暴力：

1.刷颜色：记录最后一次状态

```
#include <iostream>
#include <stdio.h>
#include <string.h>
using namespace std;
  int n,m,k;
  int ma[5005][5005];
  int r[5555];int c[5555];int a[5555];int b[5555];
int main()
{

  while(~scanf("%d%d%d",&n,&m,&k))
  {
      memset(ma,0,sizeof(ma));
      int o=0;int x=0;int y=0;
      for(int i=1;i<=k;i++)
      {
          scanf("%d%d%d",&o,&x,&y);
          if(o==1)
          {
              r[x]=y,a[x]=i;
          }
          else
          {
              c[x]=y,b[x]=i;
          }
      }
      for(int i=1;i<=n;i++)
      {
           for(int j=1;j<=m;j++)
                if(a[i]>b[j])
                    printf("%d ",r[i]);
                else printf("%d ",c[j]);
            puts("");
      }

  }
    return 0;
}
```

区间dp：Zoj 3537 Cake

给定n个点的坐标，先问这些点是否能组成一个凸包，如果是凸包，问用不相交的线来切这个凸包使得凸包只由三角形组成，根据costi, j = |xi + xj| * |yi + yj| % p算切线的费用，问最少的切割费用。

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <algorithm>
using namespace std;
#define MAX 1000
#define INF 1000000000
#define min(a,b) ((a)<(b)?(a):(b))
```

```
struct point{

    int x,y;
}p[MAX];
int cost[MAX][MAX],n,m;
int dp[MAX][MAX];


int abs(int x) {

    return x < 0 ? -x : x;
}
point save[400],temp[400];
int xmult(point p1,point p2,point p0){

    return (p1.x-p0.x)*(p2.y-p0.y)-(p2.x-p0.x)*(p1.y-p0.y);
}
bool cmp(const point& a,const point &b){

    if(a.y == b.y)return a.x < b.x;
    return a.y < b.y;
}
int Graham(point *p,int n) {

    int i;
    sort(p,p + n,cmp);
    save[0] = p[0];
    save[1] = p[1];
    int top = 1;
    for(i = 0;i < n; i++){

        while(top && xmult(save[top],p[i],save[top-1]) >= 0)top--;
        save[++top] = p[i];
    }


    int mid = top;
    for(i = n - 2; i >= 0; i--){

        while(top>mid&&xmult(save[top],p[i],save[top-1])>=0)top--;
        save[++top]=p[i];
    }
    return top;
}
int Count(point a,point b) {

    return (abs(a.x + b.x) * abs(a.y+b.y)) % m;
}


int main()
{
    int i,j,k,r,u;


    while (scanf("%d%d",&n,&m) != EOF) {
```

```
}		for (i = 0; i < n; ++i)
			scanf("%d%d",&p[i].x,&p[i].y);


		int tot = Graham(p,n); //求凸包
		if (tot < n) printf("I can't cut.\n");
		else {

			memset(cost,0,sizeof(cost));
			for (i = 0; i < n; ++i)
				for (j = i + 2; j < n; ++j)
					cost[i][j] = cost[j][i] = Count(save[i],save[j]);


			for (i = 0; i < n; ++i) {

				for (j = 0; j < n; ++j)
					dp[i][j] = INF;
				dp[i][(i+1)%n] = 0;
			}
			for (i = n - 3; i >= 0; --i)		//注意这三个for循环的顺序
				for (j = i + 2; j < n; ++j) //因为要保证在算dp[i][j]时dp[i][k]和dp[k][j]时已
经计算，所以i为逆序，j要升序
					for (k = i + 1; k <= j - 1; ++k)
						dp[i][j] = min(dp[i][j],dp[i][k]+dp[k][j]+cost[i][k]+cost[k]
[j]);


			printf("%d\n",dp[0][n-1]);
		}
	}
}
```

搜索：多校3误以为并查集那题

```
#include <stdio.h>
#include <iostream>
#include <cstring>
using namespace std;
int m[111][111];
int n;int k;
int dfs(int x)
{
   int sum=0;
   for(int i=1;i<=n;i++)
   {
      if(m[x][i])
      {sum++;
      sum+=dfs(i);}
   }
   return sum;
}
```

```
int main()
{

    while(~scanf("%d%d",&n,&k))
    {
        memset(m,0,sizeof(m));
        int ans=0;
        for(int i=1;i<=n-1;i++)
        {
            int a;int b;
            scanf("%d%d",&a,&b);
            m[a][b]=1;
        }
        for(int i=1;i<=n;i++)
        {
            if(dfs(i)==k) ans++;
        }
        cout<<ans<<endl;
    }
}
```

并查集：畅通工程，目标是使任何两个城镇之间可以交通，问最少需要建设多少条道路

```
//查询根节点
int find(int x) {
    return par[x] == x ? x : par[x] = find(par[x]);
}

//合并两个集合
void unite(int x, int y) {
    x = find(x);
    y = find(y);
    if(x == y) return;
    ans--;
    par[x] = y;
}

int main() {
    while(scanf("%d", &n), n) {
        scanf("%d", &m);
        ans = n - 1;
        INIT();
        for(int i = 0; i < m; i++) {
            int a, b;
            scanf("%d%d", &a, &b);
            unite(a, b);
        }
        printf("%d\n", ans);
    }
    return 0;
}
```

最小生成树：poj 1258 裸最小生成树

```
const int Max = 102;
const int inf = 0xfffffff;
```

```
int n, ans;
int map[Max][Max], dis[Max];   //  dis[i]表示顶点i与生成树之间的最短距离。



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



void prim()
{
    int i, j, now, min_node, min_edge;
    for(i = 1; i <= n; i ++)
        dis[i] = inf;
    now = 1;
    ans = 0;
    for(i = 1; i < n; i ++){
        dis[now] = -1;    //   将dis[]的值赋-1，表示已经加入生成树中。
        min_edge = inf;
        for(j = 1; j <= n; j ++)    //   更新每个顶点所对应的dis[]值。
            if(now != j && dis[j] >= 0){
                dis[j] = min(dis[j], map[now][j]);
                if(dis[j] < min_edge){
                    min_edge = dis[j];
                    min_node = j;
                }
            }
                        now = min_node;
                        ans += min_edge;
    }
}



int main(){
    int i, j;
    while(scanf("%d", &n) != EOF){
        for(i = 1; i <= n; i ++)
            for(j = 1; j <= n; j ++)
                scanf("%d", &map[i][j]);
                        prim();
                        printf("%d\n", ans);
    }
    return 0;
}
```

线段树：（该题没有被窝ac）

题目大意：求一个区间内不重复数字的和，例如1 1 1 3，区间[1,4]的和为4。

思路：如果采用在线算法，很难在nlogn的时间内处理，所以考虑离线算法。

首先我们把所有查询区间记录下来，然后按照区间的右值排序，接着从左到右把每一个数更新到线段树中，并记录它出现的位置。
如果一个数已经出现过，那么我们就把他上次出现的位置的值置为0，并更新它出现的位置。

因为我们的查询区间是按右值排序的，所以当前区间的左值要么和之前一样要么比之前的要大，因此把过去重复出现的数字置为0不会影响结果。
当更新到某个区间的右值时，我们就查询一次该区间的答案，并把答案记录到对应的地方。
最后把区间查询的答案按照输入顺序输出即可。

离线方法+线段树+离散化

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <vector>

#include <algorithm>
#include <iostream>
#include <queue>
#include <set>
#include <string>

using namespace std ;

#define MEM(a, v)        memset (a, v, sizeof (a))    // a for address, v for value
#define max(x, y)        ((x) > (y) ? (x) : (y))
#define min(x, y)        ((x) < (y) ? (x) : (y))

#define INF     (0x3f3f3f3f)
#define MAXN 30009
#define MAXQ 100010

#define L(x)    ((x)<<1)
#define R(x)    (((x)<<1)|1)

#define DB    /##/
typedef __int64          LL;

struct NODE {
        int x, y;
        LL  sum;
};

int              tcase, n, q, iDis;        // iDis 表示 iDiscrete
int              src[MAXN], ssrc[MAXN], idques[MAXQ], idseg[MAXN];
LL               sum[MAXN*4];

NODE  question[MAXQ];

bool cmp(const int &i, const int &j)
{
        return question[i].y < question[j].y;
}
```

```
int rank(int val)
{
	int mid, lw = 0, up = iDis;
	while (lw <= up)
	{
		mid = (lw + up) >> 1;
		if (ssrc[mid] == val)
			break;

		if (val < ssrc[mid])
			up = mid - 1;
		else
			lw = mid + 1;
	}
	return mid;
}

inline void build()
{
	MEM(sum, 0);
}

void update(int id, int lf, int rh, int pos, int f)
{
	if (lf == rh)
	{
		sum[id] = f ? src[pos] : 0;	// 不是 sum[lf] 或 sum[pos]
		return ;
	}

	int mid = (lf + rh) >> 1;
	if (pos <= mid)
		update(L(id), lf, mid, pos, f);
	else
		update(R(id), mid+1, rh, pos, f);

	sum[id] = sum[L(id)] + sum[R(id)];
}

LL query(int id, int lf, int rh, int s, int t)
{
	if (s == lf && t == rh)
		return sum[id];

	int mid = (lf + rh) >> 1;
	if (t <= mid)
		return query(L(id), lf, mid, s, t);
	else if (mid < s)
		return query(R(id), mid+1, rh, s, t);

	return query(L(id), lf, mid, s, mid) + query(R(id), mid+1, rh, mid+1, t);
}

int main()
```

```
{
        int i, j, k;
        while (scanf("%d", &tcase) != EOF)
        {
                while (tcase--)
                {
                        scanf("%d", &n);
                        for (i = 0; i < n; ++i)
                        {
                                scanf("%d", src+i);
                                ssrc[i] = src[i]; // ssrc 表示被排序过的src 即 sorted src
                        }
                        // 离散化
                        sort(ssrc, ssrc+n);
                        for (i = iDis = 1; i < n; ++i)
                        {
                                if (ssrc[iDis-1] != ssrc[i])
                                        ssrc[iDis++] = ssrc[i];
                        } // 离散化完

                        scanf("%d", &q);
                        for (i = 0; i < q; ++i)
                        {
                                scanf("%d %d", &question[i].x, &question[i].y);
                                --question[i].x; --question[i].y;
                                idques[i] = i;
                        }

                        // 对 question 的下标进行排序，相当于间接排序了 question 数组
                        sort(idques, idques+q, cmp);

                        MEM(idseg, -1);

                        // build
                        build();
                        for (i = j = 0; i < n; ++i)
                        {
                                k = rank(src[i]);
                                if (-1 != idseg[k])
                                        // 到上一次 src[i] 出现的位置（idseg[k]），将 src[i] 删除.
                                        update(1, 0, n-1, idseg[k], 0);
                                idseg[k] = i;     // 记录此次 src[i] 出现的位置
                                update(1, 0, n-1, i, 1);// 将 src[i] 加入线段树中

                                // 对排序过的查询区间进行处理
                                while (j < q && question[idques[j]].y == i)
                                {
                                        question[idques[j]].sum = query(1, 0, n-1,
                                                question[idques[j]].x, i);
                                        ++j;
                                }
                        }

                        // 按原顺序输出结果
```

```
                    for (i = 0; i < q; ++i)
                            printf ("%I64d\n", question[i].sum);
            }
        }
        return 0;
}
```

裸模版：

```
#define LL(x) ((x)<<1)
#define RR(x)  ((x)<<1|1)
#define FF(i,n) for(int i = 0 ; i < n ; i ++)
struct Seg_Tree{
   int left,right,num;
   int calmid() {
      return (left+right)/2;
   }
}tt[150000];
int num[50001];
int build(int left,int right,int idx) {
   tt[idx].left = left;
   tt[idx].right = right;
   if(left == right) {
      return tt[idx].num = num[left];
   }
   int mid = (left + right)/2;
   return tt[idx].num = build(left,mid,LL(idx)) + build(mid+1,right,RR(idx));
}

void update(int id,int x,int idx) {
   tt[idx].num += x;
   if(tt[idx].left == tt[idx].right) {
      return ;
   }
   int mid = tt[idx].calmid();
   if(id <= mid) {
      update(id,x,LL(idx));
   } else {
      update(id,x,RR(idx));
   }
}

int query(int left,int right,int idx) {
   if(left == tt[idx].left && right == tt[idx].right) {
      return tt[idx].num;
   }
   int mid = tt[idx].calmid();
   if(right <= mid) {
      return query(left,right,LL(idx));
   } else if(mid < left) {
      return query(left,right,RR(idx));
   } else {
      return query(left,mid,LL(idx)) + query(mid+1,right,RR(idx));
   }
}
```

```
int main() {
    int T;
    scanf("%d",T);
    FF(cas,T) {
        int n;
        scanf("%d",&n);
        FOR(i,1,1+n) {
            scanf("%d",num[i]);
        }
        build(1,n,1);
        printf("Case %d:\n",cas+1);
        char com[9];
        while(scanf("%s",com)) {
            if(strcmp(com,"End") == 0)    break;
            int a,b;
            scanf("%d%d",&a,&b);
            switch(com[0]) {
                case 'Q':
                    printf("%d\n",query(a,b,1));
                    break;
                case 'A':
                    update(a,b,1);
                    break;
                case 'S':
                    update(a,-b,1);
                    break;
            }
        }
    }
    return 0;
}
```

数论：长春Unknown Treasure
题意

给你n,m,num

然后给你num个数，k1,k2....,knum

然后让你求C(n,m)%(k1*k2*....*knum)

题解：

首先，C(n,m)%k我们是会求的，大概这部分子问题是一个很经典的题目。

假设你会求了，那么我们就可以由此得到num个答案，是%k1,k2,k3....knum后得到的值

然后我们就可以看出就是类似韩信点兵一样的题，三个人一组剩了2个，五个人一组剩了2个这种……

这时候，就用中国剩余定理处理处理就好了

注意ll*ll会爆，所以得手写个快速乘法

```
#include <cstdio>
#include <cmath>
#include <cstring>
#include <ctime>
#include <iostream>
#include <algorithm>
#include <set>
#include <vector>
#include <sstream>
#include <queue>
#include <typeinfo>
#include <fstream>
#include <map>
#include <stack>
typedef long long ll;
using namespace std;
//***********************************************************************************

void extend_gcd(ll a,ll &x,ll b,ll &y)
{
    if(b==0)
    {
        x=1,y=0;
        return;
    }
    ll x1,y1;
    extend_gcd(b,x1,a%b,y1);
    x=y1;
    y=x1-(a/b)*y1;
}
ll inv(ll a,ll m)
{
    ll t1,t2;
    extend_gcd(a,t1,m,t2);
    return (t1%m+m)%m;
}
ll qpow(ll x,ll y,ll m)
{
    if(!y)return 1;
    ll ans = qpow(x,y>>1,m);
    ans = ans*ans%m;
    if(y&1)ans = ans*x%m;
    return ans;
}

ll nump(ll x,ll p)
{
    ll ans = 0;
    while(x)ans+=x/p,x/=p;
    return ans;
}
ll fac(ll n,ll p,ll pk)
{
```

```
    if(n==0)return 1;
    ll ans = 1;
    for(ll i=1;i<=pk;i++)
    {
        if(i%p==0)continue;
        ans = ans*i%pk;
    }
    ans = qpow(ans,n/pk,pk);
    ll to = n%pk;
    for(ll i =1;i<=to;i++)
    {
        if(i%p==0)continue;
        ans = ans*i%pk;
    }
    return fac(n/p,p,pk)*ans%pk;
}
ll cal(ll n,ll m,ll p ,ll pi,ll pk)
{
    ll a = fac(n,pi,pk),b=fac(m,pi,pk),c=fac(n-m,pi,pk);
    ll d = nump(n,pi)-nump(m,pi)-nump(n-m,pi);
    ll ans = a%pk * inv(b,pk)%pk * inv(c,pk)%pk*qpow(pi,d,pk)%pk;
    return ans*(p/pk)%p*inv(p/pk,pk)%p;
}
ll mCnmodp(ll n,ll m,ll p)
{
    ll ans = 0;
    ll x = p;
    for(ll i =2;i*i<=x&&x>1;i++)
    {
        ll k=0,pk=1;
        while(x%i==0)
        {
            x/=i;
            k++;
            pk*=i;
        }
        if(k>0)
            ans=(ans+cal(n,m,p,i,pk))%p;
    }
    if(x>1)ans=(ans+cal(n,m,p,x,x))%p;
    return ans;
}
ll qtpow(ll x,ll y,ll M)
{
    ll ret=0LL;
    for(x%=M;y;y>>=1LL)
    {
        if(y&1LL)
        {
            ret+=x;
            ret%=M;
            if(ret<0) ret+=M;
        }
        x+=x;
        x%=M;
        if(x<0) x+=M;
```

```
    }
    return ret;
}
void solve(ll r[],ll s[],int t)
{
    ll M=1LL,ans=0LL;
    ll p[20],q[20],e[20];
    for(int i=0;i<t;i++)
        M*=r[i];
    for(int i=0;i<t;i++)
    {
        ll tmp=M/r[i],tt;
        extend_gcd(tmp,p[i],r[i],q[i]);
        p[i]%=M;
        if(p[i]<0) p[i]+=M;
        e[i]=qtpow(tmp,p[i],M);
        tt=qtpow(e[i],s[i],M);
        ans=(ans+tt)%M;
        if(ans<0) ans+=M;
    }
    printf("%I64d\n",ans);
}

ll CCC[20],DDD[20];
int main()
{
    int t;
    scanf("%d",&t);
    int num = 0;
    ll n,m,p;
    while(t--)
    {
        memset(CCC,0,sizeof(CCC));
        memset(DDD,0,sizeof(DDD));
        scanf("%I64d %I64d %d",&n,&m,&num);
        for(int i=0;i<num;i++)
        {
            scanf("%I64d",&CCC[i]);
            DDD[i]=mCnmodp(n,m,CCC[i]);
        }
        solve(CCC,DDD,num);
    }
    return 0;
}
```

1.欧几里得 求最大公约数,最小公倍数

(1)递归的写法:int gcd(int a,int b) {return b?gcd(b,a%b):a;}

(2)辗转相除法:

```
int gcd(int a,int b)
{
 if(a<b)    return gcd(b,a);
 int r;
```

```
 while(b) {r=a%b;a=b;b=r;}
 return a;
}
```

(3)stein+欧几里得 快速求解大数的最大公约数

```
i64 stein(i64 a,i64 b)
{
 if(a<b)   return stein(b,a);
 if(b==0) return a;
 if((a&1)==0&&(b&1)==0) return 2*stein(a>>1,b>>1);//a and b are even
 if((a&1)==0) return  stein(a>>1,b);   // only a is  even
 if((b&1)==0) return  stein(a,b>>1);   // only b is  even
 return    stein((a+b)>>1,(a-b)>>1);   // a and b are odd
}
```

最小公倍数: int lcm(int a,int b) {return a/gcd(a,b)*b;}

2.扩展欧几里得 求ax=b (mod m) ax+my=b 如果r=gcd(a,m)且b%r==0,则同余方程有解,其最小解为x*(b/r);

ax+by=c 如r=gcd(a,b),则存在x,y,使xa+yb=r;当x+=b,y-=a后仍然成立

因为xa+yb+ab-ab=r;==>(x+b)a+(y-a)b=r

```
int exgcd(int a,int b,int &x,int &y)
{
 if(b==0) {x=1;y=0;return a;}
 int r=exgcd(b,a%b,y,x);
 y-=x*(a/b);
 return r;
}
```

3.素数判定

(1)试除法:

```
bool  isprime(int n)
{
 int i;
 for(i=2;i<=(int)sqrt(n*1.0);i++)
     if(n%i==0)    return  false;
 return  true;
}
bool isprime(int n)
{
 if(n==2) return true;
 if(n==1||(n&1)==0)  return false;
 for(int i=3;i*i<=n;i+=2)   if(n%i==0) return fals;
 return true;
}
```

(2)miller-rabin 算法

```
bool witness(i64 a,i64 n)
{
  i64 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;
}
bool miller_rabin(i64 n)
{
  if(n==2)   return 1;
  if(n==1||(n&1)==0) return 0;
  i64 j,a;
  for(j=0;j<50;j++)
  {
   a=rand()*(n-2)/RAND_MAX+1;
   if(witness(a,n)) return 0;
  }
  return 1;
}
```

另一种写法，更好理解

```
bool witness(i64 a,i64 n)
{
 int i,j=0;
 i64 m=n-1,x,y;
 while(m%2==0)
 {
  m>>=1;
  j++;
 }
 x=pow(a,m,n);///快速幂取模
 for(i=1;i<=j;i++)
 {
  y=pow(x,2,n);
  if(y==1&&x!=1&&x!=n-1) return true;
  x=y;
 }
return y==1?false:true;
}
bool miller_rabin(i64 n)
{
 if(n==2) return true;
 if(n==1||n%2==0) return false;
 for(int i=1;i<=10;i++)
 {
  i64 a=rand()%(n-1)+1;
  if(witness(a,n)) return false;
 }
 return true;
}
```

4.素数筛法　//前17个素数 prime[18]={17,2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59}

```
bool f[100002];//保存判断是否是素数的结果，p[i]=1 是素数，p[i]=0 则不是素数
int prime[78499];//保存素数prime[0]为素数的个数
void PRIME(int M)
{
 int i,i2,k;
 for(i=0;i<=M;i+=2) f[i]=0;
```

```
 for(i=1;i<=M;i+=2) f[i]=1;
 f[1]=0;  f[2]=1;
 for(i=3;i<=(int)sqrt(1.0*M);i+=2)
 if(p[i])
 {
  i2=i+i;  k=i*i;
  while(k<=M) {f[k]=0; k+=i2;}
 }
 prime[1]=2; k=1;
 for(i=3;i<=M;i+2) if(f[i]) prime[++k]=i;
 prime[0]=k;
}
```

(2)

```
void PRIME(int M)
{
 int i,j,k;
 prime[1]=2; prime[2]=3;
 for(i=5;i<=M;i+=2)
 {
  for(j=1;prime[j]*prime[j]<=i;j++)
  if(i%prime[j]==0) goto loop;
  prime[++k]=i;
  loop:;
 }
 prime[0]=k;
}
```

5.整数分解

(1)

```
void split(int n,int *p,int *t)
{
 int i,s,top=0;
 for(i=1;i<=prime[0];i++)
 {
  s=0;
  while(n%prime[i]==0)  {s++;n/=prime[i];}
  if(s)  {p[++top]=prime[i];t[top]=s;}
  if(n==1) break;
  if(n<prime[i]*prime[i]) {p[++top]=n;t[top]=1;n=1;break;}
 }
 p[0]=t[0]=top;
}
```

(2)分解1-100000的因子,且由prime[n][]保存n的素因子(prime[n][0]为质因子的个数):

```
void split(int n)//p[]为素数表
{
 int i,x=n;
 prime[n][0]=0;
 for(i=1;i<=p[0];i++) ////if(x%p[i]==0)严重坑爹的bug
 {
  prime[n][++prime[n][0]]=p[i];
  while(x%p[i]==0) x/=p[i];
  if(x==1) break;
 }
 if(x>1) prime[n][++prime[n][0]]=x;
```

```
}
```

(3)Pollard-pho大数分解

```
i64 Pollard(i64 n,int c)
{
 i64 i=1,k=2,x=rand()%n,y=x,d;
 srand(time(NULL));
 while(true)
 {
  i++;
  x=(mod_mult(x,x,n)+c)%n;
  d=gcd(y-x,n);
  if(d>1&&d<n) return d;
  if(y==x) return n;
  if(i==k)  { y=x; k<<=1; }
 }
}
```

6.求因子和与因子个数（包含1和本身）

因子和s是积性函数,即:gcd(a,b)=1==>s(a*b)=s(a)*s(b);
如果p是素数==>s(p^X)=1+p+p^2+...+p^X=(p^(X+1)-1)/(p-1);s(p^2x)=1+p+p^2+...+p^2x=(p^(2x+1)-1)/(p-1);
求因子和:
(1)ans=1+n;

```
for(i=2;i<=n/2;i++)
if(n%i==0)
{
 if(n/i>i) ans+=i+n/i;
 else  if(n/i==i) ans+=i;
 else  break;
}
```

(2)另一种递推的写法:

```
for(i=1;i<=lmax;i++)   num[i]=1+i;
for(i=2;i<=lmax/2;i++)
 for(j=i<<1;j<=lmax;j+=i)  num[j]+=i;
```

求因子个数:
n=p1^t1*p2^t2*p3^t3***pk^tk; 因子个数为:(t1+1)(t2+1)***(tk+1)

```
for(ret=i=1;i<=prime[0]&&prime[i]<=(int)sqrt(1.0*n);i++)
if(n%prime[i]==0)
{
 k=0;
 while(n%prime[i]==0) {k++,n/=prime[i];}
 ret*=k+1;
}
if(n>1) ret*=2;
```

当求n^2的因子个数的时候:n^2=p1^(2*t1)*p2^(2*t2)***pk^(2*tk); 因子个数为:（2*t1+1)(2*t2+1)***(2*tk+1)

```
for(ret=i=1;i<=prime[0]&&prime[i]<=(int)sqrt(1.0*n);i++)
if(n%prime[i]==0)
{
 k=0;
```

```
 while(n%prime[i]==0) {k++,n/=prime[i];}
 ret*=2*k+1;
}
if(n>1) ret*=3;
```

快速求出一个比较大的区间内的所有因子和:

```
const int lmax=50005;//求出[1,50005]区间内每一个数的因子和(不包括本身),并用facsum[]数组保存
i64  facsum[lmax];
for(i=0;i<=lmax;i++)   facsum[i]=1;
 for(i=2;i*i<=lmax;i++)
 {
  for(j=i+1;j*i<=lmax;j++)  facsum[i*j]+=i+j;
  facsum[i*i]+=i;
 }
```

7.欧拉函数

(1)单独求欧拉函数

```
int eular(int n)
{
 int ret=1,i;
 for(i=2;i*i<=n;i++)
 if(n%i==0)
 {
  n/=i;  ret*=i-1;
  while(n%i==0) {n/=i;ret*=i;}
 }
 if(n>1) ret*=n-1;
 return  ret;
}
int euler(int x)
{
 int i, res=x,tmp=(int)sqrt(x*1.0)+1;
 for(i=2;i<tmp;i++)
 if(x%i==0)
 {
  res=res/i*(i-1);
  while(x%i== 0) x/=i;
 }
 if(x>1) res=res/x*(x-1);
 return res;
}
int eular(int n)
{
 int ret=n,i;
 for(i=2;i*i<=n;i++)
 if(n%i==0)
 {
  ret=ret/i*(i-1);
  while(n%i==0) n/=i;
 }
 if(n>1) ret=ret/n*(n-1);
 return ret;
}
```

先素数筛法在用欧拉函数(在此仅写其中的一个)

```
void eular(int n)
{
 int ret=n,i;
 for(i=1;i<=prime[0]&&prime[i]<=(int)sqrt(1.0*n);i++)
 if(n%prime[i]==0)
 {
  ret=ret/prime[i]*(prime[i]-1);
  while(n%prime[i]==0) n/=prime[i];
 }
 if(n>1) ret=ret/n*(n-1);
 return ret;
}
```

(2)递推求欧拉函数

```
const int  lmax=300000;
int PHI(int lmax)
{
 int i,j;
 for(i=1;i<=lmax;i++) phi[i]=i&1?i:i/2;
 for(i=3;i<=lmax;i+=2)
 if(phi[i])
 for(j=i;j<=lmax;j+=i) phi[j]=phi[j]/i*(i-1);
}
```

(3)同时求出欧拉值和素数

```
int prime[lmax][25],num[lmax],eular[lmax];//prime[n][i]表示n的第i+1个素数因子,num[n]表示n的因子个数,eular[n]表示n的欧拉值
void eular_prime()//每个数的欧拉函数值及筛选法得到数的素因子num[i]为i的因子个数
{
 eular[1]=1;
 for(int i=2;i<lmax;i++)
 {
  if(eular[i]==0)
  for(int j=i;j<lmax;j+=i)
  {
   if(eular[j]==0)  eular[j]=j;
   eular[j]=eular[j]*(i-1)/i;
   prime[j][num[j]++]=i;
  }
  //eular[i]+=eular[i-1];//进行累加（法里数列长度）
 }
}
void eular_prime()
{
 int i,j;
 eular[1]=1;
 for(i=2;i<=lmax;i++)
 if(eular[i]==0)
 for(j=i;j<=lmax;j+=i)
 {
  if(eular[j]==0) eular[j]=j;
  eular[j]=eular[j]/i*(i-1);
  prime[j][++prime[j][0]]=i;
```

```
 }
}
```
欧拉定理的一个重要应用:A^x mod m=A^(x%phi(m)+phi(m)) mod m（当x>=phi(m)时)

8.求逆元ax=1 (mod m) x是a的逆元

(1)用扩展欧几里得求

```
int Inv(int a,int m)
{
  int r,x,y;
  r=exgcd(a,m,x,y);
  if(r==1) return (x%m+m)%m;
  return -1;
}
```
(2)用快速幂取模求a*a^(p-2)=a^(p-1)=1 (mod p) p必须为素数,a的逆元是a^(p-2);

```
int pow(int a,int n)//a^n%p (n=p-2)
{//这里的做法会让a的值变化,可令t=a;用t代替a计算
 int r=1;
 while(n)
 {
  if(n&1) r=r*a%p;
  a=a*a%p;
  n>>=1;
 }
 return r;
}
```

9.快速模乘 a*b%p

```
int mul(int a,int b)
{
 int r=0;
 while(b)
 {
  if(b&1) r=(r+a)%p;
  a=(a<<1)%p;
  b>>=1;
 }
 return r;
}
```

10.求解模线性方程组(中国剩余定理)

  x=a1 mod m1
  x=a2 mod m2
  ......
  x=an mod mn 其中,a[],m[]已知,m[i]>0且m[i]与m[j]互质,求x.

设m1,m2,...,mn是两两互素的正数,则对任意的整数a1,a2,...,an,同余方程组

其解为:X=((M_1*M1*a1)+(M_2*M2*a2)+...+(M_n*Mn*an)) mod m;

其中m=m1*m2*...*mn; Mi=m/mi; M_i是Mi的逆元

```
int china(int *a,int *m,int n)
{
   int M=1,ans=0,mi,i,x,y;
   for(i=0;i<n;i++) M*=m[i];
   for(i=0;i<n;i++)
   {
```

```
        mi=M/m[i];
        exgcd(m[i],mi,x,y);
        ans=(ans+mi*y*a[i])%M;
    }
    return (ans%M+M)%M;
}
```

矩阵快速幂

```
#include<iostream>
#include<stdio.h>
#include<string.h>
using namespace std;
struct node{
    int p[25][25];
};
struct node suan(struct node a,struct node b,int n)//连个矩阵相乘
{
    int i,j,k;
    struct node c;
    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            c.p[i][j]=0;
            for(k=0;k<n;k++)
                c.p[i][j]=(c.p[i][j]+a.p[i][k]*b.p[k][j])%1000;
        }
    }
    return c;
}
struct node haha(struct node a,int n,int k)
{
    int i,j;
    struct node b;
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            if(i==j)
                b.p[i][j]=1;
            else
                b.p[i][j]=0;
    while(k)
    {
        if(k%2==1)
            b=suan(b,a,n);
        k=k/2;
        a=suan(a,a,n);
    }
    return b;
}
int main()
{
    int n,m,i,x1,x2,T;
    int S,E,k;
    struct node a,b;
    while(scanf("%d%d",&n,&m)!=EOF&&(n!=0||m!=0))
```

```
    {
        memset(a.p,0,sizeof(a.p));
        for(i=0;i<m;i++)
        {
            scanf("%d%d",&x1,&x2);
            a.p[x1][x2]=1;
        }
        scanf("%d",&T);
        while(T--)
        {
            scanf("%d%d%d",&S,&E,&k);
            b=haha(a,n,k);
            while(b.p[S][E]<0)//以后碰到取模的情况记得添加
                b.p[S][E]+=1000;
            printf("%d\n",b.p[S][E]);
        }
    }
    return 0;
}
```

矩阵转置

```
#include <iostream>
using namespace std;
const int N = 3;

int main()
{
// freopen("1.txt", "r", stdin);
int n, i, j;
int temp;
int a[N][N];
cin>>n;
while(n--)
{
for (i = 0; i < N*N; i++)
{
cin>>(*a)[i];
}
for (i = 0; i < N; i++)
{
for (j = 0; j < i; j++)
{
temp = a[i][j];
a[i][j] = a[j][i];
a[j][i] = temp;
}
}
for (i = 0; i < N*N; i++)
{
cout<<(*a)[i]<<" ";
if ((i+1) % 3 == 0)
cout<<endl;
}
cout<<endl;
}
```

```
return 0;
}
```

计算几何:

```
/**
 * 二维ACM计算几何模板
 * 注意变量类型更改和EPS
 * #include <cmath>
 * #include <cstdio>
 * By OWenT
 */

const double eps = 1e-8;
const double pi = std::acos(-1.0);
//点
class point
{
public:
    double x, y;
    point(){};
    point(double x, double y):x(x),y(y){};

    static int xmult(const point &ps, const point &pe, const point &po)
    {
        return (ps.x - po.x) * (pe.y - po.y) - (pe.x - po.x) * (ps.y - po.y);
    }

    //相对原点的差乘结果，参数：点[_Off]
    //即由原点和这两个点组成的平行四边形面积
    double operator *(const point &_Off) const
    {
        return x * _Off.y - y * _Off.x;
    }
    //相对偏移
    point operator - (const point &_Off) const
    {
        return point(x - _Off.x, y - _Off.y);
    }
    //点位置相同(double类型)
    bool operator == (const point &_Off) const
    {
        return std::fabs(_Off.x - x) < eps && std::fabs(_Off.y - y) < eps;
    }
    //点位置不同(double类型)
    bool operator != (const point &_Off) const
    {
        return ((*this) == _Off) == false;
    }
    //两点间距离的平方
    double dis2(const point &_Off) const
    {
        return (x - _Off.x) * (x - _Off.x) + (y - _Off.y) * (y - _Off.y);
    }
```

```
    //两点间距离
    double dis(const point &_Off) const
    {
        return std::sqrt((x - _Off.x) * (x - _Off.x) + (y - _Off.y) * (y - _Off.y));
    }
};

//两点表示的向量
class pVector
{
public:
    point s, e;//两点表示，起点[s]，终点[e]
    double a, b, c;//一般式,ax+by+c=0

    pVector(){}
    pVector(const point &s, const point &e):s(s),e(e){}

    //向量与点的叉乘,参数：点[_Off]
    //[点相对向量位置判断]
    double operator *(const point &_Off) const
    {
        return (_Off.y - s.y) * (e.x - s.x) - (_Off.x - s.x) * (e.y - s.y);
    }
    //向量与向量的叉乘,参数：向量[_Off]
    double operator *(const pVector &_Off) const
    {
        return (e.x - s.x) * (_Off.e.y - _Off.s.y) - (e.y - s.y) * (_Off.e.x - _Off.s.x);
    }
    //从两点表示转换为一般表示
    bool pton()
    {
        a = s.y - e.y;
        b = e.x - s.x;
        c = s.x * e.y - s.y * e.x;
        return true;
    }

    //-----------点和直线（向量）-----------
    //点在向量左边（右边的小于号改成大于号即可,在对应直线上则加上=号）
    //参数：点[_Off],向量[_Ori]
    friend bool operator<(const point &_Off, const pVector &_Ori)
    {
        return (_Ori.e.y - _Ori.s.y) * (_Off.x - _Ori.s.x)
            < (_Off.y - _Ori.s.y) * (_Ori.e.x - _Ori.s.x);
    }

    //点在直线上,参数：点[_Off]
    bool lhas(const point &_Off) const
    {
        return std::fabs((*this) * _Off) < eps;
    }
    //点在线段上,参数：点[_Off]
    bool shas(const point &_Off) const
```

```
{
    return lhas(_Off)
        && _Off.x - std::min(s.x, e.x) > -eps && _Off.x - std::max(s.x, e.x) < eps
        && _Off.y - std::min(s.y, e.y) > -eps && _Off.y - std::max(s.y, e.y) < eps;
}

//点到直线/线段的距离
//参数： 点[_Off], 是否是线段[isSegment](默认为直线)
double dis(const point &_Off, bool isSegment = false)
{
    //化为一般式
    pton();

    //到直线垂足的距离
    double td = (a * _Off.x + b * _Off.y + c) / sqrt(a * a + b * b);

    //如果是线段判断垂足
    if(isSegment)
    {
        double xp = (b * b * _Off.x - a * b * _Off.y - a * c) / ( a * a + b * b);
        double yp = (-a * b * _Off.x + a * a * _Off.y - b * c) / (a * a + b * b);
        double xb = std::max(s.x, e.x);
        double yb = std::max(s.y, e.y);
        double xs = s.x + e.x - xb;
        double ys = s.y + e.y - yb;
        if(xp > xb + eps || xp < xs - eps || yp > yb + eps || yp < ys - eps)
            td = std::min(_Off.dis(s), _Off.dis(e));
    }

    return fabs(td);
}

//关于直线对称的点
point mirror(const point &_Off) const
{
    //注意先转为一般式
    point ret;
    double d = a * a + b * b;
    ret.x = (b * b * _Off.x - a * a * _Off.x - 2 * a * b * _Off.y - 2 * a * c) / d;
    ret.y = (a * a * _Off.y - b * b * _Off.y - 2 * a * b * _Off.x - 2 * b * c) / d;
    return ret;
}
//计算两点的中垂线
static pVector ppline(const point &_a, const point &_b)
{
    pVector ret;
    ret.s.x = (_a.x + _b.x) / 2;
    ret.s.y = (_a.y + _b.y) / 2;
    //一般式
    ret.a = _b.x - _a.x;
    ret.b = _b.y - _a.y;
    ret.c = (_a.y - _b.y) * ret.s.y + (_a.x - _b.x) * ret.s.x;
    //两点式
```

```
    if(std::fabs(ret.a) > eps)
    {
        ret.e.y = 0.0;
        ret.e.x = - ret.c / ret.a;
        if(ret.e == ret. s)
        {
            ret.e.y = 1e10;
            ret.e.x = - (ret.c - ret.b * ret.e.y) / ret.a;
        }
    }
    else
    {
        ret.e.x = 0.0;
        ret.e.y = - ret.c / ret.b;
        if(ret.e == ret. s)
        {
            ret.e.x = 1e10;
            ret.e.y = - (ret.c - ret.a * ret.e.x) / ret.b;
        }
    }
    return ret;
}

//------------直线和直线（向量）-------------
//直线重合,参数：直线向量[_Off]
bool equal(const pVector &_Off) const
{
    return lhas(_Off.e) && lhas(_Off.s);
}
//直线平行，参数：直线向量[_Off]
bool parallel(const pVector &_Off) const
{
    return std::fabs((*this) * _Off) < eps;
}
//两直线交点，参数：目标直线[_Off]
point crossLPt(pVector _Off)
{
    //注意先判断平行和重合
    point ret = s;
    double t = ((s.x - _Off.s.x) * (_Off.s.y - _Off.e.y) - (s.y - _Off.s.y) * (_Off.s.x - _Off.e.x))
            / ((s.x - e.x) * (_Off.s.y - _Off.e.y) - (s.y - e.y) * (_Off.s.x - _Off.e.x));
    ret.x += (e.x - s.x) * t;
    ret.y += (e.y - s.y) * t;
    return ret;
}

//------------线段和直线（向量）----------
//线段和直线交
//参数：线段[_Off]
bool crossSL(const pVector &_Off) const
{
    double rs = (*this) * _Off.s;
    double re = (*this) * _Off.e;
    return rs * re < eps;
```

```
    }

    //------------线段和线段（向量）----------
    //判断线段是否相交(注意添加eps)，参数：线段[_Off]
    bool isCrossSS(const pVector &_Off) const
    {
        //1.快速排斥试验判断以两条线段为对角线的两个矩形是否相交
        //2.跨立试验（等于0时端点重合）
        return (
            (std::max(s.x, e.x) >= std::min(_Off.s.x, _Off.e.x)) &&
            (std::max(_Off.s.x, _Off.e.x) >= std::min(s.x, e.x)) &&
            (std::max(s.y, e.y) >= std::min(_Off.s.y, _Off.e.y)) &&
            (std::max(_Off.s.y, _Off.e.y) >= std::min(s.y, e.y)) &&
            ((pVector(_Off.s, s) * _Off) * (_Off * pVector(_Off.s, e)) >= 0.0) &&
            ((pVector(s, _Off.s) * (*this)) * ((*this) * pVector(s, _Off.e)) >= 0.0)
            );
    }
};

class polygon
{
public:
    const static long maxpn = 100;
    point pt[maxpn];//点（顺时针或逆时针）
    long n;//点的个数

    point& operator[](int _p)
    {
        return pt[_p];
    }

    //求多边形面积，多边形内点必须顺时针或逆时针
    double area() const
    {
        double ans = 0.0;
        int i;
        for(i = 0; i < n; i ++)
        {
            int nt = (i + 1) % n;
            ans += pt[i].x * pt[nt].y - pt[nt].x * pt[i].y;
        }
        return std::fabs(ans / 2.0);
    }
    //求多边形重心，多边形内点必须顺时针或逆时针
    point gravity() const
    {
        point ans;
        ans.x = ans.y = 0.0;
        int i;
        double area = 0.0;
        for(i = 0; i < n; i ++)
        {
            int nt = (i + 1) % n;
            double tp = pt[i].x * pt[nt].y - pt[nt].x * pt[i].y;
```

```
        area += tp;
        ans.x += tp * (pt[i].x + pt[nt].x);
        ans.y += tp * (pt[i].y + pt[nt].y);
    }
    ans.x /= 3 * area;
    ans.y /= 3 * area;
    return ans;
}
//判断点在凸多边形内，参数：点[_Off]
bool chas(const point &_Off) const
{
    double tp = 0, np;
    int i;
    for(i = 0; i < n; i ++)
    {
        np = pVector(pt[i], pt[(i + 1) % n]) * _Off;
        if(tp * np < -eps)
            return false;
        tp = (std::fabs(np) > eps)?np: tp;
    }
    return true;
}
//判断点是否在任意多边形内[射线法]，O(n)
bool ahas(const point &_Off) const
{
    int ret = 0;
    double infv = 1e-10;//坐标系最大范围
    pVector l = pVector(_Off, point( -infv ,_Off.y));
    for(int i = 0; i < n; i ++)
    {
        pVector ln = pVector(pt[i], pt[(i + 1) % n]);
        if(fabs(ln.s.y - ln.e.y) > eps)
        {
            point tp = (ln.s.y > ln.e.y)? ln.s: ln.e;
            if(fabs(tp.y - _Off.y) < eps && tp.x < _Off.x + eps)
                ret ++;
        }
        else if(ln.isCrossSS(l))
            ret ++;
    }
    return (ret % 2 == 1);
}
//凸多边形被直线分割,参数：直线[_Off]
polygon split(pVector _Off)
{
    //注意确保多边形能被分割
    polygon ret;
    point spt[2];
    double tp = 0.0, np;
    bool flag = true;
    int i, pn = 0, spn = 0;
    for(i = 0; i < n; i ++)
    {
        if(flag)
```

```
            pt[pn ++] = pt[i];
        else
            ret.pt[ret.n ++] = pt[i];
        np = _Off * pt[(i + 1) % n];
        if(tp * np < -eps)
        {
            flag = !flag;
            spt[spn ++] = _Off.crossLPt(pVector(pt[i], pt[(i + 1) % n]));
        }
        tp = (std::fabs(np) > eps)?np: tp;
    }
    ret.pt[ret.n ++] = spt[0];
    ret.pt[ret.n ++] = spt[1];
    n = pn;
    return ret;
}

//------------凸包-------------
//Graham扫描法，复杂度O(nlg(n)),结果为逆时针
//#include <algorithm>
static bool graham_cmp(const point &l, const point &r)//凸包排序函数
{
    return l.y < r.y || (l.y == r.y && l.x < r.x);
}
polygon& graham(point _p[], int _n)
{
    int i, len;
    std::sort(_p, _p + _n, polygon::graham_cmp);
    n = 1;
    pt[0] = _p[0], pt[1] = _p[1];
    for(i = 2; i < _n; i ++)
    {
        while(n && point::xmult(_p[i], pt[n], pt[n - 1]) >= 0)
            n --;
        pt[++ n] = _p[i];
    }
    len = n;
    pt[++ n] = _p[_n - 2];
    for(i = _n - 3; i >= 0; i --)
    {
        while(n != len && point::xmult(_p[i], pt[n], pt[n - 1]) >= 0)
            n --;
        pt[++ n] = _p[i];
    }
    return (*this);
}

//凸包旋转卡壳(注意点必须顺时针或逆时针排列)
//返回值凸包直径的平方（最远两点距离的平方）
double rotating_calipers()
{
    int i = 1;
    double ret = 0.0;
    pt[n] = pt[0];
```

```
        for(int j = 0; j < n; j ++)
        {
            while(fabs(point::xmult(pt[j], pt[j + 1], pt[i + 1])) > fabs(point::xmult(pt[j], pt[j + 1], pt[i])) +
eps)
                i = (i + 1) % n;
            //pt[i]和pt[j],pt[i + 1]和pt[j + 1]可能是对踵点
            ret = std::max(ret, std::max(pt[i].dis(pt[j]), pt[i + 1].dis(pt[j + 1])));
        }
        return ret;
    }

    //凸包旋转卡壳(注意点必须逆时针排列)
    //返回值两凸包的最短距离
    double rotating_calipers(polygon &_Off)
    {
        int i = 0;
        double ret = 1e10;//inf
        pt[n] = pt[0];
        _Off.pt[_Off.n] = _Off.pt[0];
        //注意凸包必须逆时针排列且pt[0]是左下角点的位置
        while(_Off.pt[i + 1].y > _Off.pt[i].y)
            i = (i + 1) % _Off.n;
        for(int j = 0; j < n; j ++)
        {
            double tp;
            //逆时针时为 >,顺时针则相反
            while((tp = point::xmult(pt[j], pt[j + 1], _Off.pt[i + 1]) - point::xmult( pt[j], pt[j + 1], _Off.pt[i])) >
eps)
                i = (i + 1) % _Off.n;
            //(pt[i],pt[i+1])和(_Off.pt[j],_Off.pt[j + 1])可能是最近线段
            ret = std::min(ret, pVector(pt[j], pt[j + 1]).dis(_Off.pt[i], true));
            ret = std::min(ret, pVector(_Off.pt[i], _Off.pt[i + 1]).dis(pt[j + 1], true));
            if(tp > -eps)//如果不考虑TLE问题最好不要加这个判断
            {
                ret = std::min(ret, pVector(pt[j], pt[j + 1]).dis(_Off.pt[i + 1], true));
                ret = std::min(ret, pVector(_Off.pt[i], _Off.pt[i + 1]).dis(pt[j], true));
            }
        }
        return ret;
    }

    //-----------半平面交-------------
    //复杂度:O(nlog2(n))
    //#include <algorithm>
    //半平面计算极角函数[如果考虑效率可以用成员变量记录]
    static double hpc_pa(const pVector &_Off)
    {
        return atan2(_Off.e.y - _Off.s.y, _Off.e.x - _Off.s.x);
    }
    //半平面交排序函数[优先顺序: 1.极角 2.前面的直线在后面的左边]
    static bool hpc_cmp(const pVector &l, const pVector &r)
    {
        double lp = hpc_pa(l), rp = hpc_pa(r);
```

```
        if(fabs(lp - rp) > eps)
            return lp < rp;
        return point::xmult(l.s, r.e, r.s) < 0.0;
    }
    //用于计算的双端队列
    pVector dequeue[maxpn];
    //获取半平面交的多边形（多边形的核）
    //参数：向量集合[l]，向量数量[ln];(半平面方向在向量左边）
    //函数运行后如果n[即返回多边形的点数量]为0则不存在半平面交的多边形（不存在区域或区域面积无穷大）
    polygon& halfPanelCross(pVector _Off[], int ln)
    {
        int i, tn;
        n = 0;
        std::sort(_Off, _Off + ln, hpc_cmp);
        //平面在向量左边的筛选
        for(i = tn = 1; i < ln; i ++)
            if(fabs(hpc_pa(_Off[i]) - hpc_pa(_Off[i - 1])) > eps)
                _Off[tn ++] = _Off[i];
        ln = tn;
        int bot = 0, top = 1;
        dequeue[0] = _Off[0];
        dequeue[1] = _Off[1];
        for(i = 2; i < ln; i ++)
        {
            if(dequeue[top].parallel(dequeue[top - 1]) ||
                dequeue[bot].parallel(dequeue[bot + 1]))
                return (*this);
            while(bot < top &&
                point::xmult(dequeue[top].crossLPt(dequeue[top - 1]), _Off[i].e, _Off[i].s) > eps)
                top --;
            while(bot < top &&
                point::xmult(dequeue[bot].crossLPt(dequeue[bot + 1]), _Off[i].e, _Off[i].s) > eps)
                bot ++;
            dequeue[++ top] = _Off[i];
        }

        while(bot < top &&
            point::xmult(dequeue[top].crossLPt(dequeue[top - 1]), dequeue[bot].e, dequeue[bot].s) > eps)
            top --;
        while(bot < top &&
            point::xmult(dequeue[bot].crossLPt(dequeue[bot + 1]), dequeue[top].e, dequeue[top].s) > eps)
            bot ++;
        //计算交点(注意不同直线形成的交点可能重合)
        if(top <= bot + 1)
            return (*this);
        for(i = bot; i < top; i ++)
            pt[n ++] = dequeue[i].crossLPt(dequeue[i + 1]);
        if(bot < top + 1)
            pt[n ++] = dequeue[bot].crossLPt(dequeue[top]);
        return (*this);
    }
```

```
};
class circle
{
public:
    point c;//圆心
    double r;//半径
    double db, de;//圆弧度数起点， 圆弧度数终点(逆时针0-360)

    //-------圆---------

    //判断圆在多边形内
    bool inside(const polygon &_Off) const
    {
        if(_Off.ahas(c) == false)
            return false;
        for(int i = 0; i < _Off.n; i ++)
        {
            pVector l = pVector(_Off.pt[i], _Off.pt[(i + 1) % _Off.n]);
            if(l.dis(c, true) < r - eps)
                return false;
        }
        return true;
    }

    //判断多边形在圆内（线段和折线类似）
    bool has(const polygon &_Off) const
    {
        for(int i = 0; i < _Off.n; i ++)
            if(_Off.pt[i].dis2(c) > r * r - eps)
                return false;
        return true;
    }

    //-------圆弧-------
    //圆被其他圆截得的圆弧，参数：圆[_Off]
    circle operator-(circle &_Off) const
    {
        //注意圆必须相交，圆心不能重合
        double d2 = c.dis2(_Off.c);
        double d = c.dis(_Off.c);
        double ans = std::acos((d2 + r * r - _Off.r * _Off.r) / (2 * d * r));
        point py = _Off.c - c;
        double oans = std::atan2(py.y, py.x);
        circle res;
        res.c = c;
        res.r = r;
        res.db = oans + ans;
        res.de = oans - ans + 2 * pi;
        return res;
    }
    //圆被其他圆截得的圆弧，参数：圆[_Off]
    circle operator+(circle &_Off) const
    {
```

```
        //注意圆必须相交，圆心不能重合
        double d2 = c.dis2(_Off.c);
        double d = c.dis(_Off.c);
        double ans = std::acos((d2 + r * r - _Off.r * _Off.r) / (2 * d * r));
        point py = _Off.c - c;
        double oans = std::atan2(py.y, py.x);
        circle res;
        res.c = c;
        res.r = r;
        res.db = oans - ans;
        res.de = oans + ans;
        return res;
    }

    //过圆外一点的两条切线
    //参数：点[_Off](必须在圆外),返回：两条切线(切线的s点为_Off,e点为切点)
    std::pair<pVector, pVector>  tangent(const point &_Off) const
    {
        double d = c.dis(_Off);
        //计算角度偏移的方式
        double angp = std::acos(r / d), ango = std::atan2(_Off.y - c.y, _Off.x - c.x);
        point pl = point(c.x + r * std::cos(ango + angp), c.y + r * std::sin(ango + angp)),
            pr = point(c.x + r * std::cos(ango - angp), c.y + r * std::sin(ango - angp));
        return std::make_pair(pVector(_Off, pl), pVector(_Off, pr));
    }

    //计算直线和圆的两个交点
    //参数：直线[_Off](两点式)，返回两个交点，注意直线必须和圆有两个交点
    std::pair<point, point> cross(pVector _Off) const
    {
        _Off.pton();
        //到直线垂足的距离
        double td = fabs(_Off.a * c.x + _Off.b * c.y + _Off.c) / sqrt(_Off.a * _Off.a + _Off.b * _Off.b);

        //计算垂足坐标
        double xp = (_Off.b * _Off.b * c.x - _Off.a * _Off.b * c.y - _Off.a * _Off.c) / ( _Off.a * _Off.a + 
_Off.b * _Off.b);
        double yp = (- _Off.a * _Off.b * c.x + _Off.a * _Off.a * c.y - _Off.b * _Off.c) / (_Off.a * _Off.a + 
_Off.b * _Off.b);

        double ango = std::atan2(yp - c.y, xp - c.x);
        double angp = std::acos(td / r);

        return std::make_pair(point(c.x + r * std::cos(ango + angp), c.y + r * std::sin(ango + angp)),
            point(c.x + r * std::cos(ango - angp), c.y + r * std::sin(ango - angp)));
    }
};

class triangle
{
public:
    point a, b, c;//顶点
    triangle(){}
    triangle(point a, point b, point c): a(a), b(b), c(c){}
```

```
//计算三角形面积
double area()
{
    return fabs(point::xmult(a, b, c)) / 2.0;
}

//计算三角形外心
//返回：外接圆圆心
point circumcenter()
{
    pVector u,v;
    u.s.x = (a.x + b.x) / 2;
    u.s.y = (a.y + b.y) / 2;
    u.e.x = u.s.x - a.y + b.y;
    u.e.y = u.s.y + a.x - b.x;
    v.s.x = (a.x + c.x) / 2;
    v.s.y = (a.y + c.y) / 2;
    v.e.x = v.s.x - a.y + c.y;
    v.e.y = v.s.y + a.x - c.x;
    return u.crossLPt(v);
}

//计算三角形内心
//返回：内接圆圆心
point incenter()
{
    pVector u, v;
    double m, n;
    u.s = a;
    m = atan2(b.y - a.y, b.x - a.x);
    n = atan2(c.y - a.y, c.x - a.x);
    u.e.x = u.s.x + cos((m + n) / 2);
    u.e.y = u.s.y + sin((m + n) / 2);
    v.s = b;
    m = atan2(a.y - b.y, a.x - b.x);
    n = atan2(c.y - b.y, c.x - b.x);
    v.e.x = v.s.x + cos((m + n) / 2);
    v.e.y = v.s.y + sin((m + n) / 2);
    return u.crossLPt(v);
}

//计算三角形垂心
//返回：高的交点
point perpencenter()
{
    pVector u,v;
    u.s = c;
    u.e.x = u.s.x - a.y + b.y;
    u.e.y = u.s.y + a.x - b.x;
    v.s = b;
    v.e.x = v.s.x - a.y + c.y;
    v.e.y = v.s.y + a.x - c.x;
    return u.crossLPt(v);
```

```
    }

    //计算三角形重心
    //返回：重心
    //到三角形三顶点距离的平方和最小的点
    //三角形内到三边距离之积最大的点
    point barycenter()
    {
        pVector u,v;
        u.s.x = (a.x + b.x) / 2;
        u.s.y = (a.y + b.y) / 2;
        u.e = c;
        v.s.x = (a.x + c.x) / 2;
        v.s.y = (a.y + c.y) / 2;
        v.e = b;
        return u.crossLPt(v);
    }

    //计算三角形费马点
    //返回：到三角形三顶点距离之和最小的点
    point fermentpoint()
    {
        point u, v;
        double step = fabs(a.x) + fabs(a.y) + fabs(b.x) + fabs(b.y) + fabs(c.x) + fabs(c.y);
        int i, j, k;
        u.x = (a.x + b.x + c.x) / 3;
        u.y = (a.y + b.y + c.y) / 3;
        while (step > eps)
        {
            for (k = 0; k < 10; step /= 2, k ++)
            {
                for (i = -1; i <= 1; i ++)
                {
                    for (j =- 1; j <= 1; j ++)
                    {
                        v.x = u.x + step * i;
                        v.y = u.y + step * j;
                        if (u.dis(a) + u.dis(b) + u.dis(c) > v.dis(a) + v.dis(b) + v.dis(c))
                            u = v;
                    }
                }
            }
        }
        return u;
    }
};


/*
凸包，凸包点排序逆时针输出*/

#include <cstdio>
#include <cstring>
#include <cmath>
```

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

#define sqr(a) ((a) * (a))
#define dis(a, b) sqrt(sqr(a.x - b.x) + sqr(a.y - b.y))

const int MAXN = 110;
const double PI = acos(-1.0);

struct Point {
	int x;
	int y;
	Point(double a = 0, double b = 0) : x(a), y(b) {}
	friend bool operator < (const Point &l, const Point &r) {
		return l.y < r.y || (l.y == r.y && l.x < r.x);
	}
} p[MAXN], ch[MAXN];
// p, point   ch, convex hull

double mult(Point a, Point b, Point o) {
	return (a.x - o.x) * (b.y - o.y) >= (b.x - o.x) * (a.y - o.y);
}

int Graham(Point p[], int n, Point res[]) {
	int top = 1;
	sort(p, p + n);
	if (n == 0) return 0;
	res[0] = p[0];
	if (n == 1) return 0;
	res[1] = p[1];
	if (n == 2) return 0;
	res[2] = p[2];
	for (int i = 2; i < n; i++) {
		while (top && (mult(p[i], res[top], res[top - 1])))
			top--;
		res[++top] = p[i];
	}
	int len = top;
	res[++top] = p[n - 2];
	for (int i = n - 3; i >= 0; i--) {
		while (top != len && (mult(p[i], res[top], res[top - 1])))
			top--;
		res[++top] = p[i];
	}
	return top;
}

int n;

int main() {
	while (scanf("%d%d", &p[n].x, &p[n].y) != EOF)
		n++;
	n = Graham(p, n, ch);
	int t;
	for (int i = 0; i < n; i++)
		if (ch[i].x == 0 && ch[i].y == 0) {
```

```
                    t = i;
                    break;
                }

        for (int i = t; i < n; i++)
                printf("(%d,%d)\n", ch[i].x, ch[i].y);
        for (int i = 0; i < t; i++)
                printf("(%d,%d)\n", ch[i].x, ch[i].y);
        return 0;
}
```

快速幂

```
typedef long long LL;
LL fun(LL x,LL n,)
{
    LL res=1;
    while(n>0)
    {
        if(n & 1)
            res=(res*x)%Max;
        x=(x*x)%Max;
        n >>= 1;
    }
    return res;
}
```

矩阵快速幂

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

using namespace std;
typedef vector<int> vec;
typedef vector<vec> mat;
typedef long long LL;
const int N = 10000;
mat mul(mat a,mat b)  //矩阵乘法
{
    mat c(a.size(),vec(b[0].size()));
    for(int i=0;i<a.size();i++)
    {
        for(int k=0;k<b.size();k++)
        {
            for(int j=0;j<b[0].size();j++)
                c[i][j] = ( c[i][j] + a[i][k] * b[k][j] ) % N;
        }
    }
    return c;
}

mat solve_pow(mat a,int n) //快速幂
{
    mat b(a.size(),vec(a.size()));
```

```
    for(int i=0;i<a.size();i++)
        b[i][i]=1;
    while(n>0)
    {
        if(n & 1)
            b=mul(b,a);
        a=mul(a,a);
        n >>= 1;
    }

    return b;
}
LL n;
void solve()
{
    mat a(2,vec(2));
    while(~scanf("%d",&n) && n!=-1)
    {
        a[0][0]=1,a[0][1]=1;
        a[1][0]=1,a[1][1]=0;
        a=solve_pow(a,n);
        printf("%d\n",a[1][0]);
    }
}
int main()
{
    solve();
    return 0;
}
```

//////需要懂的东西

“在一棵树上进行路径的修改、求极值、求和”乍一看只要线段树就能轻松解决，实际上，仅凭线段树是不能搞定它的。我们需要用到一种貌似高级的复杂算法——树链剖分。

树链，就是树上的路径。剖分，就是把路径分类为重链和轻链。

记siz[v]表示以v为根的子树的节点数，dep[v]表示v的深度(根深度为1)，top[v]表示v所在的链的顶端节点，fa[v]表示v的父亲，son[v]表示与v在同一重链上的v的儿子节点（姑且称为重儿子），w[v]表示v与其父亲节点的连边（姑且称为v的父边）在线段树中的位置。只要把这些东西求出来，就能用logn的时间完成原问题中的操作。

重儿子：siz[u]为v的子节点中siz值最大的，那么u就是v的重儿子。
轻儿子：v的其它子节点。
重边：点v与其重儿子的连边。
轻边：点v与其轻儿子的连边。
重链：由重边连成的路径。
轻链：轻边。

剖分后的树有如下性质：
性质1：如果(v,u)为轻边，则siz[u] * 2 < siz[v]；
性质2：从根到某一点的路径上轻链、重链的个数都不大于logn。

算法实现：

我们可以用两个dfs来求出fa、dep、siz、son、top、w。

dfs_1：把fa、dep、siz、son求出来，比较简单，略过。

dfs_2：1.对于v，当son[v]存在（即v不是叶子节点）时，显然有top[son[v]] = top[v]。线段树中，v的重边应当在v的父边的后面，记w[son[v]] = totw+1，totw表示最后加入的一条边在线段树中的位置。此时，为了使一条重链各边在线段树中连续分布，应当进行dfs_2(son[v])；

2.对于v的各个轻儿子u，显然有top[u] = u，并且w[u] = totw+1，进行dfs_2过程。

这就求出了top和w。

将树中各边的权值在线段树中更新，建链和建线段树的过程就完成了。

修改操作：例如将u到v的路径上每条边的权值都加上某值x。

一般人需要先求LCA，然后慢慢修改u、v到公共祖先的边。而高手就不需要了。

记f1 = top[u]，f2 = top[v]。

当f1 <> f2时：不妨设dep[f1] >= dep[f2]，那么就更新u到f1的父边的权值(logn)，并使u = fa[f1]。

当f1 = f2时：u与v在同一条重链上，若u与v不是同一点，就更新u到v路径上的边的权值(logn)，否则修改完成；

重复上述过程，直到修改完成。

求和、求极值操作：类似修改操作，但是不更新边权，而是对其求和、求极值。

就这样，原问题就解决了。鉴于鄙人语言表达能力有限，咱画图来看看：[转载]树链剖分

如右图所示，较粗的为重边，较细的为轻边。节点编号旁边有个红色点的表明该节点是其所在链的顶端节点。边旁的蓝色数字表示该边在线段树中的位置。图中1-4-9-13-14为一条重链。

当要修改11到10的路径时。

第一次迭代：u = 11，v = 10，f1 = 2，f2 = 10。此时dep[f1] < dep[f2]，因此修改线段树中的5号点，v = 4, f2 = 1；

第二次迭代：dep[f1] > dep[f2]，修改线段树中10--11号点。u = 2，f1 = 2；

第三次迭代：dep[f1] > dep[f2]，修改线段树中9号点。u = 1，f1 = 1；

第四次迭代：f1 = f2且u = v，修改结束。

题目：spoj375、USACO December Contest Gold Divison, "grassplant"。

**spoj375据说不“缩行”情况下最短的程序是140+行，我的是128行。

附spoj375程序(C++)：

```
#include <cstdio>
#include <algorithm>
#include <iostream>
#include <string.h>
using namespace std;
const int maxn = 10010;
struct Tedge
{ int b, next; } e[maxn * 2];
int tree[maxn];
```

```
int zzz, n, z, edge, root, a, b, c;
int d[maxn][3];
int first[maxn], dep[maxn], w[maxn], fa[maxn], top[maxn], son[maxn], siz[maxn];
char ch[10];

void insert(int a, int b, int c)
{
    e[++edge].b = b;
    e[edge].next = first[a];
    first[a] = edge;
}

void dfs(int v)
{
    siz[v] = 1; son[v] = 0;
    for (int i = first[v]; i > 0; i = e[i].next)
        if (e[i].b != fa[v])
        {
            fa[e[i].b] = v;
            dep[e[i].b] = dep[v]+1;
            dfs(e[i].b);
            if (siz[e[i].b] > siz[son[v]]) son[v] = e[i].b;
            siz[v] += siz[e[i].b];
        }
}

void build_tree(int v, int tp)
{
    w[v] = ++ z; top[v] = tp;
    if (son[v] != 0) build_tree(son[v], top[v]);
    for (int i = first[v]; i > 0; i = e[i].next)
        if (e[i].b != son[v] && e[i].b != fa[v])
            build_tree(e[i].b, e[i].b);
}

void update(int root, int lo, int hi, int loc, int x)
{
    if (loc > hi || lo > loc) return;
    if (lo == hi)
    { tree[root] = x; return; }
    int mid = (lo + hi) / 2, ls = root * 2, rs = ls + 1;
    update(ls, lo, mid, loc, x);
    update(rs, mid+1, hi, loc, x);
    tree[root] = max(tree[ls], tree[rs]);
}

int maxi(int root, int lo, int hi, int l, int r)
{
    if (l > hi || r < lo) return 0;
    if (l <= lo && hi <= r) return tree[root];
    int mid = (lo + hi) / 2, ls = root * 2, rs = ls + 1;
    return max(maxi(ls, lo, mid, l, r), maxi(rs, mid+1, hi, l, r));
}

inline int find(int va, int vb)
{
```

```
    int f1 = top[va], f2 = top[vb], tmp = 0;
    while (f1 != f2)
    {
        if (dep[f1] < dep[f2])
        { swap(f1, f2); swap(va, vb); }
        tmp = max(tmp, maxi(1, 1, z, w[f1], w[va]));
        va = fa[f1]; f1 = top[va];
    }
    if (va == vb) return tmp;
    if (dep[va] > dep[vb]) swap(va, vb);
    return max(tmp, maxi(1, 1, z, w[son[va]], w[vb]));  //
}

void init()
{
    scanf("%d", &n);
    root = (n + 1) / 2;
    fa[root] = z = dep[root] = edge = 0;
    memset(siz, 0, sizeof(siz));
    memset(first, 0, sizeof(first));
    memset(tree, 0, sizeof(tree));
    for (int i = 1; i < n; i++)
    {
       scanf("%d%d%d", &a, &b, &c);
       d[i][0] = a; d[i][1] = b; d[i][2] = c;
       insert(a, b, c);
       insert(b, a, c);
    }
    dfs(root);
    build_tree(root, root);    //
    for (int i = 1; i < n; i++)
    {
       if (dep[d[i][0]] > dep[d[i][1]]) swap(d[i][0], d[i][1]);
       update(1, 1, z, w[d[i][1]], d[i][2]);
    }
}

inline void read()
{
    ch[0] = ' ';
    while (ch[0] < 'C' || ch[0] > 'Q') scanf("%s", &ch);
}

void work()
{
    for (read(); ch[0] != 'D'; read())
    {
       scanf("%d%d", &a, &b);
       if (ch[0] == 'Q') printf("%dn", find(a, b));
                 else update(1, 1, z, w[d[a][1]], b);
    }
}

int main()
{
   for (scanf("%d", &zzz); zzz > 0; zzz--)
```

```
    {
        init();
        work();
    }
    return 0;
}
```

一、树状数组是干什么的?

平常我们会遇到一些对数组进行维护查询的操作，比较常见的如，修改某点的值、求某个区间的和，而这两种恰恰是树状数组的强项！当然，数据规模不大的时候，对于修改某点的值是非常容易的，复杂度是O(1)，但是对于求一个区间的和就要扫一遍了，复杂度是O(N)，如果实时的对数组进行M次修改或求和，最坏的情况下复杂度是O(M*N)，当规模增大后这是划不来的！而树状数组干同样的事复杂度却是O(M*lgN)，别小看这个lg，很大的数一lg就很小了，这个学过数学的都知道吧，不需要我说了。申明一下，看下面的文章一定不要急，只需要看懂每一步最后自然就懂了。

二、树状数组怎么干的?

先看两幅图（网上找的，如果雷同，不要大惊小怪～），下面的说明都是基于这两幅图的，左边的叫A图吧，右边的叫B图：

是不是很像一颗树？对，这就是为什么叫树状数组了～先看A图，a数组就是我们要维护和查询的数组，但是其实我们整个过程中根本用不到a数组，你可以把它当作一个摆设！c数组才是我们全程关心和操纵的重心。先由图来看看c数组的规则，其中c8 = c4+c6+c7+a8，c6 = c5+a6……先不必纠结怎么做到的，我们只要知道c数组的大致规则即可，很容易知道c8表示a1～a8的和，但是c6却是表示a5～a6的和，为什么会产生这样的区别的呢？或者说发明她的人为什么这样区别对待呢？答案是，这样会使操作更简单！看到这相信有些人就有些感觉了，为什么复杂度被lg了呢？可以看到，c8可以看作a1～a8的左半边和+右半边和，而其中左半边和是确定的c4，右半边其实也是同样的规则把a5～a8一分为二……继续下去都是一分为二直到不能分，可以看看B图。怎么样？是不是有点二分的味道了？对，说白了树状数组就是巧妙的利用了二分，她并不神秘，关键是她的巧妙！

她又是怎样做到不断的一分为二呢？说这个之前我先说个叫lowbit的东西，lowbit(k)就是把k的二进制的高位1全部清空，只留下最低位的1,比如10的二进制是1010,则lowbit(k)=lowbit(1010)=0010(2进制)，介于这个lowbit在下面会经常用到，这里给一个非常方便的实现方式，比较普遍的方法lowbit(k)=k&-k，这是位运算，我们知道一个数加一个负号是把这个数的二进制取反+1，如-10的二进制就是-1010=0101+1=0110，然后用1010&0110，答案就是0010了！明白了求解lowbit的方法就可以了，继续下面。介于下面讨论十进制已经没有意义（这个世界本来就是二进制的，人非要主观的构建一个十进制），下面所有的数没有特别说明都当作二进制。

上面那么多文字说lowbit，还没说它的用处呢，它就是为了联系a数组和c数组的！ck表示从ak开始往左连续求lowbit(k)个数的和，比如c[0110]=a[0110]+a[0101]，就是从110开始计算了0010个数的和，因为lowbit(0110)=0010，可以看到其实只有低位的1起作用，因为很显然可以写出c[0010]=a[0010]+a[0001]，这就为什么我们任何数都只关心它的lowbit，因为高位不起作用（基于我们的二分规则它必须如此！），除非除了高位其余位都是0，这时本身就是lowbit。

既然关系建立好了，看看如何实现a某一个位置数据跟改的，她不会直接改的（开始就说了，a根本不存在），她每次改其实都要维护c数组应有的性质，因为后面求和要用到。而维护也很简单，比如更改了a[0011]，我们接着要修改c[0011],c[0100],c[1000]，这是很容易从图上看出来的，但是你可能会问，他们之间有申明必然联系吗？每次求解总不能总要拿图来看吧？其实从0011——>0100——>1000的变化都是进行“去尾”操作，又是自己造的词--''，我来解释下，就是把尾部应该去掉的1都去掉转而换到更高位的1,记住每次变换都要有一个高位的1产生，所以0100是不能变换到0101的，因为没有新的高位1产生，这个变换过程恰好是可以借助我们的lowbit进行的，k +=lowbit(k)。

好吧，现在更新的次序都有了，可能又会产生新的疑问了：为什么它非要是这种关系啊？这就要追究到之前我们说c8可以看作a1～a8的左半边和+右半边和……的内容了，为什么c[0011]会影响到c[0100]而不会影响到c[0101]，这就是之前说的c[0100]的求解实际上是这样分段的区间c[0001]~c[0001] 和区间c[0011]~c[0011]的和，数字太小，可能这样不太理解，在比如c[0100]会影响c[1000]，为什么呢？因为c[1000]可以看作0001～0100的和加上0101~1000的和，但是0101位置的数变化并会直接作用于c[1000]，因为它的尾部1不能一下在跳两级在产生两次高位1,是通过c[0110]间接影响的，但是，c[0100]却可以跳一级产生一次高位1。

可能上面说的你比较绕了，那么此时你只需注意：c的构成性质（其实是分组性质）决定了c[0011]只会直接影响c[0100]，而c[0100]只会直接影响[1000]，而下表之间的关系恰好是也必须是k+=lowbit(k)。此时我们就是写出跟新维护树的代码：

[cpp] view plain copy print?

```
void add(int k,int num)
{
   while(k<=n)
   {
      tree[k]+=num;
      k+=k&-k;
   }
}
```

有了上面的基础，说求和就比较简单了。比如求0001～0110的和就直接c[0100]+c[0110]，分析方法与上面的恰好逆过来，而且写法也是逆过来的，具体就不累述了：

[cpp] view plain copy print?

```
int read(int k)//1~k的区间和
{
   int sum=0;
   while(k)
   {
      sum+=tree[k];
      k-=k&-k;
   }
   return sum;
}
```

三、总结一下吧

首先，明白树状数组所白了是按照二分对数组进行分组；维护和查询都是O(lgn)的复杂度，复杂度取决于最坏的情况，也是O(lgn);lowbit这里只是一个技巧，关键在于明白c数组的构成规律;分析的过程二进制一定要深入人心，当作心目中的十进制。