计算机科学与技术系

修订版

ICPC 培训讲义

算法与程序设计

目　　录

第一章　STL 简介

一、引　　言

标准模板库（Standard Template Library, 简称 STL）是 ANSI/ISO C++语言的库的一个主要组成部分。它最初是惠普实验室（Hewlett-Packard Labs）开发成功的。STL 的全称是 Standard Template Library 即标准模板库，它包括了通用数据结构和基于这些结构的算法，向外提供统一标准的公共接口，使得使用 STL 方便、快捷地建立应用程序。STL 最初是由惠普实验室的 Alexander Stepanov 和 Meng Lee 开发的，是他们在一般化编程领域研究成果的结晶，并且 David Musser 在这个领域也有重大贡献。STL 已成为 ANSI/ISO C++草案标准的一部分，使用越来越广泛。

这里我们只是简单的向大家介绍 STL 库中的各种资源的使用，通过一些短小的程序，希望能达到让大家在一定程度上掌握 STL 的基本知识，并能通过使用 STL 库开发出简洁高效的 C++代码，这是我们编写这一参考资料的主要目的。凡是学过 C++的都应该了解模板，通过模板类和模板函数，可以很方便地实现代码的复用和通用代码的建立。STL 正是基于模板建立的。比较常用的三个组件是：容器(container，常用模板化数据类型)、迭代器(interator)与算法(algorithm)。

在设计算法和编写程序的过程中，我们要经常用到链表、堆栈、队列、优先队列、集合等基础数据结构，如果每次我们都自己手动建立这些数据结构的类，那将使我们的效率大打折扣，STL 中包含了一个标准化的模板化对象容器库，有了它我们能很方便的实现上述数据结构，并且能得到更高质量的程序。

书中的所使用的代码，参考于惠普实验室的免费开放的 STL 原码，以及 SGI STL 中的部分代码。如果读者要上机运行书中的代码，以下编译器可供选择，

Windows 9.x /2000/NT/XP 环境中请选用：

Microsoft Visual C++;
IBM Visual Age C++;
C++ Builder 6.0;(版本 5.5 的行编译器可在 Borland 公司的网站上免费下载。)
Dev C++ 2.9.6 ; (该 IDE 中的编译器是 g++的 Windows 移植版本。)

Linux Redhat 8.x/9.x 环境中请选用:

g++ 2.9.5/2.9.6/3.2;
Kylix 3.0 (同时支持 Delphi 和 C++)

最佳推荐为 C++ Builder 6.0。

二、STL 组成结构

库（Library）是系列程序组件的集合，它们可以在不同的程序中重复使用，如 iostream 和 stdlib.h 、stdio.h，都是在使用 C++/C 中提供的函数及类库。通过使用模板，C++建立了一个功能强大的 STL 库。STL 的一个重大成就在于，它提供了相当多的有用算法。它是在一个有效的框架中完成这些算法的。STL 提供了大约 100 个实现算法的模板函数。熟悉了 STL 之后，你就会发现：以前所写的许多“有趣功能”的代码现在只需要用短短几行主能完全替换了。“*The Art of Computer Programming*”书中强调

算法 + 数据结构 = 程序

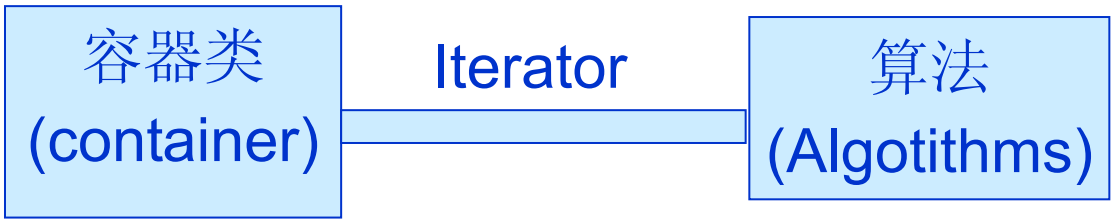
在过去，一般来说程序皆由算法加数据结构，互相配合、一起工作，完成程序的功能。但如此一来便将数据结构与算法绑再一起，不能分离，缺乏弹性。Stepanov 先生便针对此问题，采通用型设计的思维，使得程式师可以

- 以效率高的算法解决问题，此算法可处理各种数据结构即算法与数据结构互为独立
- 各种数据结构使用一致的接口(interface)，让各种算法可以透过此接口处理各种数据结构

如何做到这点？下面就简单介绍一下 STL 的组成。STL 有五个组成部分：容器、迭代器、算法、适配器、函数对象。

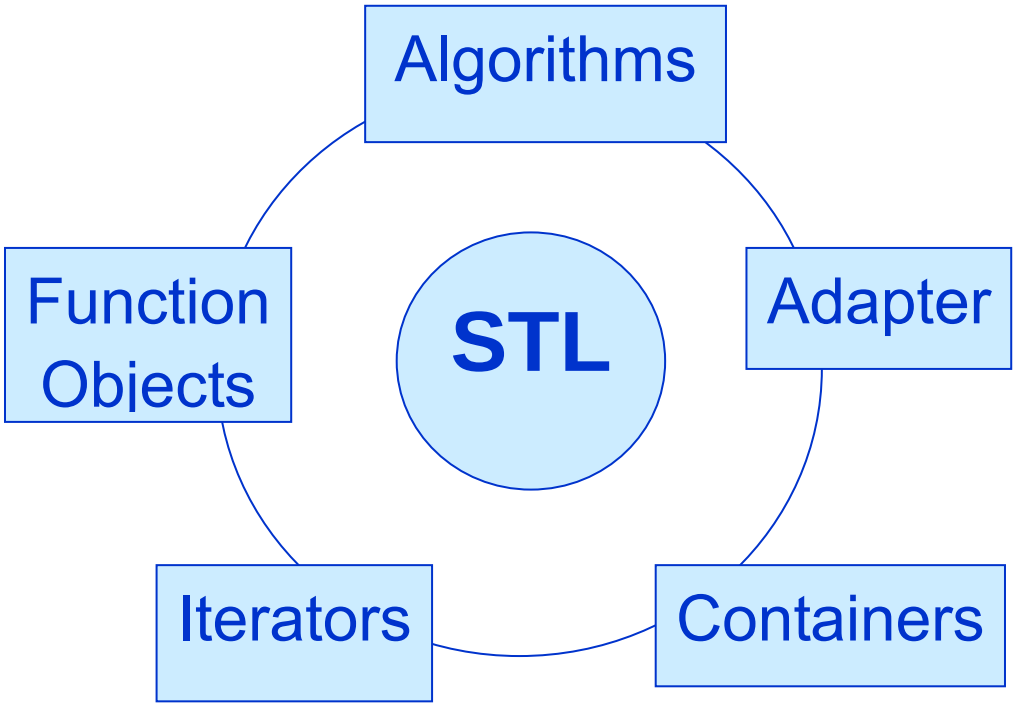
- **容器** (container)－用来储存其他物件
- **迭代器**(iterator)－好比传统 C 语言的指针，可藉之来处理容器对象
- **算法**(algorithm)－算法通过迭代器来操作容器对象
- **适配器**(adaptor)－利用基础容器对象，加以包装，改变其接口，以适应另一种需求
- **函数对象**(function object)－为 STL 中较低阶的对象，用来代替传统的**函数指针**(function pointer)

STL 的简单结构可用下图表示：其中容器类表示通常意义上的数据结构，和算法通过迭代器连接起来，做到上面的两点。



实际上，为了做到数据结构和算法之间的独立并保持相当的灵活性，STL 的实际结构

要比上图表示复杂的多，下面给出 STL 的实际组成图。



三、STL 的应用

标准的 C++库包含一个 INCLUDE 库和一个 LIB 库。Include 库里的函数和类都是以源文件的形式给出的，用户可以查看它的源代码，而 LIB 库中则以二进制的形式给出。STL 都是以源码的形式给出的，各个编译器有不同的版本。VC++中的 STL 可读性很差，变量和内部函数名完全没有真实意义，如果要看源码的话建议看其它编译器的版本。

在做题目的时候可以选用 VC++或 DEV C++，但是 VC++用的不是 g++内核，而现在的 JUDER 一般都用 g++来编译测试，所以从适应比赛角度来说用 DEV C++比较合适，它也用很好的编辑环境，而且比较小，容易存储。不过如果严格按照标准 C++的格式来编制程序的话，一般不会遇到因编译器而出错的问题。

现在大多数编译器都包含新旧两个版本的头文件，旧版本的头文件是以.h 为扩展名的，新版本的头文件则不带扩展名，两者都可以用记事本打开来查看。例如在 C/C++中使用标准输入输出流为：

```
#include <iostream.h>
#include <stdio.h>
```

而使用新版头文件则为：

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

上面的 using namespace std;一句是使用名字域 std，在 std 声明了所有 stl 中的接口，在

文件中声明一次即可。此外，新旧文件的文件名不尽相同，例如 math.h 在新版本中为 cmath，一般情况下名字不同时在原文件名前加 c 即可。

在介绍 STL 具体的数据结构之前，我们必须先了解一种叫迭代器的结构。用户不必在意它的内部实现，但在使用 STL 的时候却无论如何都不能忽略它的存在。迭代器(Iterator)可以看成是一种特殊的指针，即一个指针类，它包含一类对象的指针和可以对这种指针进行的操作，也就是说迭代器保存所操作的特定容器需要的状态信息。例如一个数组 int array[N]的迭代器可以为 int *p＝array，现在 p 指向的是数组的首地址，我们可以对 p 进行加减操作来改变它所指的元素，如：p++是将指针 p 的值直接加上整形的字长而指向数组的下一个元素。但是对于链表来说，如果 pit 是它的迭代器并且指向链表中的某个元素 T，如果要使 pit 指向链表的下一个元素则必须将 pit 内部用于定位的指针指向 T->next。其实我们可以像操作指针一样方便的操作迭代器，所不同地就是迭代器的功能比较完善，限制了对一些特殊结构类型的操作，包括随机迭代器，顺序迭代器(重载++)，双向迭代器(重载++和--)等。第一类容器提供成员函数 begin()和 end()来返回容器的首尾迭代器。例如：

```
#include <iostream>
#include <vector>
using namespace std;
vector<int> vint;
int main()
{
   int i;
   for(i=0;i<10;i++)
        vint.push_back(i);
   vector<int>::iterator pit;
   for(pit=vint.begin();pit!=vint.end();pit++)
         cout<<(*pit)<<“ ”;
   return 0;
}
```

上述代码只是让读者了解一下迭代器的使用，我们将在以后详细地介绍中具体地分析各个数据结构迭代器的用法。

请看下面 STL 中的一段代码：

```
template<class T, class U>
      struct pair {
      typedef T first_type;
      typedef U second_type
      T first;
      U second;
      pair();
      pair(const T& x, const U& y);
      template<class V, class W>
            pair(const pair<V, W>& pr);
```

};

STL 中最简单地数据结构应属于<utility>中定义地 pair<TYPE,TYPE>,它包含两个内部变量 first,second，用来将两个相关的数据配对，并重载了比较大小、判等、转化等函数，使用它可以方便地实例化自己地结构，如要建立一个坐标的数据结构则可以：

```
#include <utility>
using namespace std;
typedef pair<int,int> POINT;
```

这样在下边的代码中我们就可以使用 POINT 这个结构了。如果要使用它的判断函数，则所传递的类型必须已经重载了比较运算符或者使系统自定义类型。

例如：

```
#include <iostream>
#include <utility>

using namespace std;
typedef pair<int,int> POINT;

int main()
{
    POINT p1,p2;
    cin>>p1.first>>p1.second>>p2.first>>p2.second;
    if(p1==p2)cout<< “Equal!”<<endl;
    else cout<< “Not Equal”<<endl;
    return 0;
}
```

此外，utility 中还有一个 pair<T, U> make_pair(const T &x, const U &y)函数，用来将两个数据组成一个 pair 对象返回。

C++标准模板库提供三种顺序容器：vector、list 和 deque。Vector 类和 deque 类都是基于数组的，list 类实现链表数据结构。

【vector 类】

vector 类是一个线性数组类，内部用线性空间来存放元素，内部有两个不同的变量来标识当前元素个数_size 和当前共分配的空间数_capacity，通常情况下_size 要小于_capacity，这样当在数组中添加元素时就不用每次都重新分配空间，从而提高了时间效率，直到分配的空间使用完时才重新分配。至于每次分配空间的大小则与数组元素的个数有关，可以用 capacity()函数得到。Vector 类元素最多一般可以为 1073741823 个。Vector 类重载了下标运算符，可以像数组一样访问容器中的元素，而且 vector 可以相互赋值，这是 C 语言的数组所不能实现的。Vector 不进行下标检查，但是它通过成员函数 at 提供这个功能。使用 vector 可以方便的实现线性多维数组。例如要分别建立坐标的一维和二维数组：

```
#include <utility>
#include <vector>
using namespace std;
typedef pair<int,int>POINT；
typedef vector<POINT> Array1D;
typedef vector<vector<POINT> > Array2D;
```

在上面我们定义了一维坐标类 Array1D 和二维坐标类 Array2D，值得注意的是在定义 Array2D 的时候，后面的尖括号中间要用空格隔开，否则编译器可能误认为是移位运算符而报错。Vector 类常用的接口如下：

函数名	返回值	参数	功能
at	元 素 的 引用	要取元素的位置(int pos)	如果该位置元素存在则返回它的引用，否则进行一场处理
back	元 素 的 引用	无	返回容器最后一个元素的引用，如果容器为空则出错
begin	迭代器	无	取得元素的首迭代器
end	迭代器	无	取得元素的尾迭代器
clear	无	无	将容器清空
empty	bool 值	无	如果容器为空则为真，否则为假
erase	迭代器	要删除元素的迭代器 要删除容器段的首尾迭代器	第一个版本删除指定的元素 第二个版本删除指定的容器段 不存在则出错
front	元 素 的 引用	无	返回首元素的引用，为空则出错
insert	无	iterator it, const T& x 2、iterator it, const_iterator first, const_iterator last	在 it 这个位置插入元素 T 从 it 位置开始，依次插入 first 到 last 所指的元素
pop_back	无	无	删除最后一个元素
push_back	无	const T& x	从最后插入一个元素
resize	无	size_type n	将容器大小重设置为 n
size	int	无	返回容器中元素的个数

请看下面的实例程序：

```
#include <iostream>
#include <vector>
using namespace std;
typedef vector<int> Array1D;
```

```
typedef vector<vector<int> > Array2D;
int main()
{
	int i,j;
	Array1D sz;
	for(i=0;i<10;i++)
	   sz.push_back(i);
	for(i=0;i<10;i++)
	    cout<<sz[i]<<" ";
	cout<<"\n"<<sz.front()<<"    "<<sz.back()<<endl;
	sz.erase(sz.begin()+3);
	for(i=0;i<sz.size();i++)
	   cout<<sz[i]<<" ";
	cout<<endl;
	sz.erase(sz.begin()+2,sz.begin()+5);
	for(i=0;i<sz.size();i++)
	 cout<<sz[i]<<" ";
	 cout<<endl;
	sz.erase(sz.begin(),sz.end());//sz.clear();
	Array2D tw;
	tw.resize(5);
	for(i=0;i<5;i++)
	   tw[i].resize(5);
	for(i=0;i<5;i++)
	   for(j=0;j<5;j++)
	      tw[i][j]=i*5+j;
	for(i=0;i<tw.size();i++)
	{
	   for(j=0;j<tw[i].size();j++)
	      cout<<tw[i][j]<<" ";
	   cout<<endl;
	}
	system("PAUSE");    //press any key to continue.....
	return 0;
}
```

对应的输出应当为：

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

5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
20 21 22 23 24

请按任意键继续. . .

读者对照输出可体会各函数的用法

【list 类】

lsit 顺序容器提供在容器中任意位置进行插入与删除操作的有效实现方法。如果打多说插入和删除发生在容器首尾，在双头队列更适合。List 是双链表，每个结点包含数据和指向上一个结点、下一个结点的指针，支持双向迭代器。List 与一般的容器相比增加的成员函数有 splice、push_front、remove、unique、merge、reverse、sort 等，使用的时候要包含<list>头文件。

splice 相当于 insert，只不过是插入后将另一个链表中的该元素移除而已。remove 是删除指定值的结点。unique 有两个版本，第一个版本是 void unique()，将每个元素同它的前面一个元素比较，如果相等则 remove 该元素。第二各版本是 void unique(not_equal_to<T> pr)，用函数 pr 的返回值来作为判等的条件。

函数接口声明如下：

```
template<class T, class A = allocator<T> >
    class list {
public:
    iterator begin();
    const_iterator begin() const;
    iterator end();
    iterator end() const;
    void resize(size_type n, T x = T());
    size_type size() const;
    bool empty() const;
    reference front();
    const_reference front() const;
    reference back();
    const_reference back() const;
    void push_front(const T& x);
    void pop_front();
    void push_back(const T& x);
    void pop_back();
    iterator insert(iterator it, const T& x = T());
    void insert(iterator it, size_type n, const T& x);
    void insert(iterator it,
        const_iterator first, const_iterator last);
```

```
        void insert(iterator it,
              const T *first, const T *last);
        iterator erase(iterator it);
        iterator erase(iterator first, iterator last);
        void clear();
        void swap(list x);
        void splice(iterator it, list& x);
        void splice(iterator it, list& x, iterator first);
        void splice(iterator it, list& x, iterator first, iterator last);
        void remove(const T& x);
        void remove_if(binder2nd<not_equal_to<T> > pr);
        void unique();
        void unique(not_equal_to<T> pr);
        void merge(list& x);
        void merge(list& x, greater<T> pr);
        void sort();
        template<class Pred>
              void sort(greater<T> pr);
        void reverse();
    };
```

sort 函数有两个版本，第一个按照 operator<排序，另一个则用 pr 函数的返回值作为排序的条件。

请看下面的实例程序：

```
#include <iostream>
#include <list>
//#include <algorithm>

using namespace std;

typedef list<int> lint;
int value[]={2,4,5,7,9,12};

void print(lint &l)
{
      int i;
      lint::iterator pit;
      for(pit=l.begin();pit!=l.end();pit++)
         cout<<(*pit)<<" ";
      cout<<endl;
      /*ostream_iterator<int> output(cout," ");
      copy(l.begin(),l.end(),output);
```

```
        */
}
bool sortsp(int v1,int v2)
{
        return v1>v2;
}

int main()
{
        lint firstl,secondl;
        firstl.push_front(1);
        firstl.push_front(2);
        firstl.push_back(4);
        firstl.push_back(3);
        cout<<"The first list contains: ";
        print(firstl);
        firstl.sort();
        cout<<"After sort the first list: ";
        print(firstl);

        secondl.insert(secondl.begin(),value,value+6);
        cout<<"The second list contains: ";
        print(secondl);
        firstl.splice(firstl.end(),secondl);
        cout<<"After splice the first list contians: ";
        print(firstl);
        firstl.sort(sortsp);
        cout<<"After sort with compare function: ";
        print(firstl);

        firstl.unique();
        cout<<"After unique the first list: ";
        print(firstl);

        firstl.remove(9);
        cout<<"After remove value 9 : " ;
        print(firstl);

        secondl.insert(secondl.end(),value,value+5);
        firstl.swap(secondl);
        cout<<"After swap with the second list: ";
        print(firstl);
```

```
    system("PAUSE");//press any key to contineu... Used in dev c++
    return 0;
}
```

运行结果是：

```
The first list contains: 2 1 4 3
After sort the first list: 1 2 3 4
The second list contains: 2 4 5 7 9 12
After splice the first list contians: 1 2 3 4 2 4 5 7 9 12
After sort with compare function: 12 9 7 5 4 4 3 2 2 1
After unique the first list: 12 9 7 5 4 3 2 1
After remove value 9 : 12 7 5 4 3 2 1
After swap with the second list: 2 4 5 7 9

请按任意键继续. . .
```

以上介绍了两种容器的用法，其他的第一类容器与上面的类似，只是特性和接口函数上稍微有一些变化，下面将其他容器的接口函数给出，读者可自行编制程序来体会它们的用法。

【deque 类】

```
template<class T, class A = allocator<T> >
    class deque {
public:
    iterator begin();
    const_iterator begin() const;
    iterator end();
    iterator end() const;
    void resize(size_type n, T x = T());
    size_type size() const;
    bool empty() const;
    reference at(size_type pos);
    const_reference at(size_type pos) const;
    reference operator[](size_type pos);
    const_reference operator[](size_type pos);
    reference front();
    const_reference front() const;
    reference back();
    const_reference back() const;
    void push_front(const T& x);
    void pop_front();
    void push_back(const T& x);
    void pop_back();
```

```
        iterator insert(iterator it, const T& x = T());
        void insert(iterator it, size_type n, const T& x);
        void insert(iterator it,
              const_iterator first, const_iterator last);
        iterator erase(iterator it);
        iterator erase(iterator first, iterator last);
        void clear();
        void swap(deque x);
        };
```

【stack 类】

```
template<class T,
        class Cont = deque<T> >
        class stack {
public:
        bool empty() const;
        size_type size() const;
        value_type& top();
        const value_type& top() const;
        void push(const value_type& x);
        void pop();
        };
```

【map 类】

map 关联容器用于快速存储和读取关键字与相关值，内部是红黑树结构。map 不允许重复关键字，因此一个关键字只可以对应一个值，成为一对一映射。使用的时候要包含<map>头文件。

```
template<class Key, class T, class Pred = less<Key>, class A = allocator<T> >
        class map {
public:
          iterator begin();
        const_iterator begin() const;
        iterator end();
        iterator end() const;
        size_type size() const;
       bool empty() const;
        A::reference operator[](const Key& key);
        pair<iterator, bool> insert(const value_type& x);
        iterator insert(iterator it, const value_type& x);
        void insert(const value_type *first, const value_type *last);
        iterator erase(iterator it);
        iterator erase(iterator first, iterator last);
        size_type erase(const Key& key);
```

```
    void clear();
    void swap(map x);
    key_compare key_comp() const;
    value_compare value_comp() const;
    iterator find(const Key& key);
    const_iterator find(const Key& key) const;
    size_type count(const Key& key) const;
    iterator lower_bound(const Key& key);
    const_iterator lower_bound(const Key& key) const;
    iterator upper_bound(const Key& key);
    const_iterator upper_bound(const Key& key) const;
    pair<iterator, iterator> equal_range(const Key& key);
    pair<const_iterator, const_iterator>
        equal_range(const Key& key) const;
    };
```

下面是实例程序：

```
#include <iostream>
#include <map>
using namespace std;
int main()
{
    typedef map<int, double,less<int> >mid;
    mid pairs;
    pairs.insert(mid::value_type(15,2.7));
    pairs.insert(mid::value_type(30,111.11));
    pairs.insert(mid::value_type(15,2.7));
    pairs.insert(mid::value_type(5,1010.1));
    pairs.insert(mid::value_type(10,22.22));
    pairs.insert(mid::value_type(25,33.33));
    pairs.insert(mid::value_type(5,77.22));
    pairs.insert(mid::value_type(20,6.7));
    pairs.insert(mid::value_type(15,2.7));    //ingore for key has exsit
    mid::iterator iter;
    for(iter=pairs.begin();iter!=pairs.end();iter++)
      cout<<iter->first<<'\t'<<iter->second<<endl;
    pairs[25]=9999.99;
    pairs[40]=8765.43;
    cout<<"\nAfter subscript operations, pairs contains: "<<"\nkey\tvalue\n";
    for(iter=pairs.begin();iter!=pairs.end();iter++)
      cout<<iter->first<<"\t"<<iter->second<<"\t"<<endl;
    cout<<endl;
```

```
    system("PAUSE");
    return 0;
}
```

输出为：

```
5       1010.1
10      22.22
15      2.7
20      6.7
25      33.33
30      111.11
After subscript operations, pairs contains:
key     value
5       1010.1
10      22.22
15      2.7
20      6.7
25      9999.99
30      111.11
40      8765.43
请按任意键继续. . .
```

需要注意的是，在使用 map 的查抄(find)函数的时候，如果该关键字没找到则插入，即如果不存在的话查找与插入同步完成。

【算法 Algorithm】

STL 的数据结构很大程度上是为通用算法的建立而设计的。STL 将具有很强通用性的算法如查找、排序、拷贝、反序、替换等许多基本算法写成通用函数，提供了方便、简捷的接口。例如，你不必花费好大力气去写一个排序算法还要校验它的正确性，你只要调用 STL 中的 sort 函数即可。

Sort

```
template<class RanIt>
    void sort(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void sort(RanIt first, RanIt last, Pred pr);
```

sort 函数的第一个版本是将 operator<排序，即由小到大排序。第二个版本是以所传递函数的返回值来确定顺序。前两个参数是要排序结构的首元素和尾元素的迭代器。如果要将数组 int tt[20]前十个数由小到大排序则为：

```
#include <algorithm>
#include <iostream>

using namespace std;
```

```
int tt[20];
int main()
{
   int i;
   for(i=0;i<20;i++)
         tt[i]=20-i;
   sort(tt,tt+10);
   for(i=0;i<10;i++)
    cout<<tt[i]<<“ ”;
   return 0;
}
```

如果要将 vector<int> gg;由大到小排序：

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;
vector<int> gg;
bool sortspecial(int v1,int v2)
{
    return v1>v2;
}
int main()
{
   int i;
   for(i=0;i<20;i++)
      gg.push_back(i);
   sort(gg.begin(),gg.end(),sortspecial);
   for(i=0;i<20;i++)
      cout<<gg[i]<<“ ”;
}
```

要注意的是这个排序算法是不稳定的，即具有相同键值的元素的最终位置于开始的相对位置并不一定相同，例如<7,6>、<7,5>，如果按前一个值排序则最终两个 pair 的最终位置无法确定。如果需要保持稳定的话则可以使用 stable_sort 函数。

同时在 stdlib.h 文件中由一个快排函数 qsort，原型为：

void qsort(void *base, size_t num, size_t width, int (__cdecl *compare)(const void *elem1, const void *elem2));

用法如下：

```
#include <iostream>
```

```
#include <cstdlib>
using namespace std;
int a[9]={1,2,4,5,8,12,23,4,3};
int compare( const void *arg1, const void *arg2 );
int main( )
{
     int i;
     qsort( (void *)a, 9, sizeof(int), compare );
     /* Output sorted list: */
   for(i=0;i<9;i++)
     cout<<a[i]<<" ";
   cout<<endl;
   system("PAUSE");
   return 0;
}
int compare( const void *arg1, const void *arg2 )
{
     return *((int *)arg2)-*((int *)arg1);
}
```

输出为：

23 12 8 5 4 4 3 2 1

请按任意键继续. . .

查找:

有两个版本，一个直接查找特定值的元素，另一个按照函数的返回值来查找。

1、template<class InIt, class T>
    InIt find(InIt first, InIt last, const T& val);

```
#include <algorithm>
#include <iostream>
using namespace std;
int main()
{
      const int ARRAY_SIZE = 8 ;
      int IntArray[ARRAY_SIZE] = { 1, 2, 3, 4, 4, 5, 6, 7 } ;
      int *location ;      // stores the position of the first
                                 // matching element.
      int i ;
      int value = 4 ;
      // print content of IntArray
      cout << "IntArray { " ;
      for(i = 0; i < ARRAY_SIZE; i++)
```

```
        cout << IntArray[i] << ", " ;
    cout << " }" << endl ;
    // Find the first element in the range [first, last + 1)
    // that matches value.
    location = find(IntArray, IntArray + ARRAY_SIZE, value) ;
    //print the matching element if any was found
    if (location != IntArray + ARRAY_SIZE)    // matching element found
        cout << "First element that matches " << value
            << " is at location " << location - IntArray << endl;
    else                                                    // no matching element was
                                                            // found
        cout << "The sequence does not contain any elements"
            << " with value " << value << endl ;
    system("PAUSE");
    return 0;
}
```

Program Output is:

```
IntArray { 1, 2, 3, 4, 4, 5, 6, 7,    }
First element that matches 4 is at location 3
```

2、template<class InIt, class Pred>
    InIt find_if(InIt first, InIt last, Pred pr);

```
#include <algorithm>
#include <iostream>
using namespace std;
// returns true if n is an odd number
int IsOdd( int n)
{
    return n % 2 ;
}
int main()
{
    const int ARRAY_SIZE = 8 ;
    int IntArray[ARRAY_SIZE] = { 1, 2, 3, 4, 4, 5, 6, 7 } ;
    int *location ;      // stores the position of the first
                         // element that is an odd number
    int i ;
        // print content of IntArray
    cout << "IntArray { " ;
    for(i = 0; i < ARRAY_SIZE; i++)
        cout << IntArray[i] << ", " ;
    cout << " }" << endl ;
    // Find the first element in the range [first, last -1 ]
```

```
    // that is an odd number
    location = find_if(IntArray, IntArray + ARRAY_SIZE, IsOdd) ;
    //print the location of the first element
    // that is an odd number
    if (location != IntArray + ARRAY_SIZE)    // first odd element found
        cout << "First odd element " << *location
            << " is at location " << location - IntArray << endl;
    else                // no odd numbers in the range
        cout << "The sequence does not contain any odd numbers"
            << endl ;
    }
Program Output is:
IntArray { 1, 2, 3, 4, 4, 5, 6, 7,    }
First odd element 1 is at location 0
```

现在将经常用到的算法类据如下：

```
namespace std {
template<class InIt, class T>
    InIt find(InIt first, InIt last, const T& val);
template<class InIt, class Pred>
    InIt find_if(InIt first, InIt last, Pred pr);
size_t count(InIt first, InIt last,
        const T& val, Dist& n);
template<class InIt, class Pred, class Dist>
    size_t count_if(InIt first, InIt last,
        Pred pr);
template<class InIt, class OutIt>
    OutIt copy(InIt first, InIt last, OutIt x);
template<class T>
    void swap(T& x, T& y);
template<class FwdIt1, class FwdIt2>
    FwdIt2 swap_ranges(FwdIt1 first, FwdIt1 last, FwdIt2 x);
template<class FwdIt, class T>
    void replace(FwdIt first, FwdIt last,
        const T& vold, const T& vnew);
template<class FwdIt, class Pred, class T>
    void replace_if(FwdIt first, FwdIt last,
        Pred pr, const T& val);
template<class FwdIt, class T>
template<class FwdIt, class Gen>
    void generate(FwdIt first, FwdIt last, Gen g);
template<class OutIt, class Pred, class Gen>
```

```
    void generate_n(OutIt first, Dist n, Gen g);
template<class FwdIt, class T>
    FwdIt remove(FwdIt first, FwdIt last, const T& val);
template<class FwdIt, class Pred>
    FwdIt remove_if(FwdIt first, FwdIt last, Pred pr);
template<class InIt, class OutIt, class T>
    OutIt remove_copy(InIt first, InIt last, OutIt x, const T& val);
template<class InIt, class OutIt, class Pred>
    OutIt remove_copy_if(InIt first, InIt last, OutIt x, Pred pr);
template<class FwdIt>
    FwdIt unique(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt unique(FwdIt first, FwdIt last, Pred pr);
template<class BidIt>
    void reverse(BidIt first, BidIt last);
template<class BidIt, class OutIt>
    OutIt reverse_copy(BidIt first, BidIt last, OutIt x);
template<class BidIt, class Pred>
    BidIt partition(BidIt first, BidIt last, Pred pr);
template<class FwdIt, class Pred>
    FwdIt stable_partition(FwdIt first, FwdIt last, Pred pr);
template<class RanIt>
    void sort(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void sort(RanIt first, RanIt last, Pred pr);
template<class BidIt>
    void stable_sort(BidIt first, BidIt last);
template<class BidIt, class Pred>
    void stable_sort(BidIt first, BidIt last, Pred pr);
template<class RanIt>
    void nth_element(RanIt first, RanIt nth, RanIt last);
template<class FwdIt, class T>
    bool binary_search(FwdIt first, FwdIt last, const T& val);
template<class FwdIt, class T, class Pred>
    bool binary_search(FwdIt first, FwdIt last, const T& val,
        Pred pr);
template<class InIt1, class InIt2, class OutIt>
    OutIt merge(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt, class Pred>
    OutIt merge(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x, Pred pr);
```

```
template<class T>
    const T& max(const T& x, const T& y);
template<class T, class Pred>
    const T& max(const T&    x, const T& y, Pred pr);
template<class T>
    const T& min(const T& x, const T& y);
template<class T, class Pred>
    const T& min(const T& x, const T& y, Pred pr);
    };
```

【字符串】

在做比赛题目的时候，不免碰到关于字符串的问题，在这里就比赛相关的内容简单介绍一下。

C++中的字符串由两种，一种就是来自 C 语言的 C 串，它其实就是以\0 作为结束符的字符数组,毫无安全性可言，对其的操作也只有 string.h 中的 strlen、strcat、strcm 等函数；另一个就是 STL 中定义的功能比较强大的字符串类 string，包含在头文件<string>中。它内部实现了几乎对所有字符串进行的操作，采取的是顺序表存储，同时也可以方便地实现 C 串于 string 类之间地转换。

关于字符串的处理，输入输出是经常碰到的问题。如果输入的字符串是以空格、回车、制表符作为结束符的，可以用标准输入输出流处理(cin/cout)。例如：

```
#include <string>
#include <iostream>
using namespace std;
string str;
char ch[20];
int main()
{
    cin>>str;
    cout<<str<<endl;
    cin>>ch;
    cout<<ch;
    return 0;
}
```

如果希望输入的字符串包含空格等字符，则必须使用另外的处理方式。

如 cin.getline(ch);

getline(cin,str);

特别需要注意的是当输入一个其他类型的数如整数后要用 getline 输入字符串，则必须使用 cin.ignore()；来忽略流中的回车符。如：

```
int testcase;
```

string str;

cin>>testcase;

cin.ignore();

getline(cin,str);

string 类常用的函数有：

append · begin · compare · const_pointer · empty · end · erase · find · find_first_not_of · find_first_of · find_last_not_of · find_last_of · insert · length · operator+= · operator= · operator[] · replace · reserve · resize · size ·· substr · swap

下面是实例程序(ZJU 的 1295 题)：

题目：对于输入给定的字符串将其反序输出。输入是先输入一个整数代表字符串的行数，然后是字符串。

Sample Input：

3
Frankly, I don't think we'll make much
money out of this scheme.
madam I'm adam

Sample Output

hcum ekam ll'ew kniht t'nod I ,ylknarF
.emehcs siht fo tuo yenom
mada m'I madam

参考代码如下：

```
#include<iostream>
#include<string>
using namespace std;
void rever(string s)
{
    string temp;
    int i;
    for(i=s.size()-1;i>=0;i--)
        temp.append(1,s[i]);
    for(i=0;i<temp.size();i++)
        cout<<temp[i];
    cout<<endl;
}
int main()
{
        string instr;
```

```
    int strno=0,i;
    cin>>strno;
    char enter=cin.get();
    for(i=0;i<strno;i++)
      {
        getline(cin,instr);
        rever(instr);
      }

    return 0;
}
```

【Iterator】

首先想说明的是，标题中的这个单词许多字典里可能查不到。计算机词汇中将它译为迭代器。所谓迭代器，它的很多性质是和 C++/C 中的指针是相似，甚至可以说指针就是一种 C++编译器里内置的迭代器。在学习的过程中，可以将它与指针对照起来理解，这样可能会有助于学习。

接口描述：

```
namespace std
{
    struct input_iterator_tag;
    struct output_iterator_tag;
    struct forward_iterator_tag;
    struct bidirectional_iterator_tag;
    struct random_access_iterator_tag;

    //TEMPLATE CLASS
    template <class C, class T, class Dist,
           class Pt, class Rt>
        struct iterator;
    template <class It>
        struct iterator_traits;
    template <class T>
        struct iterator_traits<T *>;
    template <class RanIt>
        class reverse_iterator;
    template <class Cont>
        class back_insert_iterator;
    template <class Cont>
        class front_insert_iterator;
```

```
template <class Cont>
   class insert_iterator;
template <class U, class E, class T,
   class Dist>
   class istream_iterator;
template <class U, class E, class T>
   class ostream_iterator;
template <class E, class T>
   class istreambuf_iterator;
template <class E, class T>
   class ostreambuf_iterator;
   //TEMPLATE FUNCTIONS
template <class RanIt>
   bool operator==(
      const reverse_iterator<RanIt>& lhs,
      const reverse_iterator<RanIt>& rhs);
template <class U, class E, class T,
      class Dist>
   bool operator==(
      const istream_iterator<U,E,T,Dist>& lhs,
      const istream_iterator<U,E,T,Dist>& rhs);
template <class E, class T>
   bool operator==(
      const istreambuf_iterator<E, T>& lhs,
      const istreambuf_iterator<E, T>& rhs);
template <class RanIt>
   bool operator!=(
      const reverse_iterator<RanIt>& lhs,
      const reverse_iterator<RanIt>& rhs);
template <class U, class E, class T,
      class Dist>
   bool operator!=(
      const istream_iterator<U,E,T,Dist>& lhs,
      const istream_iterator<U,E,T,Dist>& rhs);
template <class E, class T>
   bool operator!=(
      const istreambuf_iterator<E, T>& lhs,
      const istreambuf_iterator<E, T>& rhs);
template <class RanIt>
   bool operator< (
      const reverse_iterator<RanIt>& lhs,
      const reverse_iterator<RanIt>& rhs);
```

```
    template <class RanIt>
       bool operator> (
          const reverse_iterator<RanIt>& lhs,
          const reverse_iterator<RanIt>& rhs);
    template <class RanIt>
       bool operator<=(
          const reverse_iterator<RanIt>& lhs,
          const reverse_iterator<RanIt>& rhs);
    template <class RanIt>
       bool operator>=(
          const reverse_iterator<RanIt>& lhs,
          const reverse_iterator<RanIt>& rhs);
    template <class RanIt>
       Dist operator- (
          const reverse_iterator<RanIt>& lhs,
          const reverse_iterator<RanIt>& rhs);
    template <class RanIt>
       reverse_iterator<RanIt> operator + (Dist n,
          const reverse_iterator<RanIt>& rhs);
    template <class Cont>
       back_insert_iterator<Cont> back_inserter(Cont& x);
    template <class Cont>
       front_insert_iterator<Cont> front_inserter(Cont& x);
    template <class Cont>
       insert_iterator<Cont> inserter(Cont& x);
    template <class InIt, class Dist>
       void advance(InIt & it, Dist n);
    template <class InIt>
       iterator_traits<InIt>::difference_type
          distance(InIt first, InIt last);
};
```

使用描述：

■ iterator_traits

```
    template <class It>
    struct iterator_traits{
          typedef typename It::iterator_category
             iterator_category;
          typedef typename It::value_type       value_type;
          typedef typename It::difference_type
             difference_type;
          typedef typename It::pointer         pointer;
```

```
    typedef typename It::reference        reference;
};
template <class T>
struct iterator_traits<T *>{
    typedef random_access_iterator_tag iterator_category;
    typedef T value_type;
    typedef ptrdiff_t difference_type;
    typedef T * pointer;
    typedef T & reference;
};
template <class T>
struct iterator_traits<const T *>{
    typedef random_access_iterator_tag iterator_category;
    typedef T value_type;
    typedef ptrdiff_t difference_type;
    typedef const T * pointer;
    typedef const T & reference;
};
```

顾名思意，**iterator_traits** 就是迭代器类型提取器，也就是说由这三个类，我们可以从迭代器中提取出我们所需的类型。这里用到了部分特化的知识，在这里为了完整性它列出来，而且这里涉汲到 STL 的实现，具体使用时不大会感觉到它的存在。不打算多讨论了。

■ struct output_iterator_tag

输出迭代器类型标签。可能使用迭代器来存取有序序列中的元素。我们可用如下的循环语句，获得一个序列：

```
for (; <not done>; ++next)
    *next=<whatever>;
```

在此，next 可以看成是指针，但在 STL 中更广泛的含义就是迭代器。这里 next 就相当于输出迭代器。一个标准的输出迭代器至少需要定义下面的操作：

- *next=<whatever>

将<whatever>的值赋给序列中将要产生的一个元素

- ++next 改变 next 的值，使之指向序列中的下一个元素。

■ struct input_iterator_tag

输入迭代器类型标签。

■ struct forward_itertor_tag

前向迭代器类型标签。

■ struct bidirectional_iterator_tag

双向迭代器类型标签。

■ struct random_access_iterator_tag

随机存取迭代器类型标签。

以上四个结构没有任何的代码，而且也没有必要抱含代码，因为，它们只是用来标识迭代器的类型。

■ advance

```
advance(first,N)//add N to X
```

■ distance

```
N=distance(first, last)
// returns the distance from first to last
```

■ reverse_iterator

对于一个给定的类型为 RanIt 的随机存取迭代器，我们可以定义一个与之相关的反转型迭代器，并用其来逆向存取序列。

以下为插入型迭代器

■ back_insert_iterator

给定一个类型为 Cont 的容器，我们可以定义一个输出迭代器 OutIt,用来向被控序列末端插入元素。

```
typdef back_insert_iterator <Cont> OutIt;
```

■ front_insert_iterator

我们同样可以定义一个输出迭代器 OutIt，用其来向被控序列前端以逆序的方式插入元素。

```
typdef front_insert_iterator<Cont> OutIt;
```

■ insert_iterator

我们还可以定义一个输出迭代器，并用它来把反转的序列添加到容器的前端：

```
typedef insert_iterator<Cont> OutIt;
```

下面主要谈一下流迭代器的使用：

■ istream_iterator

下面我们试着定义一个用于读入流的流迭代器

```
typedef istream_iterator <T,char,char_tarits<char> > InIt;
```

这是关于类型 T 的完整定义，在一般的使用中 STL 自行提供的后面两个模板特化的参数，也就是说，如果我们需要一个用于读入字符流的迭代器，我们只要而且一般都这么写：

- `typedef istream_iterator <char> InIt;`

这样一个流迭代器 InIt 可以很好的与 istream 对象协同工作。

再定义

- `InIt first(cin),last;`

这样做之后，first 就成为标准输入流的迭代器，而 last 则被 istream_iterator 的默认构造函数，初使化为流的结束符。所以之后我们可以这样做

```
while(first!=last)
    <process>(*first++);
```

就完成了对字符流的处理。

■ ostream_iterator

仿照对输入迭代器的介绍，这里只给一些代码和注释：

```
typdef ostream_iterator<T> OutIt;
OutIt next(cout," ");//space 是分隔符
while (<not done>)
{
    T x=<next value>;
    *next++ = x;
}
```

- istreambuf_iterator
- ostreambuf_iterator

这里就不介绍了。只列出。

使用<iterator>:

由于 iterator 和容器结合紧密，我们这里给出的代码也不免涉及到容器。请读者自己体会其中含义：

```
#include <iostream>
#include <vector>

using namespace std;
vector<int> myV;
int main()
{
    for (int i=0;i<10;i++)
    {
        myV.push_back(i);
    }
    vector<int>::iterator it;
    for (it=myV.begin();it!=myV.end();it++)
    {
        cout<<(*it)<<' ';
    }
    return 0;
}
```

第二章　搜　　索

搜索问题，涉及到的知识较为复杂，也是竞赛中的一个难点和热点。搜索类问题主要要求对图论的知识比较高，因为其中最常用的两个算法：深度优先和广度优先就是解搜索类问题的基础知识。希望大家在阅读这一章时，结合图论知识进行学习，这样就能做到事半功倍的效果。祝大家学习搜索愉快！

一、宽度优先搜索 BFS

宽度优先搜索算法（又称广度优先搜索）是最简便的图的搜索算法之一，这一算法也是很多重要的图的算法的原型。Dijkstra 单源最短路径算法和 Prim 最小生成树算法都采用了和宽度优先搜索类似的思想。

已知图 G=(V,E)和一个源顶点 s，宽度优先搜索以一种系统的方式探寻 G 的边，从而“发现”s 所能到达的所有顶点，并计算 s 到所有这些顶点的距离(最少边数)，该算法同时能生成一棵根为 s 且包括所有可达顶点的宽度优先树。对从 s 可达的任意顶点 v，宽度优先树中从 s 到 v 的路径对应于图 G 中从 s 到 v 的最短路径，即包含最小边数的路径。该算法对有向图和无向图同样适用。

之所以称之为宽度优先算法，是因为算法自始至终一直通过已找到和未找到顶点之间的边界向外扩展，就是说，算法首先搜索和 s 距离为 k 的所有顶点，然后再去搜索和 S 距离为 k+l 的其他顶点。

为了保持搜索的轨迹，宽度优先搜索为每个顶点着色：白色、灰色或黑色。算法开始前所有顶点都是白色，随着搜索的进行，各顶点会逐渐变成灰色，然后成为黑色。在搜索中第一次碰到一顶点时，我们说该顶点被发现，此时该顶点变为非白色顶点。因此，灰色和黑色顶点都已被发现，但是，宽度优先搜索算法对它们加以区分以保证搜索以宽度优先的方式执行。若(u,v)∈E 且顶点 u 为黑色，那么顶点 v 要么是灰色，要么是黑色，就是说，所有和黑色顶点邻接的顶点都已被发现。灰色顶点可以与一些白色顶点相邻接，它们代表着已找到和未找到顶点之间的边界。

在宽度优先搜索过程中建立了一棵宽度优先树，起始时只包含根节点，即源顶点 s.在扫描已发现顶点 u 的邻接表的过程中每发现一个白色顶点 v，该顶点 v 及边(u,v)就被添加到树中。在宽度优先树中，我们称结点 u 是结点 v 的先辈或父母结点。因为一个结点至多只能被发现一次，因此它最多只能有--个父母结点。相对根结点来说祖先和后裔关系的定义和通常一样：如果 u 处于树中从根 s 到结点 v 的路径中，那么 u 称为 v 的祖先，v 是 u 的后裔。

下面的宽度优先搜索过程 BFS 假定输入图 G=(V,E)采用邻接表表示，对于图中的每个顶点还采用了几种附加的数据结构，对每个顶点 u∈V，其色彩存储于变量 color[u]中，结点 u 的父母存于变量 π[u]中。如果 u 没有父母(例如 u=s 或 u 还没有被检索到)，则 π[u]=NIL，由算法算出的源点 s 和顶点 u 之间的距离存于变量 d[u]中，算法中使用了一个先进先出队列 Q 来存放灰色节点集合。其中 head[Q]表示队列 Q 的队头元素，Enqueue(Q,v)表示将元素 v 入队， Dequeue(Q)表示对头元素出队；Adj[u]表示图中和 u 相邻的节点集合。

```
  procedure BFS(G,S);
  begin
1.  for 每个节点u∈V[G]-{s} do
      begin
2.      color[u]←White;
3.      d[u]←∞;
4.      π[u]←NIL;
      end;
5.  color[s]←Gray;
6.  d[s]←0;
7.  π[s]←NIL;
8.  Q←{s}
9.  while Q≠φ do
     begin
10.    u←head[Q];
11.    for 每个节点v∈Adj[u] do
12.      if color[v]=White then
           begin
13.          color[v]←Gray;
14.          d[v]←d[v]+1;
15.          π[v]←u;
16.          Enqueue(Q,v);
           end;
17.    Dequeue(Q);
18.    color[u]←Black;
     end;
  end;
```

图 1 展示了用 BFS 在例图上的搜索过程。黑色边是由 BFS 产生的树枝。每个节点 u 内的值为 d[u]，图中所示的队列 Q 是第 9-18 行 while 循环中每次迭代起始时的队列。队列中每个结点下面是该结点与源结点的距离。

过程 BFS 按如下方式执行，第 1-4 行置每个结点为白色，置 d[u]为无穷大，每个结点的父母置为 NIL，第 5 行置源结点 S 为灰色，即意味着过程开始时源结点已被发现。第 6 行初始化 d[s]为 0，第 7 行置源结点的父母结点为 NIL，第 8 行初始化队列 0，使其仅含源结点 s，以后 Q 队列中仅包含灰色结点的集合。

程序的主循环在 9-18 行中，只要队列 Q 中还有灰色结点，即那些已被发现但还没有完全搜索其邻接表的结点，循环将一直进行下去。第 10 行确定队列头的灰色结点为 u。第 11-16 行的循环考察 u 的邻接表中的每一个顶点 v。如果 v 是白色结点，那么该结点还没有被发现过，算法通过执行第 13-16 行发现该结点。首先它被置为灰色，距离 d[v]置为 d[u]+1，而后 u 被记为该节点的父母，最后它被放在队列 Q 的队尾。当结点 u 的邻接表中的所有结点都被检索后，第 17-18 行使 u 弹出队列并置成黑色。

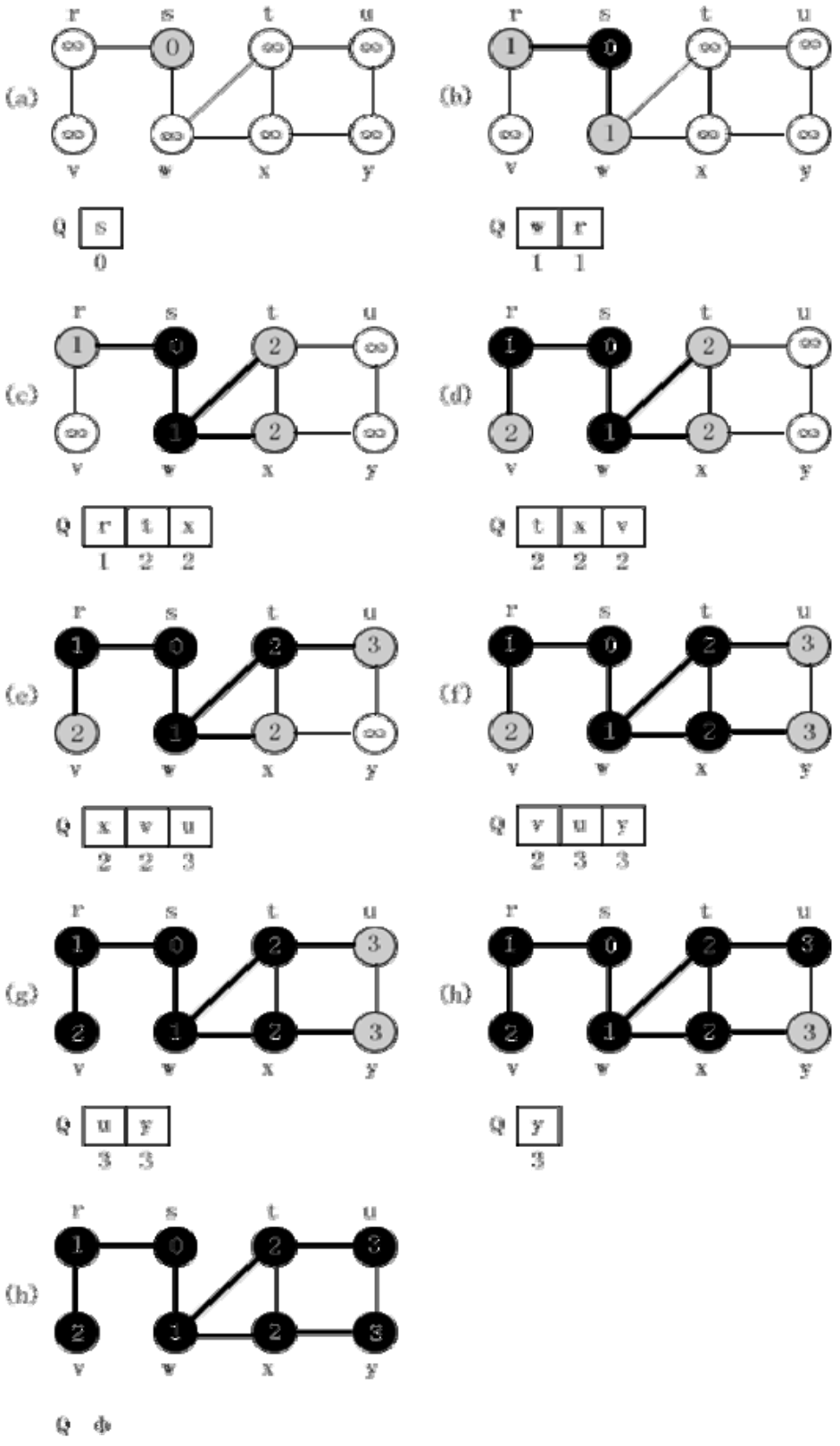


图 1 BFS 在一个无向图上的执行过程

分析：

在证明宽度优先搜索的各种性质之前，我们先做一些相对简单的工作——分析算法在图 G=(V,E)之上的运行时间。在初始化后，再没有任何结点又被置为白色。因此第 12 行的测试保证每个结点至多只能进人队列一次，因而至多只能弹出队列一次。入队和出队操作需要 O(1)的时间，因此队列操作所占用的全部时间为 O(V)，因为只有当每个顶点将被弹出队列时才会查找其邻接表，因此每个顶点的邻接表至多被扫描一次。因为所有邻接表的长度和为 Q(E)，所以扫描所有邻接表所花费时间至多为 O(E)。初始化操作的开销为 O(V)，因此过程 BFS 的全部运行时间为 O(V+E)，由此可见，宽度优先搜索的运行时间是图的邻接表大小的一个线性函数。

最短路径

在本部分的开始，我们讲过，对于一个图 G=(V,E)，宽度优先搜索算法可以得到从已知源结点 s∈V 到每个可达结点的距离，我们定义最短路径长度 δ(s,v)为从顶点 s 到顶点 v 的路径中具有最少边数的路径所包含的边数，若从 s 到 v 没有通路则为∞。具有这一距离 δ(s,v)的路径即为从 s 到 v 的最短路径（后文我们将把最短路径推广到赋权图，其中每边都有一个实型的权值，一条路径的权是组成该路径所有边的权值之和，目前讨论的图都不是赋权图）。在证明宽度优先搜索计算出的就是最短路径长度之前，我们先看一下最短路径长度的一个重要性质。

引理 1

设 G=(V,E)是一个有向图或无向图，s∈V 为 G 的任意一个结点，则对任意边(u,v)∈E，δ(s,v)≤δ(s,u)+1

证明:

如果从顶点 s 可达顶点 u，则从 s 也可达 v。在这种情况下从 s 到 v 的最短路径不可能比从 s 到 u 的最短路径加上边(u,v)更长，因此不等式成立；如果从 s 不可达顶点 u，则 δ(s,v)=∞，不等式仍然成立。

我们试图说明对每个顶点 v∈V，BFS 过程算出的 d[v]=δ(s,v)，下面我们首先证明 d[v]是 δ(s,v)的上界。

引理 2

设 G=(V,E)是一个有向或无向图，并假设算法 BFS 从 G 中一已知源结点 s∈V 开始执行，在执行终止时，对每个顶点 v∈V，变量 d[v]的值满足：d[v]≥δ(s,v)。

证明:

我们对一个顶点进入队列 Q 的次数进行归纳，我们归纳前假设在所有顶点 v∈V，d[v]≥δ(s,v)成立。

归纳的基础是 BFS 过程第 8 行当结点 s 被放入队列 Q 后的情形，这时归纳假设成立，因为对于任意结点 v∈V-{s},d[s]=0=δ(s,s)且 d[v]=∞≥δ(s,v)。

然后进行归纳，考虑从顶点 u 开始的搜索中发现一白色顶点 v，按归纳假设，d[u]≥δ(s,u)。从过程第 14 行的赋值语句以及引理 1 可知

d[v]=d[u]+1≥δ(s,u)+1≥δ(s,v)

然后，结点 v 被插入队列 Q 中。它不会再次被插入队列，因为它已被置为灰色，而第 13-16 行的 then 子句只对白色结点进行操作，这样 d[v]的值就不会改变，所以归纳假设成立。

为了证明 d[v]=δ(s,v)，首先我们必须更精确地展示在 BFS 执行过程中是如何对队列进行操作的，下面一个引理说明无论何时，队列中的结点至多有两个不同的 d 值。

引理 3

假设过程 BFS 在图 G=(V,E)之上的执行过程中，队列 Q 包含如下结点$<v_1,v_2,...,v_r>$，其中 v_1 是队列 Q 的头，v_r 是队列的尾，则 $d[v_i]\le d[v_1]+1$ 且 $d[v_i]\le d[v_{i+1}]$, i=1,2,..,r-1。

证明:

证明过程是对队列操作的次数进行归纳。初始时，队列仅包含顶点 s，引理自然正确。

下面进行归纳，我们必须证明在压入和弹出一个顶点后引理仍然成立。如果队列的头 v_1 被弹出队列，新的队头为 v_2(如果此时队列为空，引理无疑成立)，所以有 $d[v_r]\le d[v_1]+1\le d[v_2]+1$，余下的不等式依然成立，因此 v_2 为队头时引理成立。要插入一个结点入队列需仔细分析过程 BFS，在 BFS 的第 16 行，当顶点 v 加入队列成为 v_{r+1} 时，队列头 v_1 实际上就是正在扫描其邻接表的顶点 u，因此有 $d[v_{r+1}]=d[v]=d[u]+1=d[v_1]+1$，这时同样有 $d[v_r]\le d[v_1]+1=d[u]+1=d[v]=d[v_{r+1}]$，余下的不等式 $d[v_r]\le d[v_{r+1}]$仍然成立，因此当结点 v 插入队列时引理同样正确。

现在我们可以证明宽度优先搜索算法能够正确地计算出最短路径长度。

定理 1 宽度优先搜索的正确性

设 G=(V,E)是一个有向图或无向图，并假设过程 BFS 从 G 上某顶点 s∈V 开始执行，则在执行过程中，BFS 可以发现源结点 s 可达的每一个结点 v∈V，在运行终止时，对任意 v∈V,d[v]=δ(s,v)。此外，对任意从 s 可达的节点 v≠s，从 s 到 v 的最短路径之一是从 s 到 π[v]的最短路径再加上边(π[v],v)。

证明:

我们先证明结点 v 是从 s 不可达的情形。由引理 2，d[v]≥δ(s,v)=∞，根据过程第 14 行，顶点 v 不可能有一个有限的 d[v]值，由归纳可知，不可能有满足下列条件的第一个顶点存在：该顶点的 d 值被过程的第 14 行语句置为∞，因此仅对有有限 d 值的顶点，第 14 行语句才会被执行。所以若 v 是不可达的话，它将不会在搜索中被发现。

证明主要是对由 s 可达的顶点来说的。设 V_k 表示和 s 距离为 k 的顶点集合，即 $V_k=\{v\in V:\delta(s,v)=k\}$。证明过程为对 k 进行归纳。作为归纳假设，我们假定对于每一个顶点 $v\in V_k$，在 BFS 的执行中只有某一特定时刻满足：

- 结点 v 为灰色；
- d[v]被置为 k；
- 如果 v≠s，则对于某个 $u\in V_{k-1}$，π[v]被置为 u；
- v 被插入队列 Q 中；

正如我们先前所述，至多只有一个特定时刻满足上述条件。

归纳的初始情形为 k=0，此时 $V_0=\{s\}$，因为显然源结点 s 是唯一和 s 距离为 0 的结点，在初始化过程中，s 被置为灰色，d[s]被置为 0，且 s 被放人队列 Q 中，所以归纳假设成立。

下面进行归纳，我们须注意除非到算法终止，队列 Q 不为空，而且一旦某结点 u 被插入队列，d[u]和 π[u]都不再改变。根据引理 3 可知如果在算法过程中结点按次序 $v_1,v_2,...,v_r$ 被插入队列，那么相应的距离序列是单调递增的：$d[v_i]\le d[v_{i+1}]$，i=1,2,...,r-1。

现在我们考虑任意结点 $v\in V_k$，$k\ge 1$。根据单调性和 $d[v]\ge k$(由引理 2)和归纳假设，可知如果 v 能够被发现，则必在 V_{k-1} 中的所有结点进入队列之后。

由 $\delta(s,v)=k$，可知从 s 到 v 有一条具有 k 边的通路，因此必存在某结点 $u\in V_{k-1}$，且$(u,v)\in E$。不失一般性，设 u 是满足条件的第一个灰色节点(根据归纳可知集合 V_{k-1} 中的所有结点都被置为灰色)，BFS 把每一个灰色结点放入队列中，这样由第 10 行可知结点 u 最终必然会作为队头出现。当已成为队头时，它的邻接表将被扫描就会发现结点 v（结点 v 不可能在此之前被发现，因为它不与 $V_j(j<k-1)$中的任何结点相邻接，否则 v 不可能属于 V_k，并且根据假定，u 是和 v 相邻接的 V_{k-1} 中被发现的第一个结点)。第 13 行置 v 为灰色，第 14 行置 d[v]=d[u]+l=k，第 15 行置 π[v]为 u，第 16 行把 v 插入队列中。由于 v 是 V_k 中的任意结点，因此证明归纳假设成立。

在结束定理的证明前，我们注意到如果 $v\in V_k$，则据我们所知可得 $\pi[v]\in V_{k-1}$，这样我们就得到了一条从 s 到 v 的最短路径：即为从 s 到 π[v]的最短路径再通过边(π[v],v)。

宽度优先树

过程 BFS 在搜索图的同时建立了一棵宽度优先树，如图 1 所示，这棵树是由每个结点的 π 域所表示。我们正式定义先辈子图如下，对于图 G=(V,E)，源顶点为 s，其先辈子图 $G_\pi=(V_\pi,E_\pi)$满足：

$V_\pi=\{v\in V:\pi[v]\neq NIL\}\cup\{s\}$

且

$E_\pi=\{(\pi[v],v)\in E:v\in V_\pi-\{s\}\}$

如果 V_π 由从 s 可达的顶点构成，那么先辈子图 G_π 是一棵宽度优先树，并且对于所有 $v\in V_\pi$，G_π 中唯一的由 s 到 v 的简单路径也同样是 G 中从 s 到 v 的一条最短路径。由于它互相连通，且$|E_\pi|=|V_\pi|-1$(由树的性质)，所以宽度优先树事实上就是一棵树，E_π 中的边称为树枝。

当 BFS 从图 G 的源结点 s 开始执行后，下面的引理说明先辈子图是一棵宽度优先树。

引理 4

当过程 BFS 应用于某一有向或无向图 G=(V,E)时，该过程同时建立的 π 域满足条件：其先辈子图 $G_\pi=(V_\pi,E_\pi)$是一棵宽度优先树。

证明：

过程 BFS 的第 15 行语句对$(u,v)\in E$ 且 $\delta(s,v)<\infty$(即 v 从 s 可达)置 $\pi[v]=u$，因此 V_π 是由 V 中从 v 可达的顶点所组成，由于 G_π 形成一棵树，所以它包含从 s 到 V_π 中每一结点的唯一路径，由定理 1 进行归纳，我们可知其每条路径都是一条最短路径。(证毕)

下面的过程将打印出从 S 到 v 的最短路径上的所有结点，假定已经运行完 BFS 并得出了最短路径树。

```
 procedure Print_Path(G,s,v);
 begin
1.  if v=s
2.    then write(s)
```

```
3.    else if π[v]=nil
4.         then writeln('no path from ',s,' to ',v, 'exists.')
            else
              begin
5.              Print_Path(G,s,π[v]);
6.              write(v);
              end;
  end;
```

因为每次递归调用的路径都比前一次调用少一个顶点，所以该过程的运行时间是关于打印路径上顶点数的一个线性函数。

宽度搜索的基本思路是从源点出发，每一步找到当前点所有可能达到的所有情况（在图中就是相邻的点），逐层扩展，直到得到解为止。在竞赛题中一般不会使用它，而是使用它的推广算法，dijsktra,prime 等算法，对他们的掌握是比较重要的。

二、最小生成树的形成和求解(Prim and Kruskal)

1. 最小生成树的形成

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

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

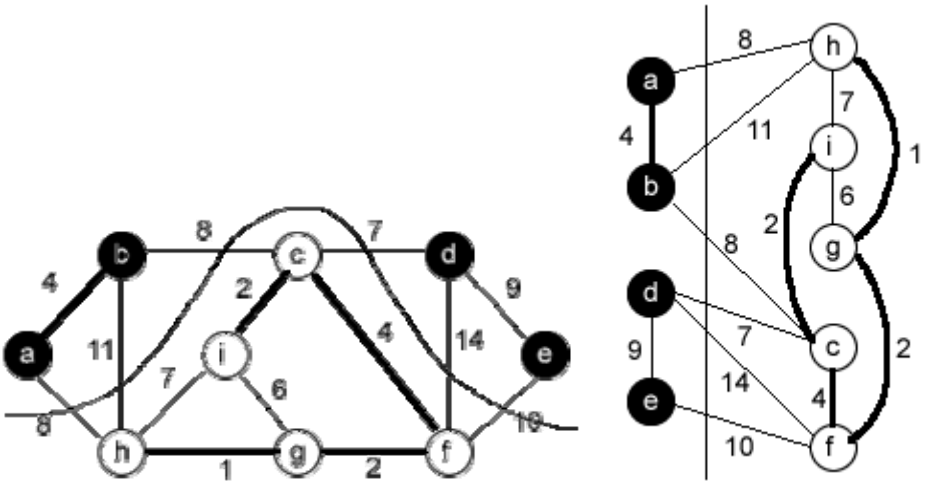
```
GENERNIC-MIT(G,W)
1. A←Æ
2. while A 没有形成一棵生成树
3    do  找出 A 的一条安全边(u,v);
4.       A←A∪{(u,v)};
5. return A
```

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

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

首先我们来定义几个概念。有向图 G=(V,E)的割(S,V-S)是 V 的一个分划。当一条边(u,v)∈E 的一个端点属于 S 而另一端点属于 V-S，则我们说边(u,v)通过割(S,V-S)。若集合 A

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



(a) (b)

图 2 从两种途径来观察图 1 所示图的割(S,V-S)

确认安全边的规则由下列定理给出。

定理 1

设图 G(V,E)是一无向连通图，且在 E 上定义了相应的实数值加权函数 w，设 A 是 E 的一个子集且包含于 G 的某个最小生成树中，割(S,V-S)是 G 的不妨害 A 的任意割且边(u,v)是穿过割(S,V-S)的一条轻边，则边(u,v)对集合 A 是安全的。

证明：

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

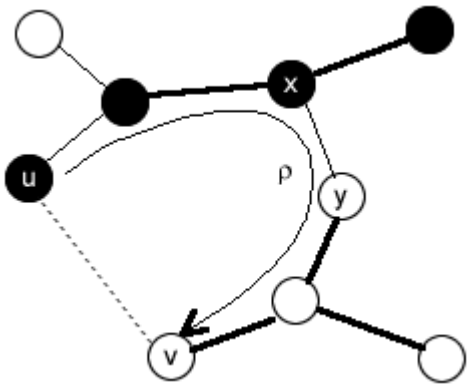


图 3 定理 1 的证明

图 3 演示出了**定理 1** 的证明。S 中的结点为黑色，V-S 中的结点为白色，边(u,v)与 T 中从 u 到 v 的通路 P 中的边构成一回路。由于 u 和 v 处于割(S,V-S)的相对的边上，因此在 T 中的通路 P 上至少存在一条边也通过割。设(x,y)为满足此条件的边。因为割不妨害 A，所以边(x,y)不属于 A。又因为(x,y)处于 T 中从 u 到 v 的唯一通路上，所以去掉边(x,y)就会把 T 分成两个子图。这时加入边(u,v)以形成一新的生成树 $T'=T-\{(x,y)\}\cup\{(u,v)\}$。

下一步我们证明 T′是一棵最小生成树。因边(u,v)是通过割(S,V-S)的一条轻边且边(x,y)也通过割，所以有 $w(u,v)\le w(x,y)$，因此 $w(T')=w(T)-w(x,y)+w(u,v)\le(T)$，但 T 是最小生成树，所以 $w(T)=w(T')$，因此 T′必定也是最小生成树。

现在还要证明(u,v)实际上是 A 的安全边。由于 $A\subseteq T$ 且 $(x,y)\notin A$，所以有 $A\subseteq T'$，则 $A\cup\{(u,v)\}\subseteq T'$。而 T′是最小生成树，因而(u,v)对 A 是安全的。□

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

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

第 2 节中论述的两种算法均使用了下列定理 1 的推论。

推论 2

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

证明：

因为割(C,V-C)不妨害 A，因此(u,v)是该割的一条轻边。

2．Kruskal 算法和 Prim 算法

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

Kruskal 算法

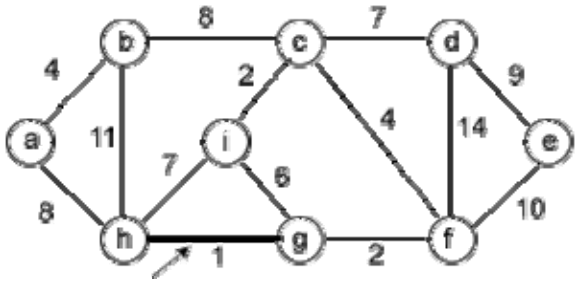
Kruskal 算法是直接基于上一节中给出的一般最小生成树算法的基础之上的。该算法找出森林中连结任意两棵树的所有边中具有最小权值的边(u,v)作为安全边，并把它添加到正在生长的森林中。设 C_1 和 C_2 表示边(u,v)连结的两棵树。因为(u,v)必是连 C_1 和其他某棵树的一条轻边，所以由推论 2 可知(u,v)对 C_1 是安全边。Kruskal 算法同时也是一种贪心算法，因为算法每一步添加到森林中的边的权值都尽可能小。

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

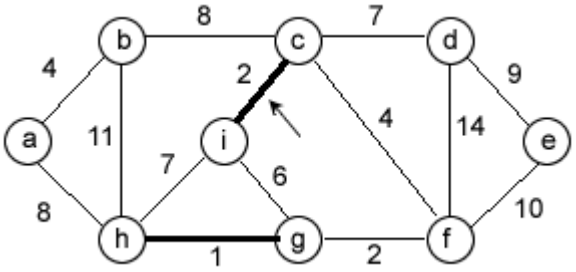
```
MST-KRUSKAL(G,w)
1. A←Φ
2. for 每个结点 v V[G]
3.   do MAKE-SET(v)
4. 根据权 w 的非递减顺序对 E 的边进行排序
5. for 每条边(u,v) E，按权的非递减次序
6.   do if FIND-SET(u) ≠ FIND-SET(v)
7.        then A←A∪{(u,v)}
8.             UNION(u,v)
9  return A
```

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

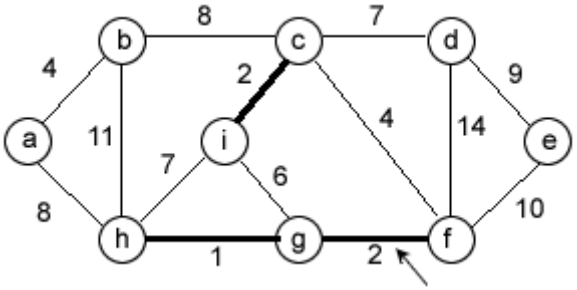
Kruskal 算法在图 G=(V,E)上的运行时间取决于分离集合这一数据结构如何实现。我们采用在分离集合中描述的按行结合和通路压缩的启发式方法来实现分离集合森林的结构，这是由于从渐近意义上来说，这是目前所知的最快的实现方法。初始化需占用时间 O(V)，第 4 行中对边进行排序需要的运行时间为 $O(E \log E)$；对分离集的森林要进行 O(E)次操作，总共需要时间为 $O(E\alpha(E,V))$，其中α函数为 Ackerman 函数的反函数。因为$\alpha(E,V)=O(\log E)$，所以 Kruskal 算法的全部运行时间为 $O(E \log E)$。


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
6
7
9
14
10
1
2

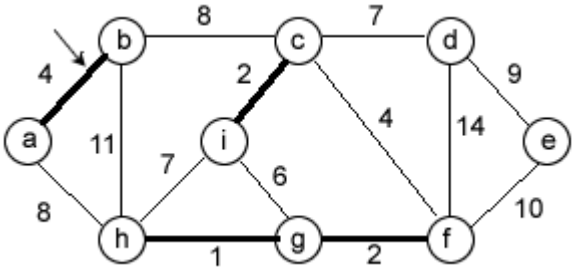
(a)


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
6
7
9
14
10
1
2

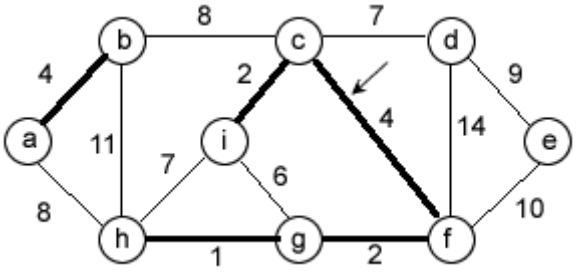
(b)


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
6
7
9
14
10
1
2

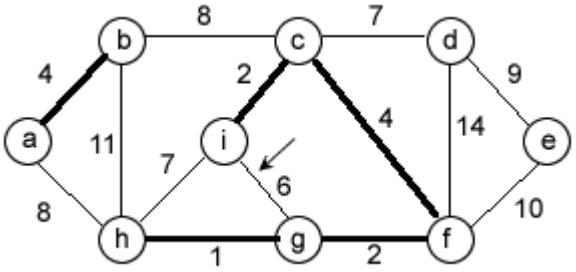
(c)


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
6
7
9
14
10
1
2

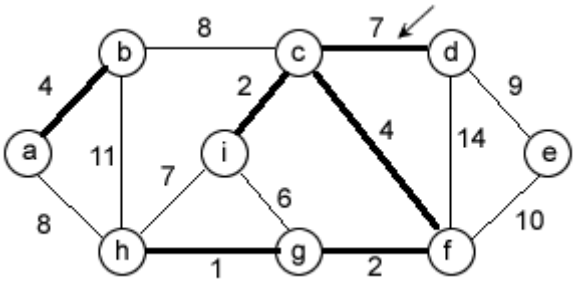
(d)


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
6
7
9
14
10
1
2

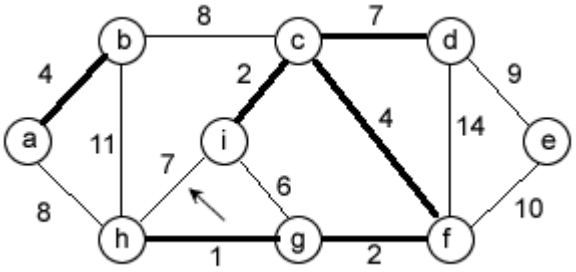
(e)


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
6
7
9
14
10
1
2

(f)


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
6
7
9
14
10
1
2

(g)


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
6
7
9
14
10
1
2

(h)

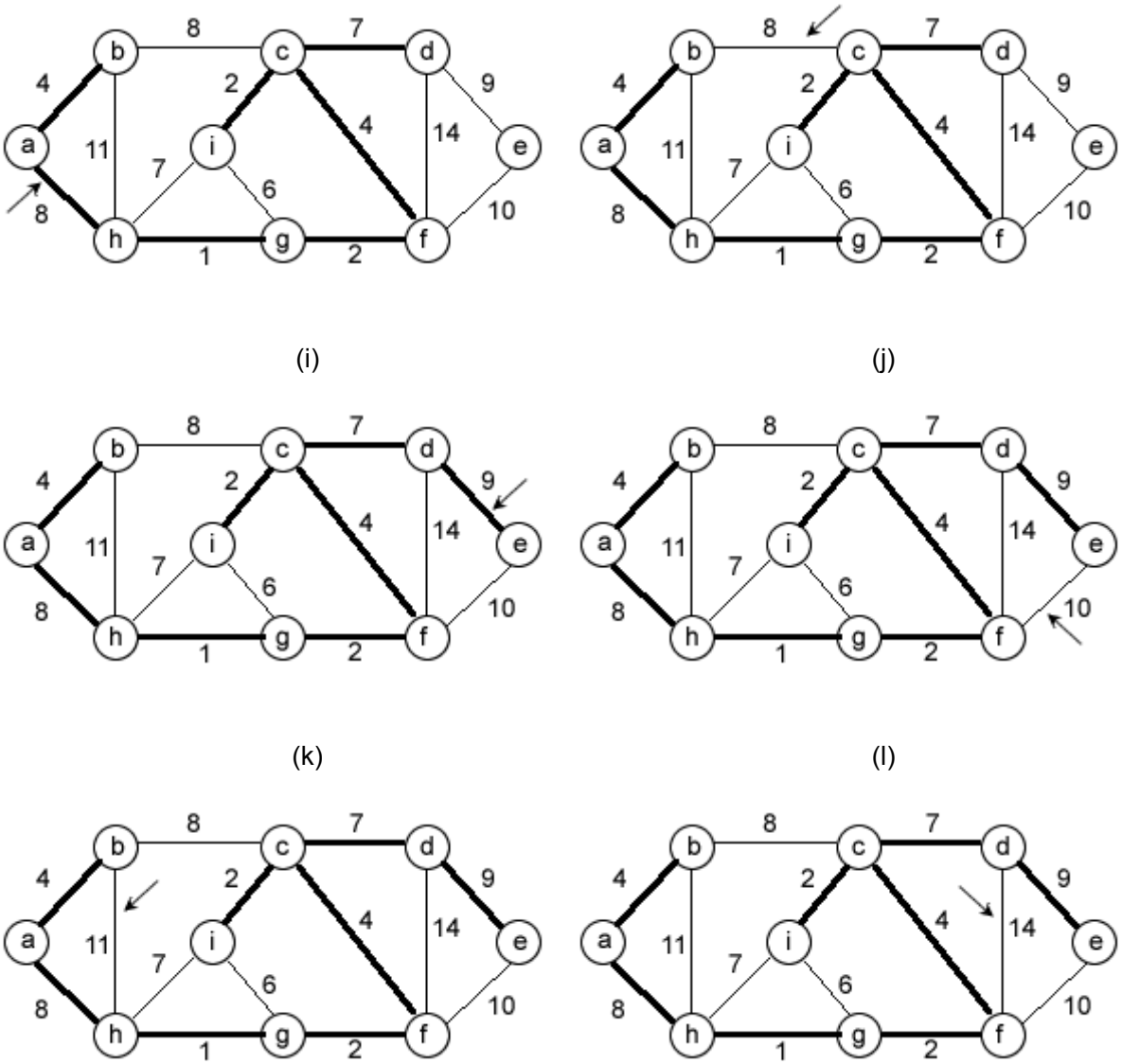


图 4 Kruskal 算法在图 1 所示的图上的执行流程

Prim 算法

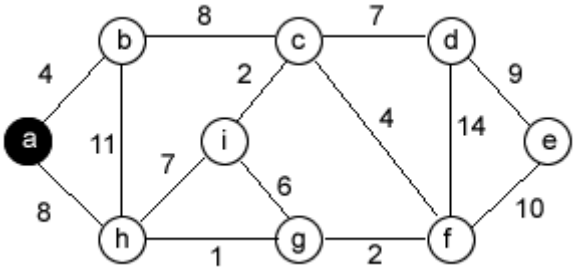
正如 Kruskal 算法一样，Prim 算法也是第上一节中讨论的一般最小生成树的特例。Prim 算法的执行非常类似于寻找图的最短通路的 Dijkstra 算法。Prim 算法的特点是集合 A 中的边总是只形成单棵树。如图 5 所示，阴影覆盖的边属于正在生成的树，树中的结点为黑色。在算法的每一步，树中的结点确定了图的一个割，并且通过该割的轻边被加进树中。树从任意根结点 r 开始形成并逐渐生长直至该树跨越了 V 中的所有结点。在每一步，连接 A 中某结点到 V-A 中某结点的轻边被加入到树中，由推论 2，该规则仅加大对 A 安全的边，因此当算法终止时，A 中的边就成为一棵最小生成树。因为每次添加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

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

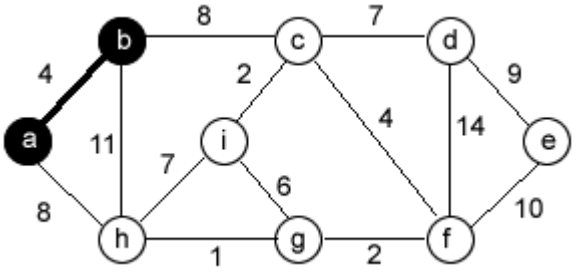
$A=\{(v,\pi[v])|v\in V-\{r\}-Q\}$

当算法终止时，优先队列 Q 为空，因此 G 的最小生成树 A 满足:

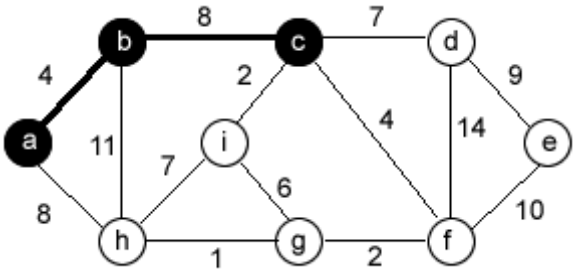
$A=\{(v,\pi[v])|v\in V-\{r\}\}$



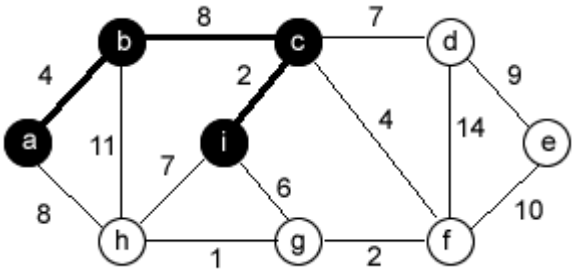
(a)



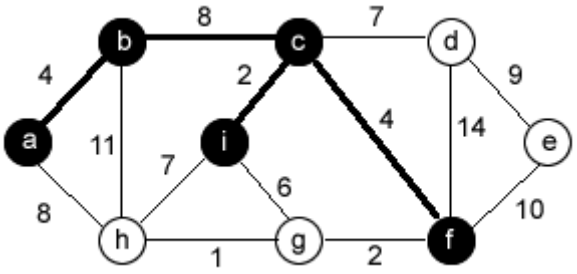
(b)



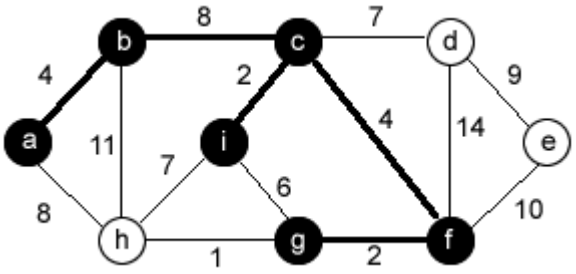
(c)



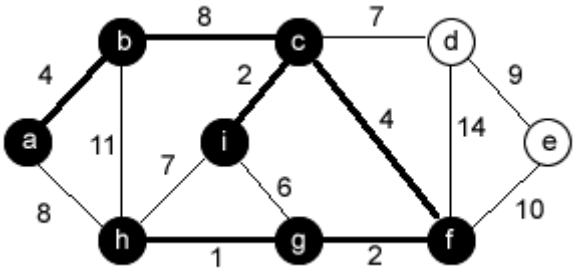
(d)



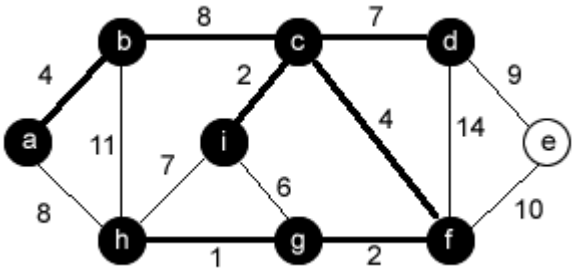
(e)



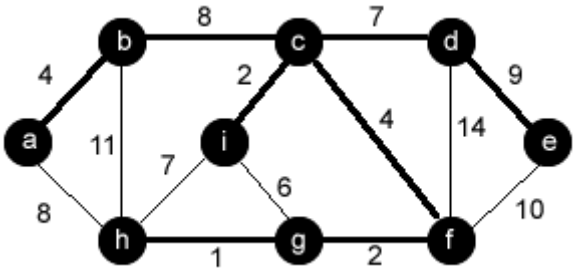
(f)



(g)



(h)



(i)

图 5 Prim 算法在图 1 所示的图上的执行流程

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

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

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

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

现在举一个用 prime 算法的例子

Swordfish

There exists a world within our world a world beneath what we call cyberspace. A world protected by firewalls, passwords and the most advanced security systems. In this world we hide our deepest secrets, our most incriminating information, and of course, a shole lot of money. This is the world of Swordfish.

We all remember that in the movie Swordfish, Gabriel broke into the World Bank Investors Group in West Los Angeles, to rob $9.5 billion. And he needed Stanley, the best hacker in the world, to help him break into the password protecting the bank system. Stanley's lovely daughter Holly was seized by Gabriel, so he had to work for him. But at the last moment, Stanley made some little trick in his hacker mission: he injected a trojan horse in the bank system, so the money would jump from one account to another account every 60 seconds, and would continue jumping in the next 10 years. Only Stanley knew when and where to get the money. If Gabriel killed Stanley, he would never get a single dollar. Stanley wanted Gabriel to release all these hostages and he would help him to find the money back.

You who has watched the movie know that Gabriel at last got the money by threatening to hang Ginger to death. Why not Gabriel go get the money himself? Because these money keep jumping, and these accounts are scattered in different cities. In order to gather up these money Gabriel would need to build money transfering tunnels to connect all these cities. Surely it will be really expensive to construct such a transfering tunnel, so Gabriel wants to find out the minimal total length of the tunnel required to connect all these cites. Now he asks you to write a computer program to find out the minimal length. Since Gabriel will get caught at the end of it anyway, so you can go ahead and write the program without feeling guilty about helping a criminal.

Input

The input contains several test cases. Each test case begins with a line contains only one integer N

(0 <= N <=100), which indicates the number of cities you have to connect. The next N lines each contains two real numbers X and Y(-10000 <= X,Y <= 10000), which are the citie's Cartesian coordinates (to make the problem simple, we can assume that we live in a flat world). The input is terminated by a case with N=0 and you must not print any output for this case.

Output

You need to help Gabriel calculate the minimal length of tunnel needed to connect all these cites. You can saftly assume that such a tunnel can be built directly from one city to another. For each of the input cases, the output shall consist of two lines: the first line contains "Case #n:", where n is the case number (starting from 1); and the next line contains "The minimal distance is: d", where d is the minimal distance, rounded to 2 decimal places. Output a blank line between two test cases.

Sample Input

```
5
0 0
0 1
1 1
1 0
0.5 0.5
0
```

Sample Output

```
Case #1:
The minimal distance is: 2.83
```

参考解答

用图的方法建模，本题实际上就是求最小生成树的路径和，用 prime 算法解决

```
void solve()//用 prime 算法的函数
{
	double legth = 0;//总权值
	double cost[100];//记录权值的数组
	int used[100];//判断点是否访问过
	for(int ix = 0; ix < n; ix++)
	{
		cost[ix] = graph[0][ix];
		used[ix] = 0;
	}
	used[0] = 1;
	for(;;)
	{
		int j = 0, i = 0;
		while(used[j] && j < n)
			j++;//找一个没访问的点
		if(j == n) break;//如果都访问过了，跳出循环
```

```
                for(i = 0; i < n; i++)
                        if(!used[i] && cost[i] < cost[j])
                                j = i;//在剩下的点中找到权值最小的点（权值
                                      //是指起始点到此点的路径长度）
                legth += cost[j];//更新总权值
                used[j] =1;//将此点置为访问过
                for(i = 0; i < n; i++)
                    if(!used[i] && graph[j][i] < cost[i])
                 cost[i] = graph[j][i];//更新未访问的所有点的权值
        }
        out.setf(ios::fixed);
        out << setprecision(2)<<legth << endl;
}
```

三、深度优先搜索 DFS

1.概述

正如算法名称，深度优先搜索所遵循的搜索策略是尽可能“深”地搜索图。在深度优先搜索中，对于最新发现的顶点，如果它还有以此为起点而未探测到的边，就沿此边继续汉下去。当结点 v 的所有边都已被探寻过，搜索将回溯到发现结点 v 有那条边的始结点。这一过程一直进行到已发现从源结点可达的所有结点为止。如果还存在未被发现的结点，则选择其中一个作为源结点并重复以上过程，整个进程反复进行直到所有结点都被发现为止。

和宽度优先搜索类似，每当扫描已发现结点 u 的邻接表从而发现新结点 v 时，深度优先搜索将置 v 的先辈域 π[v]为 u。和宽度优先搜索不同的是，前者的先辈子图形成一棵树，而后者产生的先辈子图可以由几棵树组成，因为搜索可能由多个源顶点开始重复进行。因此深度优先搜索的先辈子图的定义也和宽度优先搜索稍有不同：

$$G_\pi=(V,E_\pi),E_\pi=\{(\pi[v],v)\in E:v\in V\wedge\pi[v]\neq NIL\}$$

深度优先搜索的先辈子图形成一个由数个深度优先树组成的深度优先森林。Eπ 中的边称为树枝。和宽度优先搜索类似，深度优先在搜索过程中也为结点着色以表示结点的状态。每个顶点开始均为白色，搜索中被发现时置为灰色，结束时又被置成黑色(即当其邻接表被完全检索之后)。这一技巧可以保证每一顶点搜索结束时只存在于一棵深度优先树上，因此这些树都是分离的。

除了创建一个深度优先森林外，深度优先搜索同时为每个结点加盖时间戳。每个结点 v 有两个时间戳：当结点 v 第一次被发现(并置成灰色)时记录下第一个时间戳 d[v]，当结束检查 v 的邻接表时(并置 v 为黑色)记录下第二个时间截 f[v]。许多图的算法中都用到时间戳，他们对推算深度优先搜索进行情况是很有帮助的。

下列过程 DFS 记录了何时在变量 d[u]中发现结点 u 以及何时在变量 f[u]中完成对结点 u 的检索。这些时间戳为 1 到 2|V|之间的整数，因为对每一个 v 中结点都对应一个发现事件和一个完成事件。对每一顶点 u，有

$$d[u]<f[u] \qquad (1)$$

在时刻 d[u]前结点 u 为白色，在时刻 d[u]和 f[u]之间为灰色，以后就变为黑色。

下面的伪代码就是一个基本的深度优先搜索算法，输人图 G 可以是有向图或无向图，变量 time 是一个全局变量，用于记录时间戳。

```
  procedure DFS(G);
   begin
1    for 每个顶点 u∈V[G] do
       begin
2        color[u]←White;
3        π[u]←NIL;
       end;
4    time←0;
5    for 每个顶点 u∈V[G] do
6      if color[u]=White
7          then DFS_Visit(G,u);
end;

 procedure DFS_Visit(G,u);
  begin
1   color[u]←Gray;              Δ 白色结点 u 已被发现
2   d[u]←time←time+1;
3   for 每个顶点 v∈Adj[u] do     Δ 探寻边(u,v)
4     if color[v]=White
         then begin
5              π[v]←u;
6              DFS_Visit(G,v);
            end;
7   color[u]←Black;             Δ 完成后置 u 为黑色
8   f[u]←time←time+1;
end;
```

图 2 说明了 DFS 在图 1 所示的图上执行的过程。被算法探寻到的边要么为阴影覆盖 (如果该边为树枝)，要么成虚线形式 (其他情况)。对于非树枝的边，分别标明 B(或 F)以表示反向边、交叉边或无向边。我们用发现时刻 Z 完成时刻的形式对结点加盖时间戳。

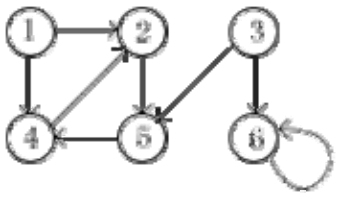


图 1 一个有向图图

过程 DFS 执行如下。第 1-3 行把所有结点置为白色，所有 π 域初始化为 NIL。第 4 行复位全局变量 time，第 5-7 行依次检索 V 中的结点，发现白色结点时，调用 DFS_Visit 去访问该结点。每次通过第 7 行调用 DFS_Visit 时，结点 u 就成为深度优先森林中一棵新树的根，当 DFS 返回时，每个结点 u 都对应于一个发现时刻 d[u]和一个完成时刻 f[u]。

每次开始调用 DFS_Visit(u)时结点 u 为白色，第 1 行置 u 为灰色，第 2 行使全局时间变量增值并存于 d[u]中，从而记录下发现时刻 d[u]，第 3-6 行检查和 u 相邻接的每个顶点 v，且若 v 为白色结点，则递归访问结点 v。在第 3 行语句中考虑到每一个结点 v∈Adj[u]时，我们可以说边(u,v)被深度优先搜索探寻。最后当以 u 为起点的所有边都被探寻后，第 7-8 行语句置 u 为黑色并记录下完成时间 f[u]。

算法 DFS 运行时间的复杂性如何？DFS 中第 1-2 行和 5-7 行的循环占用时间为 O(V)，这不包括执行调用 DFS_Visit 过程语句所耗费的时间。事实上对每个顶点 v∈V，过程 DFS_Visit 仅被调用一次，因为 DFS_Visit 仅适用于白色结点且过程首先进行的就是置结点为灰色，在 DFS_Visit(v)执行过程中，第 3-6 行的循环要执行|Adj[v]|次。因为 ∑v∈V|Adj[v]| =θ(E)，因此执行过程 DFS_Visit 中第 2-5 行语句占用的整个时间应为 θ(E)。所以 DFS 的运行时间为 θ(V+E)。

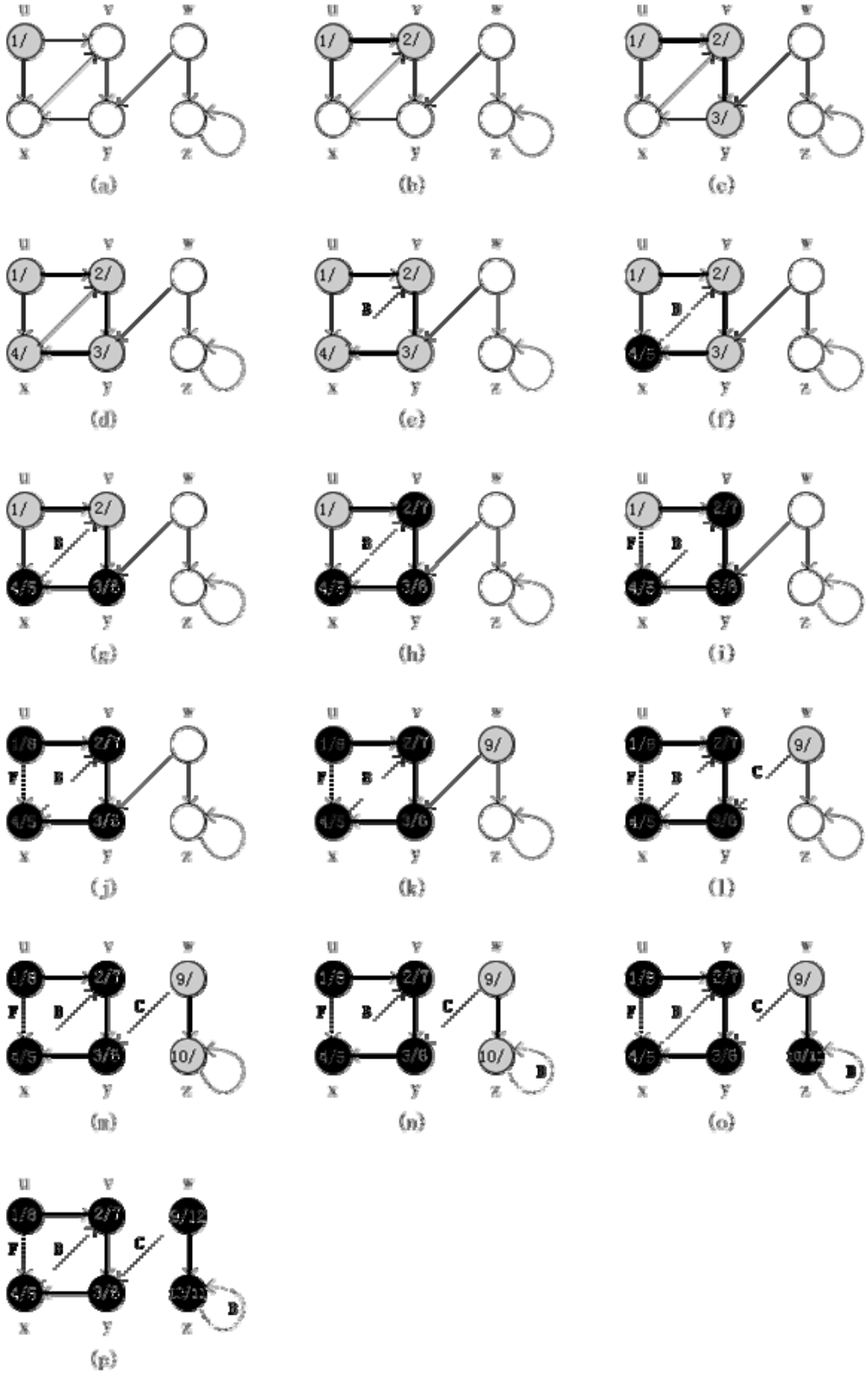


图 2 深度优先搜索算法 DFS 在有向图图 1 上的执行过程

2.深度优先搜索的性质

依据深度优先搜索可以获得有关图的结构的大量信息。也许深度优先搜索的最基本的特征是它的先辈子图 G，形成一个由树组成的森林，这是因为深度优先树的结构准确反映了 DFS_Visit 中递归调用的结构的缘故，即 u=π[v]当且仅当在搜索 u 的邻接表过程中调用了过程 DFS_Visit(v)。

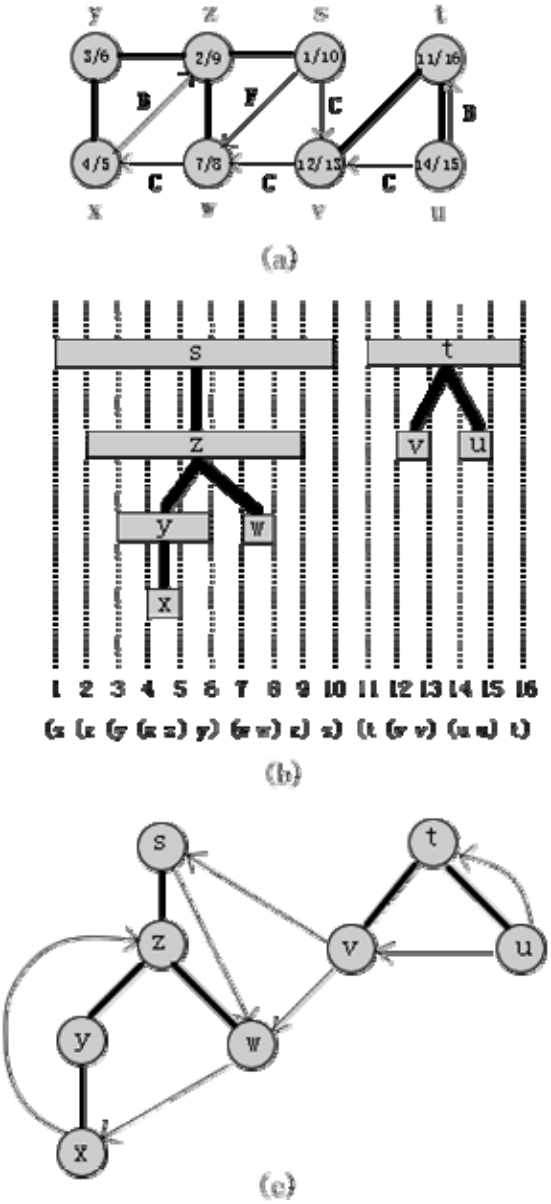


图 3 深度优先搜索的性质

深度优先搜索的另一重要特性是发现和完成时间具有括号结构，如果我们把发现顶点 u 用左括号"(u"表示，完成用右括号"u)"表示，那么发现与完成的记载在括号被正确套用的前提下就是一个完善的表达式。例如，图 3 显示了深度优先搜索的性质。(a)对一个有向图进行深度优先搜索的结果。结点的时间戳与边的类型的表示方式与图 2 相同。(b)图中的括号表示对应于每个结点的发现时刻和完成时刻的组成的区间。每个矩形跨越相应结点的发现时刻与完成时刻所设定的区间。图中还显示了树枝。如果两个区间有重叠，则必有一个区间嵌套于另一个区间内，且对应于较小区间的结点是对应于较大区间的结点的后裔。(c)对(a)中图的重新描述，使深度优先树中所有树枝和正向边自上而下，而所有反向边自下而上从后裔指向祖先。下面的定理给出了标记括号结构的另外一种办法。

定理 1 括号定理

在对有向图或无向图 G=(V,E)的任何深度优先搜索中，对于图中任意两结点 u 和 v，下述三个条件中有且仅有一条成立：

- 区间[d[u],f[u]]和区间[d[v],f[v]]是完全分离的；
- 区间[d[u],f[u]]完全包含于区间[d[v],f[v]]中且在深度优先树中 u 是 v 的后裔；
- 区间[d[v],f[v]]完全包含于区间[d[u],f[u]]中且在深度优先树中 v 是 u 的后裔。

证明：

先讨论 d[u]<d[v]的情形，根据 d[v]是否小于 f[u]又可分为两种情况:第一种情况，若 d[v]<f[u]，这样 v 已被发现时 u 结点依然是灰色，这就说明 v 是 u 的后裔，再者，因为结点 v 比 u 发现得较晚，所以在搜索返回结点 u 并完成之前，所有从 v 出发的边都已被探寻并已完成，所以在这种条件下区间[d[v],f[v]]必然完全包含于区间[d[u],f[u]]。第二种情况,若 f[u]<d[v]，则根据不等式(1)，区间[d[u],f[u]]和区间[d[v],f[v]]必然是分离的。

对于 d[v]<d[u]的情形类似可证，只要把上述证明中 u 和 v 对调一下即可。(证毕)

推论 1 后裔区间的嵌入

在有向或无向图 G 的深度优先森林中，结节 v 是结点 u 的后裔当且仅当 d[u]<d[v]<f[v]<f[u]。

证明：直接由定理 1 推得。（证毕）

在深度优先森林中若某结点是另一结点的后裔，则下述定理将指出它的另一重要特征。

定理 2 白色路径定理

在一个有向或无向图 G=(V,E)的深度优先森林中，结点 v 是结点 u 的后裔当且仅当在搜索发现 u 的时刻 d[u]，从结点 u 出发经一条仅由白色结点组成的路径可达 v。

证明：

→：假设 v 是 u 的后裔，w 是深度优先树中 u 和 v 之间的通路上的任意结点，则 w 必然是 u 的后裔，由推论 1 可知 d[u]<d[w]，因此在时刻 d[u]，w 应为白色。

←：设在时刻 d[u]，从 u 到 v 有一条仅由白色结点组成的通路，但在深度优先树中 v 还没有成为 u 的后裔。不失一般性，我们假定该通路上的其他顶点都是 u 的后裔(否则可设 v 是该通路中最接近 u 的结点，且不为 u 的后裔)，设 w 为该通路上 v 的祖先，使 w 是 u 的后裔(实际上 w 和 u 可以是同一个结点)。根据推论 1 得 f[w]≤f[u]，因为 v∈Adj[w]，对 DFS_Visit(w)的调用保证完成 w 之前先完成 v，因此 f[v]<f[w]≤f[u]。因为在时

刻 d[u]结点 v 为白色，所以有 d[u]<d[v]。由推论 1 可知在深度优先树中 v 必然是 u 的后裔。(证毕)

3. 边的分类

在深度优先搜索中，另一个令人感兴趣的特点就是可以通过搜索对输入图 G=(V,E)的边进行归类，这种归类可以发现图的很多重要信息。例如一个有向图是无回路的，当且仅当深度优先搜索中没有发现“反向边”。

根据在图 G 上进行深度优先搜索所产生的深度优先森林 G，我们可以把图的边分为四种类型。

1．树枝，是深度优先森林 G_{π} 中的边，如果结点 v 是在探寻边(u,v)时第一次被发现，那么边(u,v)就是一个树枝。
2．反向边，是深度优先树中连结结点 u 到它的祖先 v 的那些边，环也被认为是反向边。
3．正向边，是指深度优先树中连接顶点 u 到它的后裔的非树枝的边。
4．交叉边，是指所有其他类型的边，它们可以连结同一棵深度优先树中的两个结点，只要一结点不是另一结点的祖先，也可以连结分属两棵深度优先树的结点。

在图 2 和图 3 中，都对边进行了类型标示。图 3(c)还说明了如何重新绘制图 3(a)中的图，以使深度优先树中的树枝和正向边向下绘制，使反向边向上绘制。任何图都可用这种方式重新绘制。

可以对算法 DFS 进行一些修改，使之遇到边时能对其进行分类。算法的核心思想在于可以根据第一次被探寻的边所到达的结点 v 的颜色来对该边(u,v)进行分类(但正向边和交叉边不能用颜色区分出)。

1．白色表明它是树枝。
2．灰色说明它是反向边。
3．黑色说明它是正向边或交叉边。

第一种情形由算法即可推知。在第二种情形下，我们可以发现灰色结点总是形成一条对应于活动的 DFS_Visit 调用堆栈的后裔线性链，灰色结点的数目等于最近发现的结点在深度优先森林中的深度加 1，探寻总是从深度最深的灰色结点开始，因此达到另一个灰色结点的边所达到的必是它的祖先。余下的可能就是第三种情形，如果 d[u]<d[v]，则边(u,v)就是正向边，若 d[u]>d[v]，则(u,v)便是交叉边。

因此可以得到下面结论：对于某条边(u,v)，

- 当且仅当 d[u]<d[v]<f[v]<f[u]时，是树枝边或正向边；
- 当且仅当 d[v]<d[u]<f[u]<f[v]时，是反向边；
- 当且仅当 d[v]<f[v]<d[u]<f[u]时，是交叉边；

该结论的证明略。

在无向图中，由于(u,v)和(v,u)实际上是同一条边，所以对边进行这种归类可能产生歧义。在这种情况下，图的边都被归为归类表中的第一类，对应地，我们将根据算法执行过程中首先遇到的边是(u,v)还是(v,u)来对其进行归类。

下面我们来说明在深度优先搜索无向图时不会出现正向边和交叉边。

定理 3

在对无向图 G 进行深度优先搜索的过程中，G 的每条边要么是树是反向边。

证明：

设(u,v)为 G 的任意一边，不失一般性，假定 d[u]<d[v]。则因为 v 在 u 的邻接表中，所以我们必定在完成 u 之前就已发现并完成 v。如果边(u,v)第一次是按从 u 到 v 的方向被探寻到，那么(u,v)必是一树枝。如果(u,v)第一次是按从 v 到 u 的方向被探寻到，则由于该边被第一次探寻时 u 依然是灰色结点，所以(u,v)是一条反向边。(证毕)

下面是一道利用深度搜索解决的迷宫问题：

问题描述

下面是一个迷宫

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

在这个比较简单的迷宫中（用的 1 表示可以走的方块，灰度色表示出口），要求找出一条路线可以连接入口和出口。首先确定如何搜索，也就是尝试的原则，现在规定对于每个方块，从它右边开始按顺时针检查它周围的四个方块。则可以得到下列步骤：从（1，1）出发，首先找到（1， 2），位于（1， 2）的方块是可行的，于是自身的位置前进到（1， 2），经尝试，位于（2， 2）的方块可行，在（2， 2）处尝试，得到下一个可行位置（2， 3），以此类推。但是这样只能找到一条路，而不能找到所有的情况，所以当搜索遇到非法情况（包括走出迷宫）时要退回上一步尝试别的路线。直到每一步都检查了它下一步可能走的四个方向。

参考解答

```
int maze[4][6] = {     {1, 1, 0, 0, 0, 0},
                       {0, 1, 1, 1, 0, 0},
                       {1, 1, 0, 1, 0, 0},
                       {0, 0, 0, 1, 0, 0}};
vector<pair<int, int> > path;
int dir[4][2] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
void search(vector<pair<int, int> > tpath, int x, int y)
{
       if(x < 0 || y < 0 || x > 5 || y > 3)//越界返回
              return;
       if(x == 3 && y == 3)
       {
```

```
            path = tpath; //如果找到了出口,则记录下路径
            return;
        }
        for(int ix = 0; ix < 4; ix++)//四个方向搜索
        {
            if(maze[x+dir[ix][0]][ y+dir[ix][1]] == 1)
            {
                tpath.push_back(make_pair<x, y>);
                search(tpath, x+dir[ix][0], y+dir[ix][1]);
                tpath.pop_back();
            }
        }
}

int main()
{
        vector<pair<int, int> > tpath;
        search(tpath, 0, 0);//从开始点找起
}
```

这是一个常见的用递归实现的深度搜索写法。

第三章 计算几何学

一、引　言

首先，我们考虑一下哪些问题属于画法几何的范筹。

在竞赛中，常常会出现这样一些问题，它们与中学里所学的几何问题有关，问题的解决也需要这方面的知识，但问题最终的解决是可以通过算法语言用计算机描述的，我们把这样一类的题目称为画法几何问题。

举个简单的例子，请考虑下面的问题：

There are n (1<=n && n<=1000) points on the board. Please find out a certain pair of points which satisfy the following term: The distance between them is the shortest among that of any of other pairs.

Input

Input file may consist of several cases. In each case, the input datum will be formatted as. A number in the first line, represents variable n. Later, followed by n lines give the coordinates of each point. In each line, the first number is coordinate x and latter is y. Input file is terminated by a line with a single 0;

Output

Each test case must be preceded by “Case k:”, k is the sequence number, then the colon is followed a blank line, at last is the coordinates of the two points which satisfy our request above, the coordinates is separated by space. Maybe more than one pair will be candidates. you should output all of them, each pair per line.

Sample

Input:

```
3
0 0
1 0
0 1
4
0 0
0 1
1 0
1 1
0
```

Output

```
Case 1:
   0 1 1 0
Case 2:
   0 0 1 1
   0 1 1 0
```

这道题，很显然就是我们所说的画法几何的题。它将会用到几何中最简单的两点之间的距离的公式，同时，它又要求我们找出最短的一对点的坐标。这又要求用计算机语言加以描述。读者可以在此先自己想一想这道题的解法。

二、线段的性质（Line-segment properities）

线段是几何中除去点后最基本的几何元素，并且也是解决几何问题用到的最常用的几何元素。所以在此，先介绍线段的一些在解题中常常用到的性质。

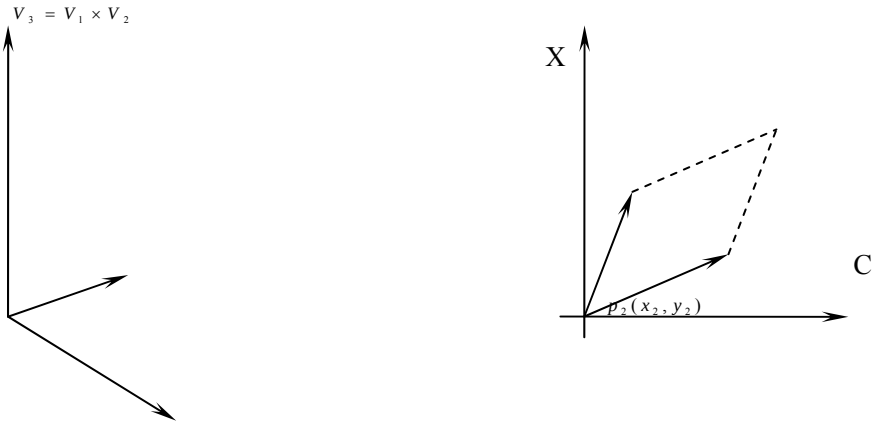
在这一部分我们要解决以下几个问题：

1 给出两个有公共起点的向量 p_0p_1 和 p_0p_2，判断以 p_0 为中心，从 p_0p_1 转到到 p_0p_2 是顺时针方向还是逆时针方向。

2 两个线段是否相交。

针对第二个问题，如果我们先计算出两条线段所在的直线的交点，然后，再看这个点是否是线段上，是不可行的。主要原因有，这个方法用到了浮点除法，在计算机中这是存在精度问题的，其次，这样计算机的效率太低，不能用于处理大数据量的问题。后面我们会介绍解决问题 2 的更为有效的算法。

1.叉积（Cross Product）



叉积，简单说来就是向量相乘的向量积。在左图中，根据向量积的性质，有

$$|\vec{V}_3| = |\vec{V}_1||\vec{V}_2|\sin\alpha$$

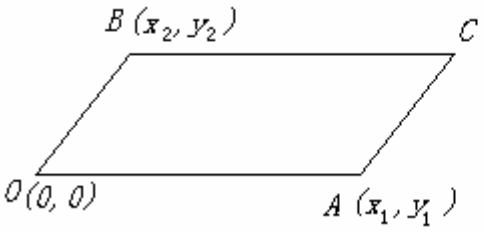
结合右图给出叉积的矩阵定义式：

$$p_1 \times p_2 = \det\begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} = x_1y_2 - x_2y_1 = -p_2 \times p_1$$

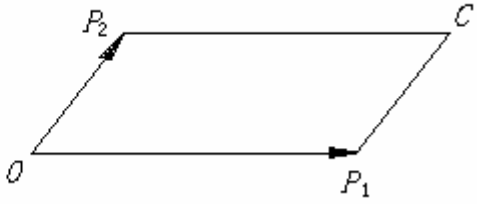
由叉积的定义式，可以看出两个有公共起点的向量的叉积，如果为正则由 p_1 转到 p_2 为逆时针方向，如果为负则为顺时针方向。特别的如果为零，则两向量共线，我们可以进一步去判断它们是共向的还是反向的。

下面是叉积的程序实现：

```
#include <utility>
using namespace std;
typedef pair<double,double> myPoint;
// return 1: clockwise, -1: counterclockwise, 0: on the same line;
int cross_product(myPoint & p1, myPoint & p2)
{
    double cp=p1.first*p2.second-p1.second*p2.first;
    if  (cp>0)  return 1;
    if  (cp<0)  return -1;
    return 0;
}
```



(图 3)



(图 4)

再谈一道叉积的应用的问题：

把叉积的定义再引申一步，如图 3，我们想求平行四边形的面积。于是有：

$$S = |OA||OB|\sin\angle\text{AOB} = |OA \times OB| = \begin{Vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{Vmatrix} = |p_1 \times p_2|$$

问题也就转化为了如图 4 所示的两个向量叉积的模的形式，由于是求模，不必考虑正负情况。

2.线段是否相交（Determining Intersections）

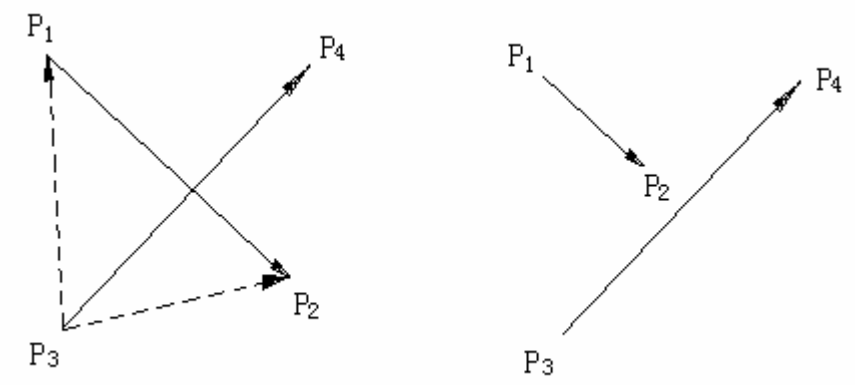
为了判断两条线段是否相交，我们提出一种新的快捷的方法。在这个方法中，将不必如从前在几何当中的做法那样，去解方程，进而看两条线段所在的直线的交点是否在线段上。之所以要避免这样的做法，原因也和前面提到的一样，计算机算除法相对较慢，另外也有较低的精确度。下面提到的这种方法，暂且叫它跨立（straddle），因为我还没有找出什么更好的名称来给它命名。

跨立的含义是：如果一条线段的一个端点在一条直线的一边，另一个端点在这条直线的另一端，我们就说这条线段跨立在这条直线上。显然，如果两条线段互相跨立，那它们一定是相交的。当然还有一些边值情况（boundary case），也就是说有可能线段的端点在另一条线段所在的直线上。

线段相交满足且只需满足如下两个条件就可以了：

1 两条线段相互跨立；

2 一条线段的一个端点在另一条线段上。（这是针对边值情况而言的）
我们进一点来说明一下如何知道，一条线段是跨立在另一条线段上的。



如左图所示，线段向量 P1P2 是跨立在 P3P4 上的。我们取端点 P3 为虚线所示向量的起点，P1 和 P2 为终点，则向量组 P3P1、P3P4 和向量组 P3P2、P3P4 的叉积符号是相反的，因为由 P3P1 转向 P3P4 为顺时针，而由 P3P2 转向 P3P4 则为逆时针。同样的，也可以判断出 P3P4 是跨立在 P1P2 上的。这样我们就可以知道，两条线段是相交的。再看一看右图，按照跨立的方法，很容易知道，这两条线段是不相交的。

下面是 C++的函数，用来判断两条线段是否相交。

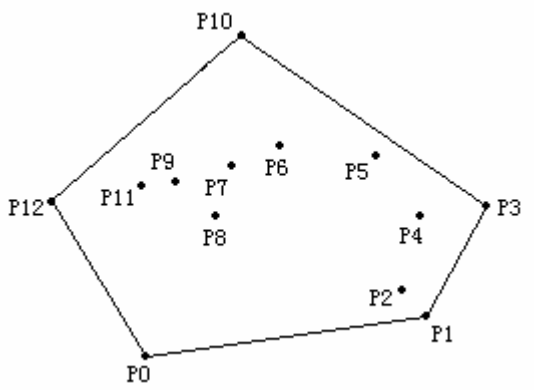
```
#include <utility>
using namespace std;

typedef pair<double,double> POINT;
//functionn dirction determines the direction that the segment
//p1p turns to p2p with respect to point p
//if return value is positive, means clockwise;
//if return value is negative, menas counter-clockwise;
// naught means on the same line;
double direction(POINT p,POINT p1,POINT p2)
{
    POINT v1,v2;
    v1.first =p2.first -p.first ;
    v1.second=p2.second-p.second;
    v2.first =p1.first -p.first;
    v2.second=p1.second-p.second;
    return v1.first*v2.second-v1.second*v2.first;
}
//on_segment determines whether the point p is on the segment p1p2
bool on_segment(POINT p,POINT p1,POINT p2)
{
    double min_x=p1.first <p2.first ?p1.first :p2.first ;
```

```
    double max_x=p1.first >p2.first ?p1.first :p2.first ;
    double min_y=p1.second<p2.second?p1.second:p2.second;
    double max_y=p1.second>p2.second?p1.second:p2.second;
    if (p.first >=min_x && p.first <=max_x
        && p.second>=min_y && p.second<=max_y) return true;
    else return false;
}
//function segment_intersect determines whether segment p1p2 and p3p4
// intersect  true yes and false no;
bool segment_intersect(POINT p1,POINT p2,POINT p3,POINT p4)
{
    double d1=direction(p3,p4,p1);
    double d2=direction(p3,p4,p2);
    double d3=direction(p1,p2,p3);
    double d4=direction(p1,p2,p4);
    if ((d1>0 && d2<0 || d1<0 && d2>0 ) &&
        ( d3>0 && d4<0 || d3<0 && d4>0))
        return true;
    else if (d1==0 && on_segment(p1,p3,p4))
        return true;
    else if (d2==0 && on_segment(p2,p3,p4))
        return true;
    else if (d3==0 && on_segment(p3,p1,p2))
        return true;
    else if (d4==0 && on_segment(p4,p1,p2))
        return true;
    else return false;
}
```

三、点集的性质（Point-set properities）

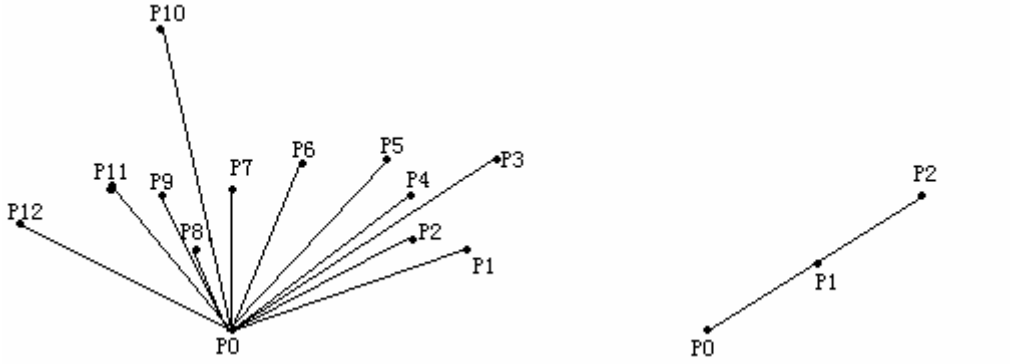
1.寻找凸包（Finding the convex hull）

凸包是一组点集中的子集，这一子集形成的凸多边形可以将点集中所有的点都围住，并且这一凸多边形的面积是最小的。我们用 Q 表示点集，用 CH（Q）表示点集 Q 的凸包。如下图所示：凸包为点 P10、P3、P1、P0 和 P12。则有 CH（Q）={P10，P3，P1，P0，P12}



这里，我们只介绍一种比较容易理解的寻找凸包的算法，称为打包法（Jarvis's march）。

具体算法是这样的，首先，我们找出点集中最下方的点，如果这样的点不止一个，就选用最左边的点（如 P0）。显然，这个点（P0）是凸包子集中的一个点。可以设想在 P0 处拴了一根皮筋的一端，另一端放在和 P0 成水平位置的右侧。现在，将皮筋，沿逆时针方向转动，首先会碰到 P1，这样就找到了另一个凸包子集中的点。以 P1 为中心，做和 P0 一样的事，会发现，我们将碰到 P3，又一个凸包的点。我们可以一直这样做下去，直到再一次遇到 P0，凸包就被找出来了。具体而言，在第一次找到 P0 点之后，以 P0 为每个矢量的起点，其它的点为矢量的终点，来比较任意两个矢量的转角，就可以对余下的点进行按极角排序。如下图：



看左图，按上所述，选 P0 为极角，将得到一组向量。先看向量 P0P1 和 P0P2，由于从 P0P1 转向 P0P2 为顺时针，于是认为点 P1 大于点 P2。对于右图中的特殊情况，由函数 direction()，得到的结果将为 0，但现在比较的结果应该是 P2 大于 P1（因为我们需要的是点 P2），于是，再用 on_segment(p2,p0,p1)，如果为真则 P2 大于 P1，这在下面的代码实现中有具体的体现。请大家认真思考这一边值情况。

下面是打包法具体的 C++代码实现：

```
#include <iostream>
#include <utility>
#include <vector>
#include <algorithm>

using namespace std;
typedef pair<double,double> POINT;
```

```
    // function dirction determines the direction that the segment
    //p1p turns to p2p with respect to point p
    //if return value is positive, means clockwise;
    //if return value is negative, menas counter-clockwise;
    // naught means on the same line;
    double direction(POINT p,POINT p1,POINT p2)
    {
        POINT v1,v2;
        v1.first =p2.first -p.first ;
        v1.second=p2.second-p.second;
        v2.first =p1.first -p.first;
        v2.second=p1.second-p.second;
        return v1.first*v2.second-v1.second*v2.first;
    }
    //function on_segment determines whether the point p is on the segment
p1p2
    bool on_segment(POINT p,POINT p1,POINT p2)
    {
        double min_x=p1.first <p2.first ?p1.first :p2.first ;
        double max_x=p1.first >p2.first ?p1.first :p2.first ;
        double min_y=p1.second<p2.second?p1.second:p2.second;
        double max_y=p1.second>p2.second?p1.second:p2.second;
        if (p.first >=min_x && p.first <=max_x
        && p.second>=min_y && p.second<=max_y) return true;
        else return false;
    }
    //point startPoint is the polor point that is needed for comparing
two other points;
    POINT startPoint;
    //function sortByPolorAngle provides the realizing of comparing two
points, which support
    //the STL function sort();
    bool sortByPolorAngle(const POINT & p1, const POINT & p2)
    {
        double d=direction(startPoint, p1, p2);
        if (d<0) return true;
        if (d>0) return false;
        if (d==0 && on_segment(startPoint, p1, p2) )return true;
        if (d==0 && on_segment(p2,startPoint,p1) ) return true;
        return false;
    }
    //here realizes the process of finding convex hull
```

```
void find_convex_hull(vector<POINT> & point)
{
    POINT p0=point[0];
    int k=0;
    for (int i=1;i<point.size();i++)
    {
        if (point[i].second<p0.second ||
            point[i].second==p0.second && point[i].first<p0.first)
            {
                p0=point[i];
                k=i;
            }
    }
    point.erase(point.begin()+k);
    point.insert(point.begin(),p0);
    vector<POINT> convex_hull;
    do {
        convex_hull.push_back(point[0]);
        startPoint=point[0];
        point.erase(point.begin());
        sort(point.begin(),point.end(),sortByPolorAngle);
        if (point[0]==convex_hull[0]) break;
        point.push_back(convex_hull[convex_hull.size()-1]);
    } while (1);
    for (int i=0;i<convex_hull.size();i++)
    {
        cout<<convex_hull[i].first<<' '
            <<convex_hull[i].second<<endl;
    }
}
```

习　题

AREA

Jerry, a middle school student, addicts himself to mathematical research. Maybe the problems he has thought are really too easy to an expert. But as an amateur, especially as a 15-year-old boy, he had done very well. He is so rolling in thinking the mathematical problem that he is easily to try to solve every problem he met in a mathematical way. One day, he found a piece of paper on the desk. His younger sister, Mary, a four-year-old girl, had drawn some lines. But those lines formed a special kind of concave polygon by accident as Fig. 1 shows.

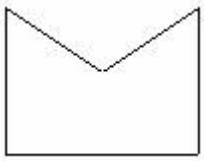


Fig. 1 The lines his sister had drawn

"Great!" he thought, "The polygon seems so regular. I had just learned how to calculate the area of triangle, rectangle and circle. I'm sure I can find out how to calculate the area of this figure." And so he did. First of all, he marked the vertexes in the polygon with their coordinates as Fig. 2 shows. And then he found the result--0.75 effortless.

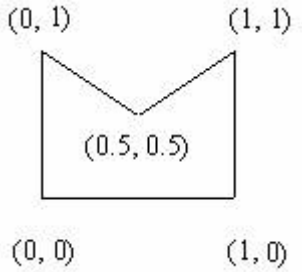


Fig.2 The polygon with the coordinates of vertexes

Of course, he was not satisfied with the solution of such an easy problem. "Mmm, if there's a random polygon on the paper, then how can I calculate the area?" he asked himself. Till then, he hadn't found out the general rules on calculating the area of a random polygon. He clearly knew that the answer to this question is out of his competence. So he asked you, an erudite expert, to offer him help. The kind behavior would be highly appreciated by him.

Input

The input data consists of several figures. The first line of the input for each figure contains a single integer n, the number of vertexes in the figure. ($0 \le n \le 1000$).

In the following n lines, each contain a pair of real numbers, which describes the coordinates of the vertexes, (xi, yi). The figure in each test case starts from the first vertex to the second one, then from the second to the third, …… and so on. At last, it closes from the nth vertex to the first one.

The input ends with an empty figure (n = 0). And this figure not be processed.

Output

As shown below, the output of each figure should contain the figure number and a colon followed by the area of the figure or the string "Impossible".

If the figure is a polygon, compute its area (accurate to two fractional digits). According to the input vertexes, if they cannot form a polygon (that is, one line intersects with another which shouldn't be adjoined with it, for example, in a figure with four lines, the first line intersects with the third one), just display "Impossible", indicating the figure can't be a polygon. If the amount of the vertexes is not enough to form a closed polygon, the output message should be "Impossible" either.

Print a blank line between each test cases.

Sample Input

```
5
0 0
0 1
0.5 0.5
1 1
1 0
4
0 0
0 1
1 0
1 1
0
```

Sample Output

```
Figure 1: 0.75
Figure 2: Impossible
```

第四章 动态规划

本章介绍了动态规划的基本思想和基本步骤，通过实例研究了利用动态规划设计算法的具体途径，讨论了动态规划的一些实现技巧，并将动态规划和其他一些算法作了比较，最后还简单介绍了动态规划的数学理论基础和当前最新的研究成果。

一、引言——由一个问题引出的算法

考虑以下问题

[例 1]最短路径问题

现有一张地图，各结点代表城市，两结点间连线代表道路，线上数字表示城市间的距离。如图 1 所示，试找出从结点 A 到结点 E 的最短距离。

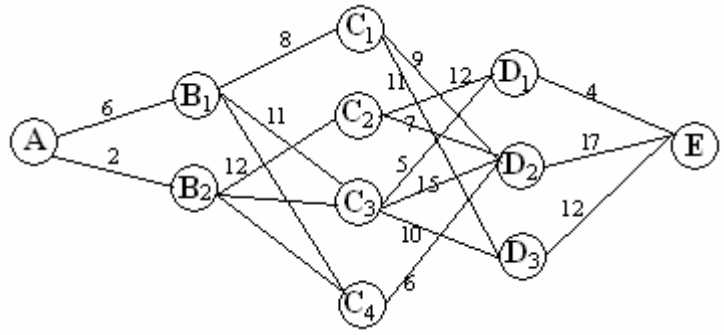


图 1

我们可以用深度优先搜索法来解决此问题，该问题的递归式为

$$MinDistance(v)=\min_{u\in\delta(v)}\{w(v,u)+MinDistance(u)\}$$

其中 $\delta(v)$ 是与 v 相邻的节点的集合，w(v,u)表示从 v 到 u 的边的长度。

具体算法如下：

```
function MinDistance(v):integer;
begin
    if v=E then return 0 else
    begin
       min:=maxint;
       for 所有没有访问过的节点 i do
          if v 和 i 相邻 then
          begin
             标记 i 访问过了;
```

```
                t:=v 到 i 的距离+MinDistance(i);
                标记 i 未访问过;
                if t<min then min=t;
            end;
        end;
    end;
```

开始时标记所有的顶点未访问过，MinDistance(A)就是从 A 到 E 的最短距离。

这个程序的效率如何呢？我们可以看到，每次除了已经访问过的城市外，其他城市都要访问，所以时间复杂度为 O(n!)，这是一个“指数级”的算法，那么，还有没有更好的算法呢？

首先，我们来观察一下这个算法。在求从 B1 到 E 的最短距离的时候，先求出从 C2 到 E 的最短距离；而在求从 B2 到 E 的最短距离的时候，又求了一遍从 C2 到 E 的最短距离。也就是说，从 C2 到 E 的最短距离我们求了两遍。同样可以发现，在求从 C1、C2 到 E 的最短距离的过程中，从 D1 到 E 的最短距离也被求了两遍。而在整个程序中，从 D1 到 E 的最短距离被求了四遍。如果在求解的过程中，同时将求得的最短距离"记录在案"，随时调用，就可以避免这种情况。于是，可以改进该算法，将每次求出的从 v 到 E 的最短距离记录下来，在算法中递归地求 MinDistance(v) 时先检查以前是否已经求过了 MinDistance(v)，如果求过了则不用重新求一遍，只要查找以前的记录就可以了。这样，由于所有的点有 n 个，因此不同的状态数目有 n 个，该算法的数量级为 O(n)。

更进一步，可以将这种递归改为递推，这样可以减少递归调用的开销。

请看图 1，可以发现，A 只和 Bi 相邻，Bi 只和 Ci 相邻,...，依此类推。这样，我们可以将原问题的解决过程划分为 4 个阶段，设 S1={A},S2={B1,B2},S3={C1,C2,C3,C4},S4={D1,D2,D3}，Fk(u)表示从 Sk 中的点 u 到 E 的最短距离，则

$$F_k(u)=\min_{u\in S_k,v\in S_{k+1}\cap\delta(u)}\{w(u,v)+F_{k+1}(v)\}$$

$$F_5(E)=0$$

显然可以递推地求出 F1(A)，也就是从 A 到 E 的最短距离。这种算法的复杂度为 O(n)，因为所有的状态总数（节点总数）为 n，对每个状态都只要遍历一次，而且程序很简洁。

```
procedure DynamicProgramming;
    begin
        F5[E]:=0;
        for i:=4 downto 1 do
            for each u ∈Sk do
            begin
                Fk[u]:=无穷大;
                for each v∈Sk+1∩δ(u) do
                    if Fk[u]>w(u,v)+Fk+1[v] then
                        Fk[u]:=w(u,v)+Fk+1[v];
            end;
            输出 F1[A];
```

```
end;
```

这种高效算法，就是**动态规划算法**。

二、动态规划的基本概念

2.1 动态规划的发展及研究内容

动态规划(dynamic programming)是运筹学的一个分支，是求解**决策过程(decision process)**最优化的数学方法。20 世纪 50 年代初美国数学家 R.E.Bellman 等人在研究**多阶段决策过程(multistep decision process)**的优化问题时，提出了著名的**最优化原理(principle of optimality)**，把多阶段过程转化为一系列单阶段问题，逐个求解，创立了解决这类过程优化问题的新方法——动态规划。1957 年出版了他的名著 *Dynamic Programming*，这是该领域的第一本著作。

动态规划问世以来，在经济管理、生产调度、工程技术和最优控制等方面得到了广泛的应用。例如最短路线、库存管理、资源分配、设备更新、排序、装载等问题，用动态规划方法比用其它方法求解更为方便。

虽然动态规划主要用于求解以时间划分阶段的动态过程的优化问题，但是一些与时间无关的静态规划(如线性规划、非线性规划)，只要人为地引进时间因素，把它视为多阶段决策过程，也可以用动态规划方法方便地求解。

2.2 多阶段决策问题

多阶段决策过程，是指这样的一类特殊的活动过程，问题可以按时间顺序分解成若干相互联系的阶段，在每一个阶段都要做出决策，全部过程的决策是一个决策序列。要使整个活动的总体效果达到最优的问题，称为**多阶段决策问题**。

引言中的例子是一个多阶段决策问题的例子，下面是另一个多阶段决策问题的例子：

[例 2] 生产计划问题

工厂生产某种产品，每单位(千件)的成本为 1(千元)，每次开工的固定成本为 3(千元)，工厂每季度的最大生产能力为 6(千件)。经调查，市场对该产品的需求量第一、二、三、四季度分别为 2，3，2，4(千件)。如果工厂在第一、二季度将全年的需求都生产出来，自然可以降低成本(少付固定成本费)，但是对于第三、四季度才能上市的产品需付存储费，每季每千件的存储费为 0.5(千元)。还规定年初和年末这种产品均无库存。试制订一个生产计划，即安排每个季度的产量，使一年的总费用(生产成本和存储费)最少。

2.3 决策过程的分类

根据过程的时间变量是离散的还是连续的，分为**离散时间决策过程(discrete-time**

decision process)，即多阶段决策过程和**连续时间决策过程(continuous-time decision process)**；根据过程的演变是确定的还是随机的，分为**确定性决策过程(deterministic decision process)**和**随机性决策过程(stochastic decision process)**，其中应用最广的是确定性多阶段决策过程。

三、动态规划模型的基本要素

一个多阶段决策过程最优化问题的动态规划模型通常包含以下要素：

1.阶段

阶段(step)是对整个过程的自然划分。通常根据时间顺序或空间特征来划分阶段，以便按阶段的次序解优化问题。阶段变量一般用 k=1,2,..,n 表示。在例 1 中由 A 出发为 k=1，由 Bi(i=1,2)出发为 k=2，依此下去从 Di(i=1,2,3)出发为 k=4，共 n=4 个阶段。在例 2 中按照第一、二、三、四季度分为 k=1,2,3,4，共 4 个阶段。

2.状态

状态(state)表示每个阶段开始时过程所处的自然状况。它应该能够描述过程的特征并且具有无后向性，即当某阶段的状态给定时，这个阶段以后过程的演变与该阶段以前各阶段的状态无关，即每个状态都是过去历史的一个完整总结。通常还要求状态是直接或间接可以观测的。

描述状态的变量称**状态变量(state variable)**。变量允许取值的范围称**允许状态集合(set of admissible states)**。用 xk 表示第 k 阶段的状态变量，它可以是一个数或一个向量。用 Xk 表示第 k 阶段的允许状态集合。在例 1 中 x2 可取 B1，B2，X2={B1,B2}。

n 个阶段的决策过程有 n+1 个状态变量，xn+1 表示 xn 演变的结果，在例 1 中 x5 取 E。

根据过程演变的具体情况，状态变量可以是离散的或连续的。为了计算的方便有时将连续变量离散化；为了分析的方便有时又将离散变量视为连续的。

状态变量简称为**状态**。

3.决策

当一个阶段的状态确定后，可以作出各种选择从而演变到下一阶段的某个状态，这种选择手段称为**决策(decision)**，在最优控制问题中也称为**控制(control)**。

描述决策的变量称**决策变量(decision variable)**。变量允许取值的范围称**允许决策集合(set of admissible decisions)**。用 uk(xk)表示第 k 阶段处于状态 xk 时的决策变量，它是 xk 的函数，用 Uk(xk)表示了 xk 的允许决策集合。在例 1 中 u2(B1)可取 C1,C2,C3。

决策变量简称**决策**。

4.策略

决策组成的序列称为**策略(policy)**。由初始状态 x1 开始的全过程的策略记作 p1n(x1)，即 p1n(x1)={u1(x1),u2(x2),...，un(xn)}。由第 k 阶段的状态 xk 开始到终止状态的后部子过程的策略记作 pkn(xk)，即 pkn(xk)={uk(xk),uk+1(xk+1),...，un(xn)}。类似地，由第 k 到第 j 阶

段的子过程的策略记作 pkj(xk)={uk(xk),uk+1(xk+1),...，uj(xj)}。对于每一个阶段 k 的某一给定的状态 xk，可供选择的策略 pkj(xk)有一定的范围，称为**允许策略集合(set of admissible policies)**，用 P1n(x1),Pkn(xk),Pkj(xk)表示。

5.状态转移方程

在确定性过程中，一旦某阶段的状态和决策为已知，下阶段的状态便完全确定。用**状态转移方程(equation of state)**表示这种演变规律，写作

$$x_{k+1} = T_k(x_k, u_k(x_k)) \quad , k = 1,2,\cdots,n \qquad (1)$$

在例 1 中状态转移方程为：xk+1=uk(xk)

6.指标函数和最优值函数

指标函数(objective function)是衡量过程优劣的数量指标，它是关于策略的数量函数，从阶段 k 到阶段 n 的指标函数用 Vkn(xk,pkn(xk))表示，k=1,2,...,n。

能够用动态规划解决的问题的指标函数应具有**可分离性**，即 Vkn 可表为 xk,uk,Vk+1 n 的函数，记为：

$$V_{kn}(x_k, u_k, x_{k+1}, \cdots, x_{n+1}) = \varphi_k(x_k, u_k, V_{k+1\ n}(x_{k+1}, \cdots, x_{n+1})) \qquad (2)$$

其中函数 φ_k 是一个关于变量 Vk+1 n **单调递增**的函数。这一性质保证了**最优化原理(principle of optimality)**的成立，是动态规划的适用前提。

过程在第 j 阶段的阶段指标取决于状态 xj 和决策 uj，用 vj(xj,uj)表示。阶段 k 到阶段 n 的指标由 vj(j=k,k+1,..n)组成，常见的形式有：

阶段指标之和，即

$$V_{kn}(x_k, u_k, \cdots, x_{n+1}) = \sum_{j=k}^{n} v_j(x_j, u_j) \qquad (3)$$

阶段指标之积，即

$$V_{kn}(x_k, u_k, \cdots, x_{n+1}) = \prod_{j=k}^{n} v_j(x_j, u_j) \qquad (4)$$

阶段指标之极大(或极小)，即

$$V_{kn}(x_k, u_k, \cdots, x_{n+1}) = \max_{k \le j \le n} v_j(x_j, u_j) \quad or \min_{k \le j \le n} v_j(x_j, u_j) \qquad (5)$$

这些形式下第 k 到第 j 阶段子过程的指标函数为 Vkj(xk,uk,xk+1,...,xj+1)。可以发现，上述(3)-(5)三个指标函数的形式都满足最优性原理。在例 1 中指标函数为(3)的形式，其中 vj(xj,uj)是边<xj,uj(xj)>的权（边的长度）,uj(xj)表示从 xj 出发根据决策 uj(xj)下一步所到达的节点。

根据状态转移方程，指标函数 Vkn 还可以表示为状态 xk 和策略 pkn 的函数，即 Vkn(xk,pkn)。在 xk 给定时指标函数 Vkn 对 pkn 的最优值称为**最优值函数(optimal value function)**，记作 fk(xk)，即

$$f_k(x_k) = \underset{p_{kn} \in P_{kn}(x_k)}{opt} V_{kn}(x_k, p_{kn}) \qquad (6)$$

其中 opt 可根据具体情况取 max 或 min。上式的意义是，对于某个阶段 k 的某个状态 xk，从该阶段 k 到最终目标阶段 n 的最优指标函数值等于从 xk 出发取遍所有能策略 pkn 所得到的最优指标值中最优的一个。

7.最优策略和最优轨线

使指标函数 Vkn 达到最优值的策略是从 k 开始的后部子过程的最优策略，记作 pkn*={uk*,..un*},p1n*又是全过程的最优策略，简称**最优策略(optimal policy)**。从初始状态 x1(=x1*)出发，过程按照 p1n*和状态转移方程演变所经历的状态序列{x1*,x2*,..,xn+1*}称**最优轨线(optimal trajectory)**。

四、动态规划的基本定理和基本方程

动态规划发展的早期阶段，从简单逻辑出发给出了所谓最优性原理，然后在最优策略存在的前提下导出基本方程，再由这个方程求解最优策略。后来在动态规划的应用过程中发现，**最优性原理不是对任何决策过程普遍成立，它与基本方程不是无条件等价，二者之间也不存在任何确定的蕴含关系**。基本方程在动态规划中起着更为本质的作用。

[基本定理]

对于初始状态 x1∈X1，策略 p1n*={u1*,..un*}是最优策略的充要条件是对于任意的 k,1<k<=n,有

$$V_{1n}(x_1, p_{1n}^*) = \varphi\left(\underset{p_{1,k-1} \in P_{1,k-1}(x_1)}{opt} [V_{1,k-1}(x_1, p_{1,k-1})], \underset{p_{kn} \in P_{kn}(x_k)}{opt} [V_{kn}(x_k, p_{kn})] \right) \qquad (8)$$

[推论]

若 p1n*∈P1n(x1)是最优策略，则对于任意的 k,1<k<n，它的子策略 pkn*对于由 x1 和 p1,k-1*确定的以 xk*为起点的第 k 到 n 后部子过程而言，也是最优策略。

上述推论称为**最优化原理**，它给出了最优策略的必要条件，通常略述为：不论过去的状态和决策如何，对于前面的决策形成的当前的状态而言，余下的各个决策必定构成最优策略。

根据基本定理的推论可以得到**动态规划的基本方程**：

$$\begin{cases} f_k(x_k) = \underset{u_k \in U_k(x_k)}{opt} \{\varphi(v_k(x_k, u_k), f_{k+1}(x_{k+1}))\}, \ x_{k+1} = T_k(x_k, u_k), \ k = 1, 2, \cdots, n \\ f_{n+1}(x_{n+1}) = \delta(x_{n+1}) \end{cases} \qquad (9)$$

其中 $f_{n+1}(x_{n+1}) = \delta(x_{n+1})$ 是决策过程的终端条件，δ 为一个已知函数。当 xn+1 只取固定的状态时称固定终端；当 xn+1 可在终端集合 Xn+1 中变动时称自由终端。最终要

求的最优指标函数满足(10)式：

$$opt\{V_{1n}\} = \underset{x_1 \in X_1}{opt}\{f(x_1)\} \qquad (10)$$

(9)式是一个递归公式，如果目标状态确定，当然可以直接利用该公式递归求出最优值（这种递归方法将在后文介绍，称作**备忘录法**），但是一般在实际应用中我们通常将该递归公式改为递推公式求解，这样一般效率会更高一些。

五、动态规划的适用条件

任何思想方法都有一定的局限性，超出了特定条件，它就失去了作用。同样，动态规划也并不是万能的。适用动态规划的问题必须满足最优化原理和无后效性。

5.1 最优化原理（最优子结构性质）

最优化原理可这样阐述：一个最优化策略具有这样的性质，不论过去状态和决策如何，对前面的决策所形成的状态而言，余下的诸决策必须构成最优策略。简而言之，一个最优化策略的子策略总是最优的。一个问题满足最优化原理又称其具有**最优子结构性质**。

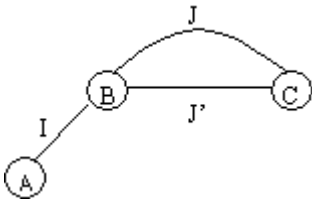


图 2

例如图 2 中，若路线 I 和 J 是 A 到 C 的最优路径，则根据最优化原理，路线 J 必是从 B 到 C 的最优路线。这可用反证法证明：假设有另一路径 J'是 B 到 C 的最优路径，则 A 到 C 的路线取 I 和 J'比 I 和 J 更优，矛盾。从而证明 J'必是 B 到 C 的最优路径。

最优化原理是动态规划的基础，任何问题，如果失去了最优化原理的支持，就不可能用动态规划方法计算。动态规划的最优化理在其指标函数的可分离性和单调性中得到体现。根据最优化原理导出的动态规划基本方程是解决一切动态规划问题的基本方法。

可以看出，例 1 是满足最优化原理的。

5.2 无后向性

将各阶段按照一定的次序排列好之后，对于某个给定的阶段状态，它以前各阶段的状态无法直接影响它未来的决策，而只能通过当前的这个状态。换句话说，每个状态都是过去历史的一个完整总结。这就是**无后向性**，又称为**无后效性**。

如果用前面的记号来描述无后向性，就是：对于确定的 xk，无论 p1,k-1 如何，最优子策略 pkn*是唯一确定的，这种性质称为无后向性。

[例 3] Bitonic 旅行路线问题

欧几里德货郎担问题是对平面给定的 n 个点确定一条连结各点的、闭合的最短游历路线问题。图 3(a)给出了七个点问题的解。Bitonic 旅行路线问题是欧几里德货郎担问题的简化，这种旅行路线先从最左边开始，严格地由左至右到最右边的点，然后再严格地由右至左到出发点，求路程最短的路径长度。图 3（b）给出了七个点问题的解。

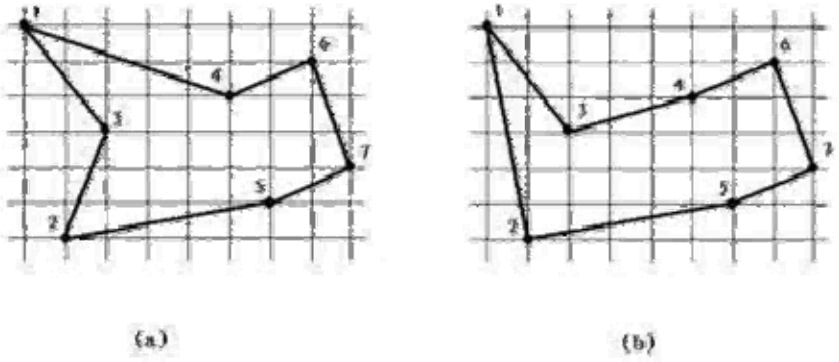


图 3

这两个问题看起来很相似。但实质上是不同的。为了方便讨论，我将每个顶点标记了号码。由于必然经过最右边的顶点 7，所以一条路(P1-P2)可以看成两条路（P1-7）与(P2-7)的结合。所以，这个问题的状态可以用两条道路结合的形式表示。我们可以把这些状态中，两条路中起始顶点相同的状态归于一个阶段，设为阶段[P1,P2]。

那么，对于 Bitonic 旅行路线问题来说，阶段[P1,P2]如果可以由阶段[Q1,Q2]推出，则必须满足的条件就是:P1<Q1 或 P2<Q2。例如，阶段[3,4]中的道路可以由阶段[3,5]中的道路加一条边 4-5 得出，而阶段[3,5]的状态却无法由阶段[3,4]中的状态得出，因为 Bitonic 旅行路线要求必须严格地由左到右来旅行。所以如果我们已经知道了阶段[3,4]中的状态，则阶段[3,5]中的状态必然已知。因此我们可以说，Bitonic 问题满足**无后向性**，可以用动态规划来解决。

有些问题乍一看好像有后向性，但如果按照某种合理的方式重新划分阶段，就可以发现其本质上是无后向性的，所以关键是阶段的合理划分。

5.3 子问题的重叠性

在例 1 中我们看到，动态规划将原来具有指数级复杂度的搜索算法改进成了具有多项式时间的算法。其中的关键在于**解决冗余**，这是动态规划算法的根本目的。动态规划实质

上是一种以空间换时间的技术，它在实现的过程中，不得不存储产生过程中的各种状态，所以它的空间复杂度要大于其它的算法。以 Bitonic 旅行路线问题为例，这个问题也可以用搜索算法来解决。动态规划的时间复杂度为 O(n2)，搜索算法的时间复杂度为 O(n!) ，但从空间复杂度来看，动态规划算法为 O(n2)，而搜索算法为 O(n)，搜索算法反而优于动态规划算法。选择动态规划算法是因为动态规划算法在空间上可以承受，而搜索算法在时间上却无法承受，所以我们舍空间而取时间。

设原问题的规模为 n，容易看出，当子问题树中的子问题总数是 n 的超多项式函数，而不同的子问题数只是 n 的多项式函数时，动态规划法显得特别有意义，此时动态规划法具有线性时间复杂性。所以，能够用动态规划解决的问题还有一个显著特征：**子问题的重叠性**。这个性质并不是动态规划适用的必要条件，但是如果该性质无法满足，动态规划算法同其他算法相比就不具备优势。

六、动态规划的基本思想

前文主要介绍了动态规划的一些理论依据，我们将前文所说的具有明显的阶段划分和状态转移方程的动态规划称为**标准动态规划**，这种标准动态规划是在研究多阶段决策问题时推导出来的，具有严格的数学形式，适合用于理论上的分析。在实际应用中，许多问题的阶段划分并不明显，这时如果刻意地划分阶段法反而麻烦。一般来说，只要该问题可以划分成规模更小的子问题，并且原问题的最优解中包含了子问题的最优解（即满足最优子化原理），则可以考虑用动态规划解决。

动态规划的实质是分治思想和**解决冗余**，因此，**动态规划**是一种将问题实例分解为更小的、相似的子问题，并存储子问题的解而避免计算重复的子问题，以解决最优化问题的算法策略。

由此可知，动态规划法与分治法和贪心法类似，它们都是将问题实例归纳为更小的、相似的子问题，并通过求解子问题产生一个全局最优解。其中贪心法的当前选择可能要依赖已经作出的所有选择，但不依赖于有待于做出的选择和子问题。因此贪心法自顶向下，一步一步地作出贪心选择；而分治法中的各个子问题是独立的 (即不包含公共的子子问题)，因此一旦递归地求出各子问题的解后，便可自下而上地将子问题的解合并成问题的解。但不足的是，如果当前选择可能要依赖子问题的解时，则难以通过局部的贪心策略达到全局最优解；如果各子问题是不独立的，则分治法要做许多不必要的工作，重复地解公共的子问题。

解决上述问题的办法是利用动态规划。该方法主要应用于最优化问题，这类问题会有多种可能的解，每个解都有一个值，而动态规划找出其中最优(最大或最小)值的解。若存在若干个取最优值的解的话，它**只取其中的一个**。在求解过程中，该方法也是通过求解局部子问题的解达到全局最优解，但与分治法和贪心法不同的是，动态规划允许这些子问题不独立，(亦即各子问题可包含公共的子子问题)也允许其通过自身子问题的解作出选择，该方法对每一个子问题只解一次，并将结果保存起来，避免每次碰到时都要重复计算。

因此，动态规划法所针对的问题有一个显著的特征，即它所对应的子问题树中的子问题呈现大量的重复。动态规划法的关键就在于，**对于重复出现的子问题，只在第一次遇到时加以求解，并把答案保存起来，让以后再遇到时直接引用，不必重新求解。**

七、动态规划算法的基本步骤

设计一个标准的动态规划算法，通常可按以下几个步骤进行：

1. 划分阶段：按照问题的时间或空间特征，把问题分为若干个阶段。注意这若干个阶段一定要是有序的或者是可排序的（即无后向性），否则问题就无法用动态规划求解。

2. 选择状态：将问题发展到各个阶段时所处于的各种客观情况用不同的状态表示出来。当然，状态的选择要满足无后效性。

3. 确定决策并写出状态转移方程：之所以把这两步放在一起，是因为决策和状态转移有着天然的联系，状态转移就是根据上一阶段的状态和决策来导出本阶段的状态。所以，如果我们确定了决策，状态转移方程也就写出来了。但事实上，我们常常是反过来做，根据相邻两段的各状态之间的关系来确定决策。

4. 写出规划方程（包括边界条件）：动态规划的基本方程是规划方程的通用形式化表达式。一般说来，只要阶段、状态、决策和状态转移确定了，这一步还是比较简单的。

动态规划的主要难点在于理论上的设计，一旦设计完成，实现部分就会非常简单。根据动态规划的基本方程可以直接递归计算最优值，但是一般将其改为递推计算，实现的大体上的框架如下：

标准动态规划的基本框架

```
1.  对 fn+1(xn+1)初始化;    {边界条件}
2.  for k:=n downto 1 do
3.     for 每一个 xk∈Xk do
4.       for 每一个 uk∈Uk(xk) do
          begin
5.          fk(xk):=一个极值;                {∞或－∞}
6.          xk+1:=Tk(xk,uk);                 {状态转移方程}
7.          t:=φ(fk+1(xk+1),vk(xk,uk));      {基本方程(9)式}
8.          if  t 比 fk(xk)更优 then fk(xk):=t; {计算 fk(xk)的最优值}
          end;
9.  t:=一个极值;                            {∞或－∞}
10. for 每一个 x1∈X1 do
11.    if f1(x1)比 t 更优 then t:=f1(x1);     {按照 10 式求出最优指标}
12. 输出 t;
```

但是，实际应用当中经常不显式地按照上面步骤设计动态规划，而是按以下几个步骤进行：

1. 分析最优解的性质，并刻划其结构特征。
2. 递归地定义最优值。
3. 以自底向上的方式或自顶向下的记忆化方法（备忘录法）计算出最优值。

4. 根据计算最优值时得到的信息，构造一个最优解。

步骤(1)--(3)是动态规划算法的基本步骤。在只需要求出最优值的情形，步骤(4)可以省略，若需要求出问题的一个最优解，则必须执行步骤(4)。此时，在步骤(3)中计算最优值时，通常需记录更多的信息，以便在步骤(4)中，根据所记录的信息，快速地构造出一个最优解。

八、动态规划的实例分析

下面我们将通过实例来分析动态规划的设计步骤和具体应用。

例 1 生产计划问题

问题描述

工厂生产某种产品，每单位(千件)的成本为 1(千元)，每次开工的固定成本为 3(千元)，工厂每季度的最大生产能力为 6(千件)。经调查，市场对该产品的需求量第一、二、三、四季度分别为 2，3，2，4(千件)。如果工厂在第一、二季度将全年的需求都生产出来，自然可以降低成本(少付固定成本费)，但是对于第三、四季度才能上市的产品需付存储费，每季每千件的存储费为 0.5(千元)。还规定年初和年末这种产品均无库存。试制订一个生产计划，即安排每个季度的产量，使一年的总费用(生产成本和存储费)最少。

参考解答

这是一个明显的多阶段问题，我们按照计划时间自然划分阶段，状态定义为每阶段开始时的存储量 xk，决策为每个阶段的产量 uk，记每个阶段的需求量（已知）为 dk，则状态转移方程为：

$$x_{k+1} = x_k + u_k - d_k, x_k \geq 0, k = 1,2,\cdots,n$$

设每个阶段开工固定成本费用为 a，生产单位数量产品的成本为 b，每阶段单位数量产品的存储费用为 c ，阶段指标为阶段的生产成本费用和存储费用之和，即：

$$v_k(x_k,u_k) = cx_k + \begin{cases} a + bu_k, u_k > 0 \\ 0 \quad\quad , u_k = 0 \end{cases}$$

指标函数 Vkn 为 vk 之和，最优值函数 fk(xk)为从第 k 阶段的状态 xk 出发到过程终结的最小费用，满足

$$f_k(x_k) = \min_{u_k \in U_k}[v_k(x_k,u_k) + f_{k+1}(x_{k+1})], k = n,\cdots,1$$

其中允许决策集合 Uk 由每阶段的最大生产能力决定，设过程终结时允许存储量为 x0n+1，则终端条件为：

$$f_{n+1}(x_{n+1}^0) = 0$$

将以上各式代入到标准动态规划的框架中，就可以求得最优解。

例 2 Bitonic 旅行路线问题

问题描述

欧几里德货郎担问题是对平面给定的 n 个点确定一条连结各点的、闭合的游历路线问题。图 1(a)给出了七个点问题的解。Bitonic 旅行路线问题是欧几里德货郎担问题的简化，这种旅行路线先从最左边开始，严格地由左至右到最右边的点，然后再严格地由右至左到出发点，求路程最短的路径长度。图 1（b）给出了七个点问题的解。

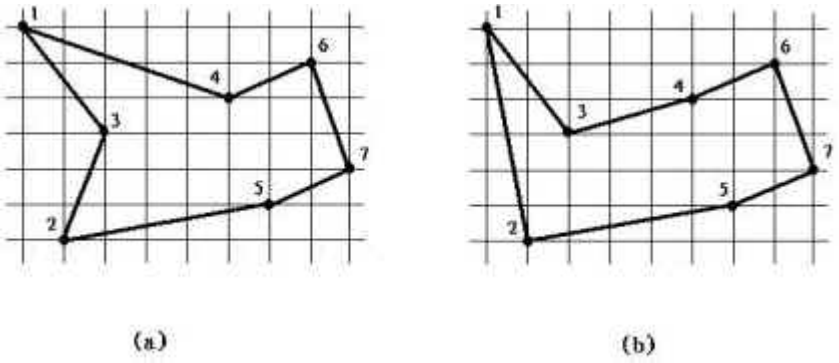


图 1

请设计一种多项式时间的算法，解决 Bitonic 旅行路线问题。

参考解答

首先将 n 个点按 X 坐标递增的顺序排列成一个序列 L=<点 1，点 2，…，点 n>。显然 L[n]为最右点，L[1]为最左点。由于点 1 往返经过二次(出发一次，返回一次)，因此点 1 拆成两个点：L[0]=L[1]=点 1。

设：

Wi,j -- 边<i,j>的距离；

Wi..j -- j-i 条连续边<i,i+l>,<i+l,i+2>，…,<j-2,j-1>,<j-1,j>的距离和$\sum_{i\le k\le j-1} W_{k,k+1}$。由点 i 至点 j 的连续边组成的路径称为连续路径；

d[i] --从点 i 出发由左至右直到 n 点后再由右至左到达点 i-1 的最短距离。d[i]的递归式如下:

$$d[i] = \min_{i\le j\le N-1}\{W_{i\cdots j} + d[j+1] + W_{j+1,i-1}\} \qquad (1 \le i \le N)$$

我们专门设置了一张记忆表 k[i](1≤i≤n)，记下使得 d[i]最小的 J 值。显然递归式的边界 d[n]=Wn,n-1 ，k[n]=n 。

由 d[i]的递归式和记忆表 k[i]的定义可以看出，在由点 i-1 至点 i 的最短路上，点 i 至 k[i]的子路径上的点序号是逐一递增的，为由左至右方向上的连续子路径。按照递归定义，若 k[i]≠N 且 k[k[i]+1]≠N，则点 k[k[i]+1]+1 至点 k[k[k[i]+1]+1]也是由左至右方向上的连续子路径；否则 k[k[i]+1]至 k[i]+1 为由右至左方向上的连续子路径，该子路径上的点序号是逐一递减的。

显然要使 d[i]最小，必须分别使 d[i+1]…d[n-1],d[n]最小,d[i]包含了 d[i+1]…d[n]最小的

子问题，因此该问题具有最优子结构和重叠子问题性质，满足动态规划法适用的条件。为了提高算法效率，我们从 d[n]出发，运用动态规划方法依次求 d[n-1],d[n-2],…,d[1]，充分利用重叠子问题。最后求得的 d[1]便是最短游历路线的距离；从点 1 出发，顺着记忆表 k 的指示可以递归递输出这条游历路线。

例 3 计算矩阵连乘积

问题描述

在科学计算中经常要计算矩阵的乘积。矩阵 A 和 B 可乘的条件是矩阵 A 的列数等于矩阵 B 的行数。若 A 是一个 p×q 的矩阵，B 是一个 q×r 的矩阵，则其乘积 C=AB 是一个 p×r 的矩阵。其标准计算公式为：

$$C_{ij}=\sum_{k=1}^{y}A_{ik}B_{kj} \qquad \text{其中} 1\le i\le p,\ 1\le j\le r$$

由该公式知计算 C=AB 总共需要 pqr 次的数乘。

现在的问题是，给定 n 个矩阵{A1,A2,…,An}。其中 Ai 与 Ai+1 是可乘的，i=1,2,…,n-1。要求计算出这 n 个矩阵的连乘积 A1A2…An。

由于矩阵乘法满足结合律，故连乘积的计算可以有许多不同的计算次序。这种计算次序可以用加括号的方式来确定。若一个矩阵连乘积的计算次序已完全确定，也就是说该连乘积已完全加括号，则我们可以通过反复调用两个矩阵相乘的标准算法计算出矩阵连乘积。完全加括号的矩阵连乘积可递归地定义为：

1. 单个矩阵是完全加括号的；
2. 若矩阵连乘积 A 是完全加括号的，则 A 可表示为两个完全加括号的矩阵连乘积 B 和 C 的乘积并加括号，即 A=(BC)。

例如，矩阵连乘积 A1A2A3 A4 可以有以下 5 种不同的完全加括号方式：

(A1(A2(A3A4)))，
(A1((A2A3)A4)),
((A1A2)(A3A4)),
((A1(A2A3))A4),
(((A1A2)A3)A4)。

每一种完全加括号方式对应于一种矩阵连乘积的计算次序，而这种计算次序与计算矩阵连乘积的计算量有着密切的关系。

为了说明在计算矩阵连乘积时加括号方式对整个计算量的影响，我们来看一个计算 3 个矩阵{A1，A2，A3}的连乘积的例子。设这 3 个矩阵的维数分别为 10×100，100×5 和 5×50。若按第一种加括号方式((A1A2)A3)来计算，总共需要 10×100×5+10×5×50=7500 次的数乘。若按第二种加括号方式(A1(A2A3))来计算，则需要的数乘次数为 100×5×50+10×100×50=75000。第二种加括号方式的计算量是第一种加括号方式的计算量的 10 倍。由此可见，在计算矩阵连乘积时，加括号方式，即计算次序对计算量有很大影响。

于是，人们自然会提出矩阵连乘积的最优计算次序问题，即对于给定的相继 n 个矩阵

{A1,A2,…,An}(其中 Ai 的维数为 pi-1×pi ，i=1,2,…,n)，如何确定计算矩阵连乘积 A1A2…An 的一个计算次序(完全加括号方式)，使得依此次序计算矩阵连乘积需要的数乘次数最少。

说明：计算两个均为 n×n 的矩阵(即 n 阶方阵)相乘还有一种 Strassen 矩阵乘法，利用分治思想将 2 个 n 阶矩阵乘积所需时间从标准算法的 O(n3)改进到 O(nlog7)=O(n2.81)。目前计算两个 n 阶方阵相乘最好的计算时间上界是 O(n2.367)。但无论如何，所需的乘法次数总随两个矩阵的阶而递增。在这道题中只考虑采用标准公式计算两个矩阵的乘积。

参考解答

解这个问题的最容易想到的方法是穷举搜索法。也就是列出所有可能的计算次序，并计算出每一种计算次序相应需要的计算量，然后找出最小者。然而，这样做计算量太大。事实上，对于 n 个矩阵的连乘积，设有 P(n)个不同的计算次序。由于我们可以首先在第 k 个和第 k+1 个矩阵之间将原矩阵序列分为两个矩阵子序列，k=1,2,…,n-1；然后分别对这两个矩阵子序列完全加括号；最后对所得的结果加括号，得到原矩阵序列的一种完全加括号方式。所以关于 P(n)，我们有递推式如下：

$$P(n)=\begin{cases}1 & ,n=1\\ \sum_{k=1}^{n-1}P(k)P(n-k) & ,n\geq 2\end{cases}$$

解此递归方程可得,P(n)实际上是 Catalan 数，即 P(n)=C(n-1)，其中，

$$C(n)=\frac{1}{n+1}\binom{2n}{n}=\Omega\left(\frac{4^n}{n^{3/2}}\right)$$

也就是说，P(n)随着 n 的增长是指数增长的。因此，穷举搜索法不是一个有效算法。

下面我们来考虑用动态规划法解矩阵连乘积的最优计算次序问题。此问题是动态规划的典型应用之一。

1.分析最优解的结构

首先，为方便起见，将矩阵连乘积 AiAi+1…Aj 简记为 Ai…j。我们来看计算 A1…n 的一个最优次序。设这个计算次序在矩阵 Ak 和 Ak+1 之间将矩阵链断开，1<=k<n，则完全加括号方式为((A1…Ak)(Ak+1…An))。照此，我们要先计算 A1…k 和 Ak+1…n，然后，将所得的结果相乘才得到 A1…n。显然其总计算量为计算 A1…k 的计算量加上计算 Ak+1…n 的计算量，再加上 A1…k 与 Ak+1…n 相乘的计算量。

这个问题的一个关键特征是：计算 A1…n 的一个最优次序所包含的计算 A1…k 的次序也是最优的。事实上，若有一个计算 A1…k 的次序需要的计算量更少，则用此次序替换原来计算 A1…k 的次序，得到的计算 A1…n 的次序需要的计算量将比最优次序所需计算量更少,这是一个矛盾。同理可知,计算 A1…n 的一个最优次序所包含的计算矩阵子链 Ak+1…n 的次序也是最优的。根据该问题的指标函数的特征也可以知道该问题满足最优化原理。另外，该问题显然满足无后向性，因为前面的矩阵链的计算方法和后面的矩阵链的计算方法无关。

2.建立递归关系

对于矩阵连乘积的最优计算次序问题，设计算 Ai…j ,1≤i≤j≤n，所需的最少数乘次数为 m[i,j]，原问题的最优值为 m[1,n]。

当 i=j 时，Ai…j=Ai 为单一矩阵，无需计算，因此 m[i,i]=0，i=1,2,…,n ；

当 i<j 时，可利用最优子结构性质来计算 m[i,j]。事实上，若计算 Ai…j 的最优次序在 Ak 和 Ak+1 之间断开，i≤k<j，则：m[i,j]=m[i,k]+m[k+1,j]+pi-1pkpj

由于在计算时我们并不知道断开点 A 的位置，所以 A 还未定。不过 k 的位置只有 j-i 个可能，即 k∈{i,i+1,…,j-1}。因此 k 是这 j-i 个位置中计算量达到最小的那一个位置。从而 m[i,j]可以递归地定义为：

$$m[i,j]=\begin{cases}0 & i=j\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & i<j\end{cases} \qquad (2.1)$$

m[i,j]给出了最优值，即计算 Ai…j 所需的最少数乘次数。同时还确定了计算 Ai…j 的最优次序中的断开位置 k，也就是说，对于这个 k 有 m[i,j] = m[i,k] + m[k+1,j] + pi-1pkpj 。若将对应于 m[i,j]的断开位置 k 记录在 s[i,j]中，则相应的最优解便可递归地构造出来。

3.计算最优值

根据 m[i,j]的递归定义(2.1)，容易写一个递归程序来计算 m[1,n]。稍后我们将看到，简单地递归计算将耗费指数计算时间。然而，我们注意到，在递归计算过程中，不同的子问题个数只有 θ(n2)个。事实上，对于 1≤i≤j≤n 不同的有序对(i,j)对应于不同的子问题。因此，不同子问题的个数最多只有

$$\binom{n}{2}+n=\Theta(n^2)$$

个。由此可见，在递归计算时，许多子问题被重复计算多次。这也是该问题可用动态规划算法求解的又一显著特征。

用动态规划算法解此问题，可依据递归式(2.1)以自底向上的方式进行计算，在计算过程中，保存已解决的子问题答案，每个子问题只计算一次，而在后面需要时只要简单查一下，从而避免大量的重复计算，最终得到多项式时间的算法。下面所给出的计算 m[i,j]动态规划算法中，输入是序列 P={p0,p1,…,pn}，输出除了最优值 m[i,j]外，还有使

m[i,j] = m[i,k] + m[k+1,j] + pi-1pkpj

达到最优的断开位置 k=s[i,j]，1≤i≤j≤n 。

```
Procedure MATRIX_CHAIN_ORDER(p); {计算矩阵链连乘的最优断开位置}
var
i,j,k,q:integer;
begin
  for j:=1 to n do {矩阵链的长度为 n}
  for i:=j downto 1 do
    begin
      if i=j then m[i,j]:=0
            else begin
```

```
                    m[i,j]:=∞;
                    for k:=i to j-1 do
                      begin
                        q:=m[i,k]+m[k+1,j]+p[i-1]*p[k]*p[j];
                        if q<m[i,j] then
                            begin
                              m[i,j]:=q;
                              s[i,j]:=k;
                               {s[i,j]记录计算 A[i..j]的最优断开位置 k}
                            end;
                      end;
                  end;
    end;
end;
```

该算法按照

m[1,1]

m[2,2] m[1,2]

m[3,3] m[2,3] m[1,3]

... ...

m[n,n] m[n-1,n] ... . m[1,n]

的顺序根据公式(2.1)计算 m[i,j]。

该算法的计算时间上界为 O(n3)。算法所占用的空间显然为 O(n2)。由此可见，动态规划算法比穷举搜索法要有效得多。

4.构造最优解

算法 MATRIX_CHAIN_ ORDER 只是计算出了最优值，并未给出最优解。也就是说，通过 MATRIX_CHAIN_ORDER 的计算，我们只知道计算给定的矩阵连乘积所需的最少数乘次数，还不知道具体应按什么次序来做矩阵乘法才能达到数乘次数最少。

然而，MATRIX_CHAIN_ORDER 已记录了构造一个最优解所需要的全部信息。事实上，s[i,j]中的数 k 告诉我们计算矩阵链 Ai…j 的最佳方式应在矩阵 Ak 和 Ak+1 之间断开，即最优的加括号方式应为(A1...k)(Ak+1…n)。因此，从 s[i,j]记录的信息可知计算 A1…n 的最优加括号方式为 (A1…s[1,n])(As[1,n]+1…n)。而计算 A1…s[1,n]的最优加括号方式为(A1…s[1,s[1,n]])(As[1,s[1,n]]+1…s[1,n])。同理可以确定计算 As[1,n]+1…n 的最优加括号方式在 s[s[1,n]+1,n]处断开。…照此递推下去，最终可以确定 As[1,n]+1…n 的最优完全加括号方式，即构造出问题的一个最优解。

下面的算法 MATRIX_CHAIN_MULTIPLY(A,s,i,j)是按 s 指示的加括号方式计算矩阵链 A={A1,A2,…,An}的子链 Ai…j 的连乘积的算法。

```
Procdeure MATRIX_CHAIN_MULTIPLY(A,s,i,j);
begin
  if j>i then
          begin
            X←MATRIX_CHAIN_MULTIPLY(A,s,i,s[i,j]);
            Y←MATRIX_CHAIN_MULTIPLY(A,s,s[i,j]+1,j);
```

```
            return(MATRIX_MULTIPLY(X,Y));{计算并返回矩阵 X*Y 的值}
          end
        else return(Ai);
  end;
```

要计算 A1…n 只要调用 MATRIX_CHAIN_MULTIPLY(A,s,1,n)即可。

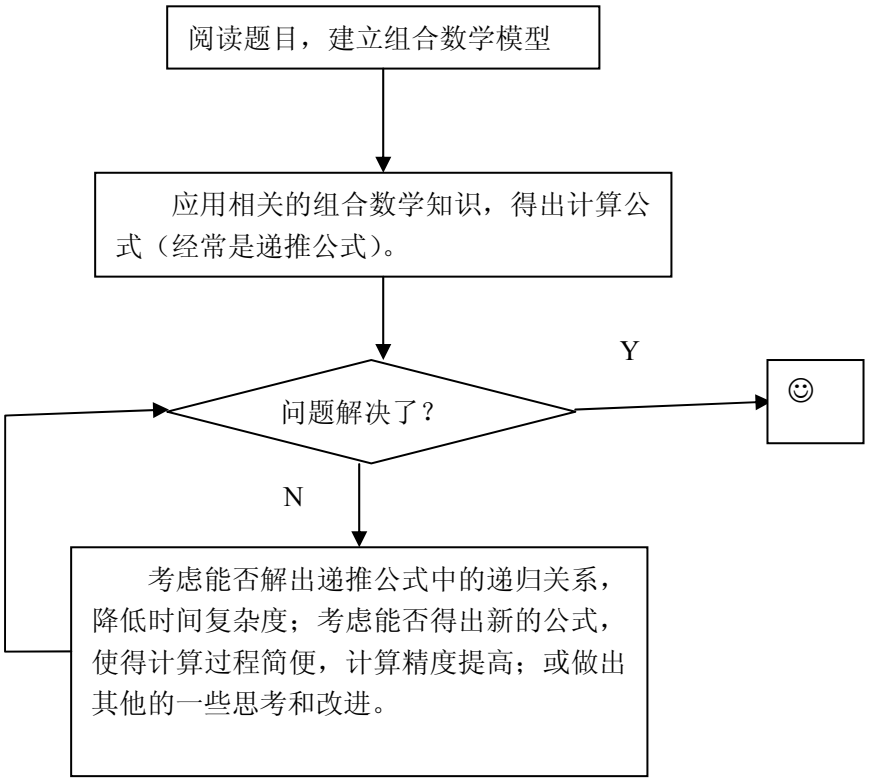
从算法 MATRIX_CHAIN_ ORDER 可以看出，该算法的有效性依赖于问题本身所具有的三个重要性质：最优子结构性质，无后向性和子问题重叠性质。一般说来，问题所具有的这三个重要性质是该问题可用动态规划算法求解的基本要素，这对于我们在设计求解具体问题的算法时，是否选择动态规划算法具有指导意义。

第五章 组合数学简介

一、概述

组合数学是计算机出现以后迅速发展起来的一门数学分支。组合数学在比赛中同样有着很重要的作用。和图论中一样，组合数学的基础知识在大多数的组合数学教科书上都能够找到，笔者在这里介绍一下组合数学在比赛中的一些应用。

这里先介绍一下比赛中解组合数学题目的一般步骤，当然我们在解题的时候应该具体题目具体对待。一般的，解组合数学题目的一般步骤如下：



二、解组合数学题目的一些方法

1. 补集转化思想

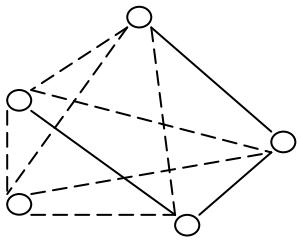
本文通过对一个重要模型——单色三角形问题，探讨了补集转化思想在组合数学问题中的应用，分析了补集转化思想在这个例子中的作用效果和价值。

最后得出结论：补集转化思想应用于组合数学问题中往往有着很好的效果，我们应该注意培养逆向思维，掌握好这种方法。

单色三角形问题

空间里有 n 个点，任意三点不共线。每两点之间都用红色或黑色线段（只有一条，非红即黑!）连接。如果一个三角形的三条边同色，则称这个三角形是单色三角形。对于给定的红色线段的列表，找出单色三角形的个数。例如下图中有 5 个点，10 条边，形成 3 个单色三角形。

输入点数 n、红色边数 m 以及这 m 条红色的边所连接的顶点标号，输出单色三角形个数 R。3<=n<=1000，0<=m<=250000。



（图中红边用虚线表示，黑边用实线表示）

很自然地，我们想到了如下算法：用一个 1000*1000 的数组记录每两个点之间边的颜色，然后枚举所有的三角形（这是通过枚举三个顶点实现的），判断它的三条边是否同色，如果同色则累加到总数 R 中（当然，初始时 R 为 0）。

这个算法怎么样呢？它的时间复杂度已经高达 $O(n^3)$。对于 n 最大达到 1000 的本题来说，实在不能说是个好算法。

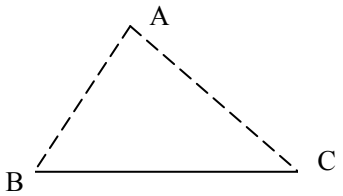
稍稍进一步分析就会发现，本题中单色三角形的个数将是非常多的，所以一切需要枚举出每一个单色三角形的方法都是不可能高效的。

单纯的枚举不可以，那么组合计数是否可行呢？从总体上进行组合计数很难想到，那么我们尝试枚举每一个点，设法找到一个组合公式来计算以这个点为顶点的单色三角形的个数。

这样似乎已经触及到问题的本质了，因为利用组合公式进行计算是非常高效的。但是仔细分析后可以发现，这个组合公式是很难找到的，因为对于枚举确定的点 A，以 A 为一个顶点的单色三角形 ABC 不仅要满足边 AB 和边 AC 同色，而且边 BC 也要和 AB、AC 边同色，于是不可能仅仅通过枚举一个顶点 A 就可以确定单色三角形。

经过上面的分析，我们得出枚举＋组合计数有可能是正确的解法，但是在组合公式的构造上我们遇到了障碍。这个障碍的本质是：

从一个顶点 A 出发的两条同色的边 AB、AC 并不能确定一个单色三角形 ABC，因为 BC 边有可能不同色。

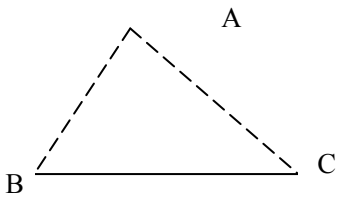


也就是说，我们无法在从同一个顶点出发的某两条边与所有的单色三角形之间建立一种确定的对应关系。

让我们换一个角度，从反面来看问题：因为每两点都有边连接，所以每三个点都可以组成一个三角形（单色或非单色的），那么所有的三角形数 S=C(n,3)=n*(n-1)*(n-2)/6。又因为单色三角形数 R 加上非单色三角形数 T 就等于 S，所以如果我们可以求出 T，那么显然，R=S－T。于是原问题就等价于：怎样高效地求出 T。

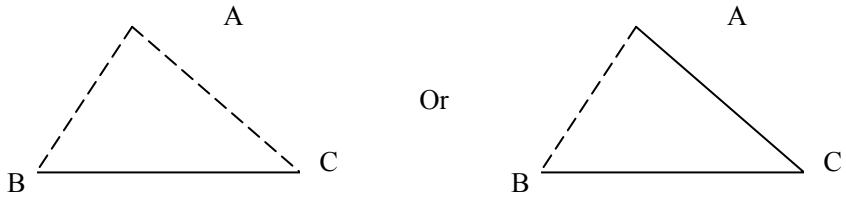
纯枚举的算法想都不用想就被排除，那么在上面分析中夭折的枚举＋组合计数的算法又怎样呢？这个算法原先的障碍是无法在“某两条边”与“单色三角形”之间建立确定的对应关系。那么有公共顶点的某两条边与非单色三角形之间是否有着确定的关系呢？对了！这种关系是明显的：

非单色三角形的三条边，共有红黑两种颜色，也就是说，只能是两条边同色，另一条边异色。假设同色的两条边顶点为 A，另外两个顶点为 B 和 C，则从 B 点一定引出两条不同色的边 BA 和 BC，同样，从 C 点引出两条不同色的边 CA 和 CB。



这样，一个非单色三角形对应着两对“有公共顶点的异色边”。

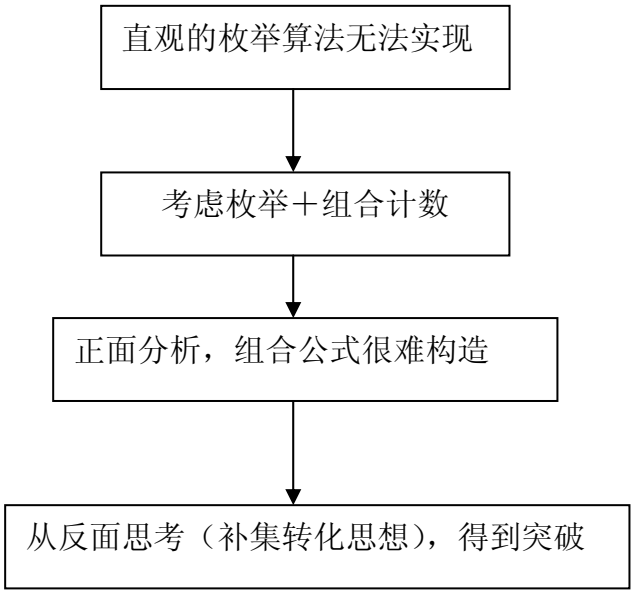
另一方面，如果从一个顶点 B 引出两条异色的边 BA、BC，则无论 AC 边是何种颜色，三角形 ABC 都只能是一个非单色三角形。也就是说，一对“有公共顶点的异色边”对应着一个非单色三角形。



很明显，我们要求的非单色三角形数 T 就等于所有“有公共顶点的异色边”的总对数 Q 的一半。而这个总对数是很好求的：每个顶点有 n-1 条边，根据输入的信息可以知道每个顶点 i 的红边数 E[i]，那么其黑边数就是 n-1-E[i]。枚举顶点 A，则根据乘法原理，以 A 为公共顶点的异色边的对数就是 E[i]*(n-1-E[i])。所以 $Q=\sum_{i=1}^{n}E[i]*(n-1-E[i])$ 。

求出 Q 之后，答案 R=S－T=n*(n-1)*(n-2)/6-Q/2。这个算法的时间复杂度仅为 O(m+n)，空间复杂度是 O(n)，非常优秀。

在这个例子中，我们经历了如下的思维过程：



补集转化思想在其中起着至关重要的作用：通过补集转化，我们才能够在原来无法联系起来的“边”和“三角形”之间建立起确定的关系，并以此构造出组合计数的公式。这样，就由单纯的枚举算法改为了枚举+组合计数的算法，大大降低了时间和空间复杂度。在这里，补集转化思想的作用体现在为找到一个本质上不同的算法创造了条件。

2. 组合数学中的递推关系

建立递推关系的关键在于寻找第 n 项与前面几项的关系式，以及初始项的值。它不是一种抽象的概念，是需要针对某一具体题目或一类题目而言的。在下文中，笔者将介绍几种典型的递推关系。

Fibonacci 数列

在所有的递推关系中，Fibonacci 数列应该是最为大家所熟悉的。Fibonacci 数列的代表问题是由意大利著名数学家 Fibonacci 提出的“兔子繁殖问题”(又称“Fibonacci 问题”)。

问题的提出：有雌雄一对兔子，假定过两个月便可繁殖雌雄各一的一对小兔子。问过 n 个月后共有多少对兔子？

解：设满 x 个月共有兔子 F_x 对，其中当月新生的兔子数目为 N_x 对。第 x-1 个月留下的兔子数目设为 O_x 对。则：

$F_x=N_x+O_x$

而 $O_x=F_{x-1}$，

$N_x=O_{x-1}=F_{x-2}$ (即第 x-2 个月的所有兔子到第 x 个月都成熟了)

∴ $F_x=F_{x-1}+F_{x-2}$ 边界条件： $F_0=0$，$F_1=1$

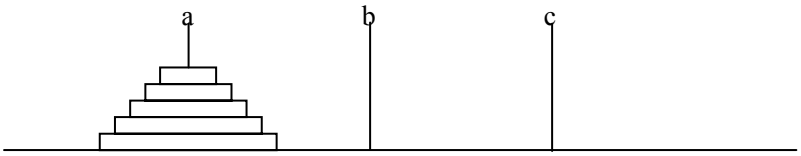
由上面的递推关系可依次得到

$F_2=F_1+F_0=1$，$F_3=F_2+F_1=2$，$F_4=F_3+F_2=3$，$F_5=F_4+F_3=5$，……。

Fabonacci 数列常出现在比较简单的组合计数问题中，例如 2*n 的“骨牌覆盖”的递推关系就是 Fabonacci 关系。

Hanoi 塔问题

问题的提出：Hanoi 塔由 n 个大小不同的圆盘和三根木柱 a,b,c 组成。开始时，这 n 个



圆盘由大到小依次套在 a 柱上，如图 1 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；

(2)圆盘只能在三个柱上存放；

(3)在移动过程中，不允许大盘压小盘。

问将这 n 个盘子从 a 柱移动到 c 柱上，总计需要移动多少个盘次？

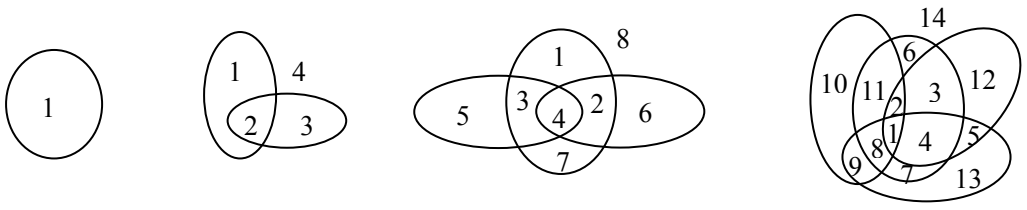
解：设 h_n 为 n 个盘子从 a 柱移到 c 柱所需移动的盘次。显然，当 n=1 时，只需把 a 柱上的盘子直接移动到 c 柱就可以了，故 $h_1=1$。当 n=2 时，先将 a 柱上面的小盘子移动到 b 柱上去；然后将大盘子从 a 柱移到 c 柱；最后，将 b 柱上的小盘子移到 c 柱上，共记 3 个盘次，故 $h_2=3$。以此类推，当 a 柱上有 $n(n\geq 2)$个盘子时，总是先借助 c 柱把上面的 n-1 个盘子移动到 b 柱上，然后把 a 柱最下面的盘子移动到 c 柱上；再借助 a 柱把 b 柱上的 n-1 个盘子移动到 c 柱上；总共移动 $h_{n-1}+1+h_{n-1}$ 个盘次。

$\therefore h_n=2h_{n-1}+1$　　边界条件：$h_{n-1}=1$

平面分割问题

问题的提出：设有 n 条封闭曲线画在平面上，而任何两条封闭曲线恰好相交于两点，且任何三条封闭曲线不相交于同一点，问这些封闭曲线把平面分割成的区域个数。

解：设 a_n 为 n 条封闭曲线把平面分割成的区域个数。 由上图可以看出：$a_2-a_1=2$；$a_3-a_2=4$；



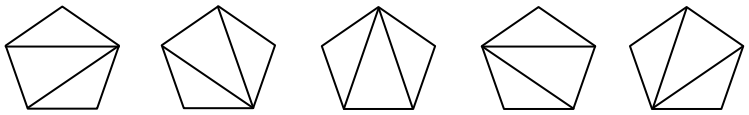
n=1　　n=2　　n=3　　n=4

$a_4-a_3=6$。从这些式子中可以看出 $a_n-a_{n-1}=2(n-1)$。当然，上面的式子只是我们通过观察 4 幅图后得出的结论，它的正确性尚不能保证。下面不妨让我们来试着证明一下。当平面上已有 n-1 条曲线将平面分割成 a_{n-1} 个区域后，第 n-1 条曲线每与曲线相交一次，就会增加一个区域，因为平面上已有了 n-1 条封闭曲线，且第 n 条曲线与已有的每一条闭曲线恰好相交于两点，且不会与任两条曲线交于同一点，故平面上一共增加 2(n-1)个区域，加上已有的 a_{n-1} 个区域，一共有 $a_{n-1}+2(n-1)$个区域。所以本题的递推关系是 $a_n=a_{n-1}+2(n-1)$，边界条件是 $a_1=1$。

Catalan 数

Catalan 数首先是由 Euler 在精确计算对凸 n 边形的不同的对角三角形剖分的个数问题时得到的，它经常出现在组合计数问题中。

问题的提出：在一个凸 n 边形中，通过不相交于 n 边形内部的对角线，把 n 边形拆分成若干三角形，不同的拆分数目用 h_n 表之，h_n 即为 Catalan 数。例如五边形有如下五种拆分方案，故 $h_5=5$。求对于一个任意的凸 n 边形相应的 h_n。



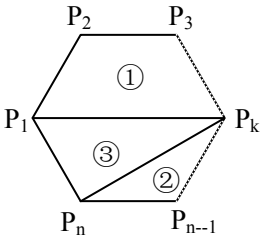


图 4

解：设 C_n 表示凸 n 边形的拆分方案总数。由题目中的要求可知一个凸 n 边形的任意一条边都必然是一个三角形的一条边，边 P_1 P_n 也不例外，再根据“不在同一直线上的三点可以确定一个三角形”，只要在 P_2，P_3，……，P_{n-1} 点中找一个点 $P_k(1<k<n)$，与 P_1、P_n 共同构成一个三角形的三个顶点，就将 n 边形分成了三个不相交的部分(如图 3 所示)，我们分别称之为区域①、区域②、区域③，其中区域③必定是一个三角形，区域①是一个凸 k 边形，区域②是一个凸 n-k+1 边形，区域①的拆分方案总数是 C_k，区域②的拆分方案数为 C_{n-k+1}，故包含$\triangle P_1P_kP_n$的 n 边形的拆分方案数为 C_kC_{n-k+1} 种，而 P_k 可以是 P_2，P_3，……，P_{n-1} 种任一点，根据加法原理，凸 n 边形的三角拆分方案总数为 $\sum_{i=2}^{n-1} C_i C_{n-i+1}$，同时考虑到计算的方便，约定边界条件 $C_2=1$。

Catalan 数是比较复杂的递推关系，也是一个很重要的递推关系模型，尤其在比赛中的应用很大。

当然上面的递推关系都可以解出，解出以后一般可以使计算的时间复杂度下降，当然要注意解出递推关系后，有可能出现无理数等，在计算机上计算时，可能会出现精度不够而造成误差，这在解题时要细细衡量，在效率和精度之间仔细权衡。

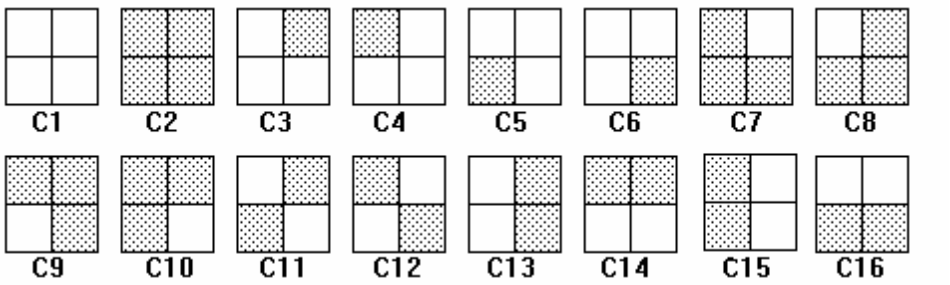
三、Pólya 原理及其应用

Pólya 原理是组合数学中，用来计算全部互异的组合状态的个数的一个十分高效、简便的工具。下面，笔者就向大家介绍一下什么是 Pólya 原理以及它的应用。请先看下面这道例题：

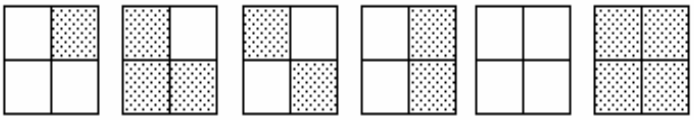
对 2*2 的方阵用黑白两种颜色涂色，问能得到多少种不同的图像？经过旋转使之吻合的两种方案，算是同一种方案。

由于该问题规模很小，我们可以先把所有的涂色方案列举出来。

一个 2*2 的方阵的旋转方法一共有 4 种：旋转 0 度、旋转 90 度、旋转 180 度和旋转



270 度。(注：本文中默认旋转即为顺时针旋转) 我们经过尝试，发现其中互异的一共只有 6 种：C3、C4、C5、C6 是可以通过旋转相互变化而得，算作同一种；C7、C8、C9、C10 是同一种；C11、C12 是同一种；C13、C14、C15、C16 也是同一种； C1 和 C2 是各自独立的两种。于是，我们得到了下列 6 种不同的方案。



但是，一旦这个问题由 2*2 的方阵变成 20*20 甚至 200*200 的方阵，我们就不能再一一枚举了，利用 Pólya 原理成了一个很好的解题方法。在接触 Pólya 原理之前，首先简单介绍 Pólya 原理中要用到的一些概念。

群：给定一个集合 $G=\{a,b,c,\ldots\}$和集合 G 上的二元运算，并满足:

(a) 封闭性：$\forall a,b\in G, \exists c\in G, a*b=c$。

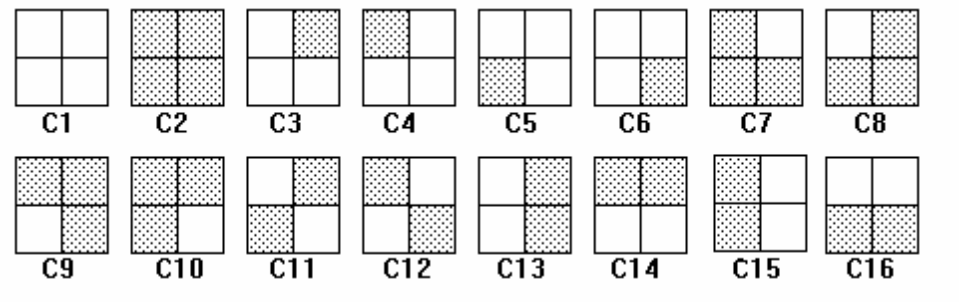
(b) 结合律：$\forall a,b,c\in G, (a*b)*c=a*(b*c)$。

(c) 单位元：$\exists e\in G, \forall a\in G, a*e=e*a=a$。

(d) 逆元：$\forall a\in G, \exists b\in G, a*b=b*a=e$，记 $b=a^{-1}$。

则称集合 G 在运算 $*$ 之下是一个群，简称 G 是群。一般 $a*b$ 简写为 ab。

置换：n 个元素 1,2,…,n 之间的一个置换 $\begin{pmatrix} 1 & 2 & \cdots & n \\ a_1 & a_2 & \cdots & a_n \end{pmatrix}$ 表示 1 被 1 到 n 中的某个数 a_1 取代，2 被 1 到 n 中的某个数 a_2 取代，直到 n 被 1 到 n 中的某个数 a_n 取代，且 $a_1,a_2,\ldots,a_n$ 互不相同。本例中有 4 个置换：



转 0°　$a1=\begin{pmatrix}1&2&3&4&5&6&7&8&9&10&11&12&13&14&15&16\\1&2&3&4&5&6&7&8&9&10&11&12&13&14&15&16\end{pmatrix}$

转 90°　$a2=\begin{pmatrix}1&2&3&4&5&6&7&8&9&10&11&12&13&14&15&16\\1&2&6&3&4&5&10&7&8&9&12&11&16&13&14&15\end{pmatrix}$

转 180°　$a3=\begin{pmatrix}1&2&3&4&5&6&7&8&9&10&11&12&13&14&15&16\\1&2&5&6&3&4&9&10&7&8&11&12&15&16&13&14\end{pmatrix}$

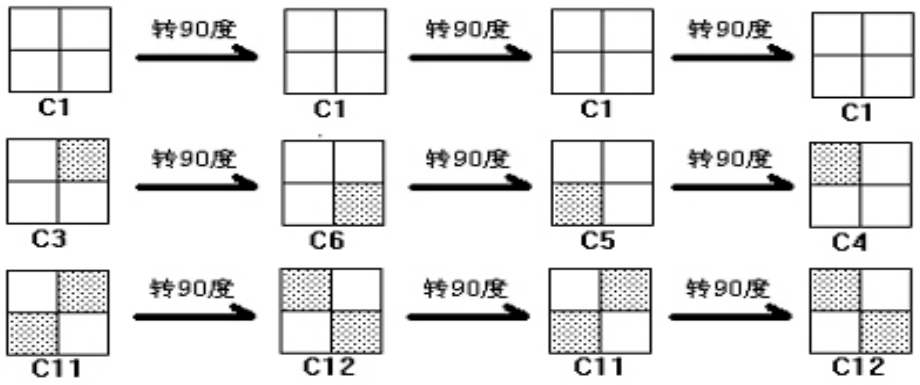
转 270°　$a4=\begin{pmatrix}1&2&3&4&5&6&7&8&9&10&11&12&13&14&15&16\\1&2&4&5&6&3&8&9&10&7&12&11&14&15&16&13\end{pmatrix}$

置换群：置换群的元素是置换，运算是置换的连接。例如：

$$\begin{pmatrix}1&2&3&4\\3&1&2&4\end{pmatrix}\begin{pmatrix}1&2&3&4\\4&3&2&1\end{pmatrix}=\begin{pmatrix}1&2&3&4\\3&1&2&4\end{pmatrix}\begin{pmatrix}3&1&2&4\\2&4&3&1\end{pmatrix}=\begin{pmatrix}1&2&3&4\\2&4&3&1\end{pmatrix}$$

可以验证置换群满足群的四个条件。

本题中置换群 G={转 0°、转 90°、转 180°、转 270°}



我们再看一个公式：$|E_k| \cdot |Z_k| = |G|$　　k=1…n

该公式的一个很重要的研究对象是群的元素个数，有很大的用处。

Z_k (K 不动置换类)：设 G 是 1…n 的置换群。若 K 是 1…n 中某个元素，G 中使 K 保持不变的置换的全体，记以 Z_k，叫做 G 中使 K 保持不动的置换类，简称 K 不动置换类。

如本例中：G 是涂色方案 1~16 的置换群。对于方案 1，四个置换都使方案 1 保持不变，所

以 $Z_1=\{a_1, a_2, a_3, a_4\}$；对于方案 3，只有置换 a_1 使其不变，所以 $Z_3=\{a_1\}$；对于方案 11，置换 a_1 和 a_3 使方案其保持不变，所以 $Z_{11}=\{a_1, a_3\}$。

E_k(等价类)：设 G 是 1…n 的置换群。若 K 是 1…n 中某个元素，K 在 G 作用下的轨迹，记作 E_k。即 K 在 G 的作用下所能变化成的所有元素的集合。

如本例中：方案 1 在四个置换作用下都是方案 1，所以 $E_1=\{1\}$；方案 3，在 a_1 下是 3，在 a_2 下变成 6，在 a_3 下变成 5，在 a_4 下变成 4，所以 $E_3=\{3,4,5,6\}$；方案 11，在 a_1、a_3 下是 11，在 a_2、a_4 下变成 12，所以 $E_{11}=\{11,12\}$。

本例中的数据，也完全符合这个定理。*如本例中：*

$|E_1| \cdot |Z_1| = 1\times4 = 4 = |G|$

$|E_3| \cdot |Z_3| = 4\times1 = 4 = |G|$

$|E_{11}| \cdot |Z_{11}| = 2\times2 = 4 = |G|$

限于篇幅，这里就不对这个定理进行证明。接着就来研究每个元素在各个置换下不变的次数的总和。见下表：

	1	2	……	16	$D(a_i)$
a1	$S_{1,1}$	$S_{1,2}$	……	$S_{1,16}$	$D(a_1)$
a2	$S_{2,1}$	$S_{2,2}$	……	$S_{2,16}$	$D(a_2)$
a3	$S_{3,1}$	$S_{3,2}$	……	$S_{3,16}$	$D(a_3)$
a4	$S_{4,1}$	$S_{4,2}$	……	$S_{4,16}$	$D(a_4)$
$\lvert Z_j \rvert$	$\lvert Z_1 \rvert$	$\lvert Z_2 \rvert$	……	$\lvert Z_{16} \rvert$	$\sum_{j=1}^{16} \lvert Z_j \rvert = \sum_{j=1}^{4} D(a_j)$

其中

$D(a_j)$ 表示在置换 a_j 下不变的元素的个数

$$S_{ij} = \begin{cases} 0 & \text{当} a_i \notin Z_j, \text{即} j \text{在} a_i \text{的变化下变动了} \\ 1 & \text{当} a_i \in Z_j, \text{即} j \text{在} a_i \text{的变化下没有变} \end{cases}$$

如本题中：涂色方案 1 在 a_1 下没变动，$S_{1,1}=1$；方案 3 在 a_3 变动了， $S_{3,3}=0$；在置换 a_1 的变化下 16 种方案都没变动，$D(a_1)=16$；在置换 a_2 下只有 1、2 这两种方案没变动，$D(a_2)=2$。一般情况下，我们也可以得出这样的结论：

$$\sum_{j=1}^{n} |Z_j| = \sum_{i=1}^{s} D(a_i)$$

我们对左式进行研究。

不妨设 $N=\{1, \cdots\cdots, n\}$中共有 L 个等价类，$N=E_1+E_2+\cdots\cdots+E_L$，则当 j 和 k 属于同一等价类时，有 $|Z_j| = |Z_k|$。所以

$$\sum_{k=1}^{n}|Z_k|=\sum_{i=1}^{L}\sum_{k\in E_i}|Z_k|=\sum_{i=1}^{L}|E_i|\cdot|Z_i|=L\cdot|G|$$

这里的 L 就是我们要求的互异的组合状态的个数。于是我们得出：

利用这个式子我们可以得到本题的解 L=(16+2+4+2)/4=6 与前面枚举得到的结果相吻合。这

$$L=\frac{1}{|G|}\sum_{k=1}^{n}|Z_k|=\frac{1}{|G|}\sum_{j=1}^{s}D(a_j)$$

个式子叫做 Burnside 引理。

但是，我们发现要计算 $D(a_j)$的值不是很容易，如果采用搜索的方法，总的时间规模为 O(n×s×p)。(n 表示元素个数，s 表示置换个数，p 表示格子数，这里 n 的规模是很大的) 下一步就是要找到一种简便的 $D(a_j)$的计算方法。先介绍一个循环的概念：

循环：记

$$(a_1a_2\cdots a_n)=\begin{pmatrix}a_1 & a_2 & \cdots & a_{n-1} & a_n\\ a_2 & a_3 & \cdots & a_n & a_1\end{pmatrix}$$

称为 n 阶循环。每个置换都可以写若干互不相交的循环的乘积，两个循环$(a_1a_2\ldots a_n)$和$(b_1b_2\ldots b_n)$互不相交是指 $a_i\neq b_j$, $i,j=1,2,\ldots,n$。例如：

$$\begin{pmatrix}1 & 2 & 3 & 4 & 5\\ 3 & 5 & 1 & 4 & 2\end{pmatrix}=(13)(25)(4)$$

这样的表示是唯一的。置换的循环节数是上述表示中循环的个数。例如(13)(25)(4)的循环节数为 3。

有了这些基础，就可以做进一步的研究，我们换一个角度来考虑这个问题。我们给 2*2 方阵的每个方块标号，如下图：

2	1
3	4

构造置换群 G'={g_1,g_2,g_3,g_4}, | G' | =4，令 g_i 的循环节数为 $c(g_i)$ (i=1,2,3,4)

在 G'的作用下，其中

g_1 表示转 0°， 即 g_1=(1)(2)(3)(4) $c(g_1)$=4

g_2 表示转 90°， 即 g_2=(4 3 2 1) $c(g_2)$=1

g_3 表示转 180°， 即 g_3=(1 3)(2 4) $c(g_3)$=2

g_4 表示转 270°， 即 g_4=(1 2 3 4) $c(g_4)$=1

我们可以发现，g_i 的同一个循环节中的对象涂以相同的颜色所得的图像数 $m^{c(g_i)}$ 正好对应 G 中置换 a_i 作用下不变的图象数，即

$2^{c(g_1)}=2^4=16=D(a_1)$ $2^{c(g_2)}=2^1=2=D(a_2)$

$2^{c(g_3)}=2^2=4=D(a_3)$ $2^{c(g_4)}=2^1=2=D(a_4)$

由此我们得出一个结论：

设 G 是 p 个对象的一个置换群，用 m 种颜色涂染 p 个对象，则不同染色方案为：

其中 G={g_1 ,…g_s}　c(g_i)为置换 g_i 的循环节数(i=1…s)

这就是所谓的 Pólya 定理。我们发现利用 Pólya 定理的时间复杂度为 O(s×p) (这里 s 表

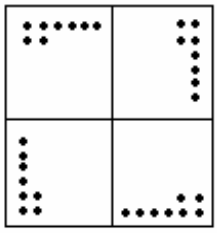
$$L = \frac{1}{|G|}(m^{c(g_1)} + m^{c(g_2)} + \cdots + m^{c(g_s)})$$

示置换个数，p 表示格子数)，与前面得到的 Burnside 引理相比之下，又有了很大的改进，其优越性就十分明显了。Pólya 定理充分挖掘了研究对象的内在联系，总结了规律，省去了许多不必要的盲目搜索，把解决这类问题的时间规模降到了一个非常低的水平。

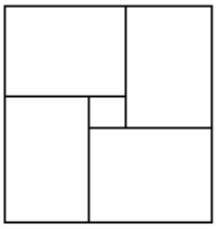
现在我们把问题改为：n×n 的方阵，每个小格可涂 m 种颜色，求在旋转操作下本质不同的解的总数。

先看一个很容易想到的搜索的方法。(见附录)

这样搜索的效率是极低的，它还有很大的改进的余地。前面，我们采用的方法是先搜后判，这样的盲目性极高。我们需要边搜边判，避免过多的不必要的枚举，我们更希望把判断条件完全融入到搜索的边界中去，消灭无效的枚举。这个美好的愿望是可以实现的。



n为偶数时



n为奇数时

我们可以在方阵中分出互不重叠的长为[(n+1)/2]，宽为[n/2]的四个矩阵。当 n 为偶数时，恰好分完；当 n 为奇数时，剩下中心的一个格子，它在所有的旋转下都不动，所以它涂任何颜色都对其它格子没有影响。令 m 种颜色为 0~m-1，我们把矩阵中的每格的颜色所代表的数字顺次(左上角从左到右，从上到下；右上角从上到下，从右到左；……)排成 m 进制数，然后就可以表示为一个十进制数，其取值范围为 0~$m^{[n^2/4]}$-1 。(因为 [n/2]*[(n+1)/2]=[n^2/4]) 这样，我们就把一个方阵简化为 4 个整数。我们只要找到每一个等价类中左上角的数最大的那个方案(如果左上角相同，就顺时针方向顺次比较) 这样，在枚举的时候其它三个数一定不大于左上角的数，效率应该是最高的。

进一步考虑，当左上角数为 i 时,(0≤i≤R-1) 令 R=$m^{[n^2/4]}$

可分为下列的 4 类：

← 其它三个整数均小于 i，共 i^3 个。

↑ 右上角为 i，其它两个整数均小于 i，共 i^2 个。

→ 右上角、右下角为 i，左下角不大于 i，共 i+1 个。

↓ 右下角为 i，其它两个整数均小于 i，且右上角的数不小于左下角的，共 i(i+1)/2 个。

因此，

$$L = \sum_{i=0}^{R-1}(i^3 + i^2 + i + 1 + \frac{1}{2}i(i+1)) = \sum_{i=0}^{R-1}(i^3 + \frac{3}{2}i^2 + \frac{3}{2}i + 1)$$

$$= \sum_{i=1}^{R}((i-1)^3 + \frac{3}{2}(i-1)^2 + \frac{3}{2}(i-1) + 1) = \sum_{i=1}^{R}(i^3 - \frac{3}{2}i^2 + \frac{3}{2}i)$$

$$= \frac{1}{4}R^2(R+1)^2 - \frac{3}{2}\times\frac{1}{6}R(R+1)(2R+1) + \frac{3}{2}\times\frac{1}{2}R(R+1)$$

$$= \frac{1}{4}(R^4 + R^2 + 2R)$$

当 n 为奇数时，还要乘一个 m。

由此我们就巧妙地得到了一个公式。但是，我们应该看到要想到这个公式需要付出不少的时间。另一方面，这种方法只能对这道题有用而不能广泛地应用于一类试题，具有很大的不定性因素。因此，如果能掌握一种适用面广的原理，就会对解这一类题有很大的帮助。

下面我们就采用 Pólya 定理。我们可以分三步来解决这个问题。

1. 确定置换群

在这里很明显只有 4 个置换：转 0°、转 90°、转 180°、转 270°。所以，置换群 G={转 0°、转 90°、转 180°、转 270°}。

2. 计算循环节个数

首先，给每个格子顺次编号（1~n^2），再开一个二维数组记录置换后的状态。最后通过搜索计算每个置换下的循环节个数，效率为一次方级。

3. 代入公式

即利用 Pólya 定理得到最后结果。

如果大家再仔细地考虑一下，就会发现这个题解还可以继续优化。对 n 分情况讨论：

$$L=\frac{1}{|G|}(m^{c(g_1)}+m^{c(g_2)}+\cdots+m^{c(g_s)})$$

← n 为偶数：在转 0°时，循环节为 n^2 个，转 180°时，循环节为 $n^2/2$ 个，转 90°和转 270°时，循环节为 $n^2/4$ 个。

↑ n 为偶数：在转 0°时，循环节为 n^2 个，转 180°时，循环节为$(n^2+1)/2$ 个，转 90°和转 270°时，循环节为$(n^2+3)/4$ 个。

把这些综合一下就得到：在转 0°时，循环节为 n^2 个，转 180°时，循环节为$[(n^2+1)/2]$ 个，转 90°和转 270°时，循环节为$[(n^2+3)/4]$个。(其中，方括号表示取整)于是就得到：

$$L=\frac{1}{4}(m^{n^2}+m^{[\frac{n^2+3}{4}]}+m^{[\frac{n^2+1}{2}]}+m^{[\frac{n^2+3}{4}]})$$

这和前面得到的结果完全吻合。

经过上述一番分析，使得一道看似很棘手的问题得以巧妙的解决，剩下的只要做一点高精度计算即可。

第六章 专题解析

一、模　拟

模拟类题目，顾名思义，是通过对一些实物（譬如各种游戏，电梯，加密过程等等）的模拟来求解的题目，是一类与实际结合较为紧密的题目，所以在历年竞赛中都会有此类题目。 而对此类题目往往没有一个严格的定义，它所涉及的算法很广，因此它和其他类别的题目如密码类，搜索类等都有交集，之所以把它单独作为一类是为了给大家提供一种解题的思考角度。

模拟类题目都会有一个共同特点：在题目中已给你一个完整的实物模型譬如游戏规则，电梯的运行规则，加密的过程描述等，你要做的只是如何将它抽象为一个数据模型并将其实现。

此类题目难易差距很大，容易的一般只要考虑其数据结构的选取，而算法只需按照步骤逐步模拟。而难的就会涉及到多种算法并由于模拟过程的繁杂而导致对边界情况和环节间的相互联系和影响考虑不周，这也是此类题目的一大难点。

还有一类题目虽然也可以用模拟的方法求解，但由于其所要求的只是结果而模拟过程一般都极其繁杂，所以我们要尽量用数学的方法甚至公式将其简化求解，在这里就不再详细讨论。

1. 模拟游戏类

下面的题目是一道典型的游戏模拟类题目，牌类是这类题目常涉及到的一个游戏类型，这在往年的竞赛题目中都有体现。在解题的过程中需要注意的是具体的游戏规则及对各种边界情况的考虑。

A simple solitaire card game called 10-20-30 uses a standard deck of 52 playing cards in which suit is irrelevant. The value of a face card (king, queen, jack) is 10. The value of an ace is one. The value of each of the other cards is the face value of the card (2, 3, 4, etc.). Cards are dealt from the top of the deck. You begin by dealing out seven cards, left to right forming seven piles. After playing a card on the rightmost pile, the next pile upon which you play a card is the leftmost pile.

For each card placed on a pile, check that pile to see if one of the following three card combinations totals 10, 20, or 30.

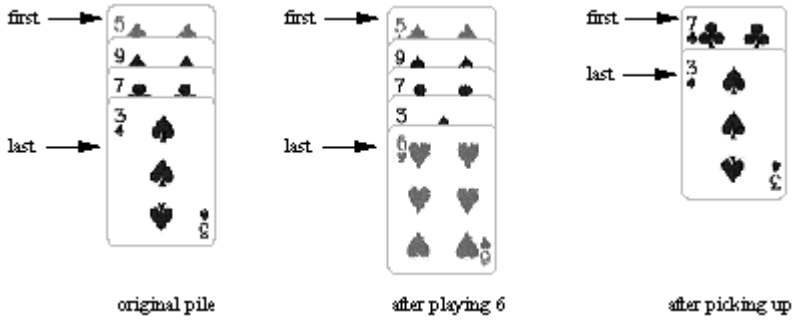
the first two and last one,

the first one and the last two, or

the last three cards.

If so, pick up the three cards and place them on the bottom of the deck. For this problem, always check the pile in the order just described. Collect the cards in the order they appear on the pile and put them at the bottom of the deck. Picking up three cards may expose three more cards that can be picked up. If so, pick them up. Continue until no more sets of three can be picked up from the pile.

For example, suppose a pile contains 5 9 7 3 where the 5 is at the first card of the pile, and then a 6 is played. The first two cards plus the last card (5 + 9 + 6) sum to 20. The new contents of the pile after picking up those three cards becomes 7 3. Also, the bottommost card in the deck is now the 6, the card above it is the 9, and the one above the 9 is the 5.



If a queen were played instead of the six, 5 + 9 + 10 = 24, and 5 + 3 + 10 = 18, but 7 + 3 + 10 = 20,

so the last three cards would be picked up， leaving the pile as 5 9.

If a pile contains only three cards when the three sum to 10, 20, or 30, then the pile "disappears" when the cards are picked up. That is, subsequent play skips over the position that the now-empty pile occupied. You win if all the piles disappear. You lose if you are unable to deal a card. It is also possible to have a draw if neither of the previous two conditions ever occurs.

Write a program that will play games of 10-20-30 given initial card decks as input.

Input

Each input set consists of a sequence of 52 integers separated by spaces and/or ends of line. The integers represent card values of the initial deck for that game. The first integer is the top card of the deck. Input is terminated by a single zero (0) following the last deck.

Output

For each input set, print whether the result of the game is a win, loss, or a draw, and print the number of times a card is dealt before the game results can be determined. (A draw occurs as soon as the state of the game is repeated.) Use the format shown in the "Output for the Sample Input" section.

Sample Input

2 6 5 10 10 4 10 10 10 4 5 10 4 5 10 9 7 6 1 7 6 9 5 3 10 10 4 10 9 2 1

10 1 10 10 10 3 10 9 8 10 8 7 1 2 8 6 7 3 3 8 2

4 3 2 10 8 10 6 8 9 5 8 10 5 3 5 4 6 9 9 1 7 6 3 5 10 10 8 10 9 10 10 7

2 6 10 10 4 10 1 3 10 1 1 10 2 2 10 4 10 7 7 10

10 5 4 3 5 7 10 8 2 3 9 10 8 4 5 1 7 6 7 2 6 9 10 2 3 10 3 4 4 9 10 1 1

10 5 10 10 1 8 10 7 8 10 6 10 10 10 9 6 2 10 10

0

Output for the Sample Input

Win : 66

Loss: 82

Draw: 73

题目分析及算法：

此题目要求判断游戏结果，可以通过模拟游戏的过程来求解。

数据结构的选用：

由于每张牌的面值在 1~10 之间，因此可用一个整型描述一张牌。可以选用 8 个（7 组牌和 1 牌堆）可以删除，添加的整型数组，每个数组的大小是 0~52，但这样较浪费空间。注意到 8 个数组的总大小始终为 52，所以我们可以用一个可删除，添加的整型数组来统一存放这 7 组牌和 1 个牌堆，选用 stl 库中的 vector,在此 vector 中设置三个标记指针——的 deck:牌堆首指针，first:当前牌组首指针，last:当前牌组尾指针；和一个整型数组 tlt[7]来记录每组牌的个数。为方便 first,last 的取值，我们用 7 个整数‘0’来分隔牌组和牌组。

Vector:

数据结构的基本操作：发牌和收牌

我们不妨将每组都已发好一张牌作为初始状态，每发一张牌就从 deck 处取出一张牌，将其插入 last 位置，然后 deck 后移一个位置，本组牌数目加 1（tlt[i]++）。

发完牌后检查是否有可以收的牌，若三张牌的总和为 10，20，30，则将其抽出，顺序移至牌堆底部即数组底部，然后 last,deck 前移三位，tlt[i]-3.此收牌过程重复进行，直到当前牌组抽不出符合条件的牌为止。

判断游戏结局：

游戏结局只有三个：

胜局：当所有牌组消失，即 tlt[7]均为 0。

败局：当牌堆为空，即 deck=59 时。

平局：无论怎样继续，胜败局都不发生，即各牌面状态循环出现。为此，我们记录每个出现的牌面状态，若有两个状态完全相同则为平局。但保留状态多占用内存也多，我们不妨保留近 50 个存入 state[50]中，每放入一个前面的状态前移，第一个将移出，然后再与前 49 个状态比较，若有相同的则判为平局。

如果当前状态非前三种，则说明游戏还可继续，继续发牌。我们用 tile 来记录当前牌组序号，其初始值为 0，每发完一张牌 tile+1,当 tile 为 6 时，发完牌后 tile 为 0，此时 last 重新赋值为第一组牌的尾指针。

注：first,last 的取值问题：

last 始终指向分隔标志‘0’。

初始时每组都有一张牌，tile=0,first=0,last=1；每发一张牌，last++；当发下一组牌时，tile+1,last+tlt[tile]+1,first 可由 last 得到即 first=last-tlt[tile];特殊的，当 tile=0 时，last=tlt[tile]。

参考代码：略

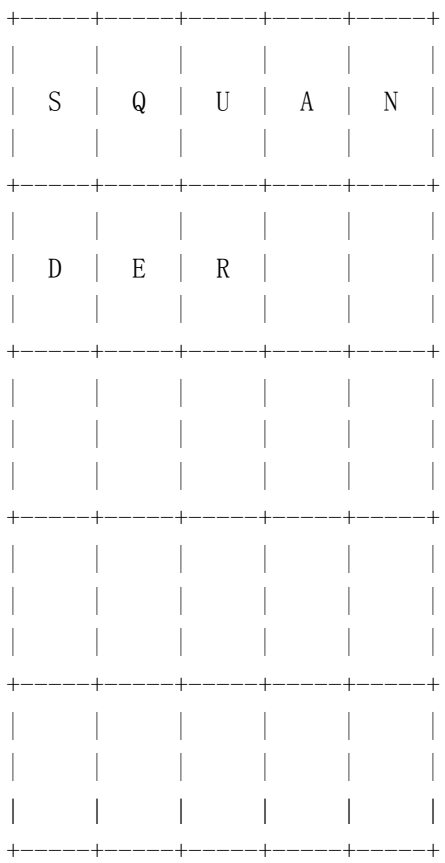
2. 模拟编码类

下面的这道题就是一道典型非游戏类模拟题，可用模拟编码的过程进行求解。给出的参考解是本人早期做的，基本没有用 stl 库。

In Dorothy Sayers' novel "Have His Carcass", Lord Peter Wimsey describes a cryptography technique that is simple for encoding and decoding, yet relatively hard to crack. Your job is to implement this technique.

Here's how it works, in Sayers' (edited) words:

You choose a key-word of six letters or more, none of which recurs. Such as, for example, SQUANDER. Then you make a diagram of five squares each way and write the key-word in the squares like this:



Then you fill up the remaining spaces with the rest of the alphabet in order, leaving out the ones you've already got.

You can't put twenty-six letters in twenty-five spaces, so you pretend you're an ancient Roman or a medieval monk and treat I and J as one letter. So you get this:

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

Now let's take a message -- What shall we say? 'All is known, fly at once' -- that classic hardy perennial. We write it down all of a piece and break it into groups of two letters, reading from left to right. It won't do to have two of the same letters coming together in a pair, so where that happens, we shove in a Q, which won't confuse the reader. Now our message runs:

AL QL IS KN OW NF LY AT ON CE

If there is an odd letter at the end, we'd add on another Q to square it up. Now we take our first group, AL. We see that they come at the corners of a rectangle in which the other corners are SP. So we put down SP for the first two letters of the coded message. In the same way, QL becomes SM and IS becomes FA.

Ah, but here's KN. They both come on the same vertical line. In this case you take the letter next below each -- TC. Next comes OW, which translates to MX. Going on, SK, PV, NP, TU...

If your first diagonal went from bottom to top, you must take it the same way again. ON = TU, NO would be UT.

CE come on the same horizontal line. In that case, you take the letter to the right of each. Since there isn't a letter to the right of C, you start again at the beginning of the line, producing DR.. Your coded message stands now:

SP SM FA TC MX SK PV NP TU DR

To make it look pretty and not give the method away, you can break it up into any lengths you like, or you can embellish it with punctuation as haphazard:

S.P. SMFA. TCMXS, KPVM, PT! UDR.

It's very ingenious. You can't guess it by way of the most frequent letter, because you get a different letter for each time, according as it's grouped to the next letter. And you can't guess individual words, because you don't know where the words begin and end. Is it at all possible to decode it without the key-word?

"Oh dear, yes," said Wimsey. "Any code ever coded can be decoded with pains and patience..."

Input

The input for your program will be a series of keywords and messages to encode, alternating line-by-line, until the end-of-file flag of 999. Using the above technique, you are to encode the message, using the word. For example:

SQUANDER

ALL IS KNOWN FLY AT ONCE

JUXTAPOSITION

THE ROOSTER CROWED AT MIDNIGHT

999

Input will be in upper case and will contain no punctuation. Unlike Sayers' example, letters may occur more than once in the key-words, in which case you must ignore all but the first occurrence of the letter.

Output

Print each message, encrypted, using two letter groups and no punctuation, separated by a single white space. Separate each message by one blank line.

For the sample input file, the first line of output will be:

SP SM FA TC MX SK PV NP TU DR

Print an I for IJ.

Notes:

If the last letter is both odd and repeated, treat it as repeated and put the 'Q' before it, not afterward. That is, ALL becomes ALQL for encoding purposes.

In the unlikely case of two Qs in a row, insert a Z between them. Also, augment odd-length messages ending with a Q by using a Z. Thus, FAQQAD becomes FAQZQADQ and HUQ becomes HUQZ for encoding.

题目简要描述：

要求设计一个编码程序，每个编码需要一个单词造一个 5*5 的表格，然后依据这个表格对给定信息进行编码。编码时将需编码的信息分解成若干段，每段两字母。而编码的过程就是找出表中每两字母所对应的另两字母。

输入：每个编码 case 分两行:第一行是给定的用来造表的单词；第二行是将要编码的信息。

输出：只需将对应字母按次序两两输出即可。

题目分析：

该问题将涉及几种特殊情况：

1 在用给定单词构造 5*5 表时:

1) 将输入的 i,j 视为一个字母 i，化入同一格.

2) 若同一词中一个字母出现两次或更多次，则只考虑第一个，将后面重复的全部忽略。

2 在将需编码的句子分解成两字母一组时：

1）当连续的两字母相同时，在其中插入 Q;当这两字母是 Q 时，插入 Z.

2）当最后只剩一个字母时，在其后补 Q；当这个字母是 Q 时，补 Z.

3 在找对应两字母时：

1）若这两字母既不同列也不同行，则每个字母的对应字母与该字母同行与另一字母同列。

2）若两字母同列，则对应字母取该字母下面的，若下面无字母时，取该列的第一个。

3）若两字母同行，则对应字母取该字母后面的，若后面无字母时，取该行第一个。

数据结构的选用及算法：

1）输入并判断：

因给定单词和所需编码的信息都以行分隔，所以用 getline(),它将读到换行符时停止并在每个字符串后加空字符。判断输入字符串为 999 时停止；否则转 2）。

2）构造 5*5 表格并构造 25 个结构体将表格中的字母和其行列标对应：用函数 sheet()完成。

int sign[26]:

用 ASCII 将该数组与 26 个字母对应起来，其初始值为 0。按顺序每读入字符串中一字母时若对应的 sign 为 0，则将其放入列 s[25]中并将 sign 置 1；若为 1，则跳过继续读下一个。

在读入子母前，将所有 j 改成 i。则 sign[9]无意义(可将其在初始时就置为 1)。

char s[25]：

用来顺序存放 5*5 表格中的字母，当字符串中的字母全部读完后，若还不满 25 则顺序检查 sign,将 sign 为 0 的对应字母放入 s 中填满。

char sh[5][5]:将 s[25]中的字母顺序放入，即得 5*5 表格。

struct pin p[26]:在构成 5*5 表格时将每个字母的行列标存入 p 中，用 ASCII 对应每个字母。

3）将输入信息编码 encode()

a)将读入信息两两字母分组

注：t1,t2 分别存放分好组的两字母。be 存放上一组的后一字母。

一） 先将 be=0,t1,t2 分别输入两字母，（注：将输入的 j 全换成 i)

二） 若 t1 不为空字符:

当 be==t1 且 be==Q 调用 fin(Z,t1)；be!=Q 则 fin(Q,t1),然后 t1=t2,输入 t2,be=t1,转二）。

当 t1==t2,t1==Q 调用 fin(t1,Z),be=Z;t1!=Q 则 fin(t1,Q),be=Q,然后 t1=t2,输入 t2,转二）。

当 t2 为空字符时，若 t1==Q 调用 fin(t1,Z),t1!=Q 则 fin(t1,Q),然后转三）。

其他情况，fin(t1,t2)并 be=t2,输入 t1,t2,转二）。

若 t1 为空字符，转三）。

三） 输出空行，结束。

b)找对应字母 fin(char,char)

先将传入两字母的对应行列坐标找到，可直接调用 p[]存放在 a1,a2,b1,b2 中再分别按情况处理

```
if (a1==a2)
    out<<sh[a1][(b1+1)%5]<<sh[a1][(b2+1)%5]<<' ';
if(b1==b2)
    out<<sh[(a1+1)%5][b1]<<sh[(a2+1)%5][b1]<<' ';
if(a1!=a2 && b1!=b2)
    out<<sh[a1][b2]<<sh[a2][b1]<<' ';
```

代码参考：略

二、密　　码

密码学是研究加密和解密变换的一门科学。通常情况下，人们将可懂的文本称为*明文*；将明文变换成的不可懂的文本称为*密文*。把明文变换成密文的过程叫*加密*；其逆过程，即把密文变换成明文的过程叫*解密*。明文与密文的相互变换是可逆的变换，并且只存在唯一的、无误差的可逆变换。完成加密和解密的算法称为*密码体制*。

在 ICPC 竞赛中，有关密码学的问题是经常出现的，一般只是对文本信息的加密，并且密码体制是给定的。竞赛中的密码问题所需求的内容归纳以来有以下几点：

1. 加密、解密过程的模拟。此类问题的求解思路很简单，难点在于密码体制相对复杂，所以读懂题目中所描述的加密和解密算法是很关键之所在。
2. 密文编码。此类问题的关键在于使用合理的数据结构编码并存储。
3. 密文解密。次类问题是经常出现的。给定了加密的算法，设计合理的解密算法求解明文。我们在这里主要讨论的就是此类问题。

明文加密的主要方法就是利用的已有的字符映射（可以是明文字符间的映射，也可以是明文字符与一串字符的映射）关系，把明文字符替换为映射字符，这样可懂得明文就变成了一组不可读懂的密文，我们把这一过程称为加密，字符映射关系就是加密算法。自然，获得这种映射关系是解密的关键，密码问题是没有其特定的算法的，但往往映射关系的求解涉及到了其它算法，比如字符串处理、图中的一些算法。

字符映射的方法多种多样，具体到某一题目中又是不同的，所以解密的算法必须特定于具体的题目中。

在解密的多种算法中，有的是根据明文和密文之间的关系求解，有的是根据某一数学公式求解，还有的是直接穷举求解。其实密码问题的算法是不固定的，大多依赖于其它问题算法的变形。在此，选出了几道具有代表性的竞赛题目，在求解的过程中介绍一些方法，希望能对读者有一定的启发。

1. Problem A

Problem A
Morse Mismatches
Samuel F. B. Morse is best known for the coding scheme that carries his name. Morse code is still used in international radio communication. The coding of text using Morse code is straightforward. Each character (case is insignificant) is translated to a predefined sequence of dits and dahs (the elements of Morse code). Dits are represented as periods (".") and dahs are represented as hyphens or minus signs ("-"). Each element is transmitted by sending a signal for some period of time. A dit is rather short, and a dah is, in perfectly formed code, three times as long as a dit. A short silent space appears between elements, with a longer space between

characters. A still longer space separates words. This dependence on the spacing and timing of elements means that Morse code operators sometimes do not send perfect code. This results in difficulties for the receiving operator, but frequently the message can be decoded depending on context.

In this problem we consider reception of words in Morse code without spacing between letters. Without the spacing, it is possible for multiple words to be coded the same. For example, if the message "dit dit dit" were received, it could be interpreted as "EEE", "EI", "IE" or "S" based on the coding scheme shown in the sample input. To decide between these multiple interpretations, we assume a particular context by expecting each received word to appear in a dictionary.

For this problem your program will read a table giving the encoding of letters and digits into Morse code, a list of expected words (context), and a sequence of words encoded in Morse code (morse). These morse words may be flawed. For each morse word, your program is to determine the matching word from context, if any. If multiple words from context match morse, or if no word matches perfectly, your program will display the best matching word and a mismatch indicator.

If a single word from context matches morse perfectly, it will be displayed on a single line, by itself. If multiple context words match morse perfectly, then select the matching word with the fewest characters. If this still results in an ambiguous match, any of these matches may be displayed. If multiple context words exist for a given morse, the matching word will be displayed followed by an exclamation point ("!").

We assume only a simple case of errors in transmission in which elements may be either truncated from the end of a morse word or added to the end of a morse word. When no perfect matches for morse are found, display the word from context that matches the longest prefix of morse, or has the fewest extra elements beyond those in morse. If multiple words in context match using these rules, any of these matches may be displayed. Words that do not match perfectly are displayed with a question mark ("?") suffixed.

The input data will only contain cases that fall within the preceding rules.

Input

The Morse code table will appear first and consists of lines each containing an uppercase letter or a digit C, zero or more blanks, and a sequence of no more than six periods and hyphens giving the Morse code for C. Blanks may precede or follow the items on the line. A line containing a single asterisk ("*"), possibly preceded or followed by blanks, terminates the Morse code table. You may assume that there will be Morse code given for every character that appears in the context section.

The context section appears next, with one word per line, possibly preceded and followed by blanks. Each word in context will contain no more than ten characters. No characters other than upper case letters and digits will appear. Thered will be at most 100 context words. A line containing only a single asterisk ("*"), possibly preceded or followed by blanks, terminates the context section.

The remainder of the input contains morse words separated by blanks or end-of-line characters. A line containing only a single asterisk ("*"), possibly preceded or followed by blanks, terminates the input. No morse word will have more than eighty (80) elements.

Output

For each input morse word, display the appropriate matching word from context followed by an exclamation mark ("!") or question mark ("?") if appropriate. Each word is to appear on a separate line starting in column one.

Sample Input

```
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       --..
0       ------
1       .-----
2       ..---
3       ...--
4       ....-
5       .....
```

6 -....
7 --...
8 ---..
9 ----.
*
AN
EARTHQUAKE
EAT
GOD
HATH
IM
READY
TO
WHAT
WROTH
*
.--.....-- .....--....
--.----.. .--.-.----..
.--.....-- .--.
..-.-.-....--.-..-.--.-.
..-- .-...--..-.--
---- ..--
*

Output for the Sample Output
WHAT
HATH
GOD
WROTH?
WHAT
AN
EARTHQUAKE
IM!
READY
TO
IM!

* 题目大意
莫斯码依靠短暂的停顿来间隔每个字母和每个单词的输入，本题目中的密文是一串没有间隔的莫斯码序列，这样一来，同一个密文可匹配不同的明文；题目假设明文都是出现在给定的字典中的，即使这样，仍可能造成多个匹配或者不匹配。题目要求对于给定的密文，输出唯一匹配的明文；如果有多个明文匹配，则选择包含字母数目最少的，如仍有多个，

选择任意一个，输出所选明文并加上"!"后缀；如果没有与之匹配的明文，则选择匹配密文前缀个数最多的，如对应多个，选择任意一个，输出所选明文并加上"?"后缀。
数据的输入、输出格式详见题目。
* 解题思路
在本题中，由于字母之间没有间隔标志，将给出的密文直接译为明文与字典中的单词进行匹配是非常困难的，也是不切合实际的。我们可以把字典中单词都译成莫斯码，然后再将给出的密文与之匹配，这样问题就转化为了字符串的匹配问题。直接有字典中的单词与密文匹配，有多个单词与密文匹配，这两种情况的处理都相对简单。我们着重讨论无单词完全匹配的处理。
题目要求当无单词与密文完全匹配的时候，选择能匹配密文前缀个数最多的单词。我们可采用这样的算法：将密文每次去掉一个后缀（一位莫斯码，'.'或'-'），再与字典中的单词比较，看密文是否为比较单词的前缀，是则找到匹配单词，否则进行循环，直到找到与密文匹配的单词。判断是否为单词前缀的方法为：如果单词莫斯码长度等于当前比较的密文，则比较二者是否相同，是则为镇，反之亦反；如长度大于密文，则删除多余后缀，再进行比较。按照题目中的说明，最后肯定是能找到匹配的单词的，算法复杂度。

* 解题程序源码：**略**

* 总结
本题的算法并不复杂，关键在于不完全匹配单词的处理问题上，可以把本题归为解密算法的模拟类。

2. Problem B

Problem B
Do the Untwist
Cryptography deals with methods of secret communication that transform a message (the plaintext) into a disguised form (the ciphertext) so that no one seeing the ciphertext will be able to figure out the plaintext except the intended recipient. Transforming the plaintext to the ciphertext is encryption; transforming the ciphertext to the plaintext is decryption. Twisting is a simple encryption method that requires that the sender and recipient both agree on a secret key k, which is a positive integer.
The twisting method uses four arrays: plaintext and ciphertext are arrays of characters, and plaincode and ciphercode are arrays of integers. All arrays are of length n, where n is the length of the message to be encrypted. Arrays are origin zero, so the elements are numbered from 0 to n - 1. For this problem all messages will contain only lowercase letters, the period, and the underscore (representing a space).
The message to be encrypted is stored in plaintext. Given a key k, the encryption method works as follows. First convert the letters in plaintext to integer codes

in plaincode according to the following rule: '_' = 0, 'a' = 1, 'b' = 2, ..., 'z' = 26, and '.' = 27. Next, convert each code in plaincode to an encrypted code in ciphercode according to the following formula: for all i from 0 to n - 1, ciphercode[i] = (plaincode[ki mod n] - i) mod 28.
(Here x mod y is the positive remainder when x is divided by y. For example, 3 mod 7 = 3, 22 mod 8 = 6, and -1 mod 28 = 27. You can use the C '%' operator or Pascal 'mod' operator to compute this as long as you add y if the result is negative.) Finally, convert the codes in ciphercode back to letters in ciphertext according to the rule listed above. The final twisted message is in ciphertext. Twisting the message cat using the key 5 yields the following:

```
Array
0
1
2
plaintext
'c'
'a'
't'
plaincode
3
1
20
ciphercode
3
19
27
ciphertext
'c'
's'
'.'
```

Your task is to write a program that can untwist messages, i.e., convert the ciphertext back to the original plaintext given the key k. For example, given the key 5 and ciphertext 'cs.', your program must output the plaintext 'cat'.
The input file contains one or more test cases, followed by a line containing only the number 0 that signals the end of the file. Each test case is on a line by itself and consists of the key k, a space, and then a twisted message containing at least one and at most 70 characters. The key k will be a positive integer not greater than 300. For each test case, output the untwisted message on a line by itself. Note: you can assume that untwisting a message always yields a unique result. (For those of you with some knowledge of basic number theory or abstract algebra, this will be the case provided that the greatest common divisor of the key k and length n is 1, which it will be for all test cases.)

Example input:
5 cs.
101 thqqxw.lui.qswer
3 b_ylxmhzjsys.virpbkr
0
Example output:
cat
this_is_a_secret
beware._dogs_barking

* 题目大意
题目介绍了一种加密方法，实际上是利用散列函数根据公匙产生字符间的映射，从而将可懂的明文加密为不可懂的密文。题目规定了明文均为小写的字母以及点和下划线，每个明文字符均事先以整数 0-27 固定其编码；将明文(plaintext)和密文(ciphertext)序列的每个字符位分离开，将整个字符序列表示为起始位为 0 的字符数组，那么明文和密文数组均对应一个同长度的整型数组 plaincode 和 ciphercode，存放各个字符的编码。明文和密文间的映射关系均由散列函数决定：ciphercode[i] = (plaincode[ki mod n] ? i) mod 28，其中 i 为数组的下标，n 为明文长度，k 为公匙。根据散列函数可将明文每个字符位加密成密文。
题目所要求的是解密，即根据已知的散列函数和提供的公匙，将密文译为明文。
* 解题思路
本题的关键在于解密函数的求解。依靠数学方法，不难推出：
plaincode[ki mod n] = (ciphercode[i] + i) mod 28
将加密过程的散列函数进行替换，即为解密方法。
在本题目的程序中使用了数组的动态创建方法，特说明如下：
在 C++的一般数组声明中，除了指定了其存储元素外，均需要指定其大小（长度），且大小是一个常量，例如 int array[10]或 const n=10; int array[n]均正确，而 int array[n]是错误的。事实上可采用变量指定数组大小，数组的声明和创建方法如下，以整型为例：
```
int n = 10;
int* array = new int[n];
```
用此方法创建的数组和一般的数组使用方法是一样的。但需要特别注意的是，当所创数组的生命周期结束后，要对数组析构，收回所占用的内存空间：delete []array;

* 解题程序源码：**略**

* 总结
本题目主要是解密散列函数的求解，对于编程技巧要求不高。此类利用散列函数进行字符映射的题目，一般均能求解出其逆映射的函数，且散列函数是双射的。在本题目的求解程序中使用了一维数组的动态创建方法，再介绍一下二维数组的动态创建方法：
数组的动态创建，实际上是人为的为其分配内存空间，所以分配后应注意收回，二维数组的创建是以一维数组为基础的，是为一维数组中的每个元素再分配空间。假设我们创建二维数组 array[7][10]:

```
int row = 7;
int col = 10;
int** array = new int*[row];
for(int i=0; i<row; i++)
        array = new int[col];
```
创建完毕，生命周期结束后进行析构：
```
    for(int i=0; i<row; i++)
            delete []array[i];
    delete []array;
```
我们还可以为每行指定不同的列数，就不再举例了。
在函数中将一维、二维数组当作参数使用时，可分别用指针和二级指针声明。例：
```
void functional(int* array, int** array)
```
这样，二维数组可当作参数传递了。

3. Problem C

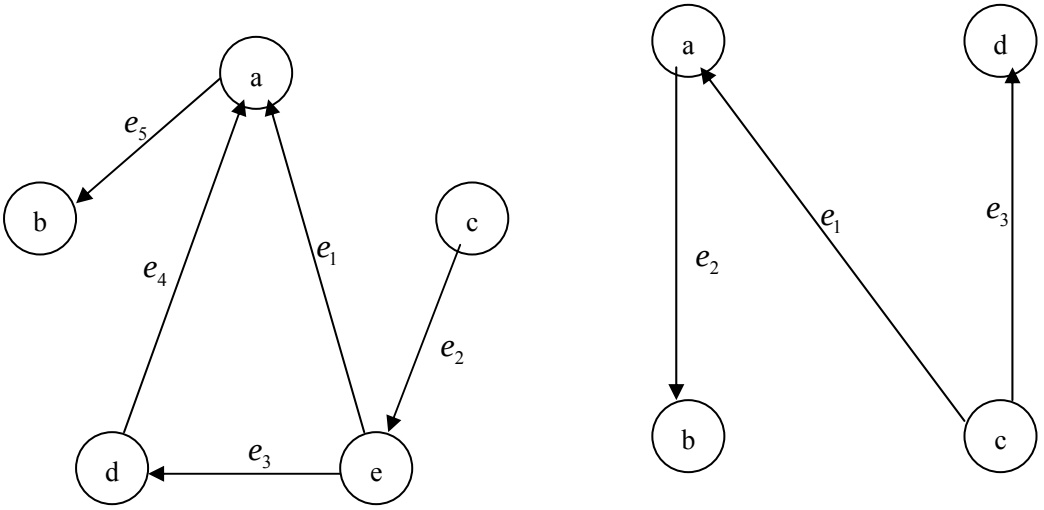
Problem C
Substitution Cipher
Antique Comedians of Malidinesia would like to play a new discovered comedy of Aristofanes. Putting it on a stage should be a big surprise for the audience so all the preparations must be kept absolutely secret. The ACM director suspects one of his competitors of reading his correspondece. To prevent other companies from revealing his secret, he decided to use a substitution cipher in all the letters mentioning the new play.
Substitution cipher is defined by a substitution table assigning each character of the substitution alphabet another character of the same alphabet. The assignment is a bijection (to each character exactly one character is assigned -- not neccessary different). The director is afraid of disclosing the substitution table and therefore he changes it frequently. After each change he chooses a few words from a dictionary by random, encrypts them and sends them together with an encrypted message. The plain (i.e. non-encrypted) words are sent by a secure channel, not by mail. The recipient of the message can then compare plain and encrypted words and create a new substitution table.
Unfortunately, one of the ACM cipher specialists have found that this system is sometimes insecure. Some messages can be decrypted by the rival company even without knowing the plain words. The reason is that when the director chooses the words from the dictionary and encrypts them, he never changes their order (the words in the dictionary are lexicographically sorted). String a1a2 ... ap is lexicografically smaller than b1b2 ... bq if there exists an integer i, i <= p, i <= q, such that aj=bj for each j, 1 <= j < i and ai < bi.
The director is interested in which of his messages could be read by the rival company.

You are to write a program to determine that.
Input Specification
The input consists of N cases. The first line of the input contains only positive integer N. Then follow the cases. The first line of each case contains only two positive integers A, 1 <= A <= 26, and K, separated by space. A determines the size of the substitution alphabet (the substitution alphabet consists of the first A lowercase letters of the english alphabet (a--z) and K is the number of encrypted words. The plain words contain only the letters of the substitution alphabet. The plain message can contain any symbol, but only the letters of the substitution alphabet are encrypted. Then follow K lines, each containing exactly one encrypted word. At the next line is encrypted message.
Output Specification
For each case, print exactly one line. If it is possible to decrypt the message uniquely, print the decrypted message. Otherwise, print the sentence 'Message cannot be decrypted.'.
Sample Input
2
5 6
cebdbac
cac
ecd
dca
aba
bac
cedab
4 4
cca
cad
aac
bca
bdac
Output for the Sample Input
abcde
Message cannot be decrypted.

* 题目大意
存在一个字符映射表，使得明文和密文字符是双射的。从字典中按单词的小写拼写次序顺序随机选出一些单词，并译为密文提供。在有些情况下，即使没有字符映射表，根据提供的有序密文我们仍可知道字符间的映射关系：字符串 a1a2 ... ap 字典序小于 b1b2 ... bq ，如果存在整数 i，i <= p, i <= q，且 aj = bj 1 <= j< I，有 ai < bi 。如果我们能到字典序序列，题目规定序列对应着字典中的前 n 个小写英文字母（a-z），所以我们就得到了

字符映射表。题目要求将给出的明文译为密文，并规定明文中不包含任何符号和除字符映射表中的任何字符。
输入、输出数据格式详见题目描述。

* 解题思路
本题的关键在于求解出密文所包含字符的字典序序列，有了这一序列即可得到字符映射表，加密、解密便迎刃而解。
由相邻（次序相邻）密文间的大小关系，可知两字符的偏序关系<a,b>(a<b)，我们可得到所有相邻字符间的偏序集。以测试数据为例：
第一组：{<e,a>, <c,e>, <e,d>, <d,a>, <a,b>, <b,c>}
第二组：{<d,a>, <a,b>, <b,c>}
通过这些偏序关系，我们可将其构成有向图：



这样一来，问题转化为了图的问题，由偏序集求解全序关系是有算法的——拓扑排序，算法过程描述如下：
1. 在图中所有结点中选取一个入度为 0 的结点，加入序列中，如有多个选任一个，如没有则排序失败；
2. 将选出结点及其连接的边从图中删除；
3. 如所有结点均被选出，则排序成功，否则，返回 1。
当然，本题目的求解还需将算法的过程改造，主要是排序成功的控制条件改造。由题目可知，最终产生的字典序列是唯一的，也就是说字符映射表是唯一的；所以有多解的拓扑排序不能产生唯一的字符映射表，那么密码就不能被破解。所以过程 1 改为：
1. 在图中所有结点中选取一个入度为 0 的结点，加入序列中，如有多个 或没有则排序失败；
按算法过程求解：
第一组：c 被首先选出，删除 c 和；选出 e，删除 e 和；选出 d，删除 d 和；选出 a，删除 a

和；最后是 b。最终得序列 c,e,d,a,b。
第二组：c 被首先选出，删除 c 和；a、d 均可选，所以排序失败。
所有字符的全序关系产生后，字符映射表自然得出。将密文译成明文，题目求解完毕。
有向图用邻接矩阵 A 存储，稍加变形：该结点有向邻接的结点标记为 1，A[i][i]存放第 I 结点的入度。以第一组数据为例：

* 解题程序源码：**略**

三、字符串处理

字符串的处理是一项基本功，是解题的工具。

在C++的STL库中有一个头文件<string.h>，里面定义了一个string类型和一些关于它的操作，熟练的掌握了string这种类型的使用技巧，可以使很多题目的处理简单化。在稍后的应用举例中我将会向大家介绍高精度算法，这是string类型的最典型的应用。

首先介绍string的定义，string实际上可以看作字符数组，可以直接加下标取出每个字符；

接下来介绍一些函数，在这里我就不再具体说明用法和用途，只介绍一些常用的函数名称，大家可以自己查看附录。常用的函数有：length, size, append, clear, insert, erase, find, find_first_of, find_last_of, find_first_not_of, find_last_not_of等等，还有一些可以方便应用的运算符：+, ==, = 等；

1. PROBLEM A

Word Reversal

Time limit: 30 Seconds Memory limit: 32768K

Total Submit: 348 Accepted Submit: 130

For each list of words, output a line with each word reversed without changing the order of the words.

This problem contains multiple test cases!

The first line of a multiple input is an integer N, then a blank line followed by N input blocks. Each input block is in the format indicated in the problem description. There is a blank line between input blocks.

The output format consists of N output blocks. There is a blank line between output blocks.

Input

You will be given a number of test cases. The first line contains a positive integer indicating the number of cases to follow. Each case is given on a line containing a list of words separated by one space, and each word contains only uppercase and lowercase letters.

Output

For each test case, print the output on one line.

Sample Input

1

3

I am happy today

To be or not to be

I want to win the practice contest

Sample Output

I ma yppah yadot
oT eb ro ton ot eb
I tnaw ot niw eht ecitcarp tsetnoc
题目大意：将一句话中的每一个单词中的每一个字母按从右向左输出
源程序：**略**

2. PROBLEM B

IBM Minus One

Time limit: 30 Seconds Memory limit: 32768K
Total Submit: 273 Accepted Submit: 162

You may have heard of the book '2001 - A Space Odyssey' by Arthur C. Clarke, or the film of the same name by Stanley Kubrick. In it a spaceship is sent from Earth to Saturn. The crew is put into stasis for the long flight, only two men are awake, and the ship is controlled by the intelligent computer HAL. But during the flight HAL is acting more and more strangely, and even starts to kill the crew on board. We don't tell you how the story ends, in case you want to read the book for yourself :-)
After the movie was released and became very popular, there was some discussion as to what the name 'HAL' actually meant. Some thought that it might be an abbreviation for 'Heuristic ALgorithm'. But the most popular explanation is the following: if you replace every letter in the word HAL by its successor in the alphabet, you get ... IBM.
Perhaps there are even more acronyms related in this strange way! You are to write a program that may help to find this out.
Input
The input starts with the integer n on a line by itself - this is the number of strings to follow. The following n lines each contain one string of at most 50 upper-case letters.
Output
For each string in the input, first output the number of the string, as shown in the sample output. The print the string start is derived from the input string by replacing every time by the following letter in the alphabet, and replacing 'Z' by 'A'.
Print a black line after each test case.
Sample Input
2
HAL
SWERC
Sample Output
String #1
IBM

String #2
TXFSD
题目大意：题目要求我们把给出的单词中的每一个字母换成它的下一个字母，如果是 Z 则换成 A；
源程序：**略**

3. 字符串处理的应用实例

下面通过介绍高精度算法来具体说明 string 类型的用法：

高精度算法：因为我们常用的类型，如 float 类型等都有精度和位数的限制，有些时候不能满足我们的实际要求，高精度算法通过程序模拟手工算法来实现整形或浮点型的基本运算，如加减乘除等，从而突破精度和位数的限制，达到我们的要求。

下面是一个写好的整型高精度算法，包括运算，大家可以参考注释来学习一下：

```
#include <string>
#include <iostream>

using namespace std;

//整数的加减乘除，取模运算
string ADD_INT (string str1, string str2);
string MULTIPLY_INT (string str1, string str2);
string MINUS_INT (string str1, string str2);
string DIVID_INT (string str1, string str2, int z);
string DIV_INT (string str1, string str2);
string MOD_INT (string str1, string str2);
//比较两个整数的大小
int comp (string str1, string str2);

int comp (string str1, string str2)
{
  if (str1.length()>str2.length())          //长度长的整数大于长度小的整数
    return 1;
    else if (str1.length()<str2.length())
           return -1;
           else return str1.compare(str2);  //若长度相等,从头到尾按位比较,
compare 函数：相等返回 0，大于返回 1，小于返回－1
}

//整数加法
string ADD_INT (string str1, string str2)
```

```
{
    //处理正负号
  int b=1;
  string str;
  if (str1[0]=='-')
    if (str2[0]=='-') {
        b=-1;
        str=ADD_INT(str1.erase(0,1),str2.erase(0,1));
      }
      else str=MINUS_INT(str2,str1.erase(0,1));
    else {
           if (str2[0]=='-')
             str=MINUS_INT(str1,str2.erase(0,1));
           else {
                //把两个整数对齐，长度不等时，在短的整数前加 0 补齐长度
                  int l1=str1.length(),l2=str2.length();
                  if (l1<l2)
                    for (int i=1; i<=l2-l1; i++)
                      str1="0"+str1;
                    else for (int i=1; i<=l1-l2; i++)
                           str2="0"+str2;
                //按位带进位加
                  int int1=0,int2=0;    //Int2 记录进位
                  for (int i=str1.length()-1; i>=0; i--) {
                      int1=(int(str1[i])-48+int(str2[i])-48+int2)%10;
                      int2=(int(str1[i])-48+int(str2[i])-48)/10;
                      str=char(int1+48)+str;
                    }
                  if (int2!=0)
                    str=char(int2+48)+str;
                }
         }
    //运算后处理符号
  if ((b==-1) && (str[0]!='0'))
    str="-"+str;
  return str;
}

//整数减法
string MINUS_INT (string str1, string str2)
{
  string str;
```

```
    //处理符号，有的可以转换成加法
  int b=1;
  if (str2[0]=='-')
    str=ADD_INT(str1,str2.erase(0,1));
    else {
           if (comp(str1,str2)==0)
             return "0";
           if (comp(str1,str2)<0) {
               swap(str1,str2);
               b=-1;
}
//按位带接位减
           int tempint=str1.length()-str2.length(),i;
           for (i=str2.length()-1; i>=0; i--) {
               if (str1[i+tempint]<str2[i]) {
                   str1[i+tempint-1]=char(int(str1[i+tempint-1])-1);
                   str=char(str1[i+tempint]-str2[i]+58)+str;
                 }
               else str=char(str1[i+tempint]-str2[i]+48)+str;
             }
           for (i=tempint-1; i>=0; i--)
             str=str1[i]+str;
         }
    //去除结果中头上多余的 0
  str.erase(0,str.find_first_not_of('0'));
  if (str.empty())
str="0";
//给结果加上符号
  if ((b==-1) && (str[0]!='0'))
    str="-"+str;
  return str;
}

//整数的乘法
string MULTIPLY_INT (string str1, string str2)
{
  string str;
    //处理符号位
  int b=1;
  if (str1[0]=='-') {
      str1=str1.erase(0,1);
      b=b*-1;
```

```
    }
  if (str2[0]=='-') {
      str2=str2.erase(0,1);
      b=b*-1;
}
//按位实现手工乘法
  for (int i=str2.length()-1; i>=0; i--) {
      int int1=0,int2=0,int3=int(str2[i])-48;
      string tempstr;
      if (int3!=0) {
          int j;
          for (j=1; j<=int(str2.length()-1-i); j++)
            tempstr="0"+tempstr;
          for (j=str1.length()-1; j>=0; j--) {
              int1=(int3*(int(str1[j])-48)+int2)%10;
              int2=(int3*(int(str1[j])-48)+int2)/10;
              tempstr=char(int1+48)+tempstr;
            }
          if (int2!=0)
            tempstr=char(int2+48)+tempstr;
        }
      str=ADD_INT(str,tempstr);
}
//去除结果中头上多余的 0
  str.erase(0,str.find_first_not_of('0'));
  if (str.empty())
        str="0";
//给结果加上符号
  if ((b==-1) && (str[0]!='0'))
    str="-"+str;
  return str;
}

//整数的除法，Z＝1 时返回商，Z=2 时返回余数
string DIVIDE_INT (string str1, string str2, int z)
{
  string quotient,residue;
  int a=1,b=1;
    //判断除数为 0
  if (str2[0]=='0') {
      quotient="ERROR!";
      residue="ERROR!";
```

```
      if (z==1)
        return quotient;
        else return residue;
    }
  if (str1[0]=='0') {
      quotient="0";
      residue="0";
}
//处理符号位
  if (str1[0]=='-') {
      str1=str1.erase(0,1);
      a=a*-1;
      b=-1;
    }
  if (str2[0]=='-') {
      str2=str2.erase(0,1);
      a=a*-1;
    }
  if (comp(str1,str2)<0) {
      quotient="0";
      residue=str1;
    }
  if (comp(str1,str2)==0) {
      quotient="1";
      residue="0";
    }
  if (comp(str1,str2)>0) {
      int length1=str1.length(), length2=str2.length();
      string tempstr;
tempstr.append(str1,0,length2-1);
//模拟手工除法
      for (int i=length2-1; i<length1; i++) {
          tempstr=tempstr+str1[i];
            //试商
          for (char ch='9'; ch>='0'; ch--) {
              string str;
              str=str+ch;
              if (comp(MULTIPLY_INT(str2,str),tempstr)<=0) {
                  quotient=quotient+ch;
                  tempstr=MINUS_INT(tempstr,MULTIPLY_INT(str2,str));
                  break;
                }
```

```
            }
          }
        residue=tempstr;
}
//去除结果中头上多余的 0
  quotient.erase(0,quotient.find_first_not_of('0'));
    //给结果加上符号
  if (quotient.empty())
    quotient="0";
  if ((a==-1) && (quotient[0]!='0'))
    quotient="-"+quotient;
  if ((b==-1) && (residue[0]!='0'))
    residue="-"+residue;
  if (z==1)
    return quotient;
    else return residue;
}

string MOD (string str1, string str2)
{
  return DIVIDE_INT(str1,str2,2);
}

string DIV (string str1, string str2)
{
  return DIVIDE_INT(str1,str2,1);
}
```

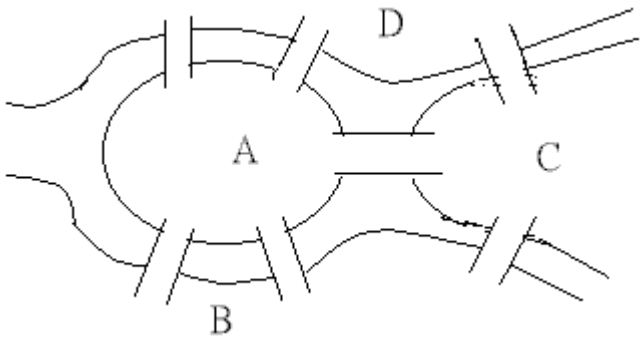
四、 图论的一些算法

（一）概述

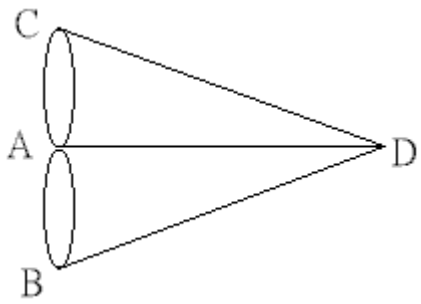
图论（Graph Theory）是数学的一个分支。它以图为研究对象。图论中的图，是由若干给定的点及连接两点的线构成的图形，这些图形通常用来描述某些事物之间的某种特定关系，用点代表事物，用连接两点的线表示相应两个事物间具有这种关系。

图论本身是应用数学的一部分，因此，历史上图论曾经被好多位数学家各自独立地建立过。关于图论的文字记载最早出现在欧拉 1736 年地论著中，他所考虑的原始问题有很强的实际背景。

图论起源于著名的柯尼斯堡七桥问题。在柯尼斯堡的普莱格而河上有七座桥将河中的岛及岛与河岸联结起来，如下图所示，A、B、C、D 表示陆地。



问题是要从这四块陆地中任何一块开始，通过每一座桥正好一次，再回到起点。然而无数次的尝试都没有成功。欧拉在 1736 年解决了这个问题，他用抽象分析法将这个问题转化为第一个图论问题：即把每一块陆地用一个点来代替，将每一座桥用连接相应的两个点的一条线来代替，从而相当于得到一个图，（如下图）。欧拉证明了这个问题没有解，并且推广了这个问题，给出了对于一个给定的图可以某种方式走遍的判定法则。这项工作使欧拉成为图论的创始人。



1859 年，英国数学家哈密顿发明了一种游戏：用一个规则的实心十二面体，它的 20 个顶点标出世界著名的 20 个城市，要求游戏者找一条沿着各边通过每个顶点刚好一次的闭

回路，即绕行世界。用图论的语言来说，游戏的目的是在十二面体的图中找出一个生成圈。这个问题后来就叫做哈密顿问题。由于运筹学、计算机科学和编码理论中的很多问题都可以化为哈密顿问题，从而引起广泛的注意和研究。

在图论的历史中，还有一个著名的问题——四色猜想。这个猜想说，在一个平面或球面上的任何地图能够只用四种颜色来着色，使得没有两个相邻的国家有相同的颜色，每个国家必须由一个单连通域构成，而两个国家相邻是指它们有一段公共的边界，而不仅仅只有一个公共点。

（二）若干特殊问题

上面所介绍的，都是图论中的一些经典问题，下面我们将重点介绍图论中的几个重要算法。

◆ 匹配问题

1 匹配的基本概念

首先，让我们把匹配的定义再明确地说一下，设 G＝[V,E]是一个无向图，M⊆E 是 G 的若干条边的集合，如果 M 中的任意两条边都没有公共端点，就称 M 是一个匹配。

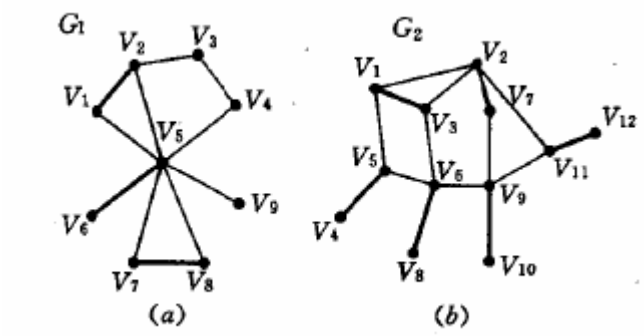


图 6－1

例如图 6－1（a）（b）中的粗线组成的边集合 M_1，M_2，是图 G_1 与 G_2 的匹配。

从给定的图 G＝[V,E]的所有匹配中，把包含边数最多的匹配找出来，就是所谓的最大匹配问题。

2 下面我们首先介绍几个概念：

1) 二分图

二分图是这样一种图：G 的顶点集合分成两个部分 X 和 Y，G 中每条边的两个端点一定是一个属于 X 而另一个属于 Y。

2) 饱和与非饱和

若匹配 M 的某条边与顶点 v 关联，则称 M 饱和顶点 v，并且称 v 是 M－饱和的，否则称 v 是 M－不饱和的。

3) 交互道

若M是二分图G＝[V,E]的一个匹配。设从G中的一个顶点到另一个顶点存在一条道路，这条道路是由属于M的边和不属于M的边交替出现组成的，则称这条道路为交互道。

4) 可增广道

若一条交互道的两端点为关于M非饱和顶点时，则称这条交互道是可增广道路。显然，一条边的两端点非饱和，则这条边也是可增广道路。

5) 最大匹配

如果M是一匹配，而不存在其他匹配M_1，使得$|M_1|>|M|$，则称M是最大匹配。其中|M|表示匹配M的边数。

Hall 定理：对于二分图G，存在一匹配M，使得X的所有顶点关于M饱和的重要条件是：

对于X的任一子集A，和A邻接的点集为Γ（A），恒有：

| Γ（A） | ≥ |A|

证明略

3 匈牙利算法：

匈牙利算法是求二分图匹配的一种算法，它的根据就是Hall定理中充分性证明的思想，现将算法描述如下：

（1） 初始任意匹配。

（2） 如X已饱和则结束，否则转（3）

（3） 在X中找一非饱和点X_0，作 $V_1 \leftarrow \{X_0\}$，$V_2 \leftarrow \varnothing$

（4） 若Γ（V_1）＝V_2，则因无法匹配而停止；否则任选一点y∈Γ（V_1）\V_2。

（5） 若y已饱和转（6），否则作

【求一条从$X_0 \rightarrow$y的可增广道路P，M←M⊕E(P)，转（2）】

（6） 由于y已饱和，故M中有一条边（y，z），作

【$V_1 \leftarrow V_1 \cup \{z\}$，$V_2 \leftarrow V_2 \cup \{y\}$,转（4）】

参考代码：

```
int hungary(){
    //从0开始
    int s[N],t[N],p,q,ret=0,i,j,k,match1[N],match2[N];

    memset(match1,-1,sizeof(match1));
     memset(match2,-1,sizeof(match2));

    for (i=0;i<n;ret+=(match1[i++]>=0)){
        memset(t,-1,sizeof(t));
        for (s[p=q=0]=i;p<=q&&match1[i]<0;p++)
            for (k=s[p],j=0;j<m&&match1[i]<0;j++)
                if (mat[k][j]&&t[j]<0){
                    s[++q]=match2[j];t[j]=k;
                    if (s[q]<0)
                        for (p=j;p>=0;j=p){
                            match2[j]=k=t[j];p=match1[k];match1[k]=j;
                        }
                }
    }
    return ret;
}
```

注意：这里是将递归改为了非递归，递归程序大家可以自己试试。

4 最佳匹配

引例：工厂有 m 个工人 n 项工作，每个工人可以完成其中的若干项工作且工作效益各不相同，我们需要制定一个分工方案，使公司的总效益最大。这就是最佳匹配的问题。

解决这种问题的常用算法是基于匈牙利算法修改顶标，详细请参考图论书。

◆ 网络流问题

1 网络流的基本概念

若有向图 G=(V,E)满足下列条件者，则称其为网络流图。

（1） 有且仅有一个顶点 Z，它的入度为 0，即 $d^-(Z)=0$，这个点便称为源点。

（2） 有且仅有一个顶点 $\bar{Z}$，它的出度为 0，即 $d^+(\bar{Z})=0$，这个点便称为汇点。

（3） 每一条边都有非负数，叫做该边的容量，边（v_i，v_j）的容量用 c_{ij} 表示。

对于网络流图 G，每一条边（i, j）都给定一个非负数 f_{ij}，这一组数满足下列两个条件时，称为这网络流的容许流，用 f 表示它：

a) $f_{ij} \le c_{ij}$

b)除源点 Z 和汇点 $\bar{Z}$ 以外所有的点 v_i，恒有：

$$\sum_j f_{ij} = \sum_k f_{ki}$$

c)对于源和汇有 $\sum_i f_{zi} = \sum_j f_{j\bar{z}} =$w, w 就叫做该网络流的流量。

2 分析与求解

首先，让我们定义两个基础概念，剩余网络和增广路径。

剩余网络：对于任意的网络流图，剩余网络和原网络有相同的顶点，对于原来有边的两个顶点，存在一条或两条边。更进一步，如果流沿边 $x \to y$ 并且流量小于边 $x \to y$ 的容量，那么存在一条边 $x \to y$，其容量为 $c_{xy} - f_{xy}$（称为残余容量）；如果流量沿边 $x \to y$ 为正数，那么存在一条反向边 $y \to x$，其容量为 f_{xy}。

增广路径：增广路径就是在剩余网络中从源点到汇点的一条可行路径。增广流量是沿着路径的所有边的容量的最小值。

让我们来举一个简单的例子：

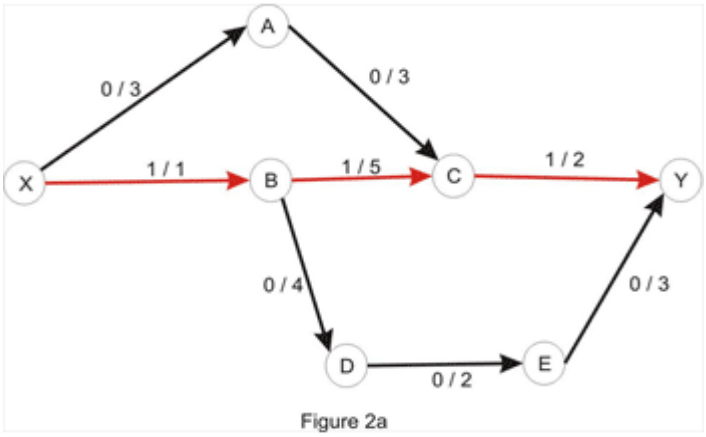


Figure 2a

其剩余网络如下所示：

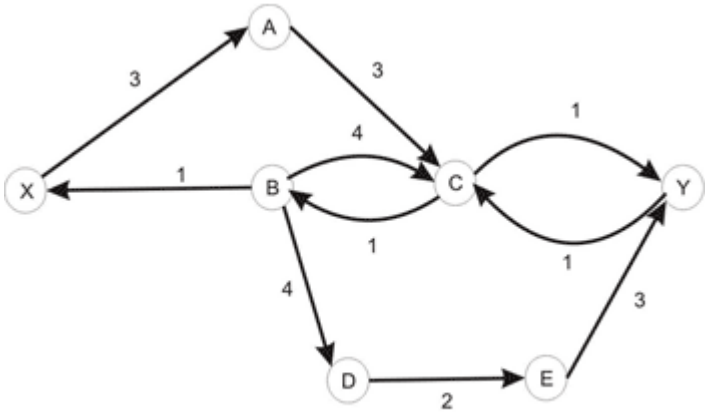


Figure 2b - The residual network of the network in 2a

考虑路径 $X \to A \to C \to Y$,我们可以将流量增加 1，边 $X \to A$ 和边 $A \to C$ 有容量 3，边 $C \to Y$ 有容量 1，所以我们取最小值得到增广流量 1。沿着路径将流量增加 1，得到下面的网络流图：

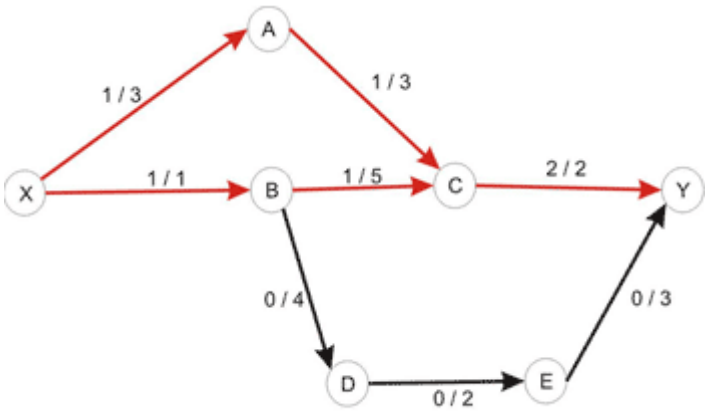


Figure 3a

现在的流量是 2，但不是最大流量，所以我们还需要继续增广，首先得到上图的剩余网络图：

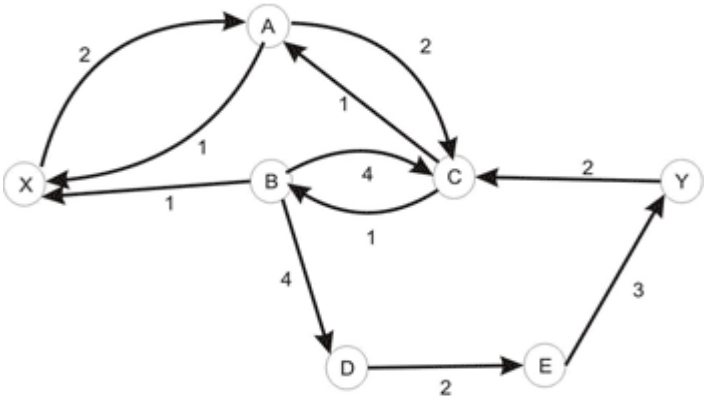


Figure 3b - The residual network of the network in 3a

显然，你可以找到的唯一增广路径为 $X \to A \to C \to B \to D \to Y$（注意在原图中不存在这样的路径，因为 $C \to B$ 为反向边，增加它的流量就相当于减少 $B \to C$ 边的流量），同样我们得到增广流量为 1。

增广之后，我们便得到下面的剩余网络图：

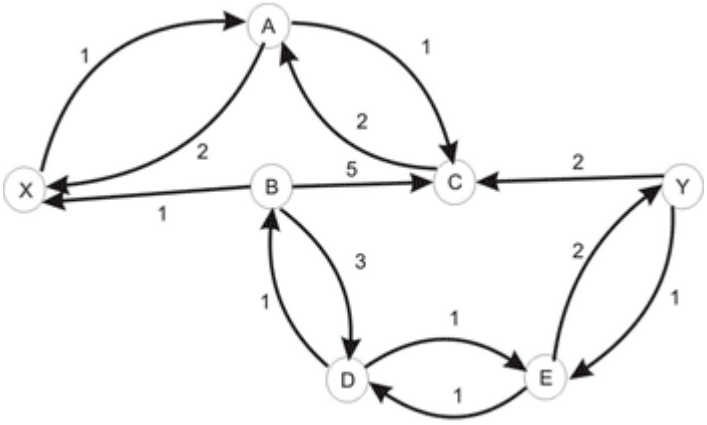


Figure 1b - The residual network of the network in 1a

可以看出已经不存在任何增广路径，所以得到该网络的最大流量为 3。

注意：该算法的结论的正确性由割切定理可以证明，证明过程略。

最大流算法小结：

1） 初始化网络，使所有流量为 0；

2） 不断地寻找增广路径并进行增广；

3） 当找不到增广路径时，算法终止。

下面我们接着讨论最大流算法中一个最重要的问题，寻找增广路径：

1） DFS

2F 算法在解决这个问题时采用的是 DFS 算法，即在剩余网络中采用 DFS 算法，试图寻找一条增广路径，当找到一条时，便停止搜索，并进行增广操作。此算法绝对可以保证正确性，而且实现起来代码非常简洁，但是有些盲目，所以导致有些时候时间效率非常差。

2） BFS

修正的 EK 算法正是采用了这种方法。我们知道，BFS 算法可以解决没有权值或权值相等的图的最短路径问题，确实，这样我们就可以得到从源点到汇点的边数最少的增广路径。这个算法可以保证最多需要 O(N*M/2)次增广，（N 表示顶点数，M 表示边数）每次 BFS 的时间复杂度为 O（M），所以整个算法的最差时间复杂度为 O（N* M^2 ），但往往实际运用中要比这个好得多。

3） PFS

这个算法在寻找增广路径时，和 Dijkstra 算法极为相似，每次都是找出可增广的流量最大的增广路径进行增广。

分析：直觉告诉我们，最后一个算法可能效果最好，因为每次都是增广最大，这样就可以减少增广次数，但是实践证明，对于不同的情况，这三种算法各有千秋，算法的好坏是要对具体问题看哪个更加适用。

五、算法的优化

在比赛中可能最令人头疼的就是想出了算法并写出代码提交后得到超时的回复，这当然和算法本身选择是否合理有关，如需要用动态规划解决的题目却选择了搜索算法。但是除了这个方面外，还有就是选择了正确的算法，但是用这个算法解决问题的时候效率不够高，在试题规定的时间范围内无法计算完数据。这就涉及到算法的优化问题。而算法的优化是建立在对算法本身的熟练掌握的基础上的，在本章中，笔者将谈一谈在平常做题的过程中对算法优化的一些体会。在此，再重申一下对算法的优化，需要对算法本身熟练掌握。

在本节中，笔者将讨论以下内容：

1．算法优化的基本思想

2．搜索的优化

3．动态规划的优化

4．一些特殊的数据结构

在本节中，笔者将不给出大部分代码，请大家自己动手完成。

（一）算法优化的基本思想

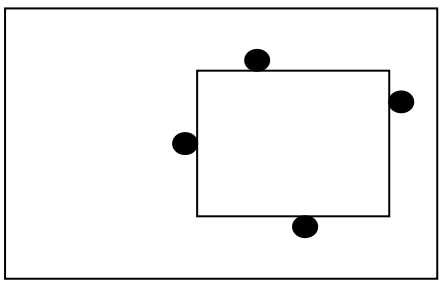
在算法设计和编程过程中，冗余计算的出现是难以避免的。冗余计算是高效率的天敌，减少冗余计算，必然会使算法和程序效率提高很多。去除冗余计算没有可套用的定理公式可用，只有认真分析、善于探索，并在做题中积累经验，才能得到去除冗计算余的好方法。

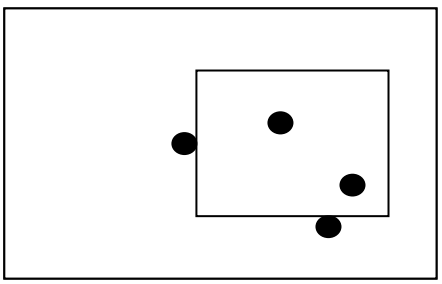
下面是一个常用的模型，笔者将通过对此模型算法的优化来说明算法优化的基本思想。

最大子矩形问题：

在一个给定的矩形中有一些障碍点，要找出内部不包含任何障碍点的，轮廓与整个矩形平行或重合的最大子矩形。矩形中障碍点的数目为 S，矩形的大小为 M*N。

首先，定义**有效子矩形**为内部不包含任何障碍点的，边界与坐标轴平行的子矩形。如下图所示，左边是有效子矩形，右边不是。





然后，定义**极大子矩形**为每条边都不能向外扩展的有效子矩形；定义**最大子矩形**为所有有效子矩形中最大的一个（或多个）。

显然，在一个有障碍点的矩形中的最大子矩形一定是一个极大子矩形（但是究竟是哪一个极大子矩形，在求解前我们并不知道）。 于是我们得出求解这个问题的基本算法思想：设计算法的思路：通过枚举所有的极大子矩形，找出最大子矩形。

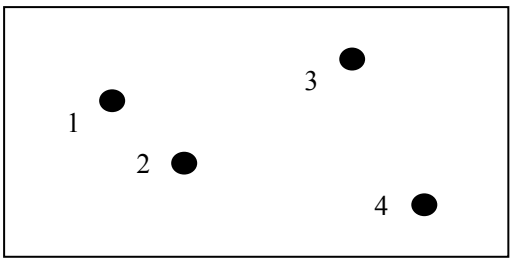
最先想到的算法可能是：枚举上下左右四个边界，然后判断组成的矩形是否是有效子矩形。算法的时间复杂度：O(S^5)。显然这个算法的效率太低，可以改进的地方：产生了大量的无效子矩形。

初步改进算法：枚举左右边界，然后对处在边界内的点排序。每两个相邻的点和左右边界一起组成一个矩形。复杂度：O(S^3)。经过初步改进后，这个算法的时间复杂度大大降低，但是还是比较高。可以改进的地方：枚举了部分不是极大子矩形的情况。

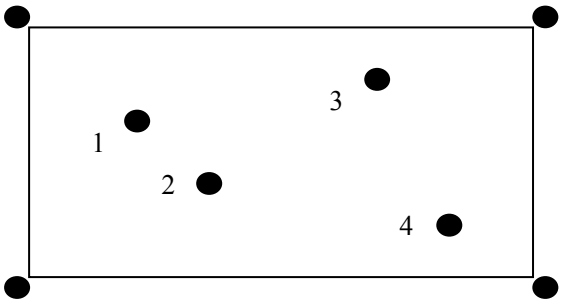
虽然上述初步改进后的算法，但是它的改进思想还是很有用的：保证每一个枚举的子矩形都是有效的；保证每一个枚举的子矩形都是极大的。显然改进思想就是避免大量的冗余计算。

两个算法的思路都是先通过一定的方式不遗漏的枚举出每一个极大子矩形，然后从中找出面积最大的一个。第一个算法的枚举方式为：枚举上下左右四个边界；第二个算法的枚举方式为：枚举左右边界。于是想到可否枚举极大子矩形的左边界，然后根据确定的左边界，找出相关的极大子矩形，并检查和处理遗漏的情况。

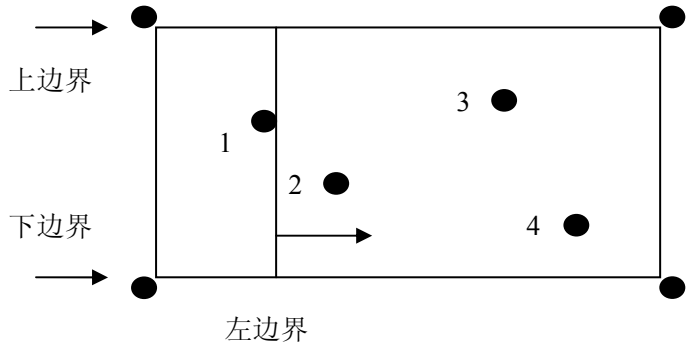
首先，将所有障碍点按横坐标从小到大的顺序将点标为 1 号点，2 号点……



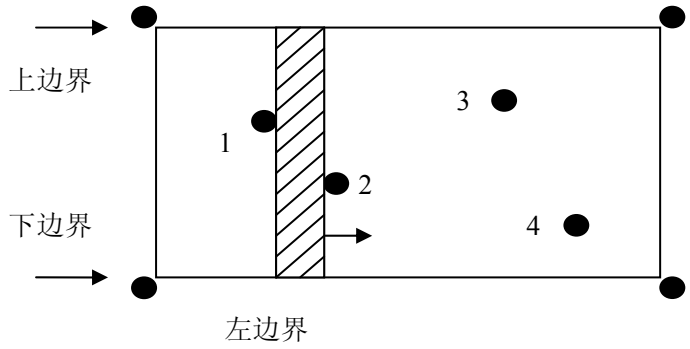
为了处理方便，在矩形的四个顶点上各增加 1 个障碍点。



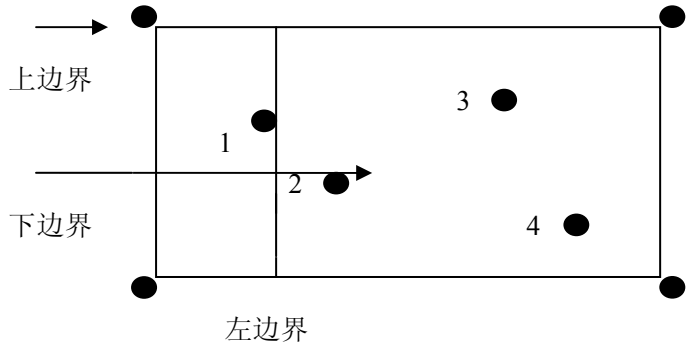
第一次取 1 号点作为所要枚举的极大子矩形的左边界，设定上下边界为极大子矩形的上下边界。



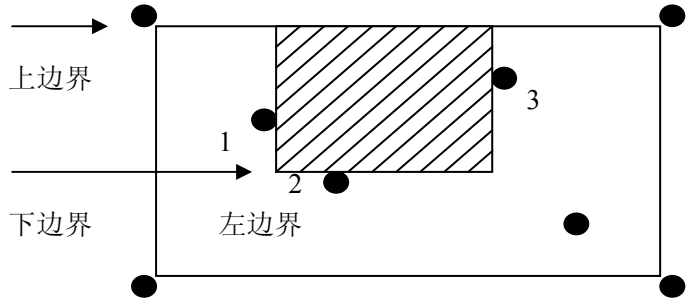
从左向右扫描，第一次遇到 2 号点，可以得到一个有效的极大子矩形，如图所示：



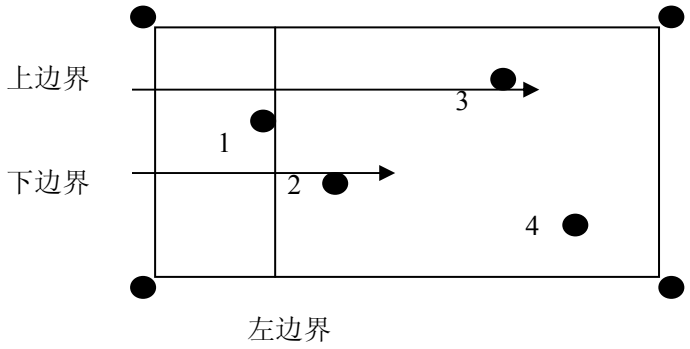
因为左边界覆盖 1 号点且右边界在 2 号点右边的有效子矩形都不能包含 2 号点，所以需要修改上下边界，2 号点在 1 号点下方，因此要修改下边界。



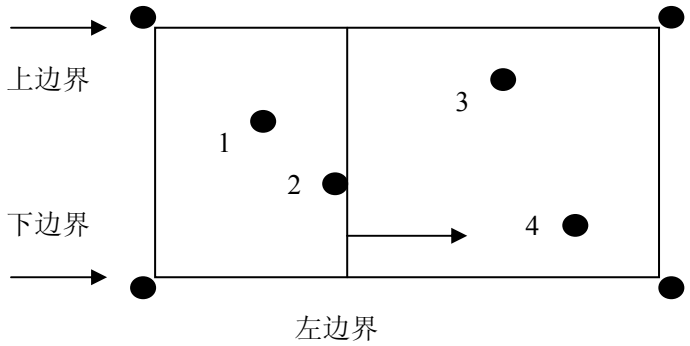
继续扫描到 3 号点，又得到一个极大有效子矩形，如图所示：



因为 3 号点在 1 号点上方，所以要修改上边界。

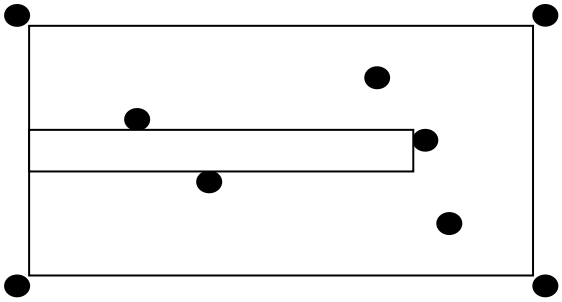


以此类推，可以得到所有以 1 号点为左边界的极大有效子矩形。然后将左边界移动到 2 号点、3 号点……横坐标的位置。开始扫描以 2 号点、3 号点……为左边界的极大子矩形。

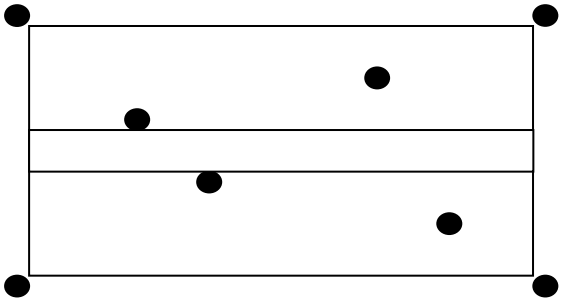


前面的做法可以找出所有左边界覆盖了一个障碍点的极大子矩形，此外，还有两类遗漏的情况：

一类是左边界与整个矩形的左边界重合，右边界覆盖一个障碍点的情况。解决方法：用类似的方法从右向左扫描一次。



另一类是左边界与整个矩形的左边界重合，且右边界也与整个矩形的右边界重合的情况。解决方法：预处理时增加特殊判断。



算法的时间复杂度为 O(S^2)，空间复杂度为 O(S)。算法的时间复杂度已经可以接受了，而且在计算过程中，计算的矩形都是极大子矩形，大大减少了冗余计算，提高了算法的效率。

（二）搜索的优化

在竞赛中，我们有时会碰到一些题目，它们既不能通过建立数学模型解决，又没有现成算法可以套用，或者必须遍历所有状况才可以得出正确结果。这时，我们就必须采用搜索算法来解决问题。

搜索算法按搜索的方式分有两类，一类是深度优先搜索，一类是广度优先搜索。我们知道，深度搜索编程简单，程序简洁易懂，空间需求也比较低，但是这种方法的时间复杂度往往是指数级的，倘若不加优化，其时间效率简直无法忍受；而广度优先搜索虽然时间复杂度比前者低一些，但其庞大的空间需求量又往往让人望而却步。

所以，对程序进行优化，就成为搜索算法编程中最关键的一环。

笔者在这里所要介绍的便是搜索算法中优化程序的一种基本方法即“剪枝”。

1）什么是剪枝

相信刚开始接触搜索算法的人，都做过类似迷宫这样的题目吧。我们在“走迷宫”的时候，

一般回溯法思路是这样的：

1、这个方向有路可走，我没走过

2、往这个方向前进

3、是死胡同，往回走，回到上一个路口

4、重复第一步，直到找着出口

这样的思路很好理解，编程起来也比较容易。但是当迷宫的规模很大时，回溯法的缺点便暴露无遗：搜索耗时令人无法忍受。

我们可不可以在向某个方向前进时，先一步判断出这样走会不会走到死胡同里呢？这样一来，搜索的时间不就可以减少了吗？

答案是：可以的。

剪枝的概念，其实就跟走迷宫避开死胡同差不多。若我们把搜索的过程看成是对一棵树的遍历，那么剪枝顾名思义，就是将树中的一些“死胡同”，不能到达我们需要的解的枝条“剪”掉，以减少搜索的时间。

搜索算法，绝大部分需要用到剪枝。然而，不是所有的枝条都可以剪掉，这就需要通过设计出合理的判断方法，以决定某一分支的取舍。在设计判断方法的时候，需要遵循一定的原则。

2）剪枝的原则

1、正确性

正如上文所述，剪枝不是想剪就能剪的。如果随便剪枝，把带有最优解的那一分支也剪掉了的话，剪枝也就失去了意义。所以，剪枝的前提是一定要保证不丢失正确的结果。

2、准确性

在保证了正确性的基础上，我们应该根据具体问题具体分析，采用合适的判断手段，使不包含最优解的枝条尽可能多的被剪去，以达到程序“最优化”的目的。可以说，剪枝的准确性，是衡量一个优化算法好坏的标准。

3、高效性 设计优化程序的根本目的，是要减少搜索的次数，使程序运行的时间减少。但为了使搜索次数尽可能的减少，我们又必须花工夫设计出一个准确性较高的优化算法，而当算法的准确性升高，其判断的次数必定增多，从而又导致耗时的增多，这便引出了矛盾。

因此，如何在优化与效率之间寻找一个平衡点，使得程序的时间复杂度尽可能降低，同样是非常重要的。倘若一个剪枝的判断效果非常好，但是它却需要耗费大量的时间来判断、比较，结果整个程序运行起来也跟没有优化过的没什么区别，这样就太得不偿失了。

综上所述，我们可以把剪枝优化的主要原则归结为六个字：正确、准确、高效。

剪枝算法按照其判断思路可大致分成两类：可行性剪枝及最优性剪枝。下面分别结合例题对这两种方法进行阐述。

3）可行性剪枝

这个方向可不可以走？走下去会不会碰到死胡同？这就是对某一枝条进行可行性剪枝的简要判断过程。

我们现来看这样一道题。

问题简述：一个规则矩形网络状的城市，城市中心坐标为（0,0）。城市包含 m 个无法通行的路障(m<=50)，采用如下规则游历城市：第一步走 1 格，第二步走 2 格，依此类推，

第 n 步走 n 格(n<=20)，除了第一步有四个方向可走，其余各步必须在前一步基础上左转或右转 90 度，最后回到出发点（0,0）。对于给定的 n、m，编程求出所有可行的路径。

现在考虑采用简单的回溯法，原因是该题的本身就已经有很强的剪枝判断了。那么我们先来分析一下用回溯法解题的思路：

用 x，y 两个变量存储当前坐标，每一步对 x，y 的值进行修改，没有遇到障碍就继续走，走完 n 步看看有没有回到（0,0），没有的话回溯搜索，直到找完所有路径。

接着，我们来看看这种算法的时间复杂度。

一共走 n 步，每步要搜索四个方向，假设在最坏的情况下，没有任何障碍物，那么它的时间复杂度应该为：O(4^n)。

很明显，这样的算法效率并不会很高，所以我们必须对程序进行剪枝，在未走完 n 步之前就提早判断出这样的走法是否可行。

当走到第 i 步时，假设当前坐标为（xi，yi），那么离（0,0）的距离就应该是 xi+yi。

而剩下的 n-i 步可以走的最远距离则是(i+1)+(i+2)+……+n，即。所以，若(i+1)+(i+2)+……+n<xi+yi 的话，就表示就算现在“回头”也没办法到达出发点了，也就是说这条分支即便再搜索下去也找不出解来，这时我们已经可以舍弃这一分支而回溯了。

这样剪枝似乎已经不错了，但是，它的效果只有当数据较大时，才能体现得明显。除了上述的优化，还有没有其他的方法呢？

我们可以这样想，这个城市是规则网格状的，东、南、西、北四个方向都是对称的。打个比方说，与（1,0）这个点对称的可以有（-1,0），（0,1），（0,-1）这三点。那可否设想，当从一个方向出发，寻找到一个解之后，将这个解旋转 90^o、180^o、270^o，不就得出其余三个解了吗？这样岂不是节省了 3/4 的搜索次数？

由这个设想出发，我们可以设计出下面的优化：

忽略所有的障碍物，第一步走固定的方向（比如东面），在这个基础上搜索路径，每找到一条路径都将其余三个“对称路径”一起判断，看看有没有经过障碍物，若没有则该路径为解之一。

请大家思考为什么要忽略障碍物并且要限定第一步的方向。

4）最优性剪枝

最优性剪枝，又称为上下界剪枝。我们可以回想一下，平时在做一些要求最优解的问题时，搜索到一个解，是不是把这个解保存起来，若下次搜索到的解比这个解更优，就又把更优解保存起来？其实这个较优解在算法中被称为“下界”，与此类似还有“上界”。在搜索中，如果已判断出这一分支的所有子节点都低于下界，或者高于上界，我们就可以将它剪枝。

如何估算出上（下）界呢？我们引入一个 “估价函数”。最优性剪枝算法的核心，就在于设计估价函数上。估价函数在不同的题目中被赋予不同的意义，比如说当前状态与目标状态相差的步数，某一数列的长度等等，都可作为估价函数的一部分。

我们再来看一道题目：

问题简述：在一个 n*m 的迷宫矩阵中，有 x 个不可逾越的障碍物，给定 x0, y0, x1, y1, 求出由(x0, y0)到(x1, y1)之间的所有最短路径。

这一道题看起来非常简单，也与上一题有不少相似之处。难道我们不能用简单的回溯法来解决它吗？

我们来分析一下时间复杂度。与上题类似，我们同样假设在最坏情况下，迷宫中没有

任何障碍，即是一个网格，从(0,0)到(n,m)的话，每一步有四个方向可以走，时间复杂度将达到 O(4^m*4^n)！假设 n=m=20，那么搜索次数将达到 2^42 次！看来这枝是非剪不可了。

最初步的剪枝当然就是将走过的方格置为“障碍”，因为若重复通过同一格，最终结果必定不是最优解。

其次，我们可以将每一次搜索出的路径长度与上界比较（初始上界＝∞），不断降低上界，一旦出现路径长超出上界而仍未到达目标点，则放弃该搜索进程。因为就算继续搜索下去，这一条路径也必然比其他路径长，不是最优解。

要完成规模更大的迷宫，仅仅这样剪枝是不够的。我们还必须采取其他的方法，比如先用动态规划求出一个准确的上界，再依据此上界进行搜索等。

当然，对于迷宫这一类题目，搜索算法并不是最好的。我们完全可以用动态规划（用 Bfs 划分阶段），再 Dfs 出所有的最短路径。在这里引用这样的一道题，目的是想让大家更好的理解最优化剪枝的思路及其应用，请大家思考如何用 Bfs+Dp+Dfs 求解这一问题。

结合本题，我们可以得出最优化剪枝算法所应该注意的一些问题：

1、与可行性剪枝一样，最优性剪枝在保证正确性的同时，同样需要注意提高准确性和高效性，估价函数的计算极为频繁，必须尽量提高其运算效率，不要“拣了芝麻，掉了西瓜”，寻找准确与高效的平衡点，确实重要。

2、在使用最优化剪枝时，一个好的估价函数往往起到“事半功倍”的效果。在搜索之前，我们可以采用一些高效率的算法，如贪心法、动态规划、标号法等等，求出一个较优解，作为上（下）界，将有解的范围缩小，以尽可能的剪去多余的枝条。如上题中，可以采用动态规划求出上界，再进行搜索，效率提高了许多。

此外，不但深度优先搜索可以运用最优性剪枝，在广度优先搜索中，同样可以采用这种剪枝方法。还有一些比较典型的方法有：分支定界法、A*算法，博弈树等等。不过，这些方法一般空间复杂度都比深度优先搜索要大得多，如何取舍就要依据不同的题目做出相应的判断了。

5）小结

在搜索算法中，几乎都需要采用程序优化，以减少时间复杂度。而本文所论述的“剪枝”算法，就是最常见的优化方法之一。其基本思想还是减少冗余的计算。

编优化程序的过程中，不论运用的是可行性剪枝还是最优性剪枝，都离不开剪枝的三个原则即正确、准确、高效。

然而，尽管可以采用众多优化算法使得程序的效率有所提高，搜索算法本身的时间复杂度不能从本质上减少是不可改变的事实，我们在考虑对一道题采用搜索算法之前，不妨先仔细想想，有没有其他更好的算法。毕竟，优化程序也是一项“吃力不讨好”的工作，与其耗费大量的时间来优化搜索程序，倒不如多想，多写，尽量以非搜索算法解决问题。

总之，我们在比赛中应当多动动脑筋，从不同角度来思考问题，采取合适的算法解决问题。

利用 Hash 表剪枝

Hash 表最大的优点，就是把数据的存储和查找消耗的时间大大降低，几乎可以看成是常数时间；而代价仅仅是消耗比较多的内存。然而在当前可利用内存越来越多的情况下，用空间换时间的做法是值得的。另外，写代码比较容易也是它的特点之一。

下面笔者介绍一下在搜索中 Hash 表怎么剪枝：

大家都知道在搜索树中每个节点都代表了一个当前状态，举个例子：Zju 上有一道题目

Knight Move：求解国际象棋棋盘上任意两点间，一个马从一点到另一点的最少步数。

很显然这样的问题用 Bfs 求解，那么搜索树中的每一个状态可以用一个二元组来表示(x,y)，即当前马正处于棋盘上(x,y) 的位置。

如果不剪枝，这题会超时，原因是有大量的节点是重复的，并且我们思考以后就会发现：当一个马重复出现在一个位置时，再由这个位置出发所得到的路径肯定不是最优的，因为我们完全可以把重复走的那段路径去掉，从而得出一条更短的路径。

现在的问题是怎么剪枝，可以这样剪枝：每当扩展一个新节点的时候，就遍历先前的节点，查找是否该节点是否已经出现过。显然棋盘上共有 64 个格子，即使全部在树中，由于不重复，所需要遍历的节点数并不多。但是如果棋盘是 100*100 的呢？这个时候我们就需要很快的查找到该节点是否出现过。这里就可以利用 Hash 表。开一个二维数组 s[i,j]。若 s[x,y]== 0，即表示该节点尚未走到，若 s[x,y]==1，即表示该节点已被访问。这样即使棋盘很大，我们也可以用 O(1)的时间就得到该节点的信息。

Hash 表在剪枝中的用途还是很大的，其基本作用就是减少查询节点是否已被遍历所用的时间。但是利用 Hash 表剪枝有个先决条件，即当节点重复时，此后所得到的解不会比不重复时更优，或者题目不允许节点重复。

其它的一些剪枝方法

当然还有很多其它的一些剪枝的方法和技巧，大家在做题的时候充分分析题目的意思和条件，并且多动脑，多动手。剪枝的思想仍然是减少冗余的计算来提高算法的效率。

（三）动态规划的优化

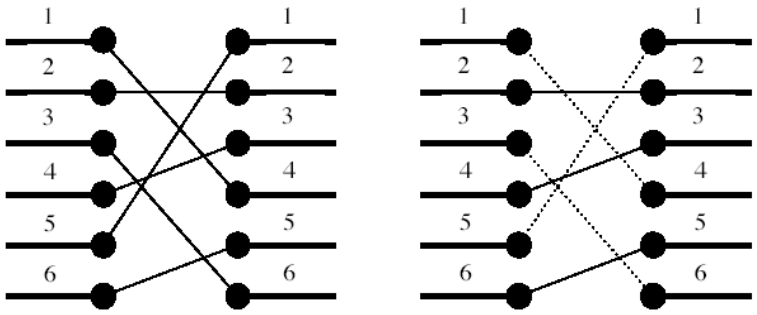
动态规划本身就是一种比较高效的算法，它的基本思想就是用空间换时间，采用记录表法对子问题的最优解进行记录，从而避免了子问题的重复计算，和递归相比节省了大量的时间，但是动态规划并不是没有优化的余地，随着参赛者水平的普遍提高，题目的数据规模越来越大，这就造成了两个问题：一是存储空间不够；二是解决问题的时间超出题目要求。下面从时间和空间两个方面来说一下动态规划的优化。

1）时间方面的优化（动态规划的剪枝）

动态规划的剪枝：在动态规划的过程中尽量减少不必要的计算，尽量减少不会得出最优解的计算，类似于搜索中的剪枝。

先给出两个题目，以此来说明：

Zju1986 Bridging Signals 题目的意思很简单，就是给出一个类似于函数双射的图，然后要求在图中的边集中选出一个子集，使得子集中的边互不相交，求解满足上述条件的所有子集中的边数最大值。如下面左图给出了一个有 6 条边的图，右图给出了它的一个最优解。数据的规模为 n<40000，n 为边数。



Zju2254 Island Country 题目大意：Alice 和 Jim 准备访问一些岛屿，这些岛屿从 1 到 n 编号，Alice 和 Jim 对这些岛屿的兴趣用一个整数来衡量，值越大表明对这个岛屿越感兴趣。但是他们两人对岛屿的兴趣不尽相同，所以他们要分开旅行。他们都按照自己对岛屿的兴趣值对导语进行访问，值低的先访问（好东西要留到后面，呵呵☺），值相同的按照编号大小进行访问，编号小的先访问。现在给出他们对岛屿的兴趣值，要求求解出他们在旅行的过程中最多遇见几次。

先说一下第 2 题，题目中的 Sample Input 如下：

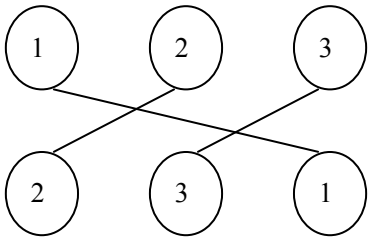
3

1 2 3

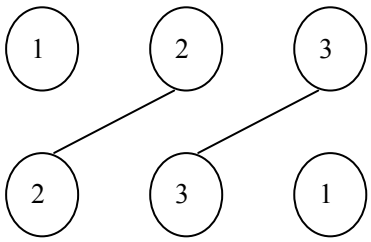
3 1 2

先输入岛屿的数量 n，n<=1000，接下来的两行分别是按岛屿的编号的顺序给出两人对这些岛屿的兴趣值。

显然本题应该用动态规划来做，先按岛屿的兴趣值对岛屿进行排序，然后就形成了两个岛屿编号的串，如题目中 Sample Input 排序后得出：



这样就转化成了和第一题相似的模型，即求解最大不相交边集。如上图中的最大不相交边集为：



现不妨称上面的序列为串 A，下面的序列为串 B。

从图中看出两人最多相遇两次。思考后得出初步算法如下：

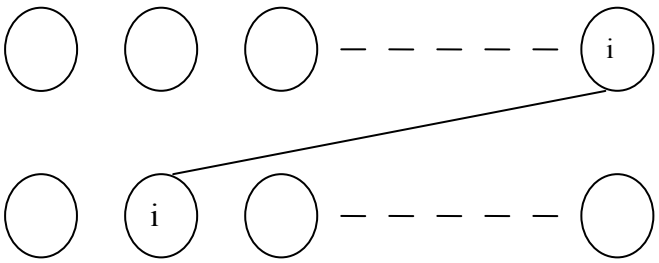
1. 先快速排序，得出两个岛屿编号的串，即串 A 和串 B。
2. 求解映射(Mapping)，即找出串 A 中的岛屿 i 在串 B 中的位置。
3. 求解最大不相交边集。

其中第 1 步算法复杂度为 O(nlogn)；第 2 步采用下述方法计算：对串 B 进行遍历，用散列 PosB[i]记录编号为 i 的岛屿在串 B 中的位置。算法复杂度为 O(n)；最重要的一步，也就是第 3 步，令状态变量 s[i]表示前 i 个岛屿，并且在第 i 个岛屿两人相遇的最优解。得出状态转移方程如下：

s[i] = max{ s[j]+1 (1<=j<i&&PosB[j]<PosB[i]) }

时间复杂度为 O(n^2)。整个算法的时间复杂度为 O(n^2)，也不算太高，似乎能够满足题目的要求，但是提交后发现超时，原因就是题目的数据量较大(n<=1000)，可是分析后，发现按照这个模型，降低算法复杂度比较困难(但不是不行)，不过在此笔者将着重介绍如何剪枝，首先分析算法中存在的不必要的计算，从而尽量减少不必要的计算时间。显然 s[i]的值只和前面的某一个 s[j]有关，但是为了寻找这个 s[j]，我们遍历了所有比 i 小的 s[j]，如何尽量快地找出满足题目要求的 j，或者如何尽量减少不必要的计算。分析后发现：

1. 会出现下述这种情况：



计算 i 的时候很明显如果按串 A 遍历，会造成很多不必要的计算；相反，如果按串 B 进行遍历，则只要计算 1 次。这是因为 i 在串 B 中比在串 A 中更靠前。

2. 也会出现下述情况：

再向前扫描的时候，前面的所有的 s[j]再加 1 也不会比 s[i]大，这就提醒我们新开一个数组 Max[j]来计算前 j 个值中的最大值，也就是

Max[j] = max{ s[k] (1<=k<=j) }

但是计算的时候当然不会按照这个 O(n)方程计算，而是应该用下面的 O(1)的方程计算：

Max[j] = max{ Max[j-1] , s[j] (j>1) }

采用这两个方法优化以后，虽然时间复杂度仍然是 O(n^2)，但是计算量大大减小，代码在规定时间内完成计算，说到这里第 1 题也可以用相同的优化方法解决。

这里优化的方法就是剪枝，另外还可以用二叉搜索树将算法复杂度降到 O(nlogn)，感兴趣的同学请参见《算法艺术与信息学竞赛》第 150 页。

当然这仅仅是从动态规划剪枝方面来讨论的，不过虽然没有降低时间复杂度，但是由于减少了大量不必要的计算，所以程序运行速度还是很快的。至于降低算法复杂度的方法，一般是利用专门的数据结构对信息进行存储，而不是像普通的动态规划那样只是用数组存储，不过这些数据结构的使用，使得编程复杂度提高，对选手的编程能力提出了更高的要求。

笔者建议当动态规划超时时，如果认为算法选择正确，那么先尽量剪枝，如果还是超时就应该考虑减低时间复杂度了。感兴趣的同学请参见《算法艺术与信息学竞赛》一书中的相关内容。

举例：最长单调序列的动态规划优化问题

最长单调序列是动态规划解决的经典问题。现在以求最长下降序列（严格下降）为例，来说明怎样用 O(nlogn)来解决它。设问题处理的对象是序列 a[1..n]。整个动态规划算法是这样实现的：

```
//Author:zheng
void Calculate()
{
    int max=1;
    s[1]=1;
    for(int i=2;i<=n;++i)
    {
        s[i]=1;
        for(int j=i-1;j;--j)
        {
            if(a[j]>a[i]&&s[j]+1>s[i])
            {
                s[i]=s[j]+1;
            }
        }
        if(max<s[i])
        {
            max=s[i];
        }
    }
    cout<<max<<endl;
}
```

这一程序非常短小精悍，其中的奥妙还是不少的。为了理解这个过程，还是从最基本的解决方法开始分析。

首先我们都知道求最长下降序列的算法：

$$s[1]=1$$

$$s[i]=MAX\{s[j]+1,1\le j<i\text{并且}a[j]>a[i]\}$$

$$P[i]=j(j\text{是上式中取}MAX\text{时的}j\text{值})$$

$$\max=MAX\{s[i](1\le i\le n)\}$$

在这个公式中 P 表示了决策。专门考虑这个 P，可能有多个决策都可以得到 M[i]得到最大值，这些决策都是等价的。那么我们当然可以对 P 进行特殊的限制，即，在所有等价的决策 j 中，P 选择 a[j]最大的那一个。

P 的选择跟我们得到结果并没有任何关系，但是希望对 P 的解释说明这样的问题：对于第 x 个数来说，它可以组成长度为 s[x]的最长下降序列，它的子问题是在 a[1..x-1]中的一个

长度为 s[x]-1 的最长下降序列，并且这个序列的最后一个数大于 a[x]。我们让 P 选择这些所有可能解中末尾数最大的，也就是说在处理完 a[1..x-1]之后，对于所有长度为 s[x]-1 的下降序列，P[x]的决策只跟其中末尾最大的一个有关，其余的同样长度的序列我们不再关心。

由此想到，用另外一个动态变化的数组 b，当我们计算完了 a[x]之后，a[1..x]中得到的所有下降序列按照长度分为 L 个等价类，每一个等价类中只需要一个作为代表，这个代表在这个等价类中末尾的数最大，我们把它记为 b[j],1≤j≤L。b[j]是所有长度为 j 的下降序列中，末尾数最大的那个序列的代表。

由于我们把 a[1..x-1]的结果都记录在了 b 中，那么处理当前的一个数 a[x]，我们无需和前面的 a[j]（1≤j≤x-1）作比较，只需要和 b[j]（1≤j≤L)进行比较。

对于 a[x]的处理，我们简单地说明。

首先，如果 a[x]<b[L]，也就是说在 a[1..x-1]中只存在长度为 1 到 L 的下降序列，其中 b[L]是作为长度为 L 的序列的代表。由于 a[x]<b[L]，显然把 a[x]接在这个序列的后面，形成了一个长度为 L+1 的序列。a[x]显然也可以接在任意的 b[j](1≤j<L)后面，形成长度为 j+1 的序列，但必然有 a[x]<b[j+1]，所以它不可能作为 b[j+1]的代表。这时 b[L+1]=a[x],即 a[x]作为长度为 L+1 的序列的代表，同时 L 应该增加 1。

另一种可能是 a[x]>=b[1]，显然这时 a[x]是 a[1..x]中所有元素中最大的，它仅能构成长度为 1 的下降序列，同时它又必然是最大的，所以它作为 b[1]的代表,b[1]=a[x]。

如果前面的情况都不存在，我们肯定可以找到一个 j,2≤j≤L,有 b[j-1]>a[x],b[j]≤a[x]，这时分析，a[x]必然接在 b[j-1]后面，新成一个新的长度为 j 的序列。这是因为，如果 a[x]接在任何 b[k]后面，1≤k<j-1，那么都有 b[k+1]>a[x]，a[x]不能作为代表。而对于任何的 b[k]，其中 j≤k≤L,b[k]<=a[x]，a[x]不能延长这个序列。由于 a[x]≥b[j],所以我们就将 b[j]更新为 a[x]。

在任何一种情况完成之后，b[1..L]显然是个下降的序列，但它并不表示长度为 L 的下降序列，这点不可混淆。

这几种情况实际上都可以归结为：处理 a[x]，令 b[L+1]为无穷小，从左到右找到第一个位置 j，使 b[j]≤a[x],然后则只要将 b[j]=a[x],如果 j=L+1,则 L 同时增加。x 处以前对应的最长下降序列长度为 M[x]=j。

这题当 a 本身是下降序列时，它退化为 $O(n^2)$的算法。这时不难想到既然 b 是一个有序数组，于是采用二分查找，于是算法的时间复杂度降为 O(nlogn)。

此外还有利用四边形不等式来加速动态规划，很多算法书上都有详细的讲解，在此就不再叙述了。感兴趣的同学可以自己查阅相关算法书籍。

2）在空间方面的优化

有时候状态数目太多，以至于存储不下。这时候就需要我们分析题目和状态转移方程，找出一些不需要存储的状态。经常碰到的情况是某些的状态在某个时刻后不会被用到，这个时候就不用存储这些状态。

（四）一些特殊的数据结构

1． 线段树：

处理涉及到图形的面积、周长等问题的时候，并不需要依赖很深的数学知识，但要提高处理此类问题的效率却又十分困难。这就需要从根本上改变算法的基础——数据结构。这里要说的就是一种特殊的数据结构——线段树。

先看一道较基础的题目：给出区间上的 n 条线段，判断这些线段覆盖到的区间大小。通过这题我们来逐步认识线段树。

定义：线段树是一棵二叉树，将一个区间划分为一个个[i，i+1]的单元区间，每个单元区间对应线段树中的一个叶子结点。每个结点用一个变量 count 来记录覆盖该结点的线段条数。

一棵线段树，记为 T(*a*, *b*)，参数 *a*,*b* 表示区间[*a*,*b*]，其中 *b*-*a* 称为区间的长度，记为 *L*。

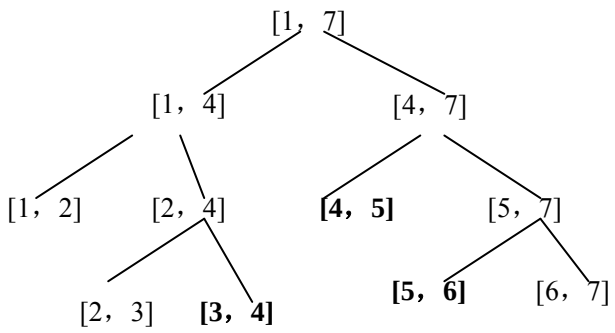
线段树 T(*a*,*b*)也可递归定义为：

若 *L*>1 : [a, (a+b) /2]为 T 的左儿子；

[(a+b)/2,b]为 T 的右儿子。

若 *L*=1 : T 为叶子节点。

区间[1，7]所对应的线段树如下图所示。区间上有一条线段[3，6]。



线段树既可以用链表存储，也可以用顺序表存储。而按顺序表存储时，可以把线段树看成一棵满二叉树（虽然有一些空节点没有用到），直接用 Tree[i]表示线段树中的节点，Tree[1]为根节点，那么左孩子为 Tree[i*2]（如果有左孩子），右孩子为 Tree[i*2+1]（如果有右孩子），于是建树的过程如下。

```
//Author:zheng
struct node{
        int Left;
        int Right;
    }Tree[TreeSize];//TreeSize 为树的大小
void BuildTree(int sn,int left,int right)
{
    int mid=(left+right)/2;
    Tree[sn].Left=left;
```

```
    Tree[sn].Right=right;
    if(right>left+1)
    {
        BuildTree(sn*2,left,mid);
        BuildTree(sn*2+1,mid,right);
    }
}
```

也可以不这样按层编号，而按照左先序编号，建树代码如下：

```
//Author:zheng
struct node{
            int Left;
            int Right;
            int LeftSon;//左孩子的编号
            int RightSon;//右孩子的编号
        }Tree[TreeSize];//TreeSize 为树的大小
Static int total;
void BuildTree(int left,int right)
{
    int now=++total,mid=(left+right)/2;
    Tree[now].Left=left;
    Tree[now].Right=right;
    if(right>left+1)
    {
        Tree[now].LeftSon=total+1;
        BuildTree(left,mid);
        Tree[now].Right=total+1;
        BuildTree(mid,right);
    }
}
```

插入一条线段：每个节点增加一个变量 Count 记录该节点被所有插入线段覆盖的次数，自根节点往下，直到一个被线段完全覆盖的节点。于是树节点的结构要作改动，插入线段的代码如下：

```
//Author:zheng
struct node{
            int Left;
            int Right;
            int LeftSon;//左孩子的编号
            int RightSon;//右孩子的编号
            int Count;//被线段覆盖的次数
        }Tree[TreeSize];//TreeSize 为树的大小
void Insert(int num,int left,int right)
{
```

```
    int mid=(left+right)/2;
    if(left<=Tree[num].Left&&right>=Tree[num].Right)
    {
        ++Tree[num].Count;
        return;
    }
    if(left<mid)
    {
        Insert(Tree[num].LeftSon,left,mid);
    }
    if(right>mid)
    {
        Insert(Tree[num].RightSon,mid,right);
    }
}
```

在线段树中删除和插入线段的方法类似，都采用递归逐层向两个子结点扫描，直到线段能够盖满结点表示的整个区间为止。请大家自己写出代码。

经过分析可以发现：在线段树中插入、删除线段的时间复杂度均为 O（logN）。线段树的深度不超过 log$L$。线段树把区间上的任意一条线段都分成不超过 2logL 条线段。这些性质为线段树能在 O(log$L$)的时间内完成一条线段的插入、删除、查找等工作，提供了理论依据。

结点的测度 m 指的是结点所表示区间中线段覆盖过的长度。

$$m=\begin{cases} j-i(count>0) \\ 0(count=0且节点为叶节点) \\ LSon.m+RSon.m(count=0且节点为内部节点) \end{cases}$$

连续线段数 line 指的是区间中互不相交的线段条数。连续线段数并不能像测度那样将两个子结点中的连续线段数简单相加。于是我们引进了两个量 lbd，rbd，分别表示区间的左右两端是否被线段覆盖。

$$lbd=\begin{cases} 0(左边端点被线段覆盖到) \\ 1(左边端点不被线段覆盖到) \end{cases}$$

$$rbd=\begin{cases} 0(右边端点被线段覆盖到) \\ 1(右边端点不被线段覆盖到) \end{cases}$$

line 可以根据 lbd，rbd 定义如下：

$$line=\begin{cases} 1(count>0) \\ 0(count=0且节点为叶节点) \\ LSon.line+RSon.line-1(count=0且节点为内部节点,LSon.rbd与RSon.lbd都为\ \ 1) \\ LSon.line+RSon.line(count=0且节点为内部节点，LSon.rbd与RSon.lbd不同时为\ 1) \end{cases}$$

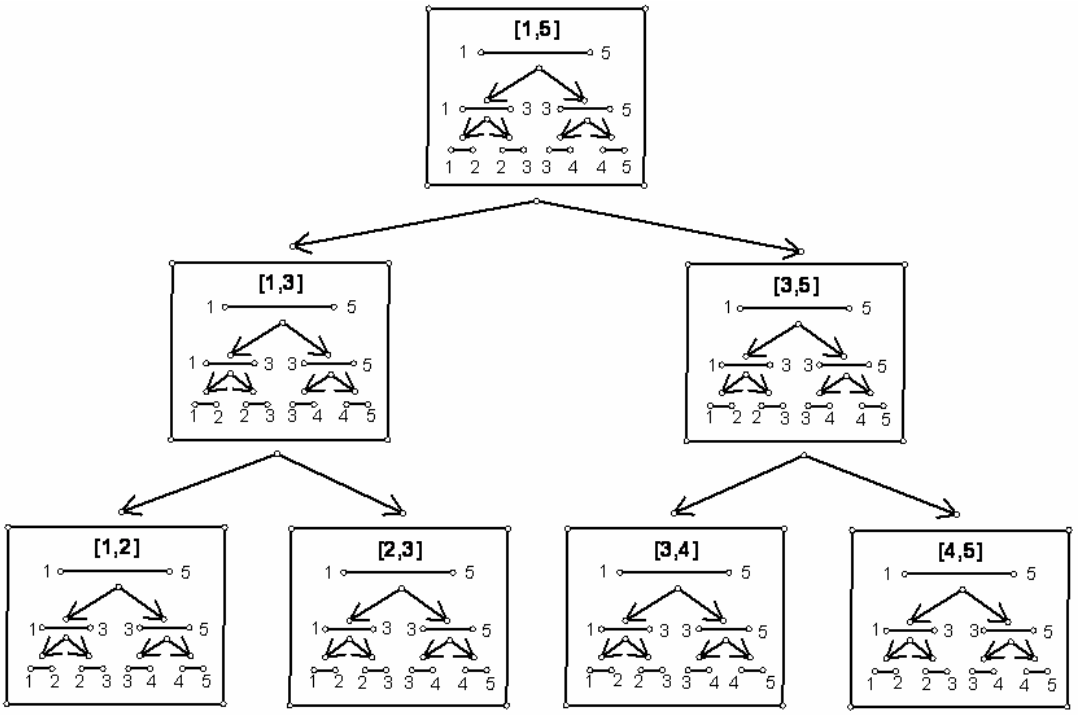
线段树能解决一些最基本的统计问题。但是如果处理一些需要进行修改的动态统计问

题，困难就出现了。这时就需要改动树节点的内容来记录一些必要的信息。

线段树的推广：

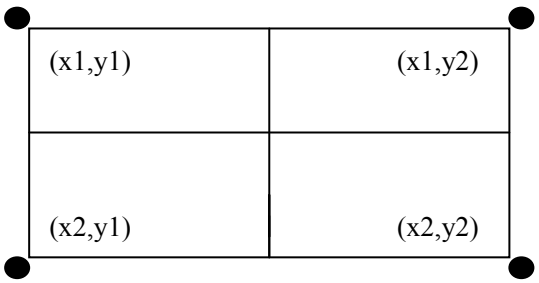
线段树处理的是线性统计问题 ，而我们往往会遇到一些平面统计问题和空间统计问题，因此我们需要推广线段树 ，使它变成 二维线段树和多维线段树。

将一维线段树改成二维线段树，有两种方法。一种就是给原来线段树中的每个结点都加多一棵线段树，即“树中有树”。如下图：

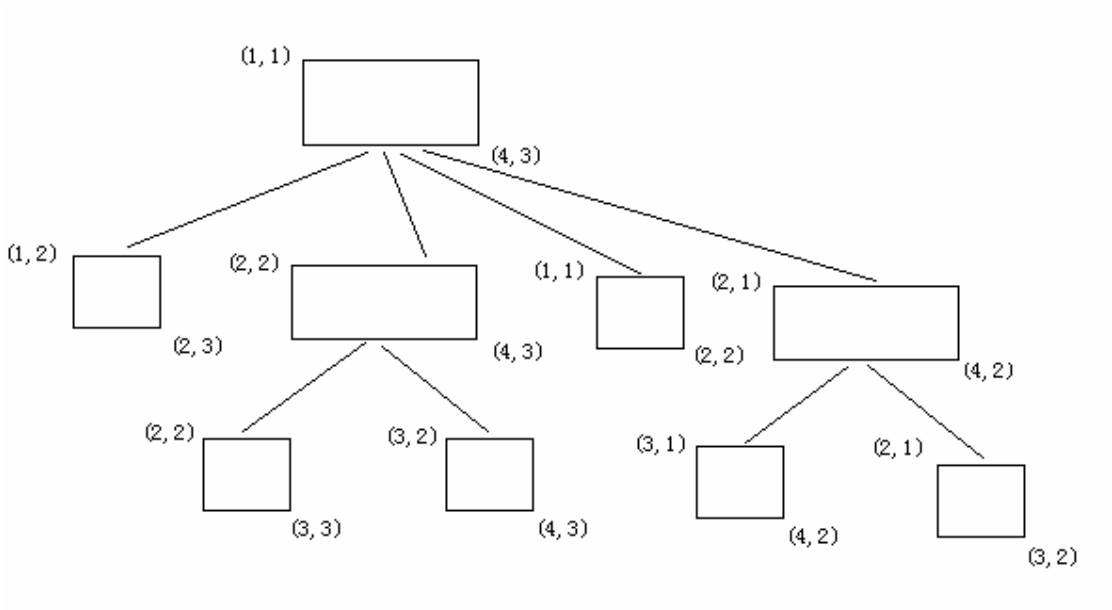


容易算出，用这种方法构造一棵矩形(x1,y1,x2,y2)的线段树的空间复杂度为O(NumOfRow*NumOfCol)。其中 NumOfRow，NumOfCol，分别表示矩形的长和宽。相应地，一次操作的时间复杂度为 O(lg(NumOfRow) * lg(NumOfCol))。由于这种线段树有两层，处理起来较麻烦。

另一种方法是将线段树结点中的线段变成矩形，从而变为矩形树。因此矩形树用的是四分的思想，每个矩形分割为 4 个子矩形。矩形(x1,y1,x2,y2)的 4 个儿子如下图所示：



这是一棵以矩形(1,1,4,3)为根的矩形树：



以(x1,y1,x2,y2)为根的矩形树的空间复杂度也为 O(NumOfRow*NumOfCol)。由于它只有一层，处理起来比第一种方法方便。而且在这种矩形树中，标记思想依然适用。而第一种方法中，标号思想在主线段树上并不适用，只能在第二层线段树上使用。但是这种方法的一次操作的时间复杂度可能会达到 O(NumOfRow)。比起第一种来就差了不少。

对于多维的问题，第一种方法几乎不可能使用。因此我们可以仿照第二种方法。例如对于 n 维的问题。我们构造以(a1,a2,a3,….,an,b1,b2,b3,….,bn)为根的线段树，其中(a1,a2,a3….,an)表示的是左上角的坐标，(b1,b2,b3,….,bn)表示的是右下角的坐标。用的是 2^n 分的思想，构造出一棵 2^n 叉树。结点的个数变为 2^n×(b1-a1)×(b2-a2)×……×(bn-an)。

2．树状数组

利用树状数组，编程的复杂度提高了，但程序的时间效率也大幅地提高。这正是利用了树结构能够减少搜索范围，将信息集中起来的优点，让更新数组和求和运算牵连尽量少的变量。

通过一个小题目来介绍一下树状数组：给出一个一维整数序列，对这个序列有两种操作：使第 k 个元素的值增加 x(k 和 x 由输入数据给定)；求解前 m 个元素的和（m 由输入数据给定）。

先将问题简化，考察一维子序列求和的算法。设该序列为 a[1]，a[2]……a[n]。

算法 1：直接在原序列中计算。显然更新元素值的时间复杂度为 O(1)；在最坏情况下，子序列求和的时间复杂度为 O(n)。

算法 2：增加数组 b，其中 b[i]＝a[1]+a[2]+……+a[i]。由于 a[i]的更改影响 b[i] ……b[n]，因此在最坏情况下更新元素值的算法复杂度为 O(n)；而子序列求和的算法复杂度仅为 O(1)。

以上两种算法，要么在更新元素值上耗费时间过长（算法 1），要么在子序列求和上无法避免大量运算（算法 2）。有没有更好的方法呢？

算法三：增加数组 C，其中 C[i]=a[i-2^k+1]+……+a[i](k 为 i 在二进制形式下末尾 0 的个数)。由 c 数组的定义可以得出：

c[1]=a[1]

c[2]=a[1]+a[2]=c[1]+a[2]

c[3]=a[3]

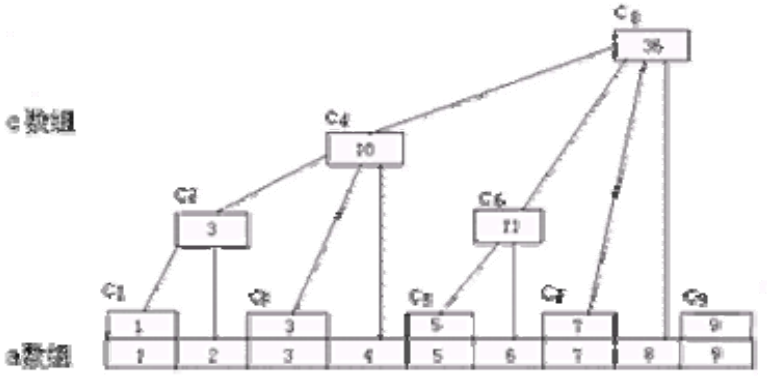
c[4]=a[1]+a[2]+a[3]+a[4]=c[2]+c[3]+a[4]

c[5]=a[5]

c[6]=a[5]+a[6]=c[5]+a[6]

……

c 数组的结构对应一棵树，因此称之为树状数组。在统计更新元素值和子序列求和的算法复杂度后，会发现两种操作的时间复杂度均为 O（logN），大大提高了算法效率。



不难想出若 a[k]所牵动的序列为 C[p_1]，C[p_2]……C[p_m]。则 p_1=k，而 p_{i+1} = p_i +2^li（li 为 p_i 在二进制中末尾 0 的个数）。由此得出更改元素值的方法：若将 x 添加到 a[k]，则 c 数组中 c[p_1]、c[p_2]、…、c[p_m]（ p_m ≤n< p_{m+1} ）受其影响，亦应该添加 x。

例如 a[1]……a[9]中，a[3] 添加 x；

p_1 =k=3　　p_2 =3+20=4

p_3 =4+22=8　　p_4 =8+23=16>9

由此得出，c[3]、c[4]、c[8]亦应该添加 x。

子序列求和可以转化为求由 a[1]开始的序列 a[1]……a[k]的和 S。而在树状数组中求 S 十分简单：根据 c[k]=a[k-2^l+1]+ … +a[k] (l 为 k 在二进制数中末尾 0 的个数)我们从 k1=k 出发，按照 ki+1=ki-2^lki（lki 为 ki 在二进制数中末尾 0 的个数）。于是递推 k2，k3，…，km (km+1=0)。由此得出：S=c[k1]+c[k2]+c[k3] + … + c[km]。

例如，计算 a[1]+a[2]+a[3]+a[4]+a[5]+a[6]+a[7]：

k1=7

k2= k1-2^l1=7-2^0=6

k3= k2-2^l2=6-2^1=4

k4= k3-2^l3=4-2^2=0

即 a[1]+a[2]+ a[3]+a[4]+ a[5]+a[6]+ a[7]=c[7]+c[6]+c[4]

对于我们提出的一维问题，用前面介绍的方法就可以轻易地解决了。注意我们所需要消耗的内存仅是一个很单一的数组，它的构造比起线段树来说要简单得多（很明显，这个

问题也可以用前面的线段树结构来解决）。

可以把这个一维的问题推广到二维。如何推广呢？模仿一维问题的解法，可以将 C[x,y] 定义为：

$$C[x,y]=\sum a[i,j]\text{其中}x-2\text{^}sx+1\le i\le x,y-2\text{^}sy+1\le i\le y$$

其中 sx 为 x 在二进制形式下末尾 0 的个数；sy 为 y 在二进制形式下末尾 0 的个数。

其具体的修改和求和过程实际上是一维过程的嵌套。这里省略其具体描述。可以发现，经过这次推广，算法的复杂度为 $\log^2 n$。而就空间而言，我们仅将一维数组简单地变为二维数组，推广的耗费是比较低的。

可以尝试类似地建立二维线段树来解决这个问题，它的复杂性要比树状数组的方法高得多。

附录 课程实验

《程序设计方法与艺术》是为培养 ICPC 竞赛选手而开的一门综合基础课，主要开课对象是学有余力的学生通过本课程的学习，学生应该在课堂学习和上机实践的基础上，进一步掌握和巩固使用程序设计知识，增强数学建模能力。最终学会融合多项技术进行 ICPC 竞赛程序开发，提高学生的创新能力和团队协作能力。

ICPC 竞赛涉及的内容多、知识面宽，主要有离散数学、组合数学、计算几何、 数论初步等数学知识，还有数据结构、算法设计、动态规划、网络流等计算机专业知识。本课程通过有选择地搜索算法、计算几何初步、动态规划以及相关专题已达到以点盖面，引导学生去学习更多的知识，达到自主学习的目的。从而提高学生算法设计与分析的素质和能力。

鉴于本课程的特点以及大多数学生在学习过程中所出现的问题，本实验指导中不是简单地安排学生编写一些算法上机通过，而是设置了兼顾不同能力层次、具有多种形式及规模的实验任务。本实践环节设置了7次实验，每次实验按排四至六小时，前6次实验分别侧重于教科书中的某一方面，最后一次实验侧重于课程知识的综合应用，要求学生对给定的问题能利用所学知识进行分析、设计、调试，并进行总结。具体实验安排可根据实验机时作适当的调整及增、删，实验内容也可根据学员的基础作适当选择。

实验一 STL 的熟悉与使用

1. 实验目的

（1） 掌握 C++中 STL 的容器类的使用。

（2） 掌握 C++中 STL 的算法类的使用。

2. 试验设备

硬件环境：PC 计算机

软件环境：

操作系统：Windows 2000 / Windows XP / Linux

语言环境：Dev cpp / gnu c++

3. 试验内容

(1) 练习 vector 和 list 的使用。

定义一个空的 vector，元素类型为 int，生成 10 个随机数插入到 vector 中，用迭代器遍历 vector 并输出其中的元素值。在 vector 头部插入一个随机数，用迭代器遍历 vector 并输出其中的元素值。用泛型算法 find 查找某个随机数，如果找到便输出，否则将此数插入 vector 尾部。用泛型算法 sort 将 vector 排序，用迭代器遍历 vector 并输出其中的元素值。删除 vector 尾部的元素，用迭代器遍历 vector 并输出其中的元素值。将 vector 清空。

定义一个 list，并重复上述实验，并注意观察结果。

(2) 练习泛型算法的使用。

定义一个 vector，元素类型为 int，插入 10 个随机数，使用 sort 按升序排序，输出每个元素的值，再按降叙排序，输出每个元素的值。练习用 find 查找元素。用 min 和 max 找出容器中的最小元素个最大元素，并输出。

实验二 搜索算法的实现

1. 实验目的

(1) 掌握宽度优先搜索算法。

(2) 掌握深度优先搜索算法。

2. 试验设备

硬件环境：PC 计算机

软件环境：

操作系统：Windows 2000 / Windows XP / Linux

语言环境：Dev cpp / gnu c++

3. 试验内容

(1) 将书上的走迷宫代码上机运行并检验结果，并注意体会搜索的思想。

(2) 八皇后问题：

在一个国际象棋棋盘上放八个皇后，使得任何两个皇后之间不相互攻击，求出所有的布棋方法。上机运行并检验结果。

思考：将此题推广到 N 皇后的情况，检验在 N 比较大的情况下，比方说 N=16 的时候，你的程序能否快速的求出结果，如果不能，思考有什么方法能够优化算法。

(3) 骑士游历问题：

在国际棋盘上使一个骑士遍历所有的格子一遍且仅一遍，对于任意给定的顶点，输出一条符合上述要求的路径。

(4) 倒水问题：

给定 2 个没有刻度容器，对于任意给定的容积，求出如何只用两个瓶装出 L 升的水，如果可以，输出步骤，如果不可以，请输出 No Solution。

实验三 计算几何算法的实现

1. 实验目的

(1) 理解线段的性质、叉积和有向面积。

(2) 掌握寻找凸包的算法。

(3) 综合运用计算几何和搜索中的知识求解有关问题。

2. 试验设备

硬件环境：PC 计算机

软件环境：

操作系统：Windows 2000 / Windows XP / Linux

语言环境：Dev cpp / gnu c++

3. 试验内容

(1) 将讲义第三章第三节中的凸包代码上机运行并检验结果。

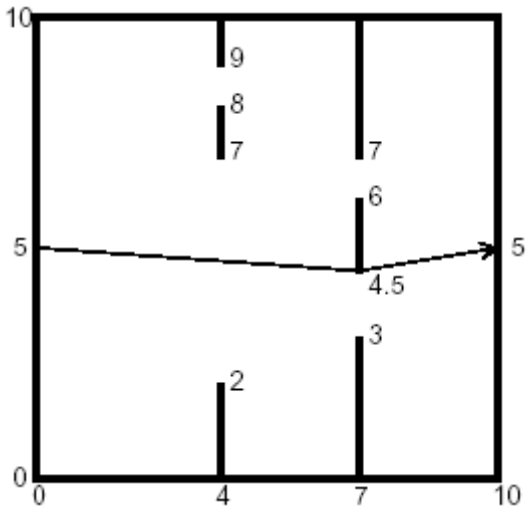
(2) 完成讲义第三章的课后习题，上机运行并检验结果。

(3) 思考：

判线段相交时，如果有个线段的端点在另一条线段上，注意可能与另一条线段上的端点重合，思考这样的情况怎么办。

(4) 房间最短路问题：

给顶一个内含阻碍墙的房间，求解出一条从起点到终点的最最短路径。房间的边界固定在 x=0,x=10,y=0 和 y=10。起点和重点固定在(0,5)和(10,5)。房间里还有 0 到 18 个墙，每个墙有两个门。输入给定的墙的个数，每个墙的 x 位置和两个门的 y 坐标区间，输出最短路的长度。下图是个例子：



实验四 动态规划算法的实现

1. 实验目的

(1) 理解动态规划的基本思想、动态规划算法的基本步骤。

(2) 掌握动态规划算法实际步骤。

2. 试验设备

硬件环境：PC 计算机

软件环境：

操作系统：Windows 2000 / Windows XP / Linux

语言环境：Dev cpp / gnu c++

3. 试验内容

(1) 求两个字符串的最长公共子序列。

X 的一个子序列是相应于 X 下标序列{1, 2, …, m}的一个子序列，求解两个序列的所有子序列中长度最大的，例如输入：pear, peach 输出：pea。

(2) 给定两个字符串 a 和 b，现将串 a 通过变换变为串 b，可用的操作为，删除串 a 中的一个字符；在串 a 的某个位置插入一个元素；将串 a 中的某个字母换为另一个字母。对于任意的串 a 和串 b，输出最少多少次能够将串变为串 b。

思考：输出变换的步骤。

(3) 输入一个矩阵，计算所有的子矩阵中和的最大值。

例如，输入

0 -2 -7 0

9 2 -6 2

-4 1 -4 1

-1 8 0 -2

输出为：15

思考：当矩阵很大时，比如 100*100 的矩阵，你的程序还能够很快的得出结果吗，如果不能，请思考如何用动态规划的思想解决。

实验五 模拟/密码类问题的建模与实现

1. 实验目的

(1) 理解模拟类问题的建模与算法实现。

(2) 掌握密码类问题的建模与算法实现。

2. 试验设备

硬件环境：PC 计算机

软件环境：

操作系统：Windows 2000 / Windows XP / Linux

语言环境：Dev cpp / gnu c++

3. 试验内容

(1) 模拟类问题

a. 完成书上的 Kitty Fishing。

b. 完成书上的 The Same Game。

c. 给定一个格子如下图所示，机器人每次的移动方向如格子中的字母所示，请根据所输入的起始位置，求出机器人的运动情况。

Grid 1

Grid 2

(2) 密码类问题

a. 将书上的代码上机运行，认真观察结果。

b. 将书上的题目自己重做一遍。

c. 给出一种加密算法如下：输入原文，将原文看做一个循环队列生成 n 个字符串，对这 n 个串按首字母稳定排序，取每个串的首字母做为密文。

例如：输入 example，得到 7 个字符串如下：

example,

xamplee,

ampleex,

mpleexa,

pleexam,

leexamp,

eexampl,按尾字母稳定排序后得到：

ampleex,

example,

eexampl,

leexamp,

mpleexa,

pleexam,

xamplee,取每个串的尾字母，得到密文 xelpame，现在要求输入一个原文对其加密，或输入一个密文对其解密。

实验六 字符串/组合数学类问题的建模与实现

1. 实验目的

(1) 理解字符串处理问题的建模与算法实现。

(2) 掌握组合数学类问题的建模与算法实现。

2. 试验设备

硬件环境：PC 计算机

软件环境：

操作系统：Windows 2000 / Windows XP / Linux

语言环境：Dev cpp / gnu c++

3. 试验内容

(1) 字符串处理

a. 将书上的代码上机运行。

b. 求 100 个字符串的最长公共子序列。

c. 自学 KMP 算法，并上机练习。

(2) 组合数学部分

a. 买票问题：

电影院卖电影票，但是没有零钱找，票价一张 5 元，买票的人为 n 个持有 5 元，m 个持有 10 元，求解出可能的买票序列的个数，使得电影院能够将票卖完。例如，n=3,m=3 时，输出 180。 当 n=5,m=3 时，输出 20160。 当 n=100,m=100 时，输出 78084937362850544906174575636850006167835245315561709237103221113302915837960727021812305347721249891502694271180162260421548791113132386639794715341864349615194342304035972003702672172665588675391253885173666662560805072022603450819556855683918208241615966079760353332695646723185180600232841669187740487348791051851871027200000000000000000000000000000000000000000000000000000000000。

b. 给定一个数字的集合{0, 1, ... , k}, 0 <= k <= 9，定义一个 tight word iff 相临的两个数字之间的差最多是 1，求解长度为 n 的 tight word 在所有的 word 中的比率。比如 k=2,n=5 时，输出 40.74074，k=8,n=7 时，输出 0.10130。

c. 巧克力问题：

有袋子里均匀地装着 c 种颜色的巧克力，每种巧克力均有无限多。每次从袋子里拿一块放在桌子上，如果桌子上已经有一块颜色相同的巧克力，就把两块巧克力都吃掉。一共取出了 n 块巧克力，问：最后桌子上有 m 块的概率为多大。例如 c=5,n=100,m=2 时，概率为 0.625。

实验七 ICPC 设计实验

1. 实验目的

综合运用所学知识，对给定问题建模并求解。

2. 试验设备

硬件环境：PC 计算机

软件环境：

操作系统：Windows 2000 / Windows XP / Linux

语言环境：Dev cpp / gnu c++

3. 试验内容

(1) 字符串调整

考虑两个字符串 X = x1x2 ... xm 和 Y = y1y2 ... yn，且字符属于集合{A, G,C, T}。现在要在 X 和 Y 中插入空格，得到字符串 X*和 Y*，要求：

a) X*和 Y*有同样的长度；

b) 如果忽略空格，X*=X，Y*=Y。

举个例子：X='GATCCGA',Y='GAAAGCAGA',插入空格后(空格用-表示)，可以得到的一种结果是

X*=G-A--TCCGA
Y*=GAAAG-CAGA

当然也可以是下面的结果：

X*=GA---TCCGA
Y*=GAAAG-CAGA.

怎么评定它们的优劣呢？我们又下面的法则：如果 xi 和 yj 对齐，得分是

$$\sigma(x_i, y_j) = \begin{cases} 2 & \text{if } x_i = y_j \\ -1 & \text{if } x_i \neq y_j \end{cases}$$

如果一个 X*(或 Y*)的连续的子串和 Y*(或 X*)的一个长度为 k 的空格子串对齐,得分是 -(4 + k).

所以，第一种结果的得分是 2 - (4 + 1) + 2 - (4 + 2) - (4 + 1) + 2 - 1 + 2 + 2 = -7；第二种结果的得分是 2 + 2 - (4 + 3) - (4 + 1) + 2 - 1 + 2 + 2 = -3。

现在你的问题是对于给定的 X 和 Y 的，找出得分最高的 X*和 Y*。对于上面的例子，最好的结果是：

X*=GA--TCCGA
Y*=GAAAGCAGA.

得分为 2+2-(4+2)-1+2-1+2+2=2。在这题中，m 和 n 最大为 500，并且 X*和 Y*中没有任何两个空格是对齐的。

输入格式：

第一行为一个整数，给出了测试的数目，接下来每两行给出一对 X 和 Y。

输出格式：

对于每一组输入，输出得分的最大值。

输入样例：

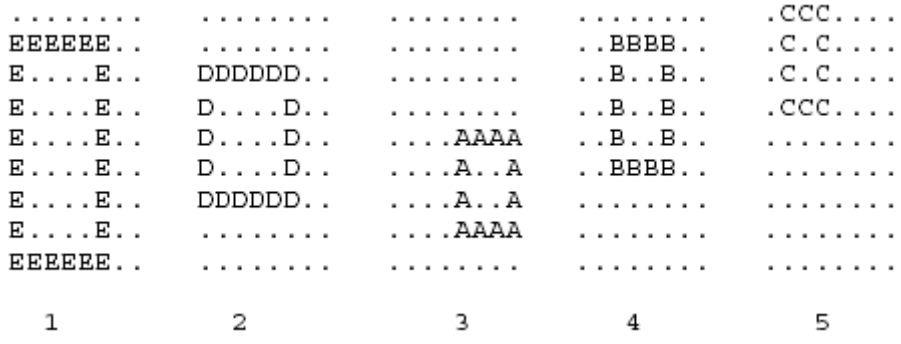
```
3
ACGGCTTAGATCCGAGAGTTAGTAGTCCTAAGCTTGCA
AGCTTAGAAAGCAGACACTTGATCCTGACGGCTTGAA
TTGAGTAGTGTTTTAGTCCTACACGACACATCAAATTCGGACAAGGCCTAGCT
TTCAAGTCCTACAATGTGTGTCAAATTCGCTTGGCCGAAAGCC
TTTGGGAACGTGTGTAGACTTCCCCATGCGATGG
AACACACACGGACTTCATGCTGG
```

输出样例：

```
18
20
2
```

(2) 框架覆盖

考虑下面的五个框架：



现在把它们安顺序一个个地叠起来，上面的框架就会覆盖掉下面框架的一部分。假设我们就得到了下面的图：

```
.CCC....
ECBCBB..
DCBCDB..
DCCC.B..
D.B.ABAA
D.BBBB.A
DDDDAD.A
E...AAAA
EEEEEE..
```

它们是按照什么顺序从下往上搭建起来的呢？答案是 EDABC。你的任务就是按照搭建后的图，判断它们的顺序。为了便于处理，有以下限制：

c) 框架的任何一边不小于 3；
d) 每个框架都没有任何一条边被完全覆盖，框架的角同时属于两条边；
e) 框架用大写字母表示且互不相同。

输入格式：

每个测试的开头是两个整数 h 和 w，表示了宽和高。接下来的 h*w 个字符描述了这个图。

输出格式：

按照从底向上输出框架搭建的顺序，如果有多种可能，按照字典序输出它们。

输入样例：

```
9
8
.CCC....
ECBCBB..
DCBCDB..
DCCC.B..
D.B.ABAA
D.BBBB.A
DDDDAD.A
E...AAAA
EEEEEE..
```

输出样例：

```
EDABC
```

(3) 猴子和香蕉

在一个动物园里，科学家们设计了一个实验来测试猴子的 IQ：把香蕉挂在很高的地方，猴子可以利用一些箱子搭成一个塔，每个箱子有三个参数(x_i, y_i 和 z_i.)，表示箱子的三维。由于箱子可以翻转，箱子 i 可以放在箱子 j 上面的充要条件是对于它们的接触面，箱子 i 的长和宽都必须严格小于箱子 j 的长和宽。

现在你的任务是计算对于给定的箱子种类集合，所能搭到的最大高度是多少，每种箱子都有无限多个。

输入格式：

输入数据包含多个测试，每个测试的第一行给出了箱子的种数 n(n<=30)，n=0 时结束，接下来的 n 行给出了每种箱子的三维。

输出格式：

对于每一个测试，输出可以搭成塔的最大高度。

输入样例：

```
1
10 20 30
2
6 8 10
5 5 5
7
1 1 1
2 2 2
3 3 3
4 4 4
5 5 5
6 6 6
7 7 7
5
31 41 59
26 53 58
97 93 23
84 62 64
33 83 27
0
```

输出样例：

```
Case 1: maximum height = 40
Case 2: maximum height = 21
Case 3: maximum height = 28
Case 4: maximum height = 342
```