ACM-ICPC

Soochow University

Author：白 岩
E-mail: Paladnix@outlook.com

Paladnix
JUST A CODER

Contents

Chapter 1

字符串

1.1 KMP

KMP 算法是单模式匹配算法，多模式串匹配应该使用 AC 自动机。可以解决的问题有单次匹配，多次可覆盖匹配，多次不可覆盖匹配，求循环节。算法复杂度是 $O(m+n)$

- 模式串：t
- 待匹配串：s
- 模式串长度：nt
- 待匹配长度：ns
- 自匹配数组：Next

```
//*******************************************************
// 求匹配问题
//*******************************************************
int ns,nt;
int s[1000005]={0},t[10005]={0};
int Next[10005]={0};
//Next数组表示如果当前位置匹配失败时，模式串在下次匹配之前已经匹配到的最大的长度。
int KMP()
{
   for(int i = 1; i < nt ; i++)
   {
      int j = i;
      while( j > 0 )
      {
         j = Next[j];
         if( t[j] == t[i] )
         {
            Next[i+1] = j + 1; break;
         }
      }
   }
   for(int i=0 ,j=0 ; i < ns ; i++)
   {
      if( j < nt && t[j] == s[i] ) j++;
      else while( j > 0 )
         {
            j = Next[j];
            if(t[j]==s[i])
            {
               j++; break;
            }
         }
      if( j == nt )
         return i - nt + 1 + 1; //起点下标从1开始;
      //如果是可覆盖的重复匹配就不return，记录下位置继续执行;
```

```
        //如果是不可覆盖的最多重复匹配，记下匹配的下标然后贪心统计就可以；
        //代码需要修改如下：
        // i++不放在for条件语句内，
        // if( j == nt )
        // {
        //  ans++;
        //  i+=nt;
        // }
        // else i++;
    }
    return -1;
}
```

求循环节的方法很简单，就是上面的 Next 的数组的应用，Next 数组的另一个理解是：当前前缀串中，后缀与前缀的最大匹配长度。求一个串的最大循环节：$cir = nt - Next[nt]$，循环并不是完全循环，最后的循环节有可能是不完整的，判断是否完整的方法也很简单，看 $nt\%cir == 0$ 就可以了。同样，判断某个前缀是否是完全循环串也是一样的道理。注意只循环一次的串要特殊判断。

1.2 扩展 KMP

扩展 KMP 算法，也是有两个过程，第一步的自匹配，用来求出一个字符串的所有的后缀与原串的最长公共前缀，与 KMP 有点反一下的感觉，KMP 的 Next 数组求出的是所有前缀的后缀与原串的最长公共前缀。第二步串间匹配，过程跟第一步的过程一模一样。

```
//*****************************************************
// 求串间或内前缀与后缀的问题
//*****************************************************
char t[50005],s[50005];
int Next[50005]={0},ret[50005]={0};
//Next是自匹配的信息。ret是串间匹配的信息。
//表示以第i个字符开始的后缀与串的最大公共前缀。
int nt,ns;

void ExtendKmp()
{
    int i,j,k;
    for( j = 0; j + 1 < nt && t[j] == t[j+1]; j ++);
    Next[1] = j;
    k = 1;
    for(i = 2; i < nt ; i ++)
    {
        int p = k + Next[k], l = Next[i-k];
        if( l + i < p ) Next[i] = l;
        else
        {
            for(j = max( 0, p-i ); i + j < nt && a[j] == a[i+j]; j ++);
            Next[i] = j;
            k = i;
        }
    }
    for( j = 0 ; j < ns && j < nt && a[j] == b[j] ; j ++);
    ret[0] = j;
    k = 0;
    for(i = 1; i < ns; i ++)
    {
        int p = k + ret[k], l = Next[i-k];
        if( l + i< p) ret[i] = l;
        else
        {
            for(j = max(0 ,p-i); j < nt && i + j < ns && a[j] == b[i+j] ; j ++);
            ret[i]=j;
            k=i;
        }
    }
}
```

扩展 KMP 没有一个特定的题目要用这个来处理，但是有很多题目是可以转化成前缀与后缀的问题，就可以用扩展 KMP 来优化。另外在字符串处理的时候有很多字符串拼接后处理的例子，这样的做法在后缀数组中使用的比较多。扩展 KMP 的 Next 数组也有很多用法。Next 数组保存的是以对应位置开头的后缀和原串的最大公共前缀。下面有几个例题：

例 1：求给定字符串所有前缀在串中出现的次数和 -hdu3336。我们计算每个后缀和前缀的最长公共部分对与答案的贡献就可以知道，只要把 Next 数组加起来就可以了。

1.3 Manachar

Manachar 有一个更炫酷的名字叫回文串自动机，所以很好理解，就是用来处理回文串问题的，将原来的串用其他符号间隔开，避免回文串长度奇偶讨论的麻烦，处理出以每个位置为中心的最长回文串的长度。算法复杂度 $O(n)$

```
//*******************************************************
// 求最长回文串问题
//*******************************************************
int n;
int p[300005];
char s[110005],str[110005<<1];
/*
   首先第一步是将原来的串扩展。避免奇偶分类讨论，在开头设置一个不用的东西，从1开始用#分隔；
   字符串的长度相应的改变；
   最后要记得在新串的最后加\0;
   */
void init()
{
    str[0] = '@';
    str[1] = '#';
    for(int i = 0; i < n; i ++)
    {
        str[i*2+2] = s[i];
        str[i*2+3] = '#';
    }
    n = 2*n + 2;
    str[n] = 0;
}
/*
   计算以每一个字符作为中心的最长回文串的长度，但是在后面可以进行剪枝，避免多余的比较，就是利用已有的回文串，
        在前面的某个字符作为中心的情况计算过了的话就可以直接用它的数据，因为他们在对称的位置上，
        在一定的范围内周边的字符是一样的，id就是用来记录上一个回文串的中心是哪一个，用来计算对称位置用的，最后扫一遍p
        数组，结果减一；
 */
void Manachar()
{
    int mx = 0,id = 0;
    for(int i = 1; i < n; i ++)
    {
        if(mx > i)
        {
            p[i] = min(p[2*id-i] , p[id]+id-i);
            //p[id*2-i]表示i的对称位置上的那个字符的回文串长度，p[id]+id-i表示的是以id为中心的回文串的边界距离i的距离
                 , 在这个距离以内的都可用对称点的数据，超出的就得自行比较了；省下一部分的比较；
        }
        else
            p[i] = 1;
        for(; str[i+p[i]] == str[i-p[i]] ; p[i]++);
        if(mx < p[i] + i)
        {
            //mx记录回文串的最远的边界；
            mx =p[i] + i;
            id = i;
        }
    }
}
```

1.4 字典树

字典树是一种数据结构，经常用于处理多模式串的问题和数字按位计算的问题，常见的字典树有 26 叉，2 叉，10 叉等，基本原理都是一样的，支持的操作有插入、删除、查询，灵活度比较大。

我的字典树是内存池的写法。

```
//*******************************************************
// 字典树模板
//*******************************************************
struct Tire
{ //2叉字典树 0-1序列
    int root, L, Next[N<<4][2], end[N<<4];
    int newNode()
    {
        Next[L][0] = Next[L][1] = -1;
        end[L] = 0;
        L++;
        return L-1;
    }
    void init()
    {
        L =0;
        root = newNode();
    }
    void insert(char buf[])
    {
        int len = strlen(buf);
        int now = root;
        for( int i = 0; i < len; i++)
        {
            if(Next[now][buf[i]] == -1) //如果是字符的话要改成buf[i]-'a';
                Next[now][buf[i]] = newNode();
            now = Next[now][buf[i]];
            end[now]++;
        }
        //如果需要支持删除操作就要把end[now]++放在for循环的里面
        //不需要删除操作的就把它放在循环的外面。
        //因为删除某个元素的时候要整个链都删掉。
    }
    void del(char buf[])
    {
        int now = root;
        int len = strlen(buf);
        for(int i = 0; i < len ; i++)
        {
            now = Next[now][buf[i]];
            end[now]--;
        }
    }

    int Query(char buf[])
    {
        int len = strlen(buf);
        int now = root;
        int ans =0;
        for(int i = 0 ; i < len;i ++)
        {
            int next = Next[now][!buf[i]];
            if(next==-1 || end[next]<=0)
            {
                now = Next[now][buf[i]];
                ans = ans*2;
            }
            else
            {
                ans = ans*2+1;
                now = Next[now][!buf[i]];
            }
        }
        return ans;
```

```
    }
}tire;
```

1.5 AC 自动机

AC 自动机是在字典树的基础上把 KMP 搞在树上，这样处理多模式串匹配的问题。在字典树的基础上新加 $fail[N]$，意思跟 KMP 的 Next 数组很像，就是失配后跳转到的位置，也就是建立了一个状态转移图，有了状态转移图的概念，在此基础上就会有矩阵加速，Dp 等套过来。

```
//******************************************************
// AC自动机模板
//******************************************************
struct Trie
{
    int next[500010][26],fail[500010],end[500010];
    int root,L;
    int newnode()
    {
        for(int i = 0;i < 26;i++)
            next[L][i] = -1;
        end[L++] = 0;
        return L-1;
    }
    void init()
    {
        L = 0;
        root = newnode();
    }
    void insert(char buf[])
    {
        int len = strlen(buf);
        int now = root;
        for(int i = 0;i < len;i++)
        {
            if(next[now][buf[i]-'a'] == -1)
                next[now][buf[i]-'a'] = newnode();
            now = next[now][buf[i]-'a'];
        }
        end[now]++;
    }
    void build()
    {
        queue<int>Q;
        fail[root] = root;
        for(int i = 0;i < 26;i++)
            if(next[root][i] == -1)
                next[root][i] = root;
            else
            {
                fail[next[root][i]] = root;
                Q.push(next[root][i]);
            }
        while( !Q.empty() )
        {
            int now = Q.front();
            Q.pop();
            for(int i = 0;i < 26;i++)
                if(next[now][i] == -1)
                    next[now][i] = next[fail[now]][i];
                else
                {
                    fail[next[now][i]]=next[fail[now]][i];
                    Q.push(next[now][i]);
                }
        }
    }
    int query(char buf[])
```

```
    {
        int len = strlen(buf);
        int now = root;
        int res = 0;
        for(int i = 0;i < len;i++)
        {
            now = next[now][buf[i]-'a'];
            int temp = now;
            while( temp != root )
            {
                res += end[temp];
                end[temp] = 0;
                temp = fail[temp];
            }
        }
        return res;
    }
    void debug()
    {
        for(int i = 0;i < L;i++)
        {
            printf("id = %3d,fail = %3d,end = %3d,chi = [",i,fail[i],end[i]);
            for(int j = 0;j < 26;j++)
                printf("%2d",next[i][j]);
            printf("]\n");
        }
    }
};
插入字符串的后需要build一下才能建起来自动机。
```

后续会放一些例题在这里，都是比较高端的题。

1.6 后缀数组

后缀数组是后缀树的简化版，但是同样功能强大。后缀数组处理出了几个数组，每个后缀的排名数组 rank，按排序先后排列的下标数组 sa，此外还有 $O(n)$ 处理出的 Height 数组，Height 数组表示的是排序相邻的两个后缀的最长公共前缀，$Height[1] = 0$。

```
//**************************************************
// 后缀数组
//**************************************************
int n,m;
int r[MAXN];
int wa[MAXN], wb[MAXN], wv[MAXN], tmp[MAXN];
int sa[MAXN]; //index range 1~n value range 0~n-1
int cmp(int *r, int a, int b, int l)
{
    return r[a] == r[b] && r[a + l] == r[b + l];
}
void da(int *r, int *sa, int n, int m)
{
    int i, j, p, *x = wa, *y = wb, *ws = tmp;
    for (i = 0; i < m; i++) ws[i] = 0;
    for (i = 0; i < n; i++) ws[x[i] = r[i]]++;
    for (i = 1; i < m; i++) ws[i] += ws[i - 1];
    for (i = n - 1; i >= 0; i--) sa[--ws[x[i]]] = i;
    for (j = 1, p = 1; p < n; j *= 2, m = p)
    {
        for (p = 0, i = n - j; i < n; i++) y[p++] = i;
        for (i = 0; i < n; i++)
            if (sa[i] >= j) y[p++] = sa[i] - j;
        for (i = 0; i < n; i++) wv[i] = x[y[i]];
        for (i = 0; i < m; i++) ws[i] = 0;
        for (i = 0; i < n; i++) ws[wv[i]]++;
        for (i = 1; i < m; i++) ws[i] += ws[i - 1];
        for (i = n - 1; i >= 0; i--) sa[--ws[wv[i]]] = y[i];
        for (swap(x, y), p = 1, x[sa[0]] = 0, i = 1; i < n; i++)
            x[sa[i]] = cmp(y, sa[i - 1], sa[i], j) ? p - 1 : p++;
```

```
    }
}
int Rank[MAXN]; //index range 0~n-1 value range 1~n
int Height[MAXN]; //index from 1 (height[1] = 0)
void calheight(int *r, int *sa, int n)
{
    int i, j, k = 0;
    for (i = 1; i <= n; ++i) Rank[sa[i]] = i;
    for (i = 0; i < n; Height[Rank[i++]] = k)
        for (k ? k-- : 0, j = sa[Rank[i] - 1]; r[i + k] == r[j + k]; ++k);
    return;
}
char s[MAXN];
int main()
{
    scanf("%s␣%s", a,s);
    n = strlen(s);
    m =0;
    for(int i=0;i<n;i++)
    {
        r[i] = (int)s[i];
        m = max(m,r[i]);
    }
    r[n] = 0;
    da(r,sa,n+1 , m+1);
    calheight(r,sa,n);
}
```

后缀数组的一些应用：

例 1：求某两个后缀的最大公共前缀 LCP。

就是求 Height 数组中的一段区间最小值，这个使用 RMQ 可以加速，这段区间就是两个后缀的 rank 之间的那段区间，左开右闭。

例 2：求最长重复字串（可覆盖）。

由于是可覆盖，求最长重复就是求两个后缀最长前缀的最大，就是 Height 数组中的最大值。

例 3：最长重复子串（不可覆盖）。

这个问题，首先二分最长的长度 k，然后把 Height 数组分组进行验证，分组的原则是组内的 Height 数组值必须 $\geqslant k$，这样符合要求的重复子串一定在一组内，对于一组内的 sa 值找到最大和最小的，差值是不是 $\geqslant k$，只要存在一组就可以。总复杂度是 $O(nlogn)$。

例 4：至少重复 k 次的子串。

做法跟上一个差不多，判断组内是否有 k 个后缀就可以了。

例 5：不相同的子串的个数。

每个子串都一定是某个后缀的前缀，原问题等价于求所有后缀的不同的前缀的个数。我们计算贡献。按照 sa 数组的顺序来思考，每次添加一个后缀有 Height[i] 个前缀是和前面一个后缀的前缀是相同的，那么这个后缀贡献的数量就是 $n - sa[i] + 1 - Height[i]$，累加后就是答案。

例 6：最长回文子串。

这个问题最好的解决方法是上面的 Manachar，但是这里介绍一种方法，就是将原来的串反过来拼接在原来的串后面，中间用一个特殊字符隔开，原来的问题就变成了最长公共前缀的问题了，枚举回文串的对称中心，然后查出 LCP 就可以啦。

例 7：最小循环节。

已知字符串是个完全循环串，我们枚举循环节的长度 k，然后判断是否整除，然后判断 suffix(1) 和 suffix(k+1) 的 LCP 是不是 $n - k$。

例 8：字符串中出现次数最多的连续重复子串。

枚举重复子串的长度 L，出现一次一定是可以的，我们直接来考虑出现两次的。记子串为 s，原串为 t，那么对于 t[0]、t[L]、t[2*L]... 中必然有两个相邻的字符出现在 s 中，枚举 t[L*i] 和 t[l*(i+1)]，那么我们以这两个为基准往两边扩展匹配，匹配到最远。这个过程事实上就是用两个基准点中间的那部分（循环节）去互相延伸，记录延伸的最大长度 K，循环的次数就是 $K/L+1$。往两边延伸的时候每次延伸一个循环节长度，验证是否是循环节，总复杂度是 $O(nlogn)$。

例 9：最长公共子串。

做法十分简单，最主要的就是要理解，每一个子串都是一个后缀的前缀，求最长公共的子串就是求两个后缀的最长公共前缀。很显然，首先我们先把两个串拼起来，中间使用一个特殊字符隔开，然后看排名相邻的两个后缀是不是来自不同的串，然后输出最大的就可以了。

例 10：求长度不小于 k 的公共子串的数量（可以相同）。
这个的总体思路应该是很明显的了。首先就是把两个字符串拼接起来，记两个串分别是 A、B，对 height 数组进行分组，然后对 B 的每个后缀，都在组内寻找对应能和之前 A 的后缀产生的公共子串的数量。这里的 A 的后缀需要用一个单调栈进行高效维护。对 A 数组也做一次，就是最后的答案了。注意在寻找的对应的后缀的数量的时候要只寻找之前的 A 的后缀，否则就会重复或是遗漏。
例 11：出现在不少于 K 个字符串中的最长公共前缀。
做法和前面的差不多。先拼接起来，然后分组，看组内是否是不少于 K 个字符串的后缀，二分答案。
例 12：n 个字符串，求出现在每一个串中并重复至少两次且不重复且不覆盖的最长子串。
拼接，然后二分答案，然后判断的时候有点麻烦。需要判断组内是否每个字符串的后缀都有，而且每个字符串的后缀都出现至少两次，并且不能重合，就是同字符串的几个在同组的后缀之间的距离是否都满足不重和。
例 13：某一字符串是否出现或翻转后出现在 n 个字符串中。
做法一样，先把字符串们翻转拼在自己后面，然后再全拼起来，然后二分答案，分组判断。

1.7 Hash

Hash 有两种写法，一种是滚动的 Hash，还有一种是 HashMap。第一种会出现 Hash 值重复的情况，一般可以使用双 Hash 来避免，第二种不存在这个问题，当 Hash 值重复的时候会挂链表，但是有可能会时间复杂度上退化。不过这都不重要。

```
//*****************************************
// 滚动Hash
//*****************************************
const int MOD = 1000000007;
struct Hash
{
    //选择B进制： 173,179,271, 277 ...
    int B, mod, len, hash[1000010], pw[1000010];
    void init(char *s, int tlen, int tb, int tmod)
    {
        B = tb;
        len = tlen;
        mod = tmod;
        pw[0] = 1;
        hash[0] = 0;
        for(int i = 0; i <= len; i ++)
        {
            pw[i] = 1ll * pw[i-1] * B % mod;
            hash[i] = (1ll * hash[i-1] * B + s[i] - 'a' + 1) % mod;
        }
    }
    int gethash(int l, int r)
    {
        int res = (hash[r] - 1ll * hash[l-1] * pw[r-l+1]) % mod;
        if( res < 0 ) res += mod;
        return res;
    }
}Hash,Hash1;
//双hash
char s[100005];
int main()
{
    scanf("%s", s+1);
    int slen = strlen( s + 1) ;
    Hash.init(s, slen, 173, MOD);
}
```

```
//*****************************************
// HashMap
//*****************************************
typedef unsigned long long ull;
const int HASH = 10007;
const int MAXN = 2010;
const int SEED = 13331;
struct HASHMAP
{
```

```
    int head[HASH], next[MAXN], size;
    ull state[MAXN];
    int f[MAXN];

    // val % HASH 的值相同的在一个链表里;
    // state记录的是该hash节点的原值。
    // f[i] 记录的是该hash节点的子串起点，相同子串记录的是最后一个串的起点;
    void init()
    {
        size = 0;
        memset(head, -1, sizeof(head));
    }
    int insert(ull val, int _id)
    {
        int h = val % HASH;
        for(int i = head[h]; i!=-1; i = next[i])
            if(val == state[i])
            {
                int tmp = f[i];
                f[i] = _id;
                return tmp;
            }
        f[size] = _id;
        state[size] = val;
        next[size] = head[h];
        head[h] = size++;
        return 0 ;
    }
}H;

ull p[MAXN];
ull s[MAXN];
char str[MAXN];
int ans[MAXN][MAXN];
int main()
{
    p[0] = 1;
    for(int i = 1; i < MAXN ; i++)
        p[i] = p[i-1] * SEED;

    scanf("%s", str);
    int n = strlen(str);
    s[0] = 0;
    for(int i = 1; i <= n; i++)
        s[i] = s[i-1] * SEED + str[i-1]; // SEED进制数
    memset(ans, 0, sizeof(ans));
    for(int L = 1; L <= n ; L++)
    {
        H.init();
        for(int i = 1; i+L-1 <= n ; i++)
        {
            int l = H.insert(s[i+L-1]-s[i-1]*p[L], i);
            ans[i][i+L-1] ++;
            ans[l][i+L-1] --;
            //如果有相同的，给新的区间加一个，给原来的减一个;
            //如果没有给0减一个;
        }
    }
    for(int i = n ; i >= 0; i--)
        for(int j = i; j <= n ;j++)
            ans[i][j] +=ans[i+1][j] + ans[i][j-1] - ans[i+1][j-1];
    //ans[u][v] 表示u，v区间内不同的字符串的个数;
    return 0;
}
```

1.8 数字快速读入

```
inline int Read()
{
  int x=0;char ch=getchar();bool positive=1;
  for (;ch<'0'||ch>'9';ch=getchar()) if (ch=='-') positive=0;
  for (;ch>='0'&&ch<='9';ch=getchar()) x=x*10+ch-'0';
  return positive?x:-x;
}

用法就是：  int x = read();
```

1.9 最小表示法

1.10 bitset 优化

Chapter 2

数据结构

2.1 邻接表 & 邻接矩阵

链式前向星是一种邻接表

```
//*******************************************************
// 链式前向星
//*******************************************************
struct Node
{
    int to, next;
    int w;
}edge[N];
int cnt,head[N];
void init()
{
    cnt = 0;
    memset(head, -1, sizeof(head));
}
void add(int u, int v, int w)
{
    edge[cnt].to = to;
    edge[cnt].w = w;
    edge[cnt].next = head[u];
    head[u] = cnt++;
}
//遍历方式
//起点为u
for(int i = head[u] ;i != -1; i=edge[i].next)
{
    int v = edge[i].to;
    .....
}
```

Vector 实现邻接表

```
//*******************************************************
// Vector
//*******************************************************
#include<vector>
struct Node
{
    int to,w;
};
vector<Node> edge[N];
//遍历方式
int len = edge[u].size();
for(int i = 0;i < len; i++)
{
    int v = edge[u][i].to;
}
```

2.2 STL

STL 模板库中有 Vector、Stack、Queue、Priority_Queue、List、Set、multiset、Map

2.2.1 Stack

一般我们会手写栈，STL 的栈的用法也十分的简单。手写就用数组模拟。
算法中还会有一种单调栈的东西，可以看做是一个框，两边可以移动，不断的移动边界直至结尾，过程中会处理出最优的结果。

```
stack.push(x);
stack.pop();
stack.top();
stack.size();
stack.empty();
stack.clear();
```

2.2.2 Queue

一般我们不手写队列。
优先队列默认是大根堆，优先集比较高的先出列。
注意：重载的时候要注意符号。

```
Q.push(x);
Q.pop();
Q.front();
Q.size();
Q.empty();
Q.clear();
priority\_queue 是 Q.top()取栈顶;
定义优先级的时候两种写法如下
struct cmp
{
    bool operator ()(int x, int y)
    {
        return  x > y; // x小的优先级高
        //也可以写成其他方式，如:  return p[x] > p[y];表示p[i]小的优先级高
    }
};
priority_queue<int, vector<int>, cmp> Q;
//其中，第二个参数为容器类型。第三个参数为比较函数。

struct node
{
    int x, y;
    bool operator < (const node & a)const
    {
        return x > a.x; //结构体中，x小的优先级高
    }
};
priority_queue<node> Q;//定义方法
```

2.2.3 Map

Map 是一种映射，键和值，键可以作为下标，值默认为 0。

```
map<string,int> Map;
Map[键] = 值;
tmp = Map[键];
Map.clear();

//找到
if( Map.find(键) ! = Map.end())

```

```
//迭代器遍历
map<string, int >::iterator it;
for(it = Map.begin(); it!=Map.end(); it++)
{
    int val = it->second;
}
```

2.2.4 Set

Set 的用法也很简单，就是集合。有一种多重集合，可以放重复元素在里面。
切记！使用迭代器删除的时候不能直接删除，要把原来的迭代器加 1 后再删除原来的迭代器指向的内容，否则会出现访问越界。或者是先记录下来要删除的元素，然后遍历完成后在循环删那些数字。

```
set<string> Set;
Set.insert(x);
Set.clear();
Set.erase(x);
Set.find(x);
Set.size();
Set.empty();
Set.lower_bound(x); //第一个大于等于x的定位器;
Set.upper_bound(x); //最后一个大于等于x的定位器;

multiset<int> mSet;
mSet.size();
mSet.count(x);
//可重复插入，统计个数的时候重复的也算，size也是。
```

2.2.5 Vector

```
vector<int> a;
a.push_back(x);
a.erase(iterator);
stack.size();
stack.empty();
stack.clear();
//遍历也是使用迭代器
```

2.2.6 Pair

```
pair<string,int> p;

make_pair : p = make_pair("abc", 5);

p.first 和 p.second
```

2.3 二叉搜索树

二叉查找树（Binary Search Tree），也称有序二叉树（ordered binary tree），排序二叉树（sorted binary tree），是指一棵空树或者具有下列性质的二叉树：
1. 若任意节点的左子树不空，则左子树上所有结点的值均小于它的根结点的值；
2. 任意节点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；
3. 任意节点的左、右子树也分别为二叉查找树。
4. 没有键值相等的节点（no duplicate nodes）。

2.3.1 Treap

2.3.2 Splay

Splay-伸展树，没有其他平衡树的冗余信息，通过树本身的旋转操作进行伸展，每次把需要调整的区间旋转到固定的子树上去，这里就是 root 的右儿子的左子树。

1. 树的根是 1 号节点，0 号节点是虚根。
2. 建树之前先建两个首尾结点，注意首尾结点内的值要特殊一点。相当于在数组的两端新加两个结点。
3. 如果我们对 $[l,r]$ 区间操作，我们把第 $l-1$ 结点调整到根的位置上，然后把 $r+1$ 结点调整到根的右儿子的位置，这样 $[l,r]$ 区间就是根的右儿子的左子树。需要注意的是，在树中我们多建了两个点，所以我们把树上第 l 个结点调整到根，把 $r+2$ 结点调整到根的右儿子。然后对整个子树进行操作，打 lazy 标记。
4. 一些其他的操作也是利用上面的这个思路，先把要操作的无论是区间还是单点都一样，调整到关键位置，然后对整个子树进行操作。
5. 用序列建树的时候下标从 1 开始存在 a 数组中。

```
/*************************************************************************
 > File Name: kb/splay-m.cpp
 > Author: Bai Yan
 > Created Time: 2015年10月04日 星期日 10时53分33秒
题目大意:
   给定一个序列；六种操作；
   1.add x y D;区间加值;
   2.REVERSE x y ; 区间翻转;
   3.REMOVE x y T; 循环右移T次;
   4.INSERT x P; 在第x个数后面插入p;
   5.DELETE x;删除第x个数;
   6.MIN x y:区间最值;
 ************************************************************************/

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

#define key_value spt[spt[root].ch[1]].ch[0]
#define ls spt[r].ch[0]
#define rs spt[r].ch[1]
#define fa spt[r].pre

using namespace std;

const int INF = 0x3fffffff;
const int N = 200010;
int a[N],n;

int root,tot1;
struct Node
{
   int pre,ch[2],key,size,add,rev,Min;
}spt[N];
int s[N],tot2;//内存池，内存池容量;

void NewNode(int &r,int father,int k)
{
   if(tot2)
      r=s[tot2--];
   else
      r=++tot1;
   spt[r].pre=father;
   spt[r].ch[0]=spt[r].ch[1]=0;
   spt[r].size=1;
   spt[r].key=spt[r].Min=k;
   spt[r].add=spt[r].rev=0;
}
//翻转的lazy;
void Update_rev(int r)
{
   if(r==0)
```

```
        return ;
    swap(spt[r].ch[0],spt[r].ch[1]);
    spt[r].rev^=1;
    return ;
}
//区间加值lazy:
void Update_add(int r,int add)
{
    if(r==0)
        return ;
    spt[r].add+=add;
    spt[r].key+=add;
    spt[r].Min+=add;
}

void Pushup(int r)
{
    spt[r].size= spt[ls].size+ spt[rs].size+ 1;
    spt[r].Min= spt[r].key;
    if(ls)
        spt[r].Min= min(spt[r].Min, spt[ls].Min);
    if(rs)
        spt[r].Min = min( spt[r].Min, spt[rs].Min);
}

void Pushdown(int r)
{
    if(spt[r].rev)
    {
        Update_rev(ls);
        Update_rev(rs);
        spt[r].rev=0;
    }
    if(spt[r].add)
    {
        Update_add(ls, spt[r].add);
        Update_add(rs, spt[r].add);
        spt[r].add=0;
    }
}
void build(int &r,int L, int R ,int father)
{
    //建一棵树, 按数列下标顺序;
    if(L>R)
        return ;
    int mid=(L+R)/2;
    NewNode(r,father,a[mid]);
    build(ls, L, mid-1, r);
    build(rs, mid+1, R, r);
    Pushup(r);
}
void init()
{
    root= tot1= tot2 =0;
    spt[root].ch[0]= spt[root].ch[1]= spt[root].size=0;
    spt[root].key= spt[root].add= spt[root].rev=spt[root].pre=0;
    spt[root].Min= INF ;//重要;
    NewNode(root , 0 ,INF);
    NewNode(spt[root].ch[1], root, INF);
    build(key_value, 1, n, spt[root].ch[1]);
    Pushup(spt[root].ch[1]);
    Pushup(root);
}
void Rotate(int x,int kind)
{
    int y=spt[x].pre;
    Pushdown(y);
    Pushdown(x);
    spt[y].ch[!kind] = spt[x].ch[kind];
    spt[spt[x].ch[kind]].pre= y;
    if(spt[y].pre)
```

```
    {
        int zfa=spt[y].pre;
        spt[zfa].ch[spt[zfa].ch[1] == y]= x;
    }
    spt[x].pre= spt[y].pre;
    spt[x].ch[kind]= y;
    spt[y].pre= x;
    Pushup(y);
}
void Splay(int r,int goal)
{
    Pushdown(r);
    while(fa != goal)
    {
        if(spt[fa].pre== goal)
        {
            Pushdown(fa);
            Pushdown(r);
            Rotate(r, spt[fa].ch[0]==r);
        }
        else
        {
            Pushdown(spt[fa].pre);
            Pushdown(fa);
            Pushdown(r);
            int zfa=spt[fa].pre;
            int kind= spt[zfa].ch[0]==fa ;
    //  printf("%d %d %d %d\n",zfa,fa,r,kind);
            if(spt[fa].ch[kind]== r)
            {
                Rotate(r, !kind);
                Rotate(r,kind);
            }
            else
            {
                Rotate(fa,kind);
                Rotate(r,kind);
            }
        }
    }
    Pushup(r);
    if(goal==0)
        root= r;
}
int get_kth(int r,int k)
{
    Pushdown(r);
    int t=spt[spt[r].ch[0]].size+1;
    if(t==k)
        return r;
    if(t>k)
        return get_kth(spt[r].ch[0],k);
    else
        return get_kth(spt[r].ch[1],k-t);
}
int get_min(int r)
{
    Pushdown(r);
    while(ls)
    {
        r=ls;
        Pushdown(r);
    }
    return r;
}
int get_max(int r)
{
    Pushdown(r);
    while(rs)
    {
        r=rs;
```

```
        Pushdown(r);
    }
    return r;
}
void add(int L,int R ,int D)
{

    int x=get_kth(root,L);
    Splay(x, 0);
    int y=get_kth(root,R+2);
    Splay(y, root);

    Update_add(key_value,D);
    Pushup(spt[root].ch[1]);
    Pushup(root);
}
void Reverse(int L,int R)
{
    Splay(get_kth(root,L), 0);
    Splay(get_kth(root,R+2), root);
    Update_rev(key_value);
    Pushup(spt[root].ch[1]);
    Pushup(root);
}
void Revolve(int l ,int r,int T)
{
    int len= r-l+1;
    T = (T%len+len)%len;
    if(T==0)
        return ;
    int c=r-T+1;
    Splay(get_kth(root, c),0);
    Splay(get_kth(root,r+2),root);
    int temp=key_value;
    key_value=0;//把这一段取出来;
    Pushup(spt[root].ch[1]);
    Pushup(root);//更新。
    Splay(get_kth(root,l),0);
    Splay(get_kth(root,l+1),root);//把左结点旋转至根的右儿子的左儿子;
    key_value=temp;
    spt[key_value].pre= spt[root].ch[1];
    Pushup(spt[root].ch[1]);
    Pushup(root);
}
void Insert(int x,int p)
{
    Splay(get_kth(root,x+1),0);//把x结点旋转成根;
    Splay(get_kth(root,x+2),root);//把x+1结点旋转成根的儿子;
    //根据二叉搜索树的性质，把连续两个数旋转成根和儿子关系的时候
    //必然儿子的左子树是空的，因为我没有介于两者之间的数;
    NewNode(key_value,spt[root].ch[1],p);
    Pushup(spt[root].ch[1]);
    Pushup(root);
}
void erase(int r)
{
    if(r)
    {
        //删掉的结点记录在内存池里，就是把一个空的结点放在这里;
        //当内存少的时候，而前面有删掉了很多结点的时候，可以重复利用;
        s[++tot2]=r;
        erase(spt[r].ch[0]);
        erase(spt[r].ch[1]);
    }
}
void Delete(int x)
{
    Splay(get_kth(root,x),0);
    Splay(get_kth(root,x+2),root);
    //把x旋转到根的右儿子的左儿子上，此时左儿子就一个元素
    //所以删掉整棵子树，也就是删掉这个点;
```

```
    erase(key_value);
    spt[key_value].pre=0;
    key_value=0;
    Pushup(spt[root].ch[1]);
    Pushup(root);
}
int Query(int l,int r)
{
    Splay(get_kth(root,l),0);
    Splay(get_kth(root,r+2),root);
    return spt[key_value].Min;
}
void print()
{
    printf("root:␣␣%d\n",root);
    for(int i=0;i<=tot1;i++)
    {
        printf("i:␣%2d␣␣pre:␣%2d␣␣ch[0]:␣%2d␣␣ch[1]:␣%2d␣␣key:␣%2d␣␣size:␣%2d\n",i,spt[i].pre,spt[i].ch[0],spt
            [i].ch[1],spt[i].key,spt[i].size);
    }
    printf("\n");
}

void check()
{
    for(int i=1;i<=tot1;i++)
    {
        printf("i:␣%2d␣␣␣%2d\n",i,get_kth(root,i));
    }
}
int main()
{
    char op[20];
    int q;
    while(scanf("%d",&n)==1)
    {
        for(int i=1;i<=n;i++)
            scanf("%d",&a[i]);
        init();
        scanf("%d",&q);
        getchar();
        while(q--)
        {
            int l,r,x;
            scanf("%s",op);
            if(op[0]=='A')
            {
                scanf("%d%d%d",&l,&r,&x);
                add(l,r,x);
            }
            else if(strcmp(op,"REVERSE")==0)
            {
                scanf("%d%d",&l,&r);
                Reverse(l,r);
            }
            else if(strcmp(op,"REVOLVE")==0)
            {
                scanf("%d%d%d",&l,&r,&x);
                Revolve(l,r,x);
            }
            else if(op[0]=='I')
            {
                scanf("%d%d",&l,&r);
                Insert(l,r);
            }
            else if(op[0]=='D')
            {
                scanf("%d",&x);
                Delete(x);
            }
            else
```

```
            {
                scanf("%d%d",&l,&r);
        //  cout<<"YES"<<endl;
                printf("%d\n",Query(l,r));
            }
        }
    }
    return 0;
}
```

2.3.3 替罪羊

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

using namespace std;

//*********************************************************
// 1.构造函数，不用初始化
// 2.Insert(x);
// 3.Rank(x);
// 4.Erase(x);
// 5.Erase_kth(k);
// 6.Kth(k);
// 7.BuildTree(a, 0, n);
//
// 可以求前驱（小于x的最大的数）和后继（大于x的最小到的数）
// 前驱：kth( Rank(x) - 1 );
// 后继：kth( Rank(x+1) );
//*********************************************************

const int MAXN = (100000 + 10);
const double alpha = 0.75;
struct Node
{
    Node * ch[2];
    int key, size, cover; // size为有效节点的数量，cover为节点总数量
    bool exist; // 是否存在（即是否被删除）
    void PushUp()
    {
        size = ch[0]->size + ch[1]->size + (int)exist;
        cover = ch[0]->cover + ch[1]->cover + 1;
    }
    bool isBad()
    { // 判断是否需要重构,左右子树是否平衡;
        return ((ch[0]->cover > cover * alpha + 5) ||
                (ch[1]->cover > cover * alpha + 5));
    }
};
struct STree
{
    protected:
        Node mem_poor[MAXN]; //内存池，直接分配好避免动态分配内存占用时间
        Node *tail, *root, *null; // 用null表示NULL的指针更方便，tail为内存分配指针，root为根
        Node *bc[MAXN]; int bc_top; // 储存被删除的节点的内存地址，分配时可以再利用这些地址

        Node * NewNode(int key)
        {
            //分配一个值为key的结点;
            Node * p = bc_top ? bc[--bc_top] : tail++;
            p->ch[0] = p->ch[1] = null;
            p->size = p->cover = 1;
            p->exist = true;
            p->key = key;
            return p;
```

```
    }

    void Travel(Node * p, vector<Node *>&v)
    {
        //对子树中根遍历，获得还没有被删除的序列；
        if (p == null) return;
        Travel(p->ch[0], v);
        if (p->exist) v.push_back(p); // 构建序列
        else bc[bc_top++] = p; // 回收
        Travel(p->ch[1], v);
    }

    Node * Divide(vector<Node *>&v, int l, int r)
    {
        //构建子树，左闭右开的区间。
        if (l >= r) return null;
        int mid = (l + r) >> 1;
        Node * p = v[mid];
        p->ch[0] = Divide(v, l, mid);
        p->ch[1] = Divide(v, mid + 1, r);
        p->PushUp(); // 自底向上维护，先维护子树
        return p;
    }

    void Rebuild(Node * &p)
    {
        //重构一个p为根的子树，先遍历一遍，在对序列重构；
        static vector<Node *>v;
        v.clear();
        Travel(p, v);
        p = Divide(v, 0, v.size());
    }

    Node ** Insert(Node *&p, int val)
    {
        //插入，返回指针的（指针）地址。
        //p是引用，是个指针；
        if (p == null)
        {
            p = NewNode(val);
            return &null;
        }
        else
        {
            p->size++; p->cover++;
            // 返回值储存需要重构的位置，若子树也需要重构，本节点开始也需要重构，以本节点为根重构
            Node ** res = Insert(p->ch[val >= p->key], val);
            if (p->isBad()) res = &p;
            return res;
            //返回最外层需要重构的结点；
        }
    }

    void Erase(Node *p, int id)
    {
        //删除第id个结点；
        p->size--;
        int offset = p->ch[0]->size + p->exist;
        //判断需要删的点在哪棵子树上。
        if (p->exist && id == offset)
        {
            p->exist = false;
            return;
        }
        else
        {
            if (id <= offset) Erase(p->ch[0], id);
            else Erase(p->ch[1], id - offset);
        }
    }

```

```
    Node **Insert(Node *&p, int id, int val)
    {
        if(p==null)
        {
            p = NewNode(val);
            return &null;
        }
        p->size++;
        p->counter++;
        Node **res;
        if(id<=p->ch[0]->size+p->exist)
            res = Insert(p->ch[0], id, val);
        else res = Insert(p->ch[1], id-p->ch[0]->size-p->exist, val);
        if(p->isBad()) res = &p;
        return res;
    }


public:

    void Init(void)
    {
        tail = mem_poor;
        null = tail++;
        null->ch[0] = null->ch[1] = null;
        null->cover = null->size = null->key = 0;
        root = null; bc_top = 0;
    }

    STree(void) { Init(); }

    void Insert(int val)
    {
        //在root子树中插入值为val的结点;
        Node ** p = Insert(root, val);
        if (*p != null) Rebuild(*p);
    }

    int Rank(int val)
    {
        Node * now = root;
        int ans = 1;
        while (now != null)
        { // 非递归求排名
            if (now->key >= val) now = now->ch[0];
            else
            {
                ans += now->ch[0]->size + now->exist;
                now = now->ch[1];
            }
        }
        return ans;
    }

    int Kth(int k)
    {
        Node * now = root;
        while (now != null)
        { // 非递归求第K大
            if (now->ch[0]->size + 1 == k && now->exist) return now->key;
            else if (now->ch[0]->size >= k) now = now->ch[0];
            else k -= now->ch[0]->size + now->exist, now = now->ch[1];
        }
    }

    void Erase(int k)
    { //删除值为k的数, 多个的话删除第一个;
        Erase(root, Rank(k));
        if (root->size < alpha * root->cover) Rebuild(root);
    }

```

```
        void Erase_kth(int k)
        {
            //删除第k个数;
            Erase(root, k);
            if (root->size < alpha * root->cover) Rebuild(root);
        }
    
    
    
    
    
    
        //用数组产生的二叉查找树，下标就是大小关系;
        //建树的时候按照正常建完全二叉树的过程去建就好了;
        //查找的时候，第i个数就是下标为i-1的数。（注意下标的问题，建树的时候是从0开始的）。
        //我们可以在区间的某个地方插入删除数。
        //
        Node * BuildTree(int *a, int l,int r)
        {
            //用数组建树，数组下标从0开始;
            //主函数调用 BuildTree(a , 0 , n);
            if(l >= r) return null;
            int mid = (l+r)/2;
            Node * p = NewNode(a[mid]);
            p->ch[0] = BuildTree(a, l , mid);
            p->ch[1] = BuildTree(a, mid+1, r);
            p->PushUp();
            return p;
        }
        void Insert(int id , int val)
        {
            //在指定位置插入值为val的结点;
            Node **p = Insert(root, id, val);
            if(*p != null ) Rebuild(*p);
        }
};
```

2.4 并查集

并查集的作用就是将集合合并，并且查找两个元素是不是在同一个集合里。不能做集合的分拆，所以如果有集合的分拆的工作就倒过来进行集合的合并，就可以使用并查集来做。
带权并查集可以记录集合内部元素之间的关系，比较典型的有 friend-enemy、石头剪刀布等。有这些关系的时候合并需要进行计算然后合并。

```
 //普通并查集
 //初始化要变成每个人是自己的父亲
int Find(int x)
{
    return x == fa[x] ? x : fa[x] = find(fa[x]);
}
void Union(int a,int b)
{
    fa[b] = fa[a];
}
```

种类并查集，每个结点都和父亲结点有个距离。当合并两个子树的时候只要将一颗树的根接在另个根上，同时计算两个根的距离，只修改根的距离。剩余子树上的距离留到 find 中去更新。下面是一个例题：Hdu3038

给一些区间，并且给出这个区间的和，按顺序从上往下，如果当前区间的和跟前面的信息冲突就算是错的，就不要了。问，有多少错误的信息。
这里有一个处理技巧，就是区间的问题，给的区间是闭区间，如果给了 $[2,6]$ 和 $[7,10]$，这两个区间是连在一起的，但是这样写就是断开了，我们把左区间统一减一，变成左开右闭的，这样就是 $(1,6]$ 和 $(6,10]$，就可以合并了。

这里在合并的时候如何计算根与根的距离,举个例子,已知区间 $sum[2,3]=5$, $sum[7,8]=2$,现在合并 $[2,7]=10$, 我们假设现在大的是根, 那么就是要计算 $sum[3,8]$, 公式就是 $sum[3,8]=sum[2,7]+sum[7,8]-sum[2,3]$。

代码如下:

```
int find(int x)
{
   if(fa[x]!=x)
   {
      int f=fa[x];
      fa[x]=find(fa[x]);
      sum[x]+=sum[f];
   }
   return fa[x];
}
main()
{
      scanf("%d%d%d",&l , &r , &s);
      l--;
      int fal = find(l) ,far = find(r);
      if(fal==far)
      {
         if( sum[l] - sum[r] != s )
             ans ++;
      }
      else
      {
        fa[fal] = far;
        sum[fal] = sum[r] - sum[l] + s;
      }
}
```

食物链并查集，集合中所有的元素都跟其他元素有吃与被吃的关系，事实上也是给每个元素记一个种类编号，只不过我们一般使用相对编号，不能确定的表示是哪一种，但是可以表示出元素的关系。我们使用向量的思维去思考这个问题。箭头的方向表示吃与被吃的关系，就可以唯一表示了。

看代码:

```
/*
   动物王国中有三类动物A,B,C，这三类动物的食物链构成了有趣的环形。
   A吃B， B吃C，C吃A。
   现有N个动物，以1－N编号。每个动物都是A,B,C中的一种，但是我们并
   不知道它到底是哪一种。
   有人用两种说法对这N个动物所构成的食物链关系进行描述：
   第一种说法是"1 X Y"，表示X和Y是同类。
   第二种说法是"2 X Y"，表示X吃Y。

   此人对N个动物，用上述两种说法，一句接一句地说出K句话，这K句话
   有的是真的，有的是假的。当一句话满足下列三条之一时，这句话就是
   假话，否则就是真话。
   1）  当前的话与前面的某些真的话冲突，就是假话；
   2）  当前的话中X或Y比N大，就是假话；
   3）  当前的话表示X吃X，就是假话。
   你的任务是根据给定的N（1 <= N <= 50,000）和K句话
   （0 <= K <= 100,000），输出假话的总数。

 */
#include<iostream>
#include<cstdio>
#include<algorithm>
using namespace std;
int fa[50010]={0};
int len[50010]={0};
int n,m;
int find(int x)
{
   if(fa[x]!=x)
   {
      int pre=fa[x];
      fa[x]=find(fa[x]);
      len[x]=(len[x]+len[pre])%3;
```

```
    }
    return fa[x];
}

int main()
{
    scanf("%d%d",&n,&m);
        int ans=0;
        for(int i=0;i<=n;i++) fa[i]=i,len[i]=0;
        while(m--)
        {
            int d,x,y;
            scanf("%d%d%d",&d,&x,&y);
            if(x>n||y>n)
                ans++;
            else if(x==y&&d==2)
                ans++;
            else
            {
                //0表示xy同类;
                //1表示根吃结点;
                //2表示结点吃根;
                //len[x]表示: fa[x]->x;方向一定要注意;
                int fx=find(x),fy=find(y);
                if(fx==fy)
                {

                    if(d==2&&(len[y]-len[x]+3)%3!=1)
                        ans++;
                    //在同一个集合里, 与根节点的偏移量相等就是同类;
                    else if(d==1&&len[x]!=len[y])
                        ans++;
                }
                else
                {
                    fa[fy]=fx;
                    len[fy]=(len[x]-len[y]+d-1+3)%3;
                    //向量思维;
                }
            }
        }
        printf("%d\n",ans);

    return 0;
}
```

friends-enemy 并查集，集合中有一些朋友和敌人的关系，我们使用和根相同的值是朋友，否则是敌人，的方式来表示。

```
int find(int x)
{
    if(x!=fa[x])
    {
        int pre=fa[x];
        fa[x]=find(fa[x]);
        val[x]^=val[pre];//**
        //friend-enemy并查集;
    }
    return fa[x];
}
scanf("%d%d␣%s",&a,&b,s);
int f1 = find(a),f2 = find(b);
if(f1==f2) continue;//因为题目本身保证没有冲突;
fa[f2]=f1;

//friend-enemy 并查集;
//val数组记录的是与根的关系;
//与根的值相同的是根的朋友, 不同的是根的敌人;
if(s[0] == 'y')
    val[f2] = val[a] ^ val[b] ^ 0;
```

```
//val初始都是0, 所以0^0=0, 1^0=1;
else
    val[f2] = val[a] ^ val[b] ^ 1;
```

2.5 离散化

```
//离散化;
for(int i=0;i<n;i++)
    t[i] = a[i];

sort(t, t+n);

n = unique(t, t+n) - t;

int pos = lower_bound(t, t+n, a[i]) - t;

pos 就是离散化后的位置。
```

2.6 树状数组

树状数组（BIT）是一种区间树，但是不是完全的区间树。它的结构支持两种区间，从头到某位置，或是从某位置到末尾。利用区间容斥的性质可以做到区间查询和修改的操作。树状数最难的地方是设计出利用树状数组的方法，本身的数据结构算法并不难。

这方面的例题有：逆序对问题、区间修改查询问题，二维树状数组问题等，其中比较典型的就是逆序对问题，求在某个数之后比某个值小的值的个数。

有几个问题需要注意。树状数组中 0 下标是不用的，所以向前求和的时候 $i > 0$，向后更新的时候 $i <= n$，

```
//*****************************************************
// 1. 区间求和
// 2. 单点加值
// 改点求段
//*****************************************************
int C[N];
int Lowbit(int x)
{
    return x&(-x);
}
int getSum(int x)
{
    int ans = 0;
    for(; x > 0; x-=Lowbit(x) )
        ans += C[x];
    return ans;
}
void Update(int x, int val)
{
    for(; x <= MAXN; x+=Lowbit(x))
        C[x] += val;
}
```

题目描述：

给一个序列，每次给一段区间加上一个值，然后查询某个点的值。

我们用一个数组 $C[i]$ 表示 i 到 MAXN 被加了多少。区间修改的时候修改区间两头就可以。即修改区间 $[a,b]$，就是让 $C[a]++$ ，同时 $C[b+1]--$；此时前缀和就是该点的值。注意树状数组中存放的是改动了多少，要加上原来的值。初始化为 0；

题目升级，查询某一段区间的和。

一般我们都是先求出前缀和，然后相减。求前缀和的时候，公式是这样子的，计算每个位置的贡献。求前缀和的时候每个位置的值又是他的前缀和，所以每个位置的贡献不一样。从前往后递减。最后的公式是：$(x+1)*sigma(C[i])-sigma(C[i]*i)$，我们建两个树状数组，分别计算 $C[i]$ 的和、$C[i]*i$ 的前缀和，然后减掉。

二维树状数组：现在一个矩阵中，有两种操作，一种是选定子矩阵，给每个元素都加上一个值；第二种操作是求某一点的值。

这就是一个二维的树状数组可以解决的问题，我们相当于建立 $log(n)$ 棵树状数组，每次修改的时候横纵都要修改。

```
void Update(int x,int y, int val)
{
   for(int i = x; i <= MAXN; i+=Lowbit(i))
      for(int j = y; j <= MAXN; j+=Lowbit(j))
         C[i][j] += val;
}
int getSum(int x,int y)
{
   int sum = 0;
   for(int i = x; i > 0; i-=Lowbit(i))
      for(int j = y; j > 0; j-=Lowbit(j))
         sum += C[i][j];
   return sum ;
}
```

求区间最值的代码在 RMQ 里面。

对于二维树状数组的子矩阵求和，一样不是一棵两棵线段树能搞定的。需要四棵线段树。对于求子矩阵和。例如，我们现在要求 (1,1)-(x,y) 这个子矩阵的和，我们对于树状数组需要的公式就是：

设：

Sum 是二维树状数组求和函数；

a[i][j] 是树状数组的每个结点的值。

则：子矩阵和 = $\sum_i^x \sum_j^y Sum[i][j]$;

这里的 Sum[i][j] 就是树状数组中的前 i，j 个的和。那么树状数组每个点的贡献就很好算了。

$\sum_i^x \sum_j^y a[i][j] * (x-i+1) * (y-j+1)$

$(x+1)*(y+1)*\sum_i^x \sum_j^y a[i][j] + (y+1)*\sum_i^x \sum_j^y a[i][j]*i + (x+1)*\sum_i^x \sum_j^y a[i][j]*j + \sum_i^x \sum_j^y a[i][j]*i*j$

上面的四项就是四棵树状数组的记录内容，注意前面的系数只有在查询的时候才会跟上，更新的时候要更新的数是修改量和乘的数。

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

  二维树状数组，  区间求和。

  题目中每个点有50个值，我们压缩到一个longlong中，因为只有0-1.
  每次区间更新的时候，我们要更新四个地方。
  对于二维矩阵的树状数组，我们记录的是从（1,1）到当前(i, j)的和。
  所以我们区间更新四个位置（x1，y1），（x1,y2+1）,(x2+1,y1)，(x2+1, y2+1);

  求和的时候从以为区间求和我们知道，推广到二维矩阵中，我们的求和就需要对应的四个量，所以我们用四颗树状数组来记录。

  更新的时候，每次区间更新更新四个位置，每个位置我们有四个值，重点就在于四个值更新的 时候。按照递推式中的量。


 **************************************************************************/
#include<iostream>
#include<cstdio>
#include<cstring>
#include<algorithm>
using namespace std;
typedef long long ll;
int n,m;

ll C[4][3010][3010]={0};
int Lowbit(int x)
{
   return x&(-x);
}

void Add(int t, int x,int y, ll val)
```

```
{
    for(int i = x; i <= n; i+=Lowbit(i))
        for(int j = y; j <= n; j+=Lowbit(j))
        {

            C[t][i][j] ^= val;
        }
}

ll getSum(int t, int x,int y)
{
    ll sum = 0;
    for(int i = x; i > 0; i-=Lowbit(i))
        for(int j = y; j > 0; j-=Lowbit(j))
        {
            sum ^= C[t][i][j];
        }
    return sum;
}

void Update(int x, int y, ll val)
{
    Add(0, x,y, val);
    if(x%2==1) Add(1, x, y, val);
    if(y%2==1) Add(2, x, y, val);
    if(x*y%2==1) Add(3, x,y, val);
}

void Update(int x1, int y1, int x2, int y2, ll val)
{
    Update(x1,y1,val);
    Update(x1,y2+1, val);
    Update(x2+1, y1, val);
    Update(x2+1, y2+1, val);
}
ll Query(int x,int y)
{
    int a[4]={0};
    if((x+1)*(y+1)%2==1) a[0] = 1;
    if((y+1)%2==1) a[1]=1;
    if((x+1)%2==1) a[2]=1;
    a[3]=1;
    ll tmp=0;
    for(int i=0;i<4;i++)
        if(a[i]==1)
            tmp ^= getSum(i,x,y);
    return tmp;
}
ll Query(int x1, int y1, int x2, int y2)
{
    ll tmp=0;
    tmp ^= Query(x2, y2);
    tmp ^= Query(x1-1,y2);
    tmp ^= Query(x2,y1-1);
    tmp ^= Query(x1-1,y1-1);
    return tmp;
}
int main()
{
    scanf("%d%d",&n,&m);
        memset(C,0,sizeof(C));
        while(m--)
        {
            char s[10];
            scanf("%s",s);
            int x1,y1,x2,y2;
            scanf("%d%d%d%d",&x1,&y1,&x2,&y2);
            if(s[0]=='P')
            {
                int k;
                scanf("%d",&k);
```

```
            ll tmp=0;
            while(k--)
            {
                int a, b;
                scanf("%d%d",&a,&b);
                if(b%2==1) tmp ^= (1ll<<a);
            }
            Update(x1,y1,x2,y2,tmp);
        }
        else
        {
            ll tmp = Query(x1, y1, x2, y2);
            for(int i=1;i<=50;i++)
            {
                int ans=1;
                if( 1ll*(tmp&(1ll<<i)) > 0 )
                    ans=2;
                printf("%d␣", ans);
            }
            printf("\n");
        }
    }
  return 0;
}
```

2.7 线段树

线段树是完全的区间树，可以对区间进行精确的操作。线段树是很多数据结构算法的基础。线段树一般使用是用于区间操作。另一种用法是用来加速实现扫描线算法。

线段树在使用的时候经常和“离散化”、“DFS 序”、“扫描线”等结合使用。

离散化前面已经讲过了。DFS 序一般用于处理树上的区间问题，将一颗子树转换成一段区间，对子树的操作就是对一段区间操作了。

我的线段树风格一般都是从 1-n 的区间。下标 0，尽量不用。

```
//
// 题意： 区间加值的模板。
// 本题是要求给一写互不重叠的矩形，求竖着划线分两半，使左右矩形面积一样。如果做不到就让左边的面积比右边的大，且差值最小。求分割线。
//
using namespace std;
typedef long long ll;
const int N = 1000005;
struct Node
{
    int l, r;
    ll sum;
    ll lazy;
}tr[N<<2];

void Build(int rt,int l,int r)
{
    tr[rt].l = l;
    tr[rt].r = r;
    tr[rt].sum = tr[rt].lazy = 0;
    if(l==r)
        return ;
    int mid = (l+r)/2;
    Build(rt<<1, l, mid);
    Build(rt<<1|1, mid+1, r);
}
void Pushup(int rt)
{
    if(tr[rt].l == tr[rt].r)
        return ;
    int lson = rt<<1;
    int rson = rt<<1|1;
```

```
    tr[rt].sum = tr[lson].sum + tr[rson].sum;
    return ;
}
void Pushdown(int rt)
{
    if(tr[rt].l == tr[rt].r)
        return ;
    int lson = rt<<1;
    int rson = rt<<1|1;
    if(tr[rt].lazy==0)
        return ;
    tr[lson].lazy += tr[rt].lazy;
    tr[rson].lazy += tr[rt].lazy;
    tr[lson].sum += tr[rt].lazy*(tr[lson].r-tr[lson].l+1);
    tr[rson].sum += tr[rt].lazy*(tr[rson].r-tr[rson].l+1);
    tr[rt].lazy = 0;
    return ;
}
void Update(int rt, int l, int r, ll val)
{
    if(tr[rt].l == l && tr[rt].r==r)
    {
        tr[rt].sum += val*(r-l+1);
        tr[rt].lazy += val;
        return ;
    }
    Pushdown(rt);
    int lson = rt<<1;
    int rson = rt<<1|1;
    if( r <= tr[lson].r )
        Update(lson, l, r, val);
    else if(l >= tr[rson].l)
        Update(rson, l, r , val);
    else
    {
        Update(lson, l , tr[lson].r, val);
        Update(rson, tr[rson].l, r, val);
    }
    Pushup(rt);
}
ll Sum(int rt, int l, int r)
{
    if(l>r)
        return 0;
    if(tr[rt].l == l && tr[rt].r == r)
        return tr[rt].sum;
    Pushdown(rt);
    int lson = rt<<1;
    int rson = rt<<1|1;
    ll ans = 0;
    if( r <= tr[lson].r )
        ans = Sum(lson, l, r);
    else if(l>=tr[rson].l)
        ans = Sum(rson, l, r);
    else ans = Sum(lson, l, tr[lson].r)+Sum(rson, tr[rson].l, r);
    Pushup(rt);
    return ans;
}
int main()
{
    int T;
    scanf("%d", &T);
    while(T--)
    {
        int R;
        int n;
        scanf("%d%d", &R, &n);
        memset(tr, 0, sizeof(tr));
        Build(1, 1, R);
        while(n--)
        {
```

```
            int x, y, w, h;
            scanf("%d%d%d%d", &x, &y, &w,&h);
            Update(1, x+1, x+w, 1ll*h);
        }
        int l = 0, r = R;
        while( l <= r )
        {
            int mid = (l+r)/2;
            ll suml = Sum(1, 1 , mid);
            ll sumr = Sum(1, mid+1, R);
            if( suml > sumr )
            {
                r = mid-1;
            }
            else l = mid+1;
        }
        int ans = r;
        ll Min = 100000000000000;
        for(int i=-2;r+i<=R; i++)
        {
            int suml = Sum(1, 1, r+i);
            int sumr = Sum(1,r+i+1 ,R);
            if( suml-sumr > Min )
                break;
            if(suml < sumr)
                continue;
            if(suml-sumr <= Min)
            {
                ans = r + i;
                Min = suml-sumr;
            }
        }
        printf("%d\n",ans);
    }
    return 0;
}
```

线段树区间合并。

```
/*
  hdu-1540 题意：一个线段，长度为n，三种操作，Dx，挖掉某个点；R，恢复最近被挖掉的点；Qx查询该点所在的连续区间的长度；
  树的节点维护三个变量，该节点左边界开始连续的个数ll，右边界开始连续的个数rl，（在该区间内），
       该区间内最大的连续区间的长度ml；
  最后一个变量是为了方便判断，主要的还是前两个；
  修改的时候先是深入单点修改单点信息，然后回溯上来更父结点，更新的时候有点复杂；
  查询的时候借助ml的长度减少计算量，快速返回；
  */
#include<cstdio>
#include<iostream>
#include<cstring>
#include<stack>
#include<algorithm>
using namespace std;
struct Node
{
    int l,r,ll,rl,ml;
}tr[200005];
void build(int rt,int l,int r)
{
    tr[rt].l=l;
    tr[rt].r=r;
    tr[rt].ll=tr[rt].rl=tr[rt].ml=r-l+1;
    if(l==r)
        return ;
    int mid=(l+r)/2;
    build(rt<<1,l,mid);
    build(rt<<1|1,mid+1,r);
}
void Update(int rt,int l,int r,int t,int val)//t是要改的点；
{
    if(l==r)        //找到单点；
```

```
    {
        if(val==1)
        {
            tr[rt].ll=tr[rt].rl=tr[rt].ml=1;//改单点的信息;
        }
        else
        {
            tr[rt].ll=tr[rt].rl=tr[rt].ml=0;
        }
        return ;
    }
    int mid=(l+r)/2;
    if(t<=mid)
    {
        Update(rt<<1,l,mid,t,val);
    }
    else
    {
        Update(rt<<1|1,mid+1,r,t,val);
    }
    tr[rt].ll=tr[rt<<1].ll;          //左子树的左侧边界的值给父节点;
    tr[rt].rl=tr[rt<<1|1].rl;
    tr[rt].ml=max(tr[rt<<1].ml,tr[rt<<1|1].ml);     //左右子树的最大连续数;
    tr[rt].ml=max(tr[rt].ml,tr[rt<<1].rl+tr[rt<<1|1].ll);  //两个子树的交接处的最大连续数;
    if(tr[rt<<1].ll==mid-l+1)
        tr[rt].ll+=tr[rt<<1|1].ll;        //如果左子树是全连续的，就更新父节点的左边界最大连续数;
    if(tr[rt<<1|1].rl==r-mid)
        tr[rt].rl+=tr[rt<<1].rl;
}
int Query(int rt,int l,int r,int t)
{
    if(l==r||tr[rt].ml==0||tr[rt].ml==r-l+1) //如果找到单点，或该区间全空，或该区间全满；就返回;
        return tr[rt].ml;
    int mid=(l+r)/2;
    if(t<=mid) //如果该点在左子树上
    {
        if(t>=tr[rt<<1].r-tr[rt<<1].rl+1) //如果该点是在子树右边界点连续范围,
             就进入左子树继续搜索该点并加上搜索右子树的左边界那个点的结果;
            return Query(rt<<1,l,mid,t)+Query((rt<<1)|1,mid+1,r,mid+1);
        else
            return Query(rt<<1,l,mid,t);
                                        //如果不再右连续块内就进左子树继续搜索;
    }
    else
    {
        if(t<=tr[rt<<1|1].l+tr[rt<<1|1].ll-1)
            return Query((rt<<1|1),mid+1,r,t)+Query(rt<<1,l,mid,mid);
        else
            return Query(rt<<1|1,mid+1,r,t);
    }
}

int main()
{
    int n,m;
    while(scanf("%d%d",&n,&m)!=EOF&&n&&m)
    {
        memset(tr,0,sizeof(tr));
        stack<int > z;
        build(1,1,n);
        for(int i=0;i<m;i++)
        {
/*   for(int j=1;j<15;j++)
            {
                printf("%2d: l:%d r:%d ll:%d rl:%d ml:%d\n",j,tr[j].l,tr[j].r,tr[j].ll,tr[j].rl,tr[j].ml);
            }
            printf("\n\n\n");
*/   char a[3];
            scanf("%s",a);
            if(a[0]=='D')
            {
```

```
            int x;
            scanf("%d",&x);
            z.push(x);
            Update(1,1,n,x,0); //0代表删除操作;
        }
        else if(a[0]=='R')
        {
            if(!z.empty())
            {
                int x=z.top();
                z.pop();
                Update(1,1,n,x,1); //1代表回复操作;
            }
        }
        else if(a[0]=='Q')
        {
            int x;
            scanf("%d",&x);
            int ans=Query(1,1,n,x);
            printf("%d\n",ans);
        }
    }
  }
  return 0;
}
```

线段树，DFS 序结合。

```
/*
  hdu3974
  dfs序建树，然后区间修改查询;
*/
#include<iostream>
#include<cstdio>
#include<cstring>
#include<algorithm>
#define MAX_N 50005
using namespace std;

int N,M;
struct Node
{
    int to,next;
}edge[MAX_N];
struct TREE
{//val记录的是区间;
    int l,r,val,lazy;
}tr[MAX_N*4];
int head[MAX_N],cnt,tot;
int Start[MAX_N],End[MAX_N];

void Addedge(int u,int v)//链式前向星存图;
{
    //tot是边的编号;
    edge[tot].to=u;//以v为起点的边；终点是u;
    edge[tot].next=head[v];//这条边的下一条边是几号边，依然是v为起点的边;
    head[v]=tot++;//更新v为起点的边的入口;
}
void dfs(int x)
{
    cnt++;//初始为0；这里是1了，从1开始编号;
    Start[x]=cnt;//起点是x的编号
    for(int i=head[x];i!=-1;i=edge[i].next)//遍历以v为起点的边;
    {
        dfs(edge[i].to);
    }
    End[x]=cnt;//遍历完子树后可以得到这个树根的编号范围就是star&end;
    //dfs先根遍历，把x为根的子树的所有的节点编号在一段区间内，连续编号，并且记录该区间的编号起始和结束，
        当线段树修改值的时候就直接修改该区间;
}
void build(int rt,int l,int r)
```

```
{
    tr[rt].l=l;
    tr[rt].r=r;
    tr[rt].val=-1;
    tr[rt].lazy=0;
    if(l==r)
        return;
    int mid=(l+r)/2;
    build(rt<<1,l,mid);
    build(rt<<1|1,mid+1,r);
}
void Pushdown(int rt)
{
    int ls=rt<<1,rs=rt<<1|1;
    if(tr[rt].lazy)
    {
        tr[rs].val=tr[ls].val=tr[rt].val;
        tr[rs].lazy=tr[ls].lazy=1;
        tr[rt].lazy=0;
    }
}
void Update(int rt,int l,int r,int x)
{
    if(tr[rt].l==l&&tr[rt].r==r)
    {
        tr[rt].val=x;
        tr[rt].lazy=1;
        return ;
    }
    Pushdown(rt);
    int ls=rt<<1,rs=rt<<1|1;
    if(l<=tr[ls].r)
    {
        if(r<=tr[ls].r)
            Update(ls,l,r,x);
        else
            Update(ls,l,tr[ls].r,x);
    }
    if(r>=tr[rs].l)
    {
        if(l>=tr[rs].l)
            Update(rs,l,r,x);
        else
            Update(rs,tr[rs].l,r,x);
    }
}
int Query(int rt,int a)
{
    if(tr[rt].l==a&&tr[rt].r==a)
    {
        return tr[rt].val;
    }
    Pushdown(rt);
    int mid=(tr[rt].l+tr[rt].r)/2;
    if(a<=mid)
        return Query(rt<<1,a);
    else
        return Query(rt<<1|1,a);
}
int main()
{
    int T,k=1;
    cin>>T;
    while(T--)
    {
        printf("Case␣#%d:\n",k++);
        cnt=0;
        tot=0;
        memset(edge,0,sizeof(edge));
        memset(head,-1,sizeof(head));
        memset(Start,-1,sizeof(Start));
```

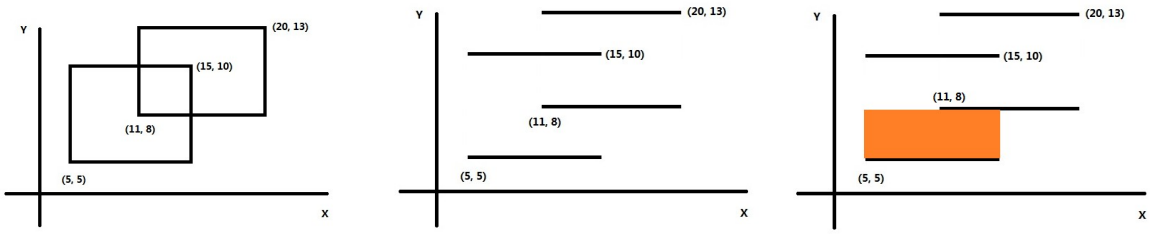
```
        memset(End,-1,sizeof(End));
        bool used[MAX_N];
        memset(used,false,sizeof(used));
        cin>>N;
        for(int i=0;i<N-1;i++)
        {
            int u,v;
            cin>>u>>v;//v是u的上级;
            used[u]=true ;//表示u是有上级的，也就是有入度的;
            Addedge(u,v);
        }
        for(int i=1;i<=N;i++)
        {
            if(!used[i])//找到入度为0的点，就是整个树的入口;
            {
                dfs(i);
                break;
            }
        }
        build(1,1,N);
        cin>>M;
        for(int i=0;i<M;i++)
        {
            char a[10];
            scanf("%s",a);
            if(a[0]=='C')
            {
                int x;
                cin>>x;
                int ans=Query(1,Start[x]);//该点的编号就是起点的编号;
                cout<<ans<<endl;
            }
            else
            {
                int a,b;
                cin>>a>>b;
                Update(1,Start[a],End[a],b);
            }
        }

    }
    return 0;
}
```

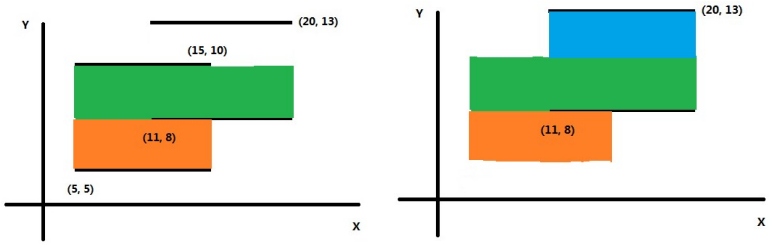
扫描线，求矩形面积并。

扫描线算法可以使用在各种面积并中，不是矩形也可以做，就是求的时候会稍微有点麻烦，比如白书中的圆形的面积并，需要求弓形的面积和四边形面积。

下面说一下扫描线算法的思路。

对于矩形并我们的处理就是分割，然后再一个一个求，那么怎么分割呢。由于是矩形，所以我们很容易想到按矩形的边划分，当然我们这里处理的是边与坐标轴平行的情况。扫描线可以横向扫也可以纵向扫，我们以纵向从下网上扫为例讲一下思路。





从上面的图我们可以看到，首先我们把每一个横边都记录下来。并且在记录的时候还要记录这条横边是上边界还是下边界。然后我们用一条线，从下往上扫，扫的时候我们维护一课线段树。线段树相当于是 x 轴的一个区间。初始化这个线段树为 0。扫描线刚学的时候这里容易理解不清楚。我们事实上就是把矩形用横线分割开，那么我们从下往上一个一个的计算面积。计算面积的时候，小矩形的长（横边）我们需要知道，宽可以用分割线间距离计算出来。但是分割后的矩形，在两个分割线的之间的部分不一定是连续的。我们可能需要扫一遍，肯定太慢了。所以我们用线段树直接求个区间和就好啦。所以知道线段树的妙用了吧，遇到下边界的时候要给区间加 1，遇到上边界要给区间减 1。这样就可以求啦。

下面说一下注意点：

1. 对于线段树记录的 x 轴的坐标，可以离散化。线段树中记录的长度是正确的就可以了。但是区间加法的时候也要对应加到离散化后的区间。如果是浮点数就一定要离散化了。

2. 线段树记录这种边长的时候，要注意，我们建的树不再是简单的点树了。之前我们建树的时候叶子结点是建到 $l == r$，这种建树的结果实际上是点树。记录几何线段的时候我们就得建线段树。其实原理都是一样的，你的叶子结点就是你的最小的单位，现在最小的单位变成了 $[l, l+1]$，所以我们的线段树的叶子结点就是一个单位为 1 的小线段。当然在离散化后，这个小线段有他实际的长度。但是也是最小的单位。

在这里我贴的代码是一个很厉害的高中生写的，代码十分的精简，甚至是没有建树的过程。代码的精简会让编码时间减小，但是这个代码中没有 lazy，事实上是牺牲了一定的运行时间，来提高编码时间。比赛的时候在时间足够的情况下不建议这么做。但是能够做到这个代码能力的人，说明他对于这个数据结构是十分透彻的理解，直接操作数组，而不依赖于树的边的关系。

3. 记录横边的时候，要记录两个东西。一个就是分割线。分割线很明显需要排序去重。第二个就是原来的横边，需要记录边的起止位置，和上下边。

```
#include<iostream>
#include<cstring>
#include<algorithm>
#include<cstdio>
using namespace std;
struct data
{
    double x1,x2,y;
    int flag;
}a[801];

double hash[201];
double sum[801];
int col[801];

inline bool cp(data a,data b)
{
    return a.y < b.y;
}

void pushup(int size,int l,int r)
{
    if(col[size])
        sum[size] = hash[r+1] - hash[l];
    else if
        ( l == r) sum[size] = 0;
    else
        sum[size] = sum[size*2] + sum[size*2+1];
}
void update(int L, int R, int flag, int l, int r, int size)
{
    if( L <= l && R >= r )
    {
        col[size] += flag;
```

```
        pushup( size, l, r );
        return;
    }
    int m = ( l + r ) / 2;
    if( L <= m)
        update( L, R, flag, l, m, size * 2 );
    if( R > m)
        update( L, R, flag, m+1, r, size * 2 + 1);
    pushup( size, l, r);
}
int main()
{
    int n;
    while(cin>>n)
    {
        if(n==0)break;
        double x1,y1,x2,y2;
        for(int i=1;i<=n;i++)
        {
            scanf("%lf%lf%lf%lf",&x1,&y1,&x2,&y2);
            a[2*i-1].x1 = a[2*i].x1=x1;
            a[2*i-1].x2 = a[2*i].x2=x2;
            a[2*i-1].y = y1;
            a[2*i].y = y2;
            a[2*i-1].flag = 1;
            a[2*i].flag=-1;
            hash[2*i-1]=x1;
            hash[2*i]=x2;
        }
        sort( a+1, a+2*n+1, cp);
        sort( hash+1, hash+2*n+1);
        int cnt = unique(hash+1, hash+2*n+1) - hash;
        memset(col,0,sizeof(col));
        memset(sum,0,sizeof(sum));
        double ans=0;
        for(int i=1;i<=2*n;i++)
        {
            int l=lower_bound( hash+1, hash+cnt, a[i].x1 ) - hash;
            int r=lower_bound( hash+1, hash+cnt, a[i].x2 ) - hash - 1;
            //求区间的时候，注意，右区间事实上减1;
            //因为树是默认 l 到 l+1 区间的。右边界必须减;
            if( l <= r )
                update( l , r, a[i].flag, 1, cnt, 1 );
            ans += sum[1] * ( a[i+1].y - a[i].y );
        }
        printf("%.2lf\n",ans);
    }
    return 0;
}
```

扫描线求矩形周长并。

求矩形的周长的并的时候事实上也是扫描线的基础的做法。有两种做法，第一种就是上面的那种，当我们求矩形的面积交的时候事实上就是先求出了长度，然后乘以高度。这里我们不用乘以高度，所以第一种做法就很简单了，就是横着做一遍再竖着做一遍，就好了。但是要注意的是我们计算边长的时候要计算这一次和上一次的差值的绝对值，才是新增的边长。

另一种做法是一遍做完。这样横着的做法还是不变的，竖着的边要在计算横边的时候一起计算出来。计算的方法比较简单，就是在分割后的一行里，有几段间隔就有几对竖边。所以我们需要对横边进行区间合并，求出有多少分隔个区间，然后就用数量乘高度就是所有的竖边的长度了。

再贴一个别人的板子，这个写的更简洁。

这个板子里，对于区间合并只需要判断接缝处是否是连续的就可以了。

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <algorithm>
using namespace std;
#define N 10010

```

```cpp
struct note
{
    int a,b,h,s;
    note(){}
    note(int l,int r,int c,int d):a(l) , b(r) , h(c) , s(d) {}
}data[N];

///线段树
#define lson l ,m ,rt<<1
#define rson m+1,r,rt<<1|1
#define fmid (l+r)>>1

#define M 22222
int len[M<<2],memseg[M<<2],lbd[M<<2],rbd[M<<2],cnt[M<<2];

void Push_up(int rt,int l,int r)
{
    if(cnt[rt])
    {
        lbd[rt] = rbd[rt] = 1;
        memseg[rt] = 2;
        len[rt] = r-l+1;
    }else if(l == r) lbd[rt] = rbd[rt] = memseg[rt] = len[rt] = 0;
    else
    {
        lbd[rt] = lbd[rt<<1];
        rbd[rt] = rbd[rt<<1|1];
        len[rt] = len[rt<<1] + len[rt<<1|1];
        memseg[rt] = memseg[rt<<1] + memseg[rt<<1|1];
        if(rbd[rt<<1] && lbd[rt<<1|1] ) memseg[rt] -= 2;
    }
}
/**
*矩形有个特点，已经出现的更新，必然会再次出现，这样的话，延迟标记也就可以不用push_down()!
*/
void update(int lx,int rx,int s,int l,int r,int rt)
{
    if(lx <= l && rx >= r)
    {
        cnt[rt] += s;
        Push_up(rt,l,r);
        return;
    }
    int m = fmid;
    if(lx <= m) update(lx,rx,s,lson);
    if(m < rx) update(lx,rx,s,rson);
    Push_up(rt,l,r);
}
//////////////////////////////////////

bool cmp(const note a,const note b)
{
    return a.h < b.h || (a.h == b.h && a.s < b.s); ///?????排序的时候可以不对s进行比较
}
int main()
{
    int n;
    while(~scanf("%d",&n))
    {
        int x,y,xx,yy;
        int ll = 10000,rr = -10000;
        int m = 0;
        for(int i = 0;i < n;i++)
        {
            scanf("%d%d%d%d",&x,&y,&xx,&yy);
            ll = min(ll,x);
            rr = max(rr,xx);
            data[m++] = note(x,xx,y,1);
            data[m++] = note(x,xx,yy,-1);
        }
        sort(data,data+m,cmp);
```

```
79
80          data[m].h = data[m-1].h;
81          int ret = 0,last = 0;
82          for(int i = 0;i < m;i++)
83          {
84              if(data[i].a != data[i].b) update(data[i].a ,data[i].b-1,data[i].s,ll,rr,1);
85              ret += abs(len[1] - last);
86              ret += memseg[1] * (data[i+1].h - data[i].h);
87              last = len[1];
88          }
89          printf("%d\n",ret);
90      }
91      return 0;
92  }
```

2.8 树链剖分

树链剖分实际上就是把一些高级数据结构推广到树上草做的一种方法。使用树链剖分的算法，将树上的关系 hash 到线段上，这样就可以使用线段树等高级数据结构来解决树上的问题。

树链，就是树上的路径。剖分，就是把路径分类为重链和轻链。

1. size[u] ： 以 u 为根的子树的结点总数。
2. 重儿子：u 的所有儿子中，size 最大的儿子就是 u 的重儿子。
3. 轻儿子：就是其他的儿子。
4. 重边：u 与他重儿子的连边。
5. 轻边：u 与他轻儿子的连边。
6. 重链：由重边组成的链路。
7. 轻链：轻边。

性质

1. 如果 $[u,v]$，边是轻边，那么 $size[v]*2<size[u]$。
2. 从根出发到所有的点的路径上的轻边和重边数量都不大于 $log(n)$。

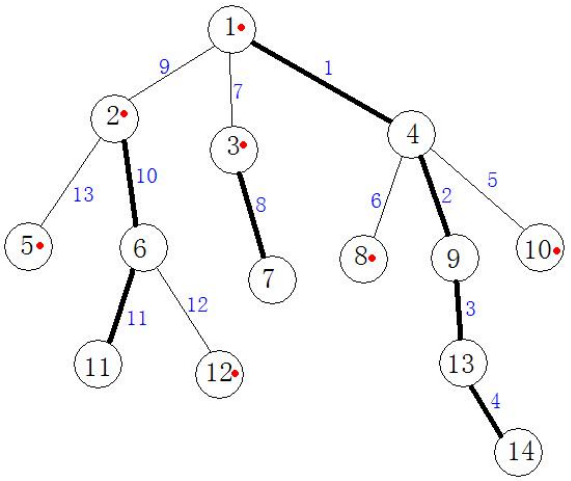
需要的信息。

1. size[N]、dep[N]：这两个就是普通的规模和深度，不需要解释。
2. fa[N] ： 父亲。
3. son[N] ： 重儿子。
4. top[N] ： 所在重链的顶端结点。
5. p[N] ： 表示 v 与其父亲的连边在线段树中的什么位置。在修改点权的时候就是该点在线段树上的位置。
6. fp[N] ： p 数组的反数组。

两遍 DFS 可以求出来上面的信息。
第一遍是找出所有的重边，第二步是把重边连起来成重链。

最后，我们需要原来的树边在线段树中呈现以下的特征：

1. 一段连续的重链要在线段树上是连续的一段。
2. 每个结点都属于唯一的一条重链。



【思路】:

1. 经过剖分算法，我们可以将树上的关系 hash 到线段上，然后使用数据结构进行维护。上面我们已经知道的每个点都在唯一的一条重链上。从上面的图可以看到有些叶子结点好像并没有在重链上，我们默认他们自己就是一个重链的顶端。

2. 单点修改。

直接在数据结构中单点修改就可以了。

3. 路径上点整体修改。

修改 u,v 路径上的所有的点的权值。当两个点在同一个重链上的时候，我们就直接找到这段线段然后修改就可以了。

当两个点不在同一条重链上的时候，例如上图中的点 7 和 9 ，我们先处理 7 所在的重链，然后跳到 3 的父亲结点，然后处理 9 所在的重链，然后也跳到了 1。这样分开一段一段的处理。同样的，即使需要跳很多次也是一样的，只不过我们要先跳比较深的那个点，就和倍增求 LCA 的时候一样。

4. 修改边权。

我们可以将所有的边唯一标示，用儿子结点去标示它和父亲的连边。这样边操作就变成了点操作，注意根结点的值，一定要能不影响结果。

下面给出点修改和边修改的两个板子。

```
/*************************************************************************
 > File Name: kb/树链剖分-a.cpp
 > Author: Bai Yan
 > Created Time: 2015年10月13日 星期二 21时18分01秒
 > 题目大意:
   HDU3966

   给定一棵树，三种操作:
   1.I : c1 c2 k 修改c1 到 c2 路径上点权加k
   2.D : c1 c2 k 路径点权减去k
   3.Q : c 查询c的点权;
*************************************************************************/

#include<iostream>
#include<cstdio>
#include<cstring>
#include<vector>
using namespace std;
const int N = 50010;


struct Edge
{
   int to,next;
}edge[N<<1];
int head[N],cnt;
```

```cpp
int top[N];//top[v]表示v所在的重链的顶端结点
int fa[N],deep[N];
int size[N];//size[v]表示以v为根的子树的结点数
int p[N];//p[v]表示v对应的位置
int fp[N];//与p数组相反
int son[N];//重儿子
int pos;
void init()
{
    cnt=0;
    memset(head,-1,sizeof(head));
    pos=1;//树状数组，编号从1开始
    memset(son,-1,sizeof(son));
}
void add(int from,int to)
{
    edge[cnt].to=to;
    edge[cnt].next= head[from];
    head[from]= cnt++;
}

void dfs1(int u ,int pre, int d)
{
    //求深度。求父亲，求size，求重儿子。
    deep[u]=d;
    fa[u]=pre;
    size[u]=1;
    for(int i=head[u]; i!=-1; i=edge[i].next)
    {
        int v=edge[i].to;
        if(v != pre)
        {
            dfs1(v,u, d+1);
            size[u]+=size[v];
            //如果u不是重儿子，或者，v的子树规模大于u的重儿子的子树规模
            if(son[u] == -1 || size[v]> size[son[u]])
                son[u]=v;
        }
    }
}
void getpos(int u,int sp)
{
    top[u]=sp;
    p[u]= pos++;
    fp[p[u]] = u; //第pos个是u点;
    if(son[u] ==-1)
        return ;
    getpos(son[u], sp);
    for(int i=head[u]; i!=-1;i=edge[i].next)
    {
        int v=edge[i].to;
        //v不是u的重儿子
        if(v!= son[u] && v!=fa[u])
            getpos(v,v);
    }
}


int c[N],n;
int lowbit(int x)
{
    return x&(-x);
}
int sum(int i)
{
    int s=0;
    while(i>0)
    {
        s+=c[i];
        i-=lowbit(i);
    }
```

```
    return s;
}
void Add(int i,int val)
{
    while(i<=n)
    {
        c[i]+=val;
        i+=lowbit(i);
    }
}
void change(int u,int v,int val)
{
    int f1= top[u],f2= top[v];
    int temp=0;
    while(f1!=f2)
    {
        //f1,u 更深;
        if(deep[f1]< deep[f2])
        {
            swap(f1,f2);
            swap(u,v);
        }
        Add(p[f1],val);
        Add(p[u]+1,-val);
        //树状数组上的区间操作要注意一下。
        u=fa[f1];
        f1=top[u];
    }
    if(deep[u]> deep[v])
        swap(u,v);
    Add(p[u], val);
    Add(p[v]+1, -val);
}
int a[N];
void check()
{
    for(int i=1;i<=n;i++)
    {
        cout<<c[p[i]]<<"␣"<<p[i]<<endl;
    }
}
int main()
{
    int M,P;
    while(scanf("%d%d%d",&n,&M,&P)==3)
    {
        int u,v;
        int c1,c2,k;
        char op[10];
        init();
        for(int i=1;i<=n;i++)
            scanf("%d",&a[i]);
        while(M--)
        {
            scanf("%d%d",&u,&v);
            add(u,v);
            add(v,u);
        }
        dfs1(1,0,0);
        getpos(1,1);
        memset(c,0,sizeof(c));
        for(int i=1;i<=n;i++)
        {
            Add(p[i],a[i]);
            Add(p[i]+1,-a[i]);
            //这里的树状数组比较重要;
        }
    // check();
        while(P--)
        {
            scanf("%s",op);
```

```
            if(op[0]=='Q')
            {
                scanf("%d",&u);
                printf("%d\n",sum(p[u]));
            }
            else
            {
                scanf("%d%d%d",&c1,&c2,&k);
                if(op[0]=='D')
                    k=-k;
                change(c1,c2,k);
            }
        }
    }
    return 0;
}
```

```
/*************************************************************************
 > File Name: kb/树链剖分-b.cpp
 > Author: Bai Yan
 > Created Time: 2015年10月14日 星期三 15时49分00秒
 > 题目大意:
    poj2763
    一棵树两种操作:
    1. 询问从当前位置到某一节点的距离;
    2. 修改第a条边边权为b
*************************************************************************/

#include<iostream>
#include<cstdio>
#include<cstring>
using namespace std;
const int N = 100010;
int n;
int fa[N],top[N],deep[N],son[N],p[N],fp[N],size[N];
int pos;
int edge_id[N],C[N];

struct Edge
{
    int from,to,next,id;
}edge[N<<2];
int head[N], cnt;
void init()
{
    cnt=0;
    memset(head,-1,sizeof(head));
    pos=1;
    memset(son,-1,sizeof(son));
}
void add(int from,int to,int ID)
{
    edge[cnt].id=ID;
    edge[cnt].from=from;
    edge[cnt].to=to;
    edge[cnt].next=head[from];
    head[from]= cnt++;
}
void dfs1(int u,int pre,int d)
{
    deep[u]=d;
    fa[u]=pre;
    size[u]=1;
    for(int i=head[u]; i!=-1;i=edge[i].next)
    {
        int v=edge[i].to;
        if(v!=pre)
        {
            edge_id[edge[i].id]= v;
            dfs1(v,u,d+1);
```

```
            size[u]+=size[v];
            if(son[u]==-1 || size[v]> size[son[u]])
                son[u]=v;
        }
    }
}
void getpos(int u,int sp)
{
    top[u]=sp;
    p[u]= pos++;
    fp[p[u]] =u;
    if(son[u]==-1)
        return ;
    getpos(son[u],sp);
    for(int i=head[u]; i!=-1; i=edge[i].next)
    {
        int v=edge[i].to;
        if(v!=son[u] && v!=fa[u])
            getpos(v,v);
    }
}
struct Node
{
    int l,r,sum;
}tr[N<<2];
void Pushup(int rt)
{
    int ls=rt<<1;
    int rs= ls+1;
    tr[rt].sum= tr[ls].sum+tr[rs].sum;
}

void build(int rt,int l,int r)
{
    tr[rt].l=l;
    tr[rt].r=r;
    tr[rt].sum=0;
    if(l==r)
        return ;
    int mid=(l+r)/2;
    build(rt<<1, l,mid);
    build(rt<<1|1, mid+1, r);
}
void Update(int rt,int k,int val)
{
    if(tr[rt].l== k && tr[rt].r==k)
    {
        tr[rt].sum=val;
        return ;
    }
    if(tr[rt].l==tr[rt].r)
        return ;
    int ls=rt<<1;
    int rs = ls+1;
    if(tr[ls].r<k)
        Update(rs,k,val);
    else
        Update(ls,k,val);
    Pushup(rt);
}
int Query(int rt, int l,int r )
{
    if(tr[rt].l==l && tr[rt].r ==r)
        return tr[rt].sum;
    int ls=rt<<1;
    int rs = ls+1;
    if(tr[ls].r<l)
        return Query(rs, l,r);
    else if(tr[rs].l>r)
        return Query(ls, l,r);
    else
```

```
        return Query(ls,l,tr[ls].r)+ Query(rs, tr[rs].l,r);
}
void change(int u,int val)
{
    int v=edge_id[u];
    Update(1,p[v],val);
}
void query(int u,int v)
{
    int f1=top[u],f2=top[v];
    int ans=0;
    while(f1!=f2)
    {
        if(deep[f1]<deep[f2])
        {
            swap(f1,f2);
            swap(u,v);
        }
        ans+= Query(1, p[f1], p[u]);
        u=fa[f1];
        f1=top[u];
    }
    if(u==v)
        printf("%d\n",ans);
    else
    {
        if(deep[u]> deep[v])
            swap(u,v);
        ans+= Query(1,p[son[u]],p[v]);
        printf("%d\n",ans);
    }
    return ;
}
int main()
{
    int q,s;
    while(scanf("%d%d%d",&n,&q,&s)!=EOF)
    {
        init();
        for(int i=1;i<n;i++)
        {
            int a,b;
            scanf("%d%d%d",&a,&b,&C[i]);
            add(a,b,i);
            add(b,a,i);
        }
        dfs1(1,0,0);
        getpos(1,1);
        build(1,1,pos-1);
        for(int i=1;i<n;i++)
            change(i,C[i]);
        while(q--)
        {
            int a,b;
            scanf("%d%d",&a,&b);
            if(a==0)
            {
                query(s,b);
                s=b;
            }
            else
            {
                int c;
                scanf("%d",&c);
                change(b,c);
            }
        }
    }
    return 0;
}
```

2.9 主席树

主席树是一种可持续化数据结构。

什么是可持续化数据结构？一般的数据结构支持我们快速修改，但是不支持我们去访问修改过程中的历史版本，也就是改完就改完了。可持续化事实上就是说，修改后的树是一颗新的树。修改之前的树我们也保存下来了。但是很明显，会超空间。

主席树是针对线段树的一种可持续化数据结构，其他的树也可以进行同样的可持续化。

我们利用修改前后两棵树的相同的部分去节省空间，既然我们只是修改了一小部分，我们可以对没有修改的部分进行重新利用。利用指针的思维我们可以很容易理解，就是如果我们有一颗子树没有被修改，那么我们就直接去指向他而不是重新建一颗一模一样的树。这样我们就实现了可持续化的省空间版。

这样的有一个弊端，就是我们在进行 lazy 标记的时候会比较麻烦。因为这个树已经不仅仅是你自己的树了，他是多颗子树公用的一段，所以我们不能擅自改变树中的数据，也就是我们在计算一些数值的时候不能将 lazy 进行 Pushdown 的操作，我们需要在函数的参数中传入 lazy 信息进行累加。

其实这是一个十分简单的过程。

那么我们一般怎么使用主席树？直接来个例题讲一讲：

1. 静态区间第 K 大

这个问题我们先将数组离散化，然后建 n 棵线段树。第 i 棵线段树存的是前 i 个数的情况，线段树的每一段 [l,r]，记录的是前 i 个数中出现在这个区间的数的个数。这样我们求区间的第 k 大就是用类似二分的过程，统计左右子区间中数的个数，然后找到第 k 个数。

求区间 $[x,y]$ 的第 k 大。

$t = root[y].[1,mid] - root[x-1].[1,mid]$，表示 $[x,y]$ 区间中在 $[1,mid]$，中的个数。

递归遍历时，参数是两棵树的根，相当于同时遍历两个棵树。不好使用递归写，要循环的写法。

2. 动态区间第 k 大

这个我们需要在主席树的基础上再套上一个树状数组。利用树状数组修改值，这样我们只需要修改 $log(n)$ 个线段树就可以了，查询的时候利用树状数组求和的过程去求区间。

下面就直接给代码了。

```

//静态区间第K大。
#include<iostream>
#include<cstdio>
#include<cstring>
#include<algorithm>
using namespace std;
const int N = 100005;
struct Node
{
    int l, r;
    int ls;
    int rs;
    int sum;
}tr[N<<5];
int root[N],cnt;

void init()
{
    memset(root,0,sizeof(root));
    memset(tr, 0, sizeof(tr));
    cnt=0;
}
int a[N], t[N];
int Build(int l ,int r)
{
    int rt = cnt++;
    tr[rt].l = l ;
    tr[rt].r = r;
    tr[rt].sum = 0;
    tr[rt].ls = tr[rt].rs = 0;
    if(l==r) return rt;
    int mid = (l+r)/2;
    tr[rt].ls = Build(l, mid);
    tr[rt].rs = Build(mid+1, r);
    return rt;
}
```

```
void Pushup(int rt)
{
    if(tr[rt].l==tr[rt].r) return ;
    int lson = tr[rt].ls;
    int rson = tr[rt].rs;
    tr[rt].sum = tr[lson].sum + tr[rson].sum;
    return ;
}

int Update(int rrt, int l ,int val)
{
    int rt = cnt++;
    tr[rt] = tr[rrt];
    if(tr[rt].l == l && tr[rt].r == l)
    {
        tr[rt].sum += val;
        return rt;
    }
    if(tr[rt].l==tr[rt].r) return rt;
    int mid = (tr[rt].l + tr[rt].r)/2;
    if(l<=mid)
        tr[rt].ls = Update(tr[rt].ls, l, val);
    else
        tr[rt].rs = Update(tr[rt].rs, l, val);
    Pushup(rt);
    return rt;
}
/*
int Update(int rrt, int pos, int val)
{
    int rt = cnt++;
    tr[rt] =tr[rrt];
}*/

int Query(int rtl, int rtr, int k,int n)
{
    int l = 1, r = n;
    while(l < r)
    {
        int mid = (l + r)/2;
        if( tr[tr[rtr].ls].sum- tr[tr[rtl].ls].sum >= k)
        {
            r = mid;
            rtl = tr[rtl].ls;
            rtr = tr[rtr].ls;
        }
        else
        {
            l = mid +1;
            k -= tr[tr[rtr].ls].sum - tr[tr[rtl].ls].sum;
            rtl = tr[rtl].rs;
            rtr = tr[rtr].rs;
        }
    }
    return l;
}
int main()
{
    int n,m;
    while(scanf("%d%d", &n,&m)!=EOF)
    {
        init();
        for(int i=1;i<=n;i++)
        {
            scanf("%d", &a[i]);
            t[i] = a[i];
        }
        sort(t+1, t+n+1);
        int num = unique(t+1, t+1+n) - t-1; // 注意这里的减一;

        root[0] = Build(1, num);
```

```
        for(int i=1;i<=n;i++)
        {
            int pos = lower_bound(t+1, t+1+num, a[i]) - t;
            root[i] = Update(root[i-1], pos, 1 );
        }
        while(m--)
        {
            int l, r, k;
            scanf("%d%d%d", &l, &r, &k);
            int ans = Query(root[l-1], root[r], k,num);
            printf("%d\n", t[ans]);
        }
    }
    return 0;
}
```

动态区间第 K 大，是使用主席树套树状数组。
主席树还是原来的主席树，树状数组的区别就是，现在的树状数组的，每个结点是一颗树。
静态区间第 K 大的做法是我们利用前缀和，求区间中小于某一数值的数的数量。利用二分的方法计算第 K 大。
同样我们还是要使用前缀和的方法去求区间第 K 大，只是现在我们要修改结点，这个就比较难办了。因为如果我们使用前缀和，当我们修改某个点的时候，我们就得修改此后的所有的点的值，这样才能不影响前缀和的正确性，显然我们如果对后面的所有的树都修改，那么肯定会时间爆炸，空间爆炸。
时间爆炸比较好理解，空间爆炸需要解释一下。我们在修改的时候事实上不是修改，我们对原来的数的位置 -1，再对新的数的位置 +1。这两个过程我们相当于建两棵新的树。这样我们就会空间爆炸了。
这里使用的处理方法是，新建一颗主席树。一个全空的主席树，来记录修改。这棵新的树套上树状数组。原来的树状数组不变。
这样这颗新的主席树就可以进行一些改动。依靠树状数组进行跳跃修改，节省时间和空间。
下面讲一下具体的实现。
对所有原来的数以及将会改变成的数进行离散化，然后建 n 棵线段树。
建一颗新的空树，对于每一个修改，我们就先 -1，再 +1。修改的树就是 log(n) 棵。
这里为什么不在原来的树上进行修改。第一、在原来的树上修改空间会浪费非常大。如果我们在原来的树上进行操作，虽然修改的树也是 log(n) 棵，但是对于每一次修改，我们都要将原来的树复制一遍，将原来的数的位置 -1，再将新的位置 +1。这样，原来空间一些不必要的信息我们也复制了一遍，就会超内存了。在一颗新树上操作的时候我们不需要将原来的树复制一遍。加入重复修改某个位置的值，我们就直接建树修改就可以了。当然我们也可以写内存回收，这样也可以避免内存爆炸。第二，如果在原树上修改代码比较复杂。。。。

```

//动态区间第K大，树状数组套主席树。
/*************************************************************************
 > File Name: DKth.cpp
 > Author: Bai Yan
 > 题意:
    动态第k大的处理中，前缀和的部分交给树状数组。
 > Created Time: 2016年09月10日 星期六 20时43分14秒
 ************************************************************************/


#include<iostream>
#include<cstdio>
#include<cstring>
#include<algorithm>
using namespace std;
const int N = 50005;

struct Node
{
    int l,r;
    int ls, rs;
    int sum;
}tr[N<<5];
int a[N], t[N];
int root[N];
int cnt;
int s[N];

```

```
struct query
{
    int kind;
    int l,r,k;
}Q[N];


int Build(int l , int r)
{
    int rt = cnt++;
    tr[rt].l = l;
    tr[rt].r = r;
    tr[rt].sum = 0;
    tr[rt].ls = tr[rt].rs = 0;
    if(l==r) return rt;
    int mid = (l+r)/2;
    tr[rt].ls = Build(l, mid);
    tr[rt].rs = Build(mid+1, r);
    return rt;
}

void Pushup(int rt)
{
    if(tr[rt].l == tr[rt].r) return ;
    int lson = tr[rt].ls;
    int rson = tr[rt].rs;
    tr[rt].sum = tr[lson].sum + tr[rson].sum;
    return ;
}

int Update(int rrt, int l, int val)
{
    int rt = cnt++;
    tr[rt] = tr[rrt];
    if(tr[rt].l==l && tr[rt].r==l)
    {
        tr[rt].sum += val;
        return rt;
    }
    int mid = (tr[rt].l + tr[rt].r)/2;
    if(l <= mid)
        tr[rt].ls = Update(tr[rt].ls, l , val);
    else tr[rt].rs = Update(tr[rt].rs, l , val);
    Pushup(rt);
    return rt;
}

int lowbit(int x){return x&(-x);}

void Modify(int x, int pos, int val,int n)
{
    for(;x<=n;x+=lowbit(x))
        s[x] = Update(s[x], pos, val);
    //新树上进行更改，之前更改的数据会记录下来。
}
int use[N];

int sum(int x)
{
    int ans =0;
    for(;x>0;x-=lowbit(x)) ans += tr[tr[use[x]].ls].sum;
    return ans;
}
int Query(int l, int r, int k ,int m)
{
    memset(use, 0, sizeof(use));
    int rtl = root[l-1];
    int rtr = root[r];
    int L = 1, R = m;
    for(int i=l-1; i>0; i-=lowbit(i)) use[i] = s[i];
    for(int i=r; i>0; i-=lowbit(i)) use[i] = s[i];
```

```
    while(L < R)
    {
        int mid = (L + R)/2;
        int tmp = sum(r) - sum(l-1) + tr[tr[rtr].ls].sum - tr[tr[rtl].ls].sum;
       if(tmp >= k )
       {
          R = mid;
          for(int i=l-1; i>0; i-=lowbit(i)) use[i] = tr[use[i]].ls;
          for(int i=r; i>0; i-=lowbit(i)) use[i] = tr[use[i]].ls;
          rtl = tr[rtl].ls;
          rtr = tr[rtr].ls;
       }
       else
       {
          L = mid+1;
          k-=tmp;
          for(int i=l-1;i>0; i-=lowbit(i)) use[i] = tr[use[i]].rs;
          for(int i=r;i>0; i-=lowbit(i)) use[i] = tr[use[i]].rs;
          rtl = tr[rtl].rs;
          rtr = tr[rtr].rs;
       }
    }
    return L;
}
int main()
{
    int T;
    scanf("%d", &T);
    while(T--)
    {
        int n,q;
        scanf("%d%d", &n,&q);
        for(int i=1;i<=n;i++)
        {
            scanf("%d", &a[i]);
            t[i] = a[i];
        }
        int m = n;
        for(int i=0;i<q;i++)
        {
            char s[10];
            int l ,r, k;
            scanf("%s␣%d%d", s, &l, &r);
            if(s[0]=='Q')
            {
                scanf("%d", &k);
                Q[i].kind = 0;
                Q[i].l = l;
                Q[i].r = r;
                Q[i].k = k;
            }
            else
            {
                Q[i].kind = 1;
                Q[i].l = l;
                Q[i].r = r;
                t[++m] = r;
            }
        }
        sort(t+1, t+1+m);
        m = unique(t+1, t+1+m) - t-1;

        cnt = 0;
        root[0] = Build(1, m);
        for(int i=1;i<=n;i++)
        {
            int pos = lower_bound(t+1, t+1+m, a[i]) - t;
            root[i] = Update(root[i-1], pos, 1);
        }

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

```
            s[i] = root[0];
        //新建一颗主席树，套在树状数组上，在这棵树上记录更改的值，这样避免内存爆炸。

        for(int i=0;i<q;i++)
        {
            if(Q[i].kind == 1)
            {
                int pos = lower_bound(t+1,t+1+m, a[Q[i].l]) - t;
                int pos1 = lower_bound(t+1,t+1+m, Q[i].r) - t;
                Modify(Q[i].l, pos, -1, n);
                Modify(Q[i].l, pos1, 1, n);
                a[Q[i].l] = Q[i].r;
            }
            else
            {
                int ans = Query(Q[i].l, Q[i].r, Q[i].k, m);
                printf("%d\n", t[ans]);
            }
        }

    }
    return 0;
}
```

树上 LCA+ 主席树

利用主席树的前缀和的思路我们可以解决这类问题，但是前缀和未必就是线性表中。在树上，我们利用每个结点到根的距离来计算任意两点之间的距离。在计算前缀和的时候我们计算的是每个结点到根结点的和，这样任意两个点（a,b）的路径上第 K 大就是:
$root[a]+root[b]-root[lca(a,b)]-root[fa[lca(a,b)]]$;
这个需要在线的 lca 算法来支持。

2.10 KD-树

KD 树是用来求临近点的算法。可以求出临近的 K 个点等。这个题的模板性比较强。

1. 求某个点的临近 K 个点。纯模板。

```
#include<queue>
#include<cstdio>
#include<cstring>
#include<algorithm>
using namespace std;
const int N=55555, K=5;
const int INF=0x3f3f3f3f;

#define sqr(x) (x)*(x)

int k,n,idx; //k为维数,n为点数
struct point
{
    int x[K];
    bool operator < (const point &u) const
    {
        //按照idx维为依据排序
        return x[idx] < u.x[idx];
    }
}po[N];

typedef pair<double,point> tp;
priority_queue<tp>nq;
//优先队列里存pair的时候按first 大根堆。

struct kdTree
{
    point pt[N<<2];
    int son[N<<2];

    void build(int l,int r,int rt=1,int dep=0)
```

```
    {
        if(l>r) return;
        son[rt]=r-l;
        son[rt*2]=son[rt*2+1]=-1;
        idx=dep%k;
        int mid=(l+r)/2;
        nth_element(po+l,po+mid,po+r+1);
        //函数的意思是， 把第po+mid个放中间，比他小的放左边，比他大的放右边。按照cmp顺序；这里重载了小于号。
        pt[rt]=po[mid];
        build(l,mid-1,rt*2,dep+1);
        build(mid+1,r,rt*2+1,dep+1);
    }
    void query(point p,int m,int rt=1,int dep=0)
    {
        if(son[rt]==-1) return;
        tp nd(0,pt[rt]);
        for(int i=0;i<k;i++) nd.first+=sqr(nd.second.x[i]-p.x[i]);
        int dim=dep%k,x=rt*2,y=rt*2+1,fg=0;
        if(p.x[dim]>=pt[rt].x[dim]) swap(x,y); //x代表当前维度上大于当前点的儿子；
        //如果在当前维度上，要求的点的值大于当前点的值，我们就遍历他的右子树，
        //但是我们不知道要不要遍历他的左子树，所以我们用fg来标记。
        //如果加入当前点，我们就要遍历一下另一个子树。
        if(~son[x]) query(p,m,x,dep+1);
        if(nq.size()<m) nq.push(nd),fg=1; //不足m个就加入队列。
        else
        {
            if(nd.first<nq.top().first) nq.pop(),nq.push(nd); //满足m个但是当前的比最大的小，就换掉。
            if(sqr(p.x[dim]-pt[rt].x[dim])<nq.top().first) fg=1; //
        }
        if(~son[y]&&fg) query(p,m,y,dep+1);
    }
}kd;
void print(point &p)
{
    for(int j=0;j<k;j++) printf("%d%c",p.x[j],j==k-1?'\n':' ');
}
int main()
{
    while(scanf("%d%d",&n,&k)!=EOF)
    {
        for(int i=0;i<n;i++) for(int j=0;j<k;j++) scanf("%d",&po[i].x[j]);
        kd.build(0,n-1);
        int t,m;
        for(scanf("%d",&t);t--;)
        {
            point ask;
            for(int j=0;j<k;j++)
                scanf("%d",&ask.x[j]);
            scanf("%d",&m);
            kd.query(ask,m);
            printf("the closest %d points are:\n", m);
            point pt[20];
            for(int j=0;!nq.empty();j++)
                pt[j]=nq.top().second,nq.pop();
            for(int j=m-1;j>=0;j--)
                print(pt[j]);
        }
    }
    return 0;
}
```

这里使用到的 nth_ element 函数是算法头文件里的。
官方的使用方法是：
nth_ element(a,a+k, a+n); 将数组中第 k 大的数放到第 k 个位置上去，比它小的在左边，比他大的在右边。但是 k 是从 0 开始的。

不经意间看到了 random_ shuffle(a.begin(), a.end());
这个函数会把数组乱序，随机乱序。

2. 会新加点的求最临近结点。

初始有 n 个黑点，在二维坐标上，然后有两种，一种是放白点，这时要查询距离白点最近的点是谁；第二种就是放黑点。可以点重合。
距离最近使用的是曼哈顿距离。

```
#include <iostream>
#include <cstdio>
#include <algorithm>
#include <cstring>
#include <vector>
#include <utility>
#include <iomanip>
#include <string>
#include <cmath>
#include <queue>
#include <assert.h>
#include <map>
#include <ctime>
#include <cstdlib>
#include <stack>
#include <set>
#define LOCAL
const int INF = 0x7fffffff;
const int MAXN = 100000 + 10;
const int maxnode = 20000 * 2 + 200000 * 20;
const int MAXM = 50000 + 10;
const int MAX = 100000000;
using namespace std;
struct Node
{//kd_tree
   int d[2], l, r;
   int Max[2], Min[2];
}t[1000000 + 10],tmp;

int n,m,root,cmp_d;
int k1, k2, k3, Ans;

bool cmp(Node a, Node b)
{
    return (a.d[cmp_d]<b.d[cmp_d]) || ((a.d[cmp_d] == b.d[cmp_d]) && (a.d[!cmp_d] < b.d[!cmp_d]));
}
void update(int p){
     if (t[p].l){
        //左边最大的横坐标值
        t[p].Max[0] = max(t[p].Max[0], t[t[p].l].Max[0]);
        t[p].Min[0] = min(t[p].Min[0], t[t[p].l].Min[0]);
        t[p].Max[1] = max(t[p].Max[1], t[t[p].l].Max[1]);
        t[p].Min[1] = min(t[p].Min[1], t[t[p].l].Min[1]);
     }
     if (t[p].r){
        t[p].Max[0] = max(t[p].Max[0], t[t[p].r].Max[0]);
        t[p].Min[0] = min(t[p].Min[0], t[t[p].r].Min[0]);
        t[p].Max[1] = max(t[p].Max[1], t[t[p].r].Max[1]);
        t[p].Min[1] = min(t[p].Min[1], t[t[p].r].Min[1]);
     }
     return;
}
//d是横竖切..
int build(int l, int r, int D){
    int mid = (l + r) / 2;
    cmp_d = D;
    //按照cmp的比较顺序在l到r中找到第mid大的元素
    nth_element(t + l + 1, t + mid + 1, t + r + 1, cmp);
    t[mid].Max[0] = t[mid].Min[0] = t[mid].d[0];
    t[mid].Max[1] = t[mid].Min[1] = t[mid].d[1];
    //递归建树
    if (l != mid) t[mid].l = build(l, mid - 1, D ^ 1);
    if (r != mid) t[mid].r = build(mid + 1, r, D ^ 1);
    update(mid);
    return mid;
}
```

```
void insert(int now){
    int D = 0, p = root;//D还是表示方向
    while (1){
         //边下传边更新
         t[p].Max[0] = max(t[p].Max[0], t[now].Max[0]);
         t[p].Min[0] = min(t[p].Min[0], t[now].Min[0]);
         t[p].Max[1] = max(t[p].Max[1], t[now].Max[1]);
         t[p].Min[1] = min(t[p].Min[1], t[now].Min[1]);
         //有没有点线段树的感觉..
         if (t[now].d[D] >= t[p].d[D]){
           if (t[p].r == 0){
             t[p].r = now;
             return;
           }else p = t[p].r;
         }else{
           if (t[p].l == 0){
             t[p].l = now;
             return;
           }else p = t[p].l;
         }
         D = D ^ 1;
    }
    return;
}
int ABS(int x) {return x < 0? -x : x;}
//dist越小代表越趋近?
int dist(int p1, int px, int py){
   int dist = 0;
   if (px < t[p1].Min[0]) dist += t[p1].Min[0] - px;
   if (px > t[p1].Max[0]) dist += px - t[p1].Max[0];
   if (py < t[p1].Min[1]) dist += t[p1].Min[1] - py;
   if (py > t[p1].Max[1]) dist += py - t[p1].Max[1];
   return dist;
}
void ask(int p){
  int dl, dr, d0;
  //哈密顿距离
  d0=ABS(t[p].d[0] - k2) + ABS(t[p].d[1] - k3);
  if(d0 < Ans) Ans = d0;
  if(t[p].l) dl = dist(t[p].l, k2, k3); else dl = 0x7f7f7f7f;
  if(t[p].r) dr = dist(t[p].r, k2, k3); else dr = 0x7f7f7f7f;
  //应该是一个启发式的过程。
  if(dl < dr){
     if(dl < Ans) ask(t[p].l);
     if(dr < Ans) ask(t[p].r);
  }else{
     if(dr < Ans) ask(t[p].r);
     if(dl < Ans) ask(t[p].l);
  }
}

void init(){
    //假设0为横坐标
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i++)
    scanf("%d%d", &t[i].d[0], &t[i].d[1]);
    root = build(1, n, 0);
}
void work(){
    for (int i = 1; i <= m ;i++){
       scanf("%d%d%d", &k1, &k2, &k3);
       if (k1 == 1){//黑棋
         ++n;
         t[n].Max[0] = t[n].Min[0] = t[n].d[0] = k2;
         t[n].Max[1] = t[n].Min[1] = t[n].d[1] = k3;
         insert(n);
       }else{
         Ans = 0x7f7f7f7f;
         ask(root);
         printf("%d\n", Ans);
       }
```

```
    }
}

int main(){

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

2.11 RMQ

RMQ : 区间最值问题（Range Minimum/Maximum Query）。

ST（Sparse Table）算法是一个非常有名的在线处理 RMQ 问题的算法，它可以在 O(nlogn) 时间内进行预处理，然后在 O(1) 时间内回答每个查询。

这是一个用 DP 方法预处理的算法。预处理出来 Max[i][k], 表示以 i 为起点，长度为 2^k 的区间内的最大值。

```

void RMQ_init() //下标从1开始，长度是n;
{
   for(int i=1; i<=n; i++) dp[i][0]=s[i];
   for(int j=1; (1<<j)<=n; j++)
      for(int i=1;i+(1<<j)-1<=n;i++)
         dp[i][j]=max(dp[i][j-1],dp[i+(1<<(j-1))][j-1]);
}

int RMQ(int L,int R)
{
   int k=0;
   while((1<<(k+1))<=R-L+1) k++;
   return max(dp[L][k],dp[R-(1<<k)+1][k]);

   //这个写法，中间会有一段重叠的地方。但是会比较块。
}

需要注意的是外层循环是k，内层循环是长度。
```

带修改值的最值问题就可以使用线段树来做了。简单点可以用树状数组 BIT。

下面给一个树状数组的写法：

树状数组求区间最值的时候跟求区间和有很大的区别。

因为最值没有前缀和的那种相减的性质。所以我们在更新的时候就需要仔细一点。

```

#include<iostream>
#include<cstdio>
#include<cstring>
#include<algorithm>
using namespace std;
const int N = 1005;
int a[N];
int Min[N],n;
int lowbit(int x){return x&(-x);}

void init_Min(int n)
{
   for(int i=1;i<=n;i++)
      for(int j=i;j<=n;j+=lowbit(j))
         Min[j] = min(Min[i], Min[j]);
}
int getMin(int l, int r)
{
   int ans = 0x3f3f3f3f;
   while( l <= r )
   {
      ans = min(a[r], ans);
```

```
        r--;
        for(;r-lowbit(r)>=l; r-=lowbit(r))
            ans = min(Min[r], ans);
    }
    return ans;
}
void Update(int x)
{
    while(x<=n)
    {
        Min[x] = a[x];
        for(int i=1;i<lowbit(x); i<<=1)
            Min[x] = min(Min[i], Min[x-i]);
        x += lowbit(x);
    }
}
int main()
{
    scanf("%d", &n);
    for(int i=1;i<=n;i++) scanf("%d",&a[i]);

    for(int i=1;i<=n;i++) Min[i] = a[i];

    init_Min(n);
    int m;
    scanf("%d", &m);
    while(m--)
    {
        int typ;
        cin>>typ;
        if(typ==1)
        {
            int l, r;
            cin>>l>>r;
            cout<<getMin(l,r);
        }
        else
        {
            int pos, x;
            cin>>pos>>x;
            a[pos] = x;
            Update(pos);
        }
    }
    return 0;
}
```

2.12 动态树问题

Chapter 3

图论

3.1 最短路

很多比赛中都会有最短路的题目，很多的题目是最短路的变种，不会直接让求最短路就结束了。
最短路有几种：
1. 单源最短路。
2. 多源多目的地的最短路。
3. 有负权甚至是负环的最短路。
4. 没有负环的最短路。

最短路可以解决的问题很广范，要会转化题目。很多的题目跟最短没有什么关系，但是可以利用最短路的更新方式去计算图上的一些东西。

常见的题目有：最短路，求最大边最小的路径，查分约束系统等。还可以结合网络流。
1. 求所有路径中，最大边最小的那个最大边。与最短路算法相似，只是在最短路中我们记录的是这最短路的总长度，现在就只需要记录这个最短路中的最大边就好了，更新的条件是不变的。
2. 还有一种是，求所有路径中最小边最大的边。这个跟上面一个有点反过来，每次找出最大的那条边，然后更新所有的路径。在 Dijkstra 的代码中给出两个程序。
3. 求一幅图的传递闭包，事实上就是一个图的可达性矩阵。可以用 floyd 也可以跑 n 次 dijkstra。例题：给一些牛之间的较量结果，问有多少牛可以确定排名。显然，按照大小关系见图，可达的牛是在他排名之后的，能达到他的是在它之前的。跑一遍然后判断就好了。
4. 差分约束系统是：在 SPFA 中讲到。
5. 判断是不是最短路上的边。只需要 $d1[from]+d2[to]+edge[i].dis=dis[n]$。其中，form 到 1 点的最短路 + to 到 n 点的最短路，加上这条边的值，等于 1 到 n 的最短路，这条边就是最短路上的边。

3.1.1 Dijkstra

Dijkstra 算法是 floyd 算法的一种简化版，单起点的最短路，复杂度是 n^2，当然也是用一种动态规划的思路，也叫做松弛。
不能处理负权。
floyd 的思路我们知道就是枚举中转点，然后比较。这里不能简单的枚举，因为不像完全的 Dp 那样不会漏或多，这里需要根据图的连通性和联通顺序进行枚举。
算法初始化 dis，优化和不优化的不一样。
然后是跑 n-1 次循环。每次找出距离起点最近的，且没有被计算过的点。然后利用这个点更新起点到没有计算过的其他点距离。
精髓在于更新的方程，查分约束系统的最短路解法就是在这个方程上推广过来的。
优先队列优化的复杂度是 $nlog(n)$。

```
/*
 Dijkstra() 邻接表 优先队列优化 可以使用链式前向星建图。
 点编号从1开始。

 poj2387
```

```
题目大意:
  1到n的最短路。
  有重边，但是邻接表没影响。
   */

#define N 1005
#define INF 0x3f3f3f3f
using namespace std;
int n,m;
struct Node
{
    int to, dis;
    Node(int a,int b)
    {
        to =a;
        dis=b;
    }
    bool operator < (const Node &A)const
    {
        if(dis== A.dis)
            return to < A.to ;
        else
            return dis > A.dis;
    }
};

vector <Node> from[N];
int dis[N];

void Dijkstra(int v)
{
    for(int i=1;i<=n;i++) dis[i]=INF;
    dis[v]=0;

    priority_queue <Node> q;
    q.push( Node(v,dis[v]) );

    while(!q.empty())
    {
        Node t=q.top(); q.pop();
        for(int i=0;i<from[t.to].size();i++)
        {
            Node temp= from[t.to][i];
            if(dis[temp.to] > t.dis+ temp.dis)
            {
                dis[temp.to]= t.dis+ temp.dis;
                q.push(Node(temp.to, dis[temp.to]));
            }
        }
    }
}
int main()
{
    scanf("%d%d",&m,&n);
    for(int i=0;i<m;i++)
    {
        int a,b,d;
        scanf("%d%d%d",&a,&b,&d);
        from[a].push_back( Node(b,d) );
        from[b].push_back( Node(a,d) );
    }
    Dijkstra(n);
    printf("%d\n",dis[1]);
    return 0;
}


// 邻接矩阵的写法。
// poj1502

#define INF 0x3f3f3f3f
```

```
#define N 105
using namespace std;
int A[N][N],dis[N],vis[N],n;

void Dijkstra(int v)
{
   memset(vis,0,sizeof(vis));
   for(int i=1;i<=n;i++)
      dis[i]=A[v][i];
   vis[v]=1;
   dis[v]=0;
   for(int i=1;i<n;i++)
   {
      int Min=INF;
      int pos=v;
      for(int j=1;j<=n;j++)
      {
         if(vis[j]==0 && Min > dis[j])
         {
            Min=dis[j];
            pos=j;
         }
      }
      vis[pos]=1;
      for(int j=1;j<=n;j++)
      {
         if(vis[j]==0 && dis[j] >dis[pos]+ A[pos][j])
            dis[j]=dis[pos]+ A [pos][j];
      }
   }
}
```

//求路径最大边最小。

```

//求路径最大边最小。
void Floyd()
{
   for(int k=0;k<=n;k++)
      for(int i=1;i<=n;i++)
         for(int j=1;j<=n;j++)
            if(k==0)
               dis[i][j]=A[j][i];
            else
               dis[i][j]=min(dis[i][j],max(dis[i][k],dis[k][j]));
}


double dijkstra()
{ //点编号从0开始。
   bool vis[maxn];
   memset(vis, false, sizeof(vis));
   for(int i=0; i<n; i++)
      d[i] = G[0][i];
   d[0] = 0;
   vis[0] = true;
   for(int i=0; i<n-1; i++)
   { // n个点跑n-1次。
      double Min = INF;
      int x;
      for(int y=0; y<n; y++)
         if(!vis[y] && m >= d[y])
            m = d[x=y];
      vis[x] = true;
      for(int y=0; y<n; y++)
         if(!vis[y])
         {
            double maxx = max(d[x], G[x][y]);
            if(d[y] > maxx) d[y] = maxx;
         }
   }
```

```
    return d[1];
}
```

求最小边最大。

```

 void Dijkstra(int v)
{
    memset(vis,0, sizeof(vis));
    for(int i=1;i<=n;i++) dis[i]=A[v][i];
    dis[v]=0;
    for(int i=1;i<n;i++)
    {
        int Max=-INF,pos=v;
        for(int j=1;j<=n;j++)
            if(vis[j]==0 && Max < dis[j] )
            {
                pos=j;
                Max=dis[j];
            }
        vis[pos]=1;
        for(int j=1;j<=n;j++)
            if(vis[j]==0 )
                dis[j]=max(dis[j],min(dis[pos],A[pos][j]));
    }
}
```

最短路记录路径。用一个 pre 数组记录每个点的前驱点，然后从最后的点开始向前找，找到起点结束。。

```

void dijkstra(int s, int e)
{
    int Min, next;
    for(int i = 1; i <= n; i++)
    {
        dist[i] = Map[s][i];
        vis[i] = false;
        pre[i] = dist[i]!=INF&&i!=s ? s : -1;//初始化要注意
    }
    vis[s] = true;
    for(int i = 2; i <= n; i++)
    {
        Min = INF;
        for(int j = 1; j <= n; j++)
        {
            if(!vis[j] && dist[j] < Min)
            {
                Min = dist[j];
                next = j;
            }
        }
        vis[next] = true;
        for(int j = 1; j <= n; j++)
        {
            if(!vis[j] && dist[j] > dist[next] + Map[next][j])
            {
                dist[j] = dist[next] + Map[next][j];
                pre[j] = next;//记录
            }
        }
    }
}
int main()
{
//打印路径。
    int now = pre[e];
    stack<int> path;
    path.push(e);
    while(1)
    {
```

```
        path.push(now);
        if(now == s)
        break;
        now = pre[now];
    }
    printf("从%d到%d的最优路线␣:␣%d", s, e, s);
    while(!path.empty())
    {
        printf("-->%d", path.top());
        path.pop();
    }
    printf("\n");
    printf("最小花费␣:␣%d\n", dist[e]);
}
```

3.1.2 Floyd

这是个动态规划的做法算法复杂度是 n^3，求出一个最短路的表。
三层循环，表示的 Dp 方程是：
从 i 到 j，有两种情况：
1. 直接到达：Dis[i][j] 2. 经过 k 点中转：Dis[i][k]+Dis[k][j];
二者取较大的。

```
void Floyd()
{
   for(int k=1;k<=n;k++)
      for(int i=1;i<=n;i++)
         for(int j=1;j<=n;j++)
            dis[i][j]=min(dis[i][j],dis[i][k]+dis[k][j]);
}

// 求路径最大边权最小的路径（不一定就是最短路）。
//只把上面的方程式换成：

dis[i][j]=min(dis[i][j],max(dis[i][k],dis[k][j]));
//dis存的是到达每个点的最小的最大边。

//dis数组需要初始化为 原图。
```

floyd 打印路径，使用 pre[i][j]，来记录点 j 的前驱点。

```
void init()
{
   for(int i=1;i<=n;i++)
      for(int j=1;j<=n;j++)
         pre[i][j] = i;
}
void floyd()
{
   int i, j, k;
   for(k = 1; k <= n; k++)
   {
      for(i = 1; i <= n; i++)
      {
         for(j = 1; j <= n; j++)
         {
            if(Map[i][j] > Map[i][k] + Map[k][j])
            {
               Map[i][j] = Map[i][k] + Map[k][j];
               pre[i][j] = pre[k][j];//记录
            }
         }
      }
   }
}

int main()
```

```
{
        int now = pre[s][e];
        stack<int> path;//记录路径
        path.push(e);
        while(1)
        {
            path.push(now);
            if(now == s)
            break;
            now = pre[s][now];
        }
        printf("从%d到%d的最优路线␣:␣%d", s, e, s);
        path.pop();
        while(!path.empty())
        {
            printf("-->%d", path.top());
            path.pop();
        }
        printf("\n");
        printf("最小花费␣:␣%d\n", Map[s][e]);
}
```

3.1.3 Bellman ford

可以处理负权边，复杂度为 $O(VE)$，可以判断是否存在负环。

```

//找正环，如果有正环，就返回1。否则返回0；
//点从1开始，边从0开始。

struct Edge
{
    int from,to;
    double r,c;
}edge[N<<1];

double dis[N];
bool Bellman(int start)
{
    for(int i=1;i<=n;i++)
        dis[i]=0.0;
    dis[start]=val;
    for(int i=1;i<n;i++)
    {
        bool flag=false;
        for(int j=0;j<M;j++)
        {
            int u=edge[j].from;
            int v=edge[j].to;
            double r=edge[j].r;
            double c=edge[j].c;
            if(dis[ v] < (dis[u ]-c)*r)
            {
                flag=true;
                dis[v]= (dis[u]-c)*r;
            }
        }
        if(!flag)
            return false;
    }
    for(int j=0;j<M;j++)
        if(dis[ edge[j].to ] < (dis[edge[j].from]-edge[j].c)*edge[j].r)
            return true;
    return false;
}


//判断是否存在负环。

```

```cpp
bool Bellman()
{
    for(int i=1;i<=n;i++)
        dis[i]=INF;
    for(int i=1;i<n;i++)
    {
        bool flag=0;
        for(int j=0;j<cnt;j++)
        {
            int from=edge[j].from;
            int to=edge[j].to;
            if(dis[to ] > dis[from ]+ edge[j].dis)
            {
                flag = 1;
                dis[to ] = dis[from ]+ edge[j].dis;
            }
        }
        if(flag==0)
            return 1;
        //本轮没有松弛，说明没有负环；
    }
    bool flag=1;
    for(int j=0;j<cnt;j++)
    {
        int from = edge[j].from;
        int to = edge[j].to;
        if(dis[to] > dis[from ] + edge[j].dis)
        {
            flag=0; break;
            //if的语句大于1句必须加括号；
        }
    }
    return flag;
}
```

3.1.4 SPFA

查分约束的题解：

poj3159 差分约束班上有 n 个同学，现在有一些糖要分给他们，设第 i 个同学得到的糖为 p[i]，分糖必须满足条件：第 i 个同学要求第 j 个同学的糖不能超过自己 k 个，即 p[j] - p[i] <= k，k >= 0。要求在满足这些条件的情况下，求出 p[n] - p[1] 的最大值。

由 p[j] - p[i] <= k 可得 p[j] <= p[i] + k 在单源最短路径的算法中有一步是“若 mindis[j] > mindis[i] + dis[i][j] ，则 mindis[j] = mindis[i] + dis[i][j]，这样就满足 mindis[j] <= mindis[i] + dis[i][j]”。因此本题可以使用单源最短路径的算法来解决，对于“第 i 个同学要求第 j 个同学的糖不能超过自己 k 个，即 p[j] - p[i] <= k ，k >= 0”这个条件，建立一条边 (i->j)=k，由于不含负权路径，因此建立完所有边之后以第 1 个同学为起点，采用 Dijkstra+Heap 求最短路径即可。除了 Dijkstra 也可以利用 Spfa+Stack 算法求解，但由于数据原因必须用 Stack，如果用 Queue 则会超时。

spfa 的复杂度是个迷。理论上说是 $O(KE)$

```cpp

//使用链式前向星建图。

//栈版。略快
int head[N],vis[N],q[N],dis[N],cnt[N];
int cnt,n,m;
struct Edge
{
    int next,to,dis;
}edge[M];

void add(int from ,int to,int dis)
{
    edge[cnt].to=to;
    edge[cnt].dis=dis;
    edge[cnt].next=head[from];
    head[from]=cnt++;
```

```
}
void SPFA(int s)
{
    memset(cnt,0,sizeof(cnt));//入栈次数。
    int top=0;
    for(int i=1;i<=n;i++)
    {
        if(i==s)
        {
            q[top++]=i;
            vis[i]=1; //在栈中标记。
            dis[i]=0;
        }
        else
        {
            vis[i]=0;
            dis[i]=INF;
        }
    }
    while(top!=0)
    {
        int u=q[--top];
        vis[u]=0;
        for(int i=head[u];~i;i=edge[i].next)
        {
            int v=edge[i].to;
            if(dis[v] > dis[u] + edge[i].dis)
            {
                dis[v]= dis[u]+ edge[i].dis;
                if(vis[v]==0 )
                {
                    vis[v]=1;
                    q[top++]=v;
                    if(++cnt[v]>n) return false; //存在负环
                }
            }
        }
    }
    return true; //不存在负环。
}


//队列版
//直接将入栈改成入队列就好了。
```

如何判断负环上的点？
如果一副图中出现负环，那么对于这个负环所能到达的所有点都是负环上的点，因为都可以变成负的。
所以只需要 dfs 一遍联通图的就可以了。

记录路径的代码。

```
void SPFA(int s, int e)
{
    queue<int> Q;
    for(int i = 1; i <= n; i++)
    dist[i] = i==s ? 0 : INF;
    memset(vis, false, sizeof(vis));
    vis[s] = true;
    Q.push(s);
    while(!Q.empty())
    {
        int u = Q.front();
        Q.pop();
        vis[u] = false;
        for(int i = head[u]; i != -1; i = edge[i].next)
        {
            Edge E = edge[i];
            if(dist[E.to] > dist[u] + E.val)
            {
                dist[E.to] = dist[u] + E.val;
```

```
            pre[E.to] = u;
            if(!vis[E.to])
            {
                vis[E.to] = true;
                Q.push(E.to);
            }
        }
    }
}
}
```

3.2 最小生成树

```

//邻接矩阵版
//复杂度是$O(V*V)$ ，V是点。用于稠密图。

int tu[M][M];
int vis[M],dis[M];
int n;

int prim()
{
    memset(pre, -1, sizeof(pre));
    //如果要记录树边，就记录直接前驱的点是谁;
    memset(vis,0,sizeof(vis));
    int Min = MAX_INT , pos = 1 , ans = 0;
    vis[1] = 1;
    for(int i=1;i<=n;i++)
        dis[i] = tu[pos][i];
    for(int i=1;i<n;i++)
    {
        Min = MAX_INT ;
        for(int j=1;j<=n;j++)
            if(vis[j] == 0 && Min > dis[j])
            {
                Min = dis[j];
                pos = j;
            }
        ans+=Min;
        vis[pos] = 1;
        for(int j=1;j<=n;j++)
            if(vis[j] == 0 && dis[j] > tu[pos][j])
            {
                dis[j] = tu[pos][j];
                pre[j] = pos; //记录前驱
            }
    }
    return ans;
}



//kruscal。
//用于稀疏图，复杂度是$Mlog(M)$，  M是边数。

using namespace std;
const int M = 100005;
struct Node
{
    int u,v,w;
    bool operator<(const Node& A)const
    {
        return w < A.w;
    }
}edge[M];

```

```cpp
int fa[N];
void init()
{
    for(int i=0;i<=n;i++)
        fa[i] = i;
}
int find(int x)
{
    return x==fa[x]?x:fa[x] = find(fa[x]);
}
int kruscal()
{
    sort(edge,edge+m);
    int cnt=1;
    init();
    for(int i=0;i<m;i++)
    {
        int fu = find(edge[i].u);
        int fv = find(edge[i].v);
        if(fu!=fv)
        {
            fa[fu] = fv;
            ans += edge[i].w;
            // is_edge[i] = true; 记录树边。
            cnt++;
            if(cnt==n) break;
        }
    }
    return ans;
}
```

3.3 次小生成树

次小生成树的算法很多，先记录一个。

```cpp
/*************************************************************************

计算次小与最小之间的差

*************************************************************************/
#include<iostream>
#include<cstring>
#include<cstdio>
#include<algorithm>
using namespace std;
const int N = 105;
const int INF = 0x3f3f3f3f;

int map[N][N], dis[N], vis[N], n, m, pre[N];
int sta[N], top; //记录最小生成树上的点;
int Dp[N][N];

int prim()
{
    top = 0;
    memset(vis, 0, sizeof(vis));
    memset(pre, -1, sizeof(pre));
    memset(Dp, 0, sizeof(Dp));
    int pos = 1, Min = INF, ans = 0;

    for(int i=1; i <= n ; i ++)
        dis[i] = map[1][i], pre[i] = 1;
    vis[1] = 1;
    sta[top++] = 1;

    for(int i = 1; i < n ; i++ )
    {
        Min = INF;
```

```
        int pos = 1;
        for(int j = 1; j <= n ; j ++)
            if(!vis[j] && dis[j] < Min)
                Min = dis[j], pos = j;

        ans += Min;
        vis[pos] = 1;
        for(int j = 0; j < top; j++)
            Dp[pos][sta[j]] = Dp[sta[j]][pos] = max(Min, Dp[sta[j]][pre[pos]]);
        sta[top++] = pos;

        for(int j = 1; j <= n; j ++)
            if(!vis[j] && dis[j] > map[pos][j])
                dis[j] = map[pos][j], pre[j] = pos;
    }
    return ans;
}

int main()
{
    int T;
    scanf("%d", &T);
    while(T--)
    {
        for(int i = 0;i < N; i++)
            for(int j = 0; j < N ;j++)
                if(i==j) map[i][j] = 0;
                else map[i][j] = INF;

        scanf("%d%d", &n,&m);
        for(int i = 0 ; i < m ; i++)
        {
            int u, v, w;
            scanf("%d%d%d", &u,&v,&w);
            map[u][v] = map[v][u] = w;
        }

        int Min_tree = prim();
        int Min = INF;
        for(int i = 1; i <= n; i++)
            for(int j = 1; j <= n; j++)
                if(i != j && i != pre[j] && j != pre[i])
                    Min = min(Min, map[i][j] - Dp[i][j]);
        //枚举不在MST中的边。
        //每次添加一个边进去，必定会形成一个环， 我们删除环上最大的边，就形成新的生成树，
        //新生成树和老的生成树之间的差就是添加进去和删除的边的差。
        //如果所有的修改中能得到一个差为0的修改，肯定不是唯一的。

        if(Min == 0)
            printf("Not␣Unique!\n");
        else printf("%d\n", Min_tree);
    }

    return 0;
}
```

3.4 最小瓶颈路

给定一个加权无向图，并给定无向图中两个结点 u 和 v，求 u 到 v 的一条路径，使得路径上边的最大权值最小。
无向图中，任意两个结点的最小瓶颈路肯定在最小生成树上。
上面的次小生成树的算法就是使用最小瓶颈路来实现的，在进行 prim 算法的过程中，计算出 Dp 数组，Dp[i][j] 表示 i 到 j 的路径上的最大边的值。然后枚举替换的边，然后计算新的生成树的值。所以 prim 版本的最小瓶颈路已经实现了。

下面给出 kruscal 的过程，我们先求出最小生成树，然后变成有根树，进行 Dp。

```cpp
/*************************************************************************
代码与上面的代码是一道题。
实现方法上不一样而已， 在poj上时间一样，但是代码长度prim写法更短。
 *************************************************************************/
#include<iostream>
#include<cstdio>
#include<cstring>
#include<algorithm>

using namespace std;

const int N = 105;
const int INF = 0x3f3f3f3f;

int n, m, fa[N], pre[N], Dp[N][N], sta[N], top;

struct Node
{
    int from,to, w, next, id;
    bool operator < (const Node &A)const
    {
        return w < A.w;
    }
}edge[N*N];

int cnt, head[N];
bool is_tree[N*N];

void init()
{
    cnt = 0;
    memset(head, -1, sizeof(head));
    memset(is_tree, 0, sizeof(is_tree));
    for(int i = 0; i < N; i ++)
        fa[i] = i;
}

void add(int u, int v , int w)
{
    edge[cnt].from = u;
    edge[cnt].to = v;
    edge[cnt].w = w;
    edge[cnt].next = head[u];
    edge[cnt].id = cnt;
    head[u] = cnt++;
}

int find(int x)
{
    return x==fa[x]? x : fa[x]=find(fa[x]);
}

int kruscal()
{
    sort(edge, edge+cnt);
    int num=1;
    int ans=0;

    for(int i = 0 ;i < cnt; i ++)
    {
        int u = edge[i].from;
        int v = edge[i].to;
        int fu = find(u);
        int fv = find(v);
        if(fu!=fv)
        {
            fa[fv] = fu;
            ans += edge[i].w;
            is_tree[edge[i].id] = true;
            num++;
            if(num>=n) break;
```

```
        }
    }
    return ans;
}

bool cmp(Node A, Node B)
{
    return A.id < B.id;
}
void Dfs(int u)
{
// cout<<u<<endl;
    for(int i = head[u]; i!=-1 ;i = edge[i].next)
    {
        if( is_tree[i] == 1 )
        {
            int v = edge[i].to;
            for(int j = 0; j< top ; j ++)
                Dp[v][j] = Dp[j][v] = max(edge[i].w , Dp[u][j]);
            sta[top++] = v;
        }
    }

    for(int i= head[u]; i!=-1; i = edge[i].next)
    {
        if( is_tree[i] == 1 )
            Dfs(edge[i].to);
    }
}
int main()
{
    int T;
    scanf("%d", &T);
    while(T--)
    {
        init();
        scanf("%d%d", &n,&m);
        for(int i = 0 ; i < m ; i++)
        {
            int u, v, w;
            scanf("%d%d%d", &u,&v, &w);
            add(u, v, w);
            add(v, u, w);
        }
        int Min_tree = kruscal();

        sort(edge, edge+cnt, cmp);
        memset(Dp, 0 , sizeof(Dp));

/*  for(int i=0;i<cnt;i++)
          printf("%d: %d %d %d %d \n",is_tree[i], edge[i].from, edge[i].to, edge[i].next, edge[i].id);
*/  int root = 0 ;
       for(int i=1;i<=n;i++)
          if(fa[i] == i){ root = i; break;}
       top=0;
       sta[top++] = root;
       Dfs(root);

    /* printf("root : %d\n",root);
        for(int i=1;i<=n;i++)
            for(int j=1;j<=n;j++)
                printf("-> %d %d %d\n", i,j, Dp[i][j]);
    */ int Min = INF;
        for(int i = 0;i < cnt; i++)
              if( is_tree[i] != 1 && is_tree[i^1]!=1)
              {
                  int u = edge[i].from , v = edge[i].to;
                  Min = min(Min, edge[i].w - Dp[u][v]);
              }
        if(Min ==0 )
            printf("Not␣Unique!\n");
```

```
        else
            printf("%d\n", Min_tree);

    }
    return 0;
}
```

3.5 最小树形图

最小树形图是有向图中的最小生成树，固定起点的，全局最小代价。

模板调用注意事项：

模板中，点的编号从 0 开始，到 n-1。

```
/*************************************************************************
题目:
给出有向图，每户人家可以自己打井或者从某些人的家中接井。我们把自己打井看做是从一个超级源点接井，
     然后就是添加一个超级源点。跑一遍最小树形图就好了。
 *************************************************************************/
#include<iostream>
#include<cstdio>
#include<cstring>
#include<algorithm>
using namespace std;
const int INF = 0x3f3f3f3f;
const int N = 1005;
int n,x,y,z;
struct Node
{
    int from , to, w, next;
}edge[N*N];
int cnt, head[N];

void init()
{
    cnt = 0;
    memset(head, -1, sizeof(head));
}
void add(int u , int v , int w)
{
    edge[cnt].from = u;
    edge[cnt].to = v;
    edge[cnt].w = w;
    edge[cnt].next = head[u];
    head[u] = cnt++;
}

int dis[N], pre[N], vis[N], id[N];
int direct_MST(int u)
{
    int ans = 0;
    while( 1 )
    {
        for(int i=0;i<n;i++)
            dis[i] = INF;
        for(int i=0;i<cnt; i++)
        {
            int u = edge[i].from, v = edge[i].to, w = edge[i].w;
            if(w < dis[v] && u!=v)
            {
                pre[v] = u;
                dis[v] = w;
            }
        }
        for(int i=0;i<n;i++)
            if(i!=u && dis[i]==INF) return -1;

        int num = 0;
```

```
        memset(vis, -1, sizeof(vis));
        memset(id, -1, sizeof(id));
        dis[u] = 0;
        for(int i=0;i<n;i++)
        {
            ans += dis[i];
            int v = i;
            while(vis[v] != i && id[v] ==-1 && v!=u)
            {
                vis[v] = i;
                v = pre[v];
            }
            if(v!=u && id[v]==-1)
            {
                for(int j = pre[v]; j!=v; j=pre[j])
                    id[j] = num;
                id[v] = num++;
            }
        }
        if(num == 0 ) break;
        for(int i = 0;i < n ; i++)
            if(id[i] == -1)
                id[i] = num++;
        for(int i = 0; i < cnt;i++)
        {
            int v = edge[i].to;
            edge[i].from = id[edge[i].from];
            edge[i].to = id[edge[i].to];
            if(edge[i].from!= edge[i].to) edge[i].w -= dis[v];
        }
        n = num;
        u = id[u];
    }
    return ans;
}

struct IN
{
    int x, y, z;
}In[N];
int Dis(int a,int b)
{
    return abs(In[a].x-In[b].x) + abs(In[a].y-In[b].y) + abs(In[a].z - In[b].z);
}

int main()
{
    while(scanf("%d%d%d%d", &n,&x, &y,&z)!=EOF)
    {
        if(x==0 && y==0 && n==0 && z==0) break;


        init();
        for(int i=1; i <= n; i++)
        {
            scanf("%d%d%d", &In[i].x, &In[i].y, &In[i].z);
            add(0, i, In[i].z*x);
        }
        for(int i = 1; i <= n; i ++)
        {
            int k, v;
            scanf("%d" , &k);
            while(k--)
            {
                scanf("%d", &v);
                if(v==i) continue;
                int w = Dis(i, v)*y;
                if(In[v].z > In[i].z ) w += z;
                add(i, v, w);
            }
        }
```

```
        n++;
        // 加了一个点，现在的点编号从0开始。 到n-1.
        int ans = direct_MST(0);
        printf("%d\n", ans);
    }
    return 0;
}
```

3.6 生成树计数

生成树计数是在无向图中，计算生成树的个数的算法。利用拉普拉斯算子进行计算。这里要记录一下理论基础。
首先是图的关联矩阵的概念。设图的关联矩阵是 B，行为点，列为边，对于每个边 $Edge_i(u,v)$，我们将 B[u][i] 置为 1，B[v][i] 置为 -1，该列的其他位置为 0。这就是图的关联矩阵。
关联矩阵自己并没有什么，但是和其转置矩阵相乘可以获得图的拉普拉斯算子。转置矩阵就是把行变成列，把列变成行。
相乘后的矩阵对角线上是每个点的度数，其他的位置，如果 u 和 v 之间有边，对应位置是 -1。快速得到的方式是用图的度数矩阵减去联通性矩阵。
度数矩阵是只有对角线处有每个点的度数的值，联通性矩阵就是有边为 1，无边为 0。
下面就是理论基础，矩阵 -树定理。对于一个图的生成树的个数就是其拉普拉斯算子的任意一个 $n-1$□□□□□□□□□□□□□□。所谓 $n-1$□□□□□□□□□ 就是选则一个 i，把 i 行和 i 列删掉后的行列式。
下面我们来看一下一些好玩的性质。我们那设拉普拉丝算子矩阵为 C。
1. C 的行列式总是为 0.
2. 如果是不联通图，其 $n-1$□□□□□□□□□ 的值为 0.
3. 如果是一颗树，其 $n-1$□□□□□□□□□ 是 1。
这里有一个很重要的公式，Binet-Cauchy 定理。还没有时间去研究。
算法复杂度是 $O(n^3)$。

```
#include<iostream>
#include<math.h>
#include<stdio.h>
#include<string.h>
using namespace std;

#define zero(x) ((x>0? x:-x)<1e-15)

int const MAXN = 100;

double a[MAXN][MAXN];
double b[MAXN][MAXN];

int g[53][53];
int n,m;

double det(double a[MAXN][MAXN], int n) {
    int i, j, k, sign = 0;
    double ret = 1, t;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            b[i][j] = a[i][j];
    for (i = 0; i < n; i++) {
        if (zero(b[i][i])) {
            for (j = i + 1; j < n; j++)
                if (!zero(b[j][i]))
                    break;
            if (j == n)
                return 0;
            for (k = i; k < n; k++)
                t = b[i][k], b[i][k] = b[j][k], b[j][k] = t;
            sign++;
        }
        ret *= b[i][i];
        for (k = i + 1; k < n; k++)
            b[i][k] /= b[i][i];
```

```
        for (j = i + 1; j < n; j++)
            for (k = i + 1; k < n; k++)
                b[j][k] -= b[j][i] * b[i][k];
    }
    if (sign & 1)
        ret = -ret;
    return ret;
}
void build(){
    while (m--) {
        int a, b;
        scanf("%d%d", &a, &b);
        // 点的编号从0开始。
        g[a-1][b-1]=g[b-1][a-1]=1;
    }
}
int main() {
    int cas;
    scanf("%d", &cas);
    while (cas--) {
        scanf("%d%d", &n, &m);
        memset(g,0,sizeof(g));
        build();
        memset(a,0,sizeof(a));

        //计算度数
        for (int i=0;i<n;i++)
        {
            int d=0;
            for (int j=0;j<n;j++)
                if (g[i][j])
                    d++;
            a[i][i]=d;
        }
        // 计算拉普莱斯算子。
        for (int i=0;i<n;i++)
            for (int j=0;j<n;j++)
                if (g[i][j])
                    a[i][j]=-1;
        //由于数据比较大，所以使用double。
        double ans = det(a, n-1);

        printf("%0.0lf\n", ans);
    }
    return 0;
}
```

```
/*************************************************************************
带取模的行列式求值，  注意求行列是的时候逆序数的问题。
这道题需要用计算集合预处理出图，在R范围内且两点之间没有其他点的时候才有连边。
 *************************************************************************/
#include<iostream>
#include<cstring>
#include<algorithm>
#include<cstdio>
#include<cmath>
using namespace std;
typedef long long ll;
const int N = 305;
const int MOD = 10007;
int R;
struct Point
{
    int x, y;
    Point(int x=0, int y=0):x(x), y(y){}
};

Point operator - (Point a , Point b){ return Point(a.x-b.x, a.y-b.y);}
int Lenth(Point a) { return a.x*a.x + a.y*a.y; }
int Cross(Point a, Point b)  {return a.x*b.y- a.y*b.x; }
```

```
int Dot(Point a, Point b) { return a.x*b.x+a.y*b.y; }
int OnSeg(Point a, Point b, Point c)
{
    if(Cross(a-b, b-c) !=0 ) return 0;
    if(Dot(a-b, a-c) < 0) return 1;
    return 0;
}
Point point[N];
int dis[N][N], inv[MOD+10], vis[N], a[N][N];

void Getdis(int n)
{
    memset(dis, 0, sizeof(dis));
    int flag ;
    for(int i=0;i<n;i++)
        for(int j=i+1; j<n;j++)
            if(Lenth(point[i] - point[j]) <= R*R)
            {
                flag = 0;
                for(int k=0;k<n;k++)
                    if(i==k || j==k) continue;
                    else if(OnSeg(point[k], point[i], point[j]))
                    { flag = 1; break; }
                if(flag==0 ) dis[i][j] = dis[j][i] = 1;
            }
}

void Exgcd(int a, int b, int &x, int &y)
{
    if(b==0) { x = 1; y = 0; return ; }
    Exgcd(b, a%b, x, y);
    int t = y;
    y = x - a/b*y;
    x = t;
}

int Dfs(int u, int n)
{
    int sum = 1;
    vis[u] = 1;
    for(int i=0;i<n;i++)
    {
        if(!vis[i] && dis[u][i]==1)
            sum+= Dfs(i,n);
    }
    return sum;
}

int det(int n)
{
    for(int i=0;i<n;i++) for(int j=0;j<n;j++) a[i][j] = (a[i][j]+MOD)%MOD;

    int ans = 1;
    for(int i=0;i<n;i++)
    {
        int k=i;
        for(int j=i+1;j<n;j++)
            if(abs(a[j][i]) > abs(a[k][i])) k = j;

        if(k !=i ) for(int j= i; j< n; j++) swap(a[i][j], a[k][j]);
        if(k !=i ) ans = -ans;
        if(ans < 0) ans += MOD;
        if(a[i][i] == 0) return -1;
        ans = (ans*a[i][i]%MOD + MOD) % MOD;
        a[i][i] = (a[i][i] + MOD) % MOD;
        for(int j=i+1; j<n;j++)
            if( a[j][i] != 0 )
            {
                a[j][i] = a[j][i]*inv[a[i][i]]%MOD;
                for(int k=i+1; k<n; k++ )
                    a[j][k] = ((a[j][k] - a[j][i]*a[i][k])%MOD + MOD)%MOD;
```

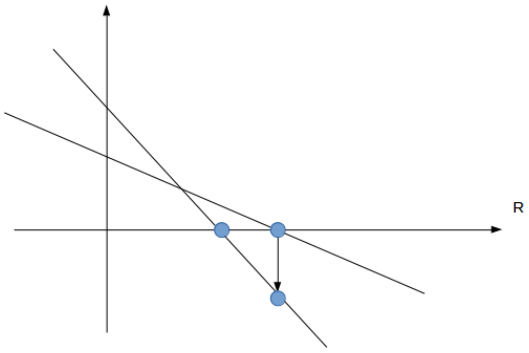
```
            }
        }
        return (ans%MOD + MOD)%MOD;
    }
    int main()
    {
        for(int i=1;i<MOD; i++)
        {
            // a*x + p*y = 1, x就是a对p的逆元
            int x, y;
            Exgcd(i, MOD, x, y);
            inv[i] = (x%MOD+MOD)%MOD;
        }

        int T;
        scanf("%d", &T);
        while(T--)
        {
            int n;
            scanf("%d%d", &n,&R);
            for(int i=0;i<n;i++)
                scanf("%d%d", &point[i].x, &point[i].y);

            Getdis(n);
            memset(vis, 0, sizeof(vis));
            if( Dfs(0, n) != n) { printf("-1\n"); continue; }

            memset(a, 0, sizeof(a));
            for(int i=0;i<n;i++)
                for(int j=0;j<n;j++)
                    if(dis[i][j]) { a[i][j] = -1; a[i][i]++; }
            int ans = det( n-1);
            printf("%d\n", ans);
        }
        return 0;
    }
```

3.7 最优比例生成树 - 0-1 分数规划

0-1 分数规划，是这样一类问题。给两个数组 a,b，分别表示价值和代价，现在要求选出一部分，我们用 $x[i]=1$ 表示选则该物品，$x[i]=0$ 表示不选则该物品，使 $\frac{\sum_{i=1}^{i=N}(a[i]*x[i])}{\sum_{i=1}^{i=N}(b[i]*x[i])}$ 最大。也就是所有物品的总价值比总代价值最大或最小。
这个问题主要包含三个问题，0-1 分数规划、最优比例生成树、最优比例环。我们这里用最优比例生成树为例子。

我们先来进行一下变形：
令：$R=\frac{\sum_{i=1}^{i=N}(a[i]*x[i])}{\sum_{i=1}^{i=N}(b[i]*x[i])}$。
令：$F(R)=\sum_{i=1}^{i=N}(a[i]*x[i])-R*\sum_{i=1}^{i=N}(b[i]*x[i])$。
则：$F(R)=\sum_{i=1}^{i=N}((a[i]-R*b[i])*x[i])$。
令：$D(R)=a[i]-R*b[i]$。
因为 b 数组是正数，所以这个 D 函数是个单调递减的函数。
上面是一些准备工作，下面我们来看一下 $F(R)$，如果 $F(R)>0$，则可知，必然存在一种 x，使新的取法获得的 R 值更大，即更优的答案。
这里需要强调一下的就是，我们关注的只有 R 的大小，其他我们定义的函数都是为这个服务的，奇遇的函数值大小都是没有任何意义的。
所以对于一个 R，只要我们求出一组 x，使得 $F(R)>0$，那么就可以证明存在一个更优的 R，直到 $F(R)=0$，此时的 R，就是我们要的最优解。
很明显我们有两种方法。第一个是二分，直接二分 R 的值，判断函数正负；第二个就是迭代法。表示迭代法比较麻烦，还是用二分吧，毕竟验证一个解和求一个解的复杂度是不一样的。
对于最小比例生成树来说，确定一个 R 后，我们将图中的边的边权改为：$a[i]-R*b[i]$，也就是改成 $D(R)$；

然后求最小生成树。
至于为什么是最小生成树呢？我找了很多博客，都没有说这个问题。我们需要来读一下题，题目的意思是，对于所有的生成树来说，我们取 R 值最小的那棵生成树。每一棵生成树都对应一个 R，且这个 R 满足：$(a[i]*x[i]-R*b[i]*x[i])=0$，对于这棵生成树上的所有边都是成立的。对于我们二分的 R，这个 R 必须要对应一个使得 $F(R)=0$ 的 x 数组，这个 R 才是合法的。如果对于这个 R，我们求最小生成树都是大于 0，那么这个 R 肯定是不合法的，而且是比合法的小。如果我们求最小生成树是小于 0 的，那么可能是存在一个 x，使得当前 R 是合法的，但是不能确定，如果我们要去验证的话，复杂度会爆炸。这个时候我们来看 $F(R)$ 这个函数，这是一个单调递减的函数，所以，如果求得的最小生成树是小于 0 的，那么无论这个 R 是否合法，一定存在一个更小的 R，也就是图像与横轴交点，使得这个 R 合法，所以只要最小生成树是小于 0 的，一定不是最优解。综上我们可以证明只有当最小生成树是 0 的时候才是合法的，在多个最小生成树为 0 的 R 中取最小的那个。下面给一个图像就可以看出来了。



对于每组 X 数组，都有一个与 R 轴的交点，也就是这个对应 X 的生成树的比例。对于同一个 R 一定有很多组 X，分别对应不同的 $F(R)$，最右边的点 R，我们求他的最小生成树，事实上是求了这个点上对于所有生成树来说的下届，如果下届是小于 0 的。也就是右下方的那个点，这个点对应的 X 一定存在一个比这个 R 更小的 R'，是与 R 轴的交点，是个合法解，所以一定会选择 R'。这就解释了函数值小于 0 的判断条件了。
数学是神器！
如果要求最大比例生成树，只要改成求最大生成树就可以了。

下面给最小比例生成树的代码：

```
1
2  //最小生成树部分要预处理边，处理方法在上面。
3
4  const double EPS = 1e-15;
5  double ratio_MST()
6  {
7     double l = 0, r = 1e6;
8     while(r - l > EPS)
9     {
10       double mid= (l + r) / 2.0;
11       if(prim(mid) < 0) r = mid;
12       else l = mid;
13    }
14    return (l+r)/2.0;
15 }
```

如果是数组中取 k 个的话，只要把 D 数组处理出来，取前 k 个就可以了。求最大就取最大 k 个，最小就取最小 k 个；

最优比例环的问题，就是在图中边上有花费，点上有收益，点和边都可以经过多次，但是花费叠加，收益不叠加。让你找出一个环，使得比例最大或最小。
很复杂的东西，首先我们要确定一个的就是，最后的答案一定是个单环，不会是环套环。这个事实上很多关于图论中环的问题都是这样的，环套环的数值处理会变得很难，一般也不会用环套环，所以不会做的时候就尝试证明单环才是我们要的，往往能证明出来。
由于是环，所以点数和边数是一样的，我们可以将点和边统一到点上或边上。
然后我们计算出 D 数组，我们只需要找环，找正环。正环不好找，我们可以找负环，如果有负环说明原来是有正环的。就和最小生成树那个一样了。

3.8 树的直径

树的直径就是树上相距最远的两个点之间的距离。从任意一点开始 Dfs 到最远的地方 u，然后从 u 开始 Dfs 走到最远的 v，之间的路径就是树的一个直径。

3.9 强连通分量

强联通分量是指在有向图中，如果，两个点之间至少有一条边能够互相到达，那么这两个点就是强联通分量。主要使用 Tarjan 的算法。常用的使用方式是强联通缩点。然后会得到一个 DAG，然后就是 DAG 上的处理了。

3.9.1 Tarjan

```
/*************************************************************************
题目就是给一个有向图，如果是强联通中的点之间传递信息不需要花费，不是双联通的点之间传递信息需要花费。
     求传递整个图的最小花费， 我们先使用强联通缩点。然后求一个最小树形图。这里处理DAG的最小树形图，可以直接使用Dp
     的思路进行。

在DAG中，就是一个塔状的结构，每个点找距离他最近的父亲。最后把这些边加起来。
 *************************************************************************/
#include<iostream>
#include<cstdio>
#include<cstring>
#include<algorithm>
using namespace std;
const int N = 50005;
const int M = 100005;

int n,m, dfn[N], low[N], sta[N<<1], top, Dindex, Bcnt, vis[N], Belong[N], ans[N];

struct Node
{
    int from, to, next, w;
}edge[M<<1];
int cnt, head[N];

void init()
{
    cnt = 0;
    memset(head, -1, sizeof(head));
}

void add(int u, int v, int w)
{
    edge[cnt].from = u;
    edge[cnt].to = v;
    edge[cnt].w = w;
    edge[cnt].next = head[u];
    head[u] = cnt++;
}

void tarjan(int u)
{
    dfn[u] = low[u] = ++Dindex;
    vis[u] = 1;
    sta[top++] = u;
    for(int i=head[u]; i != -1; i =edge[i].next)
    {
        int v = edge[i].to;
        if(!dfn[v])
        {
            tarjan(v);
            low[u] = min(low[u], low[v]);
        }
        else if(vis[v] && dfn[v]<low[u])
```

```
            low[u] = dfn[v];
    }
    if( dfn[u] == low[u])
    {
        Bcnt++;
        int v;
        do
        {
            v = sta[--top];
            vis[v] = 0;
            Belong[v] = Bcnt;
        }while( v != u );
    }
}

void solve_tarjan()
{
    //强联通分量编号从1开始。
    Bcnt = top = Dindex = 0 ;
    memset(dfn, 0, sizeof(dfn));
    for(int i = 0 ; i < n ; i++)
        if( !dfn[i] )
            tarjan(i);
}


int main()
{
    while(scanf("%d%d", &n,&m)!=EOF)
    {
        init();
        for(int i = 0;i < m; i++)
        {
            int u,v,w;
            scanf("%d%d%d", &u, &v, &w);
            add(u,v,w);
        }

        solve_tarjan();
        memset(ans, 0x3f, sizeof(ans));
        for(int i=0;i < m;i++)
        {
            int u = edge[i].from, v = edge[i].to, w = edge[i].w;
            if(Belong[u] == Belong[v]) continue;
            ans[Belong[v]] = min(ans[Belong[v]], w);
        }
        ans[Belong[0]] = 0;
        int sum = 0;
        for(int i = 1; i<= Bcnt;i++) sum += ans[i];
        printf("%d\n", sum);

    }
    return 0;
}
```

3.10 割点桥双连通分量

割点就是去掉这个点后联通分量会增加，桥就是删掉这个边后会增加联通分量。

```

//链式前向星建图，双向边。
//桥的标记是把正向和反向边一起标记掉。
//桥和割点一起求出来的，复杂度是O(V+E)

#include<iostream>
#include<cstring>

int bridge[M],cut[N];
```

```
int low[N], dfn[N], vis[N];

void cut_bridge(int u, int fa, int dep)
{
    vis[u] = 1;
    dfn[u] = dep;
    low[u] = dep;
    int son=0;
    for(int i=head[u]; i!=-1; i=edge[i].next)
    {
        int v = edge[i].to;
        if(vis[v]==1 && v!=fa)
            if(dfn[v] < low[u])
                low[u] = dfn[v];
        if(vis[v]==0)
        {
            cut_bridge(v, u, dep+1);
            son++;
            if(low[v] < low[u]) low[u] = low[v];
            if( (fa==-1 && son>1 ) || (fa!=-1 && low[v]>= dfn[u]))
                cut[u] = 1;
            if(low[v]>dfn[u]) bridge[i] = bridge[i^1] = true;
        }
    }
    vis[u] = 2;
}
```

下面是几个概念的辨析。在大白书中 P312，有详细的证明算法的过程，不懂可以去复习。

双联通分量就是任意两个点之间有两条路可达，两条路没有重合的地方。

点双联通分量：图中没有割点。

边双联通分量：图中没有桥。

双联通分量中，割点可以出现在多个双联通分量。一条边只能属于一个双联通分量。我们用边集来表示双联通分量。属于这个边集的边和他的点构成这个双联通分量。

3.10.1 点双连通分支

下面是求点双联通的代码。

```
/*****************************************************************************
给一个无向图，可能不是联通的，如果有两个环路共用一条路，这就是冲突边，桥就是多余边。
让你求冲突边和桥的数量。
这是一个点双联通分量，因为每条边都只属于一个点双联通分量，对于在一个双联通分量中的点和边，如果点数不等于边数，
    那么所有的边都是冲突边了。所以最后判断一下就好了。
 *****************************************************************************/
#include<iostream>
#include<cstdio>
#include<cstring>
#include<algorithm>
#include<vector>
using namespace std;
const int N = 10005<<2;
const int M = 100005<<2;
int S[M], top=0;
int dfn[N], low[N];
bool vis[M];
int n, m, ind, T, sum1, sum2;

struct Edge
{
    int from, to,next;
}edge[M];

int cnt,head[N];
vector<int> block[N];
void init()
{
    cnt=0;
    memset(head, -1, sizeof(head));
```

```
    for(int i=0;i<n;i++) block[i].clear();
    memset(dfn, -1, sizeof(dfn)); //第一次遍历到的顺序编号。
    sum1 = sum2 = 0;
    T = ind = 0; // T是dfs的计数器。ind是块（联通分量）计数器。
    top=0; //栈初始化。
}
void add(int u, int v)
{
    edge[cnt].to = v;
    edge[cnt].from = u;
    edge[cnt].next = head[u];
    head[u] = cnt++;
}


void tarjan(int u, int pre)
{
    dfn[u]=low[u]=T++;
    for(int i=head[u]; i!=-1; i=edge[i].next)
    {
        int v=edge[i].to;
        if(dfn[v]==-1)
        {
            S[++top] = i;
            tarjan(v, u);
            if(low[u]>low[v]) low[u]=low[v];
            if(dfn[u]<=low[v])
            {
                for(int j; top>=0 ; )
                {
                    j = S[top--];
                    if(dfn[edge[j].from] < dfn[v]) break;
                    block[ind].push_back(j);
                }
                block[ind++].push_back(i);
                if(dfn[u]<low[v]) sum1++; //这是一个桥。
            }
        }
        else if(v!=pre && dfn[v]<dfn[u])
        {
            S[++top] = i;
            if(low[u]>dfn[v]) low[u]=dfn[v];
        }
    }
}

void solve()
{
    for(int i=0;i<n;i++)
        if(dfn[i]==-1)
            tarjan(i, -1);
    //共有ind块点双联通分量。
    for(int i=0;i<ind; i++)
    {
        memset(vis, 0,sizeof(vis));
        int len = block[i].size();
        int tot=0;
        for(int j=0;j<len;j++)
        {
            int u = edge[block[i][j]].from;
            int v = edge[block[i][j]].to;
            if(!vis[u]) vis[u]=1, tot++;
            if(!vis[v]) vis[v]=1, tot++;
        }
        if(len > tot) sum2+=len;
    }
    printf("%d␣%d\n", sum1, sum2);
}

int main()
{
```

```
    while(scanf("%d%d", &n,&m)!=EOF)
    {
        if(n==0 && m==0) break;
        init();
        for(int i=0;i<m;i++)
        {
            int u, v;
            scanf("%d%d", &u, &v);
            add(u,v);
            add(v,u);
        }
        solve();
    }
    return 0;
}
```

3.10.2 边双连通分支

求边双联通分量比较简单，只要跑两遍 dfs，因为边双联通就是没有桥的联通块。所以第一遍先跑出来所有的桥，然后第二次 dfs 的时候碰到桥就不走。然后分别跑联通块就好了。

```
 /*************************************************************************
hdu2460
这道题给一个无向图，然后有Q个询问，每次询问之前会加进一条新边，问每次加进新边后桥的个数。
因为所有的桥都必然会出现再dfs树上，所以我们记录每个点在dfs树上的父亲和深度，借助LCA的思路把路径上的桥全变成非桥就好了。
桥的表示上，除了用边，在dfs树上还可以使用下结点来唯一标示。
 *************************************************************************/
#include<iostream>
#include<cstdio>
#include<cstring>
#include<algorithm>
using namespace std;
const int N = 100005;
const int M = 2000005;
int n,m,Q;
struct Edge
{
    int from , to, next;
}edge[M<<2];
int cnt,head[N];
void init()
{
    cnt=0;
    memset(head, -1, sizeof(head));
}
void add(int u, int v)
{
    edge[cnt].from = u;
    edge[cnt].to = v;
    edge[cnt].next = head[u];
    head[u] = cnt++;
}
int bridge[M],f[N], b[N];
int low[N],dfn[N],vis[N];
void Bridge(int u, int fa,int dep)
{
    f[u] = fa;
    vis[u]=1;
    dfn[u] = low[u] = dep;
    for(int i=head[u]; i!=-1; i = edge[i].next)
    {
        int v = edge[i].to;
        if(vis[v]==1 && v!=fa)
            if(dfn[v] < low[u])
                low[u] = dfn[v];
        if(!vis[v])
        {
```

```
            Bridge(v, u, dep+1);
            if(low[v] < low[u]) low[u] = low[v];
            if(low[v] > dfn[u])
            {
                b[v] = 1;
                bridge[i] = bridge[i^1] = true;
            }
        }
    }
    vis[u] = 2;
}
int LCA(int A,int B)
{
    int ans=0;
    if(dfn[A] < dfn[B])
        swap(A,B);
    while(dfn[A] > dfn[B])
    {
        ans += b[A];
        b[A]=0;
        A = f[A];
    }
    while(A!=B)
    {
        ans+= b[A] + b[B];
        b[A] = b[B] = 0;
        A = f[A];
        B = f[B];
    }
    return ans;
}
int main()
{
    int k=1;
    while(scanf("%d%d", &n,&m)!=EOF)
    {
        if(n==0 && m==0 ) break;
        printf("Case␣%d:\n",k++);

        init();
        for(int i=0;i<m;i++)
        {
            int u,v;
            scanf("%d%d", &u, &v);
            add(u,v);
            add(v,u);
        }

        memset(vis, 0,sizeof(vis));
        memset(bridge, 0,sizeof(bridge));
        memset(b, 0,sizeof(b));
        memset(dfn, -1,sizeof(dfn));
        memset(low, -1,sizeof(low));
        memset(f, -1,sizeof(f));
        Bridge(1, -1, 0);
        int num=0;
        for(int i=1;i<=n;i++)
            if(b[i]==1) num++;
        scanf("%d", &Q);
        while(Q--)
        {
            int u, v;
            scanf("%d%d", &u,&v);
            int tmp = LCA(u,v);
            num-= tmp;
            if(num<0) while(1);
            printf("%d\n",num);
        }
        printf("\n");

    }
```

```
    return 0;
}
```

边双联通缩点，将边双联通分量看做是一个点，这样就可以变成树来操作。
如果要把图补成全边双联通，就是双联通分量树中，度数为 1 的个数加 1，然后除以 2。
坑点就是万一图不联通，会出现森林的情况。森林补成边双联通的就是在树之间补 n（n 棵树），搞成环。

```

#include<iostream>
#include<cstdio>
#include<cstring>
#include<algorithm>
using namespace std;
const int N = 5010<<2;
const int M = 10005<<2;
int n,m;
struct Node
{
    int from, to, next;
}edge[2][M];
int cnt[2],head[2][N];
void init()
{
    cnt[0] = cnt[1]= 0;
    memset(head, -1, sizeof(head));
}
void add(int t,int u, int v)
{
    edge[t][cnt[t]].from =u;
    edge[t][cnt[t]].to = v;
    edge[t][cnt[t]].next = head[t][u];
    head[t][u] = cnt[t]++;
}

int bridge[M], vis[N], low[N], dfn[N];
void Bridge(int u, int fa, int dep)
{
    vis[u] = 1;
    low[u] = dfn[u] = dep;
    for(int i=head[0][u]; i!=-1; i=edge[0][i].next)
    {
        int v = edge[0][i].to;
        if(vis[v]==1 && v!=fa)
            if(low[v] < dfn[u])
                low[u] = low[v];
        if(!vis[v])
        {
            Bridge(v,u, dep+1);
            low[u] = min(low[u], low[v]);
            if(low[v] > dfn[u])
                bridge[i] = bridge[i^1] = 1;
        }
    }
    vis[u] = 2;
}
int fa[N];

void Dfs(int u, int Fa)
{
    fa[u] = Fa;
    for(int i=head[0][u]; i!=-1; i=edge[0][i].next)
    {
        if(bridge[i]==1)
            continue;
        int v = edge[0][i].to;
        if(fa[v]==-1)
            Dfs(v, Fa);
    }
}
int du[N];
```

```
int main()
{
    scanf("%d%d", &n,&m);
        init();
        while(m--)
        {
            int u, v;
            scanf("%d%d", &u,&v);
            add(0, u, v);
            add(0, v, u);
        }
        memset(vis, 0,sizeof(vis));
        memset(dfn, -1,sizeof(dfn));
        memset(low, -1,sizeof(low));
        memset(bridge, 0,sizeof(bridge));
        Bridge(1, -1, 0);

        memset(fa,-1,sizeof(fa));
        int num=1;
        int ans=0;
        for(int i=1;i<=n;i++ )
        {
            if(fa[i]==-1)
            {
                Dfs(i, num);
                num++;
            }
        }
        memset(du, 0, sizeof(du));
        for(int i=0;i<cnt[0];i+=2)
        {
            if(bridge[i]==1 && bridge[i^1]==1)
            {
                int u = edge[0][i].from;
                int v = edge[0][i].to;
                if(fa[u]==fa[v]) continue;
                du[fa[u]]++;
                du[fa[v]]++;
            }
        }
        for(int i=1;i<num;i++)
        {
            if(du[i]==1)
                ans++;
        }
        printf("%d\n",(ans+1)/2);

    return 0;
}
```

3.11 二分图

二分图有这样的性质。我们把同一边的两个点染上不同的颜色，有一种染色方式，可以使得所有的边的两个短点都是相异的颜色。也就是说，图中的点会分成两堆，所有的边都是从一堆连往另一堆，堆内的点之间没有连边。这就是二分图，我们可以用二分染色的方式判别二分图。

对于二分图我们需要区分几个概念，很容易混淆。

1. 最大匹配。给定一个二分图 G，在 G 的一个子图 M 中，M 的边集 E 中的任意两条边都不依附于同一个顶点，则称 M 是一个匹配。选择这样的子集中边数最大的子集称为图的最大匹配问题 (maximal matching problem)。
2. 完美匹配。如果一个匹配中，图中的每个顶点都和图中某条边相关联，则称此匹配为完全匹配，也称作完备，完美匹配。
3. 点覆盖。点覆盖是一个点集，使得该图中的所有的边都被覆盖掉的点集。当我们覆盖一个点的时候我们覆盖与该点所有相连的边，能够覆盖所有的边的点集是点覆盖集。极小点覆盖就是指，该点集的任意真子集都不是点覆盖集。最小点覆盖是指点数最小的覆盖集。

4. 边覆盖与点覆盖差不多，选边去覆盖点，是的所有的点都被覆盖到。极小边覆盖和最小边覆盖也是一样的概念。
5. 独立集。独立集是指一个点集，独立集中的点之间没有边，也就是两两之间不相邻，注意不是不联通。极大独立集就是，再加任何的点进来都不是独立集。最大独立集就是点最多的独立集，独立数就是指独立集的点数。
6. 团。团也是一个点集，与独立集相反，这个是团中点两两之间有边相连，也就是两两相邻，是一个完全图。极大团，最大团，团数，一样是参考独立集去理解。
7. 边独立集。是一个边集，其中任意两条边不相邻接，也就是任意两条边没有公共点。这其实就是匹配，极大，最大，边独立数，一样参考独立集的定义。
8. 支配集。支配集是一个点集，使得所有不在该集合中的点至少有一个相邻点在该集合中。也就是每条边都至少有 1 个点在支配集中。某些点支配了所有的点。极小支配集，最小支配集，支配数。
9. 边支配集。是一个边集，使得图中所有的边，至少有一个邻接边在支配集中。极小边支配集，最小边支配集，边支配数等。
10. 最小路径覆盖。就是使用路径去覆盖点。用尽量少的路径去覆盖所有的点，要求路径不能相交，每个点必须只属于一条路径，路径可以是一个点。□□□□□□□□ = □□ − □□□□□□□□□□□□，所以要使得边尽量多。我们使用一个方法，把所有的点拆成两部分，一部分在左一部分在右，然后把左右的边都建成从左到右的，然后求二分图最大匹配，最大匹配的边数就是路径覆盖中的边数，所以 □□□□□□□□ = □□ − □□□□□□□□。

二分图的一些性质。
1. 最大匹配数 = 最小点覆盖数。
2. 点独立数 = 顶点数 - 最大匹配数
3. DAG 的最小路径覆盖 = 点数 - 拆点最大匹配数。上面介绍过了。路径就按照匹配边走。
4. 最大匹配数 = 左边匹配点 + 右边未匹配点
5. 最小边覆盖 = 图中点的个数 - 最大匹配数 = 最大独立集。

3.11.1 二分图最大匹配 -匈牙利

算法思路：
找增广路，每次将增广路延长 1。直到找不到增广路。
模板有些地方需要注意，我们从左边向右边建有向边，并且记录左边点的个数。
linker 代表该点的匹配点的编号。
每次从左边的一个点出发，寻找增广路。每次 DFS 的起点也都是左边的点，终点肯定是右边的点，如果右边的点是一个未匹配点，那么我们肯定是找到了一个增广路。变换一下 linker，然后退出。linker 是右边的点的匹配点。
算法复杂度是 $O(N*M)$。

```
 /*
 * 经典的二分图最大匹配的问题。也可以深搜来做。
 * 在一个网格中有墙，我们放车在空位上，最多放多少。
 * 车的攻击范围是直线，所以我们把相交的横和纵建边，然后求最大匹配。
 *
 */
#include<iostream>
#include<cstdio>
#include<cstring>
#include<algorithm>

using namespace std;

int n;
char a[10][10];
int b[10][10];
struct Node
{
    int to,next;
}edge[10000];
int cnt =0,head[10000];
void init()
{
    cnt=0;
    memset(head, -1,sizeof(head));
```

```
}
void add(int u,int v)
{
    edge[cnt]. to = v;
    edge[cnt].next = head[u];
    head[u] = cnt++;
}
int xn;

// 二分图最大匹配模板。
int linker[1000];
int vis[1000];
bool Dfs(int u)
{
    for(int i= head[u]; i!=-1; i=edge[i].next)
    {
        int v = edge[i].to;
        if(!vis[v])
        {
            vis[v] = 1;
            if(linker[v]==-1 || Dfs(linker[v]))
            {
                linker[v] = u;
                return 1 ;
            }
        }
    }
    return false;
}
int hungary()
{
    int ans =0 ;
    memset(linker,-1,sizeof(linker));
    for(int i=1;i<=xn;i++)
    {
        memset(vis,0,sizeof(vis));
        if(Dfs(i)) ans++;
    }
    return ans;
}

int main()
{
    while(scanf("%d", &n)!=EOF &&n)
    {
        init();
        for(int i = 0; i < n ;i++)
            scanf("%s", a[i]);
        memset(b,0,sizeof(b));
        int cnt = 0;
        // 给所有的竖连续的空白编号。
        for(int i= 0 ; i <n ;i++)
        {
            for(int j=0;j<n;j++)
            {
                if(a[j][i] == 'X')
                    cnt++;
                else b[j][i] = cnt;
            }
            cnt++;
        }
        // 给所有的横着连续的空白编号，并且向竖空白建边。
        if(a[0][0]=='X')
            cnt = 0;
        else cnt = 1;
        for(int i = 0; i <n;i++)
        {
            for(int j = 0; j < n ;j ++)
            {
                if(a[i][j]=='X')
                    cnt++;
```

```
            else
                add(cnt, b[i][j]);
        }
        cnt ++;
    }
    xn = cnt; // xn 是左边点的个数。
    int ans = hungary();
    printf("%d\n",ans);
  }
  return 0;
}
```

```
 /*************************************************************************
有向图二分图染色后求最大匹配。

算法过程没有什么，计算的时候以每个点都作为起点进行增广路，
计算出的结果是有向图的2倍。所以要除以2.

 *************************************************************************/
#include<cstdio>
#include<iostream>
#include<cstring>
#include<algorithm>
using namespace std;
const int N = 205;
struct Node
{
  int to,next;
}edge[N*N];
int cnt , head[N];

void init()
{
  cnt = 0;
  memset(head, -1 ,sizeof(head));
}
void add(int u,int v)
{
  edge[cnt].to = v;
  edge[cnt].next = head[u];
  head[u] = cnt++;
}


int n,m;
int color[N];
int linker[N];
int xn;
// 二分图染色
bool Dfs_color(int u)
{
  for(int i = head[u]; i != -1; i = edge[i].next)
  {
    int v = edge[i].to;
    if(color[v] == -1)
    {
      color[v] = !color[u];
      if( ! Dfs_color( v ) )
        return false;
    }
    else if(color[v] == color[u])
      return false;
  }
  return true;
}
int vis[N];
bool Dfs(int u)
{
  for(int i = head[u] ; i != -1; i = edge[i].next)
  {
```

```
        int v = edge[i].to;
        if(!vis[v])
        {
            vis[v] = 1;
            if(linker[v] == -1 || Dfs( linker[v]))
            {
                linker[v] = u;
                return true;
            }
        }
    }
    return false;
}
int hungray()
{
    int ans = 0;
    memset(linker,-1,sizeof(linker));
    for(int i = 1; i <= xn; i++ )
    {
        memset(vis,0,sizeof(vis));
        if(Dfs(i))
            ans++;
    }
    return ans;
}
int main()
{
    while(scanf("%d%d",&n,&m)!=EOF)
    {
        if(n == 1)
        {
            printf("No\n");
            continue;
        }
        init();
        for(int i = 0;i < m ;i ++)
        {

            int u,v;
            scanf("%d%d",&u,&v);
            add(u,v);
            add(v,u);
        }
        memset(color,-1,sizeof(color));
        int flag=0;
//
        color[1] = 1;
        if(!Dfs_color(1)) flag = 1;

        if(flag==1)
        {
            printf("No\n");
            continue;
        }
        xn = n;
        int ans = hungray();
        printf("%d\n",ans/2);
    }
    return 0;
}
```

3.11.2 二分图最大匹配 -HK

算法复杂度是 $O(\sqrt{N} * M)$。
在有向图中也是一样的计算两边最大匹配。
初始化 cntx 就可以了。

```
```

```cpp
#include<iostream>
#include<cstdio>
#include<cstring>
#include<algorithm>
#include<queue>
using namespace std;
const int INF = 0x3f3f3f3f;

struct Node
{
    int to,next;
}edge[10000];
int cnt,head[1000];
int nx[1000],ny[1000];
int cntx,cnty,dis;
int dx[1000],dy[1000];
int linkx[1000],linky[1000];
int vis[1000];
void init()
{
    cnt =0;
    memset(head,-1,sizeof(head));
    memset(linkx,-1,sizeof(linkx));
    memset(linky,-1,sizeof(linky));
}
void add(int u,int v)
{
    edge[cnt].to = v;
    edge[cnt].next = head[u];
    head[u] = cnt++;
}

bool Bfs()
{
    queue<int> Q;
    memset(dx,-1,sizeof(dx));
    memset(dy,-1,sizeof(dy));
    dis = INF;
    for(int i=1;i<=cntx;i++)
        if(linkx[i]==-1)
        {
            Q.push(i);
            dx[i] = 0;
        }
    while(!Q.empty())
    {
        int u = Q.front();
        Q.pop();
        if(dx[u]>dis) break;
        for(int i=head[u];i!=-1;i=edge[i].next)
        {
            int v = edge[i].to;
            if(dy[v]==-1)
            {
                dy[v] = dx[u] +1;
                if(linky[v]==-1) dis = dy[v];
                else
                {
                    dx[linky[v]] = dy[v]+1;
                    Q.push(linky[v]);
                }
            }
        }
    }
    // printf("dis: %d\n",dis);
    return dis!=INF;
}
bool Dfs(int u )
{
    for(int i=head[u];i!=-1; i=edge[i].next)
    {
```

```
        int v = edge[i].to;
        if(!vis[v] && dy[v] == dx[u]+1)
        {
            vis[v] = 1;
            if(linky[v]!=-1 && dy[v]==dis) continue;
            else if(linky[v]==-1 || Dfs(linky[v]))
            {
                linkx[u] = v;
                linky[v] = u;
                return true;
            }
        }
    }
    return false;
}
int Maxcatch()
{
    int ans=0;
    while(Bfs())
    {
        memset(vis,0,sizeof(vis));
        for(int i=1;i<=cntx;i++)
            if(linkx[i]==-1 && Dfs(i))
                ans++;
    }
    return ans;

}
int main()
{
    init();
    scanf("%d%d", &cntx,&cnty);
    int m;
    scanf("%d", &m);
    for(int i=0;i<m;i++)
    {
        int u,v;
        scanf("%d%d", &u,&v);
        add(u,v);
    }
    int ans = Maxcatch();
    printf("Maxcatch:␣%d\n",ans);
    return 0;
}
```

3.11.3 二分图多重匹配

在二分图最大匹配中，每个点（不管是 X 方点还是 Y 方点）最多只能和一条匹配边相关联，然而，我们经常遇到这种问题，即二分图匹配中一个点可以和多条匹配边相关联，但有上限，或者说，Li 表示点 i 最多可以和多少条匹配边相关联。

二分图多重匹配分为二分图多重最大匹配与二分图多重最优匹配两种，分别可以用最大流与最大费用最大流解决。

（1）二分图多重最大匹配：

在原图上建立源点 S 和汇点 T，S 向每个 X 方点连一条容量为该 X 方点 L 值的边，每个 Y 方点向 T 连一条容量为该 Y 方点 L 值的边，原来二分图中各边在新的网络中仍存在，容量为 1（若该边可以使用多次则容量大于 1），求该网络的最大流，就是该二分图多重最大匹配的值。

（2）二分图多重最优匹配：

在原图上建立源点 S 和汇点 T，S 向每个 X 方点连一条容量为该 X 方点 L 值、费用为 0 的边，每个 Y 方点向 T 连一条容量为该 Y 方点 L 值、费用为 0 的边，原来二分图中各边在新的网络中仍存在，容量为 1（若该边可以使用多次则容量大于 1），费用为该边的权值。求该网络的最大费用最大流，就是该二分图多重最优匹配的值。

具体的网络流解法看网络流内容。

一般在点数不多的时候，我们使用匈牙利算法的改进版来求，代码量比较小。y 的点不再是 1 个匹配，可以匹配多个 x 点。这时候我们用下面的算法。

```
/*************************************************************************
num[] 是右边点的最大承载量
 *************************************************************************/

const int N = 1010; // 左边点数
const int M = 510; // 右边点数

int uN, vN;
int g[N][M], linker[M][N], vis[M], num[M];

bool Dfs(int u)
{
   for(int v=0; v< vN; v++)
   {
      if(g[u][v] && !vis[v])
      {
         vis[v] = 1;
         if(linker[v][0] < num[v])
         {
            //linker[v][0] 是计数器。
            linker[v][++linker[v][0]] = u;
            return true;
         }
      }
      for(int i=1;i<=num[v]; i++)
         if(Dfs(linker[v][i]))
         {
            linkeer[v][i] = u;
            return true;
         }
   }

   return false;
}
int Hungary()
{
   int ans = 0;
   for(int i=0;i<vN;i++)
      linker[i][0] = 0;
   for(int u = 0; u< uN; u++)
   {
      memse(vis, 0,sizeof(vis));
      if(Dfs(u)) ans ++;
   }
   return ans;
}
```

3.11.4 二分图最大权匹配 -KM

二分图如果有边权就是最大权匹配或是最小权匹配。有几个概念要区分一下。

1. 二分图的最佳匹配一定为完备匹配，在此基础上，才要求匹配的边权值之和最大或最小。
2. KM 算法是求最大权完备匹配。KM 算法的运行要求是必须存在一个完备匹配，如果求一个最大权匹配（不一定完备）该如何办？依然很简单，把不存在的边权值赋为 0。
3. KM 算法求得的最大权匹配是边权值和最大，如果我想要边权之积最大，又怎样转化？还是不难办到，每条边权取自然对数，然后求最大和权匹配，求得的结果 a 再算出 e^a 就是最大积匹配。至于精度问题则没有更好的办法了。

```
/*************************************************************************
题意是这样的，有n个人和n个物品， 每个人都给每个物品一个估价，然后让你分配这些物品，要人手一个，求赚的钱总和最大。
    最大权匹配。
 *************************************************************************/
#include<iostream>
#include<cstring>
#include<algorithm>
#include<cstdio>
using namespace std;
```

```cpp
const int N = 310;
const int INF = 0x3f3f3f3f;

int nx, ny, g[N][N];
int linker[N], lx[N], ly[N];
int slack[N], visx[N], visy[N];

bool Dfs(int x)
{
    visx[x] = 1;
    for(int y=0; y<ny; y++)
    {
        if(visy[y]) continue;
        int tmp =lx[x] + ly[y] - g[x][y];
        if(tmp == 0)
        {
            visy[y] = 1;
            if(linker[y]==-1 || Dfs(linker[y]))
            {
                linker[y] = x;
                return true;
            }
        }
        else if(slack[y] > tmp) slack[y] = tmp;
    }
    return false;
}
int KM()
{
    memset(linker, -1, sizeof(linker));
    memset(ly, 0, sizeof(ly));
    for(int i=0; i< nx; i++)
    {
        lx[i] = -INF;
        for(int j=0;j<ny;j++)
            if(g[i][j]> lx[i])
                lx[i] = g[i][j];
    }
    for(int x=0; x< nx;x++)
    {
        for(int i=0;i<ny; i++)
            slack[i] = INF;
        while(true)
        {
            memset(visx, 0, sizeof(visx));
            memset(visy, 0, sizeof(visy));
            if(Dfs(x)) break;
            int d = INF;
            for(int i=0;i<ny;i++)
                if(!visy[i] && d > slack[i])
                    d = slack[i];
            for(int i=0;i<nx;i++)
                if(visx[i])
                    lx[i] -= d;
            for(int i=0;i< ny; i++)
                if(visy[i]) ly[i]+= d;
                else slack[i] -= d;
        }
    }
    int ans = 0;
    for(int i=0;i<ny;i++)
        if(linker[i]!=-1) ans += g[linker[i]][i];
    return ans;

}
int main()
{
    int n;
    while(scanf("%d", &n)!=EOF)
    {
        for(int i=0;i<n;i++)
```

```
            for(int j=0;j<n;j++)
                scanf("%d", &g[i][j]);
        nx = ny = n;
        int ans = KM();
        printf("%d\n", ans);
    }
    return 0;
}
```

3.11.5 一般图匹配带花树

3.11.6 一般图最大加权匹配

3.12 最大流

3.12.1 EK

EK 算法的很简单，每次对原图进行 BFS，找到一条到汇点的有向路，计算出路径上最小的残量，并记录路径，然后用最小的参量顺着路径更新这条路上的参量信息。直至找不到这样的增广路。
算法复杂度大概是 $O(N*M)$。

```
/*************************************************************************
  求最大流，也就是最小割，并输出割边集。
 *************************************************************************/

#include <cstdio>
#include <iostream>
#include <algorithm>
#include <cstring>
#include <queue>
using namespace std;

const int MAXN=100;
const int INF=0x3f3f3f3f;
int g[MAXN][MAXN];//原图的流量
int flow[MAXN][MAXN];//最后求得最大流的流量
int pre[MAXN];//记录每次增广的路径;
int a[MAXN];//残量网络;  表示流到该点的流量;
int start,end;
int n;//顶点数，编号1~n

int maxflow()
{
    queue<int>q;
    memset(flow,0,sizeof(flow));
    int max_flow=0;
    while(1)
    {
        memset(a,0,sizeof(a));
        a[start]=INF;
        while(!q.empty())q.pop();
        q.push(start);
        while(!q.empty())
        {
            int u=q.front();
            q.pop();
            if(u==end) break; //这句不能不加
            for(int v=1;v<=n;v++)
              if( !a[v] && flow[u][v] < g[u][v] ) //本次没有增广过,且还有流量,
              {
                  pre[v] = u;//记录路径;
                  a[v] = min( a[u] , g[u][v] - flow[u][v] );
                  //对u的相连的点进行更新;
                  q.push(v);
              }
        }
```

```
        if(a[end]==0)break; //表示增广到最后没有流量可加;
        for(int u=end;u!=start;u=pre[u])
        {
            // 按照路径进行更新。
            flow[pre[u]][u]+=a[end];
            flow[u][pre[u]]-=a[end];
        }
        max_flow+=a[end];
    }
    return max_flow;
}

const int MAXM=550;
int x[MAXM],y[MAXM];
int m;
int main()
{
    while(scanf("%d%d",&n,&m)==2)
    {
        if(n==0&&m==0)break;
        memset(g,0,sizeof(g));
        for(int i=0;i<m;i++)
        {
            scanf("%d%d",&x[i],&y[i]);
            scanf("%d",&g[x[i]][y[i]]);
            g[y[i]][x[i]]=g[x[i]][y[i]];
        }
        start=1,end=2;
        maxflow();
        for(int i=0;i<m;i++)
        {
            // 用点来表示边，相邻两条边的两端的流量不同;则是要断开的割。
            if((!a[x[i]] && a[y[i]]) || ( a[x[i]] && !a[y[i]]) )
                printf("%d␣%d\n", x[i], y[i]);
        }
        printf("\n");
    }
    return 0;
}
```

3.12.2 ISAP

BFS 初始化 + 栈优化。十分高效。复杂度，暂时不知道，据说比 dinic 快一点。

```
 /*
最大流模板
sap
*/
#include<stdio.h>
#include<string.h>
#include<algorithm>
#include<iostream>
using namespace std;

const int MAXN=100010;//点数的最大值
const int MAXM=400010;//边数的最大值
const int INF=0x3f3f3f3f;

struct Node
{
    int from,to,next;
    int cap;
}edge[MAXM];
int tol;
int head[MAXN];
int dep[MAXN];
int gap[MAXN];//gap[x]=y :说明残留网络中dep[i]==x的个数为y

int n;//n是总的点的个数，包括源点和汇点
```

```
26
27  void init()
28  {
29      tol=0;
30      memset(head,-1,sizeof(head));
31  }
32
33  void addedge(int u,int v,int w)
34  {
35      edge[tol].from=u;
36      edge[tol].to=v;
37      edge[tol].cap=w;
38      edge[tol].next=head[u];
39      head[u]=tol++;
40      edge[tol].from=v;
41      edge[tol].to=u;
42      edge[tol].cap=0;
43      edge[tol].next=head[v];
44      head[v]=tol++;
45  }
46  void BFS(int start,int end)
47  {
48      memset(dep,-1,sizeof(dep));
49      memset(gap,0,sizeof(gap));
50      gap[0]=1;
51      int que[MAXN];
52      int front,rear;
53      front=rear=0;
54      dep[end]=0;
55      que[rear++]=end;
56      while(front!=rear)
57      {
58          int u=que[front++];
59          if(front==MAXN)front=0;
60          for(int i=head[u];i!=-1;i=edge[i].next)
61          {
62              int v=edge[i].to;
63              if(dep[v]!=-1)continue;
64              que[rear++]=v;
65              if(rear==MAXN)rear=0;
66              dep[v]=dep[u]+1;
67              ++gap[dep[v]];
68          }
69      }
70  }
71  int SAP(int start,int end)
72  {
73      int res=0;
74      BFS(start,end);
75      int cur[MAXN];
76      int S[MAXN];
77      int top=0;
78      memcpy(cur,head,sizeof(head));
79      int u=start;
80      int i;
81      while(dep[start]<n)
82      {
83          if(u==end)
84          {
85              int temp=INF;
86              int inser;
87              for(i=0;i<top;i++)
88                 if(temp>edge[S[i]].cap)
89                 {
90                     temp=edge[S[i]].cap;
91                     inser=i;
92                 }
93              for(i=0;i<top;i++)
94              {
95                  edge[S[i]].cap-=temp;
96                  edge[S[i]^1].cap+=temp;
```

```
            }
            res+=temp;
            top=inser;
            u=edge[S[top]].from;
        }
        if(u!=end&&gap[dep[u]-1]==0)//出现断层，无增广路
          break;
        for(i=cur[u];i!=-1;i=edge[i].next)
           if(edge[i].cap!=0&&dep[u]==dep[edge[i].to]+1)
             break;
        if(i!=-1)
        {
            cur[u]=i;
            S[top++]=i;
            u=edge[i].to;
        }
        else
        {
            int min=n;
            for(i=head[u];i!=-1;i=edge[i].next)
            {
                if(edge[i].cap==0)continue;
                if(min>dep[edge[i].to])
                {
                    min=dep[edge[i].to];
                    cur[u]=i;
                }
            }
            --gap[dep[u]];
            dep[u]=min+1;
            ++gap[dep[u]];
            if(u!=start)u=edge[S[--top]].from;
        }
    }
    return res;
}

int main()
{
   // freopen("in.txt","r",stdin);
  // freopen("out.txt","w",stdout);
    int start,end;
    int m;
    int u,v,z;
    int T;
    scanf("%d",&T);

    while(T--)
    {
        init();
        scanf("%d%d",&n,&m);
        int minx=10000000;
        int maxx=-10000000;
        int x,y;
        for(int i=1;i<=n;i++)
        {
            scanf("%d%d",&x,&y);
            if(minx>x)
            {
                minx=x;
                start=i;
            }
            if(maxx<x)
            {
                maxx=x;
                end=i;
            }
        }


        while(m--)
```

```
        {
            scanf("%d%d%d",&u,&v,&z);
            addedge(u,v,z);
            addedge(v,u,z);
        }
        //n一定是点的总数，这是使用SAP模板需要注意的
        int ans=SAP(start,end);
        printf("%d\n",ans);
    }
    return 0;
}
```

3.12.3 Dinic

最大流算法中 Dinic 算法的思路比较好懂。效率也还可以。复杂度是 $O(n^2 * m)$，不过这是理论值，实际效果要快的多，差不多有 $O(n^{2/3} * m)$。对于二分图来说，复杂度可以降到 $O(\sqrt{n} * m)$。
最大流算法的思路是这样的。
每次将图用 Bfs 分层，直到找到一条路径到达汇点。
然后使用 Dfs 沿阻塞流进行增广，什么是阻塞流呢？很难形象的解释，其实质就是在分层的基础上找到一条到汇点的路径并把沿途的边的流量用最小容量填满。
然后继续使用 Bfs 分层。
整个最大流就是这样的思路。下面讲一下反向弧。我们在建边的时候要建反向弧，反向弧是干什么用的呢？ 在建反向弧的时候我们把容量设为 0，当我们正向用 Dfs 增广的时候我们要把增加的流量变成负数加在反向弧上。我们用当前流到 a 的流量将原来流过 a 的流量退回到其他的路径上到达汇点，然后在继续增广当前流量。
下面是循环版的模板，效率稍微更高一些。

```
链式前向星 + 循环版，Dinic
************************************************************************/

#include<iostream>
#include<cstdio>
#include<cstring>
#include<algorithm>
#include<queue>
#include<map>
using namespace std;
const int M =410;
const int INF = 0x3fffffff;
int n,m,k;
int head[M],cnt;
struct Edge
{
    int to,next,cap,flow;
}edge[M<<2];
void init()
{
    cnt=0;
    memset(head,-1,sizeof(head));
}
void add(int from,int to,int w)
{
    edge[cnt].to=to;
    edge[cnt].cap=w;
    edge[cnt].flow=0;
    edge[cnt].next=head[from];
    head[from]=cnt++;

    edge[cnt].to=from;
    edge[cnt].cap=0;
    edge[cnt].flow=0;
    edge[cnt].next=head[to];
    head[to]=cnt++;
}
//0---源点，1到n----插座，已有，n+1到tot,转换器；
//tot+1到m+tot,用电器，m+tot+1,汇点；
```

```
int dep[M],cur[M],sta[M],Q[M<<2];
bool bfs(int s,int t)
{
    memset(dep,-1,sizeof(dep));
    int front=0, tail=0;
    Q[tail++] = s;
    dep[s]=0;
    while(front < tail)
    {
        int u=Q[front++];
        for(int i=head[u]; i!=-1; i=edge[i].next)
        {
            int v=edge[i].to;
            if(dep[v]==-1 && edge[i].cap> edge[i].flow)
            {
                dep[v]=dep[u]+1;
                if(v==t)
                    return true;
                Q[tail++] = v;
            }
        }
    }
    return false;
}

int Dinic(int s,int t)
{
    int maxflow=0;
    while(bfs(s,t))
    {
        for(int i=0;i<=t;i++)
            cur[i]=head[i];
        int u=s,tail=0;
        while(cur[s]!=-1)
        {
            if(u==t)
            {
                int tp=INF;
                for(int i=tail-1;i>=0;i--)
                    tp=min(edge[sta[i]].cap-edge[sta[i]].flow,tp);
                maxflow+=tp;
                for(int i=tail-1;i>=0;i--)
                {
                    edge[sta[i]].flow+=tp;
                    edge[sta[i]^1].flow-=tp;
                    if(edge[sta[i]].flow==edge[sta[i]].cap)
                        tail=i;
                }
                u=edge[sta[tail]^1].to;
            }
            else if(cur[u]!=-1 && edge[cur[u]].cap>edge[cur[u]].flow && dep[u]+1== dep[edge[cur[u]].to])
            {
                sta[tail++]=cur[u];
                u=edge[cur[u]].to;
            }
            else
            {
                while(u!=s && cur[u]==-1)
                    u=edge[sta[--tail]^1].to;
                cur[u]=edge[cur[u]].next;
            }
        }
    }
    return maxflow;
}
int tm[M],tot;
int main()
{
    while(scanf("%d",&n)!=EOF)
    {
        init();
```

```
        map<string ,int > Map;
        string str,str1;
        tot=0;
        for(int i=1;i<=n;i++)
        {
            cin>>str;
            Map[str]=++tot;
            add(0,i,1);//源点到插座;
        }
        scanf("%d",&m);
        for(int j=1;j<=m;j++)
        {
            cin>>str1>>str;
            if(Map[str]==0)
                Map[str]=++tot;
            tm[j]=Map[str];
        }
        scanf("%d",&k);
        for(int j=1;j<=k;j++)
        {
            cin>>str>>str1;
            if(Map[str1]==0)
                Map[str1]=++tot;
            if(Map[str]==0)
                Map[str]=++tot;
            add(Map[str1],Map[str],INF);//转换器到插座
        }
        for(int j=1;j<=m;j++)
        {
            add(tot+j,m+tot+1,1);//用电器到汇点;
            add(tm[j],j+tot,1);//插座到用电器;
        }
        int ans=Dinic(0,tot+m+1);
        printf("%d\n",m-ans);
    }
    return 0;
}
```

3.12.4 最大流判多解

最大流判断多解，就是在参量网络中寻找正环。

```

bool vis[N], no[N], sta[N], top;
bool Dfs(int u, int pre, bool flag)
{
    vis[u] = 1;
    sta[top++] = u;
    for(int i=head[u]; i!=-1; i=edge[i].next)
    {
        int v = edge[i].to;
        if(edge[i].cap <= edge[i].flow) continue;
        if(v == pre) continue;
        if(!vis[v]){
            if( Dfs(v, u, edge[i^1].flow < edge[i^1].cap)) return true;
        }
        else if(!no[v]) return true;
    }
    if(!flag)
        while(1)
        {
            int v = sta[--top];
            no[v] = true;
            if(v==u) break;
        }
    return false;
}
```

```
bool check_multiflow()
{
    memset(vis, 0,sizeof(vis));
    memset(no, 0,sizeof(no));
    top = 0;
    bool flag = Dfs(end, end, 0);
}
```

3.13 最小费用最大流

3.13.1 SPFA+ 费用流

```
/*
 * 返回最大流，cost存最小费用，如果求最大费用只要把费用存成负数存到图中就可以了。
 */
#include <cstdio>
#include <cstdlib>
#include <cmath>
#include <cstring>
#include <ctime>
#include <algorithm>
#include <iostream>
#include <sstream>
#include <string>
#include <queue>
#define oo 0x13131313
using namespace std;
const int MAXN=200;
const int MAXM=200000;
const int INF=0x3f3f3f3f;
struct Edge
{
    int to,next,cap,flow,cost;
    void get(int a,int b,int c,int d)
    {
        to=a,cap=b,cost=c;next=d;flow=0;
    }
}edge[MAXM];
int head[MAXN],tol;
int pre[MAXN],dis[MAXN];
bool vis[MAXN];
int N;
void init(int n)
{
    N=n;
    tol=0;
    memset(head,-1,sizeof(head));
}
void addedge(int u,int v,int cap,int cost)
{
    edge[tol].get(v,cap,cost,head[u]);head[u]=tol++;
    edge[tol].get(u,0,-cost,head[v]);head[v]=tol++;
}
bool spfa(int s,int t)
{
    queue<int>q;
    for(int i=0;i<N;i++)
    {
        dis[i]=INF;
        vis[i]=false;
        pre[i]=-1;
    }
    dis[s]=0;
    vis[s]=true;
    q.push(s);
    while(!q.empty())
    {
```

```
        int u=q.front();
        q.pop();
        vis[u]=false;
        for(int i= head[u];i!=-1;i=edge[i].next)
        {
            int v=edge[i].to;
            if(edge[i].cap>edge[i].flow&&
               dis[v]>dis[u]+edge[i].cost )
            {
                dis[v]=dis[u]+edge[i].cost;
                pre[v]=i;
                if(!vis[v])
                {
                    vis[v]=true;
                    q.push(v);
                }
            }
        }
    }
    if(pre[t]==-1) return false;
    else return true;
}
int minCostMaxflow(int s,int t,int &cost)
{
    int flow=0;
    cost = 0;
    while(spfa(s,t))
    {
        int Min=INF;
        for(int i=pre[t];i!=-1;i=pre[edge[i^1].to])
        {
            if(Min >edge[i].cap-edge[i].flow)
              Min=edge[i].cap-edge[i].flow;
        }
        for(int i=pre[t];i!=-1;i=pre[edge[i^1].to])
        {
            edge[i].flow+=Min;
            edge[i^1].flow-=Min;
            cost+=edge[i].cost*Min;
        }
        flow+=Min;
    }
    return flow;
}
int main()
{

}
```

3.13.2 zkw 费用流

```
    /*****************************************************************
    对二分图的效率高一些。
  最小费用最大流 z k w
 *****************************************************************/

#include<iostream>
#include<cstdio>
#include<cstring>
#include<algorithm>
#include<queue>
#include<cmath>
using namespace std;
const int N =100005;
const int INF =0x3fffffff;
struct Edge
{
    int to,next,cost,cap,flow;
```

```
    Edge(int _to=0,int _next=0,int _cap=0,int _flow=0,int _cost=0 ):to(_to),next(_next),cap(_cap),flow(_flow)
        ,cost(_cost){}
}edge[N];
int head[N],cnt=0;
void init()
{
    cnt=0;
    memset(head,-1,sizeof(head));
}
void add(int from,int to,int cap,int cost)
{
    edge[cnt]=Edge(to,head[from],cap,0,cost);
    head[from]=cnt++;
    edge[cnt]=Edge(from,head[to],0,0,-cost);
    head[to]=cnt++;
}

int n,m,dis[N],cur[N],vis[N];

//这一行变量是题意中的变量，模板不需要。
int Hx[N],Hy[N],Mx[N],My[N],totH=0,totM=0;

int aug(int u,int t,int flow)
{
    if(u==t)
        return flow;
    vis[u]=1;
    for(int i=cur[u];i!=-1;i=edge[i].next)
    {
        int v=edge[i].to;
        if(edge[i].cap>edge[i].flow && vis[v]==0 && dis[u]==dis[v]+edge[i].cost)
        {
            int tmp=aug(v,t,min(flow,edge[i].cap-edge[i].flow));
            edge[i].flow+=tmp;
            edge[i^1].flow-=tmp;
            cur[u]=i;
            if(tmp)
                return tmp;
        }
    }
    return 0;
}
bool modify_label(int tol)
{
    int d=INF;
    for(int u=0;u<tol;u++)
    {
        if(vis[u])
            for(int i=head[u];i!=-1;i=edge[i].next)
            {
                int v=edge[i].to;
                if(edge[i].cap> edge[i].flow && !vis[v])
                    d=min(d,dis[v]+edge[i].cost-dis[u]);
            }
    }
    if(d==INF)
        return false;
    for(int i=0;i<tol;i++)
    {
        if(vis[i])
        {
            vis[i]=0;
            dis[i]+=d;
        }
    }
    return true;
}

// 传入源点s， 汇点t，点的总个数tol， 编号从0开始。
// 返回一个pair，first是最小费用，second是最大流。
pair<int ,int > Mincost_Maxflow(int s,int t,int tol)
```

```
{
    int mincost=0,maxflow=0;
    memset(dis,0,sizeof(dis));
    while(1)
    {
        for(int i=0;i<tol;i++)
            cur[i]=head[i];
        while(1)
        {
            memset(vis,0,sizeof(vis));
            int tmp=aug(s,t,INF);
            if(tmp==0)
                break;
            maxflow+=tmp;
            mincost+=tmp*dis[s];
    //  cout<<tmp<<endl;
        }
        if(!modify_label(tol))
            break;
    }
    return make_pair(mincost,maxflow);
}
int main()
{
    while(scanf("%d%d",&n,&m)!=EOF)
    {
        if(n==0 && m==0)
            break;
        init();
        char s[2000];
        totH=totM=0;
        for(int i=1;i<=n;i++)
        {
            scanf("%s",s);
            for(int j=0;j<m;j++)
                if(s[j]=='H')
                {
                    Hx[totH]=i;
                    Hy[totH++]=j+1;
                }
                else if(s[j]=='m')
                {
                    Mx[totM]=i;
                    My[totM++]=j+1;
                }
        }
        //0是源点，1-totM是人，totM+1-totM+totH是房子；
        for(int i=1;i<=totM;i++)
            add(0,i,1,0);
        for(int i=1;i<=totH;i++)
            add(totM+i,totM+totH+1,1,0);
        for(int i=0;i<totM;i++)
            for(int j=0;j<totH;j++)
                add(i+1,totM+j+1,1,abs(Hx[j]-Mx[i])+abs(Hy[j]-My[i]));
        pair<int ,int > ans;
        ans=Mincost_Maxflow(0,totM+totH+1,totM+totH+2);
        printf("%d\n",ans.first);
    }
    return 0;
}
```

3.14 曼哈顿最小生成树

曼哈顿最小生成树就是在最小生成树上的演进算法。利用曼哈顿距离的特性缩小复杂度。
道理是这样的：
对于每个点的右上方来说，事实上可以将空间分成 45 度的两份，两份对称，所以我们只要考虑其中一个 45 度区域。对于这个区域，我们只要找到距离他最近的点然后建一条边就可以了。

然后有 8 个方位，但是由于边是双向的，所以我们只要建其中的四个方向就可以了。剩余的方向可以在枚举下一个点的时候建出来。画图就可以看出来，任意的四个方向都可以。都会对称回来。
然后问题就简化成了。给定一个点找距离他最近的右上角 45 区域的点。
然后我们可以发现，对于 $y > x$ 这个区域，如果对于点 $A(x_0, y_0)$ 在这个区域中有个点 $B(x_1, y_1)$。那么 $x_1 > x_0$&&$y_1 - x_1 > y_0 - x_0$。而 $dist(A, B) = x_1 - x_0 + y_1 - y_0 = x_1 + y_1 - (x_0 + y_0)$，那么对于点 A，则是找在这个区域内 $x_1 + y_1$ 最小的点。那么什么样的点满足在这个区域内呢，$(x_1 > x_0$&&$y_1 - x_1 > y_0 - x_0)$ 便是条件。
对于当前的点，我们找到其 $y_0 - x_0$ 离散化后的位置，我们在这个位置之后寻找最小的 $x_1 + y_1$。这里的树状数组用的比较诡异，正常的树状数组我们是向后更新，向前查询。但是这个的使用需求不一样。他是向后查询，向前更新。我们在排序的时候是按照 $y_1 - x_1$ 排序的，所以我们找到当前点的位置的时候，在这个位置后面的点，都是属于 $y_1 - x_1 > y_0 - x_0$ 的点。我们在这些点中寻找最小的 $x_1 + y_1$。这个都可以理解，但是有一个条件我们没有使用，就是 $x_1 > x_0$，事实上我们的处理顺序是按照 x 排序的，我们是从后面往前处理，所以在该点的右边的点在之前肯定都加进去了，保证找到的点肯定是右边的点。逆序数的思维。
对于不同的方向,我们将坐标转换。比如对于与右上角相对的左下角。我们可以发现,要满足 $x_0 > x_1$&&$y_1 - x_1 > y_0 - x_0)$，我们要找的是最小的 $-x_1 - y_1$，所以只要把 x 和 y 事先都取负数就可以了。程序中取的方向是:右边的四个方位。这样取一边可以保证逆序数那部分不需要改动，缩减代码。
很佩服做出来这个问题的人。一个小小的子问题就已经很难搞了。
这种思路的特点就是，有多个顺序条件。对于顺序条件我们可以使用逆序这种操作屏蔽掉一维顺序，然后用位置顺序再屏蔽一维顺序，接下来的一个大小顺序就可以用树状数组来搞。所以树状数组对于解决三个维度顺序的问题还是很简单高效的，所以下次做题目的时候只要能把问题抽象成按照顺序限制找值的这个模型就可以用树状数组来做。

```cpp
/************************************************************************

 ************************************************************************/
#include<iostream>
#include<cstdio>
#include<cstring>
#include<algorithm>
using namespace std;
const int INF = 0x3f3f3f3f;
const int N = 10005;
int n,K, fa[N];
struct Point
{
    int x,y,id;
    bool operator < (const Point &A)const
    {
        return x != A.x ? x < A.x : y < A.y;
    }
}point[N];

struct Edge
{
    int u,v,w;
    Edge(){}
    Edge(int a,int b, int c): u(a), v(b),w(c){}
    bool operator < (const Edge & A) const
    {
        return w < A.w;
    }
}edge[N<<3];
int m;

struct BIT
{
    int Min_val, pos;
    void init()
    {
        Min_val = INF;
        pos = -1;
    }

}bit[N];
```

```

int Find(int x)
{
    return x == fa[x] ? x : fa[x] = Find(fa[x]);
}

int Dis(int a,int b)
{
    return abs( point[a].x - point[b].x ) + abs( point[a].y - point[b].y );
}

int lowbit(int x){return x&(-x);}

void Update(int x, int val, int pos)
{
    for( int i = x; i > 0; i -= lowbit(i))
        if(val < bit[i].Min_val)
            bit[i].Min_val = val, bit[i].pos = pos;
}

int Query(int x, int num)
{
    int Min_val = INF, pos = -1;
    for(int i = x; i <= num; i += lowbit(i))
        if(bit[i].Min_val < Min_val)
            Min_val = bit[i].Min_val, pos = bit[i].pos;
    return pos;
}

int a[N], b[N];

int Manhattan_MST()
{
    for(int dir = 0; dir < 4; dir++)
    {
        if( dir % 2 == 1)
            for(int i = 1; i <= n; i++ )
                swap( point[i].x , point[i].y );
        else if( dir == 2 )
            for( int i = 1; i <= n; i++)
                point[i].x = -point[i].x;

        sort( point + 1, point + n + 1);

        for( int i = 1; i <= n; i++)
            a[i] = b[i] = point[i].y - point[i].x;

        sort( b + 1, b + n + 1 );
        int num = unique( b + 1 , b + 1 + n ) - b;

        for( int i = 1; i <= num; i++)
            bit[i].init();
        for(int i = n; i > 0; i--)
        {
            int pos = lower_bound(b + 1, b + num + 1 , a[i]) - b;
            int ans = Query(pos, num);
            if( ans != -1 )
                edge[m++] = Edge(point[i].id, point[ans].id, Dis(i, ans));
            Update(pos, point[i].x + point[i].y, i);
        }
    }

    sort( edge, edge + m);
    int cnt = n - K;
    for(int i = 0; i <= n; i++) fa[i] = i;
    for(int i = 0; i < m; i++)
    {
        int u = edge[i].u, v = edge[i].v, w = edge[i].w;
        int fu = Find(u), fv = Find(v);
        if( fu != fv )
        {
```

```
            cnt--;
            fa[fu] = fv;
            if(cnt==0)
                return w;
        }
    }
}

int main()
{
    while(scanf("%d%d", &n,&K)!=EOF)
    {
        m = 0;
        for(int i = 1; i <= n; i++)
            point[i].id = i;
        for(int i = 1; i <= n; i++)
            scanf("%d%d", &point[i].x, &point[i].y);

        int ans = Manhattan_MST();
        printf("%d\n", ans);
    }
    return 0;
}
```

3.15 LCA-最近公共祖先

3.15.1 离线 Tarjan 算法

离线算法需要预读入进来。所有的询问，然后处理。

```
#include<iostream>
#include<cstdio>
#include<cstring>
#include<algorithm>
using namespace std;
int vis[50005];
struct E
{
    int to,next,w;
}edge[50005];
int cnt=0,head[50005];
int Fa[50005];
int qu[205],qv[205];
int dis[50005];
int n,m;

void init()
{
    for(int i=0;i<50000;i++)
        Fa[i] =i;
    memset(head,-1,sizeof(head));
    cnt =0;
    memset(vis,0,sizeof(vis));
}
void add(int u,int v,int w)
{
    edge[cnt].to = v;
    edge[cnt].next = head[u];
    edge[cnt].w = w;
    head[u] = cnt++;
}
int find(int x)
{
    return x==Fa[x]?x: Fa[x] = find(Fa[x]);
}
int ans[205];
void Tarjan(int u)
{
```

```
    vis[u] = 1;
    for(int i=0;i<m;i++)
    {
        if(qu[i]==u && vis[qv[i]])
            ans[i] = dis[qv[i]] + dis[qu[i]] - 2 * dis[find(qv[i])];
        else if(qv[i]==u && vis[qu[i]])
            ans[i] = dis[qv[i]]+ dis[qu[i]]- 2*dis[find(qu[i])];
    }
    for(int i=head[u];i!=-1;i= edge[i].next)
    {
        int v = edge[i].to;
        if(!vis[v])
        {
            Tarjan(v);
            Fa[v] = u;
        }
    }
}
void Dfs(int u)
{
    for(int i=head[u]; i!=-1; i= edge[i].next)
    {
        int v = edge[i].to;
        dis[v] = edge[i].w+ dis[u];
        Dfs(v);
    }
}
int main()
{
    int T;
    scanf("%d", &T);
    while(T--)
    {
        init();
        scanf("%d%d", &n,&m);
        for(int i=0;i<n-1;i++)
        {
            int u,v,w;
            scanf("%d%d%d", &u,&v,&w);
            add(u,v,w);
            vis[v]=1;
        }

        for(int i=0;i<m;i++)
            scanf("%d%d", &qu[i],&qv[i]);
        int root=1;
        for(int i=1;i<=n;i++)
        {
            if(!vis[i])
                root = i;
        }
        memset(vis,0,sizeof(vis));
        memset(ans,0,sizeof(ans));
        memset(dis,0,sizeof(dis));
        Dfs(root);

        Tarjan(root);
        for(int i=0;i<m;i++)
            printf("%d\n",ans[i]);
    }
    return 0;
}
```

3.15.2 倍增算法

利用 ST 的思路进行倍增计算。

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

```cpp
#include<algorithm>
#include<cstring>
using namespace std;
int fa[20][1005];
int ans[1005];
int deep[1005];
int vis[1005];
struct E
{
    int to,next;
}edge[10005];
int cnt=0,head[1005];
void init()
{
    cnt =0;
    memset(head,-1,sizeof(head));
}
void add(int u,int v)
{
    edge[cnt].to = v;
    edge[cnt].next = head[u];
    head[u] = cnt++;
}
void Dfs(int u)
{
    for(int i=head[u]; i!=-1; i = edge[i].next)
    {
        int v = edge[i].to;
        deep[v] = deep[u]+1;
        Dfs(v);
    }
}
int LCA(int u,int v)
{
    if(deep[u] < deep[v])
        swap(u,v);
    int t = deep[u]- deep[v];
    int cnt =0;
    while(t)
    {
        if(t%2==1)
            u = fa[u][cnt];
        t/=2;
        cnt++;
    }
    if(u==v)
        return u;
    int k=0;
    while(u!=v)
    {
        if(fa[u][k]!=fa[v][k] || (fa[u][k]==fa[v][k] && k==0))
        {
            u = fa[u][k];
            v = fa[v][k];
            k++;
        }
        else k--;
    }
    return u;
}
int main()
{
    int n;
    while(scanf("%d", &n)!=EOF)
    {
        init();
        memset(ans,0,sizeof(ans));
        memset(vis,0,sizeof(vis));
        memset(fa,0,sizeof(fa));
        for(int i=0;i<n;i++)
        {
```

```
            int u,v,m;
            scanf("%d:(%d)", &u,&m);
            for(int j=0;j<m;j++)
            {
                scanf("%d", &v);
                add(u,v);
                fa[v][0] = u;
                vis[v] = 1;
            }
        }
        int root=1;
        for(int i=1;i<=n;i++)
            if(!vis[i])
                root = i;
        memset(deep,0,sizeof(deep));
        Dfs(root);
        for(int i=1;(1<<i)<=n;i++)
            for(int j=1;j<=n;j++)
                fa[j][i] = fa[fa[j][i-1]][i-1];
        int q;
        scanf("%d", &q);
        while(q--)
        {
            char s1[2],s2[2];
            int u,v;
            scanf("%1s%d%d%1s",s1,&u,&v,s2);
            ans[LCA(u,v)]++;
        }
        for(int i=1;i<=n;i++)
            if(ans[i]!=0)
                printf("%d:%d\n",i,ans[i]);
    }
    return 0;
}
```

3.16 欧拉序列

欧拉序列就是对树的 Dfs 遍历顺序，包括进入时的编号和出来时的编号。

3.17 欧拉路

欧拉路径： 从起点出发，每条边只经过一次，不要求回到起点。
欧拉回路： 从起点出发，经过没条路一次，并回到起点。
欧拉回路的判断：

1. 无向图： 联通图，每个点的度数都是偶数。
2. 有向图： 联通图，每个点的入度等于出度。
3. 混合图： 需要借助网络流来判定。

欧拉路径的判断：

1. 无向图： 联通图，每个点的度数都是偶数，或者只有两个点的度数是奇数，这俩个点一个是起点一个是终点。
2. 有向图： 联通图，每个点的入度等于出度，或者，只有一个点的入度比出度少 1，作为起点，只有一个点的入度比出度大 1，作为终点。其余的点入度等于出度。

下面介绍以下混合图的判定。
混合图就是有向和无向边都有的图。但是我们不能简单的看成特殊的有向图，因为如果看成双向边的话，边数就增加了，事实上我们不能增加边的数量。所以我们只能先给无向边任意指定一个方向。这样就变成有向图了。我们设 D[i] 为 i 点的出度减入度。可以发现，我们设定一个边的方向的时候，改变了这么几个信息。u 的入度 -1，出度 +1。v 的入度 +1，出度 -1。这样对每个点改变的度事实上是 2，所以原来是偶数现在还是偶数，奇数还是奇数。我们要求最终的每个点的入度等于出度，所以 D[i] == 0，所以，如果我们指定方向后的图中，只要有一个点的 D 是奇数，则不存在欧拉回路。
如果所有的 D 都是偶数，我们建立源点和汇点，对于每个 $D[i] > 0$ 的点 i，建立源点到 i 的边，容量为

$D/2$。对于 $D[i] < 0$ 建立 i 到汇点的边，容量为 $-D/2$，除此意外每条边仍然存在，并且容量为 1，表示该边最多被改变方向一次，求这个网络的最大流。如果所有源点连出去的边都满流，那么原混合图就是有欧拉回路，将网络中所有流量为 1 的中间边在原图中改变方向，新图一定就是有欧拉回路的有向图。如果源点出边没有满流就不是。
上面是判断欧拉回路，如果要判断欧拉路。如果图中有且仅有两个点的 D 是奇数，那么我们给这两个点建边，然后用上面的方法判断欧拉回路，如果有欧拉回路就是有欧拉路径，否则就没有。

3.17.1 有向图

```
 /*************************************************************************
 * 有向图 求 欧拉路径
 * 就是一个Dfs，路径倒着输出
 *************************************************************************/
#include<iostream>
#include<cstdio>
#include<cstring>
#include<algorithm>
#include<string>
#include<vector>
using namespace std;

const int N = 35;
string s[1010];
int in[N], out[N], n, ans[1010], num;

struct Edge
{
    int to, next, idx, flag;
}edge[20000];
int cnt, head[N];

void init()
{
    cnt = 0;
    memset(head, -1, sizeof(head));
    memset(in, 0, sizeof(in));
    memset(out, 0, sizeof(out));
}

void add(int u, int v , int idx)
{
    edge[cnt].to = v;
    edge[cnt].next = head[u];
    edge[cnt].idx = idx;
    edge[cnt].flag = 0;
    head[u] = cnt++;
}

void Dfs(int u)
{
    for(int i = head[u] ; i!= -1; i = edge[i].next)
        if( ! edge[i].flag )
        {
            edge[i].flag = 1;
            Dfs(edge[i].to);
            ans[num++] = edge[i].idx;
        }
}
int main()
{
    int T;
    scanf("%d", &T);
    while(T--)
    {
        scanf("%d", &n);
        for(int i=0;i<n;i++)
            cin>>s[i];
        sort(s, s+n); // 字典序排序。
```

```
        init();
        int st = 100;
        for(int i = n-1; i >= 0; i--) // 字典序大的先加入。遍历的时候从小的开始遍历。保证字典序最小。
        {
            int u = s[i][0] - 'a';
            int v = s[i][s[i].length()-1] - 'a';
            add(u , v, i);
            out[u] ++ ;
            in[v] ++;
            st = min( st , min( u , v ));
        }
        int cc1 = 0, cc2 = 0;
        for(int i = 0; i < 26; i++) // 统计每个点的度数。
            if(out[i] - in[i] == 1)
                cc1++, st = i;
            else if( out[i] - in[i] == -1)
                cc2++;
            else if(out[i] - in[i] != 0)
                cc1 = 3;
        if(!((cc1 == 0 && cc2 ==0 ) || (cc1 == 1 && cc2 == 1 )))
        { // 所有点入度等于出度，则有欧拉回路，也就一定有欧拉路。
            // 或者两个奇数的，一个起点一个终点。否则就没有。
            printf("***\n");
            continue;
        }

        num = 0;
        Dfs(st);
        if(num != n)
        {
            printf("***\n");
            continue;
        }
        for(int i = num - 1 ; i >= 0 ;i--)
        {
            cout<<s[ans[i]];
            if(i > 0) printf(".");
            else printf("\n");
        }
    }
    return 0;
}
```

3.17.2 无向图

```
 /*************************************************************************
   图求欧拉回路
   有向无向都可以。
   这个是无向图，  如果时候有向图，和前一个一样。
   求欧拉路径和欧拉回路是一样的。
 *************************************************************************/

#include<iostream>
#include<cstdio>
#include<algorithm>
#include<cstring>
using namespace std;

struct Edge
{
    int to, next, idx;
    Edge(){}
    Edge(int a, int b, int c):to(a), next(b),idx(c){}
}edge[M];
int cnt, head[N], fa[N], du[N], vis[M],path[M], num;
void init()
{
    cnt = num = 0;
```

```
    memset(head, -1, sizeof(head));
    memset(du, 0, sizeof(du));
    for(int i=0;i<N;i++) fa[i] = i;
}
void add(int u, int v, int idx)
{
    edge[cnt] = Edge(v, head[u], idx);
    head[u] = cnt++;
}
int find(int x)
{
    return x==fa[x]? x: fa[x] = find(fa[x]);
}
void Dfs(int u)
{
    for(int i= head[u]; i!=-1; i = edge[i].next)
    {
        if(!vis[edge[i].idx])
        {
            vis[edge[i].idx] = 1;
            Dfs(edge[i].to);
            path[num++] =ege[i].idx;
        }
    }
}

int solve()
{
    init();
    scanf("%d%d", &n,&m);
    for(int i = 0; i< m; i++)
    {
        int u, v;
        scanf("%d%d", &u, &v);
        add(u, v, i);
        add(v, u, i);
        du[u]++;
        du[v]++;
        int fu = find(u), fv = find(v);
        if(fu != fv)
            fa[fu] = fv;
    }
    int origin = -1;
    int flag;
    for(int i=0;i<n;i++)
        if(head[i]!=-1)
        {
            if(du[i]%2==1) return false;
            if(origin==-1) origin = i;
            if(find(i) != find(origin)) return false;
        }
    memset(vis, 0, sizeof(vis));
    if(origin!=-1) Dfs(origin);
    //路径是反的。
    return true;
}
```

3.17.3 混合图

```
 /*************************************************************************
 > File Name: oulahunhe.cpp
 > Author: Bai Yan
 > 题意:
 > Created Time: 2016年10月03日 星期一 22时06分19秒
 ************************************************************************/
#include<iostream>
#include<cstdio>
#include<cstring>
```

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

const int N = 210;
const int M = 20100;
const int INF = 0x3f3f3f3f;

struct Edge
{
    int to, next, cap, flow;
    Edge(){}
    Edge(int a, int b, int c, int d):to(a), cap(b), next(c), flow(d){}
}edge[M];
int cnt, head[N], gap[N], dep[N], pre[N], cur[N];
int in[N], out[N];
void init()
{
    cnt = 0;
    memset(head, -1, sizeof(head));
    memset(in, 0, sizeof(in));
    memset(out, 0, sizeof(out));
}
void add(int u, int v, int w, int rw=0)
{
    edge[cnt] = Edge(v, w, head[u], 0);
    head[u] = cnt++;
    edge[cnt] = Edge(u, rw, head[v], 0);
    head[v] = cnt++;
}

int sap(int s, int ed, int n)
{
    memset(gap, 0, sizeof(gap));
    memset(dep, 0, sizeof(dep));
    memcpy(cur, head, sizeof(head));
    int u = s;
    pre[u] = -1;
    gap[0] = n;
    int ans = 0;
    while(dep[s] < n)
    {
        if(u == ed)
        {
            int Min = INF;
            for(int i = pre[u]; i!=-1; i = pre[edge[i^1].to])
                Min = min (Min, edge[i].cap - edge[i].flow);
            for(int i = pre[u]; i!=-1; i= pre[edge[i^1].to])
            {
                edge[i].flow += Min;
                edge[i^1].flow -= Min;
            }
            u = s;
            ans += Min;
            continue;
        }
        bool flag = 0;
        int v;
        for(int i= cur[u]; i != -1; i = edge[i].next)
        {
            v = edge[i].to;
            if(edge[i].cap - edge[i].flow && dep[v]+1 == dep[u])
            {
                flag = 1;
                cur[u] = pre[v] = i;
                break;
            }
        }
        if(flag)
        {
            u = v;
            continue;
```

```
        }
        int Min = n;
        for(int i =head[u]; i!=-1; i= edge[i].next)
            if(edge[i].cap- edge[i].flow && dep[edge[i].to] < Min)
            {
                Min = dep[edge[i].to];
                cur[u] = i;
            }
        gap[dep[u]] -- ;
        if(!gap[dep[u]]) return ans;
        dep[u] = Min + 1;
        gap[dep[u]]++;
        if(u != s) u = edge[pre[u]^1].to;
    }
    return ans;
}
int main()
{
    int T;
    int n,m;
    scanf("%d", &T);
    while(T--)
    {
        scanf("%d%d", &n,&m);
        init();
        while(m--)
        {
            int u,v,w;
            scanf("%d%d%d", &u, &v, &w);
            in[v]++;
            out[u]++;
            if(w==0) add(u, v, 1);
            //无向边直接建成有向边。
            //有向边不管
        }
        bool flag = true;
        for(int i = 1; i<= n ;i++)
        {
            //按度数建边。
            if(out[i] - in[i] > 0)
                add(0, i , (out[i] - in[i])/2);
            else if(in[i] - out[i] > 0)
                add(i, n+1, (in[i]-out[i])/2);
            if((out[i]-in[i])&1)
                flag = false;
        }
        // 本题保证了图的联通性，如果没有保证还有并查集跑一遍判断联通性。
        //
        if(!flag)
        {
            printf("impossible\n");
            continue;
        }
        sap(0, n+1, n+2);
        for(int i = head[0]; i!=-1; i=edge[i].next)
            if(edge[i].cap > 0 && edge[i].cap > edge[i].flow)
            {
                flag = false;
                break;
            }
        if(!flag) printf("impossible\n");
        else printf("possible\n");
    }
    return 0;
}
```

3.18 树同构

判断树是否同构，一般使用树 hash，我们预随机出一个数组。然后从根开始递归的 Hash。

```
    /*************************************************************************
给定两棵树。用字符串表示Dfs顺序，判断有根树是否同构。
注意，初始化随机算子的时候会出错，这一题不初始化随机算子了。
 *************************************************************************/
#include<iostream>
#include<cstdio>
#include<cstring>
#include<algorithm>
//#include<string>
#include<cstdlib>
#include<ctime>
using namespace std;
const int MOD = 19901;

int h[10005<<2];
char s1[10005<<2],s2[10005<<2];
char *p;
int hash(int j)
{
    // j记录的是层数。
    // 这样递归的hash可以将子树的hash值集中到父亲结点上。
    int sum = h[j+5000];
    while( *p && *p++ == '0')
    {
        sum = (sum + hash( j+1 ) * h[j] ) % 19001;
    }
    return (sum*sum) %19001;
}
int main()
{
// srand((int)time(NULL));
    for(int i=0; i < 10000;i++)
        h[i] = (rand() % MOD)+1;
    int T;
    scanf("%d", &T);
    while(T--)
    {
        scanf("%s␣%s", s1, s2);
        if(strlen(s1) != strlen(s2))
        {
            printf("different\n");
            continue;
        }
        p = s1;
        int a = hash(1);
        p = s2;
        int b = hash(1);
        if(a==b)
            printf("same\n");
        else printf("different\n");
    }
    return 0;
}
```

无根树的 hash 和有根树在 hash 部分是一样的，一般的方法是选择一棵树的每个点做为根 hash 出来。更快的方法是找到树的重心。

3.19 树的重心

树的重心的定义是，以重心的为根的所有子树中最大的子树结点最少。使用 Dp 来求出。一棵树可能有多个重心。一棵树，最多只有两个重心。

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

```cpp
 > File Name: poj1655.cpp

 > 多个重心，最多两个。
 ************************************************************************/
#include<iostream>
#include<cstdio>
#include<cstring>
#include<algorithm>
using namespace std;
const int N = 50005;
const int INF = 0x3f3f3f3f;
int n, ans[N], num, Size;

int cnt, head[N], son[N], vis[N];
struct Node
{
    int to, next;
    Node(){}
    Node(int a, int b):to(a), next(b){}
}edge[2*N];

void init()
{
    cnt = num = 0;
    Size = INF;
    memset(son, 0, sizeof(son));
    memset(vis, 0, sizeof(vis));
    memset(head, -1, sizeof(head));
}

void add(int u,int v)
{
    edge[cnt] = Node(v, head[u]);
    head[u] = cnt++;
}

void Dfs(int u)
{
    int Max = 0;
    vis[u] = 1;
    for(int i = head[u]; i != -1; i=edge[i].next)
    {
        int v = edge[i].to;
        if(vis[v]) continue;
        Dfs(v);
        son[u] += son[v] + 1;
        Max = max( Max, son[v] + 1);
    }
    Max = max(Max, n - son[u] - 1);
    if( Max < Size )
    {
        Size = Max;
        num=0;
        ans[num++] = u;
    }
    else if(Max == Size)
        ans[num++] = u;
}
int main()
{
    while(scanf("%d", &n)!=EOF)
    {
        init();
        for(int i= 1; i<n;i++)
        {
            int u, v;
            scanf("%d%d", &u ,&v);
            add(u, v);
            add(v, u);
        }

```

```
        Dfs(1);
        sort(ans, ans + num);
        // ans就是重心;
        if(num>2) while(1);
        for(int i=0;i<num;i++)
            printf("%d␣", ans[i]);
        printf("\n");
    }
    return 0;
}
```

3.20 树分治

3.20.1 点分治

```
/*************************************************************************
  > File Name: poj1741.cpp
  > Author: Bai Yan
  > 题意:
  > Created Time: 2016年10月11日 星期二 08时26分21秒
 ************************************************************************/
#include<iostream>
#include<cstdio>
#include<cstring>
#include<algorithm>
using namespace std;
const int INF = 0x7fffffff;
const int N = 10005;

int n , k, dis[N],d[N], vis[N], son[N], f[N], root, num, sum, ans;

struct Node
{
    int to, next, w;
    Node(){}
    Node(int a, int b, int c):to(a), next(b), w(c){}
}edge[N<<1];

int cnt, head[N];

void init()
{
    cnt = 0;
    memset(head, -1, sizeof(head));
}

void add(int u, int v, int w)
{
    edge[cnt] = Node(v, head[u], w);
    head[u] = cnt++;
}

void Dfs_root(int u, int fa)
{
    son[u] = 1;
    f[u] = 0;
    for( int i = head[u]; i != -1; i = edge[i].next)
    {
        int v = edge[i].to;
        if( v == fa || vis[v]) continue;
        Dfs_root(v, u);
        son[u] += son[v];
        f[u] = max( f[u], son[v] );
    }
    f[u] = max( f[u], sum - son[u]);
    if( f[u] < f[root] )
        root = u;
```

```
}
void Dfs_Dis( int u, int fa)
{
    dis[num++] = d[u];
    for( int i = head[u]; i != -1; i = edge[i].next)
    {
        int v = edge[i].to;
        if( v == fa || vis[v] ) continue;
        d[v] = d[u] + edge[i].w;
        Dfs_Dis(v , u);
    }
}
int cala( int u, int now)
{
    num = 0; d[u] = now;
    Dfs_Dis( u , 0);
    sort( dis, dis + num );
    int i = 0, j = num - 1, ret = 0;
    while( i < j ) // 相向搜索。
    {
        if(dis[i] + dis[j] <= k ){ret += j-i; i++;}
        else j--;
    }
    return ret;
}
void Dfs(int u)
{
    ans += cala( root , 0 );
    vis[root] = 1;
    for( int i = head[u]; i != -1; i = edge[i].next)
    {
        int v = edge[i].to;
        if( vis[v] ) continue;
        ans -= cala( v, edge[i].w);
        sum = son[v];
        root = 0;
        Dfs_root(v, root);
        Dfs(root);
    }
}

int main()
{
    while(scanf("%d%d", &n,&k) != EOF)
    {
        if( n ==0 ) break;
        init();
        for(int i = 1; i < n; i++)
        {
            int u, v, w;
            scanf("%d%d%d", &u, &v, &w);
            add(u, v, w);
            add(v, u, w);
        }
        memset( vis, 0, sizeof(vis) );
        sum = n; root = ans = 0;
        f[0] = INF;
        Dfs_root(1, 0);
        Dfs(root);
        printf("%d\n", ans);
    }
    return 0;
}
```

3.20.2 链分治

Chapter 4

Dp-动态规划

4.1 背包

4.2 LIS-最长上升子序列

4.3 数位 Dp

Chapter 5

数论

5.1 一些理论

5.1.1 大整数取模

把大整数写成“从左向右”的形式：如：$1234 = ((1 * 10 + 2) * 10 + 3) * 10 + 4$. 然后根据 $(n + m)\%p = ((n\%p) + (m\%p))\%p$，每步取模。

```
scanf("%s%d", &s, &p);
int len = strlen(s);
if(s[0] == '0' && len == 1){
   printf("divisible\n");
   continue;
}
if(p < 0) p = -p; //判断负数
int ans = 0, st = 0;
if(s[0] == '-') st = 1;
for(int i = st; i < len; i++){
   ans = (int)(((ll) ans * 10 + s[i] - '0') % p); //注意爆int
}
if(ans == 0) printf("divisible\n");
else printf("not␣divisible\n");
```

5.1.2 哥德巴赫猜想（$1 + 1$ 问题）

大于 2 的所有偶数都可以表示为两个素数的之和，大于 5 的所有奇数均可以表示为三个素数之和。陈景润证明了：所有大于 2 的偶数均是一个素数和只有两个素数因数的合数之和（$1 + 2$ 问题），称为陈氏定理。

5.1.3 费马小定理

p 是素数，且 $a\ !\equiv 0\ (mod\ p)$，则：$a^{p-1} \equiv 1(mod\ p)$ 。

5.1.4 素数定理

$f(x) \approx \frac{x}{ln(x)}$ （$f(x)$ 为不超过 x 的素数的个数）
10^8 级别的素数筛筛选出的素数个数约为 $6 * 10^6$ 级别
10^7 级别的素数筛筛选出的素数个数约为 $7 * 10^5$ 级别

5.1.5 算数基本定理

任何一个大于 1 的自然数 n，若其不为素数则必可唯一的分解为有限个素数的乘积，即 $n = p_1^{a1} * p_2^{a2} * p_3^{a3} * ... * p_k^{ak}$，那么同时 n 的因子个数为 $num = \prod(a_i + 1)(1 \le i \le k)$【$LightOJ\ 1341$】。

5.1.6 $ax+by=c$ 的任意整数解

a,b,c 为任意整数，若 $ax+by=c$ 的一组整数解为 (x_0,y_0)，则它的任意整数解为 (x_0+kb',y_0-ka'). 其中 $a'=\frac{a}{gcd(a,b)},b'=\frac{b}{gcd(a,b)}$,$k$ 取任意整数。

证明:令 $g=gcd(a,b)$. 设另一组解为 (x_1,y_1),则 $ax_0+by_0=ax_1+by_1(=c)$. 移项:$a(x_1-x_0)=b(y_0-y_1)$. 同除以 g 可得:$\frac{a}{g}*(x_1-x_0)=\frac{b}{g}*(y_0-y_1)$. 令 $a'=\frac{a}{g},b'=\frac{b}{g}$。则 $a'(x_1-x_0)=b'(y_0-y_1)$ 此时 a' 与 b' 互质. 所以:x_1-x_0 一定是 b' 的整数倍，设倍数为 k。则有: $x_1-x_0=kb',y_0-y_1=ka'$，可得: $x_1=x_0+kb',y_1=y_0-ka'$. 结论加强: a,b,c 为任意整数，$g=gcd(a,b)$，方程 $ax+by=g$ 的一组解是 (x_0,y_0). 则当 c 是 g 的倍数时，$ax+by=c$ 的一组解为 $(\frac{x_0*c}{g},\frac{y_0*c}{g})$. 当 c 不为 g 的倍数时 $ax+by=c$ 无整数解。

5.1.7 n 是 m 的倍数，$[1,n]$ 中和 m 互素数个数

对于两个正整数 m 和 n,如果 n 是 m 的倍数,那么 $1\sim n$ 中与 m 互素的数的个数为 $\frac{n}{m}*\phi(m)$.($\phi(m)$ 为小于等于 m 的且与 m 互素的数的个数)

5.1.8 p 为奇素数，$1\sim(p-1)$ 模 p 的逆元对应全部 $1\sim(p-1)$ 中的所有数，既是单射也是满射

也就是 $(1+\frac{1}{2}+\frac{1}{3}+...+\frac{1}{(p-1)})\ mod(\ p)=(1+2+...+(p-1))\ mod(\ p)$.

5.1.9 调和级数求和

令 $h[n]$ 表示 n 级调和级数的和，即:$h[n]=\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+...+\frac{1}{n}$. 则当 $n>10^6$ 时可以用公式:

$$h[n]=(log(n*1.0)+\frac{log(n+1.0)}{2.0}+\frac{1.0}{(6.0*n*(n+1))}-\frac{1.0}{(30.0*n*n*(n+1)*(n+1))}+r$$

其中 r 为欧拉常数:$r=0.57721566490153286060651209008240243104215933593992$。上述公式误差不超过 10^{-8}.

5.1.10 如果 $gcd(a,m)=1$，那么 $gcd(a+k*m,m)=1,k\in Z$

【$POJ\ 2773$】

证明: 只需要证明 $gcd(a+m,m)=1$. 令 $d=gcd(a+m,m)$，设 $a+m=d*p_1,m=d*p_2$且$p_1>p_2$,，移项可得:$a=d*(p_1-p_2)$，如果 $d!=1$，那么 $gcd(a,m)!=1$，与原条件不符，故 $gcd(a,m)=1$，同理可证 $gcd(a+k*m,m)=1,k\in Z$.

5.1.11 指数降幂公式

$$A^x\ \%\ p=A^{x\%\phi(p)+\phi(p)}\ \%\ p\ \ (x\geq\phi(p))$$

其中 $\phi(p)$ 是 p 的欧拉函数值

5.2 素数筛

```
void GetPrime(int n ) //获得 n 以内的所有质数
{
    prime_cnt = 0;
    int m = sqrt(n + 0.5);
    memset(vis, 0, sizeof(vis));
    for(int i = 2; i <= m; i++){
        if(vis[i] == 0){
```

```
            prime[prime_cnt++] = i;
            for(int j = i * i; j < n; j+= i){
                vis[j] = 1;
            }
        }
    }
}

//线性时间素数筛：利用每个合数必有一个最小素因子，每个合数仅被它的最小素因子筛去
//代码核心：if(i % prime[j] == 0) break;
void GetPrime()
{
    prime_cnt = 0;
    memset(vis, 0, sizeof(vis));
    for(int i = 2; i < MAX_N; i++){
        if(vis[i] == 0) { prime[prime_cnt++] = i; }
        for(int j = 0; j < prime_cnt && i * prime[j] < MAX_N; j++){
            vis[i * prime[j]] = 1;
            //将所有合数标记， prime[j] 是 i * prime[j] 的最小素因子
            if(i % prime[j] == 0) break;
        }
    }
}
```

5.2.1 区间素数筛

```
const int MAX_N = 100010; // 最大区间长度

int prime_cnt;
int prime_list[MAX_N];
bool prime[MAX_N]; //prime[i] = true:i + L 是素数

void SegmentPrime(int L, int R)
{
    int len = R - L + 1; // 区间长度
    for(int i = 0; i < len; i++) prime[i] = true; // 初始全部为素数
    int st = 0;
    if(L % 2) st++; // L 是奇数
    for(int i = st; i < len; i += 2) { prime[i] = false; }
    // 相当于 i + L 是合数，把 [L, R] 的偶数都筛掉
    int m = (int)sqrt(R + 0.5);
    for(int i = 3; i <= m; i += 2){ // 用 [3, m] 之间的数筛掉 [L, R] 之间的合数
        if(i > L && prime[i - L] == false) { continue; }
        // i 是 [L, R] 区间内的合数,此时 [L, R] 区间中中以 i 为基准的合数已经被筛掉了
        int j = (L / i) * i; // 此时 j 是 i 的倍数中最接近 L (小于等于 L )的数
        if(j < L) j += i; // j >= L
        if(j == i) j += i;
        // 如果 j>=L 且 j==i 且 i 为素数，则相当于 [L, R] 中的 i 也为素数，所以要 j+=i
        j -= L; // 把 j 调整为 [0, len) 之间
        for(; j < len; j += i) {
            prime[j] = false;
        }
    }
    if(L == 1) prime[0] = false;
    if(L <= 2) prime[2 - L] = true; // 特别注意 2 的情况!
    prime_cnt = 0;
    for(int i = 0; i < len; i++){
        if(prime[i]) prime_list[prime_cnt++] = i + L;
    }
}
```

5.3 分解质因数

```
void factor(int x)
```

```
{ // factot: 存质因数, factor_cnt[i] : 存 factor[i] 的指数,  factor_num : 质因数的个数
   GetPrime();
   factor_num = 0;
   memset(factor_cnt, 0, sizeof(factor_cnt));
   for(int i = 0; i < prime_cnt && prime[i] * prime[i] <= x; i++){
        if(x % prime[i] == 0){
            int tmp = 0;
           while(x % prime[i] == 0){
               x /= prime[i];
               tmp ++;
           }
           factor[factor_num] = prime[i];
           factor_cnt[factor_num++] = tmp;
        }
   }
}
```

5.3.1 给定 n 求满足 $a^p = n$ 的最大 p

【LightOJ 1220】

需要考虑 n 的正负。对于正数 n 只要对 n 进行质因子分解，然后在所有质因子的幂中找 gcd 即可。对于负整数 n 需要考虑质因子的幂为偶数的情况，因为得到的质因子实际上是它的相反数，如果是偶数次幂的话得到的是正数，所以需要将偶数次幂除以 2 找到把幂变为奇数。例如对于 -16，分解质因子得到 2 和幂次 4．需要将 $\frac{4}{2}$ 得 2，再除以 2 得 1，所以答案就是 1．然而对于 $n = -32$ 来说，就不用了，因为得到 2 的幂次是 5 是个奇数，答案就是 5．

5.3.2 求满足 $lcm(i,j) = n(1 \le i \le j \le n \le 10^{14})$ 的 (i,j) 对数

【LightOJ 1236】

假设 n 质因子分解为：$n = p_1^{e_1} * p_2^{e_2} * p_3^{e_3} * ... * p_k^{e_k}$。对于任意 p_i 的幂 e_i，当对 i,j 进行质因数分解后，相应的到的 p_i 的幂为 a_i 和 b_i，那么一定满足 $max(a_i, b_i) = e_i$．如果 $a_i = e_i$，那么 b_i 可以取 $[0, e_i]$，共 $e_i + 1$ 种，如果 $a_i < e_i$，那么 b_i 只取 e_i，这时 a_i 可以取 $[0, e_i)$，共 e_i 种。合起来一共是 $2 * e_i + 1$ 种。所以对每个质因子这样考虑的话可以得到：$ans = \prod(2 * e_i + 1)$．但是要考虑到 $i \le j$，所以 $ans = \frac{ans}{2}$，但是上面的考虑在所有的 p_i 当中 $a_i = e_i$ 同时 $b_i = e_i$ 只考虑了一次，也就是 $i = j = n$ 只考虑了一次。所以还要 $ans = ans + 1$．

5.4 逆元

5.4.1 介绍

对于正整数 a, p 满足 $ax \equiv 1(\%\ p)$ 的最小正整数 x 称为 a 的逆元。
乘法逆元的存在条件是：$gcd(a, p) = 1$。
$\equiv$：同余符号。$a \equiv b(mod\ n)$ 的含义是"a 和 b 关于模 n 同余"，即 $a\ mod\ n = b\ mod\ n$，其充要条件是：$a - b$ 是 n 的整数倍．当 $gcd(a, n) = 1$ 时，该方程有唯一解，否则该方程无解。
特例：
1：当 p 是素数且 $a\ != 0(mod\ p)$ 时由费马小定理可得：$x \equiv a^{(p-2)}(mod\ p)$．
2：已知 $b \mid a$，则 $(\frac{a}{b})\ mod\ m = \frac{a\ mod\ (mb)}{b}$。这个式子通常用于模数和分母不互素的情况，这样就不能用费马小定理和扩展欧几里德求解，必须将模数扩大为 $m * b$，最后答案除以 b。
证明：假设 $\frac{a}{b}\ mod(m) = x$，
即：$\frac{a}{b} = k * m + x(x < m)$
$--> a = k * b * m + b * x$
$--> a\ mod\ (bm) = bx$
$--> \frac{a\ mod\ (bm)}{b} = x$

$--> \frac{a}{b}\ mod(m) = \frac{a\ mod\ (bm)}{b}\ =\ x$

5.4.2 模素数逆元连乘

如果有的题目需要用到 $1 \sim n$ 模 M 的所有逆元（M 为奇质数），如果用快速幂求解时间复杂度为 $O(M * log(M))$. 实际上有 $O(M)$ 的算法，递推式如下：

$$inv[i] = (M - \frac{M}{i}) * inv[M\ \%\ i]\%M$$

推导过程如下：
设 $t = \frac{M}{i}, k = M\%i --> t * i + k = 0\ (mod\ M) --> -t * i = k\ (mod\ M)$. 两边同时除以 $i * k$ 得：$--> -t * inv[k] = inv[i](mod\ M)$. 再把 t 和 k 替换掉最终得到：

$$inv[i] = (M - \frac{M}{i}) * inv[M\ \%\ i]\%\ M$$

初始化 $inv[1] = 1$. 这样就可以通过递推法求出 $1 \sim n$ 模 M 的所有逆元了。

```
ll inv(ll a, ll p) // 利用 inv[a] = (p - p / a) * inv[p % a] % p 求逆元
{ // 需要保证 a < p 且 gcd(a, p) = 1
   if(a == 1) return 1;
   return inv(p % a, p) * (p - p / a) % p;
}
```

5.5 欧拉函数

5.5.1 介绍

定义：$\phi[n]$：小于等于 n 且与其互素的正整数的个数
欧拉定理：$a^{\phi(n)} \equiv 1(mod\ n)\ \ (gcd(a,\ n) = 1)$
特例是费马小定理。利用这个定理求模意义下的乘法逆元:$a^{-1} \equiv a^{\phi(n)-1}(mod\ n)$
性质：

- $\phi(1) = 1$
- $\phi(N) = N \cdot \prod \frac{p-1}{p}$ (p 为 N 的所有素因数)
- $\phi(p^k) = p^k - p^{k-1} = (p-1) \cdot p^{k-1}$，其中 p 为素数
- $\phi(m * n) = \phi(m) \cdot \phi(n)$，其中 $gcd(m, n) = 1$
- $\sum_{d|n} \phi(d) = n$
- 当 p 为素数时：

$$\phi(t * p) = \begin{cases} p * \phi(t) & t\%p = 0 \\ (p-1) * \phi(t) & t\%p! = 0 \end{cases}$$

- 当 $n > 1$ 时，$1...n$ 中与 n 互质的整数和是 $\frac{n\phi(n)}{2}$
- 当 $n \geq 3$ 时，$\phi[n]$ 是偶数
- 两相邻质数之间合数的欧拉函数值小于较小的素数，所以对于一个数 x 要找到最小的 t 使得 t 的欧拉函数值 $\phi[t] \geq x$ 则 t 必然是大于 x 的第一个素数。

5.5.2 求单个数的欧拉函数

求欧拉函数：先令 $\phi[i]=i$，根据性质 2，遍历所有素数 p，令 $\phi[kp]=\frac{\phi[kp]}{p}*(p-1)$

```
// 任何一个大于 1 的自然数 n ，若其不为素数则必可唯一的分解为有限个素数的乘积
int Euler(int n)
{
    int ans = 1;
    for(int i = 2; i * i <= n; i++){
        if(n % i == 0){
            n /= i;
            ans *= (i - 1);
            while(n % i == 0){
                n /= i;
                ans *= i;
            }
        }
    }
    if(n > 1) ans *= (n -1);
    return ans;
}
```

```
int Euler(int n)
{
    int res = n, a = n;
    for(int i = 2; i * i <= a; i++){
        if(a % i == 0){
            res = res / i * (i - 1); // 先进行除法防止数据溢出
            while(a % i ==0) a /= i;
        }
    }
    if(a > 1) res = res / a * (a - 1);
    return res;
}
```

5.5.3 欧拉筛

```
void GetPhi() // 埃氏筛
{
    phi[1] = 1;
    for(int i = 2; i < MAX_N; i++){
        if(phi[i] == 0){
            for(int j = i; j < MAX_N; j += i){
                if(!phi[j]) phi[j] = j;
                phi[j] = phi[j] / i * (i - 1);
            }
        }
    }
}
```

```
bitset<MAX_N> bs;
int prime_cnt, prime[MAX_N], phi[MAX_N];
void GetPhi() // 欧拉筛,同时获得素数表和欧拉表
{
    prime_cnt = 0;
    bs.set();
    for(int i = 2; i < MAX_N; ++i ) {
        if(bs[i] == 1) {
            prime[prime_cnt++] = i;
            phi[i] = i - 1;
        }
        for(int j = 0; j < prime_cnt && i * prime[j] < MAX_N; ++ j ) {
            bs[i * prime[j]] = 0;
            if(i % prime[j]) {
                phi[i * prime[j]] = (prime[j] - 1) * phi[i];
            }else {
```

```
            phi[i * prime[j]] = prime[j] * phi[i];
            break;
        }
    }
}
}
```

[UVA 11428 GCD - Extreme (II): 给定 n 求: $G=\sum_{i=1}^{i<N}\sum_{j=i+1}^{j\le N}GCD(i,j)(n\le 4000000)$

令 $sum[n]$ 为题式中答案。考虑递推 $sum[n]=sum[n-1]+gcd(1,n)+gcd(2,n)+gcd(3,n)+...+gcd(n-1,n)$。令 $f[n]=gcd(1,n)+gcd(2,n)+gcd(3,n)+...+gcd(n-1,n)$. 设满足 $gcd(x,n)=t...①$ 的 $x(x<n)$ 的个数有 $h[t]$ 个，显然 x,t 均是 n 的约数，又因为不包含 $gcd(n,n)$，所以 $t<n$。将等式 ① 两边同除以 t 可得:$gcd(\frac{x}{t},\frac{n}{t})=1$. 所以 $h[t]=\phi[\frac{n}{t}]$. 那么枚举 n 的约数可以得到:$f[n]=\sum(t*\phi[\frac{n}{t}])$($t$ 为 n 的所有约数) $-n$(相当于去掉 $gcd(n,n)$) 需要预处理 $f[n]$ 然后对于每个 n 有:$sum[i]=sum[i-1]+f[i]$,$sum[1]=0$; 递推即可。

```
typedef long long ll;
const int MAX_N = 4000010;

int n;
ll phi[MAX_N], f[MAX_N], sum[MAX_N];

void init()
{
    GetPhi();
    memset(f, 0, sizeof(f));
    for(int i = 1; i < MAX_N; i++){
        for(int j = i; j < MAX_N; j += i){
            f[j] += i * phi[j / i];
        }
    }
}

int main()
{
    init();
    while(~scanf("%d", &n) && n){
        sum[1] = 0;
        for(int i = 2; i <= n; i++){
            sum[i] = sum[i - 1] + f[i] - i;
        }
        printf("%lld\n", sum[n]);
    }
    return 0;
}
```

5.6 扩展欧几里德

5.6.1 裴蜀定理

$ax+by=gcd(a,b)$□□□□□□(x,y).

求解过程: 因为 a$x+by=gcd(a,b)$ 且 $gcd(a,b)=gcd(b,a\%b)$。那么有: $bx+(a\%b)y=gcd(b,a\%b)$，即有:$bx+(a\%b)y=ax+by$. 所以: $bx'+(a-\frac{a}{b}*b)y'=ax+by$□ 移项 :$ay'+b(x'-\frac{a}{b}*y')=ax+by$. 得到: $x=y',y=(x'-\frac{a}{b}*y')$，也就是 $x'=y-\frac{a}{b}*x,y'=x$. 所以可以使用递归求解。

```
//需保证系数 a, b 应同为正数，但是求解出来的 x, y 可正可负
int ex_gcd(int a, int b, int& x,int& y)
{
    if(b == 0) {
        x = 1, y = 0;
        return a;
    }
    int r = ex_gcd(b, a % b, y, x);
```

```
    y -= a / b * x;
    return r;
}
```

解释 1: $y-=\frac{a}{b}*x$

由前面的推导可知: $x'=y-\frac{a}{b}*x, y'=x$, 其中 x,y 是真正解, x',y' 是递归调用时的下一层的解。所以需要将上一层的 x 传递给下一层的 y', 另一方面因为 x 和 y 是未知的所以不能直接将上一层的 $y-a/b*x$ 传递给下一层的 x', 但是可以先传递 y, 相当于人为的增加了 $\frac{a}{b}*x$, 所以当计算出真正的 x,y 时需要减去 $\frac{a}{b}*x$. 解释 2: $x=1,y=0$

递归的最后一层的方程式的是: $gcd(a,b)*x+0*y=gcd(a,b)$. 要使这个式子恒成立, 显然需要 $x=1$。递归的倒数第二层的方程式是: $k*gcd(a,b)*x'+gcd(a,b)*y'=gcd(a,b)$. 其中 k 为任意整数。约掉 $gcd(a,b)$ 可得: $k*x'+y'=1$. 此式要恒成立显然 $x'=0,y'=1$。又因为此式中的 $x'=y,y'=x-\frac{a}{b}*y$. 所以 $y=0$. 当然这时 $y'=x-\frac{a}{b}*y=1-\frac{a}{b}*0=1$. 正好也是符合的。

5.6.2 求解逆元 $ax\equiv 1(mod\ n)$ 的最小非负数解

先求解:$ax+ny=gcd(a,n)$. 如果 $gcd(a,n)=1$, 则 $ans=(x\%n+n)\%n$. 否则无解。所以题目中经常是 n 为一个很大的素数, 这样保证了 $gcd(a,n)=1$.
这里得到的是最小非负数解 x , 如果要求 x 为正数还要在 $return$ 时特判:$if(x==0)x=n;$

```
int Inv(int a, int n) // Module Inverse 模逆元
{
    int d, x, y;
    d = ex_gcd(a, n, x, y);
    if(d == 1) return (x % n + n) % n;
    // a 和 n 的最小公倍数是 1 ,此时解存在
    else return -1; //no solution
}
```

5.6.3 求所有解中 $C=|x|+|y|$ 最小值

假设基础解为 (x_0,y_0), 那么通解可以表示为:$x=x_0+k*b, y=y_0-k*a(k\in Z)$。如果令 $x=0$ 得: $k=-\frac{x_0}{b}$...①, 令 $y=0$ 得: $k=\frac{y_0}{a}$...② 当两者同时满足并根据分数的性质:$\frac{a}{b}=\frac{c}{d}=\frac{(a+c)}{(b+d)}$ 可得:$k=\frac{(y_0-x_0)}{(b+a)}$, 但是考虑到这个数的真实值可能为浮点数, 需要左右考虑。

```
ll solve(ll a, ll b, ll t) // 方程为 a * x + b * y = t
{
    ll x, y, d, res;
    d = ex_gcd(a, b, x, y);
    if(t % d) return -1;
    a /= d, b /= d, t /= d;
    x *= t, y *= t; //此时 x, y 为基础解
    res = (ll)1e18;
    ll tmpx, tmpy, c = (y - x) / (a + b);
    for(int i = c - 1; i <= c + 1; i++){ //左右考虑
        tmpx = x + i * b, tmpy = y - i * a;
        res = min(res, abs(tmpx) + abs(tmpy));
    }
    return res;
}
```

5.6.4 求最小非负整数解 x 和此时的 y

```
ll extra = 0;
if(x < 0) extra = 1;
ll t = x / b - extra;
printf("%lld␣%lld\n", x - t * b, y + t * a);
```

5.7 模线性方程组

输入正整数 a,b,n，解方程 $ax \equiv b(mod\ n).a,b,n \le 10^9$

把 $ax \equiv b(modn)$ 转化为 $ax-ny=b$，当 $d=gcd(a,n)$ 不是 b 的约数时无解，否则两边同时除以 d，得到 $a'x-n'y=b'$，即 $ax' \equiv b'(modn')$(这里 $a'=\frac{a}{d},b'=\frac{b}{d},n'=\frac{n}{d}$)．此时 a' 和 n' 已经互素，因此只需要左乘 a' 在模 n' 下的逆，则解为 $x \equiv (a')^{-1}*b'(mod\ n')$，这个解是模 n' 剩余系中的一个元素。还需要把解表示成模 n 剩余系中的元素。
令 $p=(a')^{-1}*b'$，上述解相当于 $x=p,p+n',p+2*n',p+3*n'...$。对于模 n 来说，假定 $p+i*n'$ 和 $p+j*n'$ 同余，则 $(p+i*n')-(p+j*n')=(i-j)*n'$ 是 n 的倍数。因此 $(i-j)$ 必须是 d 的倍数。也就是说，在模 n 剩余系下，$ax \equiv b(modn)$ 恰好有 d 个解，为 $p,p+n',p+2*n',...,p+(d-1)*n'$.

5.7.1 利用扩展欧几里德求解模线性同余方程 $ax \equiv b(modn)$

如果 $gcd(a,b)$ 不能整除 c 则 $ax+by=c$ 无整数解。对于 $ax \equiv b(modn)(n>0)$，上式等价于二元一次方程 $ax-ny=b$

```
bool ModularLinearEquation(int a, int b, int n)
{
    int x, y, x0;
    int d = ex_gcd(a, n, x, y); //d = gcd(a,n)
    if(b % d) return false; // d 不能整除b
    x0 = x * (b / d) % n; //basic solution
    for(int i = 1;i <= d; i++){
        printf("%d\n", (x0 + i * (n / d) % n));
    }
    return true;
}
```

[SGU 106] 给出 a,b,c,x_1,x_2,y_1,y_2 求满足 $ax+by+c=0$ 且 $x \in [x_1,x_2],y \in [y_1,y_2]$ 的 x,y 有多少组。

相当于问在直线 $ax+by+c=0$ 上有多少整点 (x,y) 满足 $x \in [x_1,x_2],y \in [y_1,y_2]$．扩展欧几里德的应用。需要特别注意 $a=0,b=0,c=0$ 的特判。

```
typedef long long ll;
double eps = 1e-8;

ll a, b, c, x1, yy1, x2, y2, ans;

ll ex_gcd(ll n, ll m, ll& x, ll& y) // 返回值是 gcd(a, b)
{
    if(m == 0){
        x = 1, y = 0;
        return n;
    }
    ll d = ex_gcd(m, n % m, y, x);
    y -= n / m * x;
    return d;
}

void ModularLinearEquation() // 模线性方程组
{
    ll x, y, x0, y0;
    ll d = ex_gcd(a, b, x, y);// 求解方程ax+by=gcd(a, b)
    if(c % d) {
        printf("0\n");
        return ;
    }
    // 此时的 x, y 是方程 ax + by = gcd(a, b) 的一组基础解
    a /= d, b /= d, c /= d;
    x0 = x * c, y0 = y * c; // x0, y0 是 ax + by = c 的一组基础解
    // a * x + b * y = c 通解满足:  a * (x0 + k * b) + b * (y0 - k * a) = c
    //所以:  x' = x0 + k * b ...①, y' = y0 - k * a ...②, k ∈ Z
    //先求出满足 1:x1 <= x0 + k * b <= x2 的 k 的范围 [l1, r1]
    //再求出满足 2:yy1 <= y0 - k * a <= y2 的 k 的范围 [l2, r2] 两者取交集即可
    //l1 = ceil((x1 - x0)/b), r1 = floor((x2 - x0)/b)
    //l2 = ceil((y0 - y2)/a), r2 = floor((y0 - yy1)/a)
```

```
    ll l1 = (ll)ceil((double)(x1 - x0) / b);
    ll r1 = (ll)floor((double)(x2 - x0) / b);
    ll l2 = (ll)ceil((double)(y0 - y2) / a);
    ll r2 = (ll)floor((double)(y0 - yy1) / a);
    ll l = max(l1, l2), r = min(r1, r2);
    ans = (r - l + 1 < 0) ? 0 : (r - l + 1);
    printf("%lld\n", ans);
    return ;
}

int main()
{
    while(~scanf("%lld%lld%lld", &a, &b, &c)){
        scanf("%lld%lld%lld%lld", &x1, &x2, &yy1, &y2);
        int flag = 0;
        c = -c; // 移项，得到方程a * x + b * y = -c
        // 下面将方程的所有系数 a, b, c 变为非负数
        if(c < 0){
            c = -c , a = -a, b = -b;
        }
        if(a < 0){
// 需要将 a 取相反数，相当于把 x 也取了相反数，所以需要将 [x1, x2] 的范围变为[-x2, -x1]
            ll tmpx1 = x1, tmpx2 = x2;
            x1 = -tmpx2, x2 = - tmpx1;
            a = -a;
        }
        if(b < 0){ // 同理
            ll tmpyy1 = yy1, tmpy2 = y2;
            yy1 = -tmpy2, y2 = -tmpyy1;
            b = -b;
        }

        if(a == 0 && b == 0) { //0 * x + 0 * y = c
            flag = 1;
            if(c == 0) ans = (x2 - x1 + 1) * (y2 - yy1 + 1);
            else ans = 0;
        }else if(a == 0){ // b != 0
            flag = 1;
            if(c % b == 0){ // 0 * x + b * y = c --> b * y = c
                ll tmpy = c / b;
                if(yy1 <= tmpy && tmpy <= y2) ans = x2 - x1 + 1;
                else ans = 0;
            }else ans = 0;
        }else if(b == 0) { //a != 0
            flag = 1;
            if(c % a == 0) { // a * x + 0 * y = c --> a * x = c
                ll tmpx = c / a;
                if(x1 <= tmpx && tmpx <= x2) ans = y2 - yy1 + 1;
                else ans = 0;
            }else ans = 0;
        }
        // 以上处理完了 a = 0 或者 b = 0 的所有特例
        if(flag) printf("%lld\n", ans);
        else ModularLinearEquation();
    }
    return 0;
}
```

5.8 中国剩余定理

给定整数 n 和 n 个整数 $a[i], m[i]$，求满足方程 $x \equiv a[i](mod\ m[i])(0 \le i < n)$ 的最小正整数解。

例如求解：满足 $n = 3, a[0] = 2, m[0] = 3, a[1] = 3, m[1] = 5, a[2] = 2, m[2] = 7$. 即：
$n\%3 = 2 --> 5 * 7 * a\%3 = 1 --> a = 2, a[0] = 2 \therefore a = 4$ 得 $:5 * 7 * 4$
$n\%5 = 3 --> 3 * 7 * b\%5 = 1 --> b = 1, a[1] = 3 \therefore b = 3$ 得 $:3 * 7 * 3$
$n\%7 = 2 --> 3 * 5 * c\%7 = 1 --> c = 1, a[2] = 2 \therefore c = 2$ 得 $:3 * 5 * 2$

累加得：$5*7*4+3*7*3+3*5*2=233$
取余得：$233\%(3*5*7)=23$，所以最终答案就是 23

5.8.1 模数互素

```
//前提条件:  m[i] > 0 ,且 m[i] 中任意两数互质
ll CRT(ll a[], ll m[], ll n)
{ // Chinese Remainder Theorem
   ll M = 1, ans = 0;
   for(int i = 0; i < n; i++) { M *= m[i]; }
   for(int i = 0; i < n; i++){
      ll x, y, e;
      e = M / m[i];
      ex_gcd(e, m[i], x, y);
      ans = (ans + e * x % M * a[i] % M) % M;
   }
   return (ans + M) % M; //最小正整数解
}
```

5.8.2 模数不互素

当 m[i] 不满足两两互素时，假设 $x\equiv a1(mod\ m1)...(1), x\equiv a2(mod\ m2)...(2)$.
令 $d=gcd(m1,m2)$. 由 (1)(2) 得：$x=a1+m1*k1, x=a2+m2*k2$. 合并可得：$m1*k1=(a2-a1)+m2*k2$
两边同除以 d 得：$m1/d*k1=(a2-a1)/d+m2/d*k2$.
也就是：$m1*k1/d\equiv(a2-a1)/d(mod\ m2/d)$.
也就是：$m1*k1=(a2-a1)(mod\ m2)$.
易知 $k1$ 有多个解，假设 k' 为 $k1$ 的最小非负整数解则：$k1\equiv k'(mod\ m2/d)$
即：$k1=k'+(m2/d)*C$（C 为某一整数）. 将其带入 $x=a1+m1*k1$
可得：$x=a1+m1*(k'+m2/d*C)$
也就是：$x=a1+m1*k'+m1*m2/d*C$。
也就是：$x=(a1+m1*k')(mod\ m1*m2/d)$
也就是：$x=(a1+m1*k')(mod\ lcm(m1,m2))....(3)$;
那么求解出 k' 后，就可以将方程 (1)(2) 合并为一个方程 (3) 了，依次迭代就能出解了。
[HDU 1573 X 问题]：求解在 $(0,N]$ 区间满足 $x\equiv a[i](mod\ m[i])(0\le i<M)$ 的 x 的个数。

利用中国剩余定理求解出最小非负整数解 $a0$(如果有解) 和解的周期 $m0$，假设解的个数为 k 个则：$a0+k*m0<=N-->k<=(N-a0)/m0,(N>=a0)$。当 $a0=0$ 时解的个数是 $(N-a0)/m0$. 当 $a0>0$ 时解的个数是 $(N-a0)/m0+1$.

```
typedef long long ll;
const int MAX_N = 15;

int T, M;
ll N, a0, m0, a[MAX_N], m[MAX_N];

ll ex_gcd(ll aa, ll bb, ll& xx, ll& yy)
{
   if(bb == 0) {
      xx = 1, yy = 0;
      return aa;
   }
   ll dd = ex_gcd(bb, aa % bb, yy, xx);
   yy -= aa / bb * xx;
   return dd;
}

//求解: m0 * x = (aa - a0) (mod mm)
bool ModularLinearEquation(ll& m0, ll& a0, ll mm, ll aa)
{
   ll x, y, d;
   d = ex_gcd(m0, mm, x, y);
   if(labs(aa - a0) % d != 0) return false;
```

```
    mm /= d;
    x = x * (aa - a0) / d % mm;
    a0 += x * m0;
    m0 *= mm;
    a0 = (a0 % m0 + m0) % m0;
    return true;
}

bool CRT(ll& m0, ll&a0)
{
    bool flag = true;
    m0 = 1, a0 = 0; //任意数 mod m0 = a0 恒成立
    for(int i = 0; i < M; i++){
        if(ModularLinearEquation(m0, a0, m[i], a[i]) == false){
            flag = false;
            break;
        }
    }
    return flag;
}

int main()
{
    scanf("%d", &T);
    while(T--){
        scanf("%lld%d", &N, &M);
        for(int i = 0; i < M; i++){
            scanf("%lld", &m[i]);
        }
        for(int i = 0; i < M; i++){
            scanf("%lld", &a[i]);
        }
        if(CRT(m0, a0) == false || N < a0) printf("0\n");
        else {
            //printf("a0 = %lld m0 = %lld\n", a0, m0);
            printf("%lld\n", (N - a0) / m0 + (a0 == 0 ? 0 : 1));
        }
    }
    return 0;
}
```

5.9 法雷级数

5.9.1 介绍

真分数：若 p,q 是正整数，$0 < \frac{p}{q} < 1$，则称 $\frac{p}{q}$ 为真分数。
定理：若 $\frac{a}{d}, \frac{c}{d}$ 是最简真分数（也可以是 $\frac{0}{1}, \frac{1}{1}$），且 $\frac{a}{b} < \frac{c}{d}$ 则有：

- 数 $\frac{a+c}{b+d}$ 是一个最简真分数
- $\frac{a}{b} < \frac{a+c}{b+d} < \frac{c}{d}$

5.9.2 n 级法雷数列

对于任意给定的自然数 n，将分母小于等于 n 的不可约的真分数按升序排列，并且在第一个分数之前加上 $\frac{0}{1}$，在最后一个分数之后加上 $\frac{1}{1}$，这个序列称为 n 级法雷数列，用 F_n 表示。例如：F_5：$\frac{0}{1}, \frac{1}{5}, \frac{1}{4}, \frac{1}{3}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}, \frac{1}{1}$.

5.9.3 法雷数列的构造

应用上面的定理，如果 $\frac{a}{b}$□$\frac{c}{d}$ 是一个法雷数列，则在它们中间可以插入 $\frac{(a+c)}{(b+d)}$，这样一直二分构造，直到不能构造为止（分母大于 n）. 例如 F_5 的构造：

$step1$：在 $\frac{0}{1}$ 和 $\frac{1}{1}$ 之间插入 $\frac{1}{2}$ 可得：$\frac{0}{1}, \frac{1}{2}, \frac{1}{1}$
$setp2$：在每对相邻两个数之间插入 1 个数得：$\frac{0}{1}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{1}{1}$
$setp3$：重复上述操作 ：$\frac{0}{1}, \frac{1}{4}, \frac{1}{3}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{2}{3}, \frac{3}{4}, \frac{1}{1}$
$setp4$：重复上述操作需保证分母不大于 5：$\frac{0}{1}, \frac{1}{5}, \frac{1}{4}, \frac{1}{3}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}, \frac{1}{1}$
构造结束。

5.9.4 求 n 级法雷级数个数

设 F_n 的个数为 $f[n]$ 个，则 Fn 比 $F(n-1)$ 增加的的分母是 n，所以增加的个数是分子比 n 小且与 n 互质的数个数，这就是欧拉函数 $\phi[n]$！
递推式：$f[n] = f[n-1] + phi[n]$。所以有 $f[n] = 1 + \phi[1] + \phi[2] + ... + \phi[n]$。

5.9.5 性质

- 因为 $n \geq 3$ 时，欧拉函数 $\phi[n]$ 是偶数，由此可得：除 $f[1] = 2$ 是偶数外，法雷级数其他各级的个数都是奇数。
- 如果 $\frac{a}{b}, \frac{c}{d}$ 是相邻的两项，则 $abs(a * d - b * c) = 1$
- 如果 $\frac{a}{b}, \frac{c}{d}, \frac{e}{f}$ 是相邻的三项，则 $\frac{a+e}{b+f} = \frac{c}{d}$。

```
// 求 MAX_N 以内每个数的法雷数
int f[MAX_N];
void GetFarey()
{
    GetEuler(); //先筛得欧拉表
    f[1] = 2;
    for(int i = 2; i < MAX_N; i++){
        f[i] = f[i - 1] + phi[i];
    }
    for(int i = 1; i < 10; i++){
        printf("%d␣", f[i]);
    }
    printf("\n");
}
```

可以边筛素数边计算欧拉函数

```
void GetFarey()
{
    memset(phi, 0, sizeof(phi));
    phi[1] = 1;
    for(int i = 2; i < MAX_N; i++){
        if(phi[i] == 0){ // i 同时也是素数
            for(int j = i; j < MAX_N; j += i){
                if(phi[j] == 0) phi[j] = j;
                phi[j] = phi[j] / i * (i - 1);
            }
        }
    }
    f[1] = 2;
    for(int i = 2; i < MAX_N; i++){
        f[i] = f[i - 1] + phi[i];
    }
}
```

构造 n 级法雷级数

```
int farey[MAX_N][2], total;
// faery[i][0],farey[i][1] 分别是第 i 个 farey 数列的分子和分母
void Make_Farey_Sequence(int a, int b, int c, int d)
{
    if(a + c > n || b + d > n) return ;
    Make_Farey_Sequence(a, b, a + c, b + d);
    // 保证了法雷序列的有序性
```

```
    farey[total][0] = a + c;
    farey[total++][1] = b + d;
    Make_Farey_Sequence(a + c, b + d, c, d);
}

int main()
{
    scanf("%d", &n);
    farey[0][0] = 0, farey[0][1] = 1;
    total = 1;
    Make_Farey_Sequence(0, 1, 1, 1);
    farey[total][0] = 1;
    farey[total++][1] = 1;
    for(int i = 0; i < total; i++){
        printf("%d/%d\n", farey[i][0], farey[i][1]);
    }
    printf("\n");
}
```

5.10 原根

5.10.1 介绍

定义：设 $m>1, gcd(a,m)=1$，使得 $a^r \equiv 1(mod\ m)$ 成立的最小 r，成为 a 对模 m 的阶。
定义 1：如果 a 模 m 的阶等于 $\phi(m)$，则称 a 为模 m 的一个原根。
定义 2：在模运算中，若存在一个整数 g 使得式子 $g^k \equiv a(mod\ n)$ ，$1 \le k < n$ 结果 a 各不相同，则称 g 为模 n 的一个原根。
例如：
$3^1 = 3 \ \ = 3^0 * 3 = 1 \times 3 = 3 \equiv 3(mod\ 7)$
$3^2 = 9 \ \ = 3^1 * 3 = 3 \times 3 = 9 \equiv 2(mod\ 7)$
$3^3 = 27 \ = 3^2 * 3 = 2 \times 3 = 6 \equiv 6(mod\ 7)$
$3^4 = 81 \ \ = 3^3 * 3 = 6 \times 3 = 18 \equiv 4(mod\ 7)$
$3^5 = 243 = 3^4 * 3 = 4 \times 3 = 12 \equiv 5(mod\ 7)$
$3^6 = 729 = 3^5 * 3 = 5 \times 3 = 15 \equiv 1(mod\ 7)$
所以 3 为模 7 的一个原根。
定理：

- 模 m 有原根的充要条件是：$m = 2, 4, p^a, 2 * p^a$（p 为奇素数）
- 如果模 m 有原根，那么一共有 $\phi(\phi(m))$ 个原根。$\phi(m)$ 为欧拉函数值
- 如果 p 为素数，那么 p 一定存在原根，且模 p 的原根的个数为 $\phi(p-1)$.

5.10.2 求模素数 p 的所有原根

对 $p-1$ 素因子分解，即 $p-1 = p_1^{a_1} * p_2^{a_2} * p_3^{a_3} * ... * p_k^{a_k}$ 是 $p-1$ 的标准分解式，若恒有 $g^{\frac{p-1}{p_i}} ! = 1(mod\ p)$ 成立则 g 就是 p 的原根。枚举 $g: 2 \sim p-1$. 对于合数求原根只需把 $p-1$ 换成 $\phi(p)$ 即可。

```
int factor[MAX_N], factor_cnt;
void Factor(int x) // 获得 x 的所有素因数
{
    factor_cnt = 0;
    int t = (int) sqrt(x + 0.5);
    for(int i = 0; prime[i] <= t; i++){
        if(x % prime[i] == 0){
            factor[factor_cnt++] = prime[i];
            while(x % prime[i] == 0) x /= prime[i];
        }
    }
    if(x > 1) factor[factor_cnt++] = x;
}
```

```

ll quick_pow(ll a, ll b, ll m)
{
    ll ans = 1, tmp = a % m;
    while(b){
        if(b & 1) ans = ans * tmp % m;
        tmp = tmp * tmp % m;
        b >>= 1;
    }
    return ans;
}

int ans[MAX_N];
int main()
{
    int p, total;
    GetPrime();
    while(~scanf("%d", &p)){
        Factor(p - 1);
        total = 0;
        for(int g = 2; g < p; g++){
            bool flag = true;
            for(int i = 0; i < factor_cnt; i++){
                int t = (p - 1) / factor[i];
                if(quick_pow(g, t, p) == 1){
                    flag = false;
                    break;
                }
            }
            if(flag){
                ans[total++] = g;
            }
        }
        for(int i = 0; i < total; i++){
            printf("%d\n", ans[i]);
        }
    }
}
```

5.11 $BSGS$ 算法

$BSGS$ 算法用于求解:$a^x = b(mod\ p)$ 在已知 $a \in [2,p), b \in [1,p),(p$ 为质数$)$ 的情况下的最小解 x。时间复杂度 $O(\sqrt{p})$。

令 $x = i * m + j$，其中 $m = ceil(\sqrt{p}), 0 \le i < m, 0 \le j < m$，那么就相当于求解: $a^{i*m+j} \equiv b(mod\ p) -- > a^j = b * a^{-i*m}(mod\ p), a^{-i*m}$ 是 a^{i*m} 模 p 的逆元。所以可以先处理出 $a^j(mod\ p)$ 的答案放入一个 $hash$ 表中【$BabyStep$】，然后枚举 $i : 0 \sim m$【$GaintStep$】，查找 $b * a^{-i*m}(mod\ p)$ 是否在 $hash$ 表中出现，如果出现，令出现的编号为 id，则答案就是 $id + i * m$. 在时间允许的情况下，$hash$ 表采用 map 也可以。为什么枚举 $i < m$ 就可以了呢? 显然枚举 $i < m$ 就枚举完了 $x < p$ 的所有情况，当 $x \ge p$ 时可以令 $x = k * p + t(t < p)$，则 $a^x(mod\ p) = a^{k*p+t}(mod\ p) = a^{k*p} * a^t(mod\ p) = a^{k*k}(mod\ p) * a^t(mod\ p)$. 由费马小定理得:$a^p \equiv 1(mod\ p)$ $(p$ 为素数$)$，那么 $a^x(mod\ p) = 1^k(mod\ p) * a^t(mod\ p) = a^t(mod\ p)$。所以只需要枚举所有 x 小于 p 的情况就可以了，也就是枚举 $i < m$ 就可以了 $(m$ 是向上取整的$)$.

```
const ll MOD = 100007;
ll hs[MOD + 100], id[MOD + 100];

ll find(ll x)
{
    ll t = x % MOD;
    while(hs[t] != x && hs[t] != -1) t = (t + 1) % MOD;
    return t;
}

void insert(ll x, ll ii)
{
    ll pos = find(x);
```

```
    if(hs[pos] == -1){
        hs[pos] = x;
        id[pos] = ii;
    }
}

ll get(ll x)
{
    ll pos = find(x);
    return hs[pos] == x ? id[pos] : -1;
}

ll inv(ll a, ll p) //求解: a * x ≡ 1 (mod p)
{
    ll x, y, d;
    d = ex_gcd(a, p, x, y); // ax + py = gcd(a, p) 本段代码省略了ex_gcd()
    return d == 1 ? (x % p + p) % p : -1;
}

ll BSGS(ll a, ll b, ll p)
{//求解a^x ≡ b (mod p)
    memset(hs, -1, sizeof(hs));
    memset(id, -1, sizeof(id));
    ll m = (ll)ceil(sqrt( p + 0.5));
    ll tmp = 1;
    for(ll i = 0; i < m; ++ i) {
        insert(tmp, i);
        tmp = tmp * a % p;
    }
    ll base = inv(tmp, p); //tmp = a ^ m % p
    ll res = b;
    for(ll i = 0; i < m; ++ i) {
        if(get(res) != -1) return i * m + get(res);
        res = res * base % p;
    }
    return -1;
}
```

5.11.1 扩展 $BSGS(gcd(a,p)\ !=\ 1)$

初始化 $cnt=0$(消因子轮数)，$d=1$(消掉的 gcd 乘积)。令 $tmp=gcd(a,p)$ 当 $tmp!=1$ 时，修改变量值:$b/=tmp$(先判断 b 是否是 tmp 的倍数)，$p\ /=\ tmp, d=\frac{a}{tmp}*d$ %p; 通过若干轮消掉 a,p 的因子使得最终 $gcd(a,p)=1$. 这时再调用普通的 $BSGS$ 得到解为 res，则最终答案是 $res+cnt$。但是这样求得的解是 $\geq cnt$ 的，需要先判断下是否有 $<cnt$ 的解。考虑 cnt 的最大值。因为每次消去的最小因子是 2，可以得到 cnt 的最大值是 $log_2 p$，先跑一遍 50 次遍历的循环是绰绰有余的。

```
ll gcd(ll x, ll y)
{
    return y == 0 ? x : gcd(y, x % y);
}

ll solve(ll a, ll b, ll p)
{
    ll tmp = 1;
    for(int i = 0; i <= 50; ++ i) {
        if(tmp == b) return i;
        tmp = tmp * a % p;
    }
    ll cnt = 0, d = 1 % p;
    while((tmp = gcd(a, p)) != 1) {
        if(b % tmp) return -1;
        b /= tmp;
        p /= tmp;
        d = a / tmp * d % p;
        cnt++;
    }
    b = b * inv(d, p) % p;
```

```
    ll ans = BSGS(a, b, p); // 这里就是调用普通的BSGS
    if(ans == -1) return -1;
    else return ans + cnt;
}
```

5.12 莫比乌斯反演

5.12.1 积性函数

定义域为 N^+ 的函数 f，对于任意两个互质的正整数 a,b□ : $gcd(a,b)=1$，均满足 $f(ab)=f(a)*f(b)$，则函数 f 被称为积性函数。假如对于任意两个正整数 a,b 均有 $f(ab)=f(a)*f(b)$，则称 f 为完全积性函数。
欧拉函数时积性函数，但不是完全积性函数。
积性函数的性质：

- $f(1)=1$
- 考虑一个大于 1 的正整数 N，设 $N=\prod p_i^{a_i}$，其中 p_i 为互不相同的质数，那么对于一个积性函数 f,$f(N)=f(\prod p_i^{a_i})=\prod f(p_i^{a_i})$，如果 f 还满足完全积性，则 $f(N)=\prod f(p_i)^{a_i}$
- 若 $f(n),g(n)$ 均为积性函数，则函数 $h(n)=f(n)g(n)$ 也为积性函数。
- 若 $f(n)$ 为积性函数，则函数 $F(n)=\sum_{d|n} f(d)$ 也是积性函数，反之亦然。

5.12.2 狄利克雷卷积

对于函数 $f,g,$ 定义它们的卷积为 $(f*g)(n)=\sum_{d|n} f(d)g(\frac{n}{d})$。
性质：

- f□$(g$□$h)=(f$□$g)$□h
- f□$(g+h)=f$□$g+f$□h
- f□$g=g$□f
- 两个积性函数的狄利克雷卷积仍是积性函数

5.12.3 莫比乌斯反演公式

$$F(n)=\sum_{d|n} f(d)\rightarrow f(n)=\sum_{d|n}\mu(d)F(\frac{n}{d})$$

$$F(n)=\sum_{n|d} f(d)\rightarrow f(n)=\sum_{n|d}\mu(\frac{d}{n})F(d)$$

5.12.4 莫比乌斯函数 μ

- $$\mu(d)=\begin{cases}1 & n=1\\ (-1)^k & n=p_1p_2\dots p_k(p_i\ are\ \ all\ \ prime\ \ numbers)\\ 0 & other\ \ cases\end{cases}$$
- $\sum_{d|n}\mu(d)=(n==1?1:0)$

- 对任意正整数 n 有:$\sum_{d|n}\frac{\mu(d)}{d}=\frac{\phi(n)}{n}$

设 $f(n)=\sum_{d|n}\phi(d)$，又有 $\sum_{d|n}\phi(d)=n$，所以 $f(n)=n$，根据莫比乌斯反演可得:
$\phi(n)=\sum_{d|n}\mu(d)f(\frac{n}{d})=\sum_{d|n}\frac{\mu(d)n}{d}$

5.12.5 线性筛求解积性函数

观察线性筛中的步骤，筛掉 n 的同时还得到了它的最小质因数 p，我们希望知道 p 在 n 中的次数，这样就能利用 $f(n)=f(p^k)f(\frac{n}{p^k})$ 求出 $f(n)$.
令 $n=pm$，由于 p 是 n 的最小质因子，若 $p^2|n$，则 $p|m$ 并且 p 也是 m 的最小质因子，这样在筛的同时记录每个合数最小质因子的次数，就能算出新筛去合数最小质因子的次数。但是这样时不够的，我们需要快速求出 $f(p^k)$，这时就要结合 f 函数的性质考虑。
例如欧拉函数 ϕ,$\phi(p^k)=(p-1)p^{k-1}$ 因此在进行筛时，如果 $p|m$，就乘上 p，否则乘上 $p-1$. 而对于莫比乌斯函数 μ，只有当 $k=1$ 时 $\mu(p^k)=-1$,$\mu(p^k)=0$，和欧拉函数一样根据 m 是否被 p 整除进行判断。

```
void GetMu()
{
    memset(vis, 0, sizeof(vis));
    mu[1] = 1;
    prime_cnt = 0;
    for(int i = 2; i < MAX_N; i++) {
        if(vis[i] == 0) {
            prime[prime_cnt++] = i;
            mu[i] = -1;
        }
        for(int j = 0; j < prime_cnt && i * prime[j] < MAX_N; j++) {
            vis[i * prime[j]] = 1;
            if(i % prime[j]) mu[i * prime[j]] = -mu[i];
            else{
                mu[i * prime[j]] = 0;
                break;
            }
        }
    }
}
```

[ZOJ 3435]:$\sum_{i=0}^{i=a}\sum_{j=0}^{j=b}\sum_{k=0}^{k=c}[gcd(i,j,k)==1],a,b,c\in[1,1000000]$

1. 当 $i=j=k=0$ 时是不成立的。

2. 当 i,j,k 中有两个为 0 时，只有三种情况 $(0,0,1),(0,1,0),(1,0,0)$.

3. 当 i,j,k 中有一个为 0 时，相当于求 $gcd(i,j)=1,gcd(i,k)=1,gcd(j,k)=1)$ 的对数。

4. 当 i,j,k 均大于 0 时，相当于求 $gcd(i,j,k)=1(i\in[1,a],j\in[1,b],k\in[1,c])$ 的对数。

对于 3.4 两种情况用莫比乌斯反演即可。

```
GetMu();
int a, b, c;
while(~scanf("%d%d%d", &a, &b, &c)){
a--, b--, c--;
if(a > b) swap(a, b);
if(a > c) swap(a, c);
if(b > c) swap(b, c);
// a <= b <= c
ll ans = 3, tmp;
```

```
int last, x, y, z;
for(int i = 1; i <= b; i = last + 1) { // 注意枚举的范围
   last = i;
   x = a / i, y = b / i, z = c / i;
   if(i <= a){
      last = min(a / x, b / y);
      last = min(last, c / z);
   }else { // 防止出现除以0
      last = min(b / y, c / z);
   }
   tmp = (ll) x * y * z + (ll)x * y + (ll)x * z + (ll)y * z;
   ans += tmp * (sum[last] - sum[i - 1]);
}
printf("%lld\n", ans);
```

$gcd(x,y)=p$ (p 为质数,$x\in[1,n],y\in[1,m]$)，有序对 (x,y) 有多少对？$n,m\in[1,10^7]$

$(2,3)$ 和 $(3,2)$ 是不同的有序对

定义:$f(d)$ 为满足 $gcd(x,y)=d$ $(x\in[1,n],y\in[1,m]$ 的 (x,y) 的对数。

定义:$F(d)$ 为满足 $d\mid gcd(x,y)$ $(x\in[1,n],y\in[1,m]$ 的 (x,y) 的对数。

那么有:$F(n)=\sum_{n|d}f(d)=\frac{n}{d}*\frac{m}{d}$，根据第二种形式的莫比乌斯反演有:

$$f(x)=\sum_{x|d}\frac{\mu(d)}{x}F(d)=\sum_{x|d}\mu(\frac{d}{x})*\frac{n}{d}*\frac{m}{d}$$

题目要求是求 $gcd(x,y)$ 为质数，对于每个质数 p 相当于求 $x\in[1,\frac{n}{p}],y\in[1,\frac{m}{p}]$ 的 $gcd(x,y)=1$ 的有序对 (x,y) 的对数. 我们枚举每个质数 p，就有 $ans=\sum_p^{min(n,m)}(\sum_d^{min(n,m)}\mu(d)*\frac{n}{pd}*\frac{m}{pd})$，直接枚举的话会 TLE，所以继续优化。令 $T=pd$，那么可得:$ans=\sum_p^{min(n,m)}(\sum_d^{min(n,m)}\mu(d)*\frac{n}{T}*\frac{m}{T})=\sum_{T=1}^{min(n,m)}\frac{n}{T}*\frac{m}{T}*(\sum_{p|T}\mu(\frac{T}{p}))$. 所以我们可以预处理出所有的 T 对应的 $\sum_{p|T}\mu(\frac{T}{p})$.

设 $sum(x)=\sum_{p|x}\mu(\frac{x}{p})$，这里 p 为素数，令 $g(x)=\mu(\frac{x}{p})$. 我们枚举每一个 k，得到 $g(kx)=\mu(\frac{kx}{p})$，分情况讨论有:

- $x\%k==0$,

$$g(kx)=\begin{cases}\mu(x) & k=p\\ 0 & k!=p\end{cases}$$

- $x\%k!=0$,

$$g(kx)=\begin{cases}\mu(x) & k=p\\ \mu(x)-g(x) & k!=p\end{cases}$$

$\lfloor\frac{N}{d}\rfloor$ 的取值只有 $2\lfloor\sqrt{N}\rfloor$ 种，同理 $\lfloor\frac{M}{d}\rfloor$ 的取值也只有 $2\lfloor\sqrt{M}\rfloor$ 种，并且相同取值对应的 d 是一个连续的区间，因此 $\lfloor\frac{N}{d}\rfloor$ 和 $\lfloor\frac{M}{d}\rfloor$ 都相同的区间最多有 $2\lfloor\sqrt{N}\rfloor+2\lfloor\sqrt{M}\rfloor$ 个，这样 d 的枚举量就缩小为 $O(\sqrt{N}+\sqrt{M})$ 了。

```
typedef long long ll;
const int MAX_N = 10000010;

bitset<MAX_N> bs;
int prime_cnt, prime[MAX_N];
ll g[MAX_N], mu[MAX_N], sum[MAX_N];
//主函数中调用GetMu()
void GetMu()
{
   bs.set();
   mu[1] = 1;
   prime_cnt = 0;
   for(int i = 2; i < MAX_N; ++i) {
      if(bs[i]) {
         prime[prime_cnt++] = i;
         mu[i] = -1;
```

```
            g[i] = 1;
        }
        for(int j = 0; j < prime_cnt && i * prime[j] < MAX_N; ++j) {
            bs[i * prime[j]] = 0;
            if(i % prime[j]) {
                mu[i * prime[j]] = - mu[i];
                g[i * prime[j]] = mu[i] - g[i];
            }else {
                mu[i * prime[j]] = 0;
                g[i * prime[j]] = mu[i];
                break;
            }
        }
    }
    for(int i = 1; i < MAX_N; ++i) {
        sum[i] = sum[i - 1] + g[i];
    }
}

inline ll solve(int n, int m)
{
    int top = min(n, m), last;
    ll ans = 0;
    for(int i = 1; i <= top; i = last + 1) {
        last = min(n / (n / i), m / (m / i));
        ans +=(ll) (n / i) * (m / i) * (sum[last] - sum[i - 1]);
    }
    return ans;
}
```

[BZOJ 2154]:$\sum_{i=1}^{i=n}\sum_{j=1}^{j=m} lcm(i,j)$ % 100000009 $(n,m \le 10^7)$

$$\begin{aligned} ans &= \sum_{d=1}^{d=n}\sum_{i=1}^{i=n}\sum_{j=1}^{j=m}\frac{i*j}{d} \ (gcd(i,j)=d) \\ &= \sum_{d=1}^{d=n}]\sum_{i=1}^{\frac{n}{d}}\sum_{j=1}^{\frac{m}{d}}\frac{i*j*d^2}{d}(gcd(i,j)=1) \\ &= \sum_{d=1}^{n} d\sum_{i=1}^{\frac{n}{d}}\sum_{j=1}^{\frac{m}{d}}(i*j)(gcd(i,j)=1) \end{aligned}$$

定义 $f(d)=\sum_{i=1}^{a}\sum_{j=1}^{b}(i*j)\ \ (gcd(i,j)=d)$
定义 $F(d)=\sum_{i=1}^{i=a}\sum_{j=1}^{j=b}(i*j)\ \ (gcd(i,j)$ % $d=0)$
易得 $F(1)=sum(a,b)=\frac{a(a+1)}{2}*\frac{b(b+1)}{2}$
反演可得:

$$\begin{aligned} f(1) &= \sum_{i=1}^{i=a}\sum_{j=1}^{j=b}(i*j)(gcd(i,j)=1,a\le b) \\ &= \sum_{x=1}^{a}\mu(x)*x^2\sum_{i=1}^{\frac{a}{x}}\sum_{j=1}^{\frac{b}{x}}(i*j) \\ &= \sum_{x=1}^{a}\mu(x)*x^2\sum_{i=1}^{\frac{a}{x}}i*\sum_{j=1}^{\frac{b}{x}}j \\ &= \sum_{x=1}^{a}\mu_(x)*x^2*sum(\frac{a}{x},\frac{b}{x}) \end{aligned}$$

$$\therefore ans=\sum_{d=1}^{n} d\sum_{x=1}^{n'}\mu(x)*x^2*sum(\frac{n'}{x},\frac{m'}{x})(n'=\frac{n}{d},m'=\frac{m}{d})$$

此时如果预处理出 $\mu(x)*x^2$ 的前缀和求 ans 的复杂度是 $O(\sqrt{n}*\sqrt{n})=O(n)$，对于本题而言是不够的。令 $T=d*x$ 可得：$ans=\sum_{T=1}^{n} sum(\frac{n}{T},\frac{m}{T})\sum_{x|T}\frac{T}{x}*x^2*\mu(x)$．令 $h[T]=\sum_{x|n}\frac{T}{x}*x^2*\mu(x)$，预处理出 $h[T]$【因为 $h(T)$ 是积性函数可以线性筛】，那么时间复杂度就变为 $O(\sqrt{n})$ 总的时间复杂度为 $O(T\sqrt{n})$(T 是测试组数)

```
typedef long long ll;
const int MAX_N =10000010;
const ll mod = 100000009;

bitset<MAX_N> bs;
int prime_cnt, prime[MAX_N / 100 * 7];
ll h[MAX_N], sum[MAX_N];
// 主函数中调用GetMu()
void GetMu()
{
    bs.set();
    prime_cnt = 0;
    h[1] = sum[1] = 1;
    for(int i = 2; i < MAX_N; ++i) {
        if(bs[i]) {
            prime[prime_cnt++] = i;
            h[i] = (ll)i * (1 - i) % mod;
        }
        for(int j = 0; j < prime_cnt && i * prime[j] < MAX_N; ++j ){
            bs[i * prime[j]] = 0;
            if(i % prime[j]) { // i 和 prime[j] 互质
                h[i * prime[j]] = h[i] * h[prime[j]] % mod;
            }else {
// 从原始式子 (T)=sigma(mu(d)*d*d*(T/d)) , 对 T 质因子分解只需要考虑前两项
                h[i * prime[j]] = h[i] * prime[j] % mod;
                break;
            }
        }
    }
    for(int i = 1; i < MAX_N; ++i ) {
        sum[i] = ((sum[i - 1] + h[i]) % mod + mod) % mod;
    }
}

inline ll work(int n, int m)
{
    ll res1 = (ll) n * (n + 1) / 2 % mod;
    ll res2 = (ll) m * (m + 1) / 2 % mod;
    return res1 * res2 % mod;
}

inline ll solve(int n, int m)
{
    int top = min(n, m), last;
    ll res = 0;
    for(int i = 1; i <= top; i = last + 1) {
        last = min( n / (n / i), m / (m / i));
        res = (res + (sum[last] - sum[i - 1] + mod) % mod
               * work(n / i, m / i) % mod) % mod;
    }
    return res;
}
```

5.13 反素数

5.13.1 介绍

定义：对于任何正整数 n，其约数个数记为 $f(n)$，例如 $f(6)=4$，如果某个正整数满足 x：对任意的正整 $i(0<i<n)$ 数，都有 $f(i)<f(n)$，那么称为 n 反素数。

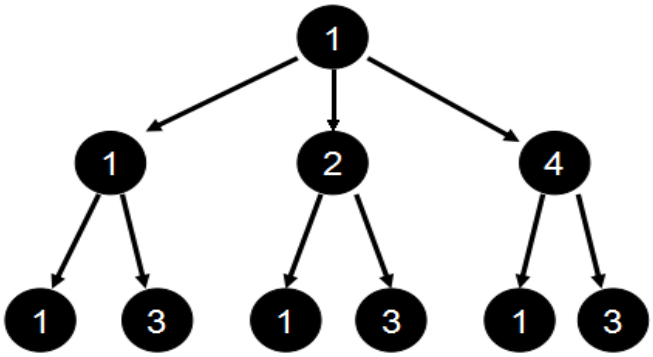
性质：
1)： 一个反素数的所有质因子必然是从 2 开始的连续若干个质数，因为反素数是保证约数个数为 x 的这个数 n 尽量小
2)： 同样的道理，如果 $n=2^{p_1}*3^{p_2}*5^{p_3}...$，那么必有 $p1 \geq p2 \geq p3...$

5.13.2 求最小的 n 使得其约数个数为 x

由算术基本定理定理我们知道：若一个数 $x=p_1{}^{a_1}p_2{}^{a_2}\square p_s{}^{a_s}$，那么 x 的约数个数

$$g(x)=\sum_{i=1}^{i=s}\prod(a_i+1)$$

例如:$12=2^2*3$，其约数个数为 6．建立搜索树：



这棵树除了第一层外，每一层对应着一个素数，从上到下递增；每一层的每一个节点对应着素数的幂，从左到右递增，每一条从根节点到叶节点的路径上数字相乘即为一个约数。我们要想获得 $g(n)=x$ 的最小正整数，就要求 n 的质因子尽可能小，且尽可能多，换而言之就是幂次尽可能为 1，所以就要对上面的树进行从上到下从左到右的 dfs 直到约数个数满足 x 。

【Codeforces 27E】给定一个数 n，求一个最小的正整数，使得的约数个数为 n.

```
typedef unsigned long long lint;
const lint inf = ~0ull;
const int prime[]= {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53};

int n;
lint ans;

void dfs(int depth, int limit, lint tmp, int num)
{
    if(num > n) return;
    if(num == n && tmp < ans) ans = tmp;
    for(int i = 1; i <= limit; ++i) { // i 相当于幂次
        if((double)tmp * prime[depth] > ans) break; // 不用扩展树的深度
        tmp *= prime[depth];
        if(n % (num * (i + 1)) == 0) {
            dfs(depth + 1, i, tmp, num * (i + 1));
        }
    }
}

int main()
{
    while(cin >> n){
        ans = inf;
        dfs(0, 63, 1, 1);
        cout << ans << endl;
    }
    return 0;
}
```

【URAL 1748】：给定一个数 n，求 $[1,n]$ 内约数个数最多的且数值最小的数，以及其约数个数 $(n \leq 10^{18})$。

```
//将搜索改为当前值 tmp > n 时终止，初始化ans = inf, cnt = 0
//主函数里调用 dfs(0, 63, 1, 1) ,初始化 prime[] 同例1
void dfs(int depth, int limit, lint tmp, int num)
{
    if(tmp > n) return;
    if(num > cnt || (num == cnt && tmp < ans)) {
        ans = tmp;
        cnt = num;
    }
    for(int i = 1; i <= limit; ++i) {
        if((double)tmp * prime[depth] > n) break;
        //需要用 double 强制类型转换，否则爆数据
        tmp *= prime[depth];
        dfs(depth + 1, i, tmp, num * (i + 1));

    }
}
```

5.14 卢卡斯定理

用于解决组合数取模问题 C_n^m %p $(m \leq n \leq 10^{18}, p$ 为素数)

当 $1 \leq m \leq n \leq 10^{18}, 2 \leq p \leq 10^5$ 且 p 是素数时，如果：

$$n = n_k p^k + n_{k-1} p^{k-1} + ... + n_1 p + n_0$$

$$m = m_k p^k + m_{k-1} p^{k-1} + ... + m_1 p + m_0$$

那么：

$$C_n^m \ \%p = \prod_{i=0}^{k} C_{n_i}^{m_i} \ (mod \ p)$$

```
inline ll C(ll a, ll b)
{ //计算组合数 C[a][b] ，如果是小数据并且模数固定可以预处理阶乘
    if(b > a) return 0;
    ll res = 1, x, y;
    for(int i = 1; i <= b; ++i) {
        x = (a + i - b) % mod;
        y = i % mod;
        res = res * x % mod * quick_pow(y, mod - 2) % mod; //模素数逆元
    }
    return res;
}
inline ll Lucas(ll n, ll m)
{
    if (n < 0 || m < 0 || m > n) return 0;
    if (m == 0) return 1;
    return C(n % mod, m % mod) * Lucas(n / mod, m / mod) % mod;
}
```

5.14.1 给定 n 求 $C_n^m (0 \leq m \leq n \leq 10^8)$ 为奇数的 m 个数

[HDU 4349]

即 C_n^m %$2 = 1$ 的 m 个数，考虑将 n 和 m 都表示成 2 的幂次组合形式，则任意的系数 n_i 和 m_i 非 0 即 1. 因为 $C_0^0 = C_1^0 = C_1^1 = 1$，所以根据 $Lucas$ 定理，如果 $n_i = 0$，则 $m_i = 0$，如果 $n_i = 1$，则 $m_i = 0$ 或 1，根据乘法原理每个 $n_i = 1$，对于 m_i 都有 2 种选择，那么 $ans = 2^{cnt}$，其中 cnt 是 n 分解为 2 的幂次形式系数为 1 的项个数

```
scanf("%d", &n);
cnt = 0;
while (n) {
    if (n & 1) cnt ++;
    n >>= 1;
}
printf("%d\n", 1 << cnt);
```

5.15 特殊方法

5.15.1 n^k 的高三位

对于给定的一个数 n，它可以写成 10^a，其中 a 为浮点数，则 $n^k=(10^a)^k=10^{a*k}=10^x*10^y$；其中 x,y 分别是 $a*k$ 的整数部分和小数部分．对于 $t=n^k$ 这个数，它的位数由 10^x 决定，它的位数上的值则有 10^y 决定，因此我们要求 t 的前三位，只需要将 10^y 求出，在乘以 100，就得到了它的前三位。$fmod(x,1)$ 可以求出 x 的小数部分。(或者使用 $floor$ 函数)

```
high_three_digits = (int)pow(10.0, 2.0 + fmod(k * 1.0 * log10(n * 1.0), 1)).
或者double t = 1.0 * k * log10(n * 1.0);
high_three_digits = (int)(pow(10.0, t - (int)floor(t) + 2.0));
```

5.15.2 约数个数之和，定义 $d(i)$ 为 i 的约数个数

$$\sum_{i=1}^{a}d(i)=\sum\lfloor\frac{a}{i}\rfloor$$

$$\sum_{i=1}^{a}\sum_{j=1}^{b}d(i*j)=\sum_{gcd(i,j)=1}\lfloor\frac{a}{i}\rfloor\lfloor\frac{b}{j}\rfloor$$

$$\sum_{i=1}^{a}\sum_{j=1}^{b}\sum_{k=1}^{c}d(i*j*k)=\sum_{gcd(i,j)=gcd(j,k)=gcd(i,k)=1}\lfloor\frac{a}{i}\rfloor\lfloor\frac{b}{j}\lfloor\frac{c}{k}\rfloor$$

这个性质可以推广到 n 维

[BZOJ 3994]: 求 $\sum_{i=1}^{n}\sum_{j=1}^{m}d(i*j)$，定义 $d(i)$ 为 i 的约数个数.$n,m\in[1,50000]$

$$ans=\sum_{gcd(i,j)=1}\lfloor\frac{n}{i}\rfloor\lfloor\frac{m}{j}\rfloor=\sum_{i=1}^{n}\lfloor\frac{n}{i}\rfloor\sum_{j=1}^{m}\lfloor\frac{m}{j}\rfloor$$

$g(n)=\sum_{i=1}^{n}\lfloor\frac{n}{i}\rfloor=\sum_{i=1}^{n}d(i)$，只需要预处理出 $g(n)$ 就可以在 $O(\sqrt{n})$ 时间范围内解决问题。如果选择分步加速的话，预处理的复杂度是 $O(n\sqrt{n})$，但是其实我们考虑每个数 i 的约数个数，然后 $g(n)$ 就是前缀和了。

在线性筛时每个合数是被最小质因子筛掉的，我们只需要记录这个每个数 m 最小质因子的幂次 $num[m],d[m]$ 记录 m 的约数个数，显然 $d(m)$ 是积性函数．对于 $m=i*prime[j]$，如果 i % $prime[j]!=0$，根据积性函数性质 $num[m]=1,d[m]=d[i]*d[prime[j]]$，否则 $num[m]=num[i]+1$，因为 m 的约数个数是:$(e_1+1)*(e_2+1)*...*(e_k+1),e_i$ 是质因子分解后各质因子的幂次，我们考虑 m 从 i 的转移过程，m 只比 i 在 $prime[j]$ 的幂次上多 1，所以 $d[m]=\frac{d[i]}{num[i]+1}*(num[i]+2)$.

```
void GetMu()//线性时间预处理
{
    bs.set();
    prime_cnt = 0;
    mu[1] = d[1] = 1;
    for(int i = 2; i < MAX_N; ++i) {
        if(bs[i]) {
            prime[prime_cnt++] = i;
```

```
            mu[i] = -1;
            num[i] = 1;
            d[i] = 2;
        }
        for(int j = 0; j < prime_cnt && i * prime[j] < MAX_N; ++j) {
            bs[i * prime[j]] = 0;
            if(i % prime[j]) {
                mu[i * prime[j]] = -mu[i];
                num[i * prime[j]] = 1;
                d[i * prime[j]] = d[i] * d[prime[j]];
            }else {
                mu[i * prime[j]] = 0;
                num[i * prime[j]] = num[i] + 1;
                d[i * prime[j]] = d[i] / (num[i] + 1) * (num[i] + 2);
                break;
            }
        }
    }
    for(int i = 1; i < MAX_N; ++i) {
        sum[i] = sum[i - 1] + mu[i];
        g[i] = g[i - 1] + d[i];
    }
}
void Get_g() // O(n * sqrt(n)) 的预处理
{
    int last;
    for(int n = 1; n < MAX_N; ++n) {
        for(int i = 1; i <= n; i = last + 1) {
            last = n / (n / i);
            g[n] += (ll) (last - i + 1) * (n / i);
        }
    }
}

inline ll solve(int n, int m)
{
    ll res = 0;
    int top = min(n, m), last;
    for(int i = 1; i <= top; i = last + 1) {
        last = min(n / (n / i), m / (m / i));
        res += (sum[last] - sum[i - 1]) * g[n / i] * g[m / i];
    }
    return res;
}
```

对于【Codeforces 235 E Number Challenge】：

$$\sum_{i=1}^{a}\sum_{j=1}^{b}\sum_{k=1}^{c} d(ijk), a, b, c \in [1, 2000]$$

利用上述结论反演可得：

$$Ans = \sum_{i=1}^{i=a} \lfloor \frac{a}{i} \rfloor \sum_{d}^{min(b,c)} (d) \sum_{d|j,(i,j)=1} \lfloor \frac{b}{j} \rfloor \sum_{d|k,(i,k)=1} \lfloor \frac{c}{k} \rfloor$$

记 $j = dj', k = dk'$，则：

$$Ans = \sum_{i=1}^{i=a} \lfloor \frac{a}{i} \rfloor \sum_{d}^{min(b,c)} (d) \sum_{gcd(i,dj')=1} \lfloor \frac{b}{dj'} \rfloor \sum_{gcd(i,dk')=1} \lfloor \frac{c}{dk'} \rfloor$$

因为 $gcd(i, dj') = 1$，我们先保证 $gcd(i, d) = 1$，然后枚举 $j' : 1 \rightarrow \frac{b}{d}$，保证 $gcd(i, j') = 1$，这样就可以使得 $gcd(i, dj') = 1$，累加即可。对于 $gcd(i, dk') = 1$ 同样处理。时间复杂度是：$O(a * b * log(b))$

```
//需要预处理 GetMu() 和 GetGcd()
inline ll work(int n, int x)
{
```

```
    ll res = 0;
    for(int i = 1; i <= n; ++i) {
        if(gcd[i][x] == 1) {
            res = (res + (n / i)) % mod;
        }
    }
    return res;
}

inline ll solve(int a, int b, int c)
{
    ll res = 0;
    int top = min(b, c);
    for(int i = 1; i <= a; ++i) {
        for(int d = 1; d <= top; ++d){
            if(gcd[i][d] == 1) {
                ll tmp = 0;
                tmp = (ll) (a / i) * mu[d] * work(b / d, i)
                        % mod * work(c / d, i) % mod;
                res = ((res + tmp) % mod + mod) % mod;
            }
        }
    }
    return res;
}
```

5.15.3 给定 k，求最小的 n 使得 n 的约数个数恰为 $n-k$ 个 $(k \leq 47777)$

【HDU 4542】

```
const int MAX_N = 50010;
int id[MAX_N];
void init()
{
    for(int i = 1; i < MAX_N; ++i) id[i] = i;
// 初始化 id[i] 最多有 i 个数与其互质
    for(int i = 1; i < MAX_N; ++i) {
        for(int j = i; j < MAX_N; j += i) id[j]--;
//根据 j 的约数有 i, id[j]--
        if(id[id[i]] == 0) id[id[i]] = i;
        // 因为我们最终是要将 id[k] 表示成约数个数为 n-k 的最小数 n ,
        // 如果此时 id[id[i]]=0 说明小于 i 的数中不存在数 x , 使得 x 的约数个数为 x-id[i]
        // 那么实际上使得约数个数恰为 n-id[i] 的最小数 n 只能是 i 了
        id[i] = 0;
// 显然小于等于 i 的数不存在数 x 使得 x 的约数个数恰为 x-i 个, 所以要赋 id[i]=0
    }
} // 对于 id[i] 等于 0 的 i , 实际上就不存在数 x 使得 x 的约数个数恰为 x-i 个
```

5.15.4 C_n^m mod $p(p = p_1 * p_2$，且 p_1, p_2 为素数）

我们把 C_n^m % p 的值记为 x，把 C_n^m % p_1 的值记为 $x1$，把 C_n^m % p_2 的值记为 $x2$，则有：

$$x \equiv x_1(\%\ p_1) \qquad x \equiv x_2(\%\ p_2)$$

因为 p_1, p_2 都是素数，所以 x_1 和 x_2 都可以用 $Lucas$ 定理求解出来。利用中国剩余定理求解同余方程。定义：

$$inv_1 \text{ 为:}(p_2 \text{ 在模 } p_1 \text{ 域下的逆元}) * p_2$$
$$inv_2 \text{ 为:}(p_1 \text{ 在模 } p_2 \text{ 域下的逆元}) * p_1$$

```
//quick_pow( a, b, c): (a ^ b) % c
inv1 = quick_pow(p2, p1 - 2, p1) * p2;
inv2 = quick_pow(p1, p2 - 2, p2) * p1;
```

那么答案就是：

$$x = (inv_1 * x_1 \ \ + inv_2 * x_2) \% \ p$$

Chapter 6

补充

6.1 Gauss 消元

```cpp
/**************************************************************************
模2 域的高斯消元
 *************************************************************************/
#include<stdio.h>
#include<algorithm>
#include<iostream>
#include<string.h>
#include<math.h>
using namespace std;

const int maxn = 1000;
int a[maxn][maxn+1], x[maxn]; //a是系数矩阵和增广矩阵，x是最后存放的解
                              // a[][maxn]中存放的是方程右面的值（bi）

int equ , var;     //equ是系数阵的行数，var是系数矩阵的列数（变量的个数）
int free_x[maxn];
int free_num , ans=100000000;

int abs1(int num) //取绝对值
{
    if (num>=0) return num;
    else
    return -1*num;
}

void Debug(void) //调试输出，看消元后的矩阵值，提交时，就不用了
{
    int i, j;
    for (i = 0; i < equ; i++)
    {
        for (j = 0; j < var + 1; j++)
        {
            cout << a[i][j] << "␣";
        }
        cout << endl;
    }
    cout << endl;
}

inline int gcd(int a, int b) //最大公约数
{
    int t;
    while (b != 0)
    {
        t = b;
        b = a % b;
        a = t;
    }
    return a;
```

```
}

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

void swap(int &a,int &b){int temp=a;a=b;b=temp;} //交换2个数

int Gauss()
{
    int ans = -1;
    int k , col = 0; // col 是当前处理的列， k 是当前处理的行
    for( k = 0; k < equ && col < var; ++k, ++col )
    {
        int max_r = k;
        for( int i = k + 1; i < equ ; i++ )
            if( a[i][col] > a[max_r][col] )
                max_r = i;
        if(max_r != k)
        {
            for(int i = k; i < var + 1; ++i)
                swap( a[k][i] , a[max_r][i] );
        }
        if( a[k][col] == 0 )
        {
            k--;
            continue;
            //如果当前行的col列为0;
            //则还在当前行上进行计算;
        }

        //使用当前行对下面所有行进行消元操作;
        for(int i = k+1;i < equ; ++i)
        {
            if(a[i][col] != 0)
            {
                int LCM = lcm(a[i][col],a[k][col]);
                int ta = LCM/a[i][col], tb = LCM/a[k][col];
                if(a[i][col]*a[k][col] < 0)
                    tb = -tb;
                for(int j = col;j < var + 1; ++j)
                    a[i][j] = (( a[i][j] * ta ) % 2 - ( a[k][j] * tb ) % 2 + 2 ) % 2; //a[i][j]只有0和1两种状态
            }
        }
    }

    // 上述代码是消元的过程，行消元完成
    // 接下来2行，判断是否无解
    // 注意K的值，k代表系数矩阵值都为0的那些行的第1行

    for(int i = k;i < equ; ++i)
        if(a[i][col] != 0) return -1; // 无解返回 -1


    //唯一解或者无穷解,k<=var
    //var-k==0 唯一解: var-k>0 无穷多解，自由解的个数=var-k
    //能执行到这，说明肯定有解了，无非是1个和无穷解的问题。
    //下面这几行很重要，保证秩内每行主元非0，且按对角线顺序排列，就是检查列

    for(int i = 0;i<equ; i++ )   //i 每一行主元素化为非零
      if(!a[i][i])
      {
          int j;
          for(j = i+1;j<var; j++)
              if(a[i][j])
                  break;
          //寻找第一个! =0 的列，交换到当前列;
          if(j == var)
              break;
          for(int k = 0;k < equ; ++k)
```

```
                swap(a[k][i],a[k][j]);
        }

    //-------------------------------------------------------------------------
    // ----处理保证对角线主元非0且顺序，检查列完成
    // free_num=k;
    // k 是矩阵的秩;
    // -------------------------------------------------------------------------

    if (var-k>0)
    {
        //无穷多解，先枚举解，然后用下面的回带代码进行回带;
        //这里省略了下面的回带的代码；不管唯一解和无穷解都可以回带，只不过无穷解//回带时，默认为最后几个自由变元=0而已。
        //下面是回带求解代码，当无穷多解时，最后几行为0的解默认为0;

        for(int tmp =0; tmp <(1<<(var-k)); tmp++)
        {
            int t = tmp;
            for(int i=k;i<var;i++)
            {
                x[i] = (t&1);
                t>>1;
            }
            for(int i = k-1;i >= 0; --i) //从消完元矩阵的主对角线非0的最后1行，开始往//回带
            {
                int tmp = a[i][var] % 2;
                for(int j = i+1;j < var; ++j) //x[i]取决于x[i+1]--x[var]啊，所以后面的解对前面的解有影响。
                    if(a[i][j] != 0)
                        tmp = ( tmp - ( a[i][j] * x[j] ) % 2 + 2 ) % 2;
                //if (a[i][i]==0) x[i]=tmp; //最后的空行时，即无穷解得
                //else
                x[i] = (tmp/a[i][i]) % 2; //上面的正常解
            }
            int anst=0;
            for(int i=0;i<var;i++)
                anst += x[i];
            if(ans == -1)
                ans = anst;
            else
                ans = min(ans,anst);
        }
       //回带结束了
    }

    if (var-k==0)//唯一解时
    {

        //下面是回带求解代码，当无穷多解时，最后几行为0的解默认为0;
        for(int i = k-1;i >= 0; --i)  //从消完元矩阵的主对角线非0的最后1行，开始往//回带
        {
            int tmp = a[i][var] % 2;
            for(int j = i+1;j < var; ++j) //x[i]取决于x[i+1]--x[var]啊，所以后面的解对前面的解有影响。
                if(a[i][j] != 0)
                    tmp = ( tmp - ( a[i][j] * x[j] ) % 2 + 2 ) % 2;

                //if (a[i][i]==0) x[i]=tmp; //最后的空行时，即无穷解得
                //else
                //while(tmp%a[i][i]) tmp += mod; 这一句是用于整数解;
            x[i] = (tmp/a[i][i]) % 2; //上面的正常解
        }

       //回带结束了
       for(int i=0;i<var;i++)
          ans += x[i];
    }
    return ans;
}


int main()
```

```
{
    memset(a,0, sizeof(a));
    for(int i=0;i<20;i++)
    {
        int t;
        scanf("%d", &t);
        a[i][20] = t^0;
    }
    for(int i=0;i<20;i++)
    {
        if(i-1>=0)
            a[i-1][i] = 1;
        a[i][i] = 1;
        if(i+1<20)
            a[i+1][i] = 1;
    }
    equ = 20;
    var = 20;
    int ans = Gauss();
    printf("%d\n",ans);
    return 0;
}
```

```
/*************************************************************************
普通的整数高斯消元
 *************************************************************************/
#include<stdio.h>
#include<algorithm>
#include<iostream>
#include<string.h>
#include<math.h>
using namespace std;

const int MAXN=50;



int a[MAXN][MAXN];  // 增广矩阵
int x[MAXN];   // 解集
bool free_x[MAXN];  // 标记是否是不确定的变元

inline int gcd(int a,int b)
{
    int t;
    while(b!=0)
    {
        t=b;
        b=a%b;
        a=t;
    }
    return a;
}
inline int lcm(int a,int b)
{
    return a/gcd(a,b)*b;//先除后乘防溢出
}

// 高斯消元法解方程组(Gauss-Jordan elimination).
// -2表示有浮点数解，但无整数解，
// -1表示无解，0表示唯一解，大于0表示无穷解，并返回自由变元的个数
// 有equ个方程，var个变元。增广矩阵行数为equ,分别为0到equ-1,列数为var+1列，分别为0到var.
int Gauss(int equ,int var)
{
    int i, j, k, ta, tb, LCM, temp, free_x_num, free_index;
    int max_r;  // 当前这列绝对值最大的行.
    int col;  // 当前处理的列

    for(int i=0;i<=var;i++)
    {
        x[i]=0;
```

```
        free_x[i]=true;
    }

    //转换为阶梯阵.
    col = 0;   // 当前处理的列
    for( k = 0; k < equ && col < var; k++, col++)
    {
        // 枚举当前处理的行.
        // 找到该col列元素绝对值最大的那行与第k行交换.(为了在除法时减小误差)
        max_r = k;
        for( i = k + 1; i < equ; i++)
            if( abs(a[i][col])>abs(a[max_r][col])) max_r = i;
        if(max_r!=k)  // 与第k行交换.
            for(j=k;j<var+1;j++) swap(a[k][j],a[max_r][j]);

        if(a[k][col]==0)
        {// 说明该col列第k行以下全是0了，则处理当前行的下一列.
            k--;
            continue;
        }

        for(i=k+1;i<equ;i++)
        {
            // 枚举要删去的行.
            if(a[i][col]!=0)
            {
                LCM = lcm(abs(a[i][col]),abs(a[k][col]));
                ta = LCM/abs(a[i][col]);
                tb = LCM/abs(a[k][col]);
                if(a[i][col]*a[k][col]<0)tb=-tb; //异号的情况是相加
                for(j=col;j<var+1;j++)
                    a[i][j] = a[i][j]*ta-a[k][j]*tb;
            }
        }
    }

    // 1. 无解的情况: 化简的增广阵中存在(0, 0, ..., a)这样的行(a != 0).
    for (i = k; i < equ; i++)
        if (a[i][col] != 0) return -1;

    // 对于无穷解来说，如果要判断哪些是自由变元，那么初等行变换中的交换就会影响，则要记录交换.

    // 2. 无穷解的情况: 在var * (var + 1)的增广阵中出现(0, 0, ..., 0)这样的行，即说明没有形成严格的上三角阵.
    // 且出现的行数即为自由变元的个数.
    if (k < var)
    {
        // 首先，自由变元有var - k个，即不确定的变元至少有var - k个.
        for (i = k - 1; i >= 0; i--)
        {
            // 第i行一定不会是(0, 0, ..., 0)的情况，因为这样的行是在第k行到第equ行.
            // 同样，第i行一定不会是(0, 0, ..., a), a != 0的情况，这样的无解的.
            free_x_num = 0; // 用于判断该行中的不确定的变元的个数，如果超过1个，则无法求解，它们仍然为不确定的变元.
            for (j = 0; j < var; j++)
                if (a[i][j] != 0 && free_x[j]) free_x_num++, free_index = j;

            if (free_x_num > 1) continue; // 无法求解出确定的变元.
            // 说明就只有一个不确定的变元free_index，那么可以求解出该变元，且该变元是确定的.
            temp = a[i][var];
            for (j = 0; j < var; j++)
                if (a[i][j] != 0 && j != free_index) temp -= a[i][j] * x[j];

            x[free_index] = temp / a[i][free_index]; // 求出该变元.
            free_x[free_index] = 0; // 该变元是确定的.
        }
        return var - k; // 自由变元有var - k个.
    }

    // 3. 唯一解的情况: 在var * (var + 1)的增广阵中形成严格的上三角阵.
    // 计算出Xn-1, Xn-2 ... X0.
    for (i = var - 1; i >= 0; i--)
    {
```

```
        temp = a[i][var];
        for (j = i + 1; j < var; j++)
        {
            if (a[i][j] != 0) temp -= a[i][j] * x[j];
        }
        if (temp % a[i][i] != 0) return -2; // 说明有浮点数解，但无整数解.
        x[i] = temp / a[i][i];
    }
    return 0;
}
int main(void)
{
    freopen("in.txt", "r", stdin);
    freopen("out.txt","w",stdout);
    int i, j;
    int equ,var;
    while (scanf("%d␣%d", &equ, &var) != EOF)
    {
        memset(a, 0, sizeof(a));
        for (i = 0; i < equ; i++)
        {
            for (j = 0; j < var + 1; j++)
            {
                scanf("%d", &a[i][j]);
            }
        }

        int free_num = Gauss(equ,var);

        if (free_num == -1) printf("无解!\n");
        else if (free_num == -2) printf("有浮点数解，无整数解!\n");
        else if (free_num > 0)
        {
            printf("无穷多解!␣自由变元个数为%d\n", free_num);
            for (i = 0; i < var; i++)
            {
                if (free_x[i]) printf("x%d␣是不确定的\n", i + 1);
                else printf("x%d:␣%d\n", i + 1, x[i]);
            }
        }
        else
            for (i = 0; i < var; i++)
                printf("x%d:␣%d\n", i + 1, x[i]);

        printf("\n");
    }
    return 0;
}
```

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

 > 带模数的高斯消元模板。要求逆元。
 **************************************************************************/
#include<iostream>
#include<cstring>
#include<algorithm>
#include<cstdio>
#include<cmath>
using namespace std;
typedef long long ll;
const int N = 305;
const int MOD = 10007;
int R;
struct Point
{
    int x, y;
    Point(int x=0, int y=0):x(x), y(y){}
};

Point operator - (Point a , Point b){ return Point(a.x-b.x, a.y-b.y);}
```

```
int Lenth(Point a) { return a.x*a.x + a.y*a.y; }
int Cross(Point a, Point b)  {return a.x*b.y- a.y*b.x; }
int Dot(Point a, Point b) { return a.x*b.x+a.y*b.y; }
int OnSeg(Point a, Point b, Point c)
{
    if(Cross(a-b, b-c) !=0 ) return 0;
    if(Dot(a-b, a-c) < 0) return 1;
    return 0;
}
Point point[N];
int dis[N][N], inv[MOD+10], vis[N], a[N][N];

void Getdis(int n)
{
    memset(dis, 0, sizeof(dis));
    int flag ;
    for(int i=0;i<n;i++)
        for(int j=i+1; j<n;j++)
            if(Lenth(point[i] - point[j]) <= R*R)
            {
                flag = 0;
                for(int k=0;k<n;k++)
                    if(i==k || j==k) continue;
                    else if(OnSeg(point[k], point[i], point[j]))
                    { flag = 1; break; }
                if(flag==0 ) dis[i][j] = dis[j][i] = 1;
            }
}

void Exgcd(int a, int b, int &x, int &y)
{
    if(b==0) { x = 1; y = 0; return ; }
    Exgcd(b, a%b, x, y);
    int t = y;
    y = x - a/b*y;
    x = t;
}

int Dfs(int u, int n)
{
    int sum = 1;
    vis[u] = 1;
    for(int i=0;i<n;i++)
    {
        if(!vis[i] && dis[u][i]==1)
            sum+= Dfs(i,n);
    }
    return sum;
}

int det(int n)
{
    for(int i=0;i<n;i++) for(int j=0;j<n;j++) a[i][j] = (a[i][j]+MOD)%MOD;

    int ans = 1;
    for(int i=0;i<n;i++)
    {
        int k=i;
        for(int j=i+1;j<n;j++)
            if(abs(a[j][i]) > abs(a[k][i])) k = j;

        if(k !=i ) for(int j= i; j< n; j++) swap(a[i][j], a[k][j]);
        if(k !=i ) ans = -ans;
        if(ans < 0) ans += MOD;
        if(a[i][i] == 0) return -1;
        ans = (ans*a[i][i]%MOD + MOD) % MOD;
        a[i][i] = (a[i][i] + MOD) % MOD;
        for(int j=i+1; j<n;j++)
            if( a[j][i] != 0 )
            {
                a[j][i] = a[j][i]*inv[a[i][i]]%MOD;
```

```
            for(int k=i+1; k<n; k++ )
                a[j][k] = ((a[j][k] - a[j][i]*a[i][k])%MOD + MOD)%MOD;
        }
    }
    return (ans%MOD + MOD)%MOD;
}
int main()
{
    for(int i=1;i<MOD; i++)
    {
        // a*x + p*y = 1, x就是a对p的逆元
        int x, y;
        Exgcd(i, MOD, x, y);
        inv[i] = (x%MOD+MOD)%MOD;
    }

    int T;
    scanf("%d", &T);
    while(T--)
    {
        int n;
        scanf("%d%d", &n,&R);
        for(int i=0;i<n;i++)
            scanf("%d%d", &point[i].x, &point[i].y);

        Getdis(n);
        memset(vis, 0, sizeof(vis));
        if( Dfs(0, n) != n) { printf("-1\n"); continue; }

        memset(a, 0, sizeof(a));
        for(int i=0;i<n;i++)
            for(int j=0;j<n;j++)
                if(dis[i][j]) { a[i][j] = -1; a[i][i]++; }
        int ans = det( n-1);
        printf("%d\n", ans);
    }
    return 0;
}
```

6.2 Java BigInteger 在 ACM 中的应用

Java 中的 BigInteger 在 ACM 中的应用

在 ACM 中的做题时，经常会遇见一些大数的问题，这是当我们用 C 或是 C++ 时就会觉得比较麻烦，就想有没有现有的现有的可以直接调用的 BigInter，那样就方便很多啦。在 java 中就有的，所以在这儿我就做一个简要的介绍吧

```

一：在java中的基本头文件（java中叫包）

import java.io.*

importjava.util.* 我们所用的输入scanner在这个包中

importjava.math.* 我们下面要用到的BigInteger就这这个包中

二：输入与输出

读入 Scanner cin=new Scanner（System.in）

While(cin.hasNext()) //相当于C语言中的！=EOF

n=cin.nextInt(); //输入一个int型整数

n=cin.nextBigInteger(); //输入一个大整数

System.out.print(n); //输出n但不换行

```

```
System.out.println(); //换行

System.out.println(n); //输出n并换行

System.out.printf( “%d\n” ,n); //类似C语言中的输出

三：定义变量

定义单个变量：

int a,b,c; //和C++ 中无区别

BigInteger a; //定义大数变量a

BigIntegerb= new BigInteger("2"); //定义大数变量 b赋值为 2;

BigDecimaln:  //定义大浮点数类 n;

定于数组：

int a[]= new int[10] //定义长度为10的数组a

BigInteger b[] =new BigInteger[100] //定义长度为100的数组a

四：表示范围

布尔型 boolean 1 true,false false

字节型 byte 8 -128-127 0

字符型 char 16  ‘\u000’ -\uffff  ‘\u0000’

短整型 short 16 -32768-32767 0

整型 int 32 -2147483648,2147483647 0

长整型 long 64 -9.22E18,9.22E18 0

浮点型 float 32 1.4E-45-3.4028E+38 0.0

双精度型 double 64 4.9E-324,1.7977E+308 0.0

BigInteger任意大的数，原则上只要你的计算机内存足够大，可以有无限位

五：常用的一些操作

A=BigInteger.ONE; //把0赋给A

B=BigInteger. valueOf (3); //把3赋给B;

A[i]=BigInteger. valueOf (10); //把10赋给A[i]

c=a.add(b) //把a与b相加并赋给c

c=a.subtract(b) //把a与b相减并赋给c

c=a.multiply(b) //把a与b相乘并赋给c

c=a.divide(b) //把a与b相除并赋给c

c=a.mod(b) // 相当于a%b

a.pow(b) //相当于a^b

a.compareTo(b):  //根据该数值是小于、等于、或大于a 返回 -1、0 或 1;

a.equals(b):  //判断两数是否相等，也可以用compareTo来代替;

a.min(b), a.max(b):  //取两个数的较小、大者;

注意以上的操作都必须是BigInteger类的。
```

```
93
94  /************************************************************
95  */
96  import java.math.*;
97  import java.util.Scanner;
98  public class Main {
99    public static void main(String[] args) {
100     Scanner cin=new Scanner(System.in);
101     BigInteger a[] = new BigInteger[300];
102     BigInteger b=new BigInteger("2");
103     a[0]=BigInteger.valueOf(1);
104     a[1]=BigInteger.valueOf(1);
105     a[2]=BigInteger.valueOf(3);
106     a[3]=BigInteger.valueOf(5);
107     int n;
108
109     for(int i=3;i<=255;i++)
110     {
111       a[i]=a[i-1].add(a[i-2].multiply(b));
112     }
113     while(cin.hasNext())
114     {
115       n=cin.nextInt();
116       System.out.println(a[n]);
117     }
118   }
119 }
120
121 /*********************************************************
122 */
123 /************************************************************************
124
125  ************************************************************************/
126 import java.io.*;
127 import java.math.BigInteger;
128 import java.util.*;
129
130 public class Main
131 {
132    public static void main(String args[])
133    {
134       Scanner cin = new Scanner(System.in);
135       int n;
136       while(cin.hasNextInt())
137       {
138          n = cin.nextInt();
139          BigInteger tmp = BigInteger.ONE;
140          for(int i = 1; i <= n; i++)
141          {
142             BigInteger I = BigInteger.valueOf( i );
143             tmp = tmp.multiply(I);
144          }
145          BigInteger sum = BigInteger.ZERO;
146          for(int i = 1; i <= n; i++)
147          {
148             BigInteger I = BigInteger.valueOf( i );
149             sum = sum.add( tmp.divide(I) );
150          }
151          sum = sum.multiply( BigInteger.valueOf(n) );
152          BigInteger ans = sum.divide( tmp );
153          sum = sum.mod(tmp);
154          BigInteger GCD = sum.gcd(tmp);
155          sum = sum.divide( GCD );
156          tmp = tmp.divide( GCD );
157
158          String Ans = ans.toString();
159          int len = Ans.length();
160
161          String b = tmp.toString();
162          int len1 = b.length();
163
```

```
            if(sum.equals(BigInteger.ZERO))
            {
                System.out.println(ans);
            }
            else
            {
                for(int i=0;i<=len;i++)
                    System.out.print("␣");
                System.out.print(sum+"\n");
                System.out.print(ans + "␣");
                for(int i = 0; i < len1; i++)
                {
                    System.out.print("-");
                }
                System.out.print("\n");
                for(int i = 0 ; i <= len ; i++)
                {
                    System.out.print("␣");
                }
                System.out.print(tmp+"\n");
            }
        }
    }
}


BigDecimal, 序列化表格
字段摘要
static BigInteger ONE
        BigInteger 的常量 1。
static BigInteger TEN
        BigInteger 的常量 10。
static BigInteger ZERO
        BigInteger 的常量 0。

构造方法摘要
BigInteger(byte[] val)
        将包含 BigInteger 的二进制补码表示形式的 byte 数组转换为 BigInteger。
BigInteger(int signum, byte[] magnitude)
        将 BigInteger 的符号-数量表示形式转换为 BigInteger。
BigInteger(int bitLength, int certainty, Random rnd)
        构造一个随机生成的正 BigInteger，它可能是一个具有指定 bitLength 的素数。
BigInteger(int numBits, Random rnd)
        构造一个随机生成的 BigInteger，它是在 0 到 (2numBits - 1)（包括）范围内均匀分布的值。
BigInteger(String val)
        将 BigInteger 的十进制字符串表示形式转换为 BigInteger。
BigInteger(String val, int radix)
        将指定基数的 BigInteger 的字符串表示形式转换为 BigInteger。

方法摘要
 BigInteger abs()
        返回其值是此 BigInteger 的绝对值的 BigInteger。
 BigInteger add(BigInteger val)
        返回其值为 (this + val) 的 BigInteger。
 BigInteger and(BigInteger val)
        返回其值为 (this & val) 的 BigInteger。
 BigInteger andNot(BigInteger val)
        返回其值为 (this & ~val) 的 BigInteger。
 int bitCount()
        返回此 BigInteger 的二进制补码表示形式中与符号不同的位的数量。
 int bitLength()
        返回此 BigInteger 的最小的二进制补码表示形式的位数，不包括 符号位。
 BigInteger clearBit(int n)
        返回其值与清除了指定位的此 BigInteger 等效的 BigInteger。
 int compareTo(BigInteger val)
        将此 BigInteger 与指定的 BigInteger 进行比较。
 BigInteger divide(BigInteger val)
        返回其值为 (this / val) 的 BigInteger。
 BigInteger[] divideAndRemainder(BigInteger val)
        返回包含 (this / val) 后跟 (this % val) 的两个 BigInteger 的数组。
 double doubleValue()
```

```
        将此 BigInteger 转换为 double。
 boolean equals(Object x)
        比较此 BigInteger 与指定的 Object 的相等性。
 BigInteger flipBit(int n)
        返回其值与对此 BigInteger 进行指定位翻转后的值等效的 BigInteger。
 float floatValue()
        将此 BigInteger 转换为 float。
 BigInteger gcd(BigInteger val)
        返回一个 BigInteger，其值是 abs(this) 和 abs(val) 的最大公约数。
 int getLowestSetBit()
        返回此 BigInteger 最右端（最低位）1 比特的索引（即从此字节的右端开始到本字节中最右端 1 比特之间的 0
            比特的位数）。
 int hashCode()
        返回此 BigInteger 的哈希码。
 int intValue()
        将此 BigInteger 转换为 int。
 boolean isProbablePrime(int certainty)
        如果此 BigInteger 可能为素数，则返回 true，如果它一定为合数，则返回 false。
 long longValue()
        将此 BigInteger 转换为 long。
 BigInteger max(BigInteger val)
        返回此 BigInteger 和 val 的最大值。
 BigInteger min(BigInteger val)
        返回此 BigInteger 和 val 的最小值。
 BigInteger mod(BigInteger m)
        返回其值为 (this mod m) 的 BigInteger。
 BigInteger modInverse(BigInteger m)
        返回其值为 (this-1 mod m) 的 BigInteger。
 BigInteger modPow(BigInteger exponent, BigInteger m)
        返回其值为 (thisexponent mod m) 的 BigInteger。
 BigInteger multiply(BigInteger val)
        返回其值为 (this * val) 的 BigInteger。
 BigInteger negate()
        返回其值是 (-this) 的 BigInteger。
 BigInteger nextProbablePrime()
        返回大于此 BigInteger 的可能为素数的第一个整数。
 BigInteger not()
        返回其值为 (~this) 的 BigInteger。
 BigInteger or(BigInteger val)
        返回其值为 (this | val) 的 BigInteger。
 BigInteger pow(int exponent)
        返回其值为 (thisexponent) 的 BigInteger。
static BigInteger probablePrime(int bitLength, Random rnd)
        返回有可能是素数的、具有指定长度的正 BigInteger。
 BigInteger remainder(BigInteger val)
        返回其值为 (this % val) 的 BigInteger。
 BigInteger setBit(int n)
        返回其值与设置了指定位的此 BigInteger 等效的 BigInteger。
 BigInteger shiftLeft(int n)
        返回其值为 (this << n) 的 BigInteger。
 BigInteger shiftRight(int n)
        返回其值为 (this >> n) 的 BigInteger。
 int signum()
        返回此 BigInteger 的正负号函数。
 BigInteger subtract(BigInteger val)
        返回其值为 (this - val) 的 BigInteger。
 boolean testBit(int n)
        当且仅当设置了指定的位时，返回 true。
 byte[] toByteArray()
        返回一个 byte 数组，该数组包含此 BigInteger 的二进制补码表示形式。
 String toString()
        返回此 BigInteger 的十进制字符串表示形式。
 String toString(int radix)
        返回此 BigInteger 的给定基数的字符串表示形式。
static BigInteger valueOf(long val)
        返回其值等于指定 long 的值的 BigInteger。
 BigInteger xor(BigInteger val)
        返回其值为 (this ^ val) 的 BigInteger。
```