CCF中学生计算机程序设计教材

CCF
中学生计算机程序设计

基础篇

中国计算机学会◎组编
江 涛 宋新波 朱全民◎主编

浅显的描述揭示深刻的内涵

信息学国际金牌教练
教你如何编程

科学出版社

CCF 中学生计算机程序设计教材

CCF 中学生计算机程序设计基础篇

中国计算机学会　组编

江　涛　宋新波　朱全民　主编

科学出版社

北京

内 容 简 介

本书由CCF组织富有程序设计教学经验的中学老师编写。通过详实的例题，循序渐进地介绍中学生计算机程序设计的各种知识，内容包括模块化编程、字符串处理、数据类型的组合、指针、数据外部存储、数据结构及其应用、简单算法、数学在程序设计中的应用、STL（标准模块库）简要说明等，旨在普及计算机科学教育，培养中学生的计算思维能力。

本书可作为中学生计算机程序设计教材，也可供广大计算机编程爱好者参考。

图书在版编目（CIP）数据

CCF中学生计算机程序设计基础篇 / 中国计算机学会组编；江涛，宋新波，朱全民主编. — 北京：科学出版社，2016.11

（CCF中学生计算机程序设计教材）

ISBN 978-7-03-050029-8

Ⅰ. C… Ⅱ. ①中… ②江… ③宋… ④朱… Ⅲ. ①程序设计-教材 Ⅳ. ①G634.671

中国版本图书馆CIP数据核字（2016）第232486号

责任编辑：杨 凯 / 责任制作：魏 谨
责任印制：张 倩 / 封面制作：杨安安

北京东方科龙图文有限公司 制作

http://www.okbook.com.cn

科学出版社出版

北京东黄城根北街16号

邮政编码：100717

http://www.sciencep.com

天津新科印刷有限公司 印刷

科学出版社发行 各地新华书店经销

*

2016年11月第 一 版 开本：720×1000 1/16

2017年1月第三次印刷 印张：16

印数：8 001 — 10 000 字数：315 000

定价：36.00元

（如有印装质量问题，我社负责调换）

CCF 中学生计算机程序设计教材
编委会名单

序

由中国计算机学会（CCF）组编的“CCF 中学生计算机程序设计教材”面世了。

早在 1984 年，邓小平就提出“计算机的普及要从娃娃抓起”。这很有先见之明，但这里的“计算机普及”是泛指，并未明确普及哪些内容。在过去的三十多年中，中小学广泛开展了计算机普及活动，2000 年教育部也曾发文，要在全国中小学开展信息技术教育。但事实上，现有的所谓“普及”大多成了对计算机工具的认识，而不是对中小学生智力的开发和思维的训练，因而效果不佳。CCF 早在 1984 年就创办了“青少年信息学奥林匹克竞赛 NOI”，这是面向那些学有余力的中学生的一项计算机科学（CS）教育活动，但具备开展这项活动的学校并不很多，每年参加 NOI 联赛的学生不过七八万，比例很小，因而普及的面并不大。

计算机科学教育的核心是算法设计和编程，这要求学生面对一个给定的现实问题要能够找到一个正确和高效的办法（算法）并将其变成计算机能理解的语言（程序设计语言），进而让计算机计算出人们需要的结果来。像快递员最佳路径算法就是一个典型的现实问题。这个过程并不容易，因为将一个问题抽象并构造一个模型，需要一定的数学基础，还得理解计算机的特点，“指挥”计算机干活。这还涉及欲求解问题的“可计算性”，因为并不是任何问题都可以由计算机求解的。计算机也并不知道什么是“问题”，是人告诉计算机，如何按照一步一步的程序求解。这个过程，就会训练一个人求解问题的能力，相应地，其具备的让计算机做事的思维能力称之为“计算思维”（Computational Thinking）。我们平常操作计算机（包括手机这些终端设备）仅仅像开关电灯那样简单，并不会使我们具备计算思维能力，而只有通过上述步骤才能训练这样的能力。随着计算机和网络的发展，未来越来越多的工作将和计算（机）有关（据美国政府的统计是 51% 以上）。我们必须知道如何让计算机做事，起码知道计算机是如何做事的，这就要求普及计算机科学教育（注意：不是计算机教育，也不是信息技术教育）。

美国政府已经把在中小学普及计算机科学当成一种国策（CS for All，每一个人学习计算机科学），并投入40亿美元落实这一项目。奥巴马总统说“在新经济形态中，计算机科学已不再是可选技能，而是同阅读、写作和算术一样的基础技能……因此，我制定了一项计划，以确保所有孩子都有机会学习计算机科学。”美国政府已明确把计算机教育列入（从幼儿园到12年级）教育体系K12中。英国从2014年起，对中小学的计算机课程进行重大改革，5岁的儿童就开始学写程序。英国教育部启动了“计算机在学校”（Computing at School，CAS）项目。新西兰等国也把计算机编程课当作中学的必修课，并为此投入资金培训教师。未来的竞争不是资源的竞争，而是人才的竞争，如果不具有计算素养和技能，则在未来的社会中处于被动地位。

CCF作为一个负责任的学术社会组织，应该勇于承担起CS普及的任务，这比NOI更加艰巨，更难。不过有NOI三十多年发展的基础，会对未来CS的普及提供有益的经验。

普及计算机科学教育的难点在于师资，而培训师资需要合适的教材。CCF组织富有程序设计教学经验的中学老师编写了“CCF中学生计算机程序设计教材”，分为入门篇、基础篇、提高篇和专业篇，只要有一定数学基础的老师，均可从入门篇修起。学习编程并不像人们想象的那么困难，只要从现实中遇到的（简单）问题出发，循序渐进，通过和计算机的互动，一旦入门就好办了，以后就可以逐步深入下去。

感谢朱全民、陈颖、徐先友、江涛、邱桂香、宋新波、汪星明、屈运华、廖晓刚等老师的贡献，他们花了两年时间写成了这套教材。感谢吴文虎教授、王宏博士审阅本书，在此向他们表示感谢。

杜子德

中国计算机学会秘书长

2016年8月29日

目 录

第1章　模块化编程——函数

我们处理任何问题都有诸多环节，比如煮饭，先要准备米，将米淘净后，然后根据米的吸水特性加水，弄好后放入电饭煲中，按下电饭煲的煮饭开关，等电饭煲灯熄灭，然后饭就做好了。现在煮饭看起来变得十分简单，其关键原因是煮饭的技术活都被电饭煲干了。电饭煲煮饭是如何控制火候和时间的，其实我们并不关心，我们所关心的是用电饭煲将饭煮熟的结果，这个方法我们叫封装。在计算机程序设计中，封装是一个非常重要的概念，它是实现程序模块化结构的重要手段。在C++中，我们通常采用函数来进行模块封装，对于函数，我们所关心的是对给定的自变量输入，能否得到我们想要的输出。有些函数系统已经做好了，可直接调用，比如诸多的数学函数、字符串处理函数等，有些函数则需要根据自己的需求进行个性开发，这章我们就重点讲授如何创作自己的函数。

1.1　自定义函数的引入

【例1.1】给出平面上两个点的坐标，求两点之间的曼哈顿距离。

提示： 平面上A点(x1,y1)与B点(x2,y2)的曼哈顿距离为：|x1-x2|+|y1-y2|。

分析： 计算曼哈顿距离可以用前面学过的分支结构来解决。

程序如下：

```
1  //eg1.1_1
2  #include<iostream>
3  using namespace std;
4  int main()
5  {
6     double x1,y1,x2,y2;
7     double dx,dy;
8     cin>>x1>>y1>>x2>>y2;
9     if(x1>x2)              //计算|x1-x2|
```

运行结果：
输入：1.2 1.5 2.5 3.8
输出：3.6

```
10        dx=x1-x2;
11      else
12        dx=x2-x1;
13      if(y1>y2)                    // 计算 |y1-y2|
14        dy=y1-y2;
15      else
16        dy=y2-y1;
17      cout<<dx+dy<<endl;
18      return 0;
19  }
```

可以使用自定义函数来计算|x1-x2|，|y1-y2|。

程序如下：

```
1   //eg1.1_2
2   #include<iostream>
3   using namespace std;
4   double abs(double x)          // 计算 x 的绝对值函数
5   {
6       if(x>0)
7          return x;
8       else
9          return -x;
10  }
11  int main()
12  {
13      double x1,y1,x2,y2;
14      double dx,dy,mht;
15      cin>>x1>>y1>>x2>>y2;
16      mht=abs(x1-x2)+abs(y1-y2);
17      cout<<mht<<endl;
18      return 0;
19  }
```

比较上面两个程序容易发现，使用自定义函数的程序有以下几个优点：

（1）程序结构清晰，逻辑关系明确，程序可读性强。

（2）解决相同或相似问题时不用重复编写代码，可通过调用函数来解决，减少代码量。

（3）利用函数实现模块化编程，各模块功能相对独立，利用“各个击破”降低调试难度。

1.2 函数的定义

前面我们用过了很多C++标准函数，但是这些标准函数并不能满足所有需求。当我们需要特定的功能函数时，这就需要我们要学会自定义函数，根据需求定制想要的功能。

1.2.1 函数定义的语法

```
返回类型  函数名(参数列表)
{
    函数体
}
```

关于函数定义的几点说明：

（1）自定义函数符合“根据已知计算未知”这一机制，参数列表相当于已知，是自变量，函数名相当于未知，是因变量。如程序eg1.1_2中的abs函数的功能是根据给定的数x计算x的绝对值，参数列表中x相当于已知——自变量，abs函数的值相当于未知——因变量。

（2）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命名。

（3）参数列表可以是空的，即无参函数，也可以有多个参数，参数之间用逗号隔开，不管有没有参数，函数名后的括号不能省略。参数列表中的每个参数，由参数类型说明和参数名组成。如程序eg1.1_2中abs函数的参数列表只有一个参数，参数数据类型是double，参数名是x。

（4）函数体是实现函数功能的语句，除了返回类型是void的函数，其他函数的函数体中至少有一条语句是“return表达式；”用来返回函数的值。执行函数过程中碰到return语句，将在执行完return语句后直接退出函数，不去执行后面的语句。

（5）返回值的类型一般是前面介绍过的int、double、char等类型，也可以是数组。有时函数不需要返回任何值，例如函数可以只描述一些过程用printf向屏幕输出一些内容，这时只需定义函数返回类型为void，并且无须使用return返回函数的值。

1.2.2 函数定义应用实例

根据上述定义，我们知道C++函数形态有以下四类：

（1）返回类型 函数名(参数列表)。

（2）返回类型 函数名( )。

（3）void函数名（参数列表）。

（4）void函数名( )。

下面我们一起来看几个例子：

【例1.2】给定两个非负整数n和m，编写函数计算组合数C_n^m。

分析：首先分析函数的功能，根据给定的n,m计算C_n^m。n,m已知，相当于自变量；C_n^m未知，相当于因变量。设计以下函数：

```
long long C(int n,int m)
```

其中，函数的返回值为C_n^m，返回类型为long long，函数名为C，参数列表中有两个参数n,m，类型都是int。函数体是实现函数功能的语句，根据$C_n^m = \frac{n!}{m! * (n-m)!}$发现需要三次用到“计算一个数的阶乘”这个功能，因此把这个功能独立出来设计一个函数来实现：

```
long long f(int n)
```

该函数的返回值为n!，返回类型为long long，函数名为f，需要一个参数n，类型为int。

综上，该函数的代码如下：

```
1  //eg1.2
2  long long f(int n)
3  {
4     long long ans=1;
5     for(int i=1;i<=n;i++)
6        ans*=i;
7     return ans;
8  }
9
10 long long C(int n,int m)
11 {
12    return f(n)/(f(m)*f(n-m));
13 }
```

提示：

（1）函数体中的语句可以是对另一个函数的调用。

（2）对于较大的n,m来说，上述程序可能会产生溢出。

【例 1.3】编写函数输出斐波那契数列的第 n 项。其中斐波那契数列 f(n)的定义如下：

$$f(1)=0,f(2)=1$$

$$f(n)=f(n-1)+f(n-2)(n>=2)$$

分析：因为该函数不需要返回值，只需要输出 f(n)即可，所以该函数的返回类型为 void，函数体部分只需计算出 f(n)再输出即可。

函数代码如下：

```
1  //eg1.3
2  void output(int n)
3  {
4     if(n<=2)
5        cout<<n-1<<endl;
6     else
7     {
8        long long p1,p2,p3;
9        p1=0;
10       p2=1;
11       for(int i=3;i<=n;i++)
12       {
13         p3=p1+p2;
14         p1=p2;
15         p2=p3;
16       }
17       cout<<p3<<endl;
18    }
19 }
```

1.3 函数调用与参数传递

上一节学习了函数的定义方法和四种不同类型的函数，在实际编程中如何调用函数呢？调用函数时参数是如何传递的？参数传递又有几种方法呢？本节将重点学习这些内容。

1.3.1 函数的调用

1. 调用方法

上一节中讲过函数一共有四种不同的类型，也可以根据返回类型分为

两大类：其中一类有返回值，如程序eg1.1_2中的abs函数、例1.2中的f函数和C函数；另一类没有返回值，如例1.3中的output函数。

对于有返回值的函数，调用时必须以值的形式出现在表达式中。比如程序eg1.1_2第16行：

```
mht=abs(x1-x2)+abs(y1-y2);
```

该语句对abs函数的调用出现在赋值语句的右边，程序eg1.2第12行：

```
return f(n)/(f(m)*f(n-m));
```

该语句对f函数的调用出现在return语句中，作为C函数的返回值。

对于没有返回值的函数，直接写“函数名(参数);”即可。如例1.3中，如果需要输出斐波那契数列的第10项，用“output(10);”即可实现。

程序可以调用任何前面已经定义的函数，如果我们需要调用在后面定义的函数，就要先声明该被调用的函数。声明方法如下：

```
返回类型  函数名(参数列表);
```

程序如下：

```
1  #include<iostream>
2  using namespace std ;
3  int sgn(int n);                          //声明sgn函数
4  int main()
5  {
6    int n;
7    cin>>n;
8    cout<<sgn(n)<<endl;
9    return 0;
10 }
11 int sgn(int n)
12 {
13   if (n==0) return 0;
14   return (n>0)?1:-1;
15 }
```

2. 形式参数与实际参数

函数调用需要理解形式参数与实际参数：

（1）函数定义中的参数名称为形式参数，如long long C(int n,int m)中的n和m是形式参数，我们完全可以把n换成a,把m换成b，再把函数体中的n换成a,m换成b,函数的功能完全一样。

（2）实际参数是指实际调用函数时传递给函数的参数的值。如调用函

数C(6,3)，这里的6,3就是实际参数，其中6传递给形式参数n，3传递给形式参数m。

3. 调用函数的执行过程

调用函数的执行过程如下：

（1）计算实际参数的值。如程序eg1.1_2中输入x1=1.2，y1=1.5，x2=2.5，y2=3.8，程序第16行调用函数abs(x1-x2)和abs(y1-y2)，首先计算参数x1-x2=-1.3，y1-y2=-2.3，这里的-1.3和-2.3就是实际参数。

（2）把实际参数传递给被调用函数的形式参数，程序执行跳到被调用的函数中。如程序eg1.1_2中，调用abs(x1-x2)时就会把-1.3赋值给函数定义中的形式参数x，然后执行abs(-1.3)。

（3）执行函数体，执行完后如果有返回值，则把返回值返回给调用该函数的地方继续执行。如上面的例子中，就会在执行完abs(-1.3)后返回1.3，再执行abs(-2.3)后返回2.3，接着就可以计算出abs(x1-x2)+abs(y1-y2)的值等于1.3+2.3=3.6，再赋值给左边的变量mht。

1.3.2 参数传递

1. 传值参数

函数通过参数来传递输入数据，参数通过传值机制来实现。

前面的程序中的函数都采用了传值参数，采用的传递方式是值传递，函数在被调用时，用克隆实参的办法得到实参的副本传递给形参，改变函数形参的值并不会影响外部实参的值。

【例1.4】编写程序利用函数交换两个变量的值。

```
1  //eg1.4_1(错误)
2  #include<iostream>
3  using namespace std;
4  void swap(int a,int b)
5  {
6      int t=a;
7      a=b;
8      b=t;
9  }
10
11 int main()
```

运行结果：
输入：3 4
输出：3 4

```
12 {
13     int x,y;
14     cin>>x>>y;
15     swap(x,y);
16     cout<<x<<' '<<y<<endl;
17     return 0;
18 }
```

以上程序中虽然swap函数交换了a,b的值，但main中的x,y并没有交换，这是因为swap函数的参数是传值参数，swap函数在被调用时，传递给形参a,b的是实参x,y的副本，swap函数中形参a,b的变化只是交换了实参x,y的副本，而实参x,y并没有被交换。

2. 引用参数

函数定义时在变量类型符号之后形式参数名之前加“&”，则该参数就是引用参数，把参数声明成引用参数，实际上改变了缺省的按值传递参数的传递机制，引用参数会直接关联到其所绑定的对象，而并非这些对象的副本，形参就像是对应实参的别名，形参的变化会保留到实参中。

例1.4采用引用参数的程序如下：

```
1  //eg1.4_2
2  #include<iostream>
3  using namespace std;
4  void swap(int &a,int &b)
5  {
6      int t=a;
7      a=b;
8      b=t;
9  }
10 int main()
11 {
12     int x,y;
13     cin>>x>>y;
14     swap(x,y);
15     cout<<x<<' '<<y<<endl;
16     return 0;
17 }
```

运行结果：
输入：3 4
输出：4 3

说明：

（1）程序第4行void swap(int &a,int &b)声明为两个整型变量起了别名，一个叫a，一个叫b，谁叫a谁叫b，要看谁来调用。

（2）讲到变量要想到有一个内存地址与之联系，主程序在第12行用“int x,y;”定义x和y，之后系统为整型变量x和y分别安排了存储空间，其内存地址分别为&x和&y。x是内存地址&x的符号名，y是内存地址&y的符号名。这里&为取地址操作符。

（3）将第14行和第4行放到一起看实参传递给形参的过程相当于声明：a是x的引用(int &a = x;)，b是y的引用(int &b = y;)。或者说a成为变量x地址(&x)的符号名的别名，b成为变量y地址(&y)的符号名的别名。

经过引用之后，a和b替代了x和y。说得更直白些，系统向形参传送的是实参的地址&x和&y。在子函数中，a是&x的符号地址，b是&y的符号地址。子函数执行时，操作a和b就等同于操作x和y。

1.4 变量的作用域

作用域描述了名称在文件的多大范围内可见可使用。C++程序中的变量按作用域来分，有全局变量和局部变量。

1.4.1 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1.5】输入两个正整数，编程计算两个数的最小公倍数。

```
1  //eg1.5
2  #include<iostream>
3  using namespace std;
4  long long x,y;          //定义全局变量x,y
5  long long gcd(long long x,long long y)
6  {
7      long long r=x%y;
```

运行结果：
输入：12 18
输出：36

```
8       while (r!=0)
9       {
10          x=y;
11          y=r;
12          r=x%y;
13      }
14      return y;
15  }
16  long long lcm()
17  {
18      return x*y/gcd(x,y);          // 访问全局变量x,y
19  }
20  int main()
21  {
22      cin>>x>>y;                    // 访问全局变量x,y
23      cout<<lcm()<<endl;
24      return 0;
25  }
```

说明：

（1）程序eg1.5中第4行定义的变量x,y在所有花括号外部，具有全局作用域，是全局变量。

（2）全局变量的作用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时，可以将这个变量定义成全局变量，如程序eg1.5中的main函数和lcm函数都用到了全局变量x,y，使用非常方便，但副作用也不可低估。

（3）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。

（4）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。

（5）全局变量在程序执行的全过程中一直占用内存单元。

（6）全局变量在定义时若没有赋初值，其默认值为0。

1.4.2 局部变量

定义在函数内部作用域为局部的变量称为局部变量。函数的形参和在该函数里定义的变量都被称为该函数的局部变量，如程序eg1.5中的计算

最大公约数的gcd函数的形参x,y和函数里定义的变量r都是gcd函数的局部变量。

使用局部变量时有以下几点说明：

（1）局部变量只在块内可见，在块外无法访问，具有块作用域。

（2）不同函数的局部变量相互独立，不能访问其他函数的局部变量。

（3）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。

（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）C++中的作用域是可以嵌套的，定义在全局作用域中的变量可以在局部作用域中使用，而定义在全局或局部作用域中的变量可以在语句作用域中使用等，并且变量还可以在内部作用域中被重新定义。定义在内部作用域的名字会自动屏蔽定义在外部作用域的相同的名字，如程序eg1.5中gcd函数中的形参x,y与全局变量x,y重名，全局变量x,y在gcd函数中被屏蔽处于无效状态。当一个局部变量的作用域结束时，它对全局变量的屏蔽就会被取消。

1.5 函数的应用

前面学习了函数的定义、调用、参数传递和全局变量、局部变量等知识点，本节将通过几个例子来看看函数在实际问题中的应用。

【例1.6】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。

输入格式：一个正整数n。1<=n<=10000。

输出格式：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。

输入样例：

100

输出样例：

3 5

```
5 7
11 13
17 19
29 31
41 43
59 61
71 73
```

分析：从2到n-2枚举每个数i，判断i和i+2是否为素数。定义isprime(n)函数判断n是否为素数即可。

编写程序如下：

```
1  //eg1.6
2  #include<iostream>
3  #include<cmath>
4  using namespace std ;
5  int n ;
6  bool isprime(int n)
7  {
8      int m=ceil(sqrt(n));  //ceil是上取整函数，sqrt是开平方
                                根，这两个函数都来自库cmath
9      for(int i=2;i<=m;i++)if(n%i==0)return false;
10     return true;
11 }
12 int main()
13 {
14     cin>>n;
15     bool empty=1;
16     for(int i=2;i<=n-2;i++)
17     {
18     if(isprime(i)&&isprime(i+2))
19     {
20        empty=0;
21        cout<<i<<' '<< i+2 <<endl;
22     }
23     }
24     if(empty)cout<<"empty";
25     return 0;
26 }
```

【例 1.7】给定一个 m*n 的矩阵 A 和 r*s 的矩阵 B，其中 0<r<=m，0<s<=n，A、B 所有元素值都是小于 100 的正整数。求 A 中一个大小为 r*s 的子矩阵 C，使得 B 和 C 的对应元素差值的绝对值之和最小，这时称 C 为最匹配的矩阵。如果有多个子矩阵同时满足条件，选择子矩阵左上角元素行号小者，行号相同时，选择列号小者。

输入格式：第 1 行是 m 和 n，以一个空格分开；之后 m 行，每行有 n 个整数，表示 A 矩阵中的各行，数与数之间以一个空格分开；第 m+2 行为 r 和 s，以一个空格分开；之后 r 行每行有 s 个整数，表示 B 矩阵中的各行，数与数之间以一个空格分开。(1<=m<=100,1<=n<=100)

输出格式：输出矩阵 C，一共 r 行，每行 s 个整数，整数之间以一个空格分开。

输入样例：

```
3 3
3 4 5
5 3 4
8 2 4
2 2
7 3
4 9
```

输出样例：

```
4 5
3 4
```

分析： 采用**枚举法**。m*n 的矩阵 A 中有 (m-r+1)*(n-s+1) 个大小为 r*s 的子矩阵，子矩阵的左上角的行号范围在 0 到 m-r 之间，列号在 0 到 n-s 之间。枚举所有 r*s 的子矩阵，计算出每个子矩阵与矩阵 B 对应元素差的绝对值的和，记录并将最小的子矩阵输出来即可。时间复杂度为 (m-r+1)*(n-s+1)*r*s=O(m*n*r*s)。

编写程序时，定义以下几个函数：

（1）void input()：输入矩阵 A 和 B。

（2）void work(int &x,int &y)：枚举所有 r*s 的子矩阵，并计算出其与矩阵 B 对应元素之差的绝对值之和，记录最小的子矩阵的左上角的行号为 x，列号为 y。

（3）int compute(int x,int y)：计算左上角行号为 x、列号为 y 的 r*s

的子矩阵与矩阵B对应元素差的绝对值之和，在work函数中被调用。计算绝对值可以调用库函数abs，但必须要在程序头部加上#include<cstdlib>。

（4）void output(int x,int y)：输出左上角行为x、列为y的子矩阵。

程序eg1.7如下：

```
1  //eg1.7
2  #include<iostream>
3  #include<cstdlib>
4  using namespace std;
5  int m,n,r,s,a[100][100],b[100][100];
6  void input()
7  {
8     cin>>m>>n;
9     for(int i=0;i<m;i++)
10       for(int j=0;j<n;j++)
11          cin>>a[i][j];
12    cin>>r>>s;
13    for(int i=0;i<r;i++)
14       for(int j=0;j<s;j++)
15          cin>>b[i][j];
16 }
17 int compute(int x,int y)
18 {
19    int ans=0;
20    for(int i=0;i<r;i++)
21     for(int j=0;j<s;j++)
22       ans+=abs(a[x+i][y+j]-b[i][j]);
23   return ans;
24 }
25 void work(int &x,int &y)
26 {
27      int mindiff=1000001;
28      for(int i=0;i<=m-r;i++)
29         for(int j=0;j<=n-s;j++)
30         {
31             int curr=compute(i,j);
32             if(curr<mindiff)
33             {
34                 mindiff=curr;
35                 x=i;
```

```
36                  y=j;
37              }
38          }
39 }
40 void output(int x,int y)
41 {
42    for(int i=0;i<r;i++)
43    {
44      for(int j=0;j<s;j++) cout<<a[x+i][y+j]<<" ";
45      cout<<endl;
46    }
47 }
48 int main()
49 {
50      int x,y;
51      input();
52      work(x,y);
53      output(x,y);
54      return 0;
55 }
```

【例 1.8】图像旋转翻转变换。给定 m 行、n 列的图像各像素点灰度值，对其依次进行一系列操作后，求最终图像。其中，可能的操作有如下四种：

A：顺时针旋转 90 度。

B：逆时针旋转 90 度。

C：左右翻转。

D：上下翻转。

输入格式：第 1 行包含两个正整数 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；接下来 m 行，每行 n 个整数，表示图像中每个像素点的灰度值，相邻两个数之间用单个空格隔开，灰度值范围在 0 到 255 之间；接下来 1 行，包含由 A、B、C、D 组成的字符串 s，表示需要按顺序执行的操作序列。s 的长度在 1 到 100 之间。

输出格式：m'行，每行包含 n'个整数，为最终图像各像素点的灰度值，其中，m'为最终图像的行数，n'为最终图像的列数。相邻两个整数之间用单个空格隔开。

输入样例：

```
2 3
```

10 0 10
100 100 10
AC

输出样例：

10 100
0 100
10 10

分析：不难发现，我们设原m*n的矩阵某元素的坐标为(x,y)，x表示行号，y表示列号，那么在四种操作后矩阵规模和元素的坐标变换如下：

A：矩阵规模变为n*m，元素坐标(x,y)变为(y,m-x+1)。

B：矩阵规模变为n*m，元素坐标(x,y)变为(n-y+1,x)。

C：矩阵规模还是m*n，元素坐标(x,y)变为(x,n-y+1)。

D：矩阵规模还是m*n，元素坐标(x,y)变为(m-x+1,y)。

方法1：枚举每一个矩阵中的元素，按照每一个操作进行变换。定义一个函数f(a,x,y,g)计算出每次操作后的新坐标，其中，参数a是当前操作类型，(x,y)是原坐标，g表明函数该返回新的行号还是列号，0表示返回行号，1表示返回列号。

f函数是这个程序的核心，它让整个程序的可读性变好了，并且还将程序的逻辑更加清楚地呈现了出来。

时间复杂度为O(m*n*L)，L表示操作序列s的长度。

编写程序eg1.8_1如下：

```
1  //eg1.8_1
2  #include<iostream>
3  #include<string>
4  #include<algorithm>      //algorithm库中提供了swap函数
5  using namespace std;
6  const int N=110;
7  int n,m,a[N][N],b[N][N];
8  string s;
9  void input()
10 {
11   cin>>m>>n;
12   for(int i=1;i<=m;i++)
13   for(int j=1;j<=n;j++)cin>>a[i][j];
14   cin>>s;
```

```
15 }
16 int f(char a,int x,int y,int g){
17   if(a=='A') if(!g)return y; else return m-x+1;
18   if(a=='B') if(!g)return n-y+1; else return x;
19   if(a=='C') if(!g)return x; else return n-y+1;
20   if(!g) return m-x+1; else return y;
21 }
22 void work(){
23   for(int k=0;k<s.length();k++)
24   {
25      for(int i=1;i<=m;i++)
26        for(int j=1;j<=n;j++)
27          b[f(s[k],i,j,0)][f(s[k],i,j,1)]=a[i][j];
28      if(s[k]=='A'||s[k]=='B')swap(m,n);
29      for(int i=1;i<=m;i++)for(int j=1;j<=n;j++)a[i][j]=b[i][j];
30   }
31 }
32 void output(){
33  for(int i=1;i<=m;i++)
34   {
35     for(int j=1;j<=n;j++)cout << a[i][j] <<' ';
36     cout << endl ;
37   }
38 }
39 int main()
40 {
41   input();
42   work();
43   output();
44   return 0;
45 }
```

***方法2：**经过深入分析，可以发现两个重要的信息：

（1）每一次操作原矩阵的所有元素变化规则相同，不用每个元素都去更新，只需求出所有操作完成后元素的变化规则，所有元素套用这个规则即可。

（2）变化规则有规律可循：元素（x,y）在一次操作后新坐标为（x1,y1），其中，x1,y1最多只有x+delta，-x+delta，y+delta,-y+delta这四种情况，其中delta是与矩阵行数或列数相关的一个常数，两次操作后也满足，进

而发现无论操作多少次都满足这个特点。

程序实现：

（1）元素a[i][j]的原始坐标为(i,j)，每一次操作后新坐标为(i',j')，i'和j'一定是有一个与i有关，另外一个与j有关。

（2）这里定义变量flag，flag=0表示i'与i有关，j'与j有关，flag=1刚好相反。

（3）定义数组c[2][2]，功能如下：

- 当flag=0时，c[0][0]记录原始行号i在i'中的正负号，c[0][1]记录对应增量，c[1][0]记录原始列号j在j'中的正负号，c[1][1]记录列号对应的增量，即

  i'=c[0][0]*i+c[0][1]; j'=c[1][0]*j+c[1][1];

- 当flag=1时，新坐标(i', j')与原始坐标(i,j)的关系如下：

  i'=c[0][0]*j+c[0][1]; j'=c[1][0]*i+c[1][1];

（4）时间复杂度为O(n*m+L)，L表示操作次数。

编写程序eg1.8_2如下：

```
1  //eg1.8_2
2  #include<iostream>
3  #include<string>
4  #include<algorithm>
5  using namespace std;
6  const int N = 110;
7  int n,m,n1,m1,num,a[N][N],b[N][N],flag,c[2][2],d[2][2];
8  string s;
9  void input()
10 {
11   cin >> m >> n;
12   m1=m;n1=n;
13   for(int i=1;i<=m;i++)
14   for(int j=1;j<=n;j++)cin>>a[i][j];
15   cin >> s;
16 }
17 void f(char a)
18 {
19   d[0][0]=-c[0][0];
20   d[0][1]=m+1-c[0][1];
21   d[1][0]=-c[1][0];
```

```
22    d[1][1]=n+1-c[1][1];
23    switch (a)
24    {
25        case 'a':
26          flag=1-flag;
27          c[0][0]=c[1][0];
28          c[0][1]=c[1][1];
29          c[1][0]=d[0][0];
30          c[1][1]=d[0][1];
31          break;
32        case 'b':
33          flag=1-flag;
34          c[1][0]=c[0][0];
35          c[1][1]=c[0][1];
36          c[0][0]=d[1][0];
37          c[0][1]=d[1][1];
38          break;
39        case 'c':
40          c[1][0]=d[1][0];
41          c[1][1]=d[1][1];
42          break;
43        case 'd':
44          c[0][0]=d[0][0];
45          c[0][1]=d[0][1];
46          break;
47    }
48 }
49 void work()
50 {
51   flag=0;c[0][0]=c[1][0]=1;c[0][1]=c[1][1]=0; //初始化
52   for(int k=0;k<s.length();k++)
53   {
54     f(s[k]);
55     if(s[k]=='a'||s[k]=='b')swap(m,n);
56   }
57   for(int i=1;i<=m1;i++)
58       for(int j=1;j<=n1;j++)
59       {
60           int t[2]={i,j};
61           int x1=c[0][0]*t[flag]+c[0][1];
```

```
62            int y1=c[1][0]*t[1-flag]+c[1][1];
63            b[x1][y1]=a[i][j];
64        }
65 }
66 void output()
67 {
68   for(int i=1;i<=m;i++)
69   {
70     for(int j=1;j<=n;j++)cout << b[i][j] <<' ';
71     cout << endl ;
72   }
73 }
74 int main()
75 {
76   input();
77   work();
78   output();
79   return 0;
80 }
```

1.6 递归函数

1.6.1 生活中的递归

我们先来看看生活中的几个递归的现象。

现象一：你乘坐电梯时，前后各有一块镜子，你会发现你在镜子的成像数不过来。那是因为你在镜子A中的成像又在镜子B中成像，镜子B的成像又在镜子A中有成像，如此反复……

现象二：小明去找A经理解决问题，A经理说："这件事情不归我管，去找B经理。"于是小明去找B经理。B经理说："这件事情不归我管，去找A经理。"如果两个经理的说辞不变，小明又始终听话，小明将永远往返于两者之间。

现象三：老和尚讲故事"从前有座山，山里有座庙，庙里有个老和尚在讲故事：从前有座山，山里有座庙，庙里有个老和尚在讲故事：从前有座山，山里有座庙，庙里有个老和尚在讲故事……"

以上现象被称为递归现象，因为"你中有我，我中有你"、"自己直接或间接地用到了自己"。

1.6.2 递归函数的定义

我们把“内部操作直接或间接地调用了自己的函数”称为递归函数。

程序中的递归函数与生活中递归现象有相似之处，也有不同之处。相似之处在于都调用了自己，不同之处在于生活中有些递归现象是无限递归，递归函数有递归终止条件。

归纳起来，递归函数有两大要素：

（1）递归关系式：对问题进行递归形式的描述。

（2）递归终止条件：当满足该条件时以一种特殊情况处理，而不是用递归关系式来处理。

如阶乘函数f(n)=n!可以定义为递归函数：

$$\begin{cases} f(1)=1 & n=1\text{时} \\ f(n)=n*f(n-1) & n>1\text{时} \end{cases}$$

其中，f(1)为递归终止条件，f(n)=n*f(n-1)为递归关系式。

对应的程序如下：

```
1  #include<iostream>
2  using namespace std ;
3  int f(int n)
4  {
5    if (n==1) return 1;                    // 递归终止条件
6    else return n*f(n-1);                  // 递归关系式
7  }
8  int main()
9  {
10 int n;
11   cin>>n;
12   cout<<f(n)<<endl;
13   return 0;
14 }
```

运行结果：
输入：5
输出：120

【例1.9】给出正整数n，用递归法求斐波那契数列第n项模1000的值。

分析：定义递归函数fib(n)表示斐波那契数列第n项的值，列出该递归函数的两大要素：

（1）递归终止条件：fib(0)=0,fib(1)=1。

（2）递归关系式：fib(n)=fib(n-1)+fib(n-2)(n>=2)。

对应的程序eg1.9如下：

```
1  //eg1.9
2  #include<iostream>
3  using namespace std;
4  int n;
5  int fib( int n)
6  {
7      if(n<= 1) return n;                    //递归终止条件
8      return(fib(n-1) + fib(n-2))%1000;      //递归关系式
9  }
10 int main()
11 {
12     cin >> n ;
13     cout <<fib(n) << endl;
14     return 0;
15 }
```

运行结果：
输入：4
输出：3

1.6.3 递归的执行过程

程序在执行函数A的函数体的过程中又调用了一个函数B，此时当前执行的位置我们称之为“当前代码行”，由于要跳出去执行函数B，此处形成一个断点，当函数B执行完后会自动回到断点处继续执行函数A。

为了更好地理解该过程，我们需要简单介绍一下调用栈。

调用栈描述的是函数之间的调用关系。它由多个栈帧组成，每个栈帧对应着一个未运行完的函数。栈帧中保存了该函数的返回地址和局部变量，不同的函数对应着不同的栈帧，因而不仅能在执行完毕后找到正确的返回地址，还很自然地保证了不同函数间的局部变量互不相干。以例1.9中n=4为例，我们来看看程序执行的顺序（表1.1）。

表1.1 程序执行的顺序

步骤	层数	当前函数	操 作
1	0	main()	调用fib(4)
2	1	fib(4)	调用fib(3)
3	2	fib(3)	调用fib(2)
4	3	fib(2)	调用fib(1)
5	4	fib(1)	返回1，回到第3层
6	3	fib(2)	调用fib(0)
7	4	fib(0)	返回0，回到第3层

续表

步骤	层数	当前函数	操 作
8	3	fib(2)	计算出fib(2)=1+0=1,返回1，回到第2层
9	2	fib(3)	调用fib(1)
10	3	fib(1)	返回1，回到第2层
11	2	fib(3)	计算出fib(3)=1+1=2,返回2，回到第1层
12	1	fib(4)	调用fib(2)
13	2	fib(2)	调用fib(1)
14	3	fib(1)	返回1，回到第2层
15	2	fib(2)	调用fib(0)
16	3	fib(0)	返回0，回到第2层
17	2	fib(2)	计算出fib(2)=1+0=1,返回1，回到第1层
18	1	fib(4)	计算出fib(4)=2+1=3，返回3，回到第0层
19	0	main()	输出3

从上述分析中可以看出，递归分为递归前进和递归返回两个过程，整个递归过程完全是动态的，不停地在各层递归函数之间调用，这样就能正确地把答案计算出来。图1.1画出了函数间的调用关系(其中数字对应上表的第几步)。

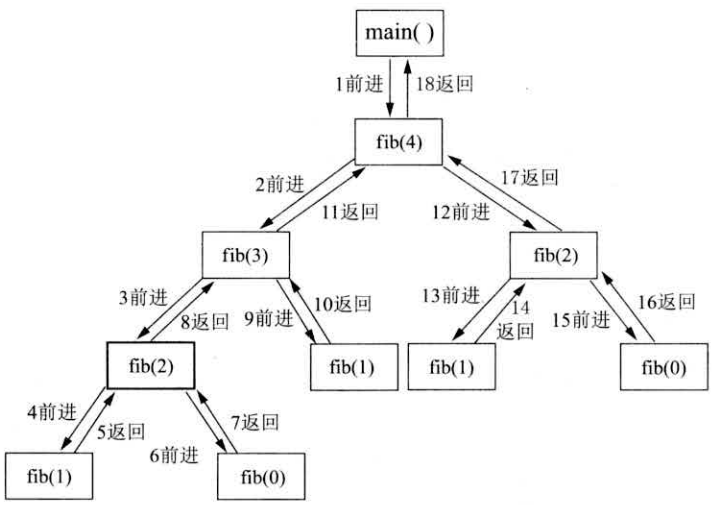


图1.1 递归函数调用关系树

1.6.4 递归函数的应用

【例1.10】用递归方法求m和n两个数的最大公约数(m>0,n>0)。

方法1：定义函数Gcd(m,n)计算m和n的最大公约数，根据**辗转相除法**得到：

（1）递归关系式：Gcd(m,n)=Gcd(n,m % n)。

（2）递归终止条件：Gcd(m,0)=m。

对应的程序eg1.10_1如下：

```
1  //eg1.10_1
2  #include<iostream>
3  using namespace std ;
4  int Gcd(int m,int n)
5  {
6    return (n==0)? m : Gcd(n,m % n);
7  }
8  int main()
9  {
10   int m,n;
11   cin>>m>>n;
12   cout<<Gcd(m,n)<<endl;
13   return 0;
14 }
```

运行结果：
输入：12 16
输出：4

方法2：这里提供另外一个**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)

该方法与方法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。

方法2对应的程序eg1.10_2如下：

```
1  //eg1.10_2
2  #include<iostream>
```

```
3  using namespace std ;
4  int Gcd(int m,int n)
5  {
6    if(m==n)return m;
7    if(m<n)return Gcd(n,m);
8    if(m & 1==0)return(n&1==0)? 2*Gcd(m/2,n/2):Gcd(m/2,n);
9    return(n & 1==0)? Gcd(m,n/2): Gcd(n,m-n);
10 }
11 int main()
12 {
13   int m,n;
14   cin>>m>>n;
15   cout<<Gcd(m,n)<<endl;
16   return 0;
17 }
```

运行结果：
输入：12 16
输出：4

【例1.11】分解因数。给出一个正整数a,要求分解成若干个正整数的乘积，即$a=a_1*a_2*a_3*...*a_n$，并且$1<a_1<=a_2<=a_3<=...<=a_n$，问这样的分解方案有多少种。注意a=a也是一种分解。

输入格式：第1行是测试数据的组数N；后面N行，每行包括一个正整数a(1<a<32768)。

输出格式：N行，每行输出一个正整数，表示分解方案数。

输入样例：

2
2
20

输出样例：

1
4

分析： 题目要求把a分解成$a_1*a_2*a_3*...*a_n$，并且$1<a_1<=a_2<=a_3<=...<=a_n$，可以依次分解$a_1,a_2,...,a_n$。在分解过程中需要记录剩下的数remain，为了确保新的因数不比之前的小，需要记录前一个因数pre。

因此定义函数fenjie(remain,pre)表示当前分解状态为剩余remain，前一个因数为pre：

（1）递归终止条件：remain=1时，完成一次分解，答案加1。

（2）递归关系式：当remain不等于1时，需要确定下一个因数i，i可

以有多个取值，i的枚举范围为pre..remain，考虑到当i不等于remain时，remain/i还需要继续分解，而下一个因数不小于i，因此需要满足remain/i>=i,i<=int(sqrt(remain)),枚举范围缩小到pre..int(sqrt(remain))，当然不要遗漏了remain。确定好下一个因数i后，递归调用fenjie(remain/i,i)进行下一次分解。

（3）主程序调用fenjie(a,2)即可。

图1.2给出以a=20为例的递归调用关系树。

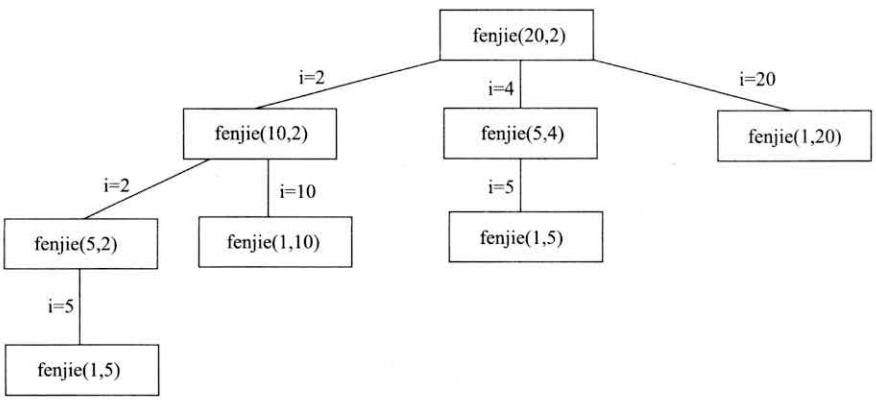


图1.2 分解因数递归函数调用关系

图1.2中4次满足递归终止条件，对应20的4种分解方案分别是20=2*2*5,20=2*10,20=4*5,20=20。

对应的程序eg1.11如下：

```
1  //eg1.11
2  #include<iostream>
3  #include<cmath>
4  using namespace std ;
5  int ans;
6  void fenjie(int remain,int pre)
7  {
8    if (remain==1){ans++;return;}
9    int m=int(sqrt(remain));
10   for(int i=pre;i<=m;i++)
11   if(remain % i==0) fenjie(remain/i,i);
12   fenjie(1,remain);
13 }
14   int main()
```

```
15  {
16    int n;
17    cin>>n;
18    for(int i=1;i<=n;i++)
19    {
20      int a;
21      cin>>a;
22      ans=0;
23      fenjie(a,2);
24      cout<<ans<<endl;
25    }
26    return 0;
27  }
```

【例1.12】汉诺塔游戏。汉诺塔由编号为1到n且大小不同的圆盘和3根柱子a,b,c组成，编号越小，盘子越小。开始时，这n个圆盘由大到小依次套在a柱上，如图1.3所示。要求把a柱上n个圆盘按下述规则移到c柱上：

（1）一次只能移一个圆盘，它必须位于某个柱子的顶部。

（2）圆盘只能在三个柱子上存放。

（3）任何时刻不允许大盘压小盘。

将这n个盘子用最少移动次数从a柱移动到c柱上，输出每一步的移动方法。

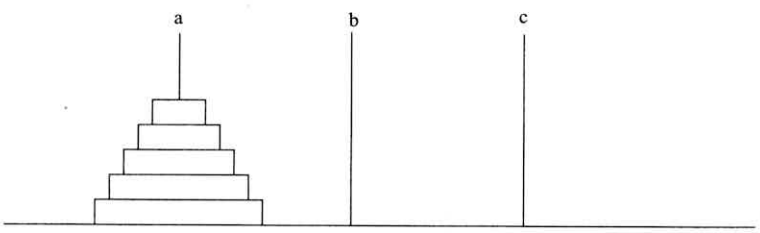


图1.3 汉诺塔

输入格式：只有1行，一个整数n(1<=n<=20),表示盘子的数量。

输出格式：输出若干行，每行的格式是“步数.Move 盘子编号 from 源柱 to 目标柱”。

输入样例：

3

输出样例：

1. Move 1 from a to c

2. Move 2 from a to b
3. Move 1 from c to b
4. Move 3 from a to c
5. Move 1 from b to a
6. Move 2 from b to c
7. Move 1 from a to c

分析：如何移动才能使得移动次数最少，考虑1到n这n个盘子哪一个盘子先移到c盘的目标位置？显然是n号圆盘！要想把n号圆盘移到c柱上，必须满足两个条件：

（1）a柱n号圆盘上面的1到n-1号圆盘都已经被移走。

（2）c柱上没有任何其他圆盘。

题目要求输出源柱和目标柱的状态，我们重新整理一下题目的要求，就是："用最少移动次数把1到n号圆盘从a柱经过b柱移到c柱"，根据上面分析，我们需要分三步走：

第1步：用最少移动次数把1到n-1号圆盘从a柱经过c柱移到b柱。

第2步：把n号圆盘直接从a柱移到c柱。

第3步：用最少移动次数把1到n-1号圆盘从b柱经过a柱移到c柱。

观察发现，第1步和第3步与原问题的本质是一样的，只是圆盘数量在减少，源柱、中间柱、目标柱的状态发生了变化。至此，递归关系比较明显，递归终止条件就是当n=1时，直接从a柱移到c柱即可。

接下来只要把上面的自然语言描述转变一下就可以了。我们定义hanoi(n,a,b,c)函数用来输出"用最少移动次数把1到n号圆盘从a柱经过b柱移到c柱"的移动序列，n>1时依次执行：

（1）hanoi(n-1,a,c,b)。

（2）输出"Move n from a to c"。

（3）hanoi(n-1,b,a,c)。

对应的程序eg1.12如下：

```
1  //eg1.12
2  #include<iostream>
3  using namespace std ;
4  int step;
5  void hanoi(int n,char a,char b,char c)
6  {
7    if(n==1)cout<<++step<<".Move"<<n<<"from"<<a<<
```

```
        "to" <<c<<endl;
8     else
9     {
10       hanoi(n-1,a,c,b);
11       cout<<++step<<".Move"<<n<<"from"<<a<<"to"<<c<<endl;
12       hanoi(n-1,b,a,c);
13    }
14}
15 int main()
16 {
17    int n;
18    cin>>n;
19    hanoi(n,'a','b','c');
20    return 0;
21 }
```

练 习

（1）数根可以通过把一个数的各个位上的数字加起来得到。如果得到的数是一位数，那么这个数就是数根。如果结果是两位数或者包括更多位的数字，那么再把这些数字加起来。如此进行下去，直到得到是一位数为止。比如，对于24来说，把2和4相加得到6，由于6是一位数，因此6是24的数根。再比如39，把3和9加起来得到12，由于12不是一位数，因此还得把1和2加起来，最后得到3，这是一个一位数，因此3是39的数根。

输入格式：一个正整数(小于10^{1000})。

输出格式：一个数字，即输入数字的数根。

输入样例：

24

输出样例：

6

（2）Pell数列a_1，a_2，a_3...的定义是这样的：$a_1=1$,$a_2=2$，...，$a_n=2*a_{n-1}+a_{n-2}(n>2)$。给出一个正整数k，要求Pell数列的第k项模上32767是多少。

输入格式：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)。

输出格式：n行，每行输出对应一个输入。输出应是一个非负整数。

输入样例：

2

1

8

输出样例：

1

408

（3）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。

输入格式：若干行，每行包含一个正整数N(1<=N<=30)，代表楼梯级数。

输出格式：不同的走法数，每行输入对应1行输出。

输入样例：

5

8

10

输出样例：

8

34

89

（4）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放法（放法数用K表示）？5，1，1和1，5，1是同一种放法。

输入格式：第1行是测试数据的数目t（0<=t<= 20）；以下每行均包含两个整数M和N(1<=M，N<=10)，以空格分开。

输出格式：对输入的每组数据M和N，用1行输出相应的K。

输入样例：

1

7 3

输出样例：

8

（5）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)

输入格式：1个正整数n（n<=20000）。

输出格式：1行，符合约定的n的2的幂次方（在表示中不能有空格）。

输入样例：

137

输出样例：

2(2(2)+2+2(0))+2(2+2(0))+2(0)

第2章　字符串处理——string类型

在《CCF中学生计算机程序设计入门篇》中，处理字符、文本的数据结构采用的是字符数组的方式，文本在计算机里称为字符串，C++专门扩展了string类型，供我们方便地处理字符串。这一章我们将学习使用string类型对字符串进行输入、输出、赋值、连接、查找、插入、删除等操作处理。

2.1　string类型的引入

【例2.1】在屏幕上输出：Hello world!。

```
1  //eg2.1
2  #include <iostream>
3  using namespace std;
4  int main()
5  {
6    cout << "Hello world!";
7    return 0;
8  }
```

运行结果：
Hello world!

说明：

（1）程序eg2.1第6句中的“Hello world!”就叫字符串，它必须用英文的双引号括起来。

（2）这种内容固定的字符串也叫字符串常量。字符串的输出和数字输出差不多，可以用流cout。

【例2.2】屏幕提示输入姓名(input name：)，键盘输入你的姓名XXX，在屏幕上输出：my name is：XXX。

```
1  //eg2.2
2  #include <iostream>
3  #include <string>
4  using namespace std;
5  int main()
```

运行结果：
input name：xiaowang
my name is：xiaowang

```
6  {
7   string name;
8    cout <<"input name:";
9    cin >> name;
10   cout <<"my name is: " << name <<endl;
11   return 0;
12 }
```

程序eg2.2为了输入字符串，第7句定义了一个字符串类型的变量name。字符串变量的输入、输出可以简单地用流cin和cout，非常方便。

说明：

（1）定义字符串变量格式：string　变量名。

（2）程序必须包含string头文件：#include <string>。

（3）输入的字符串可以认为是任意长，只受计算机内存限制。

（4）可以输入中文。

注意：用cin读入字符串时，空格和换行符都被认为是字符串的结束，因此输入的姓名中间不能有空格。

【例2.3】过滤多余的空格（来源：NOI题库）。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。

输入格式：1行，一个字符串（长度不超过200），句子的头和尾都没有空格。

输出格式：过滤之后的句子。

输入样例：

Hello　　world.This is　　c language.

输出样例：

Hello world. This is c language.

分析：cin只能一个一个读“单词”，不读空格。现在要解决两个技术：

（1）判断读入结束。

（2）字符串连接。

程序如下：

```
1  //eg2.3
2  #include <iostream>
3  #include <string>
4  using namespace std;
```

```
5  int main()
6  {
7    string s,temp;
8    cin >> s;
9    while(cin >> temp)
10   {
11     s += ' '+temp;
12   }
13   cout << s <<endl;
14   return 0;
15 }
```

说明：

（1）第9句“while(cin >> temp)”的功能是循环读入数据，在读不到的时候停止循环。

（2）第11句“s += ' '+temp”是在s的后面加一个空格和字符串变量temp，是一种字符串连接的简便方式。

注意：

（1）字符常量用单引号括起来(比如：'*')，字符串常量要用双引号(比如："abc 123")。即使一个字符的字符串也不是字符类型，例如：'A'=="A"是错误的表达式。

（2）两个字符串常量是不能直接用加号连接的，例如：

```
string s = “abc” + “def”;
```

是错误的。可以改成：

```
string s = “abc” “def”;
```

（3）多次在字符串后面追加内容时，用成员函数append是个不错的选择。例如：

```
s1.append( “s2=” ).append(s2).append( “#end\n” );
```

【例2.4】输入一个1到7的数字，表示星期一到星期日，屏幕输出相应的英文：Mon., Tue., Wed., Thur., Fri., Sat., Sun.。

程序如下：

```
1  //eg2.4
2  #include <iostream>
3  #include <string>
4  using namespace std;
```

运行结果：
输入：3
输出：Wed.

```
5  string dayName[8]={"","Mon.","Tue.","Wed.",
   "Thur.","Fri.", "Sat.","Sun."};
6  int main()
7  {
8    int day;
9    cin >> day;
10   cout << dayName[day] <<endl;
11   return 0;
12 }
```

说明：从第5句可以看出，定义string类型的数组和int类型相似，可以同时进行初始化。

2.2　string类型的基本操作

【例2.5】找第一个只出现一次的字符（来源：NOI题库）。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。

输入格式：一个字符串，长度小于100000。

输出格式：输出第一个仅出现一次的字符，若没有，则输出no。

输入样例：

abcabd

输出样例：

c

分析：这里先使用时间复杂度是$O(N^2)$的算法，下面是两重循环编写的程序。

```
1  //eg2.5
2  #include <iostream>
3  #include <string>
4  using namespace std;
5  int main()
6  {
7      string s;
8      cin >> s;
9      for(int i=0; i<s.size(); i++)
10     {
```

```
11          int c = 0;
12          for(int j=0;j<s.size();j++)
                                          // 统计与 s[i] 相同的个数
13              if(s[i]==s[j] ) c++;
14          if(c==1)
15          {
16              cout <<s[i]; return 0;
17          }
18      }
19      cout <<"no"<<endl;
20      return 0;
21  }
```

说明：

（1）第 12 句中的 size() 是 string 类型的成员函数，调用方式为 s.size()。它返回的值是 s 的大小，也就是 s 的长度。成员函数是指某个类型的特有函数，成员函数调用方式是：变量名.函数名(参数)，这样能保证编译器知道调用正确的函数。

（2）第 13、16 句中都有取字符串 s 的某个字符，s[i] 表示字符串 s 的第 i 个字符。要提醒的是，下标 i 是从 0 开始的。

（3）提交试一试，AC 了。

思考：

（1）成员函数和一般的函数有什么不同？

（2）怎样优化算法，使时间复杂度为 O(N)？

【例 2.6】统计数字字符个数（来源：NOI 题库）。输入 1 行字符，统计出其中数字字符的个数。

输入格式：1 行字符串，总长度不超过 255。

输出格式：输出为 1 行，输出字符串里面数字字符的个数。

输入样例：

Peking University is set up at 1898.

输出样例：

4

分析：读入 1 行字符放入一个字符串变量，再判断每个字符是否是数字即可。

程序如下：

```
1  //eg2.6
2  #include <iostream>
3  #include <string>
4  using namespace std;
5  int main()
6  {
7    string s;
8    getline(cin, s );
9    int c=0;
10   for(int i=0; i<s.size(); ++i)
11     if('0'<=s[i] && s[i] <= '9')
12       c++;
13   cout << c <<endl;
14   return 0;
15 }
```

运行结果：
输入：abc24e x3
输出：3

说明：

（1）由于输入的1行字符串中有空格，程序eg2.6第8句使用“getline(cin,s)”读入。getline函数默认是碰到换行符才结束，因此可以读进来空格。

（2）判断一个字符是否为数字，可以使用isdigit(c)函数（在头文件cctype里）。这里使用了“'0'<=s[i]&&s[i]<='9'”逻辑表达式方法。

提醒：假设输入格式为：

```
3 5
abc   def  12 is.
```

读入程序：

```
int n,m;
string s;
cin >> a>>b;
getline(cin, s );
```

这样读入的s是空串，即长度为0。这是因为“cin>>a>>b;”执行之后，数字5后面的回车换行符还在输入流中，getline碰到它就结束了。因此，下面程序才正确：

```
int n,m;
string s;
cin >> a>>b;
```

```
getline(cin, s );          //换行
getline(cin, s );          //读下一行
```

【例 2.7】摘录文字。输入 1 行由字母和字符“#”组成的字符串，保证“#”出现偶次。从前向后看，每两个“#”字符之间的字符串是要摘录的文字，请编程把摘录的字符串连续输出。

输入格式：1 行字符串，总长度不超过 1000000。

输出格式：“#”号对之间的字符。

输入样例：

```
a#abcd#xyz#efgh#opq.
```

输出样例：

```
abcdefgh
```

程序如下：

```
1  //eg2.7
2  #include <iostream>
3  #include <string>
4  using namespace std;
5  int main()
6  {
7    string s,ans;
8    int p;
9    cin >> s;
10   for(int i=0; i<s.size(); i++)
11       if(s[i]=='#')                    // 找到前面的 # 位置
12       {
13           p=i+1;                       // 要摘录的开始位置
14           for(i++;s[i]!='#';i++);      // 找到后面的 # 位置
15           ans+= s.substr(p,i-p);       // 取出这段，连接到 ans
16       }
17   cout << ans <<endl;
18   return 0;
19 }
```

说明：第 15 句中的成员函数 substr(开始位置，长度)是取字符串中一段子串。

【例2.8】选择你喜爱的水果（来源：NOI题库）。程序中保存了七种水果的名字，要求用户输入一个与水果有关的句子。程序在已存储的水果名字中搜索，以判断句子中是否包含七种水果的名称。如果包含，则用词组“Brussels sprouts”替换句子中出现的水果单词，并输出替换后的句子。如果句子中没有出现这些水果的名字，则输出“You must not enjoy fruit.”。假设七种水果的名字为：apples,bananas,peaches,cherries,pears,oranges,strawberries。

输入格式：有多行，每行是一个字符串（长度不超过200）。每行输入中只会有一个水果名称，不会存在1行输入包括多种水果名称或重复水果名称的情况。

输出格式：如果包含水果单词，则用词组“Brussels sprouts”替换句子中出现的水果单词，并输出替换后的句子。如果句子中没有出现这些水果的名字，则输出“You must not enjoy fruit.”。

输入样例：

```
I really love peaches on my cereal.
I'd rather have a candy bar
apples are wonderful with lunch
```

输出样例：

```
I really love Brussels sprouts on my cereal.
You must not enjoy fruit.
Brussels sprouts are wonderful with lunch
```

程序如下：

```
1  //eg2.8
2  #include <iostream>
3  #include <string>
4  using namespace std;
5  string fruits[7] = {"apples","bananas","peaches","cherries",
                       "pears","oranges","strawberries"};
6  int main()
7  {
8    string s;
9    while(getline(cin,s))
10   {
```

```
11      int pos, id=-1;        //记录查到水果的位置、是哪个水果
12      for(int i=0;i<7;i++)
13      {
14        pos=s.find(fruits[i]);  //查找水果
15        if(pos!=string::npos)  //查到一种水果
16         {id=i;break;}
17      }
18      if(id==-1)
19       cout <<"You must not enjoy fruit. "<<endl;
20      else
21      {                              //替换查找到的水果
22      s.replace(pos, fruits[id].size(),"Brussels sprouts");
23           cout << s << endl;
24      }
25     }
26    return 0;
27    }
```

说明：

（1）成员函数find(subs)是查找子字符串subs，如果查找到就返回第1个出现subs的位置；如果没找到subs，返回-1（为了兼容各C++版本，最好写成string::npos）。

（2）成员函数replace（开始位置，长度，要换上的字符串）是把字符串的一段内容换成指定的要换上的字符串。例如："s="abcde"; s.replace (1,3,"**");"的结果是s的"bcd"（下标从1开始长度为3）被"**"替换，替换后的s为"a**e"。

思考： 替换(replace)函数的功能，也可以通过删除(erase)、插入(insert)函数实现。比如程序eg2.8的第22句可改成：

```
s.erase(pos, fruits[id].size());
s.insert(pos, "Brussels sprouts" );
```

相反，删除(erase)或插入(insert)函数功能可否用替换(replace)函数实现？

2.3 string类型中字母与数字的关系

【例2.9】提取整数。有1行由小写字母和数字组成的字符串，请求出

其中所有数的和。

输入格式：一个字符串，长度小于100000。

输出格式：输出一个整数。数据保证答案不超过2^62。

输入样例：

ab123cde45enf

输出样例：

168

分析：由于答案不超过63位二进制，答案用long long型变量就可以了。

程序如下：

```
1  //eg2.9
2  #include <iostream>
3  #include <string>
4  using namespace std;
5  string s;
6  long long ans=0;
7  int main()
8  {
9    cin >> s;
10   for(int i=0; i<s.size(); ++i)
11   if('0'<=s[i] && s[i]<='9')
12   {
13       long long t=0;
14       for(;i<s.size() && '0'<=s[i] && s[i]<='9' ; i++)
15         t=t*10+(s[i]-'0');
16       ans += t;
17   }
18   cout << ans <<endl;
19   return 0;
20 }
```

说明：程序的第15句中“s[i]-'0'”是什么意思？在C++中，字符型也可以当做整数使用，整数值是它的ASCII码值，这是C++中的类型自动转换功能。比如：'0'可以看成数48，'1'可以看成49，'A'可以看成56等。因此“s[i]-'0'”就相当于一个数字字符转换成数字。

【例2.10】读入1行由0和1字符组成的二进制数字符串，请转换成十六进制数。

输入格式：一个0/1字符串，长度小于100000。

输出格式：输出1行转换后的十六进制数。

输入样例：

11010100101

输出样例：

6A5

分析：整数的二进制格式，从最低位开始，每4位对应一个十六进制数位。

程序如下：

```
1  //eg2.10
2  #include <iostream>
3  #include <string>
4  #include <algorithm>
5  using namespace std;
6  string s, ans;
7  int main()
8  {
9    cin >> s;
10   for(int i=s.size(); i>0; i-=4 )
11   {
12      int t=0;
13      for(int j=max(0,i-4);j<i;j++)
14        t = t*2 + (s[j]-'0');
15      if(t<10)ans=char(t+'0')+ans;
16      else ans =char('A'+t-10)+ans;
17   }
18   cout << ans<<endl;
19   return 0;
20 }
```

注意：

（1）要从后向前4位4位地取。

（2）要注意边界情况：max(0,i-4)，max函数在<algorithm>中。

说明：把数字转换字符要用强制类型转换。比如程序第15句的char(t+'0')是把数强制转换成字符。C++的风格是char（表达式），也可以使用早期的C语言风格——（char）表达式，比如：（char）49或（char）

（t+'0'）等形式。

【例2.11】数字加密。输入2个正整数A和B，计算出A除B的值，按精确到小数点5位输出。但为了保密需要，每位的数字都经过了转换，数字0变成字符'A'，数字1变成字符'B'，…，数字9变成字符'J'等。

输入格式：2个整数A和B，范围都在[1…10000]。

输出格式：输出加密后的浮点数，精度为小数点后5位。

输入样例：

6 7

输出样例：

A.IFHBE

解释：

6.0/7=0.85714

分析：本题的关键是把浮点数转换成字符串。用类似前面的数字与字符的转换方法，我们可以解决该题。不过灵活使用C++的stringstream流，可以方便地对各种类型进行相互转换。

程序如下：

```
1  //eg2.11_1
2  #include <iostream>
3  #include <string>
4  #include <sstream>
5  #include <iomanip>
6  using namespace std;
7  int a,b;
8  stringstream tempIO;
9  string ans;
10 int main()
11 {
12   cin >> a >> b;
13   tempIO << fixed<<setprecision(5) << double(a)/b;
14   tempIO >> ans;
15   for(int i=0; i< ans.size() ; i++)
16     if(ans[i]!='.')
17        ans[i] = char( ans[i]+'a'-'0');
```

```
18      cout << ans<<endl;
19    return 0;
20 }
```

注意：

（1）加头文件<iomanip>是为了处理输出格式，包括fixed和setprecision。

（2）头文件<sstream>提供了字符串流，tempIO在本程序中的作用相当于一个临时文本文件，先把浮点数输出，再读到字符串ans里，完成了转换。反过来转换也可以。

（3）如果要多次使用tempIO时，每次使用后要记得tempIO.flush();进行清空。

***说明：**stringstream是输入输出字符串流，可以直接对它输入和输出，如果和模板配合，可以方便地写个任意两个类型的转换函数，参考程序如下：

```
1  //eg2.11_2
2  #include <iostream>
3  #include <string>
4  #include <sstream>
5  using namespace std;
6  //=== 任意2个类型的转换样例函数 ===
7  template<class TA,class TB >
8  TB AtoB( const TA  &a)
9  {
10   stringstream st;
11   st<<a;                    // 向流中传值
12   TB b;                     // 这里存储转换结果
13   st>>b;                    // 向b中写入值
14   return b;
15 }
16 int main()
17 {
18   int x=123;
19   string y;
20   y=AtoB<int, string>(x);
21   cout << "string:"+y<<endl;
22   y="-126";
```

运行结果：
string：123
-12600

```
23    x=AtoB<string, int>(y);
24    cout << x*100<<endl;
25    return 0;
26 }
```

【例2.12】有N个人的姓名，请把他们按姓名的字典序排序输出。

输入格式：第1行，有一个整数N，N的范围是[1…10000]；下面第2行到第N+1行，每行是1个姓名。姓名由不超过50个小写字母组成。

输出格式：N行，每行一个姓名。

输入样例：

```
3
wang
liying
anqian
```

输出样例：

```
anqian
liying
wang
```

分析： 数组排序可以调用<algorithm>中的sort函数（参见STL相关章节），对于string类型，默认的大小关系就是字典序。

程序如下：

```
1  //eg2.12_1
2  #include <iostream>
3  #include <string>
4  #include <algorithm>
5  using namespace std;
6  int n;
7  string a[10000];
8  int main()
9  {
10   cin >> n;
11   for(int i=0; i<n; i++)
12     cin >> a[i];
13   sort(a, a+n);
14   for (int i=0; i<n; i++)
15     cout <<a[i]<<endl;
```

```
16   return 0;
17 }
```

说明：String类型的大小比较默认的是字典序。两个字符串s1,s2的比较有两种方式：

（1）compare( )函数。最常见的方式是s1.compare(s2)。s1和s2相等时返回0；s1字典序小于s2时返回值小于0；s1字典序大于s2时返回值大于0。compare函数可以比较某一段子串等功能，具体请查阅相关说明。

（2）用比较运算符：‘<’、‘<=’、‘==’、‘>=’、‘>’。比如：

```
if ( s1 > s2 ) cout << s1 ; else  cout <<s2;
```

字典序解释：2个字符串a和b，从前向后一个字符一个字符比较。遇到第i个不相同时，结果就是a[i]和b[i]的大小关系。如果一直相等，有一个字符串结束，长的大；一样长的话，则2字符串相等。为了说明字典序的原理，下面程序中的比较函数里面没有直接用“a>b”，而是模拟字典序定义编写。

程序如下：

```
1  //eg2.12_2
2  #include <iostream>
3  #include <string>
4  #include <algorithm>
5  using namespace std;
6  int n;
7  string a[10000];
8  //===== 模拟函数：字典序 a>b===
9  bool cmp_str( const string &a, const string &b)
10 {
11   int n=min(a.size(),b.size());
12   for(int i=0;i<n;i++)
13     if(a[i]!=b[i])return a[i]>b[i];
14   return a.size() > b.size();
15 }
16 int main()
17 {
18   cin >> n;
19   for(inti=0;i<n;i++)
20     cin >> a[i];
```

```
21    sort(a,a+n,cmp_str);
22    for (int i=0;i<n;i++)
23    cout <<a[i]<<endl;
24    return 0;
25 }
```

2.4　string类型的应用

【例2.13】车牌统计。小计喜欢研究数字，他收集了N块车牌，想研究数字0~9中某两个数字相邻出现使用在车牌上的次数。比如：68出现在100块车牌上、44出现在0块车牌上。由于车牌太多，希望你编程帮助他完成研究，并输出最多出现的车牌数。

输入格式：第1行，一个整数N，范围[1..100000]；下面有N行，每行是大写字母和数字组成的字符串，长度不超过10。

输出格式：某两个数字相邻出现在车牌的最多数。

输入样例：

```
4
YE5777
YB5677
YC8367
YA77B3
```

输出样例：

```
3
```

解释：77出现在第1、2、4块车牌上，3块是最多的。

分析：直接分解每一个车牌字符串处理的方法比较麻烦，而且容易重复计数。用类似计数排序的方法比较简单些。

程序如下：

```
1  //eg2.13
2  #include <iostream>
3  #include <fstream>
4  #include <string>
5  using namespace std;
6  int n;
7  int c[100];
```

```
8  string m[100]; //"00","01",..,"99"
9  int main()
10 {
11 //===== 初始构造 "00","01"..."99" 要匹配串
12   for(int i=0; i<10; i++)
13     for(int j=0; j<10; j++)
14     {
15         m[i*10+j] += char(i+'0');
16         m[i*10+j] += char(j+'0');
17     }
18   cin >> n;
19   for(int i=0; i<n; i++)
20   {
21       string s;
22       cin >> s;
23   //===== 对出现的匹配串计数
24       for(int j=0; j<100; j++)
25         if(s.find(m[j])!=string::npos)c[j]++;
26   }
27   int ans=0;
28   for(int i=0;i<100; i++)
29       if(c[i]>ans) ans = c[i];
30   cout << ans<<endl;
31   return 0;
32 }
```

注意：字符不能直接用“+”连接，那样是自动转换成ASCII码相加了，所以要用15和16两句。

【例2.14】单词替换（改自：NOI题库）。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。

输入格式：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。

输出格式：输出只有1行，将s中所有单词a替换成b之后的字符串。

输入样例：

```
You want someone to help you
You
I
```

输出样例：

```
I want someone to help you
```

分析：原题是第1行每个单词中间只有一个空格，现在空格数可能多于1个，只能用getlin整行读入。

程序如下：

```
1  //eg2.14
2  #include <iostream>
3  #include <fstream>
4  #include <string>
5  using namespace std;
6  int main()
7  {
8    string s,s1,s2;
9    getline(cin,s);
10   s=" "+s+" ";
11   getline(cin,s1);
12   getline(cin,s2);
13   s1=" "+s1+" ";     //2端加空格    --- 找到的是单词
14   s2=" "+s2+" ";     //2端加空格
15   for(int k=0;(k=s.find(s1,k))>=0; k+=s2.size()-1)
       s.replace(k,s1.size() ,s2);
                  //k+=s2.size()-1 保证换了的不再查找前面的!
16   cout <<s.substr(1,s.size()-2);      // 去掉2端空格
17   return 0;
18 }
```

说明：

（1）为了防止找到的子串不是单词，把s、s1和s2两端都加空格。

（2）为了防止找到被s2换过的地方出现s1，find加一个查找的开始位置参数k，这样做算法效率也提高些。

【例2.15】旋转操作。把字符串旋转一次操作等价于把字符串的最后一个字符改放到第1个字符的前面，例如：

“abcdefg”---旋转一次---“gabcdef”

现在输入一个字符串s，还有N个旋转操作。每个操作有3个参数：s,t,c，意思是要你把开始位置是s，结束位置是t的这段字符串旋转c次。例如：字符串“abcdefg”，经过操作（2,5,2）后变为“abefcdg”。

输入格式：第1行，不包含空格的字符串s，长度不超过1000；第2行，一个整数N，表示下面有N个旋转操作度（1<=N<=1000）；第3行到第N+3行，每行3个整数，即s,t,c。保证0<=s<=t<s的长度，0<=c<10000。

输出格式：输出只有1行，将s依次N次旋转操作后的字符串。

输入样例：

```
Youwantsomeonetohelpyou
3
1 5 100
0 3 20
2 15 60
```

输出样例：

```
Ynetonuwantsomeohelpyou
```

分析：本题是模拟题，要注意的是：

（1）c次旋转操作不能直接每次旋转一下地模拟，那样时间复杂度太高，要精确计算旋转后的位置，直接一次旋转到位。

（2）不要把子串取出，直接在原串中操作就可以了。如果在string类型频繁地进行插入、删除操作，速度会很慢的！要注意：find、insert、erase等函数虽然只是一个命令，但时间复杂度是O（字符串长度）。

程序如下：

```
1  //eg2.15
2  #include <iostream>
3  #include <fstream>
4  #include <string>
5  using namespace std;
6  string os,ts;
7  int n;
8  int main()
9  {
10 //  --------
11   cin >> os;
12   ts=os;
```

```
    cin >> n;
    for(int j=0; j<n; ++j)
    {
        int s,t,c;
        cin >> s>>t>>c;
        int len = t-s+1;
        c %=len;          //c大于长度，循环重复了，只取余数即可
        for(int i=s;i<=t;i++)  // 备份下来到临时字符串 ts
          ts[i] = os[i];
        for(int i=s; i<=t; i++)
          if(i+c>t) os[i+c-t] = ts[i];  // 计算目标地址
          else os[i+c] = ts[i];
    }
    cout << os<<endl;
    return 0;
}
```

说明：为了防止被换过的地方重新被取出等错误，把原来的这段先保存到ts字符串是比较简明的方法。

本章小结

string类型变量的赋值和连接操作列表

格　式	举例说明
string s（字符串）;	定义并初始化，例如： `string s("abc 123");`
string s（个数，字符）;	定义并初始化为若干相同字母，例如： `string s(10,'*');`
string s=字符串;	定义并初始化赋值，例如： `string s="abc 123";`

s=字符或字符串;	赋值语句，与成员函数assign相似，例如：
	`string str1, str2, str3;`
	`str1 = "Test string: ";`
	`str2 = 'x';`
	`str3 = str1 + str2;`
	`cout << str3  <<endl;`

字符串变量+字符串变量 字符串变量+字符串常量 字符串常量+字符串变量	运算符“+”是连接2个字符串，例如： （同上）

续表

s+=字符/字符串	运算符“+=”是自身加赋值运算符，例如：
	`string s = "Test: ";`
	`for ( char i='a'; i<='z'; i++)`
	`s += i;`
	`cout << s;`

string类型的主要函数与运算	
格　式	举例说明

size()	求字符串长度，等同于length()函数，例如：
	`s="12 34";`
	`cout << s.size();`
	结果是：5

s[下标i]	取字符串的某个字符，等同于at(下标i)函数，例如：
	`s="abcd";`
	`cout<<s[0]<<s.at(2);`
	结果是：ac

getline（cin，s）；	读入一整行（直到换行），包括读入空格

substr（开始位置i，子串长度len）；	取字符串的子串。当i+len超过原字符串长度时，只取剩下的。
	提醒：i要在字符串长度内。例如：
	`s="abcdef";`
	`cout << s.substr(3,2)<<s.substr(3,20);`
	结果是：dedef

insert（插入位置i，插入字符串s）；	在字符串的第i个位置插入s，例如：
	`s="abcdef";`
	`s.insert(2,"+") ;`
	`cout <<s;`
	结果是：ab+cdef

erase（开始位置i，删除菜单len）；	输出字符串的第i个位置后的len个字符。
	例如：
	`s="abcdef";`
	`s.erase(2,3) ;`
	`cout <<s;`
	结果是：abf

replace（开始位置i，长度len，要换上的字符串ss）	用字符串ss替换字符串中i开始长度是len的一段，例如：
	`s="abcdef";`
	`s.replace(2,1,"123") ;`
	`cout <<s;`
	结果是：ab123def

续表

find（子串subs）	查找子串subs第1次出现的位置，没有找到返回string::npos，例如：
	`s="abcdef";`
	`int i=s.find("cd");`
	`cout << i;`
	结果是：2
	提示：
	find还有一些更强大的形式，比如在某一段查找、找最后的subs位置等

string类型变量的转换总结	
知识点	举例说明

char型可以直接当整数使用	值是其ASCII码，例如：
	`'A'-'0' -5`
	结果是：65-48-5 = 12
	`'7'-'0'`
	结果是：字符'7'转数字7

整数型转char型要强制类型转换	例如： `char('0'+5)` 结果是：字符'5'

任意类型间转换可使用字符串流	例如：
	`stringstream  tempIO << "123456";`
	`int x;`
	`tempIO >> x;`

string类可直接按字典序比较大小	可直接使用关系运算符： `>, >=, <, <=, ==, !=` 例如： `"ABC" > "BCD"` 结果为假

string类型可转换到C风格的字符数组	使用函数c_str()，例如：
	`string s="filename";`
	`printf("%s",s_str() );`

C风格的字符数组可直接赋给string类型	例如：
	`char cs[]="filename";`
	`string s = cs;`

练　习

（1）合法C标识符（来源：NOI题库）。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。

输入格式：1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。

输出格式：1行，如果它是C语言的合法标识符，则输出yes，否则输出no。

输入样例：

RKPEGX9R;TWyYcp

输出样例：

no

（2）统计字符。Johe最近玩起了字符游戏，规则是这样的：读入四行字符串，其中的字母都是大写的，Johe想打印一个柱状图显示每个大写字母的频率。你能帮助他吗？

输入格式：输入文件共有4行，每行为一串字符，不超过100个字符。

输出格式：与样例的格式保持严格一致。

输入样例：

THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
THIS IS AN EXAMPLE TO TEST FOR YOUR
HISTOGRAM PROGRAM.
HELLO!

输出样例：

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

说明：

①输出的相邻字符间有一个空格。

②最后一行的26个大写字母每次必须输出。

③大写字母A所在的第一列前没有空格。

（3）字符串的展开（来源：NOIP2007）。初赛普及组的“阅读程序写结果”的问题中，我们曾给出一个字符串展开的例子：如果在输入的字符串中，含有类似于“d-h”或者“4-8”的字串，我们就把它当做一种简写，输出时，用连续递增的字母或数字串替代其中的减号，即，将上面两个子串分别输出为“defgh”和“45678”。在本题中，我们通过增加一些参数的设置，使字符串的展开更为灵活。具体约定如下：

①遇到下面的情况需要做字符串的展开：在输入的字符串中，出现了减号“-”，减号两侧同为小写字母或同为数字，且按照ASCII码的顺序，减号右边的字符严格大于左边的字符。

②参数p1：展开方式。p1=1时，对于字母子串，填充小写字母。p1=2时，对于字母子串，填充大写字母。这两种情况下数字子串的填充方式相同。p1=3时，不论是字母子串还是数字字串，都用与要填充的字母个数相同的星号“*”来填充。

③参数p2：填充字符的重复个数。p2=k表示同一个字符要连续填充k个。例如，当p2=3时，子串“d-h”应扩展为“deeefffgggh”。减号两边的字符不变。

④参数p3：是否改为逆序。p3=1表示维持原来顺序，p3=2表示采用逆序输出，注意这时候仍然不包括减号两端的字符。例如当p1=1、p2=2、p3=2时，子串“d-h”应扩展为“dggffeeh”。

⑤如果减号右边的字符恰好是左边字符的后继，只删除中间的减号，例如：“d-e”应输出为“de”，“3-4”应输出为“34”。如果减号右边的字符按照ASCII码的顺序小于或等于左边字符，输出时，要保留中间的减号，例如：“d-d”应输出为“d-d”，“3-1”应输出为“3-1”。

输入格式：第1行为用空格隔开的3个正整数，依次表示参数p1，p2，p3；第2行为一行字符串，仅由数字、小写字母和减号“-”组成。行首和行末均无空格。

40%的数据满足：字符串长度不超过5。

100%的数据满足：1<=p1<=3，1<=p2<=8，1<=p3<=2。字符串长度不超过100。

输出格式：只有1行，为展开后的字符串。

输入样例：

样例1：

```
1 2 1
abcs-w1234-9s-4zz
```

样例2：

```
2 3 2
a-d-d
```

样例3：

```
3 4 2
di-jkstra2-6
```

输出样例：

样例1：

```
abcsttuuvvw1234556677889s-4zz
```

样例2：

```
aCCCBBBd-d
```

样例3：

```
dijkstra2**********6
```

（4）同构字符串。给定一个字符串T，它的长度是LT，那么字符串T可以用字符数组T[1..LT]来表示。你可以把T的任意两个字符交换位置，你可以交换任意多次。经过交换之后的字符串被称为T的同构串。

例如：T="abac"，那么"aabc"、"aacb"、"baac"、"baca"、"bcaa"、"caab"、"caba"、"cbaa"等都是字符串T的同构串。而"baab"、"bcab"等都不是字符串T的同构串。

再给定一个字符串S，长度是LS，那么字符串S可以用字符数组S[1..LS]来表示。初始时，ans=0。对于S的任意长度是LT的一段（即一个子串），如果是字符串T的同构串，那么ans增加1。你的任务就是输出ans最后的值。

输入格式：第1行，一个字符串T。长度不超过10000，T的每个字符要么是小写字母，要么是大写字母；第2行，一个字符串S，长度是5000000，S的每个字符要么是小写字母，要么是大写字母。

输出格式：一个整数，ans最后的值。

输入样例：

```
aba
```

baababac

输出样例：

4

样例解释：

当K=1时，S[1..3]="baa"，是T的同构串。

当K=2时，S[2..4]="aab"，是T的同构串。

当K=3时，S[3..5]="aba"，是T的同构串。

当K=4时，S[4..6]="bab"，不是T的同构串。

当K=5时，S[5..7]="aba"，是T的同构串。

当K=6时，S[6..8]="bac"，不是T的同构串。

【数据规模】

对于40%的数据，T的长度不超过100，且S的长度不超过10000。

对于70%的数据，S的长度不超过1000000。

（5）统计单词数（来源：NOIP2011）。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）。

输入格式：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。

输出格式：只有1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。

输入样例：

[sample 1]

To

to be or not to be is a question

[sample 2]

to

Did the Ottoman Empire lose its power at that time

输出样例：

[sample 1]

2 0

[sample 2]

-1

数据范围：

1≤单词长度≤10。

1≤文章长度≤1,000,000。

输入输出样例1说明：输出结果表示给定的单词To在文章中出现两次，第一次出现的位置为0。

输入输出样例2说明：表示给定的单词to在文章中没有出现，输出整数-1。

（6）字符环（来源：NOI题库）。有两个由字符构成的环，请写一个程序，计算这两个字符环上最长公共字符串的长度。例如，字符串“ABCEFAGADEGKABUVKLM”的首尾连在一起，构成一个环；字符串“MADJKLUVKL”的首尾连在一起，构成另一个环；“UVKLMA”是这两个环的一个公共字符串。

输入格式：若干行，每行包括两个不包含空格的字符串。这两个字符串用空格分开。若其中某个字符串的长度为1，则表示结束；否则，每个字符串的首尾相连即为一个环。每个环上字符总数不超过255。

输出格式：为每行输入，分别输出一个整数，表示这两个字符环上最长公共字符串的长度。最后一行没有输出。

输入样例：

ABCEFA24*92(GADEGKABUVKLM

AD&30ijJKLAaUVKLM

313435t974 008bac

A 33

输出样例：

6

0

第3章　数据类型的组合——结构和联合

在处理数据时，有时需要把不同意义或不同类型的几个数据视为一个整体。C++专门有个称为结构（struct）的类型，可以把各种类型的数据放在一起，这一章我们将学习结构（struct）类型的定义和运用，同时也介绍一下相关的联合(union)类型。

3.1　结构体(struct)的引入

【例3.1】窗口重叠。在Windows操作系统中，最主要的桌面元素是窗口。通常一个窗口有4个整数定义位置：左边坐标(left)、右边坐标(right)、上边坐标(top)、下边坐标(bottom)。

现在给你两个窗口位置信息，判断它们位置是否有重叠。

输入格式：共2行，每行四个整数，表示一个窗口的位置信息。

输出格式：如果两个窗口位置重叠，输出重叠的面积；否则输出0。

输入样例：

```
10  100  20  60
60  160  50  200
```

输出样例：

```
400
```

分析：虽然我们可以用2个数组“int A[4],B[4];”保存窗口的信息数据，但A[0]、A[1]等这些不能很好表示数据代表的是什么。如果把一个窗口的数据用left、right等表示，程序的可读性会提高很多。

下面程序中通过使用结构（struct）类型来演示“好风格”的编程方法。

```
1  //eg3.1
2  #include <iostream>
3  #include <fstream>
4  #include<algorithm>
5  using namespace std;
6  //========= 定义 struct 的类型，类型名叫：tWindow
7  struct tWindow {
```

```
8     int left,
9         right,
10        top,
11        bottom;
12 };
13 //==定义2个tWindow类型的变量winA和winB表示2个窗口
14 tWindow  winA, winB,
15    tmp;                    //===临时变量
16 //==========定义1个函数，输入窗口变量
17 tWindow inData()
18 {
19    tWindow tmp;
20    cin >> tmp.left >> tmp.right
21    >> tmp.top >> tmp.bottom;
22    return tmp;
23 }
24 int main()
25 {
26 //=========输入数据============
27    winA=inData();
28    winB=inData();
29 //=========判断计算，tmp是重叠窗口=======
30    tmp.left = max(winA.left, winB.left);
31    tmp.right = min(winA.right, winB.right);
32    tmp.top =  max(winA.top, winB.top);
33    tmp.bottom = min(winA.bottom, winB.bottom);
34    int s =(tmp.right - tmp.left)*(tmp.bottom - tmp.top);
                                                  //计算面积
35    if((tmp.right <=tmp.left)||(tmp.bottom <= tmp.top))
                                                  //不重叠
36       s=0;
37 //=========输出===============
38    cout << s <<endl;
39    return 0;
40    }
```

说明：

（1）第7～12句定义了一个新的数据类型，类型名叫tWindow。

（2）tWindow是我们自己定义的类型，不是C++本身的（比如int, double, string等），因此叫自定义数据类型。

（3）第14句“tWindow winA,winB;”是定义两个tWindow类型的变量。

（4）程序后面是对结构体中成员的操作，主要是赋值、取值。

一个结构体变量可以整体赋值，例如第27句中“winA=inData();”。

结构体变量的每个成员可以单独存取，例如第20，21，30~33，35句中的tmp.left等。

【例3.2】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：

```
1  //eg3.2
2  #include<iostream >
3  #include<fstream >
4  #include<string>
5  #include<algorithm>
6  using namespace std;
7  //========= 定义一个 struct 的类型，类型名叫：tStudent
8  struct tStudent {
9    string name;
10   int cha, math;
11   int total;
12 };
13 //========== 定义一个数组 A，每个元素是 tStudent 型的
14 tStudent A[110];
15 int N;
16 int main()
17 {
18 //========= 输入数据 ============
19   cin >> N;
20   for(int i=0; i<N; i++){
21     cin >> A[i].name;                    // 输入姓名
22     cin >> A[i].cha >> A[i].math;  // 输入语文、数学分数
23     A[i].total = A[i].cha+ A[i].math;     // 计算总分
24   }
25 //========= 冒泡排序 ==============
26   for(int last=N-1;last>0;last-- )
27   for(int j=0;j<last;j++)                    // 一趟冒泡
28     if(A[j].total < A[j+1].total )          // 小的交换到后面
29       swap(A[j],A[j+1]);
30 //========= 输出 ===============
31   for (int i=0; i<N; i++)
32     cout << A[i].name<<" "
33   <<A[i].cha<<" "<<A[i].math<<" "<<A[i].total<<endl;
34     return 0;
35 }
```

说明：

（1）第8～11句定义了一个新的数据类型，类型名叫tStudent。

（2）tStudent是我们自己定义的类型，为自定义数据类型。

（3）第14句“tStudent A[110];”定义了一个A数组，每个元素都是

tStudent类型的。

（4）第21～23句是对结构体中成员的操作：赋值、取值。

结构体是C++提供给用户自己定义数据类型的一种机制，最主要的功能是把各种不同类型的数据组成一个整体。结构体的类型定义格式通常如图3.1所示。

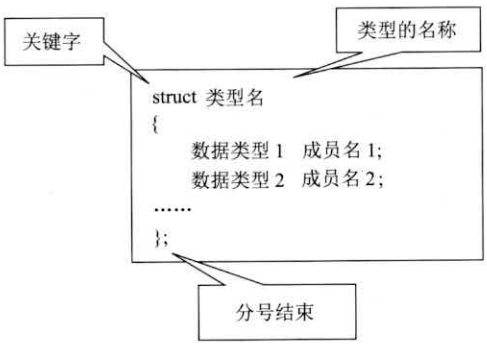


图3.1

结构体变量的定义有几种形式：

（1）struct 结构体类型名　变量名列表，例如：

```
struct tWindow a;
```

（2）结构体类型名　变量名列表，例如：

```
tWindow a;
```

（3）定义结构体类型的时候同时定义变量，例如：

```
struct tPoint       //定义坐标点类型
{
    int x, y;
}   p[100];         //同时定义了p数组变量，可保存100个坐标点
```

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：

```
tWindow a={10,50,100,200};
```

或

```
tStudent op ={ "xiaWan" , 89,90, 179 };
```

结构体的这种初始方法功能并不强大(C++11可能更好一些)，使用构造函数方法进行“初始化”会更加强大，在本章后面介绍。

3.2　结构体(struct)的使用

【例3.3】离散化基础。在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字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：

```
1  //eg3.3
2  #include<iostream>
3  #include<fstream>
4  #include<algorithm>
5  using namespace std;
6  //========= 定义 struct 的类型，类型名叫：tNode
7  struct tNode {
8    int data,                          // 数值
9      rank,                            // 排名
10     index;                           // 下标
11 };
12 int N;
13 tNode a[10001];                      // 数组
14 bool cmpData(tNode x, tNode y)
15 {
16   return x.data < y.data;
```

```
}
bool cmpIndex(tNode x, tNode y)
{
  return x.index < y.index;
}
int main()
{
//========= 输入数据 ============
    cin >>N;
    for(int i=0; i<N; i++)
      cin >> a[i].data, a[i].index = i;
//========= 根据值排序，求 rank==========
    sort(a, a+N, cmpData);
    for(int i=0; i<N; i++)
        a[i].rank = i+1;
//========= 根据下标排序 =========
    sort(a, a+N, cmpIndex);
//========= 输出 ===============
    for(int i=0; i<N; i++)
       cout <<a[i].rank<<" ";
    cout <<endl;
    return 0;
}
```

说明：

（1）sort 函数是<algorithm>提供的快速排序函数，可以自己定义比较函数。

（2）多关键字排序时显然要用结构体。

（3）根据下标排序的作用是回到原来数据的次序。

【*例 3.4】模拟链表。在今后的图论题编程中，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：

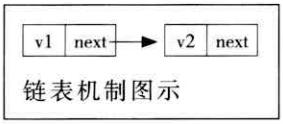
```
5 6
1 3
2 4
1 4
2 3
3 5
2 5
```

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 tNode
{
    int v;          //顶点编号
    int next;       //链表的下一个节点的下标
};
```



链表机制图示

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：

```
1  //eg3.4
2  #include <iostream>
3  #include <fstream>
4  #include<algorithm>
5  using namespace std;
6  //=========定义 struct 的类型，类型名叫：tNode
7  struct tNode {
8    int v,                  //顶点
9    next;                   //下一项，next=0 表示结束
```

```
10 };
11 int N, M;
12 tNode a[100001*2];          //数组，对于无向图，空间是边的2倍
13 int fa=0;                   //a数组的空余空间下标
14 tNode adj[5002];            //邻接链表的表头，其中v记表长度
15 void insert(int u,int v)//把v点插入到u点的邻接链表前
16 {
17   a[++fa].v = v;            //申请一个新的节点
18   a[fa].next = adj[u].next;
19   adj[u].next = fa;         //插入到u链表头
20   adj[u].v++;               //个数（链表长度）增加
21 }
22 int main()
23 {
24 //==========输入，插入数据============
25   cin >>N >>M;
26   for(int i=0; i<M; i++)
27   {
28       int u,v;
29       cin >> u >>v;
30       insert(u,v);          //插入v点到链表u
31       insert(v,u);          //插入u点到链表v
32   }
33 //==========输出===============
34   for (int i=1; i<=N; i++)
35   {
36       cout << adj[i].v<<" ";                          //表长度
37       for(int j=adj[i].next;j>0;j=a[j].next)          //遍历链表
38           cout <<a[j].v<<" ";
39       cout <<endl;}
40   return 0;
41 }
```

说明：

（1）这里adj[]的类型也是tNode，这样可以用成员v记录长度k。如果不用输出k，adj[]的类型可以定义为int类型。

（2）学会了建立邻接链表，以后写图的BFS和DFS就简单了。

【例3.5】生日相同（来源：NOI题库）。在一个有180人的大班级中，

存在两个人生日相同的概率非常大，现给出每个学生的名字，出生月日，试找出所有生日相同的学生。

输入格式：第1行，整数n，表示有n个学生，n ≤ 180；此后每行包含一个字符串和两个整数，分别表示学生的名字（名字第一个字母大写，其余小写，不含空格，且长度小于20）和出生月(1 ≤ m ≤ 12)日(1 ≤ d ≤ 31)。名字、月、日之间用一个空格分隔。

输出格式：每组生日相同的学生，输出1行，其中前两个数字表示月和日，后面跟着所有在当天出生的学生的名字，数字、名字之间都用一个空格分隔。对所有的输出，要求按日期从前到后的顺序输出。对生日相同的名字，按名字从短到长按序输出，长度相同的按字典序输出。如没有生日相同的学生，输出“None”。

输入样例：

```
6
Avril 3 2
Candy 4 5
Tim 3 2
Sufia 4 5
Lagrange 4 5
Bill 3 2
```

输出样例：

```
3 2 Tim Bill Avril
4 5 Candy Sufia Lagrange
```

分析：该题有多种方法可做。下面使用结构体和多关键字排序的方法解决。

```
1  //eg3.5
2  #include <iostream>
3  #include <fstream>
4  #include<algorithm>
5  using namespace std;
6  //========= 定义 struct 的类型，类型名叫：tStudent
7  struct tStudent {
8    string name,                          //姓名
9           m,                             //月
10          d;                             //日
```

```
11 };
12 int N;
13 tStudent a[200];                          //数组放学生信息
14 bool cmp(tStudent x, tStudent y)    //比较函数
15 {
16 //=====按日期排序
17   if(x.m != y.m) return x.m < y.m;
18   if(x.d != y.d) return x.d < y.d;
19 //=====日期相同，按姓名排序
20       if(x.name.size() != y.name.size() )
21           return x.name.size() < y.name.size();
22       return x.name < y.name;
23 }
24 int main()
25 {
26 //=========输入，插入数据 ============
27   cin >> N;
28   for(int i=0; i<N; i++)
29   {
30       cin >> a[i].name
31       >>a[i].m >> a[i].d;
32   }
33   sort(a, a+N, cmp);
34 //=========输出 ===============
35   int c=0;    //是否有生日相同
36   for(int i=0; i<N; i++)
37   {
38       if((a[i+1].m==a[i].m && a[i+1].d== a[i].d))
39       {
40           cout << a[i].m<<" "<<a[i].d
41            <<" "<<a[i].name<<" ";
42           while(a[i+1].m==a[i].m  && a[i+1].d== a[i].d)
43           {
44               c++,
45               cout<<a[++i].name<<" ";
46           }
47           cout<<endl;
48       }
49   }
50   if (c==0)
```

```
51        cout <<"None"<<endl;
52    return 0;
53 }
```

*3.3　结构体(struct)的扩展

C++的结构体功能因为类（class）技术的出现得到了很大的增强，其中和OI有关的有：成员函数和运算符重载。下面通过几个例子简单介绍一下，更具体的内容请查阅相关资料。

【例3.6】时间运算。在某个上网计费系统中，用户使用时间通常格式是：几小时几分钟。用一个结构体表示时间是个不错的方法。现在希望你设计个好的方法，能够快速方便地在程序中累加时间。

输入格式：第1行，一个整数N，范围在[1…1000]；下面有N行，每行两个整数：hi mi，表示一个用户上网时间是hi小时、mi分钟。

输出格式：1行，两个整数h和m，表示N个时间的和。

输入样例：

```
4
1 15
0 56
5 12
3 8
```

输出样例：

```
10 31
```

程序如下：

```
1  //eg3.6
2  #include<iostream>
3  #include<fstream>
4  #include<string>
5  using namespace std;
6  //========= 定义 struct 的类型，类型名叫：tTime
7  struct tTime {
8    int h,                                    // 小时
9        m;                                    // 分钟
```

```
10    tTime operator+(const tTime x)const     //对“+”重新定义
11    {
12        tTime tmp;
13        tmp.m =(m+x.m)%60;
14        tmp.h=h+x.h+(m+x.m)/60;
15        return tmp;
16    }
17 };
18 int N;
19 tTime a[1001], sum;
20 int main()
21 {
22    cin >>N;
23    sum.h=sum.m=0;
24    for(int i=0; i<N; i++)
25    {
26        cin >> a[i].h >> a[i].m;
27        sum = sum + a[i];   //两个结构体通过重新定义的+号直接相加
28    }
29    cout << sum.h<<" "<<sum.m<<endl;
30    return 0;
31 }
```

说明:

(1)第10~16行,在tTime类型里面新定义了这个类型的“+”运算,称为“+”重载。

(2)第28行使用重载运算符“+”,形式特别简洁明了。注意,由于没有重载“+=”运算符,所以不能写成“sum += a[i];”

二元运算符重载的一般格式为:

```
类型名  operator  运算符(const  类型名  变量)  const
{
    ...
}
```

运算符重载是个复杂的技术,比如还可以对复合运算符“+=”、输出符“<<”等重载,语法格式和上面不尽相同,这里就不展开讨论了。

程序中第13、14句中的h和m变量就是使用结构体中第8,9句定义的变量。

下面再看一个增加了成员函数的结构体运用。

【例3.7】集合运算。在数学上，2个集合A和B之间的运算通常有：并、差、交，分别记为A+B、A-B、A*B。数学老师想设计一款模拟集合运算的游戏，现在需要你帮忙编程。已知所有集合的元素都是小写英文字母，集合的输入、输出用字符串表示。例如：集合A={a, c, d, f }，输入输出用字符串“acdf”表示。

现在输入N个集合运算式，求运算结果。例如：

运算式：acdf - bcef

结果：ad

输入格式：第1行，一个整数N，表示有多少运算式，N范围在[1…100]；下面N行，每行一个运算式。中间运算符是'+'、'-'、'*'之一。

输出格式：共N行，对应输入的运算结果。

输入样例：

```
2
abef + cdefijk
abghio * gipqx
```

输出样例：

```
abcdefijk
gi
```

分析：本题是模拟题，比较简单，可以有多种编程方法，但使用结构体的成员函数和运算符重载会更加清晰、规范，风格漂亮。

程序如下：

```
1  //eg3.7
2  #include <iostream>
3  #include <fstream>
4  #include<string>
5  using namespace std;
6  //=========定义 struct 的类型，类型名叫：tSet
7  struct tSet {
8    bool set[26];                    //集合
9    void input()                     //输入集合成员函数
10   {
11     string s;
12     cin >> s;
```

```
13      memset(set,false,sizeof(set));
14      for(int i=0; i<s.size();i++)
15          set[s[i]-'a'] = true;
16    }
17    void output()           //输出集合成员函数
18    {
19      for(int i=0;i<26; i++)
20          if(set[i])
21            cout<<char(i+'a');
22      cout<<endl;
23    }
24    tSet operator + (const tSet x)const     //对“+”重新定义
25    {
26        tSet tmp;
27        for(int i=0;i<26;i++)
28             tmp.set[i] = set[i] || x.set[i];
29        return tmp;
30    }
31    tSet operator - (const tSet x)const         //对“-”重新定义
32    {
33        tSet tmp;
34        for(int i=0; i<26; i++)
35             tmp.set[i] = set[i] && (!x.set[i]);
36        return tmp;
37    }
38    tSet operator * (const tSet x)const  //对“*号”重新定义
39    {
40         tSet tmp;
41         for(int i=0; i<26; i++)
42              tmp.set[i] = set[i] && x.set[i];
43         return tmp;
44    }
45 };
46 int N;
47 tSet A,B,C;
48 char op;
49 int main()
50 {
51    cin >>N;
52    for (int i=0; i<N; i++)
```

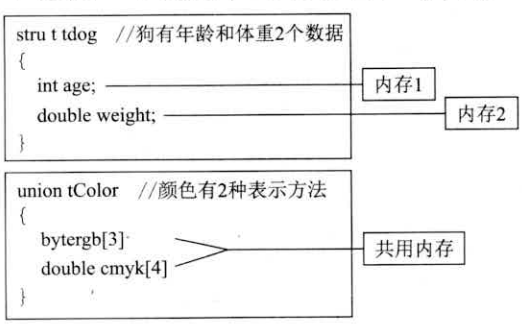
```
53    {
54        A.input();                    // 调用成员函数输入集合 A
55        cin >> op;                    // 输入运算符
56        B.input();                    // 调用成员函数输入集合 B
57        if (op=='+') C=A+B;           // 相应运算
58        else if (op=='-') C=A-B;
59        else if (op=='*') C=A*B;
60        C.output();                   // 输出
61    }
62    return 0;
63 }
```

说明：成员函数的定义方法和普通的函数很类似，但可以直接访问结构体内的成员变量，调用方法和调用数据成员一样。

*3.4　联合(union)的定义和使用

联合也称为共用体，是一种数据格式，能够储存不同类型的数据，但同一时间只能储存其中的一种类型数据。

下面用一个实例，对比联合和结构体的区别。



“tColor c;”定义了一个颜色变量c，c可以有2种方式记录颜色：rgb形式或者cmyk形式，但不能同时记录。

显然联合的用途之一：当数据项使用多种格式（但不会同时使用）时，可以节省空间。

【例3.8】注册账号。某网站收集了N个人的注册账号，账号类型有2种：身份证号、QQ号。请编程用恰当的数据结构保存信息，并统计身份证中男性和女性的人数，以及QQ号平均值（取整）。

输入格式：第1行，一个整数N，范围在[1…10000]；下面N行，每行一个字符和一个字符串。第一个字符表示账号类型，有i、q两种，第二个字符串是账号信息。

输出格式：1行，3个整数值：男性人数、女性人数、QQ号平均值。

输入样例：

```
6
i 522635197008278006
i 51170219740419175X
i 45102519760724935X
q 505165
q 34012459
i 511702198606266283
```

输出样例：

```
2 2 17258812
```

分析：如果只是单纯做本题，编程很简单。但为了展示union的使用方法，为今后编写复杂的程序打下基础，程序采用规范编写。

程序如下：

```
1  //eg3.8
2  #include <iostream>
3  #include <fstream>
4  #include<string>
5  using namespace std;
6  //========= 定义 struct 的类型，类型名叫：tID
7  struct tID {
8    char type;                    //账号类型
9    union
10   {
11       char idc[18];
12       long long qq;
13   };
14 };
15 int N;
16 tID a[10001];
17 int main()
18 {
19   //========= 输入 ========
```

```
20    cin >>N;
21    for(int i=0; i<N; i++)
22    {
23        cin >> a[i].type;
24        switch(a[i].type)
25        {
26            case 'i':
27            for(int j=0; j<18; j++)
28                cin >> a[i].idc[j];
29                break;
30                case 'q':
31                    cin>> a[i].qq;
32          }
33    }
34    //======== 处理 ============
35    long long sum=0, c=0;
36    int men=0, women =0;
37    for(int i=0; i<N; i++)
38         if(a[i].type =='i')
39            if((a[i].idc[16]-'0')%2)
40                men++;
41            else
42                women++;
43         else if(a[i].type=='q')
44            sum += a[i].qq,c++;
45    cout<< men<<" "<<women<<" "<<sum/c<<endl;
46    return 0;
47 }
```

说明：

（1）由于要识别联合变量到底是储存哪个类型，所以一般要有一个类型标识符。程序中先定义了一个struct，里面有type成员。

（2）在struct中程序第9~13嵌套定义了一个union，这里省略了类型名，更方便结构体变量存取联合的成员。

注意：union里面不能有空间不确定的数据结构类型，比如：string。

*3.5 枚举(enum)的定义和使用

C++中枚举(enum)提供了另一种创建符号常量的方式，有些场合可以替代const，甚至定义一种自定义类型。例如：

```
enum tColor
{
    red, green, blue, yellow, black, white
};
tColor x;
```

定义了一个叫tColor的枚举类型，并定义了这个类型的一个变量x。x只能取red、green、…、white几个值。

red、green、…、white可看成符号常量，对应数值0到5，可以当成整数常量运行。

枚举常量默认的值从0开始递增。但也可以使用赋值语句来人为设定枚举量的值，例如：

```
enum tBit
{
    one = 1,
    two = 2,
    three =4,
    four = 8,
    five = 16
};
```

【例3.9】取数。有一个N*N的二维网格，每格里面有1个整数。现在给定开始的位置(x,y)和方向（上、下、左、右之一），一直移动到网格的边界，计算移动过程中线路上格子里的数字和。

输入格式：1行，4个整数，第1个整数N，范围在[1..1000]，第2、3个整数是开始位置的坐标X和Y，表示在第X行Y列（编号1到N），第4个整数D表示方向，D=0表示向上，D=1表示向下，D=2表示向左，D=3表示向右；下面N行，每行N个整数，范围在[-1000..1000]。

输出格式：一个整数，数字和。

输入样例：

```
4 2 3 2
```

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

输出样例：

22

分析：输入数据中方向可以用数字表示，但编程中容易出错，特别是比较大的程序更容易混乱。在C++编程中采用枚举型可以有效减少失误。

程序如下：

```
1  //eg3.9
2  #include <iostream>
3  #include <fstream>
4  #include<string>
5  using namespace std;
6  //========== 定义 enum 类型，类型名叫：tDir
7  enum tDir
8  {
9    _up=0,
10   _down=1,
11   _left=2,
12   _right=3
13 };
14 int N,X,Y,D;
15 int a[1002][1002];
16 int main()
17 {
18   cin >>N >>X >>Y >>D;
19   for(int i=0; i<N; i++)
20        for(int j=0; j<N; j++)
21             cin >> a[i+1][j+1];     //0 行 0 列空出
22   int sum=0;
23   if(D==_up)
24        for(int i=X; i>0; i--)
25             sum+=a[i][Y];
26   else if(D==_down)
27        for(int i=X; i<=N; i++)
28             sum+=a[i][Y];
29   else if(D==_left)
```

```
30          for(int i=Y; i>0; i--)
31                sum+=a[X][i];
32    else if(D==_right)
33          for(int i=Y; i<=N; i++)
34                sum+=a[X][i];
35    cout << sum<<endl;
36    return 0;
37 }
```

说明：

（1）枚举型中的符号使用_up、_down、_left、_right，用前缀‘_’是为了尽量避免变量名冲突，比如‘left’就是C++的关键字。

（2）第23、26、29、32句中用枚举型符号，读起来不会有二义性。

本章小结

结构体的定义和使用	
知识点	举例说明

定义结构体类型：
struct 类型名
{
数据类型1 成员名1;
数据类型2 成员名2;
……
};
	struct tPoint //定义坐标点类型
	{
	int x, y;
	};

定义结构体类型的变量 struct 结构体类型名变量名列表;	struct tWindow a; 或 tWindow a;
结构体变量的成员访问 变量名.成员	cin >> A[i].name; cout <<A[i].age;
结构体变量可以整体操作	swap(A[j], A[j+1]); A[0]= temp;
结构体变量的初始化	tWindow a={10,50,100,200} ;

结构体运算符重载
类型名 operator 运算符（const 类型名变量）const
{
….
}
	tTime operator + (const tTime x) const
	{
	tTime tmp;
	tmp.m = (m+x.m)%60;
	tmp.h=h+x.h+(m+x.m)/60;
	return tmp;
	}

续表

结构体成员函数	和一般函数类似，只是使用和结构体数据成员一样要通过结构体变量调用
联合和枚举类型	
知识点	举例说明

union类型的定义：
和struct类似，但共用内存
	union tColor　　//颜色有2种表示方法
	{
	byte rgb[3];
	double cmyk[4];
	};

enume类型定义：
是一种创建符号常量的方式
	enum tColor
	{
	red, green, blue, yellow, black, white
	};
	tColor x;

练　习

（1）刚举行的万米长跑活动中，有N个人跑完了全程，所用的时间都不相同。颁奖时为了增加趣味性，随机抽了一个数K，要奖励第K名一双跑鞋。现在组委会给你N个人的姓名、成绩（用时，单位是秒），请你编程快速输出第K名的姓名。

输入格式：第1行，2个整数N和K，范围[1≤K≤N≤100]；下面N行，每行第1个是字符串表示姓名，第2个是个整数，表示这个人跑完的使用时间。

输出格式：1行，第K名的姓名。

输入样例：

```
5 3
wangxi   2306
xiaoming 3013
zhangfan  3189
chengli   4012
jiangbou  2601
```

输出样例：

```
xiaoming
```

（2）在使用离散化方法编程时，通常要知道每个数排序后的编号（rank值），相同的数对应一个编号。

输入格式：第1行，一个整数N，范围在[1…10000]；第2行，有N个整数，每个数都是int范围的。注意：可能有相同整数。

输出格式：依次输出每个数的排名。

输入样例：

5

8 2 6 9 2

输出样例：

3 1 2 4 1

（3）输入N个位数不超过1000的正整数，求它们的和。(编程使用strunc创建一个bigNum类型，并对‘+’运算符重载)

输入格式：第1行，一个正整数N，2≤N≤100；下面有N行，每行2个整数：a和b，位数都不超过1000。

输出格式：一个正整数，和。

输入样例：

1

12345 213

输出样例：

123558

第4章　功能强大的利器——指针

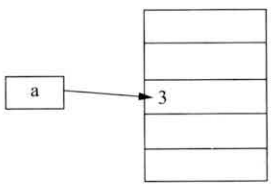
C++中被誉为功能强大、也被认为“太底层操作”的一个数据结构就是指针。理解指针的原理和运算是高水平OI选手必须具备的能力。这一章我们将学习指针的定义、申请空间、引用等知识，还将对数组、字符数组、函数等其中与指针相关联的概念作介绍。

4.1　指针概念、定义与内存分配

其实指针就像其他变量一样，所不同的是一般的变量包含的是实际的数据，而指针是一个指示器，它告诉程序在内存的哪块区域可以找到数据。先通过例子看看指针与通常的变量有什么不同。

1. 普通变量定义

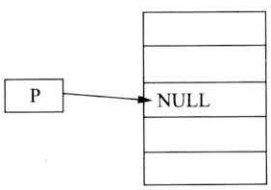
int a=3;



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的值的，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

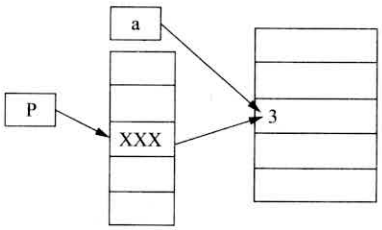
int * p=NULL;



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。p的间接操作要使用指针操作符“*”，即*p的值才是3。

表4.1

说　明	样　例
指针定义： 储存类型* 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

【例4.1】输入两个不同的浮点数，通过指针把最大值+10，并输出。程序如下：

```
1  //eg4.1
2  #include <iostream>
3  using namespace std;
4  int a,b;
5  int * p;
6  int main()
7  {
8    cin >>a >>b;
9    if(a>b)p=&a;          //p指向a
10   else p=&b;            //p指向b
```

```
11    *p += 10;                    //p 指向的整数 +=10
12    cout << *p <<endl;           // 输出 p 指向的整数
13    return 0;
14 }
```

说明：

（1）给变量 a 和变量 b 分配的保存数值的内存空间是固定不变的，程序在任何时候访问 a 和 b 都是一开始给定的地址。

（2）*p 的空间是不确定的，程序第 5 句定义时，全局变量 p 里面的值是 NULL（0）。第 9、10 句根据 a 和 b 的大小，才指向 a 或 b 的地址，p 才有了地址，*p 才有了空间。

（3）从上面特点划分：指针是动态的数据结构，int 等是静态的数据结构。指针的动态性还体现在可以动态地申请内存空间。例如："int *p;" 定义时，p 没有空间，但在程序需要时，可以通过 "p=new(int);" 语句向操作系统申请一个内存空间，并把地址赋值给 p。以后就可以进行 "*p=100;" 之类的操作了。

指针的几个相关概念的说明如表 4.2 所示。

表 4.2

指针变量	样　例	对　比
指针定义	char *p;	char c;
指针的类型	char*	char
指针所指向的类型	char	
指针所指向的值	*p	c
指针的值（即指针所指向的内存区）	p	&c
指针本身所占据的内存区	&p	

4.2 指针的引用与运算

一般的，我们可以这样看指针变量 (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

表4.3

	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。所有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4.2】输入N个整数，使用指针变量访问输出。

程序如下：

```
1  //eg4.2
2  #include <iostream>
3  using namespace std;
4  int a[100],N;
5  int main()
6  {
7      cin >> N;
8      for(int i=0; i<N; i++)
9          cin >> a[i];
10     int * p = &a[0];
11     for(int i=0; i<N; i++)
12     {
13        cout << *p <<endl;
14        p++;
15     }
16     return 0;
17 }
```

```
输入：
4
2 1 6 0
运行结果：
2 1 6 0
```

说明：程序第10句定义了指针变量int* p，初始化为数组的开始元素

地指针指向指针的指针——多重指针。

【例4.4】两重指针使用样例。

程序如下：

```
1  //eg4.4
2  #include <iostream>
3  using namespace std;
4  int a=10;
5  int *p;
6  int **pp;
7  int main()
8  {
9        p=&a;
10       pp = &p;
11       cout <<a<<"="
12       <<*p<<"="
13       << **pp<<endl;
14       return 0;
15 }
```

运行结果：

```
10=10=10
```

说明：程序第5句定义了一个指针变量p，第6句定义了一个2重指针变量pp。程序第9句使p指向了a，第10句使pp指向了p。第13句的**p通过2次间接访问了a变量的值10。

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，这个强大功能将在后面专门介绍。

4.3　指针与数组

1. C++中数组名在一定意义上可以看成是指针

【例4.5】用数组名访问数组。

程序如下：

```
1  //eg4.5
2  #include <iostream>
3  using namespace std;
4  int a[]={10,11,12,13,14,15};
5  int *p=a+4;
```

运行结果：

```
10
13
15
```

```
6  int main()
7  {
8       cout <<*a<<endl;
9       cout <<*(a+3)<<endl;
10      cout << *(++ p) <<endl;
11      return 0;
12 }
```

说明：程序第5、8、9句直接拿a当指针用，a指向数组的开始元素。

注意：

（1）第10句表明真正的指针变量p是变量，可以变。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。

（2）这也是称指针是动态数据结构、数组是静态数据结构的又一个体现。

其实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。

【例4.6】scanf使用数组名。

程序如下：

```
1  //eg4.6
2  #include <stdio.h>
3  using namespace std;
4  int a[5];
5  int main()
6  {
7    for(int i=0; i<5; i++)
8      scanf("%d", a+i);
9      for(int i=0; i<5; i++)
10         printf("%d",a[i] );
11         return 0;
12   }
```

运行结果：
输入：1 2 3 4 5
输出：1 2 3 4 5

说明：

（1）程序第8句直接拿a当指针用，a+i是指向数组的第i个元素的指针。

（2）第10句也可写成：

```
printf( "%d " ,*(a+i) );
```

2. 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4.7】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[0]+a[i]+a[2]+...+a[i]，即b[i]是a的前i+1个元素的和。

程序如下：

```
1  //eg4.7
2  #include <iostream>
3  using namespace std;
4  int N;
5  int *a;
6  int main()
7  {
8    cin >> N;
9    a = new  int [ N ] ;
10   for(int i=0; i<N; i++)
11        cin >> a[i];
12   for(int i=1; i<N; i++)
13        a[i]+=a[i-1];
14   for(int i=0; i<N; i++)
15        cout << a[i]<<" ";
16   return 0;
17 }
```

运行结果：
输入：5
1 2 3 4 5
输出：1 3 6 10 15

说明：

（1）程序第5句定义的是指针变量a，后面直接当数组名使用。

（2）第9句是新知识：向操作系统申请了连续的N个int型的空间。

动态数组的优点：在OI比赛中，对于大数据可能超空间的情况是比较纠结的事，用小数组只能得部分分，大数组可能爆空间(得0分)。使用这种“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4.8】行列转换。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…100000]，有K个位置有数据，K的范围是[1…100000]。

矩阵输入的方式是从上到下（第1行到第N行）、从左到右（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，范围都是[1..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。

输入样例：

```
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
```

输出样例：

```
73 12 56 100 34 23 56 78 55
```

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)的空间复杂度（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。

程序如下：

```
1  //eg4.8
2  #include <iostream>
3  using namespace std;
4  const int maxN = 100001;
5  int N ,M ,K;
6  int x[maxN],y[maxN], d[maxN];
7  int c[maxN];        //每列的数据个数
8  int *a[maxN];       //每列一个指针，准备申请“数组”
9  int main()
10 {
11   cin >> N >>M >>K;
12   for(int i=0; i<K; i++)
13   {                    //x[i] 和 y[i] 是第 i 个数据所在行号和列号
14     cin >> x[i] >>y[i] >>d[i];
15     c[y[i]]++;                  //统计 c 数组中每列的数据个数
16   }
17   for(int i=1; i<=M; i++)
18     a[i]=new int[c[i]];         //第 i 列指针申请“数组”空间

19   for (int i=0; i<K; i++)       //收集 K 个数据到相应的列
20   {
21     *a[y[i]]=d[i];              //数据放在相应列的数组中
22     a[y[i]]++;                  //数组指针移动到下一个位置
23   }
24   for(int i=1; i<=M; i++)       //列优先
25   {
26     a[i]=a[i]-c[i];             //指针回到每列的前面
27     for(int j=0;j<c[i];j++;a[i]++)
28       cout << *a[i] <<' ';
29   }
30   return 0;
31 }
```

说明：

（1）程序第8句定义了一个指针数组，a[i]表示第i列的指针。

（2）程序第6、7、8句定义了固定大小为maxN的数组，占用空间大小为5*4*maxN字节，约2M。还可以在第12句输入N、M后，再申请“动态数组”，当N，M比较小时，占用更小的空间。

（3）特别的，可以将指针当数组名用，即第21句可以写成：

地址，因此*p就是a[0]。第14句p++，它的意思是“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。

类似地：

（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。

注意：*p+3 和 *(p+3) 是不同的。 如果 p=&a[0]，*p+3 相当于a[0]+3，*(p+3)相当于a[3]。

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4.3】无类型指针使用样例。

程序如下：

```
1  //eg4.3
2  #include <iostream>
3  using namespace std;
4  int a=10;
5  double b=3.5;
6  void *p ;
7  int main()
8  {
9      p=&a;
10     cout << *(int*)p<<endl;
11     p=&b;
12     cout <<*(double *)p<<endl;
13     return 0;
14 }
```

运行结果：

```
10
3.5
```

说明：程序第9句p=&a，就是地址赋值，没有问题。第10句输出时，就必须明确p指向的空间存放的数据类型，类型不一样不仅空间大小不相同，存储的格式也不同。比如把第12句的*(double *)p改成*(long long *)p，输出结果将是：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

```
a[ y[i] ][0] = d[i];
```

4.4 指针与字符串

C语言没有string类，C的字符串就是字符数组，并以‘\0’为字符串结束符。这种字符串称为C风格字符串，由于是底层操作，相关的函数在编程中有特别的地位，有很多经典的指针用法。

在《CCF中学生计算机程序设计入门篇》中，已经介绍了字符数组的一些使用方法，下面作些回顾，并使用指针对其操作。

先通过表4.4来了解一下C字符串的定义和初始化。

表4.4

例 子	说 明
char s[10];	最大可以存放长度是9的字符串，因为字符串最后要加‘\0’，切记。例如：s[0]= ‘a’ ; s[1]= ‘b’ ; s[2]=0; s就是一个字符串“ab”。
char s[]={‘a’, ‘b’, ‘c’, ‘\0’ };	定义s并且初始化为一个字符串“abc”
char s[]= “abc” ;	同上，但更简洁

由于数组是静态的，一旦定义，大小就确定了，编程时要注意这点，使用时不能超出数组长度。C风格字符串可以初始化，但不可以直接赋值，“s="abcd";”是非法的，要使用strcpy函数拷贝才可以。

【例4.9】C语言的字符串编程示例。

程序如下：

```
1  //eg4.9
2  #include <cstdio>
3  #include <cstring>
4  using namespace std;
5  char a[100], b[100];          // 定义两个字符数组
6  int main()
7  {
8    strcpy(a,"abcd");           // 把字符串 "abcd" 拷贝到 a 中
9    printf("%s,len=%d\n",a,strlen(a));
                                 // 打印字符串 a 和它的长度
```

```
10   scanf("%s",b);                  // 读入字符串b，也可以用cin>>b;
11   int cmp = strcmp(a,b);          // 比较2个字符串的大小
12   if(cmp==0)                      // 相等
13     printf("%s=%s",a,b);          // 也可以用cout<<a<<"="<<b<<endl;
14   else if(cmp<0)                  // 小于
15     printf("%s<%s\n",a,b);
16   else                            // 大于
17     printf("%s>%s",a,b);
18   if(strstr(a,b)!=NULL)           // 查找子串
19     printf("%s is substr of %s\n",b,a);
20   return 0;
21 }
```

说明：

（1）程序中使用了C的字符串常见的函数：赋值strcpy、求长度strlen、比较大小strcmp、查找子串strstr等函数。

（2）这些函数都要使用#include<cstring>或#include<string.h>。为了进一步展示指针操作的广泛性，下面模拟这些函数在cstring库中的实现方法。

```
char * strcpy(char *dest ,const char *src)      // 字符串拷贝
{
  char *p=dest;
  while (*src != '\0')
  {
     *dest = *src;
     dest++;src++;
  }
  *dest = '\0';
  return p;
}
```

```
size_t strlen(const char * str)        // 字符串长度
{
    const char *cp =  str;             //str是开始指针
    while(*cp++ )                      // 找到'/0'的位置
          ;
    return (cp - str - 1 );            // 计算长度
}
```

```
int strcmp(const char * src, const char * dst)
                                                    // 字典序比较两字符串大小
{
    int ret = 0 ;
    while(!(ret=*src-*dst)&&*dst)                   // 相等并且没有结束
          ++src, ++dst;
    return( ret );
}
```

```
char * strstr ( char *buf, char *sub)  // 简单方法查找子串
{
     if(!*sub)                                      // 子串是空的特殊情况
        return buf;

     char *bp, *sp;                                 // 扫描匹配的指针
     while (*buf)
     {
        bp = buf;
        sp = sub;
        do
        {
            if(!*sp)                                // 子串匹配完成
                return buf;                         // 返回主串的位置
        } while (*bp++ == *sp++);
        buf ++;                                 // 从主串的下一个位置开始重新匹配
     }
     return 0;
}
```

这些函数的实现都是指针操作！虽然也可以用数组操作实现，但数组存取要通过下标计算内存的位置，效率低些。

仔细研究体会上面的几个经典函数的编程，会对指针的理解更深刻。

*4.5 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4.10】使用函数指针调用函数示例。

程序如下：

```
1  //eg4.10
2  #include<iostream>
3  using namespace std;
4  int test(int a);
5  int main()
6  {
7       cout<<test<<endl;     // 显示函数地址
8       int (*fp)(int a);
9       fp=test;              // 将函数 test 的地址赋给函数指针 fp
10      cout<<fp(5)<<","<<(*fp)(10)<<endl;
11      // 上面的输出 fp(5)，这是 C++ 的写法，(*fp)(10) 这是兼容 C
           语言的标准写法
12 }
13 int test(int a)
14 {
15   return a;
16 }
```

运行结果：
输出：1
5,10

说明：

（1）程序第8句定义了一个函数指针变量fp。

（2）程序第9句将函数test的地址赋给函数指针fp。

（3）程序第10句是使用fp来调用函数的两种方式。

函数指针的基础操作有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，可以看成是指针。

（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4.11所示。

【例4.11】使用typedef定义函数指针示例。

程序如下：

```
1  //eg4.11
2  #include<iostream>
3  using namespace std;
4  int add(int,int);
5  typedef  int(*addP)(int,int);
6  int main()
7  {
8      addP fp=add;
9      cout<<fp(2,5)<<endl;
10 }
11 int add(int a,int b)
12 {
13   return a+b;
14 }
```

运行结果：
输出：7

说明：

（1）程序第5句定义了一个函数指针变量类型addP。

（2）程序第8句定义了一个addP类型的函数指针fp，并赋值为add。

（3）程序第9句是使用fp来调用函数，实参为2和5，调用第11句的add函数，输出返回值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4.12】使用函数指针数组，模拟菜单功能实现方法示例。

程序如下：

```
1  //eg4.12
2  #include<iostream>
3  using namespace std;
4  void t1(){ cout << "test1";}
5  void t2(){ cout << "test2";}
6  void t3(){ cout << "test3";}
7  void t4(){ cout << "test4";}
8  void t5(){ cout << "test5";}
9  typedef  void(*Fp)();
10 int main()
11 {
12   Fp menu[] = {t1,t2,t3,t4,t5};
13   int select;
14   cin >> select;
15   menu[select]();
16   return 0;
17 }
```

说明：

（1）程序9句定义了一个函数指针变量类型Fp。

（2）程序第12句定义了一个Fp类型的函数指针数组menu，并初始化。

（3）程序第15句是使用menu[select]()来调用选择的函数。

4.6　指针的扩展

4.6.1　指针与引用

在C++中有一种新的复合类型——引用变量，它和指针变量的引用有

点相似，下面看一个样例。

【例4.13】使用指针变量和引用变量对比示例。

程序如下：

```
1  //eg4.13
2  #include<iostream>
3  using namespace std;
4  int main()
5  {
6        int a=10,b=20;
7        int *f ;
8        int &ra=a;                    //ra 相当于 a 的别名
9        f=&a;
10       cout << "a="<<*f<<endl;
11       ra+=5;
12       cout << "a="<<*f<<endl;
13       return 0;
14 }
```

运行结果：
输出：
a=10
a=15

说明：

（1）程序第7句定义了一个指针变量f。

（2）程序第8句定义了一个引用变量ra。ra相当于是a的别名，使用ra就是使用a。

（3）ra必须在定义时初始化，以后不可以通过ra=b之类命令改变为变量b的引用，因为这个语句表示把b的值赋值给ra（即20）。

（4）程序中a,*f, ra都是指向同一个内存地址。

在程序中，适当使用别名代替多重数组引用时会使程序代码简洁。比如，递推式：

```
f[a[i][j]] = (f[a[i][j]-1] + f[a[i][j]-2])*a[i][j];
```

用引用变量：

```
int &k = a[i][j];
```

让k作为a[i][j]的别名，上式简化为

```
f[k]=(f[k-1]+f[k-2])*k;
```

以前学习的函数参数一般是一种按值传递方式（即传递变量值的一个拷贝），函数对参数变量的操作与调用程序中变量无关。要避开按值传递

的限制，就要采用按指针传递的方式（比如数组）。

现在函数可以使用引用参数，使得函数中的变量名成为调用程序中的变量的别名。这种传递参数的方法称为按引用传递。函数对参数的操作就相当于对原来变量的操作。

【例4.14】函数使用引用参数示例。

程序如下：

```
1  //eg4.14
2  #include<iostream>
3  using namespace std;
4  void swapv(int a,int b);//变量参数
5  void swapp(int*fa,int*fb);    //指针参数
6  void swapr(int&a,int&b);//引用参数
7  int main()
8  {
9    int a=10,b=20;
10   swapv(a,b);                  //没有影响变量a,b
11   cout <<a<<","<<b<<endl;
12   swapp(&a,&b);                //交换了变量a,b
13   cout <<a<<","<<b<<endl;
14   swapr(a,b);                  //交换了变量a,b
15   cout <<a<<","<<b<<endl;
16   return 0;
17 }
18 void swapv(int a,int b)
19 {
20   int temp= a;
21   a=b;
22   b=temp;
23 }
24 void swapp(int *fa, int *fb)
25 {
26   int temp;
27   temp = *fa;
28   *fa=*fb;
29   *fb=temp;
30 }
31 void swapr(int & a,int & b)
32 {
```

运行结果：

输出：

10,20

20,10

10,20

```
33    int temp=a;
34    a=b;
35    b=temp;
36 }
```

说明：

（1）函数swapv使用普通变量参数，函数内（第18~23句）的a,b操作不影响主程序的变量，是错误的编写方法。

（2）函数swapp是用指针参数的方式实现的，虽然指针是传值的不能改变（fa指向a，fb指向b），但可以通过间接访问指针地址，修改指向的地址里面的值，间接修改了主程序的变量。

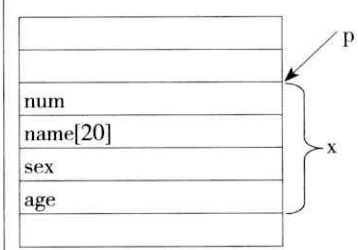
（3）函数swapr使用了引用参数，局部变量（程序31~36行）a,b就是主程序传来的变量的别名，修改它们就直接修改了主程序的变量，传递、调用都特别简单方便。

4.6.2 指向结构体变量的指针

一个结构体变量的指针就是该变量所占据的内存段的起始地址。可以设一个指针变量，用来指向一个结构体变量，此时该指针变量的值是结构体变量的起始地址。

例如：

```
struct Tstudent
{
    int num;
    char name[20];
    char sex;
    int age;
};
Tstudent x;
Tstudent *p;
```



【例4.15】使用指针变量访问结构体成员示例。

```
1  //eg4.15
2  #include<iostream>
3  using namespace std;
4  struct Tstudent
5  {
6      int num;
7      char name[20];
```

运行结果：
输出：
13
lihao
35

```
8       char sex;
9       int age;
10 };
11 Tstudent x={13,"lihao",'m',35};
12 Tstudent *p;
13 int main()
14 {
15      p=&x;
16      cout<<x.num<<endl;
17      cout<<(*p).name<<endl;
18      cout<<p->age<<endl;
19      return 0;
20 }
```

说明：

（1）程序第11句定义了一个结构体变量x，并初始化。

（2）程序第12句定义了一个结构体Tstudent的指针变量p。

（3）指针访问结构体成员有两种方式：

①(*p).name，相当于x.name。不能写成*p.name，这个语法上相当于*(p.name)。

②p->name，也相当于x.name。比较方便。

由于函数变量参数通常按值传递，如果传递一个结构体变量就会要复制整个结构体，效率不高，这时一般都使用指针或引用。

本章小结

指针变量的定义和使用	
知识点	举例说明
取地址运算符 &	int a=10; cout <<&a<<endl; //输出a的地址

定义指针变量
类型 *指针变量;
或
类型 *指针变量=地址;
	定义指针变量并初始化
	int a=10;
	int *p =&a;

指针的引用
*指针变量名
	int a=10;
	int *p=&a;
	cout << *p <<endl; //输出10

续表

指针的+、-运算
说明：p++ 指针地址不是增加1，是增加一个“类型”单位，到达下一个“元素”
	int a[ ] = {0,2,4,6,8,10,12};
	int *p=a;
	cout <<*(p+3); //输出结果是6

指针的差运算
说明：两个指针的差不是地址的差，是中间“类型”单位的差，即中间的“元素”个数
	int a[] = {0,10,20,30,40,50,60};
	int *p=&a[2];
	int *q=&a[5];
	cout <<q-p; //结果是3

数组名可看成常量指针
指针可看成数组名
	int a[] = {0,1,2,3,4,5,6};
	int *p=a+2; //a 当指针，p指向2
	cout <<*(a+4)<<”,” //输出4，a当指针
	<<p[-1]<<endl; //输出1，p当数组名

C 风格的字符串就是字符数组，常见函数有：strlen、strcpy、strcmp、strstr等	C风格字符串的操作通常都是用指针实现，具体见前面章节
函数指针	
知识点	举例说明
函数指针定义方式和函数声明类似，只是多了个*号	int (*fp)(char); 定义了一个函数指针fp，函数声明是：有一个字符参数，返回一个整数

可以结合typedef先定义一个类型，再定义变量	typedef int (* Tp)(int); //定义类型Tp
	Tp fp ; //定义函数指针变量fp

获取函数的地址 取值函数名即可	int test(int a) { return a; } fp = test;
使用指针调用函数： *指针变量； C++可以直接使用： 指针变量	cout<<(*fp) (5)<<endl; cout << fp(6)<<endl;
函数指针数组：可以用下标直接调用相应的函数	在菜单等编程中使用较多
引用和结构体指针	
知识点	举例说明
引用是C++引入的新类型	它和指针的引用相似，但有更多的特性，本章只是作简单地介绍
定义并初始化 相当于变量的别名 必须在定义时初始化	int a; int & r = a;

函数的引用参数
类似传递了指针，更简单
	void swapr(int &a, in t &b)
	{
	int temp =a; a=b; b=temp;
	}

续表

结构体指针变量
定义方法和简单类型的指针一样
	struct Tp{
	  int num;
	  char name[20];
	};
	Tp *p;

结构体成员引用方式 （1）(*指针变量名).成员名 （2）指针变量名->成员名	(*p).num = 11; p->num+=3;

练　习

（1）有一道OI题目，内存限制64M，需要输入N行、M列的一个整数(int型)数组，翻转后输出。已知N*M范围是[1..1000000]，一个int型整数需要4字节空间，因此数组最大空间只要4M。请问读入N和M后，怎样利用指针开一个恰当的二维数组？

（2）有一道OI题目，需要对10000个[-100,100]之间的整数计数排序，Pascal语言爱好者经常因为C++语言的数组下标不能是负数而“抱怨”C++，请思考利用“指针可以当成数组名”的性质，编写个C++程序可以用“下标为负数的数组”来计数排序。

（3）若有以下定义，则说法错误的是(　)。

```
int a=100,*p=&a
```

（A）声明变量p，其中*表示p是一个指针变量

（B）变量p经初始化，获得变量a的地址

（C）变量p只可以指向一个整形变量

（D）变量p的值为100

（4）若有以下定义，则赋值正确的是(　)。

```
int a ,b , *p;
float c, *q;
```

（A）p=&c　　（B）q=p　　（C）p=NULL　（D）q=new int

（5）如果x是整型变量，则合法的形式是(　)。

（A）&(x+5)　（B）*x　　（C）&*x　　（D）*&x

（6）若有语句int a[10]={0,1,2,3,4,5,6,7,8,9},*p=a；则(　)不是对a数组元素的正确引用（其中0≤i<10）。

（A）p[i]　　（B）*(*(a+i))　（C）a[p-a]　　（D）*(&a[i])

（7）以下程序的输出结果是(　)。

```
void fun(int x,int y,int *cp,int *dp)
{
   cp=x+y;
   dp=x-y;
}
void main( )
{
   int a,b,c,d;
   a=30,b=50;
   fun(a,b,&c,&d);
   cout<<c<<","<<d<<endl;
}
```

（A）50,30　　（B）30,50　　（C）80,-20　　（D）80,20

（8）设有如下定义，下面关于ptr正确叙述是(　)。

```
int (*ptr)();
```

（A）ptr是指向一维数组的指针变量

（B）ptr是指向int 型数据的指针变量

（C）ptr是指向函数的指针，该函数返回一个int型数据

（D）ptr是一个函数名，该函数的返回值是指向int型数据的指针

（9）相同数据类型的数组名和指针变量均表示地址，以下不正确的说法是(　)。

（A）数组名代表的地址不变，指针变量存放的地址可变

（B）数组名代表的存储空间不变，指针变量指向的存储空间长度可变

（C）A和B的说法均正确

（D）没有差别

（10）若已定义int a=5，下面对①、②两个语句的正确解释是(　)。

①int *p=&a;　　②*p=a;

（A）语句①和②中的*p含义相同，都表示给指针变量p赋值

（B）①和②语句的执行结果，都是把变量a的地址值赋给指针变量p

（C）①在对p进行说明的同时进行初始化，使p指向a；②变量a的值赋给指针变量p

（D）①在对p进行说明的同时进行初始化，使p指向a；②将变量a的值赋予*p

（11）有如下语句："int m=6, n=9, *p, *q; p=&m; q=&n;"，存储

结构如图1所示，若要实现图2所示的存储结构，可选用的赋值语句是(　　)。

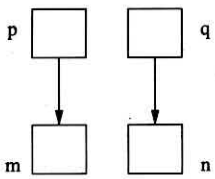


图1

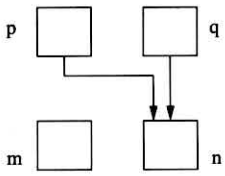


图2

（A）*p=*q;　（B）p=*q;　（C）p=q;　（D）*p=q;

（12）下列程序的输出结果是(　　)。

```
#include <stdio.h>
int main()
{
  int a[]={1,2,3,4,5,6,7,8,9,0}, *p;
  p=a;
  printf("%d\n", *p+9);
  return 0;
}
```

（A）0　（B）1　（C）10　（D）9

第5章　数据外部存储——文件

实用程序很多都要涉及数据文件操作，比如Word的doc文件、记事本的txt文件等。目前的中学OI比赛也必须使用文件作为数据输入、输出方式，掌握好相应的文件操作编程是参加比赛的必要条件。本章主要学习文本文件的读写常见方式，并针对不同的优缺点，分别以C和C++两种不同形式，介绍文件变量指针、流、读文件、写文件等编程，最后介绍比赛中使用文件时的一些技巧和优化。

5.1　数据存储的分类

前面运行程序时，我们从键盘输入数据，存放在变量中，运算结果输出到屏幕上。但有个问题：一旦重新运行，这些输入数据都要重新键盘输入、输出数据要重新计算，十分麻烦。而且对大规模数据的输入根本实现不了。把数据保存在外部存储器（相对于内存）是解决这个问题的唯一办法。

“文件”指存储在外部介质上数据的集合，简单讲就是把数据通过字节序列保存在磁盘上。在磁盘上存储、调用数据都是通过文件操作的，文件保存的方式大致有两种类型：二进制文件和文本文件。

二进制文件和文本文件有什么不同？下面举个例子说明。

数121的二进制是“1111001”，内存中就是一个字节；在源程序的文本中显示为“121”，要用3个ASCII码表示（分别为：110001,110010,110001）。如图5.1所示。

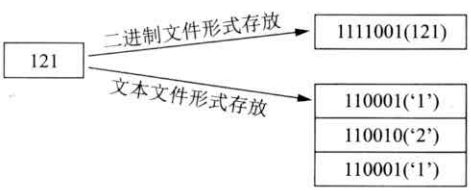


图5.1

二进制虽然效率高，不过要事先知道它的编码形式（比如：double类型的数的保存方式）才能正确解码转换，比较复杂。

文本形式的保存虽然效率不高，但可以直接按照其ASCII码翻译成文字，比较方便。信息学比赛中所有的输入输出文件都采用文本形式的文件，本章后面仅介绍文本文件的操作编程。

内存中运行的程序是怎样和磁盘上的文件打交道的呢？先用图5.2简单表示这个流程。

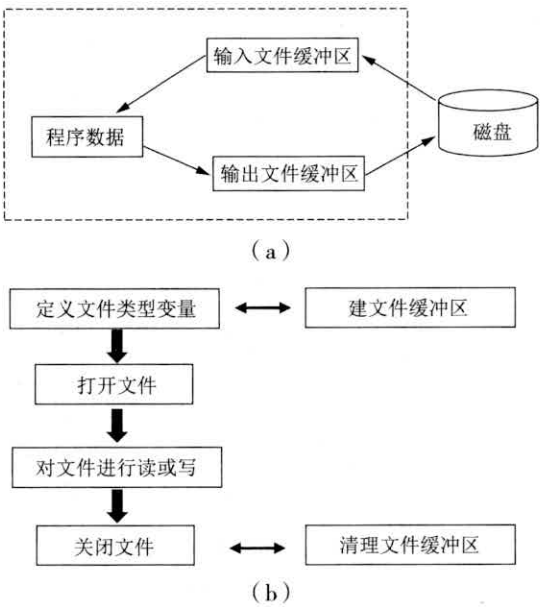


图5.2

5.2　文件类型变量的定义及引用

C++程序和文件缓冲区打交道的方式有两种：流式和I/O方式。信息学竞赛中一般使用流式文件操作。流式文件类型也分为两种：

（1）stream类的流文件。

（2）文件指针FILE。

5.2.1　stream类的流文件的操作

【例5.1】A+Bproblem。输入2个整数A和B，求它们的和A+B。

输入格式(文件ab.in)：2个整数，范围是[-1000，1000]。

输出格式（文件ab.out）：输出一个整数。

输入样例：

10 5

输出样例：

15

程序如下：

```
1  //eg5.1
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5  ifstream fin("ab.in");
6  ofstream fout("ab.out");
7  int A,B;
8  int main()
9  {
10   fin >> A >> B;
11   fout << A+B;
12   return 0;
13 }
```

说明：

（1）程序第3句包含了头文件<fstream>，即文件流。前缀“f”是“file”的缩写。

（2）程序第5句定义了一个输入流文件(ifstream)类型变量fin，初始化指向文本文件“ab.in”。

（3）程序第6句定义了一个输出流文件(ofstream)类型变量fout，初始化指向文本文件“ab.out”。

（4）程序第10句和标准输入流cin类似，用“>>”来从fin读入数据。你可以认为fin就表示图5.2的输入缓冲区。

（5）程序第11句和标准输出流cout类似，用“<<”来把数据输出到fout。你可以认为fout就表示图5.2的输出缓冲区。

注意：fin和fout只是变量名，你可以任意命名，比如：f1,f2之类的。

OI比赛要求数据文件的文件名不要带目录路径，默认在“当前目录”下，即和程序在同一文件夹里。

程序中没有关闭文件的语句（比如：fin.close(), fout.close()），不

过在程序结束时会自动关闭文件，因此我们可以在比赛中省略。

【例5.2】文件结束。已知文件中有不超过1000个的正整数，请计算它们的和（保证答案在10000以内）。

输入格式（文件sum.in）：1行，多个整数，范围是[1，1000]。

输出格式（文件sum.out）：输出一个整数。

输入样例：

```
10 5 6 9
```

输出样例：

```
30
```

分析：做早期的OI题目时会出现由于不知道数据个数要程序判断是否文件结束的情况。

程序如下：

```
1  //eg5.2_1
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5  ifstream fin("sum.in");
6  ofstream fout("sum.out");
7  int x, sum;
8  int main()
9  {
10   sum=0;
11   while(fin >> x )          // 可以读到数据就意味着文件没有结束
12       sum+=x;
13   fout << sum<<endl;
14   return 0;
15 }
```

说明：程序第11句中利用“fin>>x”如果失败就返回假的特点判断文件是否结束。

还可以利用eof()函数来判断是否文件结束（end of file），参考程序如下：

```
1  //eg5.2_2
2  #include <iostream>
3  #include <fstream>
```

```
4  using namespace std;
5  ifstream fin("sum.in");
6  ofstream fout("sum.out");
7  int x, sum;
8  int main()
9  {
10   sum=0;
11   while(!fin.eof())        // 没有文件结束就循环
12   {
13       fin >> x;
14       sum+=x;
15   }
16   fout << sum<<endl;
17   return 0;
18 }
```

OI比赛中使用上面的文件操作方法就基本可以了，作为扩展知识我们再看一个例子。

【例5.3】数据生成。为了准备今年的NOIP，老师让大牛出个题目。大牛经过努力，终于完成了例5.1的A+Bproblem的题目描述，可出测试数据时，他有个新任务：编一个程序，从键盘输入文件名和数据，自动生成数据文件。

分析：由于文件名要从键盘多次输入，文件变量不能初始化，要使用open函数，关闭时要使用close函数。

程序如下：

```
1  //eg5.3
2  #include <iostream>
3  #include <fstream>
4  #include <string>
5  using namespace std;
6  ofstream fout;
7  string fname,f1,f2;
8  int A,B;
9  int main()
10 {
11   for(int i=0; i<5; i++)
12   {
```

```
13      cout <<" 请输入文件名: ";
14      cin >> fname;          // 键盘输入文件名
15      cout <<" 请输入 A 和 B 的值 :";
16      cin >> A >> B;         // 键盘输入整数 A 和 B
17      fout.open((fname+".in").c_str());
                               // 打开输入文件 (.in), 输出 A 和 B
18      fout << A<<" "<<B <<endl;
19      fout.close();
20      fout.open((fname+".out").c_str());
                               // 打开输入文件 (.out), 输出 A+B
21      fout << A+B <<endl;
22      fout.close();
23    }
24    return 0;
25 }
```

说明：

（1）程序第6句定义了一个输入流文件变量fout。由于文件名从键盘输入，不能在定义ifstream文件变量时初始化。

（2）程序第17、20行中使用成员函数open打开文件，相当于图5.2中文件的缓冲区对应到磁盘文件中。由于open的参数是C风格的字符数组，string类型要使用c_str()函数转换。

（3）程序第19、22使用close函数把文件关闭，即把图5.2输出缓冲区的内容copy到文件中。注意：例5.1中没有使用clos是因为程序运行结束时会自动close文件。

5.2.2 文件指针FILE的操作

C++还提供了一种FILE文件结构指针类型，FILE是在<cstdio>或<stdio.h>里定义的，使用时要包含这个库。

【例5.4】排序sort。输入N个不超过1000000的正整数，请把他们递增排序后输出。

输入格式（文件sort.in）：第1行，一个整数N，范围是[1,1000000]；第2行，N个整数，范围是[1,1000000]。

输出格式（文件sort.out）：输出排序后的N个整数。

输入样例：

5

20 10 5 6 9

输出样例：

5 6 9 10 20

分析：本题的数据输入输出量很大，要考虑数据读入、输出效率。

下面程序使用效率高的fscanf和fprintf来实现输入输出，需要使用文件结构FILE指针的变量。

程序如下：

```
1  //eg5.4_1
2  #include <cstdio>
3  #include <algorithm>
4  using namespace std;
5  FILE *fin,*fout;
6  int N, a[1000001];
7  int main()
8  {
9    fin = fopen("sort.in","r"); // 打开一个输入文件
10   fout =fopen("sort.out","w");       // 打开一个输出文件
11   fscanf(fin, "%d",&N);              // 从文件流 fin 读入数据
12   for(int i=0; i<N; i++)
13        fscanf(fin, "%d",&a[i]);
14   sort(a,a+N);
15   for(int i=0; i<N; i++)
16        fprintf(fout,"%d ",a[i]);     // 输出数据到文件流 fout
17   return 0;
18 }
```

说明：

（1）程序第5句定义了两个FILE *类型的变量，这里习惯上还是使用了fin，fout作为变量名。

（2）程序第9、10句使用fopen函数打开相应文件。参数中的文件名分别是“sort.in”和“sort.out”，需要注意的是第2个参数：“r”表示以只读（read）方式打开文件；“w”表示以只写（write）方式打开文件。

（3）程序使用了fscanf()读入数据和fprintf( )输出数据。使用方式基本和格式读入scanf、格式输出printf一样，只是多了个文件指针参数。

这种方式的文件读入判断文件是否结束使用函数feof（文件指针变

量），返回值是真假。例如：

```
1  //eg5.4_2
2  #include<cstdio>
3  using namespace std;
4  FILE *fin,*fout;
5  int x, sum;
6  int main()
7  {
8    fin = fopen("sum.in","r");
9    fout = fopen("sum.out","w");
10   sum=0;
11   while(!feof(fin))              // 文件没有结束
12   {
13       fscanf(fin,"%d",&x);
14       printf("%d",x),sum+=x;
15   }
16   fprintf(fout,"%d\n",sum);
17   return 0;
18 }
```

另外这种文件操作还有一些常见函数，比如：

（1）读入字符函数：fgetc()。

（2）写入字符函数：fputc()。

（3）读入字符数组函数：fgets()。

（4）写入字符数组函数：fputs()。

5.3 文件的重定向

OI比赛中，文件功能比较单一，通常只需要同时打开一个输入文件和一个输出文件，因此文件的操作可以使用一种方便但特殊的方法：输入输出文件重定向。

标准输入、输出概念

cin或scanf使用的输入设备是键盘（控制台），也称为标准输入：stdin。

cout或printf使用的输出设备是显示器（控制台），也称为标准输出：stdout。

C++语言可以使用freopen函数把stdin和stdout重新定向到相关的文件，使原来的标准输入、输出变成了文件输入、输出。

由于兼容性等历史问题，文件操作时，C++的cin,cout要保证与prinf,scanff同步，cin、cout效率会降低。虽然有通过关闭同步功能（ios::sync_with_stdio(false);）来提高效率的函数，但在不同的C++编译器中，表现的结果不一致，比赛中使用它并不完全可靠，要针对测试环境进行测试才可用。因此，在题目中有大规模数据输入输出时，建议使用scanf和printf。

【例5.5】反向输出reverse。输入N个不超过1000000的正整数，请把它们逆向输出。

输入格式（文件reverse.in）：第1行，一个整数N，范围是[1，100000]；第2行，N个整数，范围是[1，1000000]。

输出格式（文件reverse.out）：N个整数。

输入样例：

```
5
20 10 5 6 9
```

输出样例：

```
9 6 5 10 20
```

程序如下：

```
1  //eg5.5_1
2  #include <cstdio>
3  #include <algorithm>
4  using namespace std;
5  int N, a[1000001];
6  int main()
7  {
8    freopen("reverse.in","r",stdin);
                          // 重定向只读文件 reverse.in 到 stdin
9    freopen("reverse.out","w",stdout);
                          // 重定向只写文件 reverse.out 到 stdout
10   scanf("%d",&N);    // 使用标准输入 scanf
11   for(int i=0; i<N; i++)
12       scanf("%d",&a[i]);
13   for(int i=N-1; i>=0; i--)
```

```
14        printf("%d",a[i]); // 使用标准输出 printf
15    return 0;
16 }
```

说明：

（1）程序第8句重定向只读文件reverse.in到stdin。

（2）程序第9句重定向只写文件reverse.out到stdin。

（3）程序后面的只要使用标准输入、标准输出就相当于对读写文件操作。

提示：如果不考虑输入速度问题，使用stream也可以同样重定向输入输出文件。参考程序如下：

```
1  //eg5. 5_2
2  #include<fstream>
3  #include<iostream>
4  #include <algorithm>
5  using namespace std;
6  int N, a[1000001];
7  int main()
8  {
9    freopen("reverse.in","r",stdin);
                          // 重定向只读文件 reverse.in 到 stdin
10   freopen("reverse.out","w",stdout);
                          // 重定向只读文件 reverse.in 到 stdin
11   cin>>N;              // 使用标准输入 cin
12   for(int i=0;i<N; i++)
13       cin >> a[i];
14       for(int i=N-1;i>=0; i--)
15       cout <<a[i]<<" "; // 使用标准输出 cout
16   return 0;
17 }
```

本章小结

文件操作知识点	举例说明
文件	指存储在外部介质上数据的集合。简单讲就是把数据通过字节序列保存在磁盘上
文件名	磁盘操作系统是通过文件名管理文件的

续表

文件操作知识点	举例说明
文件目录	文件在磁盘的位置
文件大小、只读、只写、扩展名等	文件的属性，详细内容略
文本文件	使用ASCII码保存内容
<fstream>类的ifstream、ofstream变量	类似cin、cout的操作
fstream类型文件的初始化和open()函数	ifstream fin(“ab.in”); f.open(文件名参数);
fstream类型文件结束判断	f.eof()函数或读入失败
文件指针(FILE *)变量	类似scanf、printf的操作
文件指针的fopen函数	fopen(文件名参数，文件属性参数) 注：文件属性参数有r、w等
标准输入	stdin
标准输出	stdout
文件重定向 freopen(文件名，属性r/w，stdin/stdot);	freopen(“ab.in”,”r”，stdin); freopen(“ab.out”,”w”，stdout);
备注： (1)如果比赛中只有一个输入文件和一个输出文件，可以使用freopn重定向方法 (2)如果数据比较多(比如多于100000个整数)，建议使用scanf,printf	

练　习

(1)速度大比拼。信息学比赛中有时输入数据很大，数据文件的读取效率就要重视了。一般地，使用scanf、prinft比使用cin、cout速度快。请根据下面的测试表，重新设计方案，在机器上用你的C++版本对各种方式文件读写进行测试。

文件读取操作速度对比表

(1)测试环境：i7+12G+固盘+devcpp5.6.1

读入方式	使用时间(毫秒)			
	10^5个整数	10^6个整数	10^7个整数	10M字符串
ifstream + fin	16	103	1001	53
FILE * +fscanf	55	345	3366	63

续表

读入方式	使用时间（毫秒）			
	10^5个整数	10^6个整数	10^7个整数	10M字符串
freopen+cin	67	440	4378	472
freopen+cin +sync_with_stdio(false)	11	103	1024	51
freopen+scanf	32	318	3112	60

（2）测试环境：i2+2G+普通硬盘+devcpp4.9.9.2

读入方式	使用时间（毫秒）			
	10^5个整数	10^6个整数	10^7个整数	10M字符串
ifstream + fin	65	982	6148	361
FILE * +fscanf	53	692	4829	170
freopen+cin	727	6449	62738	4193
freopen+cin +sync_with_stdio(false)	72	673	6538	375
freopen+scanf	55	511	4574	162

注：freopen+cin是很差的组合，数据多时要加“iso:: sync_with_stdio(false)”关闭同步。

参考程序如下：

```
// 测试文件速度程序
#include <fstream>
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <ctime>
#include <string>
using namespace std;
char s[10000005];
int x[10000005];
void testFstream(string fname, int N)
{
     ifstream fin(fname.c_str());
     fin>>N;
     for(int i=0; i<N; i++)
          fin >>x[i];
     fin.close();
}
```

```
void testFstream_str(string fname, int N)
{
     ifstream fin(fname.c_str());
//   fout <<N<<endl;
     fin >> s;
     fin.close();
}
int main()
{
     //cerr(CLOCKS_PER_SEC);
     srand(time(0));
     int st,et;

     st=clock();
     testFstream("d1.in",100000);
     et = clock();
     cout <<"10^5 integer:"<<et-st<<endl;

     st=clock();
     testFstream("d2.in",1000000);
     et = clock();
     cout <<"10^6 integer:"<<et-st<<endl;

     st=clock();
     testFstream("d3.in",10000000);
     et = clock();
     cout <<"10^7 integer:"<<et-st<<endl;

     st=clock();
     testFstream_str("d4.in",10000000);
     et = clock();
     cout <<"10^7 char:"<<et-st<<endl;
     system("pause");
   return 0;
}
```

（2）极致速度。信息学比赛中经常输入数据很大，数据文件的读取效率就要充分重视了。一般来讲，使用scanf读入就可以了，但有时还想进一步压缩读入数据的时间，可以使用下面的方法，自己编写函数，用getc读字符，“组装”成整数。

参考程序如下：

```
//测试文件速度程序
#include <fstream>
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <ctime>
#include <string>
using namespace std;
int x[10000005];

void scanf_getd(int & x)
{
   char c;                                          //跳过非数字字符
   for(c=getc(stdin); c<'0' || c>'9';c=getc(stdin));
                                                    //收集整数
   for(x=0; c>='0' && c<='9'; c=getc(stdin))
       x=x*10+c-'0';
}
void testFile(string fname, int N)
{
   freopen(fname.c_str(),"r",stdin);
   scanf_getd(N);
   for(int i=0; i<N; i++)
     scanf_getd(x[i]);

   fclose(stdin);
}

int main()
{
   srand(time(0));
   int st,et;

   st=clock();
   testFile("d1.in",100000);
   et = clock();
   cout <<"10^5 integer:"<<et-st<<endl;
```

```
    st=clock();
    testFile("d2.in",1000000);
    et = clock();

    cout <<"10^6 integer:"<<et-st<<endl;
    st=clock();
    testFile("d3.in",10000000);
    et = clock();
    cout <<"10^7 integer:"<<et-st<<endl;

    while(1);
    return 0;
}
```

运行结果：

```
10^5 integer:6
10^6 integer:48
10^7 integer:449
```

与（1）的程序相比，速度几乎快了10倍！

第6章　数据结构及其运用

在前面的章节里，我们学习了数组及其运用。数据结构则是计算机储存、组织数据的方式。特定的数据结构中的数据元素往往具有一种或多种特定关系。通常情况下，精心地选择数据结构可以带来更高的程序运行或者储存效率。打个比方，如果把程序比喻成一台机器的话，算法是机器的运行原理，数据结构就是机器上真正运作的一个个小部件。只有当你精心地选择和使用它们，你的程序才会跑得更快。接下来的章节里，我们会介绍线性表、队列和栈，以及二分和快速排序，希望读者能好好掌握。

6.1　什么是数据结构

一般来说，用计算机解决一个具体问题时，大致需要经过下列几个步骤：首先要从具体问题抽象出一个适当的数学模型，然后设计算法解决这个数学模型，最后编写程序、进行调试、调整直至得到最终解答。寻求数学模型的实质是分析问题，从中提取操作的对象，并找出这些操作对象之间含有的关系，然后用数学语言加以描述。这些操作的对象及其之间的关系在编写程序时需要被存储起来，这就要用到数据结构。

下面举几个例子让大家更好地理解数据结构。

【例6.1】食堂排队打饭，有人进来选择一个窗口排在队尾，每完成一个人，这个人就会从队首离开。如果要编写程序模拟这个过程，需要处理的对象是人，在队伍里面，除了队首每一个人都有前一个人，除了队尾每一个人都有后一个人，排队就插在队尾的后面成为新的队尾，打完饭队首就离开，下一个人成为新的队首。这些对象之间存在一种最简单的线性关系，这类数学模型可以称作是线性的数据结构。

【例6.2】计算机的目录结构，根目录是“计算机”，下面有“C盘”、“D盘”、“E盘”这些子目录，子目录下面还可以继续扩展其对应的子目录或文件。这里的对象是目录或文件，除了根目录以外，其他目录或文件都

有唯一的父目录。我们把这种数据结构称之为树。

【例6.3】五一放假，全家人外出自驾游，在手机导航软件中设置好出发地和目的地，导航会帮你选一条路线，就可以按照这个路线行驶过去。这里需要用到导航地图，地图中涉及各个途经地和这些地方之间的道路，我们把这些途经地看作点，把道路看作边，边上再添加距离、限速、红绿灯等附加信息，就可以在程序中存储起来。我们把这样的数据结构称为图。

像上面介绍的线性表、树、图这些相互之间存在一种或多种特定关系的数据元素的集合就是数据结构。

接下来几节大家将重点学习线性表、队列、栈等数据结构。

6.2　线性表的储存结构及其应用

线性结构的特点是，在数据元素的非空有限集合中：

（1）存在唯一的一个被称作“第一个”的数据元素。

（2）存在唯一的一个被称作“最后一个”的数据元素。

（3）除第一个之外，集合中的每个数据元素均只有一个前驱。

（4）除最后一个之外，集合中每个数据元素均只有一个后继。

6.2.1　线性表的储存结构

线性表是最基础最简单的数据结构，是零个或多个具有相同类型的数据元素的有限序列，数据元素的个数称为线性表的长度，长度有限。线性表中的元素具有顺序性，是按顺序储存的。中间每个元素都有一个前驱（前面的元素）和后继（后面的元素），开头元素没有前驱，结尾元素没有后继。打个比方，线性表就是火车，车厢按顺序一节拖着一节，车头前面没有车厢，车尾后面同样没有车厢。线性表储存的数据元素类型必须是相同的，储存的数据可以不同，这就好比同一火车每节车厢的长宽高都是一样的，只能装同一种类型的货物，但可以装很多不同的货物。

若将一个有 n 个数据元素的有限序列的线性表记为 $(a_1,a_2,\ldots,a_{i-1},a_i,a_{i+1},\ldots,a_n)$，则表中 a_{i-1} 领先于 a_i，a_i 领先于 a_{i+1}，称 a_{i-1} 是 a_i 的直接前驱元素，a_{i+1} 是 a_i 的直接后继元素。当 $i=1,2,\ldots,n-1$ 时，a_i 有且仅有一个直接后继，当 $i=2,3,\ldots,n$ 时，a_i 有且仅有一个直接前驱。

线性表中元素的个数 $n(n>=0)$ 定义为线性表的长度，$n=0$ 时称为空表。

在非空表中的每个数据元素都有一个确定的位置，如a_1是第一个数据元素，a_n是最后一个数据元素，a_i是第i个数据元素，称i为数据元素a_i在线性表中的位序。

线性表是一个相当灵活的数据结构，它的长度可根据需要增长或缩短，即不仅可以访问线性表中的数据元素，还可以进行插入、删除和合并等操作。

线性表在程序中按照储存结构被分为两类：顺序表和链表。接下来我们将通过一个最基本的例子来讲述顺序表和链表的表示和实现。

【例6.4】维护序列。给定一个长度为n的整数序列。现在有m个操作，操作分为三类，格式如下：

（1）1i：询问序列中第i个元素的值，保证i小于等于当前序列长度。

（2）2iv：在序列中第i个元素前加入新的元素v，保证i小于等于当前序列长度。

（3）3i：删除序列中的第i个元素，保证i小于等于当前序列长度。

输入格式：第1行输入n(1<=n<=1000)，表示序列最初的长度；第2行输入n个空格隔开的数，表示原始的整数序列；第3行输入m(1<=m<=1000)，表示操作数；第4到m+3行依次输入一个操作。

输出格式：对于操作（1）输出对应的答案，一行输出一个数。

输入样例：

```
5
6 31 23 14 5
5
1 2
2 2 7
1 2
3 3
1 3
```

输出样例：

```
31
7
23
```

1. 顺序表

顺序表是线性表的顺序表示，是用一组地址连续的存储单元(数组)依次存储线性表的数据元素。

顺序表的特点是：为表中相邻的元素 a_i 和 a_{i+1} 赋以相邻的存储位置 Locate(i)和locate(i+1)。换句话说，以元素在计算机内“物理位置相邻”来表示线性表中数据元素之间的逻辑关系。

假设线性表的每个元素需占用c个存储单元，并以所占的第一个单元的存储地址作为数据元素的存储位置，则线性表中第i+1个数据元素的存储位置 Locate(i+1)=Locate(i)+c，我们还可以推断出 Locate(i)=Locate(1)+(i−1)*c。如图6.1所示。

数据元素在线性表中的位序	存储地址	内存状态
1	b	a_1
2	b+c	a_2
...	...	...
i	b+(i−1)*c	a_i
...	...	...
n	b+(n−1)*c	a_n

图6.1　线性表的顺序存储结构示意图

只要确定了存储线性表的起始位置和数据元素在线性表中的位序，就可以知道该数据元素的存储地址，我们把这个特点称为随机存取，顺序表存储结构是一种随机存取的存储结构。由于数组类型也有随机存取的特性，因此，通常都用数组来描述顺序表。

由于数组下标是从“0”开始的，为避免出错，例6.4中可以定义一个长度为2001的整型数组a来存储序列：int a[2001]。a[i]就是表中第i个元素。

下面重点讨论顺序表的插入和删除这两种操作。

(1)插入：在长度为n的线性表$(a_1,a_2,\ldots,a_{i-1},a_i,a_{i+1},\ldots,a_n)$的第i(1<=i<=n+1)个元素前插入新的元素v，将会得到长度为n+1的线性表$(a_1,a_2,\ldots,a_{i-1},v,a_i,a_{i+1},\ldots,a_n)$。数据元素 a_{i-1} 和 a_i 之间的逻辑关系发生了变化。在顺序表中，由于逻辑上相邻的数据元素在物理位置上也是相邻的，因此，除非i=n+1，否则必须把 a_n，$a_{n-1},\ldots,a_i$ 这n−i+1个元素依次往后移动一位，再把v插入到原 a_i 处。

假设在任何位置上插入时是等概率的，则插入操作的平均移动次数为：

$$\sum_{i=1}^{n+1}\frac{n-i+1}{n+1}=\frac{n(n+1)}{2(n+1)}=\frac{n}{2}。$$

时间复杂度为O(n)。

（2）删除：删除长度为n的线性表$(a_1,a_2,\ldots,a_{i-1},a_i,a_{i+1},\ldots,a_n)$的第i(1<=i<=n)个元素，将会得到长度为n-1的线性表$(a_1,a_2,\ldots,a_{i-1},a_{i+1},\ldots,a_n)$。数据元素$a_{i-1}$、$a_i$和$a_{i+1}$之间的逻辑关系发生了变化。为了在顺序表中反映这个变化，需要把$a_{i+1},\ldots,a_n$这n-i个元素依次往前移动一位。假设任何位置删除的概率相等，则删除操作的平均移动次数为：

$$\sum_{i=1}^{n}\frac{n-i}{n}=\frac{n(n-1)}{2n}=\frac{n-1}{2}$$

时间复杂度也是O(n)。

例6.4中线性表变化如图6.2所示。

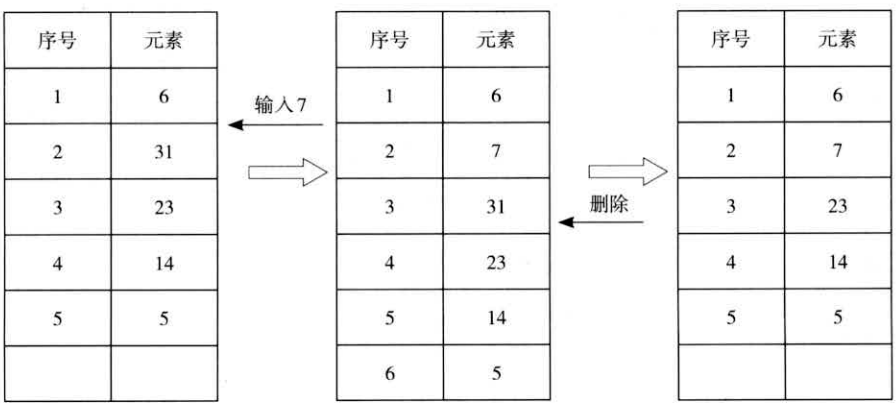


图6.2 线性表插入删除操作示意图

对应的程序eg6.4_1如下：

```
1  //eg6.4_1 用顺序表实现线性表
2  #include<iostream>
3  using namespace std;
4  int n,a[2001];
5  void InsertValue(int pos,int value)
                              // 在第 pos 个元素前插入新元素 value
6  {
7      for(int i=n;i>=pos;i--)a[i+1]=a[i];  // 后移元素
8      a[pos]=value;
9      n++;
10 }
11 void DeleteValue(int pos)     // 删除第 pos 个元素
```

```
12 {
13     for(int i=pos+1;i<=n;i++)a[i-1]=a[i];  //前移元素
14     n--;
15 }
16 int main()
17 {
18      int m;
19      cin>>n;
20      for(int i=1;i<=n;i++)cin>>a[i];
21      cin>>m;
22      for(int i=1;i<=m;i++)
23      {
24           int type,pos,value;
25           cin>>type>>pos;
26           if(type==1)cout<<a[pos]<<endl;
27           if(type==2)
28           {
29                cin>>value;
30                InsertValue(pos,value);
31           };
32           if(type==3)DeleteValue(pos);
33      }
34      return 0;
35 }
```

2. 单链表

由上一节的讨论可知，顺序表的特点是逻辑关系上相邻的两个元素在物理位置上也相邻，因此可以随机存取表中任一元素，它的存储位置可用一个简单、直观的公式来表示。然而，从另一方面来看，这个特点也铸成了这种存储结构的弱点：插入和删除操作需要移动大量元素。本节我们将讨论线性表的另一种表示方法——链表。由于链表不要求逻辑上相邻的元素在物理位置上也相邻，因此链表没有顺序表在进行插入删除的弱点，但同时也失去了顺序表可随机存取的优点。

链表的特点是用一组任意的存储单元存储线性表的数据元素，这一组存储单元不要求是连续的。因此为了表示每个数据元素 a_i 与其直接后继数据元素 a_{i+1} 之间的逻辑关系，对数据元素 a_i 来说，除了存储其本身的信息之外，还需存储一个指示其直接后继存储位置的信息。这两部分信息组成数据元素 a_i 的存储映像，称为结点（Node）。它包括两个域：其中存储数

据元素信息的域称为数据域；存储直接后继存储位置的域称为指针域。N个结点链结成一个链表，即为线性表（$a_1,a_2,\ldots,a_n$）的链式存储结构。又由于此链表的每个结点中只包含一个指针域，故又称线性链表或单链表。

图6.3为线性表（NOI,GOOD,LUCK,TO,EVERY,BODY）的线性链表存储结构，整个链表的存取必须从头指针开始进行，头指针指示链表中第一个结点的存储位置。同时，由于最后一个数据元素没有直接后继，所以线性链表中最后一个结点的指针为“空”(NULL)。

头指针 H

13

存储地址	数据域	指针域
5	LUCK	7
6	EVERY	23
7	TO	6
13	NOI	18
18	GOOD	5
23	BODY	NULL

图6.3 线性链表示意图

用线性链表表示线性表时，数据元素之间的逻辑关系是由结点中的指针指示的。换句话说，指针为数据元素之间的逻辑关系的映像，逻辑上相邻的两个数据元素其存储的物理位置不要求紧邻，由此，这种存储结构为非顺序映像或链式映像。

通常我们把链表画成用箭头相链接的结点的序列，结点之间的箭头表示链域中的指针。图6.3的线性链表可画成图6.4的形式，这是因为在使用链表时，关心的只是它所表示的线性表中数据元素之间的逻辑顺序，而不是每个数据元素在存储器中的实际位置。

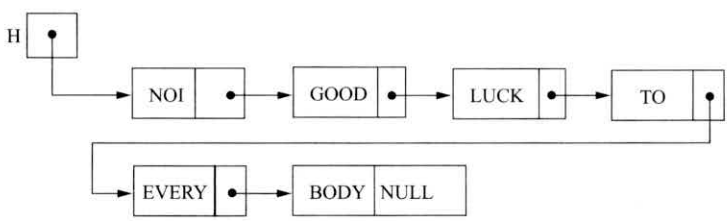


图6.4 线性链表的逻辑状态

链表可以用指针来实现，也可以用数组来实现，我们这里介绍数组的方法。为了方便，用数组实现链表需要用到以下三部分：

（1）元素数组Value[]：记录结点元素的数据域。

（2）后继数组Next[]：记录后继结点的存储位置。

（3）头结点指针Head：头结点为链表第一个结点之前附设的一个结点，数据域可以不存储任何信息，而Next[Head]记录的是链表第一个结点的存储位置。

在线性表的顺序存储结构中，由于逻辑相邻的两个元素在物理位置上紧邻，所以每个元素的存储位置都可从线性表的起始位置计算得到。而在单链表中，任何两个元素的存储位置之间没有固定的联系。然而，每个元素的存储位置都包含在其直接前驱结点的信息之中。假设p是线性表中第i个数据元素的存储位置，则Next[p]是第i+1个数据元素的存储位置。也就是说，如果Value[p]=a_i，则Value[Next[p]]=a_{i+1}。因此，单链表不是随机存取的数据结构，在单链表中取得第i个数据元素必须从头结点指针Head出发寻找。定义函数GetPos(i)返回链表中第i个元素在Value数组的存储位置，输出第i个元素就可以使用cout<<Value[GetPos(i)]，函数GetPos在链表中的实现如下所示：

```
int GetPos(int i)
{
    int hd = Head;
    for(int j=1;j<=i;j++)hd = Next[hd];
    return hd;
}
```

GetPos的时间复杂度为O(n)。

那么在单链表中，我们要如何实现插入和删除操作？

假设我们要在第i个元素前插入新的元素x，那么我们首先需要找到第i-1个元素对应的地址p，新的元素地址为q，那么我们只需要将Next[q]设为Next[p]，Next[p]设为q即可，如图6.5所示。

在链表中的删除与插入其实是类似的。假设我们当前要删掉第i个元素，我们先找到第i-1个元素的存储地址p，然后把Next[p]设置为Next[Next[p]]即可，如图6.6所示。

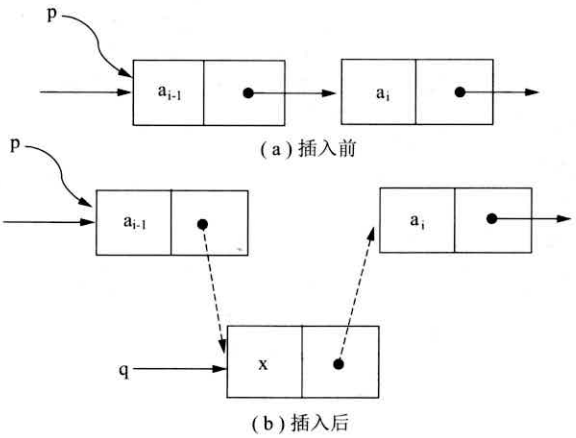


图6.5 线性链表中插入结点后指针变化情况

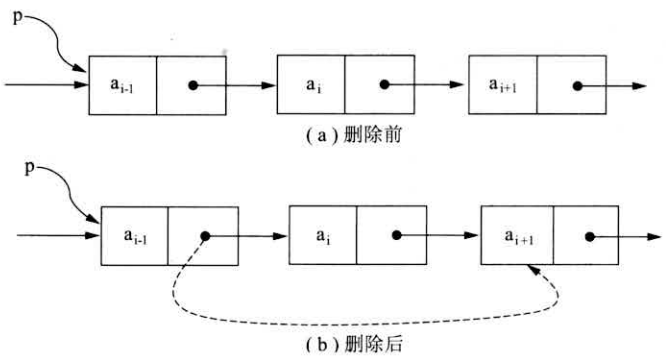


图6.6 线性表中删除结点后指针变化情况

因此，假设我们已经知道链表中第i-1个元素对应的地址，那么链表的插入和删除操作的时间复杂度就是O(1)的了。

综上，我们可以得到例6.4的链表实现程序eg6.4_2如下：

```
1  //eg6.4_2 用单链表实现线性表
2  #include<iostream>
3  using namespace std;
4  int n,Value[2001],Next[2001],Head=0;
5  int GetPos(int pos)        // 求链表第 i 个元素的存储位置
6  {
7      int hd=Head;
8      for(int j=1;j<=pos;j++)hd=Next[hd];
9      return hd;
10 }
11 void InsertValue(int pos,int val)
```

```
                                        // 在第 pos 个元素前插入新元素 val
12 {
13      int p=GetPos(pos-1);
14      Value[++n]=val;
15      Next[n]=Next[p];
16      Next[p]=n;
17 }
18 void DeleteValue(int pos)      // 删除第 pos 个元素
19 {
20      int p=GetPos(pos-1);
21      Next[p]=Next[Next[p]];
22 }
23 int main()
24 {
25      int m;
26      cin>>n;
27      for(int i=1;i<=n;i++){
28          cin>>Value[i];Next[i-1]=i;
29 }
30      cin>>m;
31      for(int i=1;i<=m;i++){
32           int type,pos,val;
33           cin>>type>>pos;
34           if(type==1)cout<<Value[GetPos(pos)]<<endl;
35           if(type==2){
36                cin>>val;InsertValue(pos,val);
37           };
38      if(type==3)DeleteValue(pos);
39      }
40      return 0;
41 }
```

3. 循环链表

循环链表是另一种形式的链式存储结构。它的特点是，表中最后一个结点的指针指向头结点，整个链表形成一个环。由此，从表中任一结点出发均可找到表中其他结点，图6.7所示是单链的循环链表。

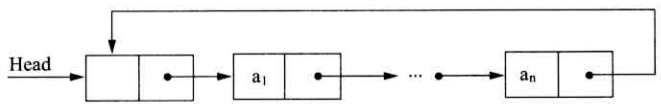


图6.7　循环链表示意图

循环链表的操作和线性链表的操作基本一致，这里不再累述。

4. 双向链表

以上讨论的链式存储结构的结点中只有一个指示直接后继的指针域，由此，从某个结点出发只能顺指针往后寻找其他结点。若要寻找结点的直接前驱，则需要从表头指针出发。换句话说，在单链表中寻找一已知结点p的后继结点的时间复杂度为O(1)，而寻找p的直接前驱的时间复杂度则为O(n)。为克服单链表这种单向性的缺点，可以利用双向链表。

顾名思义，在双向链表的结点中有两个指针域，一个指向直接后继，一个指向直接前驱。可以在前面单链表中增加前驱数组Pre[]，Pre[i]记录存储位置为i的结点的前驱结点的存储位置。

和单向循环链表类似，双向链表也可以有循环表，如图6.8所示。

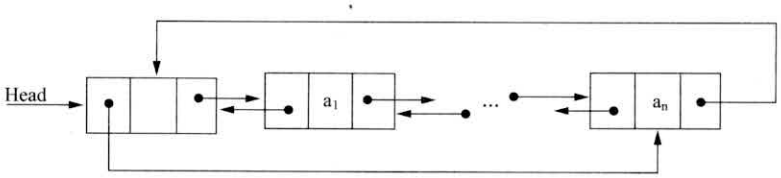


图6.8 双向循环链表示意图

在双向链表中，插入和删除操作与单链表有很大的不同，需要同时修改两个方向上的指针。图6.9显示了在结点p后面插入新结点q的变化。

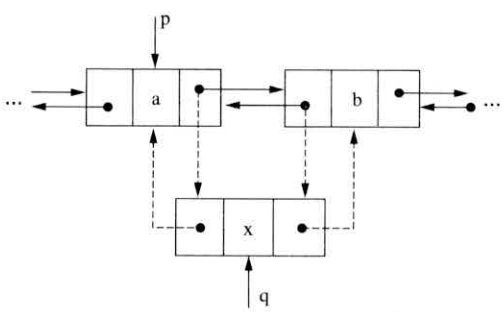


图6.9 双向链表插入结点后指针变化

图6.9所示插入操作可以通过执行下列语句来实现：

```
Next[q]=Next[p];
Next[p]=q;
Pre[Next[q]]=q;
Pre[q]=p;
```

图6.10显示了删除结点p的后继结点的变化。

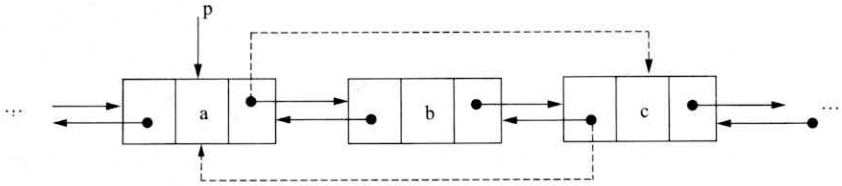


图6.10　双向链表删除结点后指针变化

图6.10所示删除操作可以通过执行下列语句来实现：

```
Next[p]=Next[Next[p]];
Pre[Next[p]]=p;
```

6.2.2　线性表的应用

下面通过一个例子来看看线性表在实际解决问题中的应用。

【例6.5】小A的烦恼。小A生活在一个神奇的国家，这个国家有N(N <= 100000)个城市，还有M(M <= 5000000)条道路连接两个城市。道路连接的两个城市可以互相直接免费到达。小A比较烦恼，因为他想知道每个城市能直接到达哪些城市，你能帮帮他吗？保证每个城市都有道路与其连接。(注：按照输入的道路顺序输出每个城市直接连接的城市)

输入格式：第1行包含两个整数N和M；接下来M行，每行两个整数，描述一条道路连接的两个城市的编号。

输出格式：输出N行，每行若干个用一个空格隔开的整数；第i行输出的是与城市i直接相连的城市编号，保证城市的出现按照道路输入的先后顺序出现。

输入样例：

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

输出样例：

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

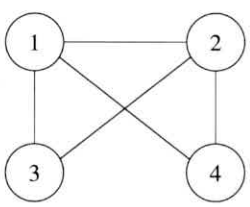
2 1

1 2

分析：每个城市能到的城市可以用一个线性表来表示，一共需要开N个线性表来存储。如果每个线性表用一个顺序表来实现，我们算一算要开多大的数组来存储。有N个城市，每个城市最多与N-1个城市直接连接，意味着每个表最大长度都有可能达到N-1，我们总共需要开$O(N^2)$大小的数组，数组大小超过9GB!内存承受不了！每个线性表用一个独立的指针实现的链表来表示是可以解决的，这里我们不作介绍。

下面我们介绍另一种方法：用数组实现的链表来表示每一个线性表，并把这N个线性表合并在一个数组中。具体实现的思路是，对于输入的一条边(a,b)，把结点a和b都加到同一个数组中，把城市a添加到城市b所在的链表的尾部，把城市b添加到城市a所在链表的尾部。要实现这个，对于城市i，我们只需记录与之相连的第一个城市Head[i]以及当前与之连接的最后一个城市Tail[i]，当一个新的城市j加入进来，只需直接插入到尾部，执行Next[Tail[i]]=j，并让j成为新的尾部即Tail[i]=j即可。这样一来，数组的总长度是边数的两倍，即2*M，大大节省了空间。输出方案时，对于城市i，从Head[i]开始顺着后继Next[]就可以把与城市i直接相连的城市按照道路输入的顺序输出来。

题目中的样例对应的示意图如图6.11所示。



i	1	2	3	4
Head[i]	4	1	2	6
Tail[i]	9	10	3	8

i	City[i]	Next[i]	i	City[i]	Next[i]
1	3	7	6	1	8
2	2	3	7	4	10
3	1	0	8	2	0
4	3	5	9	2	0
5	4	9	10	1	0

图6.11 样例链表存储示意图

对应的程序eg6.5如下：

```
1  //eg6.5
2  #include<iostream>
3  using namespace std;
4  const int N=100002;
5  int n,m,tot,Head[N],Tail[N],Next[N*10],City[N*10];
6  void Add(int x,int y)    //把城市y加到x的链表的末尾
7  {
8     City[++tot]=y;
9     if (Head[x]==0)Head[x]=tot;
10    else Next[Tail[x]]=tot;
11    Tail[x]=tot;
12 }
13 int main()
14 {
15    cin>>n>>m;
16    tot=0;
17    for(int i=1;i<=m;i++)
18    {
19        int x,y;
20        cin>>x>>y;
21        Add(x,y);
22        Add(y,x);
23    }
24    for(int i=1;i<=n;i++)
25    {
26        for(int j=Head[i];j;j=Next[j])cout<<City[j]<<" ";
27        cout<<endl;
28    }
29    return 0;
30 }
```

6.3 队列及其应用

6.3.1 队列

队列是用来存储暂未处理但需要按一定顺序处理的元素的一种数据结构，是一种先进先出（First In First Out,FIFO）的线性表，特点是先进

队的元素先出队。它只允许在表的一端进行插入，而在另一端删除元素。这和我们日常生活中的排队购物是一致的，最早进入队列的元素最早离开。在队列中，允许插入的一端叫做队尾，允许删除的一端则称为队首。假设队列为 $q=(a_1,a_2,\ldots,a_n)$，那么 a_1 是队首元素，a_n 则是队尾元素。队列中的元素是按照 $a_1,a_2,\ldots,a_n$ 的顺序进入的，退出队列也只能按照这个顺序依次退出，也就是说只有在 $a_1,a_2,\ldots,a_{n-1}$ 都离开队列之后，a_n 才能退出队列。队列的示意图如图 6.12 所示。

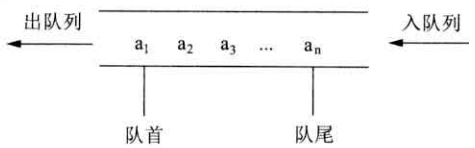


图6.12 队列示意图

队列的实现需要以下三部分：

（1）顺序表 q[m]：用来存储队列中的元素，m 是队列能存储元素的最大容量。

（2）队首指针 front：指向队首元素存储的位置。

（3）队尾指针 rear：指向队尾元素的下一个位置。

队列的基本操作：

（1）队列初始化：

```
front=0;rear=0;
```

这时队列为空，没有元素。

（2）判断队列是否为空：如果 front 等于 rear，则队列为空。

（3）判断队列是否已满：如果 rear=m，则队列已满。

（4）进队(插入元素 x)：如果队列未满，则执行 q[rear++]=x。

（5）出队：如果队列不为空，则返回队首元素 q[front]，同时 front 加 1。

（6）队列中元素个数：rear-front。

如定义 q[4] 存储 4 个元素，图 6.13 显示了队列的初始化、进队和出队操作。

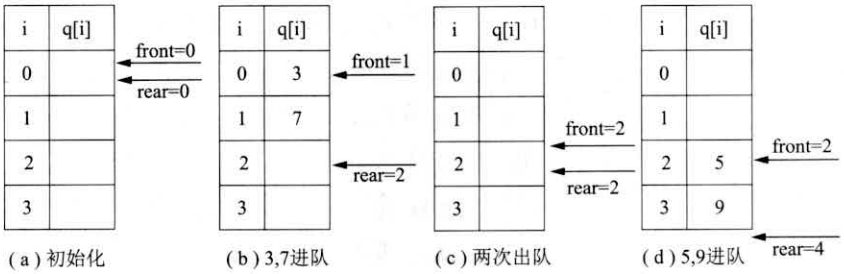


图6.13　队列基本操作示意图

如图6.13所示，在(d)后再把新元素11添加进队列是不能实现的，因为队尾指针rear已经超出了数组的上界，而实际上队列并未占满。为了充分利用空间不造成浪费，我们可以将顺序队列看作是循环的，即对于队列q[m]来说，数组下标0看作是m-1的下一个位置。示意图如图6.14所示。

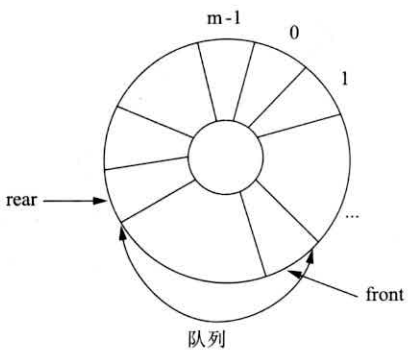


图6.14　循环队列示意图

如果采用循环队列的话，图6.13(d)队尾指针rear就应该是0，还可以进行进队操作，如图6.15所示。

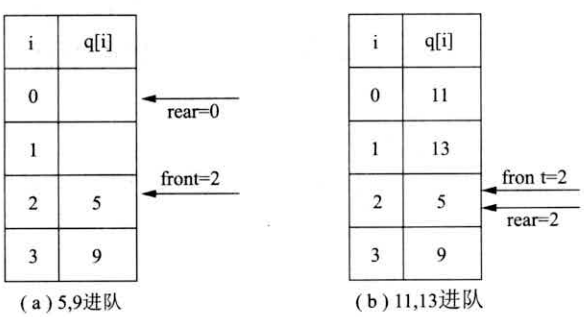


图6.15　循环队列操作示意

图6.15(b)中队列已满，front与rear指向同一位置，为了区别队列为

空的判断条件，可以考虑加标记区别队列是“空”还是“满”，也可以少用一个元素空间，以“队首指针在队尾指针的下一个位置”作为判断“满”的条件。具体循环队列的基本操作如下：

（1）“队列初始化”与“判断队列是否为空”与顺序队列是一样的。

（2）判断队列是否已满：如果(rear+1)% m==front即rear下一个位置是front，则队列已满。这种情况下，循环队列q[m]最多只能存储m-1个元素。

（3）进队(插入元素x)：如果队列未满，则执行q[rear]=x;rear=(rear+1)% m。

（4）出队：如果队列不为空，则返回队首元素q[front]同时front=(front+1)% m。

（5）队列中元素个数：(rear-front+m)% m。

队列还可以采用链式存储结构，当要实现多个队列共享内存时，选择链式分配结构则更为合适。

6.3.2 队列的应用

【例6.6】 Blah数集。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，对应以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？

输入格式：输入包含很多行，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)。

输出格式：对应每个输入，输出集合Ba的第n个元素值。

输入样例：

```
1 100
28 5437
```

输出样例：

```
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。因此除了第一个数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。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：

```
1  //eg 6.6
2  #include<iostream>
3  #include<algorithm>
4  using namespace std;
5  const int N = 1000100;
6  long long q[N];
7  int a,n;
8  void work(int a,int n)
9  {
10      int rear=2;
11      q[1]=a;
12      int two=1,three=1;
13      while(rear<=n)
14      {
15          long long t1=q[two]*2+1,t2=q[three]*3+1;
16          int t=min(t1,t2);
17          if(t1<t2)two++;else three++;
18          if (t==q[rear-1])continue;
19          q[rear++]=t;
20      }
21      cout<<q[n]<<endl;
```

```
22 }
23 int main()
24 {
25     while (cin>>a>>n) work(a,n);
26     return 0;
27 }
```

【例6.7】连通块。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他黑色格子。

输入格式：第1行，两个整数n,m(1<=n,m<=100)，表示一个n * m的方格图；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。

输出格式：1行，一个整数ans，表示图中有ans个黑色格子连通块。

输入样例：

```
3 3
1 1 1
0 1 0
1 0 1
```

输出样例：

```
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，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，则将其加入队尾，扩展完就把该格子（队首）删除。当队列为空时，一个联通块就搜索完了。

样例所对应的方格图如图6.16所示。

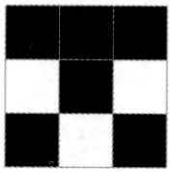


图6.16　样例对应的方格示意图

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)}，联通块数量加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。

对应的程序eg6.7如下：

```
1  //eg 6.7
2  #include <iostream>
3  using namespace std;
4  const int N = 110;
5  const int flag[4][2] = {{0,1},{0,-1},{1,0},{-1,0}};
6  int a[N][N],queue[N * N][2];
7  int n, m, ans;
8  bool p[N][N];
9  void bfs(int x,int y)
10 {
```

```
11   int front =0,rear =2;
12   queue[1][0] = x,queue[1][1] = y;
13   while (front < rear-1)
14   {
15       ++ front;
16       x = queue[front][0];
17       y = queue[front][1];
18       for(int i = 0;i < 4;++ i)
19       {
20           int x1 = x + flag[i][0];
21           int y1 = y + flag[i][1];
22           if(x1<1||y1<1||x1>n||y1>m||!a[x1][y1]||p[x1]
             [y1])continue;
23           p[x1][y1] = true;
24           queue[rear][0] = x1;
25           queue[rear++][1] = y1;
26         }
27     }
28   }
29 int main()
30   {
31    cin >> n >> m;
32    for(int i = 1;i <= n; ++ i)
33        for(int j = 1;j <= m; ++ j)cin >> a[i][j];
34    for(int i = 1;i <= n; ++ i)
35        for(int j = 1; j <= m; ++ j)
36          if(a[i][j] && !p[i][j])
37          {
38              ++ ans; bfs(i,j);
39          }
40        cout << ans << endl;
41        return 0;
42    }
```

6.4 栈及其运用

6.4.1 栈

栈是限定仅在表尾进行插入或删除操作的线性表。因此，对栈来说，

表尾端有其特殊含义，称为栈顶，相应地，表头端称为栈底。不含元素的空表称为空栈。

假设栈 $S=(a_1,a_2,\ldots,a_n)$，那么 a_1 为栈底元素，a_n 为栈顶元素。栈中元素按 $a_1,a_2,\ldots,a_n$ 的顺序进栈，退栈的第一个元素应为栈顶元素。换句话说，栈的修改是按先进后出的原则进行的。因此，栈又称为先进后出(First In Last Out，FILO)的线性表。示意图如图6.17所示。

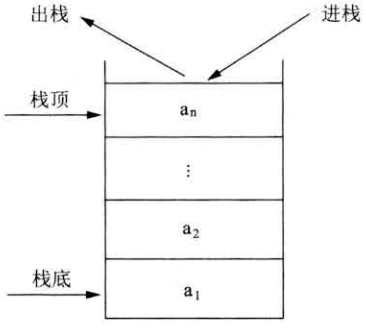


图6.17　栈的示意图

和线性表一样，栈也有两种存储表示方法：顺序栈和链式栈。我们这里主要介绍顺序栈。

顺序栈，即用顺序存储结构表示栈，一般用一个顺序表 stack 和栈顶指针 top 实现，top 指示栈顶元素在顺序栈中的位置，stack[top] 存储的就是栈顶元素，top=0 表示空栈。

栈的基本操作：

（1）初始化：top=0；stack[m] 只能存储 m−1 个元素。

（2）进栈：如果栈不满，则 stack[++ top]=item。

（3）出栈：如果栈不为空，则 item = stack[top--]；即执行出栈操作时要保证栈中有元素。

如1，2，3三个数先后进栈，从空栈开始依次进行进栈、进栈、出栈、进栈、出栈、出栈的操作，得到的出栈序列为2，3，1。示意图如图6.18所示。

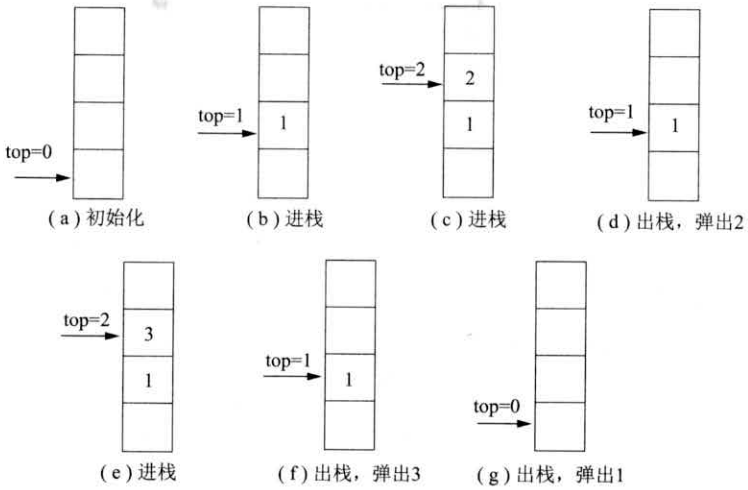


图6.18 栈的基本操作示意图

思考： 1~N这N个数按照以1,2,3,...,N的顺序进栈，通过N次进栈和N次出栈操作，最后得到1~N的出栈序列，每一个出栈序列都是1~N的一个排列，问这样的出栈序列有多少种？1~N的全排列中哪些排列不在出栈序列中？

提示： 以1,2,3的顺序进栈，可以得到以下5种出栈序列，括号中是对应的进栈出栈操作：

（1）1 2 3(进栈、出栈、进栈、出栈、进栈、出栈)。

（2）1 3 2(进栈、出栈、进栈、进栈、出栈、出栈)。

（3）2 1 3(进栈、进栈、出栈、出栈、进栈、出栈)。

（4）2 3 1(进栈、进栈、出栈、进栈、出栈、出栈)。

（5）3 2 1(进栈、进栈、进栈、出栈、出栈、出栈)。

而3 1 2不在出栈序列中，因为要先弹出3，必然1和2都在栈中，又因为1先进栈，所以1不可能在2前面出栈。

6.4.2 栈的应用

【例6.8】 括号匹配。给定一个只包含左右括号的合法括号序列，按右括号从左到右的顺序输出每一对配对的括号出现的位置(括号序列以0开始编号)。

输入格式：仅1行，表示一个合法的括号序列。

输出格式：设括号序列有n个右括号，则输出包括n行，每行两个整数l，r，表示配对的括号左括号出现在第l位，右括号出现在第r位。

输入样例：

(()())()

输出样例：

1 2
0 3
4 5

分析：维护一个栈，从左到右扫序列，如果当前括号是左括号，则将该位置加入栈中，如果是右括号，则该右括号与栈顶位置的左括号匹配，输出这对匹配括号的位置并删除栈顶的左括号。

比如说当前有一个括号序列为(())()，那么扫描的过程为：

（1）将位置0压入栈，当前栈为：{0}。

（2）将位置1压入栈，当前栈为：{0,1}。

（3）位置2为右括号，则位置2与栈顶元素1配对，输出1 2，并将栈顶弹出，当前栈为：{0}。

（4）位置3为右括号，则位置3与栈顶元素0配对，输出0 3，并将栈顶弹出，当前栈为：{ }。

（5）将位置4压入栈，当前栈为：{4}。

（6）位置5为右括号，则位置5与栈顶元素4配对，输出4 5，并将栈顶弹出，当前栈为：{ }。

程序结束，得到配对关系为：(1, 2)，(0, 3)，(4, 5)。对应的程序eg6.8如下：

```
1  //eg 6.8
2  #include <iostream>
3  #include <string>
4  using namespace std;
5  string S;
6  int Stack[105],top;
7  int main()
8  {
9    cin >> S;
10   for(int i = 0;i <int(S.length());i ++)
11     if(S[i] == '(') Stack[++top] = i;
12     else cout << Stack[top--] << " " << i << endl;
13   return 0;
14 }
```

【例6.9】铁轨。每辆火车都从A方向驶入车站C，再从B方向驶出车站C，同时它的车厢可以进行某种形式的重新组合。组合方式为：最晚驶入车站C的车厢停在最前面，可在任意时间将停在最前面的车厢驶出车站C。假设从A方向驶来的火车有n节车厢(n<1000)，分别按顺序编号为1，2，...,n。假定在进入车站之前每节车厢之间都是不连着的，并且它们可以自行移动，直接倒出在B方向的铁轨上。另外假定车站C里可以停放任意多节的车厢。但是一旦当一节车厢进入车站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方向的铁轨后，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a_1，a_2，...，a_n的顺序从B方向驶出。

请写一个程序，用来判断能否得到指定的车厢顺序。

输入格式：第1行，一个整数n，表示有n节车厢；接下来一行有n个整数，表示对应顺序。

输出格式：输出仅1行。若可以，则输出“Possible”，否则输出“Impossible”。

输入样例：

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

输出样例：

```
Possible
```

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a_i驶出。若a_i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a_i驶入，再将它从B方向驶出；若a_i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，则就把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。

程序eg6.9如下：

```
1  //eg 6.9
2  #include <iostream>
3  #include <algorithm>
4  #include <cstring>
5  using namespace std;
6  const int N = 1010;
7  int stack[N],a[N];
8  int top,n;
9  int main()
10 {
11     cin >> n;
12     for(int i = 1;i <= n;++ i)
13         cin >> a[i];
14     top = 0;
15     for(int i = 1,cur = 1;i <= n;++ i)
                        //cur为当前要从A方向驶入的车厢号
16     {
17         while(cur <= a[i])
18           stack[++ top] = cur ++;
19         if(stack[top] == a[i])
20               --top;
21         else
22         {
23           cout << "Impossible" << endl;
24           return 0;
25         }
26     }
27     cout << "Possible" << endl;
28     return 0;
29 }
```

6.5 二分及其快速排序

6.5.1 二 分

小时候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，你朋友只会回答“比x大”或者“比x小”或者“猜中”，你能保证在10次以内猜中吗？好运的话当然是一次就猜中了，但是幸运女神不会这么照顾你。

一开始猜测的区间是1到1000，你可以先猜500，运气好可以一次猜中，如果答案比500大，显然1到500都不可能有答案，下一次猜测的区间变为501到1000，如果答案比500小，那500到1000不可能有答案，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，答案就会被猜中了。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(log n)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。如上面的例子相当于1到1000从左到右排成一排，当我们猜某个数x时，如果x比答案小，那么x左边包括自己是不超过x的，现在也就都比答案小，直接排除，这里就是一种有序性。

下面通过几个例子来体会二分法的应用。

【例6.10】找数。给一个长度为n的单调增的正整数序列，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？

输入格式：第1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。

输出格式：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。

输入样例：

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

5

1

3

输出样例：

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，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a[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6.19所示。

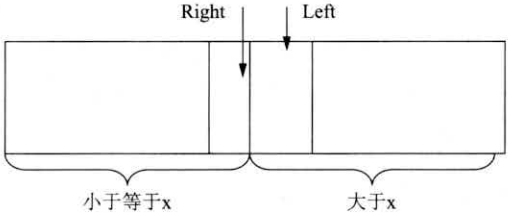


图6.19　二分循环结束示意图

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eg6.10如下：

```
1  //eg 6.10
2  #include <iostream>
3  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 6.11】 月度开销。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。

输入格式：第1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。

输出格式：一个整数，即最大月度开销的最小值。

输入样例：

```
7 5
100
400
300
```

100
500
101
400

输出样例：

500

分析：不妨设ok(i)表示是否存在一种方案使得开销最多的fajo月的开销小于等于i。判断ok(i)可以用贪心法，从左到右扫描每一天的开销，采用“物尽其用”原则，在不超过i的前提下让每一个fajo月的天数越多越好，如果最终得到的预算案小于等于M，则ok(i)=1，答案可以比i再小一些，如果大于M，则ok(i)=0,答案要比i大一些。这就满足了二分的有序性。

用L表示二分区间左边界，R表示右边界，当L<=R时：

(1) Mid=(L+R)/2。

(2) 如果ok(Mid)=1，则R=Mid-1，否则L=Mid+1。

重复执行直到L>R，由于L的左边都是小于答案的，R的右边是大于等于答案的。所以最终答案就是L。

程序如下：

```
1  //eg 6.11
2  #include <iostream>
3  using namespace std;
4  const int MAXN = 100005;
5  int A[MAXN],N,M;
6  bool ok(int limit) // 判断 limit 对应的 fajo 月数是否不超过 M
7  {
8      int c = 1;
9      for(int i = 1,sum = 0;i <= N;i ++)
10       if(sum + A[i] <= limit) sum += A[i]; else
11       {
12           ++ c;
13           sum = A[i];
14           if(sum > limit) return 0;
15       }
16     return c <= M;
17 }
18 int main()
```

```
19 {
20      cin >> N >> M;
21      for(int i = 1;i <= N;i ++) cin >> A[i];
22      int l = 1,r = 10000 * N;
23      while (l <= r)
24      {
25        int mid = l + r >> 1;
26        if(ok(mid)) r = mid - 1; else l = mid + 1;
27      }
28      cout << l << endl;
29      return 0;
30 }
```

6.5.2 快速排序

【例6.12】排序。给你一个长度为n的序列，让你给这个序列从小到大排序。(n<=100000)

输入格式：第1行，一个整数n；第2行n个整数，表示这个序列。

输出格式：1行，n个整数，表示排序好的序列。

输入样例：

6

2 4 5 1 3 7

输出样例：

1 2 3 4 5 7

分析：如果使用选择排序，时间复杂度为$O(n^2)$。对于本题来说不能在1秒内出解，超时。下面介绍快速排序。

快速排序的基本思想是：通过一趟排序将待排记录分割成独立的左右两部分，其中左边的所有元素都比右边的元素要小，接下来分别对这两部分继续进行排序，那么整个序列就有序了。

假设待排序的序列为{A[L],A[L+1],...,A[R]}，首先随机选择其中一个元素作为基准，然后按下述原则重新排列其余元素，将所有小于等于基准数的元素放在它的左边，将所有大于等于基准数的元素放在它的右边。根据基准数最后所落的位置x作为分界线，将原序列分割成两个子序列{A[L],A[L+1],...,A[x-1]}和{A[x+1],A[x+2],...,A[R]}。这个过程称为一次划分。

以上只是一次划分的基本思想，具体操作可能有一些出入：设基准值为T，定义两个指针i和j，i的初值设为L，j的初值设为R，我们把i左边的元素看作是无穷小，把j右边的元素看作是无穷大，也就是说指针i的含义是“i左边的元素都不比基准数大”，指针j的含义是“j右边的元素都不比基准数小”，区间[i,j]是待划分的区域。

当i<=j时，i从当前位置往右扫描找到第一个大于等于基准数的元素，j从当前位置往左扫描找到第一个小于等于基准数的元素，如果i<=j，则交换a[i]与a[j],交换完后a[i]小于等于基准数，指针i向右移动一位，a[j]大于等于基准数，指针j向左移动一位。

重复执行以上操作直到i>j为止。

以上划分把原序列划分成三部分：

（1）左边{A[L],A[L+1],...,A[j]}：这一部分的元素小于等于基准数T。

（2）中间{A[j+1],A[j+2],...,A[i-1]}：这一部分的元素等于基准数T。

（3）右边{A[i],A[i+1],...,A[R]}：这一部分的元素大于等于基准数T。

接下来对左边和右边进行递归处理即可。

程序eg6.12如下：

```
1  //eg 6.12
2  #include <iostream>
3  #include <algorithm>
4  #include <ctime>
5  #include <cstdlib>
6  using namespace std;
7  const int MAXN = 100005;
8  int A[MAXN],N;
9  void qsort(int L,int R)                    //快速排序
10 {
11   if(L>=R)return;
12   int i=L,j=R,T=A[rand()%(R-L+1)+L];
13   while (i<=j)
14   {
15      while (A[i]<T)i++;
16      while(A[j]>T)j--;
17      if(i<=j)
18      {
19        swap(A[i],A[j]);
20        i++;
```

```
21          j--;
22        }
23    }
24    qsort(L,j);
25    qsort(i,R);
26 }
27 int main()
28 {
29    cin >> N;
30    for(int i = 1;i <= N;i ++) cin >> A[i];
31    srand(int(time(0)));
32    qsort(1,N);
33    for(int i=1;i<=N;i++)cout<<A[i]<<" ";
34    cout<<endl;
35    return 0;
36 }
```

快速排序的写法很多，这里就上面的写法作几点说明：

（1）第15行指针i右移循环和第16行指针j左移循环中不用判越界，一定在L到R之间。观察“while(A[i]<t)i++;”和“while(A[j]>t)j--;”由于基准数T是从A[L]到A[R]中随机选择的，所以第一次执行第15、16行循环时，指针i和指针j至少会在基准数的位置处停下来，并且i<=j，交换A[i]和A[j]后，指针i加1，指针j减1。如果此时i>j，则循环结束，否则i指针继续右移j继续左移。由于此时指针i相对原始位置已经右移，i左边的数一定是小于等于T的，同样j相对原始位置已经左移，j右边的数一定是大于等于T的，因此再次执行第15、16行时，i指针最大不会超过j+1,j指针最小不会小于i-1，因此指针i与指针j不会超出L到R的范围，从而也保证了划分后左右两部分都有元素。

（2）第17行到第22行中的交换判断条件不能省略等于的情况。如果去掉等号，那么对于1 2 3，基准数T=2，第一次循环后i=2，j=2,从此将进入死循环。

（3）一次划分后，不能忽略中间{A[j+1],A[j+2],...,A[i-1]}等于基准数的这一部分。有的情况下，这一部分是不存在的，如对于2 1，T=2时，i=1，j=2，交换后i=2，j=1，中间部分是不存在的；但有的情况下这一部分是存在的，如上面的例子1 2 3，T=2，i=2，j=2，交换后i=3，j=1，此时中间部分是存在的，等于基准数2。

（4）基准数不能选择第一个或者中间的位置固定的数，这样的程序是有把程序卡得很慢的针对性的数据，建议用上面程序中随机产生基准数比较好。

比如说现在要对5个数：6，5，2，9，6排序，选择基准数T=6，那么第一次划分过程如图6.20所示。

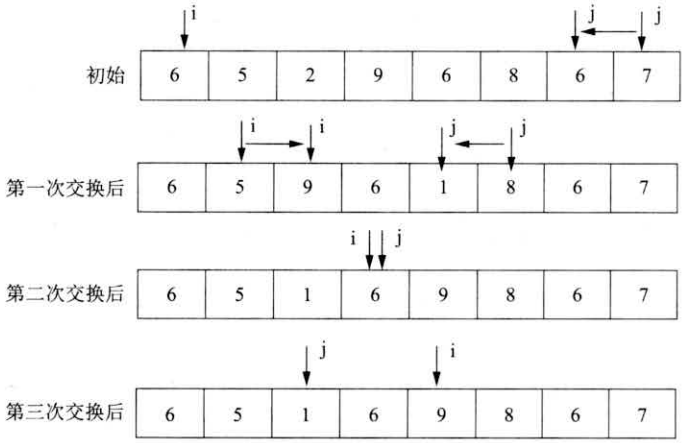


图6.20　快速排序一次划分过程示意图

一次划分结束，接下来分别对{6,5,1}{9,8,6,7}递归调用qsort进行排序即可，左边排好后为{1,5,6}，右边排好后为{6,7,8,9}，整个序列就已经完成了排序。

快速排序最好情况下时间复杂度为O(NlogN)，最坏情况下时间复杂度为$O(N^2)$，平均时间复杂度为O(NlogN)。

设T(N)为对N个元素进行快速排序所需时间。每一次划分指针i与指针j进行相向移动直到i>j为止，所以划分所需时间看作是N。

最好情况：每一次划分都会把待排序的序列均分成两半，递归的层数最少为$\log_2^N$，时间复杂度为O(N*logN)。

最坏情况：每一次划分把待排序的长度为N的序列分成长度为1和N-1的两个部分，递归深度达到N，时间复杂度为O(N*N)。

平均情况：指在每一种划分的概率均等的情况下计算时间复杂度，这里不做介绍。

【例6.13】合影效果。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）?

输入格式：第1行是人数n（2<=n<=400000，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。

输出格式：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。相邻两个数之间用单个空格隔开。

输入样例：

```
6
male 1.72
male 1.78
female 1.61
male 1.65
female 1.70
female 1.56
```

输出样例：

```
1.65 1.72 1.78 1.70 1.61 1.56
```

方法1： 这道题的比较条件比较麻烦，不像上一题就是简单的数字大小的比较。本题的比较规则如下：

（1）两人同为男生，身高矮的在左，身高高的在右。

（2）两人同为女生，身高高的在左，身高矮的在右。

（3）男生在左，女生在右。

由于每人有两个信息：性别和身高，交换时为避免两个信息都交换，定义rank[i]表示队伍中第i个人在原始队伍里面的编号，交换时直接交换rank[]即可。

程序eg6.13_1如下：

```
1  //eg 6.13_1
2  #include <iostream>
3  #include <cstdio>
4  #include <string>
5  #include <algorithm>
6  #include <ctime>
7  using namespace std;
8  const int MAXN = 400005;
```

```
string s;
double h[MAXN];
int type[MAXN],rank[MAXN],n;
void qsort(int L,int R)
{
    if(L>=R)return;
    int i=L,j=R,p=rand()%(R-L+1)+L;
    int typet=type[rank[p]];
    double ht=h[rank[p]];
    while(i<=j)
    {
      while(type[rank[i]]<typet || type[rank[i]]==typet &&
      type[rank[i]]*h[rank[i]]>typet*ht)i++;
      while(type[rank[j]]>typet || type[rank[j]]==typet
      && type[rank[j]]*h[rank[j]]<typet*ht)j--;
      if(i<=j)
      {
         swap(rank[i],rank[j]);
         i++;
         j--;
      }
    }
    qsort(L,j);
    qsort(i,R);
}
int main()
{
    cin >> n;
    for(int i = 0;i < n;i ++)
    {
      cin >> s >> h[i];
      if(s[0]=='f') type[i]=1; else type[i]=-1;
      rank[i]=i;
    }
    srand(int(time(0)));
    qsort(0,n-1);
    for(int i=0;i<n;i++)printf("%.2f%c",h[rank[i]],'');
    cout<<endl;
    return 0;
}
```

方法2：利用C++的algorithm库里的函数“sort(frist,,last,comp);”，第一个参数first是待排序数组的开始地址，第二个参数last是结束地址，是一个左闭右开区间即[first,last)，第三个参数comp是比较器，是数组中元素先后次序的判断依据。

程序如下：

```
1  //eg 6.13_2
2  #include <iostream>
3  #include <cstdio>
4  #include <string>
5  #include <algorithm>
6  using namespace std;
7  const int MAXN = 400005;
8  string s;
9  double h[MAXN];
10 int type[MAXN],rank[MAXN],n;
11 bool cmp(int a,int b)   // 比较器，函数值返回为true表示rank[a]
                             排在rank[b]的前面，否则rank[a]
                             排在rank[b]的后面
12 {
13    if(type[a] != type[b]) return type[a] < type[b];
14    if(!type[a]) return h[a] < h[b];
15    return h[a] > h[b];
16 }
17 int main()
18 {
19    cin >> n;
20    for(int i = 0;i < n;i ++)
21    {
22        cin >> s >> h[i];
23        if(s[0] == 'f') type[i] = 1; else  type[i] = 0;
24        rank[i] = i;
25 }
26 sort(rank,rank+n,cmp);
                  //这是c++自带的排序函数，在algorithm库里，第
                    一个参数是排序的开头，第二个参数是结尾，是一
                    个左闭右开区间，第三个参数是比较器
27 for(int i = 0;i < n;i ++)
28    printf("%.2f%c",h[rank[i]],(i+1)==n?'\n':' ');
                  // 这是c语言的输出，可以比较方便地输出浮点数
```

```
29 return 0;
30 }
```

【例6.14】序列第k小。给定一个长度为n的序列，问第k小的元素是多少。

输入格式：第1行，两个整数n,k；接下来一行n个数，表示这个序列。

输出格式：输出仅1行，表示第k小的元素。

输入样例：

5 3

18 23 4 5 12

输出样例：

12

分析：一种比较简单的方法是直接将这个序列排序，输出第k个数就是答案。但这个方法的复杂度为O(NlogN)，比较慢，事实上我们有一种O(N)的做法可以找到第k小的数。观察快速排序的过程，当我们按照“基准数”一次划分后，两边的大小顺序就已经相对确定了，那么我们可以只递归到第k小的值所在区间查询即可。

算法的基本流程就是，设Kth(L,R,k)表示我们要找到区间[L,R]的第k小值，那么我们先选定一个“基准数”，按照前面的分析，经过一次划分后区间[L,R]分成三部分：

（1）左边[L,j]区间是小于等于基准数的。

（2）中间[j+1,i-1]区间是等于基准数的。

（3）右边[i,R]区间是大于等于基准数的。

如果：

（1）k<=j-L+1，答案落在左边区间，只需调用Kth(L,j,k)即可。

（2）j-L+1<k<=i-L，答案落在中间区间，直接输出基准数即可。

（3）k>i-L，答案落在右边区域，只需调用Kth(i,R,k-(i-L))。

最好情况和平均情况下的时间复杂度为O(N)，最坏情况下时间复杂度为$O(N^2)$。分析方法与前面相似。

最好情况下，一次划分后，区间被均分成两部分

$$T(N) = T\left(\frac{N}{2}\right) + N$$

考虑$N = 2^k, k = \log_2^N$的情况：

$$
\begin{aligned}
T(2^k) &= T(2^{k-1}) + 2^k \\
&= T(2^{k-2}) + 2^{k-1} + 2^k \\
&= T(2^{k-3}) + 2^{k-2} + 2^{k-1} + 2^k \\
&= \cdots\cdots \\
&= 1 + 2 + \ldots + 2^k \\
&= 2^{k+1} - 1 \\
&= 2 * N - 1 \\
&= O(N)
\end{aligned}
$$

$$T(2^{k+1}) = 2^{k+2} - 1$$

当 $2^k \leqslant N < 2^{k+1}$ 时，$k = \lceil \log_2^N \rceil$，即

$$2^{k+1} - 1 \leqslant T(N) < 2^{k+2} - 1$$

因此，

$$T(N) = O(N)$$

最坏情况下的时间复杂度分析与前面的快速排序一样。

程序 eg6.14 如下：

```
1  //eg 6.14
2  #include <iostream>
3  #include <algorithm>
4  #include <ctime>
5  #include <cstdlib>
6  using namespace std;
7  const int MAXN = 1000005;
8  int A[MAXN],N,k;
9  int Kth(int l,int r,int k)
10 {
11   if(l=r)return A[l];
12   int i=l,j=r,t=A[rand()%(r-l+1)+l];
13   while (i<=j)
14   {
15     while (A[i]<t)i++;
16     while (A[j]>t)j--;
17     if(i<=j)
18     {
19       swap(A[i],A[j]);
20       i++;
21       j--;
```

```
    }
  }
    if(k<=j-l+1)return Kth(l,j,k);
    else if(k<=i-l) return t;
    else return Kth(i,r,k-(i-l));
}
int main()
{
  cin >> N>>k;
  for(int i=1;i <= N;i ++) cin >> A[i];
  srand(int(time(0)));
  cout<<Kth(1,N,k)<<endl;
  return 0;
}
```

练 习

（1）小B在一个有n个城市m条道路的国家，每条道路连接的城市可以互相到达且每条道路小B都要花1步去走过它。现在他在1号城市，问他走p步最多能走到多少个不同的城市？

输入格式：第1行，三个正整数n,m,p，意义如题；接下来m行，每行两个整数u,v，表示存在一条连接u,v的无向边。

输出格式：1行，一个整数ans，表示走p步最多能走多少个不同的城市。

输入样例：

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

输出样例：

```
4
```

数据规模：

n<=100000,m<=5000000,p<=10000

（2）现在有一列有N个车厢的火车正在铁路上行驶。一开始给定Next_i，表示i号车厢的下一个车厢为Next_i号车厢，假如Next_i=0，表

示这是最后一个车厢。现在有Q个操作，操作分为两类，一种是将某号车厢移除，那么原来在它前面的车厢将和它后面的车厢相连，另一种是询问从某号车厢开始往后第C个车厢是哪号，假如不存在，则输出-1。

输入格式：第1行，两个整数N,Q；第2行，N个整数，第i个整数表示Next_i；接下来Q行，每行为0 i或1 i C的形式，若为0 i，则表示要将i号车厢移除，若为1 i C，则表示要询问i号车厢往后第C个车厢的号码。

输出格式：对于每个询问操作，输出1行，表示对应的车厢号，若不存在，则输出-1。

输入样例：

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

输出样例：

```
-1
4
5
```

数据规模：

N,Q <= 100000

询问的C之和不超过5000000。

（3）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。

输入格式：1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。

输出格式：1行，表达式的值。可直接用printf("%f\n",v)输出表达式的值v。

输入样例：

```
* + 11.0 12.0 + 24.0 35.0
```

输出样例：

1357.000000

提示：可使用atof(str)把字符串转换为一个double类型的浮点数。atof定义在math.h中。

（4）FJ准备教他的奶牛弹奏一首歌曲，歌曲由N种音节组成，编号为1到N，而且一定按照从1到N的顺序进行弹奏，第i种音节持续B_i个节拍，节拍从0开始计数，因此从节拍0到节拍B_1-1弹奏的是第1种音节，从B_1到B_1+B_2-1弹奏的是第2种音节，依此类推。

最近奶牛对弹琴不感兴趣了，它们感觉太枯燥了。所以为了保持奶牛们注意力集中，FJ提出Q个问题，问题的格式都是“第T次节拍弹奏的是哪种音节”。

每个问题对应一个T_i，请你帮奶牛来解决。

输入格式：第1行输入两个空格隔开的整数N和Q；第2至N+1行每行包含一个整数B_i；第N+2-N+Q+1行每行包含一个整数T_i。

输出格式：Q行，每行输出对应问题的答案。

输入样例：

3 5
2
1
3
2
3
4
0
1

输出样例：

2
3
3
1
1

数据规模：

N,Q <= 50000

（5）给定N个数Ai，以及一个正整数C，问有多少对i,j，满足Ai-Aj=C。

输入格式：第1行输入两个空格隔开的整数N和C；第2至N+1行每行包含一个整数Ai。

输出格式：输出问题的答案。

输入样例：

5 3
2
1
4
2
5

输出样例：

3

数据规模：

N <= 200000

所有数字保证在32位有符号整型内。

第7章　简单算法

“程序＝算法＋数据结构”。在这一章中，我们将介绍简单的算法。早在公元前1世纪的《周髀算经》中，“算法”的概念已经有所提及。经过长时间的发展，算法艺术可谓是凝结了人类智慧的精华。只有有了算法这个灵魂，我们的程序才会有强大的力量。接下来，我们将通过“高精度处理”、“枚举”、“模拟”、“简单动态规划”、“递归回溯”这几个算法体会一下算法的奥妙。

7.1　什么是算法

算法是解题方案的准确而完整的描述，是一系列解决问题的清晰指令，算法代表着用系统的方法描述解决问题的策略机制。也就是说，算法就是解决问题的办法，能够对一定规范的输入，在有限时间内获得所要求的输出。常见的算法有递推法、递归法、穷举法、贪心算法、分治法、动态规划法、回溯法等。

算法具有五个特征：

（1）有穷性：一个算法必须能够对任何合法的输入在执行有穷步之后结束，且每一步都可在有穷时间内完成。

（2）确切性：算法中每一条指令必须有确切的含义，读者理解时不会产生歧义。并且，在任何条件下，算法只有唯一的一条执行路径，即对于相同的输入只能得出相同的输出。

（3）可行性：算法中描述的操作都是可以通过已经实现的基本操作执行有限次来实现。

（4）输入：一个算法有零个或多个输入，这些输入取自于某个特定的对象的集合。

（5）输出：一个算法有一个或多个输出，这些输出是与输入有着某些特定关系的量。

接下来我们会通过介绍秦九韶算法的中心思想与实现方法，来对这五个特征加以直观体会。

秦九韶算法是中国南宋时期的数学家秦九韶提出的一种多项式简化算法。

对于一个一元n次多项式

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

我们在代入一个特定的x进行f(x)整体求值时，若用朴素方法处理，则需要经过$\frac{n(n+1)}{2}$次乘法操作和n次加法操作。而使用秦九韶算法计算，通过去除冗余的计算步骤，只需n次乘法操作和n次加法操作就可以计算出f(x)。

它的中心思想是，将上面的n次多项式改写成以下形式：

$$\begin{aligned} f(x) &= a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 \\ &= (a_n x^{n-1} + a_{n-1} x^{n-2} + \ldots + a_1) x + a_0 \\ &= ((a_n x^{n-2} + a_{n-1} x^{n-3} + \ldots + a_2) x + a_1) x + a_0 \\ &= \ldots \\ &= ((\ldots (a_n x + a_{n-1}) x + a_{n-2}) x + \ldots) x + a_1) x + a_0 \end{aligned}$$

用朴素方法处理时，我们在计算x，x^2，…，x^{n-1}，x^n等x的次幂时会造成很多的无效操作，而秦九韶算法通过多项式的层层嵌套，巧妙地避开了不必要的计算，从而减少计算操作的次数。秦九韶算法描述如下：

第一步：输入f(x)的系数a_0，a_1，…，a_n，输入x的值。

第二步：答案Ans初始化为a_n。

第三步：对于i=1，2，…，n，循环执行Ans=Ans*x+a_{n-i}。

第四步：输出Ans。

它的算法流程图如图7.1所示。

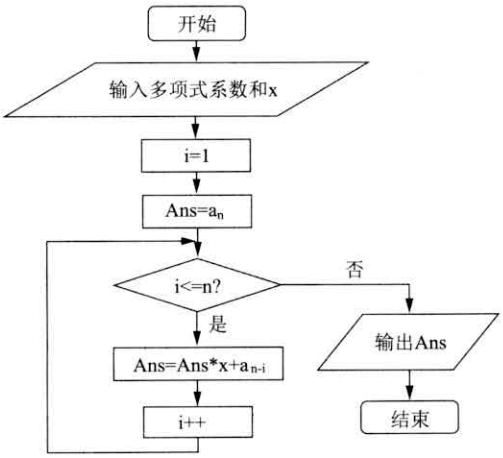


图7.1　秦九韶算法流程图

算法是程序设计的核心，是灵魂，有了算法就可以结合数据结构转变成程序。秦九韶算法对应的程序如下：

```
#include <iostream>
using namespace std;
int n,a[20],x,ans;
int main()
{
    cin>>n;
    for(int i=0;i<=n;i++)cin>>a[i];
    cin>>x;
    ans=a[n];
    for(int i=1;i<=n;i++)ans=ans*x+a[n-i];
    cout<<ans<<endl;
    return 0;
}
```

7.2 高精度数值处理

高精度数值处理是采用模拟算法对位数达上百位甚至更多位数的数字进行各种运算，包括加法、减法、乘法、除法等基础运算。

在C++中，数值的加、减、乘、除运算都已经在系统内部被定义好了，我们可以很方便地对两个变量进行简单的运算。然而其中变量的取值范围，以整数为例，最大的是long long类型，范围是$[-2^{63},2^{63})$。假如我们要对两个范围更大，比如在$[-2^{1000},2^{1000})$范围内的数进行简单基础运算，就不能用C++的内部运算器了。同理，在使用实数类型计算时，由于精确度有限，也不能精确计算小数点后的数百位的数值。那么我们该如何对两个大整数进行运算呢？

回想一下我们小学数学课上的加法“竖式”运算。

首先把加数与被加数的个位对齐，然后个位对个位、十位对十位、百位对百位，位位对应进行加法操作，有进位的要相应地进行处理。

对应地，我们不妨对每一个数位开一个整数变量进行存储。那么两个位分别相加，进位这些问题我们都可以用程序表示出来。而在实践中，对进行运算的两个数分别用两个数组进行储存会使问题变得十分方便。

以上便是对“高精度加法”算法思路的简单描述。

如图7.2所示，我们要计算出537+543的值，不妨设数组A={7,3,5}，

B = {3,4,5}，我们还需要维护一个进位数组C，以及一个答案数组Ans。如同竖式，我们逐位计算：

（1）考虑第0位（个位），Ans[0] = A[0] + B[0] = 7 + 3 = 10，但我们需要保证Ans[0] < 10，那么在第0位就发生了进位，C[1] = 10 / 10 = 1，Ans[0]变为0。

（2）考虑第1位，Ans[1] = A[1] + B[1] = 3 + 4 = 7，但由于在第0位存在进位，那么Ans[1]还要加上C[1]，因此Ans[1] = 7 + C[1] = 8，8 < 10，因此在第1位没有发生进位。

（3）考虑第2位，Ans[2] = A[2] + B[2] + C[2] = 10，发生了进位，因此C[3] = Ans[2] / 10 = 1，Ans[2]变为0。

```
      5   3   7
  +   5   4   3
  ---------------
  1   0   8   0
  1       1
```

图7.2　加法竖式运算示意图

（4）考虑第3位，Ans[3] = A[3] + B[3] + C[3] = 1。

（5）程序结束，Ans = 1 * 1000 + 0 * 100 + 8 * 10 + 0 = 1080。

一般在进行高精度加法时，我们要考虑高位进位的情况，因此要注意数组范围。

对于高精度减法的算法与加法类似，但需要注意高位可能发生被减为0以及借位的情况，要重新计算答案的位数。如图7.3所示。

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

图7.3　减法竖式运算示意图

其实这两个算法的精华，我们在小学时就已接触过了，就是各种法则，比如“各数位对齐”、“逢10进1”、“小数减大数向高位借位”等。这些烂熟于心的法则，其实就是对这个算法的精简的表述。

【例7.1】高精度加法。现有两个10000位的正整数，要求你将它们加起来输出。

分析：算法流程基本上面已经给出了，要注意的是两个10000位的数

的和可能是10001位的数。

程序eg7.1如下：

```
1  //eg7.1
2  #include <iostream>
3  #include <string>
4  #include <algorithm>
5  using namespace std;
6  const int MAXN = 10005;
7  int A[MAXN],B[MAXN],C[MAXN],Ans[MAXN],Len_A,Len_B,Len_Ans;
8  void Read(int *A,int &Len)
9  {
10   string cur;
11   cin >> cur;
12   Len = cur.length();
13   for(int i = 0;i < Len;i ++) A[i] = cur[i] - 48;
14   reverse(A,A + Len);          //对A[0]..A[Len-1]进行翻转
15 }
16 int main()
17 {
18   Read(A,Len_A);
19   Read(B,Len_B);
20   Len_Ans=max(Len_A,Len_B);
21   for(int i=0;i<=Len_Ans;i++)
22   {
23         Ans[i]=A[i]+B[i]+C[i];     //加上前一位的进位值
24         if(Ans[i]>9)C[i+1]=Ans[i]/10,Ans[i]-= 10;
                                      //处理进位
25   }
26   while(Ans[Len_Ans]>0)Len_Ans ++; //位数是否增加
27   for(int i = Len_Ans - 1;i >= 0;i --)
28         cout << Ans[i];
29   return 0;
30 }
```

说明：在程序的第14行我们需要将读入的数组整体翻转（reverse函数是C++库中的翻转函数，传入的两个参数代表翻转的左闭右开区间），因为我们读入是从高位开始读入的。

【例7.2】高精度减法。现有两个10000位的正整数，要求你将它们做减法后输出，保证A>=B。

分析：减法操作与加法相似，但需要注意借位以及最终结果为0的情况。

程序eg7.2如下：

```
1  //eg7.2
2  #include <iostream>
3  #include <string>
4  #include <algorithm>
5  using namespace std;
6  const int MAXN = 10005;
7  int A[MAXN],B[MAXN],C[MAXN],Ans[MAXN],Len_A,Len_B,Len_Ans;
8  void Read(int *A,int &Len)
9  {
10   string cur;
11   cin >> cur;
12   Len = cur.length();
13   for(int i = 0;i < Len;i ++) A[i] = cur[i] - 48;
14   reverse(A,A + Len);
15 }
16 int main()
17 {
18   Read(A,Len_A);
19   Read(B,Len_B);
20   Len_Ans = max(Len_A,Len_B);
21   for(int i = 0;i < Len_Ans;i ++)
22   {
23       Ans[i] = A[i] - B[i] - C[i];
24       if(Ans[i]<0)C[i+1]++,Ans[i]+=10;// 借位操作
25   }
26   while(Len_Ans>1 && Ans[Len_Ans-1]==0)Len_Ans--;
                                        // 判断位数是否减少
27   for(int i = Len_Ans - 1;i >= 0;i --)
28        cout << Ans[i];
29   return 0;
30 }
```

【例7.3】高精度乘法。给定两个1000位的正整数A、B，求出A*B的值。

分析：高精度乘法比高精度加法要复杂一些，我们不妨先仿照竖式计

算模拟一遍。如图7.4所示。

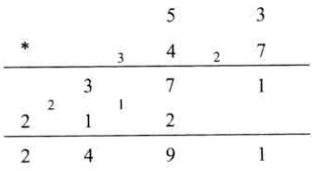


图7.4 乘法竖式运算示意图

观察图7.4，在计算53 * 47的值时，我们相当于将47拆为4 * 10 + 7来考虑，先计算出53*7，再计算出53*4，最后整体位移即可。一种更加数学化的语言是这样描述的：设A = (5 * 10 + 3)，B = (4 * 10 + 7)，那么A * B的值Ans根据乘法分配律有Ans = (5 * 4 * 100) + (5 * 7 + 3 * 4) * 10 + 3 * 7，那么我们可以每一位计算出其值，最后处理进位的情况即可。

比如说当前A = {5,3},B = {4,7},我们可以模拟这个过程，设有两个枚举变量i，j，表示当前A,B中扫描到哪个位置。

（1）i = 0，j = 0，Ans[i+j] = Ans[0] = A[0]*B[0] = 21。

（2）i = 0，j = 1，Ans[i+j] = Ans[1] = A[0]*B[1] = 12。

（3）i = 1，j = 0，Ans[i+j] = Ans[1] = Ans[1] + A[1] * B[0] = 12 + 35 = 47。

（4）i = 1，j = 1，Ans[i+j] = Ans[2] = A[1] * B[1] = 20。

（5）Ans = {20,47,21}，从低位向高位处理进位，最终得到Ans = {2,4,9,1}。

这个算法的精髓就是将一个数看成若干个10的次幂的和，利用乘法分配律各个计算，最终再处理进位的问题，更具体的细节可以看下面的代码。

```
1  //eg7.3
2  #include <iostream>
3  #include <string>
4  #include <algorithm>
5  using namespace std;
6  const int MAXN = 5005;
7  int A[MAXN],B[MAXN],Ans[MAXN],Len_A,Len_B,Len_Ans;
8  void Read(int *A,int &Len)
9  {
10   string cur;
11   cin >> cur;
12   Len = cur.length();
```

```
13    for(int i = 0;i < Len;i ++) A[i] = cur[i] - 48;
14    reverse(A,A + Len);
15 }
16 int main()
17 {
18    Read(A,Len_A);
19    Read(B,Len_B);
20    Len_Ans = Len_A + Len_B - 1;
21    for(int i = 0;i < Len_A;i ++)
22      for(int j = 0;j < Len_B;j ++)
23         Ans[i + j] += A[i] * B[j];
24    for(int i = 0;i < Len_Ans;i ++)
25      if(Ans[i]>9)Ans[i+1]+=Ans[i]/10,Ans[i]%=10;
                                          // 最后进行进位处理
26    while(Ans[Len_Ans])Len_Ans ++;
27    for(int i = Len_Ans - 1;i >= 0;i --)
28      cout << Ans[i];
29    return 0;
30 }
```

注意： 两个长度为n的正整数的乘积的位数可能达到2*n。

【例7.4】 高精度除法。给定两个1000位的正整数A、B，求出A/B的商和余数。

分析： 高精度除法同样采用竖式除法的方法，竖式除法如图7.5所示。

```
          1  0  6
      ┌──────────
  1 9 ) 2  0  1  6
        1  9
      ──────────
           1  1  6
           1  1  4
      ──────────
                 2
```

图7.5 竖式除法运算示意图

高精度除法从被除数的高位到低位依次计算出商对应位上的数字，记录在除法过程中的余数为C，高精度除法就转变为C除以B的操作，除法操作转换为减法操作，用C减去B，每减一次对应位上的商加1，直到不够减为止，再处理下一位，先产生新的余数C=C*10+A[i]，再重复上述操作。

以样例为例介绍高精度除法的过程：

（1）i=3，余数C=2，B=19，商对应的位的值Ans[3]=0。

（2）i=2，C=C*10+A[2]=20，可以从C减去B一次，因此Ans[2]=1，C=1。

（3）i=1，C=C*10+Ans[1]=11，不够减B，Ans[1]=0。

（4）i=0，C=C*10+Ans[0]=116，可以从C中减去B共6次，因此Ans[0]=6，C=2。

（5）至此，除法结束。Ans=106，C=2。

对应的程序eg7.4如下：

```
1  //eg7.4
2  #include <iostream>
3  #include <string>
4  #include <algorithm>
5  using namespace std;
6  const int MAXN = 5005;
7  int A[MAXN],B[MAXN],Ans[MAXN],C[MAXN],Len_A,Len_B,Len_Ans;
8  void Read(int *A)
9  {
10   string cur;
11   cin >> cur;
12   A[0] = cur.length();
13   for(int i = 1;i <=A[0];i ++) A[i] = cur[i-1] - 48;
14   reverse(A+1,A + A[0]+1);
15 }
16 bool Big()              // 比较余数C与除数B的大小，如果C>=B，
                              则返回true，否则返回false
17 {
18   if(C[0]>B[0]) return true;
19   if(C[0]<B[0]) return false;
20   for(int i=C[0];i>=1;i--)
21     if(C[i]<B[i]) return false;
22     else if (C[i]>B[i]) return true;
23   return true;
24 }
25 void Minus()            // 从C中减去B
26 {
27   int c=0;
28   for(int i=1;i<=C[0];i++)
```

```
29    {
30      C[i]-=B[i]+c;
31      if(C[i]<0){C[i]+=10;c=1;}
32      else c=0;
33    }
34    while(C[0]>1 && C[C[0]]==0) C[0]--;
35  }
36  void output(int *A)
37  {
38    for(int j=A[0];j>=1;j--)cout<<A[j];
39    cout<<endl;
40  }
41  int main()
42  {
43    Read(A);
44    Read(B);
45    C[0]=0;
46    for(int i=A[0];i>=1;i--)
47    {
48      for(int j=C[0];j>=1;j--)C[j+1]=C[j];
49      C[1]=A[i];
50      C[0]++;
51      while(Big())
52      {
53        Minus();
54        Ans[i]++;
55      }
56    }
57    Ans[0]=A[0];
58    while(Ans[0]>1 && Ans[Ans[0]]==0) Ans[0]--;
59    output(Ans);
60    output(C);
61    cin>>A[0];
62    return 0;
63  }
```

7.3 简单枚举算法

枚举，也叫做穷举法。从字面意思来理解，就可以知道是指列出所有

情况，并逐一进行分析。这类算法的特点在于它的正确性一般比较显然，十分的直观，编写上大多情况下也较为容易。但是缺点在于，由于需要枚举所有情况，当情况很多时效率一般很低，甚至可能达到指数级。大多数的搜索算法本质上都是枚举算法，只是在其中加入了不同的优化，从而提升了时间效率，使得正确性和效率之间达到更好的平衡。

【例7.5】质数。定义质数为因数只有1和其本身的数，对于n组询问，试判断每个数是否为质数。

输入格式：读入第1行一个正整数n，表示有n组询问；接下来n行，每行一个正整数m，表示询问m是否为质数，是，则输出“yes”，否则输出“no”。

输出格式：n行，每行一个字符串，代表答案。

数据范围：

$n \leqslant 10^3$，$1<m \leqslant 10^{10}$

分析：从定义入手，我们考虑枚举每一个1到m的数，依次判断它是不是m的因数，若统计得出m一共只存在2个因数，则能判断m是一个质数；否则m是一个合数。

面对10^6范围内的m，使用穷举法是可行的，但随着m范围的增大，枚举所有的数就显得十分冗余，并且过大的时间复杂度是我们所不能接受的。

通过观察，我们可以发现一个数的因数都是成对出现的，完全平方数的平方根这个因子除外，比如$30=1\times30=2\times15=3\times10=5\times6$，1和30是一对因子，2和15是一对因子，3和10是一对因子，5和6是一对因子，而且每一对因子中一定有一个是小于或等于$\sqrt{m}$的（设a*b=m，a<=b，则有a^2<=a*b=m，因此a<=$\sqrt{m}$）。而且只需要出现一个非1且非本身的约数，我们就可以判定这个数不是质数。于是我们可以仅仅枚举2到$\sqrt{m}$中是否有数为m的约数就可以了。

程序eg7.5如下：

```
1  //eg7.5
2  #include <iostream>
3  using namespace std;
4  int main()
5  {
6    int n;
```

```
7     long long m;
8     cin >> n;
9     for(;n;n --)
10    {
11        cin >> m;
12        if(m == 1) {cout << "no" << endl;continue;}
13        bool IsPrime = 1;
14        for(int i = 2;i * i <= m;i ++)
15            if(m % i == 0)
16            {
17              IsPrime = 0;
18              break;
19            }
20        if(IsPrime)cout<<"yes"<<endl;else cout<<"no"<<endl;
21    }
22    return 0;
23 }
```

说明：上面的算法对于m=1时需要特判。

【例7.6】垃圾炸弹。2014年足球世界杯（2014 FIFA World Cup）开踢啦！为了方便球迷观看比赛，街道上很多路口都放置了直播大屏幕，但是人群散去后总会在这些路口留下一堆垃圾。为此政府决定动用一种最新发明——“垃圾炸弹”。这种“炸弹”利用最先进的量子物理技术，爆炸后产生的冲击波可以完全清除波及范围内的所有垃圾，并且不会产生任何其他不良影响。炸弹爆炸后冲击波是以正方形方式扩散的，炸弹威力（扩散距离）以d给出，表示可以传播d条街道。

图7.6是一个d=1的“垃圾炸弹”爆炸后的波及范围。

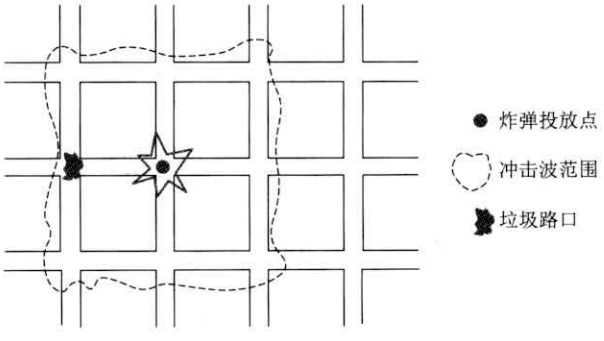


图7.6 “垃圾炸弹”爆炸冲击波范围

假设城市的布局为严格的[0,1024]*[0,1024]的网格状，由于财政问题，市政府只买得起一枚“垃圾炸弹”，希望你帮他们找到合适的投放地点，使得一次清除的垃圾总量最多（假设垃圾数量可以用一个非负整数表示，并且除设置大屏幕的路口以外的地点没有垃圾）。

输入格式：第1行给出“炸弹”威力d；第2行给出一个数组n，表示设置了大屏幕(有垃圾)的路口数目；接下来n行，每行给出三个数字x,y,i，分别代表路口的坐标(x,y)以及垃圾数量i。点坐标(x,y)保证是有效的（区间在0到1024之间），同一坐标只会给出一次。

输出格式：输出能清理垃圾最多的投放点数目，以及能够清除的垃圾总量。

输入样例：

```
1
2
4 4 10
6 6 20
```

输出样例：

```
1 30
```

数据范围：

d <= 50，n <= 1000

方法1：假如n <= 50，那么我们可以直接枚举最终的投放点(x,y)，然后枚举n个路口判断这个投放点是否能覆盖到这些路口，最终输出答案即可。但这种算法的运算时间为$1024^2 * n$，当n的规模到了1000后会超时。

注意到题目中保证了d <= 50，那么是否能从这里入手？维护数组cnt[i][j]表示炸弹投放点在(i,j)时的答案，枚举每个垃圾(x,y)，那么它能对[x-d,x+d] * [y-d,y+d]的投放点造成加1的影响，枚举[x-d,x+d] * [y-d,y+d]的所有点并对相应的cnt加1，最终扫一遍cnt就能得到答案了。

这个算法的复杂度是$O(n * d^2)$，能1秒时限内完美解决这题。

程序eg7.6_1如下：

```
1  //eg7.6_1
2  #include <iostream>
3  using namespace std;
4  const int MAXN = 1030;
5  using namespace std;
```

```
int cnt[MAXN][MAXN],n,d;
void work()
{
  for(int i = 0;i < 1025;i ++)
      for(int j = 0;j < 1025;j ++)
          cnt[i][j] = 0;
  cin >> d;
  cin >> n;
  for(int i = 1;i <= n;i ++)
  {
      int x,y,l;
      cin >> x >> y >> l;
      for(int xr = -d;xr <= d;xr ++)
            if(x + xr >= 0 && x + xr < 1025)
                for(int yr = -d;yr <= d;yr ++)
                    if(y + yr >= 0 && y + yr < 1025)
                      cnt[x + xr][y + yr] += l;
  }
  int maxval = -1,num = 0;
  for(int i = 0;i < 1025;i ++)
      for(int j = 0;j < 1025;j ++)
      if(cnt[i][j]>maxval)maxval=cnt[i][j],num=1;
      else if(cnt[i][j] == maxval)num ++;
  cout << num << " " << maxval << endl;
}
int main()
{
  work();
  return 0;
}
```

方法2：可以换个枚举的角度，枚举投放点的位置(x,y)，计算[x-d,x+d]*[y-d,y+d]中垃圾数量，这里就涉及矩阵中子矩阵的元素和，如图7.7所示。

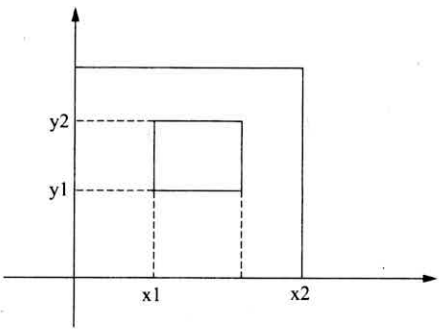


图7.7 子矩阵示意图

用Cnt[i][j]表示以(0,0)为左下角，(i,j)为右上角的子矩阵中元素的和：

Cnt[i][j]=Cnt[i-1][j]+Cnt[i][j-1]-Cnt[i-1][j-1]+a[i][j]

图7.7中子矩阵元素和可以用下式求得：

Cnt[x2][y2]-Cnt[x2][y1-1]-Cnt[x1-1][y2]+Cnt[x1-1][y1-1]

时间复杂度为O(1024*1024)，对应的程序7.6_2如下：

```
1  //eg7.6_2
2  #include <iostream>
3  #include <algorithm>
4  using namespace std;
5  const int MAXN = 1030;
6  using namespace std;
7  int cnt[MAXN][MAXN],n,d;
8  int main()
9  {
10   for(int i = 0;i < 1025;i ++)
11       for(int j = 0;j < 1025;j ++)
12          cnt[i][j] = 0;
13   cin >> d;
14   cin >> n;
15   for(int i = 1;i <= n;i ++)
16   {
17       int x,y,l;
18       cin >> x >> y >>l;
19       cnt[x][y]=l;
20   }
21   for(int i=1;i<1025;i++)
```

```
    {
        cnt[0][i]+=cnt[0][i-1];
        cnt[i][0]+=cnt[i-1][0];
    }
    for(int i=1;i<1025;i++)
        for(int j=1;j<1025;j++)
            cnt[i][j]+=cnt[i-1][j]+cnt[i][j-1]-cnt[i-1][j-1];
    int maxval = -1,num = 0;
    for(int i = 0;i < 1025;i ++)
        for(int j = 0;j < 1025;j ++)
        {
            int li=max(i-d,0),ri=min(1024,i+d);
            int dj=max(j-d,0),uj=min(1024,j+d);
            int t=cnt[ri][uj];
            if(dj>0)t-=cnt[ri][dj-1];
            if(li>0)t-=cnt[li-1][uj];
            if(li>0 && dj>0)t+=cnt[li-1][dj-1];
            if(t > maxval)maxval = t,num = 1;
            else if(t == maxval)num ++;
        }
    cout << num << " " << maxval << endl;
    return 0;
}
```

7.4 模拟算法

模拟算法，顾名思义，就是指程序完全按照题目描述的方式运行，从而获得相应的结果。模拟算法的特点在于它基本上不需要你去想怎么样去解决问题，而是给出解决方案，要你按照解决方案一步一步去实现。

【例7.7】神奇的幻方。幻方是一种很神奇的N*N矩阵：它由数字1，2，…，N*N组成。且每行、每列、两条对角线上的数字和都相同。

当N为奇数时，我们可以通过以下方式构建一个幻方：

（1）将1写在第一行的中间。

（2）按如下方式从小到大依次填写每个数K(K=2,3,…,N*N)：

①若K-1在第1行但不在最后一列，则将K填在最后1行，K-1所在列的右一列。

②若K-1在最后1列但不在第1行，则将K填在第1列，K-1所在行的上一行。

③若K-1在第1行最后1列，则将K填在K-1的正下方。

④若K-1既不在第1行，也不在最后1列，如果K-1的右上方还未填数，则将K填在K-1的右上方，否则将K填在K-1的正下方。

现给定N，请按上述方法构造N*N的幻方。

输入格式：输入文件只有1行，包含一个整数N，即幻方的大小。

输出格式：输出文件包含N行，每行N个整数，即按上述方法构造出的N*N的幻方。相邻两个整数之间用单个空格隔开。

输入样例：

```
3
```

输出样例：

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

数据范围：对于100%的数据，1<=N<=39且N为奇数。

分析： 题目里已经很明确地指出了“请按上述方法构造N×N的幻方”。显然这里只需要按照上面给出的规则用程序实现出来就可以了。

程序eg7.7如下：

```
1  //eg7.7
2  #include <iostream>
3  using namespace std;
4  const int N = 50;
5  int n, m, a[N][N], x, y;
6  int main()
7  {
8      cin >> n;
9      a[1][n / 2 + 1] = 1;
10     x = 1, y = n / 2 + 1;
11     m = n * n;
12     for(int i = 2; i <= m; ++ i)
13     {
14         int nx, ny;
15         if(x == 1 && y != n)
16         {
```

```
17              nx = n;
18              ny = y + 1;
19          }
20          else if (y == n && x != 1)
21          {
22              ny = 1;
23              nx = x - 1;
24          }
25          else if (x == 1 && y == n)
26          {
27              nx = x + 1;
28              ny = y;
29          }
30          else if (x != 1 && y != n)
31          {
32              if(!a[x - 1][y + 1])
33                  nx = x - 1, ny = y + 1;
34              else
35                  nx = x + 1, ny = y;
36          }
37          a[nx][ny] = i;
38          x = nx, y = ny;
39      }
40      for(int i = 1; i <= n; ++ i)
41      {
42          for(int j = 1; j < n; ++ j)
43              cout << a[i][j] << " ";
44          cout << a[i][n] << endl;
45      }
46      return 0;
47  }
```

【例7.8】众数。对于一个长度为n的序列$\{a_n\}$来说，其众数被定义为出现次数最多的数。

现在给定一个长度为n的序列，yc想要你求出它的众数是多少。

当然众数可能有多个，你只需要输出最小的一个就可以了。

输入格式：第1行输入n，第2行输入n个数。

输出格式：输出众数。

输入样例：

6

3 5 7 5 3 1

输出样例：

3

数据范围：

$n<10^6$, $0<a_i<1000$

分析：按定义去做，我们需要找到出现次数最多的数，那么可以考虑用一个桶存下来每个不同的数出现的次数，并找出最小的一个就可以了。

程序eg7.8如下：

```
1  //eg7.8
2  #include <iostream>
3  using namespace std;
4  const int N = 1000005, M = 1005;
5  int n, a[N], s[M];
6  int main()
7  {
8    cin >> n;
9    for(int i=1; i<=n; ++i)cin >> a[i];
10   for(int i=1; i<=n; ++i)++s[ a[i] ];
11       int ret = 0;
12   for(int i=1; i<M; ++i )
13       if(s[i]>s[ret] ) ret = i;
14   cout << ret << endl;
15   return 0;
16 }
```

7.5 简单动态规划

动态规划（Dynamic Programming，DP）是对一类最优化问题的解决方法。动态规划问题每次决策时依赖于当前状态，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，这种多阶段最优化决策解决问题的过程就称为动态规划。

动态规划的使用前提：

（1）最优化原理(最优子结构性质)：在解决原问题时，作出一个决策后，余下的问题是与原问题性质相同的子问题，只是规模或参数发生变化，

要使原问题最优，子问题必须同样最优。一个问题满足最优化原理又称其具有最优子结构性质。

（2）无后效性：利用最优化原理可以把原问题的求解转变为与原问题性质相同的子问题的求解，我们只关心子问题的最优值，我们并不关心子问题求解时的决策序列，子问题的决策序列也不会影响原问题的决策。

动态规划解决问题的步骤：

（1）确立状态：状态是一个数学形式，能刻画一个最优解的结构特征，能够把题目中的必不可少的要素包含进来，状态的定义要严谨。

（2）确定状态转移方程和边界条件：递归地定义状态的最优解，把原问题的求解转移到与原问题性质相同的子问题的求解，对于某些特殊情况赋予特殊的初值。

（3）程序实现：采用记忆化搜索避免重复计算某些状态的值，也可以使用迭代法，即找出某种顺序保证递归式子中等号右边的状态会在等号左边的状态之前先计算出来。

动态规划的设计方法有很多，技巧也不一。下面我们通过几个例子来看看动态规划解决问题的过程。

【例7.9】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。

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

在上面的样例中，从7到3到8到7到5的路径产生了最大的和30。

输入格式：第1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。所有金字塔中的整数是非负的且不大于100。

输出格式：单独的一行，包含那个可能得到的最大的和。

输入样例：

```
5
7
```

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

输出样例：

30

方法1：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；当x<N时，未到达递归出口，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])。

程序eg7.9_1如下：

```
1  //eg7.9_1
2  #include <iostream>
3  using namespace std;
4  const int MAXN = 1005;
5  int A[MAXN][MAXN],F[MAXN][MAXN],N,Ans;
6  void Dfs(int x,int y,int Curr)
7  {
8    if(x==N)
9    {
10     if(Curr>Ans)Ans=Curr;
11     return;
12   }
13   Dfs(x+1,y,Curr+A[x+1][y]);
14   Dfs(x+1,y+1,Curr+A[x+1][y+1]);
15 }
16 int main()
17 {
18   cin >> N;
19   for(int i = 1;i <= N;i ++)
20     for(int j = 1;j <= i;j ++)
21       cin >> A[i][j];
22       Ans =0;
```

```
23        Dfs(1,1,A[1][1]);
24    cout<<Ans<<endl;
25    return 0;
26 }
```

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$O(2^{N-1})$，N大了会超时。

方法2：记忆化搜索。方法1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1中的搜索进行改造。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。

对应的程序eg7.9_2如下：

```
1  //eg7.9_2
2  #include <iostream>
3  #include <algorithm>
4  using namespace std;
5  const int MAXN = 505;
6  int A[MAXN][MAXN],F[MAXN][MAXN],N;
7  int Dfs(int x,int y)
8  {
9    if (F[x][y]==-1)
10   {
11     if(x==N)F[x][y]=A[x][y];
12     else F[x][y]=A[x][y]+max(Dfs(x+1,y),Dfs(x+1,y+1));
13   }
14   return F[x][y];
15 }
16 int main()
17 {
18    cin >> N;
19    for(int i = 1;i <= N;i ++)
20      for(int j = 1;j <= i;j ++)
21         cin >> A[i][j];
22    for(int i = 1;i <= N;i ++)
23      for(int j = 1;j <= i;j ++)
24         F[i][j] = -1;
25    Dfs(1,1);
26    cout << F[1][1] << endl;
27    return 0;
28 }
```

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3： 动态规划。方法2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，路径是由各个点串联而成，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。最终答案如下：

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：

（1）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]。

（2）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]，它们是上一行的元素，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。

综上可知，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：

（1）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。

（2）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2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。

对于样例，计算出来的结果F[x][y]如表7.1所示。

表7.1 样例对应的F[x][y]值

x y	1	2	3	4	5
1	7				
2	10	15			
3	18	16	15		
4	20	25	20	19	
5	24	30	27	26	24

时间复杂度为$O(N^2)$。对应的程序eg7.9_3如下：

```
1  //eg7.9_3
2  #include <iostream>
3  #include <algorithm>
4  using namespace std;
5  const int MAXN = 1005;
6  int A[MAXN][MAXN],F[MAXN][MAXN],N;
7  int main()
8  {
9    cin >> N;
10   for(int i = 1;i <= N;i ++)
11       for(int j = 1;j <= i;j ++)
```

```
12              cin >> A[i][j];
13    F[1][1] = A[1][1];
14    for(int i = 2;i <= N;i ++)
15        for(int j = 1;j <= i;j ++)
16            F[i][j]=max(F[i-1][j-1],F[i-1][j])+A[i][j];
17    int ans =0;
18    for(int i = 1;i <= N;i ++)
19        ans = max(ans,F[N][i]);
20    cout << ans << endl;
21    return 0;
22 }
```

【例7.10】最长不下降子序列。给定一个n个数的序列A，问最长的不下降子序列长度为多少。对于一个序列p1< p2< … < pk,其为不下降子序列需要满足：$A[p_1] <= A[p_2] <= A[p_3] <= \cdots <= A[p_k]$。

输入格式：第1行一个整数n；第2行n个整数，表示A[i]。

输出格式：一个整数，表示最长的不下降子序列的长度。

输入样例：

```
7
1 7 3 5 9 4 8
```

输出样例：

```
4
```

数据范围：

```
n <= 5000
```

分析：题目要求选择其中一些数构成最长的不下降子序列，每个数顶多有“选和不选”两种选择，因此可以用搜索来完成。

定义搜索Dfs(int Pre,int Curr,int Currlen)，其中Pre表示当前不下降子序列的最后一个元素是A[Pre]，现在搜索考虑第Curr个元素，当前不下降子序列的长度为Currlen，通过确定A[Curr]选还是不选进行，这样可以把所有不下降子序列都搜索出来。

在搜索过程中可以加上最优化剪枝。程序eg7.10_1如下：

```
1  //eg7.10_1
2  #include <iostream>
3  using namespace std;
4  const int MAXN = 5005;
```

```
5  int A[MAXN],N,Ans;
6  void Dfs(int Pre,int Curr,int Currlen)
7  {
8    if(Curr==N+1)
9    {
10     if(Currlen>Ans)Ans=Currlen;
11     return;
12   }
13   if(A[Curr]>=A[Pre])Dfs(Curr,Curr+1,Currlen+1);
                              // 选A[Curr]
14   if(Currlen+N-Curr>Ans)Dfs(Pre,Curr+1,Currlen);
                              // 不选A[Currlen]，最优性剪枝
15 }
16 int main()
17 {
18  cin >> N;
19  for(int i = 1;i <= N;i ++)cin >> A[i];
20  Ans =0;
21  A[0]=-(1<<30);
22  Dfs(0,1,0);
23  cout<<Ans<<endl;
24  return 0;
25 }
```

以上程序超时，因为进行了重复搜索，拿样例1 7 3 5 9 4 8来说，搜索过程中会调用Dfs(3,4,1)和Dfs(3,4,2)，分别对应着3和1 3这两个中间状态，在实际执行过程中Dfs(3,4,1)和Dfs(3,4,2)后续的搜索是完全一样的，这里存在重复搜索。再具体一点，不下降子序列的最后一个元素相同时执行的后续搜索是一样的。

根据上述重复搜索的本质分析，我们增加记忆化，设立状态，开设全局数组，保存已经搜索过的状态，本质上就是动态规划，这里我们直接讲动态规划的做法。

1. 确定状态

定义F[i]表示以A[i]为首的最长不下降子序列的长度，也可以表示以A[i]为结尾的最长不下降子序列的长度。我们选择后者，答案为

$$\max\{F[1],F[2],...,F[N]\}$$

2. 确定状态转移方程和边界条件

如果最长不下降子序列只有A[i]一个元素，则F[i]=1；如果最长不下

降子序列不止A[i]一个元素，那一定有前驱，通过考虑前驱是哪个元素进行状态转移。

假设前驱是A[j]，则必须满足1<=j<=i-1且A[j]<=A[i]，这种情况就把以A[i]为结尾的最长不下降子序列看做是在“以A[j]为结尾的最长不下降子序列”后面增加一个A[i]，而“以A[j]为结尾的最长不下降子序列”是跟F[i]性质一样的子问题，满足最优化原理和无后效性。以A[j]为前驱的情况下，F[i]=F[j]+1。A[j]有多种选择，选择其中最大的F[j]即可。

综上，状态转移方程为：

$$F[i]=\max\{F[j],0\}+1 \quad \text{其中 } 1<=j<=i-1 \text{ 且 } A[j]<=A[i]$$

边界条件：

$$F[1]=1$$

3. 程序实现

记忆化搜索和迭代法都可以。

观察状态转移方程，计算F[i]时用到F[j]，其中1<=j<=i-1，因此我们按照i从小到大来计算就可以用迭代法完成。

样例计算出来的F[i]结果如表7.2所示。

表7.2 样例对应F[i]结果

i	1	2	3	4	5	6	7
A[i]	1	7	3	5	9	4	8
F[i]	1	2	2	3	4	3	4

时间复杂度为$O(N^2)$。程序eg7.10_2如下：

```
1  //eg7.10_2
2  #include <iostream>
3  using namespace std;
4  const int MAXN = 5005;
5  int A[MAXN],F[MAXN],N,Ans;
6  int main()
7  {
8     cin >> N;
9     for(int i = 1;i <= N;i ++)cin >> A[i];
10    F[1]=1;
11    Ans=1;
```

```
12    for(int i=2;i<=N;i++)
13    {
14       F[i]=1;
15       for(int j=1;j<=i-1;j++)
16          if(A[j]<=A[i]&&F[j]+1>F[i])F[i]=F[j]+1;
17       if(F[i]>Ans)Ans=F[i];
18    }
19    cout<<Ans<<endl;
20    return 0;
21 }
```

【例7.11】最长公共子序列。给定两个字符串S,T，求S与T的最长公共子序列。

输入格式：第1行一个字符串，表示字符串S；第2行一个字符串，表示字符串T。

输出格式：一个整数，表示最长的公共子序列长度。

数据范围：

|S|,|T| <= 5000

输入样例：

bbabxbc

batxbbc

输出样例：

5

分析：与上题类似的，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：

（1）S[x]不在公共子序列中：该情况下F[x][y]=F[x-1][y]。

（2）T[y]不在公共子序列中：该情况下F[x][y]=F[x][y-1]。

（3）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。

综上可知，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|)。对应的程序eg7.11如下：

```
1  //eg7.11
2  #include <iostream>
3  #include <string>
4  #include <algorithm>
5  using namespace std;
6  const int MAXN = 5005;
7  string S,T;
8  int F[MAXN][MAXN];
9  int main()
10 {
11    cin >> S;
12    cin >> T;
13    int ls = S.length(),lt = T.length();
14    for(int i = 1;i <= ls;i ++)
15       for(int j = 1;j <= lt;j ++)
16       {
17           F[i][j] = max(F[i - 1][j],F[i][j - 1]);
18           if(S[i - 1] == T[j - 1])
19               F[i][j] = max(F[i][j],F[i - 1][j - 1] + 1);
20       }
21    cout << F[ls][lt] << endl;
22    return 0;
23 }
```

7.6 用递归实现回溯算法

回溯算法是所有搜索算法中最为基本的一种算法，本质上是搜索算法的一种控制策略，采用了一种“走不通就掉头”的思想。有些问题在往下一步搜索时，每一步都出现很多分支，为了求得问题的解，我们必须对每一个分支都要试探，看看是否符合题目要求，若发现某一步试探不合要求，则停止从该分支往下的试探，立即返回到上一步去试探其他分支，直到找到问题的解。如果回溯到问题的初始状态之后还要返回，则表示该问题无解。

回溯法的应用范围很广，只要能把待求解的问题分成不太多的步骤，每个步骤又只有不太多的选择，都可以考虑应用回溯法。

下面通过几个具体的例子来讲解。

【例7.12】全排列。给定N（N<10），按照字典序输出所有的N排列。如当N=3时，应该输出：

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

分析：N排列是由1,2,...,N组成的一个排列，每个排列有N个数字，每个数字大于等于1小于等于N，且互不相同。该题符合回溯法的应用范围，确定N排列可以分成N个步骤，按照从左到右的顺序确定每一个数，第x步确定排列的第x个数时，前面x-1个数已经确定好，可以从1枚举到N考虑第x个数并判断该数是否在前面x-1个数中曾出现过，如果没有出现过就可以选择该数，存储下来并接着确定第x+1个数，当N个位置上的数都确定完则找到一个N排列。每执行一个递归函数，程序会自动回退到上一层对其他分支进行试探，从而把所有方案都能搜索出来。

程序eg7.12_1如下：

```
1  //eg7.12_1
2  #include <algorithm>
3  using namespace std;
4  const int N = 15;
5  int n, a[N];
```

```
6  void search(int x)      // 确定排列的第 x 个数
7  {
8     if(x==n+1)           // 到达递归出口，产生一个新的排列
9     {
10       for(int i=1;i<=n;i++)cout<<a[i]<<" ";
                          // 输出新产生的排列
11       cout<<endl;
12       return;
13    }
14    for(int i=1;i<=n;i++)  // 枚举第 x 个数的取值
15    {
16       bool ok=true;
17       for(int j=1;j<x;j++)
                          // 判断 i 是否在排列的前 x-1 个数中出现过
18          if(a[j]==i){ok=false;break;}
19       if(ok)
20       {
21          a[x]=i;         // 保存结果，把 i 存储进排列的第 x 位
22          search(x+1);    // 调用 search(x+1) 确定第 x+1 个数
23          a[x]=0;         //执行完search(x+1)后会回退到该处，需
                              要恢复到调用前的状态以试探其他取值，此
                              处a[x]=0也可以不要，因为新的赋值会把
                              a[x]原来的值覆盖掉
24       }
25    }
26 }
27 int main()
28 {
29    cin >> n;
30    search(1);
31    return 0;
32 }
```

如图 7.8 所示，以 N=3 为例画出上面程序对应的搜索树，括号中为 a 数组的取值。

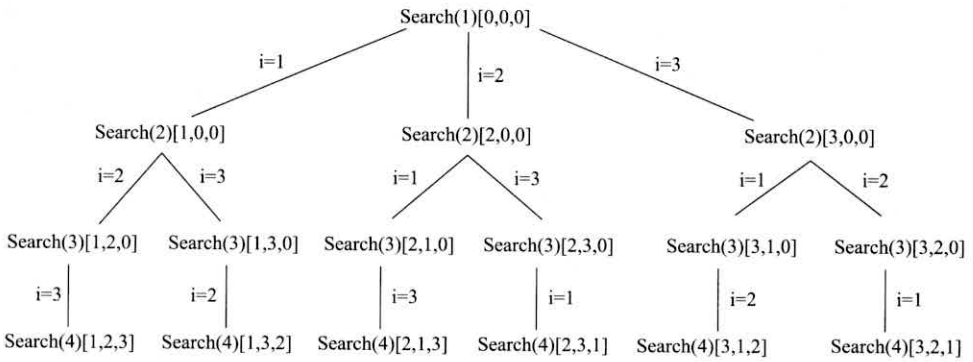


图7.8 全排列N=3的搜索树

程序7.12_1效率不高，主要是在“判断i是否在排列的前x-1个数中出现过”时使用了普通的枚举法，我们可以对这里进行优化。定义Used[]数组，如果i使用过，则Used[i]记为true，否则记为false。对应的程序eg7.12_2如下：

```
1  //eg7.12_2
2  #include <algorithm>
3  #include <iostream>
4  using namespace std;
5  const int N = 15;
6  int n, a[N];
7  bool used[N];
8  void search(int x)                    // 确定排列的第 x 个数
9  {
10     if(x==n+1)                        // 到达递归出口，产生一个新的排列
11     {
12         for(int i=1;i<=n;i++)cout<<a[i]<<" ";
13         cout<<endl;
14         return;
15     }
16     for(int i=1;i<=n;i++)             // 枚举第 x 个数的取值
17     {
18         if(!used[i])
19         {
20           a[x]=i;                     // 保存结果，把 i 存储进排列的第 x 位
21           used[i]=true;               // 把 i 标记为已经使用过
22           search(x+1);                // 调用 search(x+1) 确定第 x+1 个数
23           used[i]=false;
```

```
                                    // 回溯，i 重新标记为未使用以尝试其他情况
        }
    }
}
int main()
{
    cin >> n;
    search(1);
    return 0;
}
```

【例 7.13】N 皇后问题。在 N*N(N<=10) 的棋盘上放 N 个皇后，使得她们不能相互攻击。两个皇后能相互攻击当且仅当它们在同一行，或者同一列，或者同一条对角线上。找出一共有多少种放置方法。

分析：题目要求 N 个皇后放在 N*N 的棋盘上，并且任意两个皇后不能放在同一行同一列或同一对角线上。

先考虑“不在同一行”这个条件，这个好解决，只要每一行放一个皇后即可。因此我们定义数组 a[]，其中 a[i] 表示第 i 行皇后放置第 a[i] 列，a[i] 可以取 1 到 N。

定义递归函数 Queen(x) 表示准备放置第 x 行的皇后，在执行 Queen(x) 时前面 1 到 x-1 行的皇后已经确定好，我们可以从 1 到 N 枚举 j 作为 a[x] 的值，j 必须与 a[1] 到 a[x-1] 互不相同，这样可以确保不在同一列，还要判断第 x 行的皇后的位置 (x,j) 与前面的所有皇后 (i,a[i])(1<=i<=x-1) 不在同一对角线上，对角线有主对角线与副对角线之分。图 7.9 中为 4*4 棋盘的主副对角线示意图。

观察同一主对角线上单元格行号和列号之间的关系，如果第 x 行的皇后 (x,j) 与前面第 i 行的皇后 (i,a[i]) 在同一主对角线上，则满足 x-i=j-a[i]，如果在同一副对角线上，则满足 x-i=a[i]-j。在给第 x 行选择皇后放置列号 j 时，可以枚举前面 x-1 个皇后判断是否与第 x 行的皇后在同一列或同一对角线上。

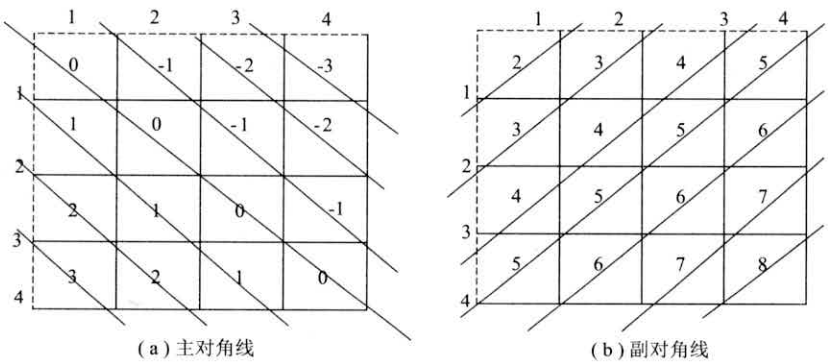
上述判断方法效率低下，对于“判断是否在同一列”我们完全像全排列那样定义数组 C[]，C[j] 表示列号 j 是否被用过，true 表示用过，false 表示未用过。

而对于“判断是否在同一主对角线上”，根据前面在同一主对角线的条件 x-i=j-a[i]，即 x-j=i-a[i] 可知，同一主对角线上的格点的行号减列

号的值是相同的，图7.9(a)所示格点中的数字就是行号与列号之差，因为对于N皇后问题来说，该值的范围为1-N到N-1，可能取负数，因此我们可以用“行号-列号+N”来标识每一条主对角线。因此我们可以定义D[]，当判断第x行选择j列放置皇后是否产生主对角线冲突时，直接判断D[N+x-j]是否为true，true表示该对角线上已经被放置过皇后。

同理，同一条副对角线上的格点的行号加列号的值是一样的，定义E[]，判断(x,j)放置皇后是否产生副对角线冲突时，判断E[x+j]是否为true即可。

因此，在枚举j时，如果可以放置，则更新C[j],D[x-j+N],E[x+j]为true，进行下一行调用Queen(x+1)，接着恢复C[j],D[x-j+N],E[x+j]为false搜索其他方案。



(a)主对角线　　(b)副对角线

图7.9 4*4棋盘中的对角线示意图

图7.10以四个皇后为例给出部分搜索树示意图。

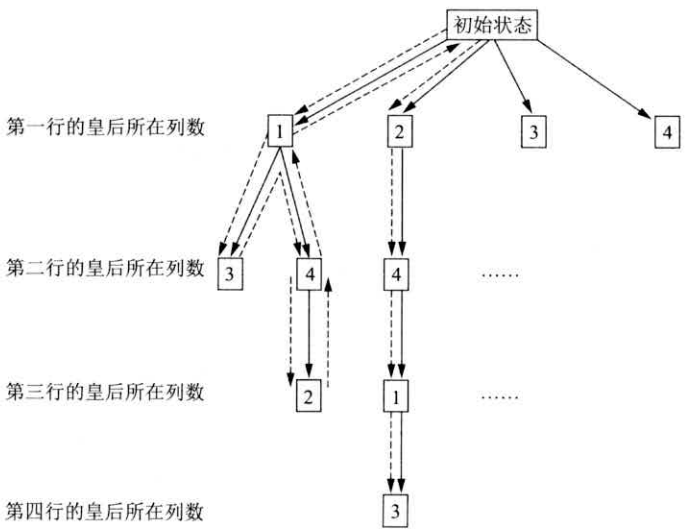


图7.10 四个皇后搜索树部分示意图

对应的程序eg7.13如下：

```
1  //eg7.13
2  #include <iostream>
3  using namespace std;
4  const int N=10;
5  bool C[N+1],D[2*N],E[2*N+1];
6  int n,Ans;
7  void Queen(int x)
8  {
9    if(x==n+1){++Ans;return;}
                  //到达递归出口，得到一个新的放置方案，方案加1
10   for(int j=1;j<=n;j++)  // 枚举列号 j 放置第 x 个皇后
11   {
12     if(!C[j]&&!D[x-j+n]&&!E[x+j])
                        //判断是否与之前的皇后有冲突
13     {
14       C[j]=D[x-j+n]=E[x+j]=true;       // 修改标记
15       Queen(x+1);          // 递归调用
16       C[j]=D[x-j+n]=E[x+j]=false;  // 恢复标记
17     }
18   }
19 }
20 int main()
21 {
22     cin>>n;
23     Queen(1);
24     cout << Ans << endl;
25     return 0;
26 }
```

练 习

（1）确定进制。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+2*1=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，则输出最小的一个。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，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。

输入格式：1行，包含三个整数p、q、r，相邻两个整数之间用单个空格隔开。

输出格式：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。

输入样例：

6 9 42

输出样例：

13

数据规模：

p、q、r的所有位都是数字，并且1 <= p、q、r <= 1 000 000

（2）最大连续子序列和。给定一个长度为n的序列，求该序列的最大的连续子序列和。

输入格式：第1行一个整数n；第2行n个整数，表示该序列。

输出格式：一个整数，表示该序列的最大连续子序列和。

输入样例：

10

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

输出样例：

8

数据规模：

1 ≤ n ≤ 100000，|序列每个元素| ≤ 1000

（3）最大子矩阵。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。

输入格式：输入是一个N * N的矩阵。输入的第1行给出N；再后面的若干行中，依次（首先从左到右给出第1行的N个整数，再从左到右给出第2行的N个整数……）给出矩阵中的N^2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]。

输出格式：输出最大子矩阵的大小。

输入样例：

4

0 -2 -7 0 9 2 -6 2

-4 1 -4 1 -1

8 0 -2

输出样例：

15

数据规模：

0 < N <= 300

（4）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。输出一个合法答案。

输入格式：1行，一个数n。

输出格式：1行，20个数，表示一个合法解。

第8章　数学在程序设计中的应用

计算机可以快速处理数据，从而解决一些运算量较大的数学问题；同时，数学的相关知识也可以帮助设计算法优化程序以减少计算机的运算量，数学与程序设计是密切相关的。这一章我们将学习一些简单的数学知识，并了解数学在程序设计中的应用。

8.1　常用数学函数

在一些题目中，我们会涉及一些数学的问题，需要求一些数学中的量。例如：求平方根和三角函数之类的，而这些量如果自己写程序来实现会比较麻烦，我们需要使用C++中的cmath库，cmath库中提供了许多可供我们直接使用的常用数学函数，只要在程序前加上#include<cmath>，我们就可以在程序中直接调用这些函数了。

【例8.1】计算器。我们需要模拟计算器实现对一个数开根或求其三角函数以及反三角函数的值的操作。读入n，表示有n个操作，接下来每一行有两个数：k,x，其中k表示操作类型，具体如下：

- 若k=1，则表示询问x的平方根。
- 若k=2，则表示询问度数为x的正弦值、余弦值以及正切值。
- 若k=3，则表示询问x的反正弦函数值和反余弦函数值。
- 若k=4，则表示询问x的反正切函数值。

所有结果均保留三位小数输出。

分析：直接调用cmath库中的函数解决问题。

程序eg8.1如下：

```
1  //eg8.1
2  #include <cmath>
3  #include<cstdio>
4  using namespace std;
5  int n,k;
6  double a,b,c,x;
```

```
const double pi=acos(-1);
// 调用反余弦函数定义 pi 常量的值
int main()
{
  scanf("%d",&n);
  for(int i=1;i<=n;i++)
  {
      scanf("%d%lf",&k,&x);
      if(k==1)
      {
        a=sqrt(x);
        // 调用开根函数 sqrt
        printf("%.3f\n",a);
      }
      if(k==2)
      {
        x=x/180*pi;                          // 把度数转换为弧度
        a=sin(x);b=cos(x);c=tan(x);          // 调用正弦余弦正切函数
        printf("%.3f %.3f %.3f\n",a,b,c);
      }
      if(k==3)
      {
        a=asin(x);b=acos(x);                 // 调用反正弦反余弦函数
        printf("%.3f %.3f\n",a,b);
      }
      if(k==4)
      {
        a=atan(x);                           // 调用反正切函数
        printf("%.3f\n",a);
      }
  }
  return 0;
}
```

运行结果：

```
输入：
4
1 9
2 30
3 -0.5
4 1
输出：
3.000
0.500 0.866 0.577
-0.524 2.094
0.785
```

cmath库中还有其他一些常用的函数，汇总如表8.1所示。

表8.1　cmath库中常用函数

序号	函数原型	函数功能
1	double sin(double x)	返回弧度x的正弦函数值

续表

序号	函数原型	函数功能
2	double cos(double x)	返回弧度 x 的余弦函数值
3	double tan(double x)	返回弧度 x 的正切函数值
4	double asin(double x)	返回 x 的反正弦函数值，x 的值在 [-1,1] 之间，返回的值是在 $[-\pi/2, \pi/2]$ 之间的弧度值
5	double acos(double x)	返回 x 的反余弦函数值，x 的值在 [-1,1] 之间，返回的值在 $[0, \pi]$ 之间
6	double atan(double x)	返回 x 的反正切函数值，返回的值在 $[-\pi/2, \pi/2]$ 之间
7	double atan2(double y, double x)	返回点 (x,y) 的极角，返回的值在 $[-\pi, \pi]$ 之间
8	double sinh(double x)	返回 x 的双曲正弦函数 $\frac{e^x - e^{-x}}{2}$ 的值
9	double cosh(double x)	返回 x 的双曲余弦函数 $\frac{e^x + e^{-x}}{2}$ 的值
10	double tanh(double x)	返回 x 的双曲正切函数 $\frac{e^x - e^{-x}}{e^x + e^{-x}}$ 的值
11	double exp(double x)	返回 x 的指数函数 e^x 的值
12	double log(double x)	返回 x 的自然对数 ln(x) 的值
13	double log10(double x)	返回 x 以 10 为底数的对数值
14	double pow(double x, double y)	返回 x 的 y 次方 x^y
15	double sqrt(double x)	返回 x 的根号值 $\sqrt{x}$
16	double ceil(double x)	返回不小于 x 的最小整数
17	double floor(double x)	返回不大于 x 的最大整数
18	int abs(int x)	返回整数 x 的绝对值 |x|
19	long labs(long x)	返回长整数 x 的绝对值 |x|
20	double fabs(double x)	返回实数 x 的绝对值 |x|

8.2 质因数的分解

质因数分解是数学领域中的一个常见问题，它被广泛运用在代数学、密码学、计算复杂性理论和量子计算机等领域。了解与掌握简单的质因数分解的方法，对于进一步学习与探究十分重要。

【例 8.2】质因数分解。对于正整数 N 的质因数分解，指的是将其写成以下形式：

$N=p_1*p_2*\ldots*p_m$

其中，$p_1,p_2,\ldots p_m$为不下降的质数。给定N，输出其质因数分解的形式。

输入格式：1个正整数N。

输出格式：N的质因数分解的形式$p_1*p_2*\ldots*p_m$，其中$p_1,p_2,\ldots,p_m$都是质数，且$p_1<=p_2<=\ldots<=p_m$。

输入样例：

60

输出样例：

2*2*3*5

分析：显然每个正整数的质因数分解的形式是唯一的。这样我们就可以设计一个简单的算法：从小到大枚举质数，如果该质数能整除N，则把该质数从N中分解出去，循环执行直到N不能被该质数整除为止，当N=1时停止枚举质数。样例中N=60时的分解过程如表8.2所示。

表8.2　N=60的分解过程

质数p	N	分解过程
p=2	15	2*2
p=3	5	2*2*3
p=5	1	2*2*3*5

该方法对应的程序eg8.2_1如下：

```
1  //eg8.2_1
2  #include<cstdio>
3  using namespace std;
4  int n;
5  int tot;                    // 表示有 tot 个质因子
6  bool IsPrime(int m)         // 测试 m 是否为一个质数
7  {
8      for(int i = 2;i*i <=m;i ++)
9          if(m % i == 0)
10             return 0;
11     return 1;
12 }
13 int main()
14 {
15     scanf("%d",&n);
```

```
16      for(int i = 2;i <= n;i ++)
17           if(IsPrime(i) && n % i == 0)
18           {
19              while(n % i==0)
20              {
21                   tot++;
22                   if(tot==1)printf("%d",i);else printf("*%d",i);
23                   n/=i;
24              }
25           }
26      return 0;
27 }
```

这个算法跑得非常慢，无法满足N较大时的需求。我们尝试去优化上面的代码。观察这一句话：

```
for(int i = 2;i <= n;i ++)
    if (IsPrime(i) && n % i == 0)
```

事实上我们没有必要去判断i是否为一个质数，因为假如n%i==0，那么i必然是一个质数！证明也十分简单，假设i不是质数，i必存在一个质数因子j，那么j<i，说明我们之前就已经枚举过了j，n已经把质数j全部分解出去了，当前的n没有了j因子，n%j不为0，与n%i==0矛盾！所以当n%i==0时,i必然为一个质数。

观察上面那句话，可以发现，i循环实际是在不停地寻找n的最小质因子。

如果n只剩一个质因子，那么n必定是质数。

如果n中不止一个质因子，设n=i*j，i<=j，i*i<=i*j=n。这就说明当n是合数时，n的最小质因子i一定满足i*i<=n，否则n就是质数。

因此，i循环的结束条件可以由i<=n改成i*i<=n，循环结束后再特判n的值，n如果等于1，则表示分解结束，n如果不等于1，则说明n还没有分解完，剩下n这个质数，直接输出即可。

优化的程序eg8.2_2如下：

```
1  //eg8.2_2
2  #include <iostream>
3  #include<cstdio>
4  using namespace std;
5  int n;
```

```
int tot;     //表示有tot个质因子
int main()
{
      scanf("%d",&n);
      for(int i = 2;i * i <= n;i ++)
           if(n % i == 0)
           {
             while (n % i==0)
             {
               tot++;
               if(tot==1)printf("%d",i);else printf("*%d",i);
               n/=i;
             }
           }
       if(n!=1)
       {
           tot++;
           if(tot==1) printf("%d",n);else printf("*%d",n);
       }
       return 0;
}
```

这个算法效率大大提升，已经可以满足大部分需求了。

8.3　最大公约数的欧几里德算法

两个数的最大公因数，也称最大公约数、最大公因子，是指两个整数共有约数中最大的一个。在本节中，我们将介绍求解两个数的最大公约数的多种算法。

最简单是枚举法：对两个整数a和b，共同的约数在1到min(a,b)之间，从大到小枚举，若判断它能同时整除a和b两个数，即为两个数的最大公约数。

【例8.3】求出正整数a和b的最大公约数。

```
//eg8.3_1
#include<iostream>
#include<algorithm>
using namespace std;
nt a,b;
```

输入：
24 30
运行结果：
6

```
int main()
{
  cin >> a>> b;
  for(int i=min(a,b); i>0; i--)
  {
        if(a%i==0 && b%i==0)
        {
          cout <<i;
          break;
        }
  }
  return 0;
}
```

说明：在C++中，a % i表示a除i的余数，a%i==0就是a能被i整除，即a是i的倍数或i是a的约数。“a%i==0 && b%i==0”成立时i为a和b的公约数。由于我们是从大到小枚举，所以遇到的第一个公约数就是最大的公约数，直接输出break就可以了。

这个算法的最坏时间复杂度为O(min(a,b))。

从数学上讲，使用上一节的分解质因数的方法，可以令我们从另一方面加深对公约数的理解。例如：a=24,b=60时，对a和b分解质因数得到：

a=2*2*2*3;

b=2*2*3*5;

收集共有的质因数是2、2、3，得到最大公约数为：2*2*3=12。

参考程序如下：

```
//eg8.3_2
#include<iostream>
#include<algorithm>
using namespace std;
int a,b,gcd;
int main()
{
  cin >> a>> b;
  gcd=1;
  for(int i=2; i*i<=min(a,b); i++)
  {
    while(a%i==0 && b%i==0)          //收集公共质因数
    {
```

```
14          a/=i;
15          b/=i;
16          gcd*=i;
17      }
18      while (a%i==0)          //a 去掉 a 有 b 没有的质因数
19          a/=i;
20      while (b%i==0)          //b 去掉 b 有 a 没有的质因数
21          b/=i;
22  }
23  if(a%b==0)                  // 剩余的 a 和 b 必有一个是质数或 1
24    gcd*=b;
25  else
26    if(b%a==0)
27      gcd*=a;
28  cout << gcd;
29  return 0;
30 }
```

说明： 由于有 i*i<=min(a,b) 的优化，时间复杂度降为 O(sqrt(min(a,b)))。循环完后，a 和 b 有一个可能没有分解完，要注意收集到 gcd 中，这个算法也可以看成是短除法的变形。

我们在例 8.3 中采取了低效的算法——枚举法。下面让我们尝试用数学知识来找到一种更好的算法。

假设 c 是 a 和 b 的最大公约数（假设 a>b），那么有 a%c=0 且 b%c=0，则 (a-b)%c=0 亦成立，即若 c 是 a 和 b 的公约数，c 亦是 b 和 a-b 的公约数；反之，即若 c 是 b 和 a-b 的公约数，c 亦是 a 和 b 的公约数。两个公约数集合相等，两个集合中的最大数也必然相等，简单讲就是：gcd(a,b) = gcd(b,a-b)。

因此我们可以采用函数进行迭代，当迭代到 a==b 时，a 为 a 和 b 的最大公约数，即答案。

【例 8.4】 用《九章算术》中的“更相减损法”求正整数 a 和 b 的最大公约数。

```
1  //eg8.4
2  #include<iostream>
3  #include <algorithm>
4  using namespace std;
```

输入：
24 30
运行结果：
6

```
int a,b;
int gcd(int a, int b)
{
  if(a==b) return a;
  if(a<b) swap(a,b);     //保证a>=b
  return  gcd(b,a-b);
}
int main()
{
  cin >> a>> b;
  cout << gcd(a,b);
  return 0;
}
```

这个算法很简洁！但可惜的是效率依然可能较低，比如说a很大，b很小时。那么我们是否再加以优化呢？

可以用例8.4类似的方法证明：gcd(a,b) = gcd(b,a%b) (a>b)。

这样的话，我们就可以设计出一个更为高效的算法——欧几里德算法。

【例8.5】使用欧几里德算法求正整数a和b的最大公约数。

```
//eg8.5
#include<iostream>
#include<algorithm>
using namespace std;
int a,b;
int gcd(int a, int b)
{
  if(b==0) return a;
  return  gcd(b,a%b);
}
int main()
{
  cin >> a>> b;
  cout << gcd(a,b);
  return 0;
}
```

欧几里德算法的时间复杂度是多少呢？

a>b时：

（1）若b<=a/2，gcd(a,b)变为gcd(b,a%b)，显然规模至少缩小一半。

（2）若b>a/2，a%b也最少缩小为a的一半。

总之经过一次迭代，gcd(a,b)的数据规模至少会变为原来的一半，所以这个算法的时间复杂度是O(logN)。

8.4　加法原理与乘法原理

在OI中经常会遇到需要我们计算事件方案数的问题，那么这个时候根据不同的具体情况，需要运用到加法原理和乘法原理计数。

加法原理：做一件事，完成它可以有N类办法，在第一类办法中有M_1种不同的方法，在第二类办法中有M_2种不同的方法，……，在第n类办法中有M_n种不同的方法，那么完成这件事共有$M_1+M_2+...+M_n$种不同方法，每一种方法都能够达成目标。

比如：小高一家人外出旅游，可以乘火车，也可以乘汽车，还可以坐飞机。经过网上查询，出发的那一天中火车有4班，汽车有3班，飞机有2班。根据加法原理，任意选择其中一个班次的方法为：4+3+2 = 9(种)。

【例8.6】方案数。有一个类似弹球的游戏，如图8.1所示，小球从上向下滑落，每次到一个“交叉”点都有2种选择：向左或向右滑落。如果有一个球从顶端滑落到底会有多少种方法？特别是中间的“交叉”点有一些是“陷阱”，小球滑到“陷阱”就不能再继续下滑了。

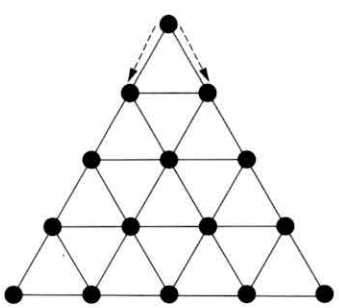


图8.1

输入格式：第1行输入两个正整数N、M（1<N、M<20），N表示三角形的层数，M表示“陷阱”的个数；第2到M+1行，每行2个整数x、y，

表示第x行的第y个“交叉”点是“陷阱”。

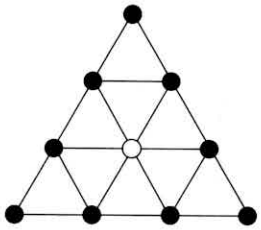


图8.2 样例数据图示

输出格式：小球从上到下的方案数。

输入样例：

4 1

3 2

输出样例：

4

分析：一般地，设顶点到三角形中间的点(x,y)的方案数为f(x,y)，可以分为2类：

(1)从(x-1,y-1)滑落来的。

(2)从(x-1,y)滑落来的。

根据加法原理：

$$f(x,y) = f(x-1,y-1)+f(x-1,y)$$

注意边界和陷阱的判断处理就可以求出所有交叉点的f(x,y)。再根据加法原理得到：

$$方案数=f(n,1)+f(n,2)+\cdots+f(n,n)$$

程序如下：

```
1  //eg8.6
2  #include<iostream>
3  #include <algorithm>
4  using namespace std;
5  const int maxN=22;
6  int f[maxN][maxN];
7  bool tr[maxN][maxN];     //陷阱标记
8  int N,M, ans;
9  int main()
10 {
```

```
11    cin >> N>> M;
12    for(int i=0;i<M; i++)
13    {
14      int x,y;
15      cin >>x >>y;
16      tr[x][y] = true;
17    }
18    if(tr[1][1])                    //处理顶点初始化
19      f[1][1]=0;
20    else f[1][1]=1;
21    for(int x=2; x<=N; x++)
22      for(int y=1; y<=x; y++)
23      {
24          if(!tr[x-1][y-1]) f[x][y]+= f[x-1][y-1];
25          if(!tr[x-1][y]) f[x][y]+= f[x-1][y];
26      }
27    ans=0;
28    for (int i=1; i<=N; i++)
29      ans+= f[N][i];
30    cout<<ans<<endl;
31    return 0;
32 }
```

乘法原理：做一件事，完成它需要分成n个步骤，做第一步有M_1种不同的方法，做第二步有M_2种不同的方法，……，做第n步有M_n种不同的方法，那么完成这件事共有$M_1*M_2*...*M_n$种不同的方法。

例如：从甲地到乙地有2条路，从乙地到丙地有3条路，从丙地到丁地也有2条路。问：从甲地经乙、丙两地到丁地，共有多少种不同的走法？

图8.3

显然根据乘法原理，共有2*3*2=12种走法。

【例8.7】单词数。Jet为了编写打字练习软件，要设计一个随机生成单词算法，生成单词的规则是，先给出一个小写字母组成的“限制单词”，

然后单词上的每个字母可以改变（仍然是小写字母）但不能变“大”。比如，“限制单词”是cb，可以生成的单词有aa,ab,ba,bb,ca,cb。

现在输入限制单词，问可能产生的单词数。

输入格式：第1行输入一个小写字母组成的单词，单词长度<100。

输出格式：可能产生的不同单词数，由于答案可能很大，输出答案模10007的结果。

输入样例：

cb

输出样例：

6

分析：由于每个字母的变化是独立的，每确定一个字母可看成完成一步，显然可以使用乘法原理计算。参考程序如下：

```
1  //eg8.7
2  #include<iostream>
3  #include <string>
4  using namespace std;
5  const int mod=10007;
6  string s;
7  int ans;
8  int main()
9  {
10   cin >> s;
11   ans=1;
12   for(int i=0; i<s.size(); i++)
13     ans = (ans*int( s[i]-'a'+1)) % mod;
14   cout << ans<<endl;
15   return 0;
16 }
```

8.5 排列与组合

排列组合是组合数学最基本的概念。所谓排列，就是指从确定的一些元素中取出指定个数的元素，且考虑不同顺序。组合则是指从确定的一些元素中仅仅取出指定个数的元素，不考虑顺序。排列组合的中心问题是研究给定要求的排列和组合可能出现的情况总数。

例如，从集合{a,b,c}中取2个元素的排列有ab,ac,ba,bc,ca,cb。取2个的组合有：{a,b},{a,c},{b,c}。

1. 排列的定义及公式

从N个不同元素中，任取M（M与N均为自然数，下同）个元素按照一定的顺序排成一列，叫做从N个不同元素中取出M个元素的一个排列；从N个不同元素中取出M个元素的所有排列的个数，叫做从N个不同元素中取出M个元素的排列数，用符号表示为A_N^M。

我们考虑当前排列，第1个位置我们有N种选择，第2个位置除去第1个位置选择的元素我们还有N-1种选择，同理第3个位置我们还有N-2种选择，以此类推，直到第M个位置我们还有N-M+1种选择，那么根据乘法原理，我们不难得到排列数公式

$$A_N^M = N \times (N-1) \times ... \times (N-M+1) = \frac{N!}{(N-M)!}$$

N!表示1*2*..*N，即N的阶乘。

2. 组合的定义及公式

从N个不同元素中，任取M个元素并成一组，叫做N从个不同元素中取出M个元素的一个组合；从N个不同元素中取出M个元素的所有组合的个数，叫做从N个不同元素中取出M个元素的组合数，用符号C_N^M表示。

对于所有的排列（A_N^M个），每一排列都有M!个排列和它的元素是一样的，即是同一个组合。比如，排列abc,acb, bac, bca, cab, cba在组合意义上是相同的。因此有：

$$C_N^M = A_N^M / M!$$

即我们可以得到组合数公式

$$C_N^M = \frac{N!}{M! * (N-M)}$$

另外，考虑N个元素中选出M个的一个组合，我们可以发现剩下没有选出的有N-M个，可以视为N个元素中选出N-M个的一个组合，而这种组合关系是一一对应的，也就是我们可以得到一个公式$C_N^M = C_N^{N-M}$。

此外，我们可以把从N个数中选M个的组合分成两类：

（1）不含第N个数的组合，方案数为C_{N-1}^M。

（2）含第N个数的组合，方案数为C_{N-1}^{M-1}。

根据加法原理，可得公式：

$$C_N^M = C_{N-1}^M + C_{N-1}^{M-1}$$

也就是说，组合数可以使用加法推算出来，这个就是著名的杨辉三角的原理。

【例8.8】求组合数。有N个任务，每个任务是求一个组合数。

输入格式：第1行输入一个正整数N<1000000；第2到第N+1行，每行2个正整数x和y，表示要计算组合数C_x^y，保证x>=y，x<1000。

输出格式：N行，每行一个组合数C_x^y，由于答案可能很大，输出答案模10007的结果。

输入样例：

```
3
6 3
10 7
20 8
```

输出样例：

```
20
120
5886
```

分析：如果每一个组合都按定义计算，不仅速度太慢，而且由于有除法运算，模10007的运算也不好处理。使用杨辉三角的方法预处理，运算只有加法，不仅速度快，而且也可以方便处理模10007运算。

```
1  //eg8.8
2  #include<iostream>
3  using namespace std;
4  const int mod=10007;
5  int C[1002][1002];
6  int N,x,y;
7  int main()
8  {
9    C[1][1]=1;
10   for(int i=2; i<=1000; i++)
11     for(int j=1; j<=i; j++)
12         C[i][j]=(C[i-1][j-1]+C[i-1][j]) % mod;
13   cin >> N;
14   for(int i=0; i<N; i++)
```

```
15    {
16      cin >> x>>y;
17      cout <<C[x+1][y+1]<<endl;
18    }
19    return 0;
20 }
```

时间复杂度为$O(10^6)$，可满足速度要求。

8.6　圆排列、可重集排列

本节研究排列的两种变形：圆排列和可重集排列。

排列是把几个数线性地排成一列，如果排成一个圆周情况会如何呢？例如排列123456放到圆上为图8.4；排列234561放到圆上为图8.5；…排列6123456放到圆上为图8.6。

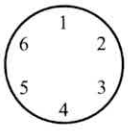


图8.4

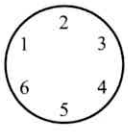


图8.5

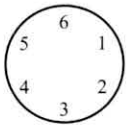


图8.6

显然，从可以转动的角度看，上面3个图是一样的圆环，圆排列认为它们都是一样的。

1. 圆排列的定义及公式

从N个不同元素中不重复地取出M个元素排列在一个圆周上，叫做N个不同元素的M-圆排列。如果一个M-圆排列通过旋转可以得到另一个M-圆排列，则认为这两个圆排列相同。

从前面的分析中知道，对于取6个元素的线性排列，有6个不同排列得到的圆排列都是相同的；以此类推，对于取M个元素的线性排列，有M个不同排列得到的圆排列都是相同的。因此从N个中取M个的圆排列数为：A_N^M/M。

另一方面，对于一个M-圆排列，我们可以切割开让它变成线性排列，有M个切的位置，因此一个M-圆排列可以对应M个不同的线性排列，结论和上面一样。所以N个不同元素的M-圆排列总数为：

$$\frac{N!}{(N-M)!*M}$$

【例8.9】求最大路径。把数1到N排放成一周，每2个相邻的数的差的平方为2个数之间的距离。问怎样排放可以使一圈的距离最大，输出这个最大值。

输入格式：第1行输入一个正整数N,N<12。

输出格式：一个整数，最大周长的值。

输入样例：

6

输出样例：

66

解释：圆排列1 5 3 4 2 6的周长为4*4+2*2+1*1+2*2+4*4+5*5=66。

分析：如果直接dfs枚举所有排列，有N!方法，最多为11!= 39916800，加上递归系数比较大，会超时。根据圆排列的原理，其实最多只有10!(=3628800)种不同的圆排列，速度不超过1秒。

具体的，我们可以认为数字1一定在第一位，只要排列后面N-1个数就可以了。参考程序如下：

```
1  //eg8.9
2  #include<iostream>
3  using namespace std;
4  bool a[15];
5  int  b[15];
6  int N,ans;
7  void dfs(int n, int len);
8  int main()
9  {
10   cin >> N;
11   ans=0;
12   a[1]=true;
13   b[1]=1;
14   dfs(1,0);
15   cout << ans<<endl;
16   return 0;
17 }
18   /*
19   使用 dfs 枚举所有可能的圆排列，找到最大值
20   n 是已经排列有几个数，
21   len 是圆圈当前的长度
```

```
22   */
23   void dfs(int n, int len)
24   {
25     if(n==N)
26     {
27       if(len+(b[n]-b[1])*(b[n]-b[1]) > ans )
28         ans= len+(b[n]-b[1])*(b[n]-b[1]);
                                  //要加上环形的最后一个和第一个的距离
29       return;
30     }
31     for(int i=2; i<=N; i++)
32         if(!a[i])
33         {
34             a[i]=true;
35             b[n+1]=i;
36             dfs(n+1,len+(b[n+1]-b[n])*(b[n+1]-b[n]));
37             a[i]=false;
38         }
39 }
```

可以看到，特别地对于圆排列，当N=M时，方案数为(N-1)!。

在全排列问题中，如果有些元素相同会怎样呢？比如集合{a,a,b}的全排列不是3!=6个，而是3个：aab,aba,baa。这是因为集合中的2个a，理论上排列有2!个，但实际上只有一个，即aa，因此集合{a,a,b}的全排列=3!/2!。

以此类推，{a,a,a,b}的全排列数为4!/3!=4。{a,a,b,b}的全排列数为4!/2!/2!=6。

2. 可重集排列的定义及公式

设可重集合$S=\{B_1*A_1,B_2*A_2,\ldots,B_m*A_m\}$，表示集合S中共有N个m种不同元素，含有$B_1$个元素$A_1$，$B_2$个元素$A_2$，…，$B_m$个元素$A_m$，$B_i$被称为元素$A_i$的重数，m被称为元素的重数。在S中任选r个元素组成的排列，称为S的r排列，当r=N时，一个相同的排列会被重复计算$B_1!*B_2!*\ldots B_m!$，所以此时排列总数为

$$\frac{N!}{B_1!*B_2!*\ldots*B_m!}$$

【例8.10】某车站有N个入口，入口每次只能进一人，M个人进站的

方案有多少种？不同入口、不同顺序进站视为不同方案。入口之间是独立的，不考虑不同入口之间的顺序。

分析： 假设 N=2，M=5 时，我们用 {A、B、C、D、E} 表示 5 个人，进站方案可表示为：第 1 个入口的方案 + 第 2 个入口的方案。比如：AC+DBE 就是一种排队方案。把“+”也看成一个元素，所有的方案数就是它们的排列数 =6!

再比如N=6,M=9时，一种方案为：A+BC+DE+FG+H+I，其中“+”表示门框，“+”将字符串分成了6段，其中每段的字母即表示在某门进站的人。所以总方案数就是一个含5个+，A~I各1个的多重集合的全排列数，即 14!/5!

所以，本题的总方案数是N-1个+和M个不同人的多重集合的全排列数，即 (N+M-1)!/(N-1)!

```
1  //eg.8.10
2  #include<cstdio>
3  using namespace std;
4  int n,m,ans,i;
5  int main()
6  {
7      scanf("%d%d".&n.&m);
8      ans=1;
9      for(i=n;i<=m+n-1;i++)ans*=i; //排列数即为 (n+m-1)!/(n-1)!
10     printf("%d\n",ans);
11     return 0;
12 }
```

练 习

（1）给定两个正整数n,m,求它们的最小公倍数lcm(n,m)。

输入格式：两个正整数n,m。

输出格式：n,m的最小公倍数。

输入样例

12 8

输出样例

24

（2）设计一个不用到mod的欧几里德算法，并且复杂度依然是logN的，尽量使用位运算加速程序。

输入格式：两个自然数n，m。

输出格式：一个自然数，表示gcd(n,m)。

输入样例：

24 16

输出样例：

8

提示：使用位运算。分为奇数偶数几种情况考虑。

（3）给定自然数n,m，求出gcd(n,m)。

输入格式：两个自然数n,m。

输出格式：一个自然数，表示gcd(n,m)。

输入样例：

24 16

输出样例：

8

提示：使用二进制欧几里德算法。

（4）给定T个正整数，要求对于每个数，都输出其质因数分解。

输入格式：第1行一个正整数T；第2行T个正整数，表示每个数。

输出格式：T行，第i行对应第i的数的质因数分解。输出形式：

p1^c1*P2^c2…*Pm^cm

其中，$p_1,p_2,\ldots p_m$为递增的互不相同的质数，ci为1是省略。

输入样例：

2

24 25

输出样例：

2^3*3

5^2

提示：维护1~T每个数最小的约数，可以发现它必然为质数。

*第9章 STL（标准模板库）简要说明

在国内的OI历史上曾经因为STL库的功能强大而被禁止使用，以维护对Pascal选手公平。后来随着C++的普及和国际上OI、ACM等比赛中选手越来越多使用STL，中国OI终于开放了STL的使用权。

STL（Standard Template Library）标准模板库，是一系列软件的统称。从根本上说，STL是一些“容器”的集合，这些“容器”有list，vector，set，map等，STL也是算法和其他一些组件的集合。前面已经学习过的<algorithm>中sort函数、<string>中string类都是STL的内容，下面我们学习更多的新内容。

STL中的概念和名称与之前Pascal、C等传统的数据类型、数据结构有很大不同，功能也比较多，本章就OI中经常使用的STL内容作一些简单介绍，完整的内容请参阅相关资料。

9.1 STL中的一些新概念

9.1.1 容器（containers）

容器可以看成是数组的拓展，STL出现了向量（vector）、栈（stack）、队列（queue）、优先队列（priority_queue）、链表（list）、集合（set）、映射（map）等容器。在教材中曾经对这些数据结构作过讲解，每种数据结构都有自己的实现细节。STL库把一切细节都隐藏起来，使用统一的接口（命令）格式，供程序员简单方便使用。

例如对于数据结构栈（stack），我们关心的是进栈、出栈、取栈顶元素等操作，实现的细节我们并不关注。如表9.1所示，STL为栈提供了相关成员函数。

表9.1

函数名	功 能
push(元素)	进栈，把一个元素压入栈顶

续表

函数名	功 能
pop( )	出栈，把栈顶元素弹出
top( )	栈顶元素，返回栈顶元素
size( )	元素个数，栈的大小——即栈中已有元素的个数
empty( )	判断栈空，等价于size为0

【例9.1】括号匹配。输入一个由(、)、[、]4种符号构成的字符串，判断其中的括号是否匹配，是，就输出“yes”，否则输出“no”。

参考程序如下：

```
1  //eg9.1
2  #include <iostream>
3  #include<string>
4  #include <stack>     //包含容器 stack
5  using namespace std;
6  string s;
7  bool check(string s)
8  {
9    stack < char > p; //定义一个栈变量 p，栈的元素是 char 类型
10   p.push('#');      //加一个前面的“哨兵”，避免空栈的判断
11   for(int i=0; i<s.size(); i++)
12   {
13     char c=s[i];                //当前的字符
14     if(c==')')                  //如果是右圆括号
15     {
16         if(p.top()!='(')        //判断栈顶是否为左圆括号
17             return false;       //不是，匹配失败
18         else p.pop();           //是，弹出栈顶的这个左圆括号
19     }
20     else if(c==']')             //如果当前字符是右方括号
21     {
22         if(p.top()!='[')        //判断栈顶是否为左方括号
23           return false;         //不是，匹配失败
24         else p.pop();           //是，弹出栈顶的这个左方括号
25     }else                       //是其他字符，即左括号
26     p.push(c);                  //把左括号压进栈
27   }
```

```
28    return (p.size()==1);
                          // 判断栈中是否只有哨兵元素，没有多余左括号
29 }
30 int main()
31 {
32    cin >> s;
33    if(check(s) )
34        cout <<"yes"<<endl;
35    else
36        cout <<"no"<<endl;
37    return 0;
38 }
```

说明：

（1）第4句头文件要包括<stack>。

（2）第9句定义容器栈(stack)变量时，要指明元素的类型，可以是程序员自定义类型。

9.1.2 算法（algorithms）

对一些编程中常用的算法，STL 提供了通用的函数供程序员直接调用。比如：遍历(for_each)、查找（find）、二分查找（binary_search、lower_bound、upper_bound）、去除重复（unique）、填充（fill）、前一个排列（pre_permutation）、下一个排列（next_permutation）、排序（sort）等。

很多时候恰当使用STL的算法库，可以使编程简单方便，更能保证编程的正确性。

【例9.2】火星人（来源：NOIP2004普及组）。人类终于登上了火星的土地并且见到了神秘的火星人。人类和火星人都无法理解对方的语言，但是我们的科学家发明了一种用数字交流的方法。这种交流方法是这样的，首先，火星人把一个非常大的数字告诉人类科学家，科学家破解这个数字的含义后，再把一个很小的数字加到这个大数上面，把结果告诉火星人，作为人类的回答。

火星人用一种非常简单的方式来表示数字——掰手指。火星人只有一只手，但这只手上有成千上万的手指，这些手指排成一列，分别编号为1，2，3……。火星人的任意两根手指都能随意交换位置，他们就是通过这方

法计数的。

一个火星人用一个人类的手演示了如何用手指计数。如果把五根手指——拇指、食指、中指、无名指和小指分别编号为1，2，3，4和5，当它们按正常顺序排列时，形成了5位数12345，当你交换无名指和小指的位置时，会形成5位数12354，当你把五个手指的顺序完全颠倒时，会形成54321，在所有能够形成的120个5位数中，12345最小，它表示1；12354第二小，它表示2；54321最大，它表示120。表9.2展示了只有3根手指时能够形成的6个3位数和它们代表的数字。

表9.2

三进制数	123	132	213	231	312	321
代表的数字	1	2	3	4	5	6

现在你有幸成为第一个和火星人交流的地球人。一个火星人会让你看他的手指，科学家会告诉你要加上去的很小的数。你的任务是，把火星人用手指表示的数与科学家告诉你的数相加，并根据相加的结果改变火星人手指的排列顺序。输入数据保证这个结果不会超出火星人手指能表示的范围。

输入格式：包括3行，第1行有一个正整数N，表示火星人手指的数目(1<=N<=10000)；第2行是一个正整数M，表示要加上去的小整数(1<=M<=100)；第3行是1到N这N个整数的一个排列，用空格隔开，表示火星人手指的排列顺序。

输出格式：只有1行，该行含有N个整数，表示改变后的火星人手指的排列顺序。每两个相邻的数中间用一个空格分开，不能有多余的空格。

输入样例

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

输出样例

```
1 2 4 5 3
```

分析：这个题目就是求M次一个排列的下一个排列是哪个？虽然可以通过用数学知识分析找到规律来编程，但使用STL的next_permutation函数简单方便，且正确性完全得到保证。

参考程序如下：

```
1  //eg9.2
2  #include <iostream>
3  #include<algorithm>
4  using namespace std;
5  int N,M, a[10010];
6  int main()
7  {
8    cin >> N>>M;
9    for(int i=0; i<N; i++)
10       cin >> a[i];
11   for(int i=0; i<M; i++)
12       next_permutation(a,a+N);
                              //求数组a[0]到a[N-1]的下一个排列
13   for(int i=0; i<N-1; i++)
14       cout<<a[i]<<" ";
15   cout<<a[N-1]<<endl;
16   return 0;
17 }
```

说明：

（1）第3句包含头文件<algorithm>。

（2）第12句调用算法库中的下一个排列函数。

9.1.3 模板(template)

我们以前编写的函数中，参数的类型都是确定的，这个特点极大限制了函数的通用性。比如C语言中求一个数的绝对值函数，整数是abs，长整数是labs，浮点数是fabs。STL中使用模板技术，统一为abs函数。再比如求2个元素的最大值，对于整数和字符串需要编写两个函数：

```
int maxA(int a, int b)
{
    if(a>b) return a;
    else return b;
}
string maxA( string a,string b)
{
    if(a>b) return a;
    else return b;
}
```

如果需要其他类型使用，也要分别写maxA函数，特别是对于我们自己定义的类型。但使用模板技术，我们可以只写一个函数：

```
template <class T>        //template引入通用的类型T
T maxA(T a, T b)          // 使用通用类型T
{
     if(a>b) return a;
     else return b;
}
```

这个函数可以对任何类型通用，即使是我们自己定义的类型（只要定义了运算符<号）。

STL中广泛使用了模板技术，因此其中的函数都具有通用性，比如排序sort函数，可以应用于任何类型。

9.1.4 迭代器(iterator)

由于容器的种类繁多，从统一性和效率考虑，STL使用了迭代器来表示数据位置用于存取数据。迭代器也可以简单地认为是容器的“专用指针”，使用时请参考本教材的有关指针的章节。

不过，为了编程简单方便，STL对vector、map等也提供了“[ ]”操作（即下标运算）。

关于迭代器的具体使用方法，后面将结合具体的容器举例说明。

9.2 几个常见的容器介绍

9.2.1 队列(queue)

数据结构中队列具有先进先出（FIFO）的特点，队尾进、队头出。STL为queue提供了一些成员函数（表9.3）。

表9.3

函数名	功 能
push（元素）	进队，把一个元素压入队尾
pop()	出队，把队头元素弹出

续表

函数名	功 能
front( )	队头元素，返回队头元素
back( )	队尾元素，返回队尾元素
size( )	元素个数，队列的大小——即队中已有元素的个数
empty( )	判断队空，等价于size为0

可以看到，queue的函数名和功能与前面stack的基本一致。STL的容器这方面做的很好，让我们容易记住和使用这些成员函数。

【例9.3】取扑克牌。有N(<100)张扑克牌放成一堆，每次从上面取一张牌翻开，再从上面取一张牌放到这堆牌的下面。即从上面奇数次取到的牌翻开放成一排，偶数次取到的牌放到下面，直到取完。

输入N张扑克牌的牌面数字，输出翻开牌的情况。例如：N=4，牌从上到下为1 2 3 4。翻开牌的情况为：1 3 2 4。

分析：使用一个队列表示这堆牌，模拟取牌过程。

参考程序如下：

```
1  //eg9.3
2  #include <iostream>
3  #include <queue>                          // 包含容器 queue
4  using namespace std;
5  queue < int > pai;                        // 定义队列变量 pai，队列
                                                里的元素是 int 型的
6  int N,x;
7  int main()
8  {
9    cin >> N;
10   for(int i=0; i<N; i++)
11   {
12       cin >> x;
13       pai.push(x);                        // 把牌放入队列中
14   }
15   for (int i=1; !pai.empty(); i++)
16   {
17     if(i%2==1)                            // 奇数张牌
18       cout <<pai.front()<<" ";            // 翻开上面牌
```

```
19      else
20        {
21            x=pai.front();                    // 取第偶数张牌
22            pai.push(x);                      // 放到后面
23        }
24      pai.pop();                              // 删除上面牌
25    }
26    return 0;
27 }
```

说明：第5句是队列（queue）的变量定义，容器的变量定义要在“<>”中说明元素的类型。

9.2.2 向量(vector)

向量(vector)可以看成是“动态数组”，即可以保存的元素个数是可变的，是OI选手经常使用的一种容器。表9.4是STL为vector提供的一些常见成员函数。

表9.4

函数名	功 能
push_back(元素)	增加一个元素到向量的后面
pop_back( )	弹出（删除）向量的最后一个元素
insert(位置,元素)	插入元素到向量的指定位置
erase(位置)	删除向量指定位置的元素
clear( )	清除向量所有元素，size变为0
运算符[ ]	取向量的第几个元素，类似数组的下标运算
front( )	取向量的第一个元素
back( )	取向量的最后一个元素
begin( )	向量的第一个元素的位置，返回第一个元素迭代器（指针）
end( )	向量的结束位置。注意：返回的迭代器是最后一个元素的后面位置，不是最后一个元素的迭代器
size( )	元素个数，向量的大小——即向量中已有元素的个数
resize(大小)	重新设定向量的大小，即可以保存多少个元素
empty( )	判断向量是否空，等价于size为0

对比数组，使用向量时要注意：向量的大小是可变的。开始时向量为空，随着不断插入元素，向量自动申请空间，容量变大。

向量添加元素的常见方法是使用push_back函数。insert是可以插入元素到向量中间的函数，位置的确定要使用迭代器，比如把数3插入向量V的第5个位置的命令为：

```
V.insert(V.begin()+5,3);
```

删除命令erase也类似。

另外向量有两种方式访问元素：迭代器和下标，具体用法见下面例子。

【例9.4】马鞍数。在一个N*N的矩形方格里，其中的M格子里有数字。如果某个格子有数字，并且是这一行和这一列的最小数，就称为马鞍数。

现在有Q个询问，每次问第x行y列的格子是否为马鞍数。如果是马鞍数，则输出所在列的所有数字。如果不是马鞍数，则输出“no”。

数据范围：

N：[100, 10000];

M：[100, 1000000];

Q：[0, 1000];

所有数字范围：[-10^8, 10^8]。

分析：如果使用N*N的二维数组，显然空间复杂度为O(4*10^8)，太大。但M比较小，我们可以使用向量分别记录每行和每列的数，空间复杂度为O(2*4*10^6)，比较小。时间复杂度为O(Q*N)<=10^7。

参考程序如下：

```
1  //eg9.4
2  #include <iostream>
3  #include<algorithm>
4  #include <vector>          //包含容器vector
5  using namespace std;
6  struct Tnode
7  {
8    int x,y;                 //一个格子的行和列坐标
9    int v;                   //格子中的数
10 };
11 bool cmp(Tnode A, Tnode B)
12 {
13   if(A.x==B.x) return A.y<B.y;
14   return A.x < B.x;
15 }
```

```
16 Tnode a[1000006];
17 vector  < int > X[1003],Y[1003];
                              //定义向量数组变量X和Y，记录每行和每
                                列的数，向量的元素是int型的
18 int minX[1003], minY[1003];  // 每行的最小数
19 int N,M,Q;
20 int main()
21 {
22   cin >> N >>M >>Q;
23   for(int i=0; i<M; i++)
24       cin >> a[i].x >> a[i].y >> a[i].v;
25       sort(a,a+M, cmp);                  // 调用 sort 按坐标排序
26   for(int i=0; i<=N; i++)
27       minX[i]=minY[i]=100000001;         // 最小值哨兵
28   for(int i=0; i<M; i++)
29   {
30       int xx=a[i].x;
31       minX[xx]=min(minX[xx],a[i].v);
32       X[xx].push_back(i);                // 收集数到相应行向量后面
33       int yy=a[i].y;
34       minY[yy]=min(minX[yy],a[i].v);
35       Y[yy].push_back(i);                // 收集数到相应列向量后面
36   }
37   for(int i=0; i<Q; i++)
38   {
39       int xx,yy;
40       cin >>xx >>yy;                     // 读入坐标
41       int j;
42       for(j=0; j<Y[yy].size(); j++)
                                           // 用下标遍历第 YY 列的向量
43         if(a[Y[yy][j]].x==xx)
                                           // 找到 X 坐标是 XX 的格子
44           break;
45       if(j==Y[yy].size()      //(xx,yy) 格子没有数字
46             ||a[Y[yy][j]].v!=minX[xx]  // 不是 xx 行最小值
47             ||a[Y[yy][j]].v!=minY[yy])  // 不是 yy 列最小值
48           cout <<"no"<<endl;            // 不是马鞍数
49       else
50       {
51           for(vector< int >::iterator it=Y[yy].begin();
```

```
52                  it!=Y[yy].end(); it++)
                                        //用迭代器遍历第 yy 列向量
53                  cout<<a[*it].v<<" ";    //打印遍历到的数
54              cout <<endl;
55          }
56      }
57      return 0;
58  }
```

说明：

（1）第17句定义了两个向量数组，分别记录N行和N列的数（在a数组中的下标）。

（2）程序第42句和第51句分别使用了下标和迭代器两种方法访问第yy列向量。

9.2.3 映射(map)

map是记录<关键字，数值>形式元素的容器。我们可以认为保存的元素按关键字次序构成一棵平衡树，插入、删除、查找一个元素的时间复杂度是O(log(N))级的。

map中的元素是一对数据：<关键字，数值>，这个概念在STL中有专门的数据结构叫pair，有一个相关函数make_pair和两个成员名：first、second。下面通过一个例子简单了解一些pair的相关知识。

【例9.5】坐标排序。输入N个不同的坐标，按从左到右、从上到下的次序重新输出。

参考程序如下：

```
1  //eg9.5
2  #include <iostream>
3  #include<algorithm>
4  using namespace std;
5  pair<int,int>a[10000];   //2个整数组成一对，每个数对表示一个
                              坐标。数组a可保存10000个坐标
6  int N;
7  int main()
8  {
9    cin >> N;
```

```
10    for (int i=0; i<N; i++)
11    {
12      int x,y;
13      cin >> x >>y;
14      a[i]= make_pair(x,y);      // 构造 pair，并赋值
15    }
16    sort(a,a+N);
17    for(int i=0; i<N; i++)
18      cout<<a[i].first<<" "<<a[i].second<<endl;
                                        //输出每个坐标的2个数值
19    return 0;
20  }
```

说明：

（1）第14句：两个数值组成一个pair整体，要使用函数make_pair。

（2）第18句：pair是一对数值，取第1个数值使用成员first，取第2个数值使用成员second。

（3）第16句：pair的默认大小比较，first成员是第一关键字，second成员是第二关键字。这里就是x轴优先排序。

表9.5是STL为map提供的一些成员函数。

表9.5

函数名	功　能
find（关键字）	返回指定关键字元素的位置迭代器
count（关键字）	统计指定关键字元素的个数。由于map每个元素的关键字都不相同，count结果只能是1或0
insert（元素）	插入元素到map中，元素一般是make_pair(关键字,值)
erase（关键字/迭代器）	删除map指定位置或指定关键字的元素
clear()	清除map所有元素，size变为0
运算符[]	取/赋值map的指定关键字的对应值，类似数组的下标运算，非常方便
begin()	map的第一个（最小）元素的位置，返回第一个元素迭代器（指针）
end()	map的结束位置。注意：返回的迭代器是最后一个元素的后面位置，不是最后一个元素的迭代器
size()	元素个数，map的大小——即map中已有元素的个数
empty()	判断map是否空，等价于size为0

【例9.6】购物。T宝网进行优惠活动，每个商品当天第一个订单可以打75折。这天网站先后依次收到了N（N<100000）个订单，订单的数据形式是：商品名　价格（单价*数量），商品名是长度不超过20的字符串。请按商品名字典序输出当天的每种商品名和其售出的总价。

分析：解决题目的关键有三个：

（1）判断一个订单是否为该商品的第一个订单。

（2）对指定商品的价格求和。

（3）依次输出结果。

使用map可以方便处理：

（1）使用count的结果可以判断某商品的订单是否出现过。

（2）使用[]可以方便赋值或累计价格。

（3）使用迭代器遍历map即可输出结果。

总时间复杂度为O(Nlog(N))。

参考程序如下：

```
1  //eg9.6
2  #include <iostream>
3  #include <map>
4  #include <string>
5  #include <iomanip>
6  using namespace std;
7  map< string, double > m;
8  int N;
9  int main()
10 {
11   cin >> N;
12   for(int i=0; i<N; i++)
13   {
14       string name;
15       double val;
16       cin >> name >> val;              //输入订单
17       if(m.count(name)==0 )            //商品名第1次出现
18           m[name]=val*0.75;            //打折
19       else
20       m[name]+=val;                    //不打折
21   }
22   for(map<string,double>::iterator ii=m.begin();
                                          //用迭代器遍历map
```

```
23      ii!=m.end(); ii++ )
24      cout <<ii->first<<" "       //输出商品名
25    <<fixed<<setprecision(2)<<ii->second<<endl;
                                    //输出付费
26    return 0;
27 }
```

说明：

（1）第7句：定义一个map，第一个数据项是string类型，第二个数据项是double型。

（2）第22句：按次序访问map，map自动按第一关键字排序。

（3）第18、20句：按“下标”法访问数据。“[ ]”里以关键字为“下标”。

（4）第24、25句：使用迭代器ii访问数据，类似指针。

9.3 几个常见的算法函数

STL中的算法库中前面使用最多的是sort函数，但也有一些其他好用的函数，下面简单介绍几个。

9.3.1 binary_search

这是个二分查找函数，一般格式为binary_search(开始位置，结束位置，要找的元素)。时间复杂度为O(logN)。如果在一个范围内找到值，返回“true”，否则返回“false”。

【例9.7】数组线段和查找。对于有N（<1000000）个正整数的数组A，问有多少段的和等于M。

分析：先求出前缀和数组a，如果存在a[i]-a[j]==M，数据j到i就是符合的一段，变形一下：枚举i，如果数组中存在值为（a[i]-M）的数，就是一个方案。

参考程序如下：

```
1  //eg9.7
2  #include <iostream>
3  #include<algorithm>
4  using namespace std;
5  int N,M, a[1000010];
```

```
int main()
{
  cin >> N >>M;
  for(int i=1; i<=N; i++)
      cin >> a[i];                    // 读入数组
    for(int i=1;i<=N;i++)             // 改造数组 a 为前缀和数组
      a[i]+=a[i-1];                   // 由于输入都是正整数，a 数组为单调的
    long long ans=0;
  for(int i=1;i<=N;i++)               // 枚举一段的右端
     if(binary_search(a,a+N,a[i]-M))
                                      // 判断左端点是否存在
          ans++;
  cout << ans<<endl;
  return 0;
}
```

9.3.2 lower_bound

binary_search 只能判断数的存在性，并不返回数据的位置。lower_bound 则二分查找出第一个大于等于指定数的位置（迭代器），如果没有找到，返回最后一个数据后面的位置。

【例 9.8】 最接近的大数。对于有 N（<1000000）个整数，有 M（<1000000）次询问，每次询问给定一个数 x，问数组中不比 x 小的最小数是哪一个？不存在时输出 123456789。

分析： N和M都很大，不能简单枚举，因此使用二分查找比较简明高效。

参考程序如下：

```
//eg9.8
#include <iostream>
#include<algorithm>
using namespace std;
int N,M,x,a[1000010];
int main()
{
  cin >> N >>M;
  for(int i=0; i<N; i++)
      cin >> a[i];                    // 读入数组
  sort(a,a+N);
```

```
12   for(int i=0; i<M; i++)
13   {
14        cin >>x;
15        int j=lower_bound(a,a+N,x)-a;
                              //通过指针的差，计算出数组下标位置
16        if(j==N) cout <<123456789<<endl;
                              //找不到不小于x的数时输出123456789
17        else cout <<a[ j ]<<endl;
18   }
19   return 0;
20 }
```

与lower_bound相近的算法函数还有upper_bound、equal_range等。

9.3.3 make_heap、push_heap、pop_heap、sort_heap

表9.6是一组关于堆操作的函数，对于要频繁求最大（或最小）的题目很有用处。

表9.6

函数名	功 能
make_heap（开始位置，结束位置）	对一段数组或向量建堆，默认是大根堆
push_heap（开始位置，结束位置）	在后面添加一个元素后，插入到堆中，堆元素个数增加1
pop_heap（开始位置，结束位置）	把堆顶元素弹出到堆尾，堆元素个数减少1，并维护堆
sort_heap（开始位置，结束位置）	对于堆，元素进一步全部排序，成递增数列

如果不使用默认的比较，函数要增加一个比较函数的参数，具体见下面例子。

【例9.9】合并果子（来源NOIP）。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1行是一个整数n（1<=n<=10000），表示果子的种类数；第2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：

3

1 2 9

输出样例：

15

数据规模：

对于30%的数据，保证有n<=1000

对于50%的数据，保证有n<=5000

对于全部的数据，保证有n<=10000

分析：经过分析和举例，可以猜想出用贪心法解决：每次挑选最小数目的两堆合并成新的一堆。具体的证明可以参考哈夫曼编码算法，这里就不再展开介绍。

每次选最小值，使用简单枚举算法复杂度为O(N*N)可能超时，应该使用高效的方法，显然堆数据结构比较合适。由于建堆默认是大根堆，一种方法就是增加个比较函数（和sort类似）；另一种方法是把所有数取反（变成负数）处理。

算法：使用2次pop_heap选出2个最小数，再用push_heap把和加入堆，重复N-1次。

参考程序如下：

```
1  //eg9.9
2  #include <iostream>
3  #include<algorithm>
```

```
4  using namespace std;
5  int N,ans,a[10010];
6  bool cmp(int a, int b)
7  {
8    return a>b;
9  }
10 int main()
11 {
12   cin >> N ;
13   for(int i=0; i<N; i++)
14        cin >> a[i];        //读入数组
15   make_heap(a,a+N,cmp); //对a[0]到a[N-1]建小根堆
16   ans=0;
17   for(int i=N;i>1;i--)   //每次合并2堆，一供合并N-1次
18   {
19        pop_heap(a,a+i,cmp);    //最小值弹到堆底a[i-1]
20        pop_heap(a,a+i-1,cmp); //最小值弹到堆底a[i-2]
21        a[i-2]+=a[i-1];
22        ans+=a[i-2];
23        push_heap(a,a+i-1,cmp); //新元素在a[i-2]，调整插入到堆中
24   }
25   cout << ans<<endl;
26   return 0;
27 }
```

说明：

（1）第6句：类似sort自己确定一个比较函数用来建立小根堆。

（2）第15句：和sort非常类似，给出数组的开头和结尾（指针）。

（3）第19、20句：堆顶元素弹出到堆底，要自己维护堆底指针。

（4）第21句：合并当前的最小两个数。

（5）第23句：push_heap会把堆底元素插入到堆的恰当位置。

（6）本题也可以使用优先队列容器（priority_queue）实现。

索 引

（TP-7439.01）
责任编辑　杨　凯
责任制作　魏　谨
封面设计　杨安安

CCF中学生计算机程序设计教材

全国青少年信息学奥林匹克竞赛

网　址：www.noi.cn　　E-mail：noi@ccf.org.cn

科学出版社互联网入口
销售：（010）64031535
E-mail：boktp@mail.sciencep.com

www.sciencep.com

ISBN 978-7-03-050029-8

定　价：36.00元