Gcd

```
int gcd(int a,int b)
{
    return b==0?a:gcd(b,a%b);
}
//若方程ax+by=c 的一组解(x0,y0)，则它的任意整数解都可以写成(x0+kb’,y0-ka’)，其中a’=a/gcd(a,b)，b’=b/gcd(a,b)
```

Exgcd

```
//对于不完全为 0 的非负整数 a,b,gcd(a，b)表示 a,b 的最大公约数，必然
//存在整数对 x,y ,使得 gcd(a,b)=ax+by。
//ax + by = m有整数解（x,y）当且仅当m是gcd(a,b)的倍数。且必然有无穷多个解。
//裴蜀定理的一个重要推论：a,b互质的充要条件是存在整数x,y使ax+by=1.
int exgcd(int a,int b,int& x,int& y)
{
    int d=a;
    if(b!=0)
    {
        d=exgcd(b,a%b,y,x);
        y-=(a/b)*x;
    }
    else
    {
        x=1;y=0;
    }
    return d;
}
```

组合数C(n,m)打表

```
//公式C[n][m]=C[n-1][m]+C[n-1][m-1]
C[1][0]=1;
C[1][1]=1;
for(int i=2;i<n;i++)
{
    C[i][0]=1;
    for(int j=1;j<m;j++)
        C[i][j]=(C[i-1][j]+C[i-1][j-1])%mod;
}
```

快速幂n^k

```
long long quick_pow(long long n,long long k)
{
    n=n%mod;   //WA 大数进来要先mod
    long long ans=1;
    while(k>0)
    {
        if(k&1)
            ans=ans*n%mod;
        n=n*n%mod;
        k>>=1;
    }
    return ans;
}
```

素数筛

```
//筛选1-n的质数,存在prime[]数组里,下标从0开始
void filterPrime(int n)
{
    bool isPrime[MAX];     //最好放全局变量
    primeCnt=0;      //全局变量
    memset(isPrime,true,sizeof(isPrime));
    isPrime[0]=false;
    isPrime[1]=false;
    for(int i=2;i<=n;i++)
    {
        if(isPrime[i]==true)
        {
            prime[primeCnt++]=i;
            for(int j=i*2;j<=n;j=j+i)
                isPrime[j]=false;
        }
    }
}
```

rmq

```
//rmq[i][j]表示从第j个数开始,长度为2^i的最大值
for(int i=1;i<=N;i++)
    scanf("%d",&rmq[0][i]);

for(int i=1;(1<<i)<=N;i++)
    for(int j=1;j+(1<<i)-1<=N;j++)
    rmq[i][j]=min(rmq[i-1][j],rmq[i-1][j+(1<<(i-1))]);

int rmqQuery(int l,int r)
{
    int k=0;
    while((1<<(k+1))<=r-l+1)
        k++;
    return min(rmq[k][l],rmq[k][r-(1<<k)+1]);
}
```

LIS

```
//输入数组A,数组长度n,最长上升子序列
int LIS(int *A,int n)
{
    int ans=0;
    B[ans]=A[0];
    for(int i=1;i<n;i++)
    {
        if(A[i]>B[ans])    //没有等号是严格LIS
            B[++ans]=A[i];
        else
        {
            int idx=(int)(upper_bound(B,B+ans,A[i])-B);
            B[idx]=A[i];
        }
    }
    return ans+1;
}
//B[i]表示 长度为i+1的LIS末尾元素最小为多少,末尾越小当然越优
//遍历A数组,维护B数组,最后B数组元素个数即为答案
//由于B数组单调递增,可二分查找
```

判断a->b->c旋转方向

```
int ccw(const struct node &a,const struct node &b,const
struct node &c)
{
    double area2 = ((double)b.x-a.x)*(c.y-a.y) - (b.y-
a.y)*(c.x-a.x);
    if      (area2 < 0) return -1; // clockwise
    else if (area2 > 0) return 1;  // counter-clockwise
    else                return 0;  // collinear
}
```

STL

```
//priorityQueue重载<
struct node{
    int data;
    int idx;
    bool operator < (const node &n) const {
            return n.data>data;
    }
};
//upper_bound(A,A+n,x)返回第一次出现x的位置
//lower_bound(A,A+n,x)返回最后一次出现x的位置的下一个
```

JAVA大数

```
import java.math.BigInteger;

public class BigIntegers {
        public static void main(String[] args) {
                String num1 = "99999999999999999999999999999";
                String num2 = "99999999999999999999999999998";
                BigInteger big1 = new BigInteger(num1);
                BigInteger big2 = new BigInteger(num2);
                System.out.println("加法操作:" + big1.add(big2));
                System.out.println("减法操作:" + big1.subtract(big2));
                System.out.println("乘法操作:" + big1.multiply(big2));
                BigInteger result[] = big1.divideAndRemainder(big2);
                System.out.println("相除后的商是:" + result[0]);
                System.out.println("相除后的余数是:" + result[1]);
        }
}

import java.math.BigDecimal;
class MyMath{
        public static double add(String num1, String num2){
                BigDecimal big1 = new BigDecimal(num1);
                BigDecimal big2 = new BigDecimal(num2);
                return big1.add(big2).doubleValue();
        }
        public static double sub(String num1, String num2){
                BigDecimal big1 = new BigDecimal(num1);
                BigDecimal big2 = new BigDecimal(num2);
                return big1.subtract(big2).doubleValue();
        }
        public static double mul(String num1, String num2){
                BigDecimal big1 = new BigDecimal(num1);
```

```
                BigDecimal big2 = new BigDecimal(num2);
                return big1.multiply(big2).doubleValue();
        }
        public static double round(double num, int scale){
                BigDecimal bd1 = new BigDecimal(num);
                BigDecimal bd2 = new BigDecimal(1);
                return bd1.divide(bd2, scale, BigDecimal.ROUND_HALF_UP).doubleValue();
        }
        public static double div(String num1, String num2, int scale){
                BigDecimal bd1 = new BigDecimal(num1);
                BigDecimal bd2 = new BigDecimal(num2);
                return bd1.divide(bd2, scale, BigDecimal.ROUND_HALF_UP).doubleValue();
        }
}
public class BigIntegers {
        public static void main(String[] args) {
                String num1 = "12345.6789";
                String num2 = "3333.23443";
                System.out.println("加法操作:" + MyMath.round(MyMath.add(num1, num2), 2));
                System.out.println("减法操作:" + MyMath.round(MyMath.sub(num1, num2), 2));
                System.out.println("乘法操作:" + MyMath.round(MyMath.mul(num1, num2), 2));
                System.out.println("除法操作:" + MyMath.div(num1, num2, 2));
        }
}
```

SPFA算法 判断负权环

Bellman–Ford的优化版本，不需要像Bellman–Ford算法一样把每条边都更新n次

vector模拟邻接表存储，dist初值为INF

首先将源点入队，若一个结点的dist更新了，将他进队去更新他的邻接点

判断负环：如一个结点进队超过n次则存在负环

SLF优化：deque维护

```cpp
#include <iostream>
#include <queue>
#include <vector>
#include <cstring>

#define MAX 10
#define INF 0xfffffff

using namespace std;

struct node{
    int u;
    int w;
};

vector<vector<struct node>> G;
int dist[MAX];
bool ifInQueue[MAX];
int updateTimes[MAX];       //若大于n,则有负环

bool SPFA(int S)
{
    memset(ifInQueue,0,sizeof(ifInQueue));
    memset(updateTimes,0,sizeof(updateTimes));
    for(int i=0;i<=N;i++)
        dist[i]=INF;
```

```
    deque<int> que;
    dist[S]=0;
    ifInQueue[S]=true;
    que.push_back(S);
    while(!que.empty())
    {
        int u=que.front();
        que.pop_front();
        ifInQueue[u]=false;
        for(int i=0;i<G[u].size();i++)
        {
            int v=G[u][i].u;
            if(dist[u]+G[u][i].w<dist[v])
            {
                dist[v]=dist[u]+G[u][i].w;
                updateTimes[v]++;
                //进队超过n次则有负环
                if(updateTimes[v]>=N)
                    return true;
                //把最短路径更新的结点入队
                if(ifInQueue[v]==false)
                {
                    ifInQueue[v]=true;
                    //SLF优化,先更新短边
                    //注意que.empty()放在前面
                    if(que.empty()==false &&
dist[v]<dist[que.front()])
                        que.push_front(v);
                    else
                        que.push_back(v);
                }
            }
        }
    }
    return false;
}
```

Dijkstra

vector模拟邻接表，注意resize，dist数组初始化为INF

priority_queue重载<，让dist小的在前

首先将源点入队，访问他的邻接点，如果这个结点未被访问且能更新变小，则更新dist数组，并将该结点入队，每次从pq中弹出一个最短的边

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

#define N 10
#define INF 0xfffffff

using namespace std;

struct node{
    int v;
    int w;
};

vector<vector<node>> v;
//vector<CNode> v[30010];  error,如果用这个，则在poj会超时。说明
vector对象的初始化，也是需要可观时间的
priority_queue<node> pq;
```

```
int dist[N];
bool vis[N];
int n,m;

bool operator < ( const node & d1, const node & d2 ) {
    return d1.w > d2.w;
void dijkstra()
{
    struct node S;    //源点
    S.v=1;
    S.w=0;
    dist[S.v]=0;
    for(int i=2;i<=n;i++)
        dist[i]=INF;
    pq.push(S);
    while(!pq.empty())
    {
        struct node p=pq.top();
        pq.pop();
        if(vis[p.v]==true)
            continue;
        vis[p.v]=true;
        for(int i=0,j=(int)v[p.v].size();i<j;i++)
        {
            int V=v[p.v][i].v;
            if(vis[V]==false && dist[p.v]+v[p.v]
[i].w<dist[V])
            {
                dist[V]=dist[p.v]+v[p.v][i].w;
                struct node q;
                q.v=V;
                q.w=dist[V];
                pq.push(q);
            }
        }
    }
}
```

Floyd

G[i][j]=min(G[i][j],G[i][k]+G[k][j])，k范围为1–N
G[i][j]=INF，G[i][i]=0
负环条件：G[i][i]<0

题意：给出m对牛的相互关系，求有多少个牛排名是确定的。
用floyed求传递闭包。如果一个牛和其余的牛关系都是确定的，那么这个牛的排名就是确定的了。

```
#include <iostream>
#include <cstdio>

#define MAX 10
#define INF 0xfffffff

using namespace std;

int G[MAX][MAX];

void Floyd();

int N,M;

int main(void)
{
    scanf("%d %d",&N,&M);
```

```
    for(int i=1;i<=N;i++)
        for(int j=1;j<=N;j++)
        {
            if(i==j)
                G[i][j]=0;
            else
                G[i][j]=INF;
        }

    for(int i=0;i<M;i++)
    {
        int u,v;
        scanf("%d %d",&u,&v);
        G[u][v]=1;
    }
    Floyd();
    int ans=0;
    for(int i=1;i<=N;i++)
    {
        int degree=0;
        for(int j=1;j<=N;j++)
        {
            if(i==j)
                continue;
            if(G[i][j]!=INF || G[j][i]!=INF)
                degree++;
        }
        if(degree==N-1)
            ans++;
    }
    printf("%d\n",ans);
    return 0;
}

void Floyd()
{
    for(int k=1;k<=N;k++)
        for(int i=1;i<=N;i++)
            for(int j=1;j<=N;j++)
                G[i][j]=min(G[i][j],G[i][k]+G[k][j]);
}
//求有向图的传递闭包
//求传递闭包,转变为图,即求任意两点是否连通
//考虑每个点的出度和入度,若等于N-1则可以确定
//所以如果这个点到另一个点的距离为INF,则不可以确定
```

DisjointSet

```
void Union(int p,int q)
{
    int i=root(p);
    int j=root(q);
    if(i==j)
        return;
    Id[i]=j;
     //加上对于根的某属性的修改
}

int root(int i)
{
    //为了记录i的爸爸
    int temp=Id[i];
    if(temp==i)
```

```
        return i;
    Id[i]=root(temp);
    //加上对某个结点的某个属性更新
    return Id[i];
}
```

题意：给出n对关系，每对关系表示a，b是情侣，问其中是否有同性恋

```
#include <iostream>
#include <cstdio>
#include <cstring>

#define MAX 10

using namespace std;

void Union(int p,int q);
int root(int i);

int Id[MAX];
int relation[MAX];     //为1则和根异性,0则为同性
bool ans=false;

int main(void)
{
    int N;
    scanf("%d",&N);
    int cnt=0;
    while(cnt<N)
    {
        cnt++;
        ans=false;
        int n,m;
        scanf("%d %d",&n,&m);
        memset(relation,0,sizeof(relation));
        for(int i=0;i<=n;i++)
            Id[i]=i;
        for(int i=0;i<m;i++)
        {
            int a,b;
            scanf("%d %d",&a,&b);
            if(root(a)==root(b))    //如果同根且relation值一样则是同性
            {
                if(relation[a]!=(relation[b]+1)%2)
        ans=true;
            }
            else
                Union(a,b);
        }
        printf("Scenario #%d:\n",cnt);
        if(ans==true)
            printf("Suspicious bugs found!\n");
        else
            printf("No suspicious bugs found!\n");
        printf("\n");
    }
    return 0;
}

void Union(int p,int q)
{
    int i=root(p);
    int j=root(q);
    if(i==j)
        return;
    Id[i]=j;
    relation[i]=(relation[p]-relation[q]+1)%2;  //why
    //根的relation一定为0
    //现在i成为了j的子树,所以要更新i的relation值
    //那p,q是异性,那就要修改i的relation值,i相对于j的性别就是
    //!(rank[p]^rank[q]),异或相同为0
    //即p,q同性那relaition[i]为1,p,q异性的话本来relaition[i]就是0
    //接着就靠root来改接下来的结点的relation值了,只要保证在下一次Union之前root
}

int root(int i)
```

```
{
    //为了记录i的爸爸
    int temp=Id[i];
    if(temp==i)
        return i;
    Id[i]=root(temp);
    relation[i]=(relation[i]+relation[temp])%2;
    //或者写成relation[i] = relation[i]^relation[temp];
    //relation[temp]是i的父亲与最新的根结点的关系
    //现在知道了relation[temp]和自己跟父亲的关系(和temp的关系)relation[i],
    //要更新自己现在和最新的根结点的关系,因为路径压缩连到根
    //可以列出来这是个异或关系
    return Id[i];
}
//这题也可以通过判二分图来做
//dfs+染色
//n阶无向图是一个二分图当且仅当图中没有无奇数圈
```

题目大意：开始有N堆砖块，编号为1,2....N，每堆都只有一个。之后可以进行两种操作：
（1）M X Y 将编号为X的砖块所在的那堆砖拿起来放到编号为Y的砖块所在的堆上；
（2）C X 查询编号为X的砖块所在的堆中，在砖块X下方的所有砖块的数目

```
#include <iostream>
#include <cstdio>
#define MAX 10

using namespace std;

int Id[MAX];
int under[MAX];          //记录该编号的方块下面有多少方块
int sum[MAX];            //若Id[i]=i,sum[i]记录该堆方块的个数

void Union(int p,int q);
int root(int i);

int main(void)
{
    for(int i=0;i<MAX;i++)
    {
        sum[i]=1;
        Id[i]=i;
        under[i]=0;
    }

    int p;
    scanf("%d",&p);
    for(int i=1;i<=p;i++)
    {
        char op[20];
        scanf("%s",op);
        if(op[0]=='M')
        {
            int a,b;
            scanf("%d %d",&a,&b);
            Union(a,b);      //把a放到b的上面
        }
        else
        {
            int a;
            scanf("%d",&a);
            root(a);
            printf("%d\n",under[a]);
        }
    }
    return 0;
}

void Union(int p,int q)
{
    int i=root(p);
    int j=root(q);
    if(i==j)
        return;
    Id[i]=j;
    under[i]=sum[j];    //因为i是根,永远这时候under[i]=0
    sum[j]+=sum[i];
}
```

```
int root(int i)
{
//    if(Id[i]==i)
//        return i;
//    int t=root(Id[i]);
//    //变量t的意义
//    //明明可以写成Id[i]=root(Id[i]];
//    //先拿t保存一下,然后因为要更新under数组,如果修改了根节点,无法一路累加under
//    //因为修改完了Id[i]就是根节点为0
//    under[i]+=under[Id[i]];
//    Id[i]=t;
    int temp=Id[i];
    if(temp==i)
        return i;
    Id[i]=root(temp);
    under[i]+=under[temp];
    return Id[i];
//under是一个相对值,相对于根的值
    //所以对于i来说,temp是他的爸爸,under[i]本来记录的值是相对于temp(i原来的爸爸)的under
值
    //temp在之前更新完了,那i现在相对于现在爸爸(即根)的under值,就是under[i]+under[temp]
}
```

Kruskal算法

vector存所有的边

排序，每次取出最小的边，判断这条边的两个端点是否形成环，直到收入了n−1条边

```
#include <iostream>
#include <cstdio>
#include <algorithm>
#include <vector>
#define MAX 10

using namespace std;

struct node{
    int u,v;
    int w;
};

bool comp(struct node a,struct node b)
{
    return a.w<b.w;
}

int Kruskal(int n);
void Union(int p,int q);
int root(int i);
bool connected(int p,int q);

vector<struct node> G;
int Id[MAX];

int main(void)
{
    int N;
    while(cin>>N)
    {
        G.clear();
        for(int i=1;i<=N;i++)
        {
            for(int j=1;j<=N;j++)
            {
                struct node p;
                if(i==j)
                {
                    scanf("%d",&p.w);
                    continue;
                }
                p.u=i;
                p.v=j;
                scanf("%d",&p.w);
                G.push_back(p);
            }
        }
```

```
        sort(G.begin(),G.end(),comp);
        printf("%d\n",Kruskal(N));
    }
    return 0;
}

void Union(int p,int q)
{
    int i=root(p);
    int j=root(q);
    if(i==j)
        return;
    Id[i]=j;
}

bool connected(int p,int q)
{
    return root(p)==root(q);
}

int root(int i)
{
//    if(Id[i]==i)
//        return i;
//    Id[i]=root(Id[i]);
//    return Id[i];
    while(Id[i]!=i)
    {
        Id[i]=Id[Id[i]];
        i=Id[i];
    }
    return i;
}

int Kruskal(int n)
{
    int ans=0;
    int cnt=0;
    for(int i=0;i<=n;i++)
        Id[i]=i;

    for(int i=0;i<G.size();i++)
    {
        struct node p=G[i];
        if(connected(p.u,p.v)==false)
        {
            Union(p.u,p.v);
            ans+=p.w;
            cnt++;
        }
        if(cnt==n-1)
            break;
    }
    return ans;
}
```

Edmonds_Karp

bfs找到一条源到汇的路，并记录下路径，求出路径上最短的边，插入反向边，并渐小原来的边，大小和那条最短的边相同
最外面再套个循环，直到更新后的图没有源到汇的路，sum求和即为答案
邻接矩阵存储，注意重边

```
#include <iostream>
#include <cstdio>
#include <queue>
#include <cstring>

#define MAX 300

using namespace std;

int G[MAX][MAX];
bool vis[MAX];
int pre[MAX];
int m,n;
```

```
unsigned int Edmonds_Karp();

int main(void)
{
    while(scanf("%d %d",&m,&n)!=EOF)
    {
        memset(G,0,sizeof(G));
        for(int i=1;i<=m;i++)
        {
            int a,b,c;
            scanf("%d %d %d",&a,&b,&c);
            G[a][b]+=c;              //可能重边
        }
        int ans=0;
        int temp;
        while(1)
        {
            temp=Edmonds_Karp();
            if(temp==0)
                break;
            ans+=temp;
        }
        printf("%d\n",ans);
    }
    return 0;
}

//1为源,n为汇
unsigned int Edmonds_Karp()
{
    //初始化
    memset(vis,0,sizeof(vis));
    memset(pre,0,sizeof(pre));
    //BFS
    deque<int> Q;
    int S=1;
    Q.push_back(S);
    vis[S]=true;
    pre[S]=0;
    bool ifRoad=false;
    while(!Q.empty())
    {
        int v=Q.front();
        Q.pop_front();
        for(int i=1;i<=n;i++)
        {
            if(G[v][i]>0 && vis[i]==false)
            {
                vis[i]=true;
                pre[i]=v;
                //找到了源到汇到路
                if(i==n)
                {
                    ifRoad=true;
                    Q.clear();          //queue没有clear
                    break;
                }
                else
                    Q.push_back(i);
            }
        }
    }
    //没有路
    if(ifRoad==false)
        return 0;
    int minRoad=0x7fffffff;
    int v=n;
    while(pre[v]!=0)
    {
        minRoad=min(minRoad,G[pre[v]][v]);
        v=pre[v];
    }
    //插反向边
    v=n;
    while(pre[v]!=0)
    {
        G[pre[v]][v]-=minRoad;
        G[v][pre[v]]+=minRoad;
        v=pre[v];
```

```
    }
    return minRoad;
}
```

Tarjan算法

vector模拟邻接表
对每个点都需要进行Tarjan算法，判断一下其是否在之前的点点Tarjan算法中已经做过了没，if(!dfn(i))，不然如果这个图并非连通，那有些点就根本没有做
dfn[i]=low[i]=idx++，将源点进栈，访问他的邻接点，是否访问过用dfn数组判断，如果没有访问过，递归调用Tarjan，并最后让v把low[v]带出来更新low[u]，如果访问过还在栈里，用low[v]更新low[u]，邻接点全访问过后，如果发现dfn[u]==low[u]，则从栈里开始弹出结点，直到弹出的是u，弹出来的这一部分就是强连通分量

题目大意：在一个牧群中，有N个奶牛，给定M对关系（A,B）表示A仰慕B，而且仰慕关系有传递性，问被所有奶牛（除了自己）仰慕的奶牛个数

技巧：求出强连通分量后进行染色，缩点，形成DAG，再利用DAG的性质，一般是判断出度入度

```
#include <iostream>
#include <cstdio>
#include <vector>
#include <stack>
#include <cstring>

#define MAX 10

using namespace std;

void Tarjan(int u);     //u当前结点

vector<vector<int>> G;
stack<int> S;
int dfn[MAX];
int low[MAX];
bool vis[MAX];          //是否在栈中
int color[MAX];
int idx=1;              //记录dfs的编号
int clr=0;             //染色标记

int main(void)
{
    int n,m;
    while(scanf("%d %d",&n,&m)!=EOF)
    {
        G.clear();
        G.resize(n+1);
        for(int i=0;i<m;i++)
        {
            int a,b;
            scanf("%d %d",&a,&b);
            G[a].push_back(b);
        }
        //初始化
        memset(vis,0,sizeof(vis));
        memset(dfn,0,sizeof(dfn));
        memset(low,0,sizeof(low));
        memset(color,0,sizeof(color));
        clr=0;
        idx=1;
        //WA的教训
        for(int i=1;i<=n;i++)
        {
            if(!dfn[i])
                Tarjan(i);
        }
        //  看一下染色结果
//              for(int i=1;i<=n;i++)
//                  printf("%d %d\n",i,color[i]);

        //把强连通分量缩点,clr的值为强连通分量的个数
        //用数组ans保存每个颜色的强连通分量的出度
        //定理:有向无环图中唯一出度为0的点，一定可以由任何点出发均可达
        //枚举所有的边
```

```
        int ans[MAX];
        memset(ans,0,sizeof(ans));
        for(int i=0;i<G.size();i++)
            for(int j=0;j<G[i].size();j++)
            {
                if(color[i]==color[G[i][j]])
                    continue;
                else
                    ans[color[i]]++;
            }
        int cnt=0;
        int idx=-1;
        for(int i=1;i<=clr;i++)
        {
            if(ans[i]==0)
            {
                cnt++;
                idx=i;
            }
        }
        if(cnt==1)
        {
            int sum=0;
            for(int i=1;i<=n;i++)
            {
                if(color[i]==idx)
                    sum++;
            }
            printf("%d\n",sum);
        }
        else
            printf("0\n");
    }
    return 0;
}

void Tarjan(int u)
{
    dfn[u]=low[u]=idx++;
    S.push(u);
    vis[u]=true;
    for(int i=0;i<G[u].size();i++)
    {
        int v=G[u][i];
        if(!dfn[v])        //如果没访问过
        {
            Tarjan(v);
            low[u]=min(low[u],low[v]);
        }
        else if(vis[v]==true)   //如果访问过还在栈里
        {
            low[u]=min(low[u],low[v]);
        }
    }
    if(dfn[u]==low[u])
    {
        int v;
        clr++;
        do
        {
            v=S.top();
            S.pop();
            color[v]=clr;
            vis[v]=false;
//            printf("%d ",v);
        }while(u!=v);
//        printf("\n");
    }
}
//定理:有向无环图中唯一出度为0的点，一定可以由任何点出发均可达
//由于无环，所以从任何点出发往前走，必然终止于一个出度为0的点

//1. 求出所有强连通分量
//2. 每个强连通分量缩成一点，则形成一个有 向无环图DAG。
//3. DAG上面如果有唯一的出度为0的点,则该点 能被所有的点可达。
//那么该点所代表的连通分 量上的所有的原图中的点，都能被原图中的所有点可达，则该连通分量的点数，
就是答案。
//4. DAG上面如果有不止一个出度为0的点，则这些点互相不可达，原问题无解，答案为0
```

TopSort

思想：维护一个入度为0的队列，依次出队，出队以后把邻接点的入度减1，如果减为0入队，直到队列空
判断有向图环：队列空后，判断每个点的入度，检查每个结点入度，如果入度有不为0的，则有环
可判断DAG是否任意两点都可到达，考虑DAG是一条长链，若存在分叉，那分叉的那两个点互相不可达到，即判断队列里是否超过两个元素。

```
#include <iostream>
#include <cstdio>
#include <vector>
#include <queue>
#include <cstring>

#define MAX 20

using namespace std;

vector< vector<int> > v;
int in[MAX];

void Topsort(int n,int m);

int main(void)
{
    int n,m;        //顶点,边数,顶点从1开始
    scanf("%d %d",&n,&m);
    memset(in,0,sizeof(in));
    v.resize(n+1);
    for(int i=1;i<=m;i++)
    {
        int a,b;
        scanf("%d %d",&a,&b);
        v[a].push_back(b);
        in[b]++;
    }
    Topsort(n,m);
    return 0;
}

void Topsort(int n,int m)
{
    queue<int> q;
    for(int i=1;i<=n;i++)
    {
        if(in[i]==0)
            q.push(i);          //入度为0的结点入队
    }
    while(!q.empty())
    {
        int u=q.front();
        printf("%d ",u);
        q.pop();
        for(int i=0;i<v[u].size();i++)
        {
            int a=v[u][i];
            in[a]--;                //修改邻接点入度
            if(in[a]==0)
                q.push(a);
        }
    }
    for(int i=1;i<=n;i++)           //检查每个点入度是否为0,如果不为0,则存在环
        if(in[i]!=0)                //判断有向图是否有环
        {
            printf("loop\n");
            break;
        }
}
```

Segment Tree 统计区间和 HDU1166

```
#include <iostream>
#include <cstdio>

#define N 51000

using namespace std;

void BuildTree(int root,int L,int R);
void Update(int root,int i,int V);
int Query(int root,int s,int t);

struct Node{
    int L,R;
    int sum;
};

struct Node T[4*N];

void BuildTree(int root,int L,int R)
{
    T[root].L=L;
    T[root].R=R;
    if(T[root].L==T[root].R)
    {
        T[root].sum=0;
        return;
    }
    int mid=(L+R)/2;
    BuildTree(root<<1,L,mid);
    BuildTree(root<<1|1,mid+1,R);
    T[root].sum=T[root<<1].sum+T[root<<1|1].sum;
}

void Update(int root,int i,int V)
{
    if(T[root].L==T[root].R)
    {
        T[root].sum+=V;
        return;
    }
    int mid=(T[root].L+T[root].R)/2;
    if(i<=mid)
        Update(root<<1,i,V);
    else
        Update(root<<1|1,i,V);
    T[root].sum=T[root<<1].sum+T[root<<1|1].sum;
}

int Query(int root,int s,int t)
{
    if(s<=T[root].L && t>=T[root].R)
        return T[root].sum;;
    int mid=(T[root].L+T[root].R)/2;
    int ans=0;
    if(s<=mid)
        ans+=Query(root<<1,s,t);
    if(t>mid)
        ans+=Query(root<<1|1,s,t);
    return ans;
}
```

Segment Tree 统计区间最值 HDU1754

```
#include <iostream>
#include <cstdio>

#define N 210000

using namespace std;

void BuildTree(int root,int L,int R);
void Update(int root,int i,int V);
int Query(int root,int s,int t);

struct Node{
    int L,R;
    int max;
};

struct Node T[4*N];

void BuildTree(int root,int L,int R)
{
    T[root].L=L;
    T[root].R=R;
    T[root].max=0;
    if(T[root].L==T[root].R)
        return;
    int mid=(L+R)/2;
    BuildTree(root<<1,L,mid);
    BuildTree(root<<1|1,mid+1,R);
    T[root].max=max(T[root<<1].max,T[root<<1|1].max);
}

void Update(int root,int i,int V)
{
    if(T[root].L==T[root].R)
    {
        T[root].max=V;
        return;
    }
    int mid=(T[root].L+T[root].R)/2;
    if(i<=mid)
        Update(root<<1,i,V);
    else
        Update(root<<1|1,i,V);
    T[root].max=max(T[root<<1].max,T[root<<1|1].max);
}

int Query(int root,int s,int t)
{
    if(s<=T[root].L && t>=T[root].R)
        return T[root].max;;
    int mid=(T[root].L+T[root].R)/2;
    int ans=0;
    if(s<=mid)
        ans=max(ans,Query(root<<1,s,t));
    if(t>mid)
        ans=max(ans,Query(root<<1|1,s,t));
    return ans;
}
```

Segment Tree 区间合并 HDU3308

题目大意:单点更新，求区间最长连续上升子序列

```cpp
#include <iostream>
#include <cstdio>

const int N=110000;

using namespace std;

void BuildTree(int root,int L,int R);
void Update(int root,int i,int V);
int Query(int root,int s,int t);
void pushUp(int root);

struct Node{
    int L,R;
    int lx,rx,mx;   //lx,r分别为取最左端点和最右端点的LCIS
};

struct Node T[4*N];
int A[N];

void pushUp(int root)
{
    //左儿子最右边的值 < 右儿子最左边的值 说明可以合并
    if(A[T[root<<1].R]<A[T[root<<1|1].L])
    {
        //lx = (左儿子的lx == 左儿子的len) ? 左儿子的len + 右儿子的lx : 左
儿子的lx
        if(T[root<<1].lx==T[root<<1].R-T[root<<1].L+1)
            T[root].lx=T[root<<1].lx+T[root<<1|1].lx;
        else
            T[root].lx=T[root<<1].lx;
        //rx = (右儿子的rx == 右儿子的len) ? 右儿子的len + 左儿子的rx : 右
儿子的rx
        if(T[root<<1|1].rx==T[root<<1|1].R-T[root<<1|1].L+1)
            T[root].rx=T[root<<1|1].rx+T[root<<1].rx;
        else
            T[root].rx=T[root<<1|1].rx;
        //mx = MAX(左儿子的rx + 右儿子的lx, 左儿子的mx, 右儿子的mx, lx,
rx)
        int max1=max(T[root<<1].mx,T[root<<1|1].mx);
        int max2=max(T[root].lx,T[root].rx);
        T[root].mx=max(max(max1,max2),T[root<<1].rx+T[root<<1|
1].lx);
    }
    //左儿子最右边的值 >= 右儿子最左边的值
    else
    {
        //lx = 左儿子的lx
        T[root].lx=T[root<<1].lx;
        //rx = 右儿子的rx
        T[root].rx=T[root<<1|1].rx;
        //mx = MAX(c左儿子的mx, 右儿子的mx)
        T[root].mx=max(T[root<<1].mx,T[root<<1|1].mx);
    }
}

void BuildTree(int root,int L,int R)
{
    T[root].L=L;
    T[root].R=R;
    if(T[root].L==T[root].R)
    {
        T[root].lx=T[root].rx=T[root].mx=1;
        return;;
```

```
    }
    int mid=(L+R)/2;
    BuildTree(root<<1,L,mid);
    BuildTree(root<<1|1,mid+1,R);
    pushUp(root);
}

void Update(int root,int i,int V)
{
    if(T[root].L==T[root].R)
    {
        A[i]=V;     //WA
        return;
    }
    int mid=(T[root].L+T[root].R)/2;
    if(i<=mid)
        Update(root<<1,i,V);
    else
        Update(root<<1|1,i,V);
    pushUp(root);
}

//和之前的不同,分三类
int Query(int root,int s,int t)
{
    if(s==T[root].L && t==T[root].R)
        return T[root].mx;
    int mid=(T[root].L+T[root].R)/2;
    if(t<=mid)
        return Query(root<<1,s,t);
    else if(s>mid)
        return Query(root<<1|1,s,t);
    else
    {
        int lx=Query(root<<1,s,mid);
        int rx=Query(root<<1|1,mid+1,t);
        int ans=0;
        //可以合并的情况
        //取min是为了防止超过区间长度,为什么会超？？？
        //因为查询出来的是[T[root].L,T[root].R]的LCIS
        //而查询区间可能只是[s,T[root].R],只是查询的区间的一部分,所以要取min
        if(A[T[root<<1].R]<A[T[root<<1|1].L])
            ans=min(mid-s+1,T[root<<1].rx)+min(t-mid,T[root<<1|
1].lx);  //WA打错
        return max(ans,max(lx,rx));
    }
}
```

Segment Tree 区间更新 POJ3468

```
#include <iostream>
#include <cstdio>

#define N 110000

using namespace std;

void BuildTree(int root,int L,int R);
void Update(int root,int s,int t,long long V);
long long Query(int root,int s,int t);

struct Node{
    int L,R;
    long long sum;
```

```
    long long inc;
    int len(){
        return R-L+1;
    }
};

struct Node T[4*N];

void pushDown(int root)
{
    if(T[root].inc!=0)
    {
        T[root<<1].sum+=T[root].inc*T[root<<1].len();
        T[root<<1|1].sum+=T[root].inc*T[root<<1|1].len();
        T[root<<1].inc+=T[root].inc;
        T[root<<1|1].inc+=T[root].inc;
        T[root].inc=0;
    }
}

void BuildTree(int root,int L,int R)
{
    T[root].L=L;
    T[root].R=R;
    T[root].inc=0;  //必须放在外面
    if(T[root].L==T[root].R)
    {
        T[root].sum=0;
        return;
    }
    int mid=(L+R)/2;
    BuildTree(root<<1,L,mid);
    BuildTree(root<<1|1,mid+1,R);
    T[root].sum=T[root<<1].sum+T[root<<1|1].sum;
}

void Update(int root,int s,int t,long long V)
{
    if(T[root].L>=s && T[root].R<=t)
    {
        T[root].sum+=V*T[root].len();
        T[root].inc+=V;
        return;
    }
    pushDown(root);
    int mid=(T[root].L+T[root].R)/2;
    if(s<=mid)
        Update(root<<1,s,t,V);
    if(t>mid)
        Update(root<<1|1,s,t,V);
    T[root].sum=T[root<<1].sum+T[root<<1|1].sum;
}

long long Query(int root,int s,int t)
{
    if(s<=T[root].L && t>=T[root].R)
        return T[root].sum;
    pushDown(root);
    int mid=(T[root].L+T[root].R)/2;
    long long ans=0;            //long long
    if(s<=mid)
        ans+=Query(root<<1,s,t);
    if(t>mid)
        ans+=Query(root<<1|1,s,t);
    return ans;
```

```
}
```

Segment Tree 区间合并+双标记 HDU3397

```
#include <iostream>
#include <cstdio>

const int N=110000;

using namespace std;

void BuildTree(int root,int L,int R);
void Update_same(int root,int s,int t,int V);
void Update_rev(int root,int s,int t);
int QueryLC1(int root,int s,int t);
int QuerySum(int root,int s,int t);
void pushUp(int root);
void pushDown(int root);

void Exchange(int root);
void solve(int root);

struct Node{
    int L,R;
    int lx1,rx1,mx1;
    int lx0,rx0,mx0;
    int sum1,sum0;   //1,0的个数
    int same;   //01的延迟标记
    int rev;    //翻转的延迟标记
    int len(){
        return R-L+1;
    }
};

struct Node T[4*N];

int main(void)
{
    int t;
    scanf("%d",&t);
    while(t--)
    {
        int n,m;
        scanf("%d %d",&n,&m);
        BuildTree(1, 0, n-1);
        for(int i=1;i<=m;i++)
        {
            int op,s,t;
            scanf("%d %d %d",&op,&s,&t);
            if(op==0)
                Update_same(1, s, t, 0);
            else if(op==1)
                Update_same(1, s, t, 1);
            else if(op==2)
                Update_rev(1, s, t);
            else if(op==3)
                printf("%d\n",QuerySum(1, s, t));
            else if(op==4)
                printf("%d\n",QueryLC1(1, s, t));
        }
    }
}

void pushUp(int root)
{
    //先赋值成儿子的个数
    T[root].lx0=T[root<<1].lx0;
    T[root].lx1=T[root<<1].lx1;
    T[root].rx0=T[root<<1|1].rx0;
    T[root].rx1=T[root<<1|1].rx1;
    T[root].sum0=T[root<<1].sum0+T[root<<1|1].sum0;
    T[root].sum1=T[root<<1].sum1+T[root<<1|1].sum1;

    //判断是否能区间合并
    if(T[root].lx1==T[root<<1].len())
        T[root].lx1+=T[root<<1|1].lx1;
    if(T[root].lx0==T[root<<1].len())
        T[root].lx0+=T[root<<1|1].lx0;
    if(T[root].rx1==T[root<<1|1].len())
        T[root].rx1+=T[root<<1].rx1;
    if(T[root].rx0==T[root<<1|1].len())
        T[root].rx0+=T[root<<1].rx0;
    T[root].mx1=max(T[root<<1].mx1,T[root<<1|1].mx1);
    T[root].mx0=max(T[root<<1].mx0,T[root<<1|1].mx0);
    T[root].mx1=max(T[root].mx1,T[root<<1].rx1+T[root<<1|1].lx1);
    T[root].mx0=max(T[root].mx0,T[root<<1].rx0+T[root<<1|1].lx0);
}
```

```
void pushDown(int root)
{
    if(T[root].same!=-1)
    {
        T[root<<1].rev=T[root<<1|1].rev=0;
        T[root<<1].same=T[root<<1|1].same=T[root].same;
        if(T[root].same==1)
        {
            T[root<<1].sum1=T[root<<1].lx1=T[root<<1].rx1=T[root<<1].mx1=T[root<<1].len();
            T[root<<1].sum0=T[root<<1].lx0=T[root<<1].rx0=T[root<<1].mx0=0;
            T[root<<1|1].sum1=T[root<<1|1].lx1=T[root<<1|1].rx1=T[root<<1|1].mx1=T[root<<1|
1].len();
            T[root<<1|1].sum0=T[root<<1|1].lx0=T[root<<1|1].rx0=T[root<<1|1].mx0=0;
        }
        else if(T[root].same==0)
        {
            T[root<<1].sum1=T[root<<1].lx1=T[root<<1].rx1=T[root<<1].mx1=0;
            T[root<<1].sum0=T[root<<1].lx0=T[root<<1].rx0=T[root<<1].mx0=T[root<<1].len();
            T[root<<1|1].sum1=T[root<<1|1].lx1=T[root<<1|1].rx1=T[root<<1|1].mx1=0;
            T[root<<1|1].sum0=T[root<<1|1].lx0=T[root<<1|1].rx0=T[root<<1|1].mx0=T[root<<1|
1].len();
        }
        T[root].same=-1;
    }
    if(T[root].rev==1)
    {
        if(T[root<<1].same!=-1)
            solve(root<<1);
        else
        {
            T[root<<1].rev=T[root<<1].rev^1;
            Exchange(root<<1);
        }
        if(T[root<<1|1].same!=-1)
            solve(root<<1|1);
        else
        {
            T[root<<1|1].rev=T[root<<1|1].rev^1;
            Exchange(root<<1|1);
        }
        T[root].rev=0;
    }
}

void BuildTree(int root,int L,int R)
{
    T[root].L=L;
    T[root].R=R;
    T[root].same=-1;
    T[root].rev=0;
    //WA,必须初始化,因为多组数据,可能后来一组数据小了,pushUp会用到之前的数据
    T[root].lx1=T[root].rx1=T[root].mx1=T[root].sum1=0;
    T[root].lx0=T[root].rx0=T[root].mx0=T[root].sum0=0;
    if(T[root].L==T[root].R)
    {
        //在BuildTree中直接插入数据,就不用写insert了
        int x;
        scanf("%d",&x);
        if(x==1)
            T[root].lx1=T[root].rx1=T[root].mx1=T[root].sum1=1;
        else if(x==0)
            T[root].lx0=T[root].rx0=T[root].mx0=T[root].sum0=1;
        return;
    }
    int mid=(L+R)/2;
    BuildTree(root<<1,L,mid);
    BuildTree(root<<1|1,mid+1,R);
    pushUp(root);
}

void Update_same(int root,int s,int t,int V)
{
    if(T[root].L>=s && T[root].R<=t)
    {
        T[root].rev=0;
        if(V==1)
        {
            T[root].lx1=T[root].rx1=T[root].mx1=T[root].sum1=T[root].len();
            T[root].lx0=T[root].rx0=T[root].mx0=T[root].sum0=0;
        }
        else if(V==0)
        {
            T[root].lx0=T[root].rx0=T[root].mx0=T[root].sum0=T[root].len();
            T[root].lx1=T[root].rx1=T[root].mx1=T[root].sum1=0;
        }
        T[root].same=V;     //延迟标记
        return;
    }
    pushDown(root);
    int mid=(T[root].L+T[root].R)/2;
```

```
    if(s<=mid)
        Update_same(root<<1,s,t,V);
    if(t>mid)
        Update_same(root<<1|1,s,t,V);
    pushUp(root);
}

//修改same标记对区间的处理
void solve(int root)
{
    T[root].same=T[root].same^1;
    if(T[root].same==1)
    {
        T[root].lx1=T[root].rx1=T[root].mx1=T[root].sum1=T[root].len();
        T[root].lx0=T[root].rx0=T[root].mx0=T[root].sum0=0;
    }
    else if(T[root].same==0)
    {
        T[root].lx0=T[root].rx0=T[root].mx0=T[root].sum0=T[root].len();
        T[root].lx1=T[root].rx1=T[root].mx1=T[root].sum1=0;
    }
}

//修改rev标记对区间的翻转
void Exchange(int root)
{
    swap(T[root].lx1,T[root].lx0);
    swap(T[root].rx1,T[root].rx0);
    swap(T[root].mx1,T[root].mx0);
    swap(T[root].sum1,T[root].sum0);
}

void Update_rev(int root,int s,int t)
{
    if(T[root].L>=s && T[root].R<=t)
    {
        //如果rev在[a,b]上遇见一个same标记,那么只需要对same进行修改即可
        if(T[root].same!=-1)
        {
            solve(root);
            return;
        }
        T[root].rev=T[root].rev^1;
        Exchange(root);
        return;
    }
    pushDown(root);
    int mid=(T[root].L+T[root].R)/2;
    if(s<=mid)
        Update_rev(root<<1,s,t);
    if(t>mid)
        Update_rev(root<<1|1,s,t);
    pushUp(root);
}

int QueryLC1(int root,int s,int t)
{
    if(s==T[root].L && t==T[root].R)
        return T[root].mx1;
    pushDown(root);
    int mid=(T[root].L+T[root].R)/2;
    if(t<=mid)
        return QueryLC1(root<<1,s,t);
    else if(s>mid)
        return QueryLC1(root<<1|1,s,t);
    else
    {
        int lx=QueryLC1(root<<1,s,mid);
        int rx=QueryLC1(root<<1|1,mid+1,t);
        int ans=0;
        ans=min(mid-s+1,T[root<<1].rx1)+min(t-mid,T[root<<1|1].lx1);
        return max(ans,max(lx,rx));
    }

}

int QuerySum(int root,int s,int t)
{
    if(s<=T[root].L && t>=T[root].R)
        return T[root].sum1;;
    pushDown(root);
    int mid=(T[root].L+T[root].R)/2;
    int ans=0;
    if(s<=mid)
        ans+=QuerySum(root<<1,s,t);
    if(t>mid)
        ans+=QuerySum(root<<1|1,s,t);
    return ans;
}
```

```
//考虑same和rev之间的关系
//假设same在[a,b]上遇到一个rev,那么把该区间上原先有的rev删除.
//如果rev在[a,b]上遇见一个same标记,那么只需要对same进行修改即可
//否则将0 1的数据进行交换
```

Segment Tree 二分+区间更新 HDU6070

```
#include <iostream>
#include <cstdio>
#include <cstring>

using namespace std;
const int N=11;
const double esp=1e-5;

struct Node{
    int L,R;
    double min;
    int inc;
};
struct Node T[4*N];
int pre[N];
int A[N];

void BuildTree(int root,int L,int R,double v);
void Update(int root,int s,int t,int v);
double Query(int root,int s,int t);
void pushDown(int root);

int main(void)
{
    int t;
    scanf("%d",&t);
    while(t--)
    {
        int n;
        scanf("%d",&n);
        for(int i=1;i<=n;i++)
            scanf("%d",&A[i]);
        //二分
        double s=0,t=1;
        while(t-s>esp)
        {
            bool flag=true;
            double mid=(s+t)/2.0;
            BuildTree(1, 1, n, mid);
            memset(pre,0,sizeof(pre));
            for(int i=1;i<=n;i++)
            {
                Update(1,pre[A[i]]+1,i,1);
                double ans=Query(1,1,i);
                if(ans<=mid*(i+1))
                {
                    flag=false;
                    break;
                }
            }
            if(flag==false)
                t=mid;
            else
                s=mid;
        }
        printf("%lf\n",s);
    }
    return 0;
}

void BuildTree(int root,int L,int R,double v)
{
    T[root].L=L;
    T[root].R=R;
    T[root].inc=0;
    if(T[root].L==T[root].R)
    {
        T[root].min=T[root].L*v;
        return;
    }
    int mid=(L+R)/2;
```

```
    BuildTree(root<<1,L,mid,v);
    BuildTree(root<<1|1,mid+1,R,v);
    T[root].min=min(T[root<<1].min,T[root<<1|1].min);
}

void Update(int root,int s,int t,int v)
{
    if(s<=T[root].L && T[root].R<=t)
    {
        T[root].min+=v;
        T[root].inc+=1;
        return;
    }
    pushDown(root);
    int mid=(T[root].L+T[root].R)/2;
    if(s<=mid)
        Update(root<<1,s,t,v);
    if(t>mid)
        Update(root<<1|1,s,t,v);
    T[root].min=min(T[root<<1].min,T[root<<1|1].min);
}

void pushDown(int root)
{
    if(T[root].inc!=0)
    {
        T[root<<1].inc+=T[root].inc;
        T[root<<1|1].inc+=T[root].inc;
        T[root<<1].min+=T[root].inc;
        T[root<<1|1].min+=T[root].inc;
        T[root].inc=0;
    }
}

double Query(int root,int s,int t)
{
    if(s<=T[root].L && t>=T[root].R)
        return T[root].min;
    pushDown(root);
    int mid=(T[root].L+T[root].R)/2;
    double ans=1e9;
    if(s<=mid)
        ans=min(ans,Query(root<<1,s,t));
    if(t>mid)
        ans=min(ans,Query(root<<1|1,s,t));
    return ans;
}

//线段树区间更新求最值
//二分答案,线段树维护区间不重复的数的个数s
//即s/(r-l+1)<=mid
//即s+mid*l<=mid*(r+1)
//=>线段树维护's+mid*l的min
```

Segment Tree 离散化 POJ2528

```
#include <iostream>
#include <cstdio>
#include <algorithm>
#include <cstring>

using namespace std;

void BuildTree(int root,int L,int R);
bool Post(int root,int s,int e);

struct node{
    int L,R;
    bool covered;
};

struct Poster{
    int L,R;
};

struct node T[1000000];
struct Poster poster[10100];        //存海报信息
int x[20200];                      //排序数组
int myHash[10000100];                    //离散化
```

```
int main(void)
{
    int n;
    int q;
    scanf("%d",&n);
    for(int j=0;j<n;j++)
    {
        scanf("%d",&q);
        int cnt=0;
        memset(myHash,0,sizeof(myHash));
        for(int i=0;i<q;i++)
        {
            scanf("%d %d",&poster[i].L,&poster[i].R);
            x[cnt++]=poster[i].L;
            x[cnt++]=poster[i].R;
        }
        sort(x,x+cnt);
        cnt=(int)(unique(x,x+cnt)-x);       //去重
        //离散化
        int interval=0;
        for(int i=0;i<cnt;i++)
        {
            myHash[x[i]]=interval++;
//            if(x[i+1]-x[i]==1)
//                interval++;
//            else
//                interval=interval+2;
        }
        BuildTree(0,0,interval-1);
        int ans=0;
        for(int i=q-1;i>=0;i--)
        {
            if(Post(0,myHash[poster[i].L],myHash[poster[i].R]))
                ans++;
        }
        printf("%d\n",ans);
    }
    return 0;
}
```

KMP

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

#define MAX 20

using namespace std;

void getNext(char *s,int len);
int KMP(char *s,char *p,int sLen,int pLen);

int KMPnext[MAX];

int main(void)
{
    char s[100];
    char p[100];
    scanf("%s",s);
    scanf("%s",p);
    getNext(p,strlen(p));
    cout<<KMP(s,p,strlen(s),strlen(p))<<endl;
    return 0;
    return 0;
}

//next数组的意义:以s[i-1]为后缀的串能与s前缀匹配next[i]长度
//s下标从0开始
void getNext(char *p,int len)
{
```

```
    KMPnext[0]=-1;
    int k=-1;
    int j=0;
    while(j<len)
    {
        if(k==-1 || p[j]==p[k])
        {
            k++;
            j++;
            KMPnext[j]=k;
        }
        else
            k=KMPnext[k];       //递归的去找
    }
//    for(int i=0;i<len;i++)
//        printf("%d ",KMPnext[i]);
}

//返回匹配下标，否则返回-1
int KMP(char *s,char *p,int sLen,int pLen)
{
    int i=0;
    int j=0;
    while(i<sLen && j<pLen)
    {
        if(j==-1 || s[i]==p[j])
        {
            i++;
            j++;
        }
        else
            j=KMPnext[j];
    }
    if(j==pLen)
        return i-j;
    else
        return -1;
}

//返回匹配次数
int KMP(char *s,char *p,int sLen,int pLen)
{
    int i=0;
    int j=0;
    int ans=0;
    while(i<sLen && j<pLen)
    {
        if(j==-1 || s[i]==p[j])
        {
            i++;
            j++;
        }
        else
            j=KMPnext[j];
        if(j==pLen)
        {
            ans++;
            j=KMPnext[j-1];
            i--;
        }
    }
    return ans;
}

//利用next数组求循环节
//对于next数组中的i,符合i%(i-next[i])==0 && next[i]!=0,则说明字符串循环
//循环节长度为:i-next[i]
//循环次数为:i/(i-next[i])
//假设s的长度为len,若s存在循环节,则最小循环节的长度L为len-next[len]
//因为next的值是前缀和后缀相等的最大长度,即len-L是最大的,len确定=>L是最小的
```

```
//如果len可以被len-next[len]整除,表明字符串S可以完全由循环节循环组成,循环周期T=len/L。
```

exKMP

```
void getextend(char *S,char *T)
{
    memset(KMPnext,0,sizeof(KMPnext));
    getKMPnext(T);
    int Slen = (int)strlen(S);
    int Tlen = (int)strlen(T);
    int a = 0;
    int MinLen = Slen>Tlen?Tlen:Slen;
    while(a<MinLen && S[a]==T[a]) a++;
    extend[0] = a, a = 0;
    for(int k = 1; k < Slen; k++)
    {
        int p = a+extend[a]-1, L = KMPnext[k-a];
        if( (k-1)+L >= p )
        {
            int j = (p-k+1)>0? (p-k+1) : 0;
            while(k+j<Slen && j<Tlen && S[k+j]==T[j] ) j++;
            extend[k] = j;a = k;
        }
        else extend[k] = L;
    }
}

void getKMPnext(char *T)
{
    int i,length =(int)strlen(T);
    KMPnext[0] = length;
    for(i = 0;i<length-1 && T[i]==T[i+1]; i++);
    KMPnext[1] = i;
    int a = 1;
    for(int k = 2; k < length; k++)
    {
        int p = a+KMPnext[a]-1, L = KMPnext[k-a];
        if( (k-1)+L >= p )
        {
            int j = (p-k+1)>0? (p-k+1) : 0;
            while(k+j<length && T[k+j]==T[j])
                j++;
            KMPnext[k] = j, a = k;
        }
        else KMPnext[k] = L;
    }
}
//KMPnext[i]表示的是T[i...Tlen-1]与T[0...Tlen-1]的最长公共前缀
//extend[i] 表示的是S[i…Slen-1]与T[0…Tlen-1]的最长公共前缀
```

AC自动机

```
#include<cstdio>
#include<cstdlib>
#include<cstring>
#include<iostream>
#include<queue>

using namespace std;

const int N=101000,L=100100;
char s[L];
int num,n;
struct node{
    int son[30];
    int fail,cnt;
}a[N*30];
queue<int> q;

void clear(int x)
```

```
{
    a[x].cnt=0;
    a[x].fail=0;
    memset(a[x].son,0,sizeof(a[x].son));
}

//插入trie树
void trie(char *c)
{
    int l=(int)strlen(c);
    int x=0;
    for(int i=0;i<l;i++)
    {
        int t=c[i]-'a'+1;
        if(!a[x].son[t])
        {
            num++;
            clear(num);
            a[x].son[t]=num;
        }
        x=a[x].son[t];
    }
    a[x].cnt++;
}

//构造失败指针
void buildAC()
{
    while(!q.empty()) q.pop();
    for(int i=1;i<=26;i++)
        if(a[0].son[i])
            q.push(a[0].son[i]);
    while(!q.empty())
    {
        int x=q.front();q.pop();
        int fail=a[x].fail;
        for(int i=1;i<=26;i++)
        {
            int y=a[x].son[i];
            if(y)
            {
                a[y].fail=a[fail].son[i];
                q.push(y);
            }
            else a[x].son[i]=a[fail].son[i];
        }
    }
}

//匹配,返回trie树上的串匹配c几次
int find(char *c)
{
    int l=(int)strlen(c);
    int x=0,ans=0;
    for(int i=0;i<l;i++)
    {
        int t=c[i]-'a'+1;
        while(x && !a[x].son[t]) x=a[x].fail;
        x=a[x].son[t];
        int p=x;
        while(p && a[p].cnt!=-1)
        {
            ans+=a[p].cnt;
            a[p].cnt=-1;
            p=a[p].fail;
        }
    }
    return ans;
}

//题意:输入n个串,是否存在一个串,其他的串都是它的子串,如果不存在输出no
```

```
int main()
{
    int T;
    scanf("%d",&T);
    while(T--)
    {
        scanf("%d",&n);
        num=0;
        clear(0);
        int maxl=-1;
        char maxs[L];
        for(int i=1;i<=n;i++)
        {
            scanf("%s",s);
            trie(s);
            int a=(int)strlen(s);
            if(a>maxl)
            {
                maxl=a;
                strcpy(maxs,s);
            }
        }
        buildAC();
        int ans=find(maxs);
        if(ans==n)
            printf("%s\n",maxs);
        else
            printf("No\n");
    }
    return 0;
}
```

后缀数组

```
#include <iostream>
#include <string>
#include <algorithm>
#include <cstring>

using namespace std;

const int MAX=200;

bool compare_sa(int i,int j);
void construct_sa(char *S, int *sa);
void construct_lcp(char *S, int *sa, int *lcp);

int n,k;
int Rank[MAX];
int tmp[MAX];
//sa[i]为后缀开始下标,i为顺序
int sa[MAX];
//lcp[i]表示 sa[i]和sa[i+1] 的公共前缀长度。
int lcp[MAX];
//以上这些都不需要memset

bool compare_sa(int i,int j)
{
    if(Rank[i]!=Rank[j])
        return Rank[i]<Rank[j];
    else
    {
        int ri= i+k<=n ? Rank[i+k] : -1;
        int rj= j+k<=n ? Rank[j+k] : -1;
        return ri<rj;
    }
}

void construct_sa(char *S, int *sa)
{
```

```
    n=(int)strlen(S);
    for(int i=0;i<=n;i++)
    {
        sa[i]=i;
        Rank[i]= i<n ? S[i] : -1;
    }
    for(k=1;k<=n;k*=2)
    {
        sort(sa,sa+n+1,compare_sa);
        tmp[sa[0]]=0;
        for(int i=1;i<=n;i++)
            tmp[sa[i]]=tmp[sa[i-1]]+(compare_sa(sa[i-1], sa[i])?1:0);
        for(int i=0;i<=n;i++)
            Rank[i]=tmp[i];
    }
}

void construct_lcp(char *S, int *sa, int *lcp)
{
    n=(int)strlen(S);
    for(int i=0;i<=n;i++)
        Rank[sa[i]]=i;
    int h=0;
    lcp[0]=0;
    for(int i=0;i<n;i++)
    {
        int j=sa[Rank[i]-1];
        if(h>0)
            h--;
        for(;j+h<n && i+h<n;h++)
            if(S[j+h]!=S[i+h])
                break;
        lcp[Rank[i]-1]=h;
    }
}
//后缀数组如何求不同子串个数
//len-sa[i]-lcp[i-1]累加
//    for(int i=1;i<=s.length();i++)
//          ans+=s.length()-sa[i]-lcp[i-1];


//std::ios::sync_with_stdio(false)
//T串是否能匹配S串
bool contain(string S,int *sa,string T)
{
    int a=0;
    int b=(int)S.length();
    while(b-a>1)
    {
        int c=(a+b)/2;
        if(S.compare(sa[c],T.length(),T)<0)
            a=c;
        else
            b=c;
    }
    return S.compare(sa[b],T.length(),T)==0;
}
```