

重要程序整理

2008 年 3 月

北京师范大学第二附属中学

Fengzee

目录

第一部分 从C到C++的平稳过渡	3
第二部分 排序与统计算法	14
排序经典：冒泡排序算法(Bubble Sort)	14
最实用的算法：快速排序(Quick Sort)	16
快速排序实现之一：Hoare划分法——王老师给我们讲过的	17
快速排序实现之二：Lomuto划分法——也许这个写起来不易出错	18
当待排数字范围不大时： $O(n)$ 时间的计数排序算法(Counting Sort)	20
顺序统计学： $O(n)$ 时间查找无序数列中第 k 大元素	22
第三部分 枚举与排列	24
第四部分 堆、二叉查找树	29
堆(Heap)	29
二叉查找树(Binary Search Tree)	33
第五部分 图的建立和搜索	41
图的邻接表存储法	41
图的DFS	42
图的BFS	46
第六部分 图论最短路算法	49
像冒泡排序一样傻的Bellman-Ford算法	50
最常用的Dijkstra算法	52
使用队列优化的最短路径算法：SPFA	54
每对顶点间的最短路径：Floyd-Warshall算法	59
第七部分 再论图	61

最小生成树的Prim算法——王老师也讲过这个	61
二分图的最大匹配：匈牙利算法(Hungary)	63
第八部分 高精度运算	71
第九部分 推荐一些有价值的OI题	76
Matrix67 递推专项训练：DP思维训练的首选	76

前言

这套程序整理，包含入门阶段必学的一些基本算法和程序设计技巧。以提供源代码、并作概括性说明的方式，帮助初学者尽快掌握最重要的一些基础算法；保持笔者所在学校 OI 组内成员知识和水平的同步性，便于日后合作学习。

本文中较多内容为源代码，旁注的说明较简略，更详细的学习，建议参看相关书籍。

阅读本文请保持轻松，知识的来源不同，所用的词汇和说法很可能不同，看不懂并不意味着你的知识有问题，而只能说明我们习惯采用的组织知识的方式不同。

本文的写作获得了李伯尧同学的提示与支持，特此感谢。

本文自由使用、更改、传播。

冯子力(Fengzee)

2008 年 3 月 1 日

第一部分 从C到C++的平稳过渡

我们已经习惯了C语言。在 2008 年寒假期间，我和黎斐南同学接触C++语言，且发现了它相对于C语言的多项优点。C++和C的区别很多人并不了解，简要地，可以理解为“C++是C的超集”，换句话说，“C是C++的

子集”。这使得大部分C语言程序，都可以不经修改、或经过极少的修改，直接作为C++程序正确运行。一般而言，C语言程序具有“.c”的扩展名，而C++语言的程序则具有“.cpp”的扩展名¹。

我建议在现阶段尽量使用 **Dev-C++的编译环境**，这将使同学之间的交流更容易。当然，Dev-C++绝非目前最好的编程环境，例如它的调试功能被普遍认为不强。

这一部分讨论从 C 到 C++的转变。

写 C++程序的第一步，就是习惯于将自己的程序保存为 cpp 格式。在此基础上，我给出 C++语言的 Hello World 程序，可以看出，它和 C 语言主要在引用头文件和输出上有所不同。

```
// Hello World Program

#include <iostream>
using namespace std;

int main()
{
    cout << "Hello World!" << endl;
    return 0;
}
```

头文件 `iostream` 在 C++中的作用，就如同 `stdio.h` 在 C 语言中的作用一样。`stdio.h` 允许了 `printf()`, `scanf()` 等函数的使用，而 `iostream` 则允许了 `cin`, `cout` 等语句的使用。正如上面的例子所显示的，这一句输出“Hello World!”并换行。

以上只是一个引例，我如果继续这样细致地写下去，几天之内是完不成这一套程序整理的。所以，下面全文引用我在 2008 年 1 月 31 日写过的一篇关于 C++的文章。你可能无法完全理解文章的所有部分，因为它限于我当时的学习内容和写作风格，你可以跳过无法理解的内容。本文整个第一部分，仅作为 C++的介绍，完全凭此学习 C++还是不够的。但是看过这些简要介绍之后，你就能够写出正确的 C++程序了。

引用内容：

¹ 准确地说，只在一部分系统中才能保证这一点。有些系统下，C++源程序文件的扩展名为“.cc”。本文为了加快写作，尽量不涉及旁系知识。全文所有的程序代码，均保证且仅保证在 Dev-C++4.9.9.2 下编译运行正常。


我于 2008 年 1 月 29 日正式决定，改用 C++ 编程。引发这一决定的直接原因，是我看到有人用 C++ 写的一套高精度。C++ 的运算符重载真是好东西。改吧，我一咬牙，开始研究 C++。话说 C++ 是 C 的超集，此话不错，多数 C 程序改个扩展名，最多再改改头文件包含后，可以在 C++ 编译器下直接通过。但是 C++ 和 C 毕竟还是有一些区别的，我总结出十条，叙述如下：

（一）省略 return 0

使用 C++ 编程方便了一些，因为可以省略掉 main() 函数末尾的 return 0;。直接省略即可，放心，没有任何后果。编译器会帮你返回 0 的。


（二）变量传引用

这个词读起来比较拗口，事实上我也没有弄清楚它为什么叫这个名字。知识点的引入需要实例来配合，我们知道在 C 语言中，如果你想交换两个数（利用那 3 句经典语句）有两种方法，一是在程序里定义一个临时变量直接换，二是使用 #define，像这样：

 程序代码

```
#define SWAP(a, b){int temp=a; a=b; b=temp;}
```

也许有人会说，第三种方法是定义一个函数。很明确，这样是不行的：


 程序代码

```
void swap(int a,int b){  
    int tem=a;  
    a=b;  
    b=tem;  
}
```

我在初学时犯过这个错误，至今记忆犹新。因为我们调用 swap(a,b) 是传给 swap() 的

参数 `a,b` 是“一去不复返”的，即使 `a,b` 是全局变量也不行，因为 C 语言规定“同名的局部变量与全局变量通明时，局部变量有效”。

在 C++ 语言中，有一种被称为“变量传引用”的方式可以解决这一问题，在这里，`swap()` 函数应当这样声明和调用：

 程序代码


```
void swap(int &a,int &b){  
    int tem=a;  
    a=b;  
    b=tem;  
}
```

```
int main(){  
    cin>>a>>b;  
    swap(a,b);  
    cout<<a<<b<<endl;  
}
```

我们不将 `&` 看作取地址符。只要这样用，`a,b` 的值在被交换时就可以传回来了。


（三）结构体的简化

C 语言中，当我们声明了结构体 `struct name{.....};` 后，需要这样才能用这个结构体声明一个变量：

 程序代码

```
struct name name_of_variables;
```

而在 C++ 语言中，省略 `struct` 就可以了：

 程序代码


```
name name_of_variables;
```

（四）using namespaces std

有时候甄别一个程序是 C 还是 C++，可能最常用的却不正规的方式就是找有没有标题上这一句。C++中在引用头文件时，常常在旁边加上一句 `using namespaces std;`，不加这一句，会导致我们对头文件的引用无效。

（五）inline+经常调用的函数

记得我在写“初冬菜鸟邀请赛”的那道皇后问题标程的时候，曾经试图调用函数来判断状态是否可行，结果 TLE。我简单地把函数取消，再把判断功能用一个 if 语句完成，就 AC 了。最后得出结论：函数的调用是需要时间的。的确，调用函数时，计算机要给自己记下来“我调用了这个函数”，而这个过程需要时间。当我们完成一些非常简单的功能，比如上面提到的交换两数，或返回两数中较大/较小的一个，调用函数就显得浪费时间。可是假如把它写到主程序中呢，又会降低程序可读性。用 `#define` 语句呢，则在 Debug 时令你无所适从。C++完美地解决了这个问题，只需在声明这类函数时加上 `inline`，就会起到类似 `#define` 的效果，编译器会帮你把这个带 `inline` 的函数移回到主叫函数中，使调用不占用时间和空间，而且不会像用 `#define` 语句那样增加 Debug 的难度。例子：

 程序代码

```
inline void swap(int &a,int &b){  
    int tem=a;  
    a=b;  
    b=tem;  
}
```


（六）const 定义常量

这又是一种替代 `#define` 语句的方式，Pascal 们多半早就是这样用的了。学习 C 时可

能没有强调这一点，现在我们要强调：能用 `const int maxn=10000` 这样的常量定义方式的，不要用 `#define MAXN 10000`。


（七）动态空间申请语句的变更

以下两段程序，作用相同，仅语言不同：

 程序代码

```
int *a=(int *) malloc(sizeof(int));  
int *b=(int *) calloc( n, sizeof(int));
```

```
free(a);  
free(b);
```

 程序代码

```
int *a=new int;  
int *b=new int[n];  
delete a;  
delete[] b;
```

是不是更简洁了？


（八）bool 型不再需要 `stdbool.h`

C++ 本身提供 `bool` 类型，不需要像 C 一样先包含头文件 `stdbool` 或干脆用 `int`。

（九）自由的变量声明位置

写 C 程序时，我们经常需要回到程序开头，加入一个之前忘记声明的变量。而 C++ 中，变量可以随用随声明。当然，出于程序可读性的考虑，我们还是建议较为集中地 声明

变量。有一个情况可以例外，那就是循环变量，还是用的时候再声明吧，那样可以减轻看着程序开头一大堆变量的头疼感。


 程序代码

```
for(int i=1;i<=n;i++).....
```

注意，这种方式声明的循环变量，出了循环体后就会失效。

（十）string 型变量

C++有专门的字符串型变量 `string`，申请后允许这样的运算：

 程序代码

```
string a={"I am "};  
string b={"a student."};  
string c=a+b;  
cout<<c<<endl;
```


程序运行结果为：I am a student.

对于 `string` 型字符串的长度，这样来计算：`name.size()`。例如上例中 `b.size()`的值为 10。

上面的文章中对 C++的说明，事实上是非常不全面的。**C++最大的新特性——类——**还完全没有在我的介绍中出现。我在 2008 年 2 月 12 日，写过一篇关于类的文章。在下面，我将引用那篇文章大约一半的内容。预先说明一点，这篇文章中，我通过一种叫做堆的数据结构来引入对“类”语法的讲解，如果你不知道什么是堆，建议先阅读本文第三部分，那里讲解了堆的作用和实现方法。

对于类的介绍文字，引用如下：


C++的类太好了，今天用类写了一个堆，才发现原来很多东西都可以用类实现。如果不了解堆是什么，可以参考《算法导论》第七十几页的地方，有很详细的讲解。类体现了 OOP 的核心思想，OOP=Object-oriented Programming，即面向对象的编程。一个类 `class` 和一个结构 `struct` 类似，都可以用来“打包”一组变量，例如在类中，我们可以这样声明一个堆：

 程序代码

```
class heap{
    int data[MAXN];
    int heapsize;
public:
    void Input();
    void MaxHeapify(int k);
    void BuildMaxHeap();
    void ExtractMax();
    void Output();
};
```

这个声明的前两行非常容易理解，一个整型数组 `data[]` 存储堆中数据，一个整数 `heapsize` 记录堆的规模。可是，后面的“`public:`”之后的东西又是什么呢？它们是几个函数的原型，从这几个函数的命名可以知道，它们是对一个堆的几种基本操作。我们为什么要这样来声明一个类？

这个问题，要从类和结构的不同点开始说起。我们知道，如果上面声明的是一个结构的话，我们完全可以下面这样的语句来对堆中的元素进行操作：


 程序代码

```
struct heap{
    int data[MAXN];
    int heapsize;
};
```

```
int main(){
    heap h;
    h.data[1]=3;
    //.....
}
```

注意 `h.data[1]=3` 这一行，它直接地利用赋值语句，修改了一个名称为 `h` 的 `heap` 类型结构中的一个数据的值。然而对于类，这样的操作则是非法的。因为 `C++` 认为，类声明中“`public:`”之前的东西是“私有”的，是不能随便访问的。只有“`public:`”之后声明的函数，可以访问这些“私有数据”。这种把数据私有化的行为称为“封装”。就好像到银行取钱，顾客绝对不可以亲自进入金库，只能由银行的工作人员代替顾客访问金库一样。这时我们就说，银行的金库是“私有的”，是被封装的。而银行中完成各种工作的工作人员，则被称为这个银行的“接口”。外层的顾客只能通过接口同银行内层的私有数据交流，这一点保证了数据的安全。


假如我们已经写出了 `BuildMaxHeap()` 过程的代码，那么，这样可以调用这个属于类的函数：

 程序代码

```
int main(){
    heap h; // "heap" is a class
    h.MaxHeapify();
    //.....
}
```

我们之所以可以用一个点来连接 `h` 与函数名 `MaxHeapify()`，是因为这个函数是类中“公有的”，是位于 `public` 之后的。事实上，我们可以在类声明的开头冠以“`private:`”，来显式地标明类中变量的私有性，不过不写也可以，因为类默认私有。

然后来说说如何写这个 `MaxHeapify()` 函数的代码，它可以写在类声明的后面，使用它之前的位置。特别地，我们需要在函数头上标记类的名称：

 程序代码

```
void heap::MaxHeapify(){  
    //.....  
}
```

类 名称和函数名称之间，应当用两个连续的冒号连接。这种写法暗示我们，如果在一个程序中声明了不止一个类，各类中是可以使用名称相同的函数的。比如我们可以 写一个 `heap::Input()`，再写一个 `BST::Input()`；前者可以读入数据存入 一个堆，而后者可以读入一些数据插入一棵二叉查找树。当然，命名是自由的，我们可以使用任何其它名称，只要这个名称不是 C++关键字。

再回过头来，讨论一些细节。我们已经知道 `cin` 和 `cout` 是 C++标准的输入输出方式，它因为不用配置类型而使用简便。然而有时候，我们还是不能放弃 `printf()` 和 `scanf()` 函数的，想使用他们的时候，应该引用 `stdio.h` 头文件，在 C++中，写法要稍微改变一下：

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

要把 “.h” 去掉，在前面加 “c”，类似地，我们还可以这样来引用其它头文件：

```
#include <cstdlib>  
#include <cmath>  
#include <ctime>  
#include <cstring>  
using namespace std;
```

关于 C++的输入输出方式，网上多数的意见认为，使用 `cin` 和 `cout` 进行输入输出时的效率远比使用 `scanf()` 和 `printf()` 低。为此，我曾经做过一个试验，并写入了 2008 年 1 月 30 日的一篇文章中。

一个关于 C++流和 C 函数输入输出速度的比较试验：

.....

有一点值得一提,据多方面的消息称,使用 C++ 的流比使用 C 的输入输出语句慢很多,甚至有些考试和竞赛在试卷上注明了这一点。人们对同一件事的看法常常不同,我后来又看了一篇文章,讲到这两者的速度其实并无差别。

为此我刚刚做了实验,用改向文件的 C 方式和 C++ 方式(我会在后面说什么叫“改向文件”)测了 10 组快排,规模由 1000 递增到 300000,控制变量: (1)均直接调用 `qsort()` 函数; (2)均写在 .cpp 文件中使用 C++ 编译器编译; (3)相同的测试数据,均由 Cena 评测计时; (4)相同的程序结构和风格。唯一的不同是输入输出方式,结果令人惊讶,如下:

引用内容

(图一为 C 的 `scanf` 和 `printf` 方式,图二为 C++ 的 `fin` 和 `fout` 流方式。)

试题: 测试 文件名: C.cpp [重新评测](#)

测试点	评测结果	得分	用时	内存
1	正确	10	0.01s	236KB
2	正确	10	0.01s	236KB
3	正确	10	0.03s	248KB
4	正确	10	0.01s	268KB
5	正确	10	0.06s	312KB
6	正确	10	0.12s	428KB
7	正确	10	0.20s	624KB
8	正确	10	0.28s	1012KB
9	正确	10	0.42s	1208KB
10	正确	10	0.43s	1400KB

得分: 100 用时: 1.57s

图一


试题: 测试 文件名: CPP.cpp [重新评测](#)

测试点	评测结果	得分	用时	内存
1	正确	10	0.01s	224KB
2	正确	10	0.01s	228KB
3	正确	10	0.01s	240KB
4	正确	10	0.03s	260KB
5	正确	10	0.03s	308KB
6	正确	10	0.07s	424KB
7	正确	10	0.12s	620KB
8	正确	10	0.31s	1004KB
9	正确	10	0.32s	1204KB
10	正确	10	0.39s	1396KB

得分: 100 用时: 1.30s

图二

看到了吗？C++的流竟然比C的scanf和printf快！你如果不信，可以下载我测试用的程序看一看：

 [点击下载此文件](#)（引用注：当时的下载链接现在可能无法使用）

压缩包中有 3 个文件，C.cpp和CPP.cpp分别是用C和C++的输入输出方式写出的快排程序，数据太大我不给了，给一个数据生成器 Data_Creator.cpp，屏幕读取规模，生成数据到文件。这次实验同样还发现，所谓的“qsort函数很慢”也好像是谣言，以前我自己写快排函数时，最大规模同样也需要 0.40s+。所以提醒各位Oler，一定要自己亲自试过的东西才可以相信。

关于 C++的内容暂时写到这里，如果想要习惯一种新语言（事实上对我们来说是半种新语言），最好的方式就是用这种语言把自己以前写过的程序都翻译一遍。请不必花大量的时间学习 C++，因为我一直认为，语言是程序设计过程中最无趣的部分，算法才是关键；C++应当在使用中慢慢习惯，这个“习惯”的过程会很快的。下面的几部分是本文的核心内容，代码全部使用 C++方式，希望能够适应这一点。

第二部分 排序与统计算法

排序经典：冒泡排序算法(BUBBLE SORT)

这是最容易理解和最经典的排序算法，由于效率不高，一般实际用处不大。我在这套程序整理中收录它，是因为冒泡排序对于入门者来说，具有独特的学习价值。首先，冒泡排序可以通过加入一个 bool 型变量，并记录数字交换的情况来进行优化，这可以看作程序优化训练的入门课程；第二，冒泡排序和下面将要介绍的计数排序算法一样具有一种叫做“稳定性”的性质，这对于多关键字排序来说至关重要。

先给出代码。程序由一个名称为`data.in`²的文件中读入数据：第一行一个整数 $N(3 \leq N \leq 1000)$ ，第二行有 N 个整数 $(-2000 \leq \text{each} \leq 2000)$ ，空格分开，表示待排序的序列³。排序后，程序将从小到大的序列输出到`result.out`文件中。

```
// Bubble Sort Algorithm

#include <fstream>
using namespace std;

ifstream fin("data.in");
ofstream fout("result.out");

inline void swap (int &a, int &b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

void BubbleSort (int *num, int N)
{
    bool flag;
    int i = 0;
    do{
        flag = true;
        for (int j=0; j<=N-2-i; j++)
            if (num[j] > num[j+1])
            {
                swap (num[j], num[j+1]);
            }
        i++;
    } while (!flag);
}
```

² 本文中所有程序，均从 `data.in` 文件读入数据，并将结果输出到 `result.out` 文件中。这种方法仅供示例，竞赛时，请千万按照题目要求书写文件名。

³ 这里所述的输入输出格式，适用于本文第二和第三部分的大多数例程。

```
        flag = false;
    }
} while (flag==false);
}
```

```
int main ()
{
    int N, num[1001];
    fin >> N;
    for (int i=0; i<N; i++)
        fin >> num[i];
    BubbleSort (num, N);
    for (int i=0; i<N; i++)
        fout << num[i] << ' ';
    return 0;
}
```

简要介绍排序的**稳定性**，我们知道，一组待排序元素的关键字不一定是两两完全不同的，它们中可能有很多相同的数字；我们还知道，有时候我们不仅需要排序一系列数字，还要同时管理它们的“卫星数据”，例如，如果我们想要将一组给定长、宽的矩形按照它们的长（而不是宽）从小到大排序，在排序过程中交换“长”这个关键字时，还要对应地交换“宽”。**我们说一种排序算法是稳定的，当且仅当排序过后，关键字相同的元素的相对位置和排序前相比没有改变。**

冒泡排序是稳定的，前提是在交换条件那里，你必须写“>”而不是“>=”。排序的稳定对多关键字排序很重要。假如我们有一批学生的 3 门课的考试成绩，要求按照数学成绩排名，数学成绩相同时按照语文成绩排名，再相同时按照英语成绩排名。这个排序任务中，数学成绩是“最重要”的，语文次之，英语最不重要。于是，我们要先用一种排序算法以最无关紧要的英语成绩为关键字排序；排好后无论结果，再用稳定的排序算法（例如冒泡排序，或后面的计数排序）按照语文成绩再次排序；最后一步，无论结果如何对成绩再按照数学成绩排序。这样以后的结果就是我们所需要的。

最实用的算法：快速排序(QUICK SORT)

快速排序实现之一：HOARE 划分法——王老师给我们讲过的

```
// Quick Sort Algorithm, Hoare

#include <fstream>
using namespace std;

ifstream fin("data.in");
ofstream fout("result.out");

inline void swap (int &a, int &b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

void QuickSort (int *num, int left, int right)
{
    if (left < right)
    {
        int i = left-1, j = right+1,
            x = num[(left+right)>>1];
        while (1)
        {
            while (num[++i] < x);
            while (num[--j] > x);
            if (i>=j) break;
            swap (num[i], num[j]);
        }
        QuickSort (num, left, i-1);
```

```
        QuickSort (num, j+1, right);
    }
}
```

```
int main ()
{
    int N, num[100001];
    fin >> N;
    for (int i=0; i<N; i++)
        fin >> num[i];
    QuickSort (num, 0, N-1);
    for (int i=0; i<N; i++)
        fout << num[i] << ' ';
    return 0;
}
```

快速排序实现之二：LOMUTO 划分法——也许这个写起来不易出错

在学会了上面的 Hoare 划分快速排序之后，你已经可以完成大多数排序任务了。为了全面，我还要再介绍一种 Lomuto 划分方法。现在，我已经习惯了这种写法，并且认为它在边界值处理问题上不易出错。你可以选择跳过这一部分，但我建议你阅读，因为后面，我们将用这种 Lomuto 划分的方法完成一个查找序列中第 k 大元素的快速程序。

```
// Quick Sort Algorithm, Lomuto
```

```
#include <fstream>
```

```
using namespace std;
```

```
ifstream fin("data.in");
```

```
ofstream fout("result.out");
```

```
inline void swap (int &a, int &b)
```

```
{
```

```
    int tmp = a;
    a = b;
    b = tmp;
}

void QuickSort (int *num, int left, int right)
{
    if (left < right)
    {
        int i = left-1, j,
            x = num[right];
        for (j=left; j<right; j++)
            if (num[j]<=x)
            {
                i ++;
                swap (num[i], num[j]);
            }
        i ++;
        swap (num[i], num[right]);
        QuickSort (num, left, i-1);
        QuickSort (num, i+1, right);
    }
}

int main ()
{
    int N, num[100001];
    fin >> N;
    for (int i=0; i<N; i++)
        fin >> num[i];
    QuickSort (num, 0, N-1);
}
```

```
    for (int i=0; i<N; i++)  
        fout << num[i] << ' ';  
  
    return 0;  
}
```

当待排数字范围不大时： $O(N)$ 时间的计数排序算法(COUNTING SORT)

快速排序需要 $O(n \cdot \log(n))$ 的时间来运行，这已经是渐近最优的，因为快速排序是一种基于比较操作的排序算法，为了确定“应该的”顺序，我们必须要用 $n \cdot \log(n)$ 次比较。然而，我们并不一定在排序时比较数字，可以想到，假如已知的待排数字都位于 0-1000 的话，我们完全可以开一个尺寸为 1001 的 `bool` 型数组，初始化为 `false`，读到一个数 `i`，就将数组的 `[i]` 处标记为 `true`。最后，只需要扫描一遍数组，遇到 `true` 就输出数字就可以了。显然，这种方法是线性时间的。

当然，以上只是一种初步的设想，现实中，我们要排除一些特殊情况，例如，两个数乃至多个数相同时应该怎么办。也许，这时我们的 `bool` 数组要改为 `int` 型，用来记录某个数字出现的次数，而不仅仅是“是否出现”这样简单的信息。

以下算法称为“计数排序”，可以完美地实现这一点，更可贵的是，它是稳定的。这个程序可以处理每个数字在 0-1000 间的排序任务：

```
// Counting Sort Algorithm  
  
#include <fstream>  
using namespace std;  
  
ifstream fin("data.in");  
ofstream fout("result.out");  
  
inline void swap (int &a, int &b)  
{  
    int tmp = a;  
    a = b;  
    b = tmp;  
}
```

```
}

void CountingSort (int *num, int N)
{
    int c[1001] = {0}, res[100001];
    for (int i=1; i<=N; i++)
        c[num[i]] ++;
    for (int i=1; i<=1000; i++)
        c[i] += c[i-1];
    for (int i=N; i>0; i--)
    {
        res[c[num[i]]] = num[i];
        c[num[i]] --;
    }
    for (int i=1; i<=N; i++)
        num[i] = res[i];
}

int main ()
{
    int N, num[100001];
    fin >> N;
    for (int i=1; i<=N; i++)
        fin >> num[i];
    CountingSort (num, N);
    for (int i=1; i<=N; i++)
        fout << num[i] << ' ';
    return 0;
}
```

顺序统计学：O(N)时间查找无序数列中第k大元素

查找一个序列中的第 k 大元素，最明显的方法是将序列排序，然后输出第 k 个位置上的数字。然而这样的方法对于一般的序列来说，需要 $O(n \cdot \log(n))$ 的时间。有没有更快的方法呢？事实证明，是的，以下给出一个平均情况下 $O(n)$ 的算法。阅读这个程序之前，你需要了解快速排序中的 **Lomuto** 划分的基本运行原理，该程序我在上面给出过。

你也许疑惑，为什么这个程序是线性的？很遗憾我无法解释，《算法导论》中有详细的数学证明，涉及递归的解法，我目前还没有能力读懂。

程序从文件 **data.in** 读入数据，第一行一个数 N 表示数字个数，第二行 N 个数表示序列内容，第三行一个数 k 表示要求返回第 k 大元素。输出文件仅一行，为第 k 大元素。

```
// Select

#include <fstream>
using namespace std;

ifstream fin("data.in");
ofstream fout("result.out");

inline void swap (int &a, int &b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

int Select (int *num, int left, int right, int k)
{
    if (left < right)
    {
        int i = left-1, j,
            x = num[right];
```

```
        for (j=left; j<right; j++)
            if (num[j]<=x)
            {
                i ++;
                swap (num[i], num[j]);
            }
        i ++;
        swap (num[i], num[right]);

        int p = i-left+1;
        if (p == k)
            return num[i];
        if (p > k)
            return Select (num, left, i-1, k);
        if (p < k)
            return Select (num, i+1, right, k-p);
    }
    return num[left];
}

int main ()
{
    int N, k, num[100001];
    fin >> N;
    for (int i=0; i<N; i++)
        fin >> num[i];
    fin >> k;
    fout << Select (num, 0, N-1, k) << endl;
    return 0;
}
```

第三部分 枚举与排列

我认为这一部分是本文中最重要的一部分，因为它是大量基础题目的基本做法，同时可以有效地学习递归的程序设计思想。如果你精通了如何方便地进行枚举，你的基本功训练便至少完成了一半。

以下程序，可以输出 1-7 的自然数的 5040 种全排列：

```
// Permute

#include <fstream>
using namespace std;

ofstream fout ("result.out");

inline void swap (int &a, int &b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

void Permute (int *num, int n)
{
    if (n==7)
    {
        for (int i=0; i<7; i++)
            fout << num[i] << ' ';
        fout << endl;
        return;
    }
    for (int i=n; i<7; i++)
    {
```

```
        swap (num[i], num[n]);
        Permute (num, n+1);
    }
}

int main ()
{
    int num[7] = {1, 2, 3, 4, 5, 6, 7};
    Permute (num, 0);
    return 0;
}
```

写作本文之前的几天，我和学校 OI 组的几位同学参加了在北京举行的 CCC 竞赛。其中的第四题是典型的枚举类问题，题目大致叙述如下：

给出 4 个整数 ($1 \leq \text{each} \leq 13$)，对他们做加减乘除加括号的任意运算，其中除法要求不得出现小数 (中间结果也不行)，每个数字只能用一次。试凑出不大于 24 的最大整数。

这道题目既用了递归枚举，也用了排列生成，很有练习价值，我的代码如下：

```
// CCC2008, Senior Division Problem 4

/* -----
   Given 4 integers (1 <= each <= 13), try to get the maximum number
   less than or equal to 24, by using basic operations + - * /.
   ----- */

#include <fstream>      // For input data from file
#include <iostream>     // For output result to screen
using namespace std;

ifstream fin("s4.in"); // Input File
```

```
int maximum; // Record the best answer for the current test case
              // Set to 0 before every test case

inline void swap (int &a, int &b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

/* Operators:
    0 -> +
    1 -> -
    2 -> *
    3 -> /
*/
int Operate (int num1, int num2, int operator)
    /* The correct English word should be "operator" instead of "operater",
       but that's a key word for C++
    */
{
    switch (operator)
    {
        case 0: return num1 + num2;
        case 1: return num1 - num2;
        case 2: return num1 * num2;
        case 3:
            if (num2 == 0 || num1 % num2 != 0) return -293;
            /* Invalid divide operation, return a nagative value to make
               the result impossible to be a good one.
```

The nagative number should be a prime.

```
        */
    return num1 / num2;
}
}
```

```
void Calculate (int *num, int *opr)
```

```
{
    int ans, ans2;

    // Case 1: ( (A $ B) $ C ) $ D
    ans = Operate (num[0], num[1], opr[0]);
    ans = Operate (ans, num[2], opr[1]);
    ans = Operate (ans, num[3], opr[2]);
    if (ans<=24 && ans>maximum)    maximum = ans;

    // Case 2: (A $ B) $ (C $ D)
    ans = Operate (num[0], num[1], opr[0]);
    ans2= Operate (num[2], num[3], opr[2]);
    ans = Operate (ans, ans2, opr[1]);
    if (ans<=24 && ans>maximum)    maximum = ans;
}
```

```
void EnumerateOperation (int *num, int *opr, int n)
```

```
    /* pow(4, 3) == 64 */
{
    if (n==3)
    {
        Calculate (num, opr);
        return;
    }
}
```

```
    for (int i=0; i<4; i++)
    {
        opr[n] = i;
        EnumerateOperation (num, opr, n+1);
    }
}
```

```
void EnumeratePermute (int *num, int n)
    /* 4! == 24 */
{
    if (n==4)
    {
        int opr[3];
        EnumerateOperation (num, opr, 0);
        return;
    }
    for (int i=n; i<4; i++)
    {
        swap (num[i], num[n]);
        EnumeratePermute (num, n+1);
    }
}
```

```
int main()
{
    int N;
    fin >> N;    // Number of test cases

    for (int i=1; i<=N; i++)
    {
        int num[4];
```

```
for (int j=0; j<4; j++)           // Input data
    fin >> num[j];                //      |
                                //      |
    maximum = 0;                  //      |
EnumeratePermute (num, 0);         // Solve the problem
                                //      |
    cout << maximum << endl;      // Output result
}

system("pause"); // This line should be removed in the contest
return 0;
}

// Program by Fengzee, 2008.2.28
```

第四部分 堆、二叉查找树

堆(HEAP)

这是本文中介绍的第一种完整意义上的数据结构。之所以说“完整意义”，是因为堆与栈、队列等不同，它不仅仅是一种程序设计技巧，还是一套可以对动态数据集合进行多种操作的数据结构。“动态数据集合”，你可以理解为一堆数据，例如字典就算是一个数据集合，“动态”表明我们要经常对数据进行修改。这些修改，可能包括插入、删除、修改数据、取最大、取最小等等……一个好的数据结构，可以在相当低的时间复杂度下完成这些操作。我们要根据题目的需要，选择合适的数据结构。

堆是编码最简单的数据结构，因为它是可以用一个一维数组存储的（其他数据结构大多需要指针操作）。所谓树，就是有一个根结点，下面一层一层岔开的那种东西。堆是一种完全二叉树。就是说，堆的第 1 层（顶层）严格有 1 个结点，第 2 层 2 个，第 3 层 4 个，第 4 层 8 个……一直到最后一个数据为止。

我们用 $a[1]$ 来存储根结点的值， $a[2]$ 、 $a[3]$ 分别存储它的左、右儿子， $a[4]$ 、 $a[5]$ 、 $a[6]$ 、 $a[7]$ 则存储根结点的 4 个孙子，即 $a[2]$ 和 $a[3]$ 的儿子……依此类推，你会发现一个非常重要的性质：即每个结点 $a[i]$ 的左儿子位于 $a[i*2]$ ，右儿子位于 $a[i*2+1]$ ，父亲则位于 $a[i/2]$ (整除)。

堆分为最大堆和最小堆两种，本文仅以最大堆为例。它的性质是：**最大堆中任意结点的两个子结点（如果有的话），不大于该结点本身。**这种性质下，显然堆的根结点即 `a[1]` 中存储的数字是最大的。

以下程序中包含各种最大堆的基本操作，程序从文件中读入数据个数 `N` 和一行 `N` 个数，将他们整理成堆后输出一遍。然后每次从堆中删去一个最大值，再输出一遍，直到堆空为止。

```
// Heap

#include <fstream>
using namespace std;

ifstream fin("data.in");
ofstream fout("result.out");

class Heap
{
    int a[10001], heapsize;

    inline int l (int i)
    {
        return i<<1;
    }

    inline int r (int i)
    {
        return (i<<1)+1;
    }

    inline int p (int i)
    {
        return i>>1;
    }

    inline void swap (int &a, int &b)
    {
        int tmp = a;
```

```
        a = b;
        b = tmp;
    }

void MaxHeapify (int i)
{
    int largest;
    if ( l(i)<=heapsize && a[l(i)]>a[i] )
        largest = l(i);
    else
        largest = i;
    if ( r(i)<=heapsize && a[r(i)]>a[largest] )
        largest = r(i);
    if (largest != i)
    {
        swap (a[largest], a[i]);
        MaxHeapify (largest);
    }
}
```

public:

```
void Input ()
{
    fin >> heapsize;
    for (int i=1; i<=heapsize; i++)
        fin >> a[i];
}

void BuildMaxHeap ()
{

```

```
        for (int i=p(heapsize); i>=1; i--)  
            MaxHeapify (i);  
    }
```

```
int ExtractMax ()  
{  
    int tmp = a[1];  
    a[1] = a[heapsize];  
    heapsize --;  
    MaxHeapify (1);  
    return tmp;  
}
```

```
void Delete (int i)  
{  
    a[i] = a[heapsize];  
    heapsize --;  
    MaxHeapify (i);  
}
```

```
void IncreaseKey (int i, int x)  
{  
    a[i] = x;  
    while (a[i] < a[p(i)])  
        swap (a[i], a[p(i)]);  
}
```

```
int Output ()  
{  
    if (!heapsize) return -1;  
    for (int i=1; i<=heapsize; i++)
```

```
        fout << a[i] << ' ';\n        fout << endl;\n        return 0;\n    }\n};
```

```
int main ()\n{\n    Heap H;\n    H.Input();\n    H.BuildMaxHeap();\n    while ( H.Output() != 0 )\n        H.ExtractMax();\n    return 0;\n}
```

二叉查找树(BINARY SEARCH TREE)

以下程序，建立一棵二叉查找树，并按照中序遍历顺序输出其结点。程序中同时包含其他常见操作：

```
// Binary Search Tree\n\n#include <fstream>\nusing namespace std;\n\nifstream fin ("data.in");\nofstream fout ("result.out");\n\nclass Tree\n{\n    struct vertex
```

```
{
    int key;
    vertex *left, *right, *p;
};
vertex *head;

public:

inline vertex *h () { return head; }

void Insert (int key)
{
    vertex *x = head, *y = NULL, *z = new vertex;
    z->key = key, z->left = z->right = NULL;
    while (x != NULL)
    {
        y = x;
        if (key <= x->key) x = x->left;
        else x = x->right;
    }
    z->p = y;
    if (y == NULL) head = z;
    else if (key <= y->key) y->left = z;
    else y->right = z;
}

void BuildTree ()
{
    head = NULL;
    int N, x;
    fin >> N;
```

```
for (int i=0; i<N; i++)
{
    fin >> x;
    Insert (x);
}

vertex *Search (int key)
{
    vertex *x = head;
    while (x!=NULL && x->key != key)
    {
        if (key <= x->key) x = x->left;
        else x = x->right;
    }
    return x;
}

void Delete (vertex *z)
{
    vertex *x, *y;
    if ( !z->left || !z->right ) y = z;
    else
    {
        x = z->right;
        while (x)
        {
            y = x;
            x = x->left;
        }
    }
}
```

```
        if (y->left)
            x = y->left;
        else
            x = y->right;
        if (x)
            x->p = y->p;

        if (!y->p)
            head = x;
        else if (y->p->left == y)
            y->p->left = x;
        else
            y->p->right = x;

        if (y!=z)
            z->key = y->key;
        delete y;
    }

    void Walk (vertex *h)
    {
        if (h == NULL) return;
        Walk (h->left);
        fout << h->key << ' ';
        Walk (h->right);
    }
};

int main ()
{
    Tree T;
```

```
T.BuildTree ();  
T.Walk (T.h());  
return 0;  
}
```


如果你认为上述程序不易理解的话，可以参考我于 2008 年 3 月 1 日写成的一篇简要介绍二叉查找树的文章，部分引用如下：

指针操作是程序设计中“最危险”的一种，因为它直接针对内存。同时指针代码的错误也是最不容易修正的一种，因为往往编译器不会给出任何提示。我们建造二叉查找树，又不得不使用指针操作。

二叉查找树是一种二叉树，保证左子树的每个结点的关键字值不大于根结点，右子树的每个结点的关键字值不小于根结点，且左子树和右子树均为二叉查找树。

这种数据结构的意义是什么呢？如果一组数据已经排序，我们可以使用二分查找的方式搜索某个想要的关键字值，然而这样组织数据，插入和删除的代价是昂贵的：每插入或删除需要 $O(n)$ 的时间。假如我们直接在无需数据上(可以是数组或链表)操作，插入和删除倒是方便了， $O(1)$ ，查找又变成了 $O(n)$ 时间。二叉查找树，就是上述两种方式的折中，它既可以如二分查找那样在 $O(n \cdot \log(n))$ 时间内找到某个特定的关键字值，又可以做到在 $O(n \cdot \log(n))$ 的时间内(而不是 $O(n)$)插入和删除数据(树较平衡的情况下)，是一种相对性能很好的数据结构。

不过，这种良好性能也带来了编码的麻烦。首先，我们要定义一个结点的结构：

 程序代码

```
struct node  
{  
    int key;  
    node *left, *right, *p;  
}
```

显然，一个最简单的结点，具有一个关键字值和三个指针，分别指向左儿子、右儿子和父亲。然后，我们通过逐个插入结点的方式，在 $O(n \cdot \log(n))$ 的时间内完成二叉查找树的建立过程。用 C++ 的类编码，可以这样写：

程序代码

```
class Tree
{
    struct node
    {
        int key;
        node *left, *right, *p;
    };
    node *head; // It's must be NULL before the first insert operation

public:

    void insert (int key)
    {
        node *x = head, *y = NULL, *z = new node;
        z->key = key; z->left = z->right = NULL;
        while (x)
        {
            y = x;
            if (key <= x->key) x = x->left;
            else x = x->right;
        }
        z->p = y;
        if (!y) head = z;
        else if (key <= y->key) y->left = z;
        else y->right = z;
    }
};
```

这个插入过程，可以这样来概括：我们用 z 指针表示待插入的新结点，它对应的空间由 `new` 语句申请，`key` 域被置为带插入的关键字值。 x, y 两个指针配合，从二叉查找树的根部开始下降，直到 x 为 `NULL` 时， y 恰好停在某一片叶子上，于是，我们将 z 的父亲 $z \rightarrow p$ 置为 y ，表示 z “被挂在 y 下面”。接下来我们判断 y 是否为 `NULL`， y 什么时候会为 `NULL` 呢？当树是空的时候。如果这样，我们则把根结点指针 `head` 置为 z ， z 是树中第一个结点。否则的话，我们要比较一下待插入的关键字值和 $y \rightarrow key$ 的大小，根据情况把 $y \rightarrow left$ 或 $y \rightarrow right$ 置为 z 。

有了这样的一棵二叉查找树，查找的过程就十分简便了。也许我们可以写一个递归的查找过程，但大多数时候，应尽量不那样做，因为函数的调用需要时间，递归比迭代多出若干个调用函数的过程。用迭代法，写出二叉查找树的搜索过程如下：

程序代码

```
node *search (int key)
{
    node x = head;
    while (x && x->key != key)
    {
        if (key <= x->key) x = x->left;
        else x = x->right;
    }
    return x;
}
```

这个过程较之插入和删除，是很容易理解的。最困难的是下面的删除过程：

程序代码

```
void Delete (node *z)
{
    node *x, *y;
    if (!z->left || !z->right)
```

```
    y = z;
else
{
    x = z->right;
    while (x)
    {
        y = x;
        x = x->left;
    }
}
if (y->left) x = y->left;
else x = y->right;
if (x) x->p = y->p;
if (!y->p) head = x;
else if (y->p->left == y) y->p->left = x;
else y->p->right = x;
if (y != z)
    z->key = y->key;
delete y;
}
```

这个过程，以传入的指向待删除结点的指针 z 为参数。删除时，存在 3 种情况：

(情况 1) z 没有儿子，这时应直接删除 z ；

(情况 2) z 有一个儿子，这时应在 z 处将树“断开”，把 z 的儿子接到 z 的父亲上去；

(情况 3) z 有两个儿子，这时 z 的后继必然在 z 下方且只有一个儿子，我们把 z 的后继的数据复制到 z 中，然后删除 z 的后继。

上面的程序代码的组织，为了编码简便，和这三种情况的组织略有出入。代码中，我们用 y 表示真正待删除的结点，可以看到，当 z 至少有一个儿子为 `NULL` 时， y 被置为 z 本身，否则，`else` 括号中的语句会帮助 y 找到 z 的后继， x 在该大括号中作为辅助变量。确定 y 之后，我们把 x 置为 y 的非空的儿子，或者当 y 为 `NULL` 时被置为 `NULL`。如果 x 没有被置为 `NULL` 的话，我们就要进行“搭接操作”，令 x 的父亲 $x \rightarrow p$ 指向 y 的

父亲 $y \rightarrow p$ ，这样就绕过了结点 y ，等待删除。和插入有点类似，此时我们的 x 结点已经接到上面去了，上面还需要有结点来接受它，根据 y 原来的地位不同， $head$ ， $y \rightarrow p \rightarrow left$ 和 $y \rightarrow p \rightarrow right$ 三者中的某一个被置为 x ，这样树的结构就正确了。最后，如果我们发现 y 和真正待删除的 z 不同的话（这意味着 y 在过程开头被置为 z 的后继而不是 z 本身），就把 y 中的数据复制到 z 中去，因为 y 马上就要给删掉了。最后一句，`delete y`，该删就删干净，这一步不写的话，过程一旦结束，就有 `sizeof (node)` 的内存永远丢失了。如果内存丢得太多，程序就要崩溃。

这段话不好理解，背起程序来更令人头疼。

第五部分 图的建立和搜索

图的邻接表存储法

图是由边和结点组成的一种数学模型，用来描述事物间的关系。很多信息学问题都可以转化为图论问题，图论算法是很重要的。图应当如何在计算机中存储呢？我们都熟悉邻接矩阵存储法：如果有一个 N 个结点的图的话，我们开一个 $N*N$ 的矩阵，其中 $a[u][v]=w$ 表示结点 u 和结点 v 之间有一条权值为 w 的边。

实际问题中，一个图经常有上万的结点，边却不一定很多。这时如果采用邻接矩阵存储，势必浪费极大的空间。我们需要邻接表存储法。首先，图中的每个结点，用这样的结构来表示：

```
struct vertex
{
    int key, w;
    vertex *next;
};
```

然后，我们声明一个 `vertex *` 型数组 `connect[]`，长度和图的结点数相同，我们把 `connect[i]` 看作一个链表的表头，链表中的元素为所有和结点 i 相连的结点。用这样的过程，向图中插入一条从 u 指向 v ，权值为 w 的边：

```
void Insert (Graph *G, int u, int v, int w)
```

```
{  
    vertex *tmp = new vertex;  
    tmp->key = v;  
    tmp->w = w;  
    tmp->next = G->connect[u];  
    G->connect[u] = tmp;  
}
```

这个过程中，我们首先定义临时变量 `tmp`，类型为指向结点的指针，然后为其开辟存储空间。边是从 `u` 到 `v` 的，我们把 `tmp->key` 置为 `v`，`tmp->w` 置为边上的权值 `w`，然后，利用最后的两行把 `tmp` 所指向的这个新建立的结点插入到链表 `connect[u]` 中。

图的DFS

这是一个非常熟悉的问题，过程不多介绍。在 `BFS` 和 `DFS` 两种搜索中，我将给出邻接矩阵存储和邻接表存储两种程序。你也许不同意我编码的方式，认为为图定义一个结构过于罗嗦。但事实上，程序设计的结构化和对象化是应该提倡的，这样当题目复杂时，我们的操作可以一目了然。

下面是邻接矩阵的 `DFS`：

```
// DFS, Matrix  
  
#include <fstream>  
using namespace std;  
  
ifstream fin ("data.in");  
ofstream fout ("result.out");  
  
struct Graph  
{  
    int gv, ge;  
    int a[1001][1001];  
    bool visited[1001];  
}
```

```
};

void DFS (Graph *G, int head)
{
    for (int i=1; i<= G->gv; i++)
        if (!G->visited[i] && G->a[head][i])
        {
            G->visited[i] = true;
            fout << i << ' ';
            DFS (G, i);
        }
}

int main ()
{
    Graph *G = new Graph;
    int u, v, w;
    fin >> G->gv >> G->ge ;
    for (int i=1; i<= G->ge; i++)
    {
        fin >> u >> v >> w;
        G->a[u][v] = G->a[v][u] = w;
    }
    fout << 1 << ' ';
    G->visited[1] = true;
    DFS (G, 1);
    return 0;
}
```

邻接表的 DFS:

```
// DFS, List
```

```
#include <fstream>
```

```
using namespace std;
```

```
ifstream fin ("data.in");
```

```
ofstream fout ("result.out");
```

```
struct Graph
```

```
{
```

```
    struct vertex
```

```
    {
```

```
        int key, w;
```

```
        vertex *next;
```

```
    };
```

```
    int gv, ge;
```

```
    vertex *connect[100001];
```

```
    bool visited[100001];
```

```
};
```

```
void Insert (Graph *G, int u, int v, int w)
```

```
{
```

```
    Graph::vertex *tmp = new Graph::vertex;
```

```
    tmp->key = v;
```

```
    tmp->w = w;
```

```
    tmp->next = G->connect[u];
```

```
    G->connect[u] = tmp;
```

```
}
```

```
void DFS (Graph *G, int head)
```

```
{
```

```
Graph::vertex *tmp = G->connect[head];
while (tmp)
{
    if (!G->visited[tmp->key])
    {
        G->visited[tmp->key] = true;
        fout << tmp->key << ' ';
        DFS (G, tmp->key);
    }
    tmp = tmp->next;
}
}
```

```
int main ()
{
    Graph *G = new Graph;
    int u, v, w;
    fin >> G->gv >> G->ge ;
    for (int i=1; i<= G->ge; i++)
    {
        fin >> u >> v >> w;
        Insert (G, u, v, w);
        Insert (G, v, u, w);
    }
    fout << 1 << ' ';
    G->visited[1] = true;
    DFS (G, 1);
    return 0;
}
```

图的BFS

邻接矩阵的 BFS:

```
// BFS, Matrix

#include <fstream>
using namespace std;

ifstream fin ("data.in");
ofstream fout ("result.out");

struct Graph
{
    int gv, ge;
    int a[1001][1001];
    bool visited[1001];
};

void BFS (Graph *G, int head)
{
    int closed = 0, open = 1, queue[10000];
    queue[1] = head;
    do{
        closed ++;
        for (int i=1; i<= G->gv; i++)
            if (!G->visited[i] && G->a[queue[closed]][i])
            {
                G->visited[i] = true;
                fout << i << ' ';
                open ++;
                queue[open] = i;
            }
    } while (open <= closed);
}
```

```
        }
    } while (closed < open);
}

int main ()
{
    Graph *G = new Graph;
    int u, v, w;
    fin >> G->gv >> G->ge ;
    for (int i=1; i<= G->ge; i++)
    {
        fin >> u >> v >> w;
        G->a[u][v] = G->a[v][u] = w;
    }
    fout << 1 << ' ';
    G->visited[1] = true;
    BFS (G, 1);
    return 0;
}
```

邻接表的 BFS:

```
// BFS, List

#include <fstream>
using namespace std;

ifstream fin ("data.in");
ofstream fout ("result.out");
```

```
struct Graph
{
    struct vertex
    {
        int key, w;
        vertex *next;
    };
    int gv, ge;
    vertex *connect[100001];
    bool visited[100001];
};

void Insert (Graph *G, int u, int v, int w)
{
    Graph::vertex *tmp = new Graph::vertex;
    tmp->key = v;
    tmp->w = w;
    tmp->next = G->connect[u];
    G->connect[u] = tmp;
}

void BFS (Graph *G, int head)
{
    int closed = 0, open = 1, queue[10000];
    queue[1] = head;
    do{
        closed ++;
        Graph::vertex *tmp = G->connect[queue[closed]];
        while (tmp)
        {
            if (!G->visited[tmp->key])
```



```
        {
            G->visited[tmp->key] = true;
            fout << tmp->key << ' ';
            open ++;
            queue[open] = tmp->key;
        }
        tmp = tmp->next;
    }
} while (closed < open);
}
```

```
int main ()
{
    Graph *G = new Graph;
    int u, v, w;
    fin >> G->gv >> G->ge ;
    for (int i=1; i<= G->ge; i++)
    {
        fin >> u >> v >> w;
        Insert (G, u, v, w);
        Insert (G, v, u, w);
    }
    fout << 1 << ' ';
    G->visited[1] = true;
    BFS (G, 1);
    return 0;
}
```

第六部分 图论最短路算法

像冒泡排序一样傻的BELLMAN-FORD算法

图论最短路算法，分为“单源最短路径算法”和“每对顶点之间的最短路径算法”，我介绍的前三种算法属于前者，Floyd-Warshall 则属于后者。所谓单源最短路径算法，指的是求解从某个给定的结点出发，到达其他所有结点的最短路径。

几乎所有的最短路径算法都基于以下定理：三角形两边之和大于第三边。在图论中，我们要对定理稍作修改：三角形两边之和大于等于第三边。利用该原理改进结点的最短路径的操作成为“松弛”，它可以这样表示：

```
if (Dist[v] > Dist[u] + w[u, v])
    Dist[v] = Dist[u] + w[u, v];
```

以上操作，可称为“通过结点 u 对结点 v 进行松弛”，需要 $O(1)$ 时间。而“利用结点 u 进行松弛”，则表示通过结点 u 对所有其他结点进行松弛，它对于 N 个结点的图需要 $O(N)$ 时间。Bellman-Ford 算法执行这样的过程：以任意顺序(为了方便通常是结点编号顺序)，利用每个结点进行松弛，重复 $N-1$ 次，此时 $\text{Dist}[]$ 数组必然被更新到最短路径值。

这个算法的时间复杂度是 $O(N^3)$ 的，最短路算法复杂度一般都不很低，所以我们采用邻接矩阵表示法。如果你还记得冒泡排序的话，我们有一个明显的优化。维护一个 `bool` 变量 `flag`，每轮开始前置为 `true`，如果这一轮中发生了松弛操作，则置为 `false`，反复循环，直到 `flag` 的值在某轮结束时为 `true` 为止。

代码如下：

```
// Bellman-Ford Algorithm
```

```
#include <fstream>
```

```
using namespace std;
```

```
ifstream fin ("data.in");
```

```
ofstream fout ("result.out");
```

```
struct Graph
```

```
{
```

```
    int gv, ge;
```

```
    int w[1001][1001];
```

```
int dist[1001];

};

void Bellman_Ford (Graph *G, int s)
{
    bool flag;
    do{
        flag = true;
        for (int j=1; j<= G->gv; j++)
            for (int k=1; k<= G->gv; k++)
                if (G->dist[k] > G->dist[j] + G->w[j][k])
                {
                    flag = false;
                    G->dist[k] = G->dist[j] + G->w[j][k];
                }
    } while (flag == false);
}

int main ()
{
    Graph *G = new Graph;
    int u, v, wei;
    fin >> G->gv >> G->ge ;
    for (int i=1; i<= G->gv; i++)
    {
        G->dist[i] = 10000;
        for (int j=1; j<= G->gv; j++)
            G->w[i][j] = 10000;
    }
    G->dist[1] = 0;
    for (int i=1; i<= G->ge; i++)
```

```
{
    fin >> u >> v >> wei;
    G->w[u][v] = G->w[v][u] = wei;
}
Bellman_Ford (G, 1);
for (int i=1; i<= G->gv; i++)
    if (G->dist[i] < 10000)
        fout << i << ' ' << G->dist[i] << endl;
return 0;
}
```

最常用的DIJKSTRA算法

毫无疑问，Bellman-Ford 算法太傻了，更聪明些，我们有 Dijkstra 算法。与机械地重复松弛最多 $N-1$ 次的做法不同，Dijkstra 只需借助每个结点松弛一次，前提是，每次的这个结点，必须是目前没有利用过的结点中期望最短路径值 Dist 最小的一个。

```
// Dijkstra Algorithm

#include <fstream>
using namespace std;

ifstream fin ("data.in");
ofstream fout ("result.out");

struct Graph
{
    int gv, ge;
    int w[1001][1001];
    int dist[1001];
    bool visited[1001];
```

```
};
```

```
void Dijkstra (Graph *G, int s)
{
    int p, minimum;
    for (int i=1; i<= G->gv; i++)
    {
        minimum = 10000;
        for (int j=1; j<= G->gv; j++)
            if (!G->visited[j] && G->dist[j]<minimum)
            {
                minimum = G->dist[j];
                p = j;
            }
        G->visited[p] = true;
        for (int j=1; j<= G->gv; j++)
            if (p!=j && G->dist[j] > G->dist[p] + G->w[p][j])
                G->dist[j] = G->dist[p] + G->w[p][j];
    }
}
```

```
int main ()
{
    Graph *G = new Graph;
    int u, v, wei;
    fin >> G->gv >> G->ge ;
    for (int i=1; i<= G->gv; i++)
    {
        G->dist[i] = 10000;
        for (int j=1; j<= G->gv; j++)
            G->w[i][j] = 10000;
```

```
}
G->dist[1] = 0;
for (int i=1; i<= G->ge; i++)
{
    fin >> u >> v >> wei;
    G->w[u][v] = G->w[v][u] = wei;
}
Dijkstra (G, 1);
for (int i=1; i<= G->gv; i++)
    if (G->dist[i] < 10000)
        fout << i << ' ' << G->dist[i] << endl;
return 0;
}
```

使用队列优化的最短路径算法：SPFA

SPFA = Shortest Path Fast Algorithm，这个名字很俗。

SPFA 算法也是一种性能优越的算法，和 Dijkstra “从小到大松弛结点”的思想不同，SPFA 利用了一个队列。它的基本想法是：开始时，队头放置源点；每次从队列中取出一个元素，利用它进行松弛；如果程序运行过程中，结点 v 的距离值 $\text{Dist}[v]$ 被改进了，那么就有可能用 v 改进其他的结点，如果我们通过查阅一个 `bool` 数组发现 v 目前不在队列中，我们便将 v 加入队列。如此执行，直到队列为空。

更详细地，我部分引用 2008 年 2 月 15 日写的一篇文章，来说明 SPFA 算法：

SPFA 是这篇日志要写的一种算法，它的性能非常好，代码实现也并不复杂。特别是当图的规模大，用邻接矩阵存不下的时候，用 SPFA 则可以很方便地面对临接表。每个人都写过广搜，SPFA 的实现和广搜非常相似。

如何求得最短路径的长度值？

首先说明，SPFA 是一种单源最短路径算法，所以以下所说的“某点的最短路径长度”，指的是“某点到源点的最短路径长度”。

我们记源点为 S ，由源点到达点 i 的“当前最短路径”为 $D[i]$ ，开始时将所有 $D[i]$ 初始化为无穷大， $D[S]$ 则初始化为 0。算法所要做的，就是在运行过程中，不断尝试减小 $D[]$ 数组的元素，最终将其中每一个元素减小到实际的最短路径。

过程中，我们要维护一个队列，开始时将源点置于队首，然后反复进行这样的操作，直到队列为空：

(1) 从队首取出一个结点 u ，扫描所有由 u 结点可以一步到达的结点，具体的扫描过程，随存储方式的不同而不同；

(2) 一旦发现有这样一个结点，记为 v ，满足 $D[v] > D[u] + w(u, v)$ ，则将 $D[v]$ 的值减小，减小到和 $D[u] + w(u, v)$ 相等。其中， $w(u, v)$ 为图中的边 $u-v$ 的长度，由于 $u-v$ 必相邻，所以这个长度一定已知（不然我们得到的也不叫一个完整的图）；这种操作叫做松弛。

引用内容

松弛操作的原理是著名的定理：“三角形两边之和大于第三边”，在信息学中我们叫它三角不等式。所谓对 i, j 进行松弛，就是判定是否 $d[j] > d[i] + w[i, j]$ ，如果该式成立则将 $d[j]$ 减小到 $d[i] + w[i, j]$ ，否则不动。

(3) 上一步中，我们认为我们“改进了”结点 v 的最短路径，结点 v 的当前路径长度 $D[v]$ 相比于以前减小了一些，于是，与 v 相连的一些结点的路径长度可能会相应地减小。注意，是可能，而不是一定。但即使如此，我们仍然要将 v 加入到队列中等待处理，以保证这些结点的路径值在算法结束时被降至最优。当然，如果连接至 v 的边较多，算法运行中，结点 v 的路径长度可能会多次被改进，如果我们因此而将 v 加入队列多次，后续的工作无疑是冗余的。这样，就需要我们维护一个 `bool` 数组 `Inqueue[]`，来记录每一个结点是否已经在队列中。我们仅将尚未加入队列的点加入队列。

算法能否结束？

对于不存在负权回路的图来说，上述算法是一定会结束的。因为算法在反复优化各个最短路径长度，总有一个时刻会进入“无法再优化”的局面，此时一旦队列读空，算法

就结束了。然而，如果图中存在一条权值为负的回路，就糟糕了，算法会在其上反复运行，通过“绕圈”来无休止地试图减小某些相关点的最短路径值。假如我们不能保证图中没有负权回路，一种“结束条件”是必要的。这种结束条件是什么呢？

思考 Bellman-Ford 算法，它是如何结束的？显然，最朴素的 Bellman-Ford 算法不管循环过程中发生了什么，一概要循环 $|V|-1$ 遍才肯结束。凭直觉我们可以感到，SPFA 算法“更聪明一些”，就是说我们可以猜测，假如在 SPFA 中，一个点进入队列——或者说一个点被处理——超过了 $|V|$ 次，那么就可以断定图中存在负权回路了。

最短路径本身怎么输出？

在一幅图中，我们仅仅知道结点 A 到结点 E 的最短路径长度是 73，有时候意义不大。这附图如果是地图的模型的话，在算出最短路径长度后，我们总要说明“怎么走”才算真正解决了问题。如何在计算过程中记录下来最短路径是怎么走的，并在最后将它输出呢？

Path [] 数组，Path[i] 表示从 S 到 i 的最短路径中，结点 i 之前的结点的编号。注意，是“之前”，不是“之后”。最短路径算法的核心思想称为“松弛”，原理是三角形不等式，方法是上文已经提及的。我们只需要在借助结点 u 对结点 v 进行松弛的同时，标记下 Path[v] = u，记录的工作就完成了。

输出时可能会遇到一点难处，我们记的是每个点“前面的”点是什么，输出却要从最前面往最后面输，这不好办。其实很好办，见如下递归方法：

程序代码

```
void PrintPath(int k){
    if( Path[k] ) PrintPath(Path[k]);
    fout<<k<<' ';
}
```

然后，我给出 SPFA 的代码。请注意，下面这段代码中没有加入统计最短路径本身的功能：

```
// Shortest Path Fast Algorithm
```

```
#include <fstream>
```

```
using namespace std;
```

```
ifstream fin ("data.in");
```

```
ofstream fout ("result.out");
```

```
struct Graph
```

```
{
```

```
    int gv, ge;
```

```
    struct vertex
```

```
    {
```

```
        int key, w;
```

```
        vertex *next;
```

```
    };
```

```
    vertex *connect[100001];
```

```
    int dist[100001];
```

```
    bool inque[100001];
```

```
};
```

```
void insert (Graph *G, int u, int v, int w)
```

```
{
```

```
    Graph::vertex *tmp = new Graph::vertex;
```

```
    tmp->key = v;
```

```
    tmp->w = w;
```

```
    tmp->next = G->connect[u];
```

```
    G->connect[u] = tmp;
```

```
}
```

```
void SPFA (Graph *G, int s)
```

```
{
    int queue[200000];
    int closed = 0, open = 1;
    queue[1] = s;
    Graph::vertex *tmp;
    do{
        closed ++;
        tmp = G->connect[queue[closed]];
        G->inque[queue[closed]] = false;
        while (tmp)
        {
            if (G->dist[tmp->key] > G->dist[queue[closed]] + tmp->w)
            {
                G->dist[tmp->key] = G->dist[queue[closed]] + tmp->w;
                if (!G->inque[tmp->key])
                {
                    G->inque[tmp->key] = true;
                    open ++;
                    queue[open] = tmp->key;
                }
            }
            tmp = tmp->next;
        }
    } while (closed < open);
}

int main ()
{
    Graph *G = new Graph;
    fin >> G->gv >> G->ge;
    int u, v, wei;
```

```
    for (int i=1; i<= G->ge; i++)
    {
        fin >> u >> v >> wei;
        insert (G, u, v, wei);
        insert (G, v, u, wei);
    }
    for (int i=1; i<= G->gv; i++)
        G->dist[i] = 10000;
    G->dist[1] = 0;
    SPFA (G, 1);
    for (int i=1; i<= G->gv; i++)
        if (G->dist[i]<10000)
            fout << i << ' ' << G->dist[i] << endl;

    return 0;
}
```

每对顶点间的最短路径：FLOYD-WARSHALL算法

前面讨论的 Bellman-Ford, Dijkstra 和 SPFA 算法，都是单源最短路径算法。如果我们想要得到任意两点间的最短路径的话，需要使用以 Floyd-Warshall 算法为代表的“每对顶点间的最短路径算法”。Floyd-Warshall 是一种动态规划算法。

```
// Floyd-Warshall Algorithm
```

```
#include <fstream>
```

```
using namespace std;
```

```
ifstream fin ("data.in");
```

```
ofstream fout ("result.out");
```

```
struct Graph
```

```
{
    int gv, ge;
    int DP[201][201][201];
};

void Floyd_Warshall (Graph *G)
{
    for (int k=1; k<= G->gv; k++)
        for (int i=1; i<= G->gv; i++)
            for (int j=1; j<= G->gv; j++)
                G->DP[i][j][k] =
                    G->DP[i][j][k-1] < G->DP[i][k][k-1] + G->DP[k][j][k-1] ?
                    G->DP[i][j][k-1] : G->DP[i][k][k-1] + G->DP[k][j][k-1] ;
}

int main ()
{
    Graph *G = new Graph;
    fin >> G->gv >> G->ge;
    int u, v, wei;
    for (int i=1; i<= G->gv; i++)
        for (int j=1; j<= G->gv; j++)
            G->DP[i][j][0] = 10000;
    for (int i=1; i<= G->ge; i++)
    {
        fin >> u >> v >> wei;
        G->DP[u][v][0] = wei;
        G->DP[v][u][0] = wei;
    }
    Floyd_Warshall (G);
    for (int i=1; i<= G->gv; i++)
```

```
{  
    for (int j=1; j<= G->gv; j++)  
    {  
        if (G->DP[i][j][G->gv]>=10000 || i==j)  
            fout << 0 <<' '  
        else  
            fout << G->DP[i][j][G->gv] <<' '  
        }  
        fout << endl;  
    }  
    return 0;  
}
```

第七部分 再论图

最小生成树的PRIM算法——王老师也讲过这个

所谓最小生成树，是指选取图中的若干条边，使得每个结点均至少和所选的一边相连，且所选边的权值总和最小。这个问题具有非常大的实际意义。例如我们想在若干台电脑间联网，每两台电脑间联网有一个特定的成本，我们想在所有电脑纳入网络的前提下，使得总成本最小。

Prim 算法是一种贪心算法，它的原理出奇的简单。我们只需从任意结点开始，不断选取目前已经纳入生成树的结点中权值最小的一个“对外的”边，直到所有结点被纳入树中，所得的生成树就是最小的。它的代码实现和 Dijkstra 算法极其相似。

```
// Prim Algorithm
```

```
#include <fstream>
```

```
using namespace std;
```

```
ifstream fin ("data.in");
ofstream fout ("result.out");

struct Graph
{
    int gv, ge;
    int a[1001][1001];
    int w[1001];
    bool visited[1001];
};

int Prim (Graph *G)
{
    int p, minimum, sum = 0;
    for (int i=1; i<= G->gv; i++)
    {
        minimum = 10000;
        for (int j=1; j<= G->gv; j++)
            if (!G->visited[j] && G->w[j]<minimum)
            {
                minimum = G->w[j];
                p = j;
            }
        G->visited[p] = true;
        sum += minimum;
        for (int j=1; j<= G->gv; j++)
            if (!G->visited[j] && G->w[j] > G->a[p][j])
                G->w[j] = G->a[p][j];
    }
    return sum;
}
```

```
int main ()
{
    Graph *G = new Graph;
    int u, v, w;
    fin >> G->gv >> G->ge ;
    for (int i=1; i<= G->gv; i++)
        for (int j=1; j<= G->gv; j++)
            G->a[i][j] = 10000;
    for (int i=1; i<= G->ge; i++)
    {
        fin >> u >> v >> w;
        G->a[u][v] = G->a[v][u] = w;
    }
    for (int i=1; i<= G->gv; i++)
        G->w[i] = 10000;
    G->w[1] = 0;
    fout << Prim (G) << endl;
    return 0;
}
```

值得注意的是，本文中 Dijkstra 算法和 Prim 算法的实现采用的是最朴素的方式，事实上，如果我们观察到这两个算法中我们需要反复选取最小值的事实，可以轻易想到一种优化方法：最小堆。堆的介绍前文已经有过。一种数据结构只有在使用中才能体现它的价值，因此，对于任意需要动用这两种算法的题目，请一定通过堆来实现，代码会长些，这是快速的代价。

二分图的最大匹配：匈牙利算法(HUNGARY)

全文引用我在 2008 年 2 月 3 日写过的一篇文章：

我们都是十六七岁的人。话说总有那么一天，很可能是十年以内，我和我周围的每一个人都会找到自己终身的伴侣。然而感情的道路是波折的，也许就在此刻，你的 脑海

中还有 3 个候选人（或许 5 个.....）。不知你可曾想过，你中意的每一个人，可能还有她们自己的三五个候选人。自然的力量是如何发挥作用，将人类的两种，从最初杂乱无章的状态下匹配成一对一对的，并使成功匹配的对数尽量多的呢？这正是本文将要讨论的问题。

二分图的概念

我到网上搜了一通，几乎每一篇介绍二分图和它的匹配问题的文章都不可避免地在开头摆出一大串令人读后头疼的定义，所以我才举了上面这个例子。所谓**二分图**，就是所有结点被分成两类的图，正如人类被分成男性和女性；某些联系**只可能**在两类之间发生，而**不可能**在同一类之间发生。就像我们总是感叹常规爱情的美好，却不能坦然地接受 BL 和 GL 一般。稍微形式化地，给出如下定义：

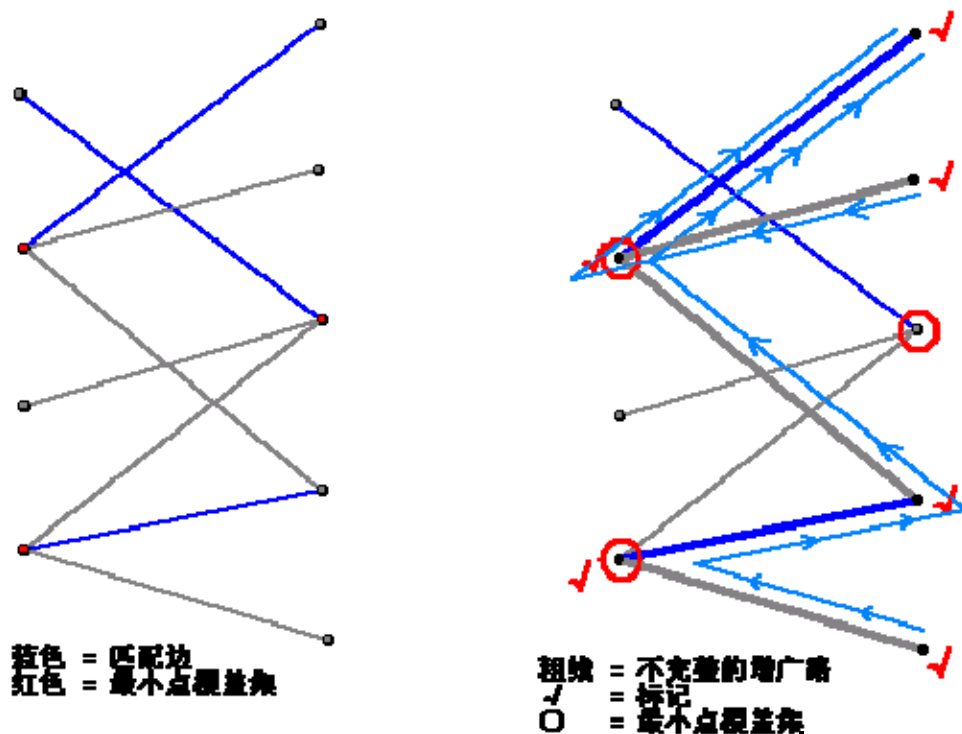
引用内容

二分图是这样一个图，它的顶点可以分类两个集合 X 和 Y ，每一条边关联的两个顶点中，恰好一个属于集合 X ，另一个属于集合 Y 。

总之，可以用这样两句话来描述二分图，它们具有相同的含义，仅表达角度不同：

1. 二分图的每一条边的两个端点分居于两个集合中；
2. 同一个集合中的任何两个结点间不存在边。

画个图表示，就更明白了：



（图片来自Matrix67.com，引在这里可能不完全恰当，请忽略多余的注释和标记，仅关注左图中点被分成左右两组、两组间互相关联的事实）

二分图的**匹配**，听起来像是一个动词，但这个词最初的定义是名词性的。它是指二分图中所有边组成的集合的一个子集（知道“子集”是什么意思吧），同时要求该子集中**没有任何**两条边具有公共端点。而二分图的**最大匹配**，则指所有的匹配中，包含边数最多的那一种匹配。换句话说，就是令这个世界上找不到另一半的人最少的那一种匹配。更进一步地，如果整个世界上再也没有任何一个人需要过 11 月 11 日那个节日，我们称这时的匹配为**完美匹配**。显然，完美匹配有可能存在，仅当二分图中两个集合中的结点数相等。


堆了一大堆的概念，目的是介绍这几个关键词：二分图、二分图的匹配、二分图的最大匹配、二分图的完美匹配。进一步读下去之前，请回溯一下这几个词的含义。

二分图最大匹配的Hungary算法

我 以前也不懂Hungary这个词的意思，今天才明白它其实就是匈牙利，下面要介绍的算法称为匈牙利算法。和通常的算法介绍不同，这次我不得不先给代码。图 采用邻接

矩阵表示，存入 `bool connect[][]`，`true` 表示连通，`false` 表示不连通。程序调用 `Max ()` 返回的值，即为该图的最大匹配数。鉴于人们往往不信任网页上拷下来的程序（至少我是这样），提供该算法的一个实例的下载，它同时也是本文后面要介绍的**例题 2** 的标程：

 [点击下载此文件](#)

 程序代码

```
bool used[MAXN]; //记录 y 中节点是否使用
int link[MAXN]; //记录当前与 y 节点相连的 x 的节点
bool connect[N][N]; //记录连接 x 和 y 的边，如果 i 和 j 之间有边则为 1，否则为 0
int n,m; //二分图中两集合 x 和 y 中点的数目
```

```
bool check(int t){
    for(int i=1;i<=m;i++){
        if(!used[i] && connect[t][i]){
            used[i]=true;
            if(!link[i] || check(link[i])){
                link[i]=t;
                return true;
            }
        }
    }
    return false;
}
```

```
int Max(){
    int ans=0;
    for(int i=1;i<=n;i++){
        memset(used,0,sizeof(used));
        if(check(i)) ans++;
    }
    return ans;
}
```

理解这段代码的难点在于 `check()` 函数，它用了递归，一般读这段代码，均需要几小时乃至若干天的工夫才会恍然大悟。我不想再写一段说明，给这段代码的理解添乱了，下面仅引用网上最常见的 3 个解说版本。仅供帮助理解，不保证完全的正确性。**首先是 1 号版本：**

引用内容

令 $g = (x, *, y)$ 是一个二分图，其中 $x = \{x_1, x_2, \dots\}$, $y = \{y_1, y_2, \dots\}$. 令 m 为 g 中的任意匹配。

1. 将 x 的所有不与 m 的边关联的顶点表上 γ ，并称所有的顶点为未扫描的。转到 2。
2. 如果在上一步没有新的标记加到 x 的顶点上，则停，否则，转 3。
3. 当存在 x 被标记但未被扫描的顶点时，选择一个被标记但未被扫描的 x 的顶点，比如 x_i ，用 (x_i) 标记 y 的所有顶点，这些顶点被不属于 m 且尚未标记的边连到 x_i 。现在顶点 x_i 是被扫描的。如果不存在被标记但未被扫描的顶点，转 4。
4. 如果在步骤 3 没有新的标记被标记到 y 的顶点上，则停，否则转 5。
5. 当存在 y 被标记但未被扫描的顶点时。选择 y 的一个被标记但未被扫描的顶点，比如 y_j ，用 (y_j) 标记 x 的顶点，这些顶点被属于 m 且尚未标记的边连到 y_j 。现在，顶点 y_j 是被扫描的。如果不存在被标记但未被扫描的顶点则转到 2。由于每一个顶点最多被标记一次且由于每一个顶点最多被扫描一次，本匹配算法在有限步内终止。

本文来自：中国自学编程网 (www.zxhc.cn) 详细出处参考：<http://www.zxhc.cn/html/sjksl/240824324400.htm>

对理解有帮助吗？再来看看 2 号版本：

引用内容

算法的思路是不停的找增广轨,并增加匹配的个数,增广轨顾名思义是指一条可以使匹配数变多的路径,在匹配问题中,增广轨的表现形式是一条"交错轨",也就是说这条由图的边组成的路径,它的第一条边是目前还没有参与匹配的,第二条边参与了匹配,第三条边没有..最后一条边没有参与匹配,并且始点和终点还没有被选择过.这样交错进行,显然他有奇数条边.那么对于这样一条路径,我们可以将第一条边改为已匹配,第二条边改为未匹配...以此类推.也就是将所有的边进行"反色",容易发现这样修改以后,匹配仍然是合法的,但是匹配数增加了一对.另外,单独的一条连接两个未匹配点的边显然也是交错轨.可以证明,当不能再找到增广轨时,就得到了一个最大匹配.这也就是匈牙利算法的思路.

似乎理解 2 号版本需要一些对“增广轨”（有时称“增广路”）一词的理解。最后，看看 **Matrix67** 大牛写的 3 号版本：

引用内容


研究了几个小时，终于明白了。说穿了，就是你从二分图中找出一条路径来，让路径的起点和终点都是还没有匹配过的点，并且路径经过的连线是一条没被匹配、一条已经匹配过，再下一条又没匹配这样交替地出现。找到这样的路径后，显然路径里没被匹配的连线比已经匹配了的连线多一条，于是修改匹配图，把路径里所有匹配过的连线去掉匹配关系，把没有匹配的连线变成匹配的，这样匹配数就比原来多 1 个。不断执行上述操作，直到找不到这样的路径为止。

利用二分图最大匹配求解的两道OI题

读到这里，可能并不容易明白二分图的最大匹配有什么现实意义，所以我举出两道例题来介绍。提供这两道题目的完整资料，Cena评测包，里面含有标程、数据（其中一道题数据太大仅提供生成器，数据没有显式的输出答案，我写的SpecialJudge里面现场根据输入数据计算），解题报告就不需要了，因为两道的标程几乎都直接引用上面的Hungary程序，非常容易读懂。点击下面的文字下载：

 [点击下载此文件](#)

第一道题目是这样的：

 引用内容

题目 1 棋盘问题(chess)

(原题摘自<组合数学>，题目数据由 Fengzee 实现)

源程序名 chess(.c/.cpp/.pas)

输入文件 chess.in

输出文件 chess.out

时间限制 1s

内存限制 10MB

考虑一个 $M \times N$ 的矩形棋盘，其中有若干点不允许放置棋子，其它点可以放置。为了避免棋子间互相攻击，我们约定，任何两枚棋子不可以出现在同一行或同一列。现给出这个棋盘上允许或不允许放置棋子的情况，请求出最多可以放置多少枚棋子。

输入文件包含两部分。第一行为两个正整数 M, N ，表示棋盘有 M 行 N 列；以下是一个 $M \times N$ 的 0-1 矩阵，0 表示不允许放置棋子，1 表示允许放置。

输出文件仅一个整数，为既定要求下最多的棋子个数。

数据规模

$2 \leq M, N \leq 1000$

输入示例

3

1 0 0

1 1 1

0 1 1

输出示例

3

还有一道题：

 引用内容

题目 2 寻找代表(unique)

(原题摘自 Matrix67 举办的“Maxwell 杯内部模拟赛”)

源程序名 unique(.c/.cpp/.pas)

输入文件 unique.in

输出文件 unique.out

时间限制 1s

内存限制 10MB

问题描述

八中一共有 n 个社团，分别用 1 到 n 编号。

八中一共有 m 个人，分别用 1 到 m 编号。每个人可以参加一个或多个社团，也可以不参加任何社团。

每个社团都需要选一个代表。我们希望更多的人能够成为代表。

输入数据

第一行输入两个数 n 和 m 。

以下 n 行每行若干个数，这些数都是不超过 m 的正整数。其中第 i 行的数表示社团 i 的全部成员。每行用一个 0 结束。

输出数据

输出最多的能够成为代表的人数。

样例输入

4 4

1 2 0

1 2 0

1 2 0

1 2 3 4 0

样例输出

3

数据范围

$1 \leq n, m \leq 200$

第八部分 高精度运算

这部分我不想做原理讲解了。理解高精度的原理是很重要的，然而对于实际应用来说，更多的是背诵，熟练背诵。下面的程序是我已经烂熟于心的，看起来有点长，但我背的时候没有花太长时间，请迅速建立背诵代码的信心。

代码引用如下：

```
// BigInt Operators

#include <iostream>
using namespace std;

const int Base = 1000000000;
const int Capacity = 1000;
typedef long long hugeint;

struct BigInt
{
```

```
int Len;
int Data[Capacity];
BigInt() : Len(0)
{
    BigInt(const BigInt &V) : Len(V.Len)
    {
        memcpy(Data, V.Data, Len*sizeof*Data);
    }
    BigInt(int V):
    Len(0)
    {
        for ( ; V>0; V/=Base)
            Data[Len++] = V%Base;
    }
    BigInt &operator = (const BigInt &V)
    {
        Len = V.Len;
        memcpy(Data, V.Data, Len*sizeof*Data);
        return *this;
    }
    int &operator [] (int Index)
    {
        return Data[Index];
    }
    int operator [] (int Index) const
    {
        return Data[Index];
    }
};
```

```
int compare (const BigInt &A, const BigInt &B)
```



```
{
    if (A.Len!=B.Len)
        return A.Len>B.Len ? 1:-1;
    int i;
    for (i=A.Len-1; i>=0 && A[i]==B[i]; i--);
    if (i<0)
        return 0;
    return A[i]>B[i] ? 1:-1;
}
```

BigInt operator + (const BigInt &A, const BigInt &B)

```
{
    int i, Carry = 0;
    BigInt R;
    for (i=0; i<A.Len || i<B.Len || Carry>0; i++)
    {
        if (i<A.Len) Carry += A[i];
        if (i<B.Len) Carry += B[i];
        R[i] = Carry%Base;
        Carry /= Base;
    }
    R.Len = i;
    return R;
}
```

BigInt operator - (const BigInt &A, const BigInt &B)

```
{
    int i, Carry = 0;
    BigInt R;
    R.Len = A.Len;
    for (i=0; i<R.Len; i++)
```

```
{
    R[i] = A[i]-Carry;
    if (i<B.Len) R[i] -= B[i];
    if (R[i]<0) Carry = 1, R[i] += Base;
    else Carry = 0;
}
while (R.Len>0 && R[R.Len-1]==0)
    R.Len--;
return R;
}

BigInt operator * (const BigInt &A, const int &B)
{
    int i;
    hugeint Carry = 0;
    BigInt R;
    for (i=0; i<A.Len || Carry>0; i++)
    {
        if (i<A.Len) Carry += hugeint(A[i])*B;
        R[i] = Carry%Base;
        Carry /= Base;
    }
    R.Len = i;
    return R;
}

istream &operator >> (istream &In, BigInt &V)
{
    char Ch;
    for (V=0; In>>Ch; )
    {
```

```
V = V*10 + (Ch-'0');
if (In.peek()<=' ') break;
}
return In;
}

ostream &operator << (ostream &Out, const BigInt &V)
{
    int i;
    Out << (V.Len==0 ? 0:V[V.Len-1]);
    for (i=V.Len-2; i>=0; i--)
        for (int j=Base/10; j>0; j/=10)
            Out << V[i]/j%10;
    return Out;
}

int main()
{
    BigInt A, B;
    cin >> A >> B;
    cout << A+B << endl;
    if (compare(A, B)>0) cout << A-B <<endl;
    else cout << B-A <<endl;
    cout << A*324423 << endl << B*328942937 << endl;
    system("pause");
    return 0;
}
```

最后的 main()函数是用来说明使用方法的。

第九部分 推荐一些有价值的OI题

MATRIX67 递推专项训练：DP思维训练的首选

这是我曾经做过的一套递推专项训练题目，感觉质量很好，建议在 90 分钟之内解决所有问题，题目引用如下：

问题一：

在所有的 N 位数中，有多少个数中有偶数个数字 3？

样例输入：2

样例输出：73

问题二：

用 1×1 和 2×2 的磁砖不重叠地铺满 $N \times 3$ 的地板，共有多少种方案？注意，旋转后相同的方案也算新的方案（这句话原题没有，是我补上的）。

样例输入：2

样例输出：3

问题三：

从原点出发，一步只能向右走、向上走或向左走。恰好走 N 步且不经过已走的点共有多少种走法？

样例输入：2

样例输出：7

问题四：

圆周上有 N 个点。连接任意多条（可能是 0 条）不相交的弦（共用端点也算相交）共有多少种方案？

样例输入：4

样例输出：9

问题五：

在网格中取一个 $N \times 1$ 的矩形，并把它当作一个无向图。这个图有 $2(N+1)$ 个顶点，有 $3(N-1)+4$ 条边。这个图有多少个生成树？

样例输入：1

样例输出：4

以下说明中的“?”代表题目的编号

源程序名

problem?.(pas/c/cpp)

输入数据

从 problem?.in 中读入一个数 N 。 $1 \leq N \leq 1000$ 。

输出数据

将答案输出到 problem?.out 中。由于结果可能很大，你只需要输出这个答案 mod 12345 的值。

时间限制

各测试点 1 秒

空间限制

你的程序将分配 64M 的运行空间

后记

我最初只想写个把程序攒到一起的“程序整理”，现在看起来接近一本书了。本文中大部分文字都出自我的笔下，只是由于它们的写作时间不同，互相引用，因此其中的关联性可能稍差。我为了用较短的时间完成整理，只能采用这种方法，请原谅。阅读时，请忽略一些难以理解的语句。本文的引文中有一些链接，转换为 PDF 格式时可能失效，文中我并没有完全标注，故在此声明。

本文收录的绝大多数程序，都是我在最近三天之内重新写成并测试通过的。工作量较大，我只使用了一组测试数据，因此我不能够完全保证所有程序没有漏洞。如果发现错误，请提示。本文自由使用、更改、传播。

冯子力(Fengzee)

2008 年 3 月 1 日