号相同)；

(2) 在数组元素中，以下标为序，按顺序找到未曾找过的最小素数minp和它的位置p(即下标号)；

(3) 从p+1开始，把凡是能被minp整除的各元素值从a数组中划去(筛掉)，也就是给

该元素值置 0;

(4) 让 p=p+1,重复执行第(2)、(3)步骤,直到 minp>floor(sqrt(N)) 为止;

(5) 打印输出 a 数组中留下来、未被筛掉的各元素值,并按每行五个数显示。

用筛法求素数的过程示意如下(图中用下划线作删去标志):

① 2 3 4 5 6 7 8 9 10 11 12 13 14 15…98 99 100　　//置数

② 2 3 4 5 6 7 8 9 10 11 12 13 14 15…98 99 100　　//筛去被 2 整除的数

③ 2 3 4 5 6 7 8 9 10 11 12 13 14 15…98 99 100　　//筛去被 3 整除的数

…

2 3 4 5 6 7 8 9 10 11 12 13 14 15…98 99 100　　//筛去被整除的数

程序如下:

```
#include<iostream>
#include<math.h>          //在 Dev C++中可调用数学函数库 cmath
#include<iomanip>
using namespace std;
const int n=100;
bool a[n+1];
int main()
{
   for (int i=0; i<=n;++i) a[i]=true;
                              //等同于 memset(a,1,sizeof(a)) ,要调用 cstring 库
   a[1]=false;
   for (int i=2; i<=sqrt(n);++i)
     if (a[i])
       for (int j=2; j<=n/i;++j)
         a[i*j]=false;
   for (int i=2,t=0; i<=n;++i)
     if (a[i])
     {
         cout<<setw(5)<<i;
         ++t;
         if (t%5==0) cout<<endl;
     }
   return 0;
}
```

【上机练习】

1. 与指定数字相同的数的个数【1.6 编程基础之一维数组 01】

2. 陶陶摘苹果【1.6 编程基础之一维数组 02】Noip2005 普及组第 1 题

3. 计算书费【1.6 编程基础之一维数组 03】

4. 数组逆序重【1.6 编程基础之一维数组 04】

5. 年龄与疾病【1.6 编程基础之一维数组 05】

6. 校门外的树【1.6 编程基础之一维数组 06】Noip2005 普及组第 2 题

7. 向量点积计算【1.6 编程基础之一维数组 09】

8. 开关灯【1.5 编程基础之循环控制 31】

10. 不高兴的津津【1.9 编程基础之顺序查找 03】Noip2004 普及组第 1 题

11. 最大值和最小值的差【1.9 编程基础之顺序查找 05】

12. 不与最大数相同的数字之和【1.9 编程基础之顺序查找 07】

13. 白细胞计数【1.9 编程基础之顺序查找 08】

14. 直方图【1.9 编程基础之顺序查找 09】

15. 最长平台【1.9 编程基础之顺序查找 12】

16. 整数去重【1.9 编程基础之顺序查找 13】

17. 铺地毯【1.9 编程基础之顺序查找 14】Noip2011 提高组第 1 题

第二节　二维数组

一、二维数组的定义

当一维数组元素的类型也是一维数组时，便构成了“数组的数组”，即二维数组。二维数组定义的一般格式：

数据类型　数组名[常量表达式1][常量表达式2]；

例如：int a[4][10]；

a数组实质上是一个有4行、10列的表格，表格中可储存40个元素。第1行第1列对应a数组的a[0][0]，第n行第m列对应数组元素a[n-1][m-1]。

说明：当定义的数组下标有多个时，我们称为多维数组，下标的个数并不局限在一个或两个，可以任意多个，如定义一个三维数组a和四维数组b：

```
int a[100][3][5];
int b[100][100][3][5];
```

多维的数组引用赋值等操作与二维数组类似。

二、二维数组元素的引用

二维数组的数组元素引用与一维数组元素引用类似，区别在于二维数组元素的引用必须给出两个下标。

引用的格式为：

＜数组名＞[下标1][下标2]

说明：显然，每个下标表达式取值不应超出下标所指定的范围，否则会导致致命的越界错误。

例如，设有定义：int a[3][5]；

则表示a是二维数组(相当于一个3*5的表格)，共有3*5=15个元素，它们是：

```
a[0][0] a[0][1] a[0][2] a[0][3] a[0][4]
a[1][0] a[1][1] a[1][2] a[1][3] a[1][4]
a[2][0] a[2][1] a[2][2] a[2][3] a[2][4]
```

因此可以看成一个矩阵(表格)，a[2][3]即表示第3行第4列的元素。

三、二维数组的初始化

二维数组的初始化和一维数组类似。可以将每一行分开来写在各自的括号里，也可以把所有数据写在一个括号里。

例如：

```
int direct[4][2]={{1,0},{0,1},{-1,0},{0,-1}}
int direct[4][2]={1,0,0,1,-1,0,0,-1}      //尽量不要用
```

```
    }
    for (int i=1; i<=10;++i)
    {
      if (i! =10) cout<<setw(30-3 * i)<<" ";//控制每行的起始位置,即空格数量
      for (int j=1; j<=i;++j) cout<<setw(6)<<a[i][j];
      cout<<endl;
    }
    return 0;
}
```

例 5.12 输入一串字符,字符个数不超过 100,且以“.”结束。判断它们是否构成回文。

【分析】 所谓回文指从左到右和从右到左读一串字符的值是一样的,如 12321,ABCBA,AA 等。先读入要判断的一串字符(放入数组 letter 中),并记住这串字符的长度,然后首尾字符比较,并不断向中间靠拢,就可以判断出是否为回文。

程序如下:

```
#include<iostream>
using namespace std;
int main()
{   char ch,letter[101];
    cout<<" Input a string:";
    cin>>ch;
    int i=0,j=1;
    while (ch! ='.')                          //读入一个字符串以'.'号结束
    {
      ++i;
      letter[i]=ch;
      cin>>ch;
    }
    while ((j<i)&&(letter[j]==letter[i]))     //判断它是否是回文
    {
        --i;++j;
    }
    if (j>=i) cout<<" Yes "<<endl;
      else  cout<<" No "<<endl;
    return 0;
}
```

例 5.13 蛇形填数

在 n * n 方阵里填入 1,2,3,…,n * n,要求填成蛇形。例如 n=4 时方阵为:

```
10 11 12 1
 9 16 13 2
 8 15 14 3
 7  6  5 4
```

四、二维数组程序设计

例 5.8　设有一程序

```
#include<cstdio>
#include<iostream>
#include<iomanip>
const int n=3;
using namespace std;
int a[n+1][n+1];
int main()
{
   for (int i=1; i<=n;++i)
   {
     for (int j=1; j<=n;++j)
        cin>>a[i][j];
   }
   for (int i=1; i<=n;++i)
   {
        for (int j=1; j<=n;++j) cout<<setw(5)<<a[j][i];
        cout<<endl;
   }
   return 0;
}
```

程序的输入：

```
2 1 3
3 3 1
1 2 1
```

程序的输出：

```
2 3 1
1 3 2
3 1 1
```

例 5.9　已知一个 6 * 6 的矩阵(方阵)，把矩阵二条对角线上的元素值加上 10，然后输出这个新矩阵。

【分析】　矩阵即表格，是一个二维数组，有 6 行 6 列共 36 个元素，每个矩阵都有二条对角线，本题难点在于对角线的元素怎么确定。

```
#include<iostream>
#include<iomanip>
using namespace std;
int a[7][7];
int main()
```

```
{
  for (int i=1; i<=6;++i)          //输入矩阵元素
    for (int j=1; j<=6;++j)
      cin>>a[i][j];
  for (int i=1; i<=6;++i)          //更改对角线上元素的值
    for (int j=1; j<=6;++j)
      if ((i==j)||(i+j==7)) a[i][j]+=10; //寻找对角线的特征
  for (int i=1; i<=6;++i)          //输出6行6列的矩阵元素
  {
      for (int j=1; j<=6;++j)
        cout<<setw(5)<<a[i][j];
      cout<<endl;
  }
  return 0;
}
```

例 5.10 大部分元素是0的矩阵称为稀疏矩阵,假设有k个非0元素,则可把稀疏矩阵用k*3的矩阵简记之,其中第一列是行号,第二列是列号,第三列是该行、该列下的非元素的值。如:

```
0 0 0 5        简记成: 1 4 5     //第1行第4列有个数是5
0 2 0 0                2 2 2     //第2行第2列有个数是2
0 1 0 0                3 2 1     //第3行第2列有个数是1
```

试编程读入一稀疏矩阵,转换成简记形式,并输出。

【分析】 本题中需要解决的主要问题是查找非零元素并记忆位置。将原始矩阵存于数组a。转换后的矩阵存于数组b,当然b数组的行数可以控制在一个小范围内。

```
#include<iostream>
#include<iomanip>
const int n=3,m=5;
using namespace std;
int main()
{
  int a[n+1][m+1],b[101][4],k=0;
  for (int i=1; i<=n;++i)          //矩阵初始
    for (int j=1; j<=m;++j)
      cin>>a[i][j];
  for (int i=1; i<=n;++i)
    for (int j=1; j<=m;++j)
      if (a[i][j]!=0)              //找到非零值,存储
      {
          ++k;
          b[k][1]=i;
          b[k][2]=j;
```

```
            b[k][3]=a[i][j];
        }
    for (int i=1; i<=k;++i)          //输出
    {
      for (int j=1; j<=3;++j) cout<<setw(3)<<b[i][j];
      cout<<endl;
    }
    return 0;
}
```

运行结果：

```
输入：0 0 0 0 5
      0 0 4 0 0
      1 0 0 0 1
输出：1 5 5
      2 3 4
      3 1 1
      3 5 1
```

例 5.11　打印杨辉三角形的前 10 行。杨辉三角形如下图：

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

图 5－1　　图 5－2

【分析】　观察图 5－1，大家不容易找到规律，但是如果将它转化为图 5－2，不难发现杨辉三角形其实就是一个二维表的小三角形部分，假设通过二维数组 yh 存储，每行首尾元素为 1，且其中任意一个非首位元素 yh[i][j]的值其实就是 yh[i－1][j－1]与 yh[i－1][j]的和，另外每一行的元素个数刚好等于行数。有了数组元素的值，要打印杨辉三角形，只需要控制好输出起始位置就行了。

```
#include<iostream>
#include<iomanip>
using namespace std;
int main()
{
  int a[11][11];
    a[1][1]=1;                          //设定第一行的值
  for (int i=2; i<=10;++i)              //从第二行开始推
  {
      a[i][1]=1;  a[i][i]=1;            //设定每一行的首尾值为 1
      for (int j=2; j<=i-1;++j)         //当前行非首尾的数
        a[i][j]=a[i-1][j-1]+a[i-1][j]; //每个数等于上一行的两个数之和
```

上面的方阵中，多余的空格只是为了便于观察规律，不必严格输出，n<=8。

【分析】 类比数学中的矩阵，我们可以用一个所谓的二维数组来储存题目中的方阵。只需声明一个 int a[MAXN][MAXN]，就可以获得一个大小为 MAXN * MAXN 的方阵。在声明时，两维的大小不必相同，因此也可以声明 int a[30][50]这样的数组，第一维下标范围是 0,1,2,…,29，第二维下标范围是 0,1,2,…,49。

让我们从 1 开始依次填写。设“笔”的坐标为(x,y)，则一开始 x=0,y=n-1，即第 0 行，第 n-1 列(别忘了行列的范围是 0 到 n-1，没有第 n 列)。“笔”的移动轨迹是：下，下，下，左，左，左，上，上，上，右，右，下，下，左，上。总之，先是下，到不能填了为止，然后是左，接着是上，最后是右。“不能填”是指再走就出界(例如 4→5)，或者再走就要走到以前填过的格子(例如 12→13)。如果我们把所有格子初始为 0，就能很方便地加以判断。

```
#include<cstdio>
#include<cstring>
#define MAXN 10
int a[MAXN][MAXN];
int main()
{
    int n,x,y,tot;
    scanf("%d",&n);
    memset(a,0,sizeof(a));
    tot=a[x=0][y=n-1]=1;
    while (tot<n*n)
    {
      while (x+1<n && ! a[x+1][y]) a[++x][y]=++tot;
      while (y-1>=0 && ! a[x][y-1]) a[x][--y]=++tot;
      while (x-1>=0 && ! a[x-1][y]) a[--x][y]=++tot;
      while (y+1<n && ! a[x][y+1]) a[x][++y]=++tot;
    }
    for(int i=0;i<n;++i)
    {
      for (int j=0;j<n;++j) printf("%3d",a[i][j]);
      printf("\n");
    }
    return 0;
}
```

【说明】

这段程序充分利用了 C++语言简洁的优势。首先，赋值 x=0 和 y=n-1 后马上要把它们作为 a 数组的下标，因此可以合并完成；tot 和 a[0][n-1]都要赋值 1，也可以合并完成。这样，我们用一条语句完成了多件事情，而且并没有牺牲程序的可读性，这段代码的含义显而易见。

那 4 条 while 语句有些难懂，不过十分相似，因此只需介绍其中的第一条：不断向下走，并且填数。我们的原则是：先判断，再移动，而不是走一步以后发现越界了再退回来。这样，我们需要进行“预判”，即是否越界，以及如果继续往下走会不会到达一个已经填过的格子。

越界只需判断 x+1<n,因为 y 值并没有修改;下一个格子是(x+1,y),因此只需 a[x+1][y]==0,简写成!a[x+1][y](其中!是"逻辑非"运算符)。

细心的读者也许会发现这里的一个"潜在 bug";如果越界,x+1 会等于 n,a[x+1][y]将访问非法内存!幸运的是,这样的担心是不必要的。&& 是短路运算符。如果 x+1<n 为假,将不会计算!a[x+1][y],也就不会越界了。

至于为什么是++tot 而不是 tot++,留给读者思考。

【上机练习】

1. 矩阵交换行【1.8 编程基础之多维数组 01】

给定一个 5*5 的矩阵(数学上,一个 r×c 的矩阵是一个由 r 行 c 列元素排列成的矩形阵列),将第 n 行和第 m 行交换,输出交换后的结果。

输入:

输入共 6 行,前 5 行为矩阵的每一行元素,元素与元素之间以一个空格分开。

第 6 行包含两个整数 m、n,以一个空格分开(1 <= m,n <= 5)。

输出:

输出交换之后的矩阵,矩阵的每一行元素占一行,元素之间以一个空格分开。

2. 同行列对角线的格【1.8 编程基础之多维数组 02】

输入三个自然数 N,i,j(1<=i<=n,1<=j<=n),输出在一个 N*N 格的棋盘中(行列均从 1 开始编号),与格子(i,j)同行、同列、同一对角线的所有格子的位置。

如:n=4,i=2,j=3 表示了棋盘中的第二行第三列的格子,如下图:



当 n=4,i=2,j=3 时,输出的结果是:

(2,1) (2,2) (2,3) (2,4)	同一行上格子的位置
(1,3) (2,3) (3,3) (4,3)	同一列上格子的位置
(1,2) (2,3) (3,4)	左上到右下对角线上的格子的位置
(4,1) (3,2) (2,3) (1,4)	左下到右上对角线上的格子的位置

输入:

一行,三个自然数 N,i,j,相邻两个数之间用单个空格隔开(1<=N<=10)。

输出:

第一行:从左到右输出同一行格子位置;

第二行:从上到下输出同一列格子位置;

第三行:从左上到右下输出同一对角线格子位置;

第四行:从左下到右上输出同一对角线格子位置。

其中每个格子位置用如下格式输出:(x,y),x 为行号,y 为列号,采用英文标点,中间无空格。相邻两个格子位置之间用单个空格隔开。

3. 计算矩阵边缘元素之和【1.8 编程基础之多维数组 03】

输入一个整数矩阵，计算位于矩阵边缘的元素之和。所谓矩阵边缘的元素，就是第一行和最后一行的元素以及第一列和最后一列的元素。

输入：

第一行分别为矩阵的行数 m 和列数 n(m<100,n<100)，两者之间以一个空格分开。

接下来输入的 m 行数据中，每行包含 n 个整数，整数之间以一个空格分开。

输出：

输出对应矩阵的边缘元素和。

4. 计算鞍点【1.8 编程基础之多维数组 05】

给定一个 5*5 的矩阵，每行只有一个最大值，每列只有一个最小值，寻找这个矩阵的鞍点。鞍点指的是矩阵中的一个元素，它是所在行的最大值，并且是所在列的最小值。

例如：在下面的例子中(第 4 行第 1 列的元素就是鞍点，值为 8)。

11　3　5　6　9

12　4　7　8　10

10　5　6　9　11

 8　6　4　7　2

15　10 11 20 25

输入：

输入包含一个 5 行 5 列的矩阵。

输出：

如果存在鞍点，输出鞍点所在的行、列及其值，如果不存在，输出" not found "。

5. 图像相似度【1.8 编程基础之多维数组 06】

给出两幅相同大小的黑白图像(用 0－1 矩阵)表示，求它们的相似度。说明：若两幅图像在相同位置上的像素点颜色相同，则称它们在该位置具有相同的像素点。两幅图像的相似度定义为相同像素点数占总像素点数的百分比。

输入：

第一行包含两个整数 m 和 n，表示图像的行数和列数，中间用单个空格隔开。1<=m<=100，1<=n<=100。

之后 m 行，每行 n 个整数 0 或 1，表示第一幅黑白图像上各像素点的颜色。相邻两个数之间用单个空格隔开。

之后 m 行，每行 n 个整数 0 或 1，表示第二幅黑白图像上各像素点的颜色。相邻两个数之间用单个空格隔开。

输出：

一个实数，表示相似度(以百分比的形式给出)，精确到小数点后两位。

6. 矩阵加法【1.8 编程基础之多维数组 08】

输入两个 n 行 m 列的矩阵 A 和 B，输出它们的和 A+B。

输入：

第一行包含两个整数 n 和 m，表示矩阵的行数和列数(1<=n<=100，1<=m<=100)。

接下来 n 行，每行 m 个整数，表示矩阵 A 的元素。

接下来 n 行，每行 m 个整数，表示矩阵 B 的元素。

相邻两个整数之间用单个空格隔开，每个元素均在 1～1000 之间。

输出：

n行,每行m个整数,表示矩阵加法的结果。相邻两个整数之间用单个空格隔开。

7. 矩阵乘法【1.8 编程基础之多维数组 09】

计算两个矩阵的乘法。n*m阶的矩阵A乘以m*k阶的矩阵B得到的矩阵C是n*k阶的,且C[i][j] = A[i][0]*B[0][j] + A[i][1]*B[1][j] + …… +A[i][m-1]*B[m-1][j](C[i][j]表示C矩阵中第i行第j列元素)。

输入:

第一行为n, m, k,表示A矩阵是n行m列,B矩阵是m行k列,n, m, k均小于100。

然后先后输入A和B两个矩阵,A矩阵n行m列,B矩阵m行k列,矩阵中每个元素的绝对值不会大于1000。

输出:

输出矩阵C,一共n行,每行k个整数,整数之间以一个空格分开。

8. 矩阵转置【1.8 编程基础之多维数组 10】

输入一个n行m列的矩阵A,输出它的转置A^T。

输入:

第一行包含两个整数n和m,表示矩阵A的行数和列数(1<=n<=100,1<=m<=100)。

接下来n行,每行m个整数,表示矩阵A的元素。相邻两个整数之间用单个空格隔开,每个元素均在1~1000之间。

输出:

m行,每行n个整数,为矩阵A的转置。相邻两个整数之间用单个空格隔开。

9. 图像旋转【1.8 编程基础之多维数组 11】

输入一个n行m列的黑白图像,将它顺时针旋转90度后输出。

输入:

第一行包含两个整数n和m,表示图像包含像素点的行数和列数。1<=n<=100,1<=m<=100。

接下来n行,每行m个整数,表示图像的每个像素点灰度。相邻两个整数之间用单个空格隔开,每个元素均在0~255之间。

输出:

m行,每行n个整数,为顺时针旋转90度后的图像。相邻两个整数之间用单个空格隔开。

10. 图像模糊处理【1.8 编程基础之多维数组 13】

给定m行n列的图像各像素点的灰度值,要求用如下方法对其进行模糊化处理:

1. 四周最外侧的像素点灰度值不变;

2. 中间各像素点新灰度值为该像素点及其上下左右相邻四个像素点原灰度值的平均(舍入到最接近的整数)。

输入:

第一行包含两个整数n和m,表示图像包含像素点的行数和列数。1<=n<=100,1<=m<=100。

接下来n行,每行m个整数,表示图像的每个像素点灰度。相邻两个整数之间用单个空格隔开,每个元素均在0~255之间。

输出:

m行,每行n个整数,为模糊处理后的图像。相邻两个整数之间用单个空格隔开。

第三节　字符类型和字符数组

无论数组的下标有几个，类型如何，但数组中全体元素的类型必须相同。数组元素的类型可以是任何类型，当它是字符型时，我们称它为字符数组。由于字符数组与字符类型的应用是计算机非数值处理的重要方面之一，所以我们把它们两个放在一起进行讨论。

下面我们举例说明字符数组的应用。

一、字符类型

字符类型为由一个字符组成的字符常量或字符变量。

字符常量定义：

```
const
    字符常量='字符'
```

字符变量定义：

```
    char 字符变量;
```

字符类型是一个有序类型，字符的大小顺序按其 ASCII 代码的大小而定。

例 5.14　按字母表顺序和逆序每隔一个字母打印。即打印出：

```
  a  c  e  g  i  k  m  o  q  s  u  w  y
  z  x  r  v  t  p  n  l  j  h  f  d  b
```

程序如下：

```
#include<iostream>
#include<iomanip>
using namespace std;
int main()
{
    for (char letter='a'; letter<='z'; letter+=2)
      cout<<setw(3)<<letter;
    cout<<endl;
    for (char letter='z'; letter>='a'; letter-=2)
      cout<<setw(3)<<letter;
    return 0;
}
```

【说明】　程序中，我们利用了字符类型是顺序类型这一特性，灵活利用字符变量当作循环变量，使程序处理起来比较直观。

二、字符数组

字符数组是指元素为字符的数组。字符数组是用来存放字符序列或字符串的。字符数

组也有一维、二维和三维之分。

1. 字符数组的定义格式

字符数组定义格式同于一般数组,所不同的是数组类型是字符型,第一个元素同样是从ch1[0]开始,而不是ch1[1]。具体格式如下:

[存储类型] char 数组名[常量表达式1]…

例如:

```
char ch1[5];          //数组 ch1 是一个具有 5 个字符元素的一维字符数组
char ch2[3][5];       //数组 ch2 是一个具有 15 个字符元素的二维字符数组
```

2. 字符数组的赋值

字符数组赋值类似于一维数组,赋值分为数组的初始化和数组元素的赋值。初始化的方式有用字符初始化和用字符串初始化两种,也有用初始值表进行初始化的。

(1)用字符初始化数组

例如:

```
char chr1[5]={'a','b','c','d','e'};
```

初始值表中的每个数据项是一个字符,用字符给数组chr1的各个元素初始化。当初始值个数少于元素个数时,从首元素开始赋值,剩余元素默认为空字符。

字符数组中也可以存放若干个字符,也可以来存放字符串。两者的区别是字符串有一结束符('\0')。反过来说,在一维字符数组中存放着带有结束符的若干个字符称为字符串。字符串是一维数组,但是一维字符数组不等于字符串。

例如:

char chr2[5]={'a','b','c','d','\0'};即在数组chr2中存放着一个字符串"abcd"。

(2)用字符串初始化数组

用一个字符串初始化一个一维字符数组,可以写成下列形式:

```
char chr2[5]="abcd";
```

使用此格式均要注意字符串的长度应小于字符数组的大小或等于字符数组的大小减1。同理,对二维字符数组来讲,可存放若干个字符串。可使用由若干个字符串组成的初始值表给二维字符数组初始化。

例如:char chr3[3][4]={"abc","mno","xyz"};在数组ch3中存放3个字符串,每个字符串的长度不得大于3。

(3)数组元素赋值

字符数组的赋值是给该字符数组的各个元素赋一个字符值。

例如:

```
char chr[3];
chr[0]='a'; chr[1]='b';chr[2]='c';
```

对二维、三维字符数组也是如此。当需要将一个数组的全部元素值赋予另一数组时,不可以用数组名直接赋值的方式,要使用字符串拷贝函数来完成。

(4)字符常量和字符串常量的区别

①两者的定界符不同,字符常量由单引号括起来,字符串常量由双引号括起来。

②字符常量只能是单个字符,字符串常量则可以是多个字符。

③可以把一个字符常量赋给一个字符变量,但不能把一个字符串常量赋给一个字符

变量。

④字符常量占一个字节，而字符串常量占用字节数等于字符串的字节数加1。增加的一个字节中存放字符串结束标志'\0'。例如：字符常量'a'占一个字节，字符串常量"a"占两个字节。

三、字符串的输入与输出

字符串可以作为一维字符数组来处理，那么字符串的输入和输出也可以按照数组元素来处理，本节不再做介绍。本节仅介绍将字符串作为一个整体进行输入和输出的语句。

1. 输入

从键盘输入一个字符数组可以使用scanf语句或gets语句。

(1)scanf语句

格式：scanf("%s",字符串名称)；

说明：

①这里的字符串名称之前不加&这个取地址符。例如：scanf("%s",&s1)是错误的。

②系统会自动在输入的字符串常量后添加'\0'标志，因此输入时，仅输入字符串的内容即可。

③输入多个字符串时，以空格分隔。

例如：scanf("%s%s%s",s1,s2,s3)；从键盘分别输入Let us go，则三个字符串分别获取了三个单词。反过来可以想到，如果仅有一个输入字符串名称的情况下，字符串变量仅获取空格前的内容。

例如：scanf("%s",s1)；从键盘分别输入Let us go，则仅有第一个单词被获取，即s1变量仅获取第一个单词Let。

(2)gets语句

格式：gets(字符串名称)；

说明：

使用gets只能输入一个字符串。

例如：gets(s1,s2)；是错误的。使用gets，是从光标开始的地方读到换行符也就是说读入的是一整行，而使用scanf是从光标开始的地方到空格，如果这一行没有空格，才读到行尾。

例如：scanf("%s",s1)；gets(s2)；对于相同的输入Hello World!。s1获取的结果仅仅是Hello，而s2获取的结果则是Hello World!

2. 输出

向屏幕输出一个字符串可以使用printf语句或puts语句。

(1)printf语句

格式：printf("%s",字符串名称)；

说明：

①用%s格式输出时，printf的输出项只能是字符串(字符数组)名称，而不能是数组元素。例如：printf("%s",a[5])；是错误的。

②输出字符串不包括字符串结束标志符'\0'。

(2) puts 语句

格式:puts(字符串名称);

说明:puts 语句输出一个字符串和一个换行符。对于已经声明过的字符串 a,printf("%s\n",a)和 puts(a)是等价的。

四、字符串 string 类型

详见《信息学奥赛一本通拓展教程》。

例 5.15 在应用计算机编辑文档的时候,我们经常遇到替换任务。如把文档中的"电脑"都替换成"计算机"。现在请你编程模拟一下这个操作。

输入两行内容,第 1 行是原文(长度不超过 200 个字符),第 2 行包含以空格分隔的两个字符 A 和 B,要求将原文中所有的字符 A 都替换成字符 B,注意:区分大小写字母。

输入样例:

I love China. I love Beijing.

I U

输出样例:

U love China. U love Beijing.

【分析】

首先要将给定的原文保存在字符数组里。然后,在原文中,从头开始寻找字符 A,找到一个字符 A,便将其替换成字符 B;继续寻找下一个字符 A,找到了就替换,……,直到将原文都处理完。如下程序只能处理单个字符替换,无法处理单词替换,I U 中间只能有一个空格。getchar()输入的使用方法详见教材的第二章第四节,单词替换详见《信息学奥赛一本通拓展教程》。

程序如下:

```
#include<cstdio>
#include<iostream>
using namespace std;
int main()
{
    char st[200];
    char A,B;
    int i,n=0;
    while((st[n++]=getchar())!='\n');  //将原文存放在字符数组 st 中
    A=getchar();getchar();B=getchar();
        //读取 A 和 B,中间 getchar()读空格,不能省略
    for (i=0;i<n;i++)
      if (st[i]==A) cout<<B;
      else cout<<st[i];
    cout<<endl;
    return 0;
```

```
}
```

例 5.16　过滤多余的空格【1.7 编程基础之字符串 23】

一个句子中也许有多个连续空格，过滤掉多余的空格，只留下一个空格。

输入：

一行，一个字符串（长度不超过 200），句子的头和尾都没有空格。

输出：

过滤之后的句子。

样例输入：

Hello　　world. This is　　c language.

样例输出：

Hello world. This is c language.

【分析】

scanf 只能一个一个读“单词”，不读空格，while (scanf("%s ",&st)==1)的功能是循环读入数据，在读不到的时候停止循环。

程序如下：

```
#include<cstdio>
using namespace std;
char st[200];
int main()
{
    while (scanf("%s ",&st)==1)
      printf("%s   ",st);          //%s 后要有一个空格，不能省略
    return 0;
}
```

例 5.17　C++中，一个字符串中的字符可以通过其对应的下标灵活使用。

```
#include<cstdio>                   // gets()调用 cstdio 库
#include<iostream>
#include<cstring>                  //strlen()调用 cstring 库，调用 string 库会编译出错
using namespace std;
int main()
{
  char st[100];
  gets(st);              //gets 为专门读字符串的函数，读取一行字符串
  for (int i=0; i<strlen(st);++i)     //输出 st 串中的第 i 个字符
    cout<<st[i];
  return 0;
}
```

五、字符串处理函数

系统提供了一些字符串处理函数,用来为用户提供一些字符串的运算。常用的字符串函数介绍如下。

函数格式	函数功能
strcat(字符串名1,字符串名2)	将字符串2连接到字符串1后边,返回字符串1的值。
strncat(字符串名1,字符串名2,长度n)	将字符串2前n个字符连接到字符串1后边,返回字符串1的值。
strcpy(字符串名1,字符串名2)	将字符串2复制到字符串1,返回字符串1的值。
strncpy(字符串名1,字符串名2,长度n)	将字符串2前n个字符复制到字符串1,返回字符串1的值。
strcmp(字符串名1,字符串名2)	比较字符串1和字符串2的大小,比较的结果由函数带回; 如果字符串1>字符串2,返回一个正整数; 如果字符串1=字符串2,返回0; 如果字符串1<字符串2,返回一个负整数。
strncmp(字符串名1,字符串名2,长度n)	比较字符串1和字符串2的前n个字符进行比较,函数返回值的情况同strcmp函数。
strlen(字符串名)	计算字符串的长度,终止符'\0'不算在长度之内。
strlwr(字符串名)	将字符串中大写字母换成小写字母
strupr(字符串名)	将字符串中小写字母换成大写字母

例5.18 对给定的10个国家名,按其字母的顺序输出。

【参考程序1】

```
#include<cstdio>
#include<iostream>
#include<cstring>
using namespace std;
int main()
{
  char t[21],cname[11][21];
  for (int i=1; i<=10; ++i)
    gets(cname[i]);              //gets为专门读字符串的函数,读取一行字符串
  for (int i=1; i<=9; ++i)
  {
      int k=i;
      for (int j=i+1; j<=10; ++j)
        if (strcmp(cname[k],cname[j])>0) k=j;
      strcpy(t,cname[i]);
```

```
        strcpy(cname[i],cname[k]);
        strcpy(cname[k],t);
    }
    for (int i=1; i<=10; ++i)
      cout<<cname[i]<<endl;
    return 0;
}
```

【参考程序 2】(详见《信息学奥赛一本通拓展教程》)

```
#include<algorithm>
#include<iostream>
#include<string>
using namespace std;
string cname[10];
int main()
{
    for (int i=0;i! =10;++i)
      getline(cin,cname[i]);
    sort(cname,cname+10);                    //利用 C++库函数排序
    for (int i=0;i! =10;++i)
      cout<<cname[i]<<endl;
    return 0;
}
```

例 5.19　字符串判等【1.7 编程基础之字符串 17】

【问题描述】

判断两个由大小写字母和空格组成的字符串在忽略大小写,且忽略空格后是否相等。

【输入】

两行,每行包含一个字符串。

【输出】

若两个字符串相等,输出 YES,否则输出 NO。

【输入样例】

a A bb BB ccc CCC

Aa BBbb CCCccc

【输出样例】

YES

【参考程序】

```
#include<cstring>
#include<cstdio>
using namespace std;
const int N=256;
char s1[N],s2[N],a[N],b[N];
```

```
int l1,l2;
int main()
{
    gets(s1); gets(s2);
    strlwr(s1); strlwr(s2);
        //将 s1,s2 同时转成大(或小)写,等同于 strupr(s1);strupr(s2);
    for (int i=0; i<strlen(s1); ++i)
      if (s1[i]! =' ') a[l1++]=s1[i];
    for (int i=0; i<strlen(s2); ++i)
      if (s2[i]! =' ') b[l2++]=s2[i];     //处理忽略空格
    if(strcmp(a,b)==0) printf(" YES\n ");
      else printf(" NO\n ");
    return 0;
}
```

例 5.20 字符串移位包含问题【1.7 编程基础之字符串 19】

【问题描述】

对于一个字符串来说,定义一次循环移位操作为:将字符串的第一个字符移动到末尾形成新的字符串。

给定两个字符串 s1 和 s2,要求判定其中一个字符串是否是另一个字符串通过若干次循环移位后的新字符串的子串。例如 CDAA 是由 AABCD 两次移位后产生的新串 BCDAA 的子串,而 ABCD 与 ACBD 则不能通过多次移位来得到其中一个字符串是新串的子串。

【输入】

一行,包含两个字符串,中间由单个空格隔开。字符串只包含字母和数字,长度不超过 30。

【输出】

如果一个字符串是另一个字符串通过若干次循环移位产生的新串的子串,则输出 true,否则输出 false。

【输入样例】

AABCD CDAA

【输出样例】

true

【参考程序】

```
#include<cstdio>
#include<cstring>
#include<iostream>
using namespace std;
const int N=61;
char s1[N],s2[N],x[N],t[N];
int l1,l2;
int main()
```

```
{
    scanf("%s%s",&s1,&s2);
    if (strlen(s1)<strlen(s2))      //将长度短的字符串作为预判断字串
    {
        strcpy(t,s1);
        strcpy(s1,s2);
        strcpy(s2,t);              //将 s1,s2 互换
    }
    strcpy(x,s1);
    if (strstr(strcat(s1,x),s2)==NULL) printf(" false\n ");
        //strstr(s1,s2)函数用于判断字符串 s2 是否是 s1 的子串
        //如果是,则该函数返回 str2 在 str1 中首次出现的地址;否则,返回 NULL
    else printf(" true\n ");
    return 0;
}
```

例 5.21　谁拿了最多奖学金【1.9 编程基础之顺序查找 04】(Noip2005 提高组第 1 题)

某校的惯例是在每学期的期末考试之后发放奖学金。发放的奖学金共有五种,获取的条件各自不同:

(1)院士奖学金,每人 8000 元,期末平均成绩高于 80 分(>80),并且在本学期内发表 1 篇或 1 篇以上论文的学生均可获得;

(2)五四奖学金,每人 4000 元,期末平均成绩高于 85 分(>85),并且班级评议成绩高于 80 分(>80)的学生均可获得;

(3)成绩优秀奖,每人 2000 元,期末平均成绩高于 90 分(>90)的学生均可获得;

(4)西部奖学金,每人 1000 元,期末平均成绩高于 85 分(>85)的西部省份学生均可获得;

(5)班级贡献奖,每人 850 元,班级评议成绩高于 80 分(>80)的学生干部均可获得;

只要符合条件就可以得奖,每项奖学金的获奖人数没有限制,每名学生也可以同时获得多项奖学金。例如姚林的期末平均成绩是 87 分,班级评议成绩 82 分,同时他还是一位学生干部,那么他可以同时获得五四奖学金和班级贡献奖,奖金总数是 4850 元。

现在给出若干学生的相关数据,请计算哪些同学获得的奖金总数最高(假设总有同学能满足获得奖学金的条件)。

输入:

第一行是一个整数 N(1 <= N <= 100),表示学生的总数。接下来的 N 行每行是一位学生的数据,从左向右依次是姓名,期末平均成绩,班级评议成绩,是否是学生干部,是否是西部省份学生,以及发表的论文数。姓名是由大小写英文字母组成的长度不超过 20 的字符串(不含空格);期末平均成绩和班级评议成绩都是 0 到 100 之间的整数(包括 0 和 100);是否是学生干部和是否是西部省份学生分别用一个字符表示,Y 表示是,N 表示不是;发表的论文数是 0 到 10 的整数(包括 0 和 10)。每两个相邻数据项之间用一个空格分隔。

输出:

包括三行,第一行是获得最多奖金的学生的姓名,第二行是这名学生获得的奖金总数。

如果有两位或两位以上的学生获得的奖金最多,输出他们之中在输入文件中出现最早的学生的姓名。第三行是这N个学生获得的奖学金的总数。

样例输入:

```
4
YaoLin 87 82 Y N 0
ChenRuiyi 88 78 N Y 1
LiXin 92 88 N N 0
ZhangQin 83 87 Y N 1
```

样例输出:

```
ChenRuiyi
9000
28700
```

程序如下:

```
#include <cstdio>
#include <cstring>
using namespace std;
int main()
{
    char s[100][20],c1,c2;
    int n,sum=0,max=0,score1,score2,num,a,b,k;
    scanf("%d",&n);
    for (int i=1;i<=n;i++)
      //scanf 读入数据,数据间有一个或多个空格,自动赋给相应的变量
    {
       scanf("%s %d %d %c %c %d",&s[i],&score1,&score2,&c1,&c2,
       &num);
       a=c1=='Y'? 1:0;                    //是否是学生干部
       b=c2=='Y'? 1:0;                    //是否是西部省份学生
       int tmp=0;
       if ((score1>80)&&(num>0)) tmp+=8000;        //院士奖学金
       if ((score1>85)&&(score2>80)) tmp+=4000;    //五四奖学金
       if (score1>90) tmp+=2000;                   //成绩优秀奖
       if (b&&(score1>85)) tmp+=1000;              //西部奖学金
       if (a&&(score2>80)) tmp+=850;               //班级贡献奖
       sum+=tmp;                                   //奖学金的总数
       if (tmp>max) {max=tmp;k=i;}                 //获奖金最高的学生
    }
    printf("%s\n%d\n%d\n",s[k],max,sum);
    return 0;
}
```

【上机练习】

1. 统计数字字符个数【1.7 编程基础之字符串 01】

输入一行字符，统计出其中数字字符的个数。

输入：

一行字符串，总长度不超过 255。

输出：

输出为 1 行，输出字符串里面数字字符的个数。

2. 找第一个只出现一次的字符【1.7 编程基础之字符串 02】

给定一个只包含小写字母的字符串，请你找到第一个仅出现一次的字符。如果没有，输出 no。

输入：

一个字符串，长度小于 100000。

输出：

输出第一个仅出现一次的字符，若没有则输出 no。

3. 基因相关性【1.7 编程基础文字符串 03】

为了获知基因序列在功能和结构上的相似性，经常需要将几条不同序列的 DNA 进行比对，以判断该比对的 DNA 是否具有相关性。

现比对两条长度相同的 DNA 序列。首先定义两条 DNA 序列相同位置的碱基为一个碱基对，如果一个碱基对中的两个碱基相同的话，则称为相同碱基对。接着计算相同碱基对占总碱基对数量的比例，如果该比例大于等于给定阈值时则判定该两条 DNA 序列是相关的，否则不相关。

输入：

有三行，第一行是用来判定出两条 DNA 序列是否相关的阈值，随后 2 行是两条 DNA 序列(长度不大于 500)。

输出：

若两条 DNA 序列相关，则输出“yes”，否则输出“no”。

4. 石头剪子布【1.7 编程基础之字符串 04】

石头剪子布，是一种猜拳游戏。起源于中国，然后传到日本、朝鲜等地，随着亚欧贸易的不断发展它传到了欧洲，到了近现代逐渐风靡世界。简单明了的规则，使得石头剪子布没有任何规则漏洞可钻，单次玩法比拼运气，多回合玩法比拼心理博弈，使得石头剪子布这个古老的游戏同时用于“意外”与“技术”两种特性，深受世界人民喜爱。

游戏规则：石头打剪刀，布包石头，剪刀剪布。

现在，需要你写一个程序来判断石头剪子布游戏的结果。

输入：

第一行是一个整数 N，表示一共进行了 N 次游戏。1 <= N <= 100。

接下来 N 行的每一行包括两个字符串，表示游戏参与者 Player1，Player2 的选择(石头、剪子或者是布)：

S1　S2

字符串之间以空格隔开 S1，S2 只可能取值在{" Rock ", " Scissors ", " Paper "}(大小写

敏感)中。

输出:

输出包括 N 行,每一行对应一个胜利者(Player1 或者 Player2),或者游戏出现平局,则输出 Tie。

5. 输出亲朋字符串【1.7 编程基础之字符串 05】

编写程序,求给定字符串 s 的亲朋字符串 s1。

亲朋字符串 s1 定义如下:给定字符串 s 的第一个字符的 ASCII 值加第二个字符的 ASCII 值,得到第一个亲朋字符;给定字符串 s 的第二个字符的 ASCII 值加第三个字符的 ASCII 值,得到第二个亲朋字符;依此类推,直到给定字符串 s 的倒数第二个字符。亲朋字符串的最后一个字符由给定字符串 s 的最后一个字符 ASCII 值加 s 的第一个字符的 ASCII 值。

输入:

输入一行,一个长度大于等于 2,小于等于 100 的字符串。字符串中每个字符的 ASCII 值不大于 63。

输出:

输出一行,为变换后的亲朋字符串。输入保证变换后的字符串只有一行。

6. 合法 C 标识符【1.7 编程基础之字符串 06】

给定一个不包含空白符的字符串,请判断是否是 C 语言合法的标识符号(注:题目保证这些字符串一定不是 C 语言的保留字)。

C 语言标识符要求:

1. 非保留字;

2. 只包含字母、数字及下划线("_")。

3. 不以数字开头。

输入:

一行,包含一个字符串,字符串中不包含任何空白字符,且长度不大于 20。

输出:

一行,如果它是 C 语言的合法标识符,则输出 yes,否则输出 no。

7. 配对碱基链【1.7 编程基础之字符串 07】

脱氧核糖核酸(DNA)由两条互补的碱基链以双螺旋的方式结合而成。而构成 DNA 的碱基共有 4 种,分别为腺嘌呤(A)、鸟嘌呤(G)、胸腺嘧啶(T)和胞嘧啶(C)。我们知道,在两条互补碱基链的对应位置上,腺嘌呤总是和胸腺嘧啶配对,鸟嘌呤总是和胞嘧啶配对。你的任务就是根据一条单链上的碱基序列,给出对应的互补链上的碱基序列。

输入:

一个字符串,表示一条碱基链。这个字符串只含有大写字母 A、T、G、C,分别表示腺嘌呤、胸腺嘧啶、鸟嘌呤和胞嘧啶。字符串长度不超过 255。

输出:

一个只含有大写字母 A、T、G、C 的字符串,为与输入的碱基链互补的碱基链。

8. 密码翻译【1.7 编程基础之字符串 09】

在情报传递过程中,为了防止情报被截获,往往需要对情报用一定的方式加密,简单的加密算法虽然不足以完全避免情报被破译,但仍然能防止情报被轻易的识别。我们给出一种最简的的加密方法,对给定的一个字符串,把其中从 a-y,A-Y 的字母用其后继字母替

代，把 z 和 Z 用 a 和 A 替代，其他非字母字符不变，则可得到一个简单的加密字符串。

输入：

输入一行，包含一个字符串，长度小于 80 个字符。

输出：

输出每行字符串的加密字符串。

9. 加密的病历单【1.7 编程基础之字符串 12】

小英是药学专业大三的学生，暑假期间获得了去医院药房实习的机会。

在药房实习期间，小英扎实的专业基础获得了医生的一致好评，得知小英在计算概论中取得过好成绩后，主任又额外交给她一项任务，解密抗战时期被加密过的一些伤员的名单。

经过研究，小英发现了如下加密规律（括号中是一个“原文 -> 密文”的例子）

1. 原文中所有的字符都在字母表中被循环左移了三个位置（dec -> abz）

2. 逆序存储（abcd -> dcba ）

3. 大小写反转（abXY -> ABxy）

输入：

一个加密的字符串。（长度小于 50 且只包含大小写字母）

输出：

输出解密后的字符串。

10. 将字符串中的小写字母转换成大写字母【1.7 编程基础之字符串 13】

给定一个字符串，将其中所有的小写字母转换成大写字母。

输入：

输入一行，包含一个字符串（长度不超过 100，可能包含空格）。

输出：

输出转换后的字符串。

11. 整理药名【1.7 编程基础之字符串 15】

医生在书写药品名的时候经常不注意大小写，格式比较混乱。现要求你写一个程序将医生书写混乱的药品名整理成统一规范的格式，即药品名的第一个字符如果是字母要大写，其他字母小写。如将 ASPIRIN、aspirin 整理成 Aspirin。

输入：

第一行一个数字 n，表示有 n 个药品名要整理，n 不超过 100。

接下来 n 行，每行一个单词，长度不超过 20，表示医生手书的药品名。药品名由字母、数字和－组成。

输出：

n 行，每行一个单词，对应输入的药品名的规范写法。

12. 验证子串【1.7 编程基础之字符串 18】

输入两个字符串，验证其中一个串是否为另一个串的子串。

输入：

输入两个字符串，每个字符串占一行，长度不超过 200 且不含空格。

输出：

若第一个串 s1 是第二个串 s2 的子串，则输出（s1） is substring of （s2）

否则，若第二个串 s2 是第一个串 s1 的子串，输出（s2） is substring of （s1）

否则，输出 No substring。

13. 删除单词后缀【1.7 编程基础之字符串 20】

给定一个单词,如果该单词以 er、ly 或者 ing 后缀结尾,则删除该后缀(题目保证删除后缀后的单词长度不为 0),否则不进行任何操作。

输入:

输入一行,包含一个单词(单词中间没有空格,每个单词最大长度为 32)。

输出:

输出按照题目要求处理后的单词。

14. 单词的长度【1.7 编程基础之字符串 24】

输入一行单词序列,相邻单词之间由 1 个或多个空格间隔,请对应计算各个单词的长度。

注意:如果有标点符号(如连字符,逗号),标点符号算作与之相连的词的一部分。没有被空格间开的符号串,都算作单词。

输入:

一行单词序列,最少 1 个单词,最多 300 个单词,单词之间用至少 1 个空格间隔。单词序列总长度不超过 1000。

输出:

依次输出对应单词的长度,之间以逗号间隔。

15. 最长最短单词【1.7 编程基础之字符串 25】

输入 1 行句子(不多于 200 个单词,每个单词长度不超过 100),只包含字母、空格和逗号。单词由至少一个连续的字母构成,空格和逗号都是单词间的间隔。

试输出第 1 个最长的单词和第 1 个最短单词。

输入:

一行句子。

输出:

第 1 行,第一个最长的单词。

第 2 行,第一个最短的单词。

提示:

如果所有单词长度相同,那么第一个单词既是最长单词也是最短单词。

16. 单词翻转【1.7 编程基础之字符串 28】

输入一个句子(一行),将句子中的每一个单词翻转后输出。

输入:

只有一行,为一个字符串,不超过 500 个字符。单词之间以空格隔开。

输出:

翻转每一个单词后的字符串,单词之间的空格需与原文一致。

17. 字符串 p 型编码【1.7 编程基础之字符串 31】

给定一个完全由数字字符('0','1','2',…,'9')构成的字符串 str,请写出 str 的 p 型编码串。例如:字符串 122344111 可被描述为"1 个 1、2 个 2、1 个 3、2 个 4、3 个 1",因此我们说 122344111 的 p 型编码串为 1122132431;类似的道理,编码串 101 可以用来描述 1111111111;00000000000 可描述为"11 个 0",因此它的 p 型编码串即为 110;100200300 可描述为"1 个 1、2 个 0、1 个 2、2 个 0、1 个 3、2 个 0",因此它的 p 型编码串为 112012201320。

输入:

输入仅一行，包含字符串 str。每一行字符串最多包含 1000 个数字字符。

输出：

输出该字符串对应的 p 型编码串。

18. 判断字符串是否为回文【1.7 编程基础之字符串 33】

输入一个字符串，输出该字符串是否回文。回文是指顺读和倒读都一样的字符串。

输入：

输入为一行字符串(字符串中没有空白字符，字符串长度不超过 100)。

输出：

如果字符串是回文，输出 yes；否则，输出 no。

19. 最高分数的学生姓名【1.9 编程基础之顺序查找 02】

输入学生的人数，然后再输入每位学生的分数和姓名，求获得最高分数的学生的姓名。

输入：

第一行输入一个正整数 N(N <= 100)，表示学生人数。接着输入 N 行，每行格式如下：

分数 姓名

分数是一个非负整数，且小于等于 100；

姓名为一个连续的字符串，中间没有空格，长度不超过 20。

数据保证最高分只有一位同学。

输出：

获得最高分数同学的姓名。

20. 连续出现的字符【1.9 编程基础之顺序查找 11】

给定一个字符串，在字符串中找到第一个连续出现至少 k 次的字符。

输入：

第一行包含一个正整数 k，表示至少需要连续出现的次数。1 <= k <= 1000。

第二行包含需要查找的字符串。字符串长度在 1 到 1000 之间，且不包含任何空白符。

输出：

若存在连续出现至少 k 次的字符，输出该字符；否则输出 No。

21. 最长单词 2【1.13 编程基础之综合应用 16】

一个以'.'结尾的简单英文句子，单词之间用空格分隔，没有缩写形式和其它特殊形式。

输入：

一个以'.'结尾的简单英文句子(长度不超过 500)，单词之间用空格分隔，没有缩写形式和其它特殊形式

输出：

该句子中最长的单词。如果多于一个，则输出第一个

第六章 函 数

第一节 函 数

前面我们曾经学习了程序设计中的三种基本控制结构(顺序、分支、循环),用它们可以组成任何程序。但在应用中,还经常用到子程序结构。

通常,在程序设计中,我们会发现一些程序段在程序的不同地方反复出现,此时可以将这些程序段作为相对独立的整体,用一个标识符给它起一个名字,凡是程序中出现该程序段的地方,只要简单地写上其标识符即可。这样的程序段,我们称之为子程序。

子程序的使用不仅缩短了程序,节省了内存空间及减少了程序的编译时间,而且有利于结构化程序设计。因为一个复杂的问题总可将其分解成若干个子问题来解决,如果子问题依然很复杂,还可以将它继续分解,直到每个子问题都是一个具有独立任务的模块。这样编制的程序结构清晰,逻辑关系明确,无论是编写、阅读、调试还是修改,都会带来极大的好处。

在一个程序中可以只有主程序而没有子程序(本章以前都是如此),但不能没有主程序,也就是说不能单独执行子程序。

在此之前,我们曾经介绍并使用了C++提供的各种标准函数,如abs(),sqrt()等等,这些系统提供的函数为我们编写程序提供了很大的方便。比如:求 sin(1)+sin(2)+…+sin(100)的值。但这些函数只是常用的基本函数,编程时经常需要自定义一些函数。

我们来看看下面一个例子:求:1!+2!+3!+…+10!=?

如果要编写程序,我们看到求阶乘的操作要执行10次,只不过每次所求的数不同。我们不至于编写10遍求阶乘的程序吧!我们希望有一个求阶乘的函数,假设为js(x),我们就可以这样求这道题了。

例 6.1 求:1!+2!+3!+…+10!

```
#include<iostream>
using namespace std;
int main()
{
    int sum=0;
    for (int i=1; i<=10; i++)
        sum+=js(i);
    cout<<" sum="<<sum<<endl;
    return 0;
}
```

现在的问题是：C＋＋不提供 js(x)这样一个标准函数，这个程序是通不过的，没关系，我们编写自己的函数。如果是C＋＋的标准函数，可以直接调用，如 abs(x)，sqrt(x)，…而 C＋＋调用的标准函数需要在程序中通过＃include 指令加入相应的库即可。

一、函数的定义

1. 函数定义的语法形式

```
数据类型　函数名(形式参数表)
{
    函数体              //执行语句
}
```

关于函数的定义有如下说明：

· 函数的数据类型是函数的返回值类型(若数据类型为 void ，则无返回值)。

· 函数名是标识符，一个程序中除了主函数名必须为 main 外，其余函数的名字按照标识符的取名规则可以任意选取，最好取有助于记忆的名字。

· 形式参数(简称形参)表可以是空的(即无参函数)，也可以有多个形参，形参间用逗号隔开，不管有无参数，函数名后的圆括号都必须有。形参必须有类型说明，形参可以是变量名、数组名或指针名，它的作用是实现主调函数与被调函数之间的关系。

· 函数中最外层一对花括号“{ }”括起来的若干个说明语句和执行语句组成了一个函数的函数体。由函数体内的语句决定该函数功能。函数体实际上是一个复合语句，它可以没有任何类型说明，而只有语句，也可以两者都没有，即空函数。

· 函数不允许嵌套定义。在一个函数内定义另一个函数是非法的，但是允许嵌套使用。

函数在没有被调用的时候是静止的，此时的形参只是一个符号，它标志着在形参出现的位置应该有一个什么类型的数据。函数在被调用时才执行，也就是在被调用时才由主调函数将实际参数(简称实参)值赋予形参。这与数学中的函数概念相似，如数学函数：

$f(x)=x^2+x+1$

这样的函数只有当自变量被赋值以后，才能计算出函数的值。

2. 函数定义的例子

定义一个函数，返回两个数中的较大数。

```
int max(int x,int y)
{
    return x>y? x:y;
}
```

该函数返回值是整型，有两个整型的形参，用来接受实参传递的两个数据，函数体内的语句是求两个数中的较大者并将其返回主调函数。

3. 函数的形式

函数的形式从结构上说可以分为三种：无参函数、有参函数和空函数。它们的定义形式都相同。

(1)无参函数

无参函数顾名思义即为没有参数传递的函数,无参函数一般不需要带回函数值,所以函数类型说明为 void。

(2)有参函数

有参函数即有参数传递的函数,一般需要带回函数值。例如 int max(int x,int y)函数。

(3)空函数

空函数即函数体只有一对花括号,花括号内没有任何语句的函数。例如,

```
函数名()
{ }
```

空函数不完成什么工作,只占据一个位置。在大型程序设计中,空函数用于扩充函数功能。

编写一个阶乘的函数,我们给此函数取一个名字 js。

```
int js(int n)                    //函数名 js,形参 int n
{                                //{}中是函数的函数体
    int s=1;
    for (int i=1; i<=n;++i)
      s*=i;
    return s;
}
```

在本例中,函数名叫 js,只有一个 int 型的自变量 n,函数 js 属 int 型。在本函数中,要用到两个变量 i,s。在函数体中,是一个求阶乘的语句,n 的阶乘的值在 s 中,最后由 return 语句将计算结果 s 值带回,js()函数执行结束,在主函数中 js()值就是 s 的值。

在这里,函数的参数 n 是一个接口参数,说得更明确点是入口参数。如果我们调用函数:js(3),那么在程序里所有有 n 的地方,n 被替代成 3 来计算。在这里,3 就被称为实参。又如:sqrt(4),ln(5),这里 4,5 叫实参。而 ln(x),sqrt(x)中的 x,y 叫形参。完整的程序如下:

```
#include<iostream>
using namespace std;
int js(int);                     //函数的声明
int main()
{
    int sum=0;
    for (int i=1; i<=10; ++i)
      sum+=js(i);    //函数的调用
    cout<<" sum="<<sum<<endl;
    return 0;
}
int js(int n)                    //定义的函数体
{
    int s=1;
    for (int i=1; i<=n; ++i)
```

```
    s*=i;
  return s;              //函数的返回值
}
```

二、函数的声明和调用

1. 函数的声明

调用函数之前先要声明函数原型。在主调函数中或所有函数定义之前，按如下形式声明：

类型说明符　被调函数名(含类型说明的形参表)；

如果是在所有函数定义之前声明了函数原型，那么该函数原型在本程序文件中任何地方都有效，也就是说在本程序文件中任何地方都可以依照该原型调用相应的函数。如果是在某个主调函数内部声明了被调用函数原型，那么该原型就只能在这个函数内部有效。

下面对 js()函数原型声明是合法的。

```
int js(int n);
```

也可以：

```
int js(int);
```

可以看到函数原型声明与函数定义时的第一行类似，只多了一个分号，成为了一个声明语句而已。

2. 函数的调用

声明了函数原型之后，便可以按如下形式调用函数：

函数名(实参列表)　　//例题中语句 sum+=js(i)；

实参列表中应给出与函数原型形参个数相同、类型相符的实参。在主调函数中的参数称为实参，实参一般应具有确定的值。实参可以是常量、表达式，也可以是已有确定值的变量、数组或指针名。函数调用可以作为一条语句，这时函数可以没有返回值。函数调用也可以出现在表达式中，这时就必须有一个明确的返回值。

3. 函数的返回值

在组成函数体的各类语句中，值得注意的是返回语句 return。它的一般形式是：

return(表达式)；　　// 例题中语句 return s；

其功能是把程序流程从被调函数转向主调函数并把表达式的值带回主调函数，实现函数的返回。所以，在圆括号表达式的值实际上就是该函数的返回值。其返回值的类型即为它所在函数的函数类型。当一个函数没有返回值时，函数中可以没有 return 语句，直接利用函数体的右花括号"}"，作为没有返回值的函数的返回。也可以有 return 语句，但 return 后没有表达式。返回语句的另一种形式是：

```
return;
```

这时函数没有返回值，而只把流程转向主调函数。

三、函数的传值调用

函数传值调用的特点是将调用函数的实参表中的实参值依次对应地传递给被调用函数

的形参表中的形参。要求函数的实参与形参个数相等,并且类型相同。函数的调用过程实际上是对栈空的操作过程,因为调用函数是使用栈空间来保存信息的。函数在返回时,如果有返回值,则将它保存在临时变量中。然后恢复主调函数的运行状态,释放被调用函数的栈空间,按其返回地址返回到调用函数。

在C++语言中,函数调用方式分传值调用和传址调用。

1.传值调用

这种调用方式是将实参的数据值传递给形参,即将实参值拷贝一个副本存放在被调用函数的栈区中。在被调用函数中,形参值可以改变,但不影响主调函数的实参值。参数传递方向只是从实参到形参,简称**单向值传递**。举个例子:

```
#include<iostream>
using namespace std;
void swap(int a,int b)
{
    int tmp=a;a=b;b=tmp;
}
int main()
{
    int c=1,d=2;
    swap(c,d);
    cout<<c<<' '<<d<<endl;
    return 0;
}                                    //程序输出为:1 2
```

在此例中,虽然在swap函数中交换了a,b两数的值,但是在main中却没有交换。因为swap函数只是交换c,d两变量副本的值,实参值没有改变,并没有达到交换的目的。

2.传址调用

这种调用方式是将实参变量的地址值传递给形参,这时形参是指针,即让形参的指针指向实参地址,这里不再是将实参拷贝一个副本给形参,而是让形参直接指向实参,这就提供了一种可以改变实参变量的值的方法。现在用传址调用来实现swap:

```
#include<iostream>
using namespace std;
void swap(int &a,int &b)             //定义swap()函数,形参是传址调用
{
    int tmp=a;a=b;b=tmp;
}
int main()
{
    int c=1,d=2;
    swap(c,d);                       //交换变量
    cout<<c<<' '<<d<<endl;
    return 0;
```

```
}  //程序输出为:2 1
```

在此例中,因为 swap 函数的参数为传址调用,&a 是指实参变量的地址值传递给形参,所以,在函数 swap 中修改 a,b 的值相当于在主函数 main 中修改 c,d 的值。

四、函数的应用举例

例 6.2　计算组合数 C(m,n)的值(n<=m<=10)。

【分析】组合数 C(m,n)可以理解为从 m 个数中任意取出 n 个数的所有情况数。求这个数值,有一个经典的计算方法:C(m,n)=m! /((m−n)! n!)。

程序如下:

```
#include<cstdio>
using namespace std;
int fac(int x);                                    //阶乘函数的声明
int main()
{
    int m,n;
    scanf("%d%d",&m,&n);
    printf("%d",fac(m)/(fac(m-n)*fac(n)));         //阶乘函数的调用
}

int fac(int x)                                     //定义阶乘函数
{
    int s=1;
    for (int i=1; i<=x; i++)
        s*=i;
    return s;                                      //阶乘函数的值返回
}
```

例 6.3　计算如图多边形的面积。

从图中可以看出,五边形的面积是三个三角形面积之和。

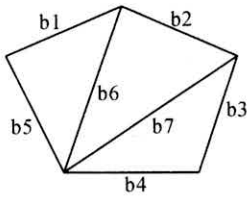


图 6−1

程序如下:

```
#include<iostream>
#include<cstdio>                                   //使用 printf 和 scanf 语句,调用 cstdio 库
#include<cmath>
using namespace std;
```

```
double area(double,double,double);
int main()
{
  double b1,b2,b3,b4,b5,b6,b7,s;
  cout<<" please input b1,b2,b3,b4,b5,b6,b7:"<<endl;
  cin>>b1>>b2>>b3>>b4>>b5>>b6>>b7;
  s=area(b1,b5,b6)+area(b2,b6,b7)+area(b3,b4,b7);  //调用三次函数 area
  printf(" s=%10.3lf\n ",s);
  return 0;
}
double area(double a,double b,double c)    //此函数为海伦-秦九韶公式
{
  double p=(a+b+c)/2;
  return sqrt(p*(p-a)*(p-b)*(p-c));
}
```

例 6.4 定义一个函数 check(n,d),让它返回一个布尔值。如果数字 d 在正整数 n 的某位中出现则送回 true,否则送回 false。

例如:check(325719,3)==true;check(77829,1)==false;

程序如下:

```
#include<iostream>
using namespace std;
bool check(int,int);
int main()
{
    int a,b;
    cout<<" input n,d "<<endl;
    cin>>a>>b;
    if (check(a,b)==true) cout<<" true "<<endl;
      else cout<<" false "<<endl;
    return 0;
}
bool check(int n,int d)
{
  while (n)                          //C++中非 0 为真
  {
      int e=n%10;
      n/=10;
      if (e==d) return true;
  }
  return false;
```

```
}
```

例 6.5　用冒泡法对数组元素按由小到大排序。(数组作为函数参数)

程序如下

```
#include<iostream>
using namespace std;
void bubble(int[],int);          //相当于 void bubble(int a[],int n);
int main()
{
    int array[10]={11,4,55,6,77,8,9,0,7,1};          //大数组应开为全局变量
    cout<<"排序前 ";
    for (int i=0; i<10;++i)
      cout<<array[i]<<',';
    cout<<endl;
    bubble(array,10);
    cout<<"排序后 ";
    for (int i=0; i<10; ++i)
      cout<<array[i]<<',';
    cout<<endl;
    return 0;
}
void bubble(int a[],int n)
{
    for (int i=1; i<n;++i)
    {
      for (int j=0; j<n-i;++j)
        if (a[j]>a[j+1])                    //判断并交换变量
        {
          int temp=a[j]; a[j]=a[j+1]; a[j+1]=temp;
        }
    }
}
```

在前面我们已经知道数组名是该数组在内存的首地址。将数组名作为参数传给函数，实际上是把数组的地址传给函数。形参数组和实参数组的首地址重合，因此在被调用函数中对数组元素值进行改变，主调函数中实参数组的相应元素值也会改变。

五、全程变量、局部变量及它们的作用域

在函数外部定义的变量称为外部变量或全局变量，在函数内部定义的变量称为内部变量或局部变量。

1. 全局变量

定义在函数外部没有被花括号括起来的变量称为全局变量,全局变量的作用域是从变量定义的位置开始到文件结束。由于全局变量是在函数外部定义的,因此对所有函数而言都是外部的,可以在文件中位于全局变量定义后面的任何函数中使用。

例 6.6 输入两个正整数,编程计算两个数的最小公倍数。

程序如下:

```
#include<iostream>
using namespace std;
int x,y;                                    //定义全局变量 x,y
int gcd(int x,int y)                        //求最大公约数,形参 x,y 是局部变量
{
    int r=x%y;                              //r 是局部变量,局部变量 x,y 屏蔽全局变量
    while(r! =0)
    {
        x=y;
        y=r;
        r=x%y;
    }
    return y;
}

int lcm()                                   //求最小公倍数
{
    return x * y/gcd(x,y);                  //这里 x,y 是全局变量
}

int main()                                  //gcd 和 lcm 在 main()前,函数不必声明
{
    cin>>x>>y;                              //这里 x,y 是全局变量
    cout<<lcm()<<endl;
    return 0;
}
```

运行结果:

输入:12 18

输出:36

使用全局变量的说明:

· 在一个函数内部,既可以使用本函数定义的局部变量,也可以使用在此函数前定义的全局变量。

· 全局变量的作用是使得函数间多了一种传递信息的方式。如果在一个程序中多个函数都要对同一个变量进行处理,即共享,就可以将这个变量定义成全局变量,使用非常方

便，但副作用也不可低估。

• 过多地使用全局变量，会增加调试难度。因为多个函数都能改变全局变量的值，不易判断某个时刻全局变量的值。

• 过多地使用全局变量，会降低程序的通用性。如果将一个函数移植到另一个程序中，需要将全局变量一起移植过去，同时还有可能出现重名问题。

• 全局变量在程序执行的全过程中一直占用内存单元。

• 全局变量在定义时若没有赋初值，其默认值为0。

2. 局部变量

(1)局部变量的作用域是在定义该变量的函数内部。换句话说，局部变量只在定义它的函数内有效。函数的形参也是局部变量。局部变量的存储空间是临时分配的，当函数执行完毕，局部变量的空间就被释放，其中的值无法保留到下次使用。

(2)由于局部变量的作用域仅局限于本函数内部，所以，在不同的函数中变量名可以相同，它们分别代表不同的对象，在内存中占据不同的内存单元，互不干扰。

(3)一个局部变量和一个全局变量是可以重名的，在相同的作用域内局部变量有效时全局变量无效，即局部变量可以屏蔽全局变量。

(4)在代码块中定义的变量的存在时间和作用域将被限制在该代码块中。如for(int i;i<=n;i++){sum+=i}中的i是在该for循环语句中定义的，存在时间和作用域只能被限制在该for循环语句中。

(5)这里需要强调的是，主函数main中定义的变量也是局部变量，这一点与其他程序设计语言不同。

(6) 全局变量数组初始全部为0，局部变量值是随机的，要初始化初值，局部变量受栈空间大小限制，大数组需要注意。通俗说，局部变量的数组不能开很大，全局变量随便。

六、函数的综合应用

例 6.7　写一个判断素数的函数，输入一个数，判断它是否是素数，是输出yes，不是输出no。

【分析】对于任意整数i，根据素数定义，我们从2开始，到sqrt(i)，找i的第一个约数，若找到第一个约数，则i必然不是素数。

程序如下：

```
#include<cstdio>
#include<cmath>
int prime(int x);                    //对于函数的声明
int main()
{
  int n;
  scanf("%d",&n);
  if (prime(n))
    printf("%s\n","yes");
  else
```

```
    printf("%s\n","no");
  return 0;
}

int prime(int x)                          //判断 x 是否素数的函数
{
  int j;
  if (x==2) return 1;
  j=2;
  while(j<=sqrt(x) && x%j! =0) j++;
  if (x%j == 0)
    return 0;
  else
    return 1;
}
```

例 6.8 编程输入十进制整数 N(N:-32767～32767),请输出它对应的二进制、八进制、十六进制数。

【分析】 这是一道进行数制转换的问题,将十进制整数转换成 R 进制的数,算法是:除以 R 取余,再将余数倒过来写出即是 R 进制的数。本例是要求把一个十进制数同时转换成二进制、八进制、十六进制数。因此可以设计一个过程同时处理这三种的进制转换。

程序如下:

```
#include<cstdlib>
#include<iostream>
using namespace std;
void TurnData(int n,int a);
char ch[6]={'A','B','C','D','E','F'};
int main()
{
    int n;
    cin>>n;
    TurnData(n,2);                        //n 转成 2 进制数
    TurnData(n,8);                        //n 转成 8 进制数
    TurnData(n,16);                       //n 转成 16 进制数
    return 0;
}
void TurnData(int n,int a)
{
    int x[17],i,j,k=0;
    cout<<n<<" turn into "<<a<<"  :"<<endl;
    if (n<0) cout<<'-';                   //负数的话,先输出负号再开始转
```

```
    j=abs(n);
    do
    {
        k++;                                    //用于统计转成 a 进制数后的总位数
        i=j%a;
        j/=a;
        x[k]=i;
    }while (j!=0);
    for (int h=k; h>=1;--h)
      if (x[h]<10) cout<<x[h];
        else  cout<<ch[x[h]-10];
    cout<<endl;
}
```

这里的过程 TurnData 中的参数不需要把什么值返回给主程序，因此设为形参即可。

例 6.9　编写一个给一个分数约分的程序。

程序如下：

```
#include<iostream>
#include<iomanip>
using namespace std;
void common(int x,int y);
int main()
{   int a,b;
    cin>>a>>b;
    common(a,b);
}
void common(int x,int y)
{   int m=x,n=y,r;                              //用辗转相除法求 x、y 的最大公约数
    do
    {     r=m%n;
          m=n;
          n=r;
    }while (r!=0);
    x/=m;                                       //用两者的最大公约数 m 对 x、y 进行约分
    y/=m;
    cout<<setw(5)<<x<<setw(5)<<y<<endl;
}
```

运行结果：

输入：12 8

输出：3 2

【课堂练习】

1. 求正整数 2 和 100 之间的完全数。

完全数:因子之和等于它本身的自然数,如 6=1+2+3

2. 编程求 2～n(n 为大于 2 的正整数)中有多少个素数。

3. 已知 $m=\frac{\max(a,b,c)}{\max(a+b,b,c)\times\max(a,b,b+c)}$,输入 a,b,c,求 m。把求三个数的最大数 max(x,y,z)分别定义成函数和过程来做。

4. 如果一个自然数是素数,且它的数字位置经过对换后仍为素数,则称为绝对素数,例如 13。试求出所有二位绝对素数。

5. 自然数 a 的因子是指能被 a 整除的所有自然数,但不含 a 本身。例如 12 的因子为:1,2,3,4,6。若自然数 a 的因子之和为 b,而且 b 的因子之和又等于 a,则称 a,b 为一对“亲和数”。求最小的一对亲和数(a<>b)。

6. 如果一个数从左边读和从右边读都是同一个数,就称为回文数。例如 6886 就是一个回文数,丘出所有的既是回文数又是素数的三位数。

7. 根据公式 $\arctan x(x)=x-x^3/3+x^5/5-x^7/7+\cdots$ 和 $\pi=6\ \arctan x(\frac{1}{\sqrt{3}})$,定义函数 arctanx(x),求当最后一项小于 10^{-6} 时 π 的值。

8. 哥德巴赫猜想的命题之一是:大于 6 的偶数等于两个素数之和。编程将 6～100 所有偶数表示成两个素数之和。

【上机练习】

1. 简单算术表达式求值【1.12 编程基础之函数与过程抽象 01】

两位正整数的简单算术运算(只考虑整数运算),算术运算为:

+,加法运算;

-,减法运算;

*,乘法运算;

/,整除运算;

%,取余运算。

算术表达式的格式为(运算符前后可能有空格):

运算数 运算符 运算数

请输出相应的结果。

输入:

一行算术表达式。

输出:

整型算数运算的结果(结果值不一定为 2 位数,可能多于 2 位或少于 2 位)。

2. 短信计费【1.12 编程基础之函数与过程抽象 02】

用手机发短信,一条短信资费为 0.1 元,但限定一条短信的内容在 70 个字以内(包括 70

个字)。如果你一次所发送的短信超过了70个字,则会按照每70个字一条短信的限制把它分割成多条短信发送。假设已经知道你当月所发送的短信的字数,试统计一下你当月短信的总资费。

输入:

第一行是整数n,表示当月发送短信的总次数,接着n行每行一个整数,表示每次短信的字数。

输出:

输出一行,当月短信总资费,单位为元,精确到小数点后1位。

3. 甲流病人初筛【1.12 编程基础之函数与过程抽象 03】

目前正是甲流盛行时期,为了更好地进行分流治疗,医院在挂号时要求对病人的体温和咳嗽情况进行检查,对于体温超过37.5度(含等于37.5度)并且咳嗽的病人初步判定为甲流病人(初筛)。现需要统计某天前来挂号就诊的病人中有多少人被初筛为甲流病人。

输入:

第一行是某天前来挂号就诊的病人数n。(n<200)

其后有n行,每行是病人的信息,包括三个信息:姓名(字符串,不含空格,最多8个字符)、体温(float)、是否咳嗽(整数,1表示咳嗽,0表示不咳嗽)。每行三个信息之间以一个空格分开。

输出:

按输入顺序依次输出所有被筛选为甲流的病人的姓名,每个名字占一行。之后在输出一行,表示被筛选为甲流的病人数量。

4. 统计单词数【1.12 编程基础之函数与过程抽象 05】Noip2011 普及组第2题

一般的文本编辑器都有查找单词的功能,该功能可以快速定位特定单词在文章中的位置,有的还能统计出特定单词在文章中出现的次数。

现在,请你编程实现这一功能,具体要求是:给定一个单词,请你输出它在给定的文章中出现的次数和第一次出现的位置。注意:匹配单词时,不区分大小写,但要求完全匹配,即给定单词必须与文章中的某一独立单词在不区分大小写的情况下完全相同(参见样例1),如果给定单词仅是文章中某一单词的一部分则不算匹配(参见样例2)。

输入:

第1行为一个字符串,其中只含字母,表示给定单词;

第2行为一个字符串,其中只可能包含字母和空格,表示给定的文章。

输出:

只有一行,如果在文章中找到给定单词则输出两个整数,两个整数之间用一个空格隔开,分别是单词在文章中出现的次数和第一次出现的位置(即在文章中第一次出现时,单词首字母在文章中的位置,位置从0开始);如果单词在文章中没有出现,则直接输出一个整数-1。

5. 机器翻译【1.12 编程基础之函数与过程抽象 07】Noip2010 提高组第1题

小晨的电脑上安装了一个机器翻译软件,他经常用这个软件来翻译英语文章。

这个翻译软件的原理很简单,它只是从头到尾,依次将每个英文单词用对应的中文含义来替换。对于每个英文单词,软件会先在内存中查找这个单词的中文含义,如果内存中有,软件就会用它进行翻译;如果内存中没有,软件就会在外存中的词典内查找,查出单词的中

文含义然后翻译,并将这个单词和译义放入内存,以备后续的查找和翻译。

假设内存中有 M 个单元,每单元能存放一个单词和译义。每当软件将一个新单词存入内存前,如果当前内存中已存入的单词数不超过 M-1,软件会将新单词存入一个未使用的内存单元;若内存中已存入 M 个单词,软件会清空最早进入内存的那个单词,腾出单元来,存放新单词。

假设一篇英语文章的长度为 N 个单词。给定这篇待译文章,翻译软件需要去外存查找多少次词典?假设在翻译开始前,内存中没有任何单词。

输入:

输入文件共 2 行。每行中两个数之间用一个空格隔开。

第一行为两个正整数 M 和 N,代表内存容量和文章的长度。

第二行为 N 个非负整数,按照文章的顺序,每个数(大小不超过 1000)代表一个英文单词。文章中两个单词是同一个单词,当且仅当它们对应的非负整数相同。

输出:

共 1 行,包含一个整数,为软件需要查词典的次数。

6. Vigenère 密码【1.12 编程基础之函数与过程抽象 08】Noip2012 提高组第 1 题

16 世纪法国外交家 Blaise de Vigenère 设计了一种多表密码加密算法——Vigenère 密码。Vigenère 密码的加密解密算法简单易用,且破译难度比较高,曾在美国南北战争中为南军所广泛使用。

在密码学中,我们称需要加密的信息为明文,用 M 表示;称加密后的信息为密文,用 C 表示;而密钥是一种参数,是将明文转换为密文或将密文转换为明文的算法中输入的数据,记为 k。在 Vigenère 密码中,密钥 k 是一个字母串,k=k1k2…kn。当明文 M=m1m2…mn 时,得到的密文 C=c1c2…cn,其中 ci=mi®ki,运算®的规则如下表所示:

Vigenère 加密在操作时需要注意:

1. ®运算忽略参与运算的字母的大小写,并保持字母在明文 M 中的大小写形式;
2. 当明文 M 的长度大于密钥 k 的长度时,将密钥 k 重复使用。

例如,明文 M=Helloworld,密钥 k=abc 时,密文 C=Hfnlpyosnd。

明文 H e l l o w o r l d

密钥 a b c a b c a b c a

密文 H f n l p y o s n d

输入:

第一行为一个字符串,表示密钥 k,长度不超过 100,其中仅包含大小写字母。第二行为一个字符串,表示经加密后的密文,长度不超过 1000,其中仅包含大小写字母。

对于 100%的数据,输入的密钥的长度不超过 100,输入的密文的长度不超过 1000,且都仅包含英文字母。

输出:

输出共 1 行,一个字符串,表示输入密钥和密文所对应的明文。

7. 素数对【1.12 编程基础之函数与过程抽象 10】

两个相差为 2 的素数称为素数对,如 5 和 7,17 和 19 等,本题目要求找出所有两个数均不大于 n 的素数对。

输入:

一个正整数 n。1<=n<=10000。

输出：

所有小于等于 n 的素数对。每对素数对输出一行，中间用单个空格隔开。若没有找到任何素数对，输出 empty。

8. 我家的门牌号【小学奥数 7649】

我家住在一条短胡同里，这条胡同的门牌号从 1 开始顺序编号。

若其余各家的门牌号之和减去我家门牌号的两倍，恰好等于 n，求我家的门牌号及总共有多少家。数据保证有唯一解。

输入：

一个正整数 n。n<100000。

输出：

一行，包含两个正整数，分别是我家的门牌号及总共有多少家，中间用单个空格隔开。

9. 质数的和与积【小学奥数 7827】

两个质数的和是 S，它们的积最大是多少？

输入：

一个不大于 10000 的正整数 S，为两个质数的和。

输出：

一个整数，为两个质数的最大乘积。数据保证有解。

10. 单词替换【1.7 编程基础之字符串 21】

输入一个字符串，以回车结束（字符串长度<=100）。该字符串由若干个单词组成，单词之间用一个空格隔开，所有单词区分大小写。现需要将其中的某个单词替换成另一个单词，并输出替换之后的字符串。

输入：

第 1 行是包含多个单词的字符串 s；

第 2 行是待替换的单词 a（长度 <= 100）；

第 3 行是 a 将被替换的单词 b（长度 <= 100）。

s，a，b 最前面和最后面都没有空格。

输出：

输出只有 1 行，将 s 中所有单词 a 替换成 b 之后的字符串。

11. 笨小猴【1.9 编程基础之顺序查找 06】Noip2008 提高组第 1 题

笨小猴的词汇量很小，所以每次做英语选择题的时候都很头疼。但是他找到了一种方法，经试验证明，用这种方法去选择选项的时候选对的几率非常大！

这种方法的具体描述如下：假设 maxn 是单词中出现次数最多的字母的出现次数，minn 是单词中出现次数最少的字母的出现次数，如果 maxn－minn 是一个质数，那么笨小猴就认为这是个 Lucky Word，这样的单词很可能就是正确的答案。

输入：

只有一行，是一个单词，其中只可能出现小写字母，并且长度小于 100。

输出：

共两行，第一行是一个字符串，假设输入的的单词是 Lucky Word，那么输出“Lucky Word”，否则输出“No Answer”；

第二行是一个整数,如果输入单词是 Lucky Word,输出 maxn－minn 的值,否则输出 0。

12. 素数回文数的个数【1.13 编程基础之综合应用 05】

求 11 到 n 之间(包括 n),既是素数又是回文数的整数有多少个。

输入:

一个大于 11 小于 1000 的整数 n。

输出:

11 到 n 之间的素数回文数个数。

13. 判决素数个数【1.13 编程基础之综合应用 10】

输入两个整数 X 和 Y,输出两者之间的素数个数(包括 X 和 Y)。

输入:

两个整数 X 和 Y($1 <= X, Y <= 10^5$)。

输出:

输出一个整数,表示 X,Y 之间的素数个数(包括 X 和 Y)。

14. 最大质因子序列【1.13 编程基础之综合应用 21】

任意输入两个正整数 m,n($1<m<n<=5000$),依次输出 m 到 n 之间每个数的最大质因子(包括 m 和 n;如果某个数本身是质数,则输出这个数自身)。

输入:

一行,包含两个正整数 m 和 n,其间以单个空格间隔。

输出:

一行,每个整数的最大质因子,以逗号间隔。

15. 区间内的真素数【1.13 编程基础之综合应用 23】

找出正整数 M 和 N 之间(N 不小于 M)的所有真素数。

真素数的定义:如果一个正整数 P 为素数,且其反序也为素数,那么 P 就为真素数。

例如,11,13 均为真素数,因为 11 的反序还是为 11,13 的反序为 31 也为素数。

输入:

输入两个数 M 和 N,空格间隔,$1<=M<=N<=100000$。

输出:

按从小到大输出 M 和 N 之间(包括 M 和 N)的真素数,逗号间隔。如果之间没有真素数,则输出 No。

16. 二进制分类【1.13 编程基础之综合应用 36】

若将一个正整数化为二进制数,在此二进制数中,我们将数字 1 的个数多于数字 0 的个数的这类二进制数称为 A 类数,否则就称其为 B 类数。

例如:

$(13)_{10}=(1101)_2$,其中 1 的个数为 3,0 的个数为 1,则称此数为 A 类数;

$(10)_{10}=(1010)_2$,其中 1 的个数为 2,0 的个数也为 2,称此数为 B 类数;

$(24)_{10}=(11000)_2$,其中 1 的个数为 2,0 的个数为 3,则称此数为 B 类数;

程序要求:求出 1～1000 之中(包括 1 与 1000),全部 A、B 两类数的个数。

输入:

无。

输出:

一行，包含两个整数，分别是 A 类数和 B 类数的个数，中间用单个空格隔开。

17. 确定进制【1.13 编程基础之综合应用 34】

$6*9=42$ 对于十进制来说是错误的，但是对于 13 进制来说是正确的。即，$6_{(13)} * 9_{(13)} = 42_{(13)}$，而 $42_{(13)}=4*13^1+2*13^0=54_{(10)}$。

你的任务是写一段程序，读入三个整数 p、q 和 r，然后确定一个进制 B(2<=B<=16) 使得 p * q = r。如果 B 有很多选择，输出最小的一个。

例如：p=11，q=11，r=121. 则有 $11_{(3)} * 11_{(3)} = 121_{(3)}$ 因为 $11_{(3)} = 1 * 3^1 + 1 * 3^0 = 4_{(10)}$ 和 $121_{(3)}=1*3^2+2*3^1+1*3^0=16_{(10)}$。对于进制 10，同样有 $11_{(10)} * 11_{(10)} = 121_{(10)}$。这种情况下，应该输出 3。如果没有合适的进制，则输出 0。

输入：

一行，包含三个整数 p、q、r。p、q、r 的所有位都是数字，并且 1 <= p、q、r <= 1,000,000。

输出：

一个整数：即使得 p*q=r 成立的最小的 B。如果没有合适的 B，则输出 0。

第二节　递归算法

一、递归概念

当函数的定义中,其内部操作又直接或间接地出现对自身的调用,则称这样的程序嵌套定义为递归定义。

递归通常把一个大型复杂的问题层层转化为一个与原问题相似的规模较小的问题来求解,递归策略只需少量的程序就可描述出解题过程所需要的多次重复计算,大大地减少了程序的代码量。递归的能力在于用有限的语句来定义对象的无限集合。用递归思想写出的程序往往十分简洁易懂。

例如,在数学上,所有偶数的集合可递归地定义为:

①0 是一个偶数;

②一个偶数与 2 的和是一个偶数。

可见,仅需两句话就能定义一个由无穷多个元素组成的集合。在程序中,递归是通过函数的调用来实现的。函数直接调用其自身,称为直接递归;函数间接调用其自身,称为间接递归。

二、递归应用

例 6.10　用递归算法求 x^n。

【分析】　把 x^n 分解成:$x^0=1$　　$(n=0)$

$x^1=x*x^0$ $(n=1)$

$x^2=x*x^1$ $(n>1)$

$x^3=x*x^2$ $(n>1)$

…　　$(n>1)$

因此将 x^n 转化为:$x*x^{n-1}$,其中求 x^{n-1} 又用求 x^n 的方法进行求解。

①定义子程序 xn(int n)求 x^n;如果 n>=1,则递归调用 xn(n-1) 求 x^{n-1};

②当递归调用到达 n=0 时终止调用,然后执行本"层"的后继语句;

③遇到子程序运行完后,就结束本次的调用,返回到上一"层"调用语句的地方,并执行其后继语句;

④继续执行步骤③,从调用中逐"层"返回,最后返回到主程序。

采用函数编写程序如下:

```
#include<iostream>
using namespace std;
int xn(int);
int x;
int main()
```

```
{
  int n;
  cin>>x>>n;
  cout<<x<<'^'<<n<<"="<<xn(n)<<endl;
  return 0;
}
int xn(int n)
{
  if (n==0) return 1;                    //递归边界
  else return x * xn(n-1);               //递归式
}
```

采用全局变量编写程序如下：

```
#include<iostream>
using namespace std;
int tt,x;                                //利用全局变量 tt 传递结果
int xn(int);
int main()
{
  int n;
  cin>>x>>n;
  xn(n);
  cout<<x<<'^'<<n<<'='<<tt<<endl;
  return 0;
}
int xn(int n)
{
  if (n==0) tt=1;
  else
  {
      xn(n-1);                           //递归调用过程 xn(n-1)求 x^(n-1)
      tt * =x;
  }
}
```

例 6.11　用递归函数求 x!

$$x! = \begin{cases} 1 & (x=0) \\ x(x-1)! & (x>0) \end{cases}$$

【分析】　根据数学中的定义把求 x! 定义为求 x*(x−1)!，其中求(x−1)! 仍采用求 x! 的方法，需要定义一个求 x! 的函数，逐级调用此函数，即：

当 x=0 时，x!=1；当 x>0 时，x!=x*(x−1)!。

假设用函数 Fac(x)表示 x 的阶乘，当 x=3 时，Fac(3)的求解方法可表示为：

Fac(3)=3*fac(2)=3*2*Fac(1)=3*2*1*Fac(0)=3*2*1*1=6

①定义函数:int fac(int n)

如果 n=0,则 fac=1;如果 n>0,则继续调用函数 fac=n*fac(n-1);

②返回主程序,打印 fac(x)的结果。

它的执行流程如图 6-6 所示:

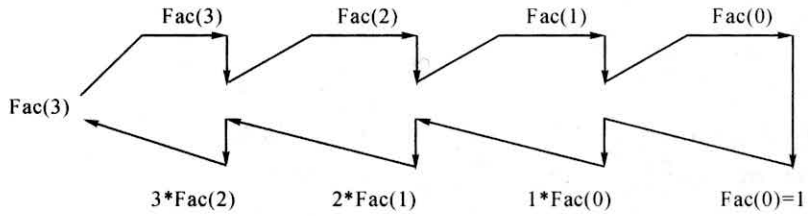


图 6-2

采用有参函数编写程序如下:

```
#include<iostream>
using namespace std;
int fac(int);
int main()
{
    int x;
    cin>>x;
    cout<<x<<"!="<<fac(x)<<endl;   //主程序调用 fac(x) 求 x !
    return 0;
}
int fac(int n)                     //函数 fac(n) 求 n !
{
    return n==0 ? 1 : n*fac(n-1);  //调用函数 fac(n-1)递归求(n-1)!
}
```

【说明】

这里出现了一个小东西,三元运算符"?:"。a? b:c 的含义是:如果 a 为真,则表达式的值是 b,否则是 c。所以 n==0 ? 1 : n*fac(n-1)很好地表达了刚才的递归定义。

采用全局变量编写程序如下:

```
#include<iostream>
using namespace std;
int t;
int fac(int);
int main()
{
    int x;
    cin>>x;
    fac(x);
    cout<<t<<endl;
```

```
    return 0;
}
int fac(int x)
{   if (x==1) t=1;
    else {
            fac(x-1);
            t*=x;
         }
}
```

例 6.12　用递归方法求两个数 m 和 n 的最大公约数。(m>0,n>0)

【方法 1】　求两个数的最大公约数,可以用枚举因子的方法,从两者中较小的数枚举到能被两个数同时整除且是最大的约数的方法;也可以用辗转相除法,这里采用递归实现辗转相除算法:

①求 m 除以 n 的余数;

②如果余数不为 0,则让 m=n,n=余数,重复步骤①,即调用子程序;

③如果余数为 0,则终止调用子程序;

④输出此时的 n 值。

程序如下:

```
#include<iostream>
using namespace std;
int gcd(int,int);
int main()
{
    int m,n;
    cin>>m>>n;
    cout<<" gcd="<<gcd(m,n)<<endl;
    return 0;
}
int gcd(int m,int n)
{
    return n==0? m:gcd(n,m%n);
}
```

【方法 2】二进制最大公约数算法

(1)递归终止条件:gcd(m,m)=m。

(2)递归关系式:

m<n 时:gcd(m,n)=gcd(n,m)

m 为偶数,n 为偶数:gcd(m,n)=2*gcd(m/2,n/2)

m 为偶数,n 为奇数:gcd(m,n)=gcd(m/2,n)

m 为奇数,n 为偶数:gcd(m,n)=gcd(m,n/2)

m 为奇数,n 为奇数:gcd(m,n)=gcd(n,m-n)

该方法和方法1相比更适合求高精度数的最大公约数,因为只涉及除2和减法操作,而辗转相除法则需要用到高精度除法。

程序如下:

```
#include<iostream>
using namespace std;
int gcd(int m,int n)
{
    if(m==n)return m;                                    //递归终止条件
    if(m<n)return gcd(n,m);
    if(m%2==0)
    {
        if(n%2==0)return 2*gcd(m/2,n/2);                 //m为偶数,n为偶数
        else return gcd(m/2,n);                          //m为偶数,n为奇数
    }
    else
    {
        if(n%2==0)return gcd(m,n/2);                     //m为奇数,n为偶数
        else return gcd(n,m-n);                          //m为奇数,n为奇数
    }
}
int main()
{
    int m,n;
    cin>>m>>n;
    cout<<gcd(m,n)<<endl;
    return 0;
}
```

例6.13 已知一个一维数组a[1..n](n<25),又已知一整数m。如能使数组a中任意几个元素之和等于m,则输出YES,反之则为NO。

【分析】 对于一个已确定的数组a[1..n]和一个确定的数m,判断能否使数组a中任意几个元素之和等于m,等价于判断能否从数组a中取任意数使其和为m。

对于a中任意元素a[n]只有取与不取两种情况:

(1)取a[n]:

则此时问题转化为:对于一个已确定的数组a[1..n-1]和一个确定的数m-a[n],判断能否使数组a[1..n-1]中任意几个元素之和等于m-a[n]。

(2)不取a[n]:

则此时问题转化为:对于一个已确定的数组a[1..n-1]和一个确定的数m,判断能否使数组a[1..n-1]中任意几个元素之和等于m。

若用函数sum(n,m)表示能否从数组a[1..n]中取任意数使其和为m,只要sum(n-1,m-a[n])和sum(n-1,m)当中有一个值为真,则sum(n,m)为真,否则为假。因此,可以用

递归来解此题。

递归终止条件为：if (a[n]==m) sum=true; else if (n==1) sum=false;

采用全局变量编写程序如下：

```
#include<iostream>
using namespace std;
const int max1=51;
int a[max1],n,m;
bool flag;
void sum(int,int);
int main()
{
  cin>>n;
  for (int i=1; i<=n;++i) cin>>a[i];
  cin>>m;
  flag=false;
  sum(n,m);
  if (flag) cout<<" YES "<<endl;
      else   cout<<" NO "<<endl;
  return 0;
}
void sum(int n,int m)
{
  if (a[n]==m) flag=true;                    //利用全局变量 flag 传递结果
  else if (n==1) return;                     //n=1 作为递归边界,不再递归下去
  else                                       //进行两种选择
  {
      sum(n-1,m-a[n]);
      sum(n-1,m);
  }
}
```

简单地说，递归算法的本质就是自己调用自己，用调用自己的方法去处理问题，可使解决问题变得简洁明了。

(1)递归程序在执行过程中，一般具有如下模式：

①将调用程序的返回地址、相应的调用前的变量都保存在系统堆栈中；

②执行被调用的函数；

③若满足退出递归的条件，则退出递归，并从栈顶上弹回返回地址、取回保存起来的变量值，继续沿着返回地址，向下执行程序；

④否则继续递归调用，只是递归调用的参数发生变化：增加一个量或减少一个量，重复执行直到递归调用结束。

(2)能够用递归算法解决的问题，一般满足如下要求：

①需要求解的问题可以化为子问题求解,其子问题的求解方法与原问题相同,只是规模上的增加或减少;

②递归调用的次数是有限的;必须有递归结束的条件(称为递归边界)。

【课堂练习】

1. 用递归的方法求 1+2+3+……+N 的值。
2. 用递归函数输出斐波那契数列第 n 项。0,1,1,2,3,5,8,13……
3. 输入一个非负整数,输出这个数的倒序数。例如输入 123,输出 321。
4. 用递归算法将一个十进制数 X 转换成任意进制数 M(M<=16)。
5. 输入一串以'!'结束的字符,按逆序输出。

【上机练习】

1. 阿克曼(Ackmann)函数 A(m,n)中,m,n 定义域是非负整数(m<=3,n<=10),函数值定义为:

akm(m,n) = n+1; (m=0 时)

akm(m,n) = akm(m−1,1); (m>0,n=0 时)

akm(m,n) = akm(m−1,akm(m, n−1)); (m,n>0 时)

2. 在程序中定义一函数 digit(n,k),它能分离出整数 n 从右边数第 k 个数字,如 digit(31859,3)=8,digit(2076,5)=0。

3. 用递归的方法求 Hermite 多项式的值

$$h_n(x)=\begin{cases}1, & n=0\\ 2x, & n=1\\ 2xh_{n-1}(x)-2(n-1)h_{n-2}(x), & n>1\end{cases}$$

对给定的 x 和正整数 n,求多项式的值。

4. 已知 $f(x,n)=\sqrt{n+\sqrt{(n-1)+\sqrt{(n-2)+\sqrt{\cdots+2+\sqrt{1+x}}}}}$,计算 x=4.2,n=10 以及 x=2.5,n=15 时的 f 的值。

5. 已知

$$f(x,n)=\cfrac{x}{n+\cfrac{x}{(n-1)+\cfrac{x}{(n-2)+\cfrac{\ddots}{\cfrac{x}{1+x}}}}}$$

用递归函数求解。

第七章　文件和结构体

文件是根据特定的目的而收集在一起的有关数据的集合。C++把每一个文件都看成是一个有序的字节流，每个文件都以文件结束标志结束，如果要操作某个文件，程序必须首先打开该文件。当一个文件被打开后，该文件就和一个流关联起来，这里的流实际上是一个字节序列。

C++将文件分为文本文件和二进制文件。二进制文件一般含有特殊的格式或计算机代码，如：图文件和可执行文件等。文本文件则是可以用任何文字处理程序阅读和编辑的简单 ASCII 文件。

下面我们学习如何编写 C++代码来实现对文本文件的输入和输出。

第一节　文件操作

C++语言提供了一批用于文件操作的标准函数，本节不是介绍文件打开函数 fopen，而是介绍另一个函数 freopen，它们都包含于标准库 cstdio 中，文件操作基本步骤如下：

(1)打开文件，将文件指针指向文件，决定打开文件类型；

(2)对文件进行读、写操作；

(3)在使用完文件后，关闭文件。

一、重定向版

【命令格式】

```
FILE * freopen( const char * filename, const char * mode, FILE * stream);
```

【参数说明】

filename：要打开的文件名

mode：文件打开的模式，和 fopen 中的模式(r/w)相同

stream：文件指针，通常使用标准流文件(stdin/stdout/stderr)

其中 stdin 是标准输入流，默认为键盘；stdout 是标准输出流，默认为屏幕；stderr 是标准错误流，一般把屏幕设为默认。通过调用 freopen，就可以修改标准流文件的默认值，实现重定向。

【使用方法】

因为文件指针使用的是标准流文件，因此我们可以不定义文件指针。接下来我们使用 freopen()函数以只读方式 r(read)打开输入文件 slyar.in。

格式：freopen("slyar.in", "r", stdin);

然后使用 freopen()函数以写入方式 w(write)打开输出文件 slyar.out。

格式:freopen("slyar.out", "w", stdout);

接下来的事情就是使用 freopen()函数的优点了,我们不再需要修改 scanf、printf、cin 和 cout,而是维持代码的原样就可以了。因为 freopen()函数重定向了标准流,使其指向前面指定的文件,省时省力。最后只要使用 fclose 关闭输入文件和输出文件即可。

格式:fclose(stdin);fclose(stdout);

若要恢复句柄,可以重新打开标准控制台设备文件,只是这个设备文件的名字是与操作系统相关的。

格式:freopen("CON", "r", stdin);

代码模版:

```
#include<cstdio>                          //使用 freopen 语句,须调用 cstdio 库
int main()
{
    freopen("slyar.in", "r", stdin);
    freopen("slyar.out", "w", stdout);
    /* 中间按原样写代码,什么都不用修改 */
    fclose(stdin);fclose(stdout);
    return 0;
}
```

例 7.1 从 in.txt 文件中读入数据,把它们的和保存 out.txt 文件中。

```
#include<cstdio>
int main()
{
    freopen("in.txt","r",stdin);       //定义输入文件名
    freopen("out.txt","w",stdout);     //定义输出文件名
    int temp,sum=0;
    while (scanf("%d",&temp)==1)//(cin>>temp)从输入文件中读入数据
                                       //在 C++中非 0 为真
      {
        sum=sum+temp;
      }
    printf("%d\n",sum);                // cout<<sum<<endl;
    fclose(stdin);fclose(stdout);      //关闭文件,可省略
    return 0;
}
```

in.txt 数据:

```
1 2 3 4 5
```

out.txt 结果:

```
15
```

【说明】 while (cin>>temp)和(scanf("%d",&temp)==1)主要是用于判断数据是否已经读完,以便及时终止循环。还可以用成员函数 eof 来判断是否达到数据流的末尾。对 scanf、printf 和 cin、cout 语句都适用。

二、fopen 版

重定向用起来很方便,但并不是所有算法竞赛都允许读写文件。甚至有的竞赛允许访问文件,但不允许使用 freopen 这样的重定向方式读写文件,可以使用 fopen 版,对 scanf 和 printf 语句适用。程序如下:

```
#include<cstdio>
using namespace std;
int main()
{
    FILE *fin,*fout;
    fin=fopen("in.txt","rb");                    //定义输入文件名
    fout=fopen("out.txt","wb");                  //定义输出文件名
    int temp,sum=0;
    while (fscanf(fin,"%d",&temp)==1)            //从输入文件中读入数据
      {
        sum=sum+temp;
      }
    fprintf(fout,"%d\n",sum);                    // cout<<sum<<endl;
    fclose(fin);fclose(fout);                    //关闭文件,可省略
    return 0;
}
```

先声明变量 fin 和 fout(暂且不用管 FILE * 为何物),把 scanf 改成 fscanf,第一个参数为 fin;把 printf 改成 fprintf,第一个参数为 fout,最后执行 fclose,关闭两个文件。

重定向和 fopen 两种方法各有优劣。重定向的方法写起来简单、自然,但是不能同时读写文件和标准输入输出;fopen 的写法稍显繁琐,但是灵活性比较大(例如可以反复打开并读写文件)。顺便说一句,如果把 fopen 版的程序改成读写标准输入输出,只需赋值 fin=stdin;fout=stdout;即可,不要调用 fopen 和 fclose。程序如下:

```
#include<cstdio>
using namespace std;
int main()
{
    FILE *fin,*fout;
    fin=stdin;
    fout=stdout;
    /*   本处语句同上   */
    fprintf(fout,"%d\n",sum);
```

```
    return 0;
}
```

三、文件输入输出流

在C++中,文件输入流(ifstream)和文件输出流(ofstream)的类,它们的默认输入输出设备都是磁盘文件。C++可以在创建对象时,设定输入或输出到哪个文件。由于这些类的定义是在fstream中进行的,因此,在使用这此类进行输入输出操作时,必须要在程序的首部利用#include指令包进fstream头文件。

例如:若想用fin作为输入对象,fout作为输出对象,则可以使用如下定义:

```
ifstream  fin("输入文件名.扩展名");
ofstream  fout("输出文件名.扩展名");
```

程序如下:

```
#include<fstream>                    //使用文件输入输出流,对cin、cout语句适用
using namespace std;
int main()
{
    ifstream fin("in.txt");          //定义输入文件名
    ofstream fout("out.txt");        //定义输出文件名
    int temp,sum=0;
    while (fin>>temp)  sum=sum+temp;      //从输入文件中读入数据
    fout<<sum<<endl;
    fin.close();fout.close();        //关闭文件,可省略
    return 0;
}
```

如果想再次使用cin和cout,是否要逐个把程序中的所有fin和fout替换为cin和cout?不用这么麻烦,只需要把fin和fout的声明语句去掉,并加上这样两行即可:

```
#define fin cin
#define fout cout
```

用条件编译,还可以让程序在本机上读写标准输入输出,比赛测试时读写文件(请读者自行实验)。

第二节　结构体

在实际问题中，一组数据往往具有不同的数据类型。例如，人口大普查时，我们需要记录每一位公民的姓名、年龄、性别、住址、身份证号码。这些信息分别要用整型、字符型、字符串型来记录。为了解决问题，C＋＋语言给出了另一种构造数据类型——“结构体”，它在数据存储方面相当于其他高级语言中的记录，但它有着面向对象的优势。

一、结构体(struct)定义和操作

1. 定义结构体及结构体变量

结构体变量的定义有两种形式：

(1)定义结构体类型的时候同时定义变量

```
struct 结构体类型名{                    //其中 struct 是关键字
    成员表；                            //可以有多个成员
    成员函数；                          //可以有多个成员函数，也可以没有
}结构体变量表；                         //可以同时定义多个结构体变量，用“,”隔开
```

例如：

```
struct student{                         //定义一个类型名叫 student 的 struct 类型
    string name;
    int chinese,math;
    int total;
} a[110];                               //同时定义了 a 数组变量
```

(2)先定义结构体再定义结构体变量

```
struct 结构体类型名{
    成员表；
    成员函数；
};
结构体名　结构体变量表                  //同样可以同时定义多个结构体变量
```

例如：

```
struct student{
    string name;
    int chinese,math;
    int total;
};
student a[110];
```

在定义结构体变量时注意，结构体变量名和结构体名不能相同。在定义结构体时，系统对其不分配实际内存。只有定义结构体变量时，系统才为其分配内存。

2. 结构体变量的特点

(1)结构体变量可以整体操作,例如:

```
swap(a[j],a[j+1]);
```

(2)结构体变量的成员访问也很方便、清晰,例如:

```
cin>>a[i].name;
```

(3)结构体变量的初始化和数组的初始化类似,例如:

```
student op={"gaoxiang",89,90,179};
```

3. 成员调用

结构体变量与各个成员之间引用的一般形式为:

结构体变量名.成员名

对于上面定义的结构体变量,我们可以这样操作:

```
cin>>a[i].name;                    //一般情况下不能写 cin>>a[i];
a[i].total=a[i].chinese+a[i].math;  //就像用整型变量一样
```

实际上结构体成员的操作与该成员类型所具有的操作是一致的。

成员运算符"."在存取成员数值时使用,其优先级最高,并具有左结合性。在处理包含结构体的结构体时,可记作:

```
strua.strub.membb
```

这说明结构体变量 strua 有结构体成员 strub;结构体变量 strub 有成员 membb。

4. 成员函数调用

结构体成员函数调用的一般形式为:

结构体变量名.成员函数

结构体成员函数默认将结构体变量作为引用参数。

例 7.2　成绩统计。输入 N 个学生的姓名和语文、数学的得分,按总分从高到低输出,分数相同的按输入先后输出。

输入格式:

第 1 行,有一个整数 N,N 的范围是[1…100];下面有 N 行,每行一个姓名,2 个整数。姓名由不超过 10 个的小写字母组成,整数范围是[0…100]。

输出格式:

总分排序后的名单,共 N 行,每行格式:姓名　语文　数学　总分。

输入样例:

```
4
gaoxiang  78  96
wangxi  70  99
liujia  90  87
zhangjin  78  91
```

输出样例:

```
liujia  90  87  177
gaoxiang  78  96  174
wangxi  70  99  169
zhangjin  78  91  169
```

分析：由于姓名是字符串，分数是整数，如果用数组保存，则要两个数组，比如：

```
string name[100];
int score[100][3];
```

这种方法不利于把一个学生的信息当成一个整体处理。

下面程序中通过使用结构(struct)类型的方法来解决这个问题。

程序如下：

```
#include<iostream>
#include<string>
using namespace std;
struct student{
    string name;
    int chinese,math;
    int total;
};                                  //定义一个 struct 的类型,类型名叫:student
student a[110];                     //定义一个数组 a,每个元素是 student 类型
int n;
int main()
{
    cin>>n;
        for (int i=0; i<n; i++)  //对结构体中成员的赋值、取值。
    {
        cin>>a[i].name;
        cin>>a[i].chinese >>a[i].math;
        a[i].total=a[i].chinese+a[i].math;
    }
    for (int i=n-1; i>0; i--)
    for (int j=0; j<i; j++)         //冒泡排序
        if (a[j].total<a[j+1].total)
            swap(a[j],a[j+1]);

    for (int i=0; i<n; i++)         //输出
        cout<<a[i].name<<' '<<a[i].chinese<<' '
        <<a[i].math<<' '<<a[i].total<<endl;
    return 0;
}
```

二、结构体(struct)的使用

例 7.3　离散化基础。以后要学习使用的离散化方法编程中，通常要知道每个数排序后的编号(rank 值)。

输入格式：

第1行，一个整数N，范围在[1…10000]；第2行，有N个不相同的整数，每个数都是int范围的。

输出格式：

依次输出每个数的排名。

输入样例：

```
5
8 2 6 9 4
```

输出样例：

```
4 1 3 5 2
```

分析：排序是必须的，关键是怎样把排名写回原来的数“下面”。程序使用了分别对数值和下标不同关键词2次排序的办法来解决这个问题，一个数据“节点”应该包含数值、排名、下标3个元素，用结构体比较好。

程序如下：

```
#include<iostream>
#include<algorithm>
using namespace std;
struct node{
    int data;                          //数值
    int rank;                          //排名
    int index;                         //下标
};                                     //定义 struct 类型
node a[10001];
int n;
bool comp1(node x,node y)
{
    return x.data<y.data;
}                                      //自定义比较函数
bool comp2(node x,node y)
{
    return x.index<y.index;
}                                      //同上
int main()
{
    cin>>n;
    for (int i=1; i<=n; i++)
    cin>>a[i].data,a[i].index=i;
    sort(a+1,a+1+n,comp1);                    //根据值排序，求排名 rank
    for (int i=1; i<=n; i++) a[i].rank=i;
    sort(a+1,a+1+n,comp2);                    //根据下标排序，回到原数据次序
```

```
    for (int i=1; i<=n; i++)
    cout<<a[i].rank<<' ';
    return 0;
}
```

例 7.4　模拟链表。在图论题编程中，通常要运用邻接链表数据结构。由于动态指针比静态的数组的存取慢，很多 OI 选手就用数组模拟指针。现在就来学习一下这种方法的编程。

有 N 个点，编号从 1 到 N。有 M 条边，每条边用连接的 2 个顶点表示，如：(3,8)，表示顶点 3 和 8 之间的边(无向边)。请输出每个顶点通过边相邻的顶点。

输入格式：

第 1 行，N 和 M 两个整数，N 范围在[1…5000]，M 范围在[1…100000]；下面有 M 行，每行两个整数，表示一条边。

输出格式：

N 行，第 i 行的第 1 个数 k 表示有多少边和 i 号顶点相连，后面有 k 个数，表示哪 k 个顶点和 i 连接为一条边。

输入样例：

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

输出样例：

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

分析：本题中邻接链表的每个节点有 2 个成员：

```
struct node
{
    int v;                          //定点编号
    int next;                       //链表的下一个节点的下标
};
```

一开始我们给足够大的数组，保证可以保存所有边。每读入一条边(a,b)，把 a 插入到 b 的链表前，再把 b 插入到 a 的链表前。

程序如下：

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

```
struct node{
    int v;
    int next;
};                                          //定义 struct 类型
node a[200001];                             //无向图,空间是边的两倍
int n,m,p;                                  //p 为 a 数组的空余空间下标
int k[5001],c[5001];                        //邻接链表的表头,k 数组记长度
void insert(int u,int v)                    //把 v 点插入到 u 点的邻接链表前
{
    a[++p].v=v;                             //申请一个新节点
    a[p].next=c[u];
    c[u]=p;                                 //插入到 u 链表头
    k[u]++;                                 //链表长度增加
}
int main()
{
    cin>>n>>m;
    for (int i=1; i<=m; i++)
    {
        int u,v;
        cin>>u>>v;
        insert(u,v);                        //插入 v 点到链表 u
        insert(v,u);                        //插入 u 点到链表 v
    }
    for (int i=1; i<=n; i++)
    {
        cout<<k[i]<<' ';                    //表的长度
        for (int j=c[i]; j>0; j=a[j].next)  //遍历链表
            cout<<a[j].v<<' ';
            cout<<endl;
    }
    return 0;
}
```

第八章　指针与链表

指针是C++语言中广泛使用的一种数据类型，运用指针编程是C++语言最主要风格之一。利用指针变量可以表示各种数据结构，能很方便地使用数组和字符串，并能像汇编语言一样处理内存地址，从而编出精炼而高效的程序，指针极大地丰富了C++语言的功能。学习指针是学习C++语言最重要的一环，能否正确理解和使用指针是我们是否掌握C++语言的一个标志。同时，指针也是C++语言中最为困难的一部分，在学习中除了要正确理解基本概念，还必须要多编程、多上机调试，只要做到这些，指针也是不难掌握的。

第一节　指针变量

一、指针变量的定义、赋值

在使用指针之前要先定义指针，对指针变量的类型说明，一般形式为：

类型说明符　*变量名；

其中，*表示这是一个指针变量，变量名即为定义的指针变量名，类型说明符表示该指针变量所指向的变量的数据类型。先通过例子看看指针与普通的变量有什么不同。

1. 普通变量定义

int　a=3；

定义了变量a，是int型的，值为3。内存中有一块内存空间是放a的值，对a的存取操作就是直接到这个内存空间存取。内存空间的位置叫地址，存放3的地址可以用取地址操作符“&”对a运算得到：&a。

2. 指针变量定义

int　*p=NULL；

定义了一个指针变量p，p指向一个内存空间，里面存放的是一个内存地址。现在赋值为NULL(其实就是0，表示特殊的空地址)。

3. 给指针变量p赋值

P=&a；

即把a变量的内存空间地址(比如：XXX)给了p。显然，直接对p存取，操作的是地址。通过这个地址间接地操作，才是整数3，如图8-1。P的间接操作要使用指针操作符“*”，即*p的值才是3。设有指向整型变量的指针变量p，如要把整型变量a的地址赋予p可以有以下两种方式：

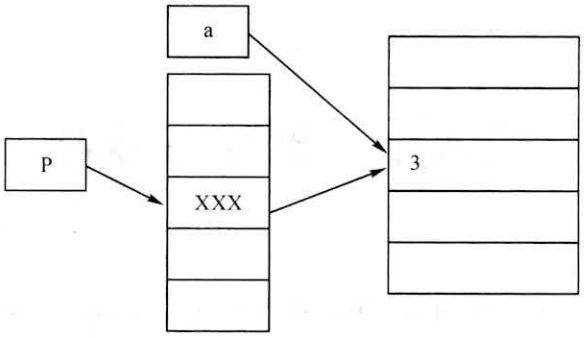


图 8-1

①指针变量初始化的方法

int a; int *p=&a;

②赋值语句的方法

int a; int *p; p=&a;

不允许把一个数赋予指针变量,故如下的赋值是错误的:int *p;p=1000;。被赋值的指针变量前不能再加"*"说明符,故如下的赋值也是错误的:*p=&a;。

指针的几个相关操作说明表

说　明	样 例
指针定义: 类型说明符　*指针变量名;	int a=10; int *p;
取地址运算符: &	p=&a;
间接运算符: *	*p=20;
指针变量直接存取的是内存地址	cout<<p; 结果可能是:0x4097ce
间接存取的才是储存类型的值	cout<<*p; 结果是:20

指针变量同普通变量一样,使用之前不仅要定义说明,而且必须被赋予具体的值,未经赋值的指针变量不能使用。如定义了 int a; int *p=&a;,则 *p 表示 p 指向的整型变量,而 p 中存放的是变量 a 占用单元的起始地址,所以 *p 实际上访问了变量 a,也就是说 *p 与 a 等价。下面举一个简单的指针使用的例子:

例 8.1　输入两个不同的数,通过指针对两个数进行相加和相乘,并输出。

```
#include<cstdio>
#include<iostream>
using namespace std;
int main()
{
    int a,b,s,t,*pa,*pb;
    pa=&a;pb=&b;
```

```
    a=10;b=20;
    s=*pa+*pb;
    t=*pa**pb;
    printf("a=%d,b=%d\n",*pa,*pb);
    printf("s=%d,t=%d\n",s,t);
    return 0;
}
```

输出：

a=10,b=20

s=30,t=200

二、指针的引用与运算

一般的，我们可以这样看指针(int *p)与普通变量(int a)的对应关系：

p	————	&a
*p	————	a
*p=3	————	a=3

下面介绍指针的一些运算。

1. 指针变量的初始化

指针的几个初始化操作说明表

	方 法	说 明
1	int *p=NULL;	NULL 是特殊的地址 0，叫零指针
2	int a; int *p=&a;	p 初始化为 a 的地址
3	int *p=new(int);	申请一个空间给 p，*p 内容不确定

要强调的是，对于定义的局部指针变量，其内容(地址)是随机的，直接对它操作可能会破坏程序或系统内存的值，引发不可预测的错误。所以编程中指针变量要保证先初始化或赋值，给予正确的地址再使用。

2. 指针变量的+、-运算

指针变量的内容是内存地址，它有两个常用的运算：加、减，这两个运算一般都是配合数组操作的。

例 8.2 输入 N 个整数，使用指针变量访问输出。

```
#include<cstdio>
using namespace std;
int a[101],n;
int main()
{
    scanf("%d",&n);
    for (int i=1; i<=n; i++)
```

```
    scanf("%d",&a[i]);
    int p=&a[1];  //定义指针变量 int p,初始化为数组开始元素的地址,即 a[1];
    for (int i=1; i<=n; i++)
    {
        printf("%d ", *p);
        p++;                    //p 指向下一个数,详见说明
    }
    return 0;
}
```

输入:4

2 1 6 0

输出:2 1 6 0

【说明】

"p++"的意思是"广义的加 1",不是 p 的值(地址)加 1,而是根据类型 int 增加 sizeof(int),即刚好"跳过"一个整数的空间,达到下一个整数。

类似的:

①"p--"就是向前"跳过"一个整数的空间,达到前一个整数。

②(p+3)就是指向后面第 3 个整数的地址。

3. 无类型指针

有时候,一个指针根据不同的情况,指向的内容是不同类型的值,我们可以先不明确定义它的类型,只是定义一个无类型的指针,以后根据需要再用强制类型转换的方法明确它的类型。

例 8.3 无类型指针运用举例。

```
#include<iostream>
using namespace std;
int a=10;
double b=3.5;
void *p;
int main()
{
    p=&a;                       //p 的地址赋值
    cout<<*(int*)p<<endl;       //必须明确 p 指向的空间的数据类型,详见说明
    p=&b;
    cout<<*(double*)p<<endl;
    return 0;
}
```

输出:10

3.5

【说明】

必须明确 p 指向的空间的数据类型,类型不一样的不仅空间大小不相同,储存的格式也

不同。如果把 cout<<*(double*)p<<endl;改成 cout<<*(long long*)p<<endl;输出的结果将是:4615063718147915776。

4. 多重指针

既然指针是指向其他类型的,指针本身也是一种类型。

C++允许递归地指针指向指针的指针——多重指针。

例 8.4　双重指针运用举例。

```
#include<cstdio>
using namespace std;
int a=10;
int *p;
int **pp;                          //定义双重指针
int main()
{
    p=&a;                          //将 p 指向 a
    pp=&p;                         //将 pp 指向 p
    printf("%d=%d=%d\n",a,*p,**pp);
                                   //**pp 通过 2 次间接访问了 a 的变量的值 10
    return 0;
}
```

输出:

```
10=10=10
```

【说明】

多重指针除了可以多次"间接"访问数据,OI 上主要的应用是动态的多维数组,这个强大的功能将在后面专门介绍。

第二节　指针与数组

一、指针与数组的关系

指向数组的指针变量称为数组指针变量。一个数组是一块连续的内存单元组成的，数组名就是这块连续内存单元的首地址。一个数组元素的首地址就是指它所占有的几个内存单元的首地址。一个指针变量既可以指向一个数组，也可以指向一个数组元素，可把数组名或第一个元素的地址赋予它。如要使指针变量指向第 i 号元素，可以把 i 元素的首地址赋予它，或把数组名加 i 赋予它。

设有数组 a，指向 a 的指针变量为 pa，则有以下关系：pa、a、&a[0]均指向同一单元，是数组 a 的首地址，也是 0 号元素 a[0]的首地址。pa+1、a+1、&a[1]均指向 1 号元素 a[1]。类推可知 pa+i、a+i、&a[i]指向 i 号元素 a[i]。pa 是变量，而 a，&a[i]是常量，在编程时应予以注意。

二、指向数组的指针

数组指针变量说明的一般形式为：

类型说明符　*指针变量名

其中类型说明符表示所指数组的类型，从一般形式可以看出，指向数组的指针变量和指向普通变量的指针变量的说明是相同的。

引入指针变量后，就可以用两种方法访问数组元素了，

例如定义了 int a[5];int *pa=a;

第一种方法为下标法，即用 pa[i]形式访问 a 的数组元素。

第二种方法为指针法，即采用 *(pa+i)形式，用间接访问的方法来访问数组元素。

例 8.5　scanf 使用数组名，用数组名或指针访问数组。

```
#include<cstdio>
using namespace std;
int main()
{
    int a[5],i,*pa=a;     //定义整型数组和指针，*pa=a 可以在下一行 pa=a;
    for (i=0;i<5;i++)
      scanf("%d",a+i);  //可写成 pa+i 和 &a[i]
    for (i=0;i<5;i++)
      printf("a[%d]=%d\n",i, *(a+i));
                          //指针访问数组，可写成 *(pa+i)或 pa[i]或 a[i]
    return 0;
}
```

输入:1 2 3 4 5
输出:a[0]=1
a[1]=2
a[2]=3
a[3]=4
a[4]=5

【说明】

①直接拿 a 当指针用,a 指向数组的开始元素,a+i 是指向数组的第 i 个元素的指针。

②指针变量 pa 是变量,可以变的,但数组 a 是静态的变量名,不可变,只能当做常量指针使用。例如:p=p+2;是合法的,a=a+2;是非法的。

③最早在使用标准输入 scanf 时就使用了指针技术,读入一个变量时要加取地址运算符"&"传递给 scanf 一个指针。对于数组,可以直接用数组名当指针。

三、指针也可以看成数组名

指针可以动态申请空间,如果一次申请多个变量空间,系统给的地址是连续的,就可以当成数组使用,这就是传说中的动态数组的一种。

例 8.6　动态数组,计算前缀和数组。b 是数组 a 的前缀和的数组定义:
b[i]=a[1]+a[2]+…+a[i],即 b[i]是 a 的 i 个元素的和。

```
#include<cstdio>
using namespace std;
int n;
int *a;                          //定义指针变量 a,后面直接当数组名使用
int main()
{
    scanf("%d",&n);
    a=new int[n+1];              //向操作系统申请了连续的 n+1 个 int 型的空间
    for (int i=1; i<=n; i++)
        scanf("%d",&a[i]);
    for (int i=2; i<=n; i++)
        a[i]+=a[i-1];
    for (int i=1; i<=n; i++)
        printf("%d ",a[i]);
    return 0;
}
```

输入:5
1 2 3 4 5
输出:1 3 6 10 15

【说明】

动态数组的优点:在 OI 中,对于大数据可能超空间的情况是比较纠结的事,用小数组只

得部分分,大数组可能爆空间(得0分)。使用“动态数组”,可以在确保小数据没问题的前提下,尽量满足大数据的需求。

例8.7 行列转换问题。

【问题描述】

矩阵可以认为是N*M的二维数组。现在有一个巨大但稀疏的矩阵。

N,M范围是:1≤N,M≤100000,有K个位置有数据,K的范围是:1≤K≤100000。

矩阵输入的方式是从上到下(第1行到第N行)、从左到右(从第1列到第M列)扫描,记录有数据的坐标位置(x,y)和值(v)。这是按照行优先的方式保存数据的。现在要求按照列优先的方式输出数据,即从左到右、从上到下扫描,输出有数据的坐标和数值。

【输入格式】

第1行,3个整数N,M,K,其中1≤N;M,K≤100000;下面有K行,每行3个整数:a,b,c,表示第a行第b列有数据c。数据在int范围内,保证是行优先的次序。

【输出格式】

1行,K个整数,是按照列优先次序输出的数。

【样列输入】

```
4 5 9
1 2 12
1 4 23
2 2 56
2 5 78
3 2 100
3 4 56
4 1 73
4 3 34
4 5 55
```

【样列输出】

```
73 12 56 100 34 23 56 78 55
```

【样例解释】

	12		23	
	56			78
	100		56	
73		34		55

【分析】

由于N*M可能会很大,直接开二维数组空间太大,不可行。解决问题的方法有很多种、下面程序使用了指针和动态数组,根据每一列的实际数据个数来申请该列的空间,使每列的“数组”长度不同。算法是O(M+N+K)的时间复杂度(即程序的运算量),O(N+K)的时间复杂度(即程序保存数据的内存大小),其他方法很难有这样优秀的效率。

【参考程序】

```
#include<cstdio>
using namespace std;
const int LP=100001;
int n,m,k;
int x[LP],y[LP],d[LP];
int c[LP];                          //每列的数据的个数
int *a[LP];                         //每列一个指针,准备申请"数组"。
                                    //这里定义了固定大小为 LP 的一个指针数组,
                                    //占用空间大小为 5*4*LP 字节,约 2M。
                                    //还可以在输入 n,m 后,再申请"动态数组",
                                    //当 n,m 比较小时,占用更小的空间。
                                    //a[i]表示第 i 列的指针。
int main()
{
    scanf("%d%d%d",&n,&m,&k);
    for (int i=1; i<=k; i++)
    {                               //x[i]和 y[i]是第 i 个数据所在行号和列号
        scanf("%d%d%d",&x[i],&y[i],&d[i]);
        c[y[i]]++;                  //统计 c 数组中每列的数据的个数
    }
    for (int i=1; i<=m; i++)
        a[i]=new int[c[i]];         //第 i 列指针申请"数组"空间
    for (int i=1; i<=k; i++)        //收集 k 个数据到相应的列中
    {
        *a[y[i]]=d[i];              //数据放在相应列的数组中
                                    //*a[y[i]]=d[i];也可以写成:a[y[i]][0]=d[i];
        a[y[i]]++;                  //数组指针移动到下一个位置
    }
    for (int i=1; i<=m; i++)        //列优先
    {
        a[i]-=c[i];                 //指针回到每列的前面
        for (int j=1; j<=c[i]; j++,a[i]++)
            printf("%d ",*a[i]);
    }
    return 0;
}
```

【说明】

特别的,可以把指针当数组名用,即 *a[y[i]]=d[i];可以写成:a[y[i]][0]=d[i];

第三节　指针与字符串

一、字符串的表示形式

在C++中,我们可以用两种方式访问字符串。

(1)用字符数组存放一个字符串,然后输出该字符串。

```
int main()
{
    char str[]="I love China!";
    printf("%s\n", str);
}
```

(2)用字符指针指向一个字符串。可以不定义字符数组,而定义一个字符指针。用字符指针指向字符串中的字符。

```
int main()
{
    char *str="I love China!";
    printf("%s\n", str);
}
```

在这里,我们没有定义字符数组,而是在程序中定义了一个字符指针变量str,用字符串常量"I love China!"对它进行初始化。C++对字符串常量是按字符数组处理的,在内存中开辟了一个字符数组用来存放该字符串常量。对字符指针变量初始化,实际上是把字符串第1个元素的地址(即存放字符串的字符数组的首元素地址)赋给str。有人认为str是一个字符串变量,以为在定义时把"I love China!"这几个字符赋给该字符串变量是不对的。

实际上,char *str="I love China!";

等价于:char *str;

str="I love China!";

可以看到,str被定义为一个指针变量,指向字符型数据,请注意它只是指向了一个字符变量或其他字符类型数据,不能同时指向多个字符数据,更不是把"I love China!"这些字符存放到str中(指针变量只能存放地址)。只是把"I love China!"的第一个字符的地址赋给指针变量str。

在输出时,要用:printf("%s\n", str);

其中"%s"是输出字符串时所用的格式符,在输出项中给出字符指针变量名,则系统先输出它所指向的一个字符数据,然后自动使str加1,使之指向下一个字符,然后再输出一个字符……如此直到遇到字符串结束标志"\0"为止。

注意:可以通过字符数组名或者字符指针变量输出一个字符串。而对一个数值型数组,是不能企图用数组名输出它的全部元素的。

例如:

```
int i[10];
```

……

printf("%d\n", i);

这样是不行的，只能逐个输出。显然 %s 可以对一个字符串进行整体的输入和输出。

二、字符串指针作函数参数

将一个字符串从一个函数传递到另外一个函数，可以用地址传递的方法，即用字符数组名作参数或用指向字符的指针变量做参数。在被调用的函数中可以改变字符串内容，在主调函数中可以得到改变了的字符串。

例 8.8　输入一个长度最大为 100 的字符串，以字符数组的方式储存，再将字符串倒序储存，输出倒序储存后的字符串。（这里以字符指针为函数参数）

```
#include<cstdio>
#include<cstring>
using namespace std;
void swapp(char &a,char &b)        //交换两个字符的位置
{
    char t;
    t=a;
    a=b;
    b=t;
}
void work(char * str)
{
    int len=strlen(str);           //strlen(str)这个函数返回的是 str 的长度，
                                   //需包含头文件 cstring
                                   //这个函数的原型是" size_t strlen(const char * str)"
    for (int i=0; i<=len/2; ++i)
      swapp(str[i],str[len-i-1]);
}
int main()
{
    char s[110];
    char * str = s;
    gets(s);
    work(str);
    printf("%s ",s);
    return 0;
}
```

输入：! anihC evol I

输出：I love China!

第四节　指针与函数

一、指针作为函数参数

指针可以作为函数的参数。在函数章节中,我们把数字作为参数传入函数中,实际上就是利用了传递指针(即传递数组的首地址)的方法。通过首地址,我们可以访问数组中的任何一个元素。

对于指向其他类型变量的指针,我们可以用同样的方式处理。

例如,我们编写如下一个函数,用于将两个整型变量的值交换。

```
void swap(int *x,int *y)
{
    int t=*x;
    *x=*y;
    *y=t;
}
```

这时,我们在其他函数中可以使用这个函数:

```
int a=5,b=3;
swap(&a,&b);
printf("a=%d,b=%d",a,b);
```

输出:a=3,b=5

在这个过程中,我们先将 a 和 b 的地址传给函数,然后在函数中通过地址得到变量 a 和 b 的值,并且对它们进行修改。当退出函数时,a 和 b 的值就已经交换了。

这里有一点值得我们注意。看如下这个过程:

```
void swap(int x,int y)
{
    int t=x;
    x=y;
    y=t;
}
```

我们调用了 swap(a,b);然而这个函数没有起作用,没有将 a 和 b 的值互换。

为什么呢?因为这里在传入变量 a 和 b 的时候,是将 a 的值赋值给函数中的形参 x,将 b 赋值给形参 y。这里接下来的操作就完全与 a 和 b 无关了,函数将变量 x 和 y 的值互换,然后退出函数。这里没有像上面例子那样传入指针,所以无法对传进来的变量进行修改。

将指针传入函数与将变量传入函数的区别在于:前者是通过指针来使用或修改传入的变量;而后者是将传入的变量的值赋给新的变量,函数对新的变量进行操作。

同理,对 scanf()函数而言,读取变量的时候我们要在变量之前加 & 运算符,即将指针传入函数。这是由于 scanf()函数通过指针将读取的值返回给引用的变量,没有 &,就无法进行正常的读取操作。

例 8.9　编写一个函数，将三个整型变量排序，并将三者中的最小值赋给第一个变量，次小值赋给第二个变量，最大值赋给第三个变量。

```
#include<cstdio>
using namespace std;
void swap(int *x,int *y)
{
    int t=*x;
    *x=*y;
    *y=t;
}
void sort(int *x,int *y,int *z)
{
    if (*x>*y) swap(x,y);
    if (*x>*z) swap(x,z);
    if (*y>*z) swap(y,z);
}
int main()
{
    int a,b,c;
    scanf("%d%d%d",&a,&b,&c);
    sort(&a,&b,&c);
    printf("%d %d %d",a,b,c);
    return 0;
}
```

输入：2 3 1

输出：1 2 3

二、函数返回指针

一个函数可以返回整数值、字符值、实型值等，也可以返回指针联系的数据（即地址）。

返回指针值的函数的一般定义形式为：

类型名 * 函数名(参数列表);

例如：

```
int *a(int a,int b)
```

a 是函数名，调用它后得到一个指向整型数据的指针（地址）。x 和 y 是函数 a 的形参，为整型。

注意：在 *a 的两侧没有括号；在 a 的两侧分别为 * 运算符和()运算符，由于()的优先级高于 *，因此 a 先于()结合。在函数前面有一个 *，表示此函数是返回指针类型的函数。最前面的 int 表示返回的指针指向整型变量。对初学 C++语言的人来说，这种定义形式可能不太习惯，容易弄错，用时要十分小心。

例 8.10 编写一个函数,用于在一个包含 N 个整数的数组中找到第一个质数,若有则返回函数的地址;否则返回 NULL(空指针)。

```
#include<cmath>
#include<cstdio>
using namespace std;
int n,a[10001];
bool isprime(int n)                          //判断 n 是不是素数
{
    if (n<2) return false;
    if (n==2) return true;
    for (int i=2; i<=sqrt(n); ++i)
      if (n%i==0)
        return false;
    return true;
}
int* find()
{
    for (int i=1; i<=n; ++i)
      if (isprime(a[i]))
        return &a[i];                        //这句也可以写成:return a+i;
    return NULL;                             //如果找不到则返回 NULL(空指针)
}
int main()
{
    scanf("%d",&n);
    for (int i=1; i<=n; ++i)
      scanf("%d",&a[i]);
    int *p=find();
    if (p!=NULL)
      printf("%d\n%d\n",p,*p);               //输出这个素数的地址和它本身
    else
      printf("can't find!");
    return 0;
}
```

输入:

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

输出:

```
(可能是)4214864
2
```

三、函数指针和函数指针数组

一个指针变量通过指向不同的整数变量的地址，就可以对其他的变量操作。

程序中不仅数据是存放在内存空间中，代码也同样存放在内存空间里。具体讲，C++的函数也保存在内存中，函数的入口地址也同样可以用指针访问。

另一方面，有些函数在编写时对要调用的辅助函数尚未确定，在执行时才能根据情况为其传递辅助函数的地址。比如 sort 函数的调用："sort(a,a+n,cmp);"其中的比较函数 cmp 是我们根据需要转给 sort 的(也可能是 cmp1,cmp2 等)，其实就是传递了函数指针。

下面我们来看一个具体例子。

例 8.11　使用函数指针调用函数示例。

```
#include<iostream>
using namespace std;
int t(int a)
{
    return a;
}
int main()
{
    cout<<t<<endl;                    //显示函数地址
    int (*p)(int a);                  //定义函数指针变量 p
    p=t;                              //将函数 t 的地址赋给函数指针 p
    cout<<p(5)<<','<<(*p)(10)<<endl;
    //输出 p(5)是 C++的写法，(*p)(10)是兼容 C，这是使用 p 来调用函数的两种方法
    return 0;
}
```

输出：

```
1
5,10
```

函数指针的基本操作有 3 个：

(1)声明函数指针。

声明要指定函数的返回类型以及函数的参数列表，和函数原型差不多。

例如，函数的原型是：int test(int);

相应的指针声明就是：int (*fp)(int);

要注意的是，不能写成：int *fp(int);

这样计算机编译程序会处理成声明一个 fp(int)的函数，返回类型是 int* 。

(2)获取函数的地址。

获取函数的地址很简单，只要使用函数名即可，例如，fp=test;

这表明函数名和数组名一样，可以看做是指针。

(3)使用函数指针来调用函数。

类似普通变量指针，可以用(*fp)来间接调用指向的函数。但 C++也允许像使用函数

名一样使用 fp,虽然有争议,但 C++确实是支持了。

函数指针还有另一种结合 typedef 的声明方式,如例 X 所示。

例 8.12 使用 typedef 定义函数指针示例。

```
#include<iostream>
using namespace std;
int sum(int a,int b)
{
    return a+b;
}
typedef int (*LP)(int,int); //定义了一个函数指针变量类型 LP
int main()
{
    LP p=sum;               //定义了一个 LP 类型的函数指针 LP,并赋值为 sum
    cout<<p(2,5);           //使用 p 来调用函数,实参为 2 和 5,调用 sum 函数,输
                              出返回值 7
    return 0;
}
```

输出:7

在软件开发编程中,函数指针的一个广泛应用是菜单功能函数的调用。通常选择菜单的某个选项都会调用相应的功能函数,并且有些软件的菜单会根据情况发生变化(上下文敏感)。如果使用 swith/case 或 if 语句处理起来会比较复杂、冗长,不利于代码的维护,可以考虑使用函数指针数组方便灵活地实现。

例 8.13 使用函数指针数组,模拟菜单功能实现方法示例。

```
#include<iostream>
using namespace std;
void t1() { cout<<" test1 "; }
void t2() { cout<<" test2 "; }
void t3() { cout<<" test3 "; }
void t4() { cout<<" test4 "; }
void t5() { cout<<" test5 "; }
typedef void(*LP)();          //定义了一个函数指针变量类型 LP
int main()
{
    LP a[]={t1,t2,t3,t4,t5}; //定义了一个 LP 类型的函数指针数组 a,并初始化
    int x;
    cin>>x;
    a[x]();                   //使用 a[x]()来调用选择的函数
    return 0;
}
```

输入:2

输出:test3

第五节　结构体指针

一、结构体指针的定义与使用

当一个指针变量用来指向一个结构体变量时，称之为结构体指针变量。

结构体指针变量的值是所指向的结构体变量的起始地址。通过结构体指针即可访问该结构体变量，这与数组指针和函数指针的情况是相同的。

结构体指针变量定义的一般形式：

结构体名 *结构体指针变量名

当然也可以在定义结构体的同时定义这个结构体指针变量。

例如：(定义一个结构体(类型为自己定义的 student)指针变量 p)

```
struct student
{
    char name[20];
    char sex;
    float score;
} *p;
```

也可写成

```
struct student
{
    char name[20];
    char sex;
    float score;
};
student *p;
```

与前面讨论的各类指针变量相同，结构体指针变量也必须要赋值后才能使用。赋值是把结构体变量的首地址赋予该指针变量，不能把结构名赋予该指针变量。

例如：如果 p 是被定义为 student 类型的结构体指针变量，boy 是被定义为 student 类型的结构体变量，则：p=&boy 是正确的，而 p=&student 是错误的。

引用结构体指针变量指向的结构体变量的成员的方法如下：

①指针名->成员名

②(*指针名).成员名

例如：

(*p).score 与 p->score 是等价的。

例 8.14　结构体指针运用举例。

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

```
struct student
{
    char name[20];
    char sex;
    int score;
} s[3]={{" xiaoming ",'f',356},
    {" xiaoliang ",'f',350},
    {" xiaohong ",'m',0}};
int main()
{
    student *p;
    printf(" Name        Sex   Score\n ");
    for (p=s; p<s+3; p++)
    printf("%-9s%3c%7d\n ",p->name,p->sex,p->score);
    return 0;
}
```

输出:

```
Name        Sex   Score
xiaoming    f     356
xiaoliang   f     350
xiaohong    m       0
```

【说明】

这里 p++起到移动指针的作用,随着 p 的变化,输出数组不同元素内容。

二、自引用结构

在一个结构体内部包含一个类型为该结构体本身的成员是否合法呢?

```
struct stu
{
    char name[20];
    int age,score;
    stu p;
};
```

这种类型的自引用是非法的,因为成员 p 是另一个完整的结构,其内部还将包含它自己的成员 p。这第 2 个成员又是一个完整的结构,它还将包含自己的成员 p……这样重复下去就永无止境了。这有点像永远不会终止的递归程序。但下面这个程序是合法的:

```
struct stu
{
    char name[20];
    int age,score;
```

```
    stu *p;
};
```

这个声明和前面那个声明的区别在于p现在是一个指针而不是结构体。编译器在结构体的长度确定之前就已经知道指针的长度，所以这种类型的自引用是合法的。

当一个结构体中有一个或是多个成员是指针，它们所指向的类型就是本结构体类型时，通常这种结构体称为“引用自身的结构体”，即“自引用结构”。这种自引用结构是实现其他一些结构的基础。

自引用结构在动态数据结构中有重要作用，甚至可以说，自引用结构是C/C++语言实现动态数据结构的基石。包括动态的链表、堆、栈、树，无不是自引用结构的具体实现。

例如，下面的定义就可以在实际操作中建立起一个链表。

```
struct node
{
    int x,y;
    node *next;
} point;
```

在下一节将对链表结构进行深入的研究。

第六节　链表结构

【存储方式的分类】顺序存储结构和链式存储结构。

【顺序存储结构】在(子)程序的说明部分就必须加以说明,以便分配固定大小的存储单元,直到(子)程序结束,才释放空间。因此,这种存储方式又称为静态存储。所定义的变量相应的称为静态变量。它的优缺点如下:

(1)优点:可以通过一个简单的公式随机存取表中的任一元素,逻辑关系上相邻的两个元素在物理位置上也是相邻的,且很容易找到前趋与后继元素;

(2)缺点:在线性表的长度不确定时,必须分配最大存储空间,使存储空间得不到充分利用,浪费了宝贵的存储资源;线性表的容量一经定义就难以扩充;在插入和删除线性表的元素时,需要移动大量的元素,浪费了时间。

【链式存储结构】在程序的执行过程中,通过两个命令向计算机随时申请存储空间或随时释放存储空间,以达到动态管理、使用计算机的存储空间,保证存储资源的充分利用。这样的存储方式称为动态存储。所定义的变量称为动态变量。它的优点如下:

可以用一组任意的存储单元(这些存储单元可以是连续的,也可以是不连续的)存储线性表的数据元素,这样就可以充分利用存储器的零碎空间。

【概念】为了表示任意存储单元之间的逻辑关系,对于每个数据元素来说,除了要存储它本身的信息(数据域、data)外,还要存储它的直接后继元素的存储位置(指针域、link 或 next)。我们把这两部分信息合在一起称为一个"结点 node"。

(1)N 个结点链接在一起就构成了一个链表。N=0 时,称为空链表。

(2)为了按照逻辑顺序对链表中的元素进行各种操作,我们需要定义一个变量用来存储整个链表的第一个结点的物理位置,这个变量称为"头指针、H 或 head"。也可以把头指针定义成一个结点,称为"头结点",头结点的数据域可以不存储任何信息,也可以存储线性表的长度等附加信息,头结点的指针域(头指针)存储指向第一个结点的指针,若线性表为空表,则头结点的指针域为空(NIL)。由于最后一个元素没有后继,所以线性表中最后一个结点的指针域为空(NIL)。

(3)由于此链表中的每个结点都只包含一个指针域,故称为"线性链表或单链表"。

一、单链表的定义

1. 类型和变量的说明

```
struct Node
{
    int data;
    Node * next;
};
Node * p;
```

2. 申请存储单元

p=new Node;　　//动态申请、空间大小由指针变量的基类型决定

3. 指针变量的赋值

指针变量名=NULL;　　//初始化，暂时不指向任何存储单元

如何表示和操作指针变量？不同于简单变量(如 A=0;)，C++规定用“指针变量名->”的形式引用指针变量(如 P->data=0;)。

二、单链表的结构、建立、输出

由于单链表的每个结点都有一个数据域和一个指针域，所以，每个结点都可以定义成一个记录。比如，有如下一个单链表，如何定义这种数据结构呢？

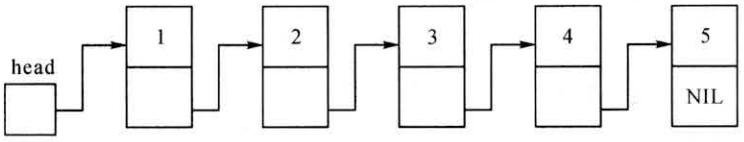


图 8-2

下面给出建立并输出单链表的程序，大家可以把它改成过程用在以后的程序当中。

```
#include<iostream>
using namespace std;
struct Node
{
    int data;
    Node *next;
};
Node *head, *p, *r;      //r 指向链表的当前最后一个结点，可以称为尾指针
int x;
int main()
{
  cin>>x;
  head=new Node;         //申请头结点
  r=head;
  while(x! =-1)          //读入的数非-1
  {
    p=new Node;          //否则，申请一个新结点
    p->data=x;
    p->next=NULL;
    r->next=p;           //把新结点链接到前面的链表中，实际上 r 是 p 的直接前趋
    r=p;                 //尾指针后移一个
    cin>>x;
  }
```

```
  p=head->next;          //头指针没有数据,只要从第一个结点开始就可以了}
  while(p->next! =NULL)
  {
    cout<<p->data<<" ";
    p=p->next;
  }
  cout<<p->data<<endl;
                         //最后一个结点的数据单独输出,也可以改用 do-while 循环
  system(" pause ");
}
```

三、单链表的操作

1. 查找"数据域满足一定条件的结点"

```
p=head->next;
while((p->data! =x)&&(p->next! =NULL))p=p->next;
                                    //找到第一个就结束
if(p->data==x)
    找到了处理;
else
    输出不存在;
```

如果想找到所有满足条件的结点,则修改如下:

```
p=head->next;
while(p->next! =NULL)          //一个一个判断
{
    if(p->data==x)             //找到一个处理一个;
    p=p->next;
}
```

2. 取出单链表的第 i 个结点的数据域

```
void get(Node * head,int i)
{
    Node * p;int j;
    p=head->next;
    j=1;
    while((p! =NULL)&&(j<i))
    {
        p=p->next;
        j=j+1;
    }
    if((p! =NULL)&&(j==i))
```

```
        cout<<p->data;
    else
        cout<<" i not exsit! ";
}
```

3. 插入一个结点在单链表中去

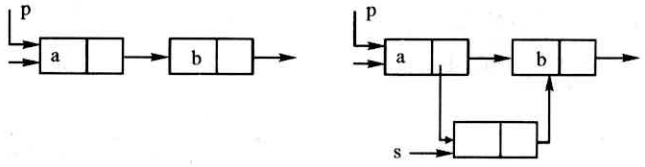


图8-3 插入结点前和后的链表变化

```
void insert(Node *head,int i,int x)        //插入X到第i个元素之前
{
    Node *p,*s;int j;
    p=head;
    j=0;
    while((p! =NULL)&&(j<i-1)) //寻找第i-1个结点,插在它的后面
    {
        p=p->next;
        j=j+1;
    }
    if(p==NULL)
      cout<<" no this position! ";
    else
    {                                      //插入
        s=new Node;
        s->data=x;
        s->next=p->next;
        p->next=s;
    }
}
```

4. 删除单链表中的第i个结点(如图8-4的"b"结点)

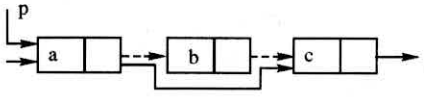


图8-4 删除一个结点前和后链表的变化

```
void delete(Node *head,int i)              //删除第i个元素
{
    Node *p,*s;int j;
    p=head;
    j=0;
```

```
    while((p->next! =NULL)&&(j<i-1))
    {
        p=p->next;
        j=j+1;
    }                                    //p 指向第 i-1 个结点
    if (p->next==NULL)
      cout<<" no this position! ";
    else
    {                                    //删除 p 的后继结点,假设为 s
        s=p->next;
        p->next=p->next->next; //或 p->next=s->next
        free(s);
    }
}
```

5. 求单链表的实际长度

```
int len(Node * head)
{
    int n=0;
    p=head
    while(p! =NULL)
    {
        n=n+1;
        p=p->next
    }
    return n;
}
```

四、双向链表

每个结点有两个指针域和若干数据域,其中一个指针域指向它的前趋结点,一个指向它的后继结点。它的优点是访问、插入、删除更方便,速度也快了。但"是以空间换时间"。

【数据结构的定义】

```
struct node
{
    int data;
    node * pre, * next;                  //pre 指向前趋,next 指向后继
}
node * head, * p, * q, * r;
```

下面给出双向链表的插入和删除过程。

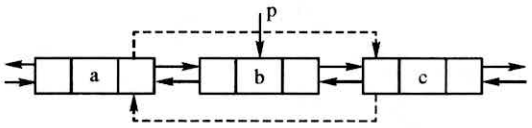


图8-5　删除P结点前后的指针变化

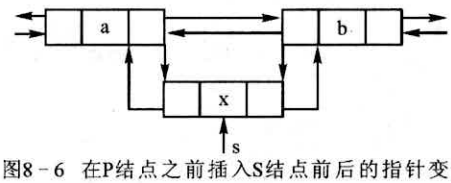


图8-6　在P结点之前插入S结点前后的指针变化

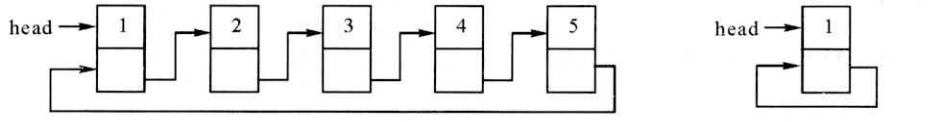
```
void insert(node * head,int i,int x)        //在双向链表的第 i 个结点之前插入 x
{
    node * s, * p;
    int j;
    s=new node;
    s->data=x;
    p=head;
    j=0;
    while((p->next! =NULL)&&(j<i))
    {
        p=p->next;
        j=j+1;
    }                                        //p 指向第 i 个结点
    if(p==NULL)
      cout<<" no this position! ";
    else
    {                                        //将结点 s 插入到结点 p 之前
        s->pre=p->pre;                       //将 s 的前趋指向 p 的前趋
        p->pre=s;                            //将 s 作为 p 的新前趋
        s->next=p;                           //将 s 的后继指向 p
        p->pre->next=s;                      //将 p 的本来前趋结点的后继指向 s
    }
}

void delete(node * head,int i)               //删除双向链表的第 i 个结点
{
    int j;
    node * p;
    p=head;
    j=0;
    while((p->next! =NULL)&&(j<i))
    {
        p=p->next;
        j=j+1;
```

```
    }                                    //p 指向第 i 个结点
    if(p==NULL)
    cout<<" no this position! ";
    else
    {                                    //将结点 P 删除
        p->pre->next=p->next;  //P 的前趋结点的后继赋值为 P 的后继
        p->next->pre=p->pre;   //P 的后继结点的前趋赋值为 P 的前趋
    }
}
```

五、循环链表

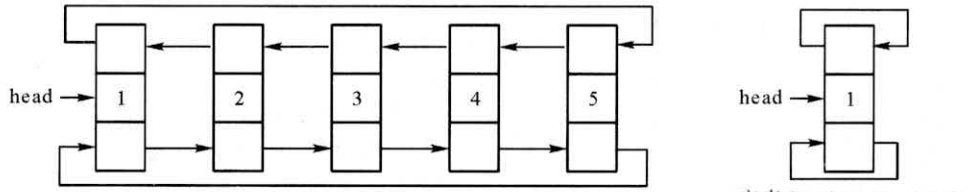
单向循环链表:最后一个结点的指针指向头结点。如下图:



非空表　　空表 (head^.next:=head)

图8-7 单向循环链表

双向循环链表:最后一个结点的指针指向头结点,且头结点的前趋指向最后一个结点。如下图:



非空表

空表(head^.next:=head)
(head^.pre:=head)

图8-8 双向循环链表

六、循环链表的应用举例

例 8.15　约瑟夫环问题

【问题描述】

有 M 个人,其编号分别为 1-M。这 M 个人按顺序排成一个圈(如图)。现在给定一个数 N,从第一个人开始依次报数,数到 N 的人出列,然后又从下一个人开始又从 1 开始依次报数,数到 N 的人又出列……如此循环,直到最后一个人出列为止。

【输入格式】

输入只有一行,包括 2 个整数 M,N。之间用一个空格分开(0 < n <= m <= 100)。

【输出格式】

输出只有一行,包括 M 个整数

【样列输入】

8 5

【样列输出】

5 2 8 7 1 4 6 3

【参考程序】

```
#include <iostream>
using namespace std;
struct node
{
    long d;
    node *next;
};
long n,m;
node *head,*p,*r;
int main()
{
    long i,j,k,l;
    cin>>n>>m;
    head=new node;
    head->d=1; head->next=NULL; r=head;
    for (i=2;i<=n;i++)
    {
        p=new node;
        p->d=i;
        p->next=NULL;
        r->next=p;
        r=p;
    }
    r->next=head; r=head;
    for (i=1;i<=n;i++)
    {
        for (j=1;j<=m-2;j++) r=r->next;
        cout<<r->next->d<<" ";
        r->next=r->next->next;
        r=r->next;
    }
}
```

【上机练习】

1. 利用指针,编写用于交换两个整型变量值的函数。

样例输入:5 6

样例输出:6 5

2. 利用指针,编写主程序,将输入字符串反序输出。

样例输入:ABCDEFGHIJK

样例输出:KJIHGFEDCBA

3. 编写一个用于在字符串中查找某字符的函数。

查找成功,函数返回该字符第一次出现的地址(指针);查找失败,返回 NULL。

编写主函数测试该函数。在主函数中输入原字符串和要查找的字符。如果找到,输出字符在原字符串中的序号;如果找不到,输出"no"。

输入格式:

输入包括两行,第一行为原字符串,第二行为要查找的字符。

输出格式:

输出包括一行,找到输出字符在原字符串中的序号(从 1 开始),找不到输出"no"。

样例输入 1:

ABCDEFGHIJKLMN

D

样例输出 1:

4

样例输入 2:

ABCDEFG

S

样例输出 2:

no

4. 约瑟夫问题(使用链表)【3.2 数据结构之指针和链表 1748】

约瑟夫问题:有 n 只猴子,按顺时针方向围成一圈选大王(编号从 1 到 n),从第 1 号开始报数,一直数到 m,数到 m 的猴子退出圈外,剩下的猴子再接着从 1 开始报数。就这样,直到圈内只剩下一只猴子时,这个猴子就是猴王,编程求输入 n,m 后,输出最后猴王的编号。

输入格式:

每行是用空格分开的两个整数,第一个是 n,第二个是 m(0 < m,n <=300)。最后一行是:0 0

输出格式:

对于每行输入数据(最后一行除外),输出数据也是一行,即最后猴王的编号。

样例输入:

6 2

12 4

8 3

0 0

样例输出：

5

1

7

5. 删除数组中的元素(链表)【3.2 数据结构之指针和链表 6378】

给定 N 个整数，将这些整数中与 M 相等的删除。

假定给出的整数序列为：1,3,3,0,－3,5,6,8,3,10,22,－1,3,5,11,20,100,3,9,3。

应该将其放在一个链表中，链表长度为 20。

要删除的数是 3，删除以后，链表中只剩 14 个元素：1 0 －3 5 6 8 10 22 －1 5 11 20 100 9。

要求：必须使用链表，不允许使用数组，也不允许不删除元素直接输出。

程序中必须有链表的相关操作：建立链表，删除元素，输出删除后链表中元素，释放链表。

输入格式：

输入包含 3 行：

第一行是一个整数 n(1 <= n <= 200000)，代表数组中元素的个数。

第二行包含 n 个整数，代表数组中的 n 个元素。每个整数之间用空格分隔；每个整数的取值在 32 位有符号整数范围以内。

第三行是一个整数 k，代表待删除元素的值(k 的取值也在 32 位有符号整数范围内)。

输出格式：

输出只有 1 行：

将数组内所有待删除元素删除以后，输出数组内的剩余元素的值，每个整数之间用空格分隔。

样例输入：

20

1 3 3 0 －3 5 6 8 3 10 22 －1 3 5 11 20 100 3 9 3

3

样例输出：

1 0 －3 5 6 8 10 22 －1 5 11 20 100 9

6. 统计学生信息(使用动态链表完成)【3.2 数据结构之指针和链表 6379】

利用链表记录输入的学生信息(学号、姓名、性别、年龄、得分、地址)。其中，学号长度不超过 20，姓名长度不超过 40，性别长度为 1，地址长度不超过 40。

输入格式：

包括若干行，每一行都是一个学生的信息，如：

00630018 zhouyan m 20 10.0 28#460

输入的最后以" end "结束。

输出格式：

将输入的内容倒序输出。每行一条记录，按照下面的格式输出：

学号 姓名 性别 年龄 得分 地址

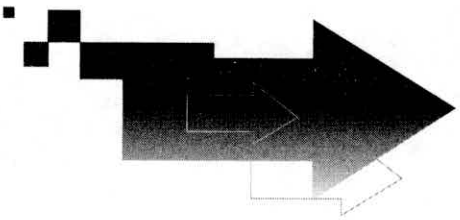
样例输入：

00630018 zhouyan m 20 10 28#4600
0063001 zhouyn f 21 100 28#460000
0063008 zhoyan f 20 1000 28#460000
0063018 zhouan m 21 10000 28#4600000
00613018 zhuyan m 20 100 28#4600
00160018 zouyan f 21 100 28#4600
01030018 houyan m 20 10 28#4600
0630018 zuyan m 21 100 28#4600
10630018 zouan m 20 10 28#46000
end

样例输出：

10630018 zouan m 20 10 28#46000
0630018 zuyan m 21 100 28#4600
01030018 houyan m 20 10 28#4600
00160018 zouyan f 21 100 28#4600
00613018 zhuyan m 20 100 28#4600
0063018 zhouan m 21 10000 28#4600000
0063008 zhoyan f 20 1000 28#460000
0063001 zhouyn f 21 100 28#460000
00630018 zhouyan m 20 10 28#4600

第二部分
基础算法



第一章　高精度计算

利用计算机进行数值计算，有时会遇到这样的问题：有些计算要求精度高，希望计算的数的位数可达几十位甚至几百位，虽然计算机的计算精度也算较高了，但因受到硬件的限制，往往达不到实际问题所要求的精度。我们可以利用程序设计的方法去实现这样的高精度计算。介绍常用的几种高精度计算的方法。

高精度计算中需要处理好以下几个问题：

(1)数据的接收方法和存贮方法

数据的接收和存贮：当输入的数很长时，可采用字符串方式输入，这样可输入位数很长的数，利用字符串函数和操作运算，将每一位数取出，存入数组中。

```
void init(int a[])                    //传入一个数组
{
  string s;
  cin>>s;                             //读入字符串 s
  a[0]=s.length();                    //用 a[0]计算字符串 s 的位数
  for(i=1;i<=a[0];i++)
    a[i]=s[a[0]-i]-'0';               //将数串 s 转换为数组 a,并倒序存储
}
```

另一种方法是直接用循环加数组方法输入数据。

(2)高精度数位数的确定

位数的确定：接收时往往是用字符串的，所以它的位数就等于字符串的长度。

(3)进位，借位处理

```
加法进位：c[i]=a[i]+b[i];
          if(c[i]>=10) { c[i]%=10;++c[i+1]; }
减法借位：if (a[i]<b[i]) {--a[i+1]; a[i]+=10; }
          c[i]=a[i]-b[i];
乘法进位：c[i+j-1]=a[i] * b[j]+x+c[i+j-1];
          x=c[i+j-1]/10;
          c[i+j-1] %=10;
```

(4)商和余数的求法

商和余数处理：视被除数和除数的位数情况进行处理。

例 1.1　高精度加法。输入两个正整数，求它们的和。

【算法分析】

输入两个数到两个变量中，然后用赋值语句求它们的和，输出。但是，我们知道，在C++语言中任何数据类型都有一定的表示范围。当两个被加数很大时，上述算法显然不能

求出精确解,因此我们需要寻求另外一种方法。在读小学时,我们做加法都采用竖式方法,如图 1-1。这样,我们方便写出两个整数相加的算法。

$$\begin{array}{r} 856 \\ +255 \\ \hline 1111 \end{array} \qquad \begin{array}{r} a_3\ a_2\ a_1 \\ +\ b_3\ b_2\ b_1 \\ \hline c_4\ c_3\ c_2\ c_1 \end{array}$$

图 1-1　　图 1-2

如果我们用数组 a、b 分别存储加数和被加数,用数组 c 存储结果。则上例有 a[1]=6,a[2]=5,a[3]=8,b[1]=5,b[2]=5,b[3]=2,c[4]=1,c[3]=1,c[2]=1,c[1]=1,两数相加如图 1-2 所示。

因此,算法描述如下:

```
int c[100];
void add(int a[],int b[])              //a,b,c 都为数组,分别存储被加数、加数、结果
{
  int  i=1,x=0;                        //x 是进位
  while ((i<=a 数组长度)||(i<=b 数组的长度))
  {
    c[i]=a[i]+b[i]+x;                  //第 i 位相加并加上次的进位
    x=c[i]/10;                         //向高位进位
    c[i]%=10;                          //存储第 i 位的值
    i++;                               //位置下标变量
  }
}
```

通常,读入的两个整数可用字符串来存储,程序设计如下:

```
#include<iostream>
#include<cstdio>
#include<cstring>
using namespace std;
int main()
{
  char a1[100],b1[100];
  int a[100],b[100],c[100],lena,lenb,lenc,i,x;
  memset(a,0,sizeof(a));
  memset(b,0,sizeof(b));
  memset(c,0,sizeof(c));
  gets(a1);
  gets(b1);                                          //输入加数与被加数
  lena=strlen(a1);
  lenb=strlen(b1);
  for (i=0;i<=lena-1;i++) a[lena-i]=a1[i]-48;  //加数放入 a 数组
```

```
    for (i=0;i<=lenb-1;i++) b[lenb-i]=b1[i]-48; //加数放入 b 数组
    lenc=1;
    x=0;
    while (lenc <=lena||lenc <=lenb)
    {
      c[lenc]=a[lenc]+b[lenc]+x;              //两数相加
      x=c[lenc]/10;
      c[lenc]%=10;
      lenc++;
    }
    c[lenc]=x;
    if (c[lenc]==0)
      lenc--;                                 //处理最高进位
    for (i=lenc;i>=1;i--)
      cout<<c[i];                             //输出结果
    cout<<endl;
    return 0;
}
```

例 1.2　高精度减法。输入两个正整数，求它们的差。

【算法分析】

类似加法，可以用竖式求减法。在做减法运算时，需要注意的是：被减数必须比减数大，同时需要处理借位。高精度减法的参考程序：

```
#include<iostream>
#include<cstdio>
#include<cstring>
using namespace std;
int main()
{
  int a[256],b[256],c[256],lena,lenb,lenc,i;
  char n[256],n1[256],n2[256];
  memset(a,0,sizeof(a));
  memset(b,0,sizeof(b));
  memset(c,0,sizeof(c));
  printf("Input minuend:");  gets(n1);  //输入被减数
  printf("Input subtrahend:"); gets(n2); //输入减数
if (strlen(n1)<strlen(n2)||(strlen(n1)==strlen(n2)&&strcmp(n1,n2)<0))
                          //strcmp()为字符串比较函数，当 n1==n2，返回 0；
                          //n1>n2 时，返回正整数；n1<n2 时，返回负整数
{                         //处理被减数和减数，交换被减数和减数
  strcpy(n,n1);           //将 n1 数组的值完全赋值给 n 数组
```

```
    strcpy(n1,n2);
    strcpy(n2,n);
    cout<<"-";                    //交换了减数和被减数,结果为负数
  }
  lena=strlen(n1); lenb=strlen(n2);
  for (i=0;i<=lena-1;i++) a[lena-i]=int(n1[i]-'0');  //被减数放入 a 数组
  for (i=0;i<=lenb-1;i++) b[lenb-i]=int(n2[i]-'0');  //减数放入 b 数组
  i=1;
  while (i<=lena||i<=lenb)
  {
    if (a[i]<b[i])
    {
      a[i]+=10;                   //不够减,那么向高位借 1 当 10
      a[i+1]--;
    }
    c[i]=a[i]-b[i];               //对应位相减
    i++;
    }
    lenc=i;
    while ((c[lenc]==0)&&(lenc>1)) lenc--;  //最高位的 0 不输出
    for (i=lenc;i>=1;i--) cout<<c[i];       //输出结果
    cout<<endl;
    return 0;
  }
```

例 1.3 高精度乘法。输入两个正整数,求它们的积。

【算法分析】

类似加法,可以用竖式求乘法。在做乘法运算时,同样也有进位,同时对每一位进行乘法运算时,必须进行错位相加,如图 1-3、图 1-4。

分析 c 数组下标的变化规律,可以写出如下关系式:$c_i=c'_i+c''_i+\cdots$由此可见,c_i跟 a[i] * b[j]乘积有关,跟上次的进位有关,还跟原 c_i的值有关,分析下标规律,有 c[i+j-1]=a[i] * b[j]+x+c[i+j-1]; x=c[i+j-1]/10 ; c[i+j-1]%=10;

```
    856
  × 25
 ------
   4280
  1712
 ------
  21400
```

图 1-3

```
          a3  a2  a1
        ×       b2  b1
 ---------------------
          c'4 c'3 c'2 c'1
      c''5 c''4 c''3 c''2
 ---------------------
      c5  c4  c3  c2  c1
```

图 1-4

高精度乘法的参考程序:

```
#include<iostream>
#include<cstring>
```

```
#include<cstdio>
using namespace std;
int main()
{
  char a1[100],b1[100];
  int a[100],b[100],c[100],lena,lenb,lenc,i,j,x;
  memset(a,0,sizeof(a));
  memset(b,0,sizeof(b));
  memset(c,0,sizeof(c));
  gets(a1);gets(b1);
  lena=strlen(a1);lenb=strlen(b1);
  for (i=0;i<=lena-1;i++) a[lena-i]=a1[i]-48;
  for (i=0;i<=lenb-1;i++) b[lenb-i]=b1[i]-48;
  for (i=1;i<=lena;i++)
  {
    x=0;                                    //用于存放进位
    for (j=1;j<=lenb;j++)                   //对乘数的每一位进行处理
    {
      c[i+j-1]=a[i]*b[j]+x+c[i+j-1];        //当前乘积+上次乘积进位+原数
      x=c[i+j-1]/10;
      c[i+j-1] %=10;
    }
    c[i+lenb]=x;                            //进位
  }
  lenc=lena+lenb;
  while (c[lenc]==0&&lenc>1)                //删除前导 0
    lenc--;
  for (i=lenc;i>=1;i--)
    cout<<c[i];
  cout<<endl;
}
```

例 1.4　高精除以低精。输入两个正整数，求它们的商(做整除)。

【算法分析】

做除法时，每一次的商的值都在 0～9，每次求得的余数连接以后的若干位得到新的被除数，继续做除法。因此，在做高精度除法时，要涉及到乘法运算和减法运算，还有移位处理。当然，为了程序简洁，可以避免高精度乘法，用 0～9 次循环减法取代得到商的值。这里，我们讨论一下高精度数除以单精度数的结果，采取的方法是按位相除法。

```
#include<iostream>
#include<cstring>
#include<cstdio>
```

```
using namespace std;
int main()
{
  char a1[100],c1[100];
  int a[100],c[100],lena,i,x=0,lenc,b;
  memset(a,0,sizeof(a));
  memset(c,0,sizeof(c));
  gets(a1);
  cin>>b;
  lena=strlen(a1);
  for (i=0;i<=lena-1;i++)
    a[i+1]=a1[i]-48;
  for (i=1;i<=lena;i++)                    //按位相除
  {
    c[i]=(x*10+a[i])/b;
    x=(x*10+a[i])%b;
  }
  lenc=1;
  while (c[lenc]==0&&lenc<lena)
    lenc++;                                //删除前导0
  for (i=lenc;i<=lena;i++)
    cout<<c[i];
  cout<<endl;
  return 0;
}
```

实质上,在做两个高精度数运算的时候,存储高精度数的数组元素可以不仅仅只保留一位数字,而可以采取保留多位数(例如一个整型或长整型数据等),这样,在做运算(特别是乘法运算)时,可以减少很多操作次数。如图1-5所示,就是采用4位保存的除法运算,其他运算也类似。具体程序可以修改上述例题予以解决,程序请读者完成。

```
示例:123456789 ÷45 = 1'2345'6789 ÷ 45
                   = 274'3484
∵ 1/45 = 0, 1%45 = 1
∴ 取 12345/45 = 274   ∵ 12345%45 = 15
∴ 取 156789/45 = 3484
∴ 答案为 2743484, 余数为 156789%45 = 9
```

图1-5

例1.5 高精除以高精,求它们的商和余数。

【算法分析】

高精除以低精是对被除数的每一位(这里的"一位"包含前面的余数,以下都是如此)都除以除数,而高精除以高精则是用减法模拟除法,对被除数的每一位都减去除数,一直减到

当前位置的数字(包含前面的余数)小于除数(由于每一位的数字小于10,所以对于每一位最多进行10次计算),具体实现程序如下:

```
#include<iostream>
#include<cstring>
using namespace std;
int a[101],b[101],c[101],d,i;
void init(int a[])
{ string s;
  cin>>s;                          //读入字符串 s
  a[0]=s.length();                 //用 a[0]计算字符串 s 的位数
  for(i=1;i<=a[0];i++)
    a[i]=s[a[0]-i]-'0';            //将数串 s 转换为数组 a,并倒序存储
}
void print(int a[])                //打印输出
{
  int i;
  if (a[0]==0){cout<<0<<endl;return;}
  for(i=a[0];i>0;i--) cout<<a[i];
  cout<<endl;
  return ;
}
int compare (int a[],int b[])
      //比较 a 和 b 的大小关系,若 a>b 则为 1,若 a<b 则为-1,若 a=b 则为 0
{ int i;
  if(a[0]>b[0]) return 1;          //a 的位数大于 b 则 a 比 b 大
  if(a[0]<b[0]) return-1;          //a 的位数小于 b 则 a 比 b 小
  for(i=a[0];i>0;i--)              //从高位到低位比较
   {if (a[i]>b[i]) return 1;
    if (a[i]<b[i]) return-1;
   }
  return 0;                        //各位都相等则两数相等。
}
void jian(int a[],int b[])         //计算 a=a-b
{ int flag,i;
  flag=compare(a,b);               //调用比较函数判断大小
  if (flag==0) {a[0]=0;return;}    //相等
  if(flag==1)                      //大于
    {for(i=1;i<=a[0];i++)
      {if(a[i]<b[i]){ a[i+1]--;a[i]+=10;}  //若不够减则向上借一位
      a[i]-=b[i];
```

```
        }
        while(a[0]>0&&a[a[0]]==0) a[0]--;  //修正 a 的位数
        return;
      }
}
void numcpy(int p[],int q[],int det)  //复制 p 数组到 q 数组从 det 开始的地方
{
  for (int i=1;i<=p[0];i++) q[i+det-1]=p[i];
  q[0]=p[0]+det-1;
}
void chugao(int a[],int b[],int c[])
{
  int i,tmp[101];
  c[0]=a[0]-b[0]+1;
  for (i=c[0];i>0;i--)
   {
     memset(tmp,0,sizeof(tmp));                    //数组清零
     numcpy(b,tmp,i);
     while(compare(a,tmp)>=0){c[i]++;jian(a,tmp);}  //用减法来模拟
   }
  while(c[0]>0&&c[c[0]]==0)c[0]--;
  return ;
}
int main()
{
  memset(a,0,sizeof(a));
  memset(b,0,sizeof(b));
  memset(c,0,sizeof(c));
  init(a);init(b);
  chugao(a,b,c);
  print(c);
  print(a);
  return 0;
}
```

例 1.6 回文数(Noip1999)

【问题描述】

若一个数(首位不为零)从左向右读与从右向左读都是一样,我们就将其称之为回文数。例如:给定一个 10 进制数 56,将 56 加 65(即把 56 从右向左读),得到 121 是一个回文数。又如,对于 10 进制数 87,

STEPl: 87+78=165　　STEP2: 165+561=726

STEP3：726+627=1353　STEP4：1353+3531=4884

在这里的一步是指进行了一次 n 进制的加法，上例最少用了 4 步得到回文数 4884。

写一个程序，给定一个 n(2<n<=10 或 n=16)进制数 m。求最少经过几步可以得到回文数。如果在 30 步以内(包含 30 步)不可能得到回文数，则输出“Impossible”。

【输入样例】　　【输出样例】

9 87　　6

【算法分析】

n 进制运算

1. 当前位规范由%10 改为% n

2. 进位处理由/10 改为/n

3. 其他运算规则不变

【参考程序】

```
#include<iostream>
#include<cstring>
using namespace std;
int n,a[101],b[101],ans,i;
void init(int a[])                    //将数串 s 转化为整数数组 a
{ string s;
  cin>>n>>s;                          //读入字符串 s
  memset(a,0,sizeof(a));              //数组 a 清 0
  a[0]=s.length();                    //用 a[0]计算字符串 s 的位数
  for(i=1;i<=a[0];i++)
    if(s[a[0]-i]>='0'&&s[a[0]-i]<='9')  a[i]=s[a[0]-i]-'0';
    else a[i]=s[a[0]-i]-'A'+10;
}
bool check(int a[])                   //判别整数数组 a 是否为回文数
{ for(i=1;i<=a[0];i++)
    if(a[i]!=a[a[0]-i+1])return false;
  return true;
}
void  jia(int a[])                    //整数数组 a 与其反序数 b 进行 n 进制加法运算
{ int i,k;
  for(i=1;i<=a[0];i++)b[i]=a[a[0]-i+1];    //反序数 b
  for(i=1;i<=a[0];i++) a[i]+=b[i];         //逐位相加
  for(i=1;i<=a[0];i++)                     //处理进位
    {a[i+1]+=a[i]/n;
     a[i]%=n;
    }
  if(a[a[0]+1]>0) a[0]++;     //修正新的 a 的位数(a+b 最多只能进一位)
}
```

```
int main()
{
  init(a);
  if(check(a)){cout<<0<<endl;return 0;}
  ans=0;                          //步数初始化为 0
  while(ans<=30)
    { ans++;
      jia(a);
      if(check(a)){cout<<ans<<endl;return 0;}
    }
  cout<<" Impossible ";           //输出无解信息
  return 0;
}
```

【上机练习】

1. 大整数加法【1.6 编程基础之一维数组 10】

求两个不超过 200 位的非负整数的和。

输入：

有两行，每行是一个不超过 200 位的非负整数，可能有多余的前导 0。

输出：

一行，即相加后的结果。结果里不能有多余的前导 0，即如果结果是 342，那么就不能输出为 0342。

样例输入：

```
2222222222222222222
3333333333333333333
```

样例输出：

```
5555555555555555555
```

2. 大整数减法【1.6 编程基础之综合应用 11】

求两个大的正整数相减的差。

输入：

共 2 行，第 1 行是被减数 a，第 2 行是减数 b(a > b)。每个大整数不超过 200 位，不会有多余的前导零。

输出：

一行，即所求的差。

样例输入：

```
99999999999999999999999999999999999999
9999999999999
```

样例输出：

```
99999999999999999999999990000000000000
```

3. 计算 2 的 N 次方【1.6 编程基础之一维数组 12】

任意给定一个正整数 N(N<=100),计算 2 的 n 次方的值。

输入:

输入一个正整数 N。

输出:

输出 2 的 N 次方的值。

样例输入:

5

样例输出:

32

4. 大整数的因子【1.6 编程基础之一维数组 13】

已知正整数 k 满足 2<=k<=9,现给出长度最大为 30 位的十进制非负整数 c,求所有能整除 c 的 k。

输入:

一个非负整数 c,c 的位数<=30。

输出:

若存在满足 c%k==0 的 k,从小到大输出所有这样的 k,相邻两个数之间用单个空格隔开;若没有这样的 k,则输出"none"。

样例输入:

30

样例输出:

2 3 5 6

5. 求 10000 以内 n 的阶乘【1.6 编程基础之一维数组 14】

求 10000 以内 n 的阶乘。

输入:

只有一行输入,整数 n(0<=n<=10000)。

输出:

一行,即 n! 的值。

样例输入:

100

样例输出:

933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000

6. 阶乘和【1.6 编程基础之一维数组 15】

用高精度计算出 S=1! +2! +3! +…+n! (n≤50),其中"!"表示阶乘,例如:5! = 5*4*3*2*1。

输入正整数 N,输出计算结果 S。

输入:

一个正整数 N。

输出:

计算结果 S。

样例输入：

5

样例输出：

153

7. 大整数乘法【1.13 编程基础之综合应用 09】

求两个不超过 200 位的非负整数的积。

输入：

有两行，每行是一个不超过 200 位的非负整数，没有多余的前导 0。

输出：

一行，即相乘后的结果。结果里不能有多余的前导 0，即如果结果是 342，那么就不能输出为 0342。

样例输入：

12345678900

98765432100

样例输出：

1219326311126352690000

8. 除以 13【1.13 编程基础之综合应用 27】

输入一个大于 0 的大整数 N，长度不超过 100 位，要求输出其除以 13 得到的商和余数。

输入：

一个大于 0 的大整数，长度不超过 100 位。

输出：

两行，分别为整数除法得到的商和余数。

样例输入：

2132104848488485

样例输出：

164008065268345

0

第二章　数据排序

信息获取后通常需要进行处理，处理后的信息其目的是便于人们的应用。信息处理方法有多种，通常有数据的排序、查找、插入、删除、归并等操作。读者已经接触了一些这方面的知识，本章重点介绍数据排序的几种方法。

1. 选择排序

(1) 基本思想：每一趟从待排序的数据元素中选出最小(或最大)的一个元素，顺序放在待排序的数列的最前，直到全部待排序的数据元素排完。

(2)排序过程：

初始关键字　[49 38 65 97 76 13 27 49]

第一趟排序后 13 [38 65 97 76 49 27 49]

第二趟排序后 13 27 [65 97 76 49 38 49]

第三趟排序后 13 27 38 [97 76 49 65 49]

第四趟排序后 13 27 38 49 [76 97 65 49]

第五趟排序后 13 27 38 49 49 [97 65 76]

第六趟排序后 13 27 38 49 49 65 [97 76]

第七趟排序后 13 27 38 49 49 65 76 [97]

最后排序结果 13 27 38 49 49 65 76　97

例 2.1　输入 n 个数，将 n 个数按从小到大的顺序输出(n<=10000)。

输入样例：

8

49 38 65 97 76 13 27 49

输出样例：

13 27 38 49 49 65 76 97

归纳上述排序过程，具体实现步骤如下：

①读入数据存放在 a 数组中。

②在 a[1]～a[n]中选择值最小的元素，与第 1 位置元素交换，则把最小值元素放入 a[1]中。

③在 a[2]～a[n]中选择值最小的元素，与第 2 位置元素交换，则把最小值元素放入 a[2]中，……

④直到第 n－1 个元素与第 n 个元素比较排序为止。

程序实现方法：用两层循环完成算法，外层循环 i 控制当前序列最小值存放的数组位置，内层循环 j 控制从 i+1 到 n 序列中选择最小的元素所在位置 k。

程序如下：

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

```
const int MAXN=10001;
int main()
{
    int n,k,i,j;
    float temp,a[MAXN];
    cin>>n;
    for (i=0;i<n;i++)
      cin>>a[i];                    //输入 n 个数
    for (i=0;i<n;i++)               //i 控制当前序列中最小值存放的数据位置
    {
        k=i;
        for (j=i+1;j<n;j++)  //在当前无序区 a[i..n]中选最小的元素 a[k]
          if (a[j]<a[k])  k=j;
        if (k!=i)                   //交换 a[i]和 a[k],将当前最小值放到 a[i]位置
        {
            temp=a[i];a[i]=a[k];a[k]=temp;
        }
    }
    for (i=0;i<n;i++)
      cout<<a[i]<<" ";
    return 0;
}
```

2. 冒泡排序

冒泡排序的思想:以 n 个人站队为例,从第 1 个开始,依次比较相邻的两个是否逆序对(高在前,矮在后),若逆序就交换这两人,即第 1 个和第 2 个比,若逆序就交换两人,接着第 2 个和第 3 个比,若逆序就交换两人,接着第 3 个和第 4 个比,若逆序就交换两人,……,直到 n−1和 n 比较,经过一轮比较后,则把最高的人排到最后,即将最高的人像冒泡一样逐步冒到相应的位置。原 n 个人的排序问题,转换为 n−1 个人的排序问题。第二轮从第 1 个开始,依次比较相邻的两个人是否逆序对,若逆序就交换两人,直到 n−2 和 n−1 比较。如此,进行 n−1 轮后,队列为有序的队列。

从上述分析中可以看出,每进行一轮的比较之后,n 个数的排序规模就转化为 n−1 个数的排序规模。

例如有 6 个元素需要排序:

6 5 3 4 1 2

第一趟排序:

5 6 3 4 1 2
5 3 6 4 1 2
5 3 4 6 1 2
5 3 4 1 6 2
5 3 4 1 2 6 第一趟结束,最后的6固定下来

第二趟排序：

5 3 4 1 2 6
3 5 4 1 2 6
3 4 5 1 2 6
3 4 1 5 2 6
3 4 1 2 5 6　第二趟结束，5固定下来

第三趟排序：

3 4 1 2 5 6
3 1 4 2 5 6
3 1 2 4 5 6　第三趟结束，4固定下来

第四趟排序：

1 3 2 4 5 6
1 2 3 4 5 6　第四趟结束，3固定下来

第五趟排序：

1 2 3 4 5 6　第五趟结束，2固定下来

五趟结束后，6 个整数就已经排序完成。排序过程中，大数慢慢地往后，相当于气泡上升，所以叫冒泡排序。

归纳上述排序过程，具体实现步骤如下：

①读入数据存放在 a 数组中。

②比较相邻的前后两个数据，如果前面数据大于后面的数据，就将两个数据交换。

③对数组的第 0 个数据到 n－1 个数据进行一次遍历后，最大的一个数据就“冒”到数组第 n－1 个位置。

④n＝n－1，如果 n 不为 0 就重复前面二步，否则排序完成。

程序实现方法：用两层循环完成算法，外层循环 i 控制每轮要进行多少次的比较，第 1 轮比较 n－1 次，第 2 轮比较 n－2 次，……，最后一轮比较 1 次。内层循环 j 控制每轮 i 次比较相邻两个元素是否逆序，若逆序就交换这两个元素。

【程序实现】

```
#include<iostream>
using namespace std;
const int MAXN=10001;
int main()
{
    int n,i,j;
    float temp,a[MAXN];
    cin>>n;
    for (i=0;i<n;i++)
      cin>>a[i];                    //输入 n 个数
    for(i=n-1; i>=1; i--)           //进行 n-1 轮冒泡
    {
        for(j=0; j<=i; j++)         //每轮进行 i 次的比较
```

```
        {
            if(a[j]>a[j+1])              //相邻两个元素比较,若逆序就交换
              swap(a[j], a[j+1]);        //交换
        }
    }
    for (i=0;i<n;i++)                    //输出排序结果
      cout<<a[i]<<" ";
    return 0;
}
```

改进的冒泡排序:

对于有些数据,我们发现,不一定要 n-1 次才能排完。例如 1 5 2 3 4 6,我们发现只需一趟排序就可以将整个序列排完,于是,我们可以设置一个布尔变量,判断是否有进行交换,如果没有交换,说明已经排序完成,进而减少几趟排序。

【程序实现】

```
bool ok;
for(int i=n-1; i>=1; i--)
{
    ok=true;                             //判断是否有交换
    for(int j=1; j<=i; j++)
    {
        if(a[j]>a[j+1])
        {
            swap(a[j],a[j+1]);
            ok=false;
        }
    }
    if(ok==true) break;      //没有交换就退出
}
```

例 2.2 车厢重组

【问题描述】

在一个旧式的火车站旁边有一座桥,其桥面可以绕河中心的桥墩水平旋转。一个车站的职工发现桥的长度最多能容纳两节车厢,如果将桥旋转 180 度,则可以把相邻两节车厢的位置交换,用这种方法可以重新排列车厢的顺序。于是他就负责用这座桥将进站的车厢按车厢号从小到大排列。他退休后,火车站决定将这一工作自动化,其中一项重要的工作是编一个程序,输入初始的车厢顺序,计算最少用多少步就能将车厢排序。

【输入文件】

输入文件有两行数据,第一行是车厢总数 N(不大于 10000),第二行是 N 个不同的数表示初始的车厢顺序。

【输出文件】

一个数据,是最少的旋转次数。

【输入样例】

4

4 3 2 1

【输出样例】

6

程序如下：

```
#include<iostream>
#include<cstdio>
using namespace std;
long n,i,j,t,s,a[10000];
int main()
{
    cin>>n;
    for (i=1; i<=n; i++)
      cin>>a[i];                    //输入 n 个车厢号
    for (i=1; i<=n-1; i++)          //冒泡排序另一种写法
      for (j=1; j<=n-i; j++)
      if (a[j]>a[j+1])              //判断车厢号是否逆序
      {
        swap(a[j],a[j+1]);
        s++;                        //统计车厢旋转的次数
        };
      cout<<s;                      //最少的旋转次数
      return 0;
}
```

3. 插入排序

插入排序思想：回忆一下打牌时抓牌的情景，为了方便打牌，抓牌时一般一边抓牌一边按花色和大小插入恰当的位置，当抓完所有的牌时，手中的牌便是有序的，这种排序方法即插入排序。

当读入一个元素时，在已经排序好的序列中，搜寻它正确的位置，再放入读入的元素。但不该忽略一个重要的问题：在插入这个元素前，应当先将它后面的所有元素后移一位，以保证插入位置的原元素不被覆盖。

例如：设 n=8，数组 a 中 8 个元素是：36，25，48，12，65，43，20，58，执行插入排序程序后，其数据变动情况：

第 0 步：[36] 25 48 12 65 43 20 58

第 1 步：[25 36] 48 12 65 43 20 58

第 2 步：[25 36 48] 12 65 43 20 58

第 3 步：[12 25 36 48] 65 43 20 58

第 4 步：[12 25 36 48 65] 43 20 58

第 5 步：[12 25 36 43 48 65] 20 58

第 6 步:[12 20 25 36 43 48 65] 58
第 7 步:[12 20 25 36 43 48 58 65]
程序如下:

```
#include<iostream>
using namespace std;
const int MAXN=10001;
int main()
{
    int n,i,j,k;
    float temp,a[MAXN];
    cin>>n;
    for (i=0;i<n;i++)
      cin>>a[i];                    //输入 n 个数
    for(i=0; i<n; i++)
    {
      for(j=i-1; j>=0; j--) //在前面有序区间中为 a[i]找合适的插入位置
        if (a[j]<a[i]) break;       //找到比 a[i]小的位置就退出,插入其后
      if (j! =i-1)
      {
        temp=a[i];                  //将比 a[i]大的数据向后移
        for(k=i-1;k>j;k--)
          a[k+1]=a[k];              //将 a[i]放在正确位置上
        a[k+1]=temp;
      }
    }
    for (i=0;i<n;i++)
      cout<<a[i]<<" ";              //输出排序的结果
    return 0;
}
```

4. 桶排序

桶排序的思想是若待排序的值在一个明显有限范围内(整型)时,可设计有限个有序桶,待排序的值装入对应的桶(当然也可以装入若干个值),桶号就是待排序的值,顺序输出各桶的值,将得到有序的序列。

例:输入 n 个 0 到 100 之间的整数,由小到大排序输出。

【程序实现】

```
#include<iostream>
#include <cstring>
using namespace std;
int main()
{
```

```
    int b[101],n,i,j,k;
    memset(b,0,sizeof(b));                    //初始化
    cin>>n;
    for (i=1;i<=n;i++)
    {
      cin>>k; b[k]++;                         //将等于 k 的值全部装入第 k 桶中
    }
    for (i=0;i<=100;i++)                      //输出排序结果
      while (b[i]>0)                          //相同的整数,要重复输出
      {
        cout<<i<<" "; b[i]--;                 //输出一个整数后,个数减 1
      }
    cout<<endl;
}
```

运行结果：

样例输入：

10

2 3 1 2 4 55 3 55 3 2

样例输出：

1 2 2 2 3 3 3 4 55 55

例 2.3　明明的随机数(Noip2006)

【问题描述】

明明想在学校中请一些同学一起做一项问卷调查，为了实验的客观性，他先用计算机生成了 N 个 1 到 1000 之间的随机整数(N≤100)，对于其中重复的数字，只保留一个，把其余相同的数去掉，不同的数对应着不同的学生的学号。然后再把这些数从小到大排序，按照排好的顺序去找同学做调查。请你协助明明完成“去重”与“排序”的工作。

【输入文件】

输入文件 random.in 有 2 行，第 1 行为 1 个正整数，表示所生成的随机数的个数：N。第 2 行有 N 个用空格隔开的正整数，为所产生的随机数。

【输出文件】

输出文件 random.out 也是 2 行，第 1 行为 1 个正整数 M，表示不相同的随机数的个数。第 2 行为 M 个用空格隔开的正整数，为从小到大排好序的不相同的随机数。

【输入样例】

10

20 40 32 67 40 20 89 300 400 15

【输出样例】

8

15 20 32 40 67 89 300 400

【分析】

本题有一个重要的特点就是每一个数都介于 0 到 1000 之间的整数，可以开设一个下标

为0～1000的数组b,b[0]记录值为0的个数,b[1]记录值为1的个数,……,b[x]记录值为x的个数,那么从小到大的顺序输出b数组值不为0的b数组下标值。

程序如下:

```
#include<iostream>
#include<cstdio>
#include<cstring>
using namespace std;
int main()
{
    int b[1001],n,i,j,m=0,x;
    memset(b,0,sizeof(b));          //初始化
    cin>>n;
    for (i=1;i<=n;i++)
    {
        cin>>x;
        if (b[x]==0)   m++;  //b[x]为0表示x为新的随机数,m加1
        b[x]++;                     //将等于x的值全部装入第x桶中
    }
    cout<<m<<endl;                  //不相同的随机数的个数
    for (i=0;i<=1000;i++)           //输出排序结果
      if (b[i]>0) cout<<i<<" ";
    cout<<endl;
    return 0;
}
```

5.快速排序

快速排序是对冒泡排序的一种改进。它的基本思想是:通过一趟排序将待排记录分割成独立的两部分,其中一部分记录的关键字均比另一部分记录的关键字小,则可分别对这两部分记录继续进行排序,以达到整个序列有序。

假设待排序的序列为{a[l],a[l+1],a[l+2],…,a[r]},首先任意选取一个记录(通常可选中间一个记作为枢轴或支点),然后重新排列其余记录,将所有关键字小于它的记录都放在左子序列中,所有关键字大于它的记录都放在右子序列中。由此可以将该"支点"记录所在的位置mid作分界线,将序列分割成两个子序列和。这个过程称作一趟快速排序(或一次划分)。

一趟快速排序的具体做法是:附设两个指针i和j,它们的初值分别为l和r,设枢轴记录取mid,则首先从j所指位置起向前搜索找到第一个关键字小于mid的记录,然后从i所指位置起向后搜索,找到第一个关键字大于mid的记录,将它们互相交换,重复这两步直至i>j为止。

快速排序算法

```
#include<iostream>
using namespace std;
void qsort(int,int);
```

```
int a[101];
int main()
{
  int n,i;
  cin>>n;
  for (i=1;i<=n;i++)
    cin>>a[i];
  qsort(1,n);
  for (i=1;i<=n;i++)
    cout<<a[i]<<" ";
  cout<<endl;
}
void qsort(int l,int r)
  {
    int i,j,mid,p;
    i=l;j=r;
    mid=a[(l+r)/2];          //将当前序列在中间位置的数定义为分隔数
    do
      {
        while (a[i]<mid) i++;//在左半部分寻找比中间数大的数
        while (a[j]>mid) j--;//在右半部分寻找比中间数小的数
        if (i<=j)
          {                  //若找到一组与排序目标不一致的数对,则交换它们
            p=a[i];a[i]=a[j];a[j]=p;
            i++;j--;         //继续找
          }
      }while(i<=j);          //注意这里不能少了等号
    if (l<j) qsort(l,j);     //若未到两个数的边界,则递归搜索左右区间
    if (i<r) qsort(i,r);
  }
```

快速排序的时间的复杂性是 $O(n\log_2 n)$,速度快,但它是不稳定的排序方法。就平均时间而言,快速排序是目前被认为是最好的一种内部排序方法。

由以上讨论可知,从时间上看,快速排序的平均性能优于前面讨论过的各种排序方法,但快速排序需一个栈空间来实现递归。若每一趟排序都将记录序列均匀地分割成长度相接近的两个子序列,则栈的最大深度为 $\log(n+1)$。

6. 归并排序

归并排序是建立在归并操作上的一种有效的排序算法,该算法是采用分治法(Divide and Conquer)的一个非常典型的应用。将已有序的子序列合并,得到完全有序的序列;即先使每个子序列有序,再使子序列段间有序。若将两个有序表合并成一个有序表,称为二路归并。

例如有 8 个数据需要排序:10 4 6 3 8 2 5 7

归并排序主要分两大步:分解、合并。

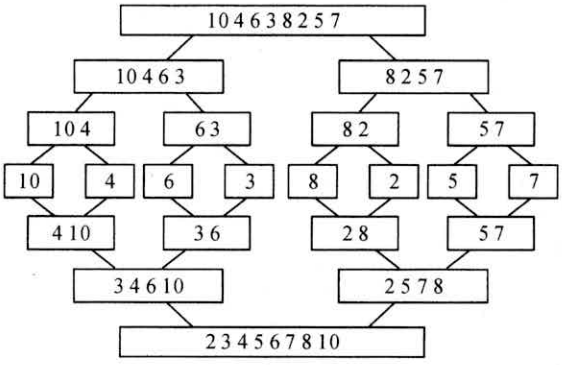


图 2-1

合并过程为:比较 a[i]和 a[j]的大小,若 a[i]≤a[j],则将第一个有序表中的元素 a[i]复制到 r[k]中,并令 i 和 k 分别加上 1;否则将第二个有序表中的元素 a[j]复制到 r[k]中,并令 j 和 k 分别加上 1,如此循环下去,直到其中一个有序表取完,然后再将另一个有序表中剩余的元素复制到 r 中从下标 k 到下标 t 的单元。归并排序的算法我们通常用递归实现,先把待排序区间[s,t]以中点二分,接着把左边子区间排序,再把右边子区间排序,最后把左区间和右区间用一次归并操作合并成有序的区间[s,t]。

【程序实现】

```
void msort(int s,int t)
{
    if(s==t) return;                //如果只有一个数字则返回,无须排序
    int mid=(s+t)/2;
    msort(s,mid);                   //分解左序列
    msort(mid+1,t);                 //分解右序列
    int i=s, j=mid+1, k=s;          //接下来合并

    while(i<=mid && j<=t)
    {
        if(a[i]<=a[j])
        {
        r[k]=a[i]; k++; i++;
        }else{
        r[k]=a[j]; k++; j++;
        }
    }
    while(i<=mid)                   //复制左边子序列剩余
    {
        r[k]=a[i]; k++; i++;
    }
```

```
    while(j<=t)                    //复制右边子序列剩余
    {
        r[k]=a[j]; k++; j++;
    }
    for(int i=s; i<=t; i++) a[i]=r[i];
}
```

归并排序的时间复杂度是O(nlogn)，速度快。同时，归并排序是稳定的排序。即相等的元素的顺序不会改变。如输入记录1(1) 3(2) 2(3) 2(4) 5(5)（括号中是记录的关键字）时输出的1(1) 2(3) 2(4) 3(2) 5(5)中的2和2是按输入的顺序。这对要排序数据包含多个信息而要按其中的某一个信息排序，要求其它信息尽量按输入的顺序排列时很重要，这也是它比快速排序优势的地方。

7. 逆序对

上述提到归并排序是稳定的排序，相等的元素的顺序不会改变，进而用其可以解决逆序对的问题。首先我们了解一下什么是逆序对。

逆序对：设A为一个有n个数字的有序集（n>1），其中所有数字各不相同。如果存在正整数i，j使得$1 \leqslant i < j \leqslant n$而且A[i] > A[j]，则<A[i]，A[j]>这个有序对称为A的一个逆序对，也称作逆序数。

例如，数组(3,1,4,5,2)的逆序对有(3,1)，(3,2)，(4,2)，(5,2)，共4个。

所谓逆序对的问题，即对给定的数组序列，求其逆序对的数量。

从逆序对定义上分析，逆序对就是数列中任意两个数满足大的在前，小的在后的组合。如果将这些逆序对都调整成顺序（小的在前，大的在后），那么整个数列就变得有序，即排序。因而，容易想到冒泡排序的机制正好是利用消除逆序来实现排序的，也就是说，交换相邻两个逆序数，最终实现整个序列有序，那么交换的次数即为逆序对的数量。

冒泡排序可以解决逆序对问题，但是由于冒泡排序本身效率不高，时间复杂度为O(n^2)，对于n比较大的情况就没用武之地了。我们可以这样认为，冒泡排序求逆序对效率之所以低，是因为其在统计逆序对数量的时候是一对一对统计的，而对于范围为n的序列，逆序对数量最大可以是(n+1) * n/2，因此其效率太低。那怎样可以一下子统计多个，而不是一个一个累加呢？这个时候，归并排序就可以帮我们来解决这个问题。

在合并操作中，我们假设左右两个区间元素为：

左边：{3 4 7 9}　　右边：{1 5 8 10}

那么合并操作的第一步就是比较3和1，然后将1取出来，放到辅助数组中，这个时候我们发现，右边的区间如果是当前比较的较小值，那么其会与左边剩余的数字产生逆序关系，也就是说1和3、4、7、9都产生了逆序关系，我们可以一下子统计出有4对逆序对。接下来3,4取下来放到辅助数组后，5与左边剩下的7、9产生了逆序关系，我们可以统计出2对。依此类推，8与9产生1对，那么总共有4+2+1对。这样统计的效率就会大大提高，便可较好地解决逆序对问题。

而在算法的实现中，我们只需略微修改原有归并排序，当右边序列的元素为较小值时，就统计其产生的逆序对数量，即可完成逆序对的统计。

【程序实现】

```
void msort(int s,int t)
{
    if(s==t) return;                    //如果只有一个数字则返回,无须排序
  int mid=(s+t)/2;
  msort(s,mid);                         //分解左序列
  msort(mid+1,t);                       //分解右序列
  int i=s, j=mid+1, k=s;                //接下来合并
  while(i<=mid && j<=t)
  {
      if(a[i]<=a[j])
      {
          r[k]=a[i]; k++; i++;
      }else{
          r[k]=a[j]; k++; j++;
          ans+=mid-i+1;                 //统计产生逆序对的数量
      }
  }
  while(i<=mid)                         //复制左边子序列剩余
  {
      r[k]=a[i]; k++; i++;
  }
  while(j<=t)                           //复制右边子序列剩余
  {
      r[k]=a[j]; k++; j++;
  }
  for(int i=s; i<=t; i++) a[i]=r[i];
}
```

其中 ans+=mid-i+1 这句代码统计新增逆序对的数量,ans 作为全局变量,用于统计逆序对的数量,此时 ans 要增加左边区间剩余元素的个数。当归并排序结束后,逆序对问题也得到解决,ans 即为逆序对的数量。

8. 各种排序算法的比较

(1)稳定性比较

插入排序、冒泡排序、二叉树排序、二路归并排序及其他线性排序是稳定的。

选择排序、希尔排序、快速排序、堆排序是不稳定的。即有跨度的交换都会导致不稳定。

(2)时间复杂性比较

插入排序、冒泡排序、选择排序的时间复杂性为 $O(n^2)$;快速排序、堆排序、归并排序的时间复杂性为 $O(n\log_2 n)$;桶排序的时间复杂性为 $O(n)$;

若从最好情况考虑，则直接插入排序和冒泡排序的时间复杂度最好，为 O(n)，其他算法的最好情况同平均情况相同；若从最坏情况考虑，则快速排序的时间复杂度为 $O(n^2)$，直接插入排序和冒泡排序虽然平均情况相同，但系数大约增加一倍，所以运行速度将降低一半，最坏情况对直接选择排序、堆排序和归并排序影响不大。

由此可知，在最好情况下，直接插入排序和冒泡排序最快；在平均情况下，快速排序最快；在最坏情况下，堆排序和归并排序最快。

(3)辅助空间的比较

桶排序、二路归并排序的辅助空间为 O(n)，快速排序的辅助空间为 $O(\log_2 n)$，最坏情况为 O(n)，其他排序的辅助空间为 O(1)。

(4)其他比较

插入、冒泡排序的速度较慢，但参加排序的序列局部或整体有序时，这种排序能达到较快的速度。而在这种情况下，快速排序反而慢了。

当 n 较小时，对稳定性不作要求时宜用选择排序，对稳定性有要求时宜用插入或冒泡排序。

若待排序的记录的关键字在一个明显有限范围内时，且空间允许是用桶排序。

当 n 较大时，关键字元素比较随机，对稳定性没要求宜用快速排序。

当 n 较大时，关键字元素可能出现本身是有序的，对稳定性没有要求时宜用堆排序。

快速排序是目前基于比较的内部排序中被认为是最好的方法，当待排序的关键字是随机分布时，快速排序的平均时间最短。

堆排序所需的辅助空间少于快速排序，并且不会出现快速排序可能出现的最坏情况。这两种排序都是不稳定的。

【上机练习】

1. 谁考了第 k 名【1.10 编程基础之简单排序 01】

在一次考试中,每个学生的成绩都不相同,现知道了每个学生的学号和成绩,求考第 k 名学生的学号和成绩。

输入:

第一行有两个整数,分别是学生的人数 n(1≤n≤100),和求第 k 名学生的 k(1≤k≤n)。

其后有 n 行数据,每行包括一个学号(整数)和一个成绩(浮点数),中间用一个空格分隔。

输出:

输出第 k 名学生的学号和成绩,中间用空格分隔。(注:请用%g 输出成绩)

样例输入:

5 3

90788001 67.8

90788002 90.3

90788003 61

90788004 68.4

90788005 73.9

样例输出:

90788004 68.4

2. 奇数单增序列【1.10 编程基础之简单排序 02】

给定一个长度为 N(不大于 500)的正整数序列,请将其中的所有奇数取出,并按升序输出。

输入:

第 1 行为 N;

第 2 行为 N 个正整数,其间用空格间隔。

输出:

增序输出的奇数序列,数据之间以逗号间隔。数据保证至少有一个奇数。

样例输入:

10

1 3 2 6 5 4 9 8 7 10

样例输出:

1,3,5,7,9

3. 成绩排序【1.10 编程基础之简单排序 03】

给出班里某门课程的成绩单,请你按成绩从高到低对成绩单排序输出,如果有相同分数则名字字典序小的在前。

输入:

第一行为 n (0 < n < 20),表示班里的学生数目;

接下来的 n 行，每行为每个学生的名字和他的成绩，中间用单个空格隔开。名字只包含字母且长度不超过 20，成绩为一个不大于 100 的非负整数。

输出：

把成绩单按分数从高到低的顺序进行排序并输出，每行包含名字和分数两项，之间有一个空格。

样例输入：

4

Kitty 80

Hanmeimei 90

Joey 92

Tim 28

样例输出：

Joey 92

Hanmeimei 90

Kitty 80

Tim 28

4. 奖学金【1.10 编程基础之简单排序 04】Noip2007 普及组第 1 题

某小学最近得到了一笔赞助，打算拿出其中一部分为学习成绩优秀的前 5 名学生发奖学金。期末，每个学生都有 3 门课的成绩：语文、数学、英语。先按总分从高到低排序，如果两个同学总分相同，再按语文成绩从高到低排序，如果两个同学总分和语文成绩都相同，那么规定学号小的同学排在前面，这样，每个学生的排序是唯一确定的。

任务：先根据输入的 3 门课的成绩计算总分，然后按上述规则排序，最后按排名顺序输出前五名名学生的学号和总分。注意，在前 5 名同学中，每个人的奖学金都不相同，因此，你必须严格按上述规则排序。例如，在某个正确答案中，如果前两行的输出数据（每行输出两个数：学号、总分）是：

7 279

5 279

这两行数据的含义是：总分最高的两个同学的学号依次是 7 号、5 号。这两名同学的总分都是 279（总分等于输入的语文、数学、英语三科成绩之和），但学号为 7 的学生语文成绩更高一些。如果你的前两名的输出数据是：

5 279

7 279

则按输出错误处理，不能得分。

输入：

包含 n+1 行：

第 1 行为一个正整数 n，表示该校参加评选的学生人数。

第 2 到 n+1 行，每行有 3 个用空格隔开的数字，每个数字都在 0 到 100 之间。第 j 行的 3 个数字依次表示学号为 j−1 的学生的语文、数学、英语的成绩。每个学生的学号按照输入顺序编号为 1～n（恰好是输入数据的行号减 1）。

输出：

共有5行，每行是两个用空格隔开的正整数，依次表示前5名学生的学号和总分。

样例输入：

样例 #1：

```
6
90 67 80
87 66 91
78 89 91
88 99 77
67 89 64
78 89 98
```

样例 #2：

```
8
80 89 89
88 98 78
90 67 80
87 66 91
78 89 91
88 99 77
67 89 64
78 89 98
```

样例输出：

样例 #1：

```
6 265
4 264
3 258
2 244
1 237
```

样例 #2：

```
8 265
2 264
6 264
1 258
5 258
```

5. 分数线划定【1.10 编程基础之简单排序 05】Noip2009 普及组第2题

世博会志愿者的选拔工作正在A市如火如荼的进行。为了选拔最合适的人才，A市对所有报名的选手进行了笔试，笔试分数达到面试分数线的选手方可进入面试。面试分数线根据计划录取人数的150%划定，即如果计划录取m名志愿者，则面试分数线为排名第m*150%(向下取整)名的选手的分数，而最终进入面试的选手为笔试成绩不低于面试分数线的

所有选手。

现在就请你编写程序划定面试分数线，并输出所有进入面试的选手的报名号和笔试成绩。

输入：

第一行，两个整数 n，m(5 ≤ n ≤ 5000，3 ≤ m ≤ n)，中间用一个空格隔开，其中 n 表示报名参加笔试的选手总数，m 表示计划录取的志愿者人数。输入数据保证 m * 150%向下取整后小于等于 n。

第二行到第 n+1 行，每行包括两个整数，中间用一个空格隔开，分别是选手的报名号 k(1000 ≤ k ≤ 9999)和该选手的笔试成绩 s(1 ≤ s ≤ 100)。数据保证选手的报名号各不相同。

输出：

第一行，有两个整数，用一个空格隔开，第一个整数表示面试分数线；第二个整数为进入面试的选手的实际人数。

从第二行开始，每行包含两个整数，中间用一个空格隔开，分别表示进入面试的选手的报名号和笔试成绩，按照笔试成绩从高到低输出，如果成绩相同，则按报名号由小到大的顺序输出。

样例输入：

6 3
1000 90
3239 88
2390 95
7231 84
1005 95
1001 88

样例输出：

88 5
1005 95
2390 95
1000 90
1001 88
3239 88

提示：

样例说明：m * 150%＝3 * 150%＝4.5，向下取整后为 4。保证 4 个人进入面试的分数线为 88，但因为 88 有重分，所以所有成绩大于等于 88 的选手都可以进入面试，故最终有 5 个人进入面试。

6. 整数奇偶排序【1.10 编程基础之简单排序 06】

给定 10 个整数的序列，要求对其重新排序。排序要求：

1. 奇数在前，偶数在后；

2. 奇数按从大到小排序；

3. 偶数按从小到大排序。

输入:

输入一行,包含 10 个整数,彼此以一个空格分开,每个整数的范围是大于等于 0,小于等于 100。

输出:

按照要求排序后输出一行,包含排序后的 10 个整数,数与数之间以一个空格分开。

样例输入:

4 7 3 13 11 12 0 47 34 98

样例输出:

47 13 11 7 3 0 4 12 34 98

7. 合影效果【1.10 编程基础之简单排序 07】

小云和朋友们去爬香山,为美丽的景色所陶醉,想合影留念。如果他们站成一排,男生全部在左(从拍照者的角度),并按照从矮到高的顺序从左到右排,女生全部在右,并按照从高到矮的顺序从左到右排,请问他们合影的效果是什么样的(所有人的身高都不同)?

输入:

第一行是人数 n(2 <=n <=40,且至少有 1 个男生和 1 个女生)。

后面紧跟 n 行,每行输入一个人的性别(男 male 或女 female)和身高(浮点数,单位米),两个数据之间以空格分隔。

输出:

n 个浮点数,模拟站好队后,拍照者眼中从左到右每个人的身高。每个浮点数需保留到小数点后 2 位,相邻两个数之间用单个空格隔开。

样例输入:

6

male 1.72

male 1.78

female 1.61

male 1.65

female 1.70

female 1.56

样例输出:

1.65 1.72 1.78 1.70 1.61 1.56

8. 病人排队【1.10 编程基础之简单排序 08】

病人登记看病,编写一个程序,将登记的病人按照以下原则排出看病的先后顺序:

1. 老年人(年龄 >=60 岁)比非老年人优先看病。

2. 老年人按年龄从大到小的顺序看病,年龄相同的按登记的先后顺序排序。

3. 非老年人按登记的先后顺序看病。

输入:

第 1 行,输入一个小于 100 的正整数,表示病人的个数;

后面按照病人登记的先后顺序,每行输入一个病人的信息,包括:一个长度小于 10 的字

符串表示病人的 ID(每个病人的 ID 各不相同且只含数字和字母),一个整数表示病人的年龄,中间用单个空格隔开。

输出:

按排好的看病顺序输出病人的 ID,每行一个。

样例输入:

5

021075 40

004003 15

010158 67

021033 75

102012 30

样例输出:

021033

010158

021075

004003

102012

9. 明明的随机数【1.10 编程基础之简单排序 09】Noip2006 普及组第 1 题

明明想在学校中请一些同学一起做一项问卷调查,为了实验的客观性,他先用计算机生成了 N 个 1 到 1000 之间的随机整数(N≤100),对于其中重复的数字,只保留一个,把其余相同的数去掉,不同的数对应着不同的学生的学号。然后再把这些数从小到大排序,按照排好的顺序去找同学做调查。请你协助明明完成“去重”与“排序”的工作。

输入:

有 2 行,第 1 行为 1 个正整数,表示所生成的随机数的个数:N;

第 2 行有 N 个用空格隔开的正整数,为所产生的随机数。

输出:

也是 2 行,第 1 行为 1 个正整数 M,表示不相同的随机数的个数。第 2 行为 M 个用空格隔开的正整数,为从小到大排好序的不相同的随机数。

样例输入:

10

20 40 32 67 40 20 89 300 400 15

样例输出:

8

15 20 32 40 67 89 300 400

10. 单词排序【1.10 编程基础之简单排序 10】

输入一行单词序列,相邻单词之间由 1 个或多个空格间隔,请按照字典序输出这些单词,要求重复的单词只输出一次。(区分大小写)

输入:

一行单词序列,最少 1 个单词,最多 100 个单词,每个单词长度不超过 50,单词之间用至

少1个空格间隔。数据不含除字母、空格外的其他字符。

输出:

按字典序输出这些单词,重复的单词只输出一次。

样例输入:

She wants to go to Peking University to study Chinese

样例输出:

Chinese

Peking

She

University

go

study

to

wants

11. 出现次数超过一半的数【1.13 编程基础之综合应用 28】

给出一个含有 n(0 < n <=1000)个整数的数组,请找出其中出现次数超过一半的数。数组中的数大于-50且小于50。

输入:

第一行包含一个整数 n,表示数组大小;

第二行包含 n 个整数,分别是数组中的每个元素,相邻两个元素之间用单个空格隔开。

输出:

如果存在这样的数,输出这个数;否则输出 no。

样例输入:

3

1 2 2

样例输出:

2

12. 统计字符数【1.13 编程基础之综合应用 29】

给定一个由 a-z 这 26 个字符组成的字符串,统计其中哪个字符出现的次数最多。

输入:

输入包含一行,一个字符串,长度不超过 1000。

输出:

输出一行,包括出现次数最多的字符和该字符出现的次数,中间以一个空格分开。如果有多个字符出现的次数相同且最多,那么输出 ascii 码最小的那一个字符。

样例输入:

abbccc

样例输出:

c 3

第三章　递推算法

递推法是一种重要的数学方法，在数学的各个领域中都有广泛的运用，也是计算机用于数值计算的一个重要算法。这种算法特点是：一个问题的求解需一系列的计算，在已知条件和所求问题之间总存在着某种相互联系的关系，在计算时，如果可以找到前后过程之间的数量关系（即递推式），那么，从问题出发逐步推到已知条件，此种方法叫逆推。无论顺推还是逆推，其关键是要找到递推式。这种处理问题的方法能使复杂运算化为若干步重复的简单运算，充分发挥出计算机擅长于重复处理的特点。

递推算法的首要问题是得到相邻的数据项间的关系（即递推关系）。递推算法避开了求通项公式的麻烦，把一个复杂的问题的求解，分解成了连续的若干步简单运算。一般说来，可以将递推算法看成是一种特殊的迭代算法。

例 3.1　数塔问题。如图 3－1 所示为一个数字三角形。请编一个程序计算从顶到底的某处的一条路径，使该路径所经过的数字总和最大。只要求输出总和。

1. 一步可沿左斜线向下或右斜线向下走；
2. 三角形行数小于等于 100；
3. 三角形中的数字为 0,1,…,99；

测试数据通过键盘逐行输入，如上例数据应以如图 3－2 所示格式输入：

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

图 3－1

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

图 3－2

【算法分析】

此题解法有多种，从递推的思想出发，设想，当从顶层沿某条路径走到第 i 层向第 i＋1 层前进时，我们的选择一定是沿其下两条可行路径中最大数字的方向前进，为此，我们可以采用倒推的手法，设 a[i][j]存放从 i,j 出发到达 n 层的最大值，则 a[i][j]＝max{a[i][j]＋a[i＋1][j],a[i][j]＋a[i＋1][j＋1]},a[1][1] 即为所求的数字总和的最大值。

【参考程序】

```
#include<iostream>
using namespace std;
int main()
{
  int n,i,j,a[101][101];
  cin>>n;
```

```
    for (i=1;i<=n;i++)
      for (j=1;j<=i;j++)
        cin>>a[i][j];                    //输入数字三角形的值
    for (i=n-1;i>=1;i--)
      for (j=1;j<=i;j++)
        {
          if (a[i+1][j]>=a[i+1][j+1])  a[i][j]+=a[i+1][j]; //路径选择
          else  a[i][j]+=a[i+1][j+1];
        }
      cout<<a[1][1]<<endl;
}
```

例 3.2 满足 $F_1=F_2=1, F_n=F_{n-1}+F_{n-2}$ 的数列称为斐波那契数列(Fibonacci),它的前若干项是 1,1,2,3,5,8,13,21,34……求此数列第 n 项(n>=3)。

即:f1=1 (n=1)

f2=1 (n=2)

fn=fn-1 + fn-2 (n>=3)

程序如下:

```
#include<iostream>
#include<cstdio>
using namespace std;
int main()
{
    int f0=1,f1=1,f2;
    int n;
    cin>>n;
    for (int i=3; i<=n; ++i)
    {
        f2=f0+f1;
        f0=f1;
        f1=f2;
    }
    printf("%d\n",f2);
    return 0;
}
```

有关 Fibonacci 数列,我们先来考虑一个简单的问题:楼梯有 n 个台阶,上楼可以一步上一阶,也可以一步上两阶。一共有多少种上楼的方法?

这是一道计数问题。在没有思路时,不妨试着找规律。n=5 时,一共有 8 种方法:

5=1+1+1+1+1

5=2+1+1+1

5=1+2+1+1

5＝1＋1＋2＋1

5＝1＋1＋1＋2

5＝2＋2＋1

5＝2＋1＋2

5＝1＋2＋2

其中有 5 种方法第 1 步走了 1 阶(背景灰色),3 种方法第 1 步走了 2 阶,没有其他可能。假设 f(n)为 n 个台阶的走法总数,把 n 个台阶的走法分成两类。

第 1 类:第 1 步走 1 阶,剩下还有 n－1 阶要走,有 f(n－1)种方法。

第 2 类:第 1 步走 2 阶,剩下还有 n－2 阶要走,有 f(n－2)种方法。

这样,就得到了递推式:f(n)＝f(n－1)＋f(n－2),不要忘记边界情况:f(1)＝1,f(2)＝2。把 f(n)的前几项列出:1,2,3,5,8,……。

再例如,把雌雄各一的一对新兔子放入养殖场中。每只雌兔在出生两个月以后,每月产雌雄各一的一对新兔子。试问第 n 个月后养殖场中共有多少对兔子。

还是先找找规律。

第 1 个月:一对新兔子 r1。用小写字母表示新兔子。

第 2 个月:还是一对新兔子,不过已经长大,具备生育能力了,用大写字母 R1 表示。

第 3 个月:R1 生了一对新兔子 r2,一共 2 对。

第 4 个月:R1 又生一对新兔子 r3,一共 3 对。另外,r2 长大了,变成 R2

第 5 个月:R1 和 R2 各生一对,记为 r4 和 r5,共 5 对。此外,r3 长成 R3。

第 6 个月:R1、R2 和 R3 各生一对,记为 r6～r8,共 8 对。此外,r4 和 r5 长大。

……

把这些数排列起来:1,1,2,3,5,8,……,事实上,可以直接推导出来递推关系 f(n)＝f(n－1)＋f(n－2):第 n 个月的兔子由两部分组成,一部分是上个月就有的老兔子 f(n－1),一部分是上个月出生的新兔子 f(n－2)(第 n－1 个月时具有生育能力的兔子数就等于第 n－2个月兔子总数)。根据加法原理,f(n)＝f(n－1)＋f(n－2)。

例 3.3　有 2＊n 的一个长方形方格,用一个 1＊2 的骨牌铺满方格。

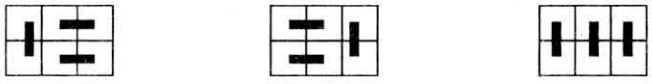


图 3－3

编写一个程序,试对给出的任意一个 n(n＞0),输出铺法总数。

【算法分析】

(1)面对上述问题,如果思考方法不恰当,要想获得问题的解答是相当困难的。可以用递推方法归纳出问题解的一般规律。

(2)当 n＝1 时,只能是一种铺法,铺法总数有示为 $x_1=1$。

(3)当 n＝2 时,骨牌可以两个并列竖排,也可以并列横排,再无其他方法,如图 3－4 所示,因此,铺法总数表示为 $x_2=2$。

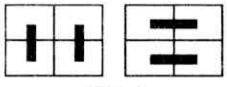


图 3－4

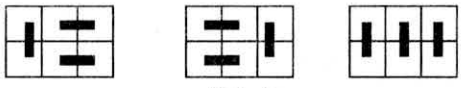


图 3－5

(4)当 n=3 时,骨牌可以全部竖排,也可以认为在方格中已经有一个竖排骨牌,则需要在方格中排列两个横排骨牌(无重复方法),若已经在方格中排列两个横排骨牌,则必须在方格中排列一个竖排骨牌(如图 3-5),再无其他排列方法,因此铺法总数表示为 $x_3=3$。

由此可以看出,当 n=3 时的排列骨牌的方法数是 n=1 和 n=2 排列方法数的和。

(5)推出一般规律:对一般的 n,要求 x_n 可以这样来考虑,若第一个骨牌是竖排列放置,剩下有 n-1 个骨牌需要排列,这时排列方法数为 x_{n-1};若第一个骨牌是横排列,整个方格至少有 2 个骨牌是横排列(1*2 骨牌),因此剩下 n-2 个骨牌需要排列,这是骨牌排列方法数为 x_{n-2}。从第一骨牌排列方法考虑,只有这两种可能,所以有:

$$x_n=x_{n-1}+x_{n-2} \quad (n>2)$$
$$x_1=1$$
$$x_2=2$$

$x_n=x_{n-1}+x_{n-2}$ 就是问题求解的递推公式,任给 n 都可以从中获得解答。例如 n=5,

$$x_3=x_2+x_1=3$$
$$x_4=x_3+x_2=5$$
$$x_5=x_4+x_3=8$$

下面是输入 n,输出 $x_1\sim x_n$ 的 C++程序:

```
#include<iostream>
using namespace std;
int main()
{
  int n,i,j,a[101];
  cout<<" input n:";                    //输入骨牌数
  cin>>n;
  a[1]=1;a[2]=2;
  cout<<" x[1]="<<a[1]<<endl;
  cout<<" x[2]="<<a[2]<<endl;
  for (i=3;i<=n;i++)                    //递推过程
    {
      a[i]=a[i-1]+a[i-2];
      cout<<" x["<<i<<"]="<<a[i]<<endl;
    }
}
```

下面是运行程序输入 n=30,输出的结果:

```
input n: 30
x[1]=1
x[2]=2
x[3]=3
...
x[29]=832040
x[30]=1346269
```

问题的结果就是有名的斐波那契数。

例 3.4　昆虫繁殖

【问题描述】

科学家在热带森林中发现了一种特殊的昆虫，这种昆虫的繁殖能力很强。每对成虫过 x 个月产 y 对卵，每对卵要过两个月长成成虫。假设每个成虫不死，第一个月只有一对成虫，且卵长成成虫后的第一个月不产卵(过 x 个月产卵)，问过 z 个月以后，共有成虫多少对？0<=x<=20，1<=y<=20，x<=z<=50

【输入格式】

x，y，z 的数值。

【输出格式】

过 z 个月以后，共有成虫对数

【输入样例】　　【输出样例】

1 2 8　　37

【参考程序】

```
#include<iostream>
using namespace std;
int main()
{
  long long a[101]={0},b[101]={0},i,j,x,y,z;
  cin>>x>>y>>z;
  for(i=1;i<=x;i++){a[i]=1;b[i]=0;}
  for(i=x+1;i<=z+1;i++)        //因为要统计到第 z 个月后，所以要 for 到 z+1
  {
    b[i]=y*a[i-x];
    a[i]=a[i-1]+b[i-2];
  }
  cout<<a[z+1]<<endl;
  return 0;
}
```

例 3.5　位数问题

【问题描述】

在所有的 n 位数中，有多少个数中有偶数个数字 3？由于结果可能很大，你只需要输出这个答案对 12345 取余的值。

【输入格式】

读入一个数 n。

【输出格式】

输出有多少个数中有偶数个数字 3。

【输入样例】　　【输出样例】

2　　73

【数据范围】

1<=n<=1000

【样例说明】

在所有的2位数字,包含0个3的数有72个,包含2个3的数有1个,共73个。

【算法分析】

方法一:排列组合(但需要运用动态规划)。

可以列出公式,在n个格子中放x个3(其中x为偶数,包括0)。

c(n,x)*9^(n-x)-c(n-1,x)*9^(n-x-1)含义为在n个格子中取x个3,且不考虑第一位的特殊情况为c(n,x)*9^(n-x),而第一位为0的情况为c(n-1,x)*9^(n-x-1),两者相减,就为答案。

方法二:递推

考虑这种题目,一般来说都是从第i-1位推导第i位,且当前位是取偶数还是取奇数的。

恍然大悟,可以用f[i][0]表示前i位取偶数个3有几种情况,f[i][1]表示前i位取奇数个3有几种情况。

则状态转移方程可以表示为:

f[i][0]=f[i-1][0]*9+f[i-1][1];f[i][1]=f[i-1][0]+f[i-1][1]*9;

边界条件:f[1][1]=1;f[1][0]=9;

【参考程序】

```
#include<iostream>
using namespace std;
int main()
{
  int f[1001][2],n,i,x;
  cin>>n;
  f[1][1]=1;f[1][0]=9;
  for(i=2;i<=n;i++)
    {
      x=f[1][0];
      if(i==n)x--;
      f[i][0]=(f[i-1][0]*x+f[i-1][1])%12345;
      f[i][1]=(f[i-1][1]*x+f[i-1][0])%12345;
    }
  cout<<f[n][0];
  return 0;
}
```

例3.6 过河卒(Noip2002)

【问题描述】

棋盘上A点有一个过河卒,需要走到目标B点。卒行走的规则:可以向下或者向右。同时在棋盘上的某一点有一个对方的马(如C点),该马所在的点和所有跳跃一步可达的点

称为对方马的控制点，如图 3－6 中的 C 点和 P1，…，P8，卒不能通过对方马的控制点。棋盘用坐标表示，A 点(0，0)、B 点(n，m)(n，m 为不超过 20 的整数)，同样马的位置坐标是需要给出的，C≠A 且 C≠B。现在要求你计算出卒从 A 点能够到达 B 点的路径的条数。

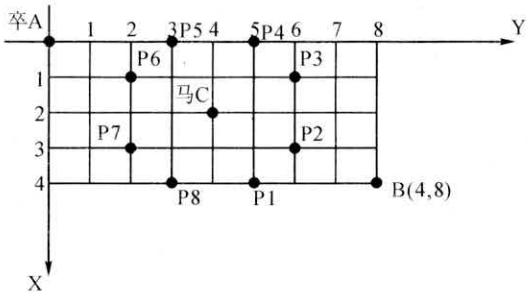


图 3－6

【算法分析】

跳马是一道老得不能再老的题目，我想每位编程初学者都学过，可能是在学回溯或搜索等算法的时候，很多书上也有类似的题目，一些比赛中也出现过这一问题的变形(如 Noip1997 初中组的第三题)。有些同学一看到这条题目就去搜索，即使你编程调试全通过了，运行时你也会发现：当 n、m＝15 就会超时。

其实，本题稍加分析就能发现，要到达棋盘上的一个点，只能从左边过来(我们称之为左点)或是从上面过来(我们称之为上点)，所以根据加法原理，到达某一点的路径数目，就等于到达其相邻的上点和左点的路径数目之和，因此我们可以使用逐列(或逐行)递推的方法来求出从起点到终点的路径数目。障碍点(马的控制点)也完全适用，只要将到达该点的路径数目设置为 0 即可。

用 f[i][j]表示到达点(i，j)的路径数目，g[i][j]表示点(i，j)有无障碍，g[i][j]＝0 表示无障碍，g[i][j]＝1 表示有障碍。

则，递推关系式如下：

```
f[i][j]=f[i-1][j]+f[i][j-1]    //i>0 且 j>0 且 g[i][j]=0
```

递推边界有 4 个：

```
f[i][j]=0                      //g[i][j]=1
f[i][0]=f[i-1][0]              //i > 0 且 g[i][0]=0
f[0][j]=f[0][j-1]              //j > 0 且 g[0][j]=0
f[0][0]=1
```

考虑到最大情况下：n＝20，m＝20，路径条数可能会超过 $2^{31}-1$，所以要用高精度。

五种典型的递推关系

1. Fibonacci 数列

在所有的递推关系中，Fibonacci 数列应该是最为大家所熟悉的。在最基础的程序设计语言 Logo 语言中，就有很多这类的题目。而在较为复杂的 Basic、Pascal、C 语言中，Fibonacci 数列类的题目因为解法相对容易一些，逐渐退出了竞赛的舞台。可是这不等于说 Fibonacci 数列没有研究价值，恰恰相反，一些此类的题目还是能给我们一定的启发的。

Fibonacci 数列的代表问题是由意大利著名数学家 Fibonacci 于 1202 年提出的“兔子繁

殖问题"(又称"Fibonacci 问题")。

问题的提出:有雌雄一对兔子,假定过两个月便可繁殖雌雄各一的一对小兔子。问过 n 个月后共有多少对兔子?

解:设满 x 个月共有兔子 F_x 对,其中当月新生的兔子数目为 N_x 对。第 x-1 个月留下的兔子数目设为 F_{x-1} 对。则:

$F_x=N_x+F_{x-1}$

$N_x=F_{x-2}$ (即第 x-2 个月的所有兔子到第 x 个月都有繁殖能力了)

$\therefore\ F_x=F_{x-1}+F_{x-2}$ 边界条件:$F_0=0,F_1=1$

由上面的递推关系可依次得到

$F_2=F_1+F_0=1,F_3=F_2+F_1=2,F_4=F_3+F_2=3,F_5=F_4+F_3=5,\cdots$

Fabonacci 数列常出现在比较简单的组合计数问题中,例如以前的竞赛中出现的"骨牌覆盖"问题。在优选法中,Fibonacci 数列的用处也得到了较好的体现。

2. Hanoi 塔问题

问题的提出:Hanoi 塔由 n 个大小不同的圆盘和三根木柱 a、b、c 组成。开始时,这 n 个圆盘由大到小依次套在 a 柱上,如图 3-7 所示。

要求把 a 柱上 n 个圆盘按下述规则移到 c 柱上:

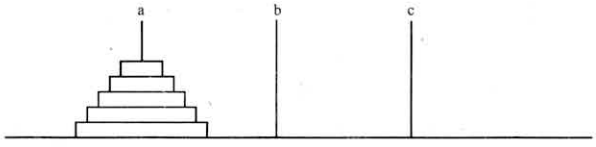


图 3-7

(1)一次只能移一个圆盘;

(2)圆盘只能在三个柱上存放;

(3)在移动过程中,不允许大盘压小盘。

问将这 n 个盘子从 a 柱移动到 c 柱上,总计需要移动多少个盘次?

解:设 h_n 为 n 个盘子从 a 柱移到 c 柱所需移动的盘次。显然,当 n=1 时,只需把 a 柱上的盘子直接移动到 c 柱就可以了,故 $h_1=1$。当 n=2 时,先将 a 柱上面的小盘子移动到 b 柱上去;然后将大盘子从 a 柱移到 c 柱;最后,将 b 柱上的小盘子移到 c 柱上,共记 3 个盘次,故 $h_2=3$。以此类推,当 a 柱上有 n(n>=2)个盘子时,总是先借助 c 柱把上面的 n-1 个盘子移动到 b 柱上,然后把 a 柱最下面的盘子移动到 c 柱上;再借助 a 柱把 b 柱上的 n-1 个盘子移动到 c 柱上;总共移动 $h_{n-1}+1+h_{n-1}$ 个盘次。

$\therefore h_n=2h_{n-1}+1$ 边界条件:$h_1=1$

3. 平面分割问题

问题的提出:设有 n 条封闭曲线画在平面上,而任何两条封闭曲线恰好相交于两点,且任何三条封闭曲线不相交于同一点,问这些封闭曲线把平面分割成的区域个数。

解:设 a_n 为 n 条封闭曲线把平面分割成的区域个数。由图 3-8 可以看出:$a_2-a_1=2$;$a_3-a_2=4$;$a_4-a_3=6$。

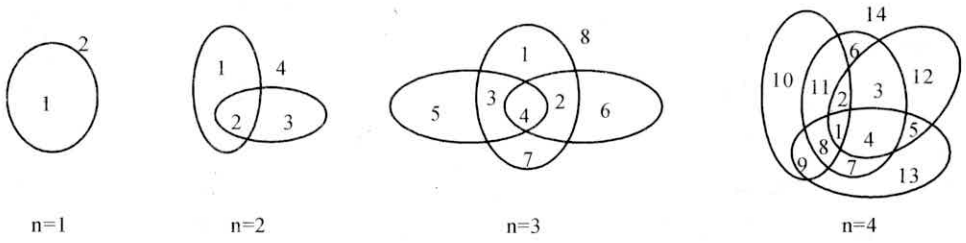


图 3－8

从这些式子中可以看出 $a_n-a_{n-1}=2(n-1)$。当然，上面的式子只是我们通过观察 4 幅图后得出的结论，它的正确性尚不能保证。下面不妨让我们来试着证明一下。当平面上已有 n－1条曲线将平面分割成 a_{n-1}个区域后，第 n－1 条曲线每与曲线相交一次，就会增加一个区域，因为平面上已有了 n－1 条封闭曲线，且第 n 条曲线与已有的每一条闭曲线恰好相交于两点，且不会与任两条曲线交于同一点，故平面上一共增加 2(n－1)个区域，加上已有的 a_{n-1}个区域，一共有 $a_{n-1}+2(n-1)$个区域。所以本题的递推关系是 $a_n=a_{n-1}+2(n-1)$，边界条件是 $a_1=1$。

平面分割问题是竞赛中经常触及到的一类问题，由于其灵活多变，常常感到棘手，下面的【例 3.7】是另一种平面分割问题，有兴趣的读者不妨自己先试着求一下其中的递推关系。

例 3.7　(1998 合肥市竞赛复试第二题)同一平面内的 n(n＜＝500)条直线，已知有 p(p＞＝2)条直线相交于同一点，则这 n 条直线最多能将平面分割成多少个不同的区域？

解：这道题目与第一部分中的平面分割问题十分相似，不同之处就在于线条的曲直以及是否存在共点线条。由于共点直线的特殊性，我们决定先考虑 p 条相交于一点的直线，然后再考虑剩下的 n－p 条直线。首先可以直接求出 p 条相交于一点的直线将平面划分成的区域数为 2p 个，然后在平面上已经有 k(k＞＝p)条直线的基础上，加上一条直线，最多可以与 k 条直线相交，而每次相交都会增加一个区域，与最后一条直线相交后，由于直线可以无限延伸，还会再增加一个区域。所以 $f_m=f_{m-1}+m\ (m>p)$，边界条件在前面已经计算过了，是 $f_p=2p$。虽然这题看上去有两个参数，但是在实际做题中会发现，本题还是属于带一个参数的递推关系。

4. Catalan 数

Catalan 数首先是由 Euler 在精确计算对凸 n 边形的不同的对角三角形剖分的个数问题时得到的，它经常出现在组合计数问题中。

问题的提出：在一个凸 n 边形中，通过不相交于 n 边形内部的对角线，把 n 边形拆分成若干三角形，不同的拆分数目用 h_n 表示，h_n 即为 Catalan 数。例如五边形有如下五种拆分方案(图 3－9)，故 $h_5=5$。求对于一个任意的凸 n 边形相应的 h_n。

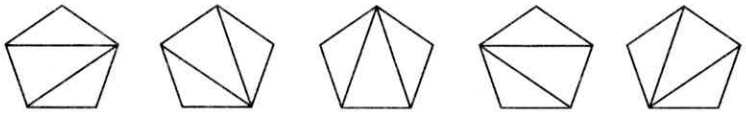


图 3－9

解：设 C_n 表示凸 n 边形的拆分方案总数。由题目中的要求可知一个凸 n 边形的任意一条边都必然是一个三角形的一条边，边 P_1P_n 也不例外，再根据“不在同一直线上的三点可以确定一个三角形”，只要在 $P_2,P_3,\cdots,P_{n-1}$点中找一个点 $P_k(1<k<n)$，与 P_1、P_n共同构成一个三角形的三个顶点，就将 n 边形分成了三个不相交的部分(如图

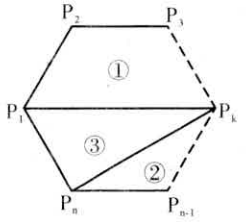


图 3－10

3－10 所示),我们分别称之为区域①、区域②、区域③,其中区域③必定是一个三角形,区域①是一个凸 k 边形,区域②是一个凸 n－k＋1 边形,区域①的拆分方案总数是 C_k,区域②的拆分方案数为 C_{n-k+1},故包含△$P_1P_kP_n$ 的 n 边形的拆分方案数为 C_kC_{n-k+1} 种,而 P_k 可以是 $P_2,P_3,\cdots,P_{n-1}$ 中任一点,根据加法原理,凸 n 边形的三角拆分方案总数为 $\sum_{i=2}^{n-1}C_iC_{n-i+1}$,同时考虑到计算的方便,约定边界条件 $C_2=1$。

Catalan 数是比较复杂的递推关系,尤其在竞赛的时候,选手很难在较短的时间里建立起正确的递推关系。当然,Catalan 数类的问题也可以用搜索的方法来完成,但是,搜索的方法与利用递推关系的方法比较起来,不仅效率低,编程复杂度也陡然提高。

5. 第二类 Stirling 数

在五类典型的递推关系中,第二类 Stirling 是最不为大家所熟悉的。也正因为如此,我们有必要先解释一下什么是第二类 Strling 数。

【定义】 n 个有区别的球放到 m 个相同的盒子中,要求无一空盒,其不同的方案数用 S(n,m)表示,称为第二类 Stirling 数。

下面就让我们根据定义来推导带两个参数的递推关系——第二类 Stirling 数。

解:设有 n 个不同的球,分别用 $b_1,b_2,\cdots,b_n$ 表示。从中取出一个球 b_n,b_n 的放法有以下两种:

①b_n 独自占一个盒子;那么剩下的球只能放在 m－1 个盒子中,方案数为 $S_2(n-1,m-1)$;

②b_n 与别的球共占一个盒子;那么可以事先将 $b_1,b_2,\cdots,b_{n-1}$ 这 n－1 个球放入 m 个盒子中,然后再将球 b_n 放入其中一个盒子中,方案数为 $mS_2(n-1,m)$。

综合以上两种情况,可以得出第二类 Stirling 数定理:

$S_2(n,m)=mS_2(n-1,m)+S_2(n-1,m-1)\quad (n>1,m>=1)$

边界条件可以由定义推导出:

$S_2(n,0)=0;S_2(n,1)=1;S_2(n,n)=1;S_2(n,k)=0(k>n)$。

第二类 Stirling 数在竞赛中较少出现,但在竞赛中也有一些题目与其类似,甚至更为复杂。读者不妨自己来试着建立其中的递推关系。

小结:通过上面对五种典型的递推关系建立过程的探讨,可知对待递推类的题目,要具体情况具体分析,通过找到某状态与其前面状态的联系,建立相应的递推关系。

【上机练习】

1. 菲波那契数列(2)【2.4 基本算法之递归变递推 1760】

菲波那契数列是指这样的数列：数列的第一个和第二个数都为 1，接下来每个数都等于前面 2 个数之和。

给出一个正整数 a，要求菲波那契数列中第 a 个数对 1000 取模的结果是多少。

输入：

第 1 行是测试数据的组数 n，后面跟着 n 行输入。每组测试数据占 1 行，包括一个正整数 a(1 <=a <=1000000)。

输出：

n 行，每行输出对应一个输入。输出应是一个正整数，为菲波那契数列中第 a 个数对 1000 取模得到的结果。

样例输入：

4

5

2

19

1

样例输出：

5

1

181

1

2. Pell 数列【2.4 基本算法之递归变递推 1788】

Pell 数列 a1，a2，a3，…的定义是这样的，a1=1，a2=2，…，$a_n=2 * a_{n-1}+a_{n-2}(n>2)$。

给出一个正整数 k，要求 Pell 数列的第 k 项模上 32767 是多少。

输入：

第 1 行是测试数据的组数 n，后面跟着 n 行输入。每组测试数据占 1 行，包括一个正整数 k (1≤k<1000000)。

输出：

n 行，每行输出对应一个输入。输出应是一个非负整数。

样例输入：

2

1

8

样例输出：

1

408

3. 上台阶【2.4 基本算法之递归变递推 3525】

楼梯有 n(100>n>0)阶台阶,上楼时可以一步上 1 阶,也可以一步上 2 阶,也可以一步上 3 阶,编程计算共有多少种不同的走法。

输入:

输入的每一行包括一组测试数据,即为台阶数 n。最后一行为 0,表示测试结束。

输出:

每一行输出对应一行输入的结果,即为走法的数目。

样例输入:

```
1
2
3
4
0
```

样例输出:

```
1
2
4
7
```

4. 流感传染【2.4 基本算法之递归变递推 6262】

有一批易感人群住在网格状的宿舍区内,宿舍区为 n*n 的矩阵,每个格点为一个房间,房间里可能住人,也可能空着。在第一天,有些房间里的人得了流感,以后每天,得流感的人会使其邻居传染上流感(已经得病的不变),空房间不会传染。请输出第 m 天得流感的人数。

输入:

第一行一个数字 n,n 不超过 100,表示有 n*n 的宿舍房间。

接下来的 n 行,每行 n 个字符,'.'表示第一天该房间住着健康的人,'#'表示该房间空着,'@'表示第一天该房间住着得流感的人。

接下来的一行是一个整数 m,m 不超过 100。

输出:

输出第 m 天,得流感的人数。

样例输入:

```
5
....#
.#.@.
.#@..
#....
.....
4
```

样例输出:

```
16
```

5. 放苹果【2.4 基本算法之递归变递推 666】

把 M 个同样的苹果放在 N 个同样的盘子里，允许有的盘子空着不放，问共有多少种不同的分法？（用 K 表示）5，1，1 和 1，5，1 是同一种分法。

输入：

第一行是测试数据的数目 t（0<=t<=20）。以下每行均包含二个整数 M 和 N，以空格分开。1<=M，N<=10。

输出：

对输入的每组数据 M 和 N，用一行输出相应的 K。

样例输入：

1

7 3

样例输出：

8

6. 吃糖果【2.6 基本算法之动态规划 1944】

名名的妈妈从外地出差回来，带了一盒好吃又精美的巧克力给名名（盒内共有 N 块巧克力，0<N<20）。妈妈告诉名名每天可以吃一块或者两块巧克力。假设名名每天都吃巧克力，问名名共有多少种不同的吃完巧克力的方案。例如：如果 N=1，则名名第 1 天就吃掉它，共有 1 种方案；如果 N=2，则名名可以第 1 天吃 1 块，第 2 天吃 1 块，也可以第 1 天吃 2 块，共有 2 种方案；如果 N=3，则名名第 1 天可以吃 1 块，剩 2 块，也可以第 1 天吃 2 块剩 1 块，所以名名共有 2+1=3 种方案；如果 N=4，则名名可以第 1 天吃 1 块，剩 3 块，也可以第 1 天吃 2 块，剩 2 块，共有 3+2=5 种方案。现在给定 N，请你写程序求出名名吃巧克力的方案数目。

输入：

输入只有 1 行，即整数 N。

输出：

输出只有 1 行，即名名吃巧克力的方案数。

样例输入：

4

样例输出：

5

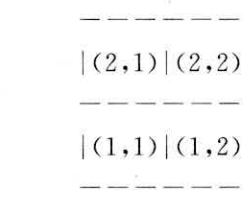
7. 移动路线【2.6 基本算法之动态规划 2718】

X 桌子上有一个 m 行 n 列的方格矩阵，将每个方格用坐标表示，行坐标从下到上依次递增，列坐标从左至右依次递增，左下角方格的坐标为（1，1），则右上角方格的坐标为（m，n）。

小明是个调皮的孩子，一天他捉来一只蚂蚁，不小心把蚂蚁的右脚弄伤了，于是蚂蚁只能向上或向右移动。小明把这只蚂蚁放在左下角的方格中，蚂蚁从

左下角的方格中移动到右上角的方格中，每步移动一个方格。蚂蚁始终在方格矩阵内移动，请计算出不同的移动路线的数目。

对于 1 行 1 列的方格矩阵，蚂蚁原地移动，移动路线数为 1；对于 1 行 2 列（或 2 行 1 列）的方格矩阵，蚂蚁只需一次向右（或向上）移动，移动路线数也为 1……对于一个 2 行 3 列的方格矩阵，如下图所示：



蚂蚁共有3种移动路线：

路线1：(1,1) → (1,2) → (1,3) → (2,3)

路线2：(1,1) → (1,2) → (2,2) → (2,3)

路线3：(1,1) → (2,1) → (2,2) → (2,3)

输入：

输入只有一行，包括两个整数m和n(0＜m＋n＜＝20)，代表方格矩阵的行数和列数，m、n之间用空格隔开。

输出：

输出只有一行，为不同的移动路线的数目。

样例输入：

2 3

样例输出：

3

8. 判断整除【2.6 基本算法之动态规划 3531】

一个给定的正整数序列，在每个数之前都插入＋号或－号后计算它们的和。比如序列：1、2、4共有8种可能的序列：

(＋1)＋(＋2)＋(＋4)＝7

(＋1)＋(＋2)＋(－4)＝－1

(＋1)＋(－2)＋(＋4)＝3

(＋1)＋(－2)＋(－4)＝－5

(－1)＋(＋2)＋(＋4)＝5

(－1)＋(＋2)＋(－4)＝－3

(－1)＋(－2)＋(＋4)＝1

(－1)＋(－2)＋(－4)＝－7

所有结果中至少有一个可被整数k整除，我们则称此正整数序列可被k整除。例如上述序列可以被3、5、7整除，而不能被2、4、6、8……整除。注意：0、－3、－6、－9……都可以认为是3的倍数。

输入：

输入的第一行包含两个数：N(2＜N＜10000)和k(2＜k＜100)，其中N代表一共有N个数，k代表被除数。第二行给出序列中的N个整数，这些整数的取值范围都0到10000之间(可能重复)。

输出：

如果此正整数序列可被k整除，则输出YES，否则输出NO。(注意：都是大写字母)

样例输入：

3 2

1 2 4

样例输出：

NO

9. 踩方格【2.6 基本算法之动态规划 4982】

有一个方格矩阵，矩阵边界在无穷远处。我们做如下假设：

a. 每走一步时，只能从当前方格移动一格，走到某个相邻的方格上；

b. 走过的格子立即塌陷无法再走第二次；

c. 只能向北、东、西三个方向走；

请问：如果允许在方格矩阵上走 n 步，共有多少种不同的方案。2 种走法只要有一步不一样，即被认为是不同的方案。

输入：

允许在方格上行走的步数 n(n<=20)。

输出：

计算出的方案数量

样例输入：

2

样例输出：

7

10. 山区建小学【2.6 基本算法之动态规划 7624】

政府在某山区修建了一条道路，恰好穿越总共 m 个村庄的每个村庄一次，没有回路或交叉，任意两个村庄只能通过这条路来往。已知任意两个相邻的村庄之间的距离为 di(为正整数)，其中，0<i<m。为了提高山区的文化素质，政府又决定从 m 个村中选择 n 个村建小学(设 0<n<=m<500)。请根据给定的 m、n 以及所有相邻村庄的距离，选择在哪些村庄建小学，才使得所有村到最近小学的距离总和最小，计算最小值。

输入：

第 1 行为 m 和 n，其间用空格间隔

第 2 行为(m−1) 个整数，依次表示从一端到另一端的相邻村庄的距离，整数之间以空格间隔。

例如：

10 3

2 4 6 5 2 4 3 1 3

表示在 10 个村庄建 3 所学校。第 1 个村庄与第 2 个村庄距离为 2，第 2 个村庄与第 3 个村庄距离为 4，第 3 个村庄与第 4 个村庄距离为 6，…，第 9 个村庄到第 10 个村庄的距离为 3。

输出：

各村庄到最近学校的距离之和的最小值。

样例输入：

10 2

3 1 3 1 1 1 1 1 3

样例输出：

18

第四章　递归算法

前面已经介绍了关于递归调用这样一种操作,而递归程序设计是C++语言程序设计中的一种重要的方法,它使许多复杂的问题变得简单,容易解决了。递归特点是:函数调用它自己本身。其中直接调用自己称为直接递归,而将a调用b,b又调用a的递归叫做间接递归。

例4.1　给定n(n>=1),用递归的方法计算1+2+3+4+…+(n-1)+n。

【算法分析】

本题可以用递归方法求解,其原因在于它符合递归的三个条件:

(1)本题是累加问题:当前和=前一次和+当前项,而前一次和的计算方法与其相同,只是数据不同s(n)=s(n-1)+n;

(2)给定n,所以是有限次的递归调用;

(3)结束条件是当n=1时,则s=1。

【参考程序】

```
#include<iostream>
using namespace std;
int fac(int);                              //递归函数
int main()
{
    int t;
    cin>>t;                                //输入t的值
    cout<<" s="<<fac(t)<<endl;             //计算1到t的累加和,输出结果
    return 0;
}
int fac(int n)
{
    if (n==1) return 1;
    return(fac(n-1)+n);                    //调用下一层递归
}
```

运行程序,当t=5时,输出结果:s=15,其递归调用执行过程是:(设t=3)

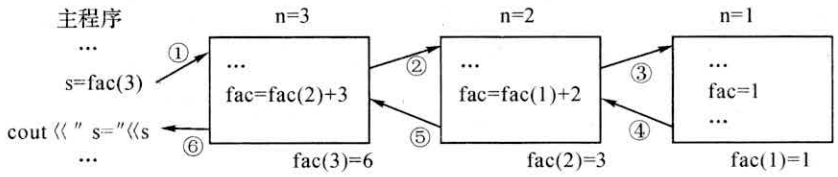


图4-1

递归调用过程，实质上是不断调用函数的过程，由于递归调用一次，所有子程序的变量（局部变量、变参等）、地址在计算机内部都有用特殊的管理方法——栈（先进后出）来管理，一旦递归调用结束，计算机便开始根据栈中存储的地址返回各子程序变量的值，并进行相应操作。

例 4.2　设有 n 个数已经按从大到小的顺序排列，现在输入 x，判断它是否在这 n 个数中，如果存在则输出“YES”，否则输出“NO”。

【算法分析】

该问题属于数据的查找问题，数据查找有多种方法，通常方法是：顺序查找和二分查找，当 N 个数排好序时，用二分查找方法速度大大加快。二分查找算法：

(1)设有 n 个数，存放在 a 数组中，待查找数为 x，用 L 指向数据的高端，用 R 指向数据的低端，mid 指向中间；

(2)若 x=a[mid]，则输出 “YES”；

(3)若 x<a[mid]，则到数据后半段查找：R 不变，L=mid+1，计算新的 mid 值，并进行新的一段查找；

(4)若 x>a[mid]，则到数据前半段查找：L 不变，R=mid−1，计算新的 mid 值，并进行新的一段查找；

(5)若 L>R 都没有查找到，则输出“NO”。

该算法符合递归程序设计的基本规律，可以用递归方法设计。

【参考程序】

```
#include<iostream>
#include<cstdlib>
using namespace std;
int a[11];
void search(int,int,int);
int main()                              //主程序
{
  int k,x,L=1,R=10;
  cout<<"输入 10 个从大到小顺序的数:"<<endl;
  for (k=1;k<=10;k++)
    cin>>a[k];
  cin>>x;
  search(x,L,R);
  system(" pause ");
  return 0;
}
void search(int x,int top,int bot)   //二分查找递归过程
{
  int mid;
  if (top<=bot)
  {
```

```
        mid=(top+bot)/2;                                    //求中间数的位置
        if (x==a[mid]) cout<<" YES "<<endl;  //找到就输出
        else
          if (x<a[mid]) search(x,mid+1,bot);  //判断在前半段还是后半段查找
          else search(x,top,mid-1);
    }
    else cout<<" NO "<<endl;
}
```

例 4.3 Hanoi 汉诺塔问题

有 n 个圆盘,依半径大小(半径都不同),自下而上套在 a 柱上,每次只允许移动最上面一个盘子到另外的柱子上去(除 a 柱外,还有 b 柱和 c 柱,开始时这两个柱子上无盘子),但绝不允许发生柱子上出现大盘子在上,小盘子在下的情况,现要求设计将 a 柱子上 n 个盘子搬移到 c 柱去的方法。

【算法分析】

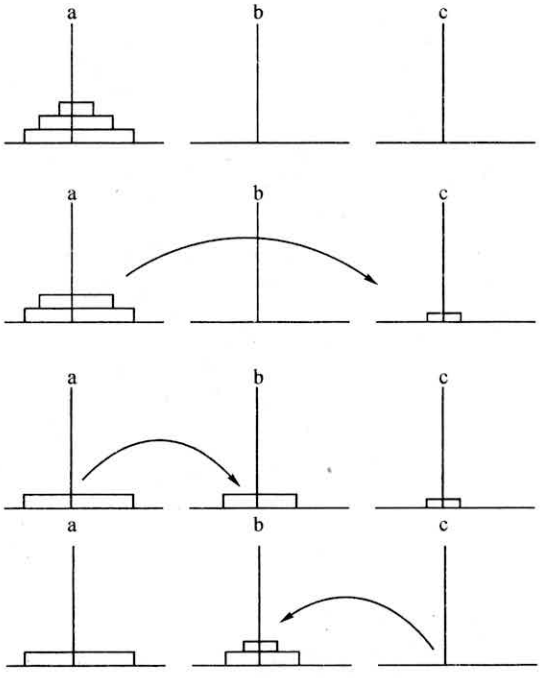
本题是典型的递归程序设计题。

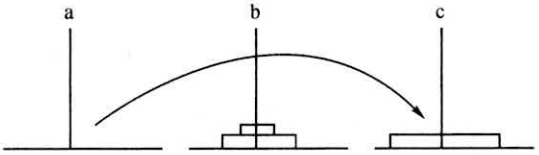
(1)当 n=1 时,只有一个盘子,只需要移动一次:a ⟶c;

(2)当 n=2 时,则需要移动三次:

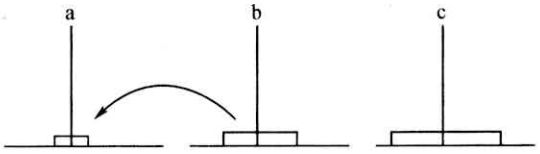
a——1 ⟶b,a——2 ⟶c,b——1 ⟶c。

(3)若 n=3,则具体移动步骤为:

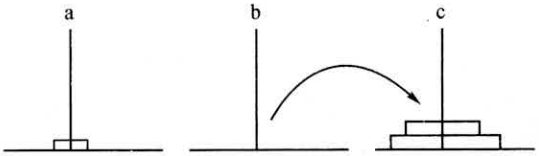




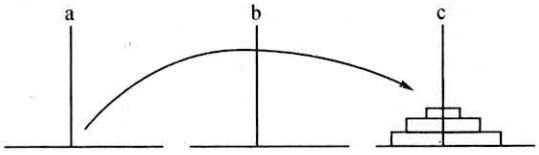
④from a to c



⑤from b to a

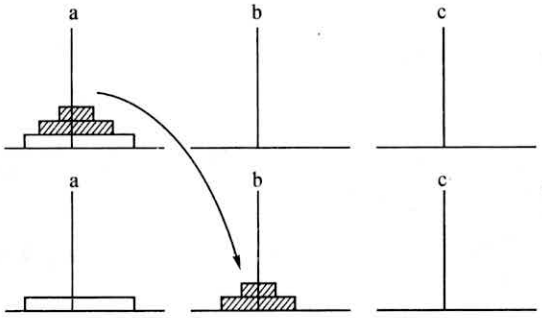


⑥from b to c



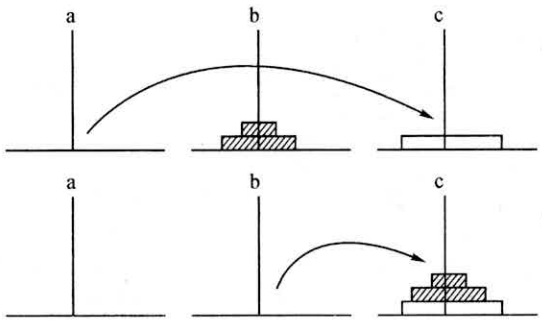
⑦from a to c

假设把第 3 步、第 4 步、第 7 步抽出来就相当于 n＝2 的情况(把上面 2 片捆在一起，视为一片)：



原问题：欲将a柱上的n片移到c柱上，b柱为过渡柱，记为（a，c，b）

①将a柱上的（n-1）片移到b柱上，{a为源柱，b为目标柱，c为过渡柱，记为（a，b，c）}



②将a柱上剩下的1片直接移到c柱上；{不需调用过程}

③将b柱上的（n-1）片移到c柱上，结束。{b为源柱，c为目标柱，a为过渡柱，记为（b，c，a）}

所以可按“n＝2”的移动步骤设计：

①若 n＝0，则退出，即结束程序；否则继续往下执行；

②用c柱作为协助过渡,将a柱上的(n-1)片移到b柱上,调用函数mov(n-1,a,b,c);

③将a柱上剩下的一片直接移到c柱上;

④用a柱作为协助过渡,将b柱上的(n-1)片移到c柱上,调用函数mov(n-1,b,c,a)。

【参考程序】

```
#include<iostream>
using namespace std;
int k=0,n;
void mov(int n,char a,char c,char b)//用b柱作为协助过渡,将a柱上的n片移到c柱上
{
  if (n==0) return; //如果n=0,则退出,即结束程序
  mov(n-1,a,b,c); //用c柱作为协助过渡,将a柱上的(n-1)片移到b柱上
  k++;
  cout <<k<<" :from "<<a <<"-->"<<c<<endl;
  mov(n-1,b,c,a); //用a柱作为协助过渡,将b柱上的(n-1)片移到c柱上
}
int  main()
{
  cout<<" n=";
  cin>>n;
  mov(n,'a','c','b');
  return 0;
}
```

程序定义了把n片从a柱移到c柱的函数mov(n,a,c,b),这个函数把移动分为以下三步来进行:

①先调用函数mov(n-1,a,b,c),把(n-1)片从a柱移到b柱,c柱作为过渡柱;

②直接执行cout <<k<<" :from "<<a <<"-->"<<c<<endl,把a柱上剩下的一片直接移到c柱上;

③调用mov(n-1,b,c,a),把b柱上的(n-1)片从b移到c柱上,a柱是过渡柱。

对于b柱上的(n-1)片如何移到c柱,仍然调用上述的三步。只是把(n-1)当成了n,每调用一次,要移到目标柱上的片数n就减少了一片,直至减少到n=0时就退出,不再调用。

mov函数中出现了自己调用自己的情况,在C++中称为递归调用,这是C++语言的一个特色。对于没有递归调用功能的程序设计语言,则需要将递归函数重新设计为非递归函数的程序。

例4.4 用递归的方法求斐波那契数列中的第n个数。

$$f_n\begin{cases}0 & n=0\\ 1 & n=1\\ f_{n-1}+f_{n-2} & n>1\end{cases}$$

【参考程序】

```
#include<iostream>
using namespace std;
int a[11];
int fib(int);
int main()
{
    int m;
    cin>>m;
    cout<<" fib("<<m<<")="<<fib(m);
    return 0;
}
int fib(int n)
{
    if (n==0) return 0;                    //满足边界条件,递归返回
    if (n==1) return 1;                    //满足边界条件,递归返回
    return (fib(n-1)+fib(n-2));            //递归公式,进一步递归
}
```

输入 15

输出 fib(15)=610

例 4.5　集合的划分

【问题描述】

设 S 是一个具有 n 个元素的集合，S={a1,a2,…,an}，现将 S 划分成 k 个满足下列条件的子集合 S1,S2,…,Sk ，且满足：

1. Si ≠ ∅
2. Si ∩ Sj = ∅　　(1<=i,j<=k　i≠j)
3. S1 ∪ S2 ∪ S3 ∪ … ∪ Sk = S

则称 S1,S2,…,Sk 是集合 S 的一个划分。它相当于把 S 集合中的 n 个元素 a1 ,a2,…,an 放入 k 个(0<k<=n<30)无标号的盒子中，使得没有一个盒子为空。请你确定 n 个元素 a1 ,a2 ,…,an 放入 k 个无标号盒子中去的划分数 S(n,k)。

【输入样例】　　【输出样例】

10 6　　22827

【算法分析】

先举个例子，设 S={1,2,3,4}，k=3，不难得出 S 有 6 种不同的划分方案，即划分数 S(4,3)=6，具体方案为：

{1,2}∪{3}∪{4}　　{1,3}∪{2}∪{4}　　{1,4}∪{2}∪{3}

{2,3}∪{1}∪{4}　　{2,4}∪{1}∪{3}　　{3,4}∪{1}∪{2}

考虑一般情况，对于任意的含有 n 个元素 a1 ,a2,…,an 的集合 S，放入 k 个无标号的盒子中去，划分数为 S(n,k)，我们很难凭直觉和经验计算划分数和枚举划分的所有方案，必须归纳出问题的本质。其实对于任一个元素 an，则必然出现以下两种情况：

1.{an}是k个子集中的一个,于是我们只要把a1,a2,…,an-1划分为k-1子集,便解决了本题,这种情况下的划分数共有S(n-1,k-1)个;

2.{an}不是k个子集中的一个,则an必与其他的元素构成一个子集。则问题相当于先把a1,a2,…,an-1划分成k个子集,这种情况下划分数共有S(n-1,k)个;然后再把元素an加入到k个子集中的任一个中去,共有k种加入方式,这样对于an的每一种加入方式,都可以使集合划分为k个子集,因此根据乘法原理,划分数共有k * S(n-1,k)个。

综合上述两种情况,应用加法原理,得出n个元素的集合{a1,a2,…,an}划分为k个子集的划分数为以下递归公式:S(n,k)=S(n-1,k-1)+k * S(n-1,k) (n>k,k>0)。

下面,我们来确定S(n,k)的边界条件,首先不能把n个元素不放进任何一个集合中去,即k=0时,S(n,k)=0;也不可能在不允许空盒的情况下把n个元素放进多于n的k个集合中去,即k>n时,S(n,k)=0;再者,把n个元素放进一个集合或把n个元素放进n个集合,方案数显然都是1,即k=1或k=n时,S(n,k)=1。

因此,我们可以得出划分数S(n,k)的递归关系式为:

S(n,k)=S(n-1,k-1)+k * S(n-1,k)　(n>k,k>0)

S(n,k)=0　(n<k)或(k=0)

S(n,k)=1　(k=1)或(k=n)

【参考程序】

```
#include<iostream>
using namespace std;
int s(int n, int k)                          //数据还有可能越界,请用高精度计算
{
    if ((n < k) || (k==0)) return 0;         //满足边界条件,退出
    if ((k==1) || (k==n)) return 1;
    return s(n-1,k-1)+k * s(n-1,k);//调用下一层递归
}
int main()
{
    int n,k;
    cin >> n >> k;
    cout << s(n,k);
    return 0;
}
```

例4.6 数的计数(Noip2001)

【问题描述】

我们要求找出具有下列性质数的个数(包括输入的自然数n)。先输入一个自然数n(n<=1000),然后对此自然数按照如下方法进行处理:

不作任何处理;

在它的左边加上一个自然数,但该自然数不能超过原数的一半;

加上数后,继续按此规则进行处理,直到不能再加自然数为止。

【输入格式】

自然数 n(n<=1000)

【输出格式】

满足条件的数

【输入样例】　　【输出样例】

6　　满足条件的数为　　6　(此部分不必输出)　　6

16

26

126

36

136

【方法一】

用递归,f(n)=1+f(1)+f(2)+…+f(n/2),当 n 较大时会超时,时间应该为指数级。

【参考程序】

```
#include<iostream>
using namespace std;
int ans;
void dfs(int m)                    //统计 m 所扩展出的数据个数
{
    int i;
    ans++;                         //每出现一个原数,累加器加 1;
    for (i=1; i<=m/2; i++)//左边添加不超过原数一半的自然数,作为新原数
      dfs(i);
}
int main()
{
    int n;
    cin >> n;
    dfs(n);
    cout << ans;
    return 0;
}
```

【方法二】 用记忆化搜索,实际上是对方法一的改进。设 h[i]表示自然数 i 满足题意三个条件的数的个数。如果用递归求解,会重复来求一些子问题。例如在求 h[4]时,需要再求 h[1]和 h[2]的值。现在我们用 h 数组记录在记忆求解过程中得出的所有子问题的解,当遇到重叠子问题时,直接使用前面记忆的结果。

【参考程序】

```
#include<iostream>
using namespace std;
int h[1001];
void dfs(int m)
```

```
{
    int i;
    if (h[m] !=-1) return;
                    //说明前面已经求得 h[m]的值,直接引用即可,不需要再递归
    h[m]=1;         //将 h[m]置为 1,表示 m 本身为一种情况
    for (i=1; i <=m/2; i++)
    {
      dfs(i);
      h[m]+=h[i];
    }
}
int main()
{
    int n;
    cin >> n;
    for (int i=1; i <=n; i++)
      h[i]=-1;                  //h 数组初始化为-1
    dfs(n);                     //由顶到下记忆化递归求解
    cout << h[n];
    return 0;
}
```

【方法三】 用递推,用 h(n)表示自然数 n 所能扩展的数据个数,则 h(1)=1, h(2)=2, h(3)=2, h(4)=4, h(5)=4, h(6)=6, h(7)=6, h(8)=10, h(9)=10。分析以上数据,可得递推公式:h(i)=1+h(1)+h(2)+…+h(i/2)。此算法的时间度为 O(n * n)。

设 h[i]—i 按照规则扩展出的自然数个数(1<=i<=n)。下表列出了 h[i]值及其方案:

i	h[i]	自然数序列
1	1	1
2	2	2 12
3	2	3 13
4	4	4 14 24 124
5	4	5 15 25 125
…	…	…
i	$1+\sum_{k=1}^{[\frac{i}{2}]}h[k]$	1 1i 2i 12i …

由于 1 为最小自然数,因此 1 无法扩展出其他自然数。自然数 i(2<=i<=n)按照规则扩展出的自然数包括自然数 i;i 左边加上 1;i 左边加上 2 按规则扩展出的 h[2]个自然数…;

由于i左邻的自然数不超过$[\frac{i}{2}]$，因此直至i左边加上h$[[\frac{i}{2}]]$个自然数(这些自然数由$[\frac{i}{2}]$按规则扩展出)为止。由此得出递推的计数公式：

$h[1]=1$

$h[i]=1+\sum_{k=1}^{[\frac{i}{2}]}h[k]$　　$(2<=i<=n)$

从1出发，按照上述公式递推至自然数n，便可得出n按规则扩展出的自然数个数h[n]。

【参考程序】

```
#include<iostream>
using namespace std;
int h[10001];
int main()
{
    int n;
    cin >> n;
    for (int i=1; i <=n; i++) //按照递增顺序计算扩展出的自然数的个数
    {
      h[i]=1;                    //扩展出的自然数包括i本身
      for (int j=1; j <=i/2; j++)
                        //i左边分别加上1…自然数[i/2]，按规则扩展出的自然数
        h[i]+=h[j];
    }
    cout << h[n];
    return 0;
}
```

【方法四】

是对方法三的改进，我们定义数组s，s(x)=h(1)+h(2)+…+h(x)，h(x)=s(x)-s(x-1)，此算法的时间复杂度可降到O(n)。

【参考程序】

```
#include<iostream>
using namespace std;
int h[1001],s[1001];
int main()
{
    int n;
    cin >> n;
    for (int i=1; i <=n; i++)
    {
      h[i]=1+s[i/2];
```

```
        s[i]=s[i-1]+h[i];          //s是h的前缀累加和
    }
    cout << h[n];
    return 0;
}
```

【方法五】

还是用递推,只要作仔细分析,其实我们还可以得到以下的递推公式:(1)当i为奇数时,h(i)=h(i-1);(2)当i为偶数时,h(i)=h(i-1)+h(i/2).

【参考程序】

```
#include<iostream>
using namespace std;
int h[1001];
int main()
{
    int n;
    cin >> n;
    h[1]=1;
    for (int i=2; i <=n; i++)
    {
      h[i]=h[i-1];
      if (i % 2==0) h[i]=h[i-1]+h[i/2];
    }
    cout << h[n];
    return 0;
}
```

【上机练习】

1. 逆波兰表达式【2.2 基本算法之递归 1696】

逆波兰表达式是一种把运算符前置的算术表达式，例如普通的表达式 2+3 的逆波兰表示法为+2 3。逆波兰表达式的优点是运算符之间不必有优先级关系，也不必用括号改变运算次序，例如(2+3) * 4 的逆波兰表示法为 * +2 3 4。本题求解逆波兰表达式的值，其中运算符包括+ - * /四个。

输入：

输入为一行，其中运算符和运算数之间都用空格分隔，运算数是浮点数。

输出：

输出为一行，表达式的值。

可直接用 printf("%f\n", v)输出表达式的值 v。

样例输入：

* +11.0 12.0+24.0 35.0

样例输出：

1357.000000

2. 全排列【2.2 基本算法之递归 1750】

给定一个由不同的小写字母组成的字符串，输出这个字符串的所有全排列。

我们假设对于小写字母有'a'<'b'<…<'y'<'z'，而且给定的字符串中的字母已经按照从小到大的顺序排列。

输入：

输出只有一行，是一个由不同的小写字母组成的字符串，已知字符串的长度在 1 到 6 之间。

输出：

输出这个字符串的所有排列方式，每行一个排列。要求字母序比较小的排列在前面。字母序如下定义：

已知 S=s1s2…sk，T=t1t2…tk，则 S<T 等价于，存在 p(1<=p<=k)，使得 s1=t1，s2=t2，…，$s_{p-1}=t_{p-1}$，sp<tp 成立。

样例输入：

abc

样例输出：

abc

acb

bac

bca

cab

cba

3. 分解因数【2.2 基本算法之递归 1751】

给出一个正整数 a，要求分解成若干个正整数的乘积，即 a=a1 * a2 * a3 * … * an，

并且 1<=a1<=a2<=a3<=…<=an,问这样的分解的种数有多少。注意到 a=a 也是一种分解。

输入:

第 1 行是测试数据的组数 n,后面跟着 n 行输入。每组测试数据占 1 行,包括一个正整数 a(1<a<32768)

输出:

n 行,每行输出对应一个输入。输出应是一个正整数,指明满足要求的分解的种数。

样例输入:

2

2

20

样例输出:

1

4

4. 菲波那契数列【2.2 基本算法之递归 1755】

菲波那契数列是指这样的数列:数列的第一个和第二个数都为 1,接下来每个数都等于前面 2 个数之和。

给出一个正整数 a,要求菲波那契数列中第 a 个数是多少。

输入:

第 1 行是测试数据的组数 n,后面跟着 n 行输入。每组测试数据占 1 行,包括一个正整数 a(1<=a<=20)

输出:

输出有 n 行,每行输出对应一个输入。输出应是一个正整数,为菲波那契数列中第 a 个数的大小。

样例输入:

4

5

2

19

1

样例输出:

5

1

4181

1

5. Pell 数列【2.2 基本算法之递归 1788】

Pell 数列 a1,a2,a3,…的定义是这样的,a1=1,a2=2,…,$a_n=2*a_{n-1}+a_{n-2}$(n>2)。

给出一个正整数 k,要求 Pell 数列的第 k 项模上 32767 是多少。

输入:

第 1 行是测试数据的组数 n，后面跟着 n 行输入。每组测试数据占 1 行，包括一个正整数 k（1≤k<1000000）。

输出：

n 行，每行输出对应一个输入。输出应是一个非负整数。

样例输入：

2

1

8

样例输出：

1

408

6. 扩号匹配问题【2.2 基本算法之递归 2705】

在某个字符串（长度不超过 100）中有左括号、右括号和大小写字母；规定（与常见的算数式子一样）任何一个左括号都从内到外与在它右边且距离最近的右括号匹配。写一个程序，找到无法匹配的左括号和右括号，输出原来字符串，并在下一行标出不能匹配的括号。不能匹配的左括号用"$"标注，不能匹配的右括号用"?"标注。

输入：

输入包括多组数据，每组数据一行，包含一个字符串，只包含左右括号和大小写字母，字符串长度不超过 100。

注意：cin.getline(str,100)最多只能输入 99 个字符！

输出：

对每组输出数据，输出两行，第一行包含原始输入字符，第二行由"$","?"和空格组成，"$"和"?"表示与之对应的左括号和右括号不能匹配。

样例输入：

((ABCD(x)

)(rttyy())sss)(

样例输出：

((ABCD(x)

$$

)(rttyy())sss)(

?　　?$

7. 爬楼梯【2.2 基本算法之递归 3089】

树老师爬楼梯，他可以每次走 1 级或者 2 级，输入楼梯的级数，求不同的走法数。

例如：楼梯一共有 3 级，他可以每次都走一级，或者第一次走一级，第二次走两级，也可以第一次走两级，第二次走一级，一共 3 种方法。

输入：

输入包含若干行，每行包含一个正整数 N，代表楼梯级数，1<=N<=30。

输出：

不同的走法数，每一行输入对应一行输出。

样例输入：

5

8

10

样例输出：

8

34

89

8. 汉诺塔问题【2.2 基本算法之递归 6261】

约 19 世纪末，在欧州的商店中出售一种智力玩具，在一块铜板上有三根杆，最左边的杆上自上而下、由小到大顺序串着由 64 个圆盘构成的塔。目的是将最左边杆上的盘全部移到中间的杆上，条件是一次只能移动一个盘，且不允许大盘放在小盘的上面。

这是一个著名的问题，几乎所有的教材上都有这个问题。由于条件是一次只能移动一个盘，且不允许大盘放在小盘上面，所以 64 个盘的移动次数是：18，446，744，073，709，551，615

这是一个天文数字，若每一微秒可能计算(并不输出)一次移动，那么也需要几乎一百万年。我们仅能找出问题的解决方法并解决较小 N 值时的汉诺塔，但很难用计算机解决 64 层的汉诺塔。

假定圆盘从小到大编号为 1，2，…

输入：

输入为一个整数后面跟三个单字符字符串。

整数为盘子的数目，后三个字符表示三个杆子的编号。

输出：

输出每一步移动盘子的记录。一次移动一行。

每次移动的记录为例如 a－＞3－＞b 的形式，即把编号为 3 的盘子从 a 杆移至 b 杆。

样例输入：

2 a b c

样例输出：

a－＞1－＞c

a－＞2－＞b

c－＞1－＞b

9. 放苹果【2.2 基本算法之递归 666】

把 M 个同样的苹果放在 N 个同样的盘子里，允许有的盘子空着不放，问共有多少种不同的分法？(用 K 表示)5，1，1 和 1，5，1 是同一种分法。

输入：

第一行是测试数据的数目 t(0＜＝t＜＝20)。以下每行均包含二个整数 M 和 N，以空格分开。1＜＝M，N＜＝10。

输出：

对输入的每组数据 M 和 N，用一行输出相应的 K。

样例输入：

1

7 3

样例输出：

8

10. 求最大公约数问题【2.2 基本算法之递归 7592】

给定两个正整数，求它们的最大公约数。

输入：

输入一行，包含两个正整数(<1,000,000,000)。

输出：

输出一个正整数，即这两个正整数的最大公约数。

样例输入：

6 9

样例输出：

3

11. 2 的幂次方表示【2.2 基本算法之递归 8758】

任何一个正整数都可以用 2 的幂次方表示。例如：

$137=2^7+2^3+2^0$

同时约定方次用括号来表示，即 a^b 可表示为 a(b)。由此可知，137 可表示为：

2(7)+2(3)+2(0)

进一步：$7=2^2+2+2^0$（2^1 用 2 表示）

$3=2+2^0$

所以最后 137 可表示为：

2(2(2)+2+2(0))+2(2+2(0))+2(0)

又如：

$1315=2^{10}+2^8+2^5+2+1$

所以 1315 最后可表示为：

2(2(2+2(0))+2)+2(2(2+2(0)))+2(2(2)+2(0))+2+2(0)

输入：

一个正整数 n(n≤20000)。

输出：

一行，符合约定的 n 的 0,2 表示(在表示中不能有空格)。

样例输入：

137

样例输出：

2(2(2)+2+2(0))+2(2+2(0))+2(0)

12. 分数求和【1.13 编程基础之综合应用 12】

输入 n 个分数并对他们求和，并用最简形式表示。所谓最简形式是指：分子分母的最大公约数为 1；若最终结果的分母为 1，则直接用整数表示。

如：5/6、10/3 均是最简形式，而 3/6 需要化简为 1/2，3/1 需要化简为 3。

分子和分母均不为 0，也不为负数。

输入：

第一行是一个整数 n,表示分数个数,1 <=n <=10;

接下来 n 行,每行一个分数,用"p/q"的形式表示,不含空格,p,q 均不超过 10。

输出:

输出只有一行,即最终结果的最简形式。若为分数,用"p/q"的形式表示。

样例输入:

2

1/2

1/3

样例输出:

5/6

13. 因子分解【1.13 编程基础之综合应用 22】

输入一个数,输出其素因子分解表达式。

输入:

输入一个整数 n (2<=n<100)。

输出:

输出该整数的因子分解表达式。

表达式中各个素数从小到大排列。

如果该整数可以分解出因子 a 的 b 次方,当 b 大于 1 时,写做 a^b ;当 b 等于 1 时,则直接写成 a。

样例输入:

60

样例输出:

2^2 * 3 * 5

14. 判断元素是否存在【1.13 编程基础之综合应用 41】

有一个集合 M 是这样生成的:(1) 已知 k 是集合 M 的元素;(2) 如果 y 是 M 的元素,那么,2y+1 和 3y+1 都是 M 的元素;(3) 除了上述二种情况外,没有别的数能够成为 M 的一个元素。

问题:任意给定 k 和 x,请判断 x 是否是 M 的元素。这里的 k 是无符号整数,x 不大于 100000,如果是,则输出 YES,否则,输出 NO

输入:

输入整数 k 和 x, 逗号间隔。

输出:

如果是,则输出 YES,否则,输出 NO

样例输入:

0,22

样例输出:

YES

第五章　搜索与回溯算法

搜索与回溯是计算机解题中常用的算法，很多问题无法根据某种确定的计算法则来求解，可以利用搜索与回溯的技术求解。回溯是搜索算法中的一种控制策略。它的基本思想是：为了求得问题的解，先选择某一种可能情况向前探索，在探索过程中，一旦发现原来的选择是错误的，就退回一步重新选择，继续向前探索，如此反复进行，直至得到解或证明无解。

如迷宫问题：进入迷宫后，先随意选择一个前进方向，一步步向前试探前进，如果碰到死胡同，说明前进方向已无路可走，这时，首先看其他方向是否还有路可走，如果有路可走，则沿该方向再向前试探；如果已无路可走，则返回一步，再看其他方向是否还有路可走；如果有路可走，则沿该方向再向前试探。按此原则不断搜索回溯再搜索，直到找到新的出路或从原路返回入口处无解为止。

递归回溯法算法框架[一]

```
int search(int k)
    {
      for (i=1;i<=算符种数;i++)
      if (满足条件)
      {
        保存结果
        if (到目的地) 输出解;
            else search(k+1);
        恢复:保存结果之前的状态{回溯一步}
      }
    }
```

递归回溯法算法框架[二]

```
int search(int k)
  {
    if  (到目的地) 输出解;
      else
        for (i=1;i<=算符种数;i++)
          if  (满足条件)
            {
              保存结果;
              search(k+1);
              恢复:保存结果之前的状态{回溯一步}
            }
  }
```

例 5.1 素数环:从 1 到 20 这 20 个数摆成一个环,要求相邻的两个数的和是一个素数。

【算法分析】

非常明显,这是一道回溯的题目。从 1 开始,每个空位有 20 种可能,只要填进去的数合法:与前面的数不相同;与左边相邻的数的和是一个素数。第 20 个数还要判断和第 1 个数的和是否素数。

【算法流程】

(1)数据初始化;(2)递归填数:判断第 i 个数填入是否合法。

①如果合法:填数;判断是否到达目标(20 个已填完):是,打印结果;不是,递归填下一个;

②如果不合法:选择下一种可能。

【参考程序】

```
#include<cstdio>
#include<iostream>
#include<cstdlib>
#include<cmath>
using namespace std;
bool b[21]={0};
int total=0,a[21]={0};
int search(int);                    //回溯过程
int print();                        //输出方案
bool pd(int,int);                   //判断素数
int main()
{
   search(1);
   cout<<total<<endl;               //输出总方案数
}
int search(int t)
{
   int i;
   for (i=1;i<=20;i++)              //有 20 个数可选
    if (pd(a[t-1],i)&&(!b[i]))//判断与前一个数是否构成素数及该数是否可用
     {
      a[t]=i;
      b[i]=1;
      if (t==20) { if (pd(a[20],a[1])) print();}
        else search(t+1);
      b[i]=0;
     }
}
int print()
```

```
{
  total++;
  cout<<"<"<<total<<">";
  for (int j=1;j<=20;j++)
    cout<<a[j]<<" ";
  cout<<endl;
}
bool pd(int x,int y)
{
  int k=2,i=x+y;
  while (k<=sqrt(i)&&i%k!=0) k++;
  if (k>sqrt(i)) return 1;
    else return 0;
}
```

例 5.2　设有 n 个整数的集合{1,2,…,n},从中任意取出 r 个数进行排列(r<n),试列出所有的排列。

【参考程序】

```
#include<cstdio>
#include<iostream>
#include<iomanip>
using namespace std;
int num=0,a[10001]={0},n,r;
bool b[10001]={0};
int search(int);                          //回溯过程
int print();                              //输出方案
int main()
{
  cout<<" input n,r:";
  cin>>n>>r;
  search(1);
  cout<<" number="<<num<<endl;            //输出方案总数
}
int search(int k)
{
   int i;
   for (i=1;i<=n;i++)
    if  (!b[i])                           //判断 i 是否可用
      {
       a[k]=i;                            //保存结果
       b[i]=1;
```

```
        if (k==r) print();
          else search(k+1);
        b[i]=0;
      }
}
int print()
{
  num++;
  for (int i=1;i<=r;i++)
    cout<<setw(3)<<a[i];
  cout<<endl;
}
```

例 5.3 任何一个大于 1 的自然数 n,总可以拆分成若干个小于 n 的自然数之和。

当 n=7,共有 14 种拆分方法:

```
7=1+1+1+1+1+1+1
7=1+1+1+1+1+2
7=1+1+1+1+3
7=1+1+1+2+2
7=1+1+1+4
7=1+1+2+3
7=1+1+5
7=1+2+2+2
7=1+2+4
7=1+3+3
7=1+6
7=2+2+3
7=2+5
7=3+4
total=14
```

【参考程序】

```
#include<cstdio>
#include<iostream>
#include<cstdlib>
using namespace std;
int a[10001]={1},n,total;
int search(int,int);
int print(int);
int main()
```

```
{
    cin>>n;
    search(n,1);                          //将要拆分的数 n 传递给 s
    cout<<" total="<<total<<endl;         //输出拆分的方案数
}
int search(int s,int t)
{
    int i;
    for (i=a[t-1];i<=s;i++)
     if (i<n)                             //当前数 i 要大于等于前 1 位数,且不超过 n
      {
       a[t]=i;                            //保存当前拆分的数 i
       s-=i;                              //s 减去数 i, s 的值将继续拆分
       if (s==0) print(t);                //当 s=0 时,拆分结束输出结果
         else search(s,t+1);              //当 s>0 时,继续递归
       s+=i;                              //回溯:加上拆分的数,以便产生所有可能的拆分
      }
}
int print(int t)
{
    cout<<n<<"=";
    for (int i=1;i<=t-1;i++)              //输出一种拆分方案
     cout<<a[i]<<"+";
    cout<<a[t]<<endl;
    total++;                              //方案数累加 1
}
```

例 5.4　八皇后问题:要在国际象棋棋盘中放八个皇后,使任意两个皇后都不能互相吃。(提示:皇后能吃同一行、同一列、同一对角线的任意棋子。)

放置第 i 个(行)皇后的算法为:

```
int search(i);
  {
      int j;
      for (第 i 个皇后的位置 j=1;j<=8;j++)  //在本行的 8 列中去试
        if (本行本列允许放置皇后)
        {
          放置第 i 个皇后;
          对放置皇后的位置进行标记;
          if (i==8) 输出                          //已经放完个皇后
            else search(i+1);                     //放置第 i+1 个皇后
          对放置皇后的位置释放标记,尝试下一个位置是否可行;
```

```
    }
}
```

【算法分析】

显然问题的关键在于如何判定某个皇后所在的行、列、斜线上是否有别的皇后;可以从矩阵的特点上找到规律,如果在同一行,则行号相同;如果在同一列上,则列号相同;如果同在/斜线上,则行列值之和相同;如果同在\斜线上,则行列值之差相同;从图 5-1 可验证:

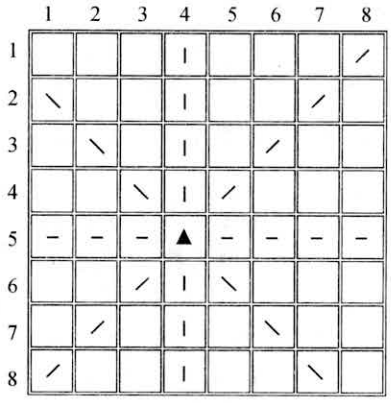


图 5-1

考虑每行有且仅有一个皇后,设一维数组 a[1..8]表示皇后的放置:第 i 行皇后放在第 j 列,用 a[i]=j 来表示,即下标是行数,内容是列数。例如:a[3]=5 就表示第 3 个皇后在第 3 行第 5 列上。

判断皇后是否安全,即检查同一列、同一对角线是否已有皇后,建立标志数组 b[1..8]控制同一列只能有一个皇后,若两皇后在同一对角线上,则其行列坐标之和或行列坐标之差相等,故亦可建立标志数组 c[1..16]、d[-7..7]控制同一对角线上只能有一个皇后。

如果斜线不分方向,则同一斜线上两皇后的行号之差的绝对值与列号之差的绝对值相同。在这种方式下,要表示两个皇后 i 和 j 不在同一列或斜线上的条件可以描述为:(a[i]!=a[j]) && (abs(i-j)!=abs(a[i]-a[j]))\{i 和 j 分别表示两个皇后的行号\}。

【参考程序】

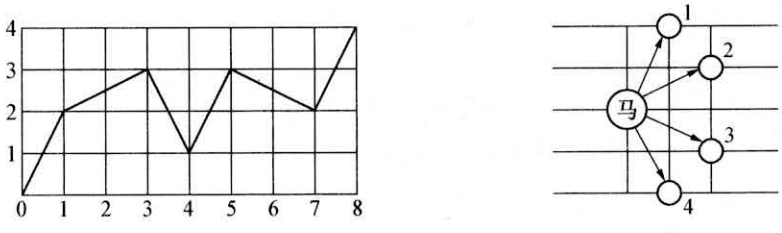
```
#include<cstdio>
#include<iostream>
#include<cstdlib>
#include<iomanip>
using namespace std;
bool d[100]={0},b[100]={0},c[100]={0};
int sum=0,a[100];
int search(int);
int print();
int main()
{
    search(1);                    //从第 1 个皇后开始放置
}
int search(int i)
```

```
{
    int j;
    for (j=1;j<=8;j++)          //每个皇后都有 8 位置(列)可以试放
      if ((!b[j])&&(!c[i+j])&&(!d[i-j+7]))  //寻找放置皇后的位置
                                //由于 C++不能操作负数组,因此考虑加 7
      {                         //放置皇后,建立相应标志值
        a[i]=j;                 //摆放皇后
        b[j]=1;                 //宣布占领第 j 列
        c[i+j]=1;               //占领两个对角线
        d[i-j+7]=1;
        if (i==8) print();      //8 个皇后都放置好,输出
          else search(i+1);     //继续递归放置下一个皇后
        b[j]=0;                 //递归返回即为回溯一步,当前皇后退出
        c[i+j]=0;
        d[i-j+7]=0;
      }
}
int print()
{
    int i;
    sum++;                      //方案数累加 1
    cout<<" sum="<<sum<<endl;
    for (i=1;i<=8;i++)          //输出一种方案
      cout<<setw(4)<<a[i];
    cout<<endl;
}
```

例 5.5　马的遍历

中国象棋半张棋盘如图 5－2(a)所示。马自左下角往右上角跳。今规定只许往右跳，不许往左跳。比如图 5－2(a)中所示为一种跳行路线，并将所经路线打印出来。打印格式为：

0,0－>2,1－>3,3－>1,4－>3,5－>2,7－>4,8…



(a)　　(b)

图 5－2

【算法分析】

如图 5－2(b)，马最多有四个方向，若原来的横坐标为 j、纵坐标为 i，则四个方向的移动

可表示为：

1:(i,j)→(i+2,j+1);　(i<3,j<8)

2:(i,j)→(i+1,j+2);　(i<4,j<7)

3:(i,j)→(i−1,j+2);　(i>0,j<7)

4:(i,j)→(i−2,j+1);　(i>1,j<8)

搜索策略：

S1:a[1]=(0,0);

S2:从 a[1]出发,按移动规则依次选定某个方向,如果达到的是(4,8)则转向 S3,否则继续搜索下一个到达的顶点;

S3:打印路径。

【参考程序】

```
#include<cstdio>
#include<iostream>
#include<cstdlib>
using namespace std;
int a[100][100],t=0;                    //路径总数和路径
int x[4]={2,1,-1,-2},                   //四种移动规则
    y[4]={1,2,2,1};
int search(int);                        //搜索
int print(int);                         //打印
int main()                              //主程序
{
    a[1][1]=0;a[1][2]=0;                //从坐标(0,0)开始往右跳第二步
    search(2);
}
int search(int i)
{
    for (int j=0;j<=3;j++)              //往 4 个方向跳
      if (a[i-1][1]+x[j]>=0&&a[i-1][1]+x[j]<=4
        &&a[i-1][2]+y[j]>=0&&a[i-1][2]+y[j]<=8) //判断马不越界
      {
        a[i][1]=a[i-1][1]+x[j];         //保存当前马的位置
        a[i][2]=a[i-1][2]+y[j];
        if (a[i][1]==4&&a[i][2]==8) print(i);
          else search(i+1);             //搜索下一步
      }
}
int print(int ii)
{
    {
```

```
        t++;
        cout<<t<<":  ";
        for (int i=1;i<=ii-1;i++)
          cout<<a[i][1]<<","<<a[i][2]<<"-->";
        cout<<" 4,8 "<<endl;
      }
  }
```

例 5.6　设有 A、B、C、D、E 五人从事 J1、J2、J3、J4、J5 五项工作，每人只能从事一项，他们的效益如图 5－3 所示。

	J1	J2	J3	J4	J5
A	13	11	10	4	7
B	13	10	10	8	5
C	5	9	7	7	4
D	15	12	10	11	5
E	10	11	8	8	4

图 5－3

每人选择五项工作中的一项，在各种选择的组合中，找到效益最高的一种组合输出。

【算法分析】

1. 用数组 f 储存搜索中工作选择的方案；数组 g 存放最优的工作选择方案；数组 p 用于表示某项工作有没有被选择了。

2. (1)选择 p(i)=0 的第 i 项工作；

(2)判断效益是否高于 max 已记录的效益，若高于则更新 g 数组及 max 的值。

3. 搜索策略：回溯法(深度优先搜索 dfs)。

【参考程序】

```
#include<cstdio>
#include<iostream>
#include<cstdlib>
#include<iomanip>
using namespace std;
int data[6][6]={{0,0,0,0,0,0},{0,13,11,10,4,7},{0,13,10,10,8,5},{0,5,9,7,
7,4},{0,15,12,10,11,5},{0,10,11,8,8,4}};
int max1=0,g[10],f[10];
bool p[6]={0};
int go(int step,int t)                    // step 是第几个人，t 是之前已得的效益
{
    for (int i=1;i<=5;i++)
     if (!p[i])                           //判断第 i 项工作没人选择
      {
        f[step]=i;                        //第 step 个人，就选第 i 项工作
```

```
            p[i]=1;                    //标记第 i 项工作被人安排了
            t+=data[step][i];          //计算效益值
            if (step<5) go(step+1,t);
                else if (t>max1)       //保存最佳效益值
                {
                  max1=t;
                  for (int j=1;j<=5;j++)
                    g[j]=f[j];         //保存最优效益下的工作选择方案
                }
            t-=data[step][i];          //回溯
            p[i]=0;
        }
}
int main()
{
  go(1,0);                             //从第 1 个人,总效益为 0 开始
  for (int i=1;i<=5;i++)
    cout<<char(64+i)<<":J "<<g[i]<<setw(3);  //输出各项工作安排情况
  cout<<endl;
  cout<<" supply:"<<max1<<endl;  //输出最佳效益值
}
```

例 5.7 选书

学校放寒假时,信息学竞赛辅导老师有 A、B、C、D、E 五本书,要分给参加培训的张、王、刘、孙、李五位同学,每人只能选一本书。老师事先让每个人将自己喜欢的书填写在如下的表格中。然后根据他们填写的表来分配书本,希望设计一个程序帮助老师求出所有可能的分配方案,使每个学生都满意。

书本 / 学生	A	B	C	D	E
张同学			Y	Y	
王同学	Y	Y			Y
刘同学		Y	Y		
孙同学				Y	
李同学		Y			Y

【算法分析】

可用穷举法,先不考虑“每人都满意”这一条件,这样只剩“每人选一本且只能选一本”这一条件。在这个条件下,可行解就是五本书的所有全排列,一共有 5!=120 种。然后在 120 种可行解中一一删去不符合“每人都满意”的解,留下的就是本题的解答。

为了编程方便,设 1、2、3、4、5 分别表示这五本书。这五个数的一种全排列就是五本书的一种分发。例如 54321 就表示第 5 本书(即 E)分给张,第 4 本书(即 D)分给王,…,第 1 本

书(即 A)分给李。“喜爱书表”可以用二维数组来表示,1 表示喜爱,0 表示不喜爱。

算法设计:S1:产生 5 个数字的一个全排列;

S2:检查是否符合“喜爱书表”的条件,如果符合就打印出来;

S3:检查是否所有的排列都产生了,如果没有产生完,则返回 S1;

S4:结束。

上述算法有可以改进的地方。比如产生了一个全排列 12345,从表中可以看出,选第一本书即给张同学的书,1 是不可能的,因为张只喜欢第 3、4 本书。这就是说,1×××一类的分法都不符合条件。由此想到,如果选定第一本书后,就立即检查一下是否符合条件,发现 1 是不符合的,后面的四个数字就不必选了,这样就减少了运算量。换句话说,第一个数字只在 3、4 中选择,这样就可以减少 3/5 的运算量。同理,选定了第一个数字后,也不应该把其他 4 个数字一次选定,而是选择了第二个数字后,就立即检查是否符合条件。例如,第一个数选 3,第二个数选 4 后,立即检查,发现不符合条件,就应另选第二个数。这样就把 34×××一类的分法在产生前就删去了,又减少了一部分运算量。

综上所述,改进后的算法应该是:在产生排列时,每增加一个数,就检查该数是否符合条件,不符合,就立刻换一个,符合条件后,再产生下一个数。因为从第 i 本书到第 i+1 本书的寻找过程是相同的,所以可以用回溯算法。算法设计如下:

```
int search(i)
 {
  for (j=1;j<=5;j++)
    {
      if (第 i 个同学分给第 j 本书符合条件)
        {
          记录第 i 个数
          if (i==5) 打印一个解;
                  else search(i+1);
          删去第 i 个数
        }
    }
}
```

【参考程序】

```
#include<cstdio>
#include<iostream>
#include<cstdlib>
using   namespace std;
int   book[6],c;
bool flag[6],like[6][6]={{0,0,0,0,0,0},{0,0,0,1,1,0},{0,1,1,0,0,1},
                         {0,0,1,1,0,0},{0,0,0,0,1,0},{0,0,1,0,0,1}};;
int   search(int);
int   print();
int   main()
```

```
{
  for (int i=1;i<=5;i++) flag[i]=1;
  search(1);                          //从第 1 个开始选书,递归
}
int search(int i)                     //递归函数
{
  for (int j=1;j<=5; j++)             //每个人都有 5 本书可选
  if (flag[j]&&like[i][j])            //满足分书的条件
  {
    flag[j]=0;                        //把被选中的书放入集合 flag 中,避免重复被选
    book[i]=j;                        //保存第 i 个人选中的第 j 本书
    if (i==5) print();                //i=5 时,所有的人都分到书,输出结果
      else search(i+1);               //i<5 时,继续递归分书
    flag[j]=1;                        //回溯:把选中的书放回,产生其他分书的方案
    book[i]=0;
  }
}
int print()
{
  c++;                                //方案数累加 1
  cout <<" answer " <<c <<":\n ";
  for (int i=1;i<=5;i++)
    cout <<i <<": " <<char(64+book[i]) <<endl; //输出分书的方案
}
```

输出结果:

```
answer 1:
1: C
2: A
3: B
4: D
5: E
```

例 5.8 跳马问题。在 5 * 5 格的棋盘上,有一只中国象棋的马,从(1,1)点出发,按日字跳马,它可以朝 8 个方向跳,但不允许出界或跳到已跳过的格子上,要求在跳遍整个棋盘。

输出前 5 个方案及总方案数。

输出格式示例:

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

【参考程序】

```
#include<cstdio>
#include<iostream>
#include<cstdlib>
#include<iomanip>
using namespace std;
int u[8]={1,2,2,1,-1,-2,-2,-1},//8 个方向上的 x,y 增量
    v[8]={-2,-1,1,2,2,1,-1,-2};
int a[100][100]={0},num=0;
                          //记每一步走在棋盘的哪一格和棋盘的每一格有没有被走过
bool b[100][100]={0};
int search(int,int,int);          //以每一格为阶段,在每一阶段中试遍 8 个方向
int print();                      //打印方案
int main()
{
  a[1][1]=1;b[1][1]=1;     //从(1,1)第一步开始走
  search(1,1,2);           //从(1,1)开始搜第 2 步该怎样走
  cout<<num<<endl;         //输出总方案(304)
}
int search(int i,int j,int n)
{
  int k,x,y;                      //这三个变量一定要定义局部变量
  if (n>25) {print();return 0;}  //达到最大规模打印、统计方案
  for (k=0;k<=7;k++)       //试遍 8 个方向
  {
    x=i+u[k];y=j+v[k];  //走此方向,得到的新坐标
    if (x<=5&&x>=1&&y<=5&&y>=1&&(!b[x][y]))
    {                             //如果新坐标在棋盘上,并且这一格可以走
      b[x][y]=1;
      a[x][y]=n;
      search(x,y,n+1);        //从(x,y)去搜下一步该如何走
      b[x][y]=0;
      a[x][y]=0;
    }
  }
}
int print()
{
  num++;                          //统计总方案
  if (num<=5)                     //打印出前 5 种方案
```

```
    {
      for (int k=1;k<=5;k++)   //打印本次方案
      {
        for (int kk=1;kk<=5;kk++)
          cout<<setw(5)<<a[k][kk];
        cout<<endl;
      }
      cout<<endl;
    }
}
```

【上机练习】

1. LETTERS【2.5 基本算法之搜索 156】

给出一个 roe * col 的大写字母矩阵,一开始的位置为左上角,你可以向上下左右四个方向移动,并且不能移向曾经经过的字母。问最多可以经过几个字母。

输入:

第一行,输入字母矩阵行数 R 和列数 S,1<=R,S<=20。

接着输出 R 行 S 列字母矩阵。

输出:

最多能走过的不同字母的个数。

样例输入:

3 6

HFDFFB

AJHGDH

DGAGEH

样例输出:

6

2. 八皇后问题【2.5 基本算法之搜索 1700】

在国际象棋棋盘上放置八个皇后,要求每两个皇后之间不能直接吃掉对方。

输入:

无输入。

输出:

按给定顺序和格式输出所有八皇后问题的解(见 Sample Output)。

样例输入:

无输入。

样例输出:

No. 1

1 0 0 0 0 0 0 0

0 0 0 0 0 0 1 0

0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
No. 2
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 1 0 0 0 0 0
…以下省略

3. 八皇后【2.5 基本算法之搜索 1756】

会下国际象棋的人都很清楚：皇后可以在横、竖、斜线上不限步数地吃掉其他棋子。如何将 8 个皇后放在棋盘上(有 8 * 8 个方格)，使它们谁也不能被吃掉！这就是著名的八皇后问题。

对于某个满足要求的 8 皇后的摆放方法，定义一个皇后串 a 与之对应，即 a=b1b2…b8，其中 bi 为相应摆法中第 i 行皇后所处的列数。已经知道 8 皇后问题一共有 92 组解(即 92 个不同的皇后串)。

给出一个数 b，要求输出第 b 个串。串的比较是这样的：皇后串 x 置于皇后串 y 之前，当且仅当将 x 视为整数时比 y 小。

输入：

第 1 行是测试数据的组数 n，后面跟着 n 行输入。每组测试数据占 1 行，包括一个正整数 b(1<=b<=92)。

输出：

输出有 n 行，每行输出对应一个输入。输出应是一个正整数，是对应于 b 的皇后串。

样例输入：

2
1
92

样例输出：

15863724
84136275

4. 迷宫【2.5 基本算法之搜索 1792】

一天 Extense 在森林里探险的时候不小心走入了一个迷宫，迷宫可以看成是由 n * n 的格点组成，每个格点只有 2 种状态，. 和 #，前者表示可以通行后者表示不能通行。同时当

Extense处在某个格点时,他只能移动到东南西北(或者说上下左右)四个方向之一的相邻格点上,Extense想要从点A走到点B,问在不走出迷宫的情况下能不能办到。如果起点或者终点有一个不能通行(为#),则看成无法办到。

输入:

第1行是测试数据的组数k,后面跟着k组输入。每组测试数据的第1行是一个正整数n(1<=n<=100),表示迷宫的规模是n * n的。接下来是一个n * n的矩阵,矩阵中的元素为.或者#。再接下来一行是4个整数ha, la, hb, lb,描述A处在第ha行, 第la列,B处在第hb行, 第lb列。注意到ha, la, hb, lb全部是从0开始计数的。

输出:

k行,每行输出对应一个输入。能办到则输出"YES",否则输出"NO"。

样例输入:

```
2
3
. # #
. . #
# . .
0 0 2 2
5
. . . . .
# # # . #
. . # . .
# # # . .
. . . # .
0 0 4 0
```

样例输出:

```
YES
NO
```

5. 红与黑【2.5 基本算法之搜索 1818】

有一间长方形的房子,地上铺了红色、黑色两种颜色的正方形瓷砖。你站在其中一块黑色的瓷砖上,只能向相邻的黑色瓷砖移动。请写一个程序,计算你总共能够到达多少块黑色的瓷砖。

输入:

包括多个数据集合。每个数据集合的第一行是两个整数W和H,分别表示x方向和y方向瓷砖的数量。W和H都不超过20。在接下来的H行中,每行包括W个字符。每个字符表示一块瓷砖的颜色,规则如下:

1)'.':黑色的瓷砖;

2)'#':白色的瓷砖;

3)'@':黑色的瓷砖,并且你站在这块瓷砖上。该字符在每个数据集合中唯一出现一次。

当在一行中读入的是两个零时，表示输入结束。

输出：

对每个数据集合，分别输出一行，显示你从初始位置出发能到达的瓷砖数（记数时包括初始位置的瓷砖）。

样例输入：

```
6 9
. . . . #.
. . . . . #
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
#@. . . #
. #. . #.
0 0
```

样例输出：

```
45
```

6. 棋盘问题【2.5 基本算法之搜索 323】

在一个给定形状的棋盘（形状可能是不规则的）上面摆放棋子，棋子没有区别。要求摆放时任意的两个棋子不能放在棋盘中的同一行或者同一列，请编程求解对于给定形状和大小的棋盘，摆放 k 个棋子的所有可行的摆放方案 C。

输入：

输入含有多组测试数据。

每组数据的第一行是两个正整数，n k，用一个空格隔开，表示了将在一个 n * n 的矩阵内描述棋盘，以及摆放棋子的数目。n <=8 , k <=n

当为-1-1 时表示输入结束。

随后的 n 行描述了棋盘的形状：每行有 n 个字符，其中 # 表示棋盘区域，. 表示空白区域（数据保证不出现多余的空白行或者空白列）。

输出：

对于每一组数据，给出一行输出，输出摆放的方案数目 C（数据保证 C<2^31）。

样例输入：

```
2 1
#.
. #
4 4
. . . #
. . #.
. #. .
```

#...
-1 -1

样例输出:

2

1

7. 取石子游戏【2.5 基本算法之搜索 6266】

有两堆石子,两个人轮流去取。每次取的时候,只能从较多的那堆石子里取,并且取的数目必须是较少的那堆石子数目的整数倍,最后谁能够把一堆石子取空谁就算赢。

比如初始的时候两堆石子的数目是 25 和 7。

25 7	→	11 7	→	4 7	→	4 3	→	1 3	→	1 0
	选手 1 取		选手 2 取		选手 1 取		选手 2 取		选手 1 取	

最后选手 1(先取的)获胜,在取的过程中选手 2 都只有唯一的一种取法。

给定初始时石子的数目,如果两个人都采取最优策略,请问先手能否获胜。

输入:

输入包含多数数据。每组数据一行,包含两个正整数 a 和 b,表示初始时石子的数目。

输入以两个 0 表示结束。

输出:

如果先手胜,输出"win",否则输出"lose"

样例输入:

34 12

15 24

0 0

样例输出:

win

lose

提示:

假设石子数目为(a,b)且 a >=b,如果[a/b] >=2 则先手必胜,如果[a/b]<2,那么先手只有唯一的一种取法。[a/b]表示 a 除以 b 取整后的值。

8. 马走日【2.5 基本算法之搜索 8465】

马在中国象棋以日字形规则移动。

请编写一段程序,给定 n * m 大小的棋盘,以及马的初始位置(x,y),要求不能重复经过棋盘上的同一个点,计算马可以有多少途径遍历棋盘上的所有点。

输入:

第一行为整数 T(T < 10),表示测试数据组数。

每一组测试数据包含一行,为四个整数,分别为棋盘的大小以及初始位置坐标 n,m,x,y。(0<=x<=n-1,0<=y<=m-1, m < 10, n < 10)

输出:

每组测试数据包含一行,为一个整数,表示马能遍历棋盘的途径总数,0 为无法遍历

一次。

样例输入：

1

5 4 0 0

样例输出：

32

9. 单词接龙【2.5 基本算法之搜索 8783】

单词接龙是一个与我们经常玩的成语接龙相类似的游戏，现在我们已知一组单词，且给定一个开头的字母，要求出以这个字母开头的最长的“龙”（每个单词都最多在“龙”中出现两次），在两个单词相连时，其重合部分合为一部分，例如 beast 和 astonish，如果接成一条龙则变为 beastonish，另外相邻的两部分不能存在包含关系，例如 at 和 atide 间不能相连。

输入：

输入的第一行为一个单独的整数 n（n<=20）表示单词数，以下 n 行每行有一个单词（只含有大写或小写字母，长度不超过 20），输入的最后一行为一个单个字符，表示“龙”开头的字母。你可以假定以此字母开头的“龙”一定存在。

输出：

只需输出以此字母开头的最长的“龙”的长度。

样例输入：

5

at

touch

cheat

choose

tact

a

样例输出：

23

10. 分成互质组【2.5 基本算法之搜索 7834】

给定 n 个正整数，将它们分组，使得每组中任意两个数互质。至少要分成多少个组？

输入：

第一行是一个正整数 n。1 <=n <=10。

第二行是 n 个不大于 10000 的正整数。

输出：

一个正整数，即最少需要的组数。

样例输入：

6

14 20 33 117 143 175

样例输出：

3

11. 放苹果【2.5 基本算法之搜索 666】

把 M 个同样的苹果放在 N 个同样的盘子里,允许有的盘子空着不放,问共有多少种不同的分法?(用 K 表示)5,1,1 和 1,5,1 是同一种分法。

输入:

第一行是测试数据的数目 t(0 <=t <=20)。以下每行均包含二个整数 M 和 N,以空格分开。1<=M,N<=10。

输出:

对输入的每组数据 M 和 N,用一行输出相应的 K。

样例输入:

1

7 3

样例输出:

8

第六章　贪心算法

一、基本概念

所谓贪心算法是指在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，他所做出的仅是在某种意义上的局部最优解。

贪心算法没有固定的算法框架，算法设计的关键是贪心策略的选择。必须注意的是，贪心算法不是对所有问题都能得到整体最优解，选择的贪心策略必须具备无后效性，即某个状态以后的过程不会影响以前的状态，只与当前状态有关。

所以对所采用的贪心策略一定要仔细分析其是否满足无后效性。

二、基本思路

1. 建立数学模型来描述问题。
2. 把求解的问题分成若干个子问题。
3. 对每一子问题求解，得到子问题的局部最优解。
4. 把子问题的解局部最优解合成原来解问题的一个解。

三、适用问题

贪心策略适用的前提是：局部最优策略能导致产生全局最优解。

四、实现框架

```
从问题的某一初始解出发；
while（能朝给定总目标前进一步）
{
    利用可行的决策，求出可行解的一个解元素；
}
由所有解元素组合成问题的一个可行解；
```

五、贪心策略的选择

因为用贪心算法只能通过解局部最优解的策略来达到全局最优解，因此，一定要注意判

断问题是否适合采用贪心算法策略,找到的解是否一定是问题的最优解。

六、典型例题

例 6.1 排队打水问题

有 n 个人排队到 r 个水龙头去打水,他们装满水桶的时间为 t1,t2,…,tn 为整数且各不相等,应如何安排他们的打水顺序才能使他们花费的时间最少?

【输入样例】

```
4 2                          //4 人打水,2 个水龙头
2 6 4 5                      //每个人的打水时间
```

【输出样例】

```
23                           //总共花费时间
```

【算法分析】

由于排队时,越靠前面的计算次数越多,因此越小的排在越前面得出的结果越小(可以用数学方法简单证明,这里就不再赘述),所以这道题可以用贪心法解答,基本步骤为:

(1)将输入的时间按从小到大排序;

(2)将排序后的时间按顺序依次放入每个水龙头的队列中;

(3)统计,输出答案。

参考程序主要框架如下:

```
cin>>n>>r;
memset(s,0,sizeof(s));          //初始化
j=0; minx=0;
for (i=1;i<=n;i++)              //用贪心法求解
{
  j++;
  if (j==r+1) j=1;              //前 r 个人为一组,第 r+1 个人回到第一个水龙头
  s[j]+=a[i];                   //加上等待时间
  minx+=s[j];
}
cout<<minx;                     //输出解答
```

例 6.2 均分纸牌(Noip2002)

有 n 堆纸牌,编号分别为 1,2,…, n。每堆上有若干张,但纸牌总数必为 n 的倍数。可以在任一堆上取若干张纸牌,然后移动。

移牌规则为:在编号为 1 的堆上取的纸牌,只能移到编号为 2 的堆上;在编号为 n 的堆上取的纸牌,只能移到编号为 n-1 的堆上;其他堆上取的纸牌,可以移到相邻左边或右边的堆上。

现在要求找出一种移动方法,用最少的移动次数使每堆上纸牌数都一样多。

例如 n=4,4 堆纸牌数分别为: ①9 ②8 ③17 ④6

移动 3 次可达到目的:

从 ③ 取 4 张牌放到④(9 8 13 10)->从③取 3 张牌放到 ②(9 11 10 10)-> 从②取 1

张牌放到①(10 10 10 10)。

【输入格式】

n(n 堆纸牌,1 <=n <=100)

a1 a2 … an (n 堆纸牌,每堆纸牌初始数,l<=ai <=10000)

【输出格式】

所有堆均达到相等时的最少移动次数。

【输入样例】　　【输出样例】

4　　3

9 8 17 6

【算法分析】

如果你想到把每堆牌的张数减去平均张数,题目就变成移动正数,加到负数中,使大家都变成 0,那就意味着成功了一半!拿例题来说,平均张数为 10,原张数 9,8,17,6,变为-1,-2,7,-4,其中没有为 0 的数,我们从左边出发:要使第 1 堆的牌数-1 变为 0,只需将-1 张牌移到它的右边(第 2 堆)-2 中;结果是-1 变为 0,-2 变为-3,各堆牌张数变为 0,-3,7,-4;同理:要使第 2 堆变为 0,只需将-3 移到它的右边(第 3 堆)中去,各堆牌张数变为 0,0,4,-4;要使第 3 堆变为 0,只需将第 3 堆中的 4 移到它的右边(第 4 堆)中去,结果为 0,0,0,0,完成任务。每移动 1 次牌,步数加 1。也许你要问,负数张牌怎么移,不违反题意吗?其实从第 i 堆移动-m 张牌到第 i+1 堆,等价于从第 i+1 堆移动 m 张牌到第 i 堆,步数是一样的。

如果张数中本来就有为 0 的,怎么办呢?如 0,-1,-5,6,还是从左算起(从右算起也完全一样),第 1 堆是 0,无需移牌,余下与上相同;再比如-1,-2,3,10,-4,-6,从左算起,第 1 次移动的结果为 0,-3,3,10,-4,-6;第 2 次移动的结果为 0,0,0,10,-4,-6,现在第 3 堆已经变为 0 了,可节省 1 步,余下继续。

参考程序主要框架如下:

```
cin>>n;
ave=0;step=0;
for (i=1;i<=n;i++)
{
   cin>>a[i]; ave+=a[i];              //读入各堆牌张数,求总张数 ave
}
 ave/=n;                              //求牌的平均张数 ave
 for (i=1;i<=n;i++) a[i]-=ave;        //每堆牌的张数减去平均数
 i=1;j=n;
 while (a[i]==0&&i<n) i++;            //过滤左边的 0
 while (a[j]==0&&j>1) j--;            //过滤右边的 0
 while (i<j)
 {
   a[i+1]+=a[i];                      //将第 i 堆牌移到第 i+1 堆中去
   a[i]=0;                            //第 i 堆牌移走后变为 0
   step++;                            //移牌步数计数
```

```
    i++;                                    //对下一堆牌进行循环操作
    while (a[i]==0&&i<j) i++;               //过滤移牌过程中产生的0
  }
cout<<step<<endl;
```

点评:基本题,本题有3点比较关键:一是善于将每堆牌数减去平均数,简化了问题;二是要过滤掉0(不是所有的0,如-2,3,0,-1中的0是不能过滤的);三是负数张牌也可以移动,这是关键中的关键。

例6.3 删数问题(Noi1994)

输入一个高精度的正整数n,去掉其中任意s个数字后剩下的数字按原左右次序组成一个新的正整数。编程对给定的n和s,寻找一种方案使得剩下的数字组成的新数最小。

输出新的正整数。(n不超过240位)

输入数据均不需判错。

【输入格式】

n

s

【输出格式】

最后剩下的最小数。

【输入样例】	【输出样例】
175438	13
4	

【算法分析】

由于正整数n的有效数位为240位,所以很自然地采用字符串类型存贮n。那么如何决定哪s位被删除呢?是不是最大的s个数字呢?显然不是,大家很容易举出一些反例。为了尽可能逼近目标,我们选取的贪心策略为:每一步总是选择一个使剩下的数最小的数字删去,即按高位到低位的顺序搜索,若各位数字递增,则删除最后一个数字;否则删除第一个递减区间的首字符,这样删一位便形成了一个新数字串。然后回到串首,按上述规则再删下一个数字。重复以上过程s次为止,剩下的数字串便是问题的解了。

例如:n=175438

s=4

删数的过程如下:

```
n=175438                    //删掉7
  15438                     //删掉5
  1438                      //删掉4
  138                       //删掉8
  13                        //解为13
```

这样,删数问题就与如何寻找递减区间首字符这样一个简单的问题对应起来。不过还要注意一个细节性的问题,就是可能会出现字符串串首有若干0的情况,甚至整个字符串都是0的情况。按以上贪心策略编制的程序框架如下:

```
输入n,s;
for (i=1;i<=s;i++)                  //一共要删除s个字符
```

```
{
  for( j=0;j<len-1;j++)          //从串首开始找,len 是 n 的长度
    if( n[j]>n[j+1])             //找到第一个符合条件的
    {
      for( k=j;k<len-1;k++) //删除字符串 n 的第 j 个字符 ,后面字符往前整
        n[k]=n[k+1];
      break;
    }
  len--;                         //长度减 1
}
for(i=0;i<=len-1;i++)
{ if (n[i]! ='0') flag=true;     //删除串首无用的零,输出答案
  if (flag) cout<<n[i];
}
```

例 6.4　拦截导弹问题(Noip1999)

某国为了防御敌国的导弹袭击,开发出一种导弹拦截系统,但是这种拦截系统有一个缺陷:虽然它的第一发炮弹能够到达任意的高度,但是以后每一发炮弹都不能高于前一发的高度。某天,雷达捕捉到敌国的导弹来袭,由于该系统还在试用阶段。所以一套系统有可能不能拦截所有的导弹。

输入导弹依次飞来的高度(雷达给出的高度不大于 30000 的正整数)。计算要拦截所有导弹最小需要配备多少套这种导弹拦截系统。

【输入格式】

n 颗依次飞来的高度(1<=n<=1000)。

【输出格式】

要拦截所有导弹最小配备的系统数 k。

【输入样例】　　【输出样例】

389　207　155　300　299　170　158　65　　2

【算法分析】

按照题意,被一套系统拦截的所有导弹中,最后一枚导弹的高度最低。

设:k 为当前配备的系统数;

l[k]为被第 k 套系统拦截的最后一枚导弹的高度,简称系统 k 的最低高度(1<=k<=n)。

我们首先设导弹 1 被系统 1 所拦截(k←1,l[k]←导弹 1 的高度)。然后依次分析导弹 2,…,导弹 n 的高度。

若导弹 i 的高度高于所有系统的最低高度,则断定导弹 i 不能被这些系统所拦截,应增设一套系统来拦截导弹 i(k←k+1,l[k]←导弹 i 的高度);若导弹 i 低于某些系统的最低高度,那么导弹 i 均可被这些系统所拦截。究竟选择哪个系统拦截可使得配备的系统数最少,我们不妨采用贪心策略,选择其中最低高度最小(即导弹 i 的高度与系统最低高度最接近)的一套系统 p(l[p]=min{l[j]|l[j]>导弹 i 的高度};l[p]←导弹 i 的高度)(i<=j<=k)。这样可使得一套系统拦截的导弹数尽可能增多。

依此类推,直至分析了 n 枚导弹的高度为止。此时得出的 k 便为应配备的最少系统数。

参考程序主要框架如下：

```
k=1;l[k]=导弹1的高度;
for (i=2;i<=n;i++)
  {
    p=0;
    for (j=1;j<=k;j++)
      if (l[j]>=导弹i的高度) {if (p==0) p=j;
                              else if (l[j]<l[p]) p=j;}  //贪心
    if (p==0) { k++;l[k]=导弹i的高度; } //增加一套新系统
      else l[p]=导弹i的高度;                    //贪心,更新第p套系统的最低高度
  }
```

输出应配备的最少系统数 k。

例 6.5 活动选择

学校在最近几天有 n 个活动，这些活动都需要使用学校的大礼堂，在同一时间，礼堂只能被一个活动使用。由于有些活动时间上有冲突，学校办公室人员只好让一些活动放弃使用礼堂而使用其他教室。

现在给出 n 个活动使用礼堂的起始时间 begini 和结束时间 endi(begini < endi)，请你帮助办公室人员安排一些活动来使用礼堂，要求安排的活动尽量多。

【输入格式】

第 1 行一个整数 n(n<=1000)；

接下来的 n 行，每行两个整数，第一个 begini，第 2 个是 endi(begini< endi <=32767)

【输出格式】

输出最多能安排的活动个数。

【输入样例】	【输出样例】
11	4
3 5	
1 4	
12 14	
8 12	
0 6	
8 11	
6 10	
5 7	
3 8	
5 9	
2 13	

【算法分析】

算法模型：给 n 个开区间(begini,endi)，选择尽量多的区间，使得两两不交。

做法：首先按照 end1<=end2<=…<=endn 的顺序排序，依次考虑各个活动，如果没有和已经选择的活动冲突，就选；否则就不选。

正确性：如果不选 end1，假设第一个选择的是 endi，则如果 endi 和 end1 不交叉则多选一个 end1 更划算；如果交叉则把 endi 换成 end1 不影响后续选择。

【参考程序】

```
#include<iostream>
using namespace std;
int n,begin[1001],end[1001];
void init()
{
  cin>>n;
  for(int i=1;i<=n;i++)
    cin>>begin[i]>>end[i];
}
void qsort(int x,int y)
{
  int i,j,mid,t;
  i=x;j=y;mid=end[(x+y)/2];
  while(i<=j)
   {
     while(end[i]<mid) i++;
     while(end[j]>mid) j--;
     if(i<=j)
      {
        t=end[j];end[j]=end[i];end[i]=t;
        t=begin[j];begin[j]=begin[i];begin[i]=t;
        i++;j--;
      }
   }
   if(x<j) qsort(x,j);
   if(i<y) qsort(i,y);
}
void solve()
{
  int ans=0;
  for(int i=1,t=-1;i<=n;i++)      //在初始化循环变量的同时，初始化 t
                //令 t=-1 可以使第一个区间与其他区间的操作相同
   if(begin[i]>=t) {ans++;t=end[i];}
                //如果当前活动与之前最后结束的活动不冲突，就接受当前活动
  cout<<ans<<endl;
}
```

```
int main()
{
  init();
  qsort(1,n);
  solve();
  return 0;
}
```

例 6.6 整数区间

请编程完成以下任务：

1. 从文件中读取闭区间的个数及它们的描述；

2. 找到一个含元素个数最少的集合，使得对于每一个区间，都至少有一个整数属于该集合，输出该集合的元素个数。

【输入格式】

首行包括区间的数目 n，1<=n<=10000，接下来的 n 行，每行包括两个整数 a，b，被一空格隔开，0<=a<=b<=10000，它们是某一个区间的开始值和结束值。

【输出格式】

第 1 行集合元素的个数，对于每一个区间都至少有一个整数属于该区间，且集合所包含元素数目最少。

【输入样例】

```
4
3 6
2 4
0 2
4 7
```

【输出样例】

```
2
```

【算法分析】

算法模型：给 n 个闭区间[ai，bi]，在数轴上选尽量少的点，使每个区间内至少有一个点。

算法：首先按 b1<=b2<=…<=bn 排序。每次标记当前区间的右端点 x，并右移当前区间指针，直到当前区间不包含 x，再重复上述操作。

如图 6－1，如果选灰色点，移动到黑色点更优。

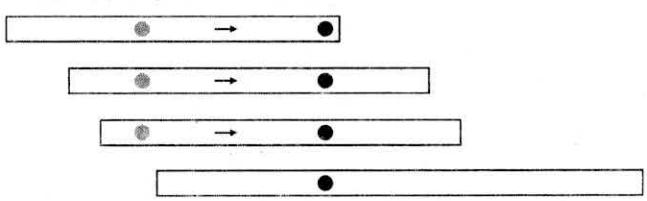


图 6--1

【参考程序】

```
#include<iostream>
using namespace std;
int a[10001],b[10001],sum=0,n,m;
```

```
void qsort(int x,int y)                    //多关键字快排
{
  int i,j,mid1,mid2,t;
  i=x;j=y;mid1=b[(x+y)/2];mid2=a[(x+y)/2];
  while(i<=j)
  {
    while(b[i]<mid1||((b[i]==mid1)&&(a[i]<mid2)))i++;
    while(b[j]>mid1||((b[j]==mid1)&&(a[j]>mid2)))j--;
    if(i<=j)
    {
      t=b[j];b[j]=b[i];b[i]=t;
      t=a[j];a[j]=a[i];a[i]=t;
      i++;j--;
    }
  }
  if(x<j) qsort(x,j);
  if(i<y) qsort(i,y);
}
int main()
{
  cin>>n;
  for(int i=1;i<=n;i++)cin>>a[i]>>b[i];
  qsort(1,n);
  for(int i=1,x=-1;i<=n;i++) //在初始化循环变量的同时,初始化 x
    {                        //令 x=-1 可以使第一个区间与其他区间的操作相同
      if (x>=a[i]) continue;     //如果当前区间包含标记点,就跳过
      sum++;  x=b[i];            //更新标记点
    }
  cout<<sum<<endl;
  return 0;
}
```

【上机练习】

1. An Easy Problem【4.6 算法之贪心 1455 poj2453】

题意:给定一个正整数 N,求最小的、比 N 大的正整数 M,使得 M 与 N 的二进制表示中有相同数目的 1。

举个例子,假如给定的 N 为 78,其二进制表示为 1001110,包含 4 个 1,那么最小的比 N 大的并且二进制表示中只包含 4 个 1 的数是 83,其二进制是 1010011,因此 83 就是答案。

输入:

输入若干行,每行一个数 n(1<=n<=1000000),输入"0"结束。

输出:

输出若干行对应的值。

样例输入:

```
1
2
3
4
78
0
```

样例输出:

```
2
4
5
8
83
```

2. 最大子矩阵【4.6 算法之贪心 1768】

已知矩阵的大小定义为矩阵中所有元素的和。给定一个矩阵,你的任务是找到最大的非空(大小至少是 1 * 1)子矩阵。

比如,如下 4 * 4 的矩阵

```
 0 -2 -7  0
 9  2 -6  2
-4  1 -4  1
-1  8  0 -2
```

的最大子矩阵是

```
 9 2
-4 1
-1 8
```

这个子矩阵的大小是 15。

输入:

输入是一个 N * N 的矩阵。输入的第一行给出 N(0<N<=100)。再后面的若干行中,

依次(首先从左到右给出第一行的 N 个整数,再从左到右给出第二行的 N 个整数……)给出矩阵中的 N2 个整数,整数之间由空白字符分隔(空格或者空行)。已知矩阵中整数的范围都在[-127,127]。

输出:

输出最大子矩阵的大小。

样例输入:

```
 4
 0 -2 -7  0
 9  2 -6  2
-4  1 -4  1
-1  8  0 -2
```

样例输出:

```
15
```

3. 金银岛【4.6 算法之贪心 1797】

某天 KID 利用飞行器飞到了一个金银岛上,上面有许多珍贵的金属,KID 虽然更喜欢各种宝石的艺术品,可是也不拒绝这样珍贵的金属。但是他只带着一个口袋,口袋至多只能装重量为 w 的物品。岛上金属有 s 个种类,每种金属重量不同,分别为 n1,n2,…,ns,同时每个种类的金属总的价值也不同,分别为 v1,v2,…,vs。KID 想一次带走价值尽可能多的金属,问他最多能带走价值多少的金属。注意到金属是可以被任意分割的,并且金属的价值和其重量成正比。

输入:

第 1 行是测试数据的组数 k,后面跟着 k 组输入。

每组测试数据占 3 行,第 1 行是一个正整数 w(1<=w<=10000),表示口袋承重上限。第 2 行是一个正整数 s(1<=s<=100),表示金属种类。第 3 行有 2s 个正整数,分别为 n1,v1,n2,v2,…,ns,vs 分别为第一种,第二种,…,第 s 种金属的总重量和总价值(1<=ni <=10000,1<=vi<=10000)。

输出:

k 行,每行输出对应一个输入。输出应精确到小数点后 2 位。

样例输入:

```
2
50
4
10 100 50 30 7 34 87 100
10000
5
1 43 43 323 35 45 43 54 87 43
```

样例输出:

```
171.93
508.00
```

4. 装箱问题【4.6 算法之贪心 19】

一个工厂制造的产品形状都是长方体,它们的高度都是 h,长和宽都相等,一共有六个

型号,他们的长宽分别为1*1,2*2,3*3,4*4,5*5,6*6。这些产品通常使用一个6*6*h的长方体包裹包装然后邮寄给客户。因为邮费很贵,所以工厂要想方设法的减小每个订单运送时的包裹数量。他们很需要有一个好的程序帮他们解决这个问题从而节省费用。现在这个程序由你来设计。

输入:

输入文件包括几行,每一行代表一个订单。每个订单里的一行包括六个整数,中间用空格隔开,分别为1*1至6*6这六种产品的数量。输入文件将以6个0组成的一行结尾。

输出:

除了输入的最后一行6个0以外,输入文件里每一行对应着输出文件的一行,每一行输出一个整数代表对应的订单所需的最小包裹数。

样例输入:

```
0 0 4 0 0 1
7 5 1 0 0 0
0 0 0 0 0 0
```

样例输出:

```
2
1
```

5. Ride to Office【4.6 算法之贪心 2404poj1922】

题意:起点与终点相隔4500米。现Charley需要从起点骑车到终点。但是,他有个习惯,沿途需要有人陪伴,即以相同的速度,与另外一个人一起骑。而当他遇到以更快的速度骑车的人时,他会以相应的速度跟上这个更快的人。先给定所有与Charley同路的人各自的速度与出发时间,问Charley以这种方式跟人,骑完4500米需要多少时间。得出的结果若是小数,则向上取整。

输入:

输入若干组数据,每组数据第一行n(1<=n<=10000),n为0,表示输入结束,接着输入n行数据,每行2个数据,表示速度v和出发时间t,如果t<0,表示陪伴人提早出发了。

输出:

输出对应若干行数据,每行输出1个数,表示最快到达的时间。

样例输入:

```
4
20   0
25 -155
27   190
30   240
2
21   0
22   34
0
```

样例输出:

```
780
```

771

6. 书架【4.6 算法之贪心 2407】

John 最近买了一个书架用来存放奶牛养殖书籍，但书架很快被存满了，只剩最顶层有空余。

John 共有 N 头奶牛($1 \leqslant N \leqslant 20,000$)，每头奶牛有自己的高度 Hi($1 \leqslant Hi \leqslant 10,000$)，N 头奶牛的总高度为 S。书架高度为 B($1 \leqslant B \leqslant S < 2,000,000,007$)。

为了到达书架顶层，奶牛可以踩着其他奶牛的背，像叠罗汉一样，直到他们的总高度不低于书架高度。当然若奶牛越多则危险性越大。为了帮助 John 到达书架顶层，找出使用奶牛数目最少的解决方案吧。

输入：

第 1 行：空格隔开的整数 N 和 B。

第 2～N+1 行：第 i+1 行为整数 Hi。

输出：

能达到书架高度所使用奶牛的最少数目

样例输入：

6 40

6

18

11

13

19

11

样例输出：

3

7. 电池的寿命【4.6 算法之贪心 2469】

小 S 新买了一个掌上游戏机，这个游戏机由两节 5 号电池供电。为了保证能够长时间玩游戏，他买了很多 5 号电池，这些电池的生产商不同，质量也有差异，因而使用寿命也有所不同，有的能使用 5 个小时，有的可能就只能使用 3 个小时。显然如果他只有两个电池一个能用 5 小时一个能用 3 小时，那么他只能玩 3 个小时的游戏，有一个电池剩下的电量无法使用，但是如果他有更多的电池，就可以更加充分地利用它们，比如他有三个电池分别能用 3、3、5 小时，他可以先使用两节能用 3 个小时的电池，使用半个小时后再把其中一个换成能使用 5 个小时的电池，两个半小时后再把剩下的一节电池换成刚才换下的电池(那个电池还能用 2.5 个小时)，这样总共就可以使用 5.5 个小时，没有一点浪费。

现在已知电池的数量和电池能够使用的时间，请你找一种方案使得使用时间尽可能的长。

输入：

输入包含多组数据。每组数据包括两行，第一行是一个整数 N($2 \leqslant N \leqslant 1000$)，表示电池的数目，接下来一行是 N 个正整数表示电池能使用的时间。

输出：

对每组数据输出一行，表示电池能使用的时间，保留到小数点后 1 位。

样例输入：

2

3 5

3

3 3 5

样例输出:

3.0

5.5

8. 寻找平面上的极大点【4.6 算法之贪心 2704】

在一个平面上,如果有两个点(x,y),(a,b),如果说(x,y)支配了(a,b),这是指 x>=a,y>=b;

用图形来看就是(a,b)坐落在以(x,y)为右上角的一个无限的区域内。

给定 n 个点的集合,一定存在若干个点,它们不会被集合中的任何一点所支配,这些点叫做极大值点。

编程找出所有的极大点,按照 x 坐标由小到大,输出极大点的坐标。

本题规定:n 不超过 100,并且不考虑点的坐标为负数的情况。

输入:

输入包括两行,第一行是正整数 n,表示是点数,第二行包含 n 个点的坐标,坐标值都是整数,坐标范围从 0 到 100,输入数据中不存在坐标相同的点。

输出:

按 x 轴坐标最小到大的顺序输出所有极大点。

输出格式为:(x1,y1),(x2,y2),…(xk,yk)。

注意:输出的每个点之间有","分隔,最后一个点之后没有",",少输出和多输出都会被判错。

样例输入:

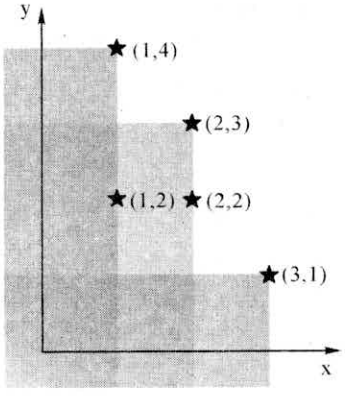
5

1 2 2 2 3 1 2 3 1 4

样例输出:

(1,4),(2,3),(3,1)

提示:



9. 最小新整数【4.6 算法之贪心 3528】

给定一个十进制正整数 n(0<n<1000000000),每个数位上数字均不为 0。n 的位数

为 m。

现在从 m 位中删除 k 位(0<k<m),求生成的新整数最小为多少?

例如:n=9128456,k=2,则生成的新整数最小为 12456

输入:

第一行 t,表示有 t 组数据;

接下来 t 行,每一行表示一组测试数据,每组测试数据包含两个数字 n,k。

输出:

t 行,每行一个数字,表示从 n 中删除 k 位后得到的最小整数。

样例输入:

2

9128456 2

1444 3

样例输出:

12456

1

10. Crossing River【4.6 算法之贪心 702】

题意:几个人过河,每次过两人一人回,速度由慢者决定,问过河所需最短时间。

输入:

输入 t 组数据,每组数据第 1 行输入 n,第 2 行输入 n 个数,表示每个人过河的时间。

输出:

输出 t 行数据,每行 1 个数,表示每组过河最少时间。

样例输入:

1

4

1 2 5 10

样例输出:

17

11. 接水问题【1.9 编程基础之顺序查找 15】Noip2010 普及组第 2 题

学校里有一个水房,水房里一共装有 m 个龙头可供同学们打开水,每个龙头每秒钟的供水量相等,均为 1。

现在有 n 名同学准备接水,他们的初始接水顺序已经确定。将这些同学按接水顺序从 1 到 n 编号,i 号同学的接水量为 wi。接水开始时,1 到 m 号同学各占一个水龙头,并同时打开水龙头接水。当其中某名同学 j 完成其接水量要求 wj 后,下一名排队等候接水的同学 k 马上接替 j 同学的位置开始接水。这个换人的过程是瞬间完成的,且没有任何水的浪费。即 j 同学第 x 秒结束时完成接水,则 k 同学第 x+1 秒立刻开始接水。若当前接水人数 n'不足 m,则只有 n'个龙头供水,其它 m−n'个龙头关闭。

现在给出 n 名同学的接水量,按照上述接水规则,问所有同学都接完水需要多少秒。

输入:

第 1 行 2 个整数 n 和 m,用一个空格隔开,分别表示接水人数和龙头个数。

第 2 行 n 个整数 w1、w2、…、wn,每两个整数之间用一个空格隔开,wi 表示 i 号同学的

接水量。

输出：

输出只有一行，1 个整数，表示接水所需的总时间。

样例输入：

样例 #1：

5 3

4 4 1 2 1

样例 #2：

8 4

23 71 87 32 70 93 80 76

样例输出：

样例 #1：

4

样例 #2：

163

提示：

输入输出样例 1 解释：

第 1 秒，3 人接水。第 1 秒结束时，1、2、3 号同学每人的已接水量为 1，3 号同学接完水，4 号同学接替 3 号同学开始接水。

第 2 秒，3 人接水。第 2 秒结束时，1、2 号同学每人的已接水量为 2，4 号同学的已接水量为 1。

第 3 秒，3 人接水。第 3 秒结束时，1、2 号同学每人的已接水量为 3，4 号同学的已接水量为 2。4 号同学接完水，5 号同学接替 4 号同学开始接水。

第 4 秒，3 人接水。第 4 秒结束时，1、2 号同学每人的已接水量为 4，5 号同学的已接水量为 1。1、2、5 号同学接完水，即所有人完成接水。

总接水时间为 4 秒。

第七章　分治算法

所谓分治就是指分而治之，即将较大规模的问题分解成几个较小规模的问题，通过对较小规模问题的求解达到对整个问题的求解。当我们将问题分解成两个较小问题求解时的分治方法称之为二分法。

你们玩过猜数字的游戏吗？你的朋友心里想一个 1000 以内的正整数，你可以给出一个数字 x，你朋友只要回答“比 x 大”或者“比 x 小”或者“猜中”，你能保证在 10 次以内猜中吗？运气好只要一次就猜中。

开始猜测是 1 到 1000 之间，你可以先猜 500，运气好可以一次猜中，如果答案比 500 大，显然答案不可能在 1 到 500 之间，下一次猜测的区间变为 501 到 1000，如果答案比 500 小，那答案不可能在 500 到 1000 之间，下一次猜测的区间变为 1 到 499。只要每次都猜测区间的中间点，这样就可以把猜测区间缩小一半。由于$\frac{1000}{2^{10}}<1$，因此不超过 10 次询问区间就可以缩小为 1，答案就会猜中了，这就是二分的基本思想。

每一次使得可选的范围缩小一半，最终使得范围缩小为一个数，从而得出答案。假设问的范围是 1 到 n，根据$\frac{n}{2^x}\leqslant 1$ 得 $x\geqslant \log_2 n$，所以我们只需要问 O(logn)次就能知道答案了。

需要注意的是使用二分法有一个重要的前提，就是有序性，下面通过几个例子来体会二分法的应用。

例 7.1　找数。

描述：

给一个长度为 n 的单调递增的正整数序列，即序列中每一个数都比前一个数大。有 m 个询问，每次询问一个 x，问序列中最后一个小于等于 x 的数是什么？

输入：

第一行两个整数 n，m。

接下来一行 n 个数，表示这个序列。

接下来 m 行每行一个数，表示一个询问。

输出：

输出共 m 行，表示序列中最后一个小于等于 x 的数是什么。假如没有，则输出－1。

样例输入：

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

样例输出：

4

1

3

分析：

我们用 Left 表示询问区间的左边界，用 Right 表示询问区间的右边界，[Left，Right]组成询问区间。一开始 Left=1，Right=n，我们可以把原始序列的左边想象成若干个无穷小的数，把序列的右边想象成无穷大的数，这样比较好理解。序列已经按照升序排好，保证了二分的有序性。

每一次二分，我们这样来做：

①取区间中间值 Mid=(Left+Right)/2；

②判断 Mid 与 x 的关系，如果 a[Mid]>x，由于序列是升序排列，所以区间[Mid，Right]都可以被排除，修改右边界 Right=Mid-1；

③如果 a[Mid]<=x，修改左边界 Left=Mid+1；

重复执行二分操作直到 Left>Right。

下面我们来分析答案的情况。循环结束示意图如图 7-1：

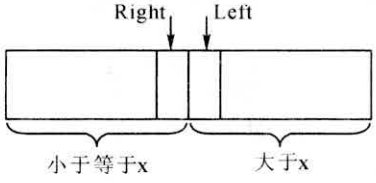


图 7-1

最终循环结束时一定是 Left=Right+1，根据二分第②步的做法我们知道 Right 的右边一定都是大于 x 的，而根据第③步我们可以知道 Left 左边一定是小于等于 x 的。

所以，一目了然，最终答案是 Right 指向的数。Right=0 就是题目中输出-1 的情况。

程序如下：

```
#include <iostream>
using namespace std;
int n,m,a[110000];
int main()
{
    cin >> n >> m;
    for (int i=1; i<=n; i++) cin >> a[i];
    a[0]=-1;
    for (int i=1; i<=m; i++)
    {
        int x;
        int left=1,right=n,mid;
        cin >> x;
        while (left <= right)
        {
```

```
            mid = (left + right) / 2;
            if (a[mid] <= x) left = mid + 1;
            else right = mid - 1;
        }
        cout << a[right] << endl;
    }
    return 0;
}
```

例 7.2　快速排序(Quicksort)

快速排序由 C. A. R. Hoare 在 1962 年提出。它的基本思想是:通过一趟排序将要排序的数据分割成独立的两部分,其中一部分的所有数据都比另外一部分的所有数据都要小,然后再按此方法对这两部分数据分别进行快速排序,整个排序过程可以递归进行,以此达到整个数据变成有序序列。

设要排序的数组是 A[0]……A[N−1],首先任意选取一个数据(通常选用数组的中间数)作为关键数据,然后将所有比它小的数都放到它前面,所有比它大的数都放到它后面,这个过程称为一趟快速排序。值得注意的是,快速排序不是一种稳定的排序算法,也就是说,多个相同的值的相对位置也许会在算法结束时产生变动。

一趟快速排序的算法是:

(1)设置两个变量 i、j,排序开始的时候:i=0,j=N−1;

(2)以数组中任意元素作为关键数据(一般以数组中间元素作为关键数据),赋值给 key,可以是 key=A[(i+j)/2];

(3)从 j 开始向前搜索,即由后开始向前搜索(j−−),找到第一个小于等于 key 的值 A[j];

(4)从 i 开始向后搜索,即由前开始向后搜索(i++),找到第一个大于等于 key 的 A[i];

(5)交换 A[i]和 A[j]的值,同时 i++,j−−;

(6)重复第 3、4、5 步,直到 i>j;

例如有 8 个元素需要排序:

6 10 11 8 4 1 9 7

一趟快速排序后:

```
6  10  11  8  4  1  9  7   key=8
↑                       ↑
i                       j
6  10  11  8  4  1  9  7   key=8
   ↑                    ↑
   i                    j
6  7   11  8  4  1  9  10  key=8
   ↑                    ↑
   i                    j
6  7   11  8  4  1  9  10  key=8
      ↑        ↑
      i        j
6  7   1   8  4  11 9  10  key=8
      ↑        ↑
      i        j
6  7   1   8  4  11 9  10  key=8
         ↑  ↑
         i  j
6  7   1   4  8  11 9  10  key=8
         ↑  ↑
         i  j
6  7   1   4  8  11 9  10  key=8
         ↑  ↑
         j  i
```

此时 i>j,并且 i 左边的数字都小于等于 key,j 右边的数字都大于等于 key,进而接下来可以分别对左边段[0, j]和右边段[i,N-1]利用同样的方法排序。

【程序实现】

```
void qsort(int le,int ri)
{
    int i=le, j=ri, mid=a[(le+ri)/2];
    while(i<=j)                    //注意这里要有等号
    {
        while(a[i]<mid) i++;  //在左边找大于等于 mid 的数
        while(a[j]>mid) j--;  //在右边找小于等于 mid 的数
        if(i<=j)
        {
            swap(a[i],a[j]);      //交换
            i++, j--;             //继续找
        }
    }
    if(le<j) qsort(le,j);          //分别递归继续排序
    if(i<ri) qsort(i,ri);
}
```

快速排序(Quicksort)是对冒泡排序的一种改进,快速排序的时间复杂度是 O(nlogn),速度快,但它是不稳定的排序方法。就平均时间而言,快速排序是目前被认为是最好的一种内部排序的方法。但快速排序需要一个栈空间来实现递归,若每一趟排序都将记录序列均匀地分割成长度相近的两个子序列,则栈的最大深度为 log(n+1)。

例 7.3 一元三次方程求解

有形如 $ax^3+bx^2+cx+d=0$ 一个一元三次方程。给出该方程中各项的系数(a、b、c、d 均为实数),并约定该方程存在三个不同实根(根的范围在-100 至 100 之间),且根与根之差的绝对值>=1。

要求由小到大依次在同一行输出这三个实根(根与根之间留有空格),并精确到小数点后 2 位。

提示:记方程 f(x)=0,若存在 2 个数 x_1 和 x_2,且 $x_1<x_2$,$f(x_1)*f(x_2)<0$,则在(x_1,x_2)之间一定有一个根。

输入:a,b,c,d　　输出:三个实根(根与根之间留有空格)

【输入样例】　　【输出样例】

1 -5 -4 20　　-2.00 2.00 5.00

【算法分析】

这是一道有趣的解方程题。为了便于求解,设方程 $f(x)=ax^3+bx^2+cx+d=0$,设根的值域(-100 至 100 之间)中有 x,其左右两边相距 0.0005 的地方有 x_1 和 x_2 两个数,即 $x_1=x-0.0005$,$x_2=x+0.0005$。x_1 和 x_2 间的距离(0.001)满足精度要求(精确到小数点后 2 位)。若出现如图 7-2 所示的两种情况之一,则确定 x 为 f(x)=0 的根。

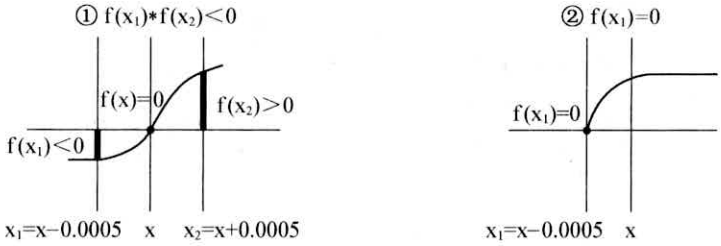


图 7－2

有两种方法计算 f(x)＝0 的根 x：

(1)枚举法

根据根的值域和根与根之间的间距要求(＞＝1)，我们不妨将根的值域扩大 100 倍(－10000＜＝x＜＝10000)，依次枚举该区间的每一个整数值 x，并在题目要求的精度内设定区间：$x_1=\frac{x-0.05}{100}$，$x_2=\frac{x+0.05}{100}$。若区间端点的函数值 $f(x_1)$ 和 $f(x_2)$ 异号或者在区间端点 x_1 的函数值 $f(x_1)=0$，则确定 $\frac{x}{100}$ 为 f(x)＝0 的一个根。

由此得出算法：

```
输入方程中各项的系数 a,b,c,d ;
for (x=－10000;x<=10000;x++)              //枚举当前根 * 100 的可能范围
{
  x1=(x－0.05)/100;x2=(x+0.05)/100;   //在题目要求的精度内设定区间
  if ((f(x1) * f(x2)<0) || (f(x1)==0))
          //若在区间两端的函数值异号或在 x1 处的函数值为 0,则确定 x/100 为根
     printf("%.2f",x/100);
}
```

其中函数 f(x)计算 $x^3+b*x^2+c*x+d$：

```
double f(double x)                        //计算 x^3+b*x^2+c*x+d
  {
    f=x*x*x+b*x*x+c*x+d;
  }                                       //f 函数
```

(2)分治法

枚举根的值域中的每一个整数 x(－100＜＝x＜＝100)。由于根与根之差的绝对值＞＝1，因此设定搜索区间[x_1，x_2]，其中 $x_1=x$，$x_2=x+1$。

(1)若 $f(x_1)=0$，则确定 x_1 为 f(x)的根；

(2)若 $f(x_1)*f(x_2)>0$，则确定根 x 不在区间[x_1，x_2]内，设定[x_2，x_2+1]为下一个搜索区间；

(3)若 $f(x_1)*f(x_2)<0$，则确定根 x 在区间[x_1，x_2]内。

如果确定根 x 在区间[x_1，x_2]内的话($f(x_1)*f(x_2)<0$)，如何在该区间找到根的确切位置。采用二分法，将区间[x_1，x_2]分成左右两个子区间：左子区间[x_1，x]和右子区间[x，x_2](其中 $x=\frac{x_1+x_2}{2}$)：

若 $f(x_1)*f(x)<=0$,则确定根在左子区间$[x_1,x]$内,将 x 设为该区间的右指针($x_2=x$),继续对左子区间进行对分;若 $f(x_1)*f(x)>0$,则确定根在右子区间$[x,x_2]$内,将 x 设为该区间的左指针($x_1=x$),继续对右子区间进行对分;

上述对分过程一直进行到区间的间距满足精度要求为止($x_2-x_1<0.001$)。此时确定 x_1 为 f(x)的根。

由此得出算法:

输入方程中各项的系数 a,b,c,d ;

```
{
  for (x=-100;x<=100;x++)              //枚举每一个可能的根
   {
    x1=x;x2=x+1;                       //确定根的可能区间
    if (f(x1)==0) printf("%.2f  ",x1); //若 x1 为根,则输出
      else if (f(x1)*f(x2)<0)          //若根在区间[x1,x2]中
       {
         while (x2-x1>=0.001)          //若区间[x1,x2]不满足精度要求,则循环
          {
            xx=(x2+x1)/2;              //计算区间[x1,x2]的中间位置
            if ((f(x1)*f(xx))<=0)      //若根在左子区间,则调整右指针
              x2=xx;
             else  x1=xx;              //若根在右子区间,则调整左指针
          }
         printf("%.2f  ",x1);          //区间[x1,x2]满足精度要求,确定 x1 为根
       }
   }
   cout<<endl;
}
double f(double x)                     //将 x 代入函数
{
    return (x*x*x*a+b*x*x+x*c+d);
}
```

其中 f(x)的函数说明如枚举法所示。

例 7.4 循环比赛日程表(match)

【问题描述】

设有 n 个选手进行循环比赛,其中 $n=2^m$,要求每名选手要与其他 n-1 名选手都赛一次,每名选手每天比赛一次,循环赛共进行 n-1 天,要求每天没有选手轮空。

输入:m

输出:表格形式的比赛安排表

【输入样例】

3

【输出样例】

1 2 3 4 5 6 7 8

2 1 4 3 6 5 8 7

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

【问题分析】

以 m=3(即 $n=2^3=8$)为例,可以根据问题要求,制定出如下表所示的一种方案:

	第一天	第二天	第三天	第四天	第五天	第六天	第七天
1	2	3	4	5	6	7	8
2	1	4	3	6	5	8	7
3	4	1	2	7	8	5	6
4	3	2	1	8	7	6	5
5	6	7	8	1	2	3	4
6	5	8	7	2	1	4	3
7	8	5	6	3	4	1	2
8	7	6	5	4	3	2	1

A　B　C　D

以表格的中心为拆分点,将表格分成 A、B、C、D 四个部分,就很容易看出有 A=D,B=C,并且这一规律同样适用于各个更小的部分。

设有 n 个选手的循环比赛,其中 $n=2^m$,要求每名选手要与其他 n−1 名选手都赛一次。每名选手每天比赛一次,循环赛共进行 n−1 天。要求每天没有选手轮空,以下是八名选手时的循环比赛表,表中第一行为八位选手的编号,下面七行依次是每位选手每天的对手。

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

【算法分析】

从八位选手的循环比赛表中可以看出,这是一个具有对称性的方阵,可以把方阵一分为四来看,那么左上角的 4 * 4 的方阵就是前四位选手的循环比赛表,而右上角的 4 * 4 的方阵就是后四位选手的循环比赛表,它们在本质上是一样的,都是 4 个选手的循环比赛表,所不同的只是选手编号不同而已,将左上角中方阵的所有元素加上 4 就能得到右上角的方阵。下方的两个方阵表示前四位选手和后四位选手进行交叉循环比赛的情况,同样具有对称性,将右上角方阵复制到左下角即得到 1、2、3、4 四位选手和 5、6、7、8 四位选手的循环比赛表,根据对称性,右下角的方阵应与左上角的方阵相同。这样,八名选手的循环比赛表可以由

四名选手的循环比赛表根据对称性生成出来.同样地,四名选手的循环比赛表可以由二名选手的循环比赛表根据对称性生成出来,而两名选手的循环比赛表可以说是已知的,这种程序设计方法叫做分治法,其基本思想是把一个规模为n的问题分成若干个规模较小的问题,使得从这些较小问题的解易于构造出整个问题的解。

程序中用数组matchlist记录n名选手的循环比赛表,整个循环比赛表从最初的1*1的方阵按上述规则生成出2*2的方阵,再生成出4*4的方阵,…,直到生成出整个循环比赛表为止。变量half表示当前方阵的大小,也是要生成的下一个方阵的大小的一半。

【参考程序】

```
#include<cstdio>
const int MAXN=1025,MAXM=11;
int matchlist[MAXN][MAXN];
int m;
int main()
{
    printf("Input m:");
    scanf("%d",&m);
    int n=1<<m,k=1,half=1;          // 1<<m 相当于 2^m
    matchlist[0][0]=1;
    while (k<=m)
    {
      for (int i=0;i<half;i++)          //构造右上方方阵
        for (int j=0;j<half;j++)
          matchlist[i][j+half]=matchlist[i][j]+half;
      for (int i=0;i<half;i++)          //对称交换构造下半部分方阵
        for (int j=0;j<half;j++)
        {
          matchlist[i+half][j]=matchlist[i][j+half];//左下方方阵等于右上方方阵
          matchlist[i+half][j+half]=matchlist[i][j];//右下方方阵等于左上方方阵
        }
      half*=2;
      k++;
    }
    for (int i=0;i<n;i++)
    {
      for (int j=0;j<n;j++)
        printf("%4d",matchlist[i][j]);
      putchar('\n');
    }
    return 0;
}
```

例 7.5　取余运算(mod)

【问题描述】

输入 b、p、k 的值，求 b^p mod k 的值。其中 b、p、k * k 为长整形数。

【输入样例】　　【输出样例】

2 10 9　　2^10 mod 9=7

【算法分析】

本题主要的难点在于数据规模很大(b、p 都是长整型数)，对于 b^p 显然不能死算，那样的话时间复杂度和编程复杂度都很大。

下面先介绍一个原理：a * b%k=(a%k) * (b%k)%k。显然有了这个原理，就可以把较大的幂分解成较小的，因而免去高精度计算等复杂过程。那么怎样分解最有效呢？显然对于任何一个自然数 p，有 p=2 * p/2+p%2，如 19=2 * 19/2+19%2=2 * 9+1，利用上述原理就可以把 b 的 19 次方除以 k 的余数转换为求 b 的 9 次方除以 k 的余数，即 $b^{19}=b^{2*9+1}=b*b^9*b^9$，再进一步分解下去就不难求得整个问题的解。

【参考程序】

```
#include<iostream>
#include<cstdio>
using namespace std;
int b,p,k,a;
int f(int p)                          //利用分治求 b^p % k
{
  if (p==0) return 1;                 // b^0 %k=1
  int tmp=f(p/2)%k;
  tmp=(tmp*tmp) % k;                  // b^p %k=(b^(p/2))^2 % k
  if (p%2==1) tmp=(tmp * b) %k;
                                      //如果 p 为奇数，则 b^p % k=((b^(p/2))^2) * b%k
  return tmp;
}
int main()
{
  cin>>b>>p>>k;                       //读入 3 个数
  int tmpb=b;                         //将 b 的值备份
  b%=k;                               //防止 b 太大
  printf("%d^%d mod %d=%d\n",tmpb,p,k,f(p)); //输出
  return 0;
}
```

例 7.6　黑白棋子的移动(chessman)

【问题描述】

有 2n 个棋子(n>=4)排成一行，开始位置为白子全部在左边，黑子全部在右边，如图 7－3为 n=5 的情形：

○○○○○●●●●●

图 7-3

移动棋子的规则是:每次必须同时移动相邻的两个棋子,颜色不限,可以左移也可以右移到空位上去,但不能调换两个棋子的左右位置。每次移动必须跳过若干个棋子(不能平移),要求最后能移成黑白相间的一行棋子。如 n=5 时,成为:

○●○●○●○●○●

图 7-4

任务:编程打印出移动过程。

【输入样例】

7

【输出样例】

```
step 0: ooooooo*******--
step 1: oooooo--*******o*
step 2: oooooo******--o*
step 3: ooooo--*****o*o*
step 4: ooooo*****--o*o*
step 5: oooo--****o*o*o*
step 6: oooo****--o*o*o*
step 7: ooo--***o*o*o*o*
step 8: ooo*o**--*o*o*o*
step 9: o--*o**oo*o*o*o*
step10: o*o*o*--o*o*o*o*
step11: --o*o*o*o*o*o*o*
```

【算法分析】

我们先从 n=4 开始试试看,初始时:

○○○○●●●●

第 1 步:○○○——●●●○●　{—表示空位}

第 2 步:○○○●○●●——●

第 3 步:○——●○●●○○●

第 4 步:○●○●○●——○●

第 5 步:——○●○●○●○●

如果 n=5 呢?我们继续尝试,希望看出一些规律,初始时:

○○○○○●●●●●

第 1 步:○○○○——●●●●○●

第 2 步:○○○○●●●●——○●

这样,n=5 的问题又分解成了 n=4 的情况,下面只要再做一下 n=4 的 5 个步骤就行了。同理,n=6 的情况又可以分解成 n=5 的情况,…,所以,对于一个规模为 n 的问题,我们很容易地就把它分治成了规模为 n-1 的相同类型子问题。

数据结构如下:数组 c[1..max]用来作为棋子移动的场所,初始时,c[1]~c[n]存放白子(用字符 o 表示),c[n+1]~c[2n]存放黑子(用字符 * 表示),c[2n+1],c[2n+2]为空位置(用字符—表示)。最后结果在 c[3]~c[2n+2]中。

【参考程序】

```
#include<iostream>
```

```
using namespace std;
int n,st,sp;
char c[101];
void print()                                    //打印
{
    int i;
    cout<<" step "<<st<<':';
    for (i=1;i<=2*n+2;i++) cout<<c[i];
    cout<<endl;
    st++;
}
void init(int n)                                //初始化
{
    int i;
    st=0;
    sp=2*n+1;
    for (i=1;i<=n;i++) c[i]='o';
    for (i=n+1;i<=2*n;i++) c[i]='*';
    c[2*n+1]='-';c[2*n+2]='-';
    print();
}
void move(int k)                                //移动一步
{
    int j;
    for (j=0;j<=1;j++)
    {
      c[sp+j]=c[k+j];
      c[k+j]='-';
    }
    sp=k;
    print();
}
void mv(int n)                                  //主要过程
{
    int i,k;
    if (n==4)                                   //n 等于 4 的情况要特殊处理
    {
      move(4); move(8); move(2); move(7); move(1);
    }
    else
```

```
    {
      move(n); move(2*n-1); mv(n-1);
    }
}
int main()
{
    cin>>n;
    init(n);
    mv(n);
}
```

例 7.7 光荣的梦想

【问题描述】

Prince 对他在这片大陆上维护的秩序感到满意,于是决定启程离开艾泽拉斯。在他动身之前,Prince 决定赋予 King_Bette 最强大的能量以守护世界、保卫这里的平衡与和谐。在那个时代,平衡是个梦想。因为有很多奇异的物种拥有各种不稳定的能量,平衡瞬间即被打破。KB 决定求助于你,帮助他完成这个梦想。

一串数列即表示一个世界的状态。

平衡是指这串数列以升序排列。而从一串无序数列到有序数列需要通过交换数列中的元素来实现。KB 的能量只能交换相邻两个数字。他想知道他最少需要交换几次就能使数列有序。

【输入格式】

第 1 行为数列中数的个数 n,第 2 行为 n <=10000 个数。表示当前数列的状态。

【输出格式】

输出一个整数,表示最少需要交换几次能达到平衡状态。

【输入样例】

```
4
2 1 4 3
```

【输出样例】

```
2
```

【算法分析】归并排序求逆序对。

归并排序:假设我们已经将区间[L , K]和区间[K+1 , R]排好序,则我们可以花费线性的复杂度将区间[L , R]排好序。接下来讨论从小到大排序(从大到小排序类似):

若 a[L]<a[K+1](a[L]为区间[L, K]第一个元素,a[K+1]为区间[K+1,R]一个元素),则 b[L]=a[L](b[L]为区间[L,R]第一个元素),因为区间[L,K] 是有序的,所以 a[L]是区间[L,K]中最小的元素(同理 a[K+1]是[K+1,R]中最小的元素)。又因为 a[L]<a[K+1],所以 b[L]=a[L](b[L]是区间[L,R]最小的元素)。取出 a[L]后,a[L+1]又是区间最小的元素,比较 a[L+1]和 a[K]决定出区间[L,R]中第二小的元素,接下来找出第三小的元素…

得出算法:

(1)将区间 [1,n]进行二分,先求出区间[1,mid]和[mid+1,n](mid 是区间[1,n]的中间点),而子区间又进行二分直到区间内只有一个元素(即区间[I,I],而只有一个元素的区间一定是有序的)。

(2)将已经排完序的子区间进行合并,即用有序的[L,K][K+1,R]求出有序的区间[L,R],最终合并出区间[1,N]。而合并区间又分为若干步:

①令变量 I=L，变量 J=K+1，变量 SUM=L，执行下一步。

②如果 I<=K 且 J<=R(即两个区间内都有元素)，若 a[I]<a[J]，则令 b[SUM]=a[I]，I=I+1，SUM=SUM+1，执行下一步；若 a[I]>=a[J]，则令 b[SUM]=a[J]，J=J+1，SUM=SUM+1，执行下一步。如果 I>K 或者 J>R(说明其中一个区间内没有元素了)，则将另一个区间剩下的元素都加入 b 数组中，执行下一步。

③如果 SUM>R(说明区间[L,R]已经排好序)，则退出循环，否则执行第 2 步。

求逆序对：在合并区间的第二步中：如果 a[J]<a[I]，则说明 a[J]比区间[I,K]中的任何一个元素都小而位置又在 J 之前，所以提供了(K−I+1)对的逆序对。所以我们只要在合并区间的第二步中加个语句：if (a[J]<a[I]) ans=ans+K−I+1 即可。

```
#include<iostream>
using namespace std;
int temp[10001],a[10001],tot=0;
void merge_sort(int left,int right)
{ if(left==right)return;
  int mid=(left+right)/2;
  merge_sort(left,mid);
  merge_sort(mid+1,right);
  int p=left,i=left,j=mid+1;
  while(i<=mid&&j<=right)
    {if(a[i]>a[j])
      { tot=tot+mid-i+1;temp[p++]=a[j++]; }
      else temp[p++]=a[i++];
    }
  while(i<=mid)temp[p++]=a[i++];
  while(j<=right)temp[p++]=a[j++];
  for(i=left;i<=right;i++) a[i]=temp[i];
}
int main()
{ int n;
  cin>>n;
  for(int i=1;i<=n;i++) cin>>a[i];
  merge_sort(1,n);
  cout<<tot;
  return 0;
}
```

【上机练习】

1. 2011【2.3 基本算法之分治 2991】

已知长度最大为 200 位的正整数 n,请求出 2011^n 的后四位。

输入:

第一行为一个正整数 k,代表有 k 组数据,k<=200 接下来的 k 行,

每行都有一个正整数 n,n 的位数<=200

输出:

每一个 n 的结果为一个整数占一行,若不足 4 位,去除高位多余的 0

样例输入:

3

5

28

792

样例输出:

1051

81

5521

2. 输出前 k 大的数【2.3 基本算法之分治 7617】

给定一个数组,统计前 k 大的数并且把这 k 个数从大到小输出。

输入:

第一行包含一个整数 n,表示数组的大小。n < 100000。

第二行包含 n 个整数,表示数组的元素,整数之间以一个空格分开。每个整数的绝对值不超过 100000000。

第三行包含一个整数 k,k < n。

输出:

从大到小输出前 k 大的数,每个数一行。

样例输入:

10

4 5 6 9 8 7 1 2 3 0

5

样例输出:

9

8

7

6

5

3. 区间合并【2.3 基本算法之分治 7620】

给定 n 个闭区间 [ai; bi],其中 i=1,2,…,n。任意两个相邻或相交的闭区间可以合并为一个闭区间。例如,[1;2] 和 [2;3] 可以合并为 [1;3],[1;3] 和 [2;4] 可以合并为 [1;

4],但是[1;2] 和 [3;4] 不可以合并。

我们的任务是判断这些区间是否可以最终合并为一个闭区间,如果可以,将这个闭区间输出,否则输出 no。

输入:

第一行为一个整数 n,3 ≤ n ≤ 50000。表示输入区间的数量。

之后 n 行,在第 i 行上(1 ≤ i ≤ n),为两个整数 ai 和 bi ,整数之间用一个空格分隔,表示区间 [ai; bi](其中 1 ≤ ai ≤ bi ≤ 10000)。

输出:

输出一行,如果这些区间最终可以合并为一个闭区间,输出这个闭区间的左右边界,用单个空格隔开;否则输出 no。

样例输入:

5

5 6

1 5

10 10

6 9

8 10

样例输出:

1 10

4. 求排列的逆序数【2.3 基本算法之分治 7622】

在 Internet 上的搜索引擎经常需要对信息进行比较,比如可以通过某个人对一些事物的排名来估计他(或她)对各种不同信息的兴趣,从而实现个性化的服务。

对于不同的排名结果可以用逆序来评价它们之间的差异。考虑 1,2,…,n 的排列 i1,i2,…,in,如果其中存在 j,k,满足 j<k 且 $i_j>i_k$,那么就称(i_j,i_k)是这个排列的一个逆序。

一个排列含有逆序的个数称为这个排列的逆序数。例如排列 263451 含有 8 个逆序(2,1),(6,3),(6,4),(6,5),(6,1),(3,1),(4,1),(5,1),因此该排列的逆序数就是 8。显然,由 1,2,…,n 构成的所有 n! 个排列中,最小的逆序数是 0,对应的排列就是 1,2,…,n;最大的逆序数是 n(n−1)/2,对应的排列就是 n,(n−1),…,2,1。逆序数越大的排列与原始排列的差异度就越大。

现给定 1,2,…,n 的一个排列,求它的逆序数。

输入:

第一行是一个整数 n,表示该排列有 n 个数(n <=100000)。

第二行是 n 个不同的正整数,之间以空格隔开,表示该排列。

输出:

输出该排列的逆序数。

样例输入:

6

2 6 3 4 5 1

样例输出:

8

5. 一元三次方程求解【2.3 基本算法之分治 7891】

有形如:$ax^3+bx^2+cx+d=0$　这样的一个一元三次方程。

给出该方程中各项的系数(a,b,c,d均为实数),并约定该方程存在三个不同实根(根的范围在-100至100之间),且根与根之差的绝对值>=1。要求由小到大依次在同一行输出这三个实根(根与根之间留有空格),并精确到小数点后2位。

输入:

一行,包含四个实数a,b,c,d,相邻两个数之间用单个空格隔开。

输出:

一行,包含三个实数,为该方程的三个实根,按从小到大顺序排列,相邻两个数之间用单个空格隔开,精确到小数点后2位。

样例输入:

1.0 -5.0 -4.0 20.0

样例输出:

-2.00 2.00 5.00

6.统计数字【2.3基本算法之分治7909】Noip2007提高组第1题

某次科研调查时得到了n个自然数,每个数均不超过1500000000(1.5*10^9)。已知不相同的数不超过10000个,现在需要统计这些自然数各自出现的次数,并按照自然数从小到大的顺序输出统计结果。

输入:

第一行是整数n,表示自然数的个数;

第2~n+1每行一个自然数。

40%的数据满足:1<=n<=1000;

80%的数据满足:1<=n<=50000;

100%的数据满足:1<=n<=200000,每个数均不超过1500 000 000(1.5*10^9)。

输出:

包含m行(m为n个自然数中不相同数的个数),按照自然数从小到大的顺序输出。每行输出两个整数,分别是自然数和该数出现的次数,其间用一个空格隔开。

样例输入:

```
8
2
4
2
4
5
100
2
100
```

样例输出:

```
2 3
4 2
5 1
100 2
```

7.查找最接近的元素【1.11编程基础之二分查找01】

在一个非降序列中,查找与给定值最接近的元素。

输入：

第一行包含一个整数 n，为非降序列长度。1 <＝n <＝100000。

第二行包含 n 个整数，为非降序列各元素。所有元素的大小均在 0～1,000,000,000 之间。

第三行包含一个整数 m，为要询问的给定值个数。1 <＝m <＝10000。

接下来 m 行，每行一个整数，为要询问最接近元素的给定值。所有给定值的大小均在 0～1,000,000,000 之间。

输出：

m 行，每行一个整数，为最接近相应给定值的元素值，保持输入顺序。若有多个值满足条件，输出最小的一个。

样例输入：

3

2 5 8

2

10

5

样例输出：

8

5

8. 二分法求函数的零点【1.11 编程基础之二分查找 02】

有函数：$f(x)=x^5-15*x^4+85*x^3-225*x^2+274*x-121$

已知 f(1.5)＞0，f(2.4)＜0 且方程 f(x)＝0 在区间[1.5,2.4] 有且只有一个根，请用二分法求出该根。

输入：

无。

输出：

该方程在区间[1.5,2.4]中的根。要求四舍五入到小数点后 6 位。

样例输入：

无

样例输出：

不提供

9. 网线主管【1.11 编程基础之二分查找 04】

仙境的居民们决定举办一场程序设计区域赛。裁判委员会完全由自愿组成，他们承诺要组织一次史上最公正的比赛。他们决定将选手的电脑用星形拓扑结构连接在一起，即将它们全部连到一个单一的中心服务器。为了组织这个完全公正的比赛，裁判委员会主席提出要将所有选手的电脑等距离地围绕在服务器周围放置。

为购买网线，裁判委员会联系了当地的一个网络解决方案提供商，要求能够提供一定数量的等长网线。裁判委员会希望网线越长越好，这样选手们之间的距离可以尽可能远一些。

该公司的网线主管承接了这个任务。他知道库存中每条网线的长度（精确到厘米），并且只要告诉他所需的网线长度（精确到厘米），他都能够完成对网线的切割工作。但是，这次，所需的网线长度并不知道，这让网线主管不知所措。

你需要编写一个程序,帮助网线主管确定一个最长的网线长度,并且按此长度对库存中的网线进行切割,能够得到指定数量的网线。

输入:

第一行包含两个整数 N 和 K,以单个空格隔开。N(1 <=N <=10000)是库存中的网线数,K(1 <=K <=10000)是需要的网线数量。

接下来 N 行,每行一个数,为库存中每条网线的长度(单位:米)。所有网线的长度至少 1m,至多 100km。输入中的所有长度都精确到厘米,即保留到小数点后两位。

输出:

网线主管能够从库存的网线中切出指定数量的网线的最长长度(单位:米)。必须精确到厘米,即保留到小数点后两位。

若无法得到长度至少为 1cm 的指定数量的网线,则必须输出"0.00"(不包含引号)。

样例输入:

```
4 11
8.02
7.43
4.57
5.39
```

样例输出:

```
2.00
```

10. 月度开销【1.11 编程基础之二分查找 06】

农夫约翰是一个精明的会计师。他意识到自己可能没有足够的钱来维持农场的运转了。他计算出并记录下了接下来 N (1 ≤ N ≤ 100,000) 天里每天需要的开销。

约翰打算为连续的 M (1 ≤ M ≤ N) 个财政周期创建预算案,他把一个财政周期命名为 fajo 月。每个 fajo 月包含一天或连续的多天,每天被恰好包含在一个 fajo 月里。

约翰的目标是合理安排每个 fajo 月包含的天数,使得开销最多的 fajo 月的开销尽可能少。

输入:

第一行包含两个整数 N,M,用单个空格隔开。

接下来 N 行,每行包含一个 1 到 10000 之间的整数,按顺序给出接下来 N 天里每天的开销。

输出:

一个整数,即最大月度开销的最小值。

样例输入:

```
7 5
100
400
300
100
500
101
400
```

样例输出：

500

提示：

若约翰将前两天作为一个月，第三、四两天作为一个月，最后三天作为一个月，则最大月度开销为500。其他任何分配方案都会比这个值更大。

11. 和为给定数【1.11 编程基础之二分查找 07】

给出若干个整数，询问其中是否有一对数的和等于给定的数。

输入：

第一行是整数 n(0 < n <=100,000)，表示有 n 个整数。

第二行是 n 个整数。整数的范围是在 0 到 10^8 之间。

第三行是一个整数 m(0 <=m <=2^30)，表示需要得到的和。

输出：

若存在和为 m 的数对，输出两个整数，小的在前，大的在后，中间用单个空格隔开。若有多个数对满足条件，选择数对中较小的数更小的。若找不到符合要求的数对，输出一行 No。

样例输入：

4

2 5 1 4

6

样例输出：

1 5

12. 不重复地输出数【1.11 编程基础之二分查找 08】

输入 n 个数，从小到大将它们输出，重复的数只输出一次。保证不同的数不超过500个。

输入：

第一行是一个整数 n。1 <=n <=100000。

之后 n 行，每行一个整数。整数大小在 int 范围内。

输出：

一行，从小到大不重复地输出这些数，相邻两个数之间用单个空格隔开。

样例输入：

5

2 4 4 5 1

样例输出：

1 2 4 5

13. 膨胀的木棍【1.11 编程基础之二分查找 09】

当长度为 L 的一根细木棍的温度升高 n 度，它会膨胀到新的长度 L'=(1+n*C)*L，其中 C 是热膨胀系数。

当一根细木棍被嵌在两堵墙之间被加热，它将膨胀形成弓形的弧，而这个弓形的弦恰好是未加热前木棍的原始位置。

你的任务是计算木棍中心的偏移距离。

输入：

三个非负实数:木棍初始长度(单位:毫米),温度变化(单位:度),以及材料的热膨胀系数。

保证木棍不会膨胀到超过原始长度的 1.5 倍。

输出:

木棍中心的偏移距离(单位:毫米),保留到小数点后第三位。

样例输入:

1000 100 0.0001

样例输出:

61.329

14. 河中跳房子【1.11 编程基础之二分查找 10】

每年奶牛们都要举办各种特殊版本的跳房子比赛,包括在河里从一个岩石跳到另一个岩石。这项激动人心的活动在一条长长的笔直河道中进行,在起点和离起点 L 远 ($1 \leqslant L \leqslant 1,000,000,000$) 的终点处均有一个岩石。在起点和终点之间,有 N ($0 \leqslant N \leqslant 50,000$) 个岩石,每个岩石与起点的距离分别为 Di ($0 < Di < L$)。

在比赛过程中,奶牛轮流从起点出发,尝试到达终点,每一步只能从一个岩石跳到另一个岩石。当然,实力不济的奶牛是没有办法完成目标的。

农夫约翰为他的奶牛们感到自豪并且年年都观看了这项比赛。但随着时间的推移,看着其他农夫的胆小奶牛们在相距很近的岩石之间缓慢前行,他感到非常厌烦。他计划移走一些岩石,使得从起点到终点的过程中,最短的跳跃距离最长。他可以移走除起点和终点外的至多 M ($0 \leqslant M \leqslant N$) 个岩石。

请帮助约翰确定移走这些岩石后,最长可能的最短跳跃距离是多少?

输入:

第一行包含三个整数 L, N, M,相邻两个整数之间用单个空格隔开。

接下来 N 行,每行一个整数,表示每个岩石与起点的距离。岩石按与起点距离从近到远给出,且不会有两个岩石出现在同一个位置。

输出:

一个整数,最长可能的最短跳跃距离。

样例输入:

25 5 2

2

11

14

17

21

样例输出:

4

提示:

在移除位于 2 和 14 的两个岩石之后,最短跳跃距离为 4(从 17 到 21 或从 21 到 25)。

第八章　广度优先搜索算法

一、广度优先搜索的过程

广度优先搜索算法(又称宽度优先搜索)是最简便的图的搜索算法之一,这一算法也是很多重要的图的算法的原型。Dijkstra 单源最短路径算法和 Prim 最小生成树算法都采用了和宽度优先搜索类似的思想。

广度优先算法的核心思想是:从初始节点开始,应用算符生成第一层节点,检查目标节点是否在这些后继节点中,若没有,再用产生式规则将所有第一层的节点逐一扩展,得到第二层节点,并逐一检查第二层节点中是否包含目标节点,若没有,再用算符逐一扩展第二层的所有节点…,如此依次扩展,检查下去,直到发现目标节点为止。即

1. 从图中的某一顶点 v0 开始,先访问 v0;
2. 访问所有与 v0 相邻接的顶点 v1,v2,…,vt;
3. 依次访问与 v1,v2,…,vt 相邻接的所有未曾访问过的顶点;
4. 循此以往,直至所有的顶点都被访问过为止。

这种搜索的次序体现沿层次向横向扩展的趋势,所以称之为广度优先搜索。

二、广度优先搜索算法描述

```
int bfs()
{
初始化,初始状态存入队列;
队列首指针 head=0; 尾指针 tail=1;
do
{
  指针 head 后移一位,指向待扩展结点;
  for (int i=1;i<=max;++i)              //max 为产生子结点的规则数
    {
    if (子结点符合条件)
      {
        tail 指针增 1,把新结点存入列尾;
        if (新结点与原已产生结点重复) 删去该结点(取消入队,tail 减 1);
          else
              if (新结点是目标结点) 输出并退出;
```

```
        }
    }
  }while(head<tail);                    //队列为空
}
```

三、广度优先搜索注意事项

1. 每生成一个子结点，就要提供指向它们父亲结点的指针。当解出现时候，通过逆向跟踪，找到从根结点到目标结点的一条路径。当然不要求输出路径，就没必要记父亲。

2. 生成的结点要与前面所有已经产生结点比较，以免出现重复结点，浪费时间和空间，还有可能陷入死循环。

3. 如果目标结点的深度与“费用”(如：路径长度)成正比，那么，找到的第一个解即为最优解，这时，搜索速度比深度搜索要快些，在求最优解时往往采用广度优先搜索；如果结点的“费用”不与深度成正比时，第一次找到的解不一定是最优解。

4. 广度优先搜索的效率还有赖于目标结点所在位置情况，如果目标结点深度处于较深层时，需搜索的结点数基本上以指数增长。

下面我们看看怎样用广度优先搜索来解决八数码问题。

图 8－1 给出广度优先搜索应用于八数码难题时所生成的搜索树。搜索树上的所有结点都标记它们所对应的状态，每个结点旁边的数字表示结点扩展的顺序。粗线条路径表明求得的一个解。从图中可以看出，扩展第 26 个结点，总共生成 46 个结点之后，才求得这个解。此外，直接观察此图表明，不存在有更短走步序列的解。

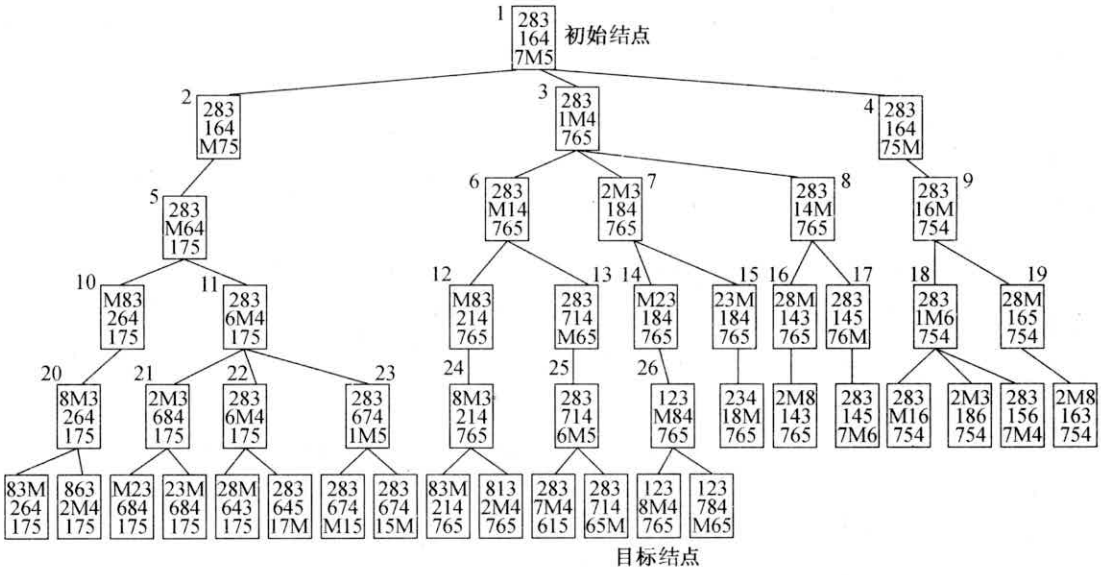


图 8－1 广度优先搜索图

例 8.1 图 8－2 表示的是从城市 A 到城市 H 的交通图。从图中可以看出，从城市 A 到城市 H 要经过若干个城市。现要找出一条经过城市最少的一条路线。

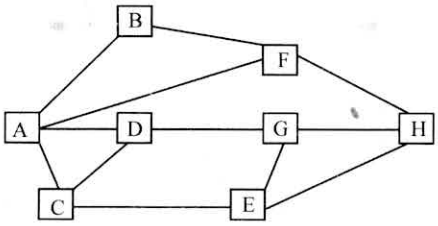


图 8－2

【算法分析】

看到这图很容易想到用邻接矩阵来表示，0 表示能走，1 表示不能走。如下表：。

	A	B	C	D	E	F	G	H
A	1	0	0	0	1	0	1	1
B	0	1	1	1	1	0	1	1
C	0	1	1	0	0	1	1	1
D	0	1	0	1	1	1	0	1
E	1	1	0	1	1	1	0	0
F	0	0	1	1	1	1	1	0
G	1	1	1	0	0	1	1	0
H	1	1	1	1	0	0	0	1

首先想到的是用队列的思想。a 数组是存储扩展结点的队列，a[i]记录经过的城市，b[i]记录前趋城市，这样就可以倒推出最短线路。具体过程如下：

(1)将城市 A 入队，队首为 0，队尾为 1。

(2)将队首所指的城市所有可直通的城市入队(如果这个城市在队列中出现过就不入队，可用一布尔数组 s[i]来判断)，将入队城市的前趋城市保存在 b[i]中。然后将队首加 1，得到新的队首城市。重复以上步骤，直到搜到城市 H 时，搜索结束。利用 b[i]可倒推出最少城市线路。

【参考程序】

```
#include<iostream>
#include<cstring>
using namespace std;
int  ju[9][9]={{0,0,0,0,0,0,0,0,0},
               {0,1,0,0,0,1,0,1,1},
               {0,0,1,1,1,1,0,1,1},
               {0,0,1,1,0,0,1,1,1},
               {0,0,1,0,1,1,1,0,1},
               {0,1,1,0,1,1,1,0,0},
               {0,0,0,1,1,1,1,1,0},
               {0,1,1,1,0,0,1,1,0},
               {0,1,1,1,1,0,0,0,1}};
int a[101],b[101];
bool s[9];                                    //初始化
```

```
int out(int d)                                    //输出过程
{
  cout<<char(a[d]+64);
  while (b[d])
    {
      d=b[d];
      cout<<"--"<<char(a[d]+64);
    }
cout<<endl;
}
void doit()
{
  int head,tail,i;
  head=0;tail=1;                                  //队首为0、队尾为1
  a[1]=1;                                         //记录经过的城市
  b[1]=0;                                         //记录前趋城市
  s[1]=1;                                         //表示该城市已经到过
  do                                              //步骤2
    {
      head++;                                     //队首加一,出队
      for (i=1;i<=8;i++)                          //搜索可直通的城市
      if ((ju[a[head]][i]==0)&&(s[i]==0))         //判断城市是否走过
        {
          tail++;                                 //队尾加一,入队
          a[tail]=i;
          b[tail]=head;
          s[i]=1;
          if (i==8)
            {
              out(tail);head=tail;break;          //第一次搜到H城市时路线最短
            }
          }
    }while (head<tail);
}
int main()                                        //主程序
{
  memset(s,false,sizeof(s));
  doit();                                         //进行bfs操作
  return 0;
}
```

输出：

H——F——A

例 8.2　一矩形阵列由数字 0 到 9 组成，数字 1 到 9 代表细胞，细胞的定义为沿细胞数字上下左右还是细胞数字则为同一细胞，求给定矩形阵列的细胞个数。如：

阵列　4 10

0234500067

1034560500

2045600671

0000000089

有 4 个细胞。

【算法分析】

(1)从文件中读入 m * n 矩阵阵列，将其转换为 boolean 矩阵存入 bz 数组中；

(2)沿 bz 数组矩阵从上到下、从左到右，找到遇到的第一个细胞；

(3)将细胞的位置入队 h，并沿其上、下、左、右四个方向上的细胞位置入队，入队后的位置 bz 数组置为 false；

(4)将 h 队的队头出队，沿其上、下、左、右四个方向上的细胞位置入队，入队后的位置 bz 数组置为 false；

(5)重复(4)，直至 h 队空为止，则此时找出了一个细胞；

(6)重复(2)，直至矩阵找不到细胞；

(7)输出找到的细胞数。

【参考程序】

```
#include<cstdio>
using namespace std;
int dx[4]={-1,0,1,0},
    dy[4]={0,1,0,-1};
int bz[100][100],num=0,n,m;
void doit(int p,int q)
{
  int x,y,t,w,i;
  int h[1000][2];
  num++;bz[p][q]=0;
  t=0;w=1;h[1][1]=p;h[1][2]=q;      //遇到的第一个细胞入队
  do
    {
      t++;                          //队头指针加 1
      for (i=0;i<=3;i++)            //沿细胞的上下左右四个方向搜索细胞
        {
          x=h[t][1]+dx[i];y=h[t][2]+dy[i];
          if ((x>=0)&&(x<m)&&(y>=0)&&(y<n)&&(bz[x][y]))
                                        //判断该点是否可以入队
```

```
            {
              w++;
              h[w][1]=x;
              h[w][2]=y;
              bz[x][y]=0;
            }                                    //本方向搜索到细胞就入队
        }
    }while (t<w);                                //直至队空为止
}
int main()
{
  int i,j;
  char s[100],ch;
  scanf("%d%d\n",&m,&n);
  for (i=0; i<=m-1;i++)
    for (j=0;j<=n-1;j++)
      bz[i][j]=1;                                //初始化
  for (i=0;i<=m-1;i++)
    {
      gets(s);
      for (j=0;j<=n-1;j++)
       if (s[j]=='0') bz[i][j]=0;
    }
  for (i=0;i<=m-1;i++)
   for (j=0;j<=n-1;j++)
    if (bz[i][j])
     doit(i,j);                                  //在矩阵中寻找细胞
  printf(" NUMBER of cells=%d ",num);
  return 0;
}
```

例 8.3 最少步数

【问题描述】

在各种棋中,棋子的走法总是一定的,如中国象棋中马走“日”。有一位小学生就想如果马能有两种走法将增加其趣味性,因此,他规定马既能按“日”走,也能如象一样走“田”字。他的同桌平时喜欢下围棋,知道这件事后觉得很有趣,就想试一试,在一个(100 * 100)的围棋盘上任选两点 A、B,A 点放上黑子,B 点放上白子,代表两匹马。棋子可以按“日”字走,也可以按“田”字走,俩人一个走黑马,一个走白马。谁用最少的步数走到左上角坐标为(1,1)的点时,谁获胜。现在他请你帮忙,给你 A、B 两点的坐标,想知道两个位置到(1,1)点可能的最少步数。

【输入样例】

12 16

18 10

【输出样例】

8

9

【算法分析】

由于A、B两点是随机输入的，因此无法找到计算最少步数的数学规律，只能通过广度优先搜索的办法求解。

(1)确定出发点

从(x,y)出发通过一次广度优先搜索，可以找到从(x,y)至棋盘上所有可达点的最少步数。而问题中要求的是黑马所在的(x1,y1)和白马所在(x2,y2)到达(1,1)目标点的最少步数。虽然两条路径的起点不一样，但是它们的终点却是一样的。如果我们将终点(1,1)作为起点，这样只需要一次广度优先搜索便可以得到(x1,y1)和(x2,y2)到达(1,1)的最少步数。

(2)数据结构

设queue——队列，存储从(1,1)可达的点(queue[k][1..2])以及到达该点所需要的最少步数(queue[k][3])(0≤k≤192+1)。队列的首指针为head，尾指针为tail。初始时，queue中只有一个元素为(1,1)，最少步数为0。

s——记录(1,1)到每点所需要的最少步数。显然，问题的答案是s[x1][y1]和s[x2][y2]。初始时，s[1][1]为0，除此之外的所有元素值设为−1。

dx、dy——移动后的位置增量数组。马有12种不同的扩展方向：

马走“日”：

(x−2,y−1)(x−1,y−2)(x−2,y+1)(x−1,y+2)(x+2,y−1)(x+1,y−2)(x+2,y+1)(x+1,y+2)

马走“田”：

(x−2,y−2)(x−2,y+2)(x+2,y−2)(x+2,y+2)

我们将i方向上的位置增量存入常量数组dx[i]、dy[i]中(0≤i≤11)：

```
int dx[12]={-2,-2,-1,1,2,2,2,2,1,-1,-2,-2},
    dy[12]={-1,-2,-2,-2,-2,-1,1,2,2,2,2,1};
```

(3)约束条件

(1)不能越出界外。由于马的所有可能的落脚点s均在s的范围内，因此一旦马越出界外，就将其s值赋为0，表示“已经扩展过，且(1,1)到达其最少需要0步”。这看上去是荒谬的，但可以简单而有效地避免马再次落入这些界外点。

(2)该点在以前的扩展中没有到达过。如果曾经到达过，则根据广度优先搜索的原理，先前到达该点所需的步数一定小于当前步数，因此完全没有必要再扩展下去。

由此得出，马的跳后位置(x,y)是否可以入队的约束条件是s[x][y]<0。

(4)算法流程

```
#include <cstdlib>
#include <cstring>
#include <iostream>
```

```
using namespace std;
int dx[12]={-2,-2,-1,1,2,2,2,2,1,-1,-2,-2},
    dy[12]={-1,-2,-2,-2,-2,-1,1,2,2,2,2,1};
int main()
{
  int s[101][101],que[10000][4]={0},x1,y1,x2,y2;
  memset(s,0xff,sizeof(s));                    //s 数组的初始化
  int head=1,tail=1;                           //初始位置入队
  que[1][1]=1;que[1][2]=1;que[1][3]=0;
  cin>>x1>>y1>>x2>>y2;                         //读入黑马和白马的出发位置
  while(head<=tail)                            //若队列非空,则扩展队首结点
  {
    for(int d=0;d<=11;d++)                     //枚举 12 个扩展方向
    {
      int x=que[head][1]+dx[d];                //计算马按 d 方向跳跃后的位置
      int y=que[head][2]+dy[d];
      if(x>0&&y>0)
        if(s[x][y]==-1)                        //若(x,y)满足约束条件
        {
          s[x][y]=que[head][3]+1;              //计算(1,1)到(x,y)的最少步数
          tail++;                              //(1,1)至(x,y)的最少步数入队
          que[tail][1]=x;
          que[tail][2]=y;
          que[tail][3]=s[x][y];
          if(s[x1][y1]>0&&s[x2][y2]>0)  //输出问题的解
          {
            cout<<s[x1][y1]<<endl;
            cout<<s[x2][y2]<<endl;
            system("pause");
            return 0;
          }
        }
    }

    head++;
  }
}
```

例 8.4 迷宫问题

如图 8-3 所示,给出一个 n * m 的迷宫图和一个入口、一个出口。

编一个程序,打印一条从迷宫入口到出口的路径。这里黑色方块的单元表示走不通(用

-1 表示),白色方块的单元表示可以走(用 0 表示)。只能往上、下、左、右四个方向走。如果无路则输出“no way.”。

入口→	0	-1	0	0	0	0	0	0	-1	
	0	0	0	0	-1	0	0	0	-1	
	-1	0	0	0	0	0	-1	-1	-1	
	0	0	-1	-1	0	0	0	0	0	→ 出口
	0	0	0	0	0	0	0	-1	-1	

图 8-3

【算法分析】

只要输出一条路径即可,所以是一个经典的回溯算法问题,本例给出了回溯(深搜)程序和广搜程序。实现见参考程序。

【深搜参考程序】

```
#include <iostream>
using namespace std;
int n,m,desx,desy,soux,souy,totstep,a[51],b[51],map[51][51];
bool f;
int move(int x, int y,int step)
{
  map[x][y]=step;                          //走一步,作标记,把步数记下来
  a[step]=x;  b[step]=y;                   //记路径
  if ((x==desx)&&(y==desy))
  {
    f=1;
    totstep=step;
  }
  else
  {
    if ((y!=m)&&(map[x][y+1]==0))          move(x,y+1,step+1); //向右
    if ((!f)&&(x!=n)&&(map[x+1][y]==0))    move(x+1,y,step+1);  //往下
    if ((!f)&&(y!=1)&&(map[x][y-1]==0))    move(x,y-1,step+1);  //往左
    if ((!f)&&(x!=1)&&(map[x-1][y]==0))    move(x-1,y,step+1);  //往上
  }
}
int main()
{
  int i,j;
  cin>>n>>m;                               //n 行 m 列的迷宫
  for (i=1;i<=n;i++)                       //读入迷宫,0 表示通,-1 表示不通
    for (j=1;j<=m;j++)
```

```
      cin>>map[i][j];
    cout<<" input the enter:";
    cin>>soux>>souy;                    //入口
    cout<<" input the exit:";
    cin>>desx>>desy;                    //出口
    f=0;                                //f=0 表示无解;f=1 表示找到了一个解
    move(soux,souy,1);
    if (f)
    {
      for (i=1;i<=totstep;i++)          //输出直迷宫的路径
      cout<<a[i]<<","<<b[i]<<endl;
    }
  else  cout<<" no way."<<endl;
  return 0;
}
```

【广搜参考程序】

```
#include <iostream>
using namespace std;
int u[5]={0,0,1,0,-1},
    w[5]={0,1,0,-1,0};
int n,m,i,j,desx,desy,soux,souy,head,tail,x,y,a[51],b[51],pre[51],map[51][51];
bool f;
int print(int d)
{
    if (pre[d]!=0) print (pre[d]);      //递归输出路径
    cout<<a[d]<<","<<b[d]<<endl;
}
int main()
{
  int i,j;
  cin>>n>>m;                            //n 行 m 列的迷宫
  for (i=1;i<=n;i++)                    //读入迷宫,0 表示通,-1 表示不通
    for (j=1;j<=m;j++)
      cin>>map[i][j];
  cout<<" input the enter:";
  cin>>soux>>souy;                      //入口
  cout<<" input the exit:";
  cin>>desx>>desy;                      //出口
  head=0;
  tail=1;
```

```
    f=0;
    map[soux][souy]=-1;
    a[tail]=soux;  b[tail]=souy; pre[tail]=0;
    while (head!=tail)                    //队列不为空
    {
      head++;
      for (i=1;i<=4;i++)                  //4 个方向
      {
        x=a[head]+u[i];  y=b[head]+w[i];
        if ((x>0)&&(x<=n)&&(y>0)&&(y<=m)&&(map[x][y]==0))
        {                                 //本方向上可以走
          tail++;
          a[tail]=x;  b[tail]=y;  pre[tail]=head;
          map[x][y]=-1;
          if ((x==desx)&&(y==desy)) //扩展出的结点为目标结点
          {
            f=1;
            print(tail);
            break;
          }
        }
      }
      if (f) break;
    }
    if (!f) cout<<" no way. "<<endl;
    return 0;
}
```

输入 1：	输出 1：	输入 2：	输出 2：
8　5	2,1	8　5	no way.
−1 −1 −1 −1 −1	2,2	−1 −1 −1 −1 −1	
0　0　0　0 −1	2,3	0　0　0　0 −1	
−1 −1 −1　0 −1	2,4	−1 −1 −1　0 −1	
−1　0　0　0 −1	3,4	−1　0　0　0 −1	
−1　0　0　1 −1	4,4	−1　0　0 −1 −1	
−1　0　0　0 −1	4,3	−1　0　0　0 −1	
−1 −1 −1　0 −1	5,3	−1 −1 −1 −1 −1	
−1　0　0　0 −1	6,3	−1　0　0　0 −1	
2　1	6,4	2　1	
8　4	7,4	8　4	
	8,4		

【上机练习】

1. Dungeon Master【2.5 基本算法之搜索 1253】

题目大意:这题是一个三维的迷宫题目,其中用'.'表示空地,'#'表示障碍物,'S'表示起点,'E'表示终点,求从起点到终点的最小移动次数,解法和二维的类似,只是在行动时除了东南西北移动外还多了上下。可以上下左右前后移动,每次都只能移到相邻的空位,每次需要花费一分钟,求从起点到终点最少要多久。

样例输入:

```
3 4 5
S . . . .
. # # # .
. # # . .
# # # . #
# # # # #
# # # # #
# # . # #
# # . . .
# # # # #
# # # # #
# . # # #
# # # # E
1 3 3
S # #
# E #
# # #
0 0 0
```

样例输出:

```
Escaped in 11 minute(s).
Trapped!
```

2. Lake Counting【2.5 基本算法之搜索 1388】

题意:有一块 N * M 的土地,雨后积起了水,有水标记为'W',干燥为'.'。八连通的积水被认为是连接在一起的。请求出院子里共有多少水洼?

样例输入:

```
 10  12
W . . . . . . . . WW .
. WWW . . . . . WWW
. . . . WW . . . WW .
```

. . . . . WW.
. . . . . W. .
. . W. . . . . W. .
. W. W. . . . . WW.
W. W. W. . . . . W.
. W. W. . . . . W.
. . W. . . . . W.

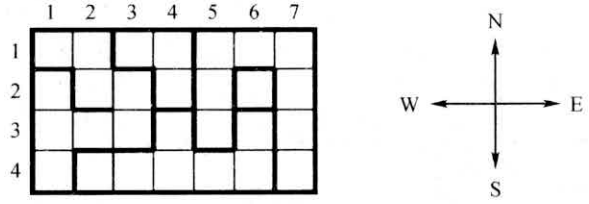
样例输出：

3

3. The Castle【2.5 基本算法之搜索 166 和 1817】

问题描述：

一座城堡被分成 m * n 个方块(m≤50,n≤50),每个方块可有 0～4 堵墙(0 表示无墙)。下面示出了建筑平面图：



图中的加粗黑线代表墙。几个连通的方块组成房间,房间与房间之间一定是用黑线(墙)隔开的。

现在要求你编一个程序,解决以下 2 个问题：

1. 该城堡中有多少个房间；

2. 最大的房间有多大；

输入格式：

平面图用一个数字表示一个方块(第 1 个房间用二进制 1011 表示,0 表示无东墙,用十进制 11 表示)。

●第一行一个整数 m(m≤50),表示房子南北方向的长度。

第二行一个整数 n(n≤50),表示房子东西方向的长度。

●后面的 m 行,每行有 n 个整数,每个整数都表示平面图对应位置的方块的特征。每个方块中墙的特征由数字 P 来描述(0≤P≤15)。数字 P 是下面的可能取的数字之和：

1(西墙 west)

2(北墙 north)

4(东墙 east)

8(南墙 south)

室内的墙被定义两次：例如方块 (1, 1)中的南墙也被位于其南面的方块(2,1)定义了一次。

●建筑中至少有两个房间。

输出格式：

第 1 行:一个整数,表示房间总数；

第 2 行:一个整数,表示最大房间的面积(方块数)；

输入样例：

```
4
7
11  6  11  6  3  10  6
 7  9  6  13  5  15  5
 1  10  12  7  13  7  5
13  11  10  8  10  12  13
```

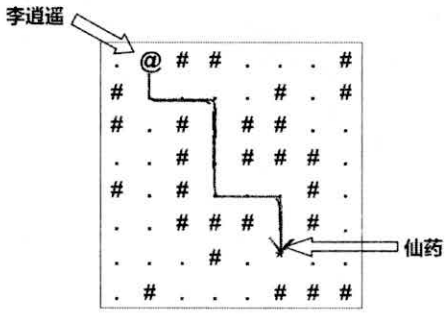
输出样例：

```
5
9
```

4. 仙岛求药【2.5 基本算法之搜索 2727】

少年李逍遥的婶婶病了，王小虎介绍他去一趟仙灵岛，向仙女姐姐要仙丹救婶婶。叛逆但孝顺的李逍遥闯进了仙灵岛，克服了千险万难来到岛的中心，发现仙药摆在了迷阵的深处。迷阵由 M×N 个方格组成，有的方格内有可以瞬秒李逍遥的怪物，而有的方格内则是安全。现在李逍遥想尽快找到仙药，显然他应避开有怪物的方格，并经过最少的方格，而且那里会有神秘人物等待着他。现在要求你来帮助他实现这个目标。

下图显示了一个迷阵的样例及李逍遥找到仙药的路线。



输入：

输入有多组测试数据. 每组测试数据以两个非零整数 M 和 N 开始，两者均不大于 20。M 表示迷阵行数，N 表示迷阵列数。接下来有 M 行，每行包含 N 个字符，不同字符分别代表不同含义：

1)'@'：少年李逍遥所在的位置；

2)'.'：可以安全通行的方格；

3)'#'：有怪物的方格；

4)'*'：仙药所在位置。

当在一行中读入的是两个零时，表示输入结束。

输出：

对于每组测试数据，分别输出一行，该行包含李逍遥找到仙药需要穿过的最少的方格数目(计数包括初始位置的方块)。如果他不可能找到仙药，则输出－1。

样例输入：

```
8 8
```

```
. @ # # . . . #
# . . . . # . #
# . # . # # . .
. . # . # # # .
# . # . . . # .
. . # # # . # .
. . . # . * . .
. # . . . # # #
6 5
. * . # .
. # . . .
. . # # .
. . . . .
. # . . .
. . . . @
9 6
. # . . # .
. # . * . #
. # # # # .
. . # . . .
. . # . . .
. . # . . .
. . # . . .
# . @ . # #
. # . . # .
0 0
```

样例输出：

```
10
8
-1
```

5. 走迷宫【2.5 基本算法之搜索 2753】

一个迷宫由 R 行 C 列格子组成，有的格子里有障碍物，不能走；有的格子是空地，可以走。

给定一个迷宫，求从左上角走到右下角最少需要走多少步（数据保证一定能走到）。只能在水平方向或垂直方向走，不能斜着走。

输入：

第一行是两个整数，R 和 C，代表迷宫的长和宽。（1<=R,C<=40）

接下来是 R 行，每行 C 个字符，代表整个迷宫。

空地格子用'.'表示，有障碍物的格子用'#'表示。

迷宫左上角和右下角都是'.'。

输出：

输出从左上角走到右下角至少要经过多少步(即至少要经过多少个空地格子)。计算步数要包括起点和终点。

样例输入：

```
5 5
. . # # #
# . . . .
# . # . #
# . # . #
# . # . .
```

样例输出：

```
9
```

6. 抓住那头牛【2.5 基本算法之搜索 2971】

农夫知道一头牛的位置，想要抓住它。农夫和牛都位于数轴上，农夫起始位于点 N(0<=N<=100000)，牛位于点 K(0<=K<=100000)。农夫有两种移动方式：

1. 从 X 移动到 X−1 或 X+1，每次移动花费一分钟

2. 从 X 移动到 2 * X，每次移动花费一分钟

假设牛没有意识到农夫的行动，站在原地不动。农夫最少要花多少时间才能抓住牛？

输入：

两个整数，N 和 K

输出：

一个整数，农夫抓到牛所要花费的最小分钟数

样例输入：

```
5 17
```

样例输出：

```
4
```

7. 走出迷宫【2.5 基本算法之搜索 6264】

当你站在一个迷宫里的时候，往往会被错综复杂的道路弄得失去方向感，如果你能得到迷宫地图，事情就会变得非常简单。

假设你已经得到了一个 n * m 的迷宫的图纸，请你找出从起点到出口的最短路。

输入：

第一行是两个整数 n 和 m(1<=n,m<=100)，表示迷宫的行数和列数。

接下来 n 行，每行一个长为 m 的字符串，表示整个迷宫的布局。字符'.'表示空地，'#'表示墙，'S'表示起点，'T'表示出口。

输出：

输出从起点到出口最少需要走的步数。

样例输入：

```
3 3
S # T
```

. # .

. . .

样例输出：

6

8. 迷宫问题【2.5 基本算法之搜索 7084】

定义一个二维数组：

```
int maze[5][5]={
  0, 1, 0, 0, 0,
  0, 1, 0, 1, 0,
  0, 0, 0, 0, 0,
  0, 1, 1, 1, 0,
  0, 0, 0, 1, 0,
};
```

它表示一个迷宫，其中的 1 表示墙壁，0 表示可以走的路，只能横着走或竖着走，不能斜着走，要求编程序找出从左上角到右下角的最短路线。

输入：

一个 5 × 5 的二维数组，表示一个迷宫。数据保证有唯一解。

输出：

左上角到右下角的最短路径，格式如样例所示。

样例输入：

```
0 1 0 0 0
0 1 0 1 0
0 0 0 0 0
0 1 1 1 0
0 0 0 1 0
```

样例输出：

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

9. 献给阿尔吉侬的花束【2.5 基本算法之搜索 7218】

阿尔吉侬是一只聪明又慵懒的小白鼠，它最擅长的就是走各种各样的迷宫。今天它要挑战一个非常大的迷宫，研究员们为了鼓励阿尔吉侬尽快到达终点，就在终点放了一块阿尔吉侬最喜欢的奶酪。现在研究员们想知道，如果阿尔吉侬足够聪明，它最少需要多少时间就能吃到奶酪。

迷宫用一个R×C的字符矩阵来表示。字符S表示阿尔吉侬所在的位置,字符E表示奶酪所在的位置,字符#表示墙壁,字符.表示可以通行。阿尔吉侬在1个单位时间内可以从当前的位置走到它上下左右四个方向上的任意一个位置,但不能走出地图边界。

输入:

第一行是一个正整数T(1<=T<=10),表示一共有T组数据。

每一组数据的第一行包含了两个用空格分开的正整数R和C(2<=R,C<=200),表示地图是一个R×C的矩阵。

接下来的R行描述了地图的具体内容,每一行包含了C个字符。字符含义如题目描述中所述。保证有且仅有一个S和E。

输出:

对于每一组数据,输出阿尔吉侬吃到奶酪的最少单位时间。若阿尔吉侬无法吃到奶酪,则输出"oop!"(只输出引号里面的内容,不输出引号)。每组数据的输出结果占一行。

样例输入:

```
3
3 4
. S . .
# # # .
. . E .
3 4
. S . .
. E . .
. . . .
3 4
. S . .
# # # #
. . E .
```

样例输出:

```
5
1
oop!
```

10. Knight Moves【2.5 基本算法之搜索 917】

题意:输入n代表有个n*n的棋盘,输入开始位置的坐标和结束位置的坐标,问一个骑士朝棋盘的八个方向走马字步,从开始坐标到结束坐标可以经过多少步。

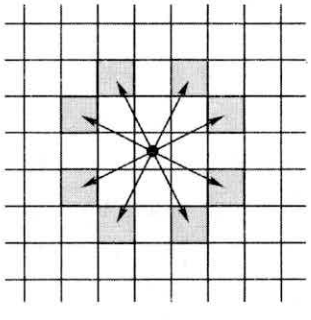


Figure 1:Possible knight moves on the board.

输入：

首先输入一个 n，表示测试样例的个数。

每个测试样例有三行。

第一行是棋盘的大小 l(4<=l<=300).

第二行和第三行分别表示马的起始位置和目标位置(0～l－1)

输出：

马移动的最小步数，起始位置和目标位置相同时输出 0。

样例输入：

```
3
8
0 0
7 0
100
0 0
30 50
10
1 1
1 1
```

样例输出：

```
5
28
0
```

第九章　动态规划

动态规划程序设计是对解最优化问题的一种途径、一种方法，而不是一种特殊算法。不像前面所述的那些搜索或数值计算那样，具有一个标准的数学表达式和明确清晰的解题方法。动态规划程序设计往往是针对一种最优化问题，由于各种问题的性质不同，确定最优解的条件也互不相同，因而动态规划的设计方法对不同的问题，有各具特色的解题方法，而不存在一种万能的动态规划算法，可以解决各类最优化问题。因此读者在学习时，除了要对基本概念和方法正确理解外，必须具体问题具体分析处理，以丰富的想象力去建立模型，用创造性的技巧去求解。我们也可以通过对若干有代表性的问题的动态规划算法进行分析、讨论，逐渐学会并掌握这一设计方法。

第一节　动态规划的基本模型

一、多阶段决策过程的最优化问题

在现实生活中，有一类活动的过程，由于它的特殊性，可将过程分成若干个互相联系的阶段，在它的每一阶段都需要作出决策，从而使整个过程达到最好的活动效果。当然，各个阶段决策的选取不是任意确定的，它依赖于当前面临的状态，又影响以后的发展，当各个阶段决策确定后，就组成一个决策序列，因而也就确定了整个过程的一条活动路线，这种把一个问题看作是一个前后关联具有链状结构的多阶段过程就称为多阶段决策过程，这种问题就称为多阶段决策问题。如图 9－1 所示：

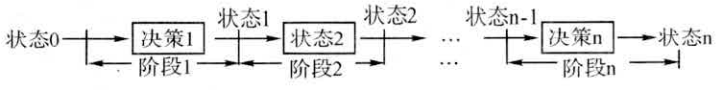


图 9－1

多阶段决策过程，是指这样的一类特殊的活动过程，问题可以按时间顺序分解成若干相互联系的阶段，在每一个阶段都要作出决策，全部过程的决策是一个决策序列。

例 9.1　最短路径问题。图 9－2 给出了一个地图，地图中的每个顶点代表一个城市，两个城市间的一条连线代表道路，连线上的数值代表道路的长度。现在想从城市 A 到达城市 E，怎样走路程最

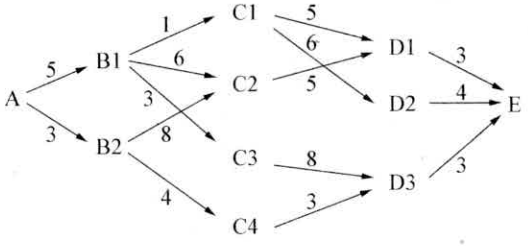


图 9－2

短？最短路程的长度是多少？

【算法分析】

把 A 到 E 的全过程分成四个阶段，用 K 表示阶段变量，第 1 阶段有一个初始状态 A，有两条可供选择的支路 A－B1、A－B2；第 2 阶段有两个初始状态 B1、B2，B1 有三条可供选择的支路，B2 有两条可供选择的支路…用 DK(X_I，$X+1_J$)表示在第 K 阶段由初始状态 X_I 到下阶段的初始状态 $X+1_J$ 的路径距离，FK(X_I)表示从第 K 阶段的 X_I 到终点 E 的最短距离，利用倒推的方法，求解 A 到 E 的最短距离。具体计算过程如下：

S1：K＝4 有

F4(D1)＝3，

F4(D2)＝4，

F4(D3)＝3；

S2：K＝3 有

F3(C1)＝MIN{ D3(C1,D1)＋F4(D1),D3(C1,D2)＋F4(D2)}

＝MIN{ 5＋3,6＋4 }＝8

F3(C2)＝D3(C2,D1)＋F4(D1)＝5＋3＝8

F3(C3)＝D3(C3,D3)＋F4(D3)＝8＋3＝11

F3(C4)＝D3(C4,D3)＋F4(D3)＝3＋3＝6

S3：K＝2 有

F2(B1)＝MIN{ D2(B1,C1)＋F3(C1),D2(B1,C2)＋F3(C2),

D2(B1,C3)＋F3(C3)}＝MIN{ 1＋8,6＋8,3＋11}＝9

F2(B2)＝MIN{ D2(B2,C2)＋F3(C2),D2(B2,C4)＋F3(C4)}

＝MIN{ 8＋8,4＋6 }＝10

S4：K＝1 有

F1(A)＝MIN{ D1(A,B1)＋F2(B1),D1(A,B2)＋F2(B2)}

＝MIN{ 5＋9,3＋10}＝13

因此由 A 点到 E 点的全过程最短路径为 A→B2→C4→D3→E；最短路程长度为 13。

从以上过程可以看出，每个阶段中，都求出本阶段的各个初始状态到终点 E 的最短距离，当逆序倒推到过程起点 A 时，便得到了全过程的最短路径和最短距离。

在上例的多阶段决策问题中，各个阶段采取的决策，一般来说是与阶段有关的，决策依赖于当前状态，又随即引起状态的转移，一个决策序列就是在变化的状态中产生出来的，故有“动态”的含义，我们称这种解决多阶段决策最优化的过程为动态规划程序设计方法。

二、动态规划的基本概念和基本模型构成

现在我们来介绍动态规划的基本概念。

1. 阶段和阶段变量

用动态规划求解一个问题时，需要将问题的全过程恰当地分成若干个相互联系的阶段，以便按一定的次序去求解。描述阶段的变量称为阶段变量，通常用 k 表示，阶段的划分一般是根据时间和空间的自然特征来划分，同时阶段的划分要便于把问题转化成多阶段决策过程，如例 9.1 中，可将其划分成 4 个阶段，即 k＝1,2,3,4。

2. 状态和状态变量

某一阶段的出发位置称为状态,通常一个阶段包含若干状态。一般地,状态可由变量来描述,用来描述状态的变量称为状态变量。如例 9.1 中,C3 是一个状态变量。

3. 决策、决策变量和决策允许集合

在对问题的处理中作出的每种选择性的行动就是决策。即从该阶段的每一个状态出发,通过一次选择性的行动转移至下一阶段的相应状态。一个实际问题可能要有多次决策和多个决策点,在每一个阶段的每一个状态中都需要有一次决策,决策也可以用变量来描述,称这种变量为决策变量。在实际问题中,决策变量的取值往往限制在某一个范围之内,此范围称为允许决策集合。如例 9.1 中,F3(C3)就是一个决策变量。

4. 策略和最优策略

所有阶段依次排列构成问题的全过程。全过程中各阶段决策变量所组成的有序总体称为策略。在实际问题中,从决策允许集合中找出最优效果的策略称为最优策略。

5. 状态转移方程

前一阶段的终点就是后一阶段的起点,对前一阶段的状态作出某种决策,产生后一阶段的状态,这种关系描述了由 k 阶段到 k+1 阶段状态的演变规律,称为状态转移方程。

三、最优化原理与无后效性原则

上面已经介绍了动态规划模型的基本组成,现在需要解决的问题是:什么样的"多阶段决策问题"才可以采用动态规划的方法求解。

一般来说,能够采用动态规划方法求解的问题,必须满足最优化原理和无后效性原则:

1. 动态规划的最优化原理

作为整个过程的最优策略具有:无论过去的状态和决策如何,对前面的决策所形成的状态而言,余下的诸决策必须构成最优策略的性质。也可以通俗地理解为子问题的局部最优将导致整个问题的全局最优,即问题具有最优子结构的性质,也就是说一个问题的最优解只取决于其子问题的最优解,而非最优解对问题的求解没有影响。在例 9.1 最短路径问题中,A 到 E 的最优路径上的任意一点到终点 E 的路径,也必然是该点到终点 E 的一条最优路径,即整体优化可以分解为若干个局部优化。

2. 动态规划的无后效性原则

所谓无后效性原则,指的是这样一种性质:某阶段的状态一旦确定,则此后过程的演变不再受此前各状态及决策的影响。也就是说,"未来与过去无关",当前的状态是此前历史的一个完整的总结,此前的历史只能通过当前的状态去影响过程未来的演变。在例 9.1 最短路径问题中,问题被划分成各个阶段之后,阶段 k 中的状态只能由阶段 k+1 中的状态通过状态转移方程得来,与其他状态没有关系,特别与未发生的状态没有关系,例如从 Ci 到 E 的最短路径,只与 Ci 的位置有关,它是由 Di 中的状态通过状态转移方程得来,与 E 状态,特别是 A 到 Ci 的路径选择无关,这就是无后效性。

由此可见,对于不能划分阶段的问题,不能运用动态规划来解;对于能划分阶段,但不符合最优化原理的,也不能用动态规划来解;既能划分阶段,又符合最优化原理的,但不具备无后效性原则,还是不能用动态规划来解;误用动态规划程序设计方法求解会导致错误的结果。

四、基本动态规划模型的应用

例 9.2　数字金字塔

观察下面的数字金字塔。写一个程序查找从最高点到底部任意处结束的路径，使路径经过数字的和最大。每一步可以从当前点走到左下方的点也可以到达右下方的点。

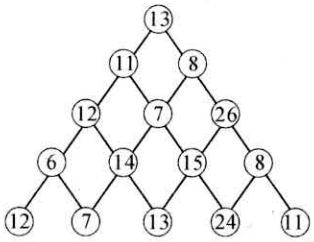


图 9－3

在上面的样例中，从 13 到 8 到 26 到 15 到 24 的路径产生了最大的和 86。

输入：

第一个行包含 R(1<= R<=1000)，表示行的数目。

后面每行为这个数字金字塔特定行包含的整数。

所有的被供应的整数是非负的且不大于 100。

输出：

单独的一行，包含那个可能得到的最大的和。

样例输入：

```
5                                    //数塔层数
13
11  8
12  7  26
6  14  15  8
12  7  13  24  11
```

样例输出：

```
86
```

方法一：搜索

问题要求的是从最高点按照规则走到最低点的路径的最大的权值和，路径起点终点固定，走法规则明确，可以考虑用搜索来解决。

定义递归函数 void Dfs(int x,int y,int Curr)，其中 x,y 表示当前已从(1,1)走到(x,y)，目前已走路径上的权值和为 Curr。

当 x=N 时，到达递归出口，如果 Curr 比 Ans 大，则把 Ans 更新为 Curr；否则向下一行两个位置行走，即递归执行 Dfs(x+1,y,Curr+A[x+1][y])和 Dfs(x+1,y+1,Curr+A[x+1][y+1])。

```
#include <iostream>
using namespace std;
const int MAXN = 1005;
```

```
int A[MAXN][MAXN],F[MAXN][MAXN],N,Ans;

void Dfs(int x,int y,int Curr)
{
    if (x==N)
    {
        if (Curr>Ans)Ans=Curr;
        return;
    }
    Dfs(x+1,y,Curr+A[x+1][y]);
    Dfs(x+1,y+1,Curr+A[x+1][y+1]);
}

int main()
{
    cin >> N;
    for(int i = 1;i <= N;i ++)
      for(int j = 1;j <= i;j ++)
    cin >> A[i][j];
    Ans =0;
    Dfs(1,1,A[1][1]);
    cout<<Ans<<endl;
    return 0;
}
```

该方法实际上是把所有路径都走了一遍,由于每一条路径都是由 N−1 步组成,每一步有“左”、“右”两种选择,因此路径总数为 2^{N-1},所以该方法的时间复杂度为 $O(2^{N-1})$,超时。

方法二:记忆化搜索

方法一之所以会超时,是因为进行了重复搜索,如样例中从(1,1)到(3,2)有“左右”和“右左”两种不同的路径,也就是说搜索过程中两次到达(3,2)这个位置,那么从(3,2)走到终点的每一条路径就被搜索了两次,我们完全可以在第一次搜索(3,2)到终点的路径时就记录下(3,2)到终点的最大权值和,下次再次来到(3,2)时就可以直接调用这个权值避免重复搜索。我们把这种方法称为记忆化搜索。

记忆化搜索需要对方法一中的搜索进行改装。由于需要记录从一个点开始到终点的路径的最大权值和,因此我们重新定义递归函数 Dfs。

定义 Dfs(x,y)表示从(x,y)出发到终点的路径的最大权值和,答案就是 Dfs(1,1)。计算 Dfs(x,y)时考虑第一步是向左还是向右,我们把所有路径分成两大类:

①第一步向左:那么从(x,y)出发到终点的这类路径就被分成两个部分,先从(x,y)到(x+1,y)再从(x+1,y)到终点,第一部分固定权值就是 A[x][y],要使得这种情况的路径权值和最大,那么第二部分从(x+1,y)到终点的路径的权值和也要最大,这一部分与前面的 Dfs(x,y)的定义十分相似,仅仅是参数不同,因此这一部分可以表示成Dfs(x+1,y)。综上,

第一步向左的路径最大权值和为 A[x][y]+Dfs(x+1,y)；

②第一步向右：这类路径要求先从(x,y)到(x+1,y+1)再从(x+1,y+1)到终点，分析方法与上面一样，这类路径最大权值和为 A[x][y]+Dfs(x+1,y+1)；

为了避免重复搜索，我们开设全局数组 F[x][y]记录从(x,y)出发到终点路径的最大权值和，一开始全部初始化为−1 表示未被计算过。在计算 Dfs(x,y)时，首先查询 F[x][y]，如果 F[x][y]不等于−1，说明 Dfs(x,y)之前已经被计算过，直接返回 F[x][y]即可，否则计算出 Dfs(x,y)的值并存储在 F[x][y]中。

```
#include <iostream>
#include <algorithm>
using namespace std;
const int MAXN = 505;
int A[MAXN][MAXN],F[MAXN][MAXN],N;

int Dfs(int x,int y)
{
  if (F[x][y]==-1)
  {
      if (x==N)F[x][y]=A[x][y];
      else F[x][y]=A[x][y]+max(Dfs(x+1,y),Dfs(x+1,y+1));
  }
  return F[x][y];
}

int main()
{
  cin >> N;
  for(int i = 1;i <= N;i ++)
      for(int j = 1;j <= i;j ++)
          cin >> A[i][j];
  for(int i = 1;i <= N;i ++)
      for(int j = 1;j <= i;j ++)
          F[i][j] = -1;
  Dfs(1,1);
  cout << F[1][1] << endl;
  return 0;
}
```

由于 F[x][y]对于每个合法的(x,y)都只计算过一次，而且计算是在 O(1)内完成的，因此时间复杂度为 $O(N^2)$。可以通过本题。

方法三：动态规划(顺推法)

方法二通过分析搜索的状态重复调用自然过渡到记忆化搜索，而记忆化搜索本质上已

经是动态规划了。下面我们完全从动态规划的算法出发换一个角度给大家展示一下动态规划的解题过程,并提供动态规划的迭代实现法。

①确定状态:

题目要求从(1,1)出发到最底层路径最大权值和,路径中是各个点串联而成,路径起点固定,终点和中间点相对不固定。因此定义 F[x][y]表示从(1,1)出发到达(x,y)的路径最大权值和。最终答案 Ans=max{F[N][1],F[N][2],...,F[N][N]}。

②确定状态转移方程和边界条件:

不去考虑(1,1)到(x,y)的每一步是如何走的,只考虑最后一步是如何走,根据最后一步是向左还是向右分成以下两种情况:

·向左:最后一步是从(x-1,y)走到(x,y),此类路径被分割成两部分,第一部分是从(1,1)走到(x-1,y),第二部分是从(x-1,y)走到(x,y),要计算此类路径的最大权值和,必须用到第一部分的最大权值和,此部分问题的性质与 F[x][y]的定义一样,就是 F[x-1,y],第二部分就是 A[x][y],两部分相加即得到此类路径的最大权值和为 F[x-1,y]+A[x,y];

·向右:最后一步是从(x-1,y-1)走到(x,y),此类路径被分割成两部分,第一部分是从(1,1)走到(x-1,y),第二部分是从(x-1,y)走到(x,y),分析方法如上。此类路径的最大权值和为 F[x-1,y-1]+A[x,y];

F[x][y]的计算需要求出上面两种情况的最大值。综上,得到状态转移方程如下:

F[x][y]=max{F[x-1,y-1],F[x-1][y]}+A[x,y]

与递归关系式还需要递归终止条件一样,这里我们需要对边界进行处理,以防无限递归下去。观察发现计算F[x][y]时需要用到 F[x-1][y-1]和 F[x-1,y],是上一行的元素,随着递归的深入,最终都要用到第一行的元素 F[1][1],F[1][1]的计算不能再使用状态转移方程来求,而是应该直接赋予一个特值 A[1][1]。这就是边界条件。

综上得:

状态转移方程:F[x][y]=max{F[x-1][y-1],F[x-1][y]}+A[x,y]

边界条件:F[1][1]=A[1][1]

现在让我们来分析一下该动态规划的正确性,分析该解法是否满足使用动态规划的两个前提:

·最优化原理(最优子结构性质):这个在分析状态转移方程时已经分析得比较透彻,明显是符合最优化原理的;

·无后效性:状态转移方程中,我们只关心 F[x-1][y-1]与 F[x-1][y]的值,计算 F[x-1][y-1]时可能有多种不同的决策对应着最优值,选哪种决策对计算 F[x][y]的决策没有影响,F[x-1][y-1]也是一样。这就是无后效性。

③程序实现:

由于状态转移方程就是递归关系式,边界条件就是递归终止条件,所以可以用递归来完成,递归存在重复调用,利用记忆化可以解决重复调用的问题,方法二已经讲过。记忆化实现比较简单,而且不会计算无用状态,但递归也会受到"栈的大小"和"递推+回归执行方式"的约束,另外记忆化实现调用状态的顺序是按照实际需求而展开,没有大局规划,不利于进一步优化。

这里介绍一种迭代法。与分析边界条件方法相似,计算 F[x][y]用到状态 F[x-1][y

-1]与F[x-1][y],这些元素在F[x][y]的上一行,也就是说要计算第x行的状态的值,必须要先把第x-1行元素的值计算出来,因此我们可以先把第一行元素F[1][1]赋为A[1][1],再从第二行开始按照行递增的顺序计算出每一行的有效状态即可。时间复杂度为$O(N^2)$。

```
#include <iostream>
#include <algorithm>
using namespace std;
const int MAXN = 1005;
int A[MAXN][MAXN],F[MAXN][MAXN],N;

int main()
{
  cin >> N;
  for(int i = 1;i <= N;i ++)
      for(int j = 1;j <= i;j ++)
          cin >> A[i][j];
  F[1][1] = A[1][1];
  for(int i = 2;i <= N;i ++)
      for(int j = 1;j <= i;j ++)
          F[i][j]=max(F[i-1][j-1],F[i-1][j])+A[i][j];
  int ans =0;
  for(int i = 1;i <= N;i ++)
      ans = max(ans,F[N][i]);
  cout << ans << endl;
  return 0;
}
```

方法四:动态规划(逆推法)

【算法分析】

(1)贪心法往往得不到最优解:本题若采用贪心法,则:13-11-12-14-13,其和为63;但存在另一条路:13-8-26-15-24,其和为86。

贪心法问题所在:眼光短浅。

(2)动态规划求解:动态规划求解问题的过程归纳为:自顶向下分析,自底向上计算。

其基本方法是:

划分阶段:按三角形的行划分阶段,若有n行,则有n-1个阶段。

A. 从根结点13出发,选取它的两个方向中的一条支路,当到倒数第二层时,每个结点其后继仅有两个结点,可以直接比较,选择最大值为前进方向,从而求得从根结点开始到底端的最大路径。

B. 自底向上计算:(给出递推式和终止条件)

①从底层开始,本身数即为最大数;

②倒数第二层的计算,取决于底层的数据:12+6=18,13+14=27,24+15=39,24+8

=32;

③倒数第三层的计算,取决于底二层计算的数据:27+12=39,39+7=46,39+26=65;

④倒数第四层的计算,取决于底三层计算的数据:46+11=57,65+8=73;

⑤最后的路径:13—8—26—15—24。

C. 数据结构及算法设计

①图形转化:直角三角形,便于搜索:向下、向右

②用三维数组表示数塔:a[x][y][1]表示行、列及结点本身数据,a[x][y][2]能够取得最大值,a[x][y][3]表示前进的方向——0 向下,1 向右;

③算法:

数组初始化,输入每个结点值及初始的最大路径、前进方向为 0;

从倒数第二层开始向上一层求最大路径,共循环 n-1 次;

从顶向下输出路径:究竟向下还是向右取决于列的值,若列的值比原先多 1 则向右,否则向下。

【参考程序】

```
#include<iostream>
#include<cstring>
using namespace std;
int main()
{
  int n,x,y;
  int a[51][51][3];
  cout<<" please input the number of rows:";
  cin>>n;
  memset(a,0,sizeof(0));
  for (x=1;x<=n;x++)                    //输入数塔的初始值
   for (y=1;y<=x;y++)
    {
      cin>>a[x][y][1];
      a[x][y][2]=a[x][y][1];
      a[x][y][3]=0;                     //路径走向,默认向下
    }
  for (x=n-1;x>=1;x--)
   for (y=1;y<=x;y++)
    if (a[x+1][y][2]>a[x+1][y+1][2])//选择路径,保留最大路径值
      { a[x][y][2]=a[x][y][2]+a[x+1][y][2]; a[x][y][3]=0; }
    else { a[x][y][2]=a[x][y][2]+a[x+1][y+1][2]; a[x][y][3]=1; }
    cout<<" max="<<a[1][1][2]<<endl; //输出数塔最大值
    y=1;
    for (x=1;x<=n-1;x++)                //输出数塔最大值的路径
     {
```

```
        cout<<a[x][y][1]<<"->";
        y=y+a[x][y][3];                    //下一行的列数
       }
     cout<<a[n][y][1]<<endl;
}
```

输入：

```
5                                          //数塔层数
13
11  8
12  7    26
6  14    15    8
12  7    13  24    11
```

输出结果：

```
max=86
13->8->26->15->24
```

例 9.3　求最长不下降序列

【问题描述】

设有由 n 个不相同的整数组成的数列，记为：b(1)，b(2)，…，b(n)且 b(i)<>b(j)(i<>j)，若存在 i1<i2<i3< … < ie 且有 b(i1)<b(i2)< … <b(ie)，则称为长度为 e 的不下降序列。程序要求，当原数列出之后，求出最长的不下降序列。

例如 13，7，9，16，38，24，37，18，44，19，21，22，63，15。其中 13，16，18，19，21，22，63 就是一个长度为 7 的不下降序列，同时也有 7，9，16，18，19，21，22，63 组成的长度为 8 的不下降序列。

【输入样例】

```
14
13 7 9 16 38 24 37 18 44 19 21 22 63 15
```

【输出样例】

```
max=8
7 9 16 18 19 21 22 63
```

【算法分析】

根据动态规划的原理，由后往前进行搜索(当然从前往后也一样)。

(1)对 b(n)来说，由于它是最后一个数，所以当从 b(n)开始查找时，只存在长度为 1 的不下降序列；

(2)若从 b(n−1)开始查找，则存在下面的两种可能性：

①若 b(n−1)<b(n)则存在长度为 2 的不下降序列 b(n−1)，b(n)。

②若 b(n−1)>b(n)则存在长度为 1 的不下降序列 b(n−1)或 b(n)。

(3)一般若从 b(i)开始，此时最长不下降序列应该按下列方法求出：

在 b(i+1)，b(i+2)，…，b(n)中，找出一个比 b(i)大的且最长的不下降序列，作为它的后继。

【数据结构】

为算法上的需要,定义一个整数类型二维数组 b[n][3]

(1)b[i][1]表示第 i 个数的数值本身;

(2)b[i][2]表示从 i 位置到达 n 的最长不下降序列长度;

(3)b[i][3]表示从 i 位置开始最长不下降序列的下一个位置,若 b[i][3]=0 则表示后面没有连接项。

【求解过程】

①从倒数第二项开始计算,后面仅有 1 项,比较一次,因 63>15,不符合要求,长度仍为 1。

②从倒数第三项开始其后有 2 项,需做两次比较,得到目前最长的不下降序列为 2,如下表:

…	11	12	13	14
		22	63	15
		2	1	1
		13	0	0

…	11	12	13	14
	21	22	63	15
	3	2	1	1
	12	13	0	0

【处理过程】

①在 i+1,i+2,…,n 项中,找出比 b[i][1]大的最长长度 L 以及位置 k;

②若 L>0,则 b[i][2]=L+1;b[i][3]=k;

最后本题经过计算,其数据存储表如下:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
b[i][1]	13	7	9	16	38	24	37	18	44	19	21	22	63	15
b[i][2]	7	8	7	6	3	4	3	5	2	4	3	2	1	1
b[i][3]	4	3	4	8	9	7	9	10	13	11	12	13	0	0

初始化:

```
for (i=1;i<=n;i++)
{
    cin>>b[i][1];
    b[i][2]=1;b[i][3]=0;
}
```

下面给出求最长不下降序列的算法:

```
for (i=n-1;i>=1;i--)
{
  l=0;k=0;
  for (j=i+1;j<=n;j++)
    if((b[j][1]>b[i][1])&&(b[j][2]>l))
    {
      l=b[j][2];
      k=j;
    }
  if (l>0)
```

```
  {
    b[i][2]=l+1;
    b[i][3]=k;
  }
}
```

下面找出最长不下降序列：

```
k=1;
for (j=1;j<=n;j++)
  if (b[j][2]>b[k][2])  k=j;
```

最长不下降序列长度为 b[k][2]序列

```
while (k! =0)
{
    cout<<" "<<b[k][1];
    k=b[k][3];
}
```

【参考程序】

```
#include<iostream>
using namespace std;
int main()
{
  int n,i,j,l,k,b[200][10];
  cout<<" input n:"<<endl;
  cin>>n;
  for (i=1;i<=n;i++)                    //输入序列的初始值
  {
    cin>>b[i][1];
    b[i][2]=1;b[i][3]=0;
  }
  for (i=n-1;i>=1;i--)                  //求最长不下降序列
   {
   l=0;k=0;
     for (j=i+1;j<=n;j++)
      if ((b[j][1]>b[i][1])&&(b[j][2]>l))
        {
          l=b[j][2];
          k=j;
        }
      if (l>0)
        {
          b[i][2]=l+1;b[i][3]=k;
```

```
        }
    }
    k=1;
    for (j=1;j<=n;j++)                    //求最长不下降序列的起始位置
     if (b[j][2]>b[k][2]) k=j;
    cout<<" max="<<b[k][2]<<endl; //输出结果
    while (k!=0)                          //输出最长不下降序列
    {
      cout<<' '<<b[k][1];
      k=b[k][3];
    }
}
```

例 9.4 拦截导弹(Noip1999)

某国为了防御敌国的导弹袭击,发展出一种导弹拦截系统。但是这种导弹拦截系统有一个缺陷:虽然它的第一发炮弹能够到达任意的高度,但是以后每一发炮弹都不能高于前一发的高度。某天,雷达捕捉到敌国的导弹来袭。由于该系统还在试用阶段,所以只有一套系统,因此有可能不能拦截所有的导弹。

输入导弹依次飞来的高度(雷达给出的高度数据是不大于 30000 的正整数,导弹数不超过 1000),计算这套系统最多能拦截多少导弹,如果要拦截所有导弹最少要配备多少套这种导弹拦截系统。

【输入样例】

389 207 155 300 299 170 158 65

【输出样例】

6(最多能拦截的导弹数)

2(要拦截所有导弹最少要配备的系统数)

【算法分析】

第一问即经典的最长不下降子序列问题,可以用一般的 DP 算法,也可以用高效算法,但这个题的数据规模不需要。

用 a[x]表示原序列中第 x 个元素,b[x]表示长度为 x 的不下降子序列的长度。当处理 a[x]时,可查找它可以连接到长度最大为多少的不下降子序列后(即与部分 b[x]比较)。假设可以连到长度最大为 maxx 的不下降子序列后,则 b[x]=maxx+1。b 数组被赋值的最大值就是第一问的答案。

第二问用贪心法即可。每颗导弹来袭时,使用能拦截这颗导弹的防御系统中上一次拦截导弹高度最低的那一套来拦截。若不存在符合这一条件的系统,则使用一套新系统。

【参考程序】(顺推法)

```
#include<iostream>
#include<cstdio>
#include<cstring>
using namespace std;
int main()
{
  freopen("in.txt", "r", stdin);
```

```
freopen("out.txt","w",stdout);
int i,j,k,x,n,maxx,m,a[10000],b[10000],h[10000];
i=1;n=0;m=0;
memset(a,0,sizeof(a));
memset(b,0,sizeof(b));
memset(h,0,sizeof(h));
while (cin>>a[i])
 {
    maxx=0;
    for (j=1;j<=i-1;j++)          //计算前 i-1 个导弹最佳拦截的方案
     if (a[j]>=a[i])
      if (b[j]>maxx)
        maxx=b[j];
    b[i]=maxx+1;                  //在前 i-1 个导弹最佳方案上+1
    if (b[i]>m) m=b[i];
    x=0;
    for (k=1;k<=n;k++)            //计算由哪一套系统拦截导弹
     if (h[k]>=a[i])
      if (x==0) x=k;
      else if (h[k]<h[x]) x=k;
                          //如果有多套系统可拦截,则选择上一次拦截高度最低的
    if (x==0) {n++;x=n;}          //新增一套导弹拦截系统
    h[x]=a[i];
    i++;
 }
cout<<m<<endl<<n<<endl;
}
```

经过计算,其数据存储表如下

i	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8
a[i]	389	207	155	300	299	170	158	65
b[i]	1	2	3	2	3	4	5	6
n 值	1			2				
h[1]	389	207	155					65
h[2]				300	299	170	158	

例 9.5　图 9-4 表示城市之间的交通路网,线段上的数字表示费用,单向通行由 A->E。试用动态规划的最优化原理求出 A->E 的最省费用。

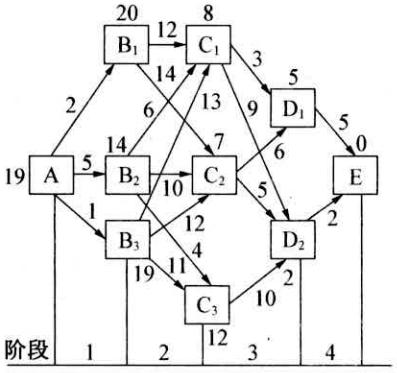


图 9-4

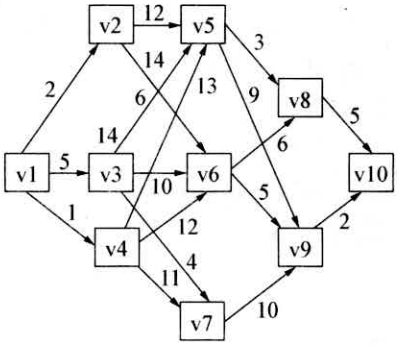


图 9-5

如图 9-5,求 v1 到 v10 的最短路径长度及最短路径。

【输入样例】

```
10
0  2  5  1  0  0  0  0  0  0
0  0  0  0 12 14  0  0  0  0
0  0  0  0  6 10  4  0  0  0
0  0  0  0 13 12 11  0  0  0
0  0  0  0  0  0  0  3  9  0
0  0  0  0  0  0  0  6  5  0
0  0  0  0  0  0  0  0 10  0
0  0  0  0  0  0  0  0  0  5
0  0  0  0  0  0  0  0  0  2
0  0  0  0  0  0  0  0  0  0
```

【输出样例】

```
minlong=19
1  3  5  8  10
```

【算法分析】逆推法

设 f[i]表示点 i 到 v10 的最短路径长度,则 f[10]=0

f[i]=min{ a[i][x]+f[x]; 当 a[i][x]>0 ,i<x<=n 时}

【参考程序】(逆推法)

```
#include<iostream>
using namespace std;
#include<cstring>
#include<cstdio>
int main()
{
    long n,i,j,x,f[100],c[100],a[100][100];
    memset(a,0,sizeof(a));
    memset(c,0,sizeof(c));
    cin>>n;
    for (i=1;i<=n;i++)          //输入各个城市之间距离
      for (j=1;j<=n;j++)
```

```
            cin>>a[i][j];
        for (i=1;i<=n;i++)
          f[i]=1000000;                    //初始化,默认每一个城市到达终点都是 1000000
        f[n]=0;
        for (i=n-1;i>=1;i--)               //从终点往前逆推,计算最短路径
         for (x=i+1;x<=n;x++) //若 f[x]=1000000 表示城市 x 到终点城市不通
          if ((a[i][x]>0)&&(f[x]!=1000000)&&(f[i]>a[i][x]+f[x]))
              {                            //a[i][x]>0 表示城市 i 和城市 x 通路
                f[i]=a[i][x]+f[x]; //城市 i 到终点最短路径的值
                c[i]=x;
              }
          cout<<" minlong="<<f[1]<<endl;   //输出最短路径的值
          x=1;
          while (x!=0)                             //输出路过的各个城市
           {
              cout<<x<<' ';
              x=c[x];
           }
           cout<<endl;
}
```

例 9.6　挖地雷

【问题描述】

在一个地图上有 n 个地窖(n<=200),每个地窖中埋有一定数量的地雷。同时,给出地窖之间的连接路径,并规定路径都是单向的,且保证都是小序号地窖指向大序号地窖,也不存在可以从一个地窖出发经过若干地窖后又回到原来地窖的路径。某人可以从任一处开始挖地雷,然后沿着指出的连接往下挖(仅能选择一条路径),当无连接时挖地雷工作结束。设计一个挖地雷的方案,使他能挖到最多的地雷。

【输入格式】

```
n                                  //地窖的个数
w1,w2,…,wn                         //每个地窖中的地雷数
x1,y1                              //表示从 x1 可到 y1,x1<y1
x2,y2
…
0,0                                //表示输入结束
```

【输出格式】

```
k1-k2-…-kv                         //挖地雷的顺序
max                                //最多挖出的地雷数
```

【输入样例】

```
6
5 10 20 5 4 5
```

【输出样例】

```
3-4-5-6
34
```

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

【算法分析】

本题是一个经典的动态规划问题。很明显,题目规定所有路径都是单向的,所以满足无后效性原则和最优化原理。设 w[i]为第 i 个地窖所藏有的地雷数,a[i][j]表示第 i 个地窖与第 j 个地窖之间是否有通路,f[i]为从第 i 个地窖开始最多可以挖出的地雷数,则有如下递归式:

f[i]=max{ w[i]+f[j]}　　(i<j<=n , a[i][j]=true)

边界:f[n]=w[n]

于是就可以通过递推的方法,从后面的 f(n)往前逐个找出所有的 f[i],再从中找一个最大的即为问题 2 的解。对于具体所走的路径(问题 1),可以通过一个向后的链接来实现。

【参考程序】

```
#include<iostream>
#include<cstring>
using namespace std;
int main()
{
    long f[201]={0},w[201],c[201]={0};
    bool a[201][201]={0};
    long i,j,n,x,y,l,k,maxx;
    memset(f,0,sizeof(f));
    memset(c,0,sizeof(c));
    memset(a,false,sizeof(a));
    cin>>n;
    for (i=1;i<=n;i++)
      cin>>w[i];                    //输入每个地窖中的地雷数
    do
      {
        cin>>x>>y;                  //表示从 x 可到 y
        if ((x!=0)&&(y!=0)) a[x][y]=true;
      }while ((x!=0)||(y!=0));
      f[n]=w[n];                    //从后面的 f[n]往前逐个找出所有的 f[i]
      for (i=n-1;i>=1;i--)
```

```
    {
      l=0;k=0;
      for (j=i+1;j<=n;j++)
       if ((a[i][j])&&(f[j]>l))
        { l=f[j]; k=j; }
      f[i]=l+w[i];              //保存从第 i 个地窖起能挖到的后继点最大地雷数
      c[i]=k;                   //k 地窖是 i 地窖最优路径
    }
    k=1;
    for (j=2;j<=n;j++)          //计算最多挖出的地雷数
      if (f[j]>f[k]) k=j;
    maxx=f[k];
    cout<<k;
    k=c[k];
    while (k!=0)                //输出挖地雷的顺序
      {
        cout<<"-"<<k;
        k=c[k];
      }
    cout<<endl;
    cout<<maxx<<endl;           //输出最多挖出的地雷数
}
```

例 9.7　友好城市

【问题描述】

Palmia 国有一条横贯东西的大河，河有笔直的南北两岸，岸上各有位置各不相同的 N 个城市。北岸的每个城市有且仅有一个友好城市在南岸，而且不同城市的友好城市不相同。

每对友好城市都向政府申请在河上开辟一条直线航道连接两个城市，但是由于河上雾太大，政府决定避免任意两条航道交叉，以避免事故。编程帮助政府做出一些批准和拒绝申请的决定，使得在保证任意两条航线不相交的情况下，被批准的申请尽量多。

【输入格式】

第 1 行，一个整数 N(1<=N<=5000)，表示城市数。

第 2 行到第 n+1 行，每行两个整数，中间用 1 个空格隔开，分别表示南岸和北岸的一对友好城市的坐标。(0<=xi<=10000)

【输出格式】

仅一行，输出一个整数，表示政府所能批准的最多申请数。

【输入样例】

```
7
22 4
2  6
```

【输出样例】

```
4
```

10 3
15 12
9 8
17 17
4 2

【算法分析】

我们将每对友好城市看成一条线段,则这道题的描述化为:有 N 条线段,问最少去掉多少条线,可以使剩下的线段互不交叉?

第一,以北岸为线的起点而南岸为线的终点;先将所有的线按照起点坐标值从小到大排序,按照每条线的终点坐标计算交叉数:求线 I 的交叉数 A[I],则检查所有 1～I－1 条线,若线 J(1<=J< I)的终点值大于线 I 的终点值,则线 I 与线 J 相交。A[I]与 A[J]同时加 1。整个搜索量最大为 5000 * 5000。

第二,将 A 数组从大到小排序,每删除一条线,则将与之相交的线的 A 值减 1,重复这个过程,直到所有 A 值都为 0。此时剩下的线则全不交叉。

如上数据,则可得下面结果:

编号	南岸	北岸	交叉数
1	1	3	2
2	2	4	2
3	3	1	2
4	4	5	1
5	5	2	3

此时,1、2、3 航线的交叉数都一样,如果删去的是 3、5 线,则剩下的 1、2、4 线互不相交,最多航线数为 3;如果删去的是 2、3 线,则还要删去第 5 线才符合要求,此时的最多航线数为 2,不是最优解。

于是,我们从上面的分析中再深入,将航线按起点坐标排好序后,如上所述,在本题中,只要线 J 的起点小于线 I 的起点,同时它的终点也小于线 I 的终点,则线 J 和线 I 不相交。因此,求所有线中最多能有多少条线不相交,实际上是从终点坐标值数列中求一个最长不下降序列。这就把题目转化为一个非常典型的动态规划题目了。

求最长不下降序列的规划方程如下:

L[i]=max{L[j]}+1;1<=j<i 且 Sj<Si。Si 为航线的终点坐标值。

从以上分析过程可以得出:当我们拿到一道题时,不要急于求解,而应先将题目的表面现象一层层像剥竹笋一样去掉,只留下最实质的内容。这时再来设计算法,往往能事半功倍。

例 9.8 合唱队形

【问题描述】

N 位同学站成一排,音乐老师要请其中的(N－K)位同学出列,使得剩下的 K 位同学排成合唱队形。

合唱队形是指这样的一种队形:设 K 位同学从左到右依次编号为 1, 2, …, K,他们的

身高分别为 T_1，T_2，…，T_K，则他们的身高满足 $T_1 < T_2 < \cdots < T_i$，$T_i > T_{i+1} > \cdots > T_K$（1<=i<=K）。

你的任务是，已知所有 N 位同学的身高，计算最少需要几位同学出列，可以使得剩下的同学排成合唱队形。

【输入格式】

输入文件 chorus.in 的第一行是一个整数 N(2 <= N <= 100)，表示同学的总数。第二行有 n 个整数，用空格分隔，第 i 个整数 T_i(130 <= T_i <= 230)是第 i 位同学的身高(厘米)。

【输出格式】

输出文件 chorus.out 包括一行，这一行只包含一个整数，就是最少需要几位同学出列。

【输入样例】

8

186 186 150 200 160 130 197 220

【输出样例】

4

【数据范围】

50%的数据满足：n <= 20；全部的数据满足：n<=100。

【算法分析】

我们按照由左而右和由右而左的顺序，将 n 个同学的身高排成数列。如何分别在这两个数列中寻求递增的、未必连续的最长子序列，就成为问题的关键。设

a 为身高序列，其中 a[i]为同学 i 的身高；

b 为由左而右身高递增的人数序列，其中 b[i]为同学 1…同学 i 间(包括同学 i)身高满足递增顺序的最多人数。显然 $b[i]=\max_{1\le j\le i-1}\{b[j]\mid \text{同学 j 的身高}<\text{同学 i 的身高}\}+1$；

c 为由右而左身高递增的人数序列，其中 c[i]为同学 n…同学 i 间(包括同学 i)身高满足递增顺序的最多人数。显然 $c[i]=\max_{i+1\le j\le n}\{c[j]\mid \text{同学 j 的身高}<\text{同学 i 的身高}\}+1$；

由上述状态转移方程可知，计算合唱队形的问题具备了最优子结构性质(要使 b[i]和 c[i]最大，子问题的解 b[j]和 c[k]必须最大(1<=j<=i-1，i+1<=k<=n))和重迭子问题的性质(为求得 b[i]和 c[i]，必须一一查阅子问题的解 b[1]…b[i-1]和 c[i+1]…c[n])，因此可采用动态程序设计的方法求解。

显然，合唱队的人数为 $\max_{1\le i\le n}\{b[i]+c[i]\}-1$(公式中同学 i 被重复计算，因此减 1)，n 减去合唱队人数即为解。

具体算法如下：

```
#include<cstring>
#include<iostream>
using namespace std;
int a[200],b[200],c[200];
main()
{
  int n,i,j,maxx;
  cin>>n;                          //读入学生数
  memset(b,0,sizeof(b));           //身高满足递增顺序的两个队列初始化
```

```
    memset(c,0,sizeof(c));
    for (i=1;i<=n;i++)                    //读每个学生的身高
     cin>>a[i];
    for (i=1;i<=n;i++)                    //按照由左而右的顺序计算 b 序列
     {
        b[i]=1;
        for (j=1;j<=i-1;j++)
          if ((a[i]>a[j])&&(b[j]+1>b[i]))
            b[i]=b[j]+1;
     }
    for (i=n;i>=1;i--)                    //按照由右而左的顺序计算 c 序列
     {
        c[i]=1;
        for (j=i+1;j<=n;j++)
          if ((a[j]<a[i])&&(c[j]+1>c[i]))
            c[i]=c[j]+1;
     }
    maxx=0;                               //计算合唱队的人数 max(其中 1 人被重复计算)
    for (i=1;i<=n;i++)
      if (b[i]+c[i]>maxx)
        maxx=b[i]+c[i];
    cout<<n-maxx+1<<endl;  //输出出列人数
}
```

这个算法的时间复杂度为 $O(n^2)$,在 1 秒时限内可解决 n<=100 范围内的问题。

例 9.9 最长公共子序列

【问题描述】

一个给定序列的子序列是在该序列中删去若干元素后得到的序列。确切地说,若给定序列 $X=<x_1,x_2,\cdots,x_m>$,则另一序列 $Z=<z_1,z_2,\cdots,z_k>$是 X 的子序列是指存在一个严格递增的下标序列$<i_1,i_2,\cdots,i_k>$,使得对于所有 j=1,2,…,k 有:

$Xi_j=Z_j$

例如,序列 Z=<B,C,D,B>是序列 X=<A,B,C,B,D,A,B>的子序列,相应的递增下标序列为<2,3,5,7>。给定两个序列 X 和 Y,当另一序列 Z 既是 X 的子序列又是 Y 的子序列时,称 Z 是序列 X 和 Y 的公共子序列。例如,若 X=<A,B,C,B,D,A,B>和 Y=<B,D,C,A,B,A>,则序列<B,C,A>是 X 和 Y 的一个公共子序列,序列 <B,C,B,A>也是 X 和 Y 的一个公共子序列。而且,后者是 X 和 Y 的一个最长公共子序列.因为 X 和 Y 没有长度大于 4 的公共子序列。

给定两个序列 $X=<x_1,x_2,\cdots,x_m>$和 $Y=<y_1,y_2,\cdots,y_n>$.要求找出 X 和 Y 的一个最长公共子序列。

【输入格式】

输入文件共有两行。每行为一个由大写字母构成的长度不超过 200 的字符串,表示序

列 X 和 Y。

【输出格式】

输出文件第一行为一个非负整数。表示所求得的最长公共子序列的长度。若不存在公共子序列，则输出文件仅有一行输出一个整数 0。否则在输出文件的第二行输出所求得的最长公共子序列(也用一个大写字母组成的字符串表示)。若符合条件的最长公共子序列不止一个，只需输出其中任意一个。

【样例输入】

ABCBDAB

BDCABA

【样例输出】

4

提示：

最长公共子串(Longest Common Substirng)和最长公共子序列(Longest Common Subsequence，LCS)的区别为：子串是串的一个连续的部分，子序列则是不改变序列的顺序，而从序列中去掉任意的元素而获得新的序列；也就是说，子串中字符的位置必须是连续的，子序列则可以不必连续。字符串长度小于等于 1000。

分析：

与最长不下降子序列(LIS)类似的，我们可以以子序列的结尾作为状态，但现在有两个子序列，那么直接以两个子序列当前的结尾作为状态即可。

①确定状态：

设 F[x][y]表示 S[1..x]与 T[1..y]的最长公共子序列的长度。答案为 F[|S|][|T|]。

②确定状态转移方程和边界条件：

分三种情况来考虑：

- S[x]不在公共子序列中：该情况下 F[x][y]=F[x−1][y]；
- T[y]不在公共子序列中：该情况下 F[x][y]=F[x][y−1]；
- S[x]=T[y]，S[x]与 T[y]在公共子序列中：该情况下，F[x][y]=F[x−1][y−1]+1。

F[x][y]取上述三种情况的最大值。综上：

状态转移方程：F[x][y]=max{F[x−1][y]，F[x][y−1]，F[x−1][y−1]+1}，其中第三种情况要满足 S[x]=T[y]；

边界条件：F[0][y]=0，F[x][0]=0。

③程序实现：

计算 F[x][y]时用到 F[x−1][y−1]，F[x−1][y]，F[x][y−1]这些状态，它们要么在 F[x][y]的上一行，要么在 F[x][y]的左边。因此预处理出第 0 行，然后按照行从小到大、同一行按照列从小到大的顺序来计算就可以用迭代法计算出来。时间复杂度为 O(|S| * |T|)。

【参考程序】

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

```
const int MAXN = 5005;
string S,T;
int F[MAXN][MAXN];

int main()
{
  cin >> S;
  cin >> T;
  int ls = S.length(),lt = T.length();
  for(int i = 1;i <= ls;i ++)
      for(int j = 1;j <= lt;j ++)
      {
          F[i][j] = max(F[i - 1][j],F[i][j - 1]);
          if (S[i - 1] == T[j - 1])
              F[i][j] = max(F[i][j],F[i - 1][j - 1] + 1);
      }
  cout << F[ls][lt] << endl;
  return 0;
}
```

例 9.10 机器分配

【问题描述】

总公司拥有高效设备 M 台,准备分给下属的 N 个分公司。各分公司若获得这些设备,可以为国家提供一定的盈利。问:如何分配这 M 台设备才能使国家得到的盈利最大?求出最大盈利值。其中 M<=15,N<=10。分配原则:每个公司有权获得任意数目的设备,但总台数不超过设备数 M。

【输入格式】

第 1 行有两个数,第一个数是分公司数 N,第二个数是设备台数 M。

接下来是一个 N * M 的矩阵,表明了第 I 个公司分配 J 台机器的盈利。

【输出格式】

第 1 行输出最大盈利值。

接下 N 行,每行 2 个数,即分公司编号和该分公司获得设备台数。

【输入样例】

```
3 3     //3 个分公司分 3 台机器
30 40 50
20 30 50
20 25 30
```

【输出样例】

```
70  //最大盈利值为 70
1 1  //第一分公司分 1 台
2 1  //第二分公司分 1 台
3 1  //第三分公司分 1 台
```

【算法分析】

按照公司的顺序来分配机器,即按照公司的顺序划分阶段,第一个阶段把 M 台设备分给第一个公司,记录下来获得的各个盈利值,然后把 M 台设备分给前两个公司,和第一个阶段比较记录下来更优的各个盈利值,一直到第 N 个阶段把 M 台机器全部分给了 N 个公司,

就可以得到最优解，下面来讨论两个阶段之间的关系，设数组 F[I][J]表示前 I 个公司分配 J 台机器的最大盈利，数组 F[I－1][K]表示前 I－1 个公司分配 K 台机器的最大盈利(1<=I<=N,1<=J<=M,0<=K<=J)，用 Value[I][J]表示第 I 个公司分配 J 台机器的盈利，则 F[I][J]可以由下面的式子中取最大值获得：

F[I－1][0]＋Value[I][J]　//前 I－1 公司分配 0 台机器最大盈利＋第 I 个公司分配 J 台的机器的盈利

F[I－1][1]＋Value[I][J－1]//前 I－1 公司分配 1 台机器最大盈利＋第 I 个公司分配 J－1 台的机器的盈利

F[I－1][2]＋Value[I][J－2]//前 I－1 公司分配 2 台机器最大盈利＋第 I 个公司分配 J－2 台的机器的盈利

F[I－1][J－1]＋Value[I][1]//前 I－1 公司 J－1 分配台机器最大盈利＋第 I 个公司分配 1 台的机器的盈利

F[I－1][J]＋Value[I][0]　//前 I－1 公司分配 J 台机器最大盈利＋第 I 个公司分配 0 台的机器的盈利

在这里用机器数用做每个阶段的状态，由于 Value[I][J]的值为定值，要使前面每个式子的值最大，就必须使第 i－1 阶段的各个状态的值最大，阶段 i 的状态只能由阶段 i－1 中的状态通过状态转移方程求得，与其他状态没有关系。由此可见，该问题具备了最优子结构和无后效性原则，适宜使用动态程序设计方法求解。状态转移方程为：F[I][J]＝MAX{F[I－1][K]＋Value[I][J－K]} (1<=I<=N,1<=J<=M,0<=K<=J)

初始值：F[0][0]＝0，F[n][m]的值即为所求最大盈利值。

【参考程序】

```
#include<iostream>
#include<cstring>
using namespace std;
int show(int,int);
long max1,f[11][20],value[11][20];
int main()
{
  long m,n,i,j,k;
  cin>>n>>m;
  for (i=1;i<=n;i++)
   for (j=1;j<=m;j++)
    cin>>value[i][j];
  for (i=1;i<=n;i++)
   for (j=1;j<=m;j++)
    {
     max1=0;
      for (k=0;k<=j;k++)
       if (f[i-1][k]+value[i][j-k]>max1)
           max1=f[i-1][k]+value[i][j-k];
      f[i][j]=max1;
    }
  cout<<f[n][m]<<endl;                    //输出最大盈利值
  show(n,m);                              //输出分配情况
}
```

```
int show(int i,int j)                          //输出各分公司分配情况
{
  int k;
  if (i==0) return 0;
  for (k=0;k<=j;k++)
   if (max1==f[i-1][k]+value[i][j-k])//递归求各公司分配的机器数量
    {
      max1=f[i-1][k];
      show(i-1,k);
      cout<<i<<" "<<j-k<<endl;
      break;
    }
}
```

【上机练习】

1. 最长上升子序列【2.6 基本算法之动态规划 1759】

一个数的序列 bi,当 $b_1<b_2<\cdots<b_S$ 的时候,我们称这个序列是上升的。对于给定的一个序列($a_1,a_2,\cdots,a_N$),我们可以得到一些上升的子序列($a_{i1},a_{i2},\cdots,a_{iK}$),这里 1<=i1<i2<…<iK<=N。比如,对于序列(1,7,3,5,9,4,8),有它的一些上升子序列,如(1,7),(3,4,8)等等。这些子序列中最长的长度是 4,比如子序列(1,3,5,8)。

你的任务,就是对于给定的序列,求出最长上升子序列的长度。

输入:

输入的第一行是序列的长度 N(1<=N<=1000)。第二行给出序列中的 N 个整数,这些整数的取值范围都在 0 到 10000。

输出:

最长上升子序列的长度。

样例输入:

```
7
1 7 3 5 9 4 8
```

样例输出:

```
4
```

2. 最大子矩阵【2.6 基本算法之动态规划 1768】

已知矩阵的大小定义为矩阵中所有元素的和。给定一个矩阵,你的任务是找到最大的非空(大小至少是 1 * 1)子矩阵。

比如,如下 4 * 4 的矩阵

```
 0 -2 -7  0
 9  2 -6  2
-4  1 -4  1
-1  8  0 -2
```

的最大子矩阵是

9 2

−4 1

−1 8

这个子矩阵的大小是 15。

输入：

输入是一个 N * N 的矩阵。输入的第一行给出 N(0<N<=100)。再后面的若干行中，依次(首先从左到右给出第一行的 N 个整数，再从左到右给出第二行的 N 个整数……)给出矩阵中的 N2 个整数，整数之间由空白字符分隔(空格或者空行)。已知矩阵中整数的范围都在[−127,127]。

输出：

输出最大子矩阵的大小。

样例输入：

```
 4
 0 −2 −7  0
 9  2 −6  2
−4  1 −4  1
−1  8  0 −2
```

样例输出：

15

3. 登山【2.6 基本算法之动态规划 1996】

五一到了，ACM 队组织大家去登山观光，队员们发现山上一个有 N 个景点，并且决定按照顺序来浏览这些景点，即每次所浏览景点的编号都要大于前一个浏览景点的编号。同时队员们还有另一个登山习惯，就是不连续浏览海拔相同的两个景点，并且一旦开始下山，就不再向上走了。队员们希望在满足上面条件的同时，尽可能多的浏览景点，你能帮他们找出最多可能浏览的景点数么？

输入：

Line 1：N (2 <=N <=1000) 景点数

Line 2：N 个整数，每个景点的海拔

输出：

最多能浏览的景点数

样例输入：

```
8
186 186 150 200 160 130 197 220
```

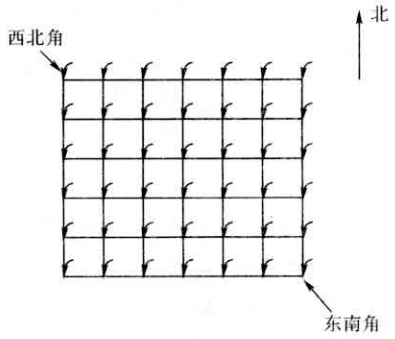
样例输出：

4

4. 摘花生【2.6 基本算法之动态规划 2728】

Hello Kitty 想摘点花生送给她喜欢的米老鼠。她来到一片有网格状道路的矩形花生地(如下图)，从西北角进去，东南角出来。地里每个道路的交叉点上都有种着一株花生苗，上面有若干颗花生，经过一株花生苗就能摘走该它上面所有的花生。Hello Kitty 只能向东或

向南走,不能向西或向北走。问 Hello Kitty 最多能够摘到多少颗花生。



输入:

第一行是一个整数 T,代表一共有多少组数据。1<=T<=100

接下来是 T 组数据。

每组数据的第一行是两个整数,分别代表花生苗的行数 R 和列数 C(1<=R,C<=100)

每组数据的接下来 R 行数据,从北向南依次描述每行花生苗的情况。每行数据有 C 个整数,按从西向东的顺序描述了该行每株花生苗上的花生数目 M(0<=M<=1000)。

输出:

对每组输入数据,输出一行,内容为 Hello Kitty 能摘到得最多的花生颗数。

样例输入:

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

样例输出:

```
8
16
```

5. 最大上升子序列和【2.6 基本算法之动态规划 3532】

一个数的序列 bi,当 b1<b2<…<bS 的时候,我们称这个序列是上升的。对于给定的一个序列(a1,a2,…,aN),我们可以得到一些上升的子序列(ai1,ai2,…,aiK),这里 1<=i1< i2<…<iK<=N。比如,对于序列(1,7,3,5,9,4,8),有它的一些上升子序列,如(1,7),(3,4,8)等等。这些子序列中序列和最大为 18,为子序列(1,3,5,9)的和。

你的任务,就是对于给定的序列,求出最大上升子序列和。注意,最长的上升子序列的和不一定是最大的,比如序列(100,1,2,3)的最大上升子序列和为 100,而最长上升子序列为(1,2,3)。

输入:

输入的第一行是序列的长度 N(1<=N<=1000)。第二行给出序列中的 N 个整数,这些整数的取值范围都在 0 到 10000(可能重复)。

输出：

最大上升子序列和.

样例输入：

7

1 7 3 5 9 4 8

样例输出：

18

6. 怪盗基德的滑翔翼【2.6 基本算法之动态规划 4977】

怪盗基德是一个充满传奇色彩的怪盗，专门以珠宝为目标的超级盗窃犯。而他最为突出的地方，就是他每次都能逃脱中村警部的重重围堵，而这也很大程度上是多亏了他随身携带的便于操作的滑翔翼。

有一天，怪盗基德像往常一样偷走了一颗珍贵的钻石，不料却被柯南小朋友识破了伪装，而他的滑翔翼的动力装置也被柯南踢出的足球破坏了。不得已，怪盗基德只能操作受损的滑翔翼逃脱。

假设城市中一共有 N 幢建筑排成一条线，每幢建筑的高度各不相同。初始时，怪盗基德可以在任何一幢建筑的顶端。他可以选择一个方向逃跑，但是不能中途改变方向（因为中森警部会在后面追击）。因为滑翔翼动力装置受损，他只能往下滑行（即：只能从较高的建筑滑翔到较低的建筑）。他希望尽可能多地经过不同建筑的顶部，这样可以减缓下降时的冲击力，减少受伤的可能性。请问，他最多可以经过多少幢不同建筑的顶部（包含初始时的建筑）？

输入：

输入数据第一行是一个整数 K（K＜100），代表有 K 组测试数据。

每组测试数据包含两行：第一行是一个整数 N（N＜100），代表有 N 幢建筑。第二行包含 N 个不同的整数，每一个对应一幢建筑的高度 h（0＜h＜10000），按照建筑的排列顺序给出。

输出：

对于每一组测试数据，输出一行，包含一个整数，代表怪盗基德最多可以经过的建筑数量。

样例输入：

3

8

300 207 155 299 298 170 158 65

8

65 158 170 298 299 155 207 300

10

2 1 3 4 5 6 7 8 9 10

样例输出：

6

6

9

7. 最低通行费【2.6 基本算法之动态规划 7614】

一个商人穿过一个 N*N 的正方形的网格,去参加一个非常重要的商务活动。他要从网格的左上角进,右下角出。每穿越中间 1 个小方格,都要花费 1 个单位时间。商人必须在(2N-1)个单位时间穿越出去。而在经过中间的每个小方格时,都需要缴纳一定的费用。

这个商人期望在规定时间内用最少费用穿越出去。请问至少需要多少费用?

注意:不能对角穿越各个小方格(即,只能向上下左右四个方向移动且不能离开网格)。

输入:

第一行是一个整数,表示正方形的宽度 N (1<=N<100);

后面 N 行,每行 N 个不大于 100 的整数,为网格上每个小方格的费用。

输出:

至少需要的费用。

样例输入:

```
5
1  4  6  8 10
2  5  7 15 17
6  8  9 18 20
10 11 12 19 21
20 23 25 29 33
```

样例输出:

```
109
```

提示:

样例中,最小值为 109=1+2+5+7+9+12+19+21+33。

8. 三角形最佳路径问题【2.6 基本算法之动态规划 7625】

如下所示的由正整数数字构成的三角形:

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

从三角形的顶部到底部有很多条不同的路径。对于每条路径,把路径上面的数加起来可以得到一个和,和最大的路径称为最佳路径。你的任务就是求出最佳路径上的数字之和。

注意:路径上的每一步只能从一个数走到下一层上和它最近的下边(正下方)的数或者右边(右下方)的数。

输入:

第一行为三角形高度 100>=h>=1,同时也是最底层边的数字的数目。

从第二行开始,每行为三角形相应行的数字,中间用空格分隔。

输出:

最佳路径的长度数值。

样例输入:

```
5
```

7

3 8

8 1 0

2 7 4 4

4 5 2 6 5

样例输出：

30

9. 拦截导弹【2.6 基本算法之动态规划 8780】

某国为了防御敌国的导弹袭击，发展出一种导弹拦截系统。但是这种导弹拦截系统有一个缺陷：虽然它的第一发炮弹能够到达任意的高度，但是以后每一发炮弹都不能高于前一发的高度。某天，雷达捕捉到敌国的导弹来袭。由于该系统还在试用阶段，所以只有一套系统，因此有可能不能拦截所有的导弹。

输入导弹依次飞来的高度（雷达给出的高度数据是不大于 30000 的正整数），计算这套系统最多能拦截多少导弹。

输入：

第一行是一个整数 N（不超过 15），表示导弹数。

第二行包含 N 个整数，为导弹依次飞来的高度（雷达给出的高度数据是不大于 30000 的正整数）。

输出：

一个整数，表示最多能拦截的导弹数。

样例输入：

8

389 207 155 300 299 170 158 65

样例输出：

6

第二节　背包问题

一、01 背包问题

问题：

有 N 件物品和一个容量为 V 的背包。第 i 件物品的费用（即体积，下同）是 w[i]，价值是 c[i]。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

基本思路：

这是最基础的背包问题，特点是：每种物品仅有一件，可以选择放或不放。

用子问题定义状态：即 f[i][v]表示前 i 件物品（部分或全部）恰放入一个容量为 v 的背包可以获得的最大价值，则其状态转移方程便是：f[i][v]＝max{f[i－1][v]，f[i－1][v－w[i]]＋c[i]}。

这个方程非常重要,基本上所有跟背包相关的问题的方程都是由它衍生出来的。所以有必要将它详细解释一下:“将前 i 件物品放入容量为 v 的背包中”这个子问题,若只考虑第 i 件物品的策略(放或不放),那么就可以转化为一个只牵扯前 i−1 件物品的问题。如果不放第 i 件物品,那么问题就转化为“前 i−1 件物品放入容量为 v 的背包中”;如果放第 i 件物品,那么问题就转化为“前 i−1 件物品放入剩下的容量为 v−w[i]的背包中”,此时能获得的最大价值就是 f[i−1][v−w[i]]再加上通过放入第 i 件物品获得的价值 c[i]。

注意 f[i][v]有意义当且仅当存在一个前 i 件物品的子集,其费用总和为 v。所以按照这个方程递推完毕后,最终的答案并不一定是 f[N][V],而是 f[N][0..V]的最大值。如果将状态的定义中的“恰”字去掉,在转移方程中就要再加入一项 f[i−1][v],这样就可以保证 f[N][V]就是最后的答案。但是若将所有 f[i][j]的初始值都赋为 0,你会发现 f[n][v]也会是最后的答案。为什么呢?因为这样你默认了最开始 f[i][j]是有意义的,只是价值为 0,就看作是无物品放的背包价值都为 0,所以对最终价值无影响,这样初始化后的状态表示就可以把“恰”字去掉。

优化空间复杂度

以上方法的时间和空间复杂度均为 O(N * V),其中时间复杂度基本已经不能再优化了,但空间复杂度却可以优化到 O(V)。

先考虑上面讲的基本思路如何实现,肯定是有一个主循环 i=1..N,每次算出来二维数组 f[i][0..V]的所有值。那么,如果只用一个数组 f[0..V],能不能保证第 i 次循环结束后 f[v]中表示的就是我们定义的状态 f[i][v]呢?f[i][v]是由 f[i−1][v]和 f[i−1][v−w[i]]两个子问题递推而来,能否保证在推 f[i][v]时(也即在第 i 次主循环中推 f[v]时)能够得到 f[i−1][v]和 f[i−1][v−w[i]]的值呢?事实上,这要求在每次主循环中我们以 v=V..0 的逆序推 f[v],这样才能保证推 f[v]时 f[v−w[i]]保存的是状态 f[i−1][v−w[i]]的值。

伪代码如下:

```
for i=1..N
for v=V..0
  f[v]=max{f[v],f[v−w[i]]+c[i]};
```

其中 f[v]=max{f[v],f[v−w[i]]+c[i]}相当于转移方程 f[i][v]=max{f[i−1][v],f[i−1][v−w[i]]+c[i]},因为现在的 f[v−w[i]]就相当于原来的 f[i−1][v−w[i]]。如果将 v 的循环顺序从上面的逆序改成顺序的话,那么则成了 f[i][v]由 f[i][v−w[i]]推知,与本题意不符,但它却是另一个重要的完全背包问题最简捷的解决方案,故学习只用一维数组解 01 背包问题是十分必要的。

例 9.11 01 背包问题

【问题描述】

一个旅行者有一个最多能装 M 公斤的背包,现在有 n 件物品,它们的重量分别是 W1,W2,…,Wn,它们的价值分别为 C1,C2,…,Cn,求旅行者能获得最大总价值。

【输入格式】

第 1 行:两个整数,M(背包容量,M<=200)和 N(物品数量,N<=30);

第 2 至 N+1 行:每行两个整数 Wi,Ci,表示每个物品的重量和价值。

【输出格式】

仅一行,一个数,表示最大总价值。

【输入样例】

10 4
2 1
3 3
4 5
7 9

【输出样例】

12

【解法一】

【算法分析】

设 f[i][v]表示前 i 件物品，总重量不超过 v 的最优价值，则 f[i][v]=max(f[i−1][v−w[i]]+c[i],f[i−1][v])；f[n][m]即为最优解。

【参考程序】

```
#include<cstdio>
using namespace std;
const int maxm=201, maxn=31;
int m, n;
int w[maxn], c[maxn];
int f[maxn][maxm];
int max(int x,int y)                    //求 x 和 y 最大值
{
    if (x < y) return y;
    else return x;
}
int main()
{
    scanf("%d%d",&m, &n);    //背包容量 m 和物品数量 n
    for (int i=1; i<=n; i++)  //在初始化循环变量部分，定义一个变量并初始化
      scanf("%d%d",&w[i],&c[i]);//每个物品的重量和价值
    for (int i=1; i<=n; i++)
                              // f[i][v]表示前 i 件物品，总重量不超过 v 的最优价值
        for (int v=m; v > 0; v--)
          if (w[i] <=v)  f[i][v]=max(f[i−1][v],f[i−1][v−w[i]]+c[i]);
            else  f[i][v]=f[i−1][v];
    printf("%d",f[n][m]);          // f[n][m]为最优解
    return 0;
}
```

使用二维数组存储各子问题时方便，但当 maxm 较大时，如 maxm=2000 时不能定义二维数组 f，怎么办，其实可以用一维数组。

【解法二】

【算法分析】

本问题的数学模型如下：设 f[v]表示重量不超过 v 公斤的最大价值，则 f[v]=max

{f[v],f[v-w[i]]+c[i]},当v>=w[i],1<=i<=n时。

【参考程序】

```
#include<cstdio>
using namespace std;
const int maxm=2001, maxn=31;
int m, n;
int w[maxn], c[maxn];
int f[maxm];
int main()
{
    scanf("%d%d",&m, &n);          //背包容量m和物品数量n
    for (int i=1; i<=n; i++)
      scanf("%d%d",&w[i],&c[i]);//每个物品的重量和价值
    for (int i=1; i<=n; i++)       //设f(v)表示重量不超过v公斤的最大价值
      for (int v=m; v>=w[i]; v--)
        if (f[v-w[i]]+c[i]>f[v])
          f[v]=f[v-w[i]]+c[i];
    printf("%d",f[m]);             // f(m)为最优解
    return 0;
}
```

总结:

01背包问题是最基本的背包问题,它包含了背包问题中设计状态、方程的最基本思想,另外,别的类型的背包问题往往也可以转换成01背包问题求解。故一定要仔细体会上面基本思路的得出方法,状态转移方程的意义,以及最后怎样优化的空间复杂度。

二、完全背包问题

问题:

有N种物品和一个容量为V的背包,每种物品都有无限件可用。第i种物品的费用是w[i],价值是c[i]。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量,且价值总和最大。

基本思路:

这个问题非常类似于01背包问题,所不同的是每种物品有无限件。也就是从每种物品的角度考虑,与它相关的策略已并非取或不取两种,而是有取0件、取1件、取2件…很多种。如果仍然按照解01背包时的思路,令f[i][v]表示前i种物品恰放入一个容量为v的背包的最大权值。仍然可以按照每种物品不同的策略写出状态转移方程,像这样:f[i][v]=max{f[i-1][v-k*w[i]]+k*c[i]|0<=k*w[i]<=v}。

将01背包问题的基本思路加以改进,得到了这样一个清晰的方法。这说明01背包问题的方程的确是很重要,可以推及其他类型的背包问题。

这个算法使用一维数组,先看伪代码:

```
for i=1..N
  for v=0..V
    f[v]=max{f[v],f[v-w[i]]+c[i]};
```

你会发现，这个伪代码与01背包问题的伪代码只有v的循环次序不同而已。为什么这样一改就可行呢？首先想想为什么01背包问题中要按照v=V..0的逆序来循环。这是因为要保证第i次循环中的状态f[i][v]是由状态f[i-1][v-w[i]]递推而来。换句话说，这正是为了保证每件物品只选一次，保证在考虑“选入第i件物品”这件策略时，依据的是一个绝无已经选入第i件物品的子结果f[i-1][v-w[i]]。而现在完全背包的特点恰是每种物品可选无限件，所以在考虑“加选一件第i种物品”这种策略时，却正需要一个可能已选入第i种物品的子结果f[i][v-w[i]]，所以就可以并且必须采用v=0..V的顺序循环。这就是这个简单的程序为何成立的道理。

这个算法也可以以另外的思路得出。例如，基本思路中的状态转移方程可以等价地变形成这种形式：f[i][v]=max{f[i-1][v],f[i][v-w[i]]+c[i]}，将这个方程用一维数组实现，便得到了上面的伪代码。

例9.12　完全背包问题

【问题描述】

设有n种物品，每种物品有一个重量及一个价值。但每种物品的数量是无限的，同时有一个背包，最大载重量为m，今从n种物品中选取若干件(同一种物品可以多次选取)，使其重量的和小于等于m，而价值的和为最大。

【输入格式】

第1行：两个整数，M(背包容量，M<=200)和N(物品数量，N<=30)；

第2至N+1行：每行两个整数Wi,Ci，表示每个物品的重量和价值。

【输出格式】

仅一行，一个数，表示最大总价值。

【输入样例】

```
10 4
2 1
3 3
4 5
7 9
```

【输出样例】

```
max=12
```

【解法一】

【算法分析】

设f[i][v]表示前i件物品，总重量不超过v的最优价值，则f[i][v]=max(f[i][v-w[i]]+c[i],f[i-1][v])；f[n][m]即为最优解。

【参考程序】

```
#include<cstdio>
using namespace std;
const int maxm=201, maxn=31;
int m, n;
```

```
int w[maxn], c[maxn];
int f[maxn][maxm];
int main()
{
    scanf("%d%d",&m, &n);        //背包容量 m 和物品数量 n
    for (int i=1; i<=n; i++)
      scanf("%d%d",&w[i],&c[i]);//每个物品的重量和价值
    for (int i=1; i<=n; i++)
                          //f[i][v]表示前 i 件物品,总重量不超过 v 的最优价值
      for (int v=1; v<=m; v++)
        if (v < w[i])  f[i][v]=f[i-1][v];
        else
            if (f[i-1][v] > f[i][v-w[i]]+c[i])  f[i][v]=f[i-1][v];
            else f[i][v]=f[i][v-w[i]]+c[i];
    printf(" max=%d",f[n][m]);    // f[n][m]为最优解
    return 0;
}
```

【解法二】

【算法分析】

本问题的数学模型如下:

设 f(v)表示重量不超过 v 公斤的最大价值,则 f(v)=max{f(v),f(v-w[i])+c[i]}(v>=w[i] ,1<=i<=n) 。

【参考程序】

```
#include<cstdio>
using namespace std;
const int maxm=2001,maxn=31;
int n,m,v,i;
int c[maxn],w[maxn];
int f[maxm];
int main()
{
    scanf("%d%d",&m,&n);        //背包容量 m 和物品数量 n
    for(i=1;i<=n;i++)
      scanf("%d%d",&w[i],&c[i]);
    for(i=1;i<=n;i++)
      for(v=w[i];v<=m;v++)     //设 f[v]表示重量不超过 v 公斤的最大价值
        if(f[v-w[i]]+c[i]>f[v])  f[v]=f[v-w[i]]+c[i];
    printf(" max=%d\n",f[m]);     // f[m]为最优解
    return 0;
```

}

一个简单有效的优化：

完全背包问题有一个很简单有效的优化，是这样的：若两件物品 i、j 满足 w[i]<=w[j] 且 c[i]>=c[j]，则将物品 j 去掉，不用考虑。这个优化的正确性显然：任何情况下都可将价值小费用高的 j 换成物美价廉的 i，得到至少不会更差的方案。对于随机生成的数据，这个方法往往会大大减少物品的件数，从而加快速度。然而这个并不能改善最坏情况的复杂度，因为有可能特别设计的数据可以一件物品也去不掉。

转化为 01 背包问题求解：

既然 01 背包问题是最基本的背包问题，那么我们可以考虑把完全背包问题转化为 01 背包问题来解。最简单的想法是，考虑到第 i 种物品最多选 V/w[i]件，于是可以把第 i 种物品转化为 V/w[i]件费用及价值均不变的物品，然后求解这个 01 背包问题。这样完全没有改进基本思路的时间复杂度，但这毕竟给了我们将完全背包问题转化为 01 背包问题的思路：将一种物品拆成多件物品。

更高效的转化方法是：把第 i 种物品拆成费用为 w[i]*2^k、价值为 c[i]*2^k 的若干件物品，其中 k 满足 w[i]*2^k<V。这是二进制的思想，因为不管最优策略选几件第 i 种物品，总可以表示成若干个 2^k 件物品的和。这样把每种物品拆成 O(log(V/w[i])+1)件物品，是一个很大的改进。

总结：

完全背包问题也是一个相当基础的背包问题，它有两个状态转移方程，分别在“基本思路”以及“O(VN)的算法”的小节中给出。希望你能够对这两个状态转移方程都仔细地体会，不仅记住，还要弄明白它们是怎么得出来的，最好能够自己想一种得到这些方程的方法。事实上，对每一道动态规划题目都思考其方程的意义以及如何得来，是加深对动态规划的理解、提高动态规划功力的好方法。

三、多重背包问题

问题：

有 N 种物品和一个容量为 V 的背包。第 i 种物品最多有 n[i]件可用，每件费用是w[i]，价值是 c[i]。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

基本算法：

这题目和完全背包问题很类似。基本的方程只需将完全背包问题的方程略微一改即可，因为对于第 i 种物品有 n[i]+1 种策略：取 0 件，取 1 件…取 n[i]件。令 f[i][v]表示前 i 种物品恰放入一个容量为 v 的背包的最大权值，则：f[i][v]=max{f[i-1][v-k*w[i]]+k*c[i]|0<=k<=n[i]}。复杂度是 O(V*∑n[i])。

转化为 01 背包问题

另一种好想好写的基本方法是转化为 01 背包求解：把第 i 种物品换成 n[i]件 01 背包中的物品，则得到了物品数为∑n[i]的 01 背包问题，直接求解，复杂度仍然是 O(V*∑n[i])。

但是我们期望将它转化为 01 背包问题之后能够像完全背包一样降低复杂度。仍然考

虑二进制的思想,我们考虑把第 i 种物品换成若干件物品,使得原问题中第 i 种物品可取的每种策略——取 0..n[i]件——均能等价于取若干件代换以后的物品。另外,取超过 n[i]件的策略必不能出现。

方法是:将第 i 种物品分成若干件物品,其中每件物品有一个系数,这件物品的费用和价值均是原来的费用和价值乘以这个系数,使这些系数分别为 1,2,4,…,2^(k-1),n[i]-2^k+1,且 k 是满足 n[i]-2^k+1>0 的最大整数(注意:这些系数已经可以组合出 1~n[i]内的所有数字)。例如,如果 n[i]为 13,就将这种物品分成系数分别为 1,2,4,6 的四件物品。

分成的这几件物品的系数和为 n[i],表明不可能取多于 n[i]件的第 i 种物品。另外这种方法也能保证对于 0..n[i]间的每一个整数,均可以用若干个系数的和表示,这个证明可以分 0..2^k-1 和 2^k..n[i]两段来分别讨论得出,并不难,希望你自己思考尝试一下。

这样就将第 i 种物品分成了 O(logn[i])种物品,将原问题转化为了复杂度为 O(V * Σlogn[i])的 01 背包问题,是很大的改进。

例 9.13 庆功会

【问题描述】

为了庆贺班级在校运动会上取得全校第一名成绩,班主任决定开一场庆功会,为此拨款购买奖品犒劳运动员。期望拨款金额能购买最大价值的奖品,可以补充他们的精力和体力。

【输入格式】

第 1 行两个数 n(n<=500),m(m<=6000),其中 n 代表希望购买的奖品的种数,m 表示拨款金额。

接下来 n 行,每行 3 个数,v、w、s 分别表示第 i 种奖品的价格、价值(价格与价值是不同的概念)和能购买的最大数量(买 0 件到 s 件均可),其中 v<=100,w<=1000,s<=10。

【输出格式】

一个数,表示此次购买能获得的最大的价值(注意!不是价格)。

【输入样例】

```
5 1000
80 20 4
40 50 9
30 50 7
40 30 6
20 20 1
```

【输出样例】

```
1040
```

【解法一】

【算法分析】朴素算法

【参考程序】

```
#include<cstdio>
using namespace std;
int v[6002], w[6002], s[6002];
int f[6002];
int n, m;
int max(int x,int y)
```

```
{
    if (x < y) return y;
      else return x;
}
int main()
{
    scanf("%d%d",&n,&m);
    for (int i=1; i<=n; i++)
      scanf("%d%d%d",&v[i],&w[i],&s[i]);
    for (int i=1; i<=n; i++)
      for (int j=m; j>=0; j--)
        for (int k=0; k<=s[i]; k++)
          {
            if (j-k*v[i]<0) break;
            f[j]=max(f[j],f[j-k*v[i]]+k*w[i]);
          }
    printf("%d",f[m]);
    return 0;
}
```

【解法二】

【算法分析】

进行二进制优化，转换为 01 背包。

【参考程序】

```
#include<cstdio>
int v[10001],w[10001];
int f[6001];
int n,m,n1;
int max(int a,int b)
{
    return a>b? a : b;      //这句话等于:if (a>b) return a; else return b;
}
int main()
{
    scanf("%d%d",&n,&m);
    for(int i=1;i<=n;i++)
    {
      int x,y,s,t=1;
      scanf("%d%d%d",&x,&y,&s);
      while (s>=t)
```

```
        {
          v[++nl]=x*t;  //相当于 nl++;  v[nl]=x*t;
          w[nl]=y*t;
          s-=t;
          t*=2;
        }
        v[++nl]=x*s;
        w[nl]=y*s;      //把 s 以 2 的指数分堆:1,2,4,…,2^(k-1),s-2^k+1,
      }
      for(int i=1;i<=nl;i++)
        for(int j=m;j>=v[i];j--)
          f[j]=max(f[j],f[j-v[i]]+w[i]);
      printf("%d\n",f[m]);
      getchar();                          //起暂停作用,便于观察输出结果
      return 0;
}
```

小结:

这里我们看到了将一个算法的复杂度由 O(V * Σn[i])改进到 O(V * Σlogn[i])的过程,还知道了存在应用超出 NOIP 范围的知识的 O(VN)算法。希望你特别注意"拆分物品"的思想和方法,自己证明一下它的正确性,并用尽量简洁的程序来实现。

四、混合三种背包问题

问题:

如果将 01 背包、完全背包、多重背包混合起来。也就是说,有的物品只可以取一次(01 背包),有的物品可以取无限次(完全背包),有的物品可以取的次数有一个上限(多重背包)。应该怎么求解呢?

01 背包与完全背包的混合:

考虑到在 01 背包和完全背包中最后给出的伪代码只有一处不同,故如果只有两类物品:一类物品只能取一次,另一类物品可以取无限次,那么只需在对每个物品应用转移方程时,根据物品的类别选用顺序或逆序的循环即可,复杂度是 O(VN)。

伪代码如下:

```
for i=1..N
  if 第 i 件物品是 01 背包
    for v=V..0
      f[v]=max{f[v],f[v-w[i]]+c[i]};
  else if 第 i 件物品是完全背包
    for v=0..V
      f[v]=max{f[v],f[v-w[i]]+c[i]};
```

再加上多重背包

如果再加上有的物品最多可以取有限次，那么原则上也可以给出 O(VN)的解法：遇到多重背包类型的物品用单调队列解即可。但如果不考虑超过 Noip 范围的算法的话，用多重背包中将每个这类物品分成 O(log n[i])个 01 背包的物品的方法也已经很优了。

例 9.14　混合背包

【问题描述】

一个旅行者有一个最多能装 V 公斤的背包，现在有 n 件物品，它们的重量分别是 W1，W2，…，Wn，它们的价值分别为 C1，C2，…，Cn。有的物品只可以取一次(01 背包)，有的物品可以取无限次(完全背包)，有的物品可以取的次数有一个上限(多重背包)。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

【输入格式】

第 1 行：两个整数，M(背包容量，M<=200)，N(物品数量，N<=30)；

第 2 至 N+1 行：每行三个整数 Wi，Ci，Pi，前两个整数分别表示每个物品的重量，价值，第三个整数若为 0，则说明此物品可以购买无数件；若为其他数字，则为此物品可购买的最多件数(Pi)。

【输出格式】

仅一行，一个数，表示最大总价值。

【输入样例】

```
10 3
2 1 0
3 3 1
4 5 4
```

【输出样例】

```
11
```

【样例说明】

选第一件物品 1 件和第三件物品 2 件。

【参考程序】

```
#include<cstdio>
using namespace std;
int m, n;
int w[31], c[31], p[31];
int f[201];
int max(int x,int y)
{
    if (x < y) return y;
      else return x;
}
int main()
{
  scanf("%d%d",&m,&n);
  for (int i=1; i<=n; i++)
    scanf("%d%d%d",&w[i],&c[i],&p[i]);
  for (int i=1; i<=n; i++)
    if (p[i]==0)                    //完全背包
```

```
    {
      for (int j=w[i]; j <=m; j++)
        f[j]=max(f[j], f[j-w[i]]+c[i]);
    }
    else
    {
      for (int j=1; j <=p[i]; j++) //01 背包和多重背包
        for (int k=m; k >=w[i]; k--)
          f[k]=max(f[k],f[k-w[i]]+c[i]);
    }
  printf("%d ",f[m]);
  return 0;
}
```

小结:

有人说,困难的题目都是由简单的题目叠加而来的。这句话是否是公理暂且存之不论,但它在本讲中已经得到了充分的体现。本来01背包、完全背包、多重背包都不是什么难题,但将它们简单地组合起来以后就得到了这样一道一定能吓倒不少人的题目。但只要基础扎实,领会三种基本背包问题的思想,就可以做到把困难的题目拆分成简单的题目来解决。

五、二维费用的背包问题

问题:

二维费用的背包问题是指:对于每件物品,具有两种不同的费用;选择这件物品必须同时付出这两种代价;对于每种代价都有一个可付出的最大值(背包容量)。问怎样选择物品可以得到最大的价值。设这两种代价分别为代价1和代价2,第i件物品所需的两种代价分别为a[i]和b[i]。两种代价可付出的最大值(两种背包容量)分别为V和U。物品的价值为c[i]。

算法:

费用加了一维,只需状态也加一维即可。设f[i][v][u]表示前i件物品付出两种代价分别为v和u时可获得的最大价值。

状态转移方程就是:f[i][v][u]=max{f[i-1][v][u],f[i-1][v-a[i]][u-b[i]]+c[i]}。如前述方法,可以只使用二维的数组:当每件物品只可以取一次时变量v和u采用逆序的循环,当物品有如完全背包问题时采用顺序的循环。当物品有如多重背包问题时拆分物品。

物品总个数的限制:

有时,"二维费用"的条件是以这样一种隐含的方式给出的:最多只能取M件物品。这事实上相当于每件物品多了一种"件数"的费用,每个物品的件数费用均为1,可以付出的最大件数费用为M。换句话说,设f[v][m]表示付出费用v、最多选m件时可得到的最大价值,则根据物品的类型(01、完全、多重)用不同的方法循环更新,最后在f[0..V][0..M]范围内寻找答案。

另外，如果要求“恰取 M 件物品”，则在 f[0..V][M]范围内寻找答案。

例 9.15　潜水员

【问题描述】

潜水员为了潜水要使用特殊的装备。他有一个带 2 种气体的气缸：一个为氧气，一个为氮气。让潜水员下潜的深度需要各种的数量的氧和氮。潜水员有一定数量的气缸。每个气缸都有重量和气体容量。潜水员为了完成他的工作需要特定数量的氧和氮。他完成工作所需气缸的总重的最低限度的是多少？

例如：潜水员有 5 个气缸。每行三个数字为：氧、氮的(升)量和气缸的重量：

3 36 120

10 25 129

5 50 250

1 45 130

4 20 119

如果潜水员需要 5 升的氧和 60 升的氮则总重最小为 249(1,2 或者 4,5 号气缸)。

你的任务就是计算潜水员为了完成他的工作需要的气缸的重量的最低值。

【输入格式】

第 1 行有 2 整数 m,n(1<=m<=21,1<=n<=79)。它们表示氧、氮各自需要的量。

第 2 行为整数 k(1<=n<=1000)表示气缸的个数。

此后的 k 行，每行包括 ai,bi,ci(1<=ai<=21,1<=bi<=79,1<=ci<=800)3 个整数。这些各自是：第 i 个气缸里的氧和氮的容量及汽缸重量。

【输出格式】

仅一行包含一个整数，为潜水员完成工作所需的气缸的重量总和的最低值。

【输入样例】

```
5 60
5
3 36 120
10 25 129
5 50 250
1 45 130
4 20 119
```

【输出样例】

```
249
```

【参考程序】

```
#include<cstdio>
#include<cstring>                    //初始化 memset 要用到
using namespace std;
int v, u, k;
int a[1001], b[1001], c[1001];
int f[101][101];
int main()
{
    memset(f,127,sizeof(f));         //初始化为一个很大的正整数
```

```
    f[0][0]=0;
    scanf("%d%d%d",&v,&u,&k);
    for (int i=1; i<=k; i++)
        scanf("%d%d%d",&a[i],&b[i],&c[i]);
    for (int i=1; i<=k; i++)
      for (int j=v; j>=0; j--)
        for (int l=u; l>=0; l--)
        {
          int t1=j+a[i],t2=l+b[i];
          if (t1 > v)  t1=v;      //若氮、氧含量超过需求,可直接用需求量代换
          if (t2> u)  t2=u;      //不影响最优解
          if (f[t1][t2] > f[j][l]+c[i])  f[t1][t2]=f[j][l]+c[i];
        }
    printf("%d",f[v][u]);
    return 0;
}
```

小结:

事实上,当发现由熟悉的动态规划题目变形得来的题目时,在原来的状态中加一维以满足新的限制是一种比较通用的方法。希望你能从本讲中初步体会到这种方法。

六、分组的背包问题

问题:

有 N 件物品和一个容量为 V 的背包。第 i 件物品的费用是 w[i],价值是 c[i]。这些物品被划分为若干组,每组中的物品互相冲突,最多选一件。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量,且价值总和最大。

算法:

这个问题变成了每组物品有若干种策略:是选择本组的某一件,还是一件都不选。也就是说设 f[k][v]表示前 k 组物品花费费用 v 能取得的最大权值,则有 f[k][v]=max{f[k-1][v],f[k-1][v-w[i]]+c[i]|物品 i 属于第 k 组}。

使用一维数组的伪代码如下:

```
for 所有的组 k
  for v=V..0
    for 所有的 i 属于组 k
      f[v]=max{f[v],f[v-w[i]]+c[i]}
```

注意这里的三层循环的顺序,"for v=V..0"这一层循环必须在"for 所有的 i 属于组 k"之外。这样才能保证每一组内的物品最多只有一个会被添加到背包中。

另外,显然可以对每组中的物品应用完全背包中"一个简单有效的优化"。

例 9.16 分组背包

【问题描述】

一个旅行者有一个最多能装 V 公斤的背包，现在有 n 件物品，它们的重量分别是 W1，W2，…，Wn，它们的价值分别为 C1，C2，…，Cn。这些物品被划分为若干组，每组中的物品互相冲突，最多选一件。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

【输入格式】

第 1 行：三个整数，V（背包容量，V<=200），N（物品数量，N<=30）和 T（最大组号，T<=10）；

第 2 至 N+1 行：每行三个整数 Wi，Ci，P，表示每个物品的重量、价值、所属组号。

【输出格式】

仅一行，一个数，表示最大总价值。

【输入样例】

```
10 6 3
2 1 1
3 3 1
4 8 2
6 9 2
2 8 3
3 9 3
```

【输出样例】

```
20
```

【参考程序】

```
#include<cstdio>
using namespace std;
int v, n, t;
int w[31], c[31];
int a[11][32], f[201];
int main()
{
    scanf("%d%d%d",&v,&n,&t);
    for (int i=1; i<=n; i++)
    {
      int p;
      scanf("%d%d%d",&w[i],&c[i],&p);
      a[p][++a[p][0]]=i;
    }
    for (int k=1; k<=t; k++)
      for (int j=v; j>=0; j--)
        for (int i=1; i<=a[k][0]; i++)
          if (j>=w[a[k][i]])
          {
            int tmp=a[k][i];
            if (f[j]<f[j-w[tmp]]+c[tmp])
```

```
                f[j]=f[j-w[tmp]]+c[tmp];
            }
        printf("%d ",f[v]);
        return 0;
    }
```

小结：

分组的背包问题将彼此互斥的若干物品称为一个组，这建立了一个很好的模型。不少背包问题的变形都可以转化为分组的背包问题。

七、有依赖的背包问题

问题：

这种背包问题的物品间存在某种“依赖”的关系。也就是说，i 依赖于 j，表示若选物品 i，则必须选物品 j。为了简化起见，我们先设没有某个物品既依赖于别的物品，又被别的物品所依赖；另外，没有某件物品同时依赖多件物品。

算法：

这个问题由 Noip2006 金明的预算方案一题扩展而来。遵从该题的提法，将不依赖于别的物品的物品称为“主件”，依赖于某主件的物品称为“附件”。由这个问题的简化条件可知所有的物品由若干主件和依赖于每个主件的一个附件集合组成。

按照背包问题的一般思路，仅考虑一个主件和它的附件集合。可是，可用的策略非常多，包括：一个也不选，仅选择主件，选择主件后再选择一个附件，选择主件后再选择两个附件…无法用状态转移方程来表示如此多的策略。（事实上，设有 n 个附件，则策略有 2^n+1 个，为指数级。）

考虑到所有这些策略都是互斥的（也就是说，你只能选择一种策略），所以一个主件和它的附件集合实际上对应于分组的背包中的一个物品组，每个选择了主件又选择了若干个附件的策略对应于这个物品组中的一个物品，其费用和价值都是这个策略中的物品的值的和。但仅仅是这一步转化并不能给出一个好的算法，因为物品组中的物品还是像原问题的策略一样多。

再考虑分组的背包中的一句话：可以对每组中的物品应用完全背包中“一个简单有效的优化”。这提示我们，对于一个物品组中的物品，所有费用相同的物品只留一个价值最大的，不影响结果。所以，我们可以对主件 i 的“附件集合”先进行一次 01 背包，得到费用依次为 0..V-w[i]所有这些值时相应的最大价值 f[0..V-w[i]]。那么这个主件及它的附件集合相当于 V-w[i]+1 个物品的物品组，其中费用为 w[i]+k 的物品的价值为 f[k]+c[i]。也就是说原来指数级的策略中有很多策略都是冗余的，通过一次 01 背包后，将主件 i 转化为 V-w[i]+1 个物品的物品组，就可以直接应用分组的背包的算法解决问题了。

更一般的问题是：依赖关系以图论中“森林”的形式给出（森林即多叉树的集合），也就是说，主件的附件仍然可以具有自己的附件集合，限制只是每个物品最多只依赖于一个物品（只有一个主件）且不出现循环依赖。

解决这个问题仍然可以用将每个主件及其附件集合转化为物品组的方式。唯一不同的是，由于附件可能还有附件，就不能将每个附件都看作一个一般的 01 背包中的物品了。若

这个附件也有附件集合，则它必定要被先转化为物品组，然后用分组的背包问题解出主件及其附件集合所对应的附件组中各个费用的附件所对应的价值。

事实上，这是一种树形 DP，其特点是每个父节点都需要对它的各个儿子的属性进行一次 DP 以求得自己的相关属性。这已经触及到了“泛化物品”的思想。看完后，你会发现这个“依赖关系树”每一个子树都等价于一件泛化物品，求某节点为根的子树对应的泛化物品相当于求其所有儿子的对应的泛化物品之和。

小结：

Noip2006 的那道背包问题，通过引入“物品组”和“依赖”的概念可以加深对这题的理解，还可以解决它的推广问题。用物品组的思想考虑那题中极其特殊的依赖关系：物品不能既作主件又作附件，每个主件最多有两个附件，可以发现一个主件和它的两个附件等价于一个由四个物品组成的物品组，这便揭示了问题的某种本质。

八、背包问题的方案总数

对于一个给定了背包容量、物品费用、物品间相互关系（分组、依赖等）的背包问题，除了再给定每个物品的价值后求可得到的最大价值外，还可以得到装满背包或将背包装至某一指定容量的方案总数。

对于这类改变问法的问题，一般只需将状态转移方程中的 max 改成 sum 即可。例如若每件物品均是 01 背包中的物品，转移方程即为 f[i][v]=sum{f[i-1][v],f[i-1][v-w[i]]+c[i]}，初始条件 f[0][0]=1。

事实上，这样做可行的原因在于状态转移方程已经考察了所有可能的背包组成方案。

例 9.17　货币系统

【问题描述】

给你一个 n 种面值的货币系统，求组成面值为 m 的货币有多少种方案。样例：设 n=3，m=10，要求输入和输出的格式如下：

【输入样例】

```
3 10   //3 种面值组成面值为 10 的方案
1      //面值 1
2      //面值 2
5      //面值 5
```

【输出样例】

```
10     //有 10 种方案
```

【算法分析 1】

设 f[j]表示面值为 j 的最大方案数，如果 f[j-k*a[i]]!=0，则 f[j]=f[j]+f[j-k*a[i]]，当 1<=i<=n，m>=j>=a[i]，1<=k<=j / a[i]。

【参考程序 1】

```
#include<cstdio>
int m, n;
int a[1001];
long long f[10001];                    //注意要用 long long
int main()
{
```

```
    scanf("%d%d",&n,&m);          //n种面值的货币,组成面值为m
    for (int i=1; i<=n; i++)
      scanf("%d",&a[i]);          //输入每一种面值
    f[0]=1;
    for (int i=1; i<=n; i++)
      for (int j=m; j>=a[i]; j--)//f[j]表示面值为j的总方案数
        for (int k=1; k<=j/a[i]; k++)
          f[j]+=f[j-k*a[i]];
    printf("%lld",f[m]);          // f[m]为最优解
    return 0;
}
```

【算法分析2】

设f[j]表示面值为j的总方案数,如果f[j-a[i]]!=0,则f[j]=f[j]+f[j-a[i]],1<=i<=n,a[i]<=j<=m。

【参考程序2】

```
#include<cstdio>
using namespace std;
int n, m;
int a[101];
long long f[10001];
int main()
{
  scanf("%d%d",&n,&m);
  for (int i=1; i<=n; i++)
    scanf("%d",&a[i]);
  f[0]=1;
  for (int i=1; i<=n; i++)
    for (int j=a[i]; j<=m; j++)
      f[j]+=f[j-a[i]];
  printf("%lld",f[m]);
  return 0;
}
```

小结:

显然,这里不可能穷尽背包类动态规划问题所有的问法。甚至还存在一类将背包类动态规划问题与其他领域(例如数论、图论)结合起来的问题,在这篇论背包问题的专文中也不会论及。但只要深刻领会前述所有类别的背包问题的思路和状态转移方程,遇到其他的变形问法,只要题目难度还属于Noip,应该也不难想出算法。

触类旁通、举一反三,应该也是一个OIer应有的品质吧!

【上机练习】

1. 采药【2.6 基本算法之动态规划 1775】

辰辰是个很有潜能、天资聪颖的孩子，他的梦想是称为世界上最伟大的医师。为此，他想拜附近最有威望的医师为师。医师为了判断他的资质，给他出了一个难题。医师把他带到个到处都是草药的山洞里对他说："孩子，这个山洞里有一些不同的草药，采每一株都需要一些时间，每一株也有它自身的价值。我会给你一段时间，在这段时间里，你可以采到一些草药。如果你是一个聪明的孩子，你应该可以让采到的草药的总价值最大。"

如果你是辰辰，你能完成这个任务吗？

输入：

输入的第一行有两个整数 T(1<=T<=1000)和 M(1<=M<=100)，T 代表总共能够用来采药的时间，M 代表山洞里的草药的数目。接下来的 M 行每行包括两个在 1 到 100 之间(包括 1 和 100)的的整数，分别表示采摘某株草药的时间和这株草药的价值。

输出：

输出只包括一行，这一行只包含一个整数，表示在规定的时间内，可以采到的草药的最大总价值。

样例输入：

```
70 3
71 100
69 1
1  2
```

样例输出：

```
3
```

2. 数字组合【2.6 基本算法之动态规划 2985】

有 n 个正整数，找出其中和为 t(t 也是正整数)的可能的组合方式。如：

n=5，5 个数分别为 1，2，3，4，5，t=5；

那么可能的组合有 5=1+4 和 5=2+3 和 5=5 三种组合方式。

输入：

输入的第一行是两个正整数 n 和 t，用空格隔开，其中 1<=n<=20，表示正整数的个数，t 为要求的和(1<=t<=1000)

接下来的一行是 n 个正整数，用空格隔开。

输出：

和为 t 的不同的组合方式的数目。

样例输入：

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

样例输出：

```
3
```

3. 宠物小精灵之收服【2.6 基本算法之动态规划 4978】

宠物小精灵是一部讲述小智和他的搭档皮卡丘一起冒险的故事。

一天,小智和皮卡丘来到了小精灵狩猎场,里面有很多珍贵的野生宠物小精灵。小智也想收服其中的一些小精灵。然而,野生的小精灵并不那么容易被收服。对于每一个野生小精灵而言,小智可能需要使用很多个精灵球才能收服它,而在收服过程中,野生小精灵也会对皮卡丘造成一定的伤害(从而减少皮卡丘的体力)。当皮卡丘的体力小于等于 0 时,小智就必须结束狩猎(因为他需要给皮卡丘疗伤),而使得皮卡丘体力小于等于 0 的野生小精灵也不会被小智收服。当小智的精灵球用完时,狩猎也宣告结束。

我们假设小智遇到野生小精灵时有两个选择:收服它,或者离开它。如果小智选择了收服,那么一定会扔出能够收服该小精灵的精灵球,而皮卡丘也一定会受到相应的伤害;如果选择离开它,那么小智不会损失精灵球,皮卡丘也不会损失体力。

小智的目标有两个:主要目标是收服尽可能多的野生小精灵;如果可以收服的小精灵数量一样,小智希望皮卡丘受到的伤害越小(剩余体力越大),因为他们还要继续冒险。

现在已知小智的精灵球数量和皮卡丘的初始体力,已知每一个小精灵需要的用于收服的精灵球数目和它在被收服过程中会对皮卡丘造成的伤害数目。请问,小智该如何选择收服哪些小精灵以达到他的目标呢?

输入:

输入数据的第一行包含三个整数:N(0<N<1000),M(0<M<500),K(0<K<100),分别代表小智的精灵球数量、皮卡丘初始的体力值、野生小精灵的数量。

之后的 K 行,每一行代表一个野生小精灵,包括两个整数:收服该小精灵需要的精灵球的数量,以及收服过程中对皮卡丘造成的伤害。

输出:

输出为一行,包含两个整数:C,R,分别表示最多收服 C 个小精灵,以及收服 C 个小精灵时皮卡丘的剩余体力值最多为 R。

样例输入

样例输入 1:

```
10 100 5
7 10
2 40
2 50
1 20
4 20
```

样例输入 2:

```
10 100 5
8 110
12 10
20 10
5 200
1 110
```

样例输出

样例输出 1：

3 30

样例输出 2：

0 100

提示：

对于样例输入 1：小智选择：(7,10) (2,40) (1,20) 这样小智一共收服了 3 个小精灵，皮卡丘受到了 70 点伤害，剩余 100－70＝30 点体力。所以输出 3 30

对于样例输入 2：小智一个小精灵都没法收服，皮卡丘也不会收到任何伤害，所以输出 0 100

4. 买书【2.6 基本算法之动态规划 6049】

小明手里有 n 元钱全部用来买书，书的价格为 10 元，20 元，50 元，100 元。

问小明有多少种买书方案？（每种书可购买多本）

输入：

一个整数 n，代表总共钱数。(0 ＜＝n ＜＝1000)

输出：

一个整数，代表选择方案种数

样例输入

样例输入 1：

20

样例输入 2：

15

样例输入 3：

0

样例输出

样例输出 1：

2

样例输出 2：

0

样例输出 3：

0

5. Charm Bracelet【2.6 基本算法之动态规划 7113】

题意：经典 0—1 背包问题，有 n 个物品，编号为 i 的物品的重量为 w[i]，价值为 c[i]，现在要从这些物品中选一些物品装到一个容量为 m 的背包中，使得背包内物体在总重量不超过 m 的前提下价值尽量大。

输入：

第 1 行：两个整数，n(物品数量，n＜＝3500)和 m(背包容量，m＜＝12880)。

第 2..n＋1 行：：每行二个整数 w[i]，c[i]，表示每个物品的重量和价值。

输出：

仅一行，一个数，表示最大总价值。

样例输入：

4 6

1 4

2 6

3 12

2 7

样例输出：

23

6. 装箱问题【2.6 基本算法之动态规划 8785】

有一个箱子容量为 V(正整数,0<=v<=20000),同时有 n 个物品(0<n<=30),每个物品有一个体积(正整数)。

要求 n 个物品中,任取若干个装入箱内,使箱子的剩余空间为最小。

输入：

第一行是一个整数 V,表示箱子容量。

第二行是一个整数 n,表示物品数。

接下来 n 行,每行一个正整数(不超过 10000),分别表示这 n 个物品的各自体积。

输出：

一个整数,表示箱子剩余空间。

样例输入：

24

6

8

3

12

7

9

7

样例输出：

0

7. 开餐馆【2.6 基本算法之动态规划 6045】

信息学院的同学小明毕业之后打算创业开餐馆.现在共有 n 个地点可供选择。小明打算从中选择合适的位置开设一些餐馆。这 n 个地点排列在同一条直线上。我们用一个整数序列 $m_1, m_2, \cdots m_n$ 来表示他们的相对位置。由于地段关系,开餐馆的利润会有所不同。我们用 p_i 表示在 m_i 处开餐馆的利润。为了避免自己的餐馆的内部竞争,餐馆之间的距离必须大于 k。请你帮助小明选择一个总利润最大的方案。

输入：

输入第一行是整数 T(1<=T<=1000),表明有 T 组测试数据。紧接着有 T 组连续的测试。每组测试数据有 3 行,

第 1 行:地点总数 n(n<100),距离限制 k(k>0 && k<1000)。

第 2 行:n 个地点的位置 m1 , m2, $\cdots m_n$(1000000>mi>0 且为整数,升序排列)。

第 3 行:n 个地点的餐馆利润 p1,p2,$\cdots p_n$(1000>pi>0 且为整数)。

输出：

对于每组测试数据可能的最大利润。

样例输入：

2

3 11

1 2 15

10 2 30

3 16

1 2 15

10 2 30

样例输出：

40

30

第三节　动态规划经典题

例 9.18　合并石子

【问题描述】

在一个操场上一排地摆放着 N 堆石子。现要将石子有次序地合并成一堆。规定每次只能选相邻的 2 堆石子合并成新的一堆，并将新的一堆石子数记为该次合并的得分。

【编程任务】

试设计一个程序，计算出将 N 堆石子合并成一堆的最小得分。

【输入格式】

第 1 行为一个正整数 N (2<=N<=100)；

以下 N 行，每行一个正整数，小于 10000，分别表示第 i 堆石子的个数(1<=i<=N)。

【输出格式】

一个正整数，即最小得分。

【输入样例】

7

13

7

8

16

21

4

18

【输出样例】

239

【算法分析】

s[i]表示前 i 堆石头的数量总和，f[i][j]表示把第 i 堆到第 j 堆的石头合并成一堆的最优值。

```
for (i=n-1; i>=1; i--)
  for (j=i+1; j<=n; j++)
```

```
    for (k=i; k<=j-1; k++)
      f[i][j]=min(f[i][j],f[i][k]+f[k+1][j]+s[j]-s[i-1]);
```

输出 f[1][n]

【参考程序】

```
#include<cstdio>
#include<cstring>
int min(int a,int b)
{
    return a > b ? b:a;      // 三目运算符,相当于 if(a>b) return b; else return a;
}
int f[101][101];
int s[101];
int n,i,j,k,x;
int main()
{
    scanf("%d",&n);
    for (i=1; i<=n; i++)
    {
      scanf("%d",&x);
      s[i]=s[i-1]+x;
    }
    memset(f,127/3,sizeof(f));          //赋值 127 是很大的正数,若无/3 后面的相加
    for (i=1; i<=n; i++) f[i][i]=0;   //可能超出 int 的范围
    for (i=n-1; i>=1; i--)
      for (j=i+1; j<=n; j++)
          for (k=i; k<=j-1; k++)
            f[i][j]=min(f[i][j],f[i][k]+f[k+1][j]+s[j]-s[i-1]);
    printf("%d\n",f[1][n]);
    return 0;
}
```

例 9.19 乘积最大[2.6 基本算法之动态规划 8782]

【问题描述】

2000 年是国际数学联盟确定的“2000——世界数学年”,又恰逢我国著名数学家华罗庚先生诞辰 90 周年。在华罗庚先生的家乡江苏金坛,组织了一场别开生面的数学智力竞赛的活动,你的一个好朋友 XZ 也有幸得以参加。活动中,主持人给所有参加活动的选手出了这样一道题目:

设有一个长度为 N 的数字串,要求选手使用 K 个乘号将它分成 K+1 个部分,找出一种分法,使得这 K+1 个部分的乘积最大。

同时，为了帮助选手能够正确理解题意，主持人还举了如下的一个例子：

有一个数字串：312，当 N=3，K=1 时会有以下两种分法：

(1)3 * 12=36

(2)31 * 2=62

这时，符合题目要求的结果是：31 * 2=62。

现在，请你帮助你的好朋友 XZ 设计一个程序，求得正确的答案。

【输入格式】

第 1 行共有 2 个自然数 N，K(6<=N<=10，1<=K<=6)。

第 2 行是一个长度为 N 的数字串。

【输出格式】

输出所求得的最大乘积(一个自然数)。

【输入样例】　　【输出样例】

4 2　　62

1231

【算法分析】

此题满足动态规划法的求解标准，我们把它按插入的乘号数来划分阶段，若插入 k 个乘号，可把问题看做是 k 个阶段的决策问题。设 f[i][k]表示在前 i 位数中插入 k 个乘号所得的最大值，a[j][i]表示从第 j 位到第 i 位所组成的自然数。用 f[i][k]存储阶段 k 的每一个状态，可以得状态转移方程：

f[i][k]=max{f[j][k-1] * a[j+1][i]}(k<=j<i)

边界值：

f[j][0]=a[1][j] (1<=j<=n)

根据状态转移方程，我们就很容易写出动态规划程序：

```
for (k=1; k<=k1; k++)                    //k1 为乘号个数
  for (i=k+1; i<=n; i++)
    for (j=k; j<i; j++)
      f[i][k]=max(f[i][k],f[j][k-1] * a[j+1][i]);
```

【参考程序】

```
#include<cstdio>
long long a[11][11],f[11][11];
long long s;
int n,i,k,k1,j;
int max(int a, int b)
{
    return a>b? a:b;
}
int main()
{
```

```
    scanf("%d%d",&n,&k1);
    scanf("%lld",&s);
    for (i=n; i>=1; i--)
    {
      a[i][i]=s%10;
      s/=10;
    }
    for (i=2; i<=n; i++)
        for (j=i-1;j>=1;j--)
          a[j][i]=a[j][i-1]*10+a[i][i];
    for (i=1; i<=n; i++)              //预处理不加乘号的情况
      f[i][0]=a[1][i];
    for (k=1; k<=k1; k++)
      for (i=k+1; i<=n; i++)
        for (j=k; j<i; j++)
          f[i][k]=max(f[i][k],f[j][k-1]*a[j+1][i]);
    printf("%lld\n",f[n][k1]);
    return 0;
}
```

例 9.20　编辑距离

【问题描述】

设 A 和 B 是两个字符串。我们要用最少的字符操作次数,将字符串 A 转换为字符串 B。这里所说的字符操作共有三种:

(1) 删除一个字符;

(2) 插入一个字符;

(3) 将一个字符改为另一个字符。

【编程任务】

对任意的两个字符串 A 和 B,计算出将字符串 A 变换为字符串 B 所用的最少字符操作次数。

【输入格式】

第 1 行为字符串 A;第 2 行为字符串 B;字符串 A 和 B 的长度均小于 200。

【输出格式】

只有一个正整数,为最少字符操作次数。

【输入样例】　　【输出样例】

sfdqxbw　　4

gfdgw

【算法分析】

状态:f[i][j]记录 ai 与 bj 的最优编辑距离。

结果:f[m][n],其中 m、n 分别是 a、b 的串长。

初值:b 串空,要删 m(a 串长)个字符;a 串空,要插 n(b 串长)个字符。

转移方程:当 a[i]=b[j]时,f[i][j]=f[i-1][j-1],

否则,f[i][j]=min(f[i-1][j-1]+1,f[i][j-1]+1,f[i-1][j]+1)

说明:f[i-1][j-1]+1:改 a[i]为 b[j];

f[i][j-1]+1:a[i]后插入 b[j-1];

f[i-1][j]+1:删 a[i]。

【参考程序】

```
#include<cstdio>
#include<cstring>
int min(int a,int b){return a<b? a:b;}
int f[202][202];
char s1[202], s2[202];
int i,j,k,m,n;
int main()
{
    scanf("%s%s",s1,s2);
    m=strlen(s1);
    n=strlen(s2);
    for (i=1; i<=m; i++) f[i][0]=i;
                          //到 i 位置为止把字符串 A 的内容全部删除
    for (i=1; i<=n; i++) f[0][i]=i;
                          //在开头给字符串 A 添上和 B 到 i 位置相同的字符
    for (i=1; i<=m; i++)
      for (j=1; j<=n; j++)
        if (s1[i-1]==s2[j-1]) f[i][j]=f[i-1][j-1];
          else f[i][j]=min(min(f[i-1][j],f[i][j-1]),f[i-1][j-1])+1;
    printf("%d\n",f[m][n]);
    return 0;
}
```

例 9.21　方格取数[2.6 基本算法之动态规划 8786]

【问题描述】

设有 N*N 的方格中,我们在其中的某些方格中填入正整数,而其他的方格中则放入数字 0。如下表所示:

A

0	0	0	0	0	0	0	0
0	0	13	0	0	6	0	0
0	0	0	0	7	0	0	0
0	0	0	14	0	0	0	0

0	21	0	0	0	4	0	0
0	0	15	0	0	0	0	0
0	14	0	0	0	0	0	0
0	0	0	0	0	0	0	0

B

某人从图中的左上角 A 出发,可以向下行走,也可以向右行走,直到到达右下角的 B 点。在走过的路上,他可以取走方格中的数(取走后的方格中将变为数字 0)。

【编程任务】

此人从 A 点到 B 点共走了两次,试找出两条这样的路径,使得取得的数字和为最大。

【输入格式】

输入文件 pane.in 第 1 行为一个整数 N(N<=10),表示 N * N 的方格图。

接下来的每行有三个整数,第一个为行号数,第二个为列号数,第三个为在该行、该列上所放的数。一行 0 0 0 表示结束。

【输出格式】

输出文件 pane.out 包含一个整数,表示两条路径上取得的最大的和。

【输入样例】

```
8
2 3 13
2 6 6
3 5 7
4 4 14
5 2 21
5 6 4
6 3 15
7 2 14
0 0 0
```

【输出样例】

```
67
```

【算法分析】

一个四重循环枚举两条路分别走到的位置。由于每个点均从上或左继承而来,故内部有四个 if,分别表示两个点从上上、上左、左上、左左继承来时,加上当前两个点所取得的最大值。a[i][j]表示(i,j)格上的值,sum[i][j][h][k]表示第一条路走到(i,j),第二条路走到(h,k)时的最优解。例:sum[i][j][h][k]=max{sum[i][j][h][k], sum[i-1][j][h-1][k]+a[i][j]+a[h][k]},表示两点均从上面位置走来。

当(i,j) <> (h,k)时:

sum[i][j][h][k]=max{sum[i-1][j][h-1][k],sum[i][j-1][h][k-1],sum[i-1][j][h][k-1],sum[i][j-1][h-1][k]}+a[i][j]+a[h][k];

当(i,j)=(h,k)时:

sum[i][j][h][k]=max{sum[i-1][j][h-1][k],sum[i][j-1][h][k-1],sum[i-1][j][h][k-1],sum[i][j-1][h-1][k]}+a[i][j]。

【参考程序】

```
#include<cstdio>
int max(int a,int b){return a>b? a : b;}
int a[51][51];
int sum[51][51][51][51];
int n,i,j,h,k,x,y,z;
int main()
{
  scanf("%d%d%d%d",&n,&x,&y,&z);
  while(x && y && z)      //C++中大于0的正整数都看作true,只有0看作false
  {
      a[x][y]=z;
      scanf("%d%d%d",&x,&y,&z);
  }
  for (i=1; i<=n; i++)
      for (j=1; j<=n; j++)
          for (h=1; h<=n; h++)
              for (k=1; k<=n; k++)
              {
                int tmp1=max(sum[i-1][j][h-1][k],sum[i][j-1][h][k-1]);
                int tmp2=max(sum[i-1][j][h][k-1],sum[i][j-1][h-1][k]);
                sum[i][j][h][k]=max(tmp1,tmp2)+a[i][j];
                if (i!=h && j!=k) sum[i][j][h][k]+=a[h][k];
              }
  printf("%d\n",sum[n][n][n][n]);
  return 0;
}
```

例 9.22　复制书稿

【问题描述】

现在要把 m 本有顺序的书分给 k 个人复制(抄写),每一个人的抄写速度都一样,一本书不允许给两个(或以上)的人抄写,分给每一个人的书,必须是连续的,比如不能把第一、第三和第四本书给同一个人抄写。

现在请你设计一种方案,使得复制时间最短。复制时间为抄写页数最多的人用去的时间。

【输入格式】

第 1 行两个整数 m,k;(k<=m<=500)

第 2 行 m 个整数,第 i 个整数表示第 i 本书的页数。

【输出格式】

共 k 行,每行两个整数,第 i 行表示第 i 个人抄写的书的起始编号和终止编号。k 行的起始编号应该从小到大排列,如果有多解,则尽可能让前面的人少抄写。

【输入样例】　　【输出样例】

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

【问题分析】

本题可以用动态规划解决,设 f(k,m)为前 m 本书交由 k 个人抄写,需要的最短时间,则状态转移方程为 $f(k,m)=\min\{\max\{f(k-1,i),\sum_{j=i+1}^{m}T_j\},i=1,2,\cdots,m-1\}$

动态规划求出的仅仅是最优值,如果要输出具体方案,还需根据动态规划计算得到的最优值,做一个贪心设计。具体来说,设最优值为 T,那么 k 个人,每个人抄写最多 T 页。从最后一本书开始按逆序将书分配给 k 人去抄写,从第 k 个人开始,如果他还能写,就给他;否则第 k 个人工作分配完毕,开始分配第 k-1 个人的工作;以后再是第 k-2 个、第 k-3 个、…直至第 1 个。一遍贪心结束后,具体的分配方案也就出来了。

【参考程序】

```
#include<iostream>
#include<cstdio>
#include<cstdlib>
using namespace std;
int max1(int,int);
int print(int,int);
int x,y,i,j,m,n,k,t,l;
int a[501],f[501][501],d[501];
int main()
{
  cin>>m>>k;
  for (i=0;i<=500;i++)
   for (j=0;j<=500;j++)
    f[i][j]=10000000;
  for (j=1;j<=m;j++)
    {
      cin>>a[j];              //输入 m 本书的页数
      d[j]=d[j-1]+a[j];       //d[j]为前 j 本书总的页数
      f[1][j]=d[j];           //f[1][j]为一个人抄前 j 本书所需时间
    }
  for (i=2;i<=k;i++)   //f[k][m]为前 m 本书交由 k 个人抄写,需要的最短时间
   for (j=1;j<=m;j++)
    for (l=1;l<=j-1;l++)
     if (max1(f[i-1][l],d[j]-d[l])<f[i][j])
      f[i][j]=max1(f[i-1][l],d[j]-d[l]);
  print(m,k);              //从第 k 个开始分配抄写方案
}
```

```
int max1(int x,int y)          //求 x 和 y 中最大值
{
  if (x>y) return x; else return y;
}
int print(int i,int j)         //递归输出抄写页数的方案
{
    int t,x;
    if (j==0) return 0;
    if (j==1)                  //第 1 个人抄写 1 到 i 本书
     {
        cout<<1<<" "<<i<<endl;
        return 0;
     }
    t=i;x=a[i];
    while (x+a[t-1]<=f[k][m])
                   //从最后一本书按逆序分配第 j 个人抄写,只要能写,就给他
     {
        x+=a[t-1];
        t--;
     }
    print(t-1,j-1);
                   //用递归过程给第 j-1 个人分配抄写方案,这时只有 t-1 本书了
    cout<<t<<" "<<i<<endl;
                   //递归返回时输出,因为递归过程是最后 1 个人先分配抄书
}
```

例 9.23　橱窗布置(flower)

【问题描述】

假设以最美观的方式布置花店的橱窗,有 F 束花,每束花的品种都不一样,同时,至少有同样数量的花瓶,被按顺序摆成一行,花瓶的位置是固定的,并从左到右,从 1 到 V 顺序编号,V 是花瓶的数目,编号为 1 的花瓶在最左边,编号为 V 的花瓶在最右边,花束可以移动,并且每束花用 1 到 F 的整数唯一标识,标识花束的整数决定了花束在花瓶中列的顺序即如果 I < J,则花束 I 必须放在花束 J 左边的花瓶中。

例如,假设杜鹃花的标识数为 1,秋海棠的标识数为 2,康乃馨的标识数为 3,所有的花束在放入花瓶时必须保持其标识数的顺序,即:杜鹃花必须放在秋海棠左边的花瓶中,秋海棠必须放在康乃馨左边的花瓶中。如果花瓶的数目大于花束的数目,则多余的花瓶必须空,即每个花瓶中只能放一束花。

每一个花瓶的形状和颜色也不相同,因此,当各个花瓶中放入不同的花束时会产生不同的美学效果,并以美学值(一个整数)来表示,空置花瓶的美学值为 0。在上述例子中,花瓶与花束的不同搭配所具有的美学值,可以用如下表格表示。

根据表格,杜鹃花放在花瓶 2 中,会显得非常好看,但若放在花瓶 4 中则显得很难看。

为取得最佳美学效果,必须在保持花束顺序的前提下,使花的摆放取得最大的美学值,如果具有最大美学值的摆放方式不止一种,则输出任何一种方案即可。题中数据满足下面条件:1<=F<=100,F<=V<=100,-50<=A_{IJ}<=50,其中A_{IJ}是花束I摆放在花瓶J中的美学值。输入整数F,V和矩阵(A_{IJ}),输出最大美学值和每束花摆放在各个花瓶中的花瓶编号。

	花瓶1	花瓶2	花瓶3	花瓶4	花瓶5
杜鹃花	7	23	-5	-24	16
秋海棠	5	21	-4	10	23
康乃馨	-21	5	-4	-20	20

【数据范围】

1<=F<=100,其中F为花束的数量,花束编号从1至F。

F<=V<=100,其中V是花瓶的数量。

-50<=A_{IJ}≤50,其中A_{IJ}是花束I在花瓶J中的美学值。

【输入格式】

输入文件是flower.in。

第1行包含两个数:F,V。

随后的F行中,每行包含V个整数,A_{IJ}即为输入文件中第(I+1)行中的第J个数。

【输出格式】

输出文件必须是名为flower.out的正文文件,文件应包含两行:

第1行是程序所产生摆放方式的美学值。

第2行必须用F个数表示摆放方式,即该行的第K个数表示花束K所在的花瓶的编号。

【输入样例】

```
3 5
7 23 -5 -24 16
5 21 -4  10 23
-21 5  -4 -20 20
```

【输出样例】

```
53
2 4 5
```

【解法一】

【算法分析】

问题实际就是给定F束花和V个花瓶,以及各束花放到不同花瓶中的美学值,要求你找出一种摆放的方案,使得在满足编号小的花放进编号小的花瓶中的条件下,美学值达到最大。

将问题进行转化,找出问题的原型。首先,看一下上述题目的样例数据表格。

将摆放方案的要求用表格表现出来,则摆放方案需要满足:每行选且只选一个数(花瓶);摆放方案的相邻两行中,下面一行的花瓶编号要大于上面一行的花瓶编号两个条件。这时可将问题转化为:给定一个数字表格,要求编程计算从顶行至底行的一条路径,使得这条路径所经过的数字总和最大(要求每行选且仅选一个数字)。同时,路径中相邻两行的数字,必须保证下一行数字的列数大于上一行数字的列数。

看到经过转化后的问题,发现问题与"数字三角形"问题十分相似,数字三角形问题的题意是:给定一个数字三角形,要求编程计算从顶至底的一条路径,使得路径所经过的数字总

和最大(要求每行选且仅选一个数字)。同时,路径中相邻两行的数字,必须保证下一行数字的列数与上一行数字的列数相等或者等于上一行数字的列数加1。

上例中已经知道:数字三角形中的经过数字之和最大的最佳路径,路径的每个中间点到最底层的路径必然也是最优的,可以用动态规划方法求解,对于"花店橱窗布置"问题经过转化后,也可采取同样的方法得出本题同样符合最优性原理。因此,可以对此题采用动态规划的方法。

【参考程序】

```
#include<iostream>
#include<cstring>
#include<cstdio>
using namespace std;
int main()
{
  int a[101][101],b[101][101],c[101][101],d[101];
                                  //a[i][j] 花束 i 放在花瓶 j 中的美学值
                                  //b[i][j] 前 i 束花放在前 j 个花瓶中的最优解
                                  //c[i][j] 在 b[i][j]的最优解中,花束 i-1 的位置
  int f,v,i,j,k,max;              //f,v 花束和花瓶的数目
  cin>>f>>v;
  for (i=1;i<=f;i++)
   for (j=1;j<=v;j++)
    cin>>a[i][j];
  memset(b,128,sizeof(b));        //这样处理,可以保证每束花都放进花瓶
  for (i=1;i<=v-f+1;i++)          //初始化第 1 束花放在第 i 个花瓶的情况
   b[1][i]=a[1][i];
  for (i=2;i<=f;i++)
   for (j=i;j<=v-f+i;j++)
    for (k=i-1;k<=j-1;k++)                //枚举花束 i-1 的位置
      if (b[i-1][k]+a[i][j]>b[i][j])
       {
        b[i][j]=b[i-1][k]+a[i][j];        //更新当前最优解
        c[i][j]=k;                        //前一个花束的位置为 k
       }
  max=-2100000000;
  for (i=f;i<=v;i++)
   if (b[f][i]>max)
    {
      max=b[f][i];                        //选择全局最优解
      k=i;                                //k 最后一束花的位置
    }
```

```
    cout<<max<<endl;                    //打印最优解
    for (i=1;i<=f;i++)
     {
       d[i]=k;
       k=c[f-i+1][k];
     }
    for (i=f;i>=2;i--)
    cout<<d[i]<<" ";
    cout<<d[1]<<endl;
}
```

由此可看出,对于看似复杂的问题,通过转化就可变成简单的经典的动态规划问题。在问题原型的基础上,通过分析新问题与原问题的不同之处,修改状态转移方程,改变问题状态的描述和表示方式,就会降低问题规划和实现的难度,提高算法的效率。由此可见,动态规划问题中具体的规划方法将直接决定解决问题的难易程度和算法的时间与空间效率,而注意在具体的规划过程中的灵活性和技巧性将是动态规划方法提出的更高要求。

【解法二】

【算法分析】

flower 一题是 Ioi1999 第一天第一题,该题如用组合的方法处理,将会造成超时。正确的方法是用动态规划,考虑角度为一束一束地增加花束,假设用 b[i][j]表示 1～i 束花放在 1 到 j 之间的花瓶中的最大美学值,其中 i<=j ,则 b[i][j]=max(b[i-1][k-1]+a[i][k]),其中 i<=k<=j,a[i][k]的含义参见题目。输出结果时,显然使得 b[f][k]取得总的最大美观值的第一个 k 值就是第 F 束花应该摆放的花瓶位置,将总的最大美观值减去 a[i][k]的值即得到前 k-1 束花放在前 k-1 个瓶中的最大美观值,依次使用同样的方法就可求出每一束花应该摆放的花瓶号。由于这一过程是倒推出来的,所以程序中用递归程序来实现。

【参考程序】

```
#include<iostream>
#include<cstdio>
#include<cstring>
using namespace std;
void print(int,int);
int max(int a,int b) { return a>b? a:b; }
int a[101][101],b[101][101];
int main()
{
  int f,v;
  cin>>f>>v;
  for (int i=1;i<=f;i++)
    for (int j=1;j<=v;j++)
      cin>>a[i][j];
  memset(b,128,sizeof(b));              //初始化 b 数组
```

```
    for (int i=0;i<101;i++) b[0][i]=0; //没有放花时,美学值为0。这也是初始化
    for (int i=1;i<=f;i++)
      for (int j=i;j<=v-f+i;j++)
      {
        for (int k=i;k<=j;k++)
          b[i][j]=max(b[i][j],b[i-1][k-1]+a[i][k]);
      }
      int c=-1000000;
      for (int i=f;i<=v;i++)
        if (b[f][i]>c)
          c=b[f][i];
      cout<<c<<endl;
      print(f,c);
}
void print(int i,int j)
{
  int n;
  if (i>0)
    {
      n=i;
      while (b[i][n]!=j)
      {
        n++;
      }
      print(i-1,j-a[i][n]);
      cout<<n<<" ";
    }
}
```

记忆化搜索的应用

一般来说,动态规划总要遍历所有的状态,而搜索可以排除一些无效状态。更重要的是搜索还可以剪枝,可能剪去大量不必要的状态,因此在空间开销上往往比动态规划要低很多。

如何协调好动态规划的高效率与高消费之间的矛盾呢?有一种折中的办法就是记忆化算法。记忆化算法在求解的时候还是按着自顶向下的顺序,每求解一个状态,就将它的解保存下来,以后再次遇到这个状态的时候,就不必重新求解了。这种方法综合了搜索和动态规划两方面的优点,因而还是很有使用价值的。

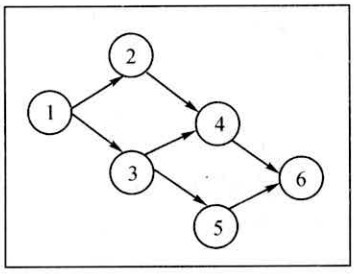


图 9-6

举一个例子:如图 9-6 所示是一个有向无环图,求从顶点 1 到顶点 6 的最长路径。(规定边的方向从左到右)

我们将从起点(顶点 1)开始到某个顶点的最长路径作为状态,用一维数组 opt 记录。opt[j]表示由起点到顶点 j 时的最长路径。显然,opt[1]=0,这是初始状态,即动态规划的边界条件。于是,我们很容易地写出状态转移方程式:opt[j]=max{opt[k]+a[k][j]}(k 到 j 有一条长度为 a[k][j]的边)。虽然有了完整的状态转移方程式,但是还是不知道动态规划的顺序。所以,还需要先进行一下拓扑排序,按照排序的顺序推下去,opt[6]就是问题的解。

可以看出,动态规划相比搜索之所以高效,是因为它将所有的状态都保存了下来。当遇到重复子问题时,它不像搜索那样把这个状态的最优值再计算一遍,只要把那个状态的最优值调出来就可以了。例如,当计算 opt[4]和 opt[5]时,都用到了 opt[3]的值。因为已经将它保存下来了,所以就没有必要再去搜索了。

但是动态规划仍然是有缺点的。一个很突出的缺点就是要进行拓扑排序。这道题的拓扑关系是很简单的,但有些题的拓扑关系是很复杂的。对于这些题目,如果也进行拓扑排序,工作量非常大。遇到这种情况,我们可以用记忆化搜索的方法,避免拓扑排序。

例 9.24 滑雪[2.6 基本算法之动态规划 90]

【问题描述】

小明喜欢滑雪,因为滑雪的确很刺激,可是为了获得速度,滑的区域必须向下倾斜,当小明滑到坡底,不得不再次走上坡或等着直升机来载他,小明想知道在一个区域中最长的滑坡。滑坡的长度由滑过点的个数来计算,区域由一个二维数组给出,数组的每个数字代表点的高度。下面是一个例子:

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

一个人可以从某个点滑向上下左右相邻四个点之一,当且仅当高度减小,在上面的例子中,一条可行的滑坡为 25-24-17-16-1(从 25 开始到 1 结束),当然 25-24…2-1 更长,事实上这是最长的一条。

【输入格式】

输入的第一行为表示区域的二维数组的行数 R 和列数 C(1<=R,C<=100),下面是 R 行,每行有 C 个数代表高度。

	[i-1,j]↓	
[i,j-1]→	[i,j]	←[i,j+1]
	[i+1,j]↑	

【输出格式】

输出区域中最长的滑坡长度。

【输入样例】

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

【输出样例】

```
25
```

【算法分析】

由于一个人可以从某个点滑向上下左右相邻四个点之一，如上表所示。当且仅当高度减小，对于任意一个点[i][j]，当它的高度小于与之相邻的四个点([i-1][j],[i][j+1],[i+1][j],[i][j-1])的高度时，这四个点可以滑向[i][j]，用 f[i][j]表示到[i][j]为止的最大长度，则 f[i][j]=max{f[i+a][j+b]}+1，其中坐标增量{(a,b)=[(1,0),(-1,0),(0,1),(0,-1)],0<i+a<=r,0<j+b<=c,High[i][j]<High[i+a][j+b]}。为了保证满足条件的 f[i+a][j+b]在 f[i][j]前算出，需要对高度排一次序，然后从大到小规划(高度)。最后再比较一下所有 f[i][j]{0<i<=r,0<j<=c}，找出其中最长的一条路线。我们还可以用记忆化搜索的方法，它的优点是不需进行排序，按照行的顺序，利用递归逐点求出区域中到达此点的最长路径，每个点的最长路径只求一次。

【参考程序】

```
#include<iostream>
#include<cstdio>
using namespace std;
int dx[5]={0,-1,0,1,0},   //x 的坐标增量
    dy[5]={0,0,1,0,-1};   //y 的坐标增量
long r,c,i,j,p,t,ans;
long m[101][101],f[101][101];
int search(int,int);
int main()
{
  cin>>r>>c;
  ans=0;
  for (i=1;i<=r;i++)
   for (j=1;j<=c;j++)
    cin>>m[i][j];          //读入每个点的高度
  for (i=1;i<=r;i++)
                  //按照行的顺序，利用递归逐点求出区域中到达此点的最长路径
   for (j=1;j<=c;j++)
    {
      t=search(i,j);
      f[i][j]=t;
      if (t>ans) ans=t;    //寻找最大长度值
    }
  cout<<ans<<endl;
}
int search(int x,int y)        //函数的作用是求到[x,y]点的最长路径
{
  int i,t,tmp,nx,ny;
   if (f[x][y]>0)              //此点长度已经求出，不必进行进一步递归，保证每
   {                           //一个点的最大长度只求一次，这是记忆化搜索的特点
```

```
        return (f[x][y]);
    }
    t=1;
    for (i=1;i<=4;i++)     //从四个方向上搜索能达到[x,y]的点
    {
        nx=x+dx[i];        //加上横、纵坐标
        ny=y+dy[i];
        if ((nx>=1)&&(nx<=r)&&(ny>=1)&&(ny<=c)&&(m[x][y]<m
          [nx][ny]))       //边界限制
        {                  //高度比较
            tmp=search(nx,ny)+1;        //递归进行记忆化搜索
            if (tmp>t) t=tmp;
        }
    }
    f[x][y]=t;
    return (t);
}
```

【上机练习】

1. 公共子序列【2.6 基本算法之动态规划 1808】

我们称序列 Z=<z1,z2,…,zk >是序列 X=<x1,x2,…,xm >的子序列当且仅当存在严格上升的序列<i_1,i_2,…,i_k>,使得对 j=1,2,…,k,有 $x_{i_j}=z_j$。比如 Z=<a,b,f,c>是 X=<a,b,c,f,b,c>的子序列。

现在给出两个序列 X 和 Y,你的任务是找到 X 和 Y 的最大公共子序列,也就是说要找到一个最长的序列 Z,使得 Z 既是 X 的子序列也是 Y 的子序列。

输入:

输入包括多组测试数据。每组数据包括一行,给出两个长度不超过 200 的字符串,表示两个序列。两个字符串之间由若干个空格隔开。

输出:

对每组输入数据,输出一行,给出两个序列的最大公共子序列的长度。

样例输入:

```
abcfbc          abfcab
programming     contest
abcd            mnp
```

样例输出:

```
4
2
0
```

2. 计算字符串距离【2.6 基本算法之动态规划 2988】

对于两个不同的字符串，我们有一套操作方法来把他们变得相同，具体方法为：

修改一个字符（如把“a”替换为“b”）

删除一个字符（如把“traveling”变为“travelng”）

比如对于“abcdefg”和“abcdef”两个字符串来说，我们认为可以通过增加/减少一个“g”的方式来达到目的。无论增加还是减少“g”，我们都仅仅需要一次操作。我们把这个操作所需要的次数定义为两个字符串的距离。

给定任意两个字符串，写出一个算法来计算出他们的距离。

输入：

第一行有一个整数 n。表示测试数据的组数。

接下来共 n 行，每行两个字符串，用空格隔开，表示要计算距离的两个字符串。

字符串长度不超过 1000。

输出：

针对每一组测试数据输出一个整数，值为两个字符串的距离。

样例输入：

3

abcdefg　abcdef

ab ab

mnklj jlknm

样例输出：

1

0

4

3. 糖果【2.6 基本算法之动态规划 2989】

由于在维护世界和平的事务中做出巨大贡献，Dzx 被赠予糖果公司 2010 年 5 月 23 日当天无限量糖果免费优惠券。在这一天，Dzx 可以从糖果公司的 N 件产品中任意选择若干件带回家享用。糖果公司的 N 件产品每件都包含数量不同的糖果。Dzx 希望他选择的产品包含的糖果总数是 K 的整数倍，这样他才能平均地将糖果分给帮助他维护世界和平的伙伴们。当然，在满足这一条件的基础上，糖果总数越多越好。Dzx 最多能带走多少糖果呢？

注意：Dzx 只能将糖果公司的产品整件带走。

输入：

第一行包含两个整数 N(1<=N<=100)和 K(1<=K<=100)。

以下 N 行每行 1 个整数，表示糖果公司该件产品中包含的糖果数目，不超过 1000000。

输出：

符合要求的最多能达到的糖果总数，如果不能达到 K 的倍数这一要求，输出 0。

样例输入：

5 7

1

2

3

4

5

样例输出:

14

提示:

Dzx 的选择是 2+3+4+5=14,这样糖果总数是 7 的倍数,并且是总数最多的选择。

4. 鸡蛋的硬度【2.6 基本算法之动态规划 7627】

最近 XX 公司举办了一个奇怪的比赛:鸡蛋硬度之王争霸赛。参赛者是来自世界各地的母鸡,比赛的内容是看谁下的蛋最硬,更奇怪的是 XX 公司并不使用什么精密仪器来测量蛋的硬度,他们采用了一种最老土的办法——从高度扔鸡蛋——来测试鸡蛋的硬度,如果一次母鸡下的蛋从高楼的第 a 层摔下来没摔破,但是从 a+1 层摔下来时摔破了,那么就说这只母鸡的鸡蛋的硬度是 a。你当然可以找出各种理由说明这种方法不科学,比如同一只母鸡下的蛋硬度可能不一样等等,但是这不影响 XX 公司的争霸赛,因为他们只是为了吸引大家的眼球,一个个鸡蛋从 100 层的高楼上掉下来的时候,这情景还是能吸引很多人驻足观看的,当然,XX 公司也绝不会忘记在高楼上挂一条幅,写上"XX 公司"的字样——这比赛不过是 XX 公司的一个另类广告而已。

勤于思考的小 A 总是能从一件事情中发现一个数学问题,这件事也不例外。"假如有很多同样硬度的鸡蛋,那么我可以用二分的办法用最少的次数测出鸡蛋的硬度",小 A 对自己的这个结论感到很满意,不过很快麻烦来了,"但是,假如我的鸡蛋不够用呢,比如我只有 1 个鸡蛋,那么我就不得不从第 1 层楼开始一层一层的扔,最坏情况下我要扔 100 次。如果有 2 个鸡蛋,那么就从 2 层楼开始的地方扔……等等,不对,好像应该从 1/3 的地方开始扔才对,嗯,好像也不一定啊……3 个鸡蛋怎么办,4 个,5 个,更多呢……",和往常一样,小 A 又陷入了一个思维僵局,与其说他是勤于思考,不如说他是喜欢自找麻烦。

好吧,既然麻烦来了,就得有人去解决,小 A 的麻烦就靠你来解决了:)

输入:

输入包括多组数据,每组数据一行,包含两个正整数 n 和 m(1<=n<=100,1<=m<=10),其中 n 表示楼的高度,m 表示你现在拥有的鸡蛋个数,这些鸡蛋硬度相同(即它们从同样高的地方掉下来要么都摔碎要么都不碎),并且小于等于 n。你可以假定硬度为 x 的鸡蛋从高度小于等于 x 的地方摔无论如何都不会碎(没摔碎的鸡蛋可以继续使用),而只要从比 x 高的地方扔必然会碎。

对每组输入数据,你可以假定鸡蛋的硬度在 0 至 n 之间,即在 n+1 层扔鸡蛋一定会碎。

输出:

对于每一组输入,输出一个整数,表示使用最优策略在最坏情况下所需要的扔鸡蛋次数。

样例输入:

100 1

100 2

样例输出:

100

14

提示：

最优策略指在最坏情况下所需要的扔鸡蛋次数最少的策略。

如果只有一个鸡蛋，你只能从第一层开始扔，在最坏的情况下，鸡蛋的硬度是 100，所以需要扔 100 次。如果采用其他策略，你可能无法测出鸡蛋的硬度（比如你第一次在第二层的地方扔，结果碎了，这时你不能确定硬度是 0 还是 1），即在最坏情况下你需要扔无限次，所以第一组数据的答案是 100。

5. 大盗阿福【2.6 基本算法之动态规划 8462】

阿福是一名经验丰富的大盗。趁着月黑风高，阿福打算今晚洗劫一条街上的店铺。

这条街上一共有 N 家店铺，每家店中都有一些现金。阿福事先调查得知，只有当他同时洗劫了两家相邻的店铺时，街上的报警系统才会启动，然后警察就会蜂拥而至。

作为一向谨慎作案的大盗，阿福不愿意冒着被警察追捕的风险行窃。他想知道，在不惊动警察的情况下，他今晚最多可以得到多少现金？

输入：

输入的第一行是一个整数 T（T<=50），表示一共有 T 组数据。

接下来的每组数据，第一行是一个整数 N（1<=N<=100,000），表示一共有 N 家店铺。第二行是 N 个被空格分开的正整数，表示每一家店铺中的现金数量。每家店铺中的现金数量均不超过 1000。

输出：

对于每组数据，输出一行。该行包含一个整数，表示阿福在不惊动警察的情况下可以得到的现金数量。

样例输入：

```
2
3
1 8 2
4
10 7 6 14
```

样例输出：

```
8
24
```

提示：

对于第一组样例，阿福选择第 2 家店铺行窃，获得的现金数量为 8。

对于第二组样例，阿福选择第 1 和 4 家店铺行窃，获得的现金数量为 10+14=24。

6. 股票买卖【2.6 基本算法之动态规划 8464】

最近越来越多的人都投身股市，阿福也有点心动了。谨记着“股市有风险，入市需谨慎”，阿福决定先来研究一下简化版的股票买卖问题。

假设阿福已经准确预测出了某只股票在未来 N 天的价格，他希望买卖两次，使得获得的利润最高。为了计算简单起见，利润的计算方式为卖出的价格减去买入的价格。

同一天可以进行多次买卖。但是在第一次买入之后，必须要先卖出，然后才可以第二次买入。现在，阿福想知道他最多可以获得多少利润。

输入：

输入的第一行是一个整数 T(T<=50),表示一共有 T 组数据。

接下来的每组数据,第一行是一个整数 N(1<=N<=100,000),表示一共有 N 天。第二行是 N 个被空格分开的整数,表示每天该股票的价格。该股票每天的价格的绝对值均不会超过 1,000,000 。

输出:

对于每组数据,输出一行。该行包含一个整数,表示阿福能够获得的最大的利润。

样例输入:

```
3
7
5 14 -2 4 9 3 17
6
6 8 7 4 1 -2
4
18 9 5 2
```

样例输出:

```
28
2
0
```

提示:

对于第一组样例,阿福可以第 1 次在第 1 天买入(价格为 5),然后在第 2 天卖出(价格为 14)。第 2 次在第 3 天买入(价格为-2),然后在第 7 天卖出(价格为 17)。一共获得的利润是(14-5)+(17-(-2))=28

对于第二组样例,阿福可以第 1 次在第 1 天买入(价格为 6),然后在第 2 天卖出(价格为 8)。第 2 次仍然在第 2 天买入,然后在第 2 天卖出。一共获得的利润是 8-6=2

对于第三组样例,由于价格一直在下跌,阿福可以随便选择一天买入之后迅速卖出。获得的最大利润为 0

7. 鸣人的影分身【2.6 基本算法之动态规划 8467】

在火影忍者的世界里,令敌人捉摸不透是非常关键的。我们的主角漩涡鸣人所拥有的一个招数——多重影分身之术——就是一个很好的例子。

影分身是由鸣人身体的查克拉能量制造的,使用的查克拉越多,制造出的影分身越强。

针对不同的作战情况,鸣人可以选择制造出各种强度的影分身,有的用来佯攻,有的用来发起致命一击。

那么问题来了,假设鸣人的查克拉能量为 M,他影分身的个数最多为 N,那么制造影分身时有多少种(用 K 表示)不同的分配方法?(影分身可以被分配到 0 点查克拉能量)

输入:

第一行是测试数据的数目 t(0<=t<=20)。以下每行均包含二个整数 M 和 N,以空格分开。1<=M,N<=10。

输出:

对输入的每组数据 M 和 N,用一行输出相应的 K。

样例输入:

1

7 3

样例输出：

8

8. 数的划分【2.6 基本算法之动态规划 8787】

将整数 n 分成 k 份，且每份不能为空，任意两份不能相同(不考虑顺序)。

例如：n=7，k=3，下面三种分法被认为是相同的。

1,1,5; 1,5,1; 5,1,1;

问有多少种不同的分法。输出一个整数，即不同的分法。

输入：

两个整数 n,k(6<n<=200,2<=k<=6)，中间用单个空格隔开。

输出：

一个整数，即不同的分法。

样例输入：

7 3

样例输出：

4

提示：

四种分法为：1,1,5;1,2,4;1,3,3;2,2,3。

9. Maximum sum【2.6 基本算法之动态规划 1481】

题意：对于给定的整数序列 A={a1, a2,…, an}，找出两个不重合连续子段，使得两子段中所有数字的和最大。我们如下定义函数 d(A)：

$$d(A)=\max_{1\le s_1\le t_1\le s_2\le t_2\le n}\left\{\sum_{i=s_1}^{t_1}a_i+\sum_{j=s_2}^{t_2}a_j\right\}$$

我们的目标就是求出 d(A)

输入：

第一行是一个整数 T(<=30)，代表一共有多少组数据。

接下来是 T 组数据。

每组数据的第一行是一个整数，代表数据个数据 n(2<=n<=50000)，第二行是 n 个整数 a1, a2,…, an(|ai| <=10000)。

输出：

输出一个整数，就是 d(A)的值。

样例输入：

1

10

1　−1　2　2　3　−3　4　−4　5　−5

样例输出：

13

提示：就是求最大子段和问题，样列取{2,2,3,−3,4}和{5}。

10. 最长公共子上升序列【2.6 基本算法之动态规划 2000】

给定两个整数序列,写一个程序求它们的最长上升公共子序列。

当以下条件满足的时候,我们将长度 N 的序列 S1,S2,…,SN 称为长度为 M 的序列 A1,A2,…,AM 的上升子序列:

存在 1<=i1<i2<…<iN<=M,使得对所有 1<=j<=N,均有 $Sj=Ai_j$,且对于所有的 1<=j <N,均有 $S_j<S_{j+1}$。

输入:

每个序列用两行表示,第一行是长度 M(1<=M<=500),第二行是该序列的 M 个整数 Ai(-2^31<=Ai<2^31)

输出:

在第一行,输出两个序列的最长上升公共子序列的长度 L。在第二行,输出该子序列。如果有不止一个符合条件的子序列,则输出任何一个即可。

样例输入:

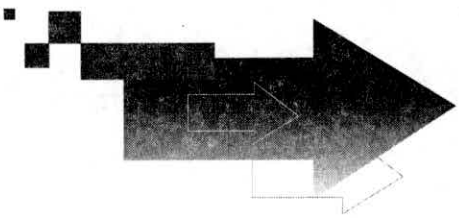
```
5
1  4  2  5  -12
4
-12  1  2  4
```

样例输出:

```
2
1 4
```

第三部分

数据结构



第一章　栈

栈是只能在某一端插入和删除的特殊线性表。

用桶堆积物品，先堆进来的压在底下，随后一件一件往上堆。取走时，只能从上面一件一件取。堆和取都在顶部进行，底部一般是不动的。

栈就是一种类似桶堆积物品的数据结构，进行删除和插入的一端称栈顶，另一堆称栈底。插入一般称为进栈(PUSH)，删除则称为退栈(POP)。栈也称为后进先出表(LIFO 表)。

一个栈可以用定长为 n 的数组 s 来表示，用一个栈指针 top 指向栈顶。若 top=0，表示栈空，top=n 时栈满。进栈时 top 加 1，退栈时 top 减 1。当 top<0 时为下溢。栈指针在运算中永远指向栈顶。

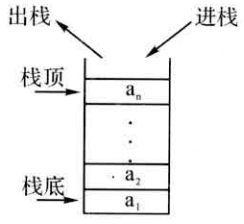


图 1－1

1. 进栈(PUSH)算法

①若 top>=n 时，则给出溢出信息，作出错处理(进栈前首先检查栈是否已满，满则溢出；不满则作②)；

②top++(栈指针加 1，指向进栈地址)；

③s[top]=x，结束(x 为新进栈的元素)。

2. 退栈(POP)算法

①若 top<=0，则给出下溢信息，作出错处理(退栈前先检查是否已为空栈，空则下溢；不空则作②)；

②x=s[top]，(退栈后的元素赋给 x)；

③top--，结束(栈指针减 1，指向栈顶)。

进栈、出栈的 C++实现过程程序：

```
#define n 100
void push(int s[],int *top,int *x)          //入栈
{
    if (*top==n) printf("overflow");
      else { (*top)++; s[*top]=*x; }
}
void pop(int s[],int *y,int *top)           //出栈
```

```
{
    if (*top==0)   printf("underflow");
      else { *y=s[*top]; (*top)--; }
}
```

对于出栈运算中的“下溢”,程序中仅给出了一个标志信息,而在实际应用中,下溢可用来作为控制程序转移的判断标志,是十分有用的。对于入栈运算中的“上溢”,则是一种致命的错误,将使程序无法继续运行,所以要设法避免。

堆栈的数组模拟

十进制数 n 和其他 d 进制数的转换是实现计算的基本问题,解决方法很多,下面给出一种算法原理:n=(n/d)*d+n%d(其中 / 为整除运算,%为求余运算)。

例如:$(1348)_{10}=(2504)_8$ 运算过程如下:

n	n / 8	n % 8
9413		

n	n / 8	n % 8
1348	168	4
168	21	0
21	2	5
2	0	2

1. 填空:

$(9413)_{10}=(\quad)_8=(\quad)_{16}=(\quad)_2$

2. 数制转化程序

```
#include<cstdlib>
#include<iostream>
using namespace std;
#define size 100
int a[size+1],n,d,i=0,j;
main()
{
  cout<<" Please enter a number(N) base 10:"; cin>>n;
  cout<<" please enter a number(d):"; cin>>d;
  do{
      a[++i]=n%d;
      n=n/d;
  }while(n!=0);
  for (j=i;j>=1;j--)cout<<a[j];
  return 0;
}
```

3. 火车站列车调度示意图如图 1-2 所示,假设调度站两侧的轨道为单向行驶轨道。

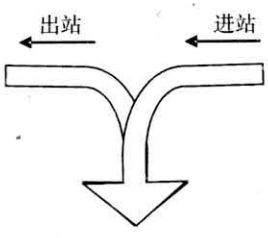


图 1-2

(1)如果进站的车厢序列为 123,则可能的出站车厢序列是什么?

(2)如果进站的车厢序列为123456,问能否得到135426和435612的出站序列?

例1.1　括号的匹配(表达式的合法性检查)

【问题描述】

假设一个表达式由英文字母(小写)、运算符(+、-、*、/)和左右小(圆)括号构成,以"@"作为表达式的结束符。请编写一个程序检查表达式中的左右圆括号是否匹配,若匹配,则返回"YES";否则返回"NO"。假设表达式长度小于255,左圆括号少于20个。

【算法分析】

假设输入的字符串存储在c中(char c[256])。

我们可以定义一个栈:char s[maxn+1];

int top;

用它来存放表达式中从左往右的左圆括号(maxn=20)。

算法的思路为:顺序(从左往右)扫描表达式的每个字符c[i],若是"(",则让它进栈;若遇到的是")",则让栈顶元素出栈;当栈发生下溢或当表达式处理完毕而栈非空时,都表示不匹配,返回"NO";否则表示匹配,返回"YES"。

【参考程序】

```
#include<cstdio>
#include<cstdlib>
#define maxn 20
using namespace std;
char c[256];
bool judge(char c[256])
{
    int top=0,i=0;
    while (c[i]!='@')
    {
        if (c[i]=='(') top++;
        if (c[i]==')')
        {
            if (top>0) top--;
              else return 0;
        }
        i++;
    }
    if (top!=0) return 0;  //检测栈是否为空。不空则说明有未匹配的括号
    else return 1;
}
main()
{
  scanf("%s",c);
  if (judge(c))printf("YES");
```

```
    else printf("NO");
    return 0;
}
```

例 1.2 编程求一个后缀表达式的值

【问题描述】

从键盘读入一个后缀表达式(字符串),只含有 0—9 组成的运算数及加(+)、减(—)、乘(*)、除(/)四种运算符。每个运算数之间用一个空格隔开,不需要判断给你的表达式是否合法。以@作为结束标志。

【算法分析】

后缀表达式的处理过程很简单,过程如下:扫描后缀表达式,凡遇操作数则将之压进堆栈,遇运算符则从堆栈中弹出两个操作数进行该运算,将运算结果压栈,然后继续扫描,直到后缀表达式被扫描完毕为止,此时栈底元素即为该后缀表达式的值。

比如,16—9*(4+3)转换成后缀表达式为:16□9□4□3□+*—,在字符数组 A 中的形式为:

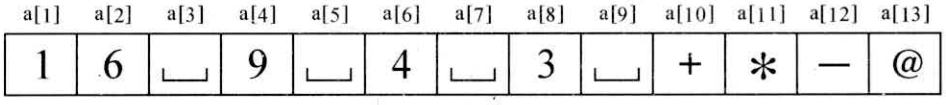


图 1—3

栈中的变化情况:

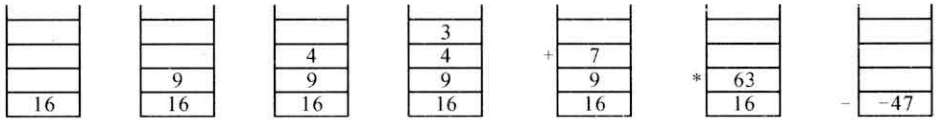


图 1—4

运行结果:—47

【参考程序】

```
#include<cstdio>
#include<cstdlib>
#include<string>
#include<cstring>
using namespace std;
int stack[101];
char s[256];
int comp(char s[256])
{
    int  i=0,top=0,x,y;
    while (i<=strlen(s)-2)
    {
      switch (s[i])
      {
        case '+':stack[--top]+=stack[top+1]; break;
```

```
            case '-':stack[--top]-=stack[top+1]; break;
            case '*':stack[--top]*=stack[top+1]; break;
            case '/':stack[--top]/=stack[top+1]; break;
            default:x=0; while (s[i]!='') x=x*10+s[i++]-'0';
              stack[++top]=x; break;
          }
          i++;
        }                                          //while
      return stack[top];
}
main()
{
  printf(" input a string(@_over):");
  gets(s);
  printf(" result=%d ",comp(s));
  return 0;
}
```

例 1.3　车厢调度

【问题描述】

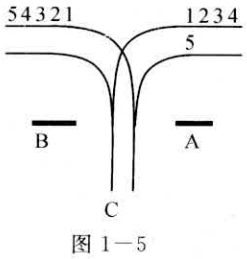


图 1-5

有一个火车站，铁路如图所示，每辆火车从 A 驶入，再从 B 方向驶出，同时它的车厢可以重新组合。假设从 A 方向驶来的火车有 n 节（n<=1000），分别按照顺序编号为 1，2，3，…，n。假定在进入车站前，每节车厢之间都不是连着的，并且它们可以自行移动到 B 处的铁轨上。另外假定车站 C 可以停放任意多节车厢。但是一旦进入车站 C，它就不能再回到 A 方向的铁轨上了，并且一旦当它进入 B 方向的铁轨，它就不能再回到车站 C。

负责车厢调度的工作人员需要知道能否使它以 a1，a2，…，an 的顺序从 B 方向驶出，请你来判断能否得到指定的车厢顺序。

【输入】

输入文件的第一行为一个整数 n，其中 n<=1000，表示有 n 节车厢，第二行为 n 个数字，表示指定的车厢顺序。

【输出】

如果可以得到指定的车厢顺序，则输出一个字符串"YES"，否则输出"NO"（注意要大写，不包含引号）。

【输入样例】

5

5 4 3 2 1

【输出样例】

YES

分析：

该题就是前面思考题的一部分。车站C相当于一个栈。我们用模拟法来做,假设我们已经处理了前i-1节从B方向驶出的车厢,我们现在要让ai驶出。若ai不在车站C中,我们就让若干车厢从A方向驶入车站C,直到ai驶入,再将它从B方向驶出;若ai在车站C中,如果它是车站C中停在最前面的,则将它从B方向驶出,否则原问题无解。

如样例中,出栈序列是3 5 4 2 1,模拟过程如下:

①一开始栈为空;

②由于3不在栈中,就需要把1,2,3依次进栈,再出栈,这样符合出栈序列第一个数是3,当前栈为{1,2};

③第2个出栈的是5,5不在栈中,就要把4,5压栈,再出栈就可以得到5,此时栈为{1,2,4};

④第3个出栈的是4,正好是栈顶元素,直接出栈,栈变为{1,2};

⑤第4个出栈的是2,正好是栈顶元素,直接出栈,栈变为{2};

⑥第5个出栈的是1,正好是栈顶元素,直接出栈,栈变为{}。

在模拟过程中没有碰到要出栈的数在栈中但不是栈顶元素的情况,所以该方案可行。

【参考程序】

```
#include <iostream>
#include <algorithm>
#include <cstring>
using namespace std;
const int N = 1010;
int stack[N],a[N];
int top,n;

int main()
{
    cin >> n;
    for (int i = 1;i <= n;++ i)
        cin >> a[i];
    top = 0;
    for (int i = 1,cur = 1;i <= n;++ i) //cur为当前要从A方向驶入的车厢号
    {
        while (cur <= a[i])
            stack[++ top] = cur ++;
        if (stack[top] == a[i])
            -- top;
        else
        {
            cout << "NO" << endl;
                return 0;
        }
```

```
    }
    cout << " YES " << endl;
    return 0;
}
```

【上机练习】

1. 表达式括号匹配(stack)

【问题描述】

假设一个表达式由英文字母(小写)、运算符(+、-、*、/)和左右小(圆)括号构成,以"@"作为表达式的结束符。请编写一个程序检查表达式中的左右圆括号是否匹配,若匹配,则返回"YES";否则返回"NO"。表达式长度小于255,左圆括号少于20个。

【输入格式】

输入文件stack.in包括一行数据,即表达式。

【输出格式】

输出文件stack.out包括一行,即"YES"或"NO"。

【输入样例1】　　【输出样例1】

2*(x+y)/(1-x)@　　YES

【输入样例2】　　【输出样例2】

(25+x)*(a*(a+b+b)@　　NO

2. 括弧匹配检验(check)

【问题描述】

假设表达式中允许包含两种括号:圆括号和方括号,其嵌套的顺序随意,如([ ]())或[([ ][ ])]等为正确的匹配,[( ])或([ ]()或(()))均为错误的匹配。

现在的问题是,要求检验一个给定表达式中的括弧是否正确匹配?

输入一个只包含圆括号和方括号的字符串,判断字符串中的括号是否匹配,匹配就输出"OK",不匹配就输出"Wrong"。输入一个字符串:[([][])],输出:OK。

【输入格式】

输入仅一行字符(字符个数小于255)。

【输出格式】

匹配就输出"OK",不匹配就输出"Wrong"。

【输入样例】　　【输出样例】

[(])　　Wrong

3. 字符串匹配问题(strs)

【问题描述】

字符串中只含有括号:()、[]、<>、{},判断输入的字符串中括号是否匹配。如果括号有互相包含的形式,从内到外必须是<>、()、[]、{},例如:输入:[()],输出:YES,而输入([])、([)]都应该输出NO。

【输入格式】

文件的第1行为一个整数n,表示以下有多少个由括号好组成的字符串。接下来的n

行,每行都是一个由括号组成的长度不超过255的字符串。

【输出格式】

在输出文件中有n行,每行都是YES或NO。

【输入样例】	【输出样例】
5	YES
{}{}<><>()()[][]	YES
{{}}{{}}<<>><<>>(())(())[[]][[]]	YES
{{}}{{}}<<>><<>>(())(())[[]][[]]	YES
{<>}{[]}<<<>><<>>>((<>))(())[[(<>)]][[]]	NO
><}{{[]}<<<>><<>>>((<>))(())[[(<>)]][[]]	

4. 计算(calc)

【问题描述】

小明在你的帮助下,破译了Ferrari设的密码门,正要往前走,突然又出现了一个密码门,门上有一个算式,其中只有"("、")"、"0－9"、"＋"、"－"、"＊"、"/"、"^"求出的值就是密码。小明数学学得不好,还需你帮他的忙。("/"用整数除法)

【输入格式】

输入文件calc.in共1行,为一个算式。

【输出格式】

输出文件calc.out共1行,就是密码。

【输入样例】	【输出样例】
1＋(3＋2)＊(7^2＋6＊9)/(2)	258

【数据范围】

100%的数据满足:算式长度<＝30,其中所有数据在$2^{31}-1$的范围内。

5. 车厢调度(train)

【问题描述】

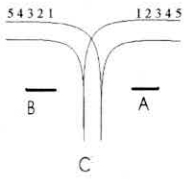


图1－6

有一个火车站,铁路如图1－5所示,每辆火车从A驶入,再从B方向驶出,同时它的车厢可以重新组合。假设从A方向驶来的火车有n节(n<＝1000),分别按照顺序编号为1,2,3,…,n。假定在进入车站前,每节车厢之间都不是连着的,并且它们可以自行移动到B处的铁轨上。另外假定车站C可以停放任意多节车厢。但是一旦进入车站C,它就不能再回到A方向的铁轨上了,并且一旦当它进入B方向的铁轨,它就不能再回到车站C。

负责车厢调度的工作人员需要知道能否使它以a1,a2,…,an的顺序从B方向驶出,请来判断能否得到指定的车厢顺序。

【输入格式】

输入文件的第一行为一个整数n,其中n<＝1000,表示有n节车厢,第二行为n个数字,表示指定的车厢顺序。

【输出格式】

如果可以得到指定的车厢顺序,则输出一个字符串"YES",否则输出"NO"(注意要大写,不包含引号)。

【输入样例】　　【输出样例】

5　　YES

5 4 3 2 1

6. 中缀表达式值(expr)

【问题描述】

输入一个中缀表达式(由0～9组成的运算数、加＋减—乘＊除/四种运算符、左右小括号组成。注意“—”也可作为负数的标志，表达式以“@”作为结束符)，判断表达式是否合法，如果不合法，请输出“NO”；否则请把表达式转换成后缀形式，再求出后缀表达式的值并输出。

注意：必须用栈操作，不能直接输出表达式的值。

【输入格式】

输入文件的第一行为一个以@结束的字符串。

【输出格式】

如果表达式不合法，请输出“NO”，要求大写。

如果表达式合法，请输出计算结果。

【输入样例】　　【输出样例】

1＋2＊8－9　　8

第二章 队 列

队列是限定在一端进行插入,另一端进行删除的特殊线性表。就像排队买东西,排在前面的人买完东西后离开队伍(删除),而后来的人总是排在队伍末尾(插入)。通常把队列的删除和插入分别称为出队和入队。允许出队的一端称为队头,允许入队的一端称为队尾。所有需要进队的数据项,只能从队尾进入,队列中的数据项只能从队头离去。由于总是先入队的元素先出队(先排队的人先买完东西),这种表也称为先进先出(FIFO)表。

队列可以用数组 Q[m+1]来存储,数组的上界 m 即是队列所容许的最大容量。在队列的运算中需设两个指针:

head:队头指针,指向实际队头元素的前一个位置。

tail:队尾指针,指向实际队尾元素所在的位置。

一般情况下,两个指针的初值设为 0,这时队列为空,没有元素。图 2-1 (a)画出了一个由 6 个元素构成的队列,数组定义 Q[11]。

Q(i) i=3,4,5,6,7,8,头指针 head=2,尾指针 tail=8。

队列中拥有的元素个数为:L=tail-head。现要让排头的元素出队,则需将头指针加 1,即 head++。这时头指针向上移动一个位置,指向 Q(3),表示 Q(3)已出队。见图 2-1 (b)。

如果想让一个新元素入队,则需尾指针向上移动一个位置,即 tail++,这时 Q(9)入队,见图 2-1 (c)。

当队尾已经处理在最上面时,即 tail=10,见图 2-1 (d),如果还要执行入队操作,则要发生"上溢",但实际上队列中还有三个空位置,所以这种溢出称为"假溢出"。

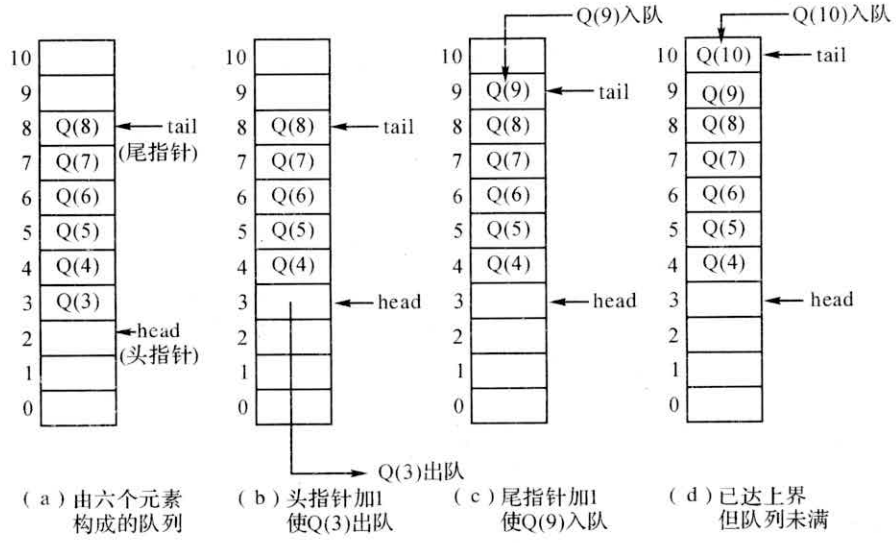


图 2-1

克服假溢出的方法有两种。一种是将队列中的所有元素向低地址区移动，显然这种方法是很浪费时间的；另一种方法是将数组存储区看成是一个首尾相接的环形区域。当存放到 n 地址后，下一个地址就“翻转”为 1。在结构上采用这种技巧来存储的队列称为循环队列，见图 2－2

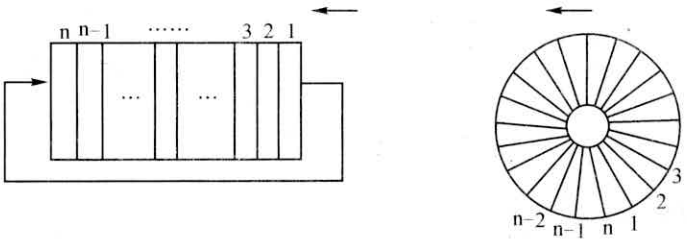


图 2－2　循环队列

循环队的入队算法如下：

1. tail＋＋；

2. 若 tail＝n＋1，则 tail＝1；

3. 若 head＝tail，尾指针与头指针重合了，表示元素已装满队列，则作上溢出错处理；

4. 否则，Q(tail)＝x，结束(x 为新入队元素)。

队列和栈一样，有着非常广泛的应用。

考虑一个分时系统，如果一台计算机联有四个终端，即允许四个用户同时使用这一台计算机。那么，计算机系统必须设立一个队列，用以管理各终端用户使用 CPU 的请求。当某个用户要求使用 CPU 时，相应的终端代号就入队(插入队尾)，而队头的终端用户则是 CPU 当前服务的对象。我们考虑最简单的情况，对于当前用户(队头)，系统每次分配一个为时间片的时间间隔，在一个时间片内，如果当前用户的作业没有结束，则该终端用户的代号出队后重新入队，插入队尾，等待下一次 CPU 服务。如果某个用户的作业运行结束，则先退出，出队后不再入队，整个运行过程就是各终端代号不断地入队、出队，CPU 轮流地为 n(n≤4)个终端用户服务。由于计算机的运行速度极快，所以，对于每个终端用户来说，似乎计算机是单独在为其服务。

和线性表一样，栈和队可以采用链表存储结构，当要实现多个栈共享内存或多个队共享内存时，选择链式分配结构则更为合适。

例 2.1　假设在周末舞会上，男士们和女士们进入舞厅时，各自排成一队。跳舞开始时，依次从男队和女队的队头上各出一人配成舞伴。规定每个舞曲只有一对跳舞者。若两队初始人数不相同，则较长的那一队中未配对者等待下一轮舞曲。现要求写一个程序，模拟上述舞伴配对问题。

【输入】

第 1 行两队的人数

第 2 行舞曲的数目

【算法分析】　设计两个队列分别存放男士和女士。每对跳舞的人一旦跳完后就回到队尾等待下次被选。如 m＝4、n＝3、k＝6。

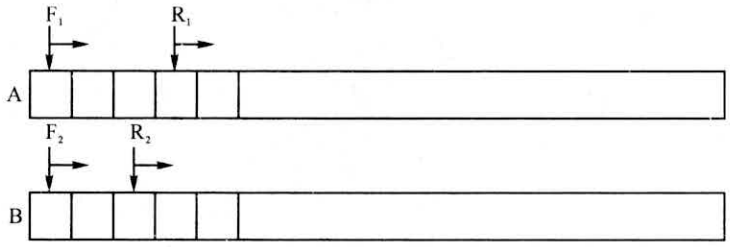


图 2-3

【参考程序】

```
#include<cstdio>
#include<iostream>
using namespace std;
int  a[10001],b[10001],k1=1,k,i,f1=1,r1,f2=1,r2;
main()
{
  int m,n;
  cin>>m>>n;
  for (i=1;i<=m;i++) a[i]=i;
  for (i=1;i<=n;i++) b[i]=i;
  cin>>k;
  r1=m; r2=n;
  while (k1<=k)
  {
      printf("%d  %d\n",a[f1],b[f2]);
      r1++; a[r1]=a[f1]; f1++;
            //第一次 a[m+1]=a[1]=1,第二次 a[m+2]=a[2]=2,如此循环
      r2++; b[r2]=b[f2]; f2++;
            //第一次 b[n+1]=b[1]=1,第二次 b[n+2]=b[2]=2,如此循环
      k1++;
  }
  return 0;
}
```

例 2.2　Blah 数集【3.4 数据结构之队列 2729】

大数学家高斯小时候偶然间发现一种有趣的自然数集合 Blah,对于以 a 为基的集合 Ba 定义如下:

(1)a 是集合 Ba 的基,且 a 是 Ba 的第一个元素;

(2)如果 x 在集合 Ba 中,则 2x+1 和 3x+1 也都在集合 Ba 中;

(3)没有其他元素在集合 Ba 中了。

现在小高斯想知道如果将集合 Ba 中元素按照升序排列,第 N 个元素会是多少?

【输入】

输入包括很多行,每行输入包括两个数字,集合的基 a(1<=a<=50))以及所求元素序

号 n(1<=n<=1000000)。

【输出】

对于每个输入,输出集合 Ba 的第 n 个元素值。

【样例输入】

1 100

28 5437

【样例输出】

418

900585

【分析】

题目要求输出集合中第 n 小的数,我们可以按照从小到大的顺序把序列中的前 n 个数计算出来,注意数集中除了第一个数 a 以外,其余每一个数 y 一定可以表示成 2x+1 或者 3x+1 的形式,其中 x 是数集中某一个数。因此除了第一数 a 以外,可以把数集 q[]的所有数分成两个子集,一个是用 2x+1 来表示的数的集合 1,另一个是用 3x+1 来表示的数的集合 2,两个集合要保持有序非常容易,只需用两个指针 two 和 three 来记录,其中 two 表示集合 1 下一个要产生的数是由 q[two]*2+1 得到,three 表示集合 2 下一个要产生的数是由 q[three]*3+1 得到。接下来比较 q[two]*2+1 和 q[three]*3+1 的大小关系:

(1) q[two]*2+1<q[three]*3+1:如果 q[two]*2+1 与 q[rear-1]不等,则把 q[two]*2+1 加到数集中,即:q[rear++]= q[two]*2+1;two++;

如果 q[two]*2+1 与 q[rear-1]相等,因为集合的唯一性,q[two]*2+1 不能加入数集,但 two 同样要加 1。

(2) q[two]*2+1>=q[three]*3+1:处理方法同上。

如此循环直到产生出数集的第 n 个数。

程序如下:

```
#include<iostream>
#include<algorithm>
using namespace std;
const int N=1000100;
long long q[N];
int a,n;
void work(int a,int n)
{
    int rear=2;
    q[1]=a;
    int two=1,three=1;
    while(rear<=n)
    {
        long long t1=q[two]*2+1,t2=q[three]*3+1;
        int t=min(t1,t2);
        if (t1<t2) two++; else three++;
```

```
            if (t==q[rear-1]) continue;
            q[rear++]=t;
        }
        cout<<q[n]<<endl;
}

int main()
{
    while (cin>>a>>n) work(a,n);
    return 0;
}
```

例 2.3 设有 n 个人依次围成一圈,从第 1 个人开始报数,数到第 m 个人出列,然后从出列的下一个人开始报数,数到第 m 个人又出列,…,如此反复到所有的人全部出列为止。设 n 个人的编号分别为 1,2,…,n,打印出列的顺序。

【算法分析】

本题我们可以用数组建立标志位等方法求解,但如果用上数据结构中循环链的思想,则更贴切题意,解题效率更高。n 人围成一圈,把一人看成一个结点,n 人之间的关系采用链接方式,即每一结点有一个前继结点和一个后继结点,每一个结点有一个指针指向下一个结点,最后一个结点指针指向第一个结点。这就是单循环链的数据结构。当 m 人出列时,将 m 结点的前继结点指针指向 m 结点的后继结点指针,即把 m 结点驱出循环链。

1. 建立循环链表。

当用数组实现本题链式结构时,数组 a[i]作为"指针"变量来使用,a[i]存放下一个结点的位置。设立指针 j 指向当前结点,则移动结点过程为 j=a[j],当数到 m 时,m 结点出链,则 a[j]=a[a[j]]。当直接用链来实现时,则比较直观,每个结点有两个域:一个数值域,一个指针域;当数到 m 时,m 出链,将 m 结点的前继结点指针指向其后继结点;

2. 设立指针,指向当前结点,设立计数器,计数数到多少人;

3. 沿链移动指针,每移动一个结点,计数器值加 1,当计数器值为 m 时,则 m 结点出链,计数器值置为 1。

4. 重复 3,直到 n 个结点均出链为止。

【参考程序】用数组实现链式结构

```
#include<cstdio>
using namespace std;
const int n=10,m=4;                    //设有 10 个人,报到 4 的人出列
int a[n+1],j=n,k=1,p=0;
main()
{
  for (int i=1;i<n;i++) a[i]=i+1;      //建立链表
  a[n]=1;                              //第 n 人指向第 1 人,形成一个环
  while (p<n)                          //n 个人均出队为止
  {
```

```
        while(k<m)                          //报数,计数器加 1
        {
          j=a[j];
          k++;
        }
        printf("%d ",a[j]); p++;            //数到 m,此人出队,计数器置 1
        a[j]=a[a[j]]; k=1;
    }
    return 0;
}
```

例 2.4　连通块。

【问题描述】

一个 n * m 的方格图,一些格子被涂成了黑色,在方格图中被标为 1,白色格子标为 0。问有多少个四连通的黑色格子连通块。四连通的黑色格子连通块指的是一片由黑色格子组成的区域,其中的每个黑色格子能通过四连通的走法(上下左右),只走黑色格子,到达该联通块中的其它黑色格子。

【输入】

第一行两个整数 n,m(1<=n,m<=100),表示一个 n * m 的方格图。

接下来 n 行,每行 m 个整数,分别为 0 或 1,表示这个格子是黑色还是白色。

【输出】

只有一行,一个整数 ans,表示图中有 ans 个黑色格子连通块。

【样例输入】

```
3 3
1 1 1
0 1 0
1 0 1
```

【样例输出】

```
3
```

【分析】

我们可以枚举每个格子,若它是被涂黑的,且它不属于已经搜索过的联通块,则由它开始,扩展搜索它所在的联通块,并把联通块里的所有黑色格子标记为已搜索过。

如何扩展搜索一个联通块呢? 我们用一个搜索队列,存储要搜索的格子。每次取出队首的格子,对其进行四连通扩展,若有黑格子,将其加入队尾,扩展完就把该格子删除。当队列为空时,一个联通块就搜索完了。

如样例所对应的方格图如下所示:

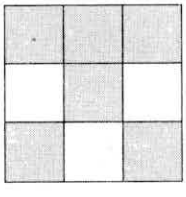


图 2-4

现在我们以样例为例模拟出这个方格图的搜索顺序：

①将(1,1)加入队列，(1,1)表示左上角这个格子，当前队列为：{(1,1)}，联通块加1等于1。

②取出队首的(1,1)，标记为已搜索并对其进行四连通扩展，扩展出(1,2)，删除(1,1)队列变为：{(1,2)}。

③取出队首的(1,2)，标记为已搜索并对其进行四连通扩展，扩展到了(1,1)，(1,3)，(2,2)，(1,1)已经被标记搜索过，所以只将(1,3)，(2,2)加入队列，删除队首(1,2)，队列变为：{(1,3)，(2,2)}。

④取出队首的(1,3)，没有扩展出新格子，删除队首，队列变为：{(2,2)}。

⑤取出队首的(2,2)，没有扩展出新格子，队列变为{}。完成以(1,1)开始的搜索。

⑥将(3,1)加入队列，队列变为：{(3,1)}，联通块数加1变为2。

⑦取出队首的(3,1)，没有扩展出新格子，删除队首，队列变为{}。完成以(3,1)开始的搜索。

⑧将(3,3)加入队列，队列变为：{(3,3)}，联通块数加1变为3。

⑨取出队首的(3,3)，没有扩展出新格子，删除队首，队列变为{}。完成以(3,3)开始的搜索。无法再加入新的元素，程序结束。

【参考程序】

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

const int N = 110;
const int flag[4][2] = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}};
int a[N][N],queue[N * N][2];
int n, m, ans;
bool p[N][N];

void bfs(int x,int y)
{
    int front =0, rear =2;
    queue[1][0] = x,queue[1][1] = y;
    while (front < rear-1)
    {
        ++ front;
        x = queue[front][0];
        y = queue[front][1];
        for (int i = 0;i < 4;++ i)
        {
            int x1 = x + flag[i][0];
```

```
            int y1 = y + flag[i][1];
            if (x1 < 1 || y1 < 1 || x1 > n || y1 > m || ! a[x1][y1] || p[x1]
            [y1]) continue;
            p[x1][y1] = true;
            queue[rear][0] = x1;
            queue[rear++][1] = y1;
        }
    }
}

int main()
{
  cin >> n >> m;
  for (int i = 1; i <= n; ++ i)
      for (int j = 1; j <= m; ++ j) cin >> a[i][j];
  for (int i = 1; i <= n; ++ i)
      for (int j = 1; j <= m; ++ j)
          if (a[i][j] && ! p[i][j])
          {
              ++ ans; bfs(i,j);
          }
  cout << ans << endl;
  return 0;
}
```

【上机练习】

1. 面积(area)

【问题描述】

编程计算由“*”号围成的下列图形的面积。面积计算方法是统计*号所围成的闭合曲线中点的数目。如图2—4所示，在10*10的二维数组中，“*”围住了15个点，因此面积为15。

```
0  0  0  0  0  0  0  0  0  0
0  0  0  0  *  *  *  0  0  0
0  0  0  0  *  0  0  *  0  0
0  0  0  0  0  *  0  0  *  0
0  0  *  0  0  0  *  0  *  0
0  *  0  *  0  *  0  0  *  0
0  *  0  0  *  *  0  *  *  0
```

0 0 * 0 0 0 0 * 0 0
0 0 0 * * * * * 0 0
0 0 0 0 0 0 0 0 0 0

图 2-4

【输入样例】

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

【输出样例】

15

2. 奇怪的电梯(lift)

【问题描述】

大楼的每一层楼都可以停电梯,而且第 i 层楼(1<=i<=N)上有一个数字 Ki(0<=Ki<=N)。电梯只有四个按钮:开,关,上,下。上下的层数等于当前楼层上的那个数字。当然,如果不能满足要求,相应的按钮就会失灵。例如:3 3 1 2 5 代表了 Ki(K1=3,K2=3,…),从一楼开始。在一楼,按"上"可以到 4 楼,按"下"是不起作用的,因为没有-2 楼。那么,从 A 楼到 B 楼至少要按几次按钮呢?

【输入格式】

输入文件共有二行,第 1 行为三个用空格隔开的正整数,表示 N,A,B(1<=N<=200,1<=A,B<=N),第 2 行为 N 个用空格隔开的正整数,表示 Ki。

【输出格式】

输出文件仅一行,即最少按键次数,若无法到达,则输出-1。

【输入样例】

5 1 5
3 3 1 2 5

【输出样例】

3

3. 产生数(produce)

【问题描述】

给出一个整数 n(n<=2000)和 k 个变换规则(k<=15)。规则:

① 1 个数字可以变换成另 1 个数字;

② 规则中,右边的数字不能为零。

例如:n=234,k=2,规则为

2 → 5

3 → 6

上面的整数 234 经过变换后可能产生出的整数为(包括原数)234、534、264、564 共 4 种不同的产生数。

求经过任意次的变换(0 次或多次),能产生出多少个不同的整数。仅要求输出不同整数个数。

【输入格式】

n

k

x1 y1

x2 y2

…

xn yn

【输出格式】

一个整数(满足条件的整数个数)。

【输入样例】

234

2

2 5

3 6

【输出样例】

4

4. 家庭问题(family)

【问题描述】

有 n 个人,编号为 1,2,…,n,另外还知道存在 k 个关系。一个关系的表达为二元组(α,β)形式,表示 α、β 为同一家庭的成员。

当 n、k 和 k 个关系给出之后,求出其中共有多少个家庭、最大的家庭中有多少人?

例如:n=6,k=3,三个关系为(1,2),(1,3),(4,5)。

此时,6 个人组成三个家庭,即:{1,2,3}为一个家庭,{4,5}为一个家庭,{6}单独为一个家庭,第一个家庭的人数为最多。

【输入格式】

文件的第一行为 n,k 两个整数(1<=n<=100)(用空格分隔);

接下来的 k 行,每行两个整数(用空格分隔)表示关系。

【输出格式】

两个整数(分别表示家庭个数和最大家庭人数)

【输入样例】

6 3

1 2

1 3

4 5

【输出样例】

3 3

第三章　树

树是一种非线性的数据结构，用它能很好地描述有分支和层次特性的数据集合。树型结构在现实世界中广泛存在，如社会组织机构的组织关系图就可以用树型结构来表示。树在计算机领域中也有广泛应用，如在编译系统中，用树表示源程序的语法结构。在数据库系统中，树型结构是数据库层次模型的基础，也是各种索引和目录的主要组织形式。在许多算法中，常用树型结构描述问题的求解过程、所有解的状态和求解的对策等。这些年的国内、国际信息学奥赛、大学生程序设计比赛等竞赛中，树型结构成为参赛者必备的知识之一，尤其是建立在树型结构基础之上的搜索算法。

在树型结构中，二叉树是最常用的结构，它的分支个数确定，又可以为空，并有良好的递归特性，特别适宜于程序设计，因此也常常将一般树转换成二叉树进行处理。

第一节　树的概念

一、树的定义

一棵树是由 n(n>0)个元素组成的有限集合，其中：

(1)每个元素称为结点(node)；

(2)有一个特定的结点，称为根结点或树根(root)；

(3)除根结点外，其余结点能分成 m(m>=0)个互不相交的有限集合 T_0，T_1，T_2，…，T_{m-1}。其中的每个子集又都是一棵树，这些集合称为这棵树的子树。

如图 3－1 是一棵典型的树：

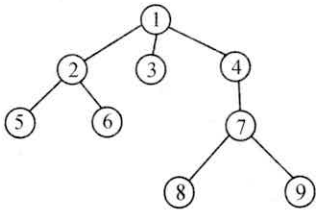


图 3－1

二、树的基本概念

(1)树是递归定义的。

(2)一棵树中至少有 1 个结点。这个结点就是根结点，它没有前驱，其余每个结点都有唯一的一个前驱结点。每个结点可以有 0 或多个后继结点。因此树虽然是非线性结构，但也是有序结构。至于前驱后继结点是哪个，还要看树的遍历方法，我们将在后面讨论。

(3)一个结点的子树个数，称为这个结点的度(degree，结点 1 的度为 3，结点 3 的度为 0)；度为 0 的结点称为叶结点(树叶 leaf，如结点 3、5、6、8、9)；度不为 0 的结点称为分支结点(如结点 1、2、4、7)；根以外的分支结点又称为内部结点(如结点 2、4、7)；树中各结点的度的

最大值称为这棵树的度(这棵树的度为 3)。

(4)在用图形表示的树型结构中,对两个用线段(称为树枝)连接的相关联的结点,称上端结点为下端结点的父结点,称下端结点为上端结点的子结点。称同一个父结点的多个子结点为兄弟结点。如结点 1 是结点 2、3、4 的父结点,结点 2、3、4 是结点 1 的子结点,它们又是兄弟结点,同时结点 2 又是结点 5、6 的父结点。称从根结点到某个子结点所经过的所有结点为这个子结点的祖先。如结点 1、4、7 是结点 8 的祖先。称以某个结点为根的子树中的任一结点都是该结点的子孙。如结点 7、8、9 都是结点 4 的子孙。

(5)定义一棵树的根结点的层次(level)为 0,其他结点的层次等于它的父结点层次加 1。如结点 2、3、4 的层次为 1,结点 5、6、7 的层次为 2,结点 8、9 的层次为 3。一棵树中所有的结点的层次的最大值称为树的深度(depth)。如这棵树的深度为 3。

(6)对于树中任意两个不同的结点,如果从一个结点出发,自上而下沿着树中连着结点的线段能到达另一结点,称它们之间存在着一条路径。可用路径所经过的结点序列表示路径,路径的长度等于路径上的结点个数减 1。如图 3-1 中,结点 1 和结点 8 之间存在着一条路径,并可用(1、4、7、8)表示这条路径,该条路径的长度为 3。注意,不同子树上的结点之间不存在路径,从根结点出发,到树中的其余结点一定存在着一条路径。

(7)森林(forest)是 m(m>=0)棵互不相交的树的集合。

三、树的存储结构

方法 1:数组,称为"父亲表示法"。

```
const int m=10;                    //树的结点数
struct node
{
    int data, parent;              //数据域,指针域
};
node tree[m];
```

优缺点:利用了树中除根结点外每个结点都有唯一的父结点这个性质。很容易找到树根,但找孩子时需要遍历整个线性表。

方法 2:树型单链表结构,称为"孩子表示法"。每个结点包括一个数据域和一个指针域(指向若干子结点),称为"孩子表示法"。假设树的度为 10,树的结点仅存放字符,则这棵树的数据结构定义如下:

```
const int m=10;                    //树的度
typedef struct node;
typedef node * tree;
struct node
{
    char data;                     //数据域
    tree child[m];                 //指针域,指向若干孩子结点
};
tree t;
```

缺陷:只能从根(父)结点遍历到子结点,不能从某个子结点返回到它的父结点。但程序中确实需要从某个结点返回到它的父结点时,就需要在结点中多定义一个指针变量存放其父结点的信息。这种结构又叫带逆序的树型结构。

方法3:树型双链表结构,称为"父亲孩子表示法"。每个结点包括一个数据域和两个指针域(一个指向若干子结点,一个指向父结点)。假设树的度为10,树的结点仅存放字符,则这棵树的数据结构定义如下:

```
const int m=10;              //树的度
typedef struct node;
typedef node * tree;         //声明 tree 是指向 node 的指针类型
struct node
{
    char data;               //数据域
    tree child[m];           //指针域,指向若干孩子结点
    tree father;             //指针域,指向父亲结点
};
tree t;
```

方法4:二叉树型表示法,称为"孩子兄弟表示法"。它是一种双链表结构,但每个结点包括一个数据域和两个指针域(一个指向该结点的第一个孩子结点,一个指向该结点的下一个兄弟结点)。假设树的度为10,树的结点仅存放字符,则这棵树的数据结构定义如下:

```
typedef struct node;
typedef node * tree;
struct node
{
    char data;               //数据域
    tree firstchild, next;   //指针域,分别指向第一个孩子结点和下一个兄弟结点
};
tree t;
```

例3.1 找树根和孩子

【问题描述】

给定一棵树,输出树的根root,孩子最多的结点max以及他的孩子。

【输入格式】

第1行:n(结点个数<=100),m(边数<=200)。

以下m行:每行两个结点x和y,表示y是x的孩子(x,y<=1000)。

【输出格式】

第1行:树根:root。

第2行:孩子最多的结点max。

第3行:max的孩子。

【输入样例】	【输出样例】
8 7	4
4 1	2

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

【参考程序】

```
#include<iostream>
using namespace std;
int n,m,tree[101]={0};
int main()
{
  int i,x,y,root,maxroot,sum=0,j,Max=0;
  cin>>n>>m;
  for(i=1;i<=m;i++)
  {
    cin>>x>>y;
    tree[y]=x;
  }
  for(i=1;i<=n;i++)   //找出树根
      if(tree[i]==0)
      {
        root=i;break;
      }
  for(i=1;i<=n;i++)   //找孩子最多的结点
  {
    sum=0;
    for(j=1;j<=n;j++)
      if(tree[j]==i) sum++;
    if(sum>Max)
    {
      Max=sum;maxroot=i;
    }
  }
  cout<<root<<endl<<maxroot<<endl;
  for(i=1;i<=n;i++)
    if(tree[i]==maxroot) cout<<i<<" ";
  return 0;
}
```

四、树的遍历

在应用树结构解决问题时,往往要求按照某种次序获得树中全部结点的信息,这种操作叫作树的遍历。遍历的方法有多种,常用的有:

A. 先序(根)遍历:先访问根结点,再从左到右按照先序思想遍历各棵子树。如图 3-2 先序遍历的结果为:125634789;

B. 后序(根)遍历:先从左到右遍历各棵子树,再访问根结点。如图 3-2 后序遍历的结果为:562389741;

C. 层次遍历:按层次从小到大逐个访问,同一层次按照从左到右的次序。如图 3-2 层次遍历的结果为:123456789;

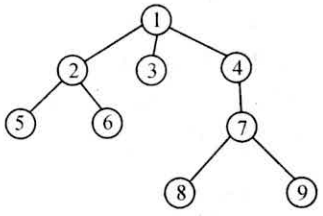


图 3-2

D. 叶结点遍历:有时把所有的数据信息都存放在叶结点中,而其余结点都是用来表示数据之间的某种分支或层次关系,这种情况就用这种方法。如图 3-2 按照这个思想访问的结果为:56389;

大家可以看出,A、B 两种方法的定义是递归的,所以在程序实现时往往也是采用递归的思想,即通常所说的"深度优先搜索"。如用先序遍历编写的过程如下:

```
void tral(tree t, int m)
{
    if (t)
    {
        cout << t->data << endl;
        for(int i=0 ; i < m ; i++)
          tral(t->child[i], m);
    }
}
```

C 方法应用也较多,实际上是我们讲的"广度优先搜索"。思想如下:若某个结点被访问,则该结点的子结点应记录,等待被访问。顺序访问各层次上结点,直至不再有未访问过的结点。为此,引入一个队列来存储等待访问的子结点,设一个队首和队尾指针分别表示出队、进队的下标。程序框架如下:

```
const int n=100;
int head, tail, i;
tree q[n];
tree p;
void work()
{
    tail=head=1;
    q[tail]=t;
    tail++;                    //队尾为空
    while(head < tail)
    {
```

```
            p=q[head];
            head++;
            cout << p->data << '';
            for(i=0 ; i < m ; i++)
            if(p->child[i])
            {
                q[tail]=p->child[i];
                tail++;
            }
        }
    }
```

例 3.2　单词查找树

【问题描述】

在进行文法分析的时候，通常需要检测一个单词是否在我们的单词列表里。为了提高查找和定位的速度，通常都画出与单词列表所对应的单词查找树，其特点如下：

1. 根结点不包含字母，除根结点外每一个结点都仅包含一个大写英文字母。

2. 从根结点到某一结点，路径上经过的字母依次连起来所构成的字母序列，称为该结点对应的单词。单词列表中的每个单词，都是该单词查找树某个结点所对应的单词。

3. 在满足上述条件下，该单词查找树的结点数最少。

4. 例如图 3－3 左边的单词列表就对应于右边的单词查找树。注意，对一个确定的单词列表，请统计对应的单词查找树的结点数(包含根结点)。

A
AN
ASP
AS
ASC
ASCII
BAS
BASIC

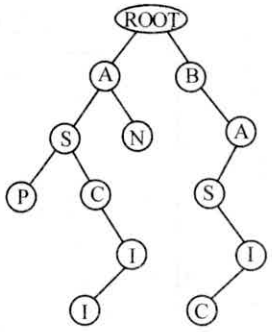


图 3－3

【输入格式】

输入文件名为 word.in，该文件为一个单词列表，每一行仅包含一个单词和一个换行/回车符。每个单词仅由大写的英文字母组成，长度不超过 63 个字母 。文件总长度不超过 32K，至少有一行数据。

【输出格式】

输出文件名为 word.out，该文件中仅包含一个整数，该整数为单词列表对应的单词查找树的结点数。

【输入样例】

A
AN
ASP
AS
ASC
ASCII
BAS
BASIC

【输出样例】

13

【算法分析】

首先要对建树的过程有一个了解。对于当前被处理的单词和当前树:在根结点的子结点中找单词的第一位字母,若存在,则进而在该结点的子结点中寻找第二位…如此下去直到单词结束,即不需要在该树中添加结点;或单词的第 n 位不能被找到,即将单词的第 n 位及其后的字母依次加入单词查找树中去。但,本问题只是问你结点总数,而非建树方案,且有32K 文件,所以应该考虑能不能不通过建树就直接算出结点数? 为了说明问题的本质,我们给出一个定义:一个单词相对于另一个单词的差:设单词 1 的长度为 L,且与单词 2 从第 N 位开始不一致,则说单词 1 相对于单词 2 的差为 L−N+1,这是描述单词相似程度的量。可见,将一个单词加入单词树的时候,须加入的结点数等于该单词树中已有单词的差的最小值。

单词的字典顺序排列后的序列则具有类似的特性,即在一个字典顺序序列中,第 m 个单词相对于第 m−1 个单词的差必定是它对于前 m−1 个单词的差中最小的。于是,得出建树的等效算法:

① 读入文件;

② 对单词列表进行字典顺序排序;

③ 依次计算每个单词对前一单词的差,并把差累加起来。注意:第一个单词相对于"空"的差为该单词的长度;

④ 累加和再加上 1(根结点),输出结果。

就给定的样例,按照这个算法求结点数的过程如下表:

原单词列表	排序后的列表	差值	总计	输出
A	A	1	12	13
AN	AN	1		
ASP	AS	1		
AS	ASC	1		
ASC	ASCII	2		
ASCII	ASP	1		
BAS	BAS	3		
BASIC	BASIC	2		

【数据结构】

先确定 32K(32 * 1024=32768 字节)的文件最多有多少单词和字母。当然应该尽可能地存放较短的单词。因为单词不重复,所以长度为 1 的单词(单个字母)最多 26 个;长度为 2 的单词最多为 26 * 26=676 个;因为每个单词后都要一个换行符(换行符在计算机中占 2 个字节),所以总共已经占用的空间为:(1+2) * 26+(2+2) * 676=2782 字节;剩余字节(32768−2782=29986 字节)分配给长度为 3 的单词(长度为 3 的单词最多有 26 * 26 * 26=17576 个)有 29986/(3+2)≈5997。所以单词数量最多为 26+676+5997=6699。

定义一个数组:string a[32768];把所有单词连续存放起来,用选择排序或快排对单词进行排序。

【参考程序】

```
#include <iostream>
#include <cstdio>
#include <string>
using namespace std;
int i, j, n, t, k;
string a[8001];                    //数组可以到 32768
string s;
int main()
{
    freopen("word.in", "r", stdin);
    freopen("word.out", "w", stdout);
    while(cin >> a[++n]);//读入文件中的单词并且存储到数组中
    n--;
    for(i=1 ; i < n ; i++)  //从小到大排序,可改为快排 sort(a+1, a+n+1)
        for(j=i+1 ; j <=n ; j++)
        if(a[i] > a[j])          //两个单词进行交换
        {
            s=a[i];
            a[i]=a[j];
            a[j]=s;
        }
    t=a[1].length();             //先累加第 1 个单词的长度
    for(i=2 ; i <=n ; i++)//依次计算每个单词对前一单词的差
    {
        j=0;
        while(a[i][j]==a[i-1][j] && j < a[i-1].length()) j++;
                                  //求两个单词相同部分的长度
        t+=a[i].length()-j;  //累加两个单词的差 a[i].length()-j
    }
    cout << t+1 << endl;
    return 0;
}
```

第二节　二叉树

一、二叉树基本概念

二叉树(binary tree,简写成 BT)是一种特殊的树型结构,它的度数为 2 的树。即二叉树的每个结点最多有两个子结点。每个结点的子结点分别称为左孩子、右孩子,它的两棵子树分别称为左子树、右子树。二叉树有 5 中基本形态:

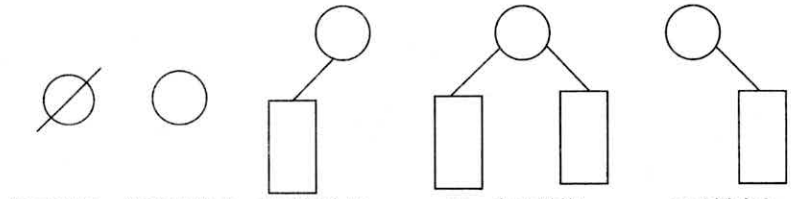


图 3-4

前面引入的树的术语也基本适用于二叉树,但二叉树与树也有很多不同,如:首先二叉树的每个结点至多只能有两个子结点,二叉树可以为空,二叉树一定是有序的,通过它的左、右子树关系体现出来。

二、二叉树的性质

【性质 1】 在二叉树的第 i 层上最多有 2^{i-1} 个结点(i>=1)。

证明:很简单,用归纳法:当 i=1 时,$2^{i-1}=1$ 显然成立;现在假设第 i-1 层时命题成立,即第 i-1 层上最多有 2^{i-2} 个结点。由于二叉树的每个结点的度最多为 2,故在第 i 层上的最大结点数为第 i-1 层的 2 倍,即 $2*2^{i-2}=2^{i-1}$。

【性质 2】 深度为 k 的二叉树至多有 2^k-1 个结点(k>=1)。

证明:在具有相同深度的二叉树中,仅当每一层都含有最大结点数时,其树中结点数最多。因此利用性质 1 可得,深度为 k 的二叉树的结点数至多为:

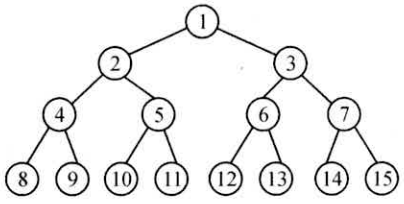
$2^0+2^1+\cdots+2^{k-1}=2^k-1$

故命题正确。

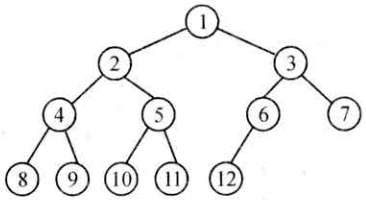
特别:一棵深度为 k 且有 2^k-1 个结点的二叉树称为满二叉树。如图 3-5A 为深度为 4 的满二叉树,这种树的特点是每层上的结点数都是最大结点数。

可以对满二叉树的结点进行连续编号,约定编号从根结点起,自上而下,从左到右,由此引出完全二叉树的定义,深度为 k,有 n 个结点的二叉树当且仅当其每一个结点都与深度为 k 的满二叉树中编号从 1 到 n 的结点一一对应时,称为完全二叉树。

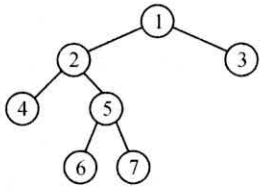
图 3-5B 就是一个深度为 4,结点数为 12 的完全二叉树。它有如下特征:叶结点只可能在层次最大的两层上出现;对任一结点,若其右分支下的子孙的最大层次为 m,则在其左分支下的子孙的最大层次必为 m 或 m+1。图 3-5C、D 不是完全二叉树,请大家思考为什么?



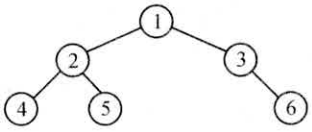
图A



图B



图C



图D

图 3－5

【性质 3】 对任意一棵二叉树，如果其叶结点数为 n_0，度为 2 的结点数为 n_2，则一定满足：$n_0=n_2+1$。

证明：因为二叉树中所有结点的度数均不大于 2，所以结点总数（记为 n）应等于 0 度结点数 n_0、1 度结点 n_1 和 2 度结点数 n_2 之和：

$n=n_0+n_1+n_2$……（式子 1）

另一方面，1 度结点有一个孩子，2 度结点有两个孩子，故二叉树中孩子结点总数是：

n_1+2n_2

树中只有根结点不是任何结点的孩子，故二叉树中的结点总数又可表示为：

$n=n_1+2n_2+1$ ……（式子 2）

由式子 1 和式子 2 得到：

$n_0=n_2+1$

【性质 4】 具有 n 个结点的完全二叉树的深度为 $floor(\log_2^n)+1$

证明：假设深度为 k，根据完全二叉树的定义，前面 k－1 层一定是满的，所以 $n>2^{k-1}-1$。但 n 又要满足 $n<=2^k-1$，所以，$2^{k-1}-1<n<=2^k-1$，变换一下为 $2^{k-1}<=n<2^k$。

以 2 为底取对数得到：$k-1<=\log_2^n<k$。而 k 是整数，所以 $k=floor(\log_2^n)+1$。

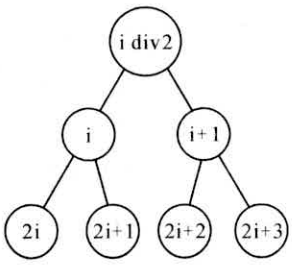
【性质 5】 对于一棵 n 个结点的完全二叉树，对任一个结点（编号为 i），有：

①如果 i＝1，则结点 i 为根，无父结点；如果 i＞1，则其父结点编号为 i/2。

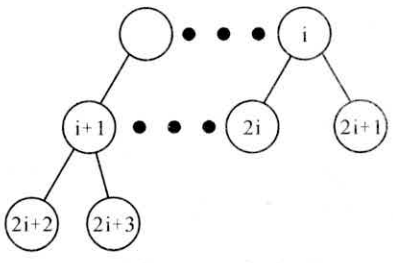
如果 2＊i＞n，则结点 i 无左孩子（当然也无右孩子，为什么？即结点 i 为叶结点）；否则左孩子编号为 2＊i。

②如果 2＊i＋1＞n，则结点 i 无右孩子；否则右孩子编号为 2＊i＋1。

证明：略，我们只要验证一下即可。总结如图 3－6：



结点i和i+1在同一层上



结点i和i+1不在同一层上

图 3－6

三、二叉树的存储结构

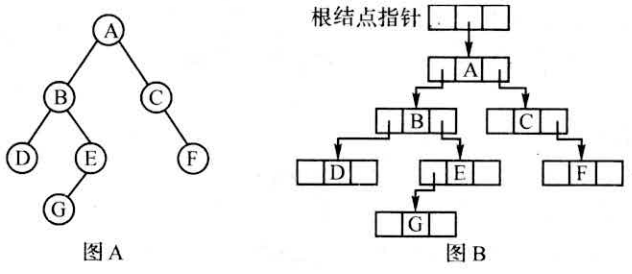
(1)链式存储结构,即单链表结构或双链表结构(同树)。数据结构修改如下:

```
typedef struct node;
typedef node * tree;
struct node
{
    char data;
    tree lchild, rchild;
};
tree bt;
```

或:

```
typedef struct node;
typedef node * tree;
struct node
{
    char data;
    tree lchild, rchild,father;
};
tree bt;
```

如图 3－7A 的一棵二叉树用单链表就可以表示成如图 3－7B 的形式:



图A　　图B

图 3－7

(2)顺序存储结构,即几个数组加一个指针变量。数据结构修改如下:

```
const int n=10;
char data[n];
char lchild[n];
char rchild[n];
int bt;                                    //根结点指针
```

二叉树在处理表达式时经常用到,一般用叶结点表示运算元,分支结点表示运算符。这样的二叉树称为表达式树。如现在有一个表达式:(a+b/c)*(d－e),可以用图 3－8 表示:

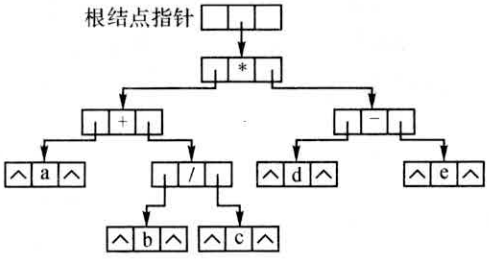


图 3－8

数据结构定义如下：

按表达式的书写顺序逐个编号，分别为 1..9，注意表达式中的所有括号在树中是不出现的，因为表达式树本身就是有序的。叶结点的左右子树均为空(用 0 表示)。

```
char data[9]={'a', '+', 'b', '/', 'c', '*', 'd', '-', 'e'};
int lchild[9]={0,1,0,3,0,2,0,7,0};
int rchild[9]={0,4,0,5,0,8,0,9,0};
int bt;                    //根结点指针，初值=6，指向'*'
```

二叉树的操作：最重要的是遍历二叉树，但基础是建一棵二叉树、插入一个结点到二叉树中、删除结点或子树等。

例 3.3　医院设置

【问题描述】

设有一棵二叉树(如图 3－9)，其中圈中的数字表示结点中居民的人口，圈边上数字表示结点编号。现在要求在某个结点上建立一个医院，使所有居民所走的路程之和为最小，同时约定，相邻结点之间的距离为 1。就本图而言，若医院建在 1 处，则距离和＝4＋12＋2＊20＋2＊40＝136；若医院建在 3 处，则距离和＝4＊2＋13＋20＋40＝81…

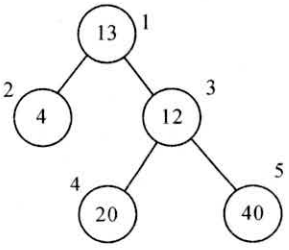


图 3－9

【输入格式】

输入文件名为 hospital.in，其中第一行一个整数 n，表示树的结点数(n<＝100)。接下来的 n 行每行描述了一个结点的状况，包含三个整数，整数之间用空格(一个或多个)分隔，其中：第一个数为居民人口数；第二个数为左链接，为 0 表示无链接；第三个数为右链接，为 0 表示无链接。

【输出格式】

输出文件名为 hospital.out，该文件只有一个整数，表示最小距离和。

【输入样例】

```
5
13 2 3
 4 0 0
12 4 5
20 0 0
40 0 0
```

【输出样例】

```
81
```

【算法分析】

这是一道简单的二叉树应用问题，问题中的结点数并不多，数据规模也不大，采用邻接

矩阵存储,用 Floyed 求出任意两结点之间的最短路径,然后穷举医院可能建立的 n 个结点位置,找出一个最小距离的位置即可。当然也可以用双链表结构或带父结点信息的数组存储结构来解决,但实际操作稍微麻烦了一点。

【参考程序】

```
#include <iostream>
#include <cstdlib>
#include <cstdio>
using namespace std;
int a[101];
int g[101][101];
int main()
{
    int n, i, j, k, l, r, min, total;
    freopen("hospital.in", "r", stdin);
    freopen("hospital.out", "w", stdout);
    cin >> n;
    for(i=1 ; i <=n ; i++)
        for(j=1 ; j <=n ; j++)
            g[i][j]=1000000;
    for(i=1 ; i <=n ; i++)              //读入、初始化
    {
        g[i][i]=0;
        cin >> a[i] >> l >> r;
        if(l > 0) g[i][l]=g[l][i]=1;
        if(r > 0) g[i][r]=g[r][i]=1;
    }
    for(k=1 ; k <=n ; k++) //用 Floyed 求任意两结点之间的最短路径
        for(i=1 ; i <=n ; i++)
        if(i != k)
            for(j=1 ; j <=n ; j++)
            if(i != j && k != j && g[i][k]+g[k][j] < g[i][j])
                g[i][j]=g[i][k]+g[k][j];
    min=0x7fffffff;
    for(i=1 ; i <=n ; i++)              //穷举医院建在 n 个结点,找出最短距离
    {
        total=0;
        for(j=1 ; j <=n ; j++)
            total+=g[i][j] * a[j];
        if(total < min) min=total;
    }
```

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

后记：

在各种竞赛中经常遇到这样的问题：n－1 条公路连接着 n 个城市，从每个城市出发都可以通过公路到达其他任意的城市。每个城市里面都有一定数量的居民，但是数量并不一定相等，每条公路的长度也不一定相等。x 公司（或者是政府）决定在某一个城市建立一个医院/酒厂/游乐场…问：将它建在哪里，可以使得所有的居民移动到那里的总耗费最小？这种题目都是本题的“变型”，一般称为“树的中心点问题”。除了简单的穷举法外，还有更好的时间复杂度为 O(n)的算法，我们将在后面的章节中继续讨论。

四、遍历二叉树

在二叉树的应用中，常常要求在树中查找具有某种特征的结点，或者对全部结点逐一进行某种处理，这就是二叉树的遍历问题。所谓二叉树的遍历是指按一定的规律和次序访问树中的各个结点，而且每个结点仅被访问一次。“访问”的含义很广，可以是对结点作各种处理，如输出结点的信息等。遍历一般按照从左到右的顺序，共有 3 种遍历方法，先（根）序遍历，中（根）序遍历，后（根）序遍历。

（一）先序遍历的操作定义如下：

若二叉树为空，则空操作，否则：

①访问根结点

②先序遍历左子树

③先序遍历右子树

先序遍历图 3－10 结果为：124753689

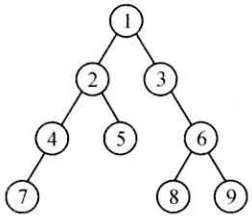


图 3－10

```
void preorder(tree bt)          //先序遍历根结点为 bt 的二叉树的递归算法
{
    if(bt)
    {
      cout << bt->data;
      preorder(bt->lchild);
      preorder(bt->rchild);
    }
}
```

(二)中序遍历的操作定义如下:

若二叉树为空,则空操作,否则:

①中序遍历左子树

②访问根结点

③中序遍历右子树

中序遍历图 3-10 结果为:742513869

```
void inorder(tree bt)              //中序遍历根结点为 bt 的二叉树的递归算法
{
    if(bt)
    {
        inorder(bt->lchild);
        cout << bt->data;
        inorder(bt->rchild);
    }
}
```

(三)后序遍历的操作定义如下:

若二叉树为空,则空操作,否则:

①后序遍历左子树

②后序遍历右子树

③访问根结点

后序遍历图 3-10 结果为:745289631

```
void postorder(tree bt)            //后序遍历根结点为 bt 的二叉树的递归算法
{
    if(bt)
    {
        postorder(bt->lchild);
        postorder(bt->rchild);
        cout << bt->data;
    }
}
```

当然,我们还可以把递归过程改成用栈实现的非递归过程,以先序遍历为例,其他的留给读者完成。

```
void preorder(tree bt)                          //先序遍历 bt 所指的二叉树
{
    tree stack[n];                              //栈
    int top=-1;                                 //栈顶指针
    tree P;
    while(bt || top)
    {
        while(bt)                               //非叶结点
```

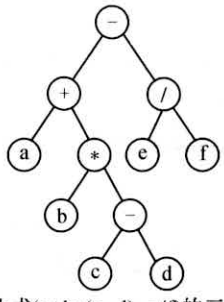
```
        {
            cout << bt->data;                    //访问根
            stack[++top]=bt->rchild;             //右子树压栈
            bt=bt->lchild;                       //遍历左子树
        }
        if(top)                                  //栈中所有元素出栈,遍历完毕
        {
            bt=stack[top--];
        }
    }
}
```

关于前面讲的表达式树，我们可以分别用先序、中序、后序的遍历方法得出完全不同的遍历结果，如对于图 3－11 遍历结果如下，它们正好对应着表达式的 3 种表示方法。

－＋a＊b－cd/ef　（前缀表示、波兰式）

a＋b＊c－d－e/f　（中缀表示）

abcd－＊＋ef/－　（后缀表示、逆波兰式）



表达式(a+b*(c-d)-e/f)的二叉树

图 3－11

五、二叉树的其他重要操作

除了“遍历”以外，二叉树的其他重要操作还有：建立一棵二叉树、插入一个结点到二叉树中、删除结点或子树等。

1. 建立一棵二叉树

```
void pre_crt(tree &bt) //按先序次序输入二叉树中结点的值生成
{
    char ch;
    ch=getchar();//二叉树的单链表存储结构,bt 为指向根结点的指针,'$'表示空树
    if(ch !='$')
    {
        bt=new node;                             //建根结点
        bt->data=ch;
        pre_crt(bt->lchild);                     //建左子树
        pre_crt(bt->rchild);                     //建右子树
    }
    else bt=NULL;
}
```

2. 删除二叉树

```
void dis(tree &bt)                               //删除二叉树
{
    if(bt)
```

```
    {
        dis(bt->lchild);                    //删左子树
        dis(bt->rchild);                    //删右子树
        delete bt;                          //释放父结点
    }
}
```

3. 插入一个结点到排序二叉树中

```
void insert(tree &bt, int n)                //插入一个结点到排序二叉树中
{
    if(bt)
    {
        if(n < bt->data) insert(bt->lchild, n);
        else if(n > bt->data) insert(bt->rchild, n);
    }
    else
    {
        bt=new node;                        //新开一个空间
        bt->data=n;
        bt->lchild=bt->rchild=NULL;
    }
}
```

4. 在排序二叉树中查找一个数,找到返回该结点,否则返回 NULL

```
tree findn(tree bt, int n)      //在二叉树中查找一个数,找到返回该结点,否则返回 NULL。
{
    if(bt)
    {
        if(n < bt->data) findn(bt->lchild, n);
        else if(n > bt->data) findn(bt->rchild, n);
        else return bt;
    }
    else return NULL;
}
```

5. 用嵌套括号表示法输出二叉树

```
void print(tree bt)                         //用嵌套括号表示法输出二叉树
{
    if(bt)
    {   cout << bt->data;
        if(bt->lchild || bt->rchild)
        {   cout << '(';
            print(bt->lchild);
```

```
            if(bt->rchild) cout << ',';
            print(bt->rchild);
            cout << ')';
        }
    }
}
```

下面我们换个角度考虑这个问题，从二叉树的遍历已经知道，任意一棵二叉树结点的先序序列和中序序列是唯一的。反过来，给定结点的先序序列和中序序列，能否确定一棵二叉树呢？又是否唯一呢？

由定义，二叉树的先序遍历是先访问根结点，再遍历左子树，最后遍历右子树，即在结点的先序序列中，第一个结点必是根，假设为 root。再结合中序遍历，因为中序遍历是先遍历左子树，再访问根，最后遍历右子树，所以结点 root 正好把中序序列分成了两部分，root 之前的应该是左子树上的结点，root 之后的应该是右子树上的结点。依此类推，便可递归得到整棵二叉树。

结论：已知先序序列和中序序列可以确定出二叉树；

已知中序序列和后序序列也可以确定出二叉树；

但，已知先序序列和后序序列却不可以确定出二叉树；为什么？举个 3 个结点的反例。

例如：已知结点的先序序列为 ABCDEFG，中序序列为 CBEDAFG。构造出二叉树，过程见图 3－12：

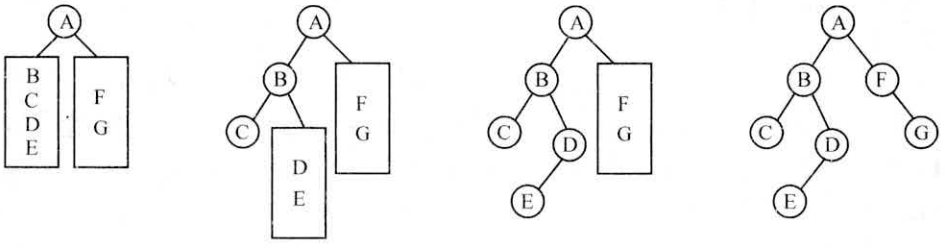


图 3－12

例 3.4　求后序遍历

【问题描述】

输入一棵二叉树的先序和中序遍历序列，输出其后序遍历序列。

【输入格式】

输入文件为 tree.in，共两行，第 1 行一个字符串，表示树的先序遍历，第 2 行一个字符串，表示树的中序遍历。树的结点一律用小写字母表示。

【输出格式】

输出文件为 tree.out，仅一行，表示树的后序遍历序列。

【输入样例】

abdec

dbeac

【输出样例】

debca

【参考程序】

```
#include <cstring>
#include <iostream>
```

```
#include <cstdio>
using namespace std;
string s1, s2;
void calc(int l1, int r1, int l2, int r2)
{
  int m=s2.find(s1[l1]);
  if(m > l2) calc(l1+1, l1+m-l2, l2, m-1);
  if(m < r2) calc(l1+m-l2+1, r1, m+1, r2);
  cout << s1[l1];
}
int main()
{
  freopen("tree.in", "r", stdin);
  freopen("tree.out", "w", stdout);
  cin >> s1 >> s2;
  calc(0, s1.length()-1, 0, s2.length()-1);
  cout << endl;
  return 0;
}
```

例 3.5 扩展二叉树

【问题描述】

由于先序、中序和后序序列中的任一个都不能唯一确定一棵二叉树，所以对二叉树做如下处理，将二叉树的空结点用“·”补齐，如图 3－13 所示。我们把这样处理后的二叉树称为原二叉树的扩展二叉树，扩展二叉树的先序和后序序列能唯一确定其二叉树。

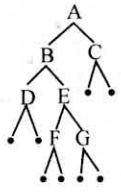


图 3－13

现给出扩展二叉树的先序序列，要求输出其中序和后序序列。

【输入样例】

ABD..EF..G..C..

【输出样例】

DBFEGAC

DFGEBCA

【参考程序】

```
#include <iostream>
#include <stdlib.h>
#include <string>
#include <cstring>
#include <algorithm>
#include <cstdio>
using namespace std;
typedef struct node;
typedef node *tree;
struct node
```

```
{
    char data;
    tree lchild, rchild;
};
tree bt;
int i;
string s;
void build(tree &bt)                    //建树
{
    if(s[++i]!='.')
    {
        bt=new node;
        bt->data=s[i];
        build(bt->lchild);
        build(bt->rchild);
    }
    else bt=NULL;
}
void printzx(tree bt)                   //输出中序序列
{
    if(bt)
    {
        printzx(bt->lchild);
        cout << bt->data;
        printzx(bt->rchild);
    }
}
void printhx(tree bt)                   //输出后序序列
{
    if(bt)
    {
        printhx(bt->lchild);
        printhx(bt->rchild);
        cout << bt->data;
    }
}
int main()
{
    freopen("tree_b.in", "r", stdin);
    freopen("tree_b.out", "w", stdout);
```

```
    cin >> s;
    i=-1;
    build(bt);
    printzx(bt);
    cout << endl;
    printhx(bt);
    cout << endl;
    return 0;
}
```

六、普通树转换成二叉树

由于二叉树是有序的,而且操作和应用更广泛,所以在实际使用时,我们经常把普通树转换成二叉树进行操作。如何转换呢?一般方法如下:

1. 将树中每个结点除了最左边的一个分支保留外,其余分支都去掉。
2. 从最左边结点开始画一条线,把同一层上的兄弟结点都连起来。
3. 以整棵树的根结点为轴心,将整棵树顺时针大致旋转 45 度。

把图 3-14A 所示的普通树转换成二叉树的过程如图 3-14B 所示:

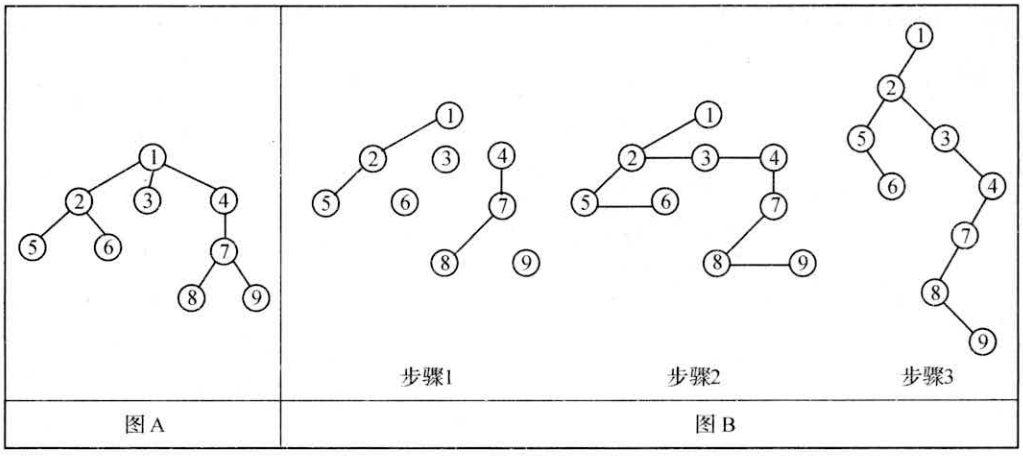


图 3-14

同样我们可以把森林也转换成二叉树处理,假设 F={T1,T2,…,Tm}是森林,则可按如下规则转换成一棵二叉树 B=(root,lb,rb)。

1. 若 F 为空,即 m=0,则 B 为空树。
2. 若 F 非空,即 m! =0,则 B 的根 root 即为森林中第一棵树的根 root(T1);B 的左子树 lb 是从 T1 中根结点的子树森林 F1={T11,T12,…,T1m}转换而成的二叉树;其右子树 rb 是从森林 F'={T2,T3,…,Tm}转换而成的二叉树。

七、树的计数问题

具有 n 个结点的不同形态的二叉树有多少棵?具有 n 个结点的不同形态的树有多少

棵？首先了解两个概念，“相似二叉树”是指两者都为空树或者两者均不为空树，且它们的左右子树分别相似。“等价二叉树”是指两者不仅相似，而且所有对应结点上的数据元素均相同。二叉树的计数问题就是讨论具有 n 个结点、互不相似的二叉树的数目 Bn。

在 n 很小时，很容易得出，B0＝1，B1＝1，B2＝2，B3＝5(图 3－15)。

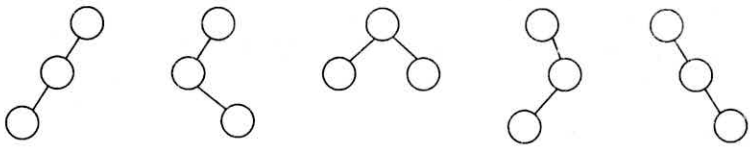


图 3－15

一般情况，一棵具有 n(n＞1)个结点的二叉树可以看成是由一个根结点、一棵具有 i 个结点的左子树和一棵具有 n－i－1 个结点的右子树组成，其中 0＜＝i＜＝n－1，

由此不难得出下列递归公式：

$B_0=1$

$Bn=\sum_{i=0}^{n=1}B_iB_{n-i-1}(n>=1)$

我们可以利用生成函数讨论这个递归公式，得出：$Bn=C_{2n}^{n}/(n+1)$。

类推到具有 n 个结点、互不相似的多叉树的数目 Tn：

由于树可以转换成二叉树且转换之后的根节点没有右儿子，所以，可以推出：$Tn=B_{n-1}$。

【课堂练习】

1. 一棵完全二叉树的结点总数为 18，其叶结点数为_____。

A. 7 个　　B. 8 个　　C. 9 个　　D. 10 个

2. 二叉树第 10 层的结点数的最大数目为_____。

A. 10　　B. 100　　C. 512　　D. 1024

3. 一棵深度为 K 的满二叉树有_____个结点。

A. 2^K-1　　B. 2^K　　C. 2 * K　　D. 2 * K－1

4. 对任何一棵二叉树 T，设 n_0、n_1、n_2 分别是度数为 0、1、2 的顶点数，则下列判断中正确的是_____。

A. $n_0=n_2+1$　　B. $n_1=n_0+1$　　C. $n_2=n_0+1$　　D. $n_2=n_1+1$

5. 一棵 n 个结点的完全二叉树，则该二叉树的高度 h 为_____。

A. n/2　　B. log(n)　　C. log(n)/2　　D. ⌊log(n)⌋＋1

6. 一棵完全二叉树上有 1001 个结点，其中叶子结点的个数是_____。

A. 250　　B. 500　　C. 254　　D. 501

7. 如果一棵二叉树有 N 个度为 2 的结点，M 个度为 1 的结点，则该树的叶子个数为_____。

A. N＋1　　B. 2 * N－1　　C. N－1　　D. M＋N－1

8. 一棵非空二叉树的先序遍历序列和后序遍历序列正好相反，则该二叉树一定满足_____。

A. 所有结点均无左孩子　　B. 所有结点均无右孩子

C. 只有一个叶子结点　　D. 是任意一棵二叉树

9. 将有关二叉树的概念推广到三叉树，则一棵有 244 个结点的完全三叉树的高度是_____。

A. 4　　B. 5　　C. 6　　D. 7

10. 在一棵具有 K 层的满三叉树中，结点总数为_____。

A. $(3^k-1)/2$　　B. 3^k-1　　C. $(3^k-1)/3$　　D. 3^k

11. 设树 T 的高度为 4，其中度为 1、2、3 和 4 的结点个数分别为 4、2、1、1，则 T 中的叶子数为_____。

A. 5　　B. 6　　C. 7　　D. 8

12. 一棵树 T 有 2 个度数为 2 的结点、有 1 个度数为 3 的结点、有 3 个度数为 4 的结点，那么树 T 有_____个树叶。

A. 14　　B. 6　　C. 18　　D. 7

13. 某二叉树中序序列为 abcdefg，后序序列为 bdcafge，则先序序列是_____。

A. egfacbd　　B. eacbdgf　　C. eagcfbd　　D. 以上答案都不对

14. 已知某二叉树的中序遍历序列是 debac，后序遍历序列是 dabec，则它的先序遍历序列是_____。

A. acbed　　B. decab　　C. deabc　　D. cedba

15. 一棵二叉树的中序遍历序列为 DGBAECHF，后序遍历序列为 GDBEHFCA，则先序遍历序列是_____。

A. ABCDFGHE　　B. ABDGCEFH　　C. ACBGDHEF　　D. ACEFHBGD

16. 已知一棵二叉树的先序序列为 ABDEGCFH，中序序列为 DBGEACHF，则该二叉树的层次序列为_____。

A. GEDHFBCA　　B. DGEBHFCA　　C. ABCDEFGH　　D. ACBFEDHG

17. 已知一棵二叉树的先序遍历结果为 ABDECFHJIG，中序遍历的结果为 DBEAJHFICG，则这棵二叉树的深度为_____。

A. 3　　B. 4　　C. 5　　D. 6

18. 二叉树的先序遍历和中序遍历如下：先序遍历：EFHIGJK，中序遍历：HFIEJKG。该二叉树根的右子树的根是_____。

A. E　　B. F　　C. G　　D. H

19. 中缀表达式 A－(B＋C/D)＊E 的后缀形式是_____。

A. AB－C＋D/E＊　B. ABC＋D/－E＊　C. ABCD/E＊＋－　D. ABCD/＋E＊－

20. 设森林 F 对应的二叉树为 B，它有 M 个结点，B 的根为 P，P 的右子树结点个数为 N，森林 F 中第一棵树的结点个数是_____。

A. M－N　　B. M－N－1

C. N＋1　　D. 条件不充分，无法确定

【练习答案】

1. C　2. C　3. A　4. A　5. D　6. D　7. A　8. A　9. B　10. A　11. D　12. A　13. B　14. D　15. B　16. C　17. C　18. C　19. D　20. A

7. [解析]此树的边数为 2N＋M，故而总共有 2N＋M＋1 个顶点。除去度为 1、2 的顶

点，度为 0 的结点（即叶子结点）有(2N+M+1)-(N+M)=N+1。答案：A。

12.［解析］设树 T 有 n 个结点，m 条边。边数为结点的度数之和，即 m=2×2+1×3+3×4=19，n=m+1=20。n 个结点中有 1+2+3=6 个分支结点，有叶结点 20-6=14 个。答案：A。

15.［解析］中序遍历 DGBAECHF 和后序遍历 GDBEHFCA 唯一对应一棵二叉树先序遍历为 ABDGCEFH。答案：B。

16.［解析］由先序序列和中序序列可以唯一地确定一棵二叉树，根据题设中的先序序列和中序序列确定的二叉树为：

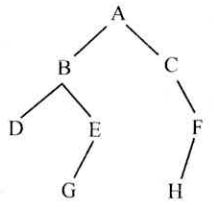


图 3-16

由此可见，其层次序列为 ABCDEFGH。答案：C

17.［解析］由题目给出二叉树先序遍历结点的顺序：ABDECFHJIG 可知结点 A 为二叉树的根结点。再由中序遍历的结点顺序：DBEAJHFICG 可知结点 A 前的 DBE 为根结点 A 的左子树的结点，而 JHFICG 为根结点 A 的右子树的结点。

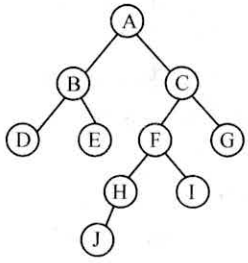


图 3-17

先来看 A 结点的左子树的结点 DBE。在先序遍历顺序中，B 结点在 A 结点之后，说明 B 结点是左子树的根结点，D 结点又在 B 结点之后，则 D 是 B 的左子树的根结点。结点 E 由中序遍历的顺序可知是 B 的右子树的根结点。同样求出右子树 JHFICG，如图 3-17。

由图可知，二叉树的深度为 5。答案：C。

19.［解析］本题主要考查学生怎样将中缀表达式转换为后缀表达式。可以先画出该二叉树，再对其进行后根遍历即为答案。答案：D。

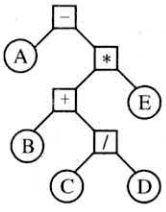


图 3-18

【上机练习】

1. 小球(drop)

【问题描述】

许多的小球一个一个的从一棵满二叉树上掉下来组成 FBT(Full Binary Tree,满二叉树),每一时间,一个正在下降的球第一个访问的是非叶子结点。然后继续下降时,或者走右子树,或者走左子树,直到访问到叶子结点。决定球运动方向的是每个结点的布尔值。最初,所有的结点都是 false,当访问到一个结点时,如果这个结点是 false,则这个球把它变成 true,然后从左子树走,继续它的旅程。如果节点是 true,则球也会改变它为 false,而接下来从右子树走。满二叉树的标记方法如图 3－19:

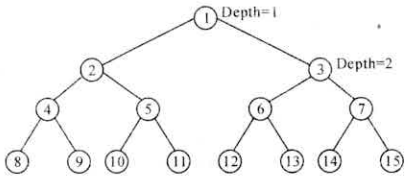


图 3－19

因为所有的结点最初为 false,所以第一个球将会访问结点 1、结点 2 和结点 4,转变结点的布尔值后在结点 8 停止。第二个球将会访问结点 1、3、6,在结点 12 停止。明显地,第三个球在它停止之前,会访问结点 1、2、5,在结点 10 停止。

现在你的任务是,给定 FBT 的深度 D 和 I,I 表示第 I 个小球下落,你可以假定 I 不超过给定的 FBT 的叶子数,写一个程序求小球停止时的叶子序号。

【输入格式】

输入文件仅一行,包含两个用空格隔开的整数 D 和 I。

其中 2<=D<=20,1<=I<=524288。

【输出格式】

对应输出第 I 个小球下落停止时的叶子序号。

【输入样例】　　【输出样例】

4 2　　12

2. 二叉树遍历(flist)

【问题描述】

树和二叉树基本上都有先序、中序、后序、按层遍历等遍历顺序,给定中序和其他一种遍历的序列就可以确定一棵二叉树的结构。

假定一棵二叉树一个结点用一个字符描述,现在给出中序和按层遍历的字符串,求该树的先序遍历字符串。

【输入格式】

输入文件 flist.in 共两行,每行是由字母组成的字符串(一行的每个字符都是唯一的),分别表示二叉树的中序遍历和按层遍历的序列。

【输出格式】

输出文件 flist.out 就一行,表示二叉树的先序序列。

【输入样例】　　【输出样例】

DBEAC　　ABDEC

ABCDE

3. FBI 树(fbi)

【问题描述】

我们可以把由“0”和“1”组成的字符串分为三类：全“0”串称为 B 串，全“1”串称为 I 串，既含“0”又含“1”的串则称为 F 串。

FBI 树是一种二叉树，它的结点类型也包括 F 结点、B 结点和 I 结点三种。由一个长度为 2^N 的“01”串 S 可以构造出一棵 FBI 树 T，递归的构造方法如下：

T 的根结点为 R，其类型与串 S 的类型相同；

若串 S 的长度大于 1，将串 S 从中间分开，分为等长的左右子串 S1 和 S2；由左子串 S1 构造 R 的左子树 T1，由右子串 S2 构造 R 的右子树 T2。

现在给定一个长度为 2^N 的“01”串，请用上述构造方法构造出一棵 FBI 树，并输出它的后序遍历序列。

【输入格式】

输入文件 fbi.in 的第 1 行是一个整数 N($0 <= N <= 10$)，第 2 行是一个长度为 2^N 的“01”串。

【输出格式】

输出文件 fbi.out 包括一行，这一行只包含一个字符串，即 FBI 树的后序遍历序列。

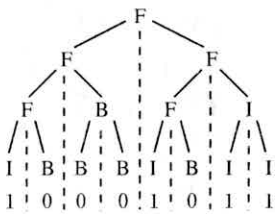


图 3－20

【输入样例】

3

10001011

【输出样例】

IBFBBBFIBFIIIFF

【数据范围】

40%的数据满足：$N <= 2$；

100%的数据满足：$N <= 10$。

4. 二叉树输出(btout)

【问题描述】

树的凹入表示法主要用于树的屏幕或打印输出，其表示的基本思想是兄弟间等长，一个结点的长度要不小于其子结点的长度。二叉树也可以这样表示，假设叶结点的长度为 1，一个非叶结点的长度等于它左右子树的长度之和。

一棵二叉树的一个结点用一个字母表示(无重复)，输出时从根结点开始：

每行输出若干个结点字符(相同字符的个数等于该结点长度)，

如果该结点有左子树就递归输出左子树；

如果该结点有右子树就递归输出右子树。

假定一棵二叉树一个结点用一个字符描述，现在给出先序和中序遍历的字符串，用树的凹入表示法输出该二叉树。

【输入格式】

输入文件 btout.in 共两行，每行是由字母组成的字符串(一行的每个字符都是唯一的)，分别表示二叉树的先序遍历和中序遍历的序列。

【输出格式】

输出文件 btout.out 的行数等于该树的结点数，每行的字母相同。

【输入样例】
ABCDEFG
CBDAFEG

【输出样例】
AAAA
BB
C
D
EE
F
G

5. 查找二叉树(tree_a)

【问题描述】

已知一棵二叉树用邻接表结构存储,中序查找二叉树中值为 x 的结点,并指出是第几个结点。

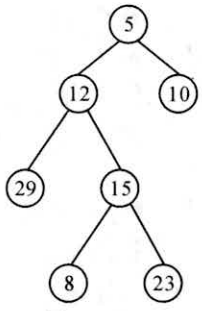


图 3－21

【输入格式】

第 1 行 n 为二叉树的结点个数,n<＝100;第 2 行 x 表示要查找的结点的值;以下第 1 列数据是各结点的值,第 2 列数据是左儿子结点编号,第 3 列数据是右儿子结点编号。

【输出格式】

输出一个数即查找的结点编号。

【输入样例】
7
15
5 2 3
12 4 5
10 0 0
29 0 0
15 6 7
8 0 0
23 0 0

【输出样例】
4

6. 对称二叉树(tree_c)

【问题描述】

如果二叉树的左右子树的结构是对称的,即两棵子树皆为空,或者皆不空,则称该二叉树是对称的。编程判断给定的二叉树是否对称。

例:如图 3－22 中的二叉树 T1 是对称的,T2 是不对称的。

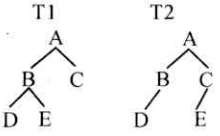


图 3－22

二叉树用顺序结构给出,若读到＃则为空,二叉树 T1＝ABCDE,T2＝ABCD＃E,如果二叉树是对称的,输出“Yes”,反之输出“No”。

【输入样例】
ABCDE

【输出样例】
Yes

第三节　堆及其应用

一、预备知识

完全二叉树：

如果一棵深度为 k 二叉树，1 至 k－1 层的结点都是满的，即满足 2^{i-1}，只有最下面的一层的结点数小于 2^{i-1}，并且最下面一层的结点都集中在该层最左边的若干位置，则此二叉树称为完全二叉树。

二、堆的定义

堆结构是一种数组对象，它可以被视为一棵完全二叉树。树中每个结点与数组中存放该结点中值的那个元素相对应，如图 3－23：

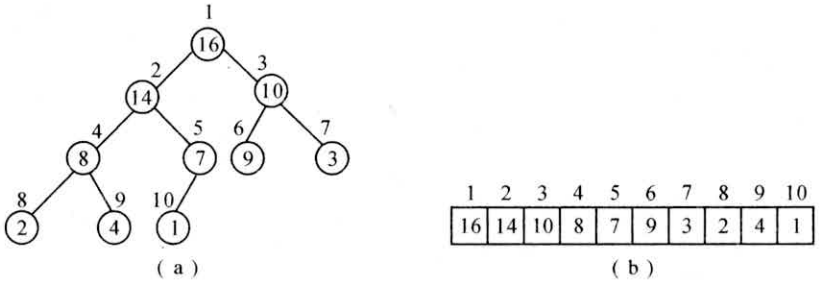


图 3－23

三、堆的性质

设数组 A 的长度为 len，二叉树的结点个数为 size，size<＝len，则 A[i]存储二叉树中编号为 i 的结点值(1<＝i<＝size)，而 A[size]以后的元素并不属于相应的堆，树的根为 A[1]，并且利用完全二叉树的性质，我们很容易求第 i 个结点的父结点(parent(i))、左孩子结点(left(i))、右孩子结点(right(i))的下标了，分别为：i/2、2i、2i＋1。

更重要的是，堆具有这样一个性质，对除根以外的每个结点 i，A[parent(i)]>＝A[i]。即除根结点以外，所有结点的值都不得超过其父结点的值，这样就推出，堆中的最大元素存放在根结点中，且每一结点的子树中的结点值都小于等于该结点的值，这种堆又称为“大根堆”；反之，对除根以外的每个结点 i，A[parent(i)]<＝A[i]的堆，称为“小根堆”。

四、堆的操作

用堆的关键部分是两个操作：put 操作，即往堆中加入一个元素；get 操作，即从堆中取出并删除一个元素。

1. 往堆中加入一个元素的算法(put)如下：

(1)在堆尾加入一个元素,并把这个结点置为当前结点。

(2)比较当前结点和它父结点的大小

如果当前结点小于父结点,则交换它们的值,并把父结点置为当前结点。转(2)。

如果当前结点大于等于父结点,则转(3)。

(3)结束。

重复 n 次 put 操作,即可建立一个小根堆。我们举一个例子看看具体过程:设 n=10,10 堆的数量分别为:3 5 1 7 6 4 2 5 4 1。

设一个堆结构 heap[11],现在先考虑用 put 操作建一个小根堆,具体方法是每次读入一个数插入到堆尾,再通过调整使得满足堆的性质(从堆尾 son=len 开始,判断它与父结点 son/2 的大小,若 heap[son]<heap[son/2],则交换这两个数,且 son=son/2,继续判断 heap[son]与 heap[son/2]的大小…直到 son=1 或者 heap[son]>=heap[son/2]为止)。开始时堆的长度 len=0。

第 1 步,读入一个数 x=3,len=1,heap[len]=x,一个数当然满足堆的性质,即:

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

第 2 步,再读入一个数 x=5,len=2,heap[len]=x,依然满足堆的性质,即:

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

第 3 步,再读入一个数 x=1,len=3,heap[len]=x,把 1 从 heap[3]调整到 heap[1],即:

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

第 4 步,再读入一个数 x=7,len=4,heap[len]=x,依然满足堆的性质,即:

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

第 5 步,再读入一个数 x=6,len=5,heap[len]=x,依然满足堆的性质,即:

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

第 6 步，再读入一个数 x=4，len=6，heap[len]=x，依然满足堆的性质，即：

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

第 7 步，再读一个数 x=2，len=7，heap[len]=x，把 2 从 heap[7]调整到 heap[3]，即：

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

第 8 步，再读一个数 x=5，len=8，heap[len]=x，把 5 从 heap[8]调整到 heap[4]，即：

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

第 9 步，再读入一个数 x=4，len=9，heap[len]=x，先把 4 从 heap[9]调整到 heap[4]，再把 4 从 heap[4]调整到 heap[2]，即：

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

当前有一个最小堆，如图 3－24

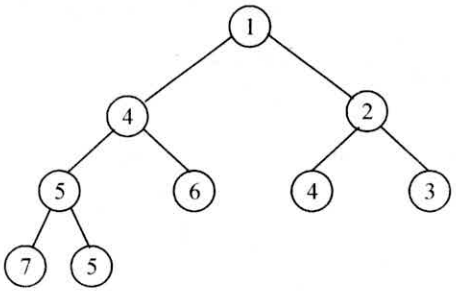


图 3－24

Heap	1	4	2	5	6	4	3	7	5

第 10 步，再读入一个数 x=1，len=10，heap[len]=x，先把 1 从 heap[10]调整到 heap[5]，再把 1 从heap[5]调整到 heap[2]，即：

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

现在，要往堆中加入一个元素 1，先将 1 加入到堆尾

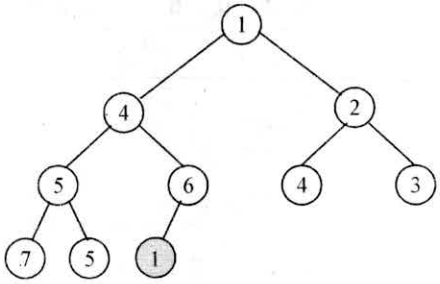


图 3－25

Heap	1	4	2	5	6	4	3	7	5	1
										son

比较当前结点 heap[son]＝1 和父结点 heap[pa]＝6 的大小，因为 heap[son]＜heap[pa]，交换它们的值，并且把父结点设置为当前结点。

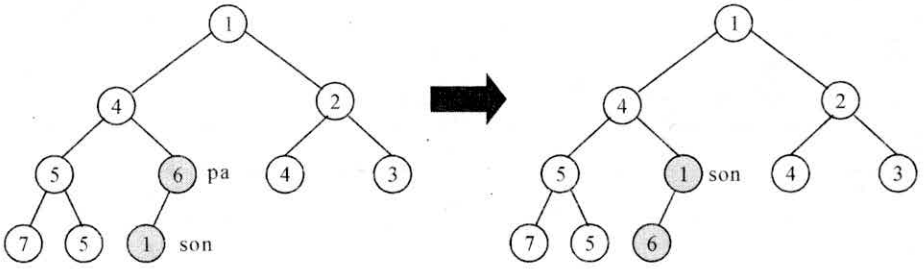


图 3－26

Heap	1	4	2	5	1	4	3	7	5	6
	1	2	3	4	5	6	7	8	9	10
					pa					son

比较当前结点 heap[son]＝1 和父结点 heap[pa]＝4 的大小，因为 heap[son]＜heap[pa]，交换它们的值，并且把父结点设置为当前结点。

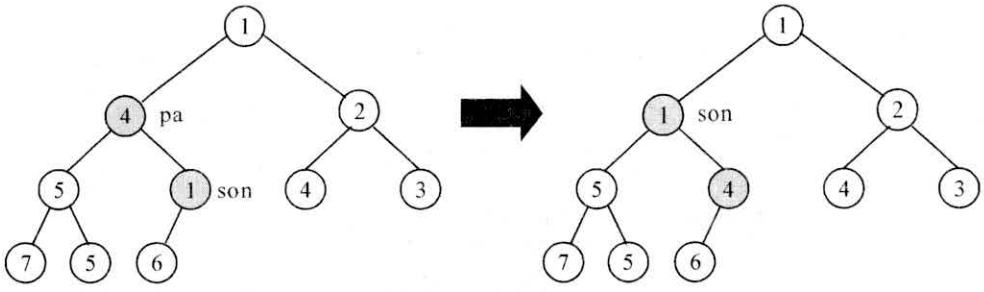


图 3－27

Heap	1	1	2	5	4	4	3	7	5	6
		pa			son					

比较当前结点 heap[son]＝1 和父结点 heap[pa]＝1 的大小，因为 heap[son]＝heap[pa]，则结束，所以插入一个元素 1 的操作结果是：

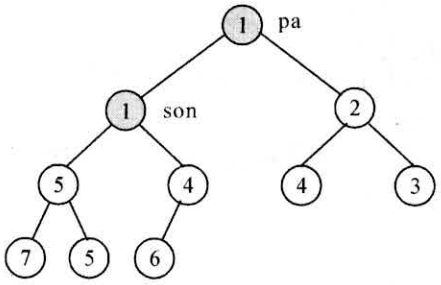


图 3－28

Heap	1	1	2	5	4	4	3	7	5	6

实际上，我们也可以直接用完全二叉树的形式描述出这个过程，得到如下的一棵完全二叉树(堆)：

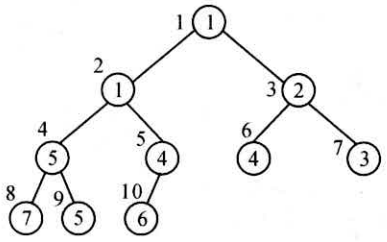


图 3－29

```
void put(int d)                          //heap[1]为堆顶
{
    int now, next;
    heap[++heap_size]=d;
    now=heap_size;
    while(now > 1)
    {
        next=now >> 1;
        if(heap[now] >=heap[next]) break;
        swap(heap[now], heap[next]);
        now=next;
    }
}
```

使用 C++标准模板库 STL(需要头文件 algorithm)：

```
void put(int d)
{
    heap[++heap_size]=d;
    //push_heap(heap+1, heap+heap_size+1);                    //大根堆
    push_heap(heap+1, heap+heap_size+1, greater<int>()); //小根堆
}
```

2.从堆中取出并删除一个元素的算法(get)如下:

(1)取出堆的根结点的值。

(2)把堆的最后一个结点(len)放到根的位置上,把根覆盖掉。把堆的长度减一。

(3)把根结点置为当前父结点 pa。

(4)如果 pa 无儿子(fa>len/2),则转(6);否则,把 pa 的两(或一)个儿子中值最小的那个置为当前的子结点 son。

(5)比较 pa 与 son 的值:

如果 pa 的值小于或等于 son,则转(6);否则,交换这两个结点的值,把 pa 指向 son,转(4)。

(6)结束。

举例:当前有一个最小堆,如图 3-30

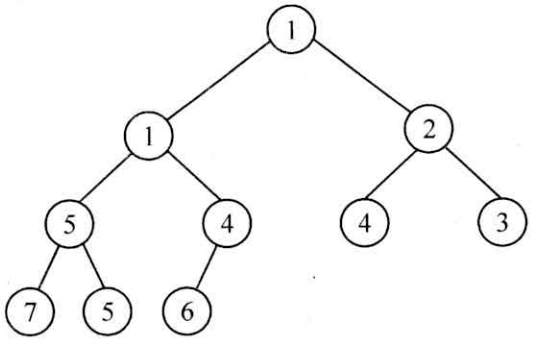


图 3-30

Heap	1	1	2	5	4	4	3	7	5	6

现在,要取出一个最小元素,即根结点,先将堆中的最后一个元素放到根结点上,并把根结点设置为当前结点 pa。

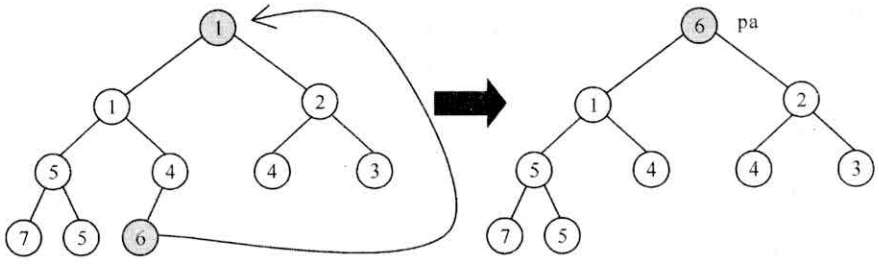


图 3-31

Heap	6	1	2	5	4	4	3	7	5

比较当前结点的两个儿子,取最小的作为当前结点的子结点 son,比较当前结点 heap[pa]=6 与其子结点 heap[son]=1,因为 heap[pa]>heap[son],则交换两个结点的值,并把子结点设置为当前结点 pa。

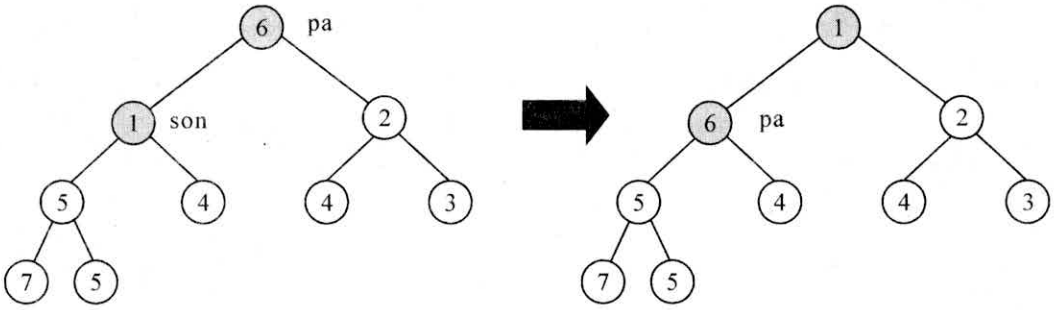


图 3－32

Heap	1	6	2	5	4	4	3	7	5

比较当前结点的两个儿子，取最小的作为当前结点的子结点 son，比较当前结点 heap[pa]=6 与其子结点 heap[son]=4，因为 heap[pa]>heap[son]，则交换两个结点的值，并把子结点设置为当前结点 pa。

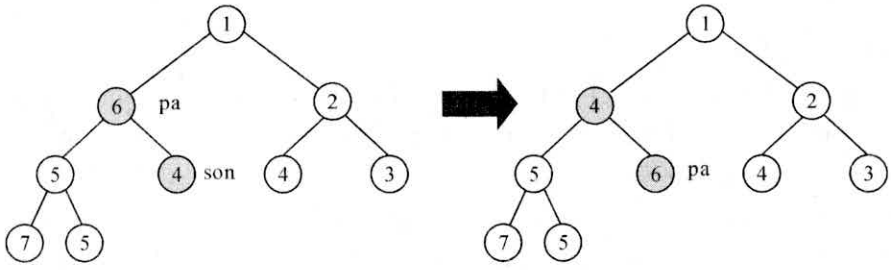


图 3－33

Heap	1	4	2	5	6	4	3	7	5

因为当前结点 heap[pa]没有儿子，所以结束，形成一个新小堆。

```
int get()                                    //heap[1]为堆顶
{
    int now, next, res;
    res=heap[1];
    heap[1]=heap[heap_size--];
    now=1;
    while(now * 2 <=heap_size)
    {
        next=now * 2;
        if (next < heap_size && heap[next+1] < heap[next]) next++;
        if (heap[now] <=heap[next]) break;
        swap(heap[now], heap[next]);
        now=next;
    }
    return res;
}
```

使用 C++标准模板库 STL(需要头文件 algorithm)：

```
int get()
{
    //pop_heap(heap+1, heap+heap_size+1);                    //大根堆
    pop_heap(heap+1, heap+heap_size+1, greater<int>());  //小根堆
    return heap[heap_size--];
}
```

五、堆的应用

例 3.6 合并果子(fruit)

【问题描述】

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

每一次合并,多多可以把两堆果子合并到一起,消耗的体力等于两堆果子的重量之和。可以看出,所有的果子经过 n-1 次合并之后,就只剩下一堆了。多多在合并果子时总共消耗的体力等于每次合并所耗体力之和。

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

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

【输入格式】

输入文件 fruit.in 包括两行,第一行是一个整数 n(1 <=n <=30000),表示果子的种类数。第二行包含 n 个整数,用空格分隔,第 i 个整数 a_i(1 <=a_i<=20000)是第 i 种果子的数目。

【输出格式】

输出文件 fruit.out 包括一行,这一行只包含一个整数,也就是最小的体力耗费值。输入数据保证这个值小于 2^{31}。

【输入样例 1】

```
3
1 2 9
```

【输出样例 1】

```
15
```

【输入样例 2】

```
10
3 5 1 7 6 4 2 5 4 1
```

【输出样例 2】

```
120
```

【数据范围】

30%的数据满足:n <=1000;

50%的数据满足:n <=5000;

100%的数据满足:n <=30000。

【问题分析】

1. 算法分析

将这个问题换一个角度描述:给定 n 个叶结点,每个结点有一个权值 w[i],将它们中两

个两个合并为树，假设每个结点从根到它的距离是 d[i]，使得最终∑(wi * di)最小。

于是，这个问题就变为了经典的 Huffman 树问题。Huffman 树的构造方法如下：

(1)从森林里取两个权和最小的子树；

(2)将它们的权和相加，得到新的子树，并且把原子树删除，将新子树插入到森林中；

(3)重复(1)～(2)，直到整个森林里只有一棵树。

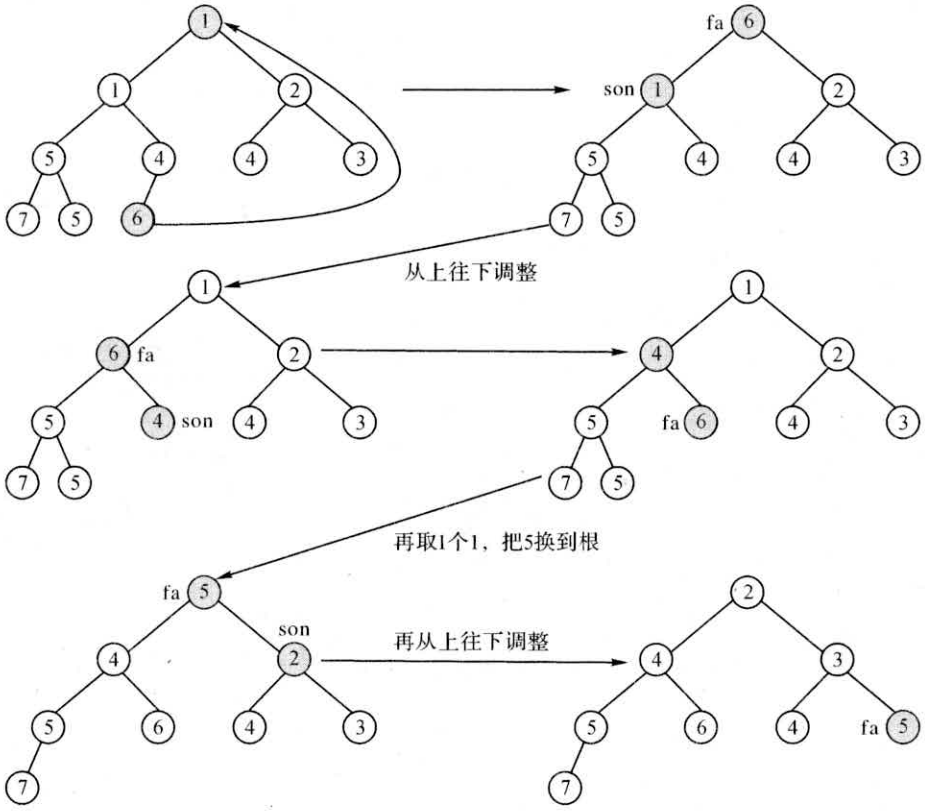
2. 数据结构

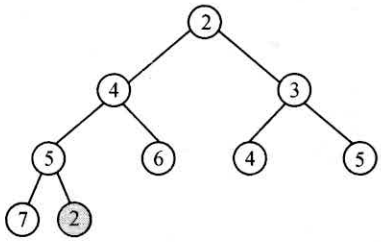
很显然，问题当中需要执行的操作是：(1) 从一个表中取出最小的数；(2) 插入一个数字到这个表中。支持动态查找最小数和动态插入操作的数据结构，我们可以选择用堆来实现。因为取的是最小元素，所以我们要用小根堆实现。

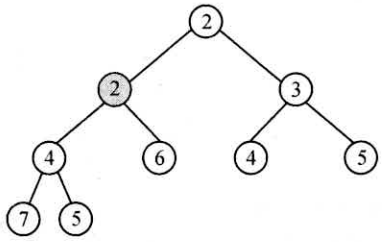
用堆的关键部分是两个操作：put 操作，即往堆中加入一个元素；get 操作，即从堆中取出并删除一个元素。

3. 操作实现

整个程序开始时通过 n 次 put 操作建立一个小根堆，然后不断重复如下操作：两次 get 操作取出两个最小数累加起来，并且形成一个新的结点，再插入到堆中。如 1＋1＝2，再把 2 插入到堆的后面一个位置，然后从下往上调整，使得包括 2 在内的数组满足堆的性质，即：







get 和 put 操作的复杂度均为 $\log_2 n$。所以建堆复杂度为 $n\log_2 n$。合并果子时，每次需要从堆中取出两个数，然后再加入一个数，因此一次合并的复杂度为 $3\log_2 n$，共 $n-1$ 次。所以整道题目的复杂度是 $n\log_2 n$。

【参考程序】

```
#include <iostream>
#include <cstdio>
using namespace std;
int heap_size, n;
int heap[30001];
void swap(int &a, int &b)      //加 & 后变量可修改
{
    int t=a;a=b;b=t;
}
void put(int d)
{
    int now, next;
    heap[++heap_size]=d;
    now=heap_size;
    while(now > 1)
    {
        next=now >> 1;
        if(heap[now] >=heap[next])return;
        swap(heap[now], heap[next]);
        now=next;
    }
}
int get()
{
    int now, next, res;
    res=heap[1];
    heap[1]=heap[heap_size--];
    now=1;
    while(now * 2 <=heap_size)
    {
        next=now * 2;
```

```
        if(next < heap_size && heap[next+1] < heap[next])next++;
        if(heap[now] <=heap[next])return res;
        swap(heap[now], heap[next]);
        now=next;
    }
return res;
}
void work()
{
    int i, x, y, ans=0;
    cin >> n;
    for(i=1 ; i <=n ; i++) //建堆,其实直接将数组排序也是建堆方法之一
    {
        cin >> x;
        put(x);
    }
    for(i=1 ; i < n ; i++)  //取、统计、插入
    {
        x=get();
        y=get();     //也可省去这一步,而直接将 x 累加到 heap[1]然后调整
        ans+=x+y;
        put(x+y);
    }
    cout << ans << endl;
}
int main()
{
    ios::sync_with_stdio(false);
  //优化。打消 iostream 的输入输出缓存,使得 cin cout 时间和 printf scanf 相差无几
    work();
    return 0;
}
```

使用 C++标准模板库 STL:

```
#include <iostream>
#include <queue>
#include <cstdio>
using namespace std;
int n;
priority_queue<int,vector<int>,greater<int> > h;     //优先队列
void work()
{
    int i, x, y, ans=0;
```

```
    cin >> n;
    for(i=1 ; i <=n ; i++) //建堆
    {
        cin >> x;
        h.push(x);
    }
    for(i=1 ; i < n ; i++)  //取、统计、插入
    {
        x=h.top();h.pop();
        y=h.top();h.pop();
        ans+=x+y;
        h.push(x+y);
    }
    cout << ans << endl;
}
int main()
{
    work();
    return 0;
}
```

例 3.7　堆排序(heapsort)

【问题描述】

假设 n 个数存放在 a[1..n]中,我们可以利用堆将它们从小到大进行排序,这种排序方法称为"堆排序"。输入两行,第 1 行为 n,第 2 行为 n 个整数,每个数之间用 1 个空格隔开。输出 1 行,为从小到大排好序的 n 个数,每个数之间也用 1 个空格隔开。

【问题分析】

一种思路是完全按照上一个例题的方法去做。

【参考程序 1】

```
#include <iostream>
#include <cstdio>
using namespace std;
int heap_size, n;
int heap[100001];
void swap(int &a, int &b)
{
    int t=a;a=b;b=t;
}
void put(int d)
{
    int now, next;
    heap[++heap_size]=d;
    now=heap_size;
```

```
    while(now > 1)
    {
        next=now >> 1;
        if(heap[now] >=heap[next])return;
        swap(heap[now], heap[next]);
        now=next;
    }
}
int get()
{
    int now, next, res;
    res=heap[1];
    heap[1]=heap[heap_size--];
    now=1;
    while(now * 2 <=heap_size)
    {
        next=now * 2;
        if(next < heap_size && heap[next+1] < heap[next])next++;
        if(heap[now] <=heap[next])return res;
        swap(heap[now], heap[next]);
        now=next;
    }
    return res;
}
void work()
{
    int i, x, y, ans=0;
    cin >> n;
    for(i=1 ; i <=n ; i++)
    {
        cin >> x;
        put(x);
    }
    for(i=1 ; i < n ; i++) cout << get() << '';
    cout << get() << endl;
}
int main()
{
    work();
    return 0;
}
```

另一种思路是考虑这样两个问题，一是如何构建一个初始(大根)堆？二是确定了最大

值后(堆顶元素 a[1]即为最大值),如何在剩下的 n-1 个数中,调整堆结构产生次大值?

对于第一个问题,我们可以这样理解,首先所有叶结点(编号为 n/2+1 到 n)都各自成堆,我们只要从最后一个分支结点(编号为 n/2)开始,不断"调整"每个分支结点与孩子结点的值,使它们满足堆的要求,直到根结点为止,这样一定能确保根(堆顶元素)的值最大。"调整"的思想如下:即如果当前结点编号为 i,则它的左孩子为 2*i,右孩子 2*i+1,首先比较 a[i]与 max(a[2*i],a[2*i+1]);如果 a[i]大,说明以结点 i 为根的子树已经是堆,不用再调整。否则将结点 i 和左右孩子中值大的那个结点 j 互换位置,互换后可能破坏以 j 为根的堆,所以必须再比较 a[j]与 max(a[2*j],a[2*j+1]),依此类推,直到父结点的值大于等于两个孩子或出现叶结点为止。这样,以 i 为根的子树就被调整成为一个堆。

编写的子程序如下:

```
void heap(int r[], int nn, int ii)
                    //一次操作,使 r 满足堆的性质,得到 1 个最大数在 r[ii]中
{
    int x, i=ii, j;
    x=r[ii];        //把待调整的结点值暂存起来
    j=2 * i;        //j 存放 i 的孩子中值大的结点编号,开始时为 i 的左孩子编号
    while(j <=nn)//不断调整,使以 i 为根的二叉树满足堆的性质
    {
        if(j < nn && r[j] < r[j+1]) j++;
                    //若 i 有右孩子且值比左孩子大,则把 j 设为右孩子的编号
        if(x < r[j]) //若父结点比孩子结点小,则调整父结点和孩子结点中值大的那
                       个结点,确保此处满足堆的性质
        {
            r[i]=r[j];
            i=j;
            j=2 * i;
        }
        else j=nn+1;            //故意让 j 超出范围,终止循环
    }
    r[i]=x;                     //调整到最终位置
}
```

经过第一步骤建立好一个初始堆后,可以确定堆顶元素值最大,我们就把它 a[1]与最后一个元素 a[n]交换,然后再对 a[1..n-1]进行调整,得到次大值与 a[n-1]交换,如此下去,所有元素便有序存放了。

主程序的框架如下:

【参考程序 2】

```
int a[100001];
int i,temp,n;
int main()
{
  输入 n 和 n 个元素;
  for(i=n / 2 ; i >=1 , i--)
```

```
    heap(a,n,i);                    //建立初始堆,且产生最大值 a[1]
  for(i=n ; i >=2 ; i--)//将当前最大值交换到最终位置上,再对前 i-1 个数调整
    {
      swap(a[1],a[i]);
      heap(a,i-1,1);
    }
  输出;
  return 0;
}
```

小结：

堆排序在数据较少时并不值得提倡，但数据量很大时，效率就会很高。因为其运算的时间主要消耗在建立初始堆和调整过程中，堆排序的时间复杂度为 $O(n\log_2 n)$，而且堆排序只需一个供交换用的辅助单元空间，是一种不稳定的排序方法。

例 3.8　鱼塘钓鱼(fishing)

【问题描述】

有 N 个鱼塘排成一排(N<100)，每个鱼塘中有一定数量的鱼，例如：N=5 时，如下表：

鱼塘编号	1	2	3	4	5
第 1 分钟能钓到的鱼的数量(1..1000)	10	14	20	16	9
每过 1 分钟钓鱼数的减少量(1..100)	2	4	6	5	3
当前鱼塘到下一个相邻鱼塘需要的时间(单位：分钟)	3	5	4	4	

即：在第 1 个鱼塘中钓鱼第 1 分钟内可钓到 10 条鱼，第 2 分钟内只能钓到 8 条鱼，…，第 5 分钟以后再也钓不到鱼了。从第 1 个鱼塘到第 2 个鱼塘需要 3 分钟，从第 2 个鱼塘到第 3 个鱼塘需要 5 分钟…

【编程任务】

给出一个截止时间 T(T<1000)，设计一个钓鱼方案，从第 1 个鱼塘出发，希望能钓到最多的鱼。

假设能钓到鱼的数量仅和已钓鱼的次数有关，且每次钓鱼的时间都是整数分钟。

【输入格式】

输入文件共 5 行，分别表示：

第 1 行为 N；

第 2 行为第 1 分钟各个鱼塘能钓到的鱼的数量，每个数据之间用一空格隔开；

第 3 行为每过 1 分钟各个鱼塘钓鱼数的减少量，每个数据之间用一空格隔开；

第 4 行为当前鱼塘到下一个相邻鱼塘需要的时间；

第 5 行为截止时间 T。

【输出格式】

输出文件仅一个整数(不超过 $2^{31}-1$)，表示你的方案能钓到的最多的鱼。

【输入样例】

```
5
10 14 20 16 9
2 4 6 5 3
3 5 4 4
```

【输出样例】

```
76
```

14

【知识准备】

最优化原理、贪心法、动态规划、用堆结构实现贪心。

【问题分析】

算法一：

我们可以这样想：如果知道了取到最大值的情况下，人最后在第i个鱼塘里钓鱼，那么用在路上的时间是固定的，因为我们不会先跑到第i个鱼塘里钓一分钟后再返回前面的鱼塘钓鱼，这样把时间浪费在路上显然不划算，再说在你没到某个鱼塘里去钓鱼之前，这个塘里的鱼也不会跑掉(即数量不会减少)。所以这时我们是按照从左往右的顺序钓鱼的，也可以看成路上是不需要时间的，即人可以自由在1～i个鱼塘之间来回走，只要尽可能选取钓到的鱼多的地方就可以了，这就是我们的贪心思想。其实，这个贪心思想并不是模拟钓鱼的过程，只是统计出在各个鱼塘钓鱼的次数。程序实现时，只要分别枚举钓鱼的终点鱼塘(从鱼塘1到鱼塘n)，每次按照上述贪心思想确定在哪些鱼塘里钓鱼，经过n次后确定后最终得到的一定是最优方案。

算法二：

其实，这道题是考虑最优性问题的，所以我们也可以用动态规划来解决，假设用opt[t][n]来表示第t分钟时，人在第n个鱼塘里钓鱼，最多所能钓到的鱼数。则：

opt[t][n]=maxinum{opt[t−k][n−1]+s}；

穷举k，s为t−k+1到t之间，除去从第n−1的鱼塘走到第n个鱼塘的时间，在第n个鱼塘中可以钓到的鱼数。

算法三：

建立以fish为关键字的大根堆，包括能钓到鱼的数量和池塘的编号。然后借助枚举创造条件，实现复杂度为O(m * nlogn)的算法。

【参考程序】

```
#include <iostream>
#include <cstdio>
using namespace std;
struct Data
{
  int fish, lake;                          //堆中结点的信息
};
Data heap[101];
int t[101], f[101], d[101];
int Max, k, t1;
void maintain(int i)                       //维护堆
{
  Data a;
  int next;
  a=heap[i];
  next=i * 2;
  while(next <=k)
  {
```

```
    if(next < k && heap[next].fish < heap[next+1].fish)next++;
    if(a.fish < heap[next].fish)
    {
      heap[i]=heap[next];
      i=next;
      next *=2;
    }
    else break;
  }
  heap[i]=a;
}
void work()
{
  int i, j, m, n;
  cin >> n;
  for(i=1 ; i <=n ; i++) cin >> f[i];
  for(i=1 ; i <=n ; i++) cin >> d[i];
  for(i=1 ; i < n ; i++) cin >> t[i];
  cin >> m;
  for(k=1 ; k <=n ; k++)                    //枚举最远走到的池塘的编号
  {
    int Time=m-t1;                          //计算剩余时间
    int ans=0;
    for(i=1 ; i <=k ; i++)                  //收集能够钓鱼的池塘的资料
    {
      heap[i].fish=f[i];
      heap[i].lake=i;
    }
    for(i=1 ; i <=k / 2 ; i++) maintain(i);  //堆的初始化
    while(Time > 0 && heap[1].fish > 0)
    {
      ans+=heap[1].fish;                    //贪心选取鱼最多的池塘
      heap[1].fish-=d[heap[1].lake];//修改鱼的数量
      maintain(1);                          //堆维护
      Time--;                               //剩余时间变少
    }
    if(ans > Max) Max=ans;                  //刷新最优解
    t1+=t[k];                               //累计走路需要的时间
  }
  cout << Max << endl;
}
int main()
```

```
{
  freopen("fishing.in", "r", stdin);
  freopen("fishing.out", "w", stdout);
  work();
  return 0;
}
```

使用 STL 的版本：

```
#include <iostream>
#include <cstdio>
#include <queue>
#define fish first
#define lake second
using namespace std;
priority_queue <pair<int, int> > heap;
int t[101], f[101], d[101];
int ans, m, Max, n, k, t1, Time;
void work()
{
    int i, j;
    cin >> n;
    for(i=1 ; i <=n ; i++) cin >> f[i];
    for(i=1 ; i <=n ; i++) cin >> d[i];
    for(i=1 ; i < n ; i++) cin >> t[i];
    cin >> m;
    for(k=1 ; k <=n ; k++)          //枚举最远走到的池塘的编号
    {
        Time=m-t1;                  //计算剩余时间
        ans=0;
        for(i=1 ; i <=k ; i++)      //收集能够钓鱼的池塘的资料
          heap.push(make_pair(f[i], i));
        while(Time > 0 && heap.top().fish > 0)
        {
          pair<int, int> a=heap.top();
          heap.pop();
          ans+=a.fish;              //贪心选取鱼最多的池塘
          a.fish-=d[a.lake];        //修改鱼的数量
          heap.push(a);             //堆维护
          Time--;                   //剩余时间变少
        }
        if(ans > Max) Max=ans;      //刷新最优解
        t1+=t[k];                   //累计走路需要的时间
    }
```

```
    cout << Max << endl;
}
int main()
{
    work();
    return 0;
}
```

【上机练习】

1. 合并果子(fruit)

【问题描述】

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

每一次合并，多多可以把两堆果子合并到一起，消耗的体力等于两堆果子的重量之和。可以看出，所有的果子经过 n－1 次合并之后，就只剩下一堆了。多多在合并果子时总共消耗的体力等于每次合并所耗体力之和。

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

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

【输入格式】

输入文件 fruit.in 包括两行，第一行是一个整数 n(1 <＝n <＝30000)，表示果子的种类数。第二行包含 n 个整数，用空格分隔，第 i 个整数 a_i(1 <＝a_i <＝20000)是第 i 种果子的数目。

【输出格式】

输出文件 fruit.out 包括一行，这一行只包含一个整数，也就是最小的体力耗费值。输入数据保证这个值小于 2^{31}。

【输入样例 1】

```
3
1 2 9
```

【输出样例 1】

```
15
```

【输入样例 2】

```
10
3 5 1 7 6 4 2 5 4 1
```

【输出样例 2】

```
120
```

2. 最小函数值(minval)

【问题描述】

有 n 个函数，分别为 F1，F2，…，Fn。定义 Fi(x)＝Ai * x^2＋Bi * x＋Ci(x∈**N** *)。给定这些 Ai、Bi 和 Ci，请求出所有函数的所有函数值中最小的 m 个(如有重复的要输出多个)。

【输入格式】

第 1 行输入两个正整数 n 和 m。

以下 n 行每行三个正整数,其中第 i 行的三个数分别位 Ai、Bi 和 Ci。输入数据保证 Ai<=10,Bi<=100,Ci<=10000。

【输出格式】

输出将这 n 个函数所有可以生成的函数值排序后的前 m 个元素。

这 m 个数应该输出到一行,用空格隔开。

【输入样例】

3 10
4 5 3
3 4 5
1 7 1

【输出样例】

9 12 12 19 25 29 31 44 45 54

【数据范围】

n,m<=10 000

第四章　图论算法

第一节　基本概念

一、什么是图

很简单，点用边连起来就叫做图，严格意义上讲，图是一种数据结构，定义为：graph＝(V,E)。V 是一个非空有限集合，代表顶点(结点)，E 代表边的集合。

二、图的一些定义和概念

1. 有向图：图的边有方向，只能按箭头方向从一点到另一点。图 4－1(a)就是一个有向图。

2. 无向图：图的边没有方向，可以双向。图 4－1(b)就是一个无向图。

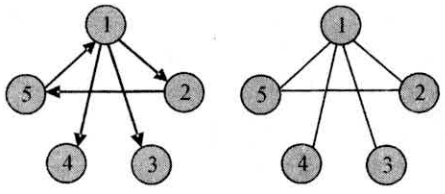


图 4－1

3. 结点的度：无向图中与结点相连的边的数目，称为结点的度。

4. 结点的入度：在有向图中，以这个结点为终点的有向边的数目。

5. 结点的出度：在有向图中，以这个结点为起点的有向边的数目。

6. 权值：边的“费用”，可以形象地理解为边的长度。

7. 连通：如果图中结点 U、V 之间存在一条从 U 通过若干条边、点到达 V 的通路，则称 U、V 是连通的。

8. 回路：起点和终点相同的路径，称为回路，或“环”。

9. 完全图：一个 n 阶的完全无向图含有 n＊(n－1)/2 条边；一个 n 阶的完全有向图含有 n＊(n－1)条边。

稠密图:一个边数接近完全图的图。
稀疏图:一个边数远远少于完全图的图。

10.强连通分量:有向图中任意两点都连通的最大子图。图4-2中,1-2-5构成一个强连通分量。特殊地,单个点也算一个强连通分量,所以图4-2有三个强连通分量:1-2-5,4,3。

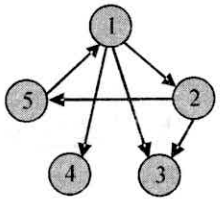


图4-2

三、图的存储结构

1.二维数组邻接矩阵存储

定义 int G[101][101];

G[i][j]的值,表示从点i到点j的边的权值,定义如下:

$$G[i][j]=\begin{cases}1\text{或权值} & \text{当 } v_i \text{ 与 } v_j \text{ 之间有边或弧时,取值为1或权值}\\ 0\text{或}\infty & \text{当 } v_i \text{ 与 } v_j \text{ 之间无边或弧时,取值为0或}\infty\text{(无穷大)}\end{cases}$$

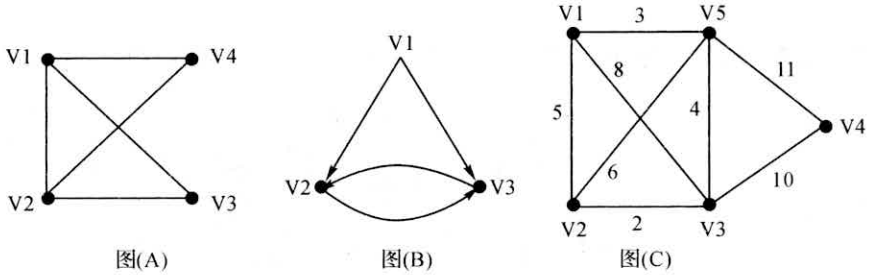


图4-3

上图中的3个图对应的邻接矩阵分别如下:

$$G(A)=\begin{bmatrix}0&1&1&1\\1&0&1&1\\1&1&0&0\\1&1&0&0\end{bmatrix}\quad G(B)=\begin{bmatrix}0&1&1\\0&0&1\\0&1&0\end{bmatrix}\quad G(C)=\begin{bmatrix}\infty&5&8&\infty&3\\5&\infty&2&\infty&6\\8&2&\infty&10&4\\\infty&\infty&10&\infty&11\\3&6&4&11&\infty\end{bmatrix}$$

下面是建立图的邻接矩阵的参考程序段:

```
#include<iostream>
using namespace std;
int i,j,k,e,n;
double g[101][101];
double w;
```

```
int main()
{
    int i,j;
    for (i=1; i <=n; i++)
      for (j=1; j <=n; j++)
        g[i][j]=0x7fffffff;
          //初始化,对于不带权的图 g[i][j]=0,表示没有边连通。这里用 0x7fffffff
            代替无穷大
    cin >> e;
    for (k=1; k <=e; k++)
    {
        cin >> i >> j >> w;   //读入两个顶点序号及权值
        g[i][j]=w;            //对于不带权的图 g[i][j]=1
        g[j][i]=w;            //无向图的对称性,如果是有向图则不要有这句
    }
    ...
    return 0;
}
```

建立邻接矩阵时,有两个小技巧:

初始化数组大可不必使用两重 for 循环。

(1)如果是 int 数组,采用 memset(g, 0x7f, sizeof(g))可全部初始化为一个很大的数(略小于 0x7fffffff);使用 memset(g, 0, sizeof(g)),全部清为 0;使用 memset(g, 0xaf, sizeof(g)),全部初始化为一个很小的数。

(2)如果是 double 数组,采用 memset(g,127,sizeof(g)),可全部初始化为一个很大的数 $1.38 * 10^{306}$;使用 memset(g, 0, sizeof(g))全部清为 0。

2. 数组模拟邻接表存储

图的邻接表存储法,又叫链式存储法。本来是要采用链表实现的,但大多数情况下只要用数组模拟即可。

以下是用数组模拟邻接表存储的参考程序段:

```
#include <iostream>
using namespace std;
const int maxn=1001,maxm=100001;
struct Edge
{
  int next;                    //下一条边的编号
  int to;                      //这条边到达的点
  int dis;                     //这条边的长度
}edge[maxm];

int head[maxn],num_edge,n,m,u,v,d;
```

```
void add_edge(int from,int to,int dis) //加入一条从 from 到 to 距离为 dis 的单向边
{
  edge[++num_edge].next=head[from];
  edge[num_edge].to=to;
  edge[num_edge].dis=dis;
  head[from]=num_edge;
}

int main()
{  num_edge=0;
   scanf("%d %d",&n,&m);                    //读入点数和边数
   for(int i=1;i<=m;i++)
   {
     scanf("%d %d %d",&u,&v,&d);            //u、v 之间有一条长度为 d 的边
     add_edge(u,v,d);
   }
   for(int i=head[1];i!=0;i=edge[i].next)   //遍历从点 1 开始的所有边
   {
       //...
   }
   //...
   return 0;
}
```

两种方法各有用武之地,需按具体情况,具体选用。

第二节　图的遍历

一、深度优先与广度优先遍历

从图中某一顶点出发系统地访问图中所有顶点，使每个顶点恰好被访问一次，这种运算操作被称为图的遍历。为了避免重复访问某个顶点，可以设一个标志数组 visited[i]，未访问时值为 false，访问一次后就改为 true。

图的遍历分为深度优先遍历和广度优先遍历两种方法，两者的时间效率都是 O(n * n)。

1. 深度优先遍历

深度优先遍历与深搜 dfs 相似，从一个点 A 出发，将这个点标为已访问 visited[i]=true;，然后再访问所有与之相连，且未被访问过的点。当 A 的所有邻接点都被访问过后，再退回到 A 的上一个点(假设是 B)，再从 B 的另一个未被访问的邻接点出发，继续遍历。

例如对右边的这个无向图深度优先遍历，假定先从 1 出发

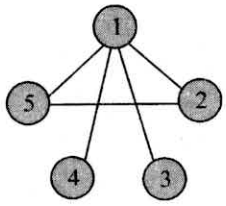


图 4－4

程序以如下顺序遍历：

1→2→5，然后退回到 2，退回到 1。

从 1 开始再访问未被访问过的点 3 ，3 没有未访问的邻接点，退回 1。

再从 1 开始访问未被访问过的点 4，再退回 1 。

起点 1 的所有邻接点都已访问，遍历结束。

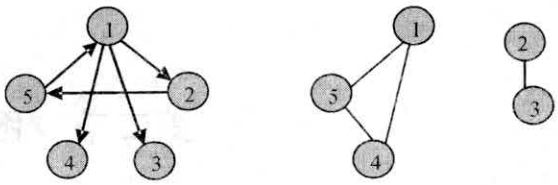
下面给出的深度优先遍历的参考程序，假设图以邻接表存储

```
void dfs(int i)                          //图用数组模拟邻接表存储，访问点 i
{
    visited[i]=true;                     //标记为已经访问过
    for (int j=1; j<=num[i]; j++)        //遍历与 i 相关联的所有未访问过的顶点
      if (!visited[g[i][j]])
        dfs(g[i][j]);
}
```

主程序如下：

```
int main()
{
  ...
  memset(visited,false,sizeof(visited));
  for (int i=1; i<=n; i++)   //每一个点都作为起点尝试访问，因为不是从任何
                             //一点开始都能遍历整个图的，例如下面的两个图
```

```
    if (!visited[i])
      dfs(i);
  ...
  return 0;
}
```



以3为起点根本不能遍历整个图　　这个非连通无向图任何一个点为起点都不能遍历整个图

图 4—5

2. 广度优先遍历

广度优先遍历并不常用，从编程复杂度的角度考虑，通常采用的是深度优先遍历。

广度优先遍历和广搜 bfs 相似，因此使用广度优先遍历一张图并不需要掌握什么新的知识，在原有的广度优先搜索的基础上，做一点小小的修改，就成了广度优先遍历算法。

二、一笔画问题

如果一个图存在一笔画，则一笔画的路径叫做欧拉路，如果最后又回到起点，那这个路径叫做欧拉回路。

我们定义奇点是指跟这个点相连的边数目有奇数个的点。对于能够一笔画的图，我们有以下两个定理。

定理 1：存在欧拉路的条件：图是连通的，有且只有 2 个奇点。

定理 2：存在欧拉回路的条件：图是连通的，有 0 个奇点。

两个定理的正确性是显而易见的，既然每条边都要经过一次，那么对于欧拉路，除了起点和终点外，每个点如果进入了一次，显然一定要出去一次，显然是偶点。对于欧拉回路，每个点进入和出去次数一定都是相等的，显然没有奇点。

求欧拉路的算法很简单，使用深度优先遍历即可。

根据一笔画的两个定理，如果寻找欧拉回路，对任意一个点执行深度优先遍历；找欧拉路，则对一个奇点执行 dfs，时间复杂度为 O(m+n)，m 为边数，n 是点数。

以下是寻找一个图的欧拉路的算法实现：

【输入格式】

第 1 行 n，m，有 n 个点，m 条边，以下 m 行描述每条边连接的两点。

【输出格式】

欧拉路或欧拉回路

【输入样例】

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

【输出样例】

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

【参考程序】

```
#include<iostream>
```

```
#include<cstring>
using namespace std;
#define maxn 101
int g[maxn][maxn];                    //此图用邻接矩阵存储
int du[maxn];                         //记录每个点的度,就是相连的边的数目
int circuit[maxn];                    //用来记录找到的欧拉路的路径
int n,e,circuitpos,i,j,x,y,start;
void find_circuit(int i)              //这个点深度优先遍历过程寻找欧拉路
{
  int j;
  for (j=1; j<=n; j++)
    if (g[i][j]==1)                   //从任意一个与它相连的点出发
    {
        g[i][j]=g[j][i]=0;            //删去这条边,避免下一次重复走过
        find_circuit(j);
    }
  circuit[++circuitpos]=i;            //记录下路径
}
int main()
{
    memset(g,0,sizeof(g));
    cin >> n >> e;
    for (i=1; i<=e; i++)
    {
      cin >> x >> y;
      g[x][y]=g[y][x]=1;              //说明 x 和 y 间有连边
      du[x]++;                        //统计每个点的度
      du[y]++;
    }
    start=1;                          //如果有奇点,就从奇点开始寻找,这样找到的就是
    for (i=1; i<=n; i++)              //欧拉路。没有奇点就从任意点开始,
        if (du[i]%2==1)               //这样找到的就是欧拉回路。(因为每一个点都是偶点)
            start=i;
    circuitpos=0;
    find_circuit(start);
    for (i=1; i<=circuitpos; i++)
        cout << circuit[i] << ' ';
    cout << endl;
    return 0;
}
```

注意以上程序具有一定的局限性,对于下面这种情况它不能很好地处理:

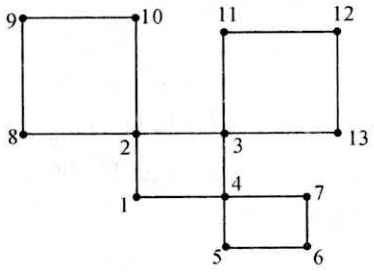


图 4-6

图 4-6 具有多个欧拉回路,而本程序只能找到一个回路。读者在遇到具体问题时,还应对程序作出相应的修改。

三、哈密尔顿环

欧拉回路是指不重复地走过所有路径的回路,而哈密尔顿环是指不重复地走过所有的点,并且最后还能回到起点的回路。

使用简单的深度优先搜索,就能求出一张图中所有的哈密尔顿环。下面给出一段参考程序:

```
#include<iostream>
#include<cstring>
using namespace std;
int start,length,x,n;
bool visited[101],vl[101];
int ans[101], num[101];
int g[101][101];
void print()
{
   int i;
   for (i=1; i<=length-1; i++)
     cout << ans[i] << ' ';
   cout << ans[length] << endl;
}
void dfs(int last,int i)   //图用数组模拟邻接表存储,访问点 i,last 表示上次访问的点
{
    visited[i]=true;              //标记为已经访问过
    vl[i]=true;                   //标记为已在一张图中出现过
    ans[++length]=i;              //记录下答案
    for (int j=1; j<=num[i]; j++)
    {
      if (g[i][j]==x&&g[i][j]!=last)   //回到起点,构成哈密尔顿环
```

```
            {
            ans[++length]=g[i][j];
            print();                    //这里说明找到了一个环,则输出 ans 数组。
            length--;
            break;
            }
        if (!visited[g[i][j]]) dfs(i,g[i][j]);
                                        //遍历与 i 相关联的所有未访问过的顶点
    }
    length--;
    visited[i]=false;                   //这里是回溯过程,注意 v1 的值不恢复。
}
int main()
{
    memset(visited,false,sizeof(visited));
    memset(v1,false,sizeof(v1));
    cin>>n;
    int m; cin>>m;
    for (int i=1; i<=m; i++)
    {
      int x,y;
      cin>>x>>y;
      g[x][++num[x]]=y;
      g[y][++num[y]]=x;
    }
    for (x=1; x <=n; x++)
      //每一个点都作为起点尝试访问,因为不是从任何一点开始都能找过整个图
        if (!v1[x])                     //如果点 x 不在之前曾经被访问过的图里
        {
            length=0;                   //定义一个 ans 数组存答案,length 记答案的长度
            dfs(0,x);
        }
    return 0;
}
```

【上机练习】

1. 珍珠(bead)

【问题描述】

有 n 颗形状和大小都一致的珍珠,它们的重量都不相同。n 为整数,所有的珍珠从 1 到 n 编号。你的任务是发现哪颗珍珠的重量刚好处于正中间,即在所有珍珠的重量中,该珍珠

的重量列(n+1)/2位。下面给出将一对珍珠进行比较的办法：

给你一架天平用来比较珍珠的重量，我们可以比出两个珍珠哪个更重一些，在作出一系列的比较后，我们可以将某些肯定不具备中间重量的珍珠拿走。

例如，下列给出对5颗珍珠进行四次比较的情况：

(1)珍珠2比珍珠1重

(2)珍珠4比珍珠3重

(3)珍珠5比珍珠1重

(4)珍珠4比珍珠2重

根据以上结果，虽然我们不能精确地找出哪个珍珠具有中间重量，但我们可以肯定珍珠1和珍珠4不可能具有中间重量，因为珍珠2、4、5比珍珠1重，而珍珠1、2、3比珍珠4轻，所以我们可以移走这两颗珍珠。

写一个程序统计出共有多少颗珍珠肯定不会是中间重量。

【输入格式】

输入文件第1行包含两个用空格隔开的整数N和M，其中1<=N<=99，且N为奇数，M表示对珍珠进行的比较次数，接下来的M行每行包含两个用空格隔开的整数x和y，表示珍珠x比珍珠y重。

【输出格式】

输出文件仅一行包含一个整数，表示不可能是中间重量的珍珠的总数。

【输入样例】

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

【输出样例】

```
2
```

2. 铲雪车(snow)

【问题描述】

随着白天越来越短夜晚越来越长，我们不得不考虑铲雪问题了。整个城市所有的道路都是双车道，因为城市预算的削减，整个城市只有1辆铲雪车。铲雪车只能把它开过的地方(车道)的雪铲干净，无论哪儿有雪，铲雪车都得从停放的地方出发，游历整个城市的街道。现在的问题是：最少要花多少时间去铲掉所有道路上的雪呢？

【输入格式】

输入数据的第1行表示铲雪车的停放坐标(x,y)，x,y为整数，单位为米。下面最多有100行，每行给出了一条街道的起点坐标和终点坐标，所有街道都是笔直的，且都是双向一个车道。铲雪车可以在任意交叉口或任何街道的末尾任意转向，包括转U型弯。铲雪车铲雪时前进速度为20千米/时，不铲雪时前进速度为50千米/时。

保证：铲雪车从起点一定可以到达任何街道。

【输出格式】

铲掉所有街道上的雪并且返回出发点的最短时间，精确到分钟。

【输入样例】

```
1
```

【输出样例】

```
3：55
```

0 0

0 0 10000 10000

5000－10000 5000 10000

5000 10000 10000 10000

【样例说明】

3 小时 55 分钟

3. 骑马修栅栏(fence)

【问题描述】

农民 John 每年有很多栅栏要修理。他总是骑着马穿过每一个栅栏并修复它破损的地方。

John 是一个与其他农民一样懒的人。他讨厌骑马，因此从来不两次经过一个一个栅栏。你必须编一个程序，读入栅栏网络的描述，并计算出一条修栅栏的路径，使每个栅栏都恰好被经过一次。John 能从任何一个顶点(即两个栅栏的交点)开始骑马，在任意一个顶点结束。

每一个栅栏连接两个顶点，顶点用 1 到 500 标号(虽然有的农场并没有 500 个顶点)。一个顶点上可连接任意多(>=1)个栅栏。所有栅栏都是连通的(也就是你可以从任意一个栅栏到达另外的所有栅栏)。

你的程序必须输出骑马的路径(用路上依次经过的顶点号码表示)。我们如果把输出的路径看成是一个 500 进制的数，那么当存在多组解的情况下，输出 500 进制表示法中最小的一个(也就是输出第一个数较小的，如果还有多组解，输出第二个数较小的…)。输入数据保证至少有一个解。

【输入格式】

第 1 行：一个整数 F(1 <=F <=1024)，表示栅栏的数目。

第 2 到 F+1 行：每行两个整数 i，j(1 <=i,j <=500)，表示这条栅栏连接 i 与 j 号顶点。

【输出格式】

输出应当有 F+1 行，每行一个整数，依次表示路径经过的顶点号。注意数据可能有多组解，但是只有上面题目要求的那一组解是认为正确的。

【输入样例】	【输出样例】
9	1
1 2	2
2 3	3
3 4	4
4 2	2
4 5	5
2 5	4
5 6	6
5 7	5
4 6	7

第三节　最短路径算法

如图4－7所示，我们把边带有权值的图称为带权图。边的权值可以理解为两点之间的距离。一张图中任意两点间会有不同的路径相连。最短路径就是指连接两点的这些路径中最短的一条。

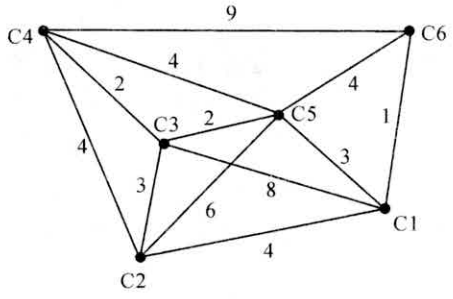


图4－7

我们有四种算法可以有效地解决最短路径问题。有一点需要读者特别注意：边的权值可以为负。当出现负边权时，有些算法不适用。

一、求出最短路径的长度

以下没有特别说明的话，dis[u][v]表示从u到v最短路径长度，w[u][v]表示连接u、v的边的长度。

1. Floyed－Warshall 算法 O(N^3)

简称Floyed(弗洛伊德)算法，是最简单的最短路径算法，可以计算图中任意两点间的最短路径。Floyed的时间复杂度是O(N^3)，适用于出现负边权的情况。

算法描述：

(a) 初始化：点u、v如果有边相连，则dis[u][v]=w[u][v]。

　　如果不相连，则dis[u][v]=0x7fffffff。

(b)
```
for (k=1; k <=n; k++)
    for (i=1; i <=n; i++)
        for (j=1; j <=n; j++)
            if (dis[i][j] >dis[i][k]+dis[k][j])
                dis[i][j]=dis[i][k]+dis[k][j];
```

(c) 算法结束：dis[i][j]得出的就是从i到j的最短路径。

算法分析 & 思想讲解：

三层循环，第一层循环中间点k，第二、第三层循环起点终点i、j，算法的思想很容易理解：如果点i到点k的距离加上点k到点j的距离小于原先点i到点j的距离，那么就用这个更短的路径长度来更新原先点i到点j的距离。

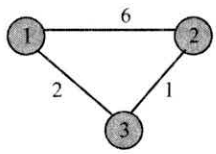


图 4－8

在图 4－8 中，因为 dis[1][3]＋dis[3][2]＜dis[1][2]，所以就用 dis[1][3]＋dis[3][2] 来更新原先 1 到 2 的距离。

我们在初始化时，把不相连的点之间的距离设为一个很大的数，不妨看作这两点相隔很远很远，如果两者之间有最短路径的话，就会更新成最短路径的长度。Floyed 算法的时间复杂度是 $O(N^3)$。

Floyed 算法变形：

如果是一个没有边权的图，把相连的两点间的距离设为 dis[i][j]＝true，不相连的两点设为 dis[i][j]＝false，用 Floyed 算法的变形：

```
for (k=1; k <=n; k++)
    for (i=1; i <=n; i++)
        for (j=1; j <=n; j++)
            dis[i][j]=dis[i][j] || (dis[i][k] && dis[k][j]);
```

用这个办法可以判断一张图中的两点是否相连。

最后再强调一点：用来循环中间点的变量 k 必须放在最外面一层循环。

例 4.1　最短路径问题

【问题描述】

平面上有 n 个点(n＜＝100)，每个点的坐标均在－10000～10000 之间。其中的一些点之间有连线。

若有连线，则表示可从一个点到达另一个点，即两点间有通路，通路的距离为两点间的直线距离。现在的任务是找出从一点到另一点之间的最短路径。

【输入格式】

输入文件为 short. in，共 n＋m＋3 行，其中：

第 1 行为整数 n。

第 2 行到第 n＋1 行(共 n 行)，每行两个整数 x 和 y，描述了一个点的坐标。

第 n＋2 行为一个整数 m，表示图中连线的个数。

此后的 m 行，每行描述一条连线，由两个整数 i 和 j 组成，表示第 i 个点和第 j 个点之间有连线。

最后一行：两个整数 s 和 t，分别表示源点和目标点。

【输出格式】

输出文件为 short. out，仅一行，一个实数(保留两位小数)，表示从 s 到 t 的最短路径长度。

【输入样例】

```
5
0 0
2 0
2 2
```

【输出样例】

```
3.41
```

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

【参考程序】

```
#include<cstdio>
#include<iostream>
#include<cmath>
#include<cstring>
using namespace std;
int a[101][3];
double f[101][101];
int n,i,j,k,x,y,m,s,e;
int main()
{
  freopen(" short. in "," r ",stdin);
  freopen(" short. out "," w ",stdout);
  cin >> n;
  for (i=1; i <=n; i++)
    cin >> a[i][1] >> a[i][2];
  cin >> m;
  memset(f,0x7f,sizeof(f));        //初始化 f 数组为最大值
  for (i=1; i <=m; i++)            //预处理出 x、y 间距离
  {
    cin >> x >> y;
    f[y][x]=f[x][y]=sqrt(pow(double(a[x][1]-a[y][1]),2)+pow(double
      (a[x][2]-a[y][2]),2));
                                   //pow(x,y)表示 x^y,其中 x,y 为 double,用 cmath 库
  }
  cin >> s >> e;
  for (k=1; k <=n; k++)            //floyed 最短路算法
   for (i=1; i <=n; i++)
    for (j=1; j <=n; j++)
      if ((i! =j) && (i! =k) && (j! =k) && (f[i][k]+f[k][j] < f[i][j]))
       f[i][j]=f[i][k]+f[k][j];
```

```
    printf("%.2lf\n",f[s][e]);
    return 0;
}
```

例 4.2　牛的旅行

【问题描述】

农民 John 的农场里有很多牧区。有的路径连接一些特定的牧区。一片所有连通的牧区称为一个牧场。但是就目前而言，你能看到至少有两个牧区不连通。现在，John 想在农场里添加一条路径（注意，恰好一条）。对这条路径有这样的限制：一个牧场的直径就是牧场中最远的两个牧区的距离（本题中所提到的所有距离指的都是最短的距离）。考虑如下的两个牧场，图 4－9 是有 5 个牧区的牧场，牧区用“＊”表示，路径用直线表示，每一个牧区都有自己的坐标：

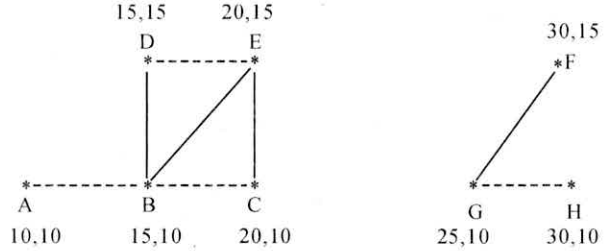


图 a　　图 b

图 4－9

图 4－9 所示的牧场的直径大约是 12.07106，最远的两个牧区是 A 和 E，它们之间的最短路径是 A－B－E。

这两个牧场都在 John 的农场上。John 将会在两个牧场中各选一个牧区，然后用一条路径连起来，使得连通后这个新的更大的牧场有最小的直径。注意，如果两条路径中途相交，我们不认为它们是连通的。只有两条路径在同一个牧区相交，我们才认为它们是连通的。

现在请你编程找出一条连接两个不同牧场的路径，使得连上这条路径后，这个更大的新牧场有最小的直径。

【输入格式】

第 1 行：一个整数 N（1 <＝N <＝150），表示牧区数；

第 2 到 N＋1 行：每行两个整数 X，Y（0 <＝X，Y <＝100000），表示 N 个牧区的坐标，每个牧区的坐标都是不一样的。

第 N＋2 行到第 2＊N＋1 行：每行包括 N 个数字（0 或 1）表示一个对称邻接矩阵。

例如，题目描述中的两个牧场的矩阵描述如下：

	A	B	C	D	E	F	G	H
A	0	1	0	0	0	0	0	0
B	1	0	1	1	1	0	0	0
C	0	1	0	0	1	0	0	0
D	0	1	0	0	1	0	0	0
E	0	1	1	1	0	0	0	0
F	0	0	0	0	0	0	1	0

G	0	0	0	0	0	1	0	1
H	0	0	0	0	0	0	1	0

输入数据中至少包括两个不连通的牧区。

【输出格式】

只有一行,包括一个实数,表示所求答案。数字保留六位小数。

【输入样例】

```
8
10 10
15 10
20 10
15 15
20 15
30 15
25 10
30 10
01000000
10111000
01001000
01001000
01110000
00000010
00000101
00000010
```

【输出样例】

```
22.071068
```

【算法分析】

用 Floyed 求出任两点间的最短路,然后求出每个点到所有可达的点的最大距离,记做 mdis[i]。(Floyed 算法)

r1=max(mdis[i])

然后枚举不连通的两点 i、j,把它们连通,则新的直径是 mdis[i]+mdis[j]+(i,j)间的距离。

r2=min(mdis[i]+mdis[j]+dis[i,j])

re=max(r1,r2)

re 就是所求。

【参考程序】

```
#include<iostream>
#include<cstdio>
#incdlue<cstring>
#include<cmath>
using namespace std;
double f[151][151],m[151],minx,r,temp,x[151],y[151],maxint=1e12;
double dist(int i,int j)
```

```
{
    return sqrt((x[i]-x[j])*(x[i]-x[j])+(y[i]-y[j])*(y[i]-y[j]));
}
int main()
{ int i,j,n,k;char c;
  cin>>n;
  for(i=1;i<=n;i++)cin>>x[i]>>y[i];
  for(i=1;i<=n;i++)
   for(j=1;j<=n;j++)
    { cin>>c;
      if(c=='1')f[i][j]=dist(i,j);
          else f[i][j]=maxint;
    }
  for(k=1;k<=n;k++)
    for(i=1;i<=n;i++)
      for(j=1;j<=n;j++)
        if(i!=j&&i!=k&&j!=k)
          if(f[i][k]<maxint-1&&f[k][j]<maxint-1)
            if(f[i][j]>f[i][k]+f[k][j])
              f[i][j]=f[i][k]+f[k][j];
  memset(m,0,sizeof(m));
  for(i=1;i<=n;i++)
   for(j=1;j<=n;j++)
    if(f[i][j]<maxint-1&&m[i]<f[i][j])m[i]=f[i][j];
  minx=1e20;
 for(i=1;i<=n;i++)
  for(j=1;j<=n;j++)
   if(i!=j&&f[i][j]>maxint-1)
    {temp=dist(i,j);
      if(minx>m[i]+m[j]+temp)minx=m[i]+m[j]+temp;
     }
  r=0;
  for(i=1;i<=n;i++)if (m[i]>minx)minx=m[i];
  printf("%.6lf ",minx);
  return 0;
}
```

2. Dijkstra 算法 O (N^2)

用来计算从一个点到其他所有点的最短路径的算法，是一种单源最短路径算法。也就是说，只能计算起点只有一个的情况。

Dijkstra 的时间复杂度是 O (N^2)，它不能处理存在负边权的情况。

算法描述：

设起点为 s，dis[v]表示从 s 到 v 的最短路径，pre[v]为 v 的前驱节点，用来输出路径。

(a)初始化：dis[v]=∞(v≠s)；dis[s]=0；pre[s]=0；

(b)for (i=1；i<=n；i++)

①在没有被访问过的点中找一个顶点 u 使得 dis[u]是最小的。

②u 标记为已确定最短路径。

③for 与 u 相连的每个未确定最短路径的顶点 v。

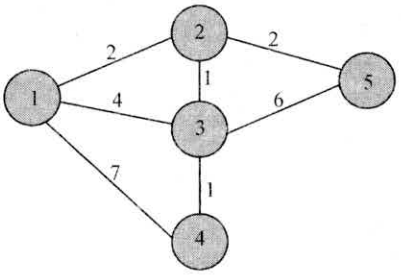
```
if  (dis[u]+w[u][v] < dis[v])
  {
    dis[v]=dis[u]+w[u][v];
    pre[v]=u;
  }
```

(c)算法结束：dis[v]为 s 到 v 的最短距离；pre[v]为 v 的前驱节点，用来输出路径。

算法分析 & 思想讲解：

从起点到一个点的最短路径一定会经过至少一个“中转点”(例如图 4－10 中 1 到 5 的最短路径，中转点是 2。特殊地，我们认为起点 1 也是一个“中转点”)。显而易见，如果我们想求出起点到一个点的最短路径，那我们必然要先求出中转点的最短路径(例如我们必须先求出点 2 的最短路径后，才能求出从起点到 5 的最短路径)。

换句话说，如果起点 1 到某一点 v0 的最短路径要经过中转点 vi，那么中转点 vi 一定是先于 v0 被确定了最短路径的点。



中转点	终点	最短路
1	1	0
1	2	2
1、2	3	3
1、2、3	4	4
1、2	5	4

求解顺序

图 4－10

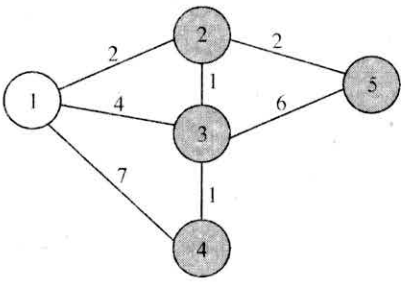
我们把点分为两类，一类是已确定最短路径的点，称为“白点”，另一类是未确定最短路径的点，称为“蓝点”。如果我们要求出一个点的最短路径，就是把这个点由蓝点变为白点。从起点到蓝点的最短路径上的中转点在这个时刻只能是白点。

Dijkstra 的算法思想，就是一开始将起点到起点的距离标记为 0，而后进行 n 次循环，每次找出一个到起点距离 dis[u]最短的点 u，将它从蓝点变为白点。随后枚举所有的蓝点 vi，如果以此白点为中转到达蓝点 vi 的路径 dis[u]+w[u][vi]更短的话，这将它作为 vi 的“更短路径”dis[vi](此时还不确定是不是 vi 的最短路径)。

就这样，我们每找到一个白点，就尝试着用它修改其他所有的蓝点。中转点先于终点变成白点，故每一个终点一定能够被它的最后一个中转点所修改，而求得最短路径。

让我们对以上这段枯燥的文字做一番模拟，加深理解。

算法开始时，作为起点的 dis[1]=0，其他的点 dis[i]=0x7fffffff。

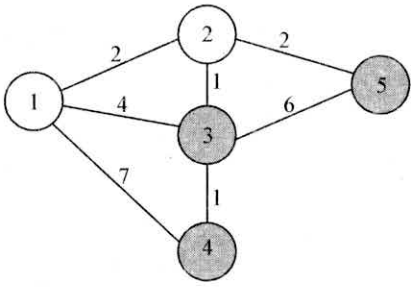


第一轮循环找到dis[1]最小，将1变成白点。

对所有的蓝点做出修改，使得dis[2]=2,dis[3]=4,dis[4]=7。

图 4－11

这时 dis[2]、dis[3]、dis[4]被它的最后一个中转点 1 修改为了最短路径。

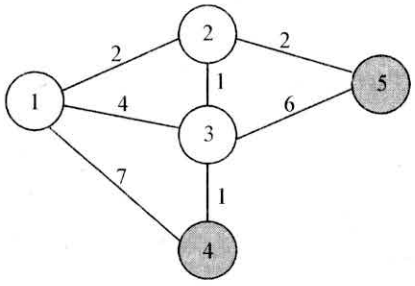


第二轮循环找到dis[2]最小，将2变成白点。

对所有的蓝点做出修改，使得dis[3]=3,dis[5]=4。

图 4－12

这时 dis[3]、dis[5]被它们的最后一个中转点 2 修改为了最短路径。



第三轮循环找到dis[3]最小，将3变成白点。

对所有的蓝点做出修改，使得dis[4]=4。发现以3为中转不能修改5，说明3不是5的最后一个中转点。

图 4－13

这时 dis[4]也被它的最后一个中转点 3 修改为了最短路径。

接下来的两轮循环将 4、5 也变成白点。N 轮循环结束后，所有的点的最短路径即能求出。

Dijkstra 无法处理边权为负的情况，例如图 4－14 这个例子。

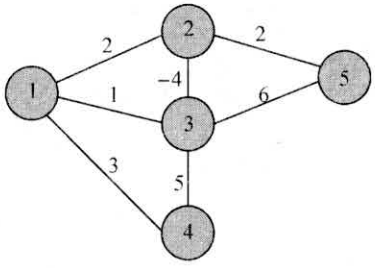


图 4－14

2 到 3 的边权值为－4，显然从起点 1 到 3 的最短路径是－2(1→2→3)，但是 dijskstra 在第二轮循环开始时会找到当前 dis[i]最小的点 3，并标记它为白点。

这时的 dis[3]＝1，然而 1 却不是从起点到点 3 的最短路径。因为 3 已被标记为白点，最短路径值 dis[3]不会再被修改了，所以我们在边权存在负数的情况下得到了错误的答案！

例 4.3　最短路径问题(dijkstra)

题目参见“Floyed 算法”，但本题要求使用 dijkstra 算法解决。

```
#include<iostream>
```

```
#include<cstdio>
#include<cstring>
#include<cmath>
using namespace std;
int a[101][3];
double c[101];
bool b[101];
double f[101][101];
int n,i,j,k,x,y,m,s,e;
double minl;
double maxx=1e30;
int main()
{
    cin >> n;
    for (i=1; i <=n; i++)
      cin >> a[i][1] >> a[i][2];
    for (i=1; i <=n; i++)
      for(j=1; j <=n; j++)
        f[i][j]=maxx;          //f 数组初始化最大值
    cin >> m;
    for (i=1; i <=m; i++)   //预处理 x、y 间距离 f[x][y]
    {
      cin >> x >> y;
      f[x][y]=f[y][x]=sqrt(pow(double(a[x][1]-a[y][1]),2)+
          pow(double(a[x][2]-a[y][2]),2));
    }
    cin >> s >> e;
    for (i=1; i <=n; i++)   c[i]=f[s][i];
    memset(b,false,sizeof(b)); //dijkstra 最短路
    b[s]=true;
    c[s]=0;
    for (i=1; i <=n-1; i++)
    {
      minl=maxx;
      k=0;
      for (j=1; j <=n; j++) //查找可以更新的点
        if ((!b[j]) && (c[j] < minl))
          {
              minl=c[j];
              k=j;
```

```
        }
      if (k==0) break;
      b[k]=true;
      for (j=1; j<=n; j++)
        if (c[k]+f[k][j] < c[j])
          c[j]=c[k]+f[k][j];
    }
    printf("%.2lf\n",c[e]);
    return 0;
}
```

例 4.4　最小花费

【问题描述】

在 n 个人中，某些人的银行账号之间可以互相转账。这些人之间转账的手续费各不相同。给定这些人之间转账时需要从转账金额里扣除百分之几的手续费，请问 A 最少需要多少钱使得转账后 B 收到 100 元？

【输入格式】

第 1 行输入两个正整数 n、m，分别表示总人数和可以互相转账的人的对数。

以下 m 行每行输入三个正整数 x、y、z，表示标号为 x 的人和标号为 y 的人之间互相转账需要扣除 z%的手续费（z<100）。

最后一行输入两个正整数 A、B。数据保证 A 与 B 之间可以直接或间接地转账。

【输出格式】

输出 A 使得 B 到账 100 元最少需要的总费用。精确到小数点后 8 位。

【输入样例】

```
3 3
1 2 1
2 3 2
1 3 3
1 3
```

【输出样例】

```
103.07153164
```

【数据范围】

1<=n<=2000

【参考程序】

```
#include<iostream>
#include<cstdio>
using namespace std;
double a[2001][2001],dis[2001]={0},minn;
int n,m,i,j,k,x,y,f[2001]={0};
void init()
{
  cin>>n>>m;
  for(i=1;i<=m;i++)
```

```
    {
      scanf("%d%d",&j,&k);
      scanf("%lf",&a[j][k]);
      a[j][k]=(100-a[j][k])/100;
      a[k][j]=a[j][k];
    }
    cin>>x>>y;
}
void dijkstra(int x)
{
    for(i=1;i<=n;i++)dis[i]=a[x][i];
    dis[x]=1;f[x]=1;
    for(i=1;i<=n-1;i++)
    { minn=0;
      for(j=1;j<=n;j++)
        if(f[j]==0&&dis[j]>minn){k=j;minn=dis[j];}
      f[k]=1;
      if(k==y)break;
      for(j=1;j<=n;j++)
       if(f[j]==0&&dis[k]*a[k][j]>dis[j])dis[j]=dis[k]*a[k][j];
    }
}
int main()
{
    init();
    dijkstra(x);
    printf("%0.81f",100/dis[y]);
    return 0;
}
```

3. Bellman-Ford 算法 O(NE)

简称 Ford(福特)算法,同样是用来计算从一个点到其他所有点的最短路径的算法,也是一种单源最短路径算法。

能够处理存在负边权的情况,但无法处理存在负权回路的情况(下文会有详细说明)。

算法时间复杂度:O(NE),N 是顶点数,E 是边数。

算法实现:

设 s 为起点,dis[v]即为 s 到 v 的最短距离,pre[v]为 v 前驱,w[j]是边 j 的长度,且 j 连接 u、v。

初始化:dis[s]=0,dis[v]=∞(v≠s),pre[s]=0

```
for (i=1; i <=n-1; i++)
  for (j=1; j <=E; j++)       //注意要枚举所有边,不能枚举点
```

```
if (dis[u]+w[j]<dis[v])    //u、v 分别是这条边连接的两个点
  {
    dis[v]=dis[u]+w[j];
    pre[v]=u;
  }
```

算法分析 & 思想讲解：

Bellman-Ford 算法的思想很简单。一开始认为起点是白点(dis[1]=0)，每一次都枚举所有的边，必然会有一些边，连接着白点和蓝点。因此每次都能用所有的白点去修改所有的蓝点，每次循环也必然会有至少一个蓝点变成白点。

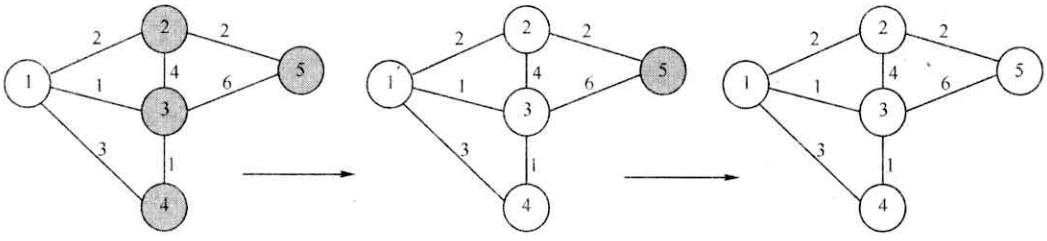


图 4－15

在上面这个简单的模拟中能看到白点的“蔓延”情况。

负权回路：

虽然 Bellman－Ford 算法可以求出存在负边权情况下的最短路径，却无法解决存在负权回路的情况。

负权回路是指边权之和为负数的一条回路，图 4－16 中②－④－⑤－③－②这条回路的边权之和为－3。在有负权回路的情况下，从 1 到 6 的最短路径是多少？答案是无穷小，因为我们可以绕这条负权回路走无数圈，每走一圈路径值就减去 3，最终达到无穷小。

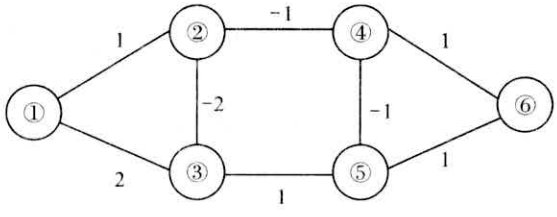


图 4－16

所以说存在负权回路的图无法求出最短路径，Bellman－Ford 算法可以在有负权回路的情况下输出错误提示。

若在 Bellman－Ford 算法的两重循环完成后，还存在某条边使得：dis[u]+w<dis[v]，则存在负权回路：

```
for 每条边(u,v)
    if  (dis[u]+w<dis[v])  return false
```

如果我们规定每条边只能走一次，在这个前提下可以求出负权回路的最短路径。这个问题就留待读者自己思考(提示：对 Floyed 做一点小处理)。

例 4.5 最短路径问题(Bellman-Ford)

题目参见"Floyed 算法",要求采用 Bellman-Ford 算法。

```
#include<iostream>
#include<cstdio>
#include<cmath>
using namespace std;
int main()
{
   double a[101][3],dis[1001],w[1001],min1;
   int n,m,x,y,k,f[1001][3],s,t;
   bool b[101];
   cin>>n;
   for (int i=1;i<=n;i++)
   scanf("%lf%lf",&a[i][1],&a[i][2]);
   cin>>m;
   for (int i=1;i<=m;i++)   //初始化数组 dis 和 f
   {
     dis[i]=0x7fffffff/3;
     f[i][1]=f[i][2]=0x7fffffff/3;
   }
   for (int i=1;i<=m;i++)
   {
     scanf("%d%d",&x,&y);
     f[i][1]=x; f[i][2]=y;
     w[i]=sqrt(pow(a[x][1]-a[y][1],2)+pow(a[x][2]-a[y][2],2));
   }
   cin>>s>>t;
   dis[s]=0;
   for (int i=1;i<=n;i++)   //算法主体
    for (int j=1;j<=m;j++)
    {
      if (dis[f[j][1]]+w[j]<dis[f[j][2]]) dis[f[j][2]]=dis[f[j][1]]+w[j];
      if (dis[f[j][2]]+w[j]<dis[f[j][1]]) dis[f[j][1]]=dis[f[j][2]]+w[j];
    }
   printf("%.2f",dis[t]);
}
```

4. SPFA 算法 O(kE)

SPFA 是 Bellman-Ford 算法的一种队列实现,减少了不必要的冗余计算。

主要思想是:

初始时将起点加入队列。每次从队列中取出一个元素,并对所有与它相邻的点进行修

改，若某个相邻的点修改成功，则将其入队。直到队列为空时算法结束。

这个算法简单的说就是队列优化的 bellman－ford，利用了每个点不会更新次数太多的特点发明的此算法。

SPFA 在形式上和广度优先搜索非常类似，不同的是广度优先搜索中一个点出了队列就不可能重新进入队列，但是 SPFA 中一个点可能在出队列之后再次被放入队列，也就是说一个点修改过其他的点之后，过了一段时间可能会获得更短的路径，于是再次用来修改其他的点，这样反复进行下去。

算法时间复杂度：O(kE)，E 是边数，K 是常数，平均值为 2。

算法实现：

dis[i]记录从起点 s 到 i 的最短路径，w[i][j]记录连接 i、j 的边的长度，pre[v]记录前趋。

team[1..n]为队列，头指针 head，尾指针 tail。

布尔数组 exist[1..n]记录一个点是否现在存在在队列中。

初始化：dis[s]=0，dis[v]=∞(v≠s)，memset(exist，false，sizeof(exist))；

起点入队 team[1]=s；head=0；tail=1；exist[s]=true；

```
do
  { 1. 头指针向下移一位，取出指向的点 u。
    2. exist[u]=false；已被取出了队列。
    3. for 与 u 相连的所有点 v    //注意不要去枚举所有点，用数组模拟邻接表存储
       if (dis[v]>dis[u]+w[u][v])
         {
            dis[v]=dis[u]+w[u][v];
            pre[v]=u;
            if (!exist[v]) //队列中不存在 v 点，v 入队
              {
                 尾指针下移一位，v 入队；
                 exist[v]=true;
              }
         }
  }
while (head < tail);
```

循环队列：

采用循环队列能够降低队列大小，队列长度只需开到 2 * n+5 即可。例题中的参考程序使用了循环队列。

例 4.6　香甜的黄油(butter)

【问题描述】

农夫 John 发现做出全威斯康辛州最甜的黄油的方法：糖。把糖放在一片牧场上，他知道 N(1<=N<=500)头奶牛会过来舔它，这样就能做出能卖好价钱的超甜黄油。当然，他将付出额外的费用在奶牛上。

农夫 John 很聪明。像以前的巴甫洛夫，他知道他可以训练这些奶牛，让它们在听到铃

声时去一个特定的牧场。他打算将糖放在那里然后下午发出铃声,以至他可以在晚上挤奶。

农夫 John 知道每头奶牛都在各自喜欢的牧场(一个牧场不一定只有一头牛)。给出各头牛在的牧场和牧场间的路线,找出使所有牛到达的路程和最短的牧场(他将把糖放在那)。

【输入格式】

第 1 行:三个数:奶牛数 N,牧场数 P(2<=P<=800),牧场间道路数 C(1<=C<=1450)。

第 2 行到第 N+1 行:1 到 N 头奶牛所在的牧场号。

第 N+2 行到第 N+C+1 行:每行有三个数:相连的牧场 A、B,两牧场间距(1<=D<=255),当然,连接是双向的。

【输出格式】

一行:输出奶牛必须行走的最小的距离和。

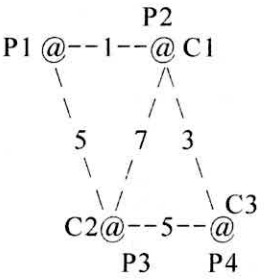
【输入样例】

```
3 4 5
2
3
4
1 2 1
1 3 5
2 3 7
2 4 3
3 4 5
```

【输出样例】

```
8  //说明:放在 4 号牧场最优
```

样例图形



【参考程序】

```
#include<iostream>
#include<cstdio>
#include<cstring>
using namespace std;
int n,p,c,i,j,x,y,t,min1,head,tail,tot,u;
int a[801][801],b[501],dis[801],num[801],w[801][801],team[1601];
bool exist[801];
int main()
{
    cin>>n>>p>>c;
```

```
for(i=1;i<=p;i++)
{
  b[i]=0;
  num[i]=0;
  for(j=1;j<=p;j++)
    w[i][j]=0x7fffffff/3;
}
for(i=1;i<=n;i++)cin>>b[i];
for(i=1;i<=c;i++)                    //邻接矩阵存储
{
  cin>>x>>y>>t;
  w[x][y]=w[y][x]=t;
  a[x][++num[x]]=y;
  a[y][++num[y]]=x;
}
min1=0x7fffffff/3;
for(i=1;i<=p;i++)
{
  for(j=1;j<=p;j++)dis[j]=0x7fffffff/3;
  memset(team,0,sizeof(team));       //队列数组初始化
  memset(exist,false,sizeof(exist)); //exist 标志初始化
  dis[i]=0;team[1]=i;head=0;tail=1;exist[i]=true; //起始点入队
  do
  {
    head++;
    head=((head-1)%1601)+1;    //循环队列处理
    u=team[head];
    exist[u]=false;
    for(j=1;j<=num[u];j++)
      if (dis[a[u][j]]>dis[u]+w[u][a[u][j]])
      {
        dis[a[u][j]]=dis[u]+w[u][a[u][j]];
        if (!exist[a[u][j]])
        {
          tail++;
          tail=((tail-1)%1601)+1;
          team[tail]=a[u][j];
          exist[a[u][j]]=true;
        }
      }
```

```
        }while(head!=tail);
        tot=0;
        for(j=1;j<=n;j++)
        tot+=dis[b[j]];
        if (tot<minl) minl=tot;
      }
  cout<<minl;
  return 0;
}
```

二、输出最短路径

1. 单源最短路径的输出

Dijkstra、Bellman-Ford、SPFA 都是单源最短路径算法，它们的共同点是都有一个数组 pre[x] 用来记录从起点到 x 的最短路径中，x 的前驱结点是哪个。每次更新，就给 pre[x]赋一个新值，结合上面的思想讲解，相信对于记录某点的前驱结点是不难理解的。那么怎么利用 pre[x]数组输出最短路径方案呢？

例 4.7 最短路径问题(输出路径)

要求改写程序，用 Dijkstra、Bellman-Ford 或 SPFA 算法输出最短路径的方案。

使用一个小小的递归过程就能解决这一问题。

```
void print(int x)
{
  if (pre[a][x]==0) return;                    //起点的前驱我们已设为 0
  print(pre[a][x]);
  cout << "->" << x;
}
//主程序中：
int main()
{
  …(进行 Dijkstra、Bellman-Ford 或 SPFA 运算)
  cout << s;  print(e);                        //s 是起点，e 是终点
}
```

2. Floyed 算法输出最短路径

Floyed 算法输出路径也是采用记录前驱点的方式。因为 floyed 是计算任意两点间最短路径的算法，dis[i][j]记录从 i 到 j 的最短路径值。故我们定义 pre[i][j]为一个二维数组，记录从 i 到 j 的最短路径中，j 的前驱点是哪一个。

例 4.8 最短路径问题(Floyed 法输出路径)

要求改写 Floyed 的程序，模仿 Dijkstra 输出路径的方法用 floyed 输出最短路径方案。

【参考程序】

```
#include<iostream>
```

```
#include<cmath>
#include<cstring>
using namespace std;
double dis[101][101];
int x[101],y[101];
int pre[101][101];
int n,i,j,k,m,a,b;
int pf(int x)
{
  return x * x;
}
void print(int x)
{
  if (pre[a][x]==0) return;              //pre[a][a]=0,说明已经递归到起点 a
  print(pre[a][x]);
  cout << "->" << x;
}
int main()
{
  cin >> n;
  for (i=1; i <=n; i++)
   cin >> x[i] >> y[i];
  memset(dis,0x7f,sizeof(dis));          //初始化数组
  cin >> m;
  memset(pre,0,sizeof(pre));             //初始化前驱数组
  for (i=1; i <=m; i++)
  {
    cin >> a >> b;
    dis[a][b]=dis[b][a]=sqrt(pf(x[a]-x[b])+pf(y[a]-y[b]));
    pre[a][b]=a;    //a 与 b 相连,自然从 a 到 b 的最短路径 b 的前驱是 a
    pre[b][a]=b;
  }
  cin >> a >> b;
  for (k=1; k <=n; k++)                  //floyed 最短路
   for (i=1; i <=n; i++)
    for (j=1; j <=n; j++)
     if ((i !=j) && (i !=k) && (j !=k))
      if (dis[i][j] > dis[i][k]+dis[k][j])
      {
       dis[i][j]=dis[i][k]+dis[k][j];
```

```
            pre[i][j]=pre[k][j];
            //从i到j的最短路径更新为i→k→j,那么i到j最短路径j的前驱就肯定与
              k到j最短路径j的前驱相同
        }
    cout << a;
    print(b);                                    //a是起点,b是终点
    return 0;
}
```

最后再稍微提一提有向图求最短路径的方法:对有向图求最短路径上面的算法可以直接使用,只需注意如果从i到j只有一条有向边,w[i][j]赋值为这条边的权值,而将w[j][i]赋值为无穷大即可。

【上机练习】

1. 信使(msner)

【问题描述】

战争时期,前线有n个哨所,每个哨所可能会与其他若干个哨所之间有通信联系。信使负责在哨所之间传递信息,当然,这是要花费一定时间的(以天为单位)。指挥部设在第一个哨所。当指挥部下达一个命令后,指挥部就派出若干个信使向与指挥部相连的哨所送信。当一个哨所接到信后,这个哨所内的信使们也以同样的方式向其他哨所送信。直至所有n个哨所全部接到命令后,送信才算成功。因为准备充足,每个哨所内都安排了足够的信使(如果一个哨所与其他k个哨所有通信联系的话,这个哨所内至少会配备k个信使)。

现在总指挥请你编一个程序,计算出完成整个送信过程最短需要多少时间。

【输入格式】

输入文件msner.in,第1行有两个整数n和m,中间用1个空格隔开,分别表示有n个哨所和m条通信线路。1<=n<=100。

第2至m+1行:每行三个整数i、j、k,中间用1个空格隔开,表示第i个和第j个哨所之间存在通信线路,且这条线路要花费k天。

【输出格式】

输出文件msner.out,仅一个整数,表示完成整个送信过程的最短时间。如果不是所有的哨所都能收到信,就输出-1。

【输入样例】

```
4 4
1 2 4
2 3 7
2 4 1
3 4 6
```

【输出样例】

```
11
```

2. 最优乘车(travel)

【问题描述】

H城是一个旅游胜地,每年都有成千上万的人前来观光。为方便游客,巴士公司在各个

旅游景点及宾馆、饭店等地都设置了巴士站并开通了一些单程巴士线路。每条单程巴士线路从某个巴士站出发，依次途经若干个巴士站，最终到达终点巴士站。

一名旅客最近到H城旅游，他很想去S公园游玩，但如果从他所在的饭店没有一路巴士可以直接到达S公园，则他可能要先乘某一路巴士坐几站，再下来换乘同一站台的另一路巴士，这样换乘几次后到达S公园。

现在用整数1,2,…,N给H城的所有的巴士站编号，约定这名旅客所在饭店的巴士站编号为1，S公园巴士站的编号为N。

写一个程序，帮助这名旅客寻找一个最优乘车方案，使他在从饭店乘车到S公园的过程中换车的次数最少。

【输入格式】

输入文件是travel.in。文件的第1行有两个数字M和N(1<=M<=100 1<N<=500)，表示开通了M条单程巴士线路，总共有N个车站。从第2行到第M行依次给出了第1条到第M条巴士线路的信息。其中第i+1行给出的是第i条巴士线路的信息，从左至右按运行顺序依次给出了该线路上的所有站号，相邻两个站号之间用一个空格隔开。

【输出格式】

输出文件是travel.out，文件只有一行。如果无法乘巴士从饭店到达S公园，则输出"NO"，否则输出你的程序所找到的最少换车次数，换车次数为0表示不需换车即可到达。

【输入样例】

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

【输出样例】

```
2
```

3. 最短路径

【问题描述】

给出一个有向图G=(V, E)和一个源点$v_0 \in V$，请写一个程序输出v_0和图G中其他顶点的最短路径。只要所有的有向环权值和都是正的，我们就允许图的边有负值。顶点的标号从1到n(n为图G的顶点数)。

【输入格式】

第1行：一个正数n(2<=n<=80)，表示图G的顶点总数。

第2行：一个整数，表示源点v_0($v_0 \in V$，v_0可以是图G中任意一个顶点)。

第3至第n+2行，用一个邻接矩阵W给出了这个图。

【输出格式】

共包含n-1行，按照顶点编号从小到大的顺序，每行输出源点v_0到一个顶点的最短距离。每行的具体格式参照样例。

【输入样例】

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

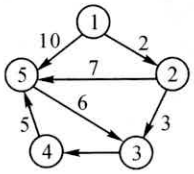
【输出样例】

```
(1-> 2)=2
(1-> 3)=5
(1-> 4)=9
(1-> 5)=9
```

---05

--6-0

样例所对应的图如下：



$$W=\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 2 & - & - & 10 \\ - & 0 & 3 & - & 7 \\ - & - & 0 & 4 & - \\ - & - & - & 0 & 5 \\ - & - & 6 & - & 0 \end{bmatrix} \end{array}$$

图 4-17

4. 热浪(heatwv)

【问题描述】

得克萨斯纯朴的民众们这个夏天正在遭受巨大的热浪！他们的得克萨斯长角牛吃起来不错，可是他们并不是很擅长生产富含奶油的乳制品。Farmer John 此时以先天下之忧而忧，后天下之乐而乐的精神，身先士卒地承担起向得克萨斯运送大量营养冰凉的牛奶的重任，以减轻得克萨斯人忍受酷暑的痛苦。

Farmer John 已经研究过可以把牛奶从威斯康星运送到得克萨斯州的路线。这些路线包括起始点和终点先一共经过 T (1 <=T <=2500)个城镇，方便地标号为 1 到 T。除了起点和终点外的每个城镇由两条双向道路连向至少两个其他城镇。每条道路有一个通过费用(包括油费、过路费等)。

给定一个地图，包含 C (1 <=C <=6200)条直接连接 2 个城镇的道路。每条道路由道路的起点 Rs、终点 Re (1 <=Rs <=T; 1 <=Re <=T)和花费(1 <=Ci <=1000)组成。求从起始的城镇 Ts (1 <=Ts <=T)到终点的城镇 Te(1 <=Te <=T)最小的总费用。

【输入格式】

第 1 行：4 个由空格隔开的整数：T，C，Ts，Te。

第 2 到第 C+1 行：第 i+1 行描述第 i 条道路。有 3 个由空格隔开的整数：Rs，Re 和 Ci。

【输出格式】

一个单独的整数表示从 Ts 到 Te 的最小总费用。数据保证至少存在一条道路。

【输入样例】

```
7 11 5 4
2 4 2
1 4 3
7 2 2
3 4 3
5 7 5
7 3 3
6 1 1
6 3 4
2 4 3
5 6 3
7 2 1
```

【输出样例】

```
7
```

【样例说明】

5->6->1->4 (3+1+3)

5. 分糖果(candy)

【问题描述】

童年的我们,将和朋友分享美好的事物作为自己的快乐。这天,C 小朋友得到了的糖果,将要把这些糖果分给要好的朋友们。已知糖果从一个人传给另一个人需要 1 秒的时间,同一个小朋友不会重复接受糖果。由于糖果足够多,如果某时刻某小朋友接受了糖果,他会将糖果分成若干份,分给那些在他身旁且还没有得到糖果的小朋友们,而且自己会吃一些糖果。由于嘴馋,小朋友们等不及将糖果发完,会在得到糖果后边吃边发。每个小朋友从接受糖果到吃完糖果需要 m 秒的时间。那么,如果第一秒 C 小朋友开始发糖,第几秒所有小朋友都吃完了糖呢?

【输入格式】

第 1 行为三个数 n、p、c,为小朋友数、关系数和 C 小朋友的编号。

第 2 行为一个数 m,表示小朋友吃糖的时间。

下面 p 行每行两个整数,表示某两个小朋友在彼此身旁。

【输出格式】

一个数,为所有小朋友都吃完了糖的时间。

【输入样例】

```
4 3 1
2
1 2
2 3
1 4
```

【输出样例】

```
5
```

【样例说明】

第一秒,糖在 1 手上。第二秒,糖传到了 2、3 的手中。第三秒,糖传到了 4 的手中,此时 1 吃完了。第四秒,2、3 吃完了。第五秒,4 吃完了。所以答案是 5。

【数据范围】

40%的数据满足:1<=n<=100;

60%的数据满足:1<=n<=1000;

100%的数据满足:1<=n<=100000;

m<=n*(n-1)/2,不会有同一个关系被描述多次的情况。

第四节　图的连通性问题

一、判断图中的两点是否连通

1. Floyed 算法

时间复杂度:$O(N^3)$

算法实现:

把相连的两点间的距离设为 dis[i][j]=true,不相连的两点设为 dis[i][j]=false,用 Floyed 算法的变形:

```
for (k=1; k<=n; k++)
  for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
      dis[i][j]=dis[i][j] || (dis[i][k] && dis[k][j]);
```

最后如果 dis[i][j]=true 的话,那么就说明 i,j 两点之间有路径连通。

有向图与无向图都适用。

2. 遍历算法

时间复杂度:$O(N^2)$

算法实现:

从任意一个顶点出发,进行一次遍历,能够从这个点出发到达的点就与起点是连通的。这样就可以求出此顶点和其他各个顶点的连通情况。所以只要把每个顶点作为出发点都进行一次遍历,就能知道任意两个顶点之间是否有路存在。

可以使用 DFS 实现。

有向图与无向图都适用。

二、最小环问题

最小环就是指在一张图中找出一个环,使得这个环上的各条边的权值之和最小。在 Floyed 的同时,可以顺便算出最小环。

记两点间的最短路为 dis[i][j],g[i][j]为边<i,j>的权值。

```
for  (k=1; k<=n; k++)
{
  for  (i=1; i<=k-1; i++)
    for  (j=i+1; j<=k-1; j++)
      answer=min(answer,dis[i][j]+g[j][k]+g[k][i]);
  for  (i=1; i<=n; i++)
    for  (j=1; j<=n; j++)
        dis[i][j]=min(dis[i][j],dis[i][k]+dis[k][j]);
}
```

answer 即为这张图的最小环。

一个环中的最大结点为 k(编号最大),与它相连的两个点为 i、j,这个环的最短长度为 g[i][k]+g[k][j]+(i 到 j 的路径中,所有结点编号都小于 k 的最短路径长度)。

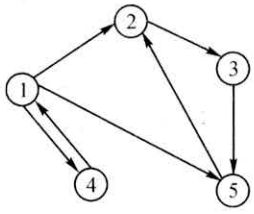
根据 Floyed 的原理,在最外层循环做了 k－1 次之后,dis[i][j]则代表了 i 到 j 的路径中,所有结点编号都小于 k 的最短路径。

综上所述,该算法一定能找到图中最小环。

三、求有向图的强连通分量

Kosaraju 算法可以求出有向图中的强连通分量个数,并且对分属于不同强连通分量的点进行标记。它的算法描述较为简单:

(1) 第一次对图 G 进行 DFS 遍历,并在遍历过程中,记录每一个点的退出顺序。以图 4－18为例:



G

图 4－18

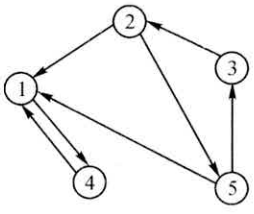
如果以 1 为起点遍历,访问结点的顺序如下:

1	2	3	5	5	3	2	1	4	4	1

结点第二次被访问即为退出之时,那么我们可以得到结点的退出顺序:

5	3	2	4	1

(2) 倒转每一条边的方向,构造出一个反图 G'。然后按照退出顺序的逆序对反图进行第二次 DFS 遍历。我们按 1、4、2、3、5 的逆序第二次 DFS 遍历:



G'

图 4－19

访问过程如下:

1	4	退出	2	5	3	退出

每次遍历得到的那些点即属于同一个强连通分量。1、4 属于同一个强连通分量,2、3、5 属于另一个强连通分量。

【上机练习】

刻录光盘(cdrom)

【问题描述】

在FJOI2010夏令营快要结束的时候,很多营员提出来要把整个夏令营期间的资料刻录成一张光盘给大家,以便大家回去后继续学习。组委会觉得这个主意不错!可是组委会一时没有足够的空光盘,没法保证每个人都能拿到刻录上资料的光盘,怎么办呢?!

DYJ分析了一下所有营员的地域关系,发现有些营员是一个城市的,其实他们只需要一张就可以了,因为一个人拿到光盘后,其他人可以带着U盘之类的东西去拷贝啊!

他们愿意某一些人到他那儿拷贝资料,当然也可能不愿意让另外一些人到他那儿拷贝资料,这与我们FJOI宣扬的团队合作精神格格不入!

现在假设总共有N个营员(2<=N<=200),每个营员的编号为1~N。DYJ给每个人发了一张调查表,让每个营员填上自己愿意让哪些人到他那儿拷贝资料。当然,如果A愿意把资料拷贝给B,而B又愿意把资料拷贝给C,则一旦A获得了资料,则B、C都会获得资料。

现在,请你编写一个程序,根据回收上来的调查表,帮助DYJ计算出组委会至少要刻录多少张光盘,才能保证所有营员回去后都能得到夏令营资料?

【输入格式】

先是一个数N,接下来的N行,分别表示各个营员愿意把自己获得的资料拷贝给其他哪些营员,即输入数据的第i+1行表示第i个营员愿意把资料拷贝给那些营员的编号,以一个0结束。如果一个营员不愿意拷贝资料给任何人,则相应的行只有1个0,一行中的若干数之间用一个空格隔开。

【输出格式】

一个正整数,表示最少要刻录的光盘数。

【输入样例】

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

【输出样例】

```
1
```

第五节　并查集

在一些有 N 个元素的集合应用问题中，我们通常是在开始时让每个元素构成一个单元素的集合，然后按一定顺序将属于同一组的元素所在的集合合并，其间要反复查找一个元素在哪个集合中。这一类问题近几年来反复出现在信息学的国际国内竞赛题中，其特点是看似并不复杂，但数据量极大，若用普通的数据结构来描述的话，往往超过了空间的限制，计算机无法承受；即使在空间上能勉强通过，运行的时间复杂度也极高，根本不可能在比赛规定的运行时间内计算出试题需要的结果，只能采用一种特殊数据结构——并查集来描述。

一、引例

例 4.9　亲戚(relation)

【问题描述】

或许你并不知道，你的某个朋友是你的亲戚。他可能是你的曾祖父的外公的女婿的外甥女的表姐的孙子。如果能得到完整的家谱，判断两个人是否是亲戚应该是可行的，但如果两个人的最近公共祖先与他们相隔好几代，使得家谱十分庞大，那么检验亲戚关系实非人力所能及。在这种情况下，最好的帮手就是计算机。为了将问题简化，你将得到一些亲戚关系的信息，如 Marry 和 Tom 是亲戚，Tom 和 Ben 是亲戚，等。从这些信息中，你可以推出 Marry 和 Ben 是亲戚。请写一个程序，对于我们的关于亲戚关系的提问，以最快的速度给出答案。

【输入格式】

输入由两部分组成。

第一部分以 N，M 开始。N 为问题涉及的人的个数(1 <= N <= 20000)。这些人的编号为 1，2，3，…，N。下面有 M 行(1 <= M <= 1000000)，每行有两个数 a_i，b_i，表示已知 a_i 和 b_i 是亲戚。

第二部分以 Q 开始。以下 Q 行有 Q 个询问(1 <= Q <= 1000000)，每行为 c_i、d_i，表示询问 c_i 和 d_i 是否为亲戚。

【输出格式】

对于每个询问 c_i、d_i，输出一行：若 c_i 和 d_i 为亲戚，则输出“Yes”，否则输出“No”。

【输入样例】

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

【输出样例】

```
Yes
No
Yes
```

5 6

2 3

3

3 4

7 10

8 9

【算法分析】

将每个人抽象成为一个点,数据给出 M 个边的关系,两个人是亲戚的时候两点间有一条边。很自然的就得到了一个 N 个顶点 M 条边的图论模型,注意到传递关系,在图中一个连通块中的任意点之间都是亲戚。对于最后的 Q 个提问,即判断所提问的两个顶点是否在同一个连通块中。

用传统的思路,可以马上反应过来,对于输入的 N 个点 M 条边,找出连通块,然后进行判断。但这种实现思路首先必须保存 M 条边,然后再进行普通的遍历算法,效率显然不高。再进一步考虑,如果把题目的要求改一改,对于边和提问相间输入,即把题目改成:

第一行是 N,M。N 为问题涉及的人的个数(1<=N<=20000)。这些人的编号为 1,2,3,…, N。下面有 M 行(1<=M<=2000000),每行有三个数 k_i、a_i、b_i。a_i、b_i 表示两个顶点,k_i 为 0 或 1,k_i 为 1 时表示这是一条边的信息,即 a_i、b_i 是亲戚关系;k_i 为 0 时表示是一个提问,根据此行以前所得到的信息,判断 a_i、b_i 是否亲戚,对于每条提问回答“Yes”或者“No”。

这个问题比原问题更复杂些,需要在任何时候回答提问的两个人的关系,并且对于信息提示还要能立即合并两个连通块。采用连通图思想显然在实现上就有所困难,因为要表示人与人之间的关系。

用集合的思路,对于每个人建立一个集合,开始的时候集合元素是这个人本身,表示开始时不知道任何人是他的亲戚。以后每次给出一个亲戚关系时,就将两个集合合并。这样实时地得到了在当前状态下的集合关系。如果有提问,即在当前得到的结果中看两元素是否属于同一集合。对于样例数据的解释如图 4-20:

输入关系	分离集合
初始状态	{1}{2}{3}{4}{5}{6}{7}{8}{9}{10}
(2,4)	{1}{2,4}{3}{5}{6}{7}{8}{9}{10}
(5,7)	{1}{2,4}{3}{5,7}{6}{8}{9}{10}
(1,3)	{1,3}{2,4}{5,7}{6}{8}{9}{10}
(8,9)	{1,3}{2,4}{5,7}{6}{8,9}{10}
(1,2)	{1,2,3,4}{5,7}{6}{8,9}{10}
(5,6)	{1,2,3,4}{5,6,7}{8,9}{10}
(2,3)	{1,2,3,4}{5,6,7}{8,9}{10}

图 4-20

由图 4-20 可以看出,操作是在集合的基础上进行的,没有必要保存所有的边,而且每

一步得到的划分方式是动态的。

那么，如何来实现以上的算法思想呢？我们就用到并查集。

二、并查集的基本思想

1. 什么叫并查集

并查集(union－find set)是一种用于分离集合操作的抽象数据类型。它所处理的是“集合”之间的关系，即动态地维护和处理集合元素之间复杂的关系，当给出两个元素的一个无序对(a,b)时，需要快速“合并”a 和 b 分别所在的集合，这其间需要反复“查找”某元素所在的集合。“并”、“查”和“集”三字由此而来。在这种数据类型中，n 个不同的元素被分为若干组，每组是一个集合，这种集合叫做分离集合(disjoint set)。并查集支持查找一个元素所属的集合以及两个元素各自所属的集合的合并。

例如，有这样的问题：初始时 n 个元素分属不同的 n 个集合，通过不断的给出元素间的联系，要求实时的统计元素间的关系(是否存在直接或间接的联系)。这时就有了并查集的用武之地了。元素间是否有联系，只要判断两个元素是否属于同一个集合；而给出元素间的联系，建立这种联系，则只需合并两个元素各自所属的集合。这些操作都是并查集所提供的。

并查集本身不具有结构，必须借助一定的数据结构以得到支持和实现。数据结构的选择是一个重要的环节，选择不同的数据结构可能会在查找和合并的操作效率上有很大的差别，但操作实现都比较简单高效。并查集的数据结构实现方法很多，数组实现、链表实现和树实现。一般用的比较多的是数组实现。

2. 并查集支持的操作

并查集的数据结构记录了一组分离的动态集合 S＝{S1,S2,…,Sk}。每个集合通过一个代表加以识别，代表即该元素中的某个元素，哪一个成员被选做代表是无所谓的，重要的是：如果求某一动态集合的代表两次，且在两次请求间不修改集合，则两次得到的答案应该是相同的。

动态集合中的每一元素是由一个对象来表示的，设 x 表示一个对象，并查集的实现需要支持如下操作：

MAKE(x)：建立一个新的集合，其仅有的成员(同时就是代表)是 x。由于各集合是分离的，要求 x 没有在其他集合中出现过。

UNIONN(x,y)：将包含 x 和 y 的动态集合(例如 Sx 和 Sy)合并为一个新的集合，假定在此操作前这两个集合是分离的。结果的集合代表是 Sx∪Sy 的某个成员。一般来说，在不同的实现中通常都以 Sx 或者 Sy 的代表作为新集合的代表。此后，由新的集合 S 代替了原来的 Sx 和 Sy。

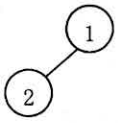
FIND(x)：返回一个指向包含 x 的集合的代表。

对于引例的问题，我们可以运用并查集简单地进行如下做法：

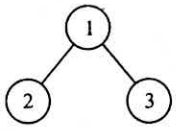
(1)元素的合并图示:



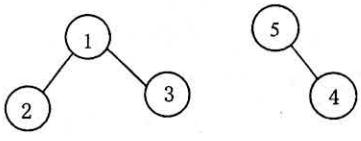
① 合并 1 和 2



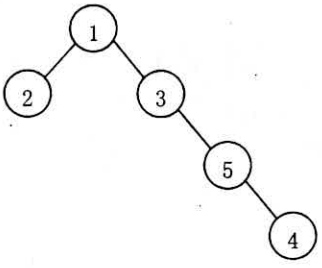
② 合并 1 和 3



③ 合并 5 和 4



④ 合并 3 和 5



用 father[i]表示元素 i 的父亲结点,进行不断并到不同的集合中。

(2)再对输入的数据进行判断:是否在同一集合。

具体程序如下:

```
#include<iostream>
#include<cstdio>
using namespace std;
#define maxn 20001
int father[maxn];
int m,n,i,x,y,q;
/*
int find(int x)                                    //用非递归的实现
{
  while (father[x] !=x) x=father[x];
  return x;
}
*/
```

```
int find(int x)                          //用递归的实现
{
  if (father[x] ! =x) return find(father[x]);
  else return x;
}
void unionn(int r1,int r2)
{
  father[r2]=r1;
}
int main()
{
  cin >> n >> m;
  for (i=1; i <=n; i++)
      father[i]=i;                       //建立新的集合,其仅有的成员是 i
  for (i=1; i <=m; i++)
  {
      scanf("%d%d",&x,&y);
      int r1=find(x);
      int r2=find(y);
      if (r1! =r2) unionn(r1,r2);
  }
  cin >> q;
  for (i=1; i <=q; i++)
  {
      scanf("%d%d",&x,&y);
      if (find (x) == find(y)) printf("Yes\n");
                  else printf("No\n");
  }
  return 0;
}
```

以上做法当数据比较特殊的时候,比如一条单链老长,数据这种“并”与“查”的方式肯定会超时。

(3)下面有一种优化的方法:

并查集的路径压缩

此种做法就是将元素的父亲结点指来指去地指,当这棵树是链的时候,可见判断两个元素是否属于同一集合需要 O(n)的时间,于是路径压缩产生了作用。

路径压缩实际上是在找完根结点之后,在递归回来的时候顺便把路径上元素的父亲指针都指向根结点。

这就是说,我们在“合并 5 和 3”的时候,不是简单地将 5 的父亲指向 3,而是直接指向根结点 1,如图 4-21:

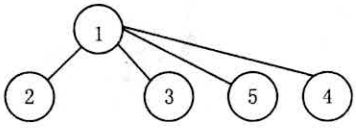


图 4-21

由此我们得到了一个复杂度几乎为常数的算法。

【程序清单】

(1)初始化:

```
for (i=1; i <=n; i++) father[i]=i;
```

因为每个元素属于单独的一个集合,所以每个元素以自己作为根结点。

(2)寻找根结点编号并压缩路径:

```
int find (int x)
{
  if (father[x] !=x) father[x]=find (father[x]);
  return father[x];
}
```

(3)合并两个集合:

```
void unionn(int x,int y)
{
  x=find(x);y=find(y);
  father[y]=x;
}
```

(4)判断元素是否属于同一集合:

```
bool  judge(int x,int y)
{
  x=find(x);
  y=find(y);
  if  (x==y)  return true;
      else return false;
}
```

这个的引题已经完全阐述了并查集的基本操作和作用。

优化的具体程序如下:

```
#include<iostream>
#include<cstdio>
using namespace std;
#define maxn 20001
int father[maxn];
```

```
int m,n,i,x,y,q;
/*
int find(int x)                                    //用非递归的实现
{
  while (father[x] !=x) x=father[x];
  return x;
}
*/
int find(int x)                                    //用递归的实现
{
  if (father[x] !=x) father[x]=find(father[x]);   //路径压缩,优化的核心
  return father[x];
}
void unionn(int r1,int r2)
{
  father[r2]=r1;
}
int main()
{
  cin >> n >> m;
  for (i=1; i <=n; i++)
      father[i]=i;                                 //建立新的集合,其仅有的成员是 i
  for (i=1; i <=m; i++)
  {
      scanf("%d%d",&x,&y);
      int r1=find(x);
      int r2=find(y);
      if (r1!=r2) unionn(r1,r2);
  }
  cin >> q;
  for (i=1; i <=q; i++)
  {
      scanf("%d%d",&x,&y);
      if (find (x) == find(y)) printf("Yes\n");
                    else printf("No\n");
  }
  return 0;
}
```

这种做法就可能不会超时了

例 4.10 格子游戏

【问题描述】

Alice 和 Bob 玩了一个古老的游戏:首先画一个 n * n 的点阵(图 4－22 中 n=3)

接着,他们两个轮流在相邻的点之间画上虚边和粗边:

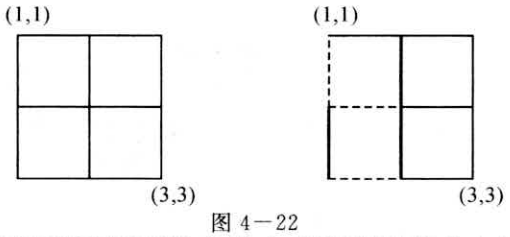


图 4－22

直到围成一个封闭的圈(面积不必为 1)为止,"封圈"的那个人就是赢家。因为棋盘实在是太大了(n <=200),他们的游戏实在是太长了!他们甚至在游戏中都不知道谁赢得了游戏。于是请你写一个程序,帮助他们计算他们是否结束了游戏?

【输入格式】

输入数据第一行为两个整数 n 和 m。m 表示一共画了 m 条线。以后 m 行,每行首先有两个数字(x, y),代表了画线的起点坐标,接着用空格隔开一个字符,假如字符是"D",则是向下连一条边,如果是"R"就是向右连一条边。输入数据不会有重复的边且保证正确。

【输出格式】

输出一行:在第几步的时候结束。假如 m 步之后也没有结束,则输出一行"draw"。

【输入样例】

```
3 5
1 1 D
1 1 R
1 2 D
2 1 R
2 2 D
```

【输出样例】

```
4
```

【参考程序】

```
#include <iostream>
using namespace std;
struct node
{ int x,y;}f[301][301],k1,k2;
int i,j,m,n,x,y;
char c;
node root(node k)
{
    if((f[k.x][k.y].x==k.x)&&(f[k.x][k.y].y==k.y))return k;
    f[k.x][k.y]=root(f[k.x][k.y]);
```

```
    return f[k.x][k.y];
}
int main()
{
  cin>>n>>m;
  for(i=1;i<=n;i++)
  for(j=1;j<=n;j++)
    {f[i][j].x=i;f[i][j].y=j;}
  for(i=1;i<=m;i++)
  {
      cin>>x>>y>>c;
      if(c=='D')
        {k1=root(f[x][y]);k2=root(f[x+1][y]);}
      if(c=='R')
        {k1=root(f[x][y]);k2=root(f[x][y+1]);}
      if((k1.x==k2.x)&&(k1.y==k2.y))
        {cout<<i<<endl;return 0;}
      else f[k1.x][k1.y]=k2;
  }
cout<<" draw "<<endl;
return 0;
}
```

三、求无向图的连通分量

这个问题其实在例 4.7 中已经提过了。这里再单独提出来强调一下，因为求无向图连通分量是个非常常用的算法。通过并查集可以使得空间上省去对边的保存，同时时间效率又是很高的。

需要特别指出的是，如果用链表来实现的话，最后任何在同一个集合（即连通块）中的元素，其代表指针的值都是相等的。而采用有根树来实现的话，算法结束后，留下的依然是树的关系，因此如果希望每个元素都指向它的根的话，还需要对每个节点进行一次 find 操作，这样每个节点的父节点都是代表此集合的节点。在某些统计问题中，往往需要这样做。

四、Kruskal 最小生成树算法

此经典算法的思想是将树上的边按照边权排序，然后从小到大分析每一条边，如果选到一条边 e=(v1,v2)，且 v1 和 v2 不在一个连通块中，就将 e 作为最小生成树的一条边，否则

忽略 e。这其中明显就包含了并查集的算法。Kruskal 算法也只有在结合了并查集后才能说是个高效的算法。

五、小结

总之在解决某些特定的问题时,并查集往往能够发挥出重要的作用,大家一定要熟悉这种算法,能举一反三。

【上机练习】

1. 团伙(group)

【问题描述】

在某城市里住着 n 个人,任何两个认识的人不是朋友就是敌人,而且满足:

(1)我朋友的朋友是我的朋友;

(2)我敌人的敌人是我的朋友。

所有是朋友的人组成一个团伙。告诉你关于这 n 个人的 m 条信息,即某两个人是朋友,或者某两个人是敌人,请你编写一个程序,计算出这个城市最多可能有多少个团伙?

【输入格式】

第 1 行为 n 和 m,1<n<1000,1<=m<=100 000;

以下 m 行,每行为 p、x、y,p 的值为 0 或 1,p 为 0 时,表示 x 和 y 是朋友,p 为 1 时,表示 x 和 y 是敌人。

【输出格式】

一个整数,表示这 n 个人最多可能有几个团伙。

【输入样例】

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

【输出样例】

```
3
```

2. 打击犯罪(black)

【问题描述】

某个地区有 n(n<=1000)个犯罪团伙,当地警方按照他们的危险程度由高到低给他们编号为 1—n,他们有些团伙之间有直接联系,但是任意两个团伙都可以通过直接或间接的方式联系,这样这里就形成了一个庞大的犯罪集团,犯罪集团的危险程度由集团内的犯罪团伙数量唯一确定,而与单个犯罪团伙的危险程度无关(该犯罪集团的危险程度为 n)。现在当地警方希望花尽量少的时间(即打击掉尽量少的团伙),使得庞大的犯罪集团分离成若干个较小的集团,并且他们中最大的一个的危险程度不超过 n/2。为达到最好的效果,他们将按顺序打击掉编号 1 到 k 的犯罪团伙,请编程求出 k 的最小值。

【输入格式】

第1行一个正整数n。接下来的n行每行有若干个正整数，第一个整数表示该行除第一个外还有多少个整数，若第i行存在正整数k，表示i、k两个团伙可以直接联系。

【输出格式】

一个正整数，为k的最小值。

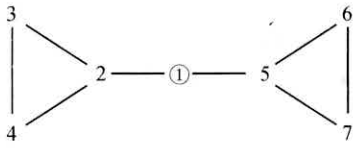
【输入样例】

7
2 2 5
3 1 3 4
2 2 4
2 2 3
3 1 6 7
2 5 7
2 5 6

【输出样例】

1

【提示】



输出1(打击掉的犯罪团伙)

3. 搭配购买(buy)

【问题描述】

Joe觉得云朵很美，决定去山上的商店买一些云朵。商店里有n朵云，云朵被编号为1，2，…，n，并且每朵云都有一个价值。但是商店老板跟他说，一些云朵要搭配来买才好，所以买一朵云则与这朵云有搭配的云都要买。但是Joe的钱有限，所以他希望买的价值越多越好。

【输入格式】

第1行n、m、w，表示n朵云，m个搭配，Joe有w的钱。

第2至n+1行，每行ci、di表示i朵云的价钱和价值。

第n+2至n+1+m行，每行ui、vi表示买ui就必须买vi，同理，如果买vi就必须买ui。

【输出格式】

一行，表示可以获得的最大价值。

【输入样例】

5 3 10
3 10
3 10
3 10
5 100

【输出样例】

1

10 1

1 3

3 2

4 2

【数据范围】

30%的数据满足:n<=100;

50%的数据满足:n<=1000;m<=100;w<=1000;

100%的数据满足:n<=10000;0<=m<=5000;w<=10000。

4. 家谱(gen)

【问题描述】

现代的人对于本家族血统越来越感兴趣,现在给出充足的父子关系,请你编写程序找到某个人的最早的祖先。

【输入格式】

输入文件由多行组成,首先是一系列有关父子关系的描述,其中每一组父子关系由二行组成,用#name的形式描写一组父子关系中的父亲的名字,用+name的形式描写一组父子关系中的儿子的名字;接下来用? name的形式表示要求该人的最早的祖先;最后用单独的一个$表示文件结束。规定每个人的名字都有且只有6个字符,而且首字母大写,且没有任意两个人的名字相同。最多可能有1000组父子关系,总人数最多可能达到50000人,家谱中的记载不超过30代。

【输出格式】

按照输入文件的要求顺序,求出每一个要找祖先的人的祖先,格式:本人的名字+一个空格+祖先的名字+回车。

【输入样例】

#George
+Rodney
#Arthur
+Gareth
+Walter
#Gareth
+Edward
? Edward
? Walter
? Rodney
? Arthur
$

【输出样例】

Edward Arthur
Walter Arthur
Rodney George
Arthur Arthur

第六节　最小生成树

一、什么是图的最小生成树(MST)

不知道大家还记不记得树的一个定理：N 个点用 N－1 条边连接成一个连通块，形成的图形只可能是树，没有别的可能。

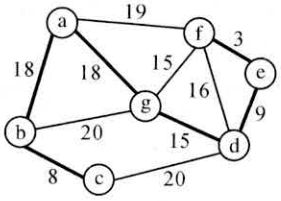


图 4－23

一个有 N 个点的图，边一定是大于等于 N－1 条的。图的最小生成树，就是在这些边中选择 N－1 条出来，连接所有的 N 个点。这 N－1 条边的边权之和是所有方案中最小的。

二、最小生成树用来解决什么问题

就是用来解决如何用最小的“代价”用 N－1 条边连接 N 个点的问题。例如：

例 4.11　城市公交网建设问题

【问题描述】

有一张城市地图，图中的顶点为城市，无向边代表两个城市间的连通关系，边上的权为在这两个城市之间修建高速公路的造价，研究后发现，这个地图有一个特点，即任一对城市都是连通的。现在的问题是，要修建若干高速公路把所有城市联系起来，问如何设计可使得工程的总造价最少？

【输入格式】

n(城市数，1<=n<=100)

e(边数)

以下 e 行，每行 3 个数 i、j、w_{ij}，表示在城市 i、j 之间修建高速公路的造价。

【输出格式】

n－1 行，每行为两个城市的序号，表明这两个城市间建一条高速公路。

【输入样例】

```
5 8
1 2  2
2 5  9
5 4  7
```

【输出样例】

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

4 1 10
1 3 12
4 3 6
5 3 3
2 3 8

1. Prim 算法

Prim 算法采用与 Dijkstra、Bellman－Ford 算法一样的“蓝白点”思想：白点代表已经进入最小生成树的点，蓝点代表未进入最小生成树的点。

算法描述：

以 1 为起点生成最小生成树，min[v]表示蓝点 v 与白点相连的最小边权。

MST 表示最小生成树的权值之和。

(a)初始化：min[v]=∞(v≠1)；min[1]=0；MST=0；

(b)for (i=1；i<=n；i++)

①寻找 min[u]最小的蓝点 u。

②将 u 标记为白点。

③MST+=min[u]。

④for 与白点 u 相连的所有蓝点 v

```
if (w[u][v]<min[v])
    min[v]=w[u][v];
```

(c)算法结束：MST 即为最小生成树的权值之和。

算法分析 & 思想讲解：

Prim 算法每次循环都将一个蓝点 u 变为白点，并且此蓝点 u 与白点相连的最小边权 min[u]还是当前所有蓝点中最小的。这样相当于向生成树中添加了 n－1 次最小的边，最后得到的一定是最小生成树。

我们通过对图 4－24 最小生成树的求解模拟来理解上面的思想。蓝点和虚线代表未进入最小生成树的点、边；白点和实线代表已进入最小生成树的点、边。

初始时所有点都是蓝点，min[1]=0，min[2、3、4、5]=∞，权值之和 MST=0。

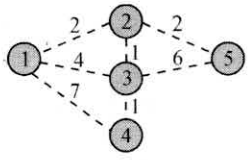
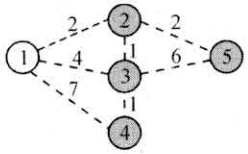


图 4－24

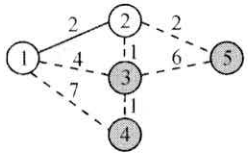
第一次循环自然是找到 min[1]=0 最小的蓝点 1。将 1 变为白点，接着枚举与 1 相连的所有蓝点 2、3、4，修改它们与白点相连的最小边权。




```
min[2]=w[1][2]=2;
min[3]=w[1][3]=4;
min[4]=w[1][4]=7;
```

图 4－25

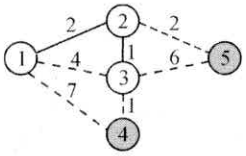
第二次循环是找到 min[2]最小的蓝点 2。将 2 变为白点，接着枚举与 2 相连的所有蓝点 3、5，修改它们与白点相连的最小边权。




```
min[3]=w[2][3]=1;
min[5]=w[2][5]=2;
```

图 4－26

第三次循环是找到 min[3]最小的蓝点 3。将 3 变为白点，接着枚举与 3 相连的所有蓝点 4、5，修改它们与白点相连的最小边权。




```
min[4]=w[3][4]=1;
由于min[5]=2<w[3][5]=6;
所以不修改min[5]的值。
```

图 4－27

最后两轮循环将点 4、5 以及边 w[2][5]、w[3][4]添加进最小生成树。

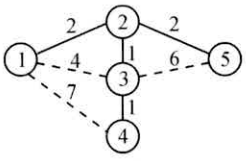


图 4－28

最后权值之和 MST＝6。

这 n 次循环，每次循环我们都能让一个新的点加入生成树，n 次循环就能把所有点囊括到其中；每次循环我们都能让一条新的边加入生成树，n－1 次循环就能生成一棵含有 n 个点的树；每次循环我们都取一条最小的边加入生成树，n－1 次循环结束后，我们得到的就是一棵最小的生成树。

这就是 Prim 采取贪心法生成一棵最小生成树的原理。

算法时间复杂度：O（N^2）。

例 4.12　最优布线问题(wire)

【问题描述】

学校有 n 台计算机，为了方便数据传输，现要将它们用数据线连接起来。两台计算机被连接是指它们有数据线连接。由于计算机所处的位置不同，因此不同的两台计算机的连接费用往往是不同的。

当然，如果将任意两台计算机都用数据线连接，费用将是相当庞大的。为了节省费用，我们采用数据的间接传输手段，即一台计算机可以间接的通过若干台计算机(作为中转)来实现与另一台计算机的连接。

现在由你负责连接这些计算机，任务是使任意两台计算机都连通(不管是直接的或间接

的)。

【输入格式】

输入文件 wire.in,第一行为整数 n(2<=n<=100),表示计算机的数目。此后的 n 行,每行 n 个整数。第 x+1 行 y 列的整数表示直接连接第 x 台计算机和第 y 台计算机的费用。

【输出格式】

输出文件 wire.out,一个整数,表示最小的连接费用。

【输入样例】

```
3
0 1 2
1 0 1
2 1 0
```

【输出样例】

```
2 (注:表示连接 1 和 2、2 和 3,费用为 2)
```

【参考程序】

```
#include<iostream>
#include<cstdio>
#include<cstring>
using namespace std;
int g[101][101];                    //邻接矩阵
int minn[101];                      //minn[i]存放蓝点 i 与白点相连的最小边权
bool u[101];                        //u[i]=True,表示顶点 i 还未加入到生成树中
                                    //u[i]=False,表示顶点 i 已加入到生成树中
int n,i,j;
int main()
{
    cin >> n;
    for (i=1; i <=n; i++)
      for (j=1; j <=n; j++)
        cin >> g[i][j];
    memset(minn,0x7f,sizeof(minn)); //初始化为 maxint
    minn[1]=0;
    memset(u,1,sizeof(u));          //初始化为 True,表示所有顶点为蓝点
    for (i=1; i <=n; i++)
    {
      int k=0;
      for (j=1; j <=n; j++)         //找一个与白点相连的权值最小的蓝点 k
        if (u[j] && (minn[j] < minn[k]))
          k=j;
      u[k]=false;                   //蓝点 k 加入生成树,标记为白点
```

```
        for (j=1; j <=n; j++)      //修改与 k 相连的所有蓝点
          if (u[j] && (g[k][j] < minn[j]))
            minn[j]=g[k][j];
    }
    int total=0;
    for (i=1; i <=n; i++)          //累加权值
      total+=minn[i];
    cout << total << endl;
    return 0;
}
```

2. Kruskal 算法

Kruskal(克鲁斯卡尔)算法是一种巧妙利用并查集来求最小生成树的算法。

首先我们把无向图中相互连通的一些点称为处于同一个连通块中。例如图 4－29：

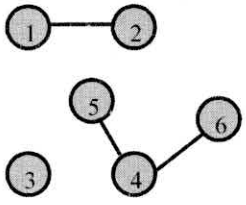


图 4－29

图中有 3 个连通块。1、2 处于一个连通块中，4、5、6 也处于一个连通块中，孤立点 3 也称为一个连通块。

Kruskal 算法将一个连通块当做一个集合。Kruskal 首先将所有的边按从小到大顺序排序(一般使用快排)，并认为每一个点都是孤立的，分属于 n 个独立的集合。然后按顺序枚举每一条边。如果这条边连接着两个不同的集合，那么就把这条边加入最小生成树，这两个不同的集合就合并成了一个集合；如果这条边连接的两个点属于同一集合，就跳过。直到选取了 n－1 条边为止。

算法描述：

1. 初始化并查集：father[x]=x；
2. tot=0；
3. 将所有边用快排从小到大排序。
4. 计数器 k=0；
5. for (i=1; i<=M; i++)　　//循环所有已从小到大排序的边

if　这是一条 u,v 不属于同一集合的边(u,v)(因为已经排序，所以必为最小)

{

①合并 u、v 所在的集合，相当于把边(u,v)加入最小生成树。

②tot=tot+W(u,v)

③k++

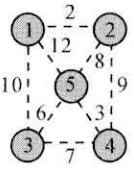
④如果 k=n－1，说明最小生成树已经生成，则 break；

}

6. 结束,tot 即为最小生成树的总权值之和。

思想讲解:

Kruskal(克鲁斯卡尔)算法开始时,认为每一个点都是孤立的,分属于 n 个独立的集合。



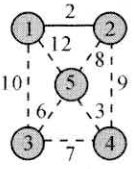
5 个集合{ {1},{2},{3},{4},{5} }

生成树中没有边

图 4-30

Kruskal 每次都选择一条最小的边,而且这条边的两个顶点分属于两个不同的集合。将选取的这条边加入最小生成树,并且合并集合。

第一次选择的是<1,2>这条边,将这条边加入到生成树中,并且将它的两个顶点 1、2 合并成一个集合。

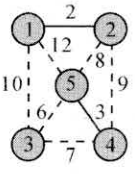


4 个集合{ {1,2},{3},{4},{5} }

生成树中有 1 条边{ <1,2> }

图 4-31

第二次选择的是<4,5>这条边,将这条边加入到生成树中,并且将它的两个顶点 4、5 合并成一个集合。

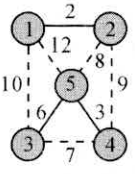


3 个集合{ {1,2},{3},{4,5} }

生成树中有 2 条边{ <1,2> ,<4,5>}

图 4-32

第三次选择的是<3,5>这条边,将这条边加入到生成树中,并且将它的两个顶点 3、5 所在的两个集合合并成一个集合。

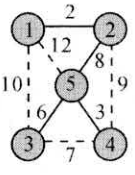


2 个集合{ {1,2},{3,4,5} }

生成树中有 3 条边{ <1,2> ,<4,5>,<3,5>}

图 4-33

第四次选择的是<2,5>这条边,将这条边加入到生成树中,并且将它的两个顶点 2、5 所在的两个集合合并成一个集合。



1 个集合{ {1,2,3,4,5} }

生成树中有 4 条边{ <1,2> ,<4,5>,<3,5>,<2,5>}

图 4－34

算法结束，最小生成树权值为 19。

通过上面的模拟能够看到，Kruskal 算法每次都选择一条最小的，且能合并两个不同集合的边，一张 n 个点的图总共选取 n－1 次边。因为每次我们选的都是最小的边，所以最后的生成树一定是最小生成树。每次我们选的边都能够合并两个集合，最后 n 个点一定会合并成一个集合。通过这样的贪心策略，Kruskal 算法就能得到一棵有 n－1 条边、连接着 n 个点的最小生成树。

Kruskal 算法的时间复杂度为 O(E * logE)，E 为边数。

例 4.13　最短网络(agrinet)

【问题描述】

农民约翰被选为他们镇的镇长！他其中一个竞选承诺就是在镇上建立起互联网，并连接到所有的农场。当然，他需要你的帮助。约翰已经给他的农场安排了一条高速的网络线路，他想把这条线路共享给其他农场。为了用最小的消费，他想铺设最短的光纤去连接所有的农场。你将得到一份各农场之间连接费用的列表，你必须找出能连接所有农场并所用光纤最短的方案。每两个农场间的距离不会超过 100000。

【输入格式】

第 1 行：农场的个数，N(3<=N<=100)。

第 2 行至结尾：后来的行包含了一个 N * N 的矩阵，表示每个农场之间的距离。理论上，它们是 N 行，每行由 N 个用空格分隔的数组成，实际上，它们限制在 80 个字符，因此，某些行会紧接着另一些行。当然，对角线将会是 0，因为不会有线路从第 i 个农场到它本身。

【输出格式】

只有一个输出，其中包含连接到每个农场的光纤的最小长度。

【输入样例】

```
4
0  4  9 21
4  0  8 17
9  8  0 16
21 17 16  0
```

【输出样例】

```
28
```

【参考程序】

```
#include<cstdio>
#include<iostream>
#include<algorithm>                    //sort()需要用到<algorithm>库
using namespace std;
struct point
{
    int x;
    int y;
    int v;
};                                     //定义结构类型，表示边
point a[9901];                         //存边
```

```
int fat[101];
int n,i,j,x,m,tot,k;
int father(int x)
{
    if (fat[x] !=x) fat[x]=father(fat[x]);
    return fat[x];
}
void unionn(int x,int y)
{
    int fa=father(x);
    int fb=father(y);
    if (fa !=fb) fat[fa]=fb;
}
int cmp(const point &a,const point &b)  //sort()自定义的比较函数
{
    if (a.v < b.v) return 1;
      else return 0;
}
int main()
{
    freopen("agrinet.in","r",stdin);
    freopen("agrinet.out","w",stdout);
    cin >> n;
    for (i=1; i <=n; i++)
      for (j=1; j <=n; j++)
      {
          cin >> x;
          if (x !=0)
          {
              m++;
              a[m].x=i; a[m].y=j; a[m].v=x;
          }
      }
    for (i=1; i <=n; i++) fat[i]=i;
    sort(a+1,a+m+1,cmp);        //C++标准库中自带的快排
    //cmp 为自定义的比较函数。表示 a 数组的 1-m 按规则 cmp 排序
    for (i=1; i <=m; i++)
    {
      if (father(a[i].x) !=father(a[i].y))
      {
```

```
            unionn(a[i].x,a[i].y);
            tot+=a[i].v;
            k++;
        }
        if (k==n-1) break;
    }
    cout << tot;
    return 0;
}
```

【上机练习】

1. 局域网(net)

【问题描述】

某个局域网内有 n(n<=100)台计算机，由于搭建局域网时工作人员的疏忽，现在局域网内的连接形成了回路，我们知道如果局域网形成回路，那么数据将不停的在回路内传输，造成网络卡的现象。因为连接计算机的网线本身不同，所以有一些连线不是很畅通，我们用 f(i,j)表示 i、j 之间连接的畅通程度(f(i,j)<=1000)，f(i,j)值越小表示 i、j 之间连接越通畅，f(i,j)为 0 表示 i、j 之间无网线连接。现在我们需要解决回路问题，我们将除去一些连线，使得网络中没有回路，并且被除去网线的 Σf(i,j)最大，请求出这个最大值。

【输入格式】

第一行两个正整数 n、k。

接下来的 k 行每行三个正整数 i、j、m，表示 i、j 两台计算机之间有网线连通，通畅程度为 m。

【输出格式】

一个正整数，Σf(i,j)的最大值。

【输入样例】

```
5 5
1 2 8
1 3 1
1 5 3
2 4 5
3 4 2
```

【输出样例】

```
8
```

2. 繁忙的都市(city)

【问题描述】

城市 C 是一个非常繁忙的大都市，城市中的道路十分的拥挤，于是市长决定对其中的道路进行改造。城市 C 的道路是这样分布的：城市中有 n 个交叉路口，有些交叉路口之间有道路相连，两个交叉路口之间最多有一条道路相连接。这些道路是双向的，且把所有的交叉路口直接或间接的连接起来了。每条道路都有一个分值，分值越小表示这个道路越繁忙，越需要进行改造。但是市政府的资金有限，市长希望进行改造的道路越少越好，于是他提出下面

的要求：

(1)改造的那些道路能够把所有的交叉路口直接或间接的连通起来。

(2)在满足要求(1)的情况下，改造的道路尽量少。

(3)在满足要求(1)(2)的情况下，改造的那些道路中分值最大值尽量小。

【编程任务】

作为市规划局的你，应当作出最佳的决策，选择哪些道路应当被修建。

【输入格式】

第1行有两个整数n、m，表示城市有n个交叉路口，m条道路。接下来m行是对每条道路的描述，u、v、c表示交叉路口u和v之间有道路相连，分值为c。(1<=n<=300，1<=c<=10000)

【输出格式】

两个整数s、max，表示你选出了几条道路，分值最大的那条道路的分值是多少。

【输入样例】

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

【输出样例】

```
3 6
```

3. 联络员(liaison)

【问题描述】

Tyvj已经一岁了，网站也由最初的几个用户增加到了上万个用户，随着Tyvj网站的逐步壮大，管理员的数目也越来越多，现在你身为Tyvj管理层的联络员，希望你找到一些通信渠道，使得管理员两两都可以联络(直接或者是间接都可以)。Tyvj是一个公益性的网站，没有过多的利润，所以你要尽可能地使费用少才可以。

目前你已经知道，Tyvj的通信渠道分为两大类，一类是必选通信渠道，无论价格多少，你都需要把所有的都选择上；还有一类是选择性的通信渠道，你可以从中挑选一些作为最终管理员联络的通信渠道。数据保证给出的通行渠道可以让所有的管理员连通。

【输入格式】

第1行n、m表示Tyvj一共有n个管理员，有m个通信渠道。

第2行到m+1行，每行四个非负整数p、u、v、w，当p=1时，表示这个通信渠道为必选通信渠道；当p=2时，表示这个通信渠道为选择性通信渠道；u、v、w表示本条信息描述的是u、v管理员之间的通信渠道，u可以收到v的信息，v也可以收到u的信息，w表示费用。

【输出格式】

最小的通信费用。

【输入样例】

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

【输出样例】

```
9
```

1 4 1 1

2 2 5 10

2 2 5 5

【样例说明】

1－2－3－4－1 存在四个必选渠道，形成一个环，互相可以到达。需要让所有管理员连通，需要连通 2 号和 5 号管理员，选择费用为 5 的渠道，所以总的费用为 9。

【注意】

u、v 之间可能存在多条通信渠道，你的程序应该累加所有 u、v 之间的必选通行渠道。

【数据范围】

30%的数据满足：n<=10，m<=100；

50%的数据满足：n<=200，m<=1000；

100%的数据满足：n<=2000，m<=10000。

4. 连接格点(grid)

【问题描述】

有一个 M 行 N 列的点阵，相邻两点可以相连。一条纵向的连线花费一个单位，一条横向的连线花费两个单位。某些点之间已经有连线了，试问至少还需要花费多少个单位才能使所有的点全部连通？

【输入格式】

第 1 行输入两个正整数 m 和 n。

以下若干行每行四个正整数 x1、y1、x2、y2，表示第 x1 行第 y1 列的点和第 x2 行第 y2 列的点已经有连线。输入保证|x1－x2|＋|y1－y2|＝1。

【输出格式】

输出使得连通所有点还需要的最小花费。

【输入样例】　　【输出样例】

2 2　　3

1 1 2 1

【数据范围】

30%的数据满足：n * m<=1000；

100%的数据满足：m，n<=1000。

第七节　拓扑排序与关键路径

一、AOV 网

在日常生活中，一项大的工程可以看作是由若干个子工程(这些子工程称为“活动”)组成的集合，这些子工程(活动)之间必定存在一些先后关系，即某些子工程(活动)必须在其他一些子工程(活动)完成之后才能开始，我们可以用有向图来形象地表示这些子工程(活动)之间的先后关系，子工程(活动)为顶点，子工程(活动)之间的先后关系为有向边，这种有向图称为“顶点活动网络”，又称“AOV 网”。

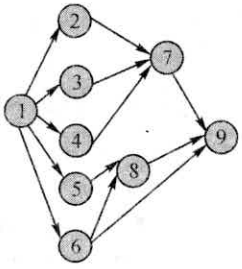


图 4－35

在 AOV 网中，有向边代表子工程(活动)的先后关系，我们把一条有向边起点的活动成为终点活动的前驱活动，同理终点的活动称为起点活动的后继活动。而只有当一个活动全部的前驱全部都完成之后，这个活动才能进行。例如在图 4－35 中，只有当工程 1 完成之后，工程 2、3、4、5、6 才能开始进行。只有当 2、3、4 全部完成之后，7 才能开始进行。

一个 AOV 网必定是一个有向无环图，即不应该带有回路。否则，会出现先后关系的自相矛盾。

图 4－36 就是一个出现环产生自相矛盾的情况。4 是 1 的前驱，想完成 1，必须先完成 4。3 是 4 的前驱，而 2 是 3 的前驱，1 又是 2 的前驱。最后造成想完成 1，必须先完成 1 本身，这显然出现了矛盾。

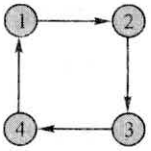


图 4－36

二、拓扑排序算法

拓扑排序算法，只适用于 AOV 网(有向无环图)。

把 AOV 网中的所有活动排成一个序列，使得每个活动的所有前驱活动都排在该活动的前面，这个过程称为“拓扑排序”，所得到的活动序列称为“拓扑序列”。

一个 AOV 网的拓扑序列是不唯一的，例如图 4－37，它的拓扑序列可以是：ABCDE，也可以是 ACBDE，或是 ADBCE。在图 4－37 所示的 AOV 网中，工程 B 和工程 C 显然可以同时进行，先后无所谓；但工程 E 却要等工程 B、C、D 都完成以后才能进行。

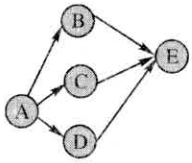


图 4－37

构造拓扑序列可以帮助我们合理安排一个工程的进度，由 AOV 网构造拓扑序列具有很高的实际应用价值。

算法思想：

构造拓扑序列的拓扑排序算法思想很简单：

(1) 选择一个入度为 0 的顶点并输出；

(2) 然后从 AOV 网中删除此顶点及以此顶点为起点的所有关联边；

(3) 重复上述两步，直到不存在入度为 0 的顶点为止；

(4) 若输出的顶点数小于 AOV 网中的顶点数，则输出“有回路信息”，否则输出的顶点序列就是一种拓扑序列。

从第四步可以看出，拓扑排序可以用来判断一个有向图是否有环。只有有向无环图才存在拓扑序列。

算法实现：

(a) 数据结构：　indgr[i]：顶点 i 的入度；

stack[]：栈

(b) 初始化：top=0（栈顶指针置零）

(c) 将初始状态所有入度为 0 的顶点压栈

(d) I=0（计数器）

(e) while 栈非空(top>0)

i. 栈顶的顶点 v 出栈；top－1；输出 v；i=i+1；

ii. for v 的每一个后继顶点 u

①indgr[u]－－；u 的入度减 1

②if indgr[u]==0　顶点 u 入栈

(f) 算法结束

这个程序采用栈来找出入度为 0 的点，栈里的顶点都是入度为 0 的点。

我们结合图 4－38 详细讲解：

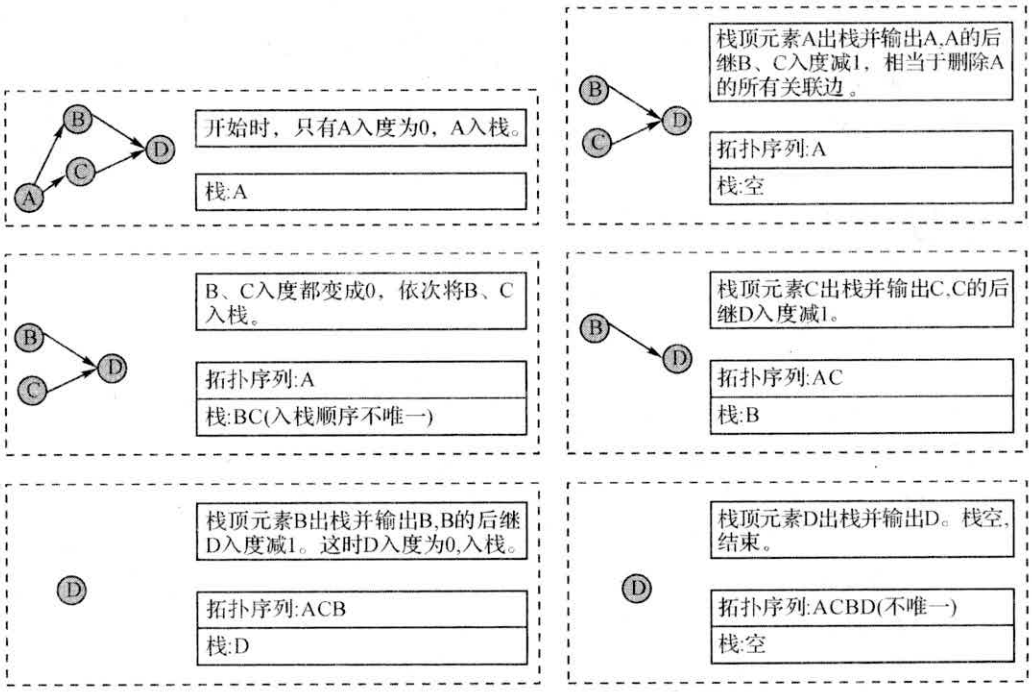


图 4－38

简单 & 高效 & 实用的算法。上述实现方法复杂度 O(V+E)。

例 4.14 家谱树

【问题描述】

有个人的家族很大,辈分关系很混乱,请你帮整理一下这种关系。

给出每个人的孩子的信息。

输出一个序列,使得每个人的后辈都比那个人后列出。

【输入格式】

第 1 行一个整数 N(1<=N<=100),表示家族的人数。

接下来 N 行,第 I 行描述第 I 个人的儿子。

每行最后是 0 表示描述完毕。

【输出格式】

输出一个序列,使得每个人的后辈都比那个人后列出。

如果有多解输出任意一解。

【输入样例】

```
5
0
4 5 1 0
1 0
5 3 0
3 0
```

【输出样例】

```
2 4 5 3 1
```

【参考程序】

```
#include<cstdio>
#include<iostream>
using namespace std;
int a[101][101];
int c[101];
int r[101];
int ans[101];
int i,j,tot,temp,num,n,m;
int main()
{
  cin >> n;
  for (i=1; i <=n; i++)
  {
    do
    {
      cin >> j;
      if (j! =0)
      {
        c[i]++;          //c[i]用来存点 i 的出度
```

```
            a[i][c[i]]=j;
            r[j]++;          //r[j]用来存点 i 的入度
        }
    }
    while (j! =0);
}
for (i=1; i <=n; i++)
    if (r[i]==0)
        ans[++tot]=i; //把图中所有入度为 0 的点入栈,栈用一维数组 ans[]表示
do
{
    temp=ans[tot];
    cout << temp << " ";
    tot--;num++;//栈顶元素出栈并输出
    for (i=1; i <=c[temp]; i++)
    {
        r[a[temp][i]]--;
        if (r[a[temp][i]]==0) //如果入度减 1 后变成 0,则将这个后继点入栈
            ans[++tot]=a[temp][i];
    }
}
while (num! =n);      //如果输出的点的数目 num 等于 n,说明算法结束
return 0;
}
```

例 4.15　奖金

【问题描述】

由于无敌的凡凡在 2005 年世界英俊帅气男总决选中胜出，Yali Company 总经理 Mr. Z 心情好，决定给每位员工发奖金。公司决定以每个人本年在公司的贡献为标准来计算他们得到奖金的多少。

于是 Mr. Z 下令召开 m 方会谈。每位参加会谈的代表提出了自己的意见："我认为员工 a 的奖金应该比 b 高！"Mr. Z 决定要找出一种奖金方案，满足各位代表的意见，且同时使得总奖金数最少。每位员工奖金最少为 100 元。

【输入格式】

第 1 行两个整数 n、m，表示员工总数和代表数；

以下 m 行，每行 2 个整数 a、b，表示某个代表认为第 a 号员工奖金应该比第 b 号员工高。

【输出格式】

若无法找到合理方案，则输出"Poor Xed"；否则输出一个数表示最少总奖金。

【输入样例】　　【输出样例】

2 1　　201

1 2

【数据范围】

80%的数据满足:n<=1000,m<=2000;

100%的数据满足:n<=10000,m<=20000。

【算法分析】

首先构图,若存在条件"a 的钱比 b 多",则从 b 引一条有向指向 a;然后拓扑排序,若无法完成排序,则表示问题无解(存在圈);若可以得到完整的拓扑序列,则按序列顺序进行递推。

设 f[i]表示第 i 个人能拿的最少奖金数;

首先所有 f[i]=100(题目中给定的最小值);

然后按照拓扑顺序考察每个点 i,若存在有向边(j,i),则表示 f[i]必须比 f[j]大,因此我们令 f[i]=Max{f[i],f[j]+1}即可;

递推完成之后所有 f[i]的值也就确定了,而答案就等于 f[1]+…+f[n]。

【参考程序】

```
#include<iostream>
using namespace std;
int a[10001][301]={0},into[10001],ans[10001],m,n,money;
void init()                          //读入数据,并构建图,统计入度
{ int i,x,y;
  cin>>n>>m;
  for(i=1;i<=m;i++)
  {
    cin>>x>>y;
    a[y][0]++;                       //记录由 y 引出边的数目
    a[y][a[y][0]]=x;                 //记录由 a[y][0]引出边的顶点
    into[x]++;                       //统计入度
  }
}
bool topsort()                       //拓扑排序
{ int t,tot,k,i,j;
  tot=0;k=0;
  while(tot<n)                       //tot 顶点个数
  {
    t=0;                             //用来判断有无环
    for(i=1;i<=n;i++)
      if(into[i]==0)
      {
        tot++;t++;money+=100;
        ans[t]=i;
        into[i]=0xfffffff;
```

```
        }
        if(t==0)return false;          //存在环
        money+=k*t;k++;
        for(i=1;i<=t;i++)              //去掉相连的边
            for(j=1;j<=a[ans[i]][0];j++)into[a[ans[i]][j]]--;
        }
        return true;
}
int main()
{
        init();money=0;
        if(topsort())cout<<money<<endl;
            else cout<<" Poor Xed "<<endl;
        return 0;
}
```

三、关键路径

利用 AOV 网络，对其进行拓扑排序能对工程中活动的先后顺序作出安排。但一个活动的完成总需要一定的时间，为了能估算出某个活动的开始时间，找出那些影响工程完成时间的最主要的活动，我们可以利用带权的有向网，图中的边表示活动，边上的权表示完成该活动所需要的时间，一条边的两个顶点分别表示活动的开始事件和结束事件，这种用边表示活动的网络，称为“AOE 网”。

其中，有两个特殊的顶点（事件），分别称为源点和汇点，源点表示整个工程的开始，通常令第一个事件（事件 1）作为源点，它只有出边没有入边；汇点表示整个工程的结束，通常令最后一个事件（事件 n）作为汇点，它只有入边没有出边；其余事件的编号为 2 到 n－1。

在实际应用中，AOE 网应该是没有回路的，并且存在唯一的入度为 0 的源点和唯一的出度为 0 的汇点。

图 4－39 表示一个具有 12 个活动的 AOE 网。图中有 8 个顶点，分别表示事件 0 到 7，其中，0 表示开始事件，7 表示结束事件，边上的权表示完成该活动所需要的时间。

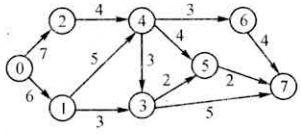


图 4－39

AOE 网络要研究的问题是完成整个工程至少需要多少时间？哪些活动是影响工程进度的关键？

下面先讨论一个事件的最早发生时间和一个活动的最早开始时间。如图 4－40，事件 V_j 必须在它的所有入边活动 $e_{ik}(1<=k<=n)$ 都完成后才能发生。活动 $e_{ik}(1<=k<=n)$ 的最早开始时间是与它对应的起点事件 V_{ik} 的最早发生时间。所有以事件 V_j 为起点事件的

出边活动 e_{jk}(1<=k<=m)的最早开始时间都等于事件 V_j 的最早发生时间。所以,我们可以从源点出发按照上述方法,求出所有事件的最早发生时间。

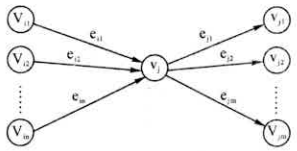


图 4-40

设数组 earliest[1..n]表示所有事件的最早发生时间,则我们可以按照拓扑顺序依次计算出 earliest[k]:

earliest[1]=0

earliest[k]=max{earliest[j]+dut[j][k]}

(其中,事件 j 是事件 k 的直接前驱事件,dut[j][k]表示边<j,k>上的权)

对于上图,用上述方法求 earliest[0..7]的过程如下:

earliest[0]=0

earliest[1]=earliest[0]+dut[0][1]=0+6=6

earliest[2]=earliest[0]+dut[0][2]=0+7=7

earliest[4]=max{earliest[1]+dut[1][4],earliest[2]+dut[2][4]}

=max{6+5,7+4}

=11

earliest[3]=max{earliest[1]+dut[1][3],earliest[4]+dut[4][3]}

=max{6+3,11+3}

=14

earliest[5]=max{earliest[3]+dut[3][5],earliest[4]+dut[4][5]}

=max{14+2,11+4}

=16

earliest[6]=earliest[4]+dut[4][6]=11+3=14

earliest[7]=max{earliest[3]+dut[3][7],earliest[5]+dut[5][7], earliest[6]+dut[6][7]}

=max{14+5,16+2,14+4}

=19

最后得到的 earliest[7]就是汇点的最早发生时间,从而可知整个工程至少需要 19 天完成。

但是,在不影响整个工程按时完成的前提下,一些事件可以不在最早发生时间发生,而向后推迟一段时间,我们把事件最晚必须发生的时间称为该事件的最迟发生时间。同样,有些活动也可以推迟一段时间完成而不影响整个工程的完成,我们把活动最晚必须开始的时间称为该活动的最迟开始时间。一个事件在最迟发生时间内仍没发生,或一个活动在最迟开始时间内仍没开始,则必然会影响整个工程的按时完成。事件 V_j 的最迟发生时间应该为:它的所有直接后继事件 V_{jk}(1<=k<=m)的最迟发生时间减去相应边<V_j,V_{jk}>上的权(活动 e_{jk}需要时间),取其中的最小值,且汇点的最迟发生时间就是它的最早发生时间,再按照逆拓扑顺序依次计算出所有事件的最迟发生时间,设用数组 lastest[1..n]表示,即:

lastest[n]=earliest[n]

lastest[j]=min{lastest[k]-dut[j][k]}

（其中，事件 k 是事件 j 的直接后继事件，dut[j,k]表示边<j,k>上的权）

对于上图，用上述方法求 lastest [0..7]的过程如下：

lastest[7]=earliest[7]=19

lastest[6]=lastest[7]-dut[6][7]=19-4=15

lastest[5]=lastest[7]-dut[5][7]=19-2=17

lastest[3]=min{lastest[5]-dut[3][5],lastest[7]-dut[3][7]}

=min{17-2,19-5}

=14

lastest[4]=min{lastest[3]-dut[4][3],lastest[5]-dut[4][5], lastest[6]

-dut[4][6]}

=min{14-3,17-4,15-3}

=11

lastest[2]=lastest[4]-dut[2][4]=11-4=7

lastest[1]=min{lastest[3]-dut[1][3],lastest[4]-dut[1][4]}

=min{14-3,11-5}

=6

lastest[0]=min{lastest[1]-dut[0][1],lastest[2]-dut[0][2]}

=min{6-6,7-7}

=0

计算好每个事件的最早和最迟发生时间后，我们可以很容易地算出每个活动的最早和最迟开始时间，假设分别用 actearliest 和 actlastest 数组表示，设活动 i 的两端事件分别为事件 j 和事件 k，如图 4-41 所示：

事件 j $\xrightarrow{\text{活动 i}}$ 事件 k

图 4-41

则：actearliest[i]=earliest[j]

actlastest[i]=lastest[k]-dut[j][k]

对于图 4-39，用上述方法求得所有活动的最早和最迟开始时间如下表：

活动	<0,1>	<0,2>	<1,3>	<1,4>	<2,4>	<3,5>	<3,7>	<4,3>	<4,5>	<4,6>	<5,7>	<6,7>
最早	0	0	6	6	7	14	14	11	11	11	16	14
最迟	0	0	11	6	7	15	14	11	13	12	17	15
余量	0	0	5	0	0	1	0	0	2	1	1	1

上表中的余量（称为开始时间余量）是该活动的最迟开始时间减去最早开始时间，余量不等于 0 的活动表示该活动不一定要在最早开始时间时就进行，可以拖延一定的余量时间再进行，也不会影响整个工程的完成。而余量等于 0 的活动必须在最早开始时间时进行，而且在规定的工期内完成，否则将影响整个工程的完成。

我们把开始时间余量为 0 的活动称为“关键活动”，由关键活动所形成的从源点到汇点的每一条路径称为“关键路径”。

图 4－39 所示的 AOE 网的关键路径如图 4－42 所示。

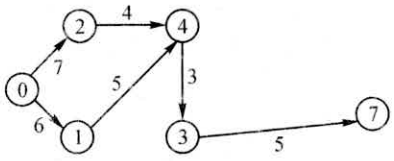


图 4－42

细心的读者可能已经发现，其实关键路径就是从源点到汇点具有最大路径长度的那些路径。这很容易理解，因为整个工程的工期就是按照最长路径计算出来的。很显然，要想缩短整个工程的工期，就应该想方设法去缩短关键活动的持续时间。读者可以根据上面的思想编程求出 AOE 网的关键路径。

【上机练习】

1. 烦人的幻灯片(slides)

【问题描述】

李教授将于今天下午作一次非常重要的演讲。不幸的事他不是一个爱整洁的人，他把自己演讲要用的幻灯片随便堆在了一起。因此，演讲之前他不得不去整理这些幻灯片。作为一个讲求效率的学者，他希望尽可能简单地完成它。教授这次演讲一共要用 n 张幻灯片(n<=26)，这 n 张幻灯片按照演讲要使用的顺序已经用数字 1～n 编了号。因为幻灯片是透明的，所以我们不能一下子看清每一个数字所对应的幻灯片。

现在我们用大写字母 A,B,C,…，再次把幻灯片依次编号。你的任务是编写一个程序，把幻灯片的数字编号和字母编号对应起来，显然这种对应应该是唯一的；若出现多种对应的情况或是某些数字编号和字母编号对应不起来，我们称对应是无法实现的。

【输入格式】

文件的第一行只有一个整数 n，表示有 n 张幻灯片，接下来的 n 行每行包括 4 个整数 xmin,xmax,ymin,ymax(整数之间用空格分开)为幻灯片的坐标，这 n 张幻灯片按其在文件中出现的顺序从前到后依次编号为 A,B,C,…，再接下来的 n 行依次为 n 个数字编号的坐标 x,y，显然在幻灯片之外是不会有数字的。

【输出格式】

若是对应可以实现，输出文件应该包括 n 行，每一行为一个字母和一个数字，中间以一个空格隔开，并且每行以字母的升序排列，注意输出的字母要大写并且定格；反之，若是对应无法实现，在文件的第一行顶格输出 None 即可。首行末无多余的空格。

【输入样例】

```
4
6 22 10 20
4 18  6 16
8 20  2 18
10 24 4  8
9 15
19 17
```

【输出样例】

```
A 4
B 1
C 2
D 3
```

11　7

21　11

2. 病毒(virus)

【问题描述】

有一天，小 y 突然发现自己的计算机感染了一种病毒！还好，小 y 发现这种病毒很弱，只是会把文档中的所有字母替换成其他字母，但并不改变顺序，也不会增加和删除字母。

现在怎么恢复原来的文档呢？小 y 很聪明，他在其他没有感染病毒的机器上，生成了一个由若干单词构成的字典，字典中的单词是按照字母顺序排列的，他把这个文件拷贝到自己的机器里，故意让它感染上病毒，他想利用这个字典文件原来的有序性，找到病毒替换字母的规律，再用来恢复其他文档。

现在你的任务是：告诉你被病毒感染了的字典，要你恢复一个字母串。

【输入格式】

第 1 行为整数 K(<=50000)，表示字典中的单词个数。

以下 K 行，是被病毒感染了的字典，每行一个单词。

最后一行是需要你恢复的一串字母。

所有字母均为小写。

【输出格式】

输出仅一行，为恢复后的一串字母。当然也有可能出现字典不完整、甚至字典是错的情况，这时请输出一个 0。

【输入样例】

6
cebdbac
cac
ecd
dca
aba
bac
cedab

【输出样例】

abcde

附录:C++常用库函数

头文件:cstring

函数原型:void * memcpy(void * dest,const void * src,sizet count);

功能:从 src 拷贝 count 个字节到 dest。

函数原型:void * memset(void * dest,int c,sizet count);

功能:设置 dest 的前 count 个字节为字符 c。

函数原型:int strcmp(const char * string1,constchar * string2)

功能:返回值<0,stringl 小于 string2;返回值=0,stringl 等于 string2;返回值>0,stringl 大于 string2。

函数原型:char * strcpy(char * strDestination,constcha * strSource)

功能:把源字符串 strSource(包括结尾的空字符)拷贝到 strDestination 所指的位置。

函数原型:size_t strlen(const char * string);

功能:返回 string 中的字符个数,不包括尾部 NULL。没有指出错误的返回值。

头文件:cctype

函数原型:int isalnum(int c);

功能:测试 c 是否字母或数字。

函数原型:int isalpha(int c);

功能:测试 c 是否字母。

函数原型:int islower(int c);

功能:测试是否小写字母。

函数原型:int isupper(int c);

功能:测试是否大写字母。

函数原型:int tolower(int c);

功能:将字符转换为小写字母。

函数原型:int toupper(int c);

功能:将字符转换为大写字母。

头文件:cmath

函数原型:int abs(int n);

功能:求绝对值。

函数原型:double acos(double x) ;
功能:计算并返回范围在 0 到 π 弧度之间的 x 的反余弦值。

函数原型:double asin(double x);
功能:计算并返回范围在 $-\pi/2$ 到 $\pi/2$ 弧度之间的 x 的反正弦值。

函数原型:double atan(double x);
double atan2(double y,double x);
功能:atan 返回 x 的反正切值,atan2 返回 y/x 的反正切值。

函数原型:double ceil(double x);
功能:对 x 向上取整,并以 double 型浮点数形式存储结果。

函数原型:double cos(double x);
功能:计算并返回 x 的余弦值(cos)。

函数原型:double exp(double x);
功能:计算并返回 e 的 x 次幂。

函数原型:double fabs(double x);
功能:计算并返回浮点参数 x 的绝对值。

函数原型:double floor(double x);
功能:向下取整,并以 double 型浮点数形式存储结果。

函数原型:double log(double x);
功能:计算并返回 x 的自然对数。

函数原型:double log10(double x);
功能:计算并返回 x 的以 10 为底的对数。

函数原型:double pow(double x,double y);
功能:计算并返回 x 的 y 次幂。

函数原型:double sin(double x);
功能:sin 返回 x 的正弦值。

函数原型:double sqrt(double x);
功能:计算并返回 x 的平方根。

函数原型:double tan(double x);
功能:tan 返回 x 的正切值。

头文件:iostream
函数原型:type max (type a,type b);
功能:比较 a 和 b 并返回其中较大者。

函数原型:type min (type a,type b);
功能:比较 a 和 b 并返回其中较小者。

头文件:cstdio
函数原型:int getchar(void);
功能:从 stdin 读取一个字符并返回所读字符,当出现读错误或遇到文件结尾时返回 EOF。

函数原型:char * gets(char * buffer);
功能:从标准输入流读取一行,并存储在 buffer 中。读入换行符但没有保存换行符。

函数原型:int putchar(int c);
功能:写一个字符到 stdout 中。

函数原型:int puts(const char * string);
功能:将 string 写到标准输出流 stdout,用换行符('\n')代替字符串的结尾的空字符('\0')。

函数原型:int sprintf(char * buffer,const char * format[, Argument] ...);
功能:将数据格式化后写到字符串中。

函数原型:int sscanf(const char * buffer,const char * format[, Argument] ...);
功能:由 buffer 读取字符数据并转换后存储到每个 argument 指定的位置中。

(详细库函数请见光盘 pdf 文件)

图书在版编目(CIP)数据

信息学奥赛一本通：C++版 / 董永建著. — 北京：科学技术文献出版社，2013.6(2017.3 重印)

ISBN 978-7-5023-7988-9

Ⅰ. ①信… Ⅱ. ①董… Ⅲ. ①C 语言—程序设计—中小学—教学参考资料… Ⅳ. ①G634.673

中国版本图书馆 CIP 数据核字(2013)第 124220 号

信息学奥赛一本通（C++版）

策划编辑：钟海丰　　责任编辑：周国臻　　责任出版：张志平

出版者	科学技术文献出版社
地址	北京市复兴路 15 号　邮编 100038
编务部	(010)58882938,58882087(传真)
发行部	(010)58882868,58882874(传真)
邮购部	(010)58882873
官方网址	www.stdp.com.cn
发行者	科学技术文献出版社发行　全国各地新华书店经销
印刷者	杭州华艺印刷有限公司
版次	2013 年 6 月第 1 版　2017 年 3 月第 5 次印刷
开本	787×1092　1/16
字数	832 千
印张	33.75
书号	ISBN 978-7-5023-7988-9
定价	50.00 元

版权所有　违法必究

购买本社图书，凡字迹不清、缺页、倒页、脱页者，本社发行部负责调换