ACM/ICPC 算法训练教程

2010 年 10 月

序

ACM 国际大学生程序设计竞赛(ACM/ICPC)是由国际计算机界历史悠久、颇具权威性的组织 ACM 学会主办，是世界上公认的规模最大、水平最高的国际大学生程序设计竞赛，其目的旨在使大学生运用计算机来充分展示自己分析问题和解决问题的能力。因历届竞赛都荟萃了世界各大洲的精英，云集了计算机界的“希望之星”，而受到国际各知名大学的重视，并受到全世界各著名计算机公司的高度关注，成为世界各国大学生最具影响力的国际级计算机类的赛事。

南京理工大学参与该项赛事八年，获得亚洲区银奖 5 个，铜奖 12 个。同时在训练中也积累了一些训练的资料。南京理工大学 ACM/ICPC 集训队根据多年训练积累，整理的《ACM/ICPC 算法训练教程》适合 ACM/ICPC 初学者，及具有一定基础的计算机算法和编程爱好者。适合作为本科及研究生算法与数据结构类课程的参考教材。

本书资料全部来自南京理工大学 ACM/ICPC 集训队内部资料。由张珂、张俊华等集训队员负责整理。因成书仓促，错误在所难免，望批评指正。

余立功

目录

第一章 算法基础

对于计算机科学来说，算法（Algorithm）的概念是至关重要的。通俗地讲，算法是指解决问题的一种方法或一个过程。严格地讲，算法是由若干条指令组成的有穷序列。计算机解题的核心是算法没汁。一个算法应该具有以下四个重要特征：

（1）输入：有零个或多个外部提供的量作为算法的输入，以刻划运算对象的初始情况。所谓 0 个输入是指算法本身定出了初始条件；

（2）输出：算法产生一个或多个输出，以反映对输人数据加工后的结果。没有输出的算法是毫无意义的；

（3）确切性：组成算法的每条指令是清晰的，无歧义的；

（4）有穷性：算法中的每条指令的执行次数和执行时间都是有限的。

程序是算法用某种程序语言的具体实现。对于一个大的软件系统而言，有时候，一个好的算法将决定整个系统的成败。因此我们说，算法是一个程序的灵魂，这一点也不夸张。

对于读者来说，为了设计出高效可行的程序，就必须为了获得一个既有效又优美的算法。但俗话说，万丈高楼平地起，一个好的算法设计是建立在对基础算法的了解基础上的。

下面，我们对一些基本的常用的算法设计思路展开讨论。这里笔者旨在算法思想的介绍，因此，也尽量选用比较浅显易懂并能说明问题的例题进行说明。

1.1 穷举（枚举）法

枚举法，俗称爆搜，指的是从可能的解的集合中一一枚举各元素，用题目给定的检验条件判定哪些是无用的，哪些是有用的。能使命题成立，即为其解。

枚举法本质上属于搜索算法。但是它与隐式图的搜索(第六章)有所不同，因为适用于枚举法求解的问题必须满足两个条件：

1.可预先确定解的个数 n；

2.解变量 A1，A2，…An 的值的可能变化范围预先确定。

例如某问题的解变量有 3 个：Al, A2, A3。其中

A1 解变量值的可能范围 A1∈{X11，X12，X13}

A2 解变量值的可能范围 A2∈{X21，X22，X23}

A3 解变量值的可能范围 A3∈{X31，X32，X33}

则问题的可能解有 27 个：

(A1，A2，A3)∈{(X11，X21，X31)，(x11，X21，X32)，(Xll，X21，X33)
(X11，X22，X31)，(x11，X22，X32)，(Xll，X22，X33)
……
(X13，X23，X31)，(x13，X23，X32)，(Xl3，X23，X33)}

上述可能解的集合中，满足题目给定的检验条件的解元素，即为问题的解。

一般来说，如果预先对问题确定了解的个数以及各个解的值域，我们则可以利用 for 语句和条件判断语句逐步求解或证明：

```
for A1:＝X11 to X11 do
  ……
for Ai:＝Xi1 to Xiq do
```

for An:＝Xn1 to Xnk do
if(A1,…, Ai,…, An)满足检验条件
then 输出问题的解(A1, …, Ai, …, An);

枚举法的特点是算法简单，对于可确定解的值域又一时找不到其它好的算法时就可用它。但枚举法又是一种比较笨拙、原始的方法，运算量大是它的弱点。算法的复杂度是指数级的。对于枚举法也不要过于悲观，从全局观点使用枚举法，计算量容易过大。但是，如果能够排除那些明显不属于解集的元素，在局部地方使用枚举法，其效果会十分显著。

zoj 1403 Safecracker

题意简述：密码序列由一系列大写字母组成，在解密序列不唯一的情况下，输出字典排序下的第一个。

解密公式：$v - w^2 + x^3 - y^4 + z^5 = target$

其中，target，数字，由题目给出。v,m,x,y,z 属于同一个集合且各不相同，该集合由题目给出，由 26 个大写字母中的任意 5 到 12 个组成。每个字母对应的数值，依次类推（A=1，B=2，……Z=26）。

思路：这是一道典型的枚举题。题目中，解的值域已经确定，解元素中的 v，m，x，y，z 都是题目给定集合中的一个元素。尽管枚举法的算法复杂度是指数级的，但鉴于给定集合最多只有 12 个元素，完全可以直接用枚举。另外，题目中的约束条件，v，m，x，y，z 各不相同，明显不属于解集的元素可以很容易的通过判断来排除。

需要注意的是，题目求的密码序列是字典排序下的第一个，所以需要事先将集合元素排好序，这里的排序算法也不用太复杂，一个简单的冒泡即可。

代码：

```
#include <iostream>
#include <math.h>
#include <string>
using namespace std;
int num;
char letter[13];
int value[12];
void process(int m)      //主要计算过程，枚举（爆搜）吧！
{
    int a,b,c,d,e;
    for(a=0;a<m;a++)
            for(b=0;b<m;b++)
                if(a!=b)                //排除明显不属于解集的元素，下同。
                for(c=0;c<m;c++)
                    if(a!=c && b!=c)
                    for(d=0;d<m;d++)
                        if(a!=d && b!=d && c!=d)
                        for(e=0;e<m;e++)
                            if(a!=e && b!=e && c!=e && d!=e)
    if(value[a]-pow(value[b],2.0)+pow(value[c],3.0)-
```

```
    pow(value[d],4.0)+pow(value[e],5.0)==num)     //找到解了，Mission completed!
                                        {
                                            cout<<char(value[a]+'A'-1)<<char(value[b]+'A'-1)
                                                <<char(value[c]+'A'-1)<<char(value[d]+'A'-1)
                                                <<char(value[e]+'A'-1)<<endl;
                                            return;
                                        }
        cout<<"no solution"<<endl;
}
int main()
{
    int a,b,c,d,e,max,m=0;
    while(cin>>num>>letter && num && strcmp(letter,"END"))
    {
        m=0;
        while(letter[m])
        {
            value[m]=letter[m]-'A'+1;
            m++;
        }
        int temp;
        for(a=0;a<m;a++)                    //简单的冒泡!!
            for(b=a+1;b<m;b++)
                if(value[a]<value[b])
                {
                    temp=value[a];
                    value[a]=value[b];
                    value[b]=temp;
                }
        process(m);
    }
    return 0;
}
```

1.2 递归法

说起递归，首先大家联想到的应该就是那个经典的 Hanoi 塔问题了。虽然这个问题也有非递归算法，但多年来，它已经是递归算法的典例了。好吧，让我们也以 Hanoi 塔开始“递归之旅”吧！

Hanoi 塔问题

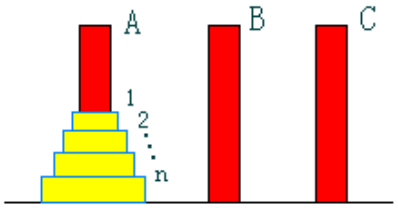
题意简述：设 a,b,c 是 3 个塔座。开始时，在塔座 a 上有一叠共 n 个圆盘，这些圆盘自下而上，由大到小地叠在一起。各圆盘从小到大编号为 1,2,…,n,现要求将塔座 a 上的这一叠圆盘移到塔座 b 上，并仍按同样顺序叠置。在移动圆盘时应遵守以下移动

规则：

规则 1：每次只能移动 1 个圆盘；

规则 2：任何时刻都不允许将较大的圆盘压在较小的圆盘之上；

规则 3：在满足移动规则 1 和 2 的前提下，可将圆盘移至 a,b,c 中任一塔座上。



思路：hanoi(n, A, B, C)表示将塔座 A 上自上而下，由大到小叠在一起的 n 个圆盘依移动规则移至塔座 B 上并仍按同样顺序叠排。在移动过程中，以塔座 C 作为辅助塔座。

move(n, a, b)表示将塔座 A 上编号为 n 的圆盘移至塔座 B 上。

代码：

```
public static void hanoi(int n, int a, int b, int c){
    if (n > 0){
        hanoi(n-1, a, c, b);
        move(n,a,b);
        hanoi(n-1, c, b, a);
    }
}
```

至此，我们可以给出递归的定义：

设一个未知函数 f，用其自身构成的已知函数 g 来定义 $f(n)=\begin{cases} g(n,f(n-1)), n>0 \\ a, n=0 \end{cases}$，

为了定义 f(n)，必须先定义 f(n 一 1)，为了定义 f(n 一 1)，又必须先定义 f(n 一 2)……上述这种用自身的简单情况来定义自己的方式称为递归定义。

一个递归定义必须是有确切含义的，也就是说，必须一步比一步简单，最后是有终结的，决不能无限循环下去。在 f(n)的定义中，当 n 为 0 时定义一个已知数 a，是最简单的情况，称为递归边界，它本身不再使用递归的定义。

与递推一样，每一个递归定义都有其边界条件。但不同的足，递推是内边界条件出发。通过递归式求 f(n)的值，从边界到求解的全过程十分清楚；而递归则是从函数自身出发来达到边界条件。在通往边界条件的递归调用过程中，系统用堆栈把每次调用的中间结果(局部变量和返回地址值)保存起来，直至求出递归边界值 f(0)＝a。然后返回调用函数。返回过程中，中间结果相继出栈恢复，f(1)＝g(1，a)→f(2)＝g(2，f(I))→…→直至求出 f(n)＝g(n，f(n—1))。

由上面递归的定义可见，递归算法的效率往往很低，费时和费内存空间。但是递归也有其长处，它能使一个蕴含递归关系且结构复杂的程序简洁精炼，增加可读性。特别是在难于找到从边界到解的全过程的情况下，如果把问题推进一步，其结果仍维持原问题的关系，则采用递归算法编程比较合适。

递归按其调用方式分为直接递归和间接递归。所谓直接递归是指递归过程 P 直接自己调用自己；而间接递归即 P 包含另一个过程 D，D 又调用 P。

递归算法适用的一般场合为：

1．数据的定义形式按递归定义

如 Fibonacci 数列：1，1，2，3，5，8，13，21，34，55，……

该数列递归定义如下：

$$F(n)=\begin{cases}1 & n=0\\ 1 & n=1\\ F(n-1)+F(n-2) & n>1\end{cases}$$

第 n 个 Fibonacci 数可递归地计算如下：

```
int fibonacci(int n)
{
    if (n <= 1) return 1;
    return fibonacci(n-1)+fibonacci(n-2);
}
```

这类递归问题可转化为递推算法，递归边界作为递推的边界条件。

上例中的第 n 个 Fibonacci 数的递推实现如下：

```
int fibonacci_1(int n)
{
        F(0) = 1;
        F(1) = 1;
        if(n <= 1) return 1;
        else{
            for(i = 2; i <=n; i++) F(i) = F(i-1) + F(i-2);
            return F(n);
        }
}
```

2．数据之间的关系(即数据结构)按递归定义。如树的遍历、图的搜索等。

3，问题解法按递归算法实现。例如回溯法等。

对于 2，3，可利用堆栈结构将其转换为非递归算法。

下面，让我们再来看看递归算法的两个简单应用。

整数划分问题

题意简述：将正整数 n 表示成一系列正整数之和：n=n1+n2+…+nk，

其中 n1≥n2≥…≥nk≥1，k≥1。

正整数 n 的这种表示称为正整数 n 的划分。求正整数 n 的不同划分个数。

例如正整数 6 有如下 11 种不同的划分：

6；

5+1；

4+2，4+1+1；

3+3，3+2+1，3+1+1+1；

2+2+2，2+2+1+1，2+1+1+1+1；

1+1+1+1+1+1。

思路：前面的几个例子中，问题本身都具有比较明显的递归关系，因而容易用递归函数直接求解。

在本例中，如果设 p(n)为正整数 n 的划分数，则难以找到递归关系，因此考虑增加一个自变量：将最大加数 n1 不大于 m 的划分个数记作 q(n,m)。可以建立 q(n,m)的如下递归关系。

(1) q(n,1)=1 n>=1

(2) q(n,m)=q(n,n); m>=n

(3) q(n,n)=1+q(n,n-1);

正整数 n 的划分由 n1=n 的划分和 n1≤n-1 的划分组成。

(4) q(n,m)=q(n,m-1)+q(n-m,m),n>m>1;

正整数 n 的最大加数 n1 不大于 m 的划分由 n1=m 的划分和 n1≤m-1 的划分组成。

在本例中，如果设 p(n)为正整数 n 的划分数，则难以找到递归关系，因此考虑增加一个自变量：将最大加数 n1 不大于 m 的划分个数记作 q(n,m)。可以建立 q(n,m)的如下递归关系。

$$q(n,m)=\begin{cases}1 & n=1,m=1\\ q(n,n) & n<m\\ 1+q(n,n-1) & n=m\\ q(n,m-1)+q(n-m,m) & n>m>1\end{cases}$$

代码：

```
int q(int n, int m)
{
    if((n < 1) || (m < 1)) return 0;
    if((n == 1) || (m == 1)) return 1;
    if(n < m) return q(n, n);
    if(n == m) return q(n, m-1) + 1;
    return q(n, m-1) + q(n-m, m);
}
```

一旦递归公式确定了，写出程序实现就不是一件难事了。

排列问题

题意简述：已知集合 $R=\{r_1,r_2,\cdots,r_n\}$，请设计一个算法生成集合 R 中 n 个元素的全排列。

思路：令 $R_j=R-\{r_j\}$。

我们记集合 X 中元素的全排列记为 perm(X)。那么 (r_j) perm(X)：表示在全排列 perm(X)的每一种排列前加上前缀 r_j 所得到的排列。所以，R 的全排列可归纳定义如下：

当 n=1 时，perm(R)=(r)，其中 r 是集合 R 中唯一的元素；

当 n>1 时，perm(R)由 (r_1) perm(R_1)，(r_2) perm(R_2)，…，(r_n) perm(R_n)构成。

代码：

```
public class Perm{
   public static void swap(Object []a,int i,int j){
      Object temp=a[i];
      a[i]=a[j];
      a[j]=temp;
   }
   public static void perm(Object []a,int k,int m,int pk,int pm){
      if(k==m){
          for(int i=pk;i<=pm;i++)
              System.out.print(a[i]);
          System.out.println();
      }else{
          for(int i=k;i<=m;i++){
                swap(a,k,i);
                perm(a,k+1,m,pk,pm);
                swap(a,k,i);
          }
      }
   }
   public static void main(String []args){
        Integer []a=new Integer[6];
        for(int i=0;i<a.length;i++){
            a[i]=new Integer(i);
        }
        perm(a,1,3,1,3);
   }
}
```

1.3 分治法

有许多算法在结构上是递归的：为了解决一个给定问题，算法要一次或多次地调用其自身来解决相关的子问题。这些算法通常采用分治策略：将原问题分成 n 个规模较小而结构与原问题相似的子问题。递归地解这些子问题，然后合并其结果就得到原问题的解。n＝2 时的分治法又称二分法。

用分治法求解一个问题，所需的时间是由子问题的个数，大小以及把这个问题分解为子问题所需的工作总量来确定的。一般来说，二分法(即把任意大小的问题尽可能地等分为两个子问题)较为有效。

分治法在每一层递归上都有三个步骤：

分解：将原问题分解成一系列子问题；

解决：递归地解各子问题。若子问题足够小，则直接解；

合并：将子问题的结果合并成原问题的解；

由此，我们可以得出，分治法的基本思想是：将规模为 n 的问题费分解为 k 个规模较小的子问题，这些子问题互相独立且与原来的问题相同。递归地解这些子问题，然后将各个子

问题地解合并得到原问题的解。它的一般的算法设计模式如下：

```
divide_and_conquer(P)
{
    if(|P|<= n0 )     process(P);
    else
    {
        divide P into smaller subinstances P1,P2,…,Pk
        for(int i=1;i<=k;i++)
            yi=divide_and_conquer(Pi);
         merge(y1,y2,…,yk);
    }
}
```

由上面的分析，我们可以看出，采用分治策略设计出的算法通常是递归算法。因而，分治法的效率通常可以用递归方程来分析。一个分治法将规模为 n 的问题分解为 a 个规模为 n/b 的子问题解决。为了方便分析，我们做如下假设：设分解阈值为 1，且解规模为 1 的问题耗费一个时间单位。另外，假设将原问题分解为 a 个子问题以及将 a 个子问题的解合并为原问题的解需要用 $s(n)$ 个单位时间，则所需要的时间可以递归表示为：

$$T(n)=\begin{cases} O(1) & \text{n = 1} \\ aT(n/b)+s(n) & \text{n > 1} \end{cases}$$

下面，我们通过两个例解来看看分治策略的应用。

快速排序

题目简述：输入数组为 $a[p:r]$，通过排序算法使其有序

思路：两步（1）划分，以 $a[p]$ 为基准元素将 $a[p:r]$ 划分成三个部分：$a[p:q-1],a[q],a[q+1:r]$，使得 $a[p:q-1]$ 中的任奕元素都小于等于 $a[q]$，$a[q+1:r]$ 中的任一元素都大于等于 $a[q]$，q 是在划分过程中确定的。（2）通过递归调用，对 $a[p:q-1]$ 和 $a[q+1:r]$ 进行同样的操作。

代码：

```
public static void quickSort(int []a,int p,int r){
if(p<r){
        int q=partition(a,p,r);
        quickSort(a,p,q-1);
        quickSort(a,q+1,r);
}
}
public static int partition(int []a,int p,int r){
    int i=p;
    int j=r+1;
    int x=a[p];
    while(true){
```

```
        while(a[++i]<x); //找到>=x 的
        while(a[--j]>x);//找到<=x 的
        if(i>=j) break;
        swap(a,i,j);
    }
    a[p]=a[j];
    a[j]=x;
    return j;
    }
```

算法分析：

例子：4，1，8，3，5，2，9，10

分析：时间复杂性与划分是否对称有关，最坏情况发生在划分过程中产生的两个区域分别包含和 $n-1$ 和 1 个元素。分割算法耗时为 $O(n)$，那么最外情形下的复杂性为：

$$T(n)=\begin{cases}O(1) & \text{n}=1\\ T(n-1)+O(n) & \text{n}>1\end{cases}$$

也就是为 $O(n^2)$

最好情形下，每次划分产生两个均匀大小的块，则时间复杂性为：

$$T(n)=\begin{cases}O(1) & \text{n}=1\\ 2T(n/2)+O(n) & \text{n}>1\end{cases}$$

也就是为 $O(n\log n)$。

这里我们可以看到，快速排序的性能取决于划分的对称性。因此可以引入随即策略提高其性能。在快速排序的第一步中，当数组还没有被划分时，在 $a[p:r]$ 随即挑一个元素作为划分基准。这样，从统计意义上来讲，得到的划分是均匀的。

```
public static int randomizedPartition(int []a,int p,int r){
    int i=random(p,r);
    swap(a,i,p);
    return partition(a,p,r);
}
public static void randomizedQuickSort(int []a,int p,int r){
    if(p<r){
        int q=randomizedPartition(a,p,r);
        randomizedQuickSort(a.p,q-1);
        randomizedQuickSort(a,q+1,r);
    }
}
```

归并排序(Merge Sort)

题目简述：同样，我们举例一个理牌的例子：一种可行的方案是将一副牌一分为二，一人洗一半。然后合并起来。

思路：将待排序的元素分成大致相同的 2 个子集合，分别对两个子集合进行排序，最终将排

序的子集合合并成排好序的集合。

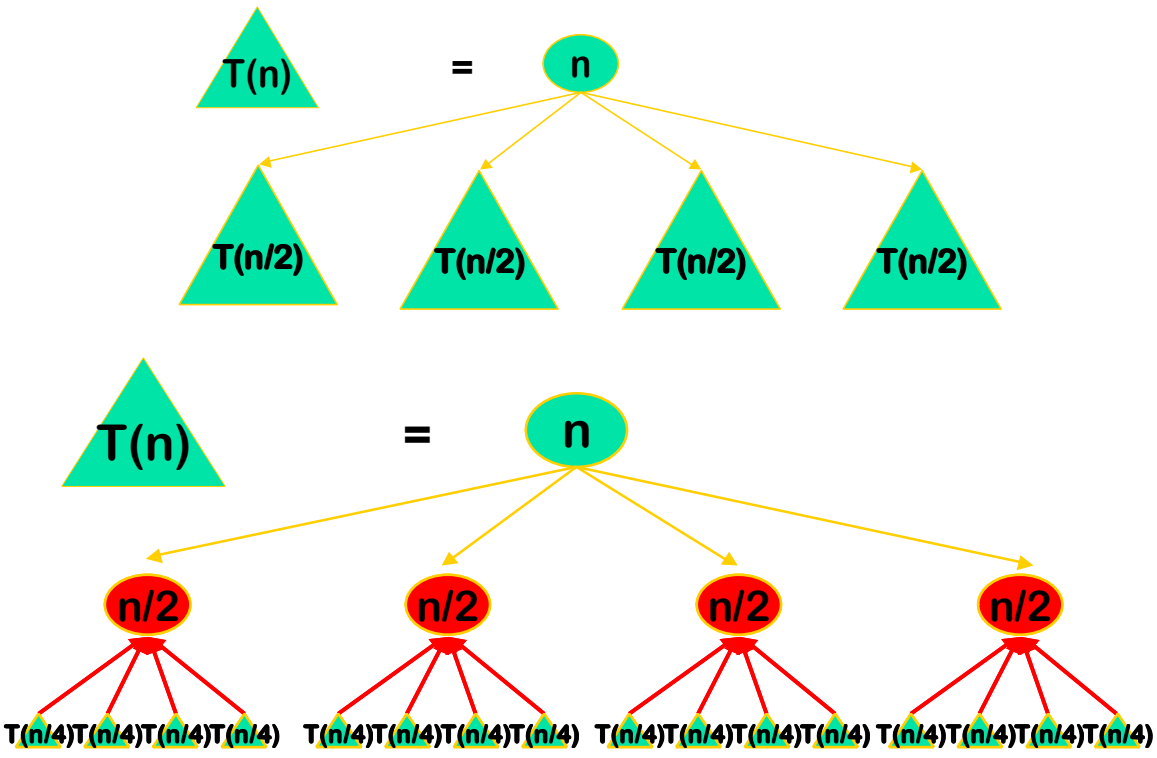
代码：

```
public static void mergeSort(int []a,int left,int right)
{
if(left<right)       //至少有两个元素
{
    int i=(left+right)/2; //取中点
    mergeSort(a,left,i);
    metgeSort(a,i+1,right);
    merge(a,b,left,i,right); //合并到数组 b
    copy(a,b,left,right);//复制回数组 a
}
}
```

至此，我们得出：

分治算法总体思想

将原问题分解为 k 个子问题，对这 k 个子问题分别求解。如果子问题的规模仍然不够小，则再划分为 k 个子问题，如此递归的进行下去，直到问题规模足够小，很容易求出其解为止。将求出的小规模的问题的解合并为一个更大规模的问题的解，自底向上逐步求出原来问题的解。



分治法的适用条件

分治法所能解决的问题一般具有以下几个特征：

(1) 该问题的规模缩小到一定的程度就可以容易地解决；

(2) 该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质

(3) 利用该问题分解出的子问题的解可以合并为该问题的解；

该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题。这条特征涉及到分治法的效率，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，此时虽然也可用分治法，但一般用动态规划较好。

1.4 贪心法

贪心法也是从问题的某一个初始解出发，向给定的目标递推。推进的每一步做一个当时看似最佳的贪心选择，不断地将问题实例归纳为更小的相似的子问题，并期望通过所做的局部最优选择产生出一个全局最优解。

问题是，这种局部贪心的选择是否可以得出全局最优解呢？在某些情况下是可以的。

删数问题

题意简述：键盘输人一个高精度的正整数 N，去掉其中任意 S 个数字后剩下的数字按原左右次序组成一个新的正整数。编程对给定的 N 和 S，寻找一种方案使得剩下的数字组成的新数最小。

输出应包括所去掉的数字的位置和组成的新的正整数。(N 不超过 240 位)

输人数据均不需判错。

思路：

首先我们必须注意，试题中正整数 N 的有效位数为 240 泣，而计算机中整数有效位(包括符号位)交其量也不过 11 位，无论如何也达不到试题的数值要求。 因此，必须采用可含 256 个字符的字串来替代整数。

以字串形式输入 N，使用尽可能逼近目标的贪心法来逐一删去其中 S 个数符，每一步总是选择一个使剩下的数最小的数符删去。之所以作出这样贪心的选择，是因为删 S 个数符的全局最优解．包含了删一个数符的子问题的最优解。

为了保证删 1 个数符后的数最小，我们按高位->低位的方向搜索递减区间。若不存在递减区间，则删尾数符；否则删递减区间的首字符，这样形成了一个新数串。然后回到串首，重复上述规则，删下一数符……依此类推，直至删除 S 个数符为止。

例如：N = '178543', S＝4, 删数过程如下:

N = '1 7 8 5 4 3'

'1 7 5 4 3'

'1 5 4 3'

'1 4 3'

'1 3'

显然，按上述规则删去 s 个数符后，剩余(N-s)个数符组成的新数必定最小。

代码：

```
void Find_Min_Integer
{
  int m=N.size();     //N 是数串
  if(s>=m) {N.erase(); return;}   //s 是被删的数字个数
  while(s>0){
        for(int i=0; (i<N.size()-1) && (N[i]<=N[i+1]); i++)   //搜索递减区间
              N.erase(i, 1);   //删去该区间的首数符
        s--;
  }
```

```
    while(N.size()>1 && N[0]=='0') N.erase(0, 1);   //删去串头的无用零
}
```

读者在看了上述题解后，自然会提出这样一个问题：对一个最优化问题，我们怎样才能知道它是否适用于贪心算法求解？没有一个通用的方法。但适用于贪心策略求解的大多数问题都有两个特点：

1.贪心选择性质——可通过做局部最优(贪心)选择来达到全局最优解

贪心策略通常是自顶向下做的。第一步为一个贪心选择，将原问题变成一个相似的、但规模更小的问题，而后的每一步都是当前看似最佳的选择。这种选择可能依赖于已作出的所有选择，但不依赖有待于做的选择或子问题的解。从求解的全过程来看，每一次贪心选择都将当前问题归纳为更小的相似子问题，而每一个选择都仅做一次，无重复回溯过程，因此贪心法有较高的时间效率，

例如[例 4.1]中，设正整数 N 有 P 位。第一步选择一个使剩下的 p-1 位数最小的数符删去．把问题归纳为在 p-1 位正整数中去掉 s-1 个数字后的新数最小的问题；第二步再选择一个使得剩下的新数最小的数符删去，又把问题归纳为在 p-2 位正整数中去掉 S-2 个数字后的新数最小的子问题……每次选择中，已删去的数不允许再删，而当前选择又不受将来删的数的影响。依照上述方法不断将问题归纳为更小的互为独立的子问题，直至删除 S 个数符为止。显然删数问题符合贪心选择性质。

2．最优子结构——问题的最优解包含了子问题的最优解

例如上例中，N＝'178543'，S＝4。第一步的贪心选择为删'8'，即第一个子问题的最优解为'8'；第二步的贪心选择为删'7'，即第二个子问题(第一个子问题的子问题)的最优解为'7'；继后的第三个，第四个于问题的最优解分别为'5'和'4'。很显然，问题的最优解包含了四个子问题的最优解、因此删数问题具有最优子结构性质。

但问题是，对所有具备最优子结构的问题是否都可采用贪心策略呢?不一定。下面,我们来考察一个经典优化问题的两种变形。

背包问题

题目简述：有一个贼在偷窃一家商店时发现有 N 件物品：第 i 件物质值 Vi 元，重 Wi 磅．(1<=i<=n)，此处 Vi 和 Wi 都是整数。他希望带走的东西越值钱越好，但他的背包中最多只能装下 W 磅的东西(W 为整数)。有两种偷窃方式：

1)0-1 背包问题

如果每件物品或被带走或被留下,小偷应该带走哪几件东西？

2)部分背包问题

如果允许小偷可带走某个物品的一部分，小偷应该带走那几件东西，每件东西的重量是多少？

思路：

两种背包问题都具有最优化结构性质：

对于 0-1 背包问题，考虑重量至多为 W 磅的最值钱的一包东西。如果我们从中去掉物品 J，余下的必须是窃贼从其余的 n 一 1 件物品中可带走的重量至多为 W-Wj 的最值钱的一件东西；如果我们再从中去掉物品 i，余下的也必须是窃贼从其余 n 一 2 件可带走的、重量至多为 W-Wj-Wi 的最值钱的一件东西……，依次类推。

```
template<class Type>
void 0_1_Knapsack (int x[],   Type w[], Type c, int n){
int *t = new int [n+1];
Sort(w, t, n);       //按物品价值排序
for (int i = 1; i <= n; i++) x[i] = 0;
```

```
for (int i = 1; i <= n && w[t[i]] <= c; i++) {x[t[i]] = 1; c -= w[t[i]];}
 }
```

对于部分背包问题，如果我们考虑从最优货物中去掉某物品 J 的重量 Wp(Wj>=Wp)，则余下的物品必须是窃贼可以从 N-1 件原有物品和物品 J 的 Wj-Wp 中可带走的、重量至多为 W-Wp 的最值钱的一件东西……，依次类推。

```
void Knapsack(int n,float M,float v[],float w[],float x[]){
Sort(n,v,w); //按物品单位价值排序
int i;
for (i=1;i<=n;i++) x[i]=0;
float c=M;
for (i=1;i<=n;i++) {
     if (w[i]>c) break;
     x[i]=1;
     c-=w[i];
}
if (i<=n) x[i]=c/w[i];
 }
```

我们可采用贪心策略来解决部分背包问题：

先对每件物品计算其每镑价值 Vi/Wi，然后按每磅价值单调递减的顺序对所有物品排序。例如，总共有三件物品和一个背包。

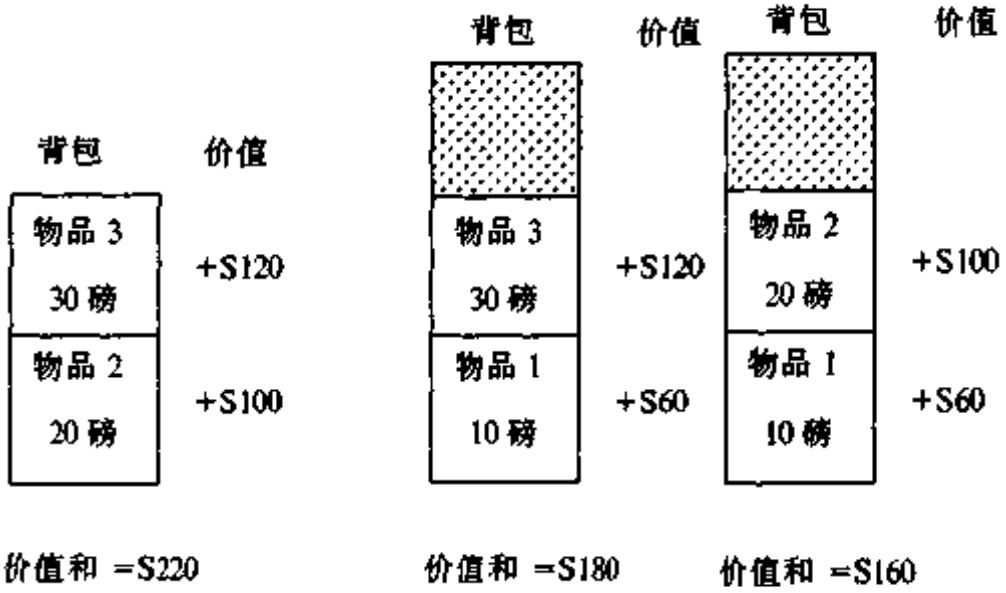
按照一种贪心策略，窃贼开始时对具有最大的每磅价值的物品尽量多拿一些。如果他拿完了该物品后仍可以取一些其它物品时，他就再取具有次大的每磅价值的物品，一直继续下去，直到不能取为止。

由此可见，在部分背包问题中，当我们在考虑是否把一件物品加到背包中时．不需要把加入该物品的子问题解与不取该物品的子问题解加以比较。由这种贪心方式所形成的所有子问题是互相独立的。

若对 0-1 背包问题采取贪心策略的话．顺序取物品 1 和 2。

但是从下图可以看出，最优解取的是物品 2 和 3，没有取物品 1。

两种包含物品 1 的可能解都不是最优解。



在 0—1 背包问题中不应取物品 1 的原因在于这样无法将背包填满，空余的空间降低了它的每磅价值. 因此当我们考虑是否要把一件物品加到背包中时，必须对把加进该物品的子问题的解与不取该物品的子问题的解进行比较。由这种方式形成的子问题导致了许多重迭的子问题。对于这一类问题虽具有最优化结构性质、但产生的子问题互为重迭的试题，一般采用动态程序设计的方法求解，也就是我们常说的动态规划。我们将在第八章介绍这一算法。

下面我们介绍一种能确定贪心法何时产生最优解的理论。虽然它还未覆盖贪心法所适用的各种情况，但它确实能应用于许多重要的场合，并且这个理论发展很快，正在覆盖更多的应用。

借助于拟阵工具，可建立关于贪心算法的较一般的理论。这个理论对确定何时使用贪心算法可以得到问题的整体最优解十分有用。

1、拟阵

拟阵 M 定义为满足下面 3 个条件的有序对(S,I)：

(1)S 是非空有限集。

(2)I 是 S 的一类具有遗传性质的独立子集族，即若 B∈I，则 B 是 S 的独立子集，且 B 的任意子集也都是 S 的独立子集。空集∅必为 I 的成员。

(3)I 满足交换性质,即若 A∈I,B∈I 且|A|<|B|,则存在某一元素 x∈B-A,使得 A∪{x}∈I。

例如，设 S 是一给定矩阵中行向量的集合，I 是 S 的线性独立子集族，则由线性空间理论容易证明(S,I)是一拟阵。拟阵的另一个例子是无向图 G=(V,E)的图拟阵 $M_G=(S_G,I_G)$。

给定拟阵 M=(S,I)，对于 I 中的独立子集 A∈ I，若 S 有一元素 x∉ A，使得将 x 加入 A 后仍保持独立性，即 A∪{x} ∈ I,则称 x 为 A 的可扩展元素。

当拟阵 M 中的独立子集 A 没有可扩展元素时，称 A 为极大独立子集。

下面的关于极大独立子集的性质是很有用的。

定理 4.1：拟阵 M 中所有极大独立子集大小相同。

这个定理可以用反证法证明。

若对拟阵 M=(S,I)中的 S 指定权函数 W，使得对于任意 x∈ S，有 W(x)>0，则称拟阵 M 为带权拟阵。依此权函数，S 的任一子集 A 的权定义为 $W(A)=\sum_{x\in A}W(x)$。

2、关于带权拟阵的贪心算法

许多可以用贪心算法求解的问题可以表示为求带权拟阵的最大权独立子集问题。

给定带权拟阵 M=(S,I)，确定 S 的独立子集 A∈I 使得 W(A)达到最大。这种使 W(A)最大的独立子集 A 称为拟阵 M 的最优子集。由于 S 中任一元素 x 的权 W(x)是正的，因此，最优子集也一定是极大独立子集。

例如，在最小生成树问题可以表示为确定带权拟阵的最优子集问题。求带权拟阵的最优子集 A 的算法可用于解最小生成树问题。

下面给出求带权拟阵最优子集的贪心算法。该算法以具有正权函数 W 的带权拟阵 M=(S,I)作为输入，经计算后输出 M 的最优子集 A。

```
Set greedy (M,W){
    A=∅;
    将 S 中元素依权值 W（大者优先）组成优先队列;
    while (S!=∅) {
        S.removeMax(x);
        if (A∪{x}∈I) A=A∪{x};
    }
```

```
    return A
}
```

引理 4.2(拟阵的贪心选择性质)

设 M=(S,I)是具有权函数 W 的带权拟阵，且 S 中元素依权值从大到小排列。又设 x∈ S 是 S 中第一个使得{x}是独立子集的元素，则存在 S 的最优子集 A 使得 x∈ A。

算法 greedy 在以贪心选择构造最优子集 A 时，首次选入集合 A 中的元素 x 是单元素独立集中具有最大权的元素。此时可能已经舍弃了 S 中部分元素。可以证明这些被舍弃的元素不可能用于构造最优子集。

引理 4.3：

设 M=(S,I)是拟阵。若 S 中元素 x 不是空集Æ 的可扩展元素，则 x 也不可能是 S 中任一独立子集 A 的可扩展元素。

引理 4.4(拟阵的最优子结构性质)

设 x 是求带权拟阵 M＝(S, I)的最优子集的贪心算法 greedy 所选择的 S 中的第一个元素 。那么，原问题可简化为求带权拟阵 M'=(S',I')的最优子集问题，其中：

S'={y|y∈ S 且{x,y} ∈ I}

I'={B|B⊆ S-{x}且 B∪{x} ∈ I}

M'的权函数是 M 的权函数在 S'上的限制(称 M'为 M 关于元素 x 的收缩)。

定理 4.5(带权拟阵贪心算法的正确性)

设 M＝(S,I)是具有权函数 W 的带权拟阵，算法 greedy 返回 M 的最优子集。

3.任务时间表问题

具有截止时间和误时惩罚的单位时间任务时间表问题可描述如下。

(1) n 个单位时间任务的集合 S={1,2,…,n}；

(2) 任务 i 的截止时间 ,1≤i≤n,1≤ ≤n，即要求任务 i 在时间 之前结束；

(3) 任务 i 的误时惩罚 ,1≤i≤n,即任务 i 未在时间 之前结束将招致的惩罚；若按时完成则无惩罚。

任务时间表问题要求确定 S 的一个时间表（最优时间表）使得总误时惩罚达到最小。给定一个单位时间任务的有限集 S。关于 S 的一个时间表用于描述 S 中单位时间任务的执行次序。时间表中第 1 个任务从时间 0 开始执行直至时间 1 结束，第 2 个任务从时间 1 开始执行至时间 2 结束，…，第 n 个任务从时间 n-1 开始执行直至时间 n 结束。这个问题看上去很复杂，然而借助于拟阵，可以用带权拟阵的贪心算法有效求解。对于一个给定的 S 的时间表，在截止时间之前完成的任务称为及时任务，在截止时间之后完成的任务称为误时任务。S 的任一时间表可以调整成及时优先形式，即其中所有及时任务先于误时任务，而不影响原时间表中各任务的及时或误时性质。类似地，还可将 S 的任一时间表调整成为规范形式，其中及时任务先于误时任务，且及时任务依其截止时间的非减序排列。

首先可将时间表调整为及时优先形式，然后再进一步调整及时任务的次序。任务时间表问题等价于确定最优时间表中及时任务子集 A 的问题。一旦确定了及时任务子集 A，将 A 中各任务依其截止时间的非减序列出，然后再以任意次序列出误时任务，即 S-A 中各任务，由此产生 S 的一个规范的最优时间表。

对时间 t=1,2,…,n，设(A)是任务子集 A 中所有截止时间是 t 或更早的任务数。考察任务子集 A 的独立性。

引理 4.6：对于 S 的任一任务子集 A，下面的各命题是等价的。

(1) 任务子集 A 是独立子集。

(2) 对于 t=1,2,…,n， (A)≤t。

(3) 若 A 中任务依其截止时间非减序排列，则 A 中所有任务都是及时的。

任务时间表问题要求使总误时惩罚达到最小，这等价于使任务时间表中的及时任务的惩罚值之和达到最大。下面的定理表明可用带权拟阵的贪心算法解任务时间表问题。

定理 4.7：设 S 是带有截止时间的单位时间任务集，I 是 S 的所有独立任务子集构成的集合。则有序对(S,I)是拟阵。

由**定理 4. 5** 可知，用带权拟阵的贪心算法可以求得最大权(惩罚)独立任务子集 A，以 A 作为最优时间表中的及时任务子集，容易构造最优时间表。

具体算法 GreedyJob 可描述如下：

```
int GreedyJob(int n,int d[],int J[])
{
    d[0] = 0;J[0] = 0;
    int k = 1;
    J[1] = 1;
    for(int i = 2; i <= n; i++){
        int r = k;
        while((d[J[r]] > d[i]) && (d[J[r]] != r)) r = r - 1;
        if((d[J[r]] <= d[i]) && (d[J[r]] > r)){
            for(int m = k; m > r; m--) J[m+1] = J[m];
            J[r+1] = i;
            k = k + 1;
        }
    }
    return k;
}
```

例如，给定单位时间任务集 S 及各任务的截止时间和误时惩罚如下：

i	1	2	3	4	5	6	7
d[i]	4	2	4	3	1	4	6
w[i]	70	60	50	40	30	20	10

算法 GreedyJob 先选择任务 1，2，3，4，然后舍弃任务 5，6，最后再选择任务 7。算法求得的最优时间表为{1，2，3，4，7，5，6}。其总误时惩罚为 w[5] + w[6] = 50，达到最小。

1.5 模拟法

有些自然界和日常生活中的事件，若用计算机很难建立枚举、递归等算法，甚至于建立不了数学模型。例如下题

Web Navigation （poj 1028）

题目简述：

本题是一个模拟 Web 浏览器的程序。

默认主页为 http://www.acm.org/

输入为指令，输出为当前页面的网址。

输入的指令有四种：

1）VISIT（后接一个网址）：访问它后面紧接着的网址

2）BACK：访问当前网页的前一个

3）FORWARD：访问当前网页的后一个

4）QUIT：退出（关闭浏览器）

思路：

必须记录访问过的网址，因为是顺序关系，才用数组存储，下标为 0 的元素为默认页；有一个变量记录当前页面在数组中的位置；

BACK 和 FORWARD 指令可能出现非法情况：

BACK 到默认页再 BACK—>输出 Ignored

FORWARD 到最后一个访问到的页面再 FORWARD—>输出 Ignored

这样，就又有一个问题：要记录最后一个访问到的页面，对于这个问题只需在遇到 VISIT 指令的时候顺便操作

上面的指令的对应操作就变得很简单：

VISIT：将网址写入数组后输出，记录当前位置，这个位置也即最后一个访问到的页面

BACK:当前页面减 1，输出网址，注意判断非法情况

FORWARD:当前页面加 1，输出网址，注意判断非法情况

QUIT:退出

代码：

```
#include <iostream>
#include <string>
using namespace std;
int checkCommand(char* command)
{
  if(strcmp(command,"VISIT")==0) return 0;
  if(strcmp(command,"BACK")==0) return 1;
  if(strcmp(command,"FORWARD")==0) return 2;
  if(strcmp(command,"QUIT")==0) return 3;
}
int main()
{
  char browser[101][71]={"http://www.acm.org/"};
  char command[10];
  int i=0;
  while(1)
  {
        cin>>command;
        int n=checkCommand(command);
        int cur;
        switch (n)
        {
        case 0:
              i++;
```

```
                cur=i;
                cin>>browser[i];
                cout<<browser[i]<<endl;
                break;
            case 1:
                i--;
                if(i>=0) cout<<browser[i]<<endl;
                else
                {
                    cout<<"Ignored"<<endl;
                    i++;
                }
                break;
            case 2:
                i++;
                if(i<=cur) cout<<browser[i]<<endl;
                else
                {
                    cout<<"Ignored"<<endl;
                    i--;
                }
                break;
            case 3:
                return 0;
            }
        }
        return 0;
    }
```

本章参考资料：

王晓东 《计算机算法设计与分析（第 2 版）》

第二章：数据结构

2.1 引言

数据结构指的是数据在计算机内存储的形式。不同的存储形式与不同的算法有着密切的关系。运用一些特殊的数据结构，会使你的程序的时间复杂度大大降低。下面，就让我们一步一步走进数据结构的殿堂，领略其无穷的魅力！

2.2 基本数据结构

2.2.1 堆栈

堆栈是一种先进后出的数据结构，与递归很相似。它有一个栈顶指针，相信所有的数据结构书上都有堆栈的介绍，在此不作深入说明。下面主要介绍 STL 模板库中的堆栈：

stack 模板类的定义在<stack>头文件中，stack 模板类需要两个模板参数，一个是元素类型，一个容器类型，但只有元素类型是必要的，在不指定容器类型时，默认的容器类型为 deque。

① 栈的定义：

```
stack<int> s;//定义一个堆栈 s,栈内元素为 int 类型
```

② 入栈操作：

```
s.push(x);//将整型变量 x 压入栈顶
```

③ 出栈操作：

```
s.pop();//需要注意的是，出栈操作只删除栈顶元素，并不返回该元素
```

④ 访问栈顶元素：

```
s.top();//返回栈顶元素  int y=s.top();将栈顶元素赋值给 y,此操作并不删除栈顶元素
```

⑤ 判断栈是否为空：

```
s.empty();//栈为空时，返回 true,否则返回 false
```

⑥ 返回栈内元素个数：

```
s.size();
```

下面通过具体例题来介绍堆栈的用法：

题意简述：给定坐标系上一系列相邻的矩形，高度不等,求把这些矩形连起来看能够得到的最大矩形的面积.

输入：一个整数 n,表示矩形个数，n 等于-1 表示输入结束.接下来每行 2 个整数 w,h,代表每个矩形的宽和高.

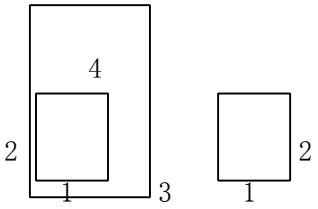
输出：能得到的最大矩形的面积.

Sample

3

1 2
3 4
1 2

得到答案为 12



此样例中最大矩形即中间那个 3*4 的矩形

思路：

⑴ 设置一个堆栈用来存储矩形.遇到比上一个矩形高度高或相等的新矩形就入栈

⑵ 如果遇到比上一个矩形高度低的矩形就要进行出栈合并以及最大矩形面积计算的处理.假设当前新矩形的高和宽分别为 data.h, data.w, 利用 totalw 统计出栈的矩形的宽度和,然后一直进行出栈操作,每出栈一个矩形就将其宽度加到 totalw 上,统计将出栈矩形高度乘以 totalw 与 ans 比较进行更新.出栈矩形的高度乘以 totalw 表示的是以这个矩形的高为大矩形的高可以得到的最大矩形面积.出栈操作一直进行到栈顶矩形的高度小于等于新矩形的高度.最后将新矩形的高度 data.h 以及 totalw + data.w 作为一个新的元素压入栈中.这样就完成了这种情况下对新矩形的操作

⑶ 最后栈中还有一些列的矩形,这些矩形构成了一个高度非降序的矩形序列.然后再进行一边出栈处理就可以了.

(注意理解栈内矩形高度是非降序的)

代码：

```
#include <iostream>
#include <stack>
using namespace std;
struct ab//矩形
{
   int w;//宽
   int h;//高
}data;
int main ()
{
   int n,ans,i,lasth,totalw,cursize;
   while(scanf("%d",&n)&&n!=-1)
   {
       ans=0;
       stack <ab> s;//定义一个空栈
       lasth=0;//上次进栈的矩形高度
       for(i=0;i<n;i++)
       {
           scanf("%d%d",&data.w,&data.h);
           if(data.h>=lasth)
```

```
            {
                s.push(data);//进栈
            }
            else
            {
                totalw =0;//总宽
                cursize = 0;//当前面积
                while(!s.empty()&&s.top().h>data.h)
                {

                    totalw+=s.top().w;
                    cursize=totalw*s.top().h;
                    if(cursize>ans)
                        ans=cursize;
                    s.pop();
                }
                totalw+=data.w;
                data.w=totalw;
                s.push(data);//新矩形压栈
            }
            lasth=data.h;
        }
        totalw =0;//总宽
        cursize = 0;//当前面积
        while(!s.empty())
        {
            totalw+=s.top().w;
            cursize=totalw*s.top().h;
            if(cursize>ans)
                ans=cursize;
            s.pop();
        }
        printf("%d\n",ans);
    }
    return 0;
}
```

2.2.2 队列

说到堆栈，就不得不说说与其相对应的队列，它是一种先进先出的数据结构。它有两个指针：一个队首指针，用来出队；一个队尾指针，用来进队。队列同样也有 STL 模板 queue 以及优先队列 priority_queue.这些具体在后面常用的 STL 中介绍。

2.2.3 图的邻接矩阵与邻接表

众所周知，图的存储方式一般有邻接矩阵和邻接表两种。这两种存储方式各有各的特点 。

就时间效率而言，如果是遍历图，邻接矩阵是 0(n^2)的（n 表示顶点数），而邻接表是 0(e)的（e 表示图中边的数目）。仅仅是换了一下数据的存储结构，时间复杂度就降低了，如果图中边不多，那降低的是相当可观的。但如果操作中要判断顶点 a 和 b 是否相连，那就要用邻接矩阵来得快了。还有就是有时候用邻接矩阵的话空间开不下来，但如果用邻接表动态分配空间就可以。

2.3 查找与排序

排序和查找并不需要对所有方式都能很熟练的掌握，但你必须保证自己对于各种情况都有一个在时间复杂度上满足最低要求的解决方案。

2.3.1 二分查找

二分查找是一种时间效率较高的查找。
二分查找要求：1.必须采用顺序存储结构 2.必须按关键字大小有序排列。
优缺点：折半查找法的优点是比较次数少，查找速度快，平均性能好;其缺点是要求待查表为有序表，且插入删除困难。因此，折半查找方法适用于不经常变动而查找频繁的有序列表。
算法思想：首先将给定 key 与字典中间位置上元素的关键码(key)比较，如果相等，则检索成功；否则，若 key 小，则在字典前半部分中继续进行二分法检索；若 key 大，则在字典后半部分中继续进行二分法检索。这样，经过一次比较就缩小一半的检索区间，如此进行下去，直到检索成功或检索失败。
所以，二分查找的时间复杂度为 O（log(n)）的。
代码：

```
int Bsearch(int key,int data[],int num)//在总数为 num 的数组 data 中二分查找 key
{
 int low=0,high=num-1,mid;//数组下标为 0~num-1
 while(low<=high)
 {
  mid=(low+high)/2;
  if(data[mid]==key)
   return mid;
  if(data[mid]>key)
   high=mid-1;
  else
   low=mid+1;
 }
return -1;//查找不成功
}
```

2.3.2 排序算法分析

2.3.2.1 快速排序

快速排序是使用的最多的一种排序了，原因就在于 STL 模板库中的 qsort 和 sort.写起来非常方便.下面介绍下 qsort 的用法:

qsort 需要包含头文件#include <algorithm>,它有四个参数，第一个为要排序的数组的首地址，第二个参数为要排序的元素的个数，第三个参数为每个元素所占的空间大小，第四个参数是需要自己写的比较函数,即告诉它应该按什么来排序.一种典型的写法如下:

qsort(s,n,sizeof(s[0]),cmp);下面具体介绍它的各种用法,主要是 cmp 函数的写法,cmp 函数有两个参数,均为 const void *类型，有一个返回值，为 int 类型.

① 对一维整型数组排序:

```
int cmp ( const void *a , const void *b )
{
return *(int *)a - *(int *)b;
}
```

如果整型数组为 a[],其中元素个数为 n,则 qsort(a,n,sizeof(a[0]),cmp);即完成了对 a[]数组的升序排序. 如果要按降序排列，可将 return *(int *)a - *(int *)b;改为 return *(int *)b - *(int *)a;

② 对 char 类型数组排序：

```
int cmp( const void *a , const void *b )
{
  return *(char *)a - *(char *)b;
}
char word[100];
```

qsort(word,100,sizeof(word[0]),cmp);即完成了对 word 中的字符按升序排列.

③ 对 double 类型数组排序：

```
double in[100];
int cmp( const void *a , const void *b )
{
return *(double *)a > *(double *)b ? 1 : -1;
}
```

qsort(in,100,sizeof(in[0]),cmp);

值得注意的是，cmp 里不能像对 int 排序那样写 return *(double *)a - *(double *)b,因为返回值是整型，这样返回值是 double 会出错且不易察觉.

④ 对字符串排序：

```
char s[100][100];
int cmp(const void *x,const void *y)
{
  return strcmp((char *)x,(char *)y);
}
qsort(s,100,sizeof(s[0]),cmp);
```

⑤ 对结构体一级排序：

```
struct node
{
  int a;
  int b;
}s[100];
int cmp(const void *x,const void *y)
{
  node xx=*(node *)x;
```

```
    node yy=*(node *)y;
    return xx.a-yy.a;
}
qsort(s,100,sizeof(s[0]),cmp);
```

按 a 的升序对结构体 s 进行排序

⑥ 对结构体二级排序:

```
    struct node
{
    int a;
    int b;
}s[100];
int cmp(const void *x,const void *y)
{
    node xx=*(node *)x;
    node yy=*(node *)y;
    if(xx.a!=yy.a)
     return xx.a-yy.a;
     else
     return xx.b-yy.b;
}
qsort(s,100,sizeof(s[0]),cmp);
```

先按 a 进行升序排序，如果 a 相同,按 b 的升序排列.

关于 qsort 的一些问题：

时间复杂度最好是 O(n),最坏为 O(n^2),平均为 O(nlog(n));

快排是不稳定的,这个不稳定表现在如果相同的比较元素,可能顺序不一样,假设我们有这样一个序列,3,3,3,但是这三个 3 是有区别的,我们标记为 3a,3b,3c,快排后的结果不一定就是 3a,3b,3c 这样的排列,所以在某些特定场合我们要用结构体来使其稳定.

即给每个元素加一个各不相同的标号，运用结构体二级排序，如果元素的值相等就按标号排序，这样就能保证稳定了。

快排的比较函数的两个参数必须都是 const void *的,这个要特别注意,写 x 和 y 只是我的个人喜好,写成 cmp 也只是我的个人喜好.推荐在 cmp 里面重新定义两个指针来强制类型转换,特别是在对结构体进行排序的时候.

快排 qsort 的第三个参数,那个 sizeof,推荐是使用 sizeof(s[0])这样,特别是对结构体,往往自己定义 2*sizeof(int)这样的会出问题,用 sizeof(s[0])既方便又保险.

如果要对数组进行部分排序,比如对一个 s[n]的数组排列其从 s[i]开始的 m 个元素,只需在第一个和第二个参数上进行一些修改:qsort(&s[i],m,sizeof(s[i]),cmp);

下面再简单介绍下 sort:

头文件与 qsort 一样，sort 有三个参数：第一个为待排序数组的首地址，第二个为尾地址，第三个为 cmp 函数. 如果是对基本类型排序(int, float, double 等)，cmp 可省略不写,默认的是升序排列. 例如对 int a[100]排序可写为 sort(a, a+100)；

如果是对结构体排序，就必须要自己写 cmp 函数了, sort 的 cmp 函数返回一个 bool 类型, 例如

```
struct node
{
  int a;
  int b;
}s[100];
```

要按 a 的升序对 s 进行排列,cmp 函数为

```
bool cmp(node x,node y)
{
   return x.a<y.a;
}
```

调用 sort(s,s+100,cmp);即完成了按 a 的升序排列,如果是降序排列，即使对整型数组排序也要自己写 cmp 函数了,将 x.a<y.a 改为 x.a>y.a 即可.

2.3.2.2 归并排序

归并排序法是将两个（或两个以上）有序表合并成一个新的有序表，即把待排序序列分为若干个子序列，每个子序列是有序的。然后再把有序子序列合并为整体有序序列。

如　设有数列{6，202，100，301，38，8，1}

初始状态：　[6] [202] [100] [301] [38] [8] [1]　　比较次数

i=1 [6 202] [100 301] [8 38] [1]　　3

i=2 [6 100 202 301] [1 8 38]　　4

i=3　[1 6 8 38 100 202 301]　　4

总计：　11 次

代码：

```
void MergeSort(int array[], int first, int last) //对 array 数组的从下标 first~last 进行归并排序
{
    int mid = 0;
    if(first<last)
    {
    mid = (first+last)/2;
    MergeSort(array, first, mid); //对左半边进行归并排序
    MergeSort(array, mid+1,last); //对右半边进行归并排序
    Merge(array,first,mid,last); //左右两边进行合并
    }
}
```

关于合并操作，即若左边为[6　202],右边为[100　301]合并操作将其变为[6 100 202 301]

代码如下：

```
int left_t[MAX];
int right_t[MAX]; //将左半边的数和右半边分离出来，放入其中
void merge(int a[],int p,int mid,int r)//将原数组 a[]中 p~mid 和 mid+1~r 进行合并,结果放入
a[]中
{
        int n1=mid-p+1;//左边数的个数
        int n2=r-mid,i,j; //右边数的个数
```

```
    for(i=1;i<=n1;i++)
    {
            left_t[i]=a[p+i-1];//将左半边的数存入 left_t[]中
    }
   for(i=1;i<=n2;i++)
    {
            right_t[i]=a[mid+i];//将右半边的数存入 right_t[]中
    }
    left_t[n1+1]=0x7fffffff;//最后一位设置一个最大值,这样下面合并不必判断结束
    right_t[n2+1]=0x7fffffff;
    i=1;//i 和 j 分别为 left_t[]  和 right_t[]的指针
   j=1;
   for(int k=p;k<=r;k++)//重置 a[]数组的 p~r
   {
            if(left_t[i]<=right_t[j])//左边数较小，将左边的数放入 a[]
            {
                    a[k]=left_t[i];
                    i=i+1;
            }
            else//右边数较小，将左边的数放入 a[]
            {
                a[k]=right_t[j];
                j=j+1;
            }
   }
}
```

归并排序有个作用，可以求逆序数:

例题：将一个序列进行冒泡排序，求总共需要交换的次数。(PKU 2299)

分析：对于一个数 a,如果后面有个比它小的数 b,他们俩总是需要交换一次，而 a 后面比 a 大的数总是不会和 a 交换的。所以对于 a,所要进行交换的次数是 a 后面比 a 小的数的个数（其实，如果 a 前面有个比 a 大的数 c,c 和 a 也是要进行交换的，这个交换算 c 的）。这样，总共需要交换的次数就是每个数后面比它小的数之和。也即这列数的逆序数。

如果对于每个数都向后直接找比它小的数的个数，这样算法复杂度是 O(n^2)，但如果用归并排序，是 O（nlogn）的。具体怎么做呢？考虑归并排序中合并的过程,例如当前对左边的有序序列 left[m]和右边的有序序列 right[m]合并，因为 left[]数组中的数在原序列中总是位于 right[]数组之前的，所以当某个 left[i]>right[j]时，说明在原序列中 left[i]后面有个比它小的 right[j],而且因为 left 是有序的，left[i~m]均大于 right[j],因而序列的逆序数应该加上 m-i+1。

这样，要求得整个序列的逆序数，只要在归并排序时的合并操作中，当要放入右边数时，将序列的逆序数加上 n1-i+1。即定义个全局变量 int count=0 表示数列的逆序数，在

```
else
{
  a[k]=right_t[j];
```

```
    j=j+1;
  }
```
中加入 count+=n1-i+1;即可。

2.3.2.3 拓扑排序

对一个有向无环图(Directed Acyclic Graph 简称 DAG)G 进行拓扑排序，是将 G 中所有顶点排成一个线性序列，使得图中任意一对顶点 u 和 v，若<u，v> ∈E(G)，则 u 在线性序列中出现在 v 之前。简单的说，由某个集合上的一个偏序得到该集合上的一个全序，这个操作称之为拓扑排序。

注意：

①若将图中顶点按拓扑次序排成一行，则图中所有的有向边均是从左指向右的。

②若图中存在有向环，则不可能使顶点满足拓扑次序。

③一个 DAG 的拓扑序列通常表示某种方案切实可行。

拓扑排序算法：

首先建图，保存每个顶点的入度。找到一个入度为 0 的点 a 输出，并在图中删去所有 a 的出边<a,b>并将 b 的入度减一。再找一个入度为 0 的点.....如此循环下去，直至找不到入度为 0 的点，这时可能排序完成(所有点均已输出)，也可能原数据不能构成拓扑序列，即存在环(还有点没输出)。

为便于考察每个顶点的入度，可保存各顶点当前的入度。为避免每次选入度为 0 的顶点时扫描整个存储空间，可设一个栈或队列暂存所有入度为零的顶点。在开始排序前，扫描整个存储空间，将入度为零的顶点均入栈(队)。以后每次选入度为零的顶点时，只需做出栈(队)操作即可。

题意简述：有 1 到 N 质量的球需要进行标号，给定 M 个约束，<a,b>使得标号 a 的球重量轻于标号 b 的球质量，求标号方案，按标号顺序输出各标号的球的重量，如果有多个解，输出字典序最小的序列，无解输出-1.(PKU 3687)

思路：如果是要求按球的重量顺序输出标号序列，直接建图正向进行拓扑排序即可，在有多个顶点入度为 0 时选择标号最小的一个。而题目要求的是重量的序列，则不可以这样做。反例 1,4; 4,2; 3,5 就可以说明。按照拓扑排序得到的是 1,3,4,2,5，这个序列为标签序列，即重量为 1 的球标号为 1，重量为 2 的球标号为 3……而题目要求的是球的重量序列,即为 1,4,2,3,5 即标号为 1 的球重量为 1，标号为 2 的球重量为 4……而如果我拓扑后得到的序列是 1,4,2,3,5 那么标号序列为 1,3,4,2,5 显然字典序比上一种方法小。可能你已经想到，本题即是要求一种拓扑序列:编号最小的节点要尽量排在前面;在满足上一个条件的基础上，编号第二小的节点要尽量排在前面；在满足前两个条件的基础上，编号第三小的节点要尽量排在前面……依此类推。对于上例中 1->4->2 和 3->5 两条链，1 肯定是要先输出的，再后面究竟是先输出 4 还是 3，跟 4 和 3 本身的大小没有关系，因为最小的 2 在 4 的后面，所以应该先输出 4，2，这样保证了第二小的尽量排在前面。这样，对于若干条平行的路径，小的头部不一定排在前面，但是大的尾部一定排在后面。但是，对于 4 和 3 两个头我并不知道尾部的情况，这里，我们可以采取建立反图，即将上例中的图建为 2->4->1 和 5->3，对于当前的 2 和 5，5 一定是排在后面的，这样拓扑答案是从尾到头逆着输出的。这样将 5 排在尾部，目前拓扑排序序列为：_,_,_,_,5,下面大的是 3: _,_,_,3,5,再然后分别是 2，4，1 拓扑序列则为 1,4,2,3,5.这样的标号序列为 1,3,4,2,5 是字典序最小的。

2.4 并查集

在一些有 N 个元素的集合应用问题中，我们通常是在开始时让每个元素构成一个单元素的集合，然后按一定顺序将属于同一组的元素所在的集合合并，其间要反复查找一个元素在哪个集合中。这类题目特点是看似并不复杂，但数据量极大，若用正常的数据结构来描述的话，往往在空间上过大，计算机无法承受；即使在空间上勉强通过，运行的时间复杂度也极高，根本就不可能在比赛规定的运行时间内计算出试题需要的结果，只能采用一种全新的抽象的特殊数据结构——并查集。

并查集类似一个森林，每个结点均有一个 father[x]来表示它的父亲结点。

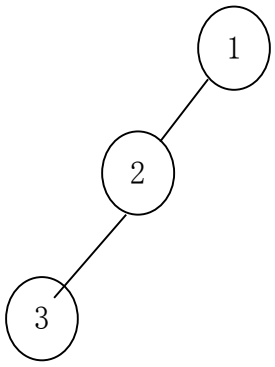
先介绍下并查集的初始化：

```
void Make_Set(int x)
 {
      father[x] = x; //根据实际情况指定的父节点可变化,这里用本身作为跟
  }
```

初始化集合 x，集合中只有 x 一个元素，将其根设为自身，这样每个集合的标号即为根结点的标号，当找到 father[x]=x 时，这个 x 即为根。

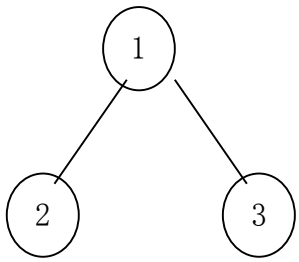
下面来看看它的两个重要操作：查找和合并。

查找:Find_Set(x)即查找元素 x 所在集合.



如图，我想知道元素 3 所在的集合。通过 father[3]=2,father[2]=1,再由 father[1]=1 确定了 3 元素所在集合为 1.可见，每次查找元素 x 所在的集合所需运算次数等于元素 x 所在的深度。

这里的查找有个路径压缩的优化,即得到 3 集合所在集合为 1 时,可以直接将 father[3]设置为 1，而且在查找 3 时会查找 2，可能还会有其他的元素，将这些元素的 father[]通通都设置为 1，这样，下次查找 3 的子孙所在集合的时候，上例中，查找次数就都缩短了 1.当查找 3 的祖辈所在集合的时候，查找次数就都只为 1 了。路径压缩后集合变为:



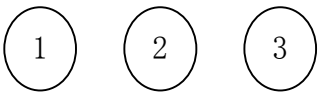
代码为：

```
int Find_Set(int x)
```

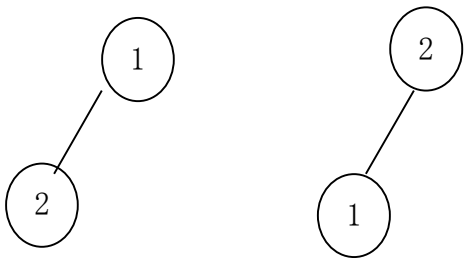
```
{
    if (x != father[x])
    {
        father[x] = Find_Set(father[x]); //这个回溯时的压缩路径是精华
    }
    return father[x];
}
```

这句 father[x] = Find_Set(father[x]);是递归的，回溯时将所有递归的元素的 father[]均设置为集合根结点。此优化正是并查集的优势所在。

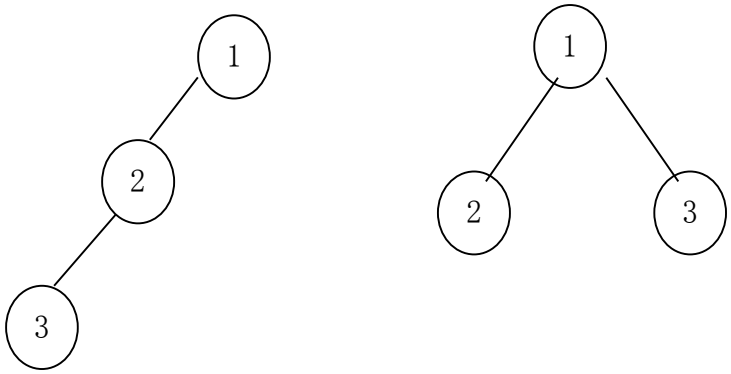
合并：Union(x,y)即合并 x 和 y 所在的两个集合，我们只需要把其中一个集合的根结点设置为令一个集合根结点的孩子即可。例如现在有三个结点。



合并 1 和 2，可以是



这两种情况，效果一样，但是再和 3 合并，可以是



这两种情况就不同了，在查找元素 3 所在集合的时候，前者先找到 father[3]=2,再找到 father[2]=1,两次找到 3 所在集合为 1，而后者只需要一次查找。这里后者优于前者，所以在合并时有一个启发式合并的优化，即记录每个集合的深度，当要合并两个集合时，将深度较小的集合挂在深度大的集合的下面。

合并代码：

```
void Union(int x, int y)
{
    int    fx = Find_Set(x);
    int    fy = Find_Set(y);
     if (fx == fy) return;
     if (rank[fx] > rank[fy]) //y 合并到 x
     {
```

```
        father[fy] = fx;
         if(rank[fy]+1>rank[fx])
         rank[fx]=rank[fy]+1;//这里的秩为深度
    }
    else
    {
        father[fx] = fy;
         if(rank[fx]+1>rank[fy])
         rank[fy]=rank[fx]+1;
    }
}
```

上面的这个用来启发式合并的秩可以自己改变，总之要使得问题能优化，这个秩也可设为集合中元素的个数。需要说明一点，这个秩，仅仅只有根结点的 rank[]值保存的是正确的信息，rank[x]并不等于 rank[fx],而且我们可以通过查找得到 x 所在集合的根结点 fx,所以 rank[x]不是必须的。（此优化效果不是很明显，作用不大，有时这样分情况讨论麻烦，也可直接将 y 挂到 x 上面）

并查集适用于所有集合的合并与查找的操作，进一步还可以延伸到一些图论中判断两个元素是否属于同一个连通块时的操作。由于使用启发式合并和路径压缩技术，可以讲并查集的时间复杂度近似的看作 O(1)，空间复杂度是 O(N)，这样就将一个大规模的问题转变成空间极小、速度极快的简单操作。

题意简述：Tadu City 里面有两个黑帮团伙 Gang Dragon 和 Gang Snake，一共有 n 名团伙成员(还不知道属于这两个黑帮的哪一个)。现在警察局有一些信息,每条信息包含 2 个人编号，表示这 2 个人属于不同的帮派。问给你 2 个人的编号，能否确定他们是否属于同一个帮派。(PKU 1703)

思路：典型的并查集的题目，有人可能会想：只需两个集合表示两个团伙，仔细想想，这样不好。当我给出你(1,2),(5,6)和(1,5),这样 2 和 5 属于一个集合，1 和 6 属于一个集合，怎么表示呢。刚开始（1，2）可能将 1 和 2 都 Make_Set（），但没有记录下 1 和 2 属于不同团队，下次(1,5)的时候不能找到应该将 2 和 5 合并的信息。

介绍一种常用做法，希望读者掌握：只要两者的关系确定了，就将他们放入同一集合内，而另外增加一个表示关系的数组 rela[]来表示该结点与其父亲的关系，0 表示是同一类，1 表示不同团伙。初始时集合只有自己一个元素，rela 设置为 0.

初始化为：

```
void Make_Set(int x)
 {
      father[x] = x;
      rela[x]=0;
}
```

当查找一个元素所在集合时,因为要进行路径压缩,一些结点的父亲可能会变为根结点，这样他们的 rela 值就要改。当其本来 rela 值为 0，其父亲 rela 值为 0（回溯时父亲已经先挂到了根结点上，这时父亲的 rela 即为与根结点的关系），那么该结点的 rela 修改为 0。同理,(0,1)->1,(1,0)->1,(1,1)->0 可见 rela[x]=rela[x]^rela[father[x]](^为异或运算)。

```
int Find_Set(int x)
{
    if (x != father[x])
    {
        int temp=father[x];
        father[x] = Find_Set(father[x]);
        rela[x]=(rela[temp]^rela[x]);//注意这里一定要在回溯时修改，此时 x 的父亲结点已经直接挂到了根结点上
    }
    return father[x];
}
```

合并集合时，合并 x 和 y 所在的集合,fx 和 fy 分别为集合根结点。这里我们仅仅将 fy 挂到 fx 下面，即合并了两个集合，这时候，只有 fy 的父亲结点改变了，所以要重新设置它的 rela 值，在查找 x 和 y 所在集合 fx,fy 的时候，由于路径压缩，已经将 x 直接挂在了 fx 上，y 直接挂到了 fy 上，现在枚举 rela[x]和 rela[y]的值，看 rela[fy]的情况。rela[x]为 0，说明 x 和 fx 是同类,rela[y]为 0,说明 y 和 fy 为同类，由于 x 和 y 一定是异类，所以 fy 和 fx 为异类,rela[fy]=1。同理，最终得到 rela[fy]=!(rela[x]^rela[y]);

```
void Union(int x, int y)
{
    int fx = Find_Set(xx);
    int fy = Find_Set(yy);
    if (fx == fy) return;
    father[fy] = fx;
    rela[fy]=!(rela[x]^rela[y]);
}
```

2.5 堆（优先队列）

堆是一个完全二叉树，可以用一维数组来实现。堆中存储的数是局部有序的。有两种不同的堆：最大值堆和最小值堆。最大值堆的任意一个结点的值都大于或等于其任意一个子结点存储的值。最小值堆的任意一个结点的值都小于或等于其任意一个子结点存储的值。所以在最小堆中，根结点一定是值最小的结点；最大堆中，根结点一定是值最大的结点。这点和优先队列一致，都是将优先级最高的放在队首。

堆(heap)经常被用来实现优先队列(priority_queue)：可以把元素加入到优先队列中也可以从队列中取出优先级最高的元素。

Insert(x):把 x 加入优先队列中

DeleteMin ():获取优先级最高的元素 x，并把它从优先队列中删除

除了实现优先队列，堆还有其他用途，因此操作比优先队列多

Getmin():获得最小值

Delete(x):删除任意已知结点

IncreaseKey(x, p):把 x 的优先级升为 p

Build(x):把数组 x 建立成最小堆

显然, 用堆很容易实现优先队列

堆是用数组实现的，最小堆的元素保存在 heap[1..hs]内

根在 heap[1]

K 的左儿子是 2k, K 的右儿子是 2k+1,

K 的父亲是[k/2]

① 删除优先级最高的元素, DeleteMin ():

此操作分为三步：

1， 直接删除根

2， 用最后一个元素代替根

3， 将堆向下重新调整

如果是最小堆，从根开始，选取当前结点 p 的较小儿子. 如果比 p 大, 调整停止, 否则交换 p 和儿子, 继续调整。

调整堆算法实现：

```
 void down(int p)
{
    //向下调整算法，p 代表当前节点，q 代表子节点
     int q=p*2;
     a=heap[p];//保存当前节点的值
     while (q<=hlength)//hlength 为堆中元素个数
     {
         if (q<hlength&&heap[q]>heap[q+1])
         //选择两个子节点中的一个最小的
             q++;
         if(heap[q]>=a) break;//如果子节点比当前节点大，就结束
         else//否则就交换
         {
             heap[p]=heap[q];
             p=q;
             q=p*2;
         }
     }
     heap[p]=a;//安排原来的节点
 }
```

删除最小元素,返回该最小元素：

```
int DeleteMin ()
{
 //删除最小元素算法
 int r=heap[1];//取出最小的元素
 heap[1]=heap[hlength--];//然后把最后一个叶子节点赋给根节点
 down(1);//调用向下调整算法
 return r;
}
```

② 在堆中插入一个新元素 Insert(x):

插入元素是先添加到末尾，再向上调整。

向上调整：比较当前结点 p 和父亲，如果父亲比 p 小，停止；否则交换父亲和 p，继续调整从下往上逐层向下调整. 所有的叶子无需调整，因此从 hs/2 开始. 可用数学归纳法证明循环变量为 i 时，第 i+1，i+2，…n 均为最小堆的根。

向上调整算法实现：

```
void up(int p)
{
//向上调整算法，p 代表当前节点，而 q 代表父母节点
int q=p/2;//获取当前节点的父母节点
a=heap[p];//保存当前节点的值
while (q<=hlength&&a<heap[q])
{
    //如果当前节点的值比父母节点的值小，就交换
    heap[p]=heap[q];
    p=q;
    q=p/2;
}
heap[p]=a;//最后安排原来的节点
}
```

插入元素算法

```
void insert(int a)
{
heap[++hlength]=a;//先往堆里插入节点值
up(hlength);//进行向上调整
}
```

③ 将 x 的优先级上升为 p：IncreaseKey(x，p)

```
int IncreaseKey(int p, int a)
{
//把 p 的优先级升为 a
    heap[p]=a;
    up(p);
}
```

④ 建立堆 Build(x)

```
void build()
{
//建堆算法
for(int i=hlength/2;i>0;i--)
down(i); //从最后一个非终端节点开始进行调整
}
```

时间复杂度分析：

向上调整/向下调整

- 每层是常数级别，共 logn 层，因此 O(logn)

插入/删除

- 只调用一次向上或向下调整，因此都是 O(logn)

后面会介绍 STL 中的优先队列，直接调用 STL 模板库，写起来非常方便简洁。

哈希表是一种高效的数据结构，主要体现在数据的查找上，几乎可以看成是常数时间。

在排序时，我们知道最低复杂度是 O(nlogn)。但对于一些特殊情况可以更快：现有 N(N>10000)个整数，范围在 0 至 10000,如何排序？我们建立一个数组 int num[10001],初始化为 0，num[i]表示 N 个数中有多少个数等于 i。这样我们每读入一个数 x，就将 num[x]++。最后从 num[0]~num[10000]依次取出这些数，复杂度为 O(n)。这个思想就是 hash：将某个对象对应到一个关键值，然后通过关键值归类，放入到一个表中（哈希表），今后可以根据关键值迅速查找。

Hash 表中的冲突：

哈希表式将某个对象对应到一个关键值，然而，可能是不同的对象对应到了相同的关键值，这就叫做冲突。如果刚才的数据范围是 1 至 10^10，则范围过大，可以对数据取模 p，得到一个较小的数字 key，作为关键值，再插入到 Hash 表，则 Hash 表的大小只需要等于 p。p 一般选择为较大的素数，或者也可以改为计算&0x1fffff 等类似方法避免取模运算较慢的问题。但是这样就会出现两个不同的数对应到同一关键值，例如 0 和 p，我们将 Hash 表的每一个位置做成一个链表，插入到链表中即可。这叫做开散列法。处理冲突的方法还有开放地址法和建立公共溢出区法，这里不再介绍。

PKU 2051

题意简述：给出任务的 id（各个任务唯一）和执行间隔（各个任务不唯一）；要求按照执行的时间顺序来输出要求的前 K 个任务 id 号；当两个任务在同一个时间执行时，先输出 id 小的.

思路：要求按照时间的先后顺序来输出，那么可以构造优先队列，每次取队头，然后再根据取出任务的执行情况将下一次的执行时间插入到队列里；直到取出满足要求的 个数任务为止。

2.6 Hash 表

Hash 中最常用的就是字符串 hash。要查找一个字符串是不是在数据中有，一般只能将已有字符串与其一一进行 strcmp，但是如果我们把每个字符串对应到一个整数值 key，出现这个字符串的话就将 hash[key]置为 1，要查找字符串 s 的时候，只需要计算出它的关键值 keys,看看 hash[keys]是不是为 1 即可。（前提是没有冲突的情况下）。常用的字符串哈希法有 Rabin-Karp，ELFHash，BKDRHash，APHash，DJBHash，JSHash，RSHash，SDBMHash，PJWHash 等等。另外还有以 MD5 和 SHA1 为代表的杂凑函数，这些函数几乎不可能找到碰撞。

①Rabin-Karp：如果字符串中可能出现的字符有 k 个，则可以将字符串对应到 k 进制数。

例如，如果字符串只可能为小写字母组成，则 acm 就对应到 0×26^2+2*26+12

log(2^63)/log(26)=13.403001373861878677 19

当字符串长度不超过 13 的时候，用 long long 作关键值类型，加上字符串长度作为限制，每个字符串唯一对应关键值。当字符串长度超过 13 的时候，就要进一步验证。如果链表中有值，不代表一定是你要找的字符串，两种方法可以进一步判定：

1,逐个字符比较字符串

2,建立另外一个或者多个 hash 函数，比较他们的其他 hash 关键值是否相同。

②ELFHash：

```
int ELFhash(char *key)
{
      unsigned long h=0;
      while(*key)
      { h=(h<<4)+*key++;
      unsigned long g=h&0Xf0000000L;
      if(g) h^=g>>24;
      h&=~g;
      }
      return h%MOD;//返回 hash 关键值，这个 MOD 最好用素数
}
```

③BKDRHash:

```
unsigned int BKDRHash(char *str)
{
    unsigned int seed = 131; // 31 131 1313 13131 131313 etc..
    unsigned int hash = 0;
    while ( *str )
    {
       hash = hash * seed + (*str++);
    }
    return (hash & 0x7FFFFFFF) % Mod ;
}
```

④SDBMHash:

```
unsigned int SDBMHash(char *str)
{
 unsigned int hash = 0;

 while (*str)
 {
  // equivalent to: hash = 65599*hash + (*str++);
  hash = (*str++) + (hash << 6) + (hash << 16) - hash;
 }

 return (hash & 0x7FFFFFFF) % Mod ;
}
```

⑤RSHash:

```
unsigned int RSHash(char *str)
{
 unsigned int b = 378551;
 unsigned int a = 63689;
 unsigned int hash = 0;

 while (*str)
 {
```

```
 hash = hash * a + (*str++);
 a *= b;
 }

 return (hash & 0x7FFFFFFF) % Mod;
}
```
⑥JSHash:
```
unsigned int JSHash(char *str)
{
 unsigned int hash = 1315423911;

 while (*str)
 {
  hash ^= ((hash << 5) + (*str++) + (hash >> 2));
 }

 return (hash & 0x7FFFFFFF) % Mod;
}
```
⑦PJWHash:
```
unsigned int PJWHash(char *str)
{
 unsigned int BitsInUnignedInt = (unsigned int)(sizeof(unsigned int) * 8);
 unsigned int ThreeQuarters = (unsigned int)((BitsInUnignedInt   * 3) / 4);
 unsigned int OneEighth  = (unsigned int)(BitsInUnignedInt / 8);
 unsigned int HighBits    = (unsigned int)(0xFFFFFFFF) << (BitsInUnignedInt - OneEighth);
 unsigned int hash     = 0;
 unsigned int test     = 0;

 while (*str)
 {
  hash = (hash << OneEighth) + (*str++);
  if ((test = hash & HighBits) != 0)
  {
   hash = ((hash ^ (test >> ThreeQuarters)) & (~HighBits));
  }
 }

 return (hash & 0x7FFFFFFF) % Mod;
}
```
⑧DJBHash:
```
unsigned int DJBHash(char *str)
{
 unsigned int hash = 5381;
```

```
 while (*str)
 {
  hash += (hash << 5) + (*str++);
 }

 return (hash & 0x7FFFFFFF) % Mod;
}
```

⑨APHash:

```
unsigned int APHash(char *str)
{
 unsigned int hash = 0;
 int i;

 for (i=0; *str; i++)
 {
  if ((i & 1) == 0)
  {
   hash ^= ((hash << 7) ^ (*str++) ^ (hash >> 3));
  }
  else
  {
   hash ^= (~((hash << 11) ^ (*str++) ^ (hash >> 5)));
  }
 }

 return (hash & 0x7FFFFFFF) % Mod;
}
```

这么多 HASH 函数没有必要都了解，有常用的一两种就足够了。

（PKU 2503）

题意简述：就是输入几组对应的字符串，其中一个是 English，另一个是 foreign language，开始是输入“字典”，然后是根据 foreign language 查询“字典”，没有时输出”eh”.

思路：运用 hash 函数直接对字符串进行定位，注意解决冲突。至于 HASH 函数，可以任意选择，总体上 BKDRHash 的效果最好，ELFHASH 冲突还是很多的。贴一下常用的 ELFHASH

代码：

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#define M 100003 //  取余的数
struct node{
    int hash;
    struct node *next;
}*link[M]={NULL};
char word[100000][11],dialect[100000][11];
```

```
int ELFhash(char *key)// ELFHASH 函数
{
    unsigned long h=0,g;
    while (*key)
    {
        h=(h << 4) + *key++;
        g=h & 0xf0000000L;
        if (g)
            h ^= g >> 24;
        h &= ~g;
    }
    return h % M;
}
int main()
{
    int i,e,n=0;
    char str[50];
    struct node *p;
    gets(str);
    while (strcmp(str,"")!=0)//创建字典
    {
        for (i=0;str[i]!=' ';i++)
            word[n][i]=str[i];
        word[n][i++]='\0';
        strcpy(dialect[n],str+i);
        e=ELFhash(dialect[n]);
        p=(struct node *)malloc(sizeof(struct node));
        p->hash=n;
        p->next=link[e];
        link[e]=p;
        n++;
        gets(str);
    }
    while (gets(str)!=NULL)//查询
    {
        e=ELFhash(str);
        p=link[e];
        while (p!=NULL)//处理冲突
        {
            if (strcmp(str,dialect[p->hash])==0)
                break;
            p=p->next;
        }
        if (p==NULL)
```

```
            printf("eh\n");
        else printf("%s\n",word[p->hash]);
    }
    return 0;
}
```

2.7 线段树

2.7.1 离散类型线段树

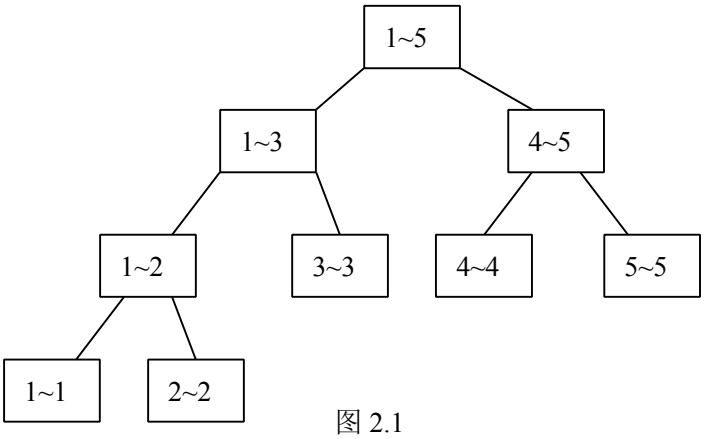


图 2.1

线段树是一种很重要的数据结构。用于对数据的更新和查询。主要优势体现在对段的处理上。比如：我要将数组 a[]的从 a[i]~a[j]的元素均加上一个数 b,那么我们将做 j-i+1 次。如果我们有一个段就表示 a[i]~a[j],那么我们直接将这个段上的和值 sum 加上 b*(j-i+1)即可，仅需一次操作。(这里我们只关心一个段的和)。再者，我们若想要知道 a[i]~a[j]的和，我们需查询 a[i],a[i+1]....a[j]的值，再将其相加，也是做 j-i+1 次，如果我们查询段 a[i]~a[j],直接取出该段的 sum 值即可，当 j-i+1 很大时，查询这个段的复杂度也会远远低于前面的做法。

那么,我们怎样来表示这个一段呢？要表示多少个段呢？下面我们介绍线段树是怎么做的：

线段树是一棵树，树中的每个结点表示了一个段。结点定义如下：

```
struct node
{
  int l;//线段的左端点
  int r;//线段的右端点
  int value;//线段上的值，上例中即为这个线段中各元素的和
};
```

对于每个结点，应该还有两个指针来表示该结点的左右孩子，我们用一个数组 tree[]来模拟一个树，那么，对于结点 tree[v],它的左孩子结点为 tree[v*2],右孩子结点为 tree[v*2+1](对于这一点,自己画一棵完全二叉树对其进行编号即能明白).那么左右孩子是什么呢?在线段树中，左右孩子分别是父结点的子区间，如父结点代表段 l~r,令 int mid=(l+r)/2; 则左孩子表示的区间为 l~mid,右孩子表示的区间为 mid+1~r。叶子结点表示

的区间是 x~x 的一个值，即是原数组中的一个元素.

那么对于 value 值的设置又是怎样的呢？我们是从叶子结点递归设置这个 value 的。就拿这里的区间和来说，叶子结点 x~x 的 value 即是 a[x](a[]为原数据数组).对于每个非叶子结点，其都有两个孩子结点，它的 value 即是两个孩子结点的 value 相加。完整的建树工作如下：

```
void bulid(int v,int l,int r)//对结点 v 进行建立，区间为 l~r
{
    tree[v].l=l;
    tree[v].r=r;
    if(l==r)
    {
       //进行结点的初始化
       tree[v].value=a[r];
       return ;
    }
    int mid=(l+r)/2;
    bulid(v*2,l,mid);
    bulid(v*2+1,mid+1,r);
    //根据左右儿子更新当前结点
    tree[v].value=tree[v*2].value+tree[v*2+1].value;
}
```

线段树主要进行的工作是：更新和查询。

更新：即维护树上的 value 值.

在对某一段 a[i]~a[j]上的所有元素都加上一个值 c 的时候，在线段树上是怎么操作的呢？首先找到这个段(可能是几个分段，比如找到两个段 i~k,k+1~j),将每个段上的 value 都加上 c*(r-l+1),r 和 l 表示这个段的区间,r-l+1 即这个段的长度.是不是这样就完成了呢？不是！比如我给 2~5 这个段加上了 c*4,那么，当我下次查询 2~3 的和的时候，这个加上 c*2 的操作并没有体现(该操作只是在 2~5 及其祖辈结点上有体现)。怎么解决这个问题？有一种做法是找到 2~5 结点更新后，再对其所有子孙结点进行更新，即也会给表示 2~3 的结点加上 c*2,但是这种做法不好。我们来看看，对长度为 n 的区间 i~j 进行更新，复杂度为 0(nlogn),而直接在原数据数组中更新只需要 n,线段树在更新上显得没有优势，而这样做往往也是会超时的。

下面介绍一种记录增量的方法，即给每个结点再加一个域 int add;记录更新操作的增量 c.初始时每个结点的 add 值均为 0,当对 2~5 区间进行更新操作后，给该节点的 add 加上一个值 c.再下次要对 2~3 结点进行更新或查询时，我们再将 add 传递到下面的孩子结点去。具体做法请看代码：

```
void update(int v,int l,int r,int m)//更新区间 l~r,加上数 m
{
    if(tree[v].l==l&&tree[v].r==r)//找到了，更新并记录增量
    {
        tree[v].value+=m*(r-l+1);
        tree[v].add=m;
        return ;//记得 return ,实际儿子没有更新，省时
    }
```

```
    if(tree[v].add)//上面没走掉，传递增量
    {
        tree[v*2].add+=tree[v].add;//儿子本身可能也有增量
        tree[v*2+1].add+=tree[v].add;
        tree[v].add=0;//传递了
    }
    int mid=(tree[v].l+tree[v].r)/2;
    if(r<=mid)
    {
        update(v*2,l,r,m);//只对左儿子进行更新
    }
    else
    {
        if(l>mid)
            update(v*2+1,l,r,m);//只对右儿子进行更新
        else//区间横跨了左右儿子区间，对其两者均进行更新
        {
            update(v*2,l,mid,m);
            update(v*2+1,mid+1,r,m);
        }
    }
}
```

查询：即查询区间 l~r 上的 value 值

```
void quary(int v,int l,int r)//当前查询结点为 v,要查询的区间是 l~r
{
    if(tree[v].l==l&&tree[v].r==r)
    {
        ans+=tree[v].value;//ans 为区间 l~r 上的和，找到了一个分区
        return ;
    }
    if(tree[v].add)//上面没走掉，传递增量
    {
            tree[v*2].add+=tree[v].add;//儿子本身可能也有增量
        tree[v*2+1].add+=tree[v].add;
        tree[v].add=0;//传递了
    }
    int mid=(tree[v].l+tree[v].r)/2;
    if(r<=mid)
    {
        quary(v*2,l,r);//要查询的区间都在左儿子
    }
    else
    {
        if(l>mid)
```

```
            quary(v*2+1,l,r);//要查询的区间都在右儿子
        else//要查询的区间分跨左右孩子
        {
            quary(v*2,l,mid);
            quary(v*2+1,mid+1,r);
        }
    }
}
```

线段树的用法非常灵活，下面介绍几个例题。离散化也是经常配合线段树使用的，因为线段树所需空间是 4*MAX 的，MAX 为原数据个数.

题意简述: 经典涂色问题. 题目大意很简单,有一个 N 长度的布,M 次操作:给区间[a,b]涂色，每次涂的颜色都不同，后涂的颜色可能会覆盖掉先前的颜色，问最后在布上能看到多少种颜色？（PKU 2528）

思路：建立线段树，每个结点增加一个信息 color 来表示该区间段上的颜色情况:-1 表示为尚未涂任何颜色，0 表示为混合颜色，>0 的任何数 a 表示该段为纯色 a。这样在第 k 次涂色对[a,b]涂色的时候（即给区间[a,b]涂色颜色 k），找到区间[a,b](可能是多个区间[a,k1],[k1,k2],[k2,b])，将区间的 color 都置 k,另外，应该将[a,b]区间的全部子孙结点的 color 也置为 k,但是前面讲过这样复杂度会较高,在此记录这个增量 k.然后对[a,b]的所有祖辈结点进行更新（实际这个更新是在回溯时进行的）：当两个儿子结点中有 color 为 0 的，父亲的 color 有为 0；两个儿子都不为 0 但不相等的，父亲也为 0.只有当两个儿子都不为 0 且相等，父亲结点的 color 也设为此值。这样最后统计的时候，从根开始往下寻找，找到 color 不为 0 的点后将不再向下查找，且将记录颜色种类的 ans 变量+1。

但是，此题中的 N 非常大（10^7），开线段树的空间是 4N，显然开不下来的。这里要介绍一个经常配合线段树使用的技术：离散化。 离散化，说白了就是用下标来代替本来的数据，目的是减小数组或线段树区间的大小。比如 数据：1 3 6 10 离散后为 0 1 2 3，对 1~10 的操作变为 0~3 的操作。 为什么能这样？ 起先必须要对原数据排序，这样保证了 1~10 中间的数也在操作范围内。比如不排序 1 10 3 6 离散后 0 1 2 3 ，你只是对 0~1 操作了，实际上 3 和 6 也在 1~10 区间内，也是要进行操作的，不排序就错了。 更重要的一点是： 离散化后会丢失信息！ 比如涂色问题中，我对 1~10，1~3，6~10 依次涂色（颜色可覆盖），最后能看到几种颜色？离散化后对 0~3，0~1，2~3 涂色，最后只能看到 2 种颜色（0 和 1 是一种，2 和 3 是一种）。但是，实际上，我会看到 3 种颜色（1~3 是一种，4~5 是一种，6~10 是一种）。为什么会这样呢，因为对于原始数据两个相邻的数 a 和 b 之间是没有数据的，但是离散化后就不一样了。这样可以看成是离散化后相邻数据 1 和 2 之间还有 1.5 这个数据。解决办法是，将点看成线段，比如，1 看成是【1，2），如此一来我就有了：0~1 是第二种颜色，1~2 是第一种颜色，2~3 是第三种颜色。但是，问题又来了：我如果要对 3~3 涂色，即离散化后的 1~1 涂色，该怎么办呢？将 1~1 看成是【1~2），即将点后面的空白段也加进去，一般地，这样对点的 a~b 涂色就变成了对线段的【a,b+1)涂色，这样就没有 a~a 的情况了。

题目简述：求相交矩形周长。给定 N 个矩形，这几个矩形相交的图形的周长。（PKU 1177）

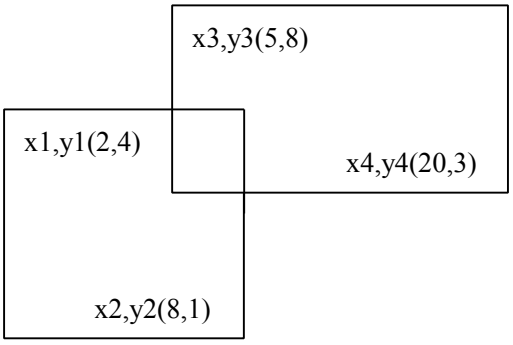


图 2.2

思路：我们先求出竖线的周长总和。用扫描线法：即将所有竖线按 x 从小到大进行排序，再标记是矩形的左边还是右边(左边用 0 标记，右边用 1 标记)上例中我们得到 4 条竖线分别为(1,4,0),(3,8,0),(1,4,1),(3,8,1)。我们建立一棵线段树，长度为 1~maxy,其中 maxy 为所有竖线中纵坐标 y 的最大值。然后遇到标记为 0 的线段就加入线段树，遇到标记为 1 的线段就从线段树中删除，用 pre 表示上次线段树中被覆盖的总长，用 len 表示加入当前线段后线段树的总长。开始时，pre 和 len 均是为 0 的。则每加入或删除一条线段，ans+=abs(pre-len);

首先，加入第一条线段后，len 为 4，pre 为 0，ans+=4,就将第一条竖线加入了周长中并将 pre 设置为 len=4。第二次，加入(3,8),现在线段树被覆盖的段为[1,8],这时 len=8,而 pre=4,ans+=abs(4-8).就将第二条竖线中的能作为周长的上部分长度 4 加入了答案中，把 pre 设置为 len=8。接着遇到第三条竖线，是标记为 1 的出边（1，4），这样将（1，4）删除，原先线段树中被覆盖的是（1，4），（3，8），其中（3，4）被覆盖了两次，删除（1，4）后线段树中被覆盖的段是（3，8），这时的（3，4）仅仅被覆盖一次了,所以这时候的 len=6,pre=8,ans+=abs(8-6);就将第三条竖线中能作为图形周长的下部分 2 加入了答案……

现在想想这个过程，线段树的每个结点需要记录什么信息？首先，这个结点中被覆盖的段的长度是需要的。由刚才第三条竖线的操作可知，还要记录这个段被覆盖了多少次。具体的这两个值在更新中的维护操作请读者自己思考完成，另外此题仍需要进行离散化。至于横边周长的求法同竖边一样，对横边按 y 进行排序，仍旧这样扫描。

2.7.2 树状数组

树状数组和线段树一样，也是一个结点表示一个线段，只不过线段树是二分来表示这个线段的。而树状数组不是：对于原数据 a[], 树状数组 c[], c[n]表示的段是 c[n]=a[n-2^k+1]+...+a[n], 其中 k 为 n 在二进制下末尾 0 的个数。比如 n=6(110),对应的 k 为 1。这个 c[]数组就是树状数组。具体每个结点表示的段见下图：

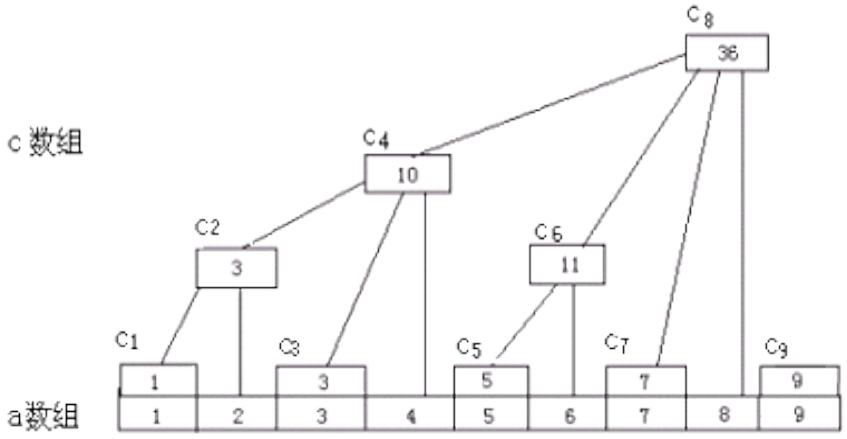


图 2.3

具体如何根据 n 计算 2^k 呢？2^k=n and (n xor (n-1)) 。Lowbit(x)函数就是用来根据 n 返回 2^k，

```
int Lowbit(int x)
{
      return x&(x^(x-1));//return x&(-x);也行
}
```

树状数组和线段树一样，主要做的工作也是查询和更新：

查询：怎样由 c[]数组求出原数据中区间[a,b]的和呢？由于 c[]数组中一个元素其实表示的是一段区间的和，那么这个操作会比直接在 a[]数组上效率要高。但怎么样找到区间呢？比如求区间[1,8]的和，直接取出 c[8]即可；如果是求[4,8]的和，貌似没有几个加起来是这个区间的，我们可以用[1,8]的和减去[1,3]的和，即 c[8]-(c[3]+c[2])。即将求[a,b]的和 sum(a,b)变为 sum(a,b)=sum(1,b)-sum(1,a-1);下面的问题就是如何求 sum(1,x)？从 x 开始向下依次求得各段：

```
int sum(int end)//计算原数组 1~end 的和
{
      int sum=0;
      while(end>0)
      {
          sum += c[end];
          end -= Lowbit(end);
      }
      return sum;
}
```

可以看出，这个算法就是将这一个个区间的和全部加起来，复杂度是 O(log(n))，为什么呢？以下给出证明： n = n - lowbit(n)这一步实际上等价于将 n 的二进制的最后一个 1 减去。而 n 的二进制里最多有 log(n)个 1，所以查询效率是 log(n)的。

树状数组另一个重要的操作：更新。即给原数据 a[x]加上一个数。可见，由于 c[]数组每个元素表示了一段，也就是说，这个 x 可能在若个个元素中都存在，所有要更新的点也不仅仅是一个。这个是从 x 开始向上依次更新包含 x 的各段

```
void update(int pos,int num)//给原数组 pos 下标的位置加上一个数 num
```

```
{
    while(pos<=N)//N 为元素个数
    {
        c[pos] += num;
        pos += Lowbit(pos);
    }
}
```

修改一个节点，必须修改其所有祖先，最坏情况下为修改第一个元素，最多有 log(n)的祖先。

树状数组的局限性：

有人可能要问：如果我想对一个区间进行更新怎么办呢？很遗憾，树状数组不能有效地进行这个操作，你可以像对原数组更新一样，一个一个地更新每个点(复杂度比在原数组上操作高 O(nlogn)的)。那么，我想对一个点查询呢？要查询 x,树状数组必须用 sum(1,x)-sum(1,x-1)，而直接在原数组 a[]中直接取得 a[x]即可。可见，树状数组适用性的约束是很大的。一般只适用于对点更新(O(logn))，对区间进行查询(O(logn))。而如果直接在原数据 a[]上操作，对点更新时 O(1)的，对区间查询时 O(n)的复杂度。而且，由于对区间查询时要进行转化:sum(a,b)=sum(1,b)-sum(1,a-1)，在不满足这个减法规则的转换时，也不能用树状数组。例如求区间[a,b]的最大值，它并不等于[1,b]的最大值减去[1,a-1]的最大值，这时候树状数组也不能用。（可以用线段树解决，线段树的应用非常广泛，也很灵活）。有一点值得说明的是：如果需要对区间进行更新，而对点进行查找，可以将 sum()函数和 update()函数内部实现调换一下，这样复杂度同样和树状数组一样，然而这个 c[]数组已经不是定义上的树状数组了。

```
int sum(int end)//计算原数组下标为 end 的值
{
     int sum=0;
     while(end<=N)
     {
          sum += c[end];
          end += Lowbit(end);
        }
        return sum;
    }
void update(int pos,int num)//给原数组 1~pos 下标的位置加上一个数
{
    while(pos>0)
    {
         c[pos] += num;
         pos -= Lowbit(pos);
    }
}
```

对于二维树状数组，代码的简洁程度让你不敢相信：

```
void update(int i, int j , int k)//给 a[i][j]这一点加上 k
{
    while(i<=n)
    {
         int temp=j;
```

```
            while(temp<=n)
            {
                c[i][temp]+=k;
                temp+=lowbit(temp);
            }
            i+=lowbit(i);
        }
    }
}
int sum(int i,int j)//查询 a[1][1]~a[i][j]的和（对原数组而言）
{
    int sum;
    while(i>0)
    {
        int temp=j;
        while(temp>0)
        {
            sum+=cc[i][temp];
            temp-=lowbit(temp);
        }
        i-=lowbit(i);
    }
    return sum;
}
```

转换函数也变为要求区间[x1][y1]~[x2][y2]，

SUM=sum(x2,y2)-sum(x1-1,y2)-sum(x2,y1-1)+sum(x1-1,y1-1);

树状数组和线段树的比较：

在很多的情况下, 线段树都可以用树状数组实现. 凡是能用树状数组的一定能用线段树.

当题目不满足减法原则的时候, 就只能用线段树, 不能用树状数组. 例如数列操作如果让我们求出一段数字中最大或者最小的数字, 就不能用树状数组了.

树状数组是一个可以很高效的进行区间统计的数据结构。在思想上类似于线段树，比线段树节省空间，编程复杂度比线段树低，但适用范围比线段树小。能用树状数组就不要用线段树了。

树状数组和线段树都在题目要求进行频繁的修改和查询的时候使用。线段树的使用很灵活，主要是思考每个结点需要记录什么，这个记录的值在修改过程中怎么维护。

第三章 数论

数论就是指研究整数性质的一门理论。整数的基本元素是素数，所以，数论的本质是对素数性质的研究。2000 年前，欧几里得证明了有无穷个素数。既然有无穷个，就一定有一个表示所有素数的素数通项公式，或者叫素数普遍公式。它是和平面几何学同样历史悠久的学科。高斯誉之为“数学中的皇冠” 按照研究方法的难易程度来看，数论大致上可以分为初等数论（古典数论）和高等数论（近代数论）。

初等数论主要包括整除理论、同余理论、连分数理论。它的研究方法本质上说，就是利用整数环的整除性质。 初等数论也可以理解为用初等数学方法研究的数论。其中最高的成就包括高斯的“二次互反律”等。

高等数论则包括了更为深刻的数学研究工具。它大致包括代数数论、解析数论、算术代数几何等等。 数论曾一度被认为是纯数学学科，与计算机无关。今天有关数论的算法正在程序设计中被广泛使用。例如有些试题看起来似乎属于盲目搜索类型，但一旦使用了数论算法后，便可使可使求解过程即简便又高效。

本章介绍一些数论的基本概念及相关算法. 它们都是我们应用数论的基础。

3.1 和素数有关的问题

3.1.1 素数基础

如何求 1~n 的所有素数?
如何判断一个数 n 是否为素数?
如何求两个数的最大公约数?
如何给一个数 n 分解素因数?

问题 1: 1~n 的素数
假设要求 1~100 的素数
2 是素数, 删除 2*2, 2*3, 2*4, …, 2*50
第一个没被删除的是 3, 删除 3*3, 3*4,3*5,…,3*33
第一个没被删除的是 5, 删除 5*5, 5*6, … 5*20
得到素数 p 时, 需要删除 p*p, p*(p+1), …
p*[n/p], 运算量为[n/p]-p, 其中 p 不超过 n1/2(想一想, 为什么)

问题 2: 素数判定
枚举法: O(n1/2), 指数级别
改进的枚举法: O(phi(n1/2))=O(n1/2/logn), 仍然是指数级别
概率算法: Miller-Rabin 测试+ Lucas-Lehmer 测试

Miller-Rabin 测试

对于奇数 n，记 n=2r*s+1，其中 s 为奇数

随机选 a(1<=a<=n-1)，n 通过测试的条件是

as≡1(mod n)，或者存在 0<=j<=r-1 使得 a2^j*s≡-1(mod n)

素数对于所有 a 通过测试，合数通过测试的概率不超过 1/4

只测试 a=2，3，5，7，则 2.5*1013 以内唯一一个可以通过所有测试的数为 3215031751

思考：区间内的素数

给出 n，m(n<=106，m<=105)，求 n~n+m 之间的素数有多少个

哪种方法快？筛还是依次素数判定？

下面是 Miller_Rabin 函数：

```
int witness(int a, int n)//采用概率算法
{
    int x, d=1, i = ceil(log(n - 1.0) / log(2.0)) - 1;
    for ( ; i >= 0; i--)
    {
        x = d;
        d = (d * d) % n;
        if (d==1 && x!=1 && x!=n-1)
            return 1;
        if (((n-1) & (1<<i)) > 0)
            d = (d * a) % n;
    }
    return (d == 1 ? 0 : 1);
}

int miller(int n, int s = 50)//对质数判断错误的概率仅为(1/2)^50
{
    if (n == 2)
        return 1;
    if ((n % 2) == 0)
        return 0;
    int j, a;
    for (j = 0; j < s; j++)
    {
        a = rand() * (n-2) / RAND_MAX + 1;
        if (witness(a, n))
            return 0;
    }
    return 1;
}
```

3.1.2 素数问题举例

prime gap（Poj 3518）

题意简述：输入一个整数（大于1且小于1299710），输出与这个整数相邻的2个素数之间的长度，若该整数本身就是素数，则直接输出0。

思路：对于素数的判断，可以直接判断，也可以打表将素数全部打出来，这样题目就比较容易了。

代码：

```
#include <iostream>
```

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

int a[1300000];

int main()
{
	int i,p,t,s,n,j,k;
	for(i=0;i<1300000;i++)
		a[i]=1;
	a[1]=0;
	for(i=4;i<1300000;i=i+2)//所有是2的倍数（2除外）都不是质数
		a[i]=0;
	p=2;
	while( p*p<1300000 )
	{
		p++;
		while( a[p]==0 )
			p++;
		t=p*p;
		s=2*p;
		while( t<1300000 )
		{
			a[t]=0;
			t=t+s;
		}
	}//同理，这样可以打出质数表
	while( scanf("%d",&n)!=EOF )
	{
		if( n==0 )
			break;
		if( a[n]==1 )
			printf("0\n");
		else
		{
			j=n-1; k=n+1;
			while( a[j]==0 || a[k]==0 )
			{
				if( a[j]==0 )
					j--;
				if( a[k]==0 )
					k++;
			}//直接判断就可以算出答案
			printf("%d\n",k-j);
		}
	}
	return 0;
}
```

2.素数判断

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

int isprime (int n )//用简单的方法直接判断n是否为质数
{
	int flag=0;
```

```
    if (n==2||n==3) return 1;
    for (int i=2;i*i<=n;i++ )
    {
        if (n%i==0)
        {
            flag=1;
            break;
        }
    }//枚举复杂度为o(sqrt(n))
    if (flag==0) return 1;
    else    return 0;
}

int main()
{
    int n;
    int i,j;

    while (scanf ("%d",&n),n)
    {

        if (isprime(n))
            printf ("%d\n",0);
        else
        {
            for (i=n;;i--)
                if (isprime(i))
                    break;//从n开始简单枚举即可
            for (j=n;;j++)
                if (isprime(j))
                    break;
            printf ("%d\n",j-i);
        }
     }

     return 0 ;
}
```

小结：针对于这一题，方法一对空间的要求比较大，当对于多case的样例时，采用将质数表打出来的方法能够很好的节省时间，而对于单case的测试样例时，可以节省很大的空间。当然关于素数的样例还有很多，变化也比较多，这里不一一赘述。

Calculation 2（hdu 3501）

题意简述：题目意思很清晰，就是求小于n并且与n不互质的数的和。

思路：由于数据范围比较大，显然需要用到__int64，暴力解决是不可能的。当然这道题目可以考虑使用积性函数做，下面介绍一下使用欧拉函数做的方法。

设小于N且与N互质的正整数之和, 设为ans.

不妨设这些数为a[0],a[1], a[2], ..., a[phi(N)-1],

由gcd(N, a[i]) =1,那么gcd(N,N - a[i]) =1

这样, N - a[0], N - a[1], ..., N - a[phi(N)-1]与原数列相同, 从而:

ans = a[0] + a[1] + ... + a[phi(N) -1]

ans= (N - a[0]) + (N - a[1]) + ... + (N - a[phi[N]-1]);两式相加得ans=N*phi[N]/2;那么结果就显然了res=(N-1)*N/2-ans;

代码：

```
#include<iostream>
```

```
using namespace std;
#define Mod 1000000007
int main()
{
	__int64 n;

	while (scanf ("%I64d",&n),n)
	{
		__int64 tmp=n;
		__int64 ans=n,res=(n-1)*n/2%Mod;//res为所有数的和

		for (int i=2;i*i<=tmp;i++)
		{
			if (tmp%i==0)
			{
				ans=ans*(i-1)/i;
				while (tmp%i==0)
					tmp/=i;
			}
		}//算出tmp的欧拉函数

		if (tmp>1)
			ans=ans*(tmp-1)/tmp;

		if (tmp==n)
			ans=n-1;

		ans=ans*n/2%Mod;
		res-=ans;//套用上面的公式

		if (res<0)//结果为负时利用模除的性质
		{
			res=(res+Mod*2)%Mod;
		}
		printf("%I64d\n",res);
	}

	return 0;
}
```

3.2 最大公约数　欧几里得及扩展欧几里得算法

3.2.1 欧几里得与扩展欧几里得简介

欧几里德算法又称辗转相除法，用于计算两个正整数a，b的最大公约数。

欧几里得算法：

求两个整数的最大公约数gcd(a,b)；

定理：gcd(a,b) = gcd(b,a mod b) (a>b 且a mod b 不为0)

```
int gcd(int a,int b)
{
```

```
    if(a%b==0) return b;
    else
        return (b,a%b);
}
```

扩展欧几里得算法:

求解线性方程 ax+by=c

1.应用：

线性方程 ax+by=c ，已知 a,b,c，求解 x,y.

2.基本思路：

ax+by=c 有解 => c=k*gcd(a,b)=kd（因为 d=gcd(a,b)=>d|(ax+by)）

我们先考虑求解 ax+by=d

由欧几里得算法,d=bx'+(a mod b)y'=bx'+(a-[a/b]b)y'=ay'+b(x'-[a/b])y'

则由上述两式子，我们可以得出 x=y' ,y=x'-[a/b]y' 这样子，在欧几里得算法添加 x,y 变量，最后得到解。（可结合下面代码源代码进行理解）

接下来我们来看看 ax'+by'=d 和 ax+by=c 之间的关系

（c/d)ax'+(c/d)by'=(c/d)d 即可以得到 x=(c/d)x',y=(c/d)y'所以可以得到 ax+by 的一组解

那么 ax+by=c 所有解的形式是什么呢？a(x+qb)+b(y-qa)=c; q 为任意整数(注意，当要求 y-qa 的最小正整数 min 时,由 y-qa>=0， q 取[y/a]最小，min=y-[y/a]y,但是,[y/a]可能为 0，如果 y 是负数，min 此时也为负数，不好，此时令 min+=a 就可以取得最小正整数值了（[y/a]=0 所以|y|<a),这段可以自己找个例子好好理解下啊）

扩展 GCD 求x, y使得gcd(a, b) = a * x + b * y;

```
int extgcd(int a, int b, int & x, int & y)
{
    if (b == 0)
    {
        x=1;
        y=0;
        return a;
    }
    int d = extgcd(b, a % b, x, y);//进行递归调用
    int t = x;
    x = y;
    y = t - a / b * y;
    return d;
}
```

3.2.2 欧几里得与扩展欧几里得举例

C Looooops（Poj 2115）

题意简述:for (variable = A; variable != B; variable += C)statement;求多少步能达到 B.如果不能就输出 FOREVER.

思路:扩展 GCD 算法. 求 a+cx=b(mod 2^k) 最小正整数解（0=<a,b,c<2^k,这个范围比较重要）我们通过扩展欧几里德求得一解 x,（无解就输出 FOREVER）x%=(2^k/d),if(x<0) x+==(2^k/d).x 为答案。

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

__int64 gcd(__int64 a,__int64 b)
{
	if(b==0)
		return a;
	else
		return gcd(b,a%b);
}

void ex_gcd(__int64 a,__int64 b,__int64 &x,__int64 &y)//扩展欧几里得算出方程整数解
{
	if(b==0)
	{
		x=1;
		y=0;
		return ;
	}
	ex_gcd(b,a%b,x,y);
	__int64 t=x;
	x=y;
	y=t-(a/b)*y;
	return ;
}

int main()
{
	__int64 a,b,c,k;
	__int64 x,y;

	while(scanf("%I64d%I64d%I64d%I64d",&a,&b,&c,&k)!=EOF && (a||b||c||k))
	{
		__int64 A=c;
		__int64 B=(__int64)1<<k;
		__int64 C=b-a;
		__int64 D=gcd(A,B); //转化为模线性方程

		if(C%D!=0)
			printf ("FOREVER\n");
		else
		{
			ex_gcd(A,B,x,y);
			x=x*(C/D);
			__int64 t=B/D;
			x=(x%t+t)%t;
			printf("%I64d\n",x);
		}

	}

	return 0;
}
```

3.3 整数因子分解

3.3.1 整数因子分解简介

分解因数可以转换为求最小素因子(找到最小素因子后递归求解)。分解素因数后得到惟一分解式 sum{pi^ki}, 可以求出约数个数, 即所有 ki+1 的乘积(由乘法原理容易证明)

方法一: 试除法，就是对于每个小于该整数的素数进行试除，直到除尽，这样便可以得到对该整数的因子分解。

方法二: pollard-rho 算法

下面我们来讨论将整数 n 分解为素数的积的问题。素数测试过程只对 n 进行合数判定，并未列出 n 的素数因子。对一个大整数进行因子分解似乎要比仅确定 n 是素数还是合数困难得多，即便用当今超级得计算机和现行得最佳算法，要对任意一个 200 位得十进制整数进行因子分解也还是不可行的。

这里，我们给出一种启发性方法，虽然它既不能保证其运行时间也不能确保其运行成功，不过在实际应用中，该算法还是非常有效的，因为它极可能找出大整数的素数因子。

该算法计算出下述序列

X_1，X_2，X_3，X_4，……X_1，X_{i+1}，……

其中 X_1，为 0.. n 一 1 之间的一个随机整数，其它数按下述递归式得出：

$X_i \longleftarrow (X^2_{i-1} - c) \bmod n$　　(2≤i)　c 是一个在 1..(n—1)间随机选取的整数

设 Y 为距离当前项 X_i 最近的 2 的次幂项。我们从第 1 项开始，逐项地延长序列。每生成一项 X_i 后检查若(y—X_i)与 n 的最大公约数 d 在 2.. n 一 1 之间，即 2≤gcd (y —X_i，n)≤n 一 1,则 d 为素数因子。上述过程一直进行至 y＝X_i 为止。(因为 0 与 n 的最大公约数为 0,n 无法分解出因子)

我们用一个 pollard_rho（c，n）的函数来描述这一算法。该函数输入序列递归式 $X_i \longleftarrow (X^2_{i-1} - c) \bmod n$ 中的 C 值和待因子分解的 n， 返回 n 的一个因子。但问题是这因子可能是还可继续分解的合数。为了真正分解出 n 的素数因子，我们采用了分治策略：

用 miller_rabin(n，2)函数测试当前因子 n，若为素数，则打印素数因子 n；否则重复下述过程：

调用 pollard_rho（c，n）分解出 n 的又一个因子 t_1;（n＝t_1*t_2）

递归分解 n 的因子 t_1;

递归分解 n 的因子 t_2;

直至无法分解下去为止。

我们用一个 rho(n)的递归程序描述这一分治过程，下面是程序题解。出于执行子函数 miller_rabin 时，需要通过反复平方法计算 $a^{n-1} \bmod n$ 的值。在迭代过程中，如果当前模取幂 $a^c \bmod n$ 中的 a^c 值超过计算机允许的整数最大值，程序可能会引出错误结论。

```
void divisors(int n)
{
    int i=n,j;

    for (j=0;n>1;j++)
    {
```

```
        while (n%prim[j]==0)       //对每一个小于n的质因子直至除尽
        {
            d[i][j]++;
            n/=prim[j];//记录所求结果
        }
    }

    if (j>m)
        m=j;
}
```

其中 prim[i]是素数表

3.3.2 整数因子分解举例

Divisors（Poj 2992）

题意简述：给出 n 跟 k，求出组合数有多少个约数。

思路：要知道一个数 n 约数的个数，首先要知道 n 的质因子分解：

$$n = p_1^{e_1} p_2^{e_2} \cdots_l^{e_l}$$

接着就可以按下式来算所有约数的个数了：

$$sum = (e_1 + 1) * (e_2 + 1) * \cdots * (e_l + 1)$$

其次，要知道一个数 n 的质因子分解，要打素数表，然后对每个小于 n 的素数 p 试除，看 n 含有多少个 p 的因子。

题目是要求 n!/(k!(n-k)!的约数个数，如果直接算出 n!/(k!(n-k)!会很麻烦，可以分别对 n!，k!，(n-k)!做如上分解，然后对于同一个底p_i，用 n 在p_i的指数e_i减去 k 对应的指数及(n-k)对应的指数，就得到 n!/(k!(n-k)!在底p_i的指数了。

可以用递推的方法来求 n!的质因子分解，具体见代码。

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

int p[83]={2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,
103,107,109,113,127,131,137,139,149,151,157,163,167,173,179,181,191,193,197,199,211,223,
227,229,233,239,241,251,257,263,269,271,277,281,283,293,307,311,313,317,331,337,347,349,
353,359,367,373,379,383,389,397,401,409,419,421,431};

int d[432][83]={0},m=0;

void divisors(int n)
{
    int i=n,j;

    for (j=0;n>1;j++)
    {
        while (n%p[j]==0)
        {
            d[i][j]++;
```

```
                n/=p[j];
            }
        }

        if (j>m)
            m=j;
}

int main()
{
        int l,n,k;

        for (int i=2;i<432;i++)
        {
            for (int j=0,k=i-1;j<m;j++)
                d[i][j]=d[k][j];
            divisors(i);
        }

        while (scanf("%d%d",&n,&k)!=EOF)
        {
            l=n-k;

            __int64 j;
            int i;

            for (i=0,j=1;i<83;i++)
                j*=(d[n][i]-d[k][i]-d[l][i]+1);      // d[n][i]-d[k][i]-d[l][i]+1为改质因子的幂指数
            //套用求因子的公式即可算出答案
            printf("%I64d\n",j);
        }

        return 0;
}
```

3．4 阶乘相关算法

有关阶乘的算法，不外乎两个方面：一是高精度计算；二是与数论相关。

1．高精度计算阶乘

实际上虽然是没有技术含量的问题，但是又会经常用到，所以还是得编写，优化它的计算。

首先看小于等于 12 的阶乘计算(计算结果不会超出 32 位范围)：

```
int factorial(int n) {
  if (n == 1 || n == 0) return 1;
  return factorial(n-1)*n;
}
```

这个递归程序简单明了，非常直观，然而一旦 n > 12，则超过 32 位 int 型的范围出现错误结果，所以上面这个递归程序仅适合 n <= 12 的阶乘计算，为了计算较大 n 的阶乘，需要将高精度乘法算法纳入到阶乘计算中来，高精度乘法过程可以如下简单的描述：(其中 A * B = C，A[0]，B[0]，C[0]分别存储长度)

当然在 ACM 竞赛中常常使用 Java 语言来处理大数的问题。

2．与数论有关

由于阶乘到后面越来越大，巧妙的利用数论求得一些有趣的数字（数值）等成为阶乘算法的设计点，下面给出几道相关的问题与分析：

(1)计算阶乘末尾第一个非 0 数字

这是一个比较经典的问题,下面这个简单易懂的算法：

观察 n!，可以发现在乘的过程中，对于任意 n > 1，n!的末尾第一个非 0 数字都是偶数。我们只需保留最后一位非零数。当要乘的数中含有因数 5 时，我们可以把所有的因数 5 都当作 8 来乘。这是因为:

...x2*5=...10（舍）或...60，最后一位非零数为 6。而恰好 2*8=16，末位为 6。

...x4*5=...70（舍）或...20，最后一位非零数为 2。而恰好 4*8=32，末位为 2。

...x6*5=...30（舍）或...80，最后一位非零数为 8。而恰好 6*8=48，末位为 8。

...x8*5=...90（舍）或...40，最后一位非零数为 4。而恰好 8*8=64，末位为 4。

(对于 n > 1 时，最后一位不会出现 1, 7, 3, 9，而永远是 2, 4, 6, 8 的循环出现)

因此，在迭代作乘法时，主要就是计算因子 5 的数量，同时可见因子 5 的个数以 4 为循环节（即只需要取它的数量对 4 取模）。那么对于不同情况下的因子 5 的数量，可以通过 res[5][4] = {{0,0,0,0}, {2,6,8,4}, {4,2,6,8}, {6,8,4,2}, {8,4,2,6}}来得到，使用 nonzero[i]表示 i 的阶乘的最后一位，那么：

如果 t 是偶数，则直接乘：nonzero[i] = (nonzero[i-1]*t)%10。

否则 nonzero[i] = res[((nonzero[i-1]*t)%10)/2][five];

其中 t 是除掉所有因子 5 的结果，five 为因子 5 数量对 4 的模。

相关题目：http://acm.zju.edu.cn/show_problem.php?pid=1222 题。

不过这一道题注意的是，它的输入 n 并非只在 32 位 int 数值范围内，而是有很大的长度，所以计算这道题目时，需要利用到高精度除法(n/=5)和高精度加法(cnt+=n)。

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

#define MAXN 10000

char buf[1000];

int lastdigit()
{
    const int mod[20]={1,1,2,6,4,2,2,4,2,8,4,
        4,8,4,6,8,8,6,8,2};//最后末尾只可能出现这几个数

    int len=strlen(buf),a[MAXN],i,c,ret=1;

    if (len==1)
        return mod[buf[0]-'0'];

    for (i=0;i<len;i++)
        a[i]=buf[len-1-i]-'0';

    for (;len;len-=!a[len-1])
    {
        ret=ret*mod[a[1]%2*10+a[0]]%5;
```

```
        for (c=0,i=len-1;i>=0;i--)
            c=c*10+a[i],a[i]=c/5,c%=5;

    }//对2和5进行模除
    return ret+ret%2*5;//算出结果
}

int main ()
{
    while (scanf ("%s",buf)!=EOF)
    {
        printf ("%d\n",lastdigit());
        memset (buf,'\0',sizeof (buf));
    }
    return 0;
}
```

（2）阶乘末尾有多少个 0

分析发现，实际上形成末尾 0，就是因子 5 的数量，而计算 1~n 之间包含一个因子 i 的个数的简单算法就是：

```
cnt = 0;
while (n)
{
    n /= i;
    cnt += n;
}
```

因此，直接将 i 换成 5，就可以得到因子 5 的数量，也即 n!末尾 0 的数量。

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

int main()
{
    int f[13];
    f[0] = 1;
    for(int i = 1; i < 13; i++)
        f[i] = f[i - 1] * 5;
    int tc;
    scanf("%d", &tc);

    while(tc--)
    {
        int n;
        scanf("%d", &n);
        int tmp, sum = 0;
        for(int i = 1; i < 13; i++)
        {
            if((tmp = n / f[i]) == 0)
                break;
            sum += tmp;//直接加上存在的5的幂指数就能算出答案
        }
        printf("%d\n", sum);
    }

    return 0;
}
```

(3)判断数值 m 是否可以整除 n!

算法：使用素因子判断法

A. 首先直接输出两种特殊情况：

m == 0 则 0 肯定不会整除 n!；

n >= m 则 m 肯定可以整除 n!

B. 那么就只剩最后一种情况：m > n，我们从 m 的最小素因子取起，设素因子为 i 那么可以求得 m 的素因子 i 的个数 nums1；再检查闭区间 i ~ n 之间的数，一共包含多少个素因子 i，就可以简单的利用上面(2)中所介绍的数学公式进行计算得到 nums2。如果 nums2 < nums1，就表示 1 ~ n 中包含素因子的数量 < 除数 m 包含素因子 i 的数量，那么 m 必然不能整除 n!，置 ok = false。

C. 最后：如果 !ok or m > n or m == 0 则不能整除；否则可以整除

(4)数字 N 能否表示成若干个不相同的阶乘的和：

这里可以选择的阶乘为：0! ~ 9!，实际上这一题与数论无关，与搜索有关。

分析，由于可供选择的阶乘数量较少，直接可以利用 DFS 搜索来做：

A. 首先将 0 ~ 9 的阶乘作一个表 A[10]；再设置一个可以组成“和”的数组 ans[N]。

B. 深度优先搜索方法：

```
search(n) {
  for(i = n; i <= 9; i++) {
      sum += A[i];//求和如果sum在ans数组中不存在，则将sum插入到ans[]数组中
      search(n+1);
      sum -= A[i];            //回溯
  }
}
```

C. 最后对于输入 n，就在 ans 数组中查找是否存在 n，如果存在，则表示 n 可以表示成不同的阶乘和，否则不行。

第四章 计算几何

计算几何是几何学的一个重要分支，也是计算机科学的一个分支，研究解决几何问题的算法。在现代工程与数学、计算机图形学、机器人学、VLSI 设计、计算机辅助设计等学科领域中都有重要应用。

计算几何问题的输入一般是关于一组几何物体（如点、线）的描述；输出常常是有关这些物体的问题的回答，如直线是否相交，点围成的面积等问题。

在最近几年的各项信息学竞赛中也出现了一些计算几何题，让人觉得耳目一新，但实际情况表明，信息学竞赛中的计算几何题得分率往往是最低的。究其原因是多样的，一是这类题目要求学生有深厚的计算几何知识，而这些知识平时应用很少，不仅中学生很少接触到，甚至普通的大学计算机专业学生也不会去研究；二是要求学生有很强的数学功底（如高等数学）和良好的空间思维能力，对计算几何中一些经典的算法熟练掌握，综合应用，精度要求也很高，编程量往往也很大；三是这类题目的每个测试点往往包含多组测试数据，造成时间不够；再者，选手遇到这类题目，有时会因为理解和思维上的困难而无法下手，导致得零分；有时又会因为考虑问题不够全面导致一些致命的错误。但是，一旦你熟练掌握了计算几何中的一些基本问题和经典算法，那么，它解决问题往往是最高效的。因为：数学工具越高级，解决问题就越高效。

信息学竞赛中的计算几何题有一些不同于一般几何题的、很明显的特点。它们不会是证明题，依靠计算机进行的证明是极为罕见的。计算机擅长的是高速运算，所以这类题目中一般都有一个（或多个）解析几何中的坐标系，需要大量繁琐的、人力难以胜任的计算工作，这也许正是它们被称为计算几何的原因吧！计算几何题的几种常见问题类型有：

1、计算求解题

解这一类题除了需要有扎实的解析几何的基础，还要全面地看待问题，仔细地分析题目中的特殊情况，比如求直线的斜率时，直线的斜率为无穷大；求两条直线的交点时，这两条直线平行等等。这些都是要靠平时学习时的积累。

2、存在性问题

这一类问题可以用计算的方法来直接求解，如果求得了可行解，则说明是存在的，否则就是不存在的，但是模型的效率同模型的抽象化程度有关，模型的抽象化程度越高，它的效率也就越高。几何模型的的抽象化程度是非常低的，而且存在性问题一般在一个测试点上有好几组测试数据，几何模型的效率显然是远远不能满足要求的，这就需要对几何模型进行一定的变换，转换成高效率的模型。

3、最佳值问题

这类问题是计算几何题中比较难的问题，一般没有什么非常有效的算法能够求得最佳解，最常用的是用近似算法去逼近最佳解，近似算法的优劣也完全取决于得出的解与最优解的近似程度。

4.1 计算几何的基础——矢量

4.1.1 矢量基础

矢量分析是高等数学的一个分支，主要应用于物理学（如力学分析）。在一些计算几何问题中，矢量和矢量运算的一些独特的性质往往能发挥出十分突出的作用，使问题的求解过程变得简洁而高效。熟练掌握一些矢量分析的方法，并灵活地加以运用，就能轻松地解决许多看似复杂的计算几何题，或者会对我们解这类题目有很大帮助，甚至还有一些计算几何题是非用矢量方法不能解决的。

1、直线

(1)直线的方程：

一般形式为：ax+by+c=0，或y=kx+b。k 称为直线的斜率，b 称为截矩。

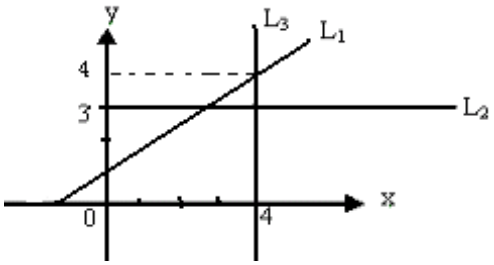


图4-1

如上图中，直线L_1的方程为：x-y+1=0，L_2：y=3，L_3：x=4。

特例：若a≠0，则一般方程可为：x+by+c=0；若b≠0，则一般方程可为：ax+y+c=0。

(2) 直线的斜率：

如下图：过两点P_1，P_2可以决定一条直线。斜率为：$k=\frac{y_2-y_1}{x_2-x_1}$,如上图中，直线$L_1$的斜率为1

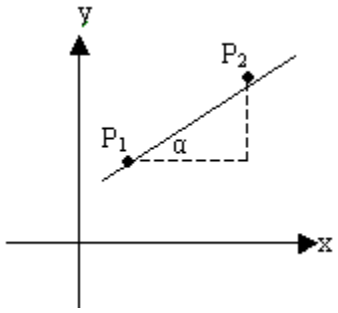


图4-2

特别地：当$y_1=y_2$时，斜率k=0(如上图中的L_2)，

当 $x_1=x_2$时，k 不存在(如上图中的L_3)。

而当x_1与x_2无限接近时，斜率k 趋于无穷大，这在编程时要特别小心。

(3) 两条直线垂直，则它们的斜率乘积等于-1。

2、线段

(1) 凸组合：两个不同的点P1(x1,y1)、P2(x2,y2)的凸组合是满足下列条件的点P3(x3,y3)：

对某个0≤σ≤1，有x3=σx1+(1-σ)x2, y3=σy1+(1-σ)y2

一般也可以写成：P3=σP1+(1-σ)P2

直观上看，P3 通过直线P1P2，并且处于P1 和P2 之间（也包括P1，P2 两点）的任意点。

(2) 线段：线段$P1P2$是两个相异点P1，P2 的凸组合的集合，其中P1，P2 称为线段的端点

3、向量（矢量）的概念

(1) 矢量：从平面上取一固定点$P_1(P_1=(X_1,Y_1))$，从P_1出发向某点P_2 $(P_2=(X_2,Y2))$引一条线段，有方向且有长度，这样的线段叫做有向线段，记作$\overrightarrow{P_1P_2}$，它的长度记作$\overrightarrow{P_1P_2}=\sqrt{(X_1-X_2)^2+(Y_1-Y_2)^2}$，点$P_1$叫做它的始点，点$P_2$叫做它的终点；如果$P_1$是原点(0，0)，则我们把有向线段$\overrightarrow{P_1P_2}$简记为向量$P_2$；如果不规定方向， 则记线段为$\overrightarrow{P_1P_2}$。

(2) 矢量的斜率：既然矢量是有方向的，那么矢量的斜率k 就是有正负之分的，具体如下：

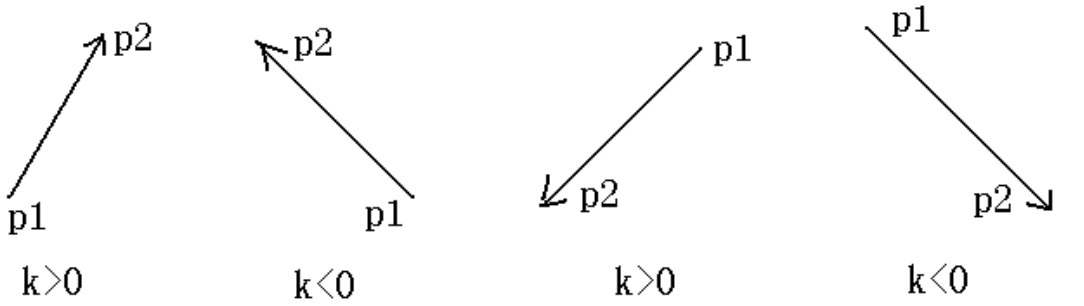


图4-3

(3) 设OA=a，则有向线段OA的长度叫做向量（矢量）a 的长度或模。记作|a|。

(4) 夹角：两个非0 矢量a、b，在空间任取一点0，作OA=a，OB=b，则角∠AOB 叫做矢量a 与b 的夹角，记作<a，b>。若<a，b>=π/2，则称a 与b 互相垂直，记作a⊥b。

4、矢量的加减法

以点0 为起点、A 为端点作矢量a，以点A 为起点、B 为端点作矢量b，则以点0 为起点、B为端点的矢量称为a 与b 的和a+b，如下中图。

从A 点作AB'，要求AB'的模等于|b|，方向与b相反，即AB'=-b，则以0为起点、B’为端点的矢量称为a 与b 的差a-b，如下右图。

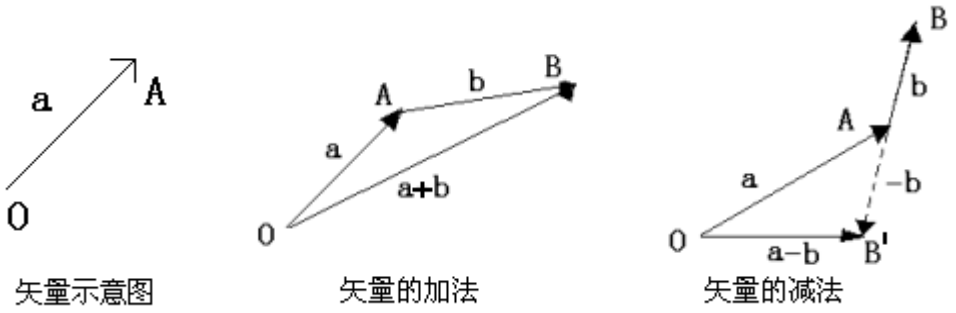


图4-4

5、矢量的分解

定理：如果空间三个矢量a, b, c 不共面，那么对任一矢量p，一定存在一个且仅一个有序实数组x, y, z，使得：p=xa+yb+zc。含义与物理上的合力和力的分解一样。

6、射影

已知向量$\overrightarrow{AB}$=a 和轴l，e 是l 上与l 同方向的单位向量，作点A 在l 上的射影A’，作点B在l 上的射影B’，则$\overrightarrow{A'B'}$叫做向量$\overrightarrow{AB}$在轴l 上或在e 方向上的（正）射影。

可以证明：$\overrightarrow{A'B'}$=|$\overrightarrow{AB}$|cos<a, e>

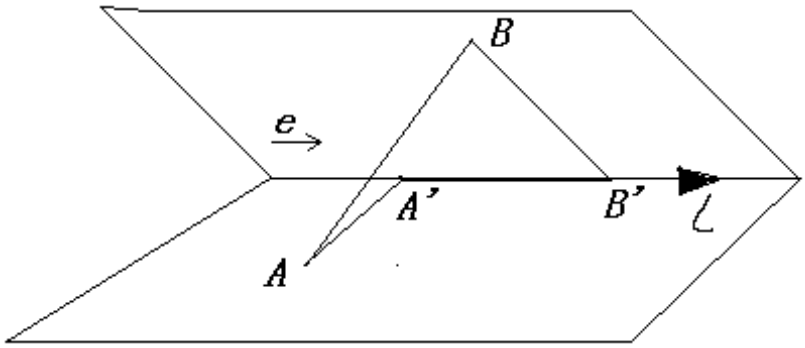


图4-5

7、矢量的数量积（点乘）

两个矢量的数量积是一个数，大小等于这两个矢量的模的乘积再乘以它们夹角的余弦。

a·b=|a||b|cos<a, b>

用上面讲到的矢量的分解可以证明，数量积等于两个矢量的对应支量乘积之和。

$a\cdot b=a_xb_x+a_yb_y+a_zb_z$

数量积的性质：

(1) a·e=|a||e|cos<a, e>=|a|cos<a, e>

(2) a⊥b 等价于a·b=0, 即$a_xb_x+a_yb_y+a_zb_z=0$

(3) 自乘：$|a|^2=a\cdot a$

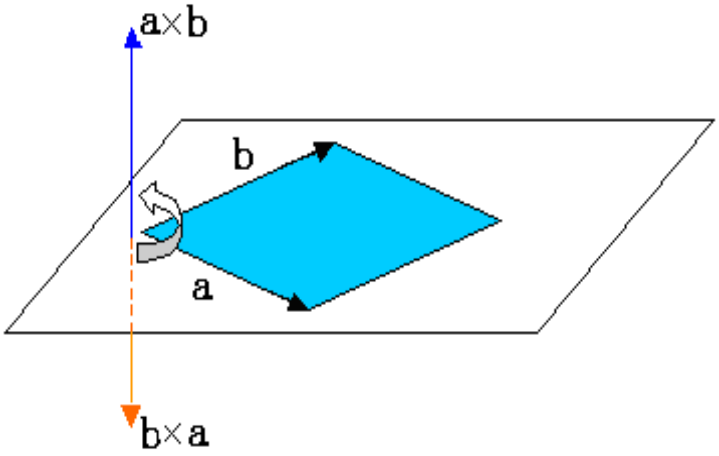
(4) 结合律：（λ·a）·b = λ(a·b)

(5) 交换律：a·b = b·a

(6) 分配律：a·(b + c) = a·b + a·c

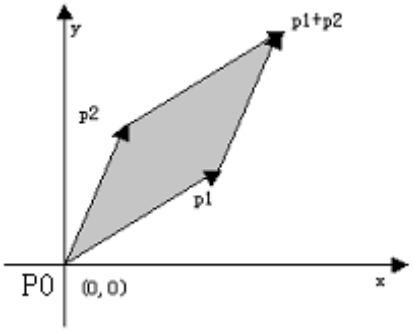
8、矢量的矢量积（叉乘、叉积）

(1) 矢量积的一般含义：两个矢量a 和b 的矢量积是一个矢量，记作a×b，其模等于由a 和b作成的平行四边形的面积，方向与平行四边形所在平面垂直，当站在这个方向观察时，a 逆时针转过一个小于π的角到达b 的方向。这个方向也可以用物理上的右手螺旋定则判断：右手四指弯向由A 转到B 的方向（转过的角小



于π），拇指指向的就是矢量积的方向。如右图（上）。　　图4-6

(2) 我们给出叉积的等价而更有用的定义，把叉积定义为一个矩阵的行列式：



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

如右图，如果$p_1 \times p_2$为正数，则相对原点(0,0)来说，p_1 在p_2的顺时针方向；如果$p_1 \times p_2$ 为负数，则p_1 在p_2的逆时针方向。如果$p_1 \times p_2$ =0，则1 p 和2 p 模相等且共线，方向相同或相反。

图4-7

也可以用矢量的分解证明：

$$A \cdot B = \begin{vmatrix} i & j & k \\ Ax & Ay & Az \\ Bx & By & Bz \end{vmatrix} = |i| \begin{vmatrix} Ay & Az \\ By & Bz \end{vmatrix} - |j| \begin{vmatrix} Ax & Az \\ Bx & Bz \end{vmatrix} + |k| \begin{vmatrix} Ax & Ay \\ Bx & By \end{vmatrix}$$

即：
$$(A \cdot B)x = AyBz - AzBy$$
$$(A \cdot B)y = AzBx - AxBz$$
$$(A \cdot B)z = AxBy - AyBx$$

注：i, j, k 分别为 x, y, z 方向上的单位矢量

(3) 矢量的叉积对于计算几何有着重要的意义，是很多算法的核心。用叉积可以判断从一个矢量到另一个矢量的旋转方向，可以求同时垂直于两个矢量的直线（矢量）方向，还能用来计算面积……在计算几何中大有用武之地，下面的例子中有更详尽的说明。

我们用函数则 multiply (P0，P1，P2)描述上述叉积的计算过程：

```
/*******************************************************************************
```

r=multiply(P0，P1，P2),得到$(P_1 - P_0)*(P_2 - P_0)$的叉积

r>0：则$\overrightarrow{P_0P_1}$在$\overrightarrow{P_0P_2}$的顺时针方向

r=0：则$\overrightarrow{P_0P_1}$在$\overrightarrow{P_0P_2}$共线

r<0：则$\overrightarrow{P_0P_1}$在$\overrightarrow{P_0P_2}$的逆时针方向

```
*******************************************************************************/
double multiply(POINT P0,POINT P1,POINT P2)
{
   return((P1.x- P0.x)*( P2.y- P0.y)-( P2.x- P0.x)*( P1.y- P0.y));
}
```

(4) 矢量的旋转：实际应用中，经常需要从一个矢量转到另一个矢量，这个过程称为矢量

的旋转，旋转的方向由叉积判定。很多例子都用到矢量的旋转这个方法和相应特性。

下面介绍坐标系中的矩形

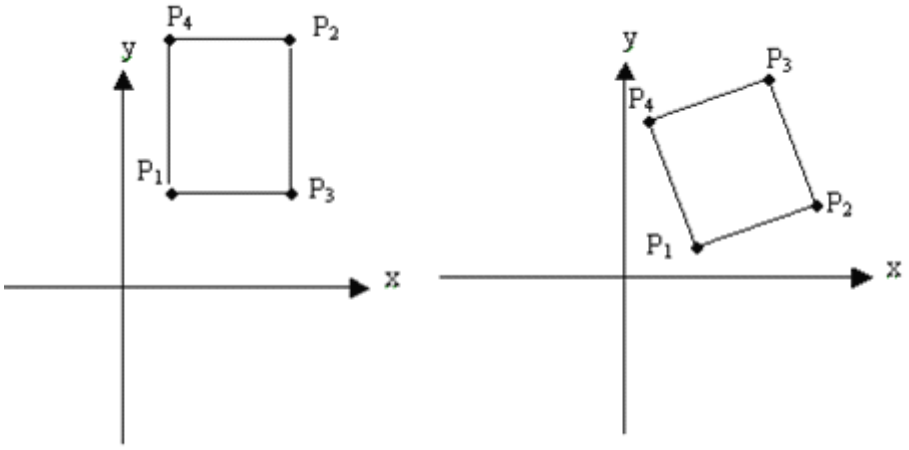


图4-8

若矩形边平行坐标轴，如(a)图，此时给出2 个点的坐标：$P_1(x_1, y_1)$， $P_2(x_2, y_2)$，则其它两点坐标为：$P_3(x_2, y_1)$， $P_4(x_1, y_2)$；

若矩形边不平行坐标轴，如(b)图，此时给出三点坐标：$P_1(x_1, y_1)$， $P_2(x_2, y_2)$，$P_3(x_3, y_3)$，则第4 点坐标可以推得，利用切割法可以得到：

$$\begin{cases} x_4 + x_2 = x_1 + x_3 \\ y_4 + y_2 = y_1 + y_3 \end{cases} \Longrightarrow \begin{cases} x_4 = x_1 + x_3 - x_2 \\ y_4 = y_1 + y_3 - y_2 \end{cases}$$

另外还有确定两条连续的有向线段 $\overrightarrow{P_0P_1}$ 和 $\overrightarrow{P_1P_2}$ 在 P_1 点是向左转还是向右转

有了叉积的基础，我们毋需通过对角 $\angle P_0P_1P_2$ 的计算来确定两条有向线段 $\overrightarrow{P_0P_1}$ 和 $\overrightarrow{P_1P_2}$ 在 P_1 点的转向，而仅需要添加一条辅助的有向线段 $\overrightarrow{P_0P_2}$，并检查 $\overrightarrow{P_0P_2}$ 是在有向线段 $\overrightarrow{P_0P_1}$ 的顺时针方向还是逆时针方向。为了做到这一点，我们计算出叉积

$$(P_2 - P_0)*(P_1 - P_0) = (X_2 - X_0)(Y_1 - Y_0) - (X_1 - X_0)(Y_2 - Y_0)$$

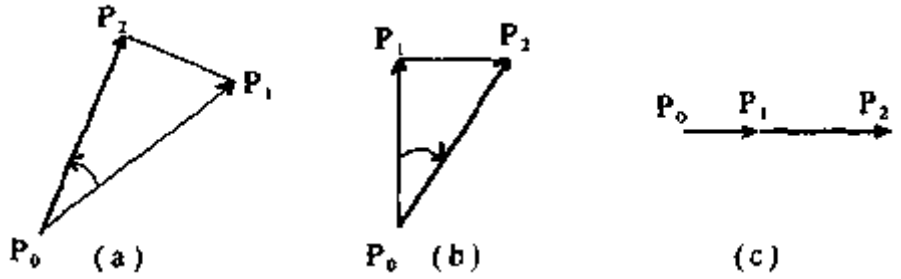


图 4-9

若该叉积为正，则 $\overrightarrow{P_0P_2}$ 在 $\overrightarrow{P_0P_1}$ 的逆时针方向，即 P_1 点向左转，(见图 4.1-3(a)。

若该叉积为负，则 $\overrightarrow{P_0P_2}$ 在 $\overrightarrow{P_0P_1}$ 的顺时针方向，即 P_1 点向右转，(见图 4.1-3(b)。

若叉积为 0，说明 P_0 P_1 和 P_2 共线(见图 4-9(c)。

显然．我们可以通过调用函数 multi(P_0 , P_1 , P_2)计算出叉积($P_2 - P_0$)*($P_1 - P_0$)，以确定两

条连续的有向线段 $\overrightarrow{P_0P_1}$ 和 $\overrightarrow{P_1P_2}$ 在 P_1 点的转向。

题意简述：任给三条直线的方程(斜率表示法，a1,c1,a2,c2,a3,c3)，求它们所围成的三角形的面积。

思路：算法分析很容易，过程如下：

设3 条直线的方程分别为：y = a1 * x + c1; y = a2 * x + c2; y = a3 * x + c3;

首先要判断这三条直线是否能构成三角形，若能则求出三个交点，再由交点求出三边的长度，再用海伦公式求出三角形的面积: s = sqrt(p*(p-a)*(p-b)*(p-c)) (p = (a+b+c)/2)；

构成三角形的条件是：三条直线的斜率互不相等，即a1,a2,a3 互不相等；

求两条直线的交点坐标，方法是求两个直线方程所列成的二元一次方程组的解，如:y=a*x+b，y=c*x+d，交点坐标为（(d-b)/(a-c),a*(d-b)/(a-c)+b）；

由两个点的坐标求两点之间距离只要用“两点间直线距离的公式”即可，如（a,b）、（c,d）两点间的距离为s=sqrt(sqr(a-c)+sqr(b-d))；

代码：略。

(5) 确定两条线段 $\overrightarrow{P_1P_2}$ 和 $\overrightarrow{P_3P_4}$ 是否相交

我们分两步确定两条线段是否相交

第一步：确定两条线段是否不相交——快速排斥试验

先通过下述方法求出两条线段的界限框(包含某线段且四边分别平行于 X 轴和 Y 轴的最小矩形)。

线段 $\overrightarrow{P_1P_2}$ 的界限框用矩形 $\left(\hat{P}_1,\hat{P}_2\right)$ 表示，其左下角的点为

$\hat{P}_1=\left(\min\left(X_1,X_2\right),\min\left(Y_1,Y_2\right)\right)$，右上角的点为 $\hat{P}_2=\left(\max\left(X_1,X_2\right),\max\left(Y_1,Y_2\right)\right)$；

线段 $\overrightarrow{P_3P_4}$ 的界限框用矩形 $\left(\hat{P}_3,\hat{P}_4\right)$ 表示，其左下角的点为

$\hat{P}_3=\left(\min\left(X_3,X_4\right),\min\left(Y_3,Y_4\right)\right)$，右上角的点为 $\hat{P}_4=\left(\max\left(X_3,X_4\right),\max\left(Y_3,Y_4\right)\right)$；

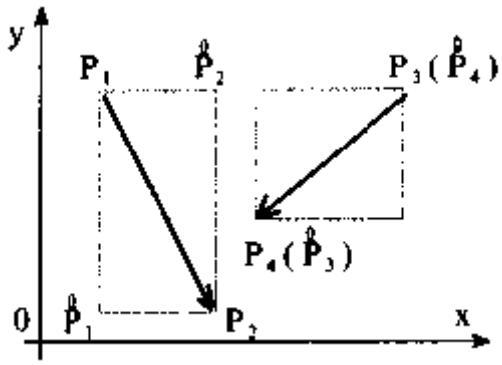


图 4-10

很显然，当 $\overrightarrow{P_1P_2}$ 界限框右上角 $\hat{P}_2$ 的 X 坐标 (max (X_1, X_2))≥ $\overrightarrow{P_3P_4}$ 界限框左下角 $\hat{P}_3$ 的 X 坐标(min (X_3, X_4))，并且 $\overrightarrow{P_3P_4}$ 界限框右上角 $\hat{P}_4$ 的 X 坐标 (max (X_3, X_4))≥ $\overrightarrow{P_1P_2}$ 界限框左下角 $\hat{P}_1$ 的 X 坐标(min (X_1, X_2))，则两个矩形在 X 方向相交；同样，当 $\overrightarrow{P_1P_2}$ 界限框右上角 $\hat{P}_2$ 的 Y 坐标(max (Y_1, Y_2))≥ $\overrightarrow{P_3P_4}$ 界限框左下角 $\hat{P}_3$ 的 Y 坐标(min (Y_3, Y_4))，并且 $\overrightarrow{P_3P_4}$ 限框右上角 $\hat{P}_4$ 的 Y 坐标(max (Y_3, Y_4))≥ $\overrightarrow{P_1P_2}$ 界限框左下角 $\hat{P}_1$ 的 Y 坐标(min (Y_1, Y_2))，则两个矩形在 Y 方向相交。

如果两条线段的界限框不能同时在 X 方向和 Y 方向相交，则这两条线段不可能相交。因此通过快速排斥试验，可以确定两条线段是否有相交的可能。

在两个矩形同时在两个方问相交的前提下，我们做第二步：确定一条线段 $\overrightarrow{P_1P_2}$ (或 $\overrightarrow{P_3P_4}$)是否跨立另一条线段 $\overrightarrow{P_3P_4}$ 或($\overrightarrow{P_1P_2}$)所在的直线——跨立实验。

所谓跨立是指某线段的两个端点是否分别处于另一线段所在直线的两旁，或者是否有其中一个端点处于另一线段所在的直线上。我们可以运用叉积的方法来确定。其设计思想是，从 P_1 出发向另一线段的两个端点 P_3 和 P_4 引出两条辅助线段 $\overrightarrow{P_1P_3}$ 、 $\overrightarrow{P_1P_4}$ 。然后计算两个叉积：由$(P_3 - P_1)*(P_2 - P_1)$确定 $\overrightarrow{P_1P_3}$ 在 $\overrightarrow{P_1P_2}$ 的哪个方向；由$(P_4 - P_1)*(P_2 - P_1)$ 确定 $\overrightarrow{P_1P_4}$ 在 $\overrightarrow{P_1P_2}$ 的哪个方向。

若两个叉积的正负号相同，说明 $\overrightarrow{P_1P_3}$ 和 $\overrightarrow{P_1P_4}$ 同在 $\overrightarrow{P_1P_2}$ 的一边，即 $\overrightarrow{P_3P_4}$ 不能跨立 $\overrightarrow{P_1P_2}$ 所在的直线。例如(图 4-11(b))；

若两个叉积的正负号相反，说明 $\overrightarrow{P_1P_3}$ 和 $\overrightarrow{P_1P_4}$ 同在 $\overrightarrow{P_1P_2}$ 的两边，即 $\overrightarrow{P_3P_4}$ 跨立 $\overrightarrow{P_1P_2}$ 所在的直线。例如(图 4-11(a))；

若任何一个叉积为 0，由于 $\overrightarrow{P_1P_2}$ 和 $\overrightarrow{P_3P_4}$ 已通过了快速排斥试验，因此 P_3 和 P_4 两点中

必有一点处于线段 $\overrightarrow{P_1P_2}$ 所在的直线上。例如图 4-11(c)。

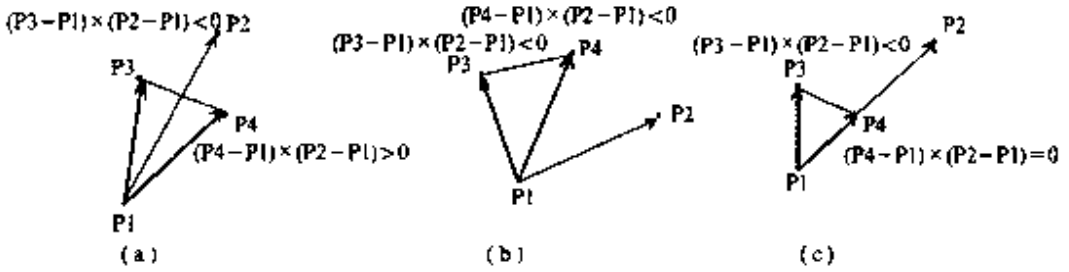


图 4-11

显然两条线段相交当且仅当它们能够通过快速排斥试验并且每一条线段能够跨立另条线段所在的直线。

下面是程序题解：

```
//如果线段 u 和 v 相交(包括相交在端点处)时，返回 true
bool intersect(LINESEG u,LINESEG v)
{
    return( (max(u.s.x,u.e.x)>=min(v.s.x,v.e.x))&&        //排斥实验
    (max(v.s.x,v.e.x)>=min(u.s.x,u.e.x))&&
    (max(u.s.y,u.e.y)>=min(v.s.y,v.e.y))&&
    (max(v.s.y,v.e.y)>=min(u.s.y,u.e.y))&&
    (multiply(v.s,u.e,u.s)*multiply(u.e,v.e,u.s)>=0)&&     //跨立实验
    (multiply(u.s,v.e,v.s)*multiply(v.e,u.e,v.s)>=0));
}
```

题意简述：在平面直角坐标系中，两点确定一条直线，题目给出四个点，前面两点和后面两点一共确定两条直线，求出这两条直线的交点。

思路：简单来看题目是比较简单的，根据两点能够求出直线方程，然后通过求解二元一次方程组求出交点。思路比较清晰，但是如果不再加思考，一般会得到一个 WA，这就是思考不够全面：首先根据两点确定的直线不一定存在斜率，因此直线方程应选择一般式即 ax+by+c=0，这样一般会比较严谨，另外一方面，得到的两条直线同样存在很多情况，比如两条直线存在平行、相交、重合三种情况，因而必须分别考虑（当然判断是比较简单的）。

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

typedef struct
{
    int x,y;
}Point;

typedef struct
```

```
{
    int a,b,c;
}Line;

int line(int x1,int y1,int x2,int y2)//向量(x1,y1),(x2,y2)
{
    return x1*y2-y1*x2;
    //结果为 0 时，向量平行
}

Line lineform(int x1,int y1,int x2,int y2)//ax+by+c=0;
{
    Line temp;
    temp.a=y2-y1;
    temp.b=x1-x2;
    temp.c=x2*y1-x1*y2;
    return temp;
}

double x,y;

void lineintersect(Line l1,Line l2)//求两直线的交点
{
    double d = l1.a * l2.b - l2.a * l1.b;
    x = (l2.c * l1.b - l1.c * l2.b) / d;
    y = (l2.a * l1.c - l1.a * l2.c) / d;
}

int main()
{
    int n;
    Point p1,p2,p3,p4;
    Line l1,l2;
    scanf("%d",&n);
    printf("INTERSECTING LINES OUTPUT\n");
    while(n--)
    {

    scanf("%d%d%d%d%d%d%d%d",&p1.x,&p1.y,&p2.x,&p2.y,&p3.x,&p3.y,&p4.x,&p4.y);
        if(line(p2.x-p1.x,p2.y-p1.y,p3.x-p1.x,p3.y-p1.y)==0&&line(p2.x-p1.x,p2.y-p1.y,p4.x-
p1.x,p4.y-p1.y)==0)
            printf("LINE\n");
        else
            if(line(p2.x-p1.x,p2.y-p1.y,p4.x-p3.x,p4.y-p3.y)==0)
```

```
                printf("NONE\n");
            else
            {
                l1=lineform(p1.x,p1.y,p2.x,p2.y);
                l2=lineform(p3.x,p3.y,p4.x,p4.y);
                lineintersect(l1,l2);
                printf("POINT %.2lf %.2lf\n",x,y);
            }
    }
    printf("END OF OUTPUT\n");
    return 0;
}
```

4.2 确定任意一对线段是否相交

下面，我们将讨论关于在一组线段中确定其中任意两条线段是否相交的问题。该算法输入由若干条线段组成的集合。若这组线段中任意两条线段相交，则返回布尔标志 true；否则返回 false。

算法使用了一种称为“扫除”的技术，即假设一条垂直扫除线沿 X 轴方向从左到右顺序移动，穿过已知的若干线段。移动过程中，每遇到一个线段端点，它就将穿过扫除线的所有线段放入一个动态数据结构中，并利用它们之间的关系进行排序，核查是否有相交点存在。

为了使问题简化，算法做了两点假设

1．没有输入线段是垂直的；

2．未有三条输人线段相交于同一点的情形。

由于不存在垂直的输入线段. 所以任何与给定的垂直扫除线相交的输入线段与其只能有一个交点。我们可以将此时与垂直扫除线相交的线段技交点的 Y 坐标值单调递增的顺序放在一个序列 T 中。当线段的左端点遇到扫除线时，线段就进入序列 T，当其右端点遇到扫除线时使随之离开序列 T。

具体点讲，考察两条不相交线段 S_1 和 S_2。如果垂直扫除线平移至 X 位置时与这两条线段相交，则我们说这些线段在 X 是可比的。如果 S_1 和 S_2 在 X 可比并且 S_1 与在 X 的扫除线的交点比 S_2 在同一条扫除线的交点高，则我们说在 X 处 S_1 位于 S_2 之上．写作 S：＞：S 2，反映在此时的 T 序列中，S_1 位于 S_2 前。

例如在图 4-12 中，我们有下述关系：

$a >_r c$，即在 r 处的 T 序列为 {a, b, c}；

$a >_t b$，$b >_t c$，$a >_l c$，即在 t 处的 T 序列为{a, b, c}；

$b >_u c$、即在 u 处的 T 序列为{ b、c }；

线段 d 与其它任何线段都不可比。

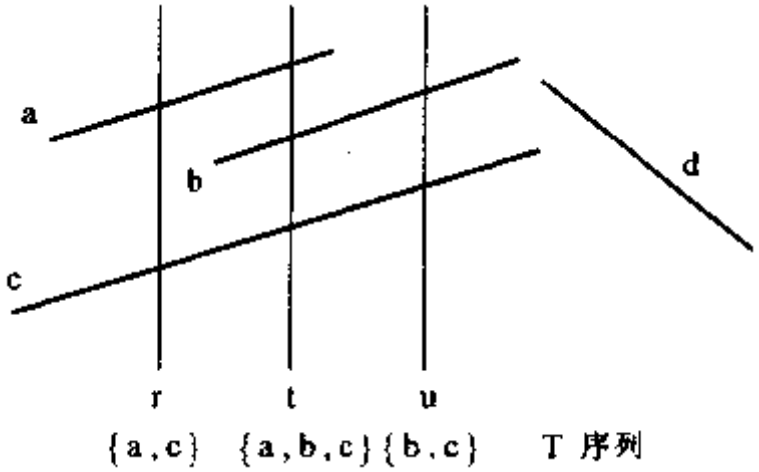


图 4-12

随着垂直扫除由左至右平移，T 序列中线段的数量和相对位置将由于下述原因而发生变化。

1．扫除线遇到某线段的左端点时，该线段进入 T 序列；遇到某线段的右端点时，该线段离开 T 序列；

2．扫除线穿过两线段交点的后的某区域（即在该区域内的 T 序列中，这两条线段是相邻的)，它们在 T 序列的位置将对换。

例如图 4-13 中，扫除线 V 和 W 分别位于线段 e 和 f 的交点的左边和右边。在交点的左边 r 时，$e >_r f$ 序列为{e，f}；而在交点右边的 w 时，$f >_w e$ 序列则变为{ f , e)。由于没有第三条线段相交于同一点，所以在交点附近必有一个连续区域(如图 4-13 中的阴影区域)，使得垂直扫除线在该区域活动时，相交线段 e 和 f 在序列 T 中是连续的。

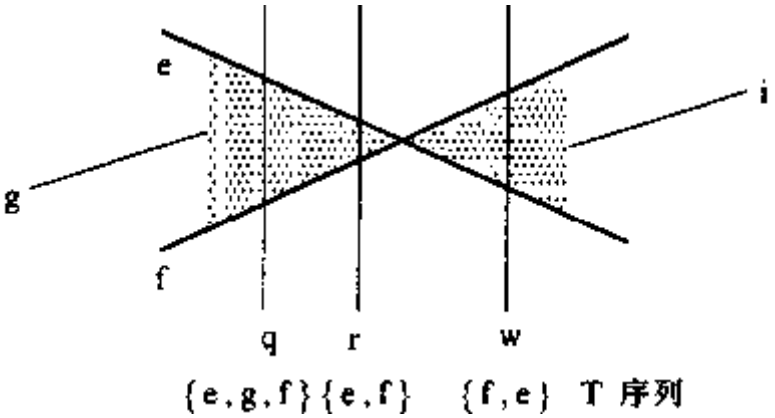


图 4-13

由此可见，典型的扫除算法，要安排下列两个数据集合：

(1). 由了序列给出与扫除续相交的线段之间的关系，即线段按相交点 Y 座标递增的顺序排列；

(2). 由左至右定义扫除线的一系列暂停位置，我们称每个这样的哲停住置为一个事件点。一般来说，事件点为每条线段的端点。当遇到线段左端点时，就把线段按排序要求插入 T 序列中；当遇到线段有端点时，就把它从 T 小删去；每当两条线段在 T 序列中第一次变为相邻时，我们就检查它们是否相交。

我们使用一个称为 any_sagment_intersect 的函数来描述算法流程。函数输入由 n 条线段组成的集合 S。如果 S 中任何一对线段相交．算法就返回布尔值 true；否则返回布尔

值 false。由于在“扫除”过程中，T 序列需按排序要求频繁地进行插入或删除，因此我们将它放入一个动态的指针链表中。

题意简述：给定两条线段的起点和终点，判断这两条线段是否相交。

思路：见以上分析.

代码：

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

const double eps=1e-8;

struct point
{
    double x, y;
};

struct line
{
    point s,t;
};

double min(double a, double b)
{
    return a < b ? a : b;
}

double max(double a, double b)
{
    return a > b ? a : b;
}

bool inter(point a, point b, point c, point d)//判断两条线段是否相交
{
    if ( min(a.x, b.x) > max(c.x, d.x) ||
         min(a.y, b.y) > max(c.y, d.y) ||
         min(c.x, d.x) > max(a.x, b.x) ||
         min(c.y, d.y) > max(a.y, b.y) ) return 0;
    double h, i, j, k;
    h = (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
    i = (b.x - a.x) * (d.y - a.y) - (b.y - a.y) * (d.x - a.x);
    j = (d.x - c.x) * (a.y - c.y) - (d.y - c.y) * (a.x - c.x);
    k = (d.x - c.x) * (b.y - c.y) - (d.y - c.y) * (b.x - c.x);
    return h * i <= eps && j * k <= eps;//注意计算几何中常见的相等和不等判断
}
```

```
int main ()
{
    int tc;
    scanf ("%d",&tc);

    while (tc--)
    {
        line l1,l2;
        scanf
("%lf%lf%lf%lf%lf%lf%lf%lf",&l1.s.x,&l1.s.y,&l1.t.x,&l1.t.y,&l2.s.x,&l2.s.y,&l2.t.x,&l2.t.y);
        if (inter(l1.s,l1.t,l2.s,l2.t))
            printf ("intersect\n");
        else
            printf ("no intersection\n");
    }

    return 0;
}
```

4.3 寻找凸包

4.3.1 寻找凸包

有一个点集 Q = {P_0, …, P_{n-1}}，它的凸包 ch(q)是一个最小的凸多边形 P，且满足 Q 中的每个点或者在 P 的边界上，或者在 P 的内部。在直观上，我们可以把 Q 中的每个点看作露在板外的铁钉.那么众包就是包含所有铁钉的一个拉紧的橡皮绳所构成的形状.例如图 4-14。

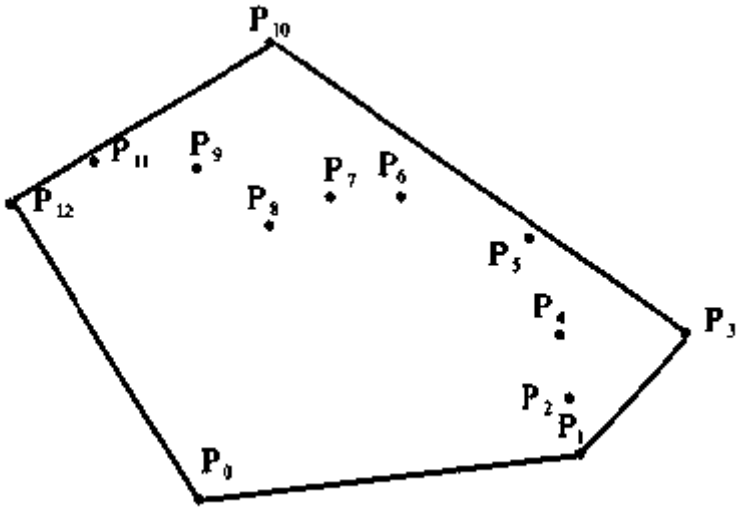


图 4-14

在计算一组点的凸包的基础上．可以引出一些计算几何学问题的算法。例如考虑二维

的最远点对问题：

已知平面上一组 n 个点，并希望找出彼此间最远的两个点。很显然，这两个点必定是凸包的顶点。运用计算凸包的算法求最远点对问题．可使算法效率提高不少。

下面我们将阐述两种计算 CH(Q)的算法。这两种算法都按逆时针方向输出凸包的顶点。

1 、graham 扫描法

首先我们必须明确，在点集 Q 小处于最低位置(Y 坐标值最小)的一个点 P_0 是凸包 ch(q)的一个顶点。如果这样的顶点有多个。则选取最左边的点为 P_0，我们从 P_0 出发，按逆时针方向寻找凸包的顶点。寻找过程中，除 P_0 外的每个点都要被扫描一次，其顺序是依据各点在逆时针方向上相对 P_0 的极角的递增次序。极角的大小、可以通过计算叉积

$$(P_i - P_0)*(P_j - P_0)$$

来确定：

若叉积＞0，则说明相对 P_0 来说，P_j 的极角大于 P_i 的极角，P_i 先于 P_j 被扫描；

若叉积＜0，则说明 P_j 的极角小于 P_i 的极角，P_j 先于 P_i 被扫描；

若叉积＝0，则说明两个极角相等。在这种情况下，由于 P_i 和 P_j 中距离 P_0 较近的点不可能是凸包的顶点，因此我们只需扫描其中一个与 P_0 距离较远的点。

例如图4.3 - 2 说明了图4-15 小的点按相对于P_0的极角进行排序后得到的扫描序列＜P_1、P_2、P_3＞。

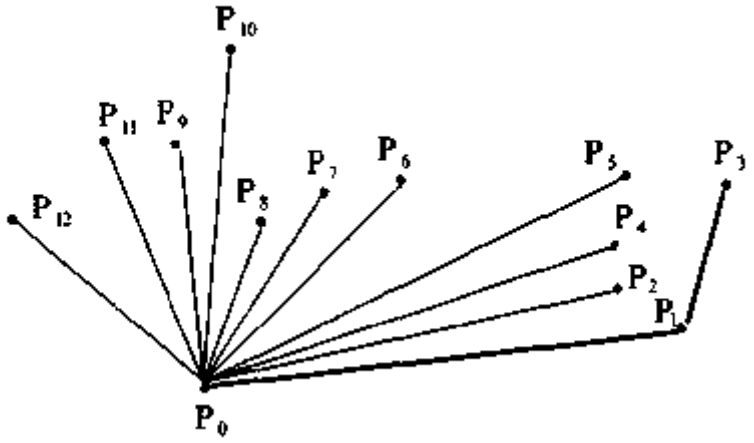


图 4-15

接下来的问题是如何确定当前被扫描的点 P_i (1≤i≤n - 1)是凸包上的点呢?我们设置一个关于候选点的堆栈 S 来解决凸包问题。初始时 P_0 和排序后的 P_1，P_2 作为初始凸包相继人栈。扫描过程中，Q 集合中的其它点都被推人堆栈一次，而不是凸包 ch(q)顶点的点最终将弹出堆栈。当算法结束时．堆栈 S 中仅包含 ch(q)的顶点，其顺序为各点在边界上出现的逆时针方向排列的顺序。

出于我们是沿逆时针方向通过凸包的，因此如果栈顶元素是凸包的顶点，则它应该向左转指向当前被扫描的点 P_i。如果它不是向左转，则它不属于凸包中的顶点，应从堆栈 S 中移出。在弹出了所有非左转的顶点后，我们就把 P_i 推入堆栈 S，继续扫描序列中的下一个点 P_{i+1}。

判断当前栈顶元素是否向左转指向扫描点 P_j，只需计算下述叉积的值。

$(P_i - P_{top-1})*(P_{top} - P_{top-1})$　(其中 top 为栈顶指针，P_{top-1} 为次钱顶元素)

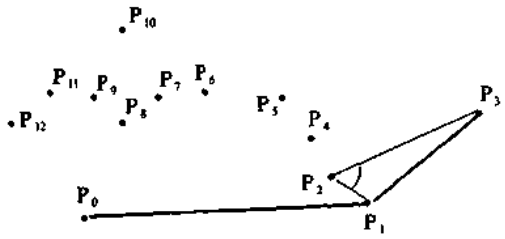
若叉积大于等于 0，说明线段 $\overline{P_{top-1}P_i}$ 在 $\overline{P_{top-1}P_{top}}$ 的顺时针方向或共线，P_{top} 未向左转。

在依次扫描了 $P_3..P_{n-1}$ 后，堆栈 S 从底 P 到顶部依次是按逆时针方向排列的 ch(q)中的顶点。

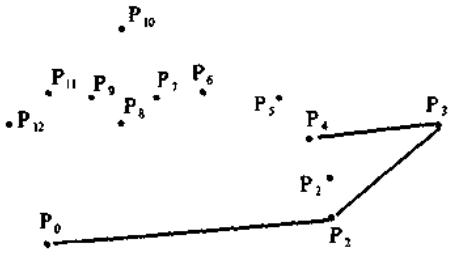
算法流程如下：
例如 graham_scan (q)输入图 4-16 所示的点集 q 后，其执行过程如下：

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

#define MAXN 10005

struct point
{
    double x, y;
};
bool mult(point sp, point ep, point op)
{
    return (sp.x - op.x) * (ep.y - op.y)>= (ep.x - op.x) * (sp.y - op.y);
}

bool operator < (const point &l, const point &r)
{
    return l.y < r.y || (l.y == r.y && l.x < r.x);
}

int graham(point pnt[], int n, point res[])
{
    int i, len, k = 0, top = 1;
    sort(pnt, pnt + n);
    if (n == 0) return 0; res[0] = pnt[0];
    if (n == 1) return 1; res[1] = pnt[1];
    if (n == 2) return 2; res[2] = pnt[2];
    for (i = 2; i < n; i++)
    {
        while (top && mult(pnt[i], res[top], res[top-1]))
            top--;
        res[++top] = pnt[i];
    }
    len = top;
    res[++top] = pnt[n - 2];
    for (i = n - 3; i >= 0; i--)
    {
        while (top!=len && mult(pnt[i], res[top], res[top-1]))
            top--;
```

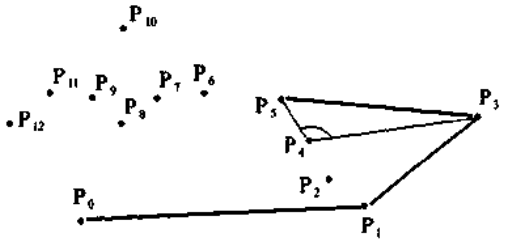
```
            res[++top] = pnt[i];//排序后对凸包上点的添加
        }
        return top; //  返回凸包中点的个数
}

int main ()
{
        int tc;
        scanf ("%d",&tc);

        point pnt[MAXN],res[MAXN];

        while (tc--)
        {
                int n;
                scanf ("%d",&n);

                for (int i=0;i<n;i++)
                {
                        scanf ("%lf%lf",&pnt[i].x,&pnt[i].y);
                }

                int ans=graham(pnt,n,res);
                printf ("%d\n",ans);
        }

        return 0;
}
```


P10
P11
P9
P7
P6
P12
P8
P5
P4
P3
P2
P0
P1

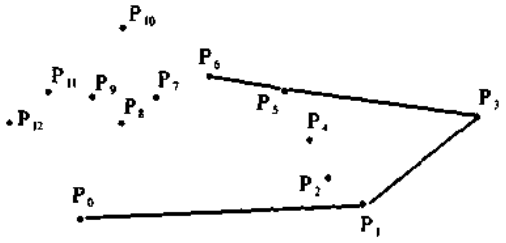
(a)


P10
P11
P9
P7
P6
P12
P8
P5
P4
P3
P2
P0
P1

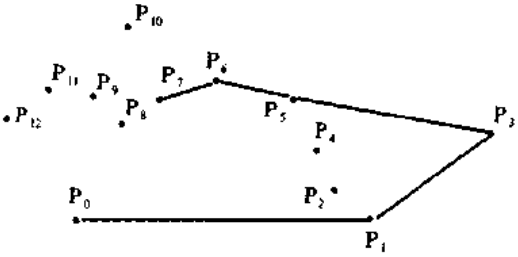
(b)


P10
P11
P9
P7
P6
P12
P8
P5
P3
P4
P2
P0
P1

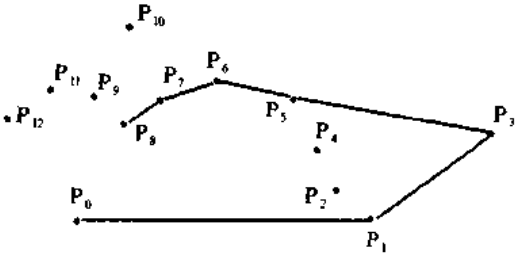
(c)


P10
P11
P9
P7
P6
P12
P8
P5
P3
P4
P2
P0
P1

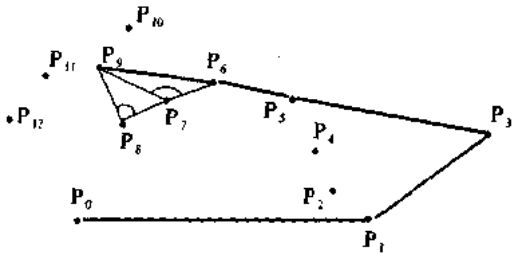
(d)


P10
P11
P9
P7
P6
P12
P8
P5
P3
P4
P0
P2
P1

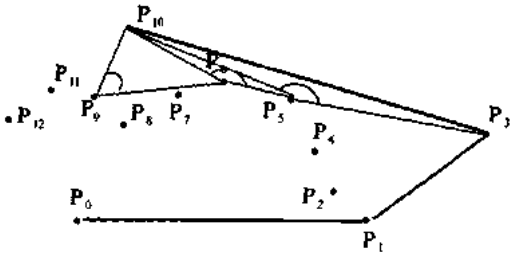
(e)


P10
P11
P9
P7
P6
P12
P8
P5
P3
P4
P0
P2
P1

(f)


P10
P11
P9
P6
P12
P8
P7
P5
P3
P4
P0
P2
P1

(g)


P10
P11
P9
P8
P7
P5
P3
P12
P4
P0
P2
P1

(h)

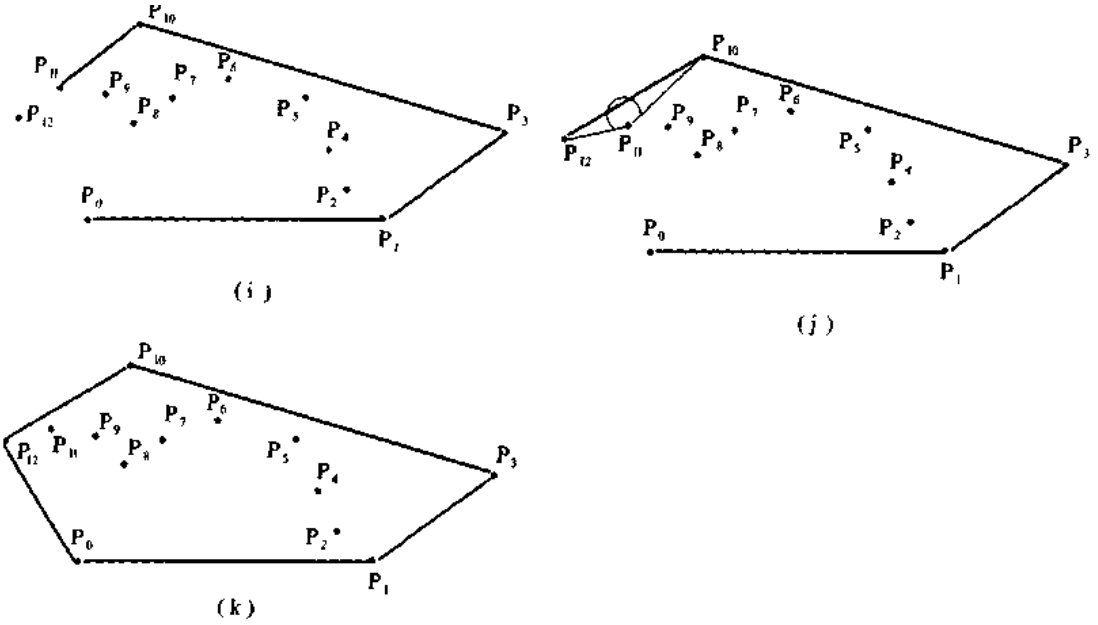


图 4-16

4.3.2 凸包举例

题目描述：给你一些树的坐标位置，让你求一个用这些树做篱笆桩所围成的多边行的最大面积（准确的说是这些面积的能够养牛的头数，每头牛至少需要 50 平方米的草坪）；

分析：题目意思是一个简单的凸包问题，直接找出凸包，求出凸多边形面积，然后就基本完成。

```
#include<iostream>
#include<stack>
#include<algorithm>
using namespace std;
#define ml 100005
struct node{
    int x,y;
};
node a[ml],b[ml];
stack<node> st;

int direction(node aa,node bb,node cc)//判断是否在同一方向，利用叉积判断
{
    int x1=bb.x-aa.x , y1=bb.y-aa.y , x2=cc.x-aa.x , y2=cc.y-aa.y;
    return x1*y2-x2*y1;
}

bool cmp1(node aa,node bb)
```

```
{
    if(aa.y==bb.y) return aa.x<bb.x;
    return aa.y<bb.y;
}

int   cmp2(node aa,node bb)
{
    int d=direction(a[1],aa,bb);
    if(d>0) return 1;
    return 0;
}

node next_to_top()//距离顶点附近的点
{
    node tn1,tn2;
    tn1=st.top(),st.pop();
    tn2=st.top();
    st.push(tn1);
    return tn2;
}

node st_top()
{
    return st.top();
}

int main()
{
    int i,k,n,x,y;
    node tn1,tn2;
    while(scanf("%d",&n)!=EOF)
    {
        for(i=1;i<=n;i++)
            scanf("%d%d",&x,&y),a[i].x=x,a[i].y=y;

        if( n<3 )
        {
            printf("0\n");
            continue;
        }

        sort(a+1,a+1+n,cmp1);//对点进行位置的排序
        sort(a+2,a+1+n,cmp2);//对点进行极角的排序
```

```
        b[1]=a[1] , b[2]=a[2];

        for(k=2 , i=3;i<=n;i++)
        {
            if(direction(b[1],b[k],a[i])==0)
            {
if(abs( a[i].x-b[1].x )>=abs( b[k].x-b[1].x ) && abs(a[i].y-b[1].y) >= abs(b[k].y-b[1].y) )
                    b[k]=a[i];
            }
            else
                b[++k]=a[i];
        }

        while(!st.empty())
            st.pop();

        st.push(b[1]);
        st.push(b[2]);
        st.push(b[3]);

        for(i=4;i<=k;i++)
        {
            while(1)
            {
                tn1=next_to_top();
                tn2=st_top();
                int d=direction(tn1,b[i],tn2);
                if (d<0)
                    break;
                st.pop();
            }
            st.push(b[i]);//不断地插入点，此时的点就是凸包上的点
        }

        k=st.size();

        for(i=k;i>=1;i--)
        {
            b[i]=st.top();
            st.pop();
        }

        double sum=0;
```

```
        for(i=3;i<=k;i++)
            sum=sum+abs( direction(b[1],b[i],b[i-1]) ); //算出凸包的面积

        printf("%d\n",(int)(sum/100));
    }
    return 0;
}
```

4.4 寻找最近点对

4.4.1 寻找最近点对方案

我们通常将平面上两点 $P_i = (X_i, Y_i)$和 $P_j = (X_j, Y_j)$之间的距离称为欧式距离

$$d_{i,j} = \sqrt{(X_i - X_j)^2 + (Y_i - Y_j)^2}$$

如果 P_i 和 P_j 重合，在这种情况下它们之间的距离 d_{ij} 为 0。

现在我们考虑在 n＞2 个点的集合 Q 中寻找最近点对的问题。这个问题有很大的实用价值，例如一个控制空中或海上交通系统可能需要了解两个最近的交通工具，以检测可能产生的相撞事故。

显然，n 个点的集合 Q 中共有 c(n，2)个点对。最原始的算法是通过计算所有这些点对的距离来找出最近点对。这种算法的效率直接依赖于 n 值，n 愈大，运行时间愈长。为了提高时效，我们将描述一种应用于该问题的分治算法。

算法每次递归调用的输人为点的子集 $P = (P_r..P_R) \subseteq Q$和数组 Y。P 中的所有点按其 Y 坐标单调递增的顺序排列，其编号序列存人数组 Y。同样，我们也得将所有点按 X 坐标单调递增的顺序排列，其编号序列存人数组 X。显然这种排序性质在每次递归调用中是保持不变的，因此我们只要在调用分治程序前进行一次预排序，分别求出数组 X 和数组 Y。

输入为 P、Y 的递归调用首先检查| P |＜3。如果是这样，则计算所有 c (| P |，2)个点对的距离并返回最近的点对及其距离。如果| P |＞3，则递归调用采取如下分治策略。

划分：

找出一条垂直平分线，将点集 P 划分为点数分别为 P / 2 (P 为偶数)或者[P / 2]和[P/2] +1 (P 为奇数)的两个点集 P_l 和 P_R。其中 P_l 上的所有点在垂直平分线上或其左侧；P_R 上的所有点在垂直平分线上或其右侧。数组 Y 被划分为两个数组 Y_l 和 Y_R。类似地，数组 X 也被划分为 X_l 和 X_R。划分后的 X_l 和 Y_l 包含了 P_l 中的点，并分别按 X 坐标和 Y 坐标单调递增的次序进行徘序；X_R 和 Y_R 包含了 P_R 中的点，也按 X 坐标和 Y 坐单调递增的次序进行排序。

解决：

把 P 划分为 P_l 和 P_R 后再进行两次递归调用：

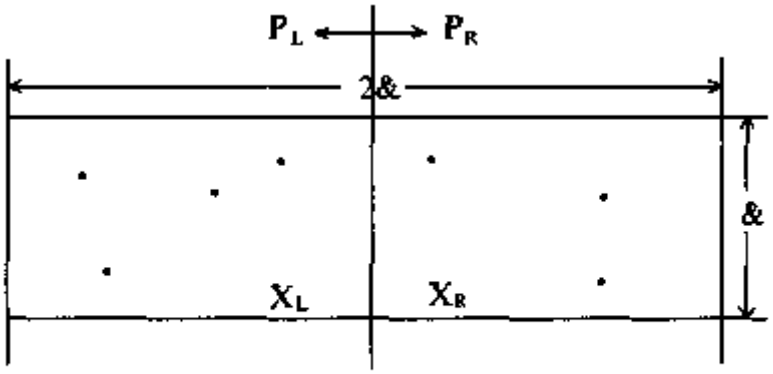
第一次：找出 P_l 中的最近点对，调用的输人为子集 P_l 和数组 Y_l，返回最近点对的距离 $\&_l$ ；

第二次：找出 P_R 中的最近点对，调用的输入为子集 P_R 和数组 Y_R，返回最近点对的距离 $\&_R$ ；

置 $\& = \min(\&_l, \&_R)$ ；

合并：

问题是可能还存在其距离小于＆的点对。如果存在这样的一个点对的话，则点对的一个点在 P_1 中，另一个点在 P_n 中，两个点必定相距垂直平分线＆单位之内，即它们必定处于以垂直平分线为中心、宽度为 2＆的垂直带形区域内。更精确一点讲，由于点对之间的垂直距离也少于＆单位，因此，它们是在以垂直平分线为中心的某个＆*2＆的矩形区域内。



垂直平分线

图 4.4 - 1

为了找出这样的点对(如果存在)，算法要做如下工作：

1．建立一个数组 Y'，Y’仅含 Y 数组中所有在宽度为 2＆的垂直带形区域内的点，这些点按其 Y 坐标单调递增的顺序存储于 Y'；

2．对数组 Y’中的每个点 P，求出 P 到紧随 P 后的 7 个点间的距离，并纪录下 Y’的所有点对中找出的最近点对的距离＆'。为什么对点 P，在 Y’中仅需考虑紧随 P 后的 7 个点呢?

设最近点对为 $X_L \in P_L$，$X_R \in P_R$ 。由于 X_L 和 X_R 只能在某个＆*2＆的矩形区域内出现，且 P_L 中所有点对的距离和 P_R 中的所有点对的距离都至少为＆单位，因此 P 中至多有 8 个点可能位于该＆*2＆矩形内(垂直平分线上的点即可 P_L，亦可属于 P_R)，而 Y’中 X_L 位于 X_R 之前，那么即使 X_L 在 Y’中尽可能早出现而 X_R 尽可能晚出现，X_R 也必定是跟随 X_L 的 7 个位置中的一个。

3．如果＆’＜＆，则垂直带形区域内的确包含比我们根据递归调用所找出的最近距离＆更近的点对，于是返回该点对及其距离＆'；否则就返回递归调用中发现的最近点对及其距离＆。

综上所述，最近点对算法 Nearest（S）流程如下：

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

const int N = 100005;
const double MAX = 10e100, eps = 0.00001;

struct Point
{
```

```
    double x, y;
    int index;
};//点以及点的序号的结构体
Point a[N], b[N], c[N];

double closest(Point *, Point *, Point *, int, int);//二分进行分割和查找
double dis(Point, Point);//点与点之间的距离
int cmp_x(const void *, const void*);
int cmp_y(const void *, const void*);//两种比较
int merge(Point *, Point *, int, int, int);//合并点集
inline double min(double, double);//求最小值

int main()
{
    int n, i;
    double d;

    while (scanf ("%d",&n),n)
    {
        for (i = 0; i < n; i++)
            scanf("%lf%lf", &(a[i].x), &(a[i].y));
        qsort(a, n, sizeof(a[0]), cmp_x);
        for (i = 0; i < n; i++)
            a[i].index = i;
        memcpy(b, a, n *sizeof(a[0]));
        qsort(b, n, sizeof(b[0]), cmp_y);
        d = closest(a, b, c, 0, n - 1);
        printf("%.2lf\n", d);
        scanf("%d", &n);
    }
    return 0;
}

double closest(Point a[],Point b[],Point c[],int p,int q)
{
    if (q - p == 1)
        return dis(a[p], a[q]);
    if (q - p == 2)
    {
        double x1 = dis(a[p], a[q]);
        double x2 = dis(a[p + 1], a[q]);
        double x3 = dis(a[p], a[p + 1]);
        if (x1 < x2 && x1 < x3)
            return x1;
```

```
        else
            if (x2 < x3)
                return x2;
            else
                return x3;
    }
    int i, j, k, m = (p + q) / 2;
    double d1, d2;
    for (i = p, j = p, k = m + 1; i <= q; i++)
        if (b[i].index <= m) c[j++] = b[i];
    //数组 c 左半部保存划分后左部的点, 且对 y 是有序的.
        else c[k++] = b[i];
        d1 = closest(a, c, b, p, m);
        d2 = closest(a, c, b, m + 1, q);
        double dm = min(d1, d2);
        //数组 c 左右部分分别是对 y 坐标有序的, 将其合并到 b.
        merge(b, c, p, m, q);
        for (i = p, k = p; i <= q; i++)
            if (fabs(b[i].x - b[m].x) < dm)
                c[k++] = b[i];
        //找出离划分基准左右不超过 dm 的部分, 且仍然对 y 坐标有序.
        for (i = p; i < k; i++)
            for (j = i + 1; j < k && c[j].y - c[i].y < dm; j++)
            {
                double temp = dis(c[i], c[j]);
                if (temp < dm) dm = temp;
            }
            return dm;
}

double dis(Point p, Point q)
{
    double x1 = p.x - q.x, y1 = p.y - q.y;
    return sqrt(x1 *x1 + y1 * y1);
}

int merge(Point p[], Point q[], int s, int m, int t)
{
    int i, j, k;
    for (i=s, j=m+1, k = s; i <= m && j <= t;)
    {
        if (q[i].y > q[j].y)
            p[k++] = q[j], j++;
        else
```

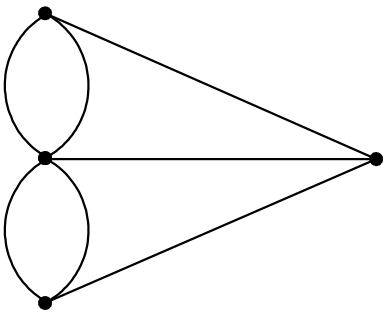
```
                p[k++] = q[i], i++;
        }
        while (i <= m)
            p[k++] = q[i++];
        while (j <= t)
            p[k++] = q[j++];
        memcpy(q + s, p + s, (t - s + 1) *sizeof(p[0]));
        return 0;
}

int cmp_x(const void *p, const void *q)
{
        double temp = ((Point*)p)->x - ((Point*)q)->x;
        if (temp > 0)
            return 1;
        else
            if (fabs(temp) < eps)
                return 0;
            else
                return - 1;
}

int cmp_y(const void *p, const void *q)
{
        double temp = ((Point*)p)->y - ((Point*)q)->y;
        if (temp > 0)
            return 1;
        else
            if (fabs(temp) < eps)
                return 0;
            else
                return - 1;
}
inline double min(double p, double q)
{
        return (p > q) ? (q): (p);
}
```

第五章 图算法

对一个已列表给出各顶点及边(包括标有权的边)的图进行搜索，实际上就是以一种系统的方式沿着图的边访问所有顶点。一个图形的搜索算法可以使我们发现图的许多结构信息。许多有关图的算法开始都通过搜索输入图来获取结构信息；另外还有一些图形算法也是由基本的图形搜索算法经过简单扩充而成的。因此图形搜索技术是图形算法领域的核心。

5.1 引言

关于图论最早的论文可以追溯到 1736 年，当时欧拉（Leonhard Eular）发表了一篇论文，给出了图论中的一个一般性的理论，其中包括现在被称为 **Königsberg 七桥问题**的解。其背景是一个非常有趣的问题：在 Königsberg 城郊的 Pregerl 河上有两个小岛，小岛和河两岸的陆地由 7 座桥相连（如图 6.1（a）），问题是如何从河岸或岛上的某一个位置出发，能经过 7 座桥正好各一次，最后回到出发地。有人尝试了很多次，始终没能成功。Euler 在论文中，给出了该问题的数学模型，用 4 个点代表 4 个被河隔开的陆地（两岸和岛屿），把桥表示为连接两个陆地之间的边，则得到图 6.1（b）所示的图，从而问题变为如何从图中的某个点出发，经过所有的边正好一次，最后回到这个点。在研究这个图的基础上，Euler 在论文中证明了该七桥问题无解，并且给出了一些规律性的理论。把实际问题抽象成点和边构成的图进行研究，标志着图论研究的开始。



（a）

图 5.1 Königsberg 七

图论中几个重要的结论也是在 19 世纪得到的，但图论 E 20 世纪 20 年代以后，其主要原因是它在许多不同领域内的应用，包括计算机科学、化学、运筹学、电子工程、语言学和经济学等。

5.2 最小生成树

5.2.1 Prim 算法

prim 算法求最小生成树是数据结构中图的一种重要应用,它的要求是从一个带权无向完全图

中选择 n－1 条边并使这个图仍然连通(也即得到了一棵生成树),同时还要考虑使树的权最小。

5.2.1.1 适用范围

MST(Minimum Spanning Tree,最小生成树)，无向图(有向图的是最小树形图)，多用于稠密图（点少边多）。

5.2.1.2 算法描述

```
void    relax(int u,int v);              //松弛操作
{
   If( w<dis[v])
   {
         pre[v]=u;
         dis[v]=w;
   }
}
```

初始化：dis[v]=maxint(v∈V,v≠s); dis[s]=0; pre[s]=s; S={s};tot=0

For i:=1 to n

1.取顶点 v∈V-S 使得 W(u,v)=min{W(u,v)|u∈S,v∈V-S,(u,v)∈E}

2.S=S+{v};tot=tot+W(u,v);输出边(u,v)

3.For V-S 中每个顶点 v do Relax(u,v,Wu,v)

算法结束：tot 为 MST 的总权值

此算法的精妙之处在于对求权值最小的边这一问题的分解。按照常规的思路,这一问题应该这样解决:分别从集合 V－U 和 U 中取一顶点,从邻接矩阵中找到这两个顶点行成的边的权值,设 V－U 中有 m 个顶点,U 中有 n 个顶点,则需要找到 m＊n 个权值,在这 m＊n 个权值中,再查找权最小的边。循环每执行一次,这一过程都应重复一次，相对来说计算量比较大。而本算法则利用了变量tree中第k＋1到第 n－1 号元素来存放到上一循环为止的一些比较结果,如以第 k＋1 号元素为例,其存放的是集合 U 中某一顶点到顶点 tree.en 的边,这条边是到该点的所有边中权值最小的边,所以,求权最小的边这一问题,通过比较第 k＋1 号到第 n－1 号元素的权的大小就可以解决,每次循环只用比较 n－k－2 次即可，从而大大减小了计算量。

5.2.1.3 算法优化

使用二叉堆(Binary Heap)来实现每步的 DeleteMin(ExtractMin)操作

算法复杂度从 O(V^2)降到 O((V+E)log V)。推荐对稀疏图使用。

使用 Fibonacci Heap 可以将复杂度降到 O(E+V log V)，但因为编程复杂度太高，不推荐在信息学竞赛及大学生程序设计比赛中使用。

5.2.1.4 应用举例

Jungle Roads(ZJU1061)

题目简述：村庄不超过 26 个，以大写字母命名。线路不超过 75 条，每一条的维修费用不超过 100。每个村庄至多与 15 个村庄相连。求可连通线路的最小费用。

思路：最小生成树，

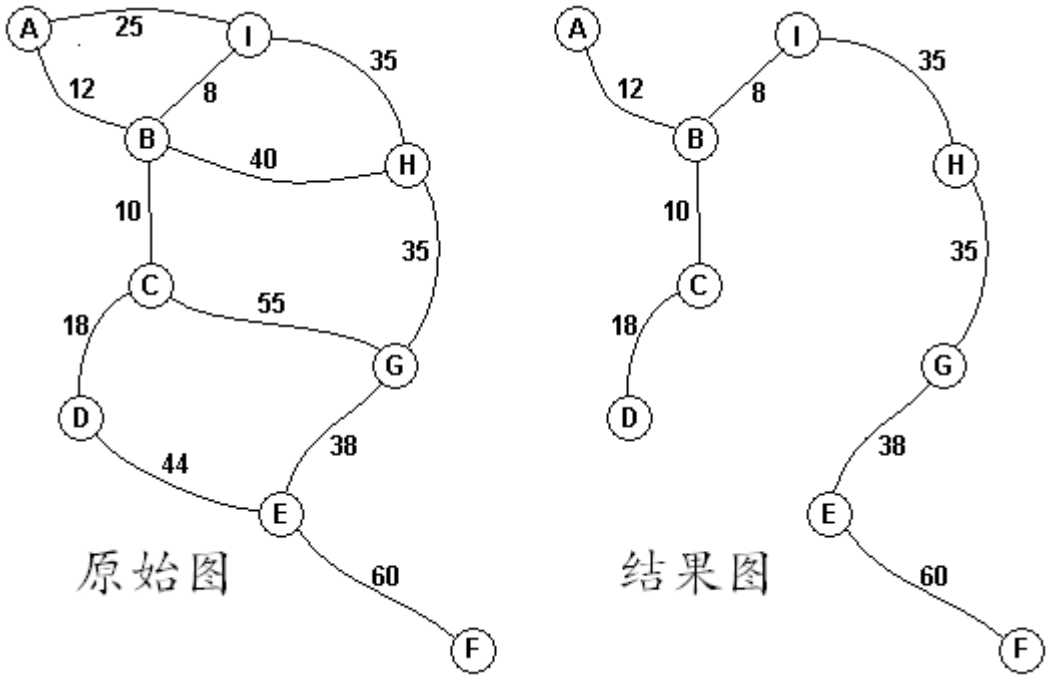


图 5.2

代码：

```
#include <iostream.h>
#define MAXINT 1000000

int iN;
int Graph[26][26];

bool Input()
{
    cin >> iN;
    if(!iN)
        return false;
    int i, j, a, b, value;
    char ch1, ch2;
    for (i = 0; i < 26; ++i)
    {
        for (j = 0; j < 26; ++j)
        {
            Graph[i][j] = MAXINT;
        }
    }
    for (i = 0; i < iN -1; ++i)
    {
        cin >> ch1 >> j;
        a = ch1-'A';
        while(j--)
```

```
            {
                cin >> ch2 >> value;
                b = ch2 - 'A';
                Graph[a][b] = Graph[b][a] = value;
            }
        }
        return true;
}

int Calculate()
{
    int distance[26];
    bool reached[26];
    int i, j;
    int Result = 0;
    for (i = 0; i < iN; ++i)
    {
        distance[i] = Graph[0][i];
        reached[i] = false;
    }
    reached[0] = true;
    for (i = 0; i < iN-1; ++i)
    {
        int min = MAXINT, v = -1;
        for (j = 0; j < iN; ++j)
            if(!reached[j] && distance[j] < min)
            {
                min = distance[j];
                v = j;
            }
            Result += min;
            reached[v] = true;
            for (j = 0; j < iN; ++j)
                if(!reached[j] && Graph[j][v] < distance[j])
                    distance[j] = Graph[j][v];
    }
    return Result;
}

int main()
{
    while(Input())
    {
        cout << Calculate() << endl;
```

```
    }
    return 0;
}
```

5.2.2 Kruskal 算法

K r u s k a l 算法每次选择 n- 1 条边，所使用的贪婪准则是：从剩下的边中选择一条不会产生环路的具有最小耗费的边加入已选择的边的集合中。注意到所选取的边若产生环路则不可能形成一棵生成树。K r u s k a l 算法分 e 步，其中 e 是网络中边的数目。按耗费递增的顺序来考虑这 e 条边，每次考虑一条边。当考虑某条边时，若将其加入到已选边的集合中会出现环路，则将其抛弃，否则，将它选入。

5. 2. 2. 1 适用范围

MST(Minimum Spanning Tree,最小生成树)，无向图(有向图的是最小树形图)，多用于稀疏图（点多变少），边已经按权值排好序给出。

5. 2. 2. 2 算法描述

基本思想：每次选不属于同一连通分量(保证无圈)且边权值最小的 2 个顶点，将边加入 MST，并将所在的 2 个连通分量合并，直到只剩一个连通分量。

```
将边按非降序排列(Quicksort,O(E log E))
While 合并次数少于|V|-1
    取一条边(u,v)(因为已经排序，所以必为最小)
    If u,v 不属于同一连通分量 then
        合并 u,v 所在的连通分量
        输出边(u,v)
    合并次数增 1；tot=tot+W(u,v)
算法结束：tot 为 MST 的总权值
```

5. 2. 2. 3 算法分析

检查 2 个顶点是否在同一连通分量可以使用并查集实现(连通分量看作等价类)。

我们可以看到，算法主要耗时在将边排序上。如果边已经按照权值顺序给出，那太棒了……

另外一种可以想到的实现方法为：O(n)时间关于边权建二叉小根堆；每次挑选符合条件的边时使用堆的 DelMin 操作。这种方法比用 Qsort 预排序的方法稍微快一些，编程复杂度基本一样。附程序。

如果边权有一定限制，即<=某常数 c，则可以使用线性时间排序以获得更好的时间效率 。

5.3 最短路算法

5.3.1 Dijkstra 算法

Dijkstra(迪杰斯特拉)算法是典型的单源最短路径算法，用于计算一个节点到其他所有节点的最短路径。主要特点是以起始点为中心向外层层扩展，直到扩展到终点为止。Dijkstra 算法是很有代表性的最短路径算法，在很多专业课程中都作为基本内容有详细的介绍，如数

据结构，图论，运筹学等等。Dijkstra 一般的表述通常有两种方式，一种用永久和临时标号方式，一种是用 OPEN, CLOSE 表的方式，这里均采用永久和临时标号的方式。注意该算法要求图中不存在负权边。

5.3.1.1 适用范围

单源最短路径(从源点 s 到其它所有顶点 v);有向图&无向图(无向图可以看作(u,v),(v,u)同属于边集 E 的有向图)，所有边权非负(任取(i,j)∈E 都有 Wij≥0);

5.3.1.2 算法描述

结定一个带正权的有向图 D（V，E）(即每条边（U，V）∈E，W(U，V≥0)和起始顶点 V0。求从 V0 到 D 中其余各顶点间的最短路径。

创建两个表，OPEN, CLOSE。

OPEN 表保存所有已生成而未考察的节点，CLOSED 表中记录已访问过的节点。

1． 访问路网中里起始点最近且没有被检查过的点，把这个点放入 OPEN 组中等待检查。

2． 从 OPEN 表中找出距起始点最近的点，找出这个点的所有子节点，把这个点放到 CLOSE 表中。

3． 遍历考察这个点的子节点。求出这些子节点距起始点的距离值，放子节点到 OPEN 表中。

4． 重复 2，3，步。直到 OPEN 表为空，或找到目标点。

5.3.1.3 算法优化

使用二叉堆(Binary Heap)来实现每步的 DeleteMin(ExtractMin，即算法步骤 b 中第 1 步)操作，算法复杂度从 O(V^2)降到 O((V+E)log V)。推荐对稀疏图使用。

使用 Fibonacci Heap(或其他 Decrease 操作 O(1),DeleteMin 操作 O(logn)的数据结构)可以将复杂度降到 O(E+V log V)；如果边权值均为不大于 C 的正整数，则使用 Radix Heap 可以达到 O(E+V log C)。但因为它们编程复杂度太高，不推荐在信息学竞赛中使用。

5.3.1.4 应用举例

昂贵的聘礼(PKU1062)

题目简述：年轻的探险家来到了一个印第安部落里。在那里他和酋长的女儿相爱了，于是便向酋长去求亲。酋长要他用 10000 个金币作为聘礼才答应把女儿嫁给他。探险家拿不出这么多金币，便请求酋长降低要求。酋长说："嗯，如果你能够替我弄到大祭司的皮袄，我可以只要 8000 金币。如果你能够弄来他的水晶球，那么只要 5000 金币就行了。"探险家就跑到大祭司那里，向他要求皮袄或水晶球，大祭司要他用金币来换，或者替他弄来其他的东西，他可以降低价格。探险家于是又跑到其他地方，其他人也提出了类似的要求，或者直接用金币换，或者找到其他东西就可以降低价格。不过探险家没必要用多样东西去换一样东西，因为不会得到更低的价格。探险家现在很需要你的帮忙，让他用最少的金币娶到自己的心上人。另外他要告诉你的是，在这个部落里，等级观念十分森严。地位差距超过一定限制的两个人之间不会进行任何形式的直接接触，包括交易。他是一个外来人，所以可以不受这些限制。但是如果他和某个地位较低的人进行了交易，地位较高的的人不会再和他交易，他们认为这样等于是间接接触，反过来也一样。因此你需要在考虑所有的情况以后给他提供一个最好的方案。

为了方便起见，我们把所有的物品从 1 开始进行编号，酋长的允诺也看作一个物品，并且编号总是 1。每个物品都有对应的价格 P，主人的地位等级 L，以及一系列的替代品 Ti 和该替代品所对应的"优惠"Vi。如果两人地位等级差距超过了

M，就不能"间接交易"。你必须根据这些数据来计算出探险家最少需要多少金币才能娶到酋长的女儿。

思路：

1.酋长的级别不一定是最大的。

2.级别差超过 m 的两个人不能以任何形式接触，不单指相邻的两人。比如说 A 级别 1，B 级别 2，C 级别 3，级别差要求是 1，那么先跟 A 交易，再跟 B 交易，然后跟 C 交易这种情况是不能存在的。亦即交易中级别最大与最小不超过 m。

代码：

```
const int inf=0x7fffffff;
int M,N,dmax,dmin,tmin;
int c[105][105];
int degree[105];
int dist[105];
bool s[105],use[105];

bool In(int a)
{
    return abs(dmax-a)<=M&&abs(dmin-a)<=M;
}

int dij()
{
    int i,j,k;
    for(i=0;i<=N;i++) {dist[i]=c[0][i];s[i]=0;}
    s[0]=1;use[0]=1;

    for(i=1;i<=N;i++)
    {
        tmin=inf;
        for(j=1;j<=N;j++)//寻找当前未被纳入符合最短距离的结点
        {
            if(!s[j]&&use[j]&&tmin>dist[j])
            {
                k=j;
                tmin=dist[j];
            }
        }
        s[k]=1;//将 k 号结点纳入最短路
        for(j=1;j<=N;j++)//对与 k 有关的结点进行更新
        {
            if(!s[j]&&use[j]&&dist[k]<inf&&c[k][j]<inf&&dist[j]>dist[k]+c[k][j])
                dist[j]=dist[k]+c[k][j];
```

```
            }
        }
        return dist[1];
}
void solove()
{
        int i,j,m,an,t;an=inf;
        for(i=0;i<=N;i++)
        {
                m=degree[i];dmax=m;dmin=m-M;//枚举最大等级,确实等级范围
                for(j=0;j<=N;j++)
                {
                        if(In(degree[j])) use[j]=1;
                        else use[j]=0;
                }
                t=dij();
                if(an>t) an=t;//记录下最优值
        }
        cout<<an<<endl;
}
int main()
{
        int i,j,k,m,n;
        while(cin>>M>>N)
        {
                for(i=0;i<=N;i++) for(j=0;j<=N;j++) c[i][j]=inf;
                for(i=1;i<=N;i++)
                {
                        cin>>c[0][i];
                        cin>>degree[i]>>k;
                        for(j=1;j<=k;j++)
                        {
                                cin>>m>>n;
                                c[m][i]=n;
                        }
                }
                degree[0]=degree[1];
                dmax=dmin=degree[0];
                solove();
        }
        return 0;
}
```

5.3.2 Bellman-Ford 算法

Bellman-Ford 算法与另一个非常著名的 Dijkstra 算法一样，用于求解单源点最短路径问题。Bellman-ford 算法除了可求解边权均非负的问题外，还可以解决存在负权边的问题（意义是什么，好好思考），而 Dijkstra 算法只能处理边权非负的问题，因此 Bellman-Ford 算法的适用面要广泛一些。但是，原始的 Bellman-Ford 算法时间复杂度为 O（VE）,比 Dijkstra 算法的时间复杂度高，所以常常被众多的大学算法教科书所忽略，就连经典的《算法导论》也只介绍了基本的 Bellman-Ford 算法，在国内常见的基本信息学奥赛教材中也均未提及，因此该算法的知名度与被掌握度都不如 Dijkstra 算法。

5.3.2.1 适用范围

单源最短路径(从源点 s 到其它所有顶点 v);有向图&无向图(无向图可以看作(u,v),(v,u)同属于边集 E 的有向图);边权可正可负(如有负权回路输出错误提示);差分约束系统;

5.3.2.2 算法描述

Bellman-Ford(G,w,s) ： boolean //图 G ，边集 函数 w ，s 为源点
1 for each vertex v ∈ V（G） do //初始化 1 阶段
2 d[v] ←+∞
3 d[s] ←0; //1 阶段结束
4 for i=1 to |v|-1 do //2 阶段开始，双重循环。
5 for each edge（u,v） ∈E(G) do //边集数组要用到，穷举每条边。
6 If d[v]> d[u]+ w(u,v) then //松弛判断
7 d[v]=d[u]+w(u,v) //松弛操作 2 阶段结束
8 for each edge（u,v） ∈E(G) do
9 If d[v]> d[u]+ w(u,v) then
10 Exit false
11 Exit true

5.3.2.3 算法分析

算法时间复杂度 O(VE)。因为算法简单，适用范围又广，虽然复杂度稍高，但因为它实践中的效果不错，仍不失为一个很实用的算法。

5.3.2.4 算法实现

```
#define MAX 100
#define MAXNUM 1000000
//MAXNUM 为一个绝对大的数,必须大于起点到任何一点的最长路径
typedef struct graphnode
{
        int vexnum;
        int arcnum;
        double gra[MAX][MAX];
}Graph;
//Graph 用于存储图的相关信息，邻接矩阵存储
//gra[i][j]存储的信息为起点为点 i 终点为点 j 的边的长度，为 0 则这两点之间没有边
int arc[MAX][MAX];
//arc 矩阵用于存储边的信息，第 i 条边的起点和重点分别存在 arc[i][0]和 arc[i][1]中
double dis[MAX];
```

```
//dis 矩阵用于存储当前起点到各个点间的最短路径，如无路径其值为 MAXNUM
void bellman(Graph *G)
{
        int i,j;
        bool sign;
        for(i=0;i<G->vexnum;i++)
                dis[i]=MAXNUM;
        //开始搜索前 dis 数组全部赋值为 MAXNUM
        sign=true;
        dis[0]=0;
        //起点到自己的最短路径为 0
        for(i=1;i<G->vexnum&&sign;i++)
        {
                sign=false;
                for(j=0;j<G->arcnum;j++)
                {
if(dis[arc[j][0]]<MAXNUM&&dis[arc[j][1]]>dis[arc[j][0]]+G->gra[arc[j][0]][arc[j][1]])
                        {
                                //如果边 j 起点可达
                                //并且经过这条边的路径小于 dis[边 j 的终点]存储的值
                                //则把较小的值赋值给 dis[边 j 的终点]
                                dis[arc[j][1]]=dis[arc[j][0]]+G->gra[arc[j][0]][arc[j][1]];
                                sign=true;
                        }
                }
        }
        //当搜索到所有的边都不能在当前状况下提供一条更短的路径时，搜索结束
        //如果 dis[i]!=MAXNUM,则 dis[i]的值就是起点到该点的最短路径
        //如果 dis[i]==MAXNUM,则起点与该点不可达
}//利用该算法求无向图的最短路径问题时，应把无向图转化为有向图
```

5.3.3 Floyd-Warshall 算法

Floyd-Warshall 算法是解决任意两点间的[最短路径](#)的一种算法，可以正确处理[有向图](#)（Directed Graph）或负数的代价(negtive cost)的最短路径问题（不能有负环）。Floyd-Warshall 算法的时间复杂度为 O(N^3)。

5.3.3.1 适用范围

APSP(All Pairs Shortest Paths)，稠密图效果最佳边权可正可负。

5.3.3.2 算法描述

```
初始化：dis[u,v]=w[u,v]
For k:=1 to n
    For i:=1 to n
        For j:=1 to n
```

If dis[i,j]>dis[i,k]+dis[k,j] Then

Dis[I,j]:=dis[I,k]+dis[k,j];

算法结束：dis 即为所有点对的最短路径矩阵

5.3.3.3 算法分析

此算法简单有效，由于三重循环结构紧凑，对于稠密图，效率要高于执行|V|次 Dijkstra 算法。时间复杂度 O(n^3)。

考虑下列变形：如(I,j)∈E 则 dis[I,j]初始为 1,else 初始为 0，这样的 Floyd 算法最后的最短路径矩阵即成为一个判断 I,j 是否有通路的矩阵。

更简单的，我们可以把 dis 设成 boolean 类型，则每次可以用“dis[I,j]:=dis[I,j]or(dis[I,k]and dis[k,j])”来代替算法描述中的下划线部分，可以更直观地得到 I,j 的连通情况。

5.3.3.4 应用举例

Fiber Network （ZJU1967）

题意简述: 给出一个有向图，从一点运送东西到另一个点要通过中介公司，公司名字是 a 到 z 一个字符, 若干询问，从某点到某点，找出能运送的公司，中途不换公司

思路：最多只有 26 个字符，所以可以看成是 26 个互不相干的图，然后再 26 次 floyd。 复杂度是 O(26*200^3).

此题有多组数据，每组数据的第一行是 n---结点数，接下来一直读到 0 0 为止描述了一些有向边，每条边的权值是一些字母，表示这条边可以由这些字母代表的公司导通，接下来又一些询问，到 0 0 结束，询问从 i 到 j 的每条路径中，哪些公司是必须用到的 。

如下图：

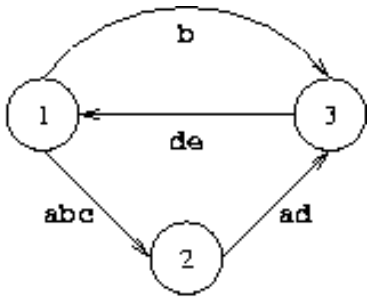
询问 1 → 3， 1 → 2 → 3，必经 a

1 → 3，必经 b，所以输出 ab

询问 2 → 1， 2 → 3 → 1 必经 d，除此没有其他路径，输出 d

如果不存在这样的公司，输出-

每组数据后有一个空行。



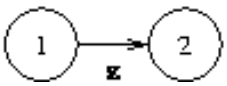


图 5.3

代码：

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

#define MAXN 201
#define MAXS 27
```

```
int n;
int node[MAXN][MAXN];
int power[MAXS];

void Init(){
    int i;
    power[0] = 1;
    for(i = 1;i < 26; ++i)
        power[i] = power[i - 1] << 1;
}

bool Input(){
    scanf("%d",&n);
    if(n == 0) return false;
    memset(node,0,sizeof(node));
    int i,j,k;
    int a,b;
    char str[MAXS];
    while(scanf("%d%d",&a,&b) && (a != 0 || b != 0)){
        scanf("%s",str);
        for(i = 0;i < (int)strlen(str); ++i)
            node[a][b] |= power[str[i] - 'a'];
    }
    for(k = 1;k <= n; ++k){
        for(i = 1;i <= n; ++i) if(i != k){
            for(j = 1;j <= n; ++j) if(k != j){
                node[i][j] |= node[i][k] & node[k][j];
            }
        }
    }
    bool flag;
    while(scanf("%d%d",&a,&b) && (a != 0 || b != 0)){
        flag = false;
        for(i = 0;i < 26; ++i){
            if((node[a][b] & power[i]) != 0){
                printf("%c",'a' + i);
                flag = true;
            }
        }
        if(flag == false)
            printf("-");
        printf("\n");
    }
```

```
    return true;
}

int main(){
    Init();
    while(Input())
        printf("\n");
    return 0;
}
```

5.4 无向图的连通性

5.4.1 无向连通图割点和桥

设 G＝(V，E)是一无向连通图。如果去掉 G 的某顶点后，G 就不是连通图，这样的顶点称为割点，如果去掉某一边就不成为连通图，这样的边称为桥。试用深度优先搜索来确定一个无向连通图的所有割点和桥。

5.4.1.1 适用范围

无向连通图，求边连通分量以及双连通分量。

5.4.1.2 算法描述

若我们对无向图实施深度优先搜索后找出一个割点 V，则 V 必然具备下述特征：

1．若 V 是深度优先树的根，则该根至少有两个子女(见图 5．3—4(a))；

2．若 V 不是深度优先树的根，则 V 有一个后裔 U，从 U 或 U 的后代到 V 的祖先点之间不存在反向边，见图 53—4(b)。

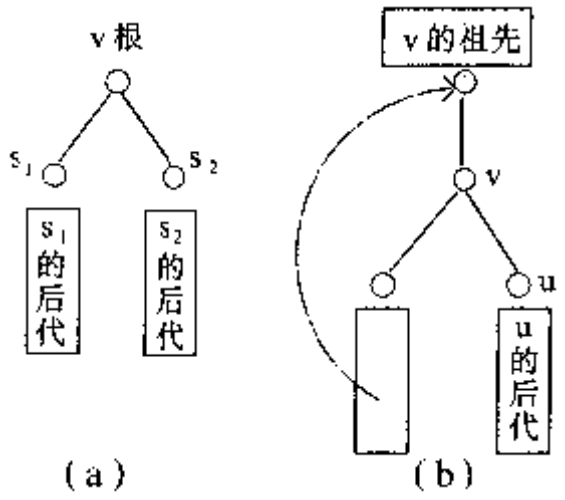


图 5.4

证明略去，读者可从图 5.3—4 中体会灵活的思路。图例中 V 是割点。

我们可以直接利用深度优先搜索过程中得出的结构信息：P(前趋表)，b(度数表)，L(标号函数表)，d(发现时刻表)进行割点的识别。为了避免对割点重复输出，还在算法中增设了割点表 a。若 a[V]＝0，表示当前还未判断出 V 是割点；若 a[V]＝1，表示已判断出 V 是割点，应避免重复输出。

显然，对应于割点的第 1 个特征来说，若(P[V]＝0)and(b[V]>1)，则根结点 V 为割点；对应于割点的第 2 个特征来说，若(P[P[V]]>0)and(a[P[V]]＝0)and(L[V]≥d[P[V]]，则非根结点 P[V]为割点。

同样我们也可以通过深度优先搜索得到的结构信息找出无向连通图的桥。所谓桥，是指那些不属于任何简单回路的边。在深度优先搜索中，当搜索到一条树枝边（U，V）时发现 V 及其后代不存在一条连接 U 的祖先点的返回边，即 LOW[V]>d[U]，则说明(U,V)不属于图的任何简单回路，(U,V)即为桥。

5.4.1.3 应用举例

Network(PKU1311)

题目简述：TLC 电话线路公司正在新建一个电话线路网络。他们将一些地方（用 1 到 N 的整数标明，任何 2 个地方的号码都不相同）连接起来。

这些线路是双向的，连接 2 个地方，并且每个地方电话线路都是连接到一个电话交换机。每个地方都有一个电话交换机。从每个地方都可以达到其他一些地方（如果有线路连接的话），然后这些线路不一定必须是直接连接的，也可以是通过几个电话交换机。但是有时会因为电力不足导致某个地方的交换机不能工作。TLC 的官员意识到一旦出现这种情况（在某个地方的交换机不工作，即这个节点与其他节点之间的线路都断开了），除了这个出现故障的地方是不可达外，还可能导致其他一些（本来连通的）地方也不再连通。我们称这个地方为关键点。

现在 TLC 的官员努力想写一个程序来找到关键节点的数目。请帮助他们。

输入包括多个 blocks，每个 blocks 描述了一个网络。每个 block 的第一行是一个整数 N，代表节点的数目(N<100)。

接下来，每个网络至多有 N 行信息，每行包括一个地方的号码，以及有直接线路通往其他地方的号码。这些行（至多 N 行）信息完整地描述了网络，也就是，该网络中每条直接连接 2 个地方的线路将会至少出现在某一行中。每行中的数字都用空格隔开。每个网络的信息用单独一行（只包括数字 0）表示结束。 N=0 表示输入信息结束。

对每个网络，输出起关键节点的个数

思路：

首先，对于单独的一个点，或者 2 个点，我们不予考虑，因为我们没法认定。

分情况：

如果待确定的点不是搜索树的根节点，那么关键点可以认为是这样的点 V，他的子孙节点中存在一个节点 P，从 P 出发继续向下拓展（也就是不再走已经走过的边，而是向下兜圈子，然后通过某个后向边回到前面已经访问过的点，这里可以看出后向边的价值，后向边在 dfs 中是不允许扩展的，但是我们说的“兜回到祖先”就是通过某个后向边实现的）dfs 不能达到 V 的祖先，事实上，只要有任意一个子孙节点符合这个特性，那么 v 就是关键点，因为原本子孙和祖先是连通的，删除它之后，他的子孙至少无法走到他的祖先（包括它的兄弟姐妹。。。）。

如果是根节点，他如果只有一个子树，那么他不是关键点，如果有多于一个的子树，那么他必然是关键点。

具体实现是这样的，我们定义一个点上的函数 dfn(v)，表示 v 这个点在 dfs 过程中第一次访问的顺序，然后，为了达到上面的定义，我们再定义一个点上的函数 low(v)，low(v)表示从 v 出发，继续向下拓展搜索，所能达到的最早的点，也就是 dfn 最小的

点。显然，low(v)有：low(v)=min(dfn(v),low(s),dfn(w))。其中，s 是所有儿子节点，w 是 v 的后向边

割顶：连通图 G 的一个顶点子集 V，如果删除这个顶点子集和它所附带的边后，图便不再连通。则称 V 是 G 的割顶集。最小割顶集中顶点的个数，称为 G 的连通度。连通度等于 1 时，割顶集中的那个顶点叫做割顶。

规定不连通图的边连通度为 0；完全图的边连通度为总顶点数－1；

连通图的两个特征：

1 连通度<=边连通度<=顶点数。

2 顶点数大于 2 的 2 连通图的充分必要条件是任两个顶点在一个圈上。

块的概念：

没有割点的连通子图，这个子图中的任何一对顶点之间至少存在两条不相交的路径，或者说要使两个站点同时发生故障，至少两个站点同时发生故障，这种二连通分支称为块．显然各个块之间的关系有如下两种：

1 互不连接。

2 通过割顶连接(割顶可以属于不同的块，也可以两个块公有一个割顶)。

引申：无向图寻找块，关键是找割顶．

满足是割顶的条件：

1 如果 u 不是根，u 成为割顶的充要条件：当且仅当存在 u 的一个儿子顶点 s,从 s 或者 s 的后代点到 u 的祖先点之间不存在后向边．

2 如果 u 是根，则 u 成为割顶当且仅当它不止有一个儿子点．

怎样求割顶：

引入一个标号函数；low(u)=min{dfn(u),low(s),dfn(w)}； s 是 u 的一个儿子，(u,w)是后向边。显然 low(u)值是 u 或者 u 的后代所能追溯到的最早(序号小)的祖先序号．

利用标号函数 low，分析求割顶的步骤：

顶点 u 不为根且为割顶的条件是当且仅当 u 有一个儿子 s，使得 low(s)>=dfn(u),即 s 和 s 的后代不会追溯到比 u 更早的祖先点．

low(u)的计算步骤：

1 low(u)=dfn(u); //u 在 dfs 过程中首次被访问

2 low(u)=min{low(u),dfn(w)} //检查后向边(u,w)时

3 low(u)=min{low(u),low(s)} //u 的儿子 s 的关联边被检查时

注意：对任何顶点 u 计算 low(u)的值是不断修改的，只有当以 U 为根的 dfs 子树和后代的 low 值,dfn 值全部出现以后才停止。

代码：

```
#include <stdio.h>
#define maxn 30
typedef struct
{
    int k,l;
}L;
L l[maxn];
int g[maxn][maxn];
int st[maxn];
int n,p,x,i;
int k[maxn];
```

```
int len;
int min(int a,int b)
{
	return a<b?a:b;
}
void push(int a)
{
	p++;
	st[p]=a;
}
int pop()
{
	 return st[p--];
}
void init()
{
	memset(l,0,sizeof(l));
	p=0;push(1);
	i=1;
	len=-1;
	l[1].k=1;l[1].l=1;
	memset(k,-1,sizeof(k));
	x=0;
}
bool ink(int a)
{
	for(int i=0;i<len;i++){
		if(k==a) return true;
	}
	return false;
}
void algorithm(int v)
{
	int u;
	for(u=1;u<=n;u++){
		if(g[v]>0){
			if(l.k==0){
				i++;
				l.k=i;
				l.l=i;
				push(u);
				algorithm(u);
				l[v].l=min(l[v].l,l.l);
```

```
                if(l.l>=l[v].k){
                        if(v!=1 && !ink(v)){
                        len++;
                        k[len]=v;
                    }
                    if(v==1) x++;
                    while(st[p]!=v){

                        printf("%d   ",pop());
                    }
                }
            }
            else l[v].l=min(l[v].l,l.k);
        }
    }
}
int main()
{
    algorithm(1);
    if(x>1){
    len++;
    k[len]=1;
    }
}
```

5.4.2 欧拉图问题

能够遍历完所有的边而不能有重复的图称为欧拉图。含有图中每一条边的路径称作欧拉路径，闭合的欧拉路径称作欧拉回路。

5.4.2.1 适用范围

一个图是欧拉图的充要条件：

1）每个顶皆为偶次；或

2）该图可以划分为不相交的圈的并集。

一个图中有欧拉路径的充要条件是该图中有零个或两个奇次顶。

5.4.2.2 算法描述

通过上面这两个定理，我们可以准确地判断给定的图是不是欧拉图。若该图中有两个奇次顶，这两个奇次顶其中一个是欧拉路径的起点，另一个是终点，这个图亦不会有欧拉回路 。这些名字中的“欧拉”是因为欧拉解决了柯尼斯堡七桥问题，并发现了一笔画所需的性质。

欧拉路径：要找出欧拉路径，就要循环地找出出发点。按以下步骤：

任取一个起点,开始下面的步骤

如果该点没有相连的点，就将该点加进路径中然后返回。

如果该点有相连的点，就列一张相连点的表然后遍历它们直到该点没有相连的点。(遍历一个点，删除一个点）

处理当前的点,删除和这个点相连的边, 在它相邻的点上重复上面的步骤,把当前这个点

加入路径中。

欧拉回路：如下图，每条边经过一次且仅一次的称为欧拉回路(euler cycle, euler circuit).

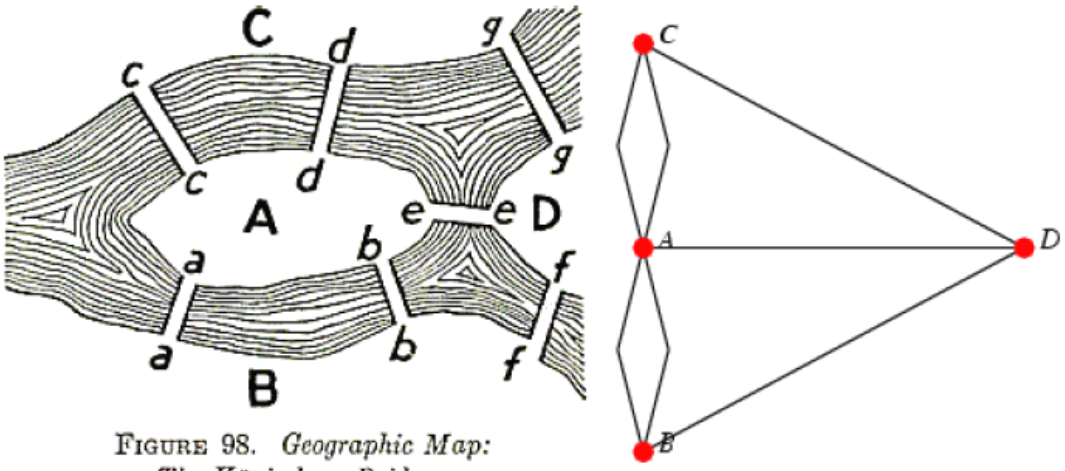


FIGURE 98. *Geographic Map: The Königsberg Bridges.*

图 5.5

算法：从一个点 u 出发 DFS，每次标记边(u,v)和(v,u)，递归调用 EulerRoute(v)，然后把(u,v)放到栈中。

邮递员问题、特殊的中国邮路问题、字典序最小欧拉回路等都是欧拉回路的问题。以下例题仅就欧拉路径进行解答。

5.4.2.3 应用举例

Play on Words (ZJU2016)

题目简述：以首尾单词为点，连一条有向边，构成有向图，然后判断是否是欧拉路。

思路：把所有的联系分堆，看是否可以合并

欧拉路的判断。

性质，如果是回路，对于所有的点，都有入度等于出度。

如果不是回路，则其中有两个点 v 和 t，分别为起点和终点，他们的出入度相差 1。

代码：

```
#include<stdio.h>
#define cham 26
char shuru[100000][2];
int visited[27][2],vtxnu,a[27][27],youxiang[27][27];/*图所用变量*/
int N;
void ru();
void init();
int liantong();
int success();
void dft(int index);

int main()
{
   int T;
   int count;
   scanf("%d",&T);
```

```
    {
        for(count=0;count<T;count++)
        {
            scanf("%d",&N);
            ru();
            init();/*完成邻接矩阵*/
            if(!liantong())
            {
                printf("The door cannot be opened.\n");
                continue;
            }
            if(success())
                printf("Ordering is possible.\n");
            else
                printf("The door cannot be opened.\n");
        }
    }
    return 0;
}
void ru()
{
    int i,j;
    char s;
    scanf("%c",&s);
    for(i=0;i<N;i++)
    {
        scanf("%c",&shuru[i][0]);
        for(j=1;1;j++)
        {
            scanf("%c",&s);
            if(s=='\n')
                break;
            shuru[i][1]=s;
        }
    }
}

void init()
{
    int i,j,k;
    for(i=0;i<26;i++)
    {
        visited[i][0]=0;
        visited[i][1]=0;
```

```
    }/*初始*/

    for(i=0;i<cham;i++)
        for(j=0;j<cham;j++)
        {
            a[i][j]=0;
            youxiang[i][j]=0;
        }/*初始*/


    for(i=0;i<N;i++)
    {
        j= (int)shuru[i][0]-97 ;
        k= (int)shuru[i][1]-97 ;
        youxiang[j][k]++;
        a[j][k]=1;
        a[k][j]=1;
        visited[j][1]=1;
        visited[k][1]=1;
    }
}

void dft(int index)
{
    int i;
    visited[index][0]=1;
    for(i=0;i<26;i++)
    {
        if(visited[i][1]==1 && a[index][i]==1 && visited[i][0]==0)
            dft(i);
    }
}
//深度优先遍历
int liantong()
{
    int i;
    for(i=0;i<26;i++)
    {
        visited[i][0]=0;/*初始化 visited[]*/
    }
    for(i=0;1;i++)
    {
        if(visited[i][1]==1)
            break;
```

```
    }
    dft(i);
    for(i=0;i<26;i++)
    {
        if(visited[i][0]==0 && visited[i][1]==1)
            return 0;
    }
    return 1;
}

int success()
{
    int i,j;
    int hang,lie,big=0,small=0,ling=0;
    int cha[27];
    for(i=0;i<26;i++)
        cha[i]=0;/*初始*/
    for(i=0;i<26;i++)
    {
        hang=0;lie=0;
        for(j=0;j<26;j++)
        {
            hang+=youxiang[i][j];
            lie+=youxiang[j][i];
        }
        cha[i]=hang-lie;
    }
    for(i=0;i<26;i++)
    {
        if(cha[i]<-1 || cha[i]>1)
            return 0;
    }/*剩下的都是 0,-1,1*/
    for(i=0;i<26;i++)
    {
        if(cha[i]==0)
            ling++;
        else if(cha[i]==1)
            big++;
        else if(cha[i]==-1)
            small++;
    }
    if( ling==26 || (big==1 && small==1) )
        return 1;
    else
```

```
    return 0;
}
```

5.5 有向图的强连通分量

有向图中, u 可达 v 不一定意味着 v 可达 u. 相互可达则属于同一个强连通分量(Strongly Connected Component, SCC)

最关键通用部分：强连通分量一定是图的深搜树的一个子树。

5.5.1 Kosaraju 算法

5. 5. 1. 1 算法思路

这个算法可以说是最容易理解，最通用的算法，其比较关键的部分是同时应用了原图 G 和反图 GT。(步骤 1)先用对原图 G 进行深搜形成森林(树)，（步骤 2）然后任选一棵树对其进行深搜(注意这次深搜节点 A 能往子节点 B 走的要求是 EAB 存在于反图 GT)，能遍历到的顶点就是一个强连通分量。余下部分和原来的森林一起组成一个新的森林，继续步骤 2 直到 没有顶点为止。

5. 5. 1. 2 算法描述

当然，基本思路实现起来是比较麻烦的(因为步骤 2 每次对一棵树进行深搜时，可能深搜到其他树上去，这是不允许的，强连通分量只能存在单棵树中(由开篇第一句话可知))，我们当然不这么做，我们可以巧妙的选择第二深搜选择的树的顺序，使其不可能深搜到其他树上去。想象一下，如果步骤 2 是从森林里选择树，那么哪个树是不连通(对于 GT 来说)到其他树上的呢？就是最后遍历出来的树，它的根节点在步骤 1 的遍历中离开时间最晚，而且可知它也是该树中离开时间最晚的那个节点。这给我们提供了很好的选择，在第一次深搜遍历时，记录时间 i 离开的顶点 j，即 numb[i]=j。那么，我们每次只需找到没有找过的顶点中具有最晚离开时间的顶点直接深搜(对于 GT 来说)就可以了。每次深搜都得到一个强连通分量。

分析到这里，我们已经知道怎么求强连通分量了。但是，大家有没有注意到我们在第二次深搜选择树的顺序有一个特点呢？如果在看上述思路的时候，你的脑子在思考，相信你已经知道了!!! 它就是：如果我们把求出来的每个强连通分量收缩成一个点，并且用求出每个强连通分量的顺序来标记收缩后的节点，那么这个顺序其实就是强连通分量收缩成点后形成的有向无环图的拓扑序列。为什么呢？首先，应该明确搜索后的图一定是有向无环图呢？废话，如果还有环，那么环上的顶点对应的所有原来图上的顶点构成一个强连通分量，而不是构成环上那么多点对应的独自的强连通分量了。然后就是为什么是拓扑序列，我们在改进分析的时候，不是先选的树不会连通到其他树上（对于反图 GT 来说），也就是后选的树没有连通到先选的树，也即先出现的强连通分量收缩的点只能指向后出现的强连通分量收缩的点。那么拓扑序列不是理所当然的吗？这就是 Kosaraju 算法的一个隐藏性质。

1）对 G 进行深度优先搜索并按递归调用完成的先后顺序对各顶点编号；

2）改变 G 的每条边的方向，构造出新的有向图 Gr

3）按 1）中的确定的顶点编号，从编号最大的顶点开始对 Gr 进行深度优先搜索。如果搜索的过程中没有访问遍 Gr 的所有顶点，则从未被访问过的顶点中选取编号最大的顶点，并从此顶点开始继续做深度优先搜索；

4）在最后得到的 Gr 的深度优先生成森林中，每棵树上的顶点组成 G 的一个强连通分

支。

5.5.1.3 应用举例

WTommy's Trouble(TJU 2233)

题目简述：给一个图。这个图满足一个规律。假如 a 得到某个信息。那么 a 可以到达的点都会得到这个信息。但是通知不同点需要耗费的时间不同，问你通知谁可以让所有的人都得到信息并且耗费的时间最小

思路：

对于本题，强联通分量只需通知一个人即可，因此取通知花费最少的一个作为代表；某个点可以代替其以后所有的点（因为不需要 WTommy 花费时间）。可以在求强联通后缩点形成一个新的 DAG（有向无环图）。当然直接 dfs 一遍，累加其代表元的花费即可。

求 SCC： 两遍 DFS，第一遍记录其结束时间，第二遍按照结束时间从晚到早的顺序遍历其 反图 即可（可以同时找出其花费时间最小的作为其代表元，直接赋值给根，以后只需用到根的值即可，不会用到其他的值了）。

最后，DFS 一遍，累加根值的和即可。

代码：

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

using namespace std;

const int maxn = 10001;

vector<int>g[maxn];
vector<int>gb[maxn];

int n,mk[maxn],list[maxn],num;
int color[maxn],col;
int times[maxn];
int in_d[maxn];
int ans[maxn];

void back(int v)
{
    mk [v]=1;color[v] = col;
    for(int u=0;u < gb[v].size(); u++)if(!mk [gb[v][u]])back(gb[v][u]);
}

void dfs(int u)
{
    mk [u]=1;
    for(int v=0;v < g[u].size();v++)if(!mk [g[u][v]])dfs(g[u][v]);
    list [num--] = u;
}
```

```
void scc()
{
    memset(mk,0,sizeof(mk));
    num=n;
    int i;
    for(i=1;i <= n; i++)if(!mk[i])dfs(i);
    memset(mk,0,sizeof(mk));
    for(i=1;i <= n; i++)if(!mk [list [i]]){back(list [i]);col++;}
}
int main()
{
    int m;
    int i,j;
    int a,b;
    int aaa ;

    while(scanf("%d%d",&n,&m))
    {
        if(n == 0 && m ==0 )break;

        col = 1;
        for(i = 1; i <= n; i++){g[i].clear();gb[i].clear();}

        memset(color,0,sizeof(color));
        memset(in_d,0,sizeof(in_d));
        memset(ans,-1,sizeof(ans));

        for(i = 1; i <= n; ++i)
            scanf("%d",&times[i]);
        while(m--)
        {
            scanf("%d%d",&a,&b);
            g[a].push_back (b);
            gb[b].push_back(a);
        }
        scc();
        for(i = 1; i <= n; ++i)cout<<color[i]<<" ";cout<<endl;
        for(i = 1; i <= n; ++i)
            for(j = 0; j < g[i].size(); ++j)
                if(color[i] != color[g[i][j]])
                    in_d[color[g[i][j]]]++;
        for(i = 1; i <= n; ++i)
        {
```

```
                if(in_d[color[i]])ans[color[i]] = 0;
                else if((times[i] < ans[color[i]]) || ans[color[i]] == -1)
                    ans[color[i]] = times[i];
            }
            aaa = 0;
            for(i = 1; i < col; i++)aaa += ans[i];
            cout<<aaa<<endl;
        }
        return 0;
    }
```

5.5.2 Tarjan 算法

5. 5. 2. 1 算法思路

这个算法思路不难理解，由开篇第一句话可知，任何一个强连通分量，必定是对原图的深度优先搜索树的子树。那么其实，我们只要确定每个强连通分量的子树的根，然后根据这些根从树的最低层开始，一个一个的拿出强连通分量即可。那么身下的问题就只剩下如何确定强连通分量的根和如何从最低层开始拿出强连通分量了。

5. 5. 2. 2 算法描述

如何确定强连通分量的根?在这里我们维护两个数组，一个是 indx[1..n]，一个是 mlik[1..n]，其中 indx[i]表示顶点 i 开始访问时间，mlik[i]为与顶点 i 邻接的顶点未删除顶点j 的 mlik[j]和 mlik[i]的最小值(mlik[i]初始化为 indx[i])。这样，在一次深搜的回溯过程中，如果发现 mlik[i]==indx[i]那么，当前顶点就是一个强连通分量的根，为什么呢？因为如果它不是强连通分量的根，那么它一定是属于另一个强连通分量，而且它的根是当前顶点的祖宗，那么存在包含当前顶点的到其祖宗的回路，可知 mlik[i]一定被更改为一个比 indx[i]更小的值。

至于如何拿出强连通分量，这个其实很简单，如果当前节点为一个强连通分量的根，那么它的强连通分量一定是以该根为根节点的(剩下节点)子树。在深度优先遍历的时候维护一个堆栈，每次访问一个新节点，就压入堆栈。现在知道如何拿出了强连通分量了吧？是的，因为这个强连通分量时最先被压人堆栈的，那么当前节点以后压入堆栈的并且仍在堆栈中的节点都属于这个强连通分量。当然有人会问真的吗？假设在当前节点压入堆栈以后压入并且还存在，同时它不属于该强连通分量，那么它一定属于另一个强连通分量，但当前节点是它的根的祖宗，那么这个强连通分量应该在此之前已经被拿出。

1)计算当前结点的层号 lv[x]，并在并查集中建立仅包含 x 结点的集合，即 root[x]:=x。

2)依次处理与该结点关联的询问。

3)递归处理 x 的所有孩子。

4)root[x]:=root[father[x]]（对于根结点来说，它的父结点可以任选一个，反正这是最后一步操作了）。

5.5.3 Gabow 算法

5.5.2.1 算法思路

这个算法其实就是 Tarjan 算法的变异体，我们观察一下，只是它用第二个堆栈来辅助求出强连通分量的根，而不是 Tarjan 算法里面的 indx[]和 mlik[]数组。那么，我们说一下如何使用第二个堆栈来辅助求出强连通分量的根。

5.5.2.2 算法描述

我们使用类比方法，在 Tarjan 算法中，每次 mlik[i]的修改都是由于环的出现(不然，mlik[i]的值不可能变小)，每次出现环，在这个环里面只剩下一个 mlik[i]没有被改变(深度最低的那个)，或者全部被改变，因为那个深度最低的节点在另一个环内。那么 Gabow 算法中的第二堆栈变化就是删除构成环的节点，只剩深度最低的节点，或者全部删除，这个过程是通过出栈来实现，因为深度最低的那个顶点一定比前面的先访问，那么只要出栈一直到栈顶那个顶点的访问时间不大于深度最低的那个顶点。其中每个被弹出的节点属于同一个强连通分量。那有人会问：为什么弹出的都是同一个强连通分量？因为在这个节点访问之前，能够构成强连通分量的那些节点已经被弹出了，这个对 Tarjan 算法有了解的都应该清楚，那么 Tarjan 算法中的判断根我们用什么来代替呢？想想，其实就是看看第二个堆栈的顶元素是不是当前顶点就可以了。　　现在，你应该明白其实 Tarjan 算法和 Gabow 算法其实是同一个思想的不同实现，但是，Gabow 算法更精妙，时间更少(不用频繁更新 mlik[])。

第六章　搜索算法

6.1 概要

搜索算法是利用计算机的高性能来有目的的穷举一个问题解空间的部分或所有的可能情况，从而求出问题的解的一种方法。

搜索算法实际上是根据初始条件和扩展规则构造一颗“解答树”并寻找符合目标状态的节点的过程。所有的搜索算法从最终的算法实现上来看，都可以划分成两个部分——控制结构（扩展节点的方式）和产生系统（扩展节点），而所有的算法优化和改进主要都是通过修改其控制结构来完成的。其实，在这样的思考过程中，我们已经不知不觉地将一个具体的问题抽象成了一个图论的模型——树，即搜索算法的使用第一步在于搜索树的建立。

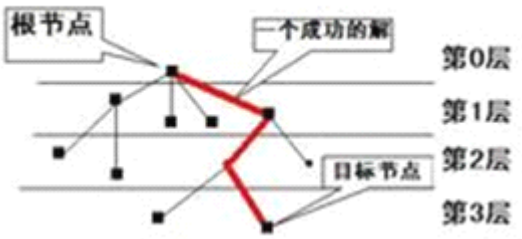


图 6.1

由图一可以知道，这样形成的一棵树叫搜索树。初始状态对应着根结点，目标状态对应着目标结点。排在前的结点叫父结点，其后的结点叫子结点，同一层中的结点是兄弟结点，由父结点产生子结点叫扩展。完成搜索的过程就是找到一条从根结点到目标结点的路径，找出一个最优的解。这种搜索算法的实现类似于图或树的遍历，通常可以有两种不同的实现方法，即深度优先搜索（DFS——Depth First search）和广度优先搜索（BFS——Breadth First Search）。

6.2 深度优先搜索（DFS——Depth First search）

深度优先搜索是一种在开发爬虫早期使用较多的方法。它的目的是要达到被搜索结构的叶结点(即那些不包含任何超链的 HTML 文件) 。在一个 HTML 文件中，当一个超链被选择后，被链接的 HTML 文件将执行深度优先搜索，即在搜索其余的超链结果之前必须先完整地搜索单独的一条链。深度优先搜索沿着 HTML 文件上的超链走到不能再深入为止，然后返回到某一个 HTML 文件，再继续选择该 HTML 文件中的其他超链。当不再有其他超链可选择时，说明搜索已经结束。

事实上,深度优先搜索属于图算法的一种,英文缩写为 DFS 即 Depth First Search.其过程简要来说是对每一个可能的分支路径深入到不能再深入为止,而且每个节点只能访问一次.

举例说明之:下图是一个无向图,如果我们从 A 点发起深度优先搜索(以下的访问次序并不是唯一的,第二个点既可以是 B 也可以是 C,D),则我们可能得到如下的一个访问过程:A->B->E(没有路了!回溯到 A)->C->F->H->G->D(没有路,最终回溯到 A,A 也没有未访问的

相邻节点,本次搜索结束).

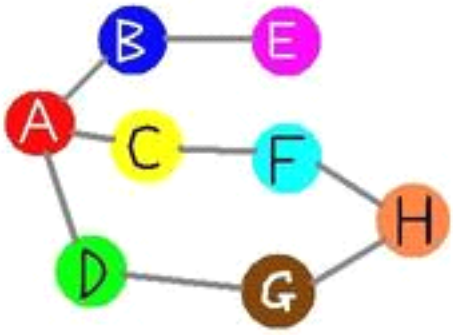


图 6.2

简要说明深度优先搜索的特点:每次深度优先搜索的结果必然是图的一个连通分量.深度优先搜索可以从多点发起.如果将每个节点在深度优先搜索过程中的"结束时间"排序(具体做法是创建一个 list,然后在每个节点的相邻节点都已被访问的情况下,将该节点加入 list 结尾,然后逆转整个链表),则我们可以得到所谓的"拓扑排序",即 topological sort.

那么 DFS 用代码怎么实现呢?写 DFS 的时候我们会用到递归求解

```
void DFS (int a)
{
   work();
   DFS(a+1);
}
```

上面一段代码就是用递归求解实现了 DFS.

我们现在来看道题:

POJ_1011 Sticks

题目大意：n 根小棍，拼成若干根长棍，要这些长棍的长度相等，并且小棍刚好都用完，问能拼成的长棍的最短长度是多少。

思路：首先把小棍按长度，从大到小排序(为了进行贪心选择)，并计算这些小棍的总长度，拼成的长棍的长度从最长的小棍开始搜索，如果小棍的总长度能整除该长棍长度，则可能完成拼凑，问题简化成 n 根小棍，长度已知，拼成 num_stick 根长度为 l 的长棍，能否完成，若能，输出 1，结束；否则，增加长棍的长度，继续搜索，最坏的情况下，所有的小木棍一起，能拼成一根长木棍，所以，搜索终能成功。

代码：

```
bool dfs (int left,int c,int v)     //left 还剩余的长度   c 当前组合好的棍子数   v 搜索到第几根
{
    int i;
    if(left==0)
    {
        if(sum/val-2==c)return 1;
        for(i=0;;i++)if(use[i]==0)break;
        use[i]=1;
        if(dfs(val-a[i],c+1,i))return 1;
        use[i]=0;
```

```
            return 0;
        }
        else
        {
            if(v>=n-1)return 0;
            for(i=v;i<n;i++)
            {
                if(use[i])continue;
                if(a[i]==a[i-1]&&use[i-1]==0)continue;
                if(a[i]>left)continue;
                use[i]=1;
                if(dfs(left-a[i],c,i))return 1;
                use[i]=0;
            }
            return 0;
        }
    }
```

//use[i]表示是否使用过第 i 跟棍子,val 是我当前搜索的使棍子组合达到的长度(这个值满足能被总长度整除),a[i]是题目中给的第 i 根棍子的长度(已经按照升序排列),sum 是所有棍子加起来的总长度

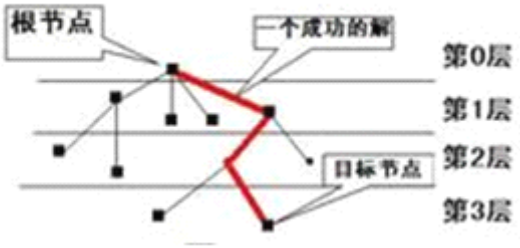
上面是程序的主要部分,我们进行搜索的时候,每次搜索一个状态的时候,我们要知道当前拼成一根棍子还需要的长度,当前的已经拼好的棍子数目,还有当前搜索到了第几根.然后我们就可以继续搜索下去.

如果当前剩余为 0 时,表示当前一根棍子已经拼好,我们就再重新找一根新的棍子开始搜索,如果拼好的棍子数达到了我要求的棍子数目的时候,则整个搜索就结束了.

如果单单这样做的话,那么就超时了.所以,伴随着搜索算法的一个核心就是剪枝.

那么什么是剪枝呢?

我们知道,搜索是对一棵状态树进行的,我们还是看图 6.1



对于这样的一棵搜索数,我们可以深搜找一条可行的路径,也就是一个可行解,那么,剪枝是什么呢?如果我们在某一条路上(还没有到达叶子节点),我们已经知道了从这条路走是不行的,那么我们就可以进行剪枝了,也就是说我们可以不必再访问其儿子节点而直接查找其兄弟节点.我们还是看上面那个题目.那么对于这题,我们的剪枝在哪呢?

if(a[i]>left)continue;

这句话是什么意思呢?由于我们的棍子是按长度升序排列的,那么当当前的长度大于我所需要的长度的时候,也就是说后面的所有的棍子的长度都大于我组成当前某根棍子的长度的时候,我当前的想拼成的那根棍子是不可能拼成的,那么我们就没有必要再往下搜索了,也就是说我们不需要再搜索其儿子节点,我们应该直接搜索其兄弟节点,所以这里我们用了

continue 来继续找其兄弟节点.

还有很多人会看不懂的一句话 if(sum/val-2==c)return 1;

为什么我们直接搜索到 sum/val-2 呢?最多也是减 1 啊.减 1 大家都能理解,因为我当前搜索完的一根还没有加上去.但是减 2 该怎么理解呢?我们可以这么想,如果我们要拼成 m 根棍子,我们现在已经拼成了 m-1 根棍子了,而且我们确保了总长度是能整除所拼成的每根棍子长度的,那么我们还剩下一根没有拼的棍子是不是一定能够拼成呢?答案显然是正确的.那么我们还有必要去搜索这根一定能拼成的棍子么?所以这个剪枝是对成功子树的剪枝.

综上,剪枝不一定是对不成功子树的剪枝,有时候还可以针对成功子树的剪枝.剪枝时一定要切合题意,看哪些状态是无效的,或者哪些状态能够推出无效,或者哪些状态必然有效,那么对于到达那些状态,我们可以直接对其剪掉.

对于深搜,还有一个非常重要的剪枝,就是记忆化搜索.何为记忆化搜索呢?我们可以这样来理解,假设我们要完成一系列任务(任务直接是有联系的,比如要完成 A 首先要完成 C),对于 A 任务,我想知道最快要多少时间完成,那么对 A 进行搜索,搜索一条路,使 A 能完成,并且保证 A 完成时间是最少的.假设我们现在对 B 进行搜索,我发现如果要完成 B,那么必定要先完成 A,则按照一般的想法,继续对 A 进行搜索,可是,在 B 之前,我们已经知道了要完成 A 的最短时间,那么,我们是不是可以直接拿来用呢?所以,我们就可以把每次搜索的值记录下来,等到下次再需要用的时候直接拿来用就可以了,而不需要再搜索了.

滑雪（PKU1088）

题目简述： Michael 喜欢滑雪百这并不奇怪， 因为滑雪的确很刺激。可是为了获得速度，滑的区域必须向下倾斜，而且当你滑到坡底，你不得不再次走上坡或者等待升降机来载你。Michael 想知道载一个区域中最长底滑坡。区域由一个二维数组给出。数组的每个数字代表点的高度。下面是一个例子

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

一个人可以从某个点滑向上下左右相邻四个点之一，当且仅当高度减小。在上面的例子中，一条可滑行的滑坡为 24-17-16-1。当然 25-24-23-...-3-2-1 更长。事实上，这是最长的一条。

思路： 多阶段决策，从大到小，的 dp，保证每个数叠加的时候，前一个比他小的数已经算好，用一个新的数组将它们排好序，记住原来的位置，四个方向判断，但要判断那个方向是最大的

rec[i][j].ans=rec[][].ans+1; if(rec[i][j]><rec[][]);

代码：

```
struct node
{
   int x,y;
};
int map[105][105];
int ans[105][105];
void dfs (node a,int m)
```

```
{
    node temp;
    if(ans[a.x][a.y]!=0)
    {
        if(m+ans[a.x][a.y]-1>mmax)mmax=m+ans[a.x][a.y]-1;
        if(m+ans[a.x][a.y]-1>mm)mm=m+ans[a.x][a.y]-1;
        return ;
    }
    if(a.y+1<c&&map[a.x][a.y+1]<map[a.x][a.y])
    {
        temp.x=a.x;
        temp.y=a.y+1;
        dfs(temp,m+1);
    }
    if(a.y-1>=0&&map[a.x][a.y-1]<map[a.x][a.y])
    {
        temp.x=a.x;
        temp.y=a.y-1;
        dfs(temp,m+1);
    }
    if(a.x+1<r&&map[a.x+1][a.y]<map[a.x][a.y])
    {
        temp.x=a.x+1;
        temp.y=a.y;
        dfs(temp,m+1);
    }
    if(a.x-1>=0&&map[a.x-1][a.y]<map[a.x][a.y])
    {
        temp.x=a.x-1;
        temp.y=a.y;
        dfs(temp,m+1);
    }

    if(m>mmax)mmax=m;
    if(m>mm)mm=m;
}
```

6.3 广度优先搜索（BFS——Breadth First Search）

广度优先搜索算法（又称宽度优先搜索）是最简便的图的搜索算法之一，这一算法也是很多重要的图的算法的原型。Dijkstra 单源最短路径算法和 Prim 最小生成树算法都采用了和宽度优先搜索类似的思想。其别名又叫 BFS，属于一种盲目搜寻法，目的是系统地展开并检查图中的所有节点，以找寻结果。换句话说，它并不考虑结果的可能位址，彻底地搜索整张图，直到找到结果为止。

BFS 并不使用经验法则算法。从算法的观点，所有因为展开节点而得到的子节点都会被加进一个先进先出的伫列中。一般的实作里，其邻居节点尚未被检验过的节点会被放置在一个被称为 open 的容器中（例如伫列或是链表），而被检验过的节点则被放置在被称为 closed 的容器中。（open-closed 表）

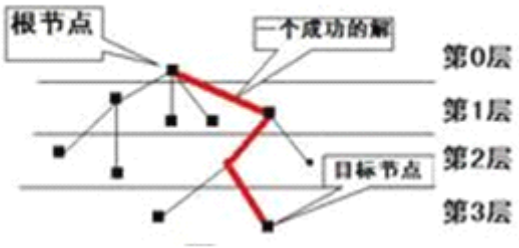
已知图 G=(V,E)和一个源顶点 s，宽度优先搜索以一种系统的方式探寻 G 的边，从而“发现”s 所能到达的所有顶点，并计算 s 到所有这些顶点的距离(最少边数)，该算法同时能生成一棵根为 s 且包括所有可达顶点的宽度优先树。对从 s 可达的任意顶点 v，宽度优先树中从s到 v 的路径对应于图 G 中从 s 到 v 的最短路径，即包含最小边数的路径。该算法对有向图和无向图同样适用。

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末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。

同样一切的搜索都是对一课状态树的搜索,那么宽搜是怎么搜索的呢?我们还是看图 6.1.



对于深搜,我们是一直往下找找到一个可行解,那么宽搜的话,我们是这样搜索的,每次,我把当前节点的所有子节点都访问一遍(不管是不是一个可行解),那么作为这一层的状态,接着,我在从这层的状态找出这层状态的下一层状态,一直找,知道找到目标节点则结束.

同样的我们看一道题。

（POJ3278）

题目简述:给你一个起点和一个终点,每个点能够到达的状态是 x-1,x+1 和 x*2,求最小几步可以到达终点.

思路：对于每个状态,都可以到达三个子状态,那么从起点开始,第二层的状态就有 3 个,第三层的状态有 9 个,依次下去,知道到达终点的状态结束,那么我们求出的步数就可以达到最小.因为对于相同层的状态,我所需的步数是相同的,而且对于层与层之间的状态来说,我先搜完前一层的所有状态才开始搜下一层的状态.也就是说前一个节点的步数一定小于等于后一个节点的步数.所以当第一次到达了终点的时候,我就得到了最短的路径.

代码：

```
struct node
{
  int num;
  int count;
};
node queue[1000000];
bool flag[1000005]={0};
int top,end;
int move (int aaa,int bbb)
{
  if(flag[aaa+1]==0&&aaa+1<=100000)
  {
      queue[top].num=aaa+1;   //向右走一步
      flag[aaa+1]=1;
      queue[top++].count=bbb+1;
  }
  if(top>=999999)top=0;
  if(flag[aaa-1]==0&&(aaa-1)>=0)
  {
      queue[top].num=aaa-1;   //向左走一步
      flag[aaa-1]=1;
      queue[top++].count=bbb+1;
  }
  if(top>=999999)top=0;
  if(flag[2*aaa]==0&&2*aaa<=100000)
  {
      queue[top].num=2*aaa;   //跳跃一步
      flag[2*aaa]=1;
      queue[top++].count=bbb+1;
  }
  if(top>=999999)top=0;
  return 0;
}
while(queue[end].num!=K)
{
  if(end>=999999)end=0;
  move(queue[end].num,queue[end].count);
  end++;
}
//用 queue 队列来存储所有状态,循环队列
```

6.4 启发式搜索(介于深搜与宽搜之间的搜索)

启发式搜索就是在状态空间中的搜索对每一个搜索的位置进行评估，得到最好的位置，再从这个位置进行搜索直到目标。这样可以省略大量无谓的搜索路径，提到了效率。在启发式搜索中，对位置的估价是十分重要的。采用了不同的估价可以有不同的效果。我们先看看估价是如何表示的。

启发中的估价是用估价函数表示的，如：

最佳优先搜索的最广为人知的形式称为 A*搜索(发音为“A 星搜索”).它把到达节点的耗散 g(n)

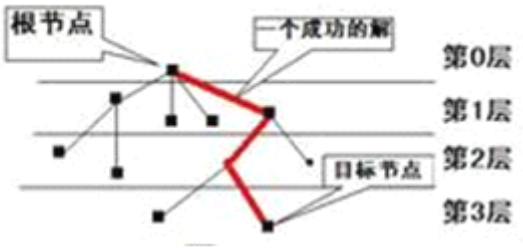
和从该节点到目标节点的消耗 h(n)结合起来对节点进行评价：f(n)=g(n)+h(n)

因为以 g(n)给出了从起始节点到节点 n 的路径耗散，而 h(n)是从节点 n 到目标节点的最低耗散路径的估计耗散值，因此 f(n)=经过节点 n 的最低耗散解的估计耗散.这样，如果我们想要找到最低耗散解，首先尝试找到 g(n)+h(n)值最小的节点是合理的。可以发现这个策略不只是合理的:倘若启发函数 h(n)满足一定的条件，A*搜索既是完备的也是最优的。

如果把 A*搜索用于 Tree-Search,它的最优性是能够直接分折的。在这种情况下，如果 h(n)是一个可采纳启发式--也就是说，倘若 h(n)从不会过高估计到达目标的耗散--A*算法是最优的。可采纳启发式天生是最优的，因为他们认为求解问题的耗散是低于实际耗散的。因为 g(n)是到达节点 n 的确切耗散，我们得到一个直接的结论：f(n)永远不会高估经过节点 n 的解的实际耗散.

启发算法有：蚁群算法，遗传算法、模拟退火算法等.

那么同样看图 6.1



对于我每一个状态,我都有一个)评估函数,f(n)=g(n)+h(n)(保证 h(n)不大于实际代价).g(n)是到达 n 状态时的实际代价,h(n)是我假设的一个函数,是对于 n 状态到达目标状态预期的值.那么我们可以这么搜索,对于每个状态,我们可以找出其所有的子状态,然后根据 f(n)的代价,选取一个代价最下的往下搜索.这样的话,我就可以先选取最优的,而不是盲目的对所有状态进行搜索.那么我们可以省去所有比较”不好”的状态的搜索.

我们可以这么想,假设我 g(n)始终为 0 的话,那么我是不是就可以转化为深搜.因为当我从起点开始搜索到一个状态的时候,当前搜索到的状态的 h(n)一定是最小的(因为没有 g(n)).这样的话那么我始终就对该状态一直往下搜索,直到搜索结束(找到或者没有找到).这样的话也就是先找一条路,走完后再找另一条路.

那么如果我 f(n)始终为 0 的时候,那么我是不是可以转化为宽搜呢?因为我 g(n)始终是实际代价,我宽搜的时候也是每次都是实际代价,那么这样的话就和宽搜一样了.

我们来看一个 A 星算法可以解的题.。

POJ_2449

题目简述：给你一张 n*n 的图,有 m 条路(有向),求从 S 到 T 的第 K 短路的路径长度.

思路：我们可以用宽搜或者深搜来实现,宽搜或者深搜的时候,我们把所有的路径长度都找出来,然后在求出第 K 短的路径长度,但是显然这么做是超时的.那么我们是不是可以用个介于深搜和宽搜直接的 A 星来求解呢?

我们这么设 h(x),h(x)为 x 点到达目标的最短路径.这时我们每次寻找状态的时候,只要在我当前可行的状态里面找一个估价最小的状态来搜索(估价为 f(n)=g(n)+h(n)),这样每次到达终点的时候,我们可以确保是当前所有状态的最短路径.那么求 K 短只要求出第 K 次到达目标时的路径长度就可以了.

代码：

```
/*A*的估计函数 f(n) = g(n) + h(n),
g(n)表示起点到 n 的最小值
h(n)为顶点 n 到终点 t 的最短路(满足 h(n)<=h'(n),)
h'(n)为顶点 n 到终点 t 实际值
*/
void dijstra(int s,int n)  //s 为起始点  n 为总点数目 Q 为一个优先队列
{
  memset(h,127,sizeof(h));//h 是估计函数
  memset(visted,false,sizeof(visted));
  while(Q.size()) Q.pop();//用优先队列维护 f(n)
  h[s]=0;
  node q;
  int res;
  q.cost=h[s];
  q.v=s;
     Q.push(q);
  while(Q.size())
  {
      q=Q.top();
      Q.pop();
      res=q.v;
      if(visted[res]) continue;
      visted[res]=true;
      int index=edge1[res].next;
      Edge e;
         while(index!=-1)
      {
          e=edge1[index];
          if(!visted[e.t] && h[e.t]>h[res]+e.wight)//更新放入队列
          {
              h[e.t]=h[res]+e.wight;
              q.cost=h[e.t];
              q.v=e.t;
              Q.push(q);
          }
```

```
                index=e.next;
            }
        }
    }
```

6.5 双向宽搜

广度双向搜索的概念

所谓双向搜索指的是搜索沿两个力向同时进行：

正向搜索：从初始结点向目标结点方向搜索；

逆向搜索：从目标结点向初始结点方向搜索；

当两个方向的搜索生成同一子结点时终止此搜索过程。

广度双向搜索算法

广度双向搜索通常有两中方法：

1. 两个方向交替扩展

2. 选择结点个数较少的那个力向先扩展.

方法 2 克服了两方向结点的生成速度不平衡的状态，明显提高了效率。

算法说明：

设置两个队列 c:array[0..1,1..maxn] of jid，分别表示正向和逆向的扩展队列。

设置两个头指针 head:array[0..1] of integer 分别表示正向和逆向当前将扩展结点的头指针。

设置两个尾指针 tail:array[0..1] of integer 分别表示正向和逆向的尾指针。

maxn 表示队列最大长度。

双向宽搜和一般宽搜比较:

双向宽搜的状态数是两个三角形,而一般的宽搜是一个大的三角形,则双向宽搜的状态数目是一般宽搜的一半.

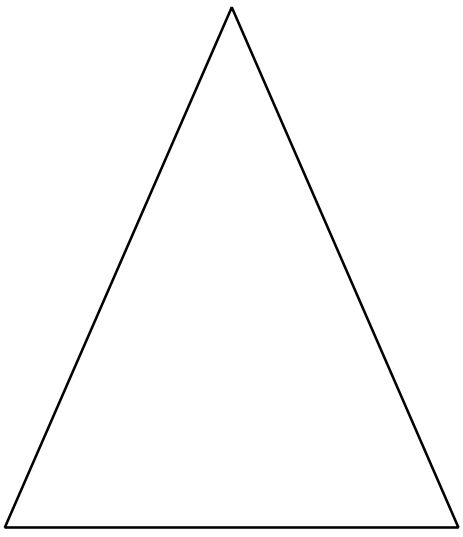


图 6.3 单向宽搜的状态树

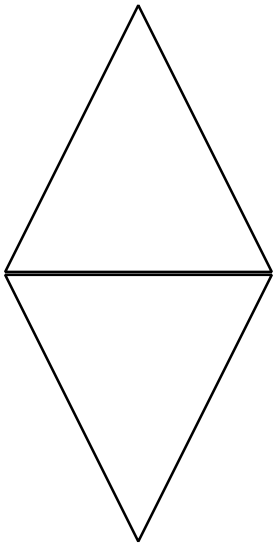


图 6.4 双向宽搜的状态树

则使用双向宽搜时间一般是单向宽搜的一半.

6.6 总结

搜索是比较常用的算法,理论上对于所有能转化为状态的问题,用搜索都是可以求出答案的(只是时间问题).搜索的精髓是剪枝(个人观点),向 A 星搜索,与其说是一种搜索方法,不如说是一种剪枝方法,A 星算法,只是采用合适的策略来使得某些状态被放在队列的底层,使得这些状态不能被访问到,也就是说把这些状态给剪掉了.

对于搜索的剪枝,一定要善于思考,善于发现,使得你的搜索向着你预期的方向走,尽量少走”冤枉路”.

第七章 网络流算法

许多系统包含了流量问题。例如，公路系统有车辆流，控制系统有信息流，供水系统中有水流，金融系统中有现金流等等。这些流都从一个产生该物质的“源”出发，经过一个系统流向另一个消耗该物质的“汇”。“源”以固定的速度产生该物质，并且“汇”用同样的速度消耗该物质。所谓流量是指该物质在系统中的运行速度。

7.1 网络流的一般概念

7.1.1 网络流的一些基本符号

V：整个图中的所有结点的集合.
E：整个图中所有边的集合.
G = (V,E) ：表示整个图.
s：网络的源点
t：网络的汇点.
c(u,v)：对于每条边(u,v),有一个容量 c(u,v)　(c(u,v)>=0)。如果 c(u,v)=0，则表示(u,v)不存在在网络中。如果原网络中不存在边(u,v)，则令 c(u,v)=0
f(u,v)：每条边(u,v),有一个流量 f(u,v).

7.1.2 网络流的三个性质

1、容量限制:　f[u,v]<=c[u,v]
2、反对称性：f[u,v] = - f[v,u]
3、流量平衡:　对于不是源点也不是汇点的任意结点,流入该结点的流量和等于流出该结点的流量和。
结合反对称性,流量平衡也可以写成:

$$\sum_{u \in V} f(v,u) = 0$$

只要满足这三个性质,就是一个合法的网络流.

7.2 最大流

7.2.1 网络流的定义

定义一个网络的流量（记为|f|）=$\sum_{v\in V} f(s,v)$

最大流问题，就是求在满足网络流性质的情况下，|f|的最大值

7.2.2 增广路法

1．算法的一般步骤：

步骤 1. 从发点 s 到收点 t 选定一条路，使这条路通过的所有弧 Vij 的前面约束量 cij 都大于 0，如果找不到这样的路，说明已经求得最大流，转步骤 4。

步骤 2. 在选定的路上，找到最小的容许量 cij 定为 P。

步骤 3. 对选定的路上每条弧的容量作以下修改，对于与路同向的弧，将 cij 修改为 cij-P，对于与路反向的弧，将 cij 修改为 cij+P。修改完毕后再转入步骤 1。

步骤 4.用原图中各条弧上起点与终点数值减去修改后的图中对应点的数值，得到正负号相反的两个数，并将从正到负的方向用箭头表示。这样，就得到一个最大流量图 。

2．算法的具体实现过程如下：

讨论如下图的求解过程

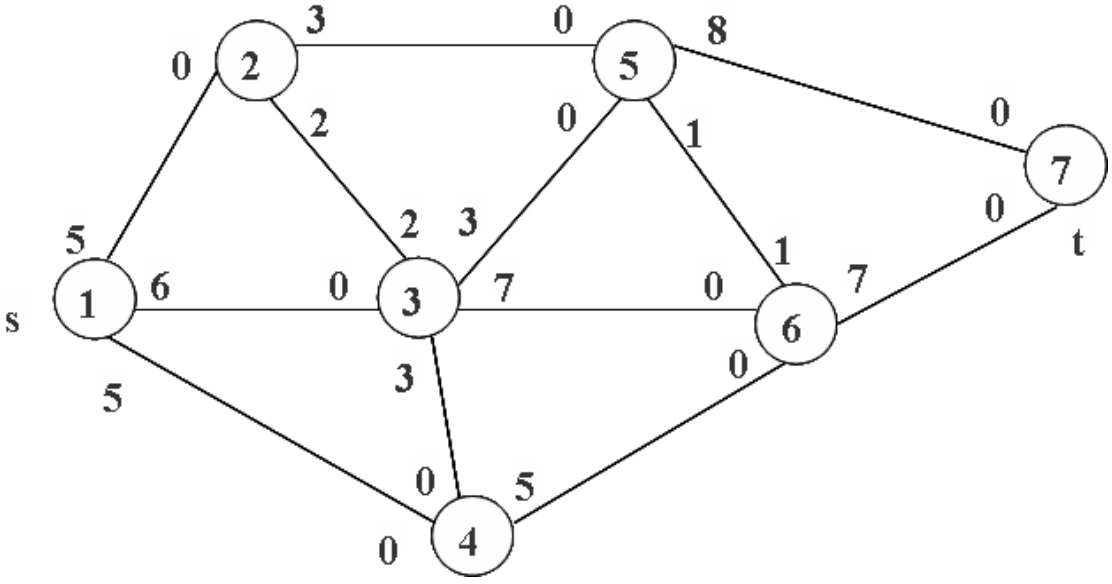


图 7-1

每条弧V_{ij} 上标有两个数字，其中，靠近点 i 的是c_{ij} ，靠近点 j 的是c_{ji} 。如① $\frac{5}{}$ $\frac{0}{}$② 表示从①到②的最大通过量是 5（百辆），从②到①的最大通过量是 0；② $\frac{2}{}$ $\frac{2}{}$③表示从②到③和从③到②都可以通过 2；等等。

下面我们用增广路法解决上面的问题：

第一次修改：

从发点 s 到收点 t 找一条路，使得这条路上的所有弧前面的约束量 $c_{ij}>0$. 从图 7-1 中可以看出，显然，①—③—⑥—⑦就是满足这样的条件的一条路。

在路①—③—⑥—⑦中, $c_{13}=6$,$c_{36}=7$, $c_{67}=7$,所以取 $P=c_{13}=6$。

在路①—③—⑥—⑦中，修改每一条弧的容量：

$c_{13}=6\text{-}P=6\text{-}6=0$　　$c_{31}=0+P=0+6=6$

$c_{36}=7-P=7\text{-}6=1$　　$c_{63}=0+P=0+6=6$

$c_{67}=7-P=7\text{-}6=1$　　$c_{76}=0+P=0+6=6$

修改过后的图如下：

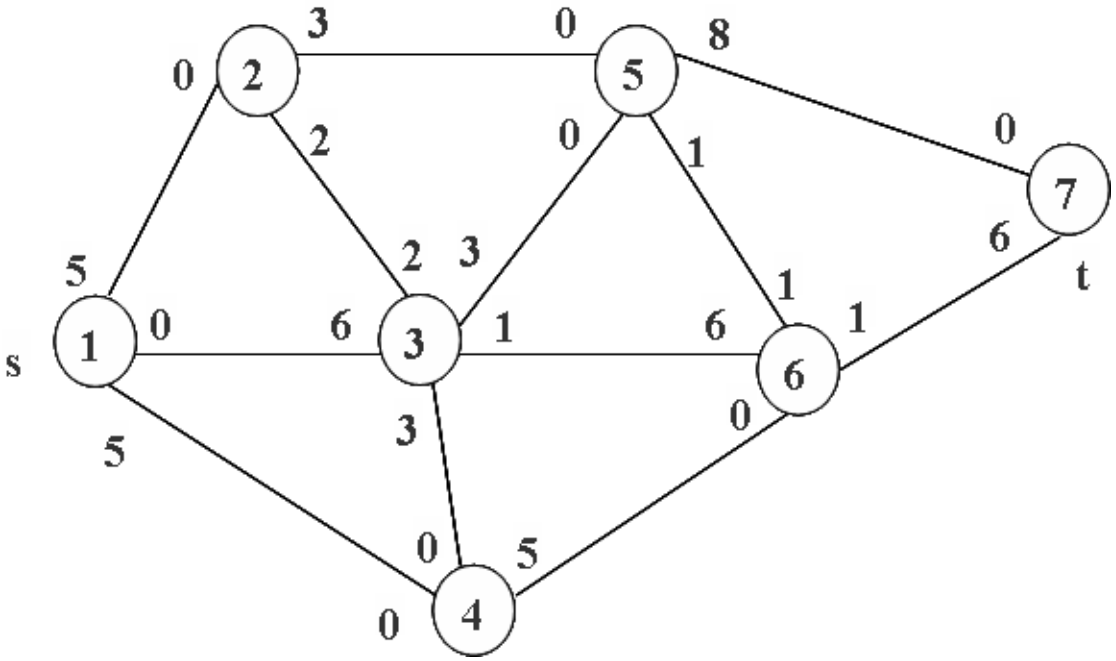


图 7-2

第二次修改：

选定①—②—⑤—⑦，在这条路中，由于$P=c_{25}=3$，所以，将c_{12} 改为 2，C25 改为 0, c_{57} 改为 5，c_{21}、c_{52} c_{75} 改为 3。修改后的图变为图 7-3。

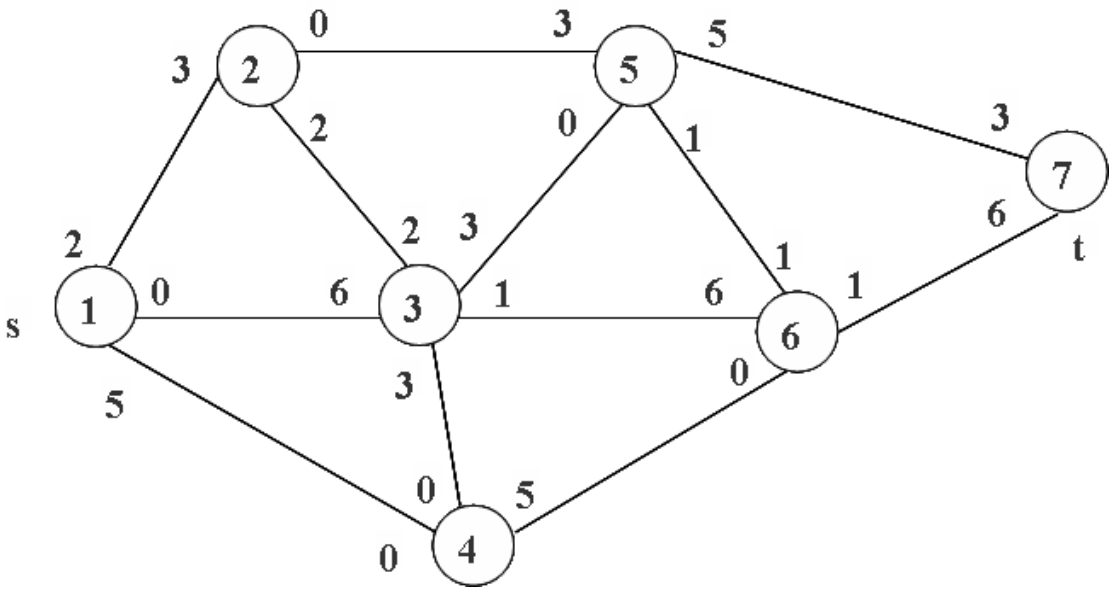


图 7-3

第三次修改：

取①—②—③—⑤—⑦，在这条，由于$P = c_{12} = 2$，所以将c_{12} 改为 0，c_{21}改为 5，c_{23} 改为 0，C32 改为 4，c_{35}改为 1，c_{53} 改为 2，c_{57}改为 3，C75 改为 5。修改后的图变为图 7-4

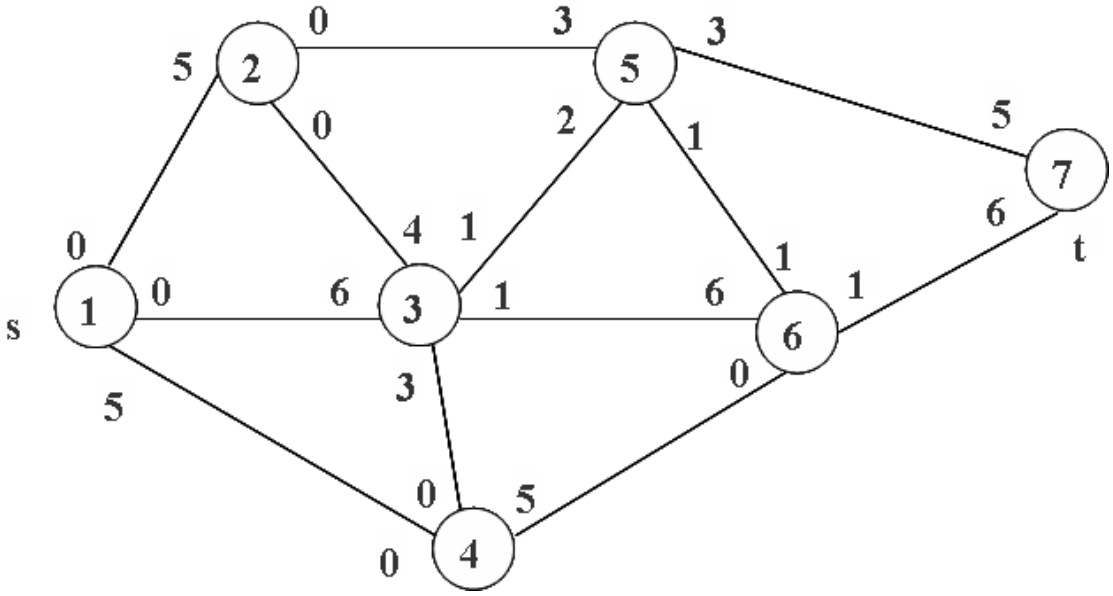


图 7-4

第四次修改：

选定①—④—⑥—⑦，在这条路中，由于 $P=c_{67}=1$，所以将 c_{14}改为 4，c_{41} 改为 1，c_{46}改为 4，c_{64}改为 1，c_{67}改为 0，c_{76} 改为 7。修改后的图为变为图 7-5 。

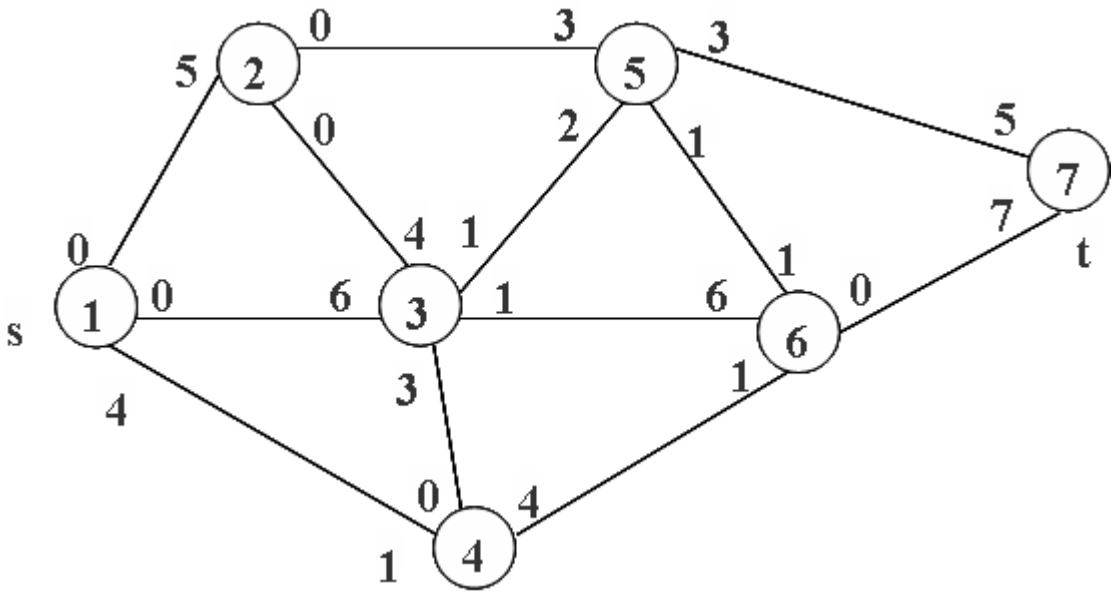


图 7-5

第五次修改：

选定①—④—⑥—⑤—⑦,在这条路中，由于 $P=c_{65}=1$，所以将 c_{14}和 c_{46}均改为 3，c_{65}改为 0，c_{57}改为 2，c_{41}、c_{64}、c_{56}均改为 2，c_{75}改为 6。修改后的图变为图 7-6。

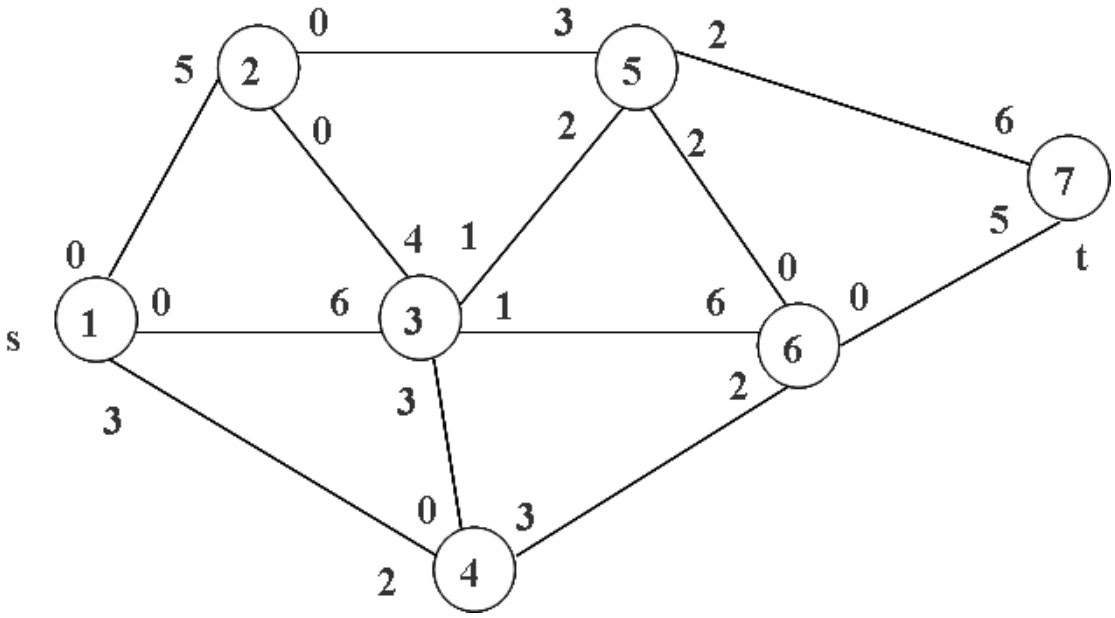


图 7.6

第六次修改：

取①—④—⑥—③—⑤—⑦，在这条路中，由于 $P=c_{35}=1$，所以将 c_{14}和 c_{46}均改为 2，c_{63}改为 5，c_{35}改为 0，c_{57}改为 1，c_{41}、c_{64}、c_{53}均改为 3，c_{36}改为 2，c_{75}改为 7。修改后的图变为图 7-7。

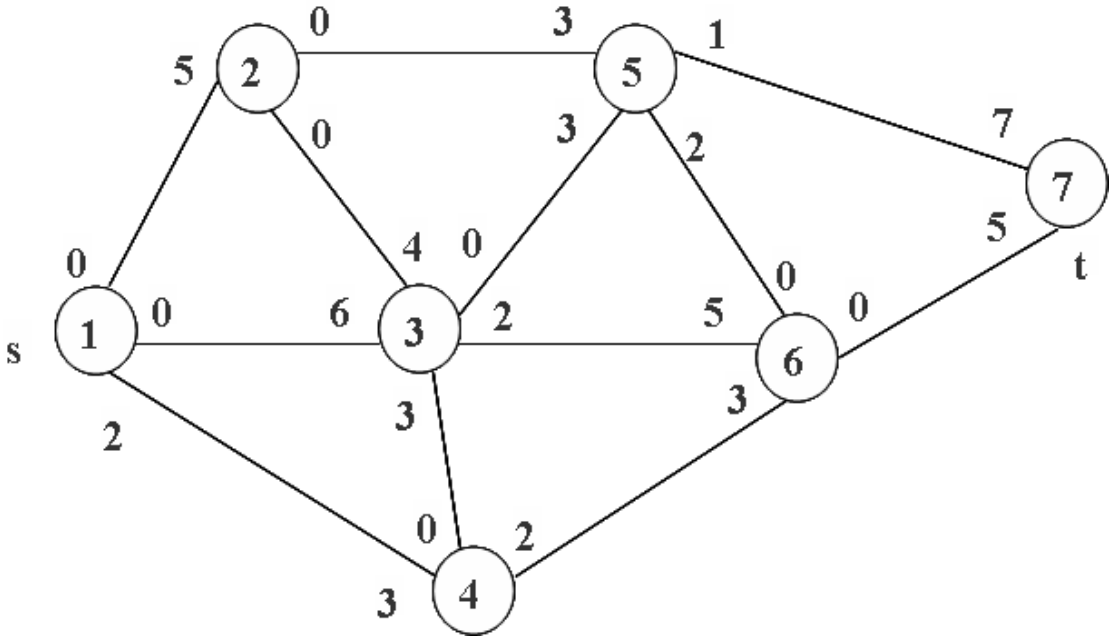


图 7-7

在图 7.7 中，从发点①到收点⑦，再也不存在连通的起点容量都大于零的弧了，所以图 7.7 为最大流图。

转入步骤（4），用原图中各条弧上起点与终点数值减去修改后的图上各点的数值，将得到正负号相反的两个数，将这个数标在弧上，并将从正到负的方向用箭头表示,这样就得到最大流量图。例如原来弧 (3,6) 是③ $\overset{7}{\underset{0}{—}}$ ⑥，现在是③ $\overset{2}{\underset{5}{—}}$ ⑥，相减为±5，③那边为正，我们就记作③ $\overset{5}{\rightarrow}$ ⑥。这样，就得到图 7.8，即最大流量。依这样的调度方式，可以从发点 *s* 调运 14 到收点 *t*。

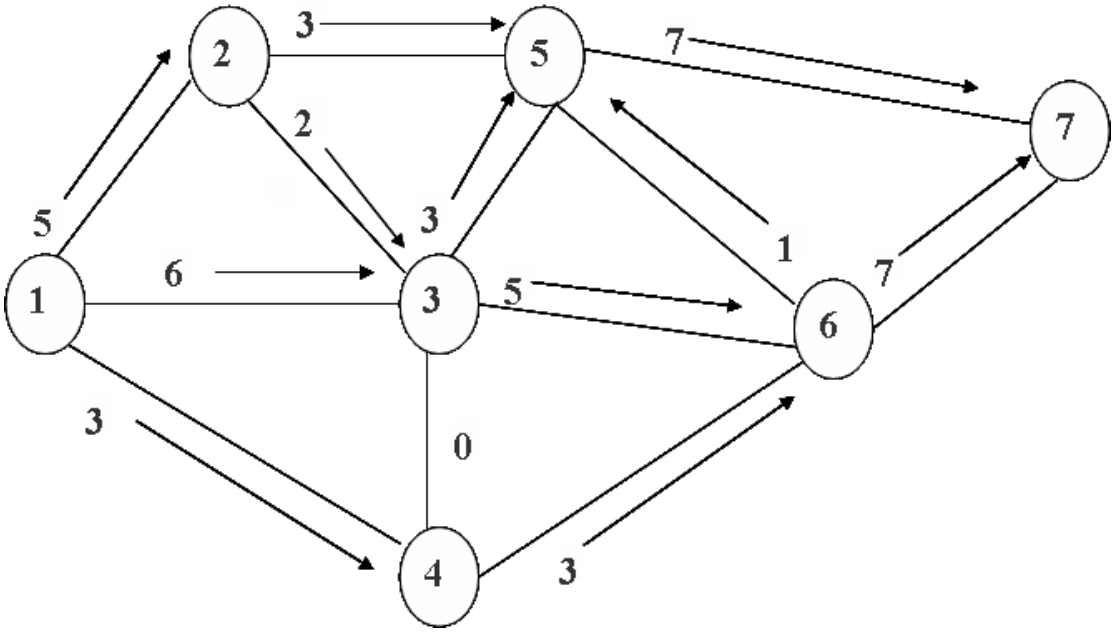


图 7-8

3．算法的具体证明过程(选看)

(1)残量网络

为了更方便算法的实现，一般根据原网络定义一个残量网络，其中 r(u,v)表示残量

网络的容量。r(u,v) = c(u,v) – f(u,v),通俗的讲：就是对于某一条边，还能再有多少流量经过。Gf 表示残量网络，Ef 表示残量网络的边集。

从残量网络中，我们就可以清楚的看到从一个顶点到另外一个顶点还可以有多少的流量。增广路算法的每一次增广实际就是在残量网络中找到一条从起点到终点的路径。

(2)割

一个割(S,T)由两个点集组成,S 和 T 交集为空，且 S 和 T 的并集为 V,s 属于 S,t 属于 T

在下面的证明过程中，X,Y,S,T 都是表示点集

f(X,Y) = $\sum_{x\in X}\sum_{y\in Y} f(x,y)$ ，即：X 中的任意一点与 Y 中的任意一点组成的所有边上的流量之和.(边的方向为从 X 中的结点到 Y 中的结点)

c,r 等函数都有类似的定义.(点集间的容量和、点集间的残量网络容量和)

结论一：

1. f(X,X) = 0　　(由流量反对称性)
2. f(X,Y) = -f(Y,X)　　(由流量反对称性)
3. f(X ∪ Y,Z) = f(X,Z) + f(Y,Z)　　(显然)
4. f(X,Y ∪ Z) = f(X,Y) + f(X,Z)　　(显然)

结论二：

不包含 s 和 t 的点集,于它相关联的边上的流量之和为 0.

证明：

f(X,V) = $\sum_{x\in X}[\sum_{y\in Y} f(x,y)]$

= $\sum_{x\in X}[0]$　　（由流量平衡定理）

= 0

结论三：

任意割的流量等于整个网络的流量.

证明:

f(S,T) = f(S,V) - f(S,S) (由结论 1)

= f(S,V)　　(由结论 1)

= f(S,V) + f(S - s,V)　(同上)

= f(s,V)　　(由结论 2)

= |f|　　(由|f|的定义)

结论四：

网络的流量小于等于任意一个割的容量.(注意这个与结论 3 的区别.这里是容量)

即|f| <= c(S,T)

证明：|f| = f(S,T) = $\sum_{x\in S}\sum_{y\in T} f(x,y)$　　(由定义)

$\sum_{x\in S}\sum_{y\in T} c(x,y)$

<= （由流量限制）

= c(S,T)

定理：

网络流中这三个条件等价（在同一个时刻）：

1、f 是最大流

2、残量网络中找不到增广路径

3、|f| = c(S,T)

证明：

1 -> 2 证明：显然.假设有增广路径,由于增广路径的容量至少为 1,所以用这个增广路径增广过后的流的流量肯定要比 f 的大,这与 f 是最大流矛盾.

2-> 3 证明：定义 S = s ∪ {v | 在残量网络中 s 到 v 有一条路径}；T = V- S. 则(S,T)是一个割.|f| = f(S,T) (结论 3)而且,r(S,T) = 0. 假设不为 0,则在残量网络中, 两个集合间必定有边相连,设在 S 的一端为 v,在 T 的一端为 u. 那么,s 就可以通过 v 到达 u,那么根据 S 的定义,u 就应该在 S 中.矛盾. 所以,|f| = f(S,T) = c(S,T) - r(S,T) = c(S,T)

3 -> 1 证明：|f| <= c(S,T) (结论 4)

因为我们已经有|f| = c(S,T),如果最大流的流量是|f|+d (d>0),那么|f|+d 肯定不能满足上面的条件.如果 最大流最小割定理不能从 2 推出 3,那么存在这样一种可能性：

尽管找不到增广路径了,但由于前面的错误决策,导致 f 还没有到达最大流,却不能通过修改当前流来得到最大流.

但由于最大流最小割定理的三个条件互相等价(1->2,2->3,3->1), 一个流是最大流当且仅当它没有增广路径.

增广路的 C++实现如下：

```
int maxflow(int s,int e)
{
    memset(pre,-1,sizeof(pre));
    memset(visited,false,sizeof(visited));
    queue<int> que;
    que.push(s);
    visited[s]=true;
    while(!que.empty())
    {
        int u=que.front();
        que.pop();
        for (int i=0;i<n;i++)
            if (g[u][i]!=0&&!visited[i])
            {
                visited[i]=true;
                pre[i]=u;
                que.push(i);
            }
```

```
}
while(pre[e]!=-1)
{
    int min=MAXINT;
    int k=e;
    while(k!=s)
    {
        min=g[pre[k]][k]<min? g[pre[k]][k] : min;
        if (g[k][pre[k]]!=0)
            min=g[k][pre[k]]<min? g[k][pre[k]] : min;
        k=pre[k];
    }
    k=e;
    while(k!=s)
    {
        f[pre[k]][k]+=min;
        g[pre[k]][k]-=min;
        if (f[k][pre[k]]!=0)
        {
            f[k][pre[k]]-=min;
            g[k][pre[k]]+=min;
        }
        k=pre[k];
    }
    memset(pre,-1,sizeof(pre));
    memset(visited,false,sizeof(visited));

    while(!que.empty()) que.pop();
    que.push(s);
    visited[s]=true;
    while(!que.empty())
    {
        int u=que.front();
        que.pop();
        for (int i=0;i<n;i++)
            if (g[u][i]!=0&&!visited[i])
            {
                visited[i]=true;
                pre[i]=u;
                que.push(i);
            }
    }
}
```

```
    int flow=0;
    for (int i=0;i<n;i++)
        if (map[s][i]!=0)
            flow+=f[s][i];
    return flow;
}
```

补充：

由上面的证明过程，有向图的最小割可以通过转化为求最大流的方法来实现，对于无向图，求最小流的方法如何。

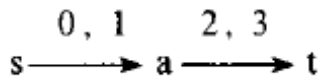
7.3 有上下界网络的最大流和最小流

设网络 D = (V,A,B,C)。其中 A 中的每条弧对应两个数字 B(e)和 C(e),分别表示弧容量的上界和下界。求满足下列条件的最大流：

1. B(e) <= F(e) <= C(e)
2. $\sum_{e\in\beta(V)} F(e) - \sum_{e\in\alpha(V)} F(e) = 0 \quad (v \neq s, t)$

(注：$\alpha(V)$是以 V 为结尾的弧集，$\beta(V)$是以 V 为头的弧集)

算法分析：增广路法例举的网络结构是有上下界的网络的一种特例，即 B(e) = 0。当 B(e) ＞0 时，这种有上下界的网络就不易定存在流了。如图所示的网络，弧上的第一个数字为 B(e),第二个数字为 C(e)。由于 0 <= f(s,a) <= 1， 2 <= f(a,t) <= 3.对于顶点 a 来讲，F(a,t) 与 f(s,a)不可能相等，所以该网络不存在流。



那么如何判断一个有上下界的网络 N 有颗星流呢？我们容易想到一种思路- ----将容量有上下界的网络转换成”每条弧仅对应一个容量 C(e) >= 0”的附加网 N.然后用增广路法对其求最大流。根据求解结果判断 N 网是否有可行流，具体的转换方法如下：

(1)新增加两个顶点$\overline{s}$和$\overline{t}$，$\overline{s}$为附加源，$\overline{t}$为附加汇

(2)对原网络每个顶点 U,加一条新弧$e=U\overline{t}$，这条弧的容量为所有以 U 为结尾的弧的容量的下限之和，即$C(e)=\sum\overline{B}(e)$，其中$e \in \alpha(U)$

(3)对原网络 N 的每个顶点 U,加一条新弧，$e=\overline{s}U$，这条弧的容量为所有以 U 为开头的弧的容量下限之和，即$\overline{C}(e)=\sum B(e),\ e \in \beta(U)$

原网络的弧在 N 中仍然保留，弧容量$\overline{C}(e)$修正为$C(e)-B(e)$

(4)再添两条新弧$e=st,\ e'=ts$,e 的容量$\overline{C}(e)=\infty$，e‘的容量修正为

$\overline{C}(e') = \infty$。

用标号法求附加网 N 的最大流，若结果能使流出$\overline{S}$的一切弧都满载，则 N 网有可行流$F(e)=\overline{F}(e)+B(e)$。否则 N 网无可行流。

若$\overline{N}$网的最大流满足上述条件，我们则可以使用增广路法，将可行流 F 放大，最终求出 N 网的最大流。

举例如下：图示 7.12 是一个容量有上下界的网络，弧的第一个数为 B(e),第二个数为 C(e).

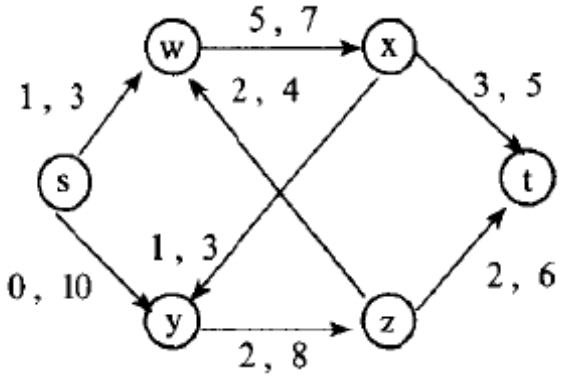


图 7-9

作附加网络 $\overline{N}$，用增广路法求 $\overline{N}$网上的最大流$\overline{F}$，如图 7.13

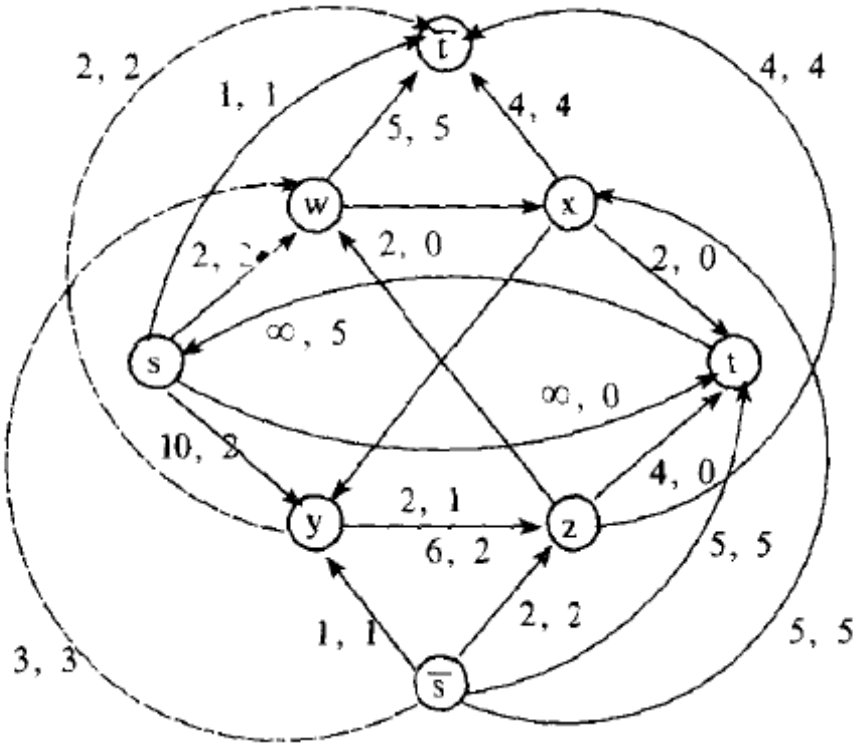


图 7-10

弧上的第一个数为$\overline{C}(e)$，第二个数为$\overline{F}(e)$。由于流出$\overline{S}$的弧皆满载，可

以得出原图网络 N 中有可行流 $F(e)=\overline{F}(e)+B(e)$。图 7.14 将 $\overline{F}$ 转换为 N 网的可行流，弧上的三个数字依次为 B(e),C(e)和 F(e)。

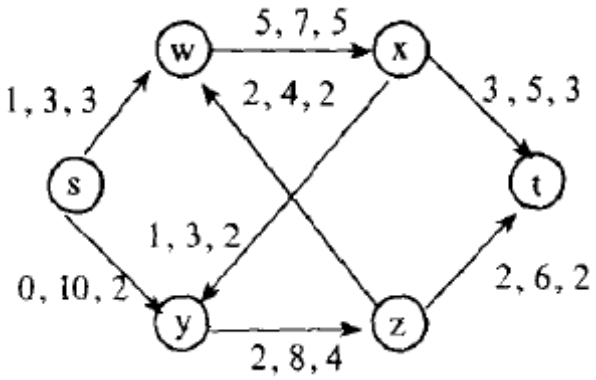


图 7-11

用增广路法将上述可行流放大，得到最大流 F.图示如 7.15：

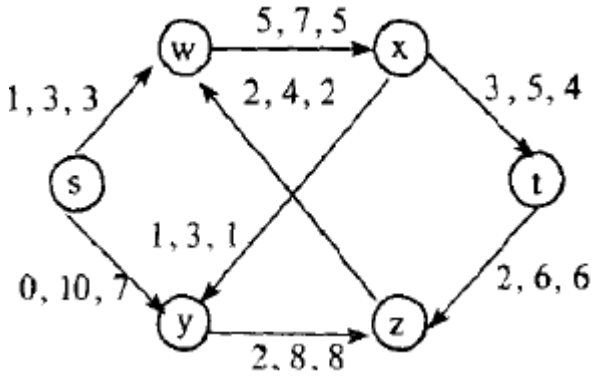


图 7-12

容量有上下界的网络最大流的 C++实现如下：

```
    void _max_flow(int n,int mat[][MAXN],int source,int sink,int flow[][MAXN])
{
    int pre[MAXN],que[MAXN],d[MAXN],p,q,t,i,j;
    do
    {
        for (i=0;i<n;pre[i++]=0);
        pre[t=source]=source+1,d[t]=inf;
        for (p=q=0;p<=q&&!pre[sink];t=que[p++])
            for (i=0;i<n;i++)
                if (!pre[i]&&mat[t][i]&&(j=mat[t][i]-flow[t][i])>0)
                    pre[que[q++]=i]=t+1,d[i]=d[t]<j?d[t]:j;
                else
                    if (!pre[i]&&mat[i][t]&&(j=flow[i][t])>0)
                        pre[que[q++]=i]=-t-1,d[i]=d[t]<j?d[t]:j;
        for (i=sink;pre[i]&&i!=source;)
            if (pre[i]>0)
                flow[pre[i]-1][i]+=d[sink],i=pre[i]-1;
```

```
                else
                    flow[i][-pre[i]-1]-=d[sink],i=-pre[i]-1;
        }
        while (pre[sink]);
    }

    int limit_max_flow(int n,int mat[][MAXN],int bf[][MAXN],int source,int sink,int
flow[][MAXN])
    {
        int i,j,sk,ks;
        if (source==sink)
            return inf;
        for (mat[n][n+1]=mat[n+1][n]=mat[n][n]=mat[n+1][n+1]=i=0;i<n;i++)
            for (mat[n][i]=mat[i][n]=mat[n+1][i]=mat[i][n+1]=j=0;j<n;j++)
                mat[i][j]-=bf[i][j],mat[n][i]+=bf[j][i],mat[i][n+1]+=bf[i][j];
        sk=mat[source][sink],ks=mat[sink][source];
        mat[source][sink]=mat[sink][source]=inf;
        for (i=0;i<n+2;i++)
            for (j=0;j<n+2;flow[i][j++]=0);
        _max_flow(n+2,mat,n,n+1,flow);
        for (i=0;i<n;i++)
            if (flow[n][i]<mat[n][i])
                return -1;
        flow[source][sink]=flow[sink][source]=0;
        mat[source][sink]=sk,mat[sink][source]=ks;
        _max_flow(n,mat,source,sink,flow);
        for (i=0;i<n;i++)
            for (j=0;j<n;j++)
                mat[i][j]+=bf[i][j],flow[i][j]+=bf[i][j];
        for (j=i=0;i<n;j+=flow[source][i++]);
        return j;
    }
```

补充：容量有上下界的网络的最小流

同样：由于下界的存在，容量有上下界的网络也存在最小流，程序实现如下：

```
    void _max_flow(int n,int mat[][MAXN],int source,int sink,int flow[][MAXN])
    {
        int pre[MAXN],que[MAXN],d[MAXN],p,q,t,i,j;
        do{
            for (i=0;i<n;pre[i++]=0);
            pre[t=source]=source+1,d[t]=inf;
            for (p=q=0;p<=q&&!pre[sink];t=que[p++])
                for (i=0;i<n;i++)
                    if (!pre[i]&&mat[t][i]&&(j=mat[t][i]-flow[t][i])>0)
                        pre[que[q++]=i]=t+1,d[i]=d[t]<j?d[t]:j;
```

```
                else if (!pre[i]&&mat[i][t]&&(j=flow[i][t])>0)
                    pre[que[q++]=i]=-t-1,d[i]=d[t]<j?d[t]:j;
            for (i=sink;pre[i]&&i!=source;)
                if (pre[i]>0)
                    flow[pre[i]-1][i]+=d[sink],i=pre[i]-1;
                else
                    flow[i][-pre[i]-1]-=d[sink],i=-pre[i]-1;
        }
        while (pre[sink]);
    }

    int limit_max_flow(int n,int mat[][MAXN],int bf[][MAXN],int source,int sink,int
flow[][MAXN])
    {
        int i,j,sk,ks;
        if (source==sink)
            return inf;
        for (mat[n][n+1]=mat[n+1][n]=mat[n][n]=mat[n+1][n+1]=i=0;i<n;i++)
            for (mat[n][i]=mat[i][n]=mat[n+1][i]=mat[i][n+1]=j=0;j<n;j++)
                mat[i][j]-=bf[i][j],mat[n][i]+=bf[j][i],mat[i][n+1]+=bf[i][j];
        sk=mat[source][sink],ks=mat[sink][source];
        mat[source][sink]=mat[sink][source]=inf;
        for (i=0;i<n+2;i++)
            for (j=0;j<n+2;flow[i][j++]=0);
        _max_flow(n+2,mat,n,n+1,flow);
        for (i=0;i<n;i++)
            if (flow[n][i]<mat[n][i])
                return -1;
        flow[source][sink]=flow[sink][source]=0;
        mat[source][sink]=sk,mat[sink][source]=ks;
        _max_flow(n,mat,source,sink,flow);
        for (i=0;i<n;i++)
            for (j=0;j<n;j++)
                mat[i][j]+=bf[i][j],flow[i][j]+=bf[i][j];
        for (j=i=0;i<n;j+=flow[source][i++]);
        return j;
    }
```

7.4 网络流的预流推进算法

前面两节所讲的网络流都是基于增广路算法，本节内容是网络流的预流推进算法，其效率比起增广路算法有明显的改进。

如果给你一个网络流,让你手算出它的最大流,你会怎么算?

一般人都会尝试着从源点出发,让每条边的流量尽可能得大,然后一点点往汇点推,直到遇到一条比较窄的弧,原先的流量过不去了,这才减少原先的流量.

一个直观的想法

大致的思路：从源点出发，逐步推进。

称当前状态下不满足流量平衡的结点为“溢出的结点”.(对于结点 u,f(V,u) > 0)

令 e(u) = f(V,u),称为 u 点的赢余，直观地描述，就是“流入的比流出的多多少”。

e(v1)=4,e(v2)=3。不断将溢出的结点中的赢余往后继点推进,直到赢余都聚集在 t.

如果多推了一些流量, 我们可以再把它推回来. (如 e(v2)=3,但这 3 个单位的赢余已经没地方去了,只能推回来.)(沿着后向弧)这副图是原网络而不是残量网络,因此没把后项弧画出来)

此时 e(v2)=3.正确的回推法是往(v2,s)推 1,往(v2,v1)推 2,然后使得这 2 个单位的赢余可以从(v1,t)推到 t 上。

但程序没有全局观,它万一往(v2,s)推了 3 个单位怎么办?我们总不能尝试所有的可能性吧,那样就变成搜索了.

为了解决上面的问题，我们得出了一个引导机制：

把流推错可能导致产生的流不是最大流.

我们需要有一个能引导流的推进方向的机制,当它发现我们先前的推进是错误的时候,能沿着正确的后向弧回推回来.

由于建立了后向弧,正推与回推在程序中并无却别，都是在推残量网络中的一条边.

高度标号

高度标号就是这样的一个引导机制.

我们规定,如果一个结点溢出了,那么他的多余的流量只能流向高度标号比自己低的结点.(“水往低处流”)

当然,高度标号不可能事先知道往哪些方向推才是正确的.它将按情况动态改变自己的值,从而正确地引导流向.

重标号处理

当一个结点有赢余(溢出了), 周围却没有高度比它低的结点时候,我们就用重标号操作使它的标号上升到比周围最低的结点略高一点,使他的赢余能流出去.

赢余千万不能困在某个结点里.对于任意一个非源非汇的结点，有赢余就意味着它不满足流量平衡，也就意味着整个网络流不是一个真正合法的网络流。

对于上例的这种情况,v2 中过多的赢余最终会沿着(v2,v1)、(v2,s)流回去(虽然他们一开始流错了方向,但后来又被回推,等于说是被改正了)。

只有当非源非汇的结点中的赢余全部流到汇点或流回源点后，这个流才重新合法。

高度函数

高度函数 h(v)返回一个 v 的高度标号。

高度函数有三个基本条件：

h(s) = |V|

h(t) = 0

对于 Ef(残量网络)中的每一条边(u,v),(r(u,v)>0)

h(u) <= h(v) + 1

这第三个条件看上去有些奇怪.既然 r(u,v)>0,那就表示从 u 到 v 还可以增加流量,那 h(u)就应该比 h(v)高才对.的确,我们后面还将规定,只有在 h(u)>h(v)的时候才能应用推进操作(将一个结点的盈余推进到另一个结点的操作).而高度函数为了满足其合法性,还要满足上述的这三个条件.后面我们将利用这三个条件证明预流推进算法的正确性。

h(u) <= h(v)+1.这个条件实质上是要求高度不能下降的太快，即水只能在高度相差不多的地方缓缓流过，不能像瀑布一样从很高的地方流到很低的地方。(否则就有流错的危险)这和 A*算法中的启发函数必须“相容”的条件类似。h 函数的缓慢下降，保证了算法的正确性。后面我们将看到这个条件的作用.

两个关键操作

推进操作(将一个结点的盈余推到另一个结点)

重标号操作(更改一个结点的高度值,使其的盈余能朝着更多的地方流动)

推进操作

使用对象：一条边(u,v)

使用条件：

e(u)>0，r(u,v)>0, h(u) = h(v)+1

（u 溢出，(u,v)在残量网络中，两者的高度差为 1）

推进量为 e(u)与 r(u,v)的最小值。

推进时同时更改相关的 r 与 e 的值。

推进操作 伪代码

Procedure Push(u,v)

X ß min{ e(u), r(u,v) }

Dec(r(u,v), x) Inc(r(v,u), x)

Dec(e(u), x) Inc(e(v), x)

重标号操作

使用对象： 一个结点 u

使用条件：

结点 u 溢出；残量网络中周围所有的点的高度都不比它低。

Relabel(u)

u(u) = min{ h(v) | (u,v)是残量网络总的边 } + 1

使用了重标号操作后,至少存在一个(u,v)满足 h(u)=h(v)+1.

预流初始化

一开始的时候，我们要让和源点 s 相关连的边都尽可能的充满。但由于 s 没有溢出，不符合推进操作的使用条件，我们需要另写一段初始化的代码。还得做的一件事是初始化高度函数.

h(s) = n h(v) = 0 (v<>s)

对于所有与 s 相关联的点 v，

Inc(e(v), c(s,v)), Dec(e(s), c(s,v))

将边(s,v)反向，变成(v,s) （在残量网络中）。

初始化过后，e(s)变成负数。

对于一个溢出的结点，两个关键操作（推进和重标号）能且只能应用一个。

证明：对于一个溢出的结点 u,和所有与他相关联的点 v((u,v)在残量网络中存在),必然有 h(u) <= h(v) + 1.(由高度函数的定义).

根据 v 分成两种情况:1).所有 v 都有 h(u)<h(v)+1 2).至少存在一个 v,使得 h(u)=h(v)+1.

而 1)2)互为否命题,不能同时成立或同时不成立.那么 1)对应重标号,2)对应推进,两者必能应用一个且只能应用一个.

程序框架

由上述结论,得到了一个一般的预流推进算法.(好短)

Init-Preflow

While 存在一个溢出的结点
选一个结点,应用相应的关键操作(推进或重标号).
当不存在溢出结点时(s,t 不算),算法结束,得到一个可行流,并且还是最大流.

预流推进的算法的代码如下：

```
/*
    Title : Pre-Label MaxFlow For Problem 1228, Power Network
#include <stdio.h>
#include <string.h>
//-----------------Max Flow---------------------//
/*
最大流 Algo : RelabelToFront Preflow
main()里要做的事：
1) 定义类 Graph 的实例 G。
2) 给边容量矩阵 G.c[][]赋值,若无边，赋为
3) 把顶点的个数赋给 G.V。
4) 调用 G.MaxFlow(s, t)，返回最大流值。同时由 G.f[][]可知流值分配。
*/
const int MAX = 405;              //最多结点数量
class Graph
{
public:
    //构造图结构时只需初始化 V 域和 c 域即可。
    int V;
    int c[MAX][MAX];    //边容量矩阵,若无边，则为
    int f[MAX][MAX];    //流矩阵
    int maxflow;
    int MaxFlow(int , int);
private:
    struct flownode
    {
        int adjnum;
        flownode * next;
    };
    int e[MAX];         //过剩容量数组
    int h[MAX];                 //Height 数组
    flownode * N[MAX];          //Neighbour lists
    //
    void InitializePreflow(int);
    void PushPreflow(int, int);
    void RelabelPreflow(int);
    void DischargePreflow(int);
    //
    void listInsert(flownode * , int);
```

```
        void listErase(flownode *);
        void listDestroy(flownode * &);
        void neighborDestroy();
};
void Graph::InitializePreflow(int s)
{
        int u;
        flownode *itr;
        for(u = 0; u < V; ++u)
        {
                h[u] = 0;
                e[u] = 0;
        }
        memset(f,0,sizeof(int) * MAX * V);
        h[s] = V;
        //
        for(itr = N[s]->next; itr; itr = itr->next)
        {
                u = itr->adjnum;
                f[s][u] = c[s][u];
                f[u][s] = -c[s][u];
                e[u] = c[s][u];
                e[s] -= c[s][u];
        }
}
void Graph::PushPreflow(int u, int v)
{
        //Applies when: u is overflowing, cf[u][v]>0, and h[u] = h[v]
        +1.
                //cf[u][v] = c[u][v] - f[u][v];
                //Action: Push df[u][v] = min{e[u], cf[u][v]) units of flow from u to v
                int Cf = c[u][v] - f[u][v];
        int Df = e[u] < Cf ? e[u] : Cf;
        f[u][v] += Df;
        f[v][u] = -f[u][v];
        e[u] -= Df;
        e[v] += Df;
}
void Graph::RelabelPreflow(int u)
{
        //Applies when: u is overflowing and for all v in V such that
        (u,v) is in Ef,
                //                      we have h[u] <= h[v].
                //Action: h[u] = 1 + min{h[v] : (u,v) in Ef
```

```
            int min = V * 2;
        flownode *itr;
        int w;
        for( itr = N[u]->next; itr; itr = itr->next)
        {
            w = itr->adjnum;
            if( (c[u][w] > f[u][w] ) && (min > h[w]) )
                min = h[w];
        }
        h[u] = 1 + min;
}
void Graph::DischargePreflow(int u)
{
        int v;
        flownode *current = N[u]->next;
        while(e[u] > 0)
        {
            if( !current )
            {
                RelabelPreflow(u);
                current = N[u]->next;
            }else
            {
                v = current->adjnum;
                if( (c[u][v] > f[u][v]) && (h[u] == h[v] + 1))
                    PushPreflow(u, v);
                else current=current->next;
            }
        }
}
void Graph::listInsert(flownode * head, int i)
{
        flownode * p;
        p = new flownode;
        p->adjnum = i;
        p->next = head->next;
        head->next = p;
}
void Graph::listErase(flownode * p)
{
        flownode * q;
        q = p->next;
        p->next = q->next;
        delete q;
```

```
}
void Graph::listDestroy(flownode * & head)
{
    flownode * q;
    while(head->next)
    {
        q = head->next;
        head->next = q->next;
        delete q;
    }
    delete head;
    head = NULL;
}
void Graph::neighborDestroy()
{
    for(int i = 0; i<V; ++i)
        listDestroy(N[i]);
}
int Graph::MaxFlow(int s, int t)
{
    flownode *L;
    flownode *itr;
    int i, u, oldheight;
    //构造每个结点的 Neightbour list;
    for(u = 0; u < V; ++u)
    {
        N[u] = new flownode;
        N[u]->next = NULL;
    }
    for(u = 0; u < V; ++u)
    {
        for(i = 0; i < u; ++i)
            if( c[u][i] || c[i][u] )
            {
                listInsert(N[u],i);
                listInsert(N[i],u);
            }
    }
    //构造 L
    L = new flownode;
    L ->next = NULL;
    for( i = 0; i < V; ++i)
    {
        if(i == s || i == t) continue;
```

```
            listInsert(L,i);
        }
        //初始化 preflow
        InitializePreflow(s);
        //

        for(itr = L; itr->next; itr = itr->next )
        {
            u = itr->next->adjnum;
            oldheight = h[u];
            DischargePreflow(u);
            if(h[u] > oldheight)
            {
                listErase(itr);
                listInsert(L,u);
                itr = L;
            }
        }
        listDestroy(L);
        maxflow = 0;
        for( itr = N[s]->next; itr; itr = itr->next)
        {
            if(f[s][itr->adjnum]    )
                maxflow += f[s][itr->adjnum];
        }
        neighborDestroy();
        return maxflow;
}
//----------------------Max Flow----------------------//
Graph G;
int main()
{
        int V,e,i,cap;
        int p,c,a,b,ans;
        char ch;

        //      freopen("in.txt","r",stdin);
        //      freopen("e:\\1228.in","r",stdin);
        //      freopen("e:\\ss2.out","w",stdout);

        while(scanf("%d %d %d %d",&V,&p,&c,&e)==4)
        {
            G.V = V+2;
            memset(G.c, 0 ,sizeof(int) * MAX * G.V);
```

```
            for(i = 0; i < e; ++i)
            {
                while((ch=getchar()) !='(');
                scanf("%d,%d)%d",&a,&b,&cap);
                G.c[a][b] = cap;
            }
            for(i = 0; i < p; ++i)
            {
                while((ch=getchar()) !='(');
                scanf("%d)%d",&a, &cap);
                G.c[V][a] = cap;
            }
            for(i = 0; i < c; ++i)
            {
                while((ch=getchar()) !='(');
                scanf("%d)%d",&a, &cap);
                G.c[a][V+1] = cap;
            }
            ans=G.MaxFlow(V,V+1);
            printf("%d\n",ans);
        }
        return 0;
    }
```

7.5 最小费用最大流

上面的问题都是讨论流的问题，但是现实生活中的好多问题都和费用有关，在最大流的前提下，要怎样才能使得费用最少。

求最小费用最大流通常有两种方法。用 dijakstra 算法来代入上面的增广路径。第二种是晓负环算法。

先介绍第一种算法：在增广路算法的时候，我们在找增广路算法用的是宽度搜索，或者为了提高效率，我们使用的类似 Dijkstra 的迭代寻找。但是在最小费用算法里面，为了保证所选的道路是费用最低的，我们有了限制，所以现在增广的原则变成了每次选择费用最低的增广路径。

而选择费用最低的增广路径就类似 Dijkstra 算法，我们就是在这个基础上进行了稍微的修改，就是在每个节点不是求到源点的最短距离，而是求到源点的所有路径中，费用最高的那条边最低的路径最为权值进行迭代。

第八章 动态规划

8.1 动态规划简介

一、动态规划的基本思想：

关键：状态, 阶段, 决策, 划分子问题, 状态转移方程

动态规划算法通常用于求解具有某种最优性质的问题。在这类问题中，可能会有许多可行解。每一个解都对应于一个值，我们希望找到具有最优值的解。动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解。与分治法不同的是，适合于用动态规划求解的问题，经分解得到子问题往往不是互相独立的。若用分治法来解这类问题，则分解得到的子问题数目太多，有些子问题被重复计算了很多次。如果我们能够保存已解决的子问题的答案，而在需要时再找出已求得的答案，这样就可以避免大量的重复计算，节省时间。我们可以用一个表来记录所有已解的子问题的答案。不管该子问题以后是否被用到，只要它被计算过，就将其结果填入表中。这就是动态规划法的基本思路。具体的动态规划算法多种多样，但它们具有相同的填表格式。

二、设计动态规划法的步骤：

1．找出最优解的性质，并刻画其结构特征；

2．递归地定义最优值（写出动态规划方程）；

3．以自底向上的方式计算出最优值；

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，步骤 3 中记录的信息也较少；若需要求出问题的一个最优解，则必须执行步骤 4，步骤 3 中记录的信息必须足够多以便构造最优解。

三、动态规划问题的特征：

动态规划算法的有效性依赖于问题本身所具有的两个重要性质：最优子结构性质和子问题重叠性质。

1．最优子结构：当问题的最优解包含了其子问题的最优解时，称该问题具有最优子结构性质。

2．重叠子问题：在用递归算法自顶向下解问题时，每次产生的子问题并不总是新问题，有些子问题被反复计算多次。动态规划算法正是利用了这种子问题的重叠性质，对每一个子问题只解一次，而后将其解保存在一个表格中，在以后尽可能多地利用这些子问题的解。

四、应用动态规划解题的条件：

1．满足最优化原则

2．满足无后效性

8.2 动态规划例题

Pigy Bank (POJ 1384)

题意简述：输入 T 组测试，然后输入猪仔空的时候的 E 重量，和装满时的重量 F,再输入 N 中硬币种类，输入硬币的面值 P 和一个硬币的重量，求可达到规定质量的最小硬币数量。

思路：首先考个数组 calc[i]记录总的金额，i 代表不同重量时候的最优方案，由 1 遍历到 MaxWeight，记录不同金额的最优值状态转移方程：calc[i] = min{calc[temp] + coin[j].den};开始想到的是开一个二维数组来保存硬币的个数，然后计算 calc[505][10005]，然后计算出最小的总值就可以了，但是提交上去内存超了，然后把 int 改成 usigned int 我计算一下内存应该为 10005 * 505 * 2 = 9868k < 10000 k还是超内存这时我发现应该poj 的编译器usigned int 应该也是占4位的，无奈只好改一下代码了。

代码：

```
#include <stdio.h>
#include <string.h>
#define INF 0x11111111
int bag[10010],w[501],v[501],n;

int main()
{
  int i,j,k,t,a,b,tot;
  scanf("%d",&t);
  while (t--)
  {
        memset(bag,0x11,sizeof(bag));
        scanf("%d%d%d",&a,&b,&n);
        for (i=0;i<n;i++)
              scanf("%d%d",v+i,w+i);
        tot=b-a;
        if (tot<0)
              tot=-tot;
        bag[0]=0;
        for (i=0;i<n;i++)
              for (j=0;j<=tot;j++)
                    if (bag[j]!=INF && j+w[i]<=tot && bag[j+w[i]]>bag[j]+v[i])
                          bag[j+w[i]]=bag[j]+v[i];
        if (bag[tot]!=INF)
        printf("The minimum amount of money in the piggy-bank is %d.\n",bag[tot]);
        else
              puts("This is impossible.");
  }
  return 0;
}
```

Traveling in Solar System （ZOJ 1161）

题意简述：3114 年，Bernie 打算带妹妹 Rosie 去太阳系旅游。因此他向旅游部门要了关于这个夏天旅游地点的列表。Rosie 想在她 3 个月的假期内玩遍太阳系，但是 Bernie 却没有足够的钱。于是 Bernie 问了 Rosie 哪些地点她比较喜欢，Rosie 就给出了她对单子上所有地方的喜好程度。你作为 Bernie 最好的朋友，他希望你能帮他安排一个合理的计划，充分利用他的钱并且让他妹妹满意。

思路：枚举走过的湖的个数 x,最终将在湖 x 停下来，那么在走路上消耗的时间可以算出来 t[1]+t[2]+...+t[n-1],在这个前提下，我们可以从一个湖瞬移到另外一个湖（这个是关键!!!）。为了掉的更多的鱼，我们每次都选择瞬移到与最多的湖去钓 5 分钟（可以通过优先队列来实现）,这样得到的鱼肯定是最多的。

代码：

```
#include<stdio.h>
#define MAX_DATA 100
#define MAX_MONEY 5000
int w[MAX_DATA], v[MAX_DATA], state[MAX_MONEY + 1], prev[MAX_MONEY + 1], newadd[MAX_MONEY + 1];
char temp[100];
inline bool OnResult(int idx, int & key)
{
  int p = idx;
  while(p != -1)
  {
        if(newadd[p] == key) return true;
        p = prev[p];
  }
  return false;
}
int main()
{
  int n, s, ss, i, j, k, money, cnt, maxp, tcost, t;
  scanf("%d", &n);
  for(i = 0; i < n; ++i)
  {
        cnt = 0;
        scanf("%d RMB", &money);
        scanf("%d", &s);
        for(j = 0; j < s; ++j)
        {
              scanf("%s%d", temp, &ss);
              for(k = 0; k < ss; ++k) scanf("%d days %d RMB", &t, &w[cnt++]);
        }
        for(j = 0; j < cnt; ++j) scanf("%d", &v[j]);
        // 初始化
```

```
        for(j = 0; j <= money; ++j)
        {
            state[j] = 0;
            newadd[j] = prev[j] = -1;
        }
        tcost = maxp = 0;
        // 0、1 背包问题（稍有不同）
        for(j = 1; j <= money; ++j)
        {
            state[j] = state[j - 1];
            prev[j] = prev[j - 1];
            newadd[j] = newadd[j - 1];
            for(k = 0; k < cnt; ++k)
            {
                if(j >= w[k] && state[j] < state[j - w[k]] + v[k])
                {
                    if(OnResult(j - w[k], k)) continue;
                    state[j] = state[j - w[k]] + v[k];
                    if(maxp < state[j])
                    {
                        maxp = state[j];
                        tcost = j;
                    }
                    prev[j] = j - w[k];
                    newadd[j] = k;
                }
            }
        }
        printf("%d %d\n", tcost, maxp);
    }
    return 1;
}
```

Dividing up(HIT1141

题意简述：给你 6 堆石子各自的石头数，第 i 堆石头每块重量为 i，问能否把所有石头分成 2 堆，使其重量相等？石头总数最大为２０００００.

思路：这个题很容易想到动态规划的做法，令 S=∑(i*a[i])，若 S 为奇数，则不可能实现，否则令 Mid=S/2，则问题转化为能否从给定的大理石中选取部分大理石，使其价值和为 Mid。m[i, j]，0≤i≤6，0≤j≤Mid，表示能否从价值为 1..i 的大理石中选出部分大理石，使其价值和为 j，若能，则用 true 表示，否则用 false 表示。则状态转移方程为:

m[i, j]=m[i, j] OR m[i-1,j-i*k] (0≤k≤a[i])

规划的边界条件为：m[i,0]=true； 0≤i≤6

若 m[i, Mid]=true，0≤i≤6，则可以实现题目要求，否则不可能实现。

时间复杂度 ６＊２０００００＊６００００

能否有更优的动态规划做法呢？先来看上述做法在什么地方能够更快．状态能更少？转移能否更少？

m[i, j]=m[i, j] OR m[i-1,j-i*k] (0≤k≤a[i])

当 k 由 x 变成 x ＋ 1 时没有用到 k ＝ x 的结论，我们是否能在这个地方做做文章呢？状态与上述一样，增加一个状态向量 usednum[][] 。usednum[i][x]=y 表示第 i 堆石头最少用了 y 块能达到重量 x 状态转移如下

m[i][j]= m[i][j] or a[i]>=usednum[i][j]>0

usednum[i][j]= min{usednum[i][j], if (m[i-1][j-k]) 1, use[i][j-k]+1}

这里的 k 是第 i 堆石头每块的重量

时间复杂度 6*60000

其实这一类动态规划问题都可以通过增加 usednum[][]状态向量实现优化，而这类问题的特点是有很多”堆”,每“堆”中的东西是没有区别的。

代码：

```
#include<cstdio>
#include<cstring>
int main()
{
  int stone[7];
  int cas=0;
  while(true)
  {
        cas++;
        int i,j,k,sum=0;
        for(i=1;i<7;i++)
        {
              scanf("%d",&stone[i]);
              sum+=i*stone[i];
        }
        if(sum==0) break;
        printf("Collection #%d:\n",cas);
        if(sum%2!=0)
        {
              printf("Can't be divided.\n\n");
              continue;
        }
        int half=sum/2;
        bool dp[65536];
        int usednum[65536];
        memset(dp,false,sizeof(dp));
        dp[0]=true;
        int up=0;
        for(k=1;k<7;k++)
        if(stone[k]>0)
        {
```

```
                if(dp[half]) break;
                memset(usednum,0,sizeof(usednum));
                for(i=0;i<=up;i++)
                {
                    if(dp[i]) {
                        j=i+k;
                        if(j>half) break;
                        if(!dp[j]) usednum[j]=1;
                        if(j>up) up=j;
                    }
                    else if(usednum[i]>0&&usednum[i]<stone[k])
                    {
                        j=i+k;
                        if(j>half) break;
                        if(!dp[j]) {
                            if(usednum[j]==0||usednum[i]+1<usednum[j])
                                usednum[j]=usednum[i]+1;
                        }
                        if(j>up) up=j;
                    }
                }
                    for(i=0;i<=up;i++)
                    {
                        if(usednum[i]>0) dp[i]=true;
                    }
            }
            if(dp[half]) printf("Can be divided.\n\n");
            else printf("Can't be divided.\n\n");
              }
              return 0;
        }
```

Strategic Game（ZOJ1134）

题意简述：一个点代表一个城市，如果放一个兵放在一个城市就可以控制它及与之相连的城市。问最少用多少兵控制所有的城市。输入：N----------城市的个数，这个数小于1500。城市的编号从 0 到 N-1 接下来每一行首先给出一个城市的编号，接着是与之相连的城市的个数，及相应的编号。输出：最少用多少兵控制所有的城市。

思路：本题的数学模型在题目中已明示，我们将其用数学语言重述一遍：在图 G=(V,E)中找一顶点集 S，使得对任意(u,v)∈E 有 u∈S 或 v∈S，且|S|最小。很明显，这是求图的最小顶点覆盖集。

图的最小顶点覆盖是 NP 问题，至今还没有有效算法。而本题 N 的规模可达 1500，显然用搜索是无法完成的。但本题的图十分特殊是一棵树。为了进一步利用数 据结构的特殊性，我们不妨先手工分析一个例子。左图为一个有 13 个顶点的树，其中用红色圈出的顶点为找到的最小覆盖集。很明显，叶节点不可能属于覆盖集， 但叶节

点的父节点一定属于覆盖集，以此为基础再判断上一层次节点是否属于覆盖集。左图的节点编号从大到小正好是我们判断的顺序，该顺序恰好是树的前序遍历 顺序，由此我们可得出本题的算法：

1：procedure Min-Vertex-Cover(T);
2：begin
3：　　按前序遍历顺序重新编排整棵树的节点编号。
4：　　for I:=1 to n do cover[I]:=0;
5：　　for I:=n Downto 2 do
6：　　if (cover[I]=0) and (cover[parent[I]]=0) then
7：　　cover[parent[I]]:=1;
8：end;

该算法是一个贪心算法。算法中用数组 cover 来标记选入覆盖点集的树结点，即当结点 i 被选入覆盖点集，则 cover[i]=1，否则 cover[i] =0。为了说明算法的正确性，必须证明关于树的最小顶点覆盖问题满足贪心选择性质并且具有最优子结构性质。具体的证明过程，在《若干 NP 完全问题的特殊情 形》一文中已有详细的叙述。

另外，本题的图是一棵树，很容易想到动态规划（树形 dp)

设 dp[i][0]表示节点 i 不设置在最小覆盖集的情况下以 i 为根的子树的最小覆盖集中元素的个数，dp[i][1]表示节点 i 再最小覆盖集中的情况下以 i 为根的子树的最小覆盖集中元素的个数，则状态转移方程为:

当 i 不是叶子节点时：

dp[i][0]=dp[j1][1]+dp[j2][1]+...+dp[jm][1] （j1,j2,j3,..,jm 都是 i 的子节点)

dp[i][1]=min(dp[j1][0],dp[j1][1])+min(dp[j2][0],dp[j2][1])+...+min(dp[jm][0],dp[jm][1])+1 （j1,j2,j3,..,jm 都是 i 的子节点)

当 i 是叶子节点时:

dp[i][0]=0

dp[i][1]=1

那么最后的结果就是 min(dp[root][0],dp[root][1])

代码：

```
//贪心算法
#include <stdio.h>
#include <string.h>
#define MAXN 1500

int parent[MAXN+1],covert[MAXN+1],cover[MAXN+1];
int count,n;

void preorder(int root) {
    covert[count++]=root;
    for(int i=0;i<n;i++)
        if(parent[i]==root)
            preorder(i);
}
```

```
int main() {
    int j,m,t,root;
    while(scanf("%d",&n)!=EOF) {
        for(int i=0;i<n;i++) parent[i]=-1;
        for(int i=0;i<n;i++) {
            scanf("%d:(%d)",&j,&m);
            for(int k=0;k<m;k++) {
                scanf("%d",&t);
                parent[t]=j;
            }
        }
        memset(cover,0,sizeof cover);
        count=1;
        root=-1;
        do{
            root++;
        }while(parent[root]!=-1);
        preorder(root);
        count=0;
        for(int i=n;i>=2;i--)
            if(cover[covert[i]]==0&&cover[parent[covert[i]]]==0) {
                cover[parent[covert[i]]]=1;
                count++;
            }
        printf("%d\n",count);
    }
    return 0;
}

//树形 dp
#include <stdio.h>
#include <string.h>
#define MAXN 1500

int parent[MAXN+1];
int dp[MAXN+1][3];
int n;

inline int min(int a,int b) { return a<b?a:b; }
void dfs(int root) {
    bool leaf=true;
    int s1=0,s2=0;
    for(int i=0;i<n;i++) {
        if(parent[i]==root) {
```

```
                leaf=false;
                dfs(i);
                s1+=dp[i][1];
                s2+=min(dp[i][0],dp[i][1]);
            }
        }
        if(leaf) {
            dp[root][0]=0;
            dp[root][1]=1;
        }else{
            dp[root][0]=s1;
            dp[root][1]=s2+1;
        }
    }

    int main() {
        int i,j,k,m,t,root;
        while(scanf("%d",&n)!=EOF) {
            memset(parent,255,sizeof parent);
            for(i=0;i<n;i++) {
                scanf("%d:(%d)",&j,&m);
                for(k=0;k<m;k++) {
                    scanf("%d",&t);
                    parent[t]=j;
                }
            }
            root=-1;
            do{ root++; } while(parent[root]!=-1);
            dfs(root);
            printf("%d\n",min(dp[root][0],dp[root][1]));
        }
        return 0;
    }
```

最长公共子串

题意简述：对于给定的 n 个序列 $S_1, S_2, \cdots, S_n$，编程计算其最长公共子序列的长度。

思路：LCS 问题就是求两个字符串最长公共子串的问题。解法就是用一个矩阵来记录两个字符串中所有位置的两个字符之间的匹配情况，若是匹配则为 1，否则为 0。然后求出对角线最长的 1 序列，其对应的位置就是最长匹配子串的位置。

当字符匹配的时候，我们并不是简单的给相应元素赋上 1，而是赋上其左上角元素的值加一。我们用两个标记变量来标记矩阵中值最大的元素的位置，在矩阵生成的过程中来判断当前生成的元素的值是不是最大的，据此来改变标记变量的值，那么到矩阵

完成的时候，最长匹配子串的位置和长度就已经出来了。

代码：

```
//////////////////////////
//最长公共子序列
//////////////////////
///c[i][j]保存字符串 {xi},{yj},(长度分别为 i,j)的最长公共子序列的字符个数
///i=0 或者是 j=0 时， c[i][j]必定为零, i,j>=0 且 xi=yj, c[i][j]=c[i-1][j-1]+1
///若 i,j>0 但 xi!=yj, c[i][j]=max{ c[i-1][j] , c[i][j-1] }
#include <stdio.h>
#include <string.h>
int c[1001][1001];
void lcs(int a, int b, char x[], char y[], int c[][1001])
{
    int i,j;
    for(i=1;i<a;i++)
        c[i][0]=0; //if b==0, set c[i][j]=0;
    for(i=1;i<b;i++)
        c[0][i]=0; // if a==0; set c[i][j]=0;
    for(i=1;i<=a;i++) // if a!=0,b!=0 loop
        for(j=1;j<=b;j++)
        {
            if(x[i-1]==y[j-1])
                c[i][j]=c[i-1][j-1]+1;
            else if (c[i-1][j]>=c[i][j-1])
                c[i][j]=c[i-1][j];
            else
                c[i][j]=c[i][j-1];
        }

}
int main()
{
    char x[1001],y[1001];
    while ( scanf("%s%s",x,y)!=EOF )
    {
        int a=strlen(x);
        int b=strlen(y);
        memset(c,0,sizeof(c));
        lcs(a,b,x,y,c);
        printf("%d\n",c[a][b]);
    }
    return 0;
}
```

回文串

题意简述：求一个给定的字符串至少添加几个字母使之变成回文串？

思路：第一种思想：一种最明了的分析方法就是规定 dp(i,j)表示对于[i,j）一段字符串至少添加几个字母使之变成回文串，那么容易写出：

dp(i,j)=min{ dp(i+1,j-1)&&a[i]==a[j-1] | dp(i+1,j)+1 | dp(i,j-1)+1　}

第二种思想：我们还可以这么想：记原字符串为 a，令其反转串(strrev)为 b，找 a,b 两字符串的关系，这时我们换一种思想，是不是求出 a 与 b 之间的最长公共子序列(LCS)，就把问题解决了么？如果a,b的LCS长度为len，那么最终答案不就是 strlen(a)-len？，

问题转化成 LCS 问题，构造的另一种状态转移方程也随即得出。

代码：

```
//（b=strrev(a) ，求 a,b 的 LCS）：
#include<iostream>
#include<string.h>
using namespace std;
char a[1000],b[1000];
int dp[1000][1000];
int main()
{
    int i,j;
    while(scanf("%s",a)==1)      //多组输入数据
    {
        int len=strlen(a);
        memset(dp,0,sizeof(dp));
        for(i=0,j=len-1;i<len;i++,j--)  //将 a 反转
            b[i]=a[j];
        b[len]='\0';
        for(i=1;i<=len;i++)              //用动态规划求两个字符串的最长公共子序列
        {
            for(j=1;j<=len;j++)
            {
                if(a[i-1]==b[j-1])
                    dp[i][j]=dp[i-1][j-1]+1;
                else if(dp[i-1][j]<dp[i][j-1])
                    dp[i][j]=dp[i][j-1];
                else
                    dp[i][j]=dp[i-1][j];
            }
        }
        printf("%d\n",len-dp[len][len]);
    }
    return 0;
}
```

第九章 字符串

9.1 KMP 算法

9.1.1 算法介绍

KMP 算法的产生是基于一个简单的问题——串的模式匹配，即在一个给定串（称为主串或目标串，用 s 表示）中寻找另一个串（称为模式串，用 t 表示）出现的位置。如 s=“s0s1s2…sn-1” p=“p0p1…pm-1” ，若 s[i..i+m-1]=p[0..m-1]，则称从位置 i 开始的匹配成功，亦称模式 t 在目标 s 中出现，我们所找的即为位置 i；若 s[i..i+m-1] ≠p[0..m-1]，则称从位置 i 开始的匹配失败。

朴素的算法就是对于合法的位置 0≦i≦n-m 依次将目标串中的子串和模式串 p[0..m-1]进行比较，对于以 i 为起始位置的 s 的子串 s[i..i+m-1]若与 p 匹配成功则返回位置 i，若匹配不成功则跳出，将起始位置后移一位继续寻找 s 的子串与 p 匹配，总的时间复杂度为 O（n*m）。

然而当我们遇到一个问题，如：s=“00000000000000000000000000001” t=“000000000001”，朴素的算法中显然很多次的匹配都是不必要的，尤其当随着 s 和 t 的长度增大、前面的无效比较次数增多，n*m 的时间复杂度显然不满足时间限制。

KMP 的思路是，每当一趟匹配过程中出现字符比较不等时，不再回溯 i，而是将模式串尽量后移，时间复杂度是 O（n+m）。

下面我们用一个简单的例子示例 KMP 算法。s=“ababcabcacbab”，t=“abcac”，

第一次匹配：ab**a**bcabcacbab

ab**c** （i=3，j=3） next[3]=1

第二次匹配：ababca**b**cacbab

abca**c** （i=7，j=5） next[5]=2

第三次匹配：ababcabca**c**bab

abca**c** （i=11，j=6，匹配成功）

next[j]=k 表明当模式串第 j 个字符与主串中第 i 个字符不匹配时，在模式中应需重新和主串中 i 进行比较的字符的位置。如 next[7]=4 的意义为与模式串前 6 个比较成功，但第 7 个不匹配，则下面同模式串第 4 个开始比较。

由此可见，KMP 算法最主要的就是求 next 函数，下面介绍用数学归纳法求 next 函数

s=“s0s1s2…sn-1” p=“p0p1…pm-1” 若 next[j]=k，则有（1）p 的前 k-1 个字符号应该同从主串 s 第 i-1 个倒数的 k-1 个字符相同。即：p1p2…pk-1=si-k+1si-k+2…si-1（2）主串 s 从第 i-1 个倒数的 k-1 个字符应该同子串第 j 个开始倒数的 k-1 个字符也应该相同。即：pj-k+1pj-k+2…pj-1=si-k+1si-k+2…si-1，则由（1）、（2）得出（3）

p1p2…pk-1=pj-k+1pj-k+2…pj-1，这是 next[j]=k 的充要条件。

则具体的求法为

（1） next[1]=0

（2） 设 next[j]=k，表明

'p1p2…pk-1'='pj-k+1pj-k+2…pj-1'

(a)若 pk=pj，则'p1p2…pk'='pj-k+1pj-k+2…pj' 且 next[k-1]保证 k 为最大的满足此式的值，则 next[j+1]=next[j]+1

(b)若 pk！=pj，此时模式串既是主串又是模式串，应将模式向右滑动至以模式中第 next[k]个字符和主串中的第 j 个字符相比较。记 next[k]=k',若 pj=pk'，则 ' p1p2…pk' '=pj-k'+1pj-k'+2…pj', 则 next[j+1]=next[k]+1

若 pj！=pk'，同理，将模式串后移至将第 next[k']个字符和 pj 对齐，直至匹配成功或不存在任何 k'（1<k'<j）满足等式，则 next[j+1]=1

函数代码及讲解如下：

```
 void get_next(char s[],int next[])//char s[]从 0 开始记，next[]从
                                          //1 开始记,记录的是位置
{
    int i,j;i=1;next[1]=0;j=0;
    while(i<lens)
   {
      if(j==0||s[i-1]==s[j-1])//当 j=0 时，即要从子串第一个再开始
                    //比较，而主串要后移 1；当第 i 个和第 j(next[i-1])
                    //个字符相等时，next 值关系是加一关系
     {
          ++i;++j;
//优化：此时求第 i 个字符的 next，本应是 j=next[i]，若第 i 个和第 j 个相
//等，则不需再比，主串与子串肯定不等，要继续右滑
          /*if(s[i-1]==s[j-1] next[i]=next[j];
            else                    next[i]=j;*/
       next[i]=j;
      }
    else   j=next[j];//子串的 j 指针转到 next[j]继续比较
    }
}
```

9.1.2 next 指针

KMP 算法的精髓是 next[],实际上，next[]的用处不只限于模式匹配，还有其他很多的妙用，这里我们举两个例子，其它的留给读者自己探索。

题意简述：求一个字符串的最大循环次数，题目具体是给定一个字符串 s，找一个最小的 k，使得 s 为若干个长度为 k 的相同字符串的连接，输出 s/k;。比如 abababab 由三个 ab 重复得来，则 ab 是最小的循环节，那么 6/2=3 就是最大循环次数 。

思路：如果有一个字符串 s,长度是 len,它的失败函数是 next，如果 len 能被 len-next[len]整除，那么 len-next[len]就是我们要求的那个子串的长度，与之对应的字符串，就是我们想得到的子串。

为什么呢？ 假设我们有一个字符串 abababab,那么 next[6]=4, 对吧，由于 next 的性质是匹配失败后，下一个能继续进行匹配的位置，也就是说，把字符串的前四个字母，abab,平移 2 个单位，这个 abab 一定与原串的 abab 重合（否则就不满足失败函数的性质），这说明了什么呢，由于字符串进行了整体平移，而平移后又要

重叠，那么必有 s[1]=s[3],s[2]=s[4],s[3]=s[5], s[4]=s[6].说明长度为 2 的字符串在原串中一定重复出现，这就是 len-next[len]的含义！

代码：

```
#include<iostream>
#include<string.h>
using namespace std;
int lens,next[1000002];
char s[1000002];
void get_next(char s[],int next[])
{
    int i,j;
    i=1;
    next[1]=0;
    j=0;
    while(i<lens)
    {
        if(j==0||s[i-1]==s[j-1])
        {
            ++i;
            ++j;
            if(s[i-1]==s[j-1])
                next[i]=next[j];
            else
                next[i]=j;
            next[i]=j;
        }
        else
            j=next[j];
    }
}
int main()
{
     int m,i;
     while(scanf("%s",s))
     {
         if(strcmp(s,".")==0)
             break;
         lens=strlen(s);
         get_next(s,next);
         m=lens-next[lens];
         if(lens%m==0&&s[lens-1]==s[next[lens]-1])
             printf("%d\n",lens/m);
         else
             printf("1\n");
```

```
    }
    return 0;
}
```

题意简述： 给定一个字符串，如果一个长度为 i 的前缀同时又是此串的后缀，i 叫一个公共前后缀的长度，求所有的公共前后缀的长度,按递增顺序排列。

题目分析： 由 next 函数的性质，最长的公共前后缀必然满足 next[len]&&s[len-1]==s[next[len]-1],然后递归下去，在最长的公共前后缀里再找。

代码：

```
//求所有的公共前后缀的长度
#include<iostream>
#define MAX   400001
using namespace std;
int next[MAX];
int lens;
char s[MAX];
void get_next(char s[],int next[])
{
    int i,j;
    i=1;
    next[1]=0;
    j=0;
    while(i<lens)
    {
        if(j==0||s[i-1]==s[j-1])
        {
            i++;
            j++;
            next[i]=j;
        }
        else
            j=next[j];
    }
}
void output(int i)
{
    if(next[i]!=0&&s[i-1]==s[next[i]-1])//从 next[lens]开始，注意条件 s[i-1]==s[next[i]-1]
                                        //一定要加上，这样才是共同的前后缀
    {
        output(next[i]);
        printf("%d ",next[i]);
    }
}
int main()
{
```

```
        while(cin>>s)//s 从 0 开始计数
        {
            lens=strlen(s);
            get_next(s,next);
            output(lens);
            printf("%d\n",lens);
        }
        return 0;
    }
```

9.2 字典树（Tries）

树形结构，也叫单词查找树、trie 树，优点是利用字符串的公共前缀来节约存储空间，最大限度地减少无谓的字符串比较。根节点不包含字符，除根节点外每一个节点都只包含一个字符。 从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串。 每个节点的所有子节点包含的字符都不相同。

存储结构：

```
struct node
{
    node *next[26];//子结点，若某个字符是此结点的子结点时，将开辟对应字符的空间，证
                   //明存在
    int count;//记录以该字符结尾的单词的个数
    node()
    {
        count=0;
        memset(next,NULL,sizeof(next));
    }
};
```

如下图所示，单词 say she shr he her 的字典树

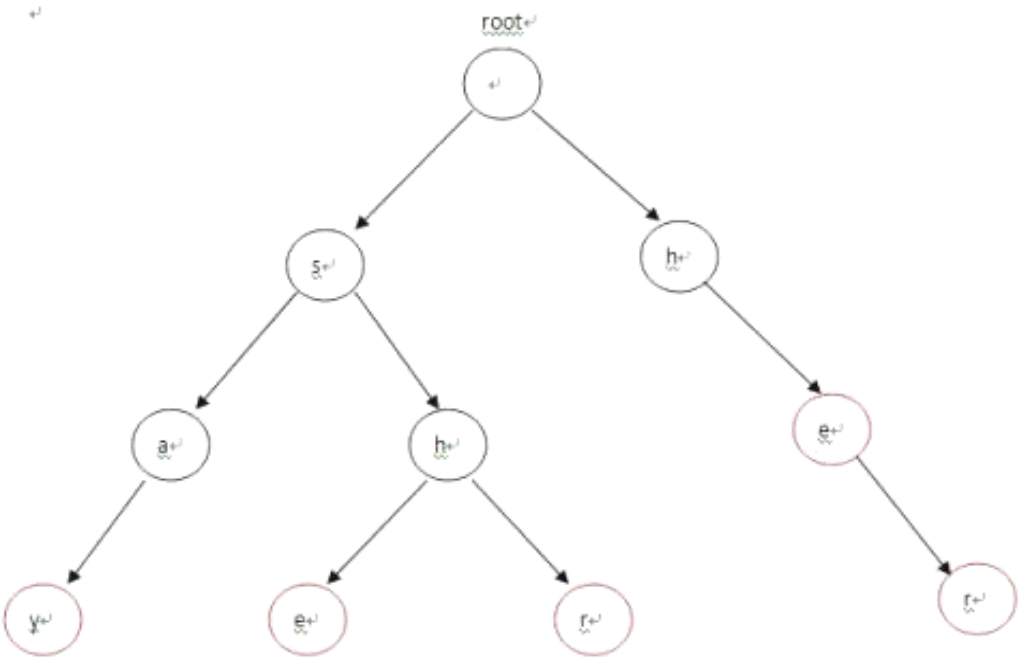


图 9-1

字典树的基本操作有:查找 插入和删除,查找很简单，当然删除操作比较少见，但对于单个 word 的删除操作很简单，只要将结尾字符的 count 值减一即可，因为一个结点的 count 值大于 1 只意味着相同单词出现了多次，如果删除一次则 count 值减一，若是彻底地删除，即不再存在此单词，则 count 值置为 0 即可。在这里只介绍字典树的插入操作。

```
void insert(char *keyword,node *&root){
    node *p=root;
    int index,i=0;
    while(keyword[i])
    {
        index=keyword[i]-'a';
        if(p->next[index]==NULL)
            p->next[index]=new node();
        p=p->next[index];
        i++;
    }
    p->count++;//后面每次到达一个结点都要加 count 值，然后一次性地置为-1，可以理解
//count 代表以此结点结尾的单词的个数
}
```

题意简述： 在有道搜索框中，当输入一个或者多个字符时，搜索框会出现一定数量的提示，如下图所示：

图 9-2

现在给你 N 个单词和一些查询，请输出提示结果，为了简化这个问题，只需要输出以查询词为前缀的并且按字典序排列的最前面的 8 个单词，如果符合要求的单词一个也没有请只输出当前查询词。

输入：

第一行是一个正整数 N，表示词表中有 N 个单词。接下来有 N 行，每行都有一个单词，注意词表中的单词可能有重复，请忽略掉重复单词。所有的单词都由小写字母组成。接下来的一行有一个正整数 Q，表示接下来有 Q 个查询。接下来 Q 行，每行有一个单词，表示一个查询词，所有的查询词也都是由小写字母组成，并且所有的单词以及查询的长度都不超过 20，且都不为空其中：N<=10000,Q<=10000

输出：

对于每个查询，输出一行，按顺序输出该查询词的提示结果，用空格隔开。

思路: 多 case 输入，每个 case 的需要查询的单词个数 10^4，已知的单词库中的单词个数 10^4，每个单词长度最多 20，则若暴力查询时间达到 10^8 数量级，对于 1000ms 时限效率低不可行，则考虑将单词库建成一个字典树，查询时在树上顺次找与查询词最后一个字母匹配的字母，若找不到，则输出查询词即可，若找得到，则以此字母为根节点的前8棵子树即为所求，深搜可得结果。

代码：

```
#include<iostream>
#include<stdio.h>
#define MAX 200001
using namespace std;
char keynode[21],str[10000];
int len;
struct node
{
    node *fail;
    node *next[26];
    int count;
```

```
    node()
    {
        fail=NULL;
        count=0;
        memset(next,NULL,sizeof(next));
    }
}*q[MAX];
//构建 Trie 树
void insert(char *keyword,node *&root)
{
    node *p=root;
    int index,i=0;
    while(keyword[i])
    {
        index=keyword[i]-'a';
        if(p->next[index]==NULL)
            p->next[index]=new node();
        p=p->next[index];
        i++;
    }
    p->count++;
}
//深搜得出所有解
int ans;
void dfs(node *p)
{
    int i;int k=0;
    char ch;
    if(p->count!=0)
    {
        if(ans==8)    return;
        printf("%s ",str);
        ans++;
    }
    for(i=0;i<26;i++)
    {
        if(p->next[i]!=NULL)
        {    k=1;
            ch=i+'a';
            str[len]=ch;
            str[++len]='\0';//后面要加一个'\0'代表数组结束
            dfs(p->next[i]);
            len--; //注意 len 的妙用，全局变量，然后每次当一个函数跳出后就 len--，回到
                   //父亲结点的 len 值，等下一个路径的赋值
```

```
            }
        }
}
int main()
{
        int index,i,k,n;
        scanf("%d",&k);
        getchar();
        node *root=new node();
        while(k--)
        {
                scanf("%s",keynode);
                getchar();
                insert(keynode,root);
        }
        scanf("%d",&n);
        getchar();
        while(n--)
        {
                ans=0;
                scanf("%s",str);
                getchar();
                len=strlen(str);
                node *p=root;
                i=0;
                while(str[i])
                {
                        index=str[i]-'a';
                        if(p->next[index]!=NULL)
                        {
                                i++;
                                p=p->next[index];
                        }
                        else break;
                }
                if(i!=len)
                {
                        printf("%s\n",str);
                        continue;
                }
                dfs(p);
                printf("\n");

        }
```

```
    return 0;
}
```

9.3 AC 自动机

AC 自动机解决的最主要的问题是多模式匹配，就是给出 n 个单词，再给出一段包含 m 个字符的文章，让你找出有多少个单词在文章里出现过。那么它的特殊之处在哪里呢？有了 9.1 节对 KMP 算法的了解，那么读者就应该明白 next 函数的妙用，同样 AC 自动机的失败指针具有同样的功能，也就是说当我们的模式串在 Tire 上进行匹配时，如果与当前节点的关键字不能继续匹配的时候，就应该去当前节点的失败指针所指向的节点继续进行匹配。概括起来，AC 自动机算法分为 3 步：构造一棵 Trie 树，构造失败指针和模式匹配过程。

下面以一个例子来讲解 AC 自动机的构造和应用过程。

题意简述：给定 5 个单词：say she shr he her，然后给定一个字符串 yasherhs。问一共有多少单词在这个字符串中出现过。

思路：第一步，定义一些数据结构。树上结点的结构如下：

```
struct node
{
    node *fail;          //失败指针
    node *next[26]; //Tire 每个节点的个子节点（最多个字母）
    int count;           //是否为该单词的最后一个节点
    node()//构造函数初始化
    {
        fail=NULL;
        count=0;
        memset(next,NULL,sizeof(next));
    }
}*q[500001];              //队列，方便用于 bfs 构造失败指针
char keyword[51];       //输入的单词
char str[1000001];      //模式串
int head,tail;          //队列的头尾指针
```

第二步，构造 trie 树。将这五个单词构造成如图 9.2.1 的字典树：

```
void insert(char *str,node *root)
{
    node *p=root;
    int i=0,index;
    while(str[i])
    {
        index=str[i]-'a';
        if(p->next[index]==NULL) p->next[index]=new node();
        p=p->next[index];
        i++;
    }
```

```
    p->count++;        //在单词的最后一个节点 count+1，代表一个单词
  }
```

第三步，构造失败指针。构造失败指针的过程概括起来就一句话：设这个节点上的字母为 C，沿着他父亲的失败指针走，直到走到一个节点，他的儿子中也有字母为 C 的节点。然后把当前节点的失败指针指向那个字母也为 C 的儿子。如果一直走到了 root 都没找到，那就把失败指针指向 root。

```
void build_ac_automation(node *root)
{
    int i;
     root->fail=NULL;
     q[head++]=root;
     while(head!=tail)
     {
          node *temp=q[tail++];
          node *p=NULL;
          for(i=0;i<26;i++)
          {
               if(temp->next[i]!=NULL)
               {
                    if(temp==root) temp->next[i]->fail=root;
                    else
                    {
                         p=temp->fail;
                         while(p!=NULL)
                         {
                              if(p->next[i]!=NULL)
                              {
                                   temp->next[i]->fail=p->next[i];
                                   break;
                              }
                              p=p->fail;
                         }
                         if(p==NULL) temp->next[i]->fail=root;
                    }
                    q[head++]=temp->next[i];
               }
          }
    }
}
```

失败指针的构造完毕后，trie 树应呈如下形式所示：

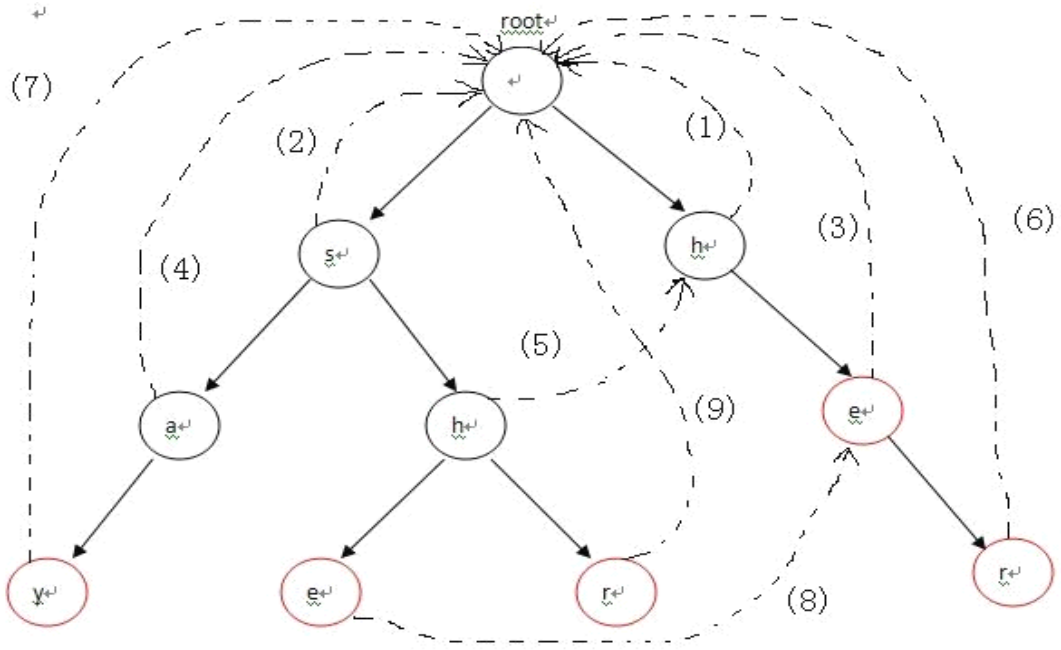


图 9-3

从代码结合例子来看一下失败指针的构造过程，首先 root 的 fail 指针指向 NULL，然后 root 入队，进入循环。第 1 次循环的时候，我们需要处理 2 个节点：root->next[‘h’-‘a’](节点 h) 和 root->next[‘s’-‘a’](节点 s)。把这 2 个节点的失败指针指向 root，并且先后进入队列，失败指针的指向对应图 9.3.1 中的(1)，(2)两条虚线；第 2 次进入循环后，从队列中先弹出 h，接下来 p 指向 h 节点的 fail 指针指向的节点，也就是 root；进入第 13 行的循环后，p=p->fail 也就是 p=NULL，这时退出循环，并把节点 e 的 fail 指针指向 root，对应图 9.3.1 中的(3)，然后节点 e 进入队列；第 3 次循环时，弹出的第一个节点 a 的操作与上一步操作的节点 e 相同，把 a 的 fail 指针指向 root，对应图 9.3.1 中的(4)，并入队；第 4 次进入循环时，弹出节点 h(图中左边那个)，这时操作略有不同。在程序运行到 14 行时，由于 p->next[i]!=NULL(root 有 h 这个儿子节点，图中右边那个)，这样便把左边那个 h 节点的失败指针指向右边那个 root 的儿子节点 h，对应图 9.3.1 中的(5)，然后 h 入队。以此类推：在循环结束后，所有的失败指针就是图 9.3.1 中的这种形式。

第四步，模式匹配过程即在 AC 自动机上查找模式串中出现过哪些单词。匹配过程分两种情况：(1)当前字符匹配，表示从当前节点沿着树边有一条路径可以到达目标字符，此时只需沿该路径走向下一个节点继续匹配即可，目标字符串指针移向下个字符继续匹配；(2)当前字符不匹配，则去当前节点失败指针所指向的字符继续匹配，匹配过程随着指针指向 root 结束。重复这 2 个过程中的任意一个，直到模式串走到结尾为止。

```
int query(node *root){
    int i=0,cnt=0,index,len=strlen(str);
    node *p=root;
    while(str[i]){
        index=str[i]-'a';
        while(p->next[index]==NULL && p!=root) p=p->fail;
        p=p->next[index];
        p=(p==NULL)?root:p;
```

```
            node *temp=p;
            while(temp!=root && temp->count!=-1){
                cnt+=temp->count;
                temp->count=-1;
                temp=temp->fail;
            }
            i++;
        }
        return cnt;
}
```

看一下模式匹配这个详细的流程，其中模式串为 yasherhs。对于 i=0,1。Trie 中没有对应的路径，故不做任何操作；i=2,3,4 时，指针 p 走到左下节点 e。因为节点 e 的 count 信息为1，所以 cnt+1，并且讲节点 e 的 count 值设置为-1，表示改单词已经出现过了，防止重复计数，最后 temp 指向 e 节点的失败指针所指向的节点继续查找，以此类推，最后 temp 指向 root，退出 while 循环，这个过程中 count 增加了 2。表示找到了 2 个单词 she 和 he。当 i=5 时，程序进入第 5 行，p 指向其失败指针的节点，也就是右边那个 e 节点，随后在第 6 行指向 r 节点，r 节点的 count 值为 1，从而 count+1，循环直到 temp 指向 root 为止。最后 i=6,7 时，找不到任何匹配，匹配过程结束。

那么至此，AC 自动机介绍完毕，AC 自动机的精髓就是 fail 指针，实际上就是在树上的 KMP，但 AC 自动机和 KMP 解决的问题不同，希望读者认真体会。

9.4 后缀数组

后缀数组是字符串处理的强有力工具，尤其在信息学竞赛中，所以在此介绍一下后缀数组的基本定义、构造方法、后缀数组的应用。

9.4.1 基本概念

约定一个字符集Σ；
待处理的字符串记为为 S，约定 len(S)=n；
规定 S 以字符“$”结尾，即 S[n]=“$”；
“$”小于Σ中所有的字符；
除了 S[n]=“$”之外，S 的其他字符都属于Σ；
对于待处理的字符串 S，其 i 开头的后缀表示为 Suffix(i)；
【后缀数组 SA】后缀数组保存的是一个字符串的所有后缀的排序结果。其中 SA[i]保存的是字符串所有的后缀中第 i 小的后缀的开头位置。
【名次数组 Rank】名次数组 Rank[i]保存的是后缀 i 在所有后缀中从小到大排列的“名次”。为了叙述方便，以第 k 个字符开始的后缀称为后缀 k。
简单的说，后缀数组是“排第几的是谁？”，名次数组是“你排第几？”。容易看出，后缀数组和名次数组为互逆运算。
以字符串“aabaaaab”为例。其所有后缀如下图所示：

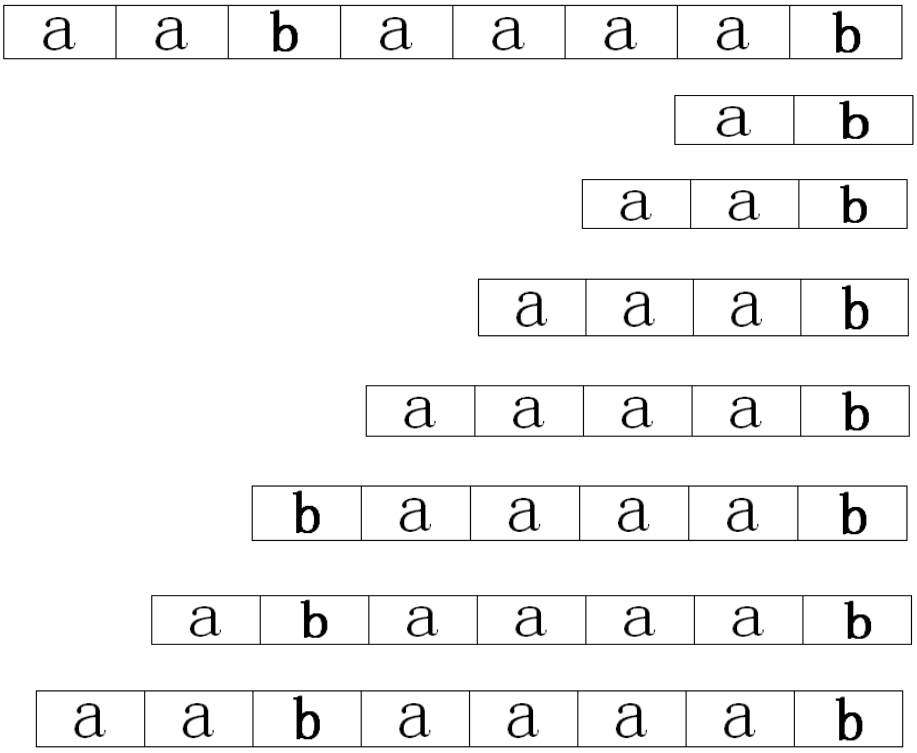


图 9-4

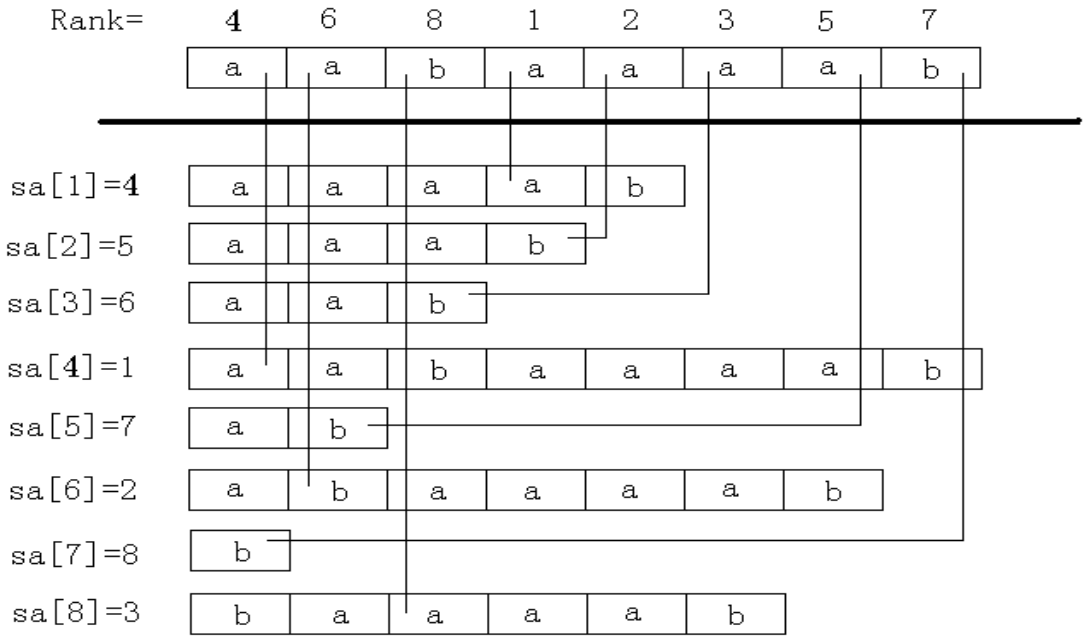


图 9-5

9.4.2 后缀数组的构造

后缀数组的构造主要是倍增算法(Doubling Algorithm)，首先介绍一些表示方法

对字符串 u，定义 uk = $\begin{cases} u[1..k] & ,len(u)\geqslant k \\ u & ,len(u)<k \end{cases}$

定义一个比较运算符

定义 k-前缀比较关系<k，=k 和≤k

对两个字符串 u,v，

u<kv　当且仅当　uk<vk

u=kv　当且仅当　uk=vk

u≤kv　当且仅当　uk≤vk

那么对于此比较运算符有如下性质：

性质 1 Suffix(i)=2kSuffix(j)等价于 Suffix(i)=kSuffix(j) 且
Suffix(i+k)=kSuffix(j+k)。

性质 2 Suffix(i)<2kSuffix(j) 等价于 Suffix(i)<kS(j) 或(Suffix(i)=kSuffix(j) 且
Suffix(i+k)<kSuffix(j+k))。

那么由这两个性质，若我们知道 k-前缀比较关系，那么就可以根据 k-前缀比较关系来表达 2k-前缀比较关系。且 1-前缀比较关系实际上是对字符串的第一个字符进行比较。那么整体的倍增算法求后缀数组的思路就是：

（1）可直接根据开头字符对所有后缀进行排序求出 SA1，采用快速排序，复杂度 O(nlogn)。然后根据 SA1 在 O(n)时间内求出 Rank1。总的来说，可以在 O(nlogn)时间内求出 SA1 和 Rank1。

（2）用 k-前缀比较关系来表达 2k-前缀比较关系。每次可以将参与比较的前缀长度加倍根据 SAk、Rankk 求出 SA2k、Rank2k。参与比较的前缀长度达到 n 以上时结束。

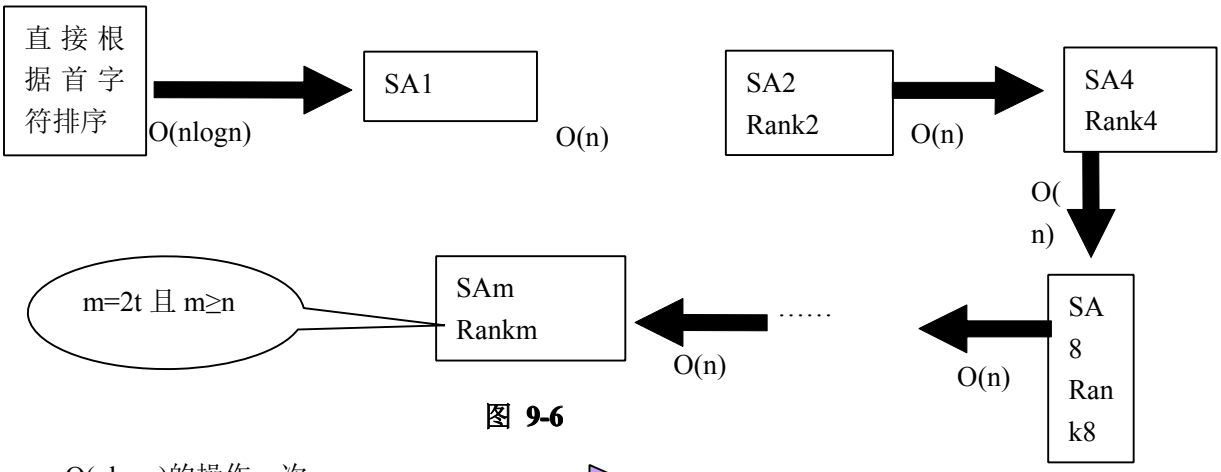


图 9-6

O(nlogn)的操作一次
O(n)的操作[logn]次

总复杂度为 O(nlogn)，即可以在 O(nlogn)时间内求出后缀数组 SA 和名次数组 Rank 数组。

图 9-6

源代码：

```
/*后缀数组
参数：
    输入字符串 str[];
```

```
    返回后缀数组 sa[];返回名次数组 rank[]
说明：
    复杂度 O(nlogn)
    需要 stdlib.h 辅助全局变量 power, lstrank[],cmpstr[]
*/
int power;
int lstrank[N];
int cmpstr[N];

int cmpChar(const void *a,const void *b)
{
    return cmpstr[*(int*)a] - cmpstr[*(int*)b];
}

int cmpLst(const void *A,const void *B)
{
    int a = *(int*)A;
    int b = *(int*)B;
    if(lstrank[a]==lstrank[b])
        return lstrank[a+power]-lstrank[b+power];
    return lstrank[a]-lstrank[b];
}

void suffixArray(int str[],int sa[],int rank[])
{
    int i,j,n=nday;

    for(i=0;i<n;i++) sa[i]=i;
    memcpy(cmpstr,str,sizeof(cmpstr));
    qsort(sa,n,sizeof(int),cmpChar);
    for(i=j=0;i<n;i++)
    {
        if(i>0&& str[sa[i]]!=str[sa[i-1]]) j++;
        rank[sa[i]]=j;
    }

    for(power=1;rank[sa[n-1]]<n-1; power*=2)
    {
        memcpy(lstrank,rank, sizeof(int)*n);
        qsort(sa,n,sizeof(int), cmpLst);
        for(i=j=0;i<n;i++)
        {
            if(i>0&&cmpLst(&sa[i-1], &sa[i])) j++;
            rank[sa[i]]=j;
```

```
        }
    }
}
```

9.4.3 后缀数组的应用

后缀数组处理字符串问题很强大，这里只是介绍后缀数组应用的冰山一角。

（1）height 数组。Height 数组利用后缀数组求出，是下面求各种字符串问题的关键。

Height 数组定义如下：

height[i]=suffix(sa[i-1])和 suffix(sa[i])的最长公共前缀，也就是排名相邻的两个后缀的最长公共前缀。（注意 height[1]=0,因为 sa[0]始终为那个特殊字符）

（2）两个后缀的最长公共前缀

对正整数 i,j 定义 LCP(i,j)=lcp(Suffix(SA[i]),Suffix(SA[j])，其中 i,j 均为 1 至 n 的整数。LCP(i,j)也就是后缀数组中第 i 个和第 j 个后缀的最长公共前缀的长度。

经过仔细分析，我们发现 LCP 函数有一个非常好的性质：

设 i<j，则 LCP(i,j)=min{LCP(k-1,k)|i+1≤k≤j} （LCP Theorem）（证明略）

那么既然我们已经求出 height 数组，对于 j 和 k，不妨设 rank[j]<rank[k],则有以下性质：suffix(j)和 suffix(k)的最长公共前缀为 height[rank[j]+1], height[rank[j]+2], height[rank[j]+3], … ,height[rank[k]]中的最小值。

例如，字符串为“aabaaaab”，求后缀“abaaaab”和后缀“aaab”的最长公共前缀。

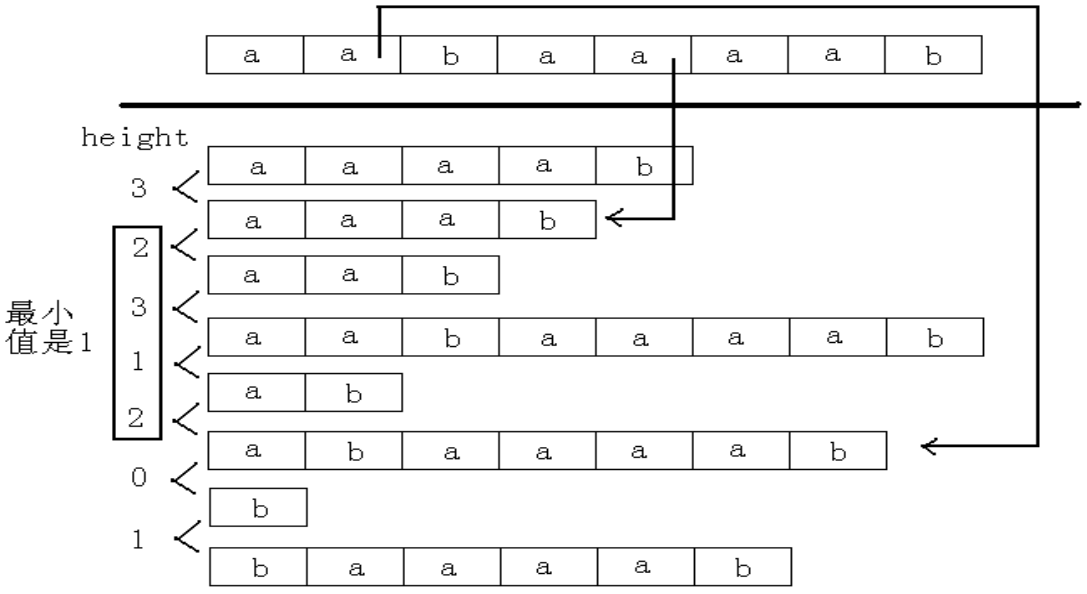


图 9-7

（3）可重叠最长重复子串。

字符串 R 在字符串 L 中至少出现两次，则称 R 是 L 的重复子串。 给定一个字符串，求最长重复子串，这两个子串可以重叠。

算法分析：求 height 数组里的最大值即可。首先求最长重复子串，等价于求两个后缀的最长公共前缀的最大值。因为任意两个后缀的最长公共前缀都是 height 数组里某一段的最小值，那么这个值一定不大于 height 数组里的最大值。所以最长重复子串的长度就是 height 数组里的最大值。这个做法的时间复

杂度为 O(n)。

（4）不可重叠最长重复子串。

算法分析：这题比上一题稍复杂一点。先二分答案，把题目变成判定性问题：假设当时枚举的长度为 k，判断是否存在两个长度为 k 的子串是相同的，且不重叠。解决这个问题的关键还是利用 height 数组。首先，根据 height 数组，将排好序的后缀分成若干组，使得每组后缀中，后缀之间的 height 值不小于 k。这样分组之后，不难看出，如果某组后缀数量大于 1，那么它们之中存在一个公共前缀，其长度为它们之间的 height 值的最小值。而我们分组之后，每组后缀之间 height 值的最小值大于等于 k。所以，后缀数大于 1 的分组中，有可能存在满足题目限制条件的长度不小于 k 的子串。只要判断满足题目限制条件成立，那么说明存在长度至少为 k 的合法子串。

另外，限制条件是不重叠，判断的方法是，一组后缀中，起始位置最大的后缀的起始位置减去起始位置最小的后缀的起始位置>=k。满足这个条件的话，那么这两个后缀的公共前缀不但出现两次，而且出现两次的起始位置间隔大于等于 k，所以不会重叠。

例如：字符串为“aabaaaab”，当 k=2 时，后缀分成了 4 组，如下图：

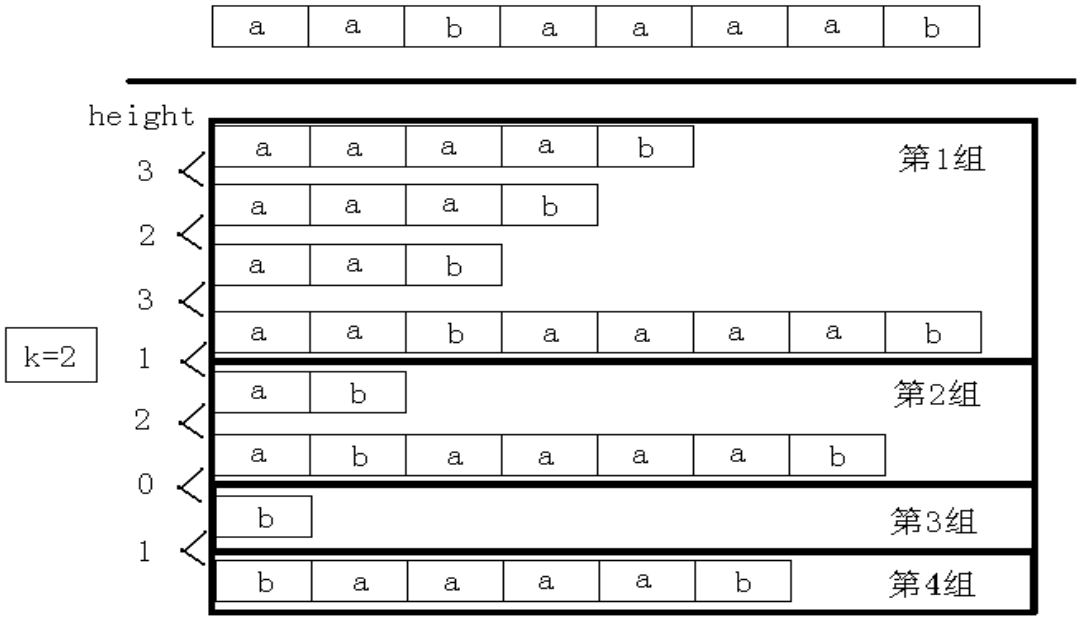


图 9-8

建议读者深刻理解这种 height 分组方法以及判断重叠与否的方法，在后缀数组处理字符串的问题中起到举足轻重的作用。

（5）可重叠的 k 次最长重复子串。

给定一个字符串，求至少出现 k 次的最长重复子串，这 k 个子串可以重叠。

算法分析：这题的做法和上一题差不多，也是先二分答案，然后将后缀分成若干组。不同的是，这里要判断的是有没有一个组的后缀个数不小于 k。如果有，那么存在 k 个相同的子串满足条件，否则不存在。这个做法的时间复杂度为 O(nlogn)。