1 图论 3
1.1 术语 3
1.2 独立集、覆盖集、支配集之间关系 3
1.3 DFS 4
1.3.1 割顶 6
1.3.2 桥 6
1.3.3 强连通分量 7
1.4 最小点基 7
1.5 拓扑排序 7
1.6 欧拉路 8
1.7 哈密顿路（正确？） 9
1.8 Bellman-ford 9
1.9 差分约束系统（用 bellman-ford 解） 10
1.10 dag 最短路径 10
1.11 二分图匹配 11
1.11.1 匈牙利算法 11
1.11.2 KM 算法 12
1.12 网络流 15
1.12.1 最大流 15
1.12.2 上下界的网络的最大流 17
1.12.3 上下界的网络的最小流 17
1.12.4 最小费用最大流 18
1.12.5 上下界的网络的最小费用最小流 21
2 数论 21
2.1 最大公约数 gcd 21
2.2 最小公倍数 lcm 22
2.3 快速幂取模 B^LmodP(O(logb)) 22
2.4 Fermat 小定理 22
2.5 Rabin-Miller 伪素数测试 22
2.6 Pollard-rho 22
2.7 扩展欧几里德算法 extended-gcd 24
2.8 欧拉定理 24
2.9 线性同余方程 $ax \equiv b \pmod{n}$ 24
2.10 中国剩余定理 25
2.11 Discrete Logging($B^L == N \pmod{P}$) 26
2.12 N!最后一个不为 0 的数字 27
2.13 2^14 以内的素数 27
3 数据结构 31
3.1 堆（最小堆） 31
3.1.1 删除最小值元素： 31
3.1.2 插入元素和向上调整： 32
3.1.3 堆的建立 32
3.2 并查集 32
3.3 树状数组 34

3.3.1 LOWBIT 34
3.3.2 修改 a[p] 34
3.3.3 前缀和 A[1]+…+A[p] 34
3.3.4 一个二维树状数组的程序 35
3.4 线段树 36
3.5 字符串 38
3.5.1 字符串哈希 38
3.5.2 KMP 算法 40
4 计算几何 42
4.1 直线交点 42
4.2 判断线段相交 42
4.3 三点外接圆圆心 43
4.4 判断点在多边形内 43
4.5 两圆交面积 44
4.6 最小包围圆 44
4.7 经纬度坐标 46
4.8 凸包 47
5 Problem 49
5.1 RMQ-LCA 49
5.1.1 Range Minimum Query(RMQ) 49
5.1.2 Lowest Common Ancestor (LCA) 53
5.1.3 Reduction from LCA to RMQ 56
5.1.4 From RMQ to LCA 58
5.1.5 An<O(N), O(1)> algorithm for the restricted RMQ 61
5.1.6 An AC programme 62
5.2 最长公共子序列 LCS 65
5.3 最长上升子序列/最长不下降子序列(LIS) 66
5.3.1 O(n^2) 66
5.3.2 O(nlogn) 67
5.4 Joseph 问题 68
5.5 0/1 背包问题 69
6 组合数学相关 70
6.1 The Number of the Same BST 70
6.2 排列生成 72
6.3 逆序 73
6.3.1 归并排序求逆序 73
7 数值分析 73
7.1 二分法 73
7.2 迭代法(x=f(x)) 74
7.3 牛顿迭代 75
7.4 数值积分 75
7.5 高斯消元 76
8 其它 78

1 图论

1.1 术语

支配集：D 是图 G 的一个顶点子集，对于 G 的任一顶点 U，要么 U 是 D 集合的一个顶点元素，要么与 D 中的一个顶点相邻，那么 D 称为图 G 的一个支配集。

极小支配集：若在 D 集中去除任何元素后 D 不再是支配集，则支配集 D 是极小支配集。

最小支配集：G 的所有支配集中顶点个数最少的支配集。

点基：设 G＝(V，E)是一个有向图，B 是若干个顶点组成的 V 的子集。如果对于任意的 Vj∈V，都存在一个 Vi∈B，使得 Vi 是 Vj 的前代，则称 B 是一个点基。

最高强连通分支：设[Si]是有向图 G 的一个强连通分支，如果在 G 中，不存在终点属于[Si]而起点不熟于[Si]的弧，就称[Si]为最高强连通分支。

最小路径覆盖是指在有向无环图 G 里面，覆盖所有点的最少不相交简单边的集合，即每个点只属于一条边。
最小路径覆盖数＝G 的定顶点数－最小路径覆盖中的边数。

覆盖集：K 是 G 图的一个顶点子集且 G 的每一边至少有一个端点属于 K，则称 K 是 G 的一个覆盖。

独立集：I 是 G 的一个顶点子集，I 中任两个顶点不相邻，则称 I 是图 G 的一个独立集。

极大独立集：若独立集 I 中增加任一个除 I 集外的顶点后 I 不是独立集。

完全二分图

最佳匹配：权和最大的匹配。

完全匹配（完备匹配）：如果一个匹配中，图中的每个顶点都和图中某条边相关联。

1.2 独立集、覆盖集、支配集之间关系

(1)一个独立集是极大独立集，当且仅当它是一个支配集；
(2)I 是独立集，当且仅当 G 中除 I 集外的所有顶点是一个覆盖集；
I 是极大(最大)独立集,当且仅当 G 中除 I 集外的所有顶点是一个极小(最小)覆盖集，即

α(G)（覆盖数）十β(G)（覆盖数）= G 的顶点数
(3)极大独立集必为极小支配集，但是极小支配集未必是极大独立集。

求所有极小支配集的公式：

$$\phi(V1，V2，\cdots，Vm)=\Pi(Vi+\Sigma U)$$

一个顶点同与它相邻的所有顶点进行加法运算组成一个因子项，几个因子项再连乘。连乘过程中展开成积之和形式。每一积项给出一个极小支配集，所有积项给
出了一切极小支配集。其中最小者为最小支配集。

极小覆盖集的计算方法：

$$\phi(V1，V2，\cdots，Vm)=\Pi(Vi+\Pi U)$$

某顶点的所有相邻顶点进行积运算后再与该顶点进行和运算，组成一个因子项。几个因子项连乘，并根据逻辑运算定律展开成积之和形式。每一积项给出一个极小覆盖集，所有积项给出了一切极小覆盖集，其中最小者为最小覆盖集。

极大独立集的计算方法：

V-极小覆盖集

1.3 DFS

图的 DFS 信息构建
g 矩阵: g[i][j] -> 0: 无边
1: 可重复访问边
-1: 非可重复访问边
说明:以为在无向图中 u->v 访问之后就不能再从 v->u 访问了故{u, v}访问了之后{v, u}要置-1。如果是有向图 则没有这个规则
gc 矩阵:gc[i][j]-> 0 : 无边
1 : 树枝边
2 : 反向边
3 : 正向边
4 : 交叉边
d 数组: 顶点的开始访问时间表
f 数组: 顶点的结束访问时间表
c 数组: 顶点颜色表 0 白色 -1 灰色 1 黑色
p 数组: 顶点的前驱表
l 数组: 顶点的 L 值(最顶层的祖先层数)
b 数组: 顶点的度数表

关于标号函数 LOW()
LOW(U)代表的是与 U 以及 U 的子孙直接相连的结点的最高辈分（深度）
d[U] U 首次被访问时
LOW[U] = min(LOW[U], d[W]) 访问边{U,W}
min(LOW[U], LOW[S]) U 的儿子 S 的关联边全部被访问时

```
const int maxn = 100;
```

```
int n, g[maxn][maxn], gc[maxn][maxn];
int d[maxn], f[maxn], l[maxn], p[maxn], c[maxn], b[maxn];
int time;

void dfs_visit(int u)//递归搜索以 U 为根的深度优先树
{
    int v;
    c[u] = -1;      //置顶点为灰色//去掉这句之后适用于有向图(后面设置不//可访问亦同)
    time++; d[u] = time, l[u] = time;
    for(v = 1; v<=n; v++)
        if(g[u][v] > 0)
            if(c[v] == 0)              //如果 v 是白色节点
            {
                g[v][u] = -1;           //不可再访问
                gc[u][v] = 1;           //树枝边
                b[u]++;                   //度数
                p[v] = u;                 //记录父亲节点
                dist_visit(v);          //递归搜索以 v 为根的深度优先树
                if(l[v] < l[u])         //v 是 u 的后代
                l[u] = l[v];       //u 的儿子 v 的关联边搜索完后计算父亲的//low 值
                g[v][u] = 1;            //恢复可访问标志
            }
            else
            {
                if(c[v] < 0)       //若顶点为灰色
                {
                    if(l[v] < l[u])   //u 与 v 相连
                        l[u] = l[v];
                    gc[u][v] = 2;      //反向边
                }
                else                   //黑色
                {
                    if(d[v] > d[u]) gc[u][v] = 3;            //正向边
                    else  gc[u][v] = 4;              //交叉边
                }
            }
    c[u] = 1;                  //DFS 完毕  置黑色吧
    time++; f[u] = time;
}

void dfs()
{
    int u;
    memset(gc, 0, sizeof(gc));
```

```
    memset(c, 0, sizeof(c));
    memset(b, 0, sizeof(b));
    time = 0;
    for(u = 1; u <= n; u++)
        if(c[u] == 0)
        {
            p[u] = 0;
            dfs_visit(u);
        }
}

void DFS(int k, int father, int deep)
{
    int i, tot;
    C[k] = -1; //灰色
    D[k] = deep; //记录深度
    Ancestor[k] = deep, tot = 0;
    for(i = 1; i<=n; ++i)
    {
        if((节点 i 和 k 相连) && (i != father) && (C[i] == -1))
            Ancestor[k] = Min(Ancestor[k], D[i]);
        if((节点 i 与 k 相连) && (C[i] == 0))
        {
            DFS(i, k, deep + 1);
            tot++, Ancestor[k] = Min(Ancestor[k], Ancestor[i]);
            if((k == Root) && (tot > 1) || ( (k != Root) && (Ancestor[i] >= D[k]) ))
                Cut[k] = true;
            if((Ancestor[i] > D[k]))  Brige[k][i] = true;
        }
    }
    C[k] = 1; //黑色
    time++, A[i] = time;
}
```

1.3.1 割顶

判定规则 1: 如果 root 节点有多于一个 1 子节点 则 root 是割点
判定规则 2: 如果一个节点 u 有某一个子节点 v 不含到 u 的祖先节点的后向边，则 u 为割点
即对于 u 的子节点 v, u 是割点的条：k == Root) && (tot > 1) || ((k != Root) && (Ancestor[i] >= D[k]))

1.3.2 桥

不属于任何简单回路的边，条件：(Ancestor[i] > D[k])

1.3.3 强连通分量

强连通分量：
(1) 对图进行 DFS，计下时间戳 Ai；
(2) 改变 G 的每一条边的方向，构造出新图 Gr；
(3) 按 Ai 由大到小对 Gr 进行 DFS，构建森林，每棵树构成强连通分量。

1.4 最小点基

(1)找出图 G＝(V，A)的所有强连通分支；
(2)从强连通分支中找出所有最高的强连通分支；
(3)从每一个最高的强连通分支中取一个权最小的顶点，组成的顶点集 B 就是一个 G 的权最小的点基。

1.5 拓扑排序

```
int topol[501], kk;  //存储拓扑排序的结果
int adj[k][j]; //邻接矩阵
int topo(int count[],int vet)
{
    int i,j,k;
    int top=-1;
    for(i=1;i<=vet;i++)
        if(0==count[i])
        {
            count[i]=top;
            top=i;
         }
    for(i=1;i<=vet;i++)
    {
        if(-1==top) return 0; //有环
        k=top;
        top=count[top];
        topol[++k] = k;
        for(j=1;j<=vet;j++)
        {
            if(adj[k][j])
            {
                count[j]--;
                if(0==count[j])
                {
                    count[j]=top;
                    top=j;
```

```
                }
            }
        }
    }
    return 1;
}
```

1.6 欧拉路

计算图的欧拉路径数目：
设 degree[i]表示结点 i 的入边数目减去出边数目，欧拉路径数＝(Σ|degree[i]|)/2+图中没有奇点的连通块个数。
算法步骤如下：
(1):任取一起始节点 VS;
(2):设路 EE：=[[v1,v2],[v2,v3],....,[vn,vs]]选出，则从 G\EE 中选边 ee，使 ee 与 VS 相连，除非没有其它选择，EE 不应为 G\EE 的桥（即去 G\EE 应保持连通性）;
(3):重复步骤(2)，直至不能进行下去为止。

```
int graph[50][50];
int degree[50];
vector<int> path;          //存储欧拉路
int n;

void FindCircuit(int u)
{
    if(degree[u] == 0)
    {
        path.push_back(u);
        return;
    }
    int v;
    for(v = 1; v <= n; ++v)
        if(graph[u][v])
        {
            --graph[u][v];
            --graph[v][u];
            --degree[u];
            --degree[v];
            FindCircuit(v);
        }
    path.push_back(u);
}
```

1.7 哈密顿路（正确？）

(1) 从任意一个顶点开始，在它的任意一端邻接一个顶点，构造一条越来越长的链到不能在加长为止。设链为：$r: y_1 - y_2 - \quad - y_m$

(2)检查 y1 和 ym 是否邻接：

A）如果 y1 和 ym 不邻接，则转到 3)；否则转到 B)；

B）如果 m＝n，就说明这条链已经把所有的点历遍了，则停止构造圈并输出哈密顿圈 $r: y_1 - y_2 - \quad - y_m$。否则 y1 和 ym 不邻接且 m<n，转到 C)；

C）找出一个不在 r 上的顶点 z 和在 r 上的顶点 yk，满足 z 和 yk 是邻接的，将 r 用下面长度为 m＋1 的链来代替：$z - y_k - \quad - y_m - y_1 - \quad - y_{k-1}$ 再转到 2)；

(3) 找到一个顶点 yk（1<k<m)，满足 y1 和 yk 邻接，且 y_{k-1}和 ym 也是邻接的，将 r 用下面的链来替代：$y_1 - \quad - y_{k-1} - y_m - \quad - y_k$。

1.8 Bellman-ford

```
void relax(int u, int v)
{
    if(d[v] > d[u] + e[u][v])
        d[v] = d[u] + e[u][v];
}

void bellman_ford()
{
    int i, j, k;
    for(i = 2; i <= n; i++)
        d[i] = 10000000000;
    for(k = 1; k < n; k++)
        for(i = 1; i <= n; i++)
            for(j = 1; j <= n; j++)
                    relax(i, j);      //对每条边松弛
    /*for(i = 1; i <= n; i++)
        for(j = 1; j <= n; j++)
            if(d[j] > d[i] + e[i][j])
                return false;
    return true;*/
}
```

1.9 差分约束系统（用 bellman-ford 解）

寻找满足：

$$Xj-Xi \leqslant bk(1 \leqslant i,j \leqslant n,\ 1 \leqslant k \leqslant m)$$

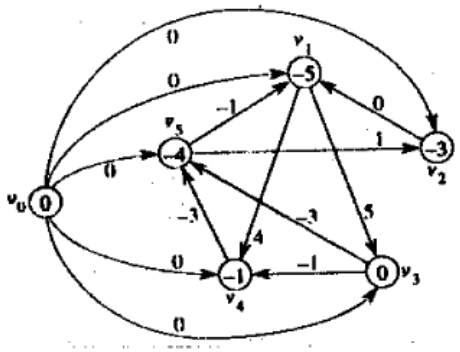
的可行解 X。

更形式地，已知某差分约束系统 $Ax \leqslant b$，相应的约束图为一有向加权图 $G=(V, E)$，其中：

$V=\{v_0, v_1, \cdots, v_n\}$

$E=\{(v_i, v_j): x_j - x_i \leqslant b_k$ 是一约束条件$\}$

$\bigcup\{(v_0, v_1), (v_0, v_2), (v_0, v_3), \cdots, (v_0, v_n)\}$



1.10 dag 最短路径

```
int n;
int graph[501][501];
int top[501], k;       //存储拓扑排序的结果(倒转数组)
int d[501];
int flag[501];

void relax(int p, int q)
{
      if(d[q] < d[p] + graph[p][q])
            d[q] = d[p] + graph[p][q];
}

void dfs(int t)
{
      int i;
      flag[t] = 1;
      for(i = 1; i <= n; i++)
            if(!flag[i] && graph[t][i])
            {
                  dfs(i);
            }
```

```
    top[++k] = t;
}
void work()
{
    k =0;
    dfs(s);
    d[s] = 0;
    for(i = k; i > 0; i--)
        for(j = 1; j <= n; j++)
            if(graph[top[i]][j])  relax(top[i], j);
}
```

1.11 二分图匹配

一个关于二分图的性质：
最大匹配数＋最大独立集＝X＋Y

1.11.1 匈牙利算法

算法轮廓：
(1) 置 M 为空；
(2) 找出一条增广路径 P，通过取反操作获得更大的匹配 M’代替 M；
(3) 重复(2)操作直到找不出增广路径为止。

```
int graph[m][n];// 二分图的邻接矩阵
int check[m];    //记录点是否被扫描过
int match[m];    //存储了匹配的方案，初始化为-1

int Dfs(int p)
{
    int i,t;
    for(i = 1; i <= m; i++)
    if(graph[i][p] && !check[i])
    {
        check[i]=1;                //不会检查第二次
        t=match[i];
        match[i]=p;
        if(t==-1 || Dfs(t))    return 1;
        match[i]=t;
    }
    return 0;
}

int Match()
{
```

```
    int i, res=0;
    for(i = 1; i <= n;i++)
    {
        memset(check,0,sizeof(check));
        if(Dfs(i)) res++;
    }
    return res;
}
```

1.11.2 KM 算法

Kuhn－Munkras 算法流程：
(1) 初始化可行顶标的值；
(2) 用匈牙利算法寻找完备匹配；
(3) 若未找到完备匹配则修改可行顶标的值；
(4) 重复(2)(3)直到找到相等子图的完备匹配为止。
Version 1：O(n^4)

```
const int size = 160;
const int INF = 100000000;   //   相对无穷大
bool map[size][size];        // 二分图的相等子图, map[i][j] = true  代表 Xi 与 Yj 有边
bool xckd[size], yckd[size];   // 标记在一次 DFS 中，Xi 与 Yi 是否在交错树上
int match[size]; //保存匹配信息，其中 i 为 Y 中的顶点标号，match[i]为 X 中顶点标号

//寻找是否有以 Xp 为起点的增广路径，返回值为是否含有增广路
bool DFS(int p, const int n)
{
    int i;
    for(i = 0; i < n; i++)
    {
        if(!yckd[i] && map[p][i])
        {
            yckd[i] = true;
            int t = match[i];
            match[i] = p;
            if(t == -1 || DFS(t, n))     return true;
            match[i] = t;
            if(t != -1) xckd[t] = true;
        }
    }
    return false;
}

void KM_Perfect_Match(const int n, const int edge[][size])
{
```

```
int i, j;
int lx[size], ly[size];     //   KM 算法中 Xi 与 Yi 的标号
for(i = 0; i < n; i++)
{
    lx[i] = -INF;
    ly[i] = 0;
    for(j = 0; j < n; j++)lx[i] = max(lx[i], edge[i][j]);
}
bool perfect = false;
while(!perfect)
{
    //初始化邻接矩阵
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n; j++)
        {
            if(lx[i]+ly[j] == edge[i][j]) map[i][j] = true;
            else map[i][j] = false;
        }
    }
    // 匹配过程
    int live = 0;
    memset(match, -1, sizeof(match));
    for(i = 0; i < n; i++)
    {
        memset(xckd, false, sizeof(xckd));
        memset(yckd, false, sizeof(yckd));
        if(DFS(i, n)) live++;
        else
        {
            xckd[i] = true;
            break;
        }
    }
    if(live == n) perfect = true;
    else
    {
        // 修改标号过程
        int ex = INF;
        for(i = 0; i < n; i++)
        {
            for(j = 0; xckd[i] && j < n; j++)
            {
                if(!yckd[j]) ex = min(ex, lx[i]+ly[j]-edge[i][j]);
```

```
                    }
                }
                for(i = 0; i < n; i++)
                {
                    if(xckd[i]) lx[i] -= ex;
                    if(yckd[i]) ly[i] += ex;
                }
            }
        }
        int cost = 0;
        for(i = 0; i < n; i++)cost += edge[match[i]][i];
        // cost 为最大匹配的总和, match[]中保存匹配信息
}
```

Version 2：O(n^3)

```
const int INF = 0x3fffffff;
int n;
int w[20][20];   //二分的权值
int lx[20],ly[20];
int fx[20],fy[20];   //标记在一次 DFS 中，Xi 与 Yi 是否在交错树上
int matx[20],maty[20]; //保存匹配信息

int path(int u)
{
    int v;
    fx[u] = 1;
    for (v=0; v<n; v++)
        if ((lx[u]+ly[v]==w[u][v])&&(fy[v]<0)) {
            fy[v] = 1;
            if ((maty[v]<0) || (path(maty[v]))) {
                matx[u] = v;
                maty[v] = u;
                return 1;
            }
        }
    return 0;
}

int KM()
{
    int ret = 0,i,j,k,p;
    memset(ly,0,sizeof(ly));
    for (i=0; i<n; i++)
    {
```

```
            lx[i] = -INF;
            for (j=0; j<n; j++)   if (w[i][j] > lx[i])    lx[i] = w[i][j];
        }
        memset(matx,-1,sizeof(matx));
        memset(maty,-1,sizeof(maty));
        for (i=0; i<n; i++) {
            memset(fx,-1,sizeof(fx));
            memset(fy,-1,sizeof(fy));
            if (!path(i))
            {
                i--;
                p = INF;
                for (k=0; k<n; k++)
                    if (fx[k] > 0)
                        for (j=0; j<n; j++)
                            if ((fy[j]<0) && (lx[k]+ly[j]-w[k][j]<p))
                                p = lx[k]+ly[j]-w[k][j];
                for (j=0; j<n; j++) ly[j] += (fy[j]<0 ? 0:p);
                for (k=0; k<n; k++) lx[k] -= (fx[k]<0 ? 0:p);
            }
        }
        for (i=0; i<n; i++) ret += w[maty[i]][i];
        return ret;
}
```

1.12 网络流

1.12.1 最大流

寻求最大流的标号法(增广路算法：Ford,Fulkerson)

从一个可行流(一般取零流)开始，不断进行以下的标号过程与调整过程，直到找不到关于 f 的可增广路径为止。

(1)标号过程

在这个过程中，网络中的点分为已标号点和未标号点，已标号点又分为已检查和未检查两种。每个标号点的标号信息表示两个 部分：第一标号表明它的标号从哪一点得到的，以便从 vt 开始反向追踪找出也增广路径；第二标号是为了表示该顶点是否已检查过。

标号开始时，给 vs 标上(s，0)，这时 vs 是标号但未检查的点，其余都是未标号的点，记为(0，0)。

取一个标号而未检查的点 vi，对于一切未标号的点 vj：

A．对于弧(vi，vj)，若 fij<cij，则给 vj 标号(vi，0)，这时，vj 点成为标号而未检查的点。

B．对于弧(vi，vj)，若 fji>0，则给 vj 标号(-vi，0)，这时，vj 点成为标号而未检查的点。

于是 vi 成为标号且已检查的点，将它的第二个标号记为 1。重复上述步骤，一旦 vt 被标上号，表明得到一条从 vi 到 vt 的增广路径 p，转入调整过程。

若所有标号都已检查过去，而标号过程进行不下去时，则算法结束，这时的可行流就是最大流。

(2)调整过程

从 vt 点开始，通过每个点的第一个标号，反向追踪，可找出增广路径 P。例如设 vt 的第一标号为 vk(或-vk)，则弧(vk，vt)(或 相应地(vt，vk))是 p 上弧。接下来检查 vk 的第一标号，若为 vi(或-vi)，则找到(vi，vk)(或相应地(vk，vi))。再检查 vi 的第一 标号，依此类推，直到 vs 为止。这时整个增广路径就找到了。在上述找增广路径的同时计算 Q：

Q=min{min(cij-fij)，minf*ij}

对流 f 进行如下的修改：

f'ij = fij+Q (vi，vj)∈ P 的前向弧的集合

f'ij = fij-Q (vi，vj)∈ P 的后向弧的集合

f'ij = f*ij (vi，vj)不属于 P 的集合

接着，清除所有标号，对新的可行流 f'，重新进入标号过程。

```
const int maxn = 1001;
const int inf = 0x7fffffff;
int g[maxn][maxn]; //网络
int f[maxn][maxn]; //残留网络
int v[maxn], pre[maxn]; //用于标号
int n, s, t; //节点数,源,汇

inline int min(int x, int y) { return x<y?x:y; }

//更新残留网络
void update()
{
    int p,q;
    int minval=v[q=t];
    while (q!=s)
    {
        p=pre[q];
        if (p>=n) p-=n, f[q][p]-=minval;
        else f[p][q]+=minval;
        q=p;
    }
}

int max_flow()
{
    int queue[maxn], p, q, k, i;
    while (true)
    {
        memset(pre, 0xff, sizeof(pre[0])*n);
        pre[s] = s;
        v[s] = inf;
```

```
            queue[p=q=0]=s;
            //寻找增广路
            while (p<=q)
            {
                k=queue[p++];
                for (i=0;i<n;i++)
                    if (pre[i]<0)
                    {
                        if (g[k][i] > f[k][i])
                            v[i]=min(g[k][i]-f[k][i], v[k]), pre[i]=k, queue[++q]=i;
                        if (f[i][k] > 0)
                            v[i]=min(f[i][k], v[k]), pre[i]=k+n, queue[++q]=i;
                    }
                if (pre[t]>=0) break;
            }
            if (pre[t]<0) break;
            update();
        }
        for (k=i=0;i<n;i++) k+=f[s][i];
        return k;
}
```

1.12.2 上下界的网络的最大流

转换成每条孤仅对应一个容量 C(e)＞0 附加网络：
(1)新增加两个顶点 s’和 t’,s’称为附加源，t’称为附加汇；
(2)对原网络 N 的每个顶点 U，加一条新弧 e＝Ut’，这条弧的容量为所有以 U 为尾的弧的容量下限之和，即 C’(e)＝ΣB(e)；
(3)对原网络 N 的每个顶点 U，加一条新弧 e＝s’U，这条弧的容量为所有以 U 为头的弧的容量下限之和，即 C’(e)＝ΣB(e)；
(4)原网络 N 的每条弧在 C’仍保留，弧容量 C’(e)修正为 C(e)一 B(e)；
(5)再添两条新弧 e=st, e’＝ts，e 的容量 C’(e)＝inf，e’的容量 C’(e’)＝inf。
求附加网 N’的最大流，若结果能使流出 s’的一切弧皆满载(即 s’出发的所有弧 e，C’(e)＝F’(e))，则网络有可行流 F(e)＝F’(e)十 B(e)。否则原网无可行流。
若 N’网的最大流满足上述条件，将可行流 F 放大，最终求出 N 网的最大流。

1.12.3 上下界的网络的最小流

(1) 按上述的办法构造附加网 N’；
(2) 求 N’网的最大流;
(3) 流出附加源 s’的弧都满载？
(4) 否，N 网无最小流；
(5) 求出 N 网和可行流 F(e)＝F’(e)十 B(e)，倒向求解；
以 t 为源点 s 为汇点，最终求出的最大流即为从 s 到 t 的最小流。

1.12.4 最小费用最大流

构造一个赋权有向图 b(f)，它的顶点是原网络 N 中的顶点，而把 N 中每一条弧(vi，vj)变成两个相反方向的弧(vi，vj)和(vj，vi)。定义 w(f)中的弧的权如下：

如果($f_{ij}<c_{ij}$)，则 $b_{ij}=w_{ij}$ ；如果($f_{ij}=c_{ij}$)，则 $b_{ij}=+oo$

如果($f_{ij}>0$)，则 $b_{ji}=-w_{ij}$ ；如果($f_{ij}=c_{ij}$)，则 $b_{ji}=+oo$

于是在网络 N 中找关于 f 的最小费用增广路径就等价于在赋权有向图 b(f)中，寻求从 vs 到 vt 的最短路径。求最短路有三种经典算法，它们分别是 Dijkstra 算法、Floyd 算法和迭代算法(ford 算法）。由于在本问题中，赋权有向图 b(f)中存在负权，故我们只能用后两种方法求最短路，其中对于本问题最高效的算法是迭代算法。为了程序的实现方便，我们只要对原网络做适当的调整。将原网络中的每条弧(vi，vj)变成两条相反的弧：

前向弧(vi,vj)，其容量 c_{ij} 和费用 w_{ij} 不变，流量为 f_{ij}；

后向弧(vj,vi)，其容量 0 和费用-w_{ij}，流量为-f_{ij}。

事实上，对于原网络的数据结构中，这些单元已存在，在用标号法求最大流时是闲置的。这样我们就不必产生关于流 f 的有向图 b(f)。同时对判断(vi,vj)的流量可否改时，可以统一为判断是否“$f_{ij}<c_{ij}$”。因为对于后向弧(vj，vi)，若 fij>0，则 $f_{ji}=-f_{ij}<0=c_{ji}$。

```
#define Q (adj[w][v].cap<0? -adj[w][v].flow : adj[w][v].cap-adj[w][v].flow)
#define maxV 100   //dfs 增广路径-最大流
#define maxWT 99999
struct link {int flow;int cap;}adj[maxV][maxV];
int st[maxV];      int vt[maxV];          int V;        int ft;
int flowdfs(int w,int t){
    int v,d;
    vt[w]=1;
    if(w==t)
    {    ft=1;return maxWT;}
    for(v=0;v<V;v++)
        if(Q>0 && vt[v]==0)
        {
            d=flowdfs(v,t);
            st[v]=w;
            if(ft==1) return (Q>d?d:Q);
        }
    return 0;}
int main()
{
    int n,a,b,c,i,j,d;
    int s,t,x,flow=0;
    scanf("%d%d",&V,&n);
    for(i=0;i<V;i++)
        for(j=0;j<V;j++)
            adj[i][j].flow=adj[i][j].cap=0;
    for(i=1;i<=n;i++) {
        scanf("%d%d%d",&a,&b,&c);
```

```
            adj[a][b].cap=c;
            adj[b][a].cap=-c;
        }
        scanf("%d%d",&s,&t);
        memset(vt,0,sizeof(vt));
        ft=0;
        while((d=flowdfs(s,t))!=0) {
            flow+=d;
            for(x=t;x!=s;x=st[x])
            {
                adj[st[x]][x].flow += d;
                adj[x][st[x]].flow -= d;
            }
            memset(vt,0,sizeof(vt));ft=0;
        }
        printf("maxflow = %d\n",flow);
        return 0;
}
*********************************************************************************
#define Q (adj[w][v].cap<0?-adj[w][v].flow:adj[w][v].cap-adj[w][v].flow)
#define maxV 105     //最小费用最大流
#define maxWT 99999
struct link
{
        int flow, cost, cap;
}adj[maxV][maxV];
int V;
int wt[maxV]; //记录最短距离
int st[maxV];  //记录最短路

int maxflow(int s,int t)
{
        bool ft;
        int w, v, d;
        for(v=0;v<V;v++)  wt[v]=maxWT;
        wt[s]=0;
        do
        {
            ft=1;
            for(w=0;w<V-1;w++)
                if(wt[w]<maxWT)
                    for(v=1;v<V;v++)
                        if(Q > 0&& adj[w][v].cap!=0)
                            if(wt[w] + adj[w][v].cost < wt[v])
```

```
                    {
                        wt[v]=wt[w]+adj[w][v].cost;
                        st[v]=w;
                        ft=0;
                    }
        }while(!ft);
        if(wt[t]<maxWT)
        {
            for(v=t,d=maxWT;v!=s;v=st[v])
            {
                w=st[v];
                d=(Q > d ? d : Q);
            }
            return d;
        }
        else        return 0;
}

int netcost()
{
        int total=0,i,j;
        for(i=0;i<V;i++)
            for(j=0;j<V;j++)
                if(adj[i][j].cap>0)    total+=adj[i][j].flow*adj[i][j].cost;
        return total;
}

int main()
{
        int n,a,b,Cap,Cost,i,j,d;
        int s,t,x,flow=0,cost;
        scanf("%d%d",&V,&n);
        for(i=0;i<V;i++)
            for(j=0;j<V;j++)
                adj[i][j].flow=adj[i][j].cap=adj[i][j].cost=0;
        for(i=1;i<=n;i++)
        {
            scanf("%d%d%d%d",&a,&b,&Cap,&Cost);
            adj[a][b].cap=Cap;
            adj[b][a].cap=-Cap;
            adj[a][b].cost=Cost;
            adj[b][a].cost=-Cost;
        }
        scanf("%d%d",&s,&t);
```

```
    while((d=maxflow(s,t))!=0)
    {
        flow+=d;
        for(x=t;x!=s;x=st[x])
        {
            adj[st[x]][x].flow += d;
            adj[x][st[x]].flow -= d;
        }
    }
    printf("maxflow = %d\n",flow);
    cost=netcost();
    printf("mincost = %d\n",cost);
    return 0;
}
```

1.12.5 上下界的网络的最小费用最小流

(1) 同上下界的网络的最小流;

(2) 为了寻求关于 F 的最小费用，我们构造一个赋权有向图 W(F)，它的顶点是原 N 网的顶点，而把 N 中的每一条弧(Vi，Vj)变成两条相反方向的弧(Vi，Vj)和(Vj，Vi)，定义 W(F)中的弧的权为：

Wij = -Aij(Fij>Bij),Wij = 0(Fij<=Bij && j == i)

在 W(F)中寻求一条从 Vs 到 Vt 的最短路;

(3) 若存在最短路，则在原 N 网中得到相应的可改进路 P，这条可改进路的前向弧 P 十为最短路上 F 值大于 B 值的弧的集合，后向弧 P 一为最短路上 F 值小于或等于 B 值的弧的集合。

在可改进路 P 上对 F 进行调整，调整量为：

Q＝min[min(P+)(F(e)一 B(e))，min(p-)(c(e)-F(e))

从 t 顶点出发，向 s 顶点倒推，按下述规律缩小可行流:

F(e) = F(e)+Q(P-) F(e) = F(e)-Q(P+)

2 数论

2.1 最大公约数 gcd

```
int gcd(int a,int b)
{
    If(b>a)    return gcd(b, a);
    if(b==0) return a;
    else return gcd(b,a%b);
}
```

2.2 最小公倍数 lcm

lcm(a, b) = a*b/gcd(a, b)

2.3 快速幂取模 B^LmodP(O(logb))

```
int power(int B, int L, int P)
{
    int acc = 1, q;
    for(q = L; q; q >>= 1)
    {
        if(q & 1)  acc = (_int64)acc * B % P;
        B = (_int64)B * B % P;
    }
    return acc;
}
```

2.4 Fermat 小定理

如果 p 是一个素数，则对任意 a 有 $a^p \equiv a \pmod{p}$。特别的，如果 p 不能整除 a，则还有 $a^{p-1} \equiv 1 \pmod{p}$。

2.5 Rabin-Miller 伪素数测试

```
int Rabin-Miller(int n)
{
    int i;
    for(i = 1; i <= s; ++i)
    {
        int a = rand()%(n-2)+2;
        if(power(a, n-1, n) != 1)  return 0;
    }
    return 1;
}
```

2.6 Pollard-rho

在开始时选取 0～n-1 范围内的随机数，然后递归地由

$$x_i = (x_{i-1}^2 - 1) \bmod n$$

产生无穷序列 $x_1, x_2, \quad , x_k,$

对于 i=2k，以及 2k<j≤2k+1，算法计算出 xj-xi 与 n 的最大公因子 d=gcd(xj-xi，n)。如果 d 是 n 的非平凡因子(非 1 和本身)，则实现对 n 的一次分割，算法输出 n 的因子 d。

```
void Pollard(int n)
{// 求整数 n 因子分割的拉斯维加斯算法
    int i=1;
    int x=rand(); // 随机整数
    int y=x;
    int k=2;
    while (true)
    {
        i++;
        x=(x*x-1)%n;       //
        int d=gcd(y-x,n);   // 求 n 的非平凡因子
        if ((d>1) && (d<n)) cout<<d<<endl;
        if (i==k)
        {
            y=x;
            k*=2;
        }
    }
}
int pollard-rho(n)
{
    int a;
    do
    {
        a = rand%(n+1);
    }
    while (a == 0 || a == 2)
    int x = rand();
    int y = x;
    int k = 2;
    int i = 1;
    while(1)
    {
        i = i + 1
        x = (x * x - a) % n
        int e = abs(x - y)
        int d = gcd(e, n)
        if(d != 1 && d != n) return d
        else if( i == k)
```

```
            {
                y = x;
                k = k << 1;
            }
        }
}
```

2.7 扩展欧几里德算法 extended-gcd

一定存在整数 x,y，使得 ax+by=gcd(a,b)。

```
int extendedgcd (int a, int b, int&x, int& y)
{
    if(!b)
    {
        x = 1, y = 0;
        return a;
    }
    int r = gcd(b, a%b, x, y);
    int t = x;
    x = y;
    y = t – a/b*y;
    return r;
}
int extendedgcd(int a, int b, __int64 &x, __int64 &y)
{
    if(!b)
    {
        x = 1 , y = 0;
        return   a;
    }
    int r = extendedgcd (b, a%b, y, x);
    y -= x * ( a / b );
}
```

2.8 欧拉定理

欧拉函数: 1~n 中和 n 互素的元素个数 ϕ(n)

Euler 定理:若 gcd(a, n)=1 则 a^ϕ(n)=1 (mod n), 当 b 很大时 a^b =ºa^(b mod ϕ(n))(mod n)

对于质数 p, ϕ(p) = p-1, ϕ(p^n) = p^(n-1)* ϕ(p); 当(m,n)=1 时, ϕ(mn)= ϕ(m)* ϕ(n)。

2.9 线性同余方程 $ax \equiv b \pmod{n}$

方法一：利用 Euler 函数

$a*a^{(\varphi(n)-1)} \equiv 1 \rightarrow a(b* a^{(\varphi(n)-1)}) \equiv b$
关键：求 $a^b \bmod n$
$a, a^2, a^4, a^8, a^{16}, \ldots$
同余方程可以做乘法，b 做二进制展开选择
方法二：扩展的 Euclid 算法
存在整数 y，使得 $ax-ny=b$
记 d=(a,n)，a'=a/d, n'=n/d，必须有 d|b
a' x-n' y=1*(b/d)
注意：x 不唯一，所有 x 相差 n/d 的整数倍

2.10 中国剩余定理

方程组 $x \equiv ai \pmod{mi}$, mi 两两互素,在 0<=x<M=m1m2…mk 内有唯一解,记 Mi=M/mi，则(Mi,mi)=1 存在 pi,qi，Mipi+miqi=1 记录 ei=Mipi，则

j=i 时 ei≡1(mod mj)
j!=i 时 ei≡0(mod mj)

则 e1a1+e2a2+…+ekak 就是一个解，调整得到[0,m)内的唯一解。

```
const int MM = 21252, M[4] = {0, 924, 759, 644};//MM 为 mi 相乘，Mi=MM/mi
const int m[4] = {0, 23, 28, 33};
int a[4];

int chinese_remain_theorem()
{
    int ans = 0;
    int i;
    for(i = 1; i <= 3; i++)
    {
        __int64 x, y;
        extendedgcd(M[i], m[i], x, y);
        x = (x + m[i])%m[i];
        ans = (ans + M[i] * x * a[i])%MM;
    }
    return ans;
}

__int64 chinese_remain_theorem ()
{
    __int64 ansa = a [0] , ansb = b [0] , a1 , a2 , d , p1 , p2;
    for ( int i = 1; i < N; i ++ )
    {
        a1 = ansa , a2 = a [i] , d = b [i] - ansb;
        extended_gcd ( a1 , a2 , p1 , p2 );
        p2 = ( ( - p2 ) * d ) % ansa;
        ansa *= a2;
```

```
            ansb = a2 * p2 + b [i];
            ansb = ( ansb % ansa + ansa ) % ansa;
        }
    return ansb ? ansb : ansa;
}
```

2.11 Discrete Logging($B^L == N \pmod P$)

giant steps-baby steps algorithm:
Input: A cyclic group G of order n, having a generator α and an element β.
Output: A value x satisfying α^x ≡ β (mod n).

m ← Ceiling(√n)
For all j where 0 ≤ j <= m:
Compute α^j mod n and store the pair (j, αj) in a table.
Compute α^(-m).
γ ← β .
For i = 0 to m:
Check to see if γ is the second component (αj) of any pair in the table.
If so, return im + j.
If not, γ ← γ * α ^(-m) mod n.

```
map<int, int> pp;
int p, b, n;

fun()
{
    int m;
    m = sqrt(p);
    int i;
    for(i = 0; i <= m; i++)
    {
        int re = power(b, i, p);
        if(pp.find(re) == pp.end())     pp[re] = i;
    }
    int a = power(b, p-1-m, p);
    int gama = n;
    for(i = 0; i <= m; i++)
    {
        if(pp.find(gama) != pp.end())
        {
            printf("%d\n", i*m+pp[gama]);
            goto done;
        }
        gama = (_int64)gama * a % p;
```

```
        }
        printf("no solution\n");
done:;
}
```

2.12 N!最后一个不为 0 的数字

```
int table[] = {1, 1, 2, 6, 4, 4, 4, 8, 4, 6};

int DIV2(int m)
{
        if(m == 2)      return 6;
        if(m == 4)      return 2;
        if(m == 6)      return 8;
        if(m == 8)      return 4;
}

int f(int n)
{
        if(n < 10)
        {
                if(n < 5)   return table[n];
                else  return DIV2(table[n]);
        }
        int ans = (f(n / 5) * table[n%10] * 6) % 10;
        for(int i = 1; i <= n / 5 % 4; ++i)    ans = DIV2(ans);
        return ans;
}

int main()
{
        int n;
        while(scanf("%d", &n) == 1)
        {
                printf("%5d -> %d\n", n, f(n));
        }
}
```

2.13 2^14 以内的素数

```
{
        2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, \
        73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, \
        157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, \
```

```
239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, \
331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, \
421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, \
509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, \
613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701, \
709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797, 809, 811, \
821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911, \
919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997, 1009, 1013, 1019, \
1021, 1031, 1033, 1039, 1049, 1051, 1061, 1063, 1069, 1087, 1091, 1093, 1097, \
1103, 1109, 1117, 1123, 1129, 1151, 1153, 1163, 1171, 1181, 1187, 1193, 1201, \
1213, 1217, 1223, 1229, 1231, 1237, 1249, 1259, 1277, 1279, 1283, 1289, 1291, \
1297, 1301, 1303, 1307, 1319, 1321, 1327, 1361, 1367, 1373, 1381, 1399, 1409, \
1423, 1427, 1429, 1433, 1439, 1447, 1451, 1453, 1459, 1471, 1481, 1483, 1487, \
1489, 1493, 1499, 1511, 1523, 1531, 1543, 1549, 1553, 1559, 1567, 1571, 1579, \
1583, 1597, 1601, 1607, 1609, 1613, 1619, 1621, 1627, 1637, 1657, 1663, 1667, \
1669, 1693, 1697, 1699, 1709, 1721, 1723, 1733, 1741, 1747, 1753, 1759, 1777, \
1783, 1787, 1789, 1801, 1811, 1823, 1831, 1847, 1861, 1867, 1871, 1873, 1877, \
1879, 1889, 1901, 1907, 1913, 1931, 1933, 1949, 1951, 1973, 1979, 1987, 1993, \
1997, 1999, 2003, 2011, 2017, 2027, 2029, 2039, 2053, 2063, 2069, 2081, 2083, \
2087, 2089, 2099, 2111, 2113, 2129, 2131, 2137, 2141, 2143, 2153, 2161, 2179, \
2203, 2207, 2213, 2221, 2237, 2239, 2243, 2251, 2267, 2269, 2273, 2281, 2287, \
2293, 2297, 2309, 2311, 2333, 2339, 2341, 2347, 2351, 2357, 2371, 2377, 2381, \
2383, 2389, 2393, 2399, 2411, 2417, 2423, 2437, 2441, 2447, 2459, 2467, 2473, \
2477, 2503, 2521, 2531, 2539, 2543, 2549, 2551, 2557, 2579, 2591, 2593, 2609, \
2617, 2621, 2633, 2647, 2657, 2659, 2663, 2671, 2677, 2683, 2687, 2689, 2693, \
2699, 2707, 2711, 2713, 2719, 2729, 2731, 2741, 2749, 2753, 2767, 2777, 2789, \
2791, 2797, 2801, 2803, 2819, 2833, 2837, 2843, 2851, 2857, 2861, 2879, 2887, \
2897, 2903, 2909, 2917, 2927, 2939, 2953, 2957, 2963, 2969, 2971, 2999, 3001, \
3011, 3019, 3023, 3037, 3041, 3049, 3061, 3067, 3079, 3083, 3089, 3109, 3119, \
3121, 3137, 3163, 3167, 3169, 3181, 3187, 3191, 3203, 3209, 3217, 3221, 3229, \
3251, 3253, 3257, 3259, 3271, 3299, 3301, 3307, 3313, 3319, 3323, 3329, 3331, \
3343, 3347, 3359, 3361, 3371, 3373, 3389, 3391, 3407, 3413, 3433, 3449, 3457, \
3461, 3463, 3467, 3469, 3491, 3499, 3511, 3517, 3527, 3529, 3533, 3539, 3541, \
3547, 3557, 3559, 3571, 3581, 3583, 3593, 3607, 3613, 3617, 3623, 3631, 3637, \
3643, 3659, 3671, 3673, 3677, 3691, 3697, 3701, 3709, 3719, 3727, 3733, 3739, \
3761, 3767, 3769, 3779, 3793, 3797, 3803, 3821, 3823, 3833, 3847, 3851, 3853, \
3863, 3877, 3881, 3889, 3907, 3911, 3917, 3919, 3923, 3929, 3931, 3943, 3947, \
3967, 3989, 4001, 4003, 4007, 4013, 4019, 4021, 4027, 4049, 4051, 4057, 4073, \
4079, 4091, 4093, 4099, 4111, 4127, 4129, 4133, 4139, 4153, 4157, 4159, 4177, \
4201, 4211, 4217, 4219, 4229, 4231, 4241, 4243, 4253, 4259, 4261, 4271, 4273, \
4283, 4289, 4297, 4327, 4337, 4339, 4349, 4357, 4363, 4373, 4391, 4397, 4409, \
4421, 4423, 4441, 4447, 4451, 4457, 4463, 4481, 4483, 4493, 4507, 4513, 4517, \
4519, 4523, 4547, 4549, 4561, 4567, 4583, 4591, 4597, 4603, 4621, 4637, 4639, \
4643, 4649, 4651, 4657, 4663, 4673, 4679, 4691, 4703, 4721, 4723, 4729, 4733, \
```

```
4751, 4759, 4783, 4787, 4789, 4793, 4799, 4801, 4813, 4817, 4831, 4861, 4871, \
4877, 4889, 4903, 4909, 4919, 4931, 4933, 4937, 4943, 4951, 4957, 4967, 4969, \
4973, 4987, 4993, 4999, 5003, 5009, 5011, 5021, 5023, 5039, 5051, 5059, 5077, \
5081, 5087, 5099, 5101, 5107, 5113, 5119, 5147, 5153, 5167, 5171, 5179, 5189, \
5197, 5209, 5227, 5231, 5233, 5237, 5261, 5273, 5279, 5281, 5297, 5303, 5309, \
5323, 5333, 5347, 5351, 5381, 5387, 5393, 5399, 5407, 5413, 5417, 5419, 5431, \
5437, 5441, 5443, 5449, 5471, 5477, 5479, 5483, 5501, 5503, 5507, 5519, 5521, \
5527, 5531, 5557, 5563, 5569, 5573, 5581, 5591, 5623, 5639, 5641, 5647, 5651, \
5653, 5657, 5659, 5669, 5683, 5689, 5693, 5701, 5711, 5717, 5737, 5741, 5743, \
5749, 5779, 5783, 5791, 5801, 5807, 5813, 5821, 5827, 5839, 5843, 5849, 5851, \
5857, 5861, 5867, 5869, 5879, 5881, 5897, 5903, 5923, 5927, 5939, 5953, 5981, \
5987, 6007, 6011, 6029, 6037, 6043, 6047, 6053, 6067, 6073, 6079, 6089, 6091, \
6101, 6113, 6121, 6131, 6133, 6143, 6151, 6163, 6173, 6197, 6199, 6203, 6211, \
6217, 6221, 6229, 6247, 6257, 6263, 6269, 6271, 6277, 6287, 6299, 6301, 6311, \
6317, 6323, 6329, 6337, 6343, 6353, 6359, 6361, 6367, 6373, 6379, 6389, 6397, \
6421, 6427, 6449, 6451, 6469, 6473, 6481, 6491, 6521, 6529, 6547, 6551, 6553, \
6563, 6569, 6571, 6577, 6581, 6599, 6607, 6619, 6637, 6653, 6659, 6661, 6673, \
6679, 6689, 6691, 6701, 6703, 6709, 6719, 6733, 6737, 6761, 6763, 6779, 6781, \
6791, 6793, 6803, 6823, 6827, 6829, 6833, 6841, 6857, 6863, 6869, 6871, 6883, \
6899, 6907, 6911, 6917, 6947, 6949, 6959, 6961, 6967, 6971, 6977, 6983, 6991, \
6997, 7001, 7013, 7019, 7027, 7039, 7043, 7057, 7069, 7079, 7103, 7109, 7121, \
7127, 7129, 7151, 7159, 7177, 7187, 7193, 7207, 7211, 7213, 7219, 7229, 7237, \
7243, 7247, 7253, 7283, 7297, 7307, 7309, 7321, 7331, 7333, 7349, 7351, 7369, \
7393, 7411, 7417, 7433, 7451, 7457, 7459, 7477, 7481, 7487, 7489, 7499, 7507, \
7517, 7523, 7529, 7537, 7541, 7547, 7549, 7559, 7561, 7573, 7577, 7583, 7589, \
7591, 7603, 7607, 7621, 7639, 7643, 7649, 7669, 7673, 7681, 7687, 7691, 7699, \
7703, 7717, 7723, 7727, 7741, 7753, 7757, 7759, 7789, 7793, 7817, 7823, 7829, \
7841, 7853, 7867, 7873, 7877, 7879, 7883, 7901, 7907, 7919, 7927, 7933, 7937, \
7949, 7951, 7963, 7993, 8009, 8011, 8017, 8039, 8053, 8059, 8069, 8081, 8087, \
8089, 8093, 8101, 8111, 8117, 8123, 8147, 8161, 8167, 8171, 8179, 8191, 8209, \
8219, 8221, 8231, 8233, 8237, 8243, 8263, 8269, 8273, 8287, 8291, 8293, 8297, \
8311, 8317, 8329, 8353, 8363, 8369, 8377, 8387, 8389, 8419, 8423, 8429, 8431, \
8443, 8447, 8461, 8467, 8501, 8513, 8521, 8527, 8537, 8539, 8543, 8563, 8573, \
8581, 8597, 8599, 8609, 8623, 8627, 8629, 8641, 8647, 8663, 8669, 8677, 8681, \
8689, 8693, 8699, 8707, 8713, 8719, 8731, 8737, 8741, 8747, 8753, 8761, 8779, \
8783, 8803, 8807, 8819, 8821, 8831, 8837, 8839, 8849, 8861, 8863, 8867, 8887, \
8893, 8923, 8929, 8933, 8941, 8951, 8963, 8969, 8971, 8999, 9001, 9007, 9011, \
9013, 9029, 9041, 9043, 9049, 9059, 9067, 9091, 9103, 9109, 9127, 9133, 9137, \
9151, 9157, 9161, 9173, 9181, 9187, 9199, 9203, 9209, 9221, 9227, 9239, 9241, \
9257, 9277, 9281, 9283, 9293, 9311, 9319, 9323, 9337, 9341, 9343, 9349, 9371, \
9377, 9391, 9397, 9403, 9413, 9419, 9421, 9431, 9433, 9437, 9439, 9461, 9463, \
9467, 9473, 9479, 9491, 9497, 9511, 9521, 9533, 9539, 9547, 9551, 9587, 9601, \
9613, 9619, 9623, 9629, 9631, 9643, 9649, 9661, 9677, 9679, 9689, 9697, 9719, \
9721, 9733, 9739, 9743, 9749, 9767, 9769, 9781, 9787, 9791, 9803, 9811, 9817, \
```

```
9829, 9833, 9839, 9851, 9857, 9859, 9871, 9883, 9887, 9901, 9907, 9923, 9929, \
9931, 9941, 9949, 9967, 9973, 10007, 10009, 10037, 10039, 10061, 10067, \
10069, 10079, 10091, 10093, 10099, 10103, 10111, 10133, 10139, 10141, 10151, \
10159, 10163, 10169, 10177, 10181, 10193, 10211, 10223, 10243, 10247, 10253, \
10259, 10267, 10271, 10273, 10289, 10301, 10303, 10313, 10321, 10331, 10333, \
10337, 10343, 10357, 10369, 10391, 10399, 10427, 10429, 10433, 10453, 10457, \
10459, 10463, 10477, 10487, 10499, 10501, 10513, 10529, 10531, 10559, 10567, \
10589, 10597, 10601, 10607, 10613, 10627, 10631, 10639, 10651, 10657, 10663, \
10667, 10687, 10691, 10709, 10711, 10723, 10729, 10733, 10739, 10753, 10771, \
10781, 10789, 10799, 10831, 10837, 10847, 10853, 10859, 10861, 10867, 10883, \
10889, 10891, 10903, 10909, 10937, 10939, 10949, 10957, 10973, 10979, 10987, \
10993, 11003, 11027, 11047, 11057, 11059, 11069, 11071, 11083, 11087, 11093, \
11113, 11117, 11119, 11131, 11149, 11159, 11161, 11171, 11173, 11177, 11197, \
11213, 11239, 11243, 11251, 11257, 11261, 11273, 11279, 11287, 11299, 11311, \
11317, 11321, 11329, 11351, 11353, 11369, 11383, 11393, 11399, 11411, 11423, \
11437, 11443, 11447, 11467, 11471, 11483, 11489, 11491, 11497, 11503, 11519, \
11527, 11549, 11551, 11579, 11587, 11593, 11597, 11617, 11621, 11633, 11657, \
11677, 11681, 11689, 11699, 11701, 11717, 11719, 11731, 11743, 11777, 11779, \
11783, 11789, 11801, 11807, 11813, 11821, 11827, 11831, 11833, 11839, 11863, \
11867, 11887, 11897, 11903, 11909, 11923, 11927, 11933, 11939, 11941, 11953, \
11959, 11969, 11971, 11981, 11987, 12007, 12011, 12037, 12041, 12043, 12049, \
12071, 12073, 12097, 12101, 12107, 12109, 12113, 12119, 12143, 12149, 12157, \
12161, 12163, 12197, 12203, 12211, 12227, 12239, 12241, 12251, 12253, 12263, \
12269, 12277, 12281, 12289, 12301, 12323, 12329, 12343, 12347, 12373, 12377, \
12379, 12391, 12401, 12409, 12413, 12421, 12433, 12437, 12451, 12457, 12473, \
12479, 12487, 12491, 12497, 12503, 12511, 12517, 12527, 12539, 12541, 12547, \
12553, 12569, 12577, 12583, 12589, 12601, 12611, 12613, 12619, 12637, 12641, \
12647, 12653, 12659, 12671, 12689, 12697, 12703, 12713, 12721, 12739, 12743, \
12757, 12763, 12781, 12791, 12799, 12809, 12821, 12823, 12829, 12841, 12853, \
12889, 12893, 12899, 12907, 12911, 12917, 12919, 12923, 12941, 12953, 12959, \
12967, 12973, 12979, 12983, 13001, 13003, 13007, 13009, 13033, 13037, 13043, \
13049, 13063, 13093, 13099, 13103, 13109, 13121, 13127, 13147, 13151, 13159, \
13163, 13171, 13177, 13183, 13187, 13217, 13219, 13229, 13241, 13249, 13259, \
13267, 13291, 13297, 13309, 13313, 13327, 13331, 13337, 13339, 13367, 13381, \
13397, 13399, 13411, 13417, 13421, 13441, 13451, 13457, 13463, 13469, 13477, \
13487, 13499, 13513, 13523, 13537, 13553, 13567, 13577, 13591, 13597, 13613, \
13619, 13627, 13633, 13649, 13669, 13679, 13681, 13687, 13691, 13693, 13697, \
13709, 13711, 13721, 13723, 13729, 13751, 13757, 13759, 13763, 13781, 13789, \
13799, 13807, 13829, 13831, 13841, 13859, 13873, 13877, 13879, 13883, 13901, \
13903, 13907, 13913, 13921, 13931, 13933, 13963, 13967, 13997, 13999, 14009, \
14011, 14029, 14033, 14051, 14057, 14071, 14081, 14083, 14087, 14107, 14143, \
14149, 14153, 14159, 14173, 14177, 14197, 14207, 14221, 14243, 14249, 14251, \
14281, 14293, 14303, 14321, 14323, 14327, 14341, 14347, 14369, 14387, 14389, \
14401, 14407, 14411, 14419, 14423, 14431, 14437, 14447, 14449, 14461, 14479, \
```

```
    14489, 14503, 14519, 14533, 14537, 14543, 14549, 14551, 14557, 14561, 14563, \
    14591, 14593, 14621, 14627, 14629, 14633, 14639, 14653, 14657, 14669, 14683, \
    14699, 14713, 14717, 14723, 14731, 14737, 14741, 14747, 14753, 14759, 14767, \
    14771, 14779, 14783, 14797, 14813, 14821, 14827, 14831, 14843, 14851, 14867, \
    14869, 14879, 14887, 14891, 14897, 14923, 14929, 14939, 14947, 14951, 14957, \
    14969, 14983, 15013, 15017, 15031, 15053, 15061, 15073, 15077, 15083, 15091, \
    15101, 15107, 15121, 15131, 15137, 15139, 15149, 15161, 15173, 15187, 15193, \
    15199, 15217, 15227, 15233, 15241, 15259, 15263, 15269, 15271, 15277, 15287, \
    15289, 15299, 15307, 15313, 15319, 15329, 15331, 15349, 15359, 15361, 15373, \
    15377, 15383, 15391, 15401, 15413, 15427, 15439, 15443, 15451, 15461, 15467, \
    15473, 15493, 15497, 15511, 15527, 15541, 15551, 15559, 15569, 15581, 15583, \
    15601, 15607, 15619, 15629, 15641, 15643, 15647, 15649, 15661, 15667, 15671, \
    15679, 15683, 15727, 15731, 15733, 15737, 15739, 15749, 15761, 15767, 15773, \
    15787, 15791, 15797, 15803, 15809, 15817, 15823, 15859, 15877, 15881, 15887, \
    15889, 15901, 15907, 15913, 15919, 15923, 15937, 15959, 15971, 15973, 15991, \
    16001, 16007, 16033, 16057, 16061, 16063, 16067, 16069, 16073, 16087, 16091, \
    16097, 16103, 16111, 16127, 16139, 16141, 16183, 16187, 16189, 16193, 16217, \
    16223, 16229, 16231, 16249, 16253, 16267, 16273, 16301, 16319, 16333, 16339, \
    16349, 16361, 16363, 16369, 16381
}
```

3 数据结构

3.1 堆（最小堆）

存储方式：
最小堆的元素保存在 heap[1..hs]内；
根在 heap[1]；
K 的左儿子是 2k, K 的右儿子是 2k+1；
K 的父亲是[k/2]。

3.1.1 删除最小值元素：

直接删除根；
用最后一个元素代替根上元素；
向下调整。

```
void sink(int p)
{
    int q=p<<1, a = heap[p];
    while(q<=hs)                    //hs 为堆的大小
    {
        if(q<hs&&heap[q+1]<heap[q])q++;
```

```
            if(heap[q]>=a) break;
            heap[p]=heap[q]; p=q; q=p<<1;
        }
        heap[p] = a;
}
```

3.1.2 插入元素和向上调整：

插入元素是先添加到末尾, 再向上调整；
向上调整: 比较当前结点 p 和父亲, 如果父亲比 p 小，停止; 否则交换父亲和 p, 继续调整

```
void swim(int p)
{
        int q = p>>1, a = heap[p];
        while(q && a<heap[q]){ heap[p]=heap[q]; p=q; q=p>>1; }
        heap[p] = a;
}
```

3.1.3 堆的建立

从下往上逐层向下调整. 所有的叶子无需调整,因此从 hs/2 开始.

```
void insert(int a)
{
        heap[++hs]=a;
        swim(hs);
}

int getmin()
{
        int r=heap[1]; heap[1]=heap[hs--];
        sink(1); return r;
}

int decreaseKey(int p, int a){ heap[p]=a; swim(p); }

void build(){ for(int i=hs/2;i>0;i--) sink(i); }
```

3.2 并查集

```
＃i nclude<iostream>
using namespace std;
int N,M,Q;
int pre[20000],rank[20000];
void makeset(int x)
```

```
 {
       pre[x]=-1;
       rank[x]=0;
 }
int find(int x)
 {
       int r=x;
       while(pre[r]!=-1)
         r=pre[r];
   //压缩路径
       while(x!=r)
        {
              int q=pre[x];
              pre[x]=r;
              x=q;
        }
      return r;
 }
void unionone(int a,int b)
 {
       int t1=find(a);
       int t2=find(b);
       if(rank[t1]>rank[t2])
          pre[t2]=t1;
      else
          pre[t1]=t2;
      if(rank[t1]==rank[t2])
         rank[t2]++;
 }
int main()
{
    int i,a,b,c,d;
      while(cin>>N>>M)
       {
            for(i=1;i<=N;i++)
              makeset(i);
           for(i=1;i<=M;i++)
              {
                    cin>>a>>b;
                    if(find(a)!=find(b))
                      unionone(a,b);
              }
           cin>>Q;
           for(i=1;i<=Q;i++)
```

```
            {
                cin>>c>>d;
                if(find(c)==find(d))
                  cout<<"YES"<<endl;
                else
                  cout<<"NO"<<endl;
            }
        }
    return 0;
}
```

3.3 树状数组

3.3.1 LOWBIT

设 C[i]=a[i-2^k+1]+…+a[i],其中 k 为 i 在二进制下末尾 0 的个数，令 LOWBIT(i)=2^k

```
inline int lowbit(int x)
{
    return x & (x ^ ( x - 1 ) );
}
```

3.3.2 修改 a[p]

```
void Add(int p , int d)
{
    while(p <= n)
    {
        C[p] += d;
        p += lowbit(p);
    }
}
```

3.3.3 前缀和 A[1]+…+A[p]

```
int Sum(int p)
{
    int ret = 0;
    while( p )
    {
        ret += C[p];
        p -= lowbit(p);
    }
```

```
    return ret;
}
```

3.3.4 一个二维树状数组的程序

```
const int maxn=1025;
int a[maxn][maxn];
int s, i,j, count;

inline int lowbit(int a)
{return -a&a;}

inline void update(int    &x,int &y,int &t)
{
    for(i=x;i<=s;i+=lowbit(i))
        for(j=y;j<=s;j+=lowbit(j))
            a[i][j] += t;
}

int sum(int    &x1,int  &y1,int  &x2,int  &y2)
{
    count=0;
    for (i=x2;i;i-=lowbit(i))
    {
        for(j=y2;j;j-=lowbit(j))
            count += a[i][j];
        for(j=y1-1;j;j-=lowbit(j))
            count -= a[i][j];
    }
    for(i=x1-1;i;i-=lowbit(i))
    {
        for(j=y2;j;j-=lowbit(j))
            count -= a[i][j];
        for(j=y1-1;j;j-=lowbit(j))
            count += a[i][j];
    }
    return count;
}

int main()
{
    int key,x,y,t,x1, y1, x2, y2;
    while(1)
    {
```

```
            scanf("%d", &key);
            if(key == 3)    break;
            else  if(key == 0)
                  scanf("%d",&s);
            else  if(key == 1)
            {
                  scanf("%d %d %d",&x,&y,&t);
                  ++x, ++y;
                  update(x,y,t);
            }
            else
            {
                  scanf("%d %d %d %d",&x1,&y1,&x2,&y2);
                  ++x1, ++x2, ++y1, ++y2;
                  printf("%d\n", sum(x1,y1,x2,y2));
            }
      }
      return 0;
}
```

3.4 线段树

Version 1：

```
struct segmenttree
{
      int a;
      int b;
      int count;
}tree[MAX];

void insert(int i, int a, int b)
{
      if(a == tree[i].a && b == tree[i].b)
      {
            tree[i].count++;
            return;
      }
      int m = (tree[i].a + tree[i].b) / 2;
      if(b <= m)       insert(2*i, a, b);
      else if(a >= m)        insert(2*i+1, a, b);
      else
      {
            insert(2*i, a, m);
            insert(2*i+1, m, b);
```

```
    }
    tree[i].count = tree[2*i].count + tree[2*i+1].count;
}

int find(int i, int a, int b)
{
    if(a == tree[i].a && b == tree[i].b)        return tree[i].count;
    int m = (tree[i].a + tree[i].b) / 2;
    if(b <= m)       return find(2*i, a, b);
    else if(a >= m)        return find(2*i+1, a, b);
    else         return find(2*i, a, m) + find(2*i+1, m, b);
}

void init(int i, int a, int b)     //初始化
{
    tree[i].a = a;
    tree[i].b = b;
    tree[i].count = 0;
    if(a+1 < b)
    {
        int m = (a + b) / 2;
        init(2*i, a, m);
        init(2*i+1, m, b);
    }
}
```

Version 2：

```
struct node{int s,t,c;}t[61000];
char hash[10010];

void make_tree(int nd,int st,int ed)
{
    if (nd>tn) tn=nd;
    t[nd].s=st;t[nd].t=ed;t[nd].c=0;
    if (ed-st>1)
    {
        make_tree(nd+nd,st,(st+ed)/2);
        make_tree(nd+nd+1,(st+ed)/2,ed);
    }
}

void insert_tree(int nd,int st,int ed,int col)
{
    if (t[nd].c!=col)
```

```
    {
        if (t[nd].s==st&&t[nd].t==ed)
        {
            t[nd].c=col;
            return ;
        }
        int m=(t[nd].s+t[nd].t)/2;
        if (t[nd].c>=0)
        {
            t[nd+nd].c=t[nd+nd+1].c=t[nd].c;
            t[nd].c=-1;
        }
        if (m<=st)        insert_tree(nd+nd+1,st,ed,col);
        else  if (m>=ed)      insert_tree(nd+nd,st,ed,col);
        else
        {
            insert_tree(nd+nd,st,m,col);
            insert_tree(nd+nd+1,m,ed,col);
        }
    }
}

void count_tree(int nd)
{
    if (t[nd].c>0)          hash[t[nd].c]=1;
    else
    {
        if (nd+nd<=tn) count_tree(nd+nd);
        if (nd+nd+1<=tn) count_tree(nd+nd+1);
    }
}
```

3.5 字符串

3.5.1 字符串哈希

```
// RS Hash Function
unsigned int RSHash(char *str)
{
    unsigned int b = 378551;
    unsigned int a = 63689;
    unsigned int hash = 0;
    while (*str)
    {
        hash = hash * a + (*str++);
        a *= b;
```

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

// JS Hash Function
unsigned int JSHash(char *str)
{
    unsigned int hash = 1315423911;
    while (*str)
    {
        hash ^= ((hash << 5) + (*str++) + (hash >> 2));
    }
    return (hash & 0x7FFFFFFF);
}

// P. J. Weinberger Hash Function
unsigned int PJWHash(char *str)
{
    unsigned int BitsInUnignedInt =
        (unsigned int)(sizeof(unsigned int) * 8);
    unsigned int ThreeQuarters =
        (unsigned int)((BitsInUnignedInt  * 3) / 4);
    unsigned int OneEighth =
        (unsigned int)(BitsInUnignedInt / 8);
    unsigned int HighBits =
        (unsigned int)(0xFFFFFFFF) << (BitsInUnignedInt - OneEighth);
    unsigned int hash = 0;
    unsigned int test = 0;
    while (*str)
    {
        hash = (hash << OneEighth) + (*str++);
        if ((test = hash & HighBits) != 0)
        {
            hash = ((hash ^ (test >> ThreeQuarters)) & (~HighBits));
        }
    }
    return (hash & 0x7FFFFFFF);
}

// ELF Hash Function
unsigned int ELFHash(char *str)
{
    unsigned int hash = 0;
    unsigned int x    = 0;
    while (*str)
    {
        hash = (hash << 4) + (*str++);
        if ((x = hash & 0xF0000000L) != 0)
        {
            hash ^= (x >> 24);
            hash &= ~x;
        }
    }
    return (hash & 0x7FFFFFFF);
}
```

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

// SDBM Hash Function
unsigned int SDBMHash(char *str)
{
    unsigned int hash = 0;
    while (*str)
    {
        hash = (*str++) + (hash << 6) + (hash << 16) - hash;
    }
    return (hash & 0x7FFFFFFF);
}

// DJB Hash Function
unsigned int DJBHash(char *str)
{
    unsigned int hash = 5381;
    while (*str)
    {
        hash += (hash << 5) + (*str++);
    }
    return (hash & 0x7FFFFFFF);
}

// AP Hash Function
unsigned int APHash(char *str)
{
    unsigned int hash = 0;
    int i;
    for (i=0; *str; i++)
    {
        if ((i & 1) == 0)
            hash ^= ((hash << 7) ^ (*str++) ^ (hash >> 3));
        else
            hash ^= (~((hash << 11) ^ (*str++) ^ (hash >> 5)));
    }
    return (hash & 0x7FFFFFFF);
}
```

3.5.2 KMP 算法

```
int next[20];
void get_nextval (char T[],int next[])
{
```

```
    int j,k;
    j=0;
    next[0]=-1;
    k=-1;
    while (T[j+1]!='\0')
    {
        if (k == -1||T[j]==T[k])
        {
            ++j,++k;
            if(T[j]!=T[k])  next[j]=k;
            else  next[j]=next[k];
        }
        else        k=next[k];
    }
}

int Index_KMP(char S[],char T[],int pos)
{
    int i,j;
    i = pos-1;
    j=0;
    while (S[i]!='\0'&&T[j]!='\0')
    {
        if(j ==-1||S[i]==T[j])      i++,j++;
        else  j = next[j];
    }
    if (T[j]=='\0')   return (i-j);
    else        return 0;
}

int main()
{
    int m;
    char S[20]="abcdfadabddfa";
    char T[20]="bc";
    get_nextval(T,next);
    m=Index_KMP(S,T,1);
    printf("%d",m);
    getch();
    return 0；
}
```

4 计算几何

4.1 直线交点

```
struct Point{double x,y;}in[max]; //// 直线交点函数
int LineIntersect(Point A,Point B,Point C,Point D,double &x,double &y)
{
    double a1,a2,b1,b2,c1,c2;
    double Delta , Delta_x , Delta_y;
    a1 = B.x - A.x;       b1 = C.x - D.x;       c1 = C.x - A.x;
    a2 = B.y - A.y;       b2 = C.y - D.y;       c2 = C.y - A.y;
    Delta=a1*b2-a2*b1; Delta_x=c1*b2-c2*b1;Delta_y=a1*c2-a2*c1;
    if(Delta)
    {
        x = A.x+a1*(Delta_x/Delta);
        y = A.y+a2*(Delta_x/Delta);
        return 1; //返回 1:  表示两条直线相交,且交点是(x , y)
    }
    else
    {
        if(!Delta_x && !Delta_y) return -1; //返回是-1: 表示两条直线是重合关系
        else return 0; //返回 0:表示两条直线是平行不相交关系
    }
}
```

4.2 判断线段相交

```
bool segmentIntersect(point x1, point y1, point x2, point y2,
                      point x3, point y3, point x4, point y4)
{
    if(x3 > x1 && x3 > x2 && x4 > x1 && x4 > x2)
        return false;
    if(x3 < x1 && x3 < x2 && x4 < x1 && x4 < x2)
        return false;
    if(y3 > y1 && y3 > y2 && y4 > y1 && x4 > y2)
        return false;
    if(y3 < y1 && y3 < y2 && y4 < y1 && x4 < y2)
        return false;
    point x5 = x4-x3, y5 = y4-y3;
    point x6 = x1-x3, y6 = y1-y3;
    point x7 = x2-x3, y7 = y2-y3;
    if((x6*y5-x5*y6)*(x7*y5-x5*y7)>0)    return false;
```

```
    return true;
}
```

4.3 三点外接圆圆心

```
double getRadiusBy3Points(point x1 , point y1 , point x2 , point y2 ,   //外接圆圆心
                point x3 , point y3 ,   point &x , point &y )
{
    //由  ( x - x1 )^2 + ( y - y1 )^2 = ( x - x2 )^2 + ( y - y2 )^2  得
    //2*( x2 - x1 )*x + 2*( y2 - y1 )*y = x2^2 - x1^2 + y2^2 - y1^2
    //同理得
    //2*( x3 - x2 )*x + 2*( y3 - y2 )*y = x3^2 - x2^2 + y3^2 - y2^2
    //由行列式解方程得  x , y
    double a11 , a12 , a21 , a22 , b1 , b2 ;
    double d , d1 , d2 ;
    a11 = 2 * ( x3 - x2 ) ;
    a12 = 2 * ( y3 - y2 ) ;
    a21 = 2 * ( x2 - x1 ) ;
    a22 = 2 * ( y2 - y1 ) ;
    b1 = x3*x3 - x2*x2 + y3*y3 - y2*y2 ;
    b2 = x2*x2 - x1*x1 + y2*y2 - y1*y1 ;
    d = a11*a22 - a12*a21 ;
    d1 = b1*a22 - a12*b2 ;
    d2 = a11*b2 - b1*a21 ;
    x = d1 / d ;
    y = d2 / d ;
    return    (x1 - x)*(x1 - x) + (y1 - y)*(y1 - y) ; // x , y  是圆心坐标
}
```

4.4 判断点在多边形内

```
typedef struct { double x, y; } Point;      Point poly[MAX+10]; //判断点是否在多边形内
#define EPS 1e-8
#define SQR(x) ((x)*(x))
double dist2d(Point a, Point b) {return sqrt(SQR(a.x-b.x) + SQR(a.y-b.y));}
int pt_in_poly(Point *p, int n, Point a)
{
    int i, j, c = 0;
    for (i = 0, j = n-1; i < n; j = i++)
    {
        if (dist2d(p[i],a)+dist2d(p[j],a)-dist2d(p[i],p[j]) < EPS)
            return 1;
        if ((((p[i].y<=a.y) && (a.y<p[j].y)) || ((p[j].y<=a.y) && (a.y<p[i].y))) &&
        (a.x < (p[j].x-p[i].x) * (a.y - p[i].y) / (p[j].y-p[i].y) + p[i].x))
```

```
                c = !c;
        }
        return c;
}
```

4.5 两圆交面积

```
#define pi 2*acos(0)            //求两圆交面积
int main()
{
        double x1,y1,r1,x2,y2,r2,a1,a2,Ans,d;
        while(scanf("%lf%lf%lf%lf%lf%lf",&x1,&y1,&r1,&x2,&y2,&r2)!=EOF)
        {
                d=sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));
                if (r1+r2<=d) {printf("0.000\n",0);continue;}
                if (fabs(r1-r2)>=d)
                {printf("%.3lf\n",pi*(r1>r2?r2:r1)*(r1>r2?r2:r1)); continue;}
                a1 = (r1*r1 + d*d - r2*r2) / (2 * r1 * d);
                a2 = (r2*r2 + d*d - r1*r1) / (2 * r2 * d);
                a1 = 2 * acos(a1);
                a2 = 2 * acos(a2);
                Ans = r1*r1 * (a1 - sin(a1)) / 2;
                Ans = Ans + r2*r2 * (a2 - sin(a2)) / 2;
                printf("%.3lf\n",Ans);
        }
        return 0;
}
```

4.6 最小包围圆

```
typedef double point;
const int mm = 1001;
point x[mm], y[mm], xc, yc;
int flag[mm];
int n, p1, p2, p3;
double r;

void getans()
{
        p1 = p2 = p3 = 0;
        int i, j;
        for(i = 1;    i<= n; ++i)
                flag[i] = 0;
        double maxd = 0;
```

```
for(i = 1; i <= n; ++i)
    for(j = i+1; j<=n; ++j)
        if(maxd < (x[j]-x[i])*(x[j]-x[i])+(y[i]-y[j])*(y[i]-y[j]))
            maxd = (x[j]-x[i])*(x[j]-x[i])+(y[i]-y[j])*(y[i]-y[j]), p1 = i, p2= j;
flag[p1]=flag[p2] = 1;
while(1)
{
    if(p1 && p2 && p3)
    {
        // getRadiusBy3Points 参见前面
        r = getRadiusBy3Points(x[p1], y[p1], x[p2], y[p2], x[p3], y[p3],
            xc, yc);
    }
    else
    {
        xc = (x[p1]+x[p2])/2;
        yc = (y[p1]+y[p2])/2;
        r = (x[p1]-xc)*(x[p1]-xc)+(y[p1]-yc)*(y[p1]-yc);
    }
    maxd = 0;
    int ind = 0,tmp;
    for(i = 1; i <= n; ++i)
        if(!flag[i])
        {
            double rr = (x[i]-xc)*(x[i]-xc)+(y[i]-yc)*(y[i]-yc);
            if(rr <= r) flag[i] = 1;
            else
            {
                if(maxd < rr)
                {
                    maxd = rr, ind = i;
                    //break;
                }
            }
        }
    if(ind == 0)    break;
    point x41 = x[ind]-x[p1], y41 = y[ind]-y[p1];
    point x42 = x[ind]-x[p2], y42 = y[ind]-y[p2];
    point x12 = x[p1]-x[p2], y12 = y[p1]-y[p2];
    if(p1 && p2 && p3)
    {
        // segmentIntersect 参见前面
        if(segmentIntersect(x[p2], y[p2],x[p3], y[p3],x[p1], y[p1],x[ind], y[ind]))
            tmp = p1, p1 = p2, p2 = tmp;
```

```
                else  if(segmentIntersect(x[p2], y[p2],x[p1], y[p1],x[p3], y[p3],
                      x[ind], y[ind]))
                      tmp = p3, p3 = p2, p2 = tmp;
                point x32 = x[p3]-x[p2], y32 = y[p3]-y[p2];
                point x43 = x[ind]-x[p3], y43 = y[ind]-y[p3];
                if((x41*x41+y41*y41+x12*x12+y12*y12-x42*x42-y42*y42)*
                      sqrt(x43*x43+y43*y43)*sqrt(x32*x32+y32*y32)<
                      (x43*x43+y43*y43+x32*x32+y32*y32-x42*x42-y42*y42)*
                      sqrt(x41*x41+y41*y41)*sqrt(x12*x12+y12*y12))
                      tmp = p1, p1 = p3, p3 =tmp;
                if(x41*x41+y41*y41+x12*x12+y12*y12<=x42*x42+y42*y42)
                      p1 = ind, p3 = 0;
                else  p3 = ind;
          }
          else
          {
                if(x41*x41+y41*y41+x12*x12+y12*y12<=x42*x42+y42*y42)
                      p1 = ind;
                else  if(x42*x42+y42*y42+x12*x12+y12*y12<=x41*x41+y41*y41)
                      p2 = ind;
                else  p3 = ind;
          }
     }
}

int main()
{
     while(scanf("%d", &n)==1 && n)
     {
          int i;
          for(i = 1; i <= n; ++i)      scanf("%lf %lf", x+i, y+i);
          getans();
          printf("%.3f\n", sqrt(r));
     }
     return 0;
}
```

4.7 经纬度坐标

```
//giving the latitude (between -90 and +90 degrees) and longitude (between -180 and +180 degrees)
of an airport.
double hudu(double a){ return pi*a/180;}
double yuxr(double x,double y,double a)//已知 x,y 和夹角 a，求 z
{return sqrt(x*x+y*y-2*x*y*cos(a));}
```

```
double dt_th(double a,double b,double a1,double b1)
{
        double x1,y1,x,y,dx,dL,dth,h;
        a=hudu(a),b=hudu(b);
        x=R*cos(a),y=R*sin(a);
        a1=hudu(a1),b1=hudu(b1);
        x1=R*cos(a1),y1=R*sin(a1);
        dx=yuxr(x,x1,b-b1);
        dL=sqrt(dx*dx+(y-y1)*(y-y1));// dL 为两点直线距离
        dth=2*asin(dL/(2*R));
       return dth;
}//两点纬度和经度（角度）为 a,b 和 a1,b1，求两点夹角
// tmp=dt_th(p[i].w,p[i].j,p[j].w,p[j].j);
```

4.8 凸包

```
//Graham 扫描法
#define pi 3.14159265358979323846264338327 95

struct vertex
{
      int x;
      int y;
}v[1001];
int n, l, low, t;
int stack[1001];

int cmp(const void * a, const void * b)
{
      int x, y, c, d;
      x = ((struct vertex *)a)->x - v[0].x;
      y = ((struct vertex *)a)->y - v[0].y;
      c = ((struct vertex *)b)->x - v[0].x;
      d = ((struct vertex *)b)->y - v[0].y;
      if(x * d == y * c)    return (x * x + y * y - c * c - d * d);
      else  return y * c - x * d;
}

int left(int a, int b, int c)
{
      int x, y, p, q;
      x = v[a].x - v[c].x;
      y = v[a].y - v[c].y;
      p = v[b].x - v[c].x;
```

```
    q = v[b].y - v[c].y;
    if(q * x > p * y)       return 1;
    return 0;
}

int main()
{
    scanf("%d %d", &n, &l);
    int i, m = 20000, a = 20000;
    for(i = 1; i <= n; i++)
    {
        scanf("%d %d", &v[i].x, &v[i].y);
        if(m > v[i].y)
        {
            m = v[i].y,a = v[i].x,low = i;
        }
        else if(m == v[i].y)
        {
            if(a>v[i].x)
            {
                a = v[i].x,low = i;
            }
        }
    }
    v[0].x = v[low].x;
    v[0].y = v[low].y;
    qsort(v+1, n, sizeof(struct vertex), cmp);
    stack[1] = 1;
    t = 3;
    int j = 3, k = 2;
    while((v[j].x - v[1].x) * (v[j-1].y - v[1].y) == (v[j].y - v[1].y) * (v[j-1].x- v[1].x))
        j++,k++;
    stack[2] = k;
    k = j,j++;
    while((v[j].x - v[1].x) * (v[j-1].y - v[1].y) == (v[j].y - v[1].y) * (v[j-1].x- v[1].x)&&j<=n)
        j++,k++;
    stack[3] = k;
    for(i = j; i <= n; i++)
    {
        while((v[i].x - v[1].x) * (v[i+1].y - v[1].y) ==
            (v[i].y - v[1].y) * (v[i+1].x- v[1].x)&&i+1<=n)    i++;
        while(left(i, stack[t], stack[t-1]))    t--;
        stack[++t] = i;
    }
```

```
    double min = 0;
    for(i = 1; i < t; i++)
    {
        min += sqrt((v[stack[i+1]].x-v[stack[i]].x)*(v[stack[i+1]].x-v[stack[i]].x)
            +(v[stack[i+1]].y-v[stack[i]].y)*(v[stack[i+1]].y-v[stack[i]].y));
    }
    min += sqrt((v[stack[1]].x-v[stack[t]].x)*(v[stack[1]].x-v[stack[t]].x)
            +(v[stack[1]].y-v[stack[t]].y)*(v[stack[1]].y-v[stack[t]].y));
    min += 2 * pi * l;
    printf("%d\n", (int)(min+0.5));
    return 0;
}
```

5 Problem

5.1 RMQ-LCA

Notations

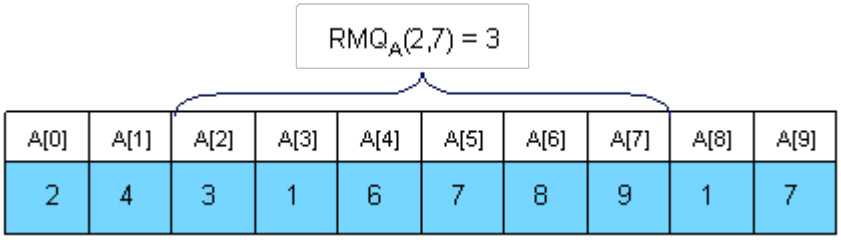
Suppose that an algorithm has preprocessing time **f(n)** and query time **g(n)**. The notation for the overall complexity for the algorithm is **<f(n), g(n)>**.

We will note the position of the element with the minimum value in some array **A** between indices **i** and **j** with $\mathbf{RMQ_A(i, j)}$.

The furthest node from the root that is an ancestor of both **u** and **v** in some rooted tree **T** is $\mathbf{LCA_T(u, v)}$.

5.1.1 Range Minimum Query(RMQ)

Given an array **A[0, N-1]** find the position of the element with the minimum value between two given indices.



5.1.1.1 Trivial algorithms for RMQ

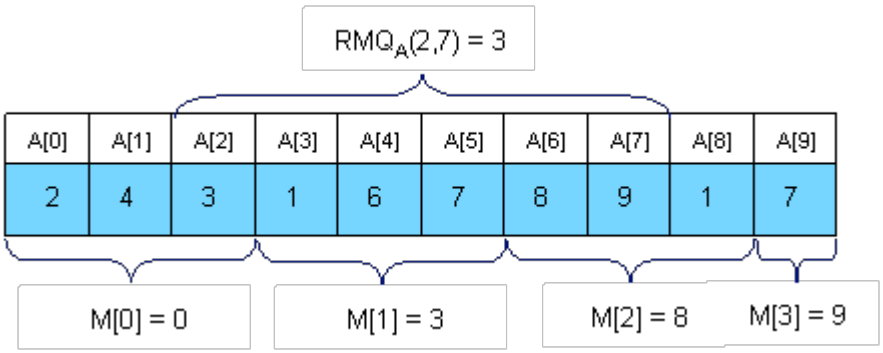
For every pair of indices (i, j) store the value of RMQA(i, j) in a table M[0, N-1][0, N-1]. Trivial computation will lead us to an <O(N3), O(1)> complexity. However, by using an easy dynamic programming approach we can reduce the complexity to <O(N2), O(1)>. The preprocessing

function will look something like this:

```
void process1(int M[MAXN][MAXN], int A[MAXN], int N)
{
    int i, j;
    for (i =0; i < N; i++)      M[i][i] = i;
    for (i = 0; i < N; i++)
        for (j = i + 1; j < N; j++)
            if (A[M[i][j - 1]] < A[j]) M[i][j] = M[i][j - 1];
            else  M[i][j] = j;
}
```

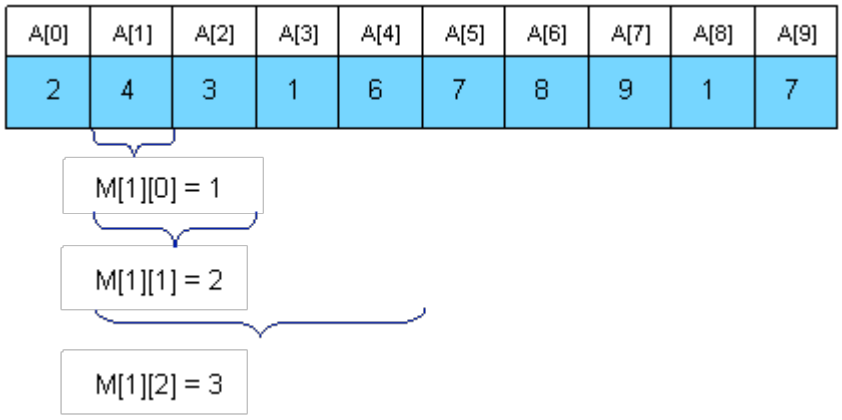
5.1.1.2 An <O(N), O(sqrt(N))> solution

An interesting idea is to split the vector in **sqrt(N)** pieces. We will keep in a vector **M[0, sqrt(N)-1]** the position for the minimum value for each section. **M** can be easily preprocessed in **O(N)**. Here is an example:



5.1.1.3 Sparse Table (ST) algorithm

A better approach is to preprocess **RMQ** for sub arrays of length 2^k using dynamic programming. We will keep an array **M[0, N-1][0, logN]** where **M[i][j]** is the index of the minimum value in the sub array starting at **i** having length 2^j. Here is an example:



For computing **M[i][j]** we must search for the minimum value in the first and second half of the interval. It's obvious that the small pieces have 2^{j-1} length, so the recurrence is:

```
void process2(int M[MAXN][LOGMAXN], int A[MAXN], int N)
{
      int i, j;
   //initialize M for the intervals with length 1
      for (i = 0; i < N; i++)      M[i][0] = i;
   //compute values from smaller to bigger intervals
      for (j = 1; 1 << j <= N; j++)
            for (i = 0; i + (1 << j) - 1 < N; i++)
                  if (A[M[i][j - 1]] < A[M[i + (1 << (j - 1))][j - 1]])
                        M[i][j] = M[i][j - 1];
                  else
                        M[i][j] = M[i + (1 << (j - 1))][j - 1];
}
```

Once we have these values preprocessed, let's show how we can use them to calculate **$RMQ_A(i, j)$**. The idea is to select two blocks that entirely cover the interval **[i..j]** and find the minimum between them. Let **k = [log(j - i + 1)]**. For computing **$RMQ_A(i, j)$** we can use the following formula:

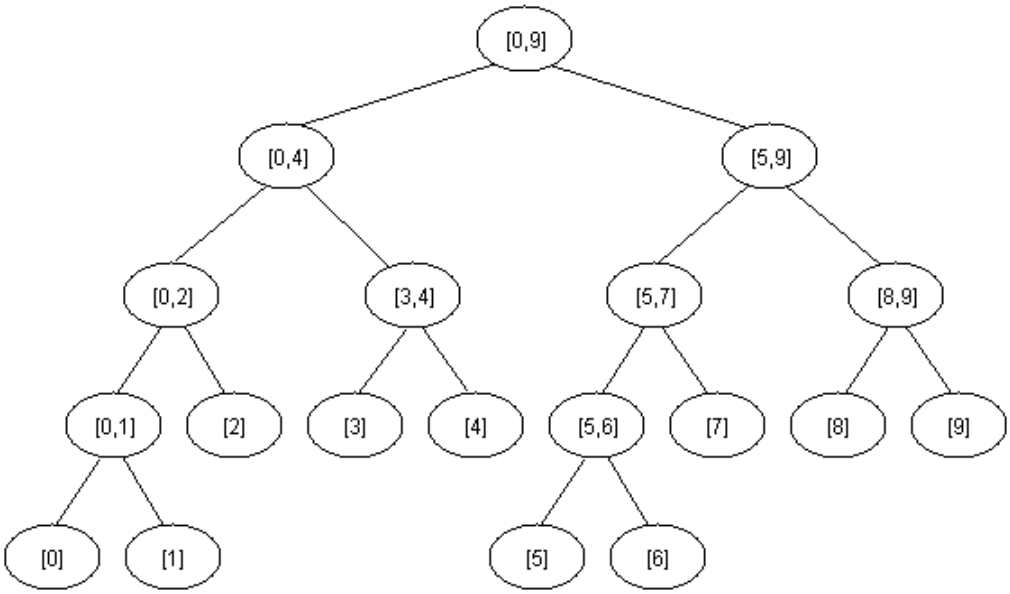
5.1.1.4 Segment trees

For solving the RMQ problem we can also use segment trees. A segment tree is a heap-like data structure that can be used for making update/query operations upon array intervals in logarithmical time. We define the segment tree for the interval **[i, j]** in the following recursive manner:

the first node will hold the information for the interval **[i, j]**

if i<j the left and right son will hold the information for the intervals **[i, (i+j)/2]** and **[(i+j)/2+1, j]**

Notice that the height of a segment tree for an interval with **N** elements is **[logN] + 1**. Here is how a segment tree for the interval **[0, 9]** would look like:



For solving the RMQ problem using segment trees we should use an array **M[1, 2 * $2^{[\log N]+1}$]** where **M[i]** holds the minimum value position in the interval assigned to node **i**. At the beginning all elements in **M** should be **-1**. The tree should be initialized with the following function (**b** and **e** are the bounds of the current interval):

```
void initialize(intnode, int b, int e, int M[MAXIND], int A[MAXN], int N)
{
    if (b == e)
        M[node] = b;
    else
    {
  //compute the values in the left and right subtrees
        initialize(2 * node, b, (b + e) / 2, M, A, N);
        initialize(2 * node + 1, (b + e) / 2 + 1, e, M, A, N);
  //search for the minimum value in the first and
  //second half of the interval
        if (A[M[2 * node]] <= A[M[2 * node + 1]])
            M[node] = M[2 * node];
        else
            M[node] = M[2 * node + 1];
      }
}
```

The function above reflects the way the tree is constructed. When calculating the minimum position for some interval we should look at the values of the sons. You should call the function with **node = 1**, **b = 0** and **e = N-1**.

We can now start making queries. If we want to find the position of the minimum value in some interval **[i, j]** we should use the next easy function:

```
int query(int node, int b, int e, int M[MAXIND], int A[MAXN], int i, int j)
{
    int p1, p2;
```

```
        //if the current interval doesn't intersect
    //the query interval return -1
        if (i > e || j < b)        return -1;
        //if the current interval is included in
    //the query interval return M[node]
        if (b >= i && e <= j)      return M[node];

    //compute the minimum position in the
    //left and right part of the interval
        p1 = query(2 * node, b, (b + e) / 2, M, A, i, j);
        p2 = query(2 * node + 1, (b + e) / 2 + 1, e, M, A, i, j);

    //return the position where the overall
    //minimum is
        if (p1 == -1)    return M[node] = p2;
        if (p2 == -1)    return M[node] = p1;
        if (A[p1] <= A[p2])return M[node] = p1;
        return M[node] = p2;
}
```

You should call this function with **node = 1**, **b = 0** and **e = N - 1**, because the interval assigned to the first node is **[0, N-1]**.

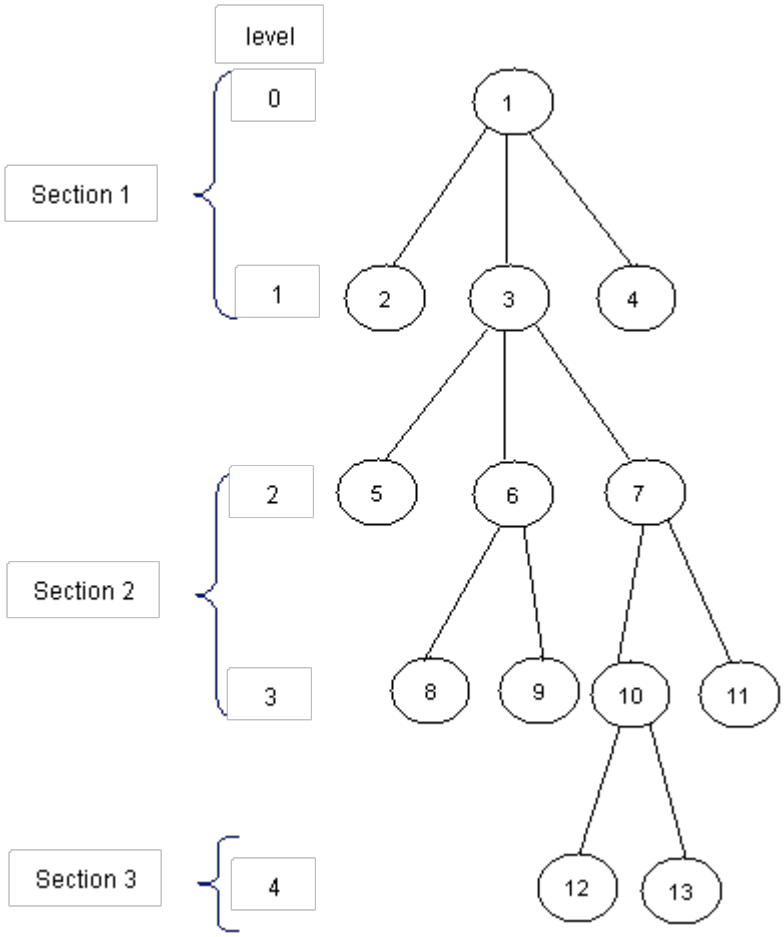
It's easy to see that any query is done in **O(log N)**. Notice that we stop when we reach completely in/out intervals, so our path in the tree should split only one time.

Using segment trees we get an **<O(N), O(logN)>** algorithm. Segment trees are very powerful, not only because they can be used for RMQ. They are a very flexible data structure, can solve even the dynamic version of RMQ problem, and have numerous applications in range searching problems.

5.1.2 Lowest Common Ancestor (LCA)

5.1.2.1 An <O(N), O(sqrt(N))> solution

Dividing our input into equal-sized parts proves to be an interesting way to solve the RMQ problem. This method can be adapted for the LCA problem as well. The idea is to split the tree in **sqrt(H)** parts, were **H** is the height of the tree. Thus, the first section will contain the levels numbered from **0 to sqrt(H) - 1**, the second will contain the levels numbered from **sqrt(H) to 2 * sqrt(H) - 1**, and so on. Here is how the tree in the example should be divided:



Now, for each node, we should know the ancestor that is situated on the last level of the upper next section. We will preprocess this values in an array **P[1, MAXN]**. Here is how **P** should look like for the tree in the example (for simplity, for every node **i** in the first section let **P[i] = 1**):

P[1]	P[2]	P[3]	P[4]	P[5]	P[6]	P[7]	P[8]	P[9]	P[10]	P[11]	P[12]	P[13]
1	1	1	1	3	3	3	3	3	3	3	10	10

Notice that for the nodes situated on the levels that are the first ones in some sections, **P[i] = T[i]**. We can preprocess **P** using a depth first search (**T[i]** is the father of node **i** in the tree, **nr** is **[sqrt(H)]** and **L[i]** is the level of the node **i**):

```
void dfs(int node, int T[MAXN], int N, int P[MAXN], int L[MAXN], int nr)
{
      int k;
   //if node is situated in the first
   //section then P[node] = 1
   //if node is situated at the beginning
   //of some section then P[node] = T[node]
   //if none of those two cases occurs, then
   //P[node] = P[T[node]]
      if (L[node] < nr)     P[node] = 1;
      else  if(!(L[node] % nr)) P[node] = T[node];
```

```
    else  P[node] = P[T[node]];
    for each son k of node    dfs(k, T, N, P, L, nr);
}
```

Now, we can easily make queries. For finding **LCA(x, y)** we we will first find in what section it lays, and then trivially compute it. Here is the code:

```
int LCA(int T[MAXN], int P[MAXN], int L[MAXN], int x, int y)
{
  //as long as the node in the next section of
  //x and y is not one common ancestor
  //we get the node situated on the smaller
  //lever closer
    while (P[x] != P[y])
        if (L[x] > L[y])      x = P[x];
        else  y = P[y];
  //now they are in the same section, so we trivially compute the LCA
    while (x != y)
        if (L[x] > L[y])      x = T[x];
        else  y = T[y];
    return x;
}
```

5.1.2.2 Another easy solution in <O(N logN, O(logN)>

If we need a faster solution for this problem we could use dynamic programming. First, let's compute a table P[1,N][1,logN] where P[i][j] is the 2^j-th ancestor of i. For computing this value we may use the following recursion:

The preprocessing function should look like this:

```
void process3(int N, int T[MAXN], int P[MAXN][LOGMAXN])
{
    int i, j;
  //we initialize every element in P with -1
    for (i = 0; i < N; i++)
        for (j = 0; 1 << j < N; j++)
            P[i][j] = -1;
  //the first ancestor of every node i is T[i]
    for (i = 0; i < N; i++)      P[i][0] = T[i];
  //bottom up dynamic programing
    for (j = 1; 1 << j < N; j++)
        for (i = 0; i < N; i++)
            if (P[i][j - 1] != -1)  P[i][j] = P[P[i][j - 1]][j - 1];
```

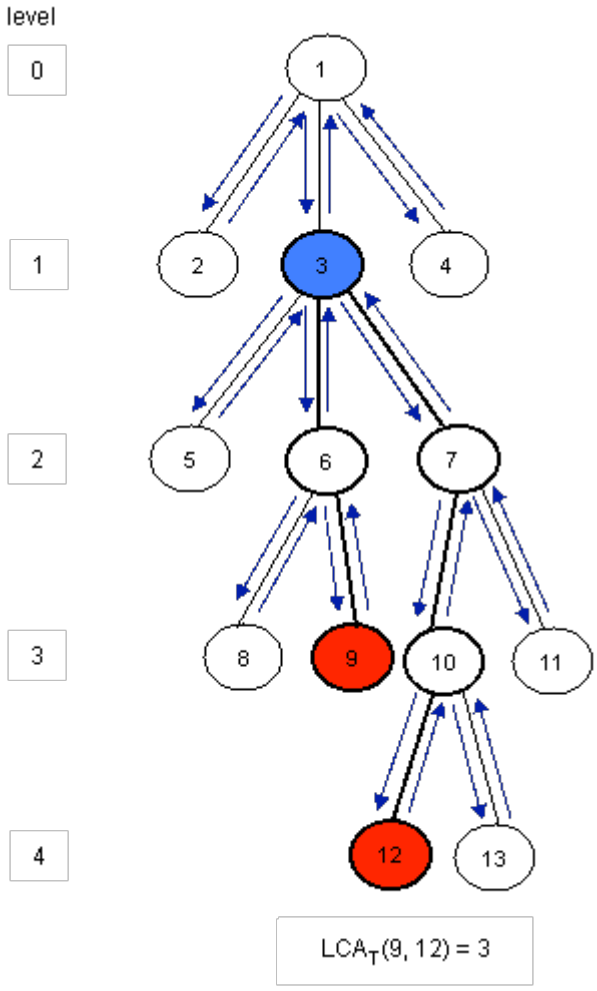
```
}
```

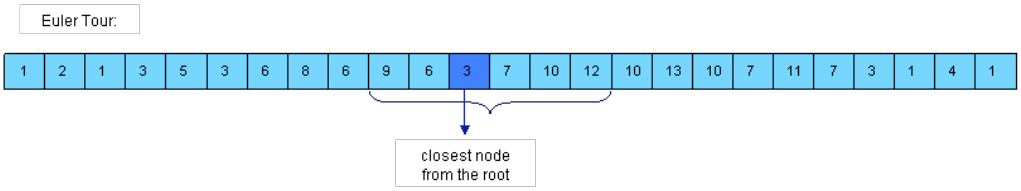
This takes **O(N logN)** time and space. Now let's see how we can make queries. Let **L[i]** be the level of node **i** in the tree. We must observe that if **p** and **q** are on the same level in the tree we can compute **LCA(p, q)** using a meta-binary search. So, for every power **j** of **2 (**between **log(L[p])** and **0,** in descending order), if **P[p][j] != P[q][j]** then we know that **LCA(p, q)** is on a higher level and we will continue searching for **LCA(p = P[p][j], q = P[q][j])**. At the end, both **p** and **q** will have the same father, so return **T[p]**. Let's see what happens if **L[p] != L[q]**. Assume, without loss of generality, that **L[p] < L[q]**. We can use the same meta-binary search for finding the ancestor of **p** situated on the same level with **q**, and then we can compute the **LCA** as described below. Here is how the query function should look:

```
int query(int N, int P[MAXN][LOGMAXN], int T[MAXN],
   int L[MAXN], int p, int q)
{
      int tmp, log, i;
   //if p is situated on a higher level than q then we swap them
      if (L[p] < L[q])       tmp = p, p = q, q = tmp;
   //we compute the value of [log(L[p)]
      for (log = 1; 1 << log <= L[p]; log++);
      log--;
   //we find the ancestor of node p situated on the same level
   //with q using the values in P
      for (i = log; i >= 0; i--)
            if (L[p] - (1 << i) >= L[q])      p = P[p][i];
      if (p == q)       return p;
   //we compute LCA(p, q) using the values in P
      for (i = log; i >= 0; i--)
            if (P[p][i] != -1 && P[p][i] != P[q][i])   p = P[p][i], q = P[q][i];
      return T[p];
}
```

5.1.3 Reduction from LCA to RMQ

Now, let's show how we can use RMQ for computing LCA queries. Actually, we will reduce the LCA problem to RMQ in linear time, so every algorithm that solves the RMQ problem will solve the LCA problem too. Let's show how this reduction can be done using an example:





Notice that **LCA_T(u, v)** is the closest node from the root encountered between the visits of **u** and **v** during a depth first search of **T**. So, we can consider all nodes between any two indices of **u** and **v** in the Euler Tour of the tree and then find the node situated on the smallest level between them. For this, we must build three arrays:

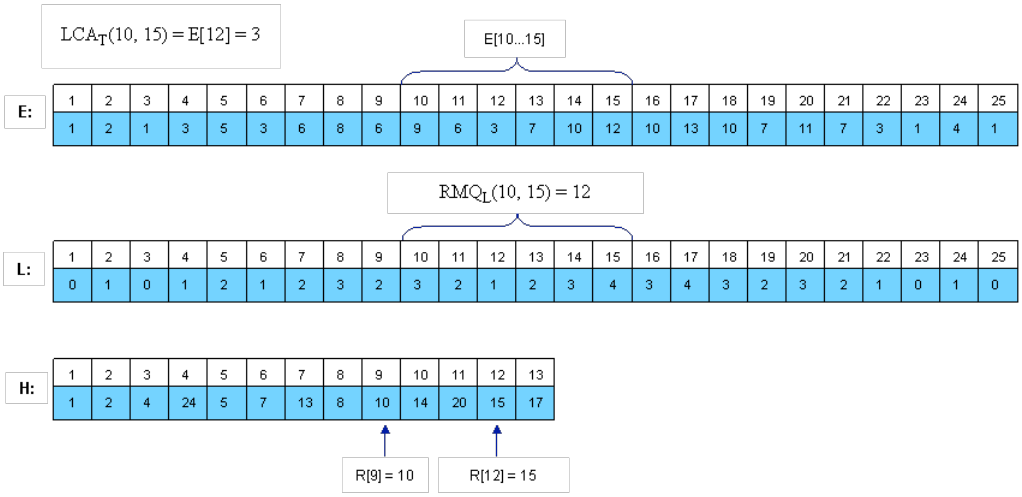
E[1, 2*N-1] - the nodes visited in an Euler Tour of **T**; **E[i]** is the label of **i-th** visited node in the tour

L[1, 2*N-1] - the levels of the nodes visited in the Euler Tour; **L[i]** is the level of node **E[i]**

H[1, N] - H[i] is the index of the first occurrence of node **i** in **E** (any occurrence would be good, so it's not bad if we consider the first one)

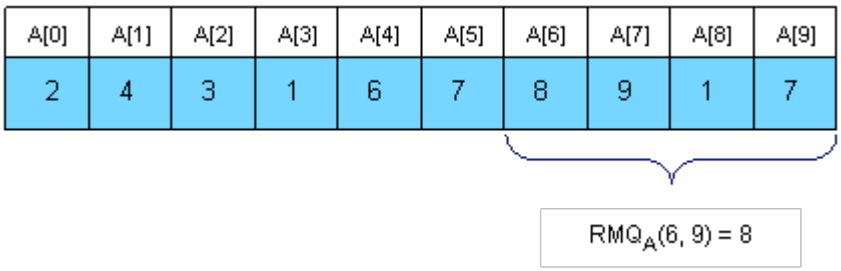
Assume that **H[u] < H[v]** (otherwise you must swap **u** and **v**). It's easy to see that the nodes between the first occurrence of **u** and the first occurrence of **v** are **E[H[u]...H[v]]**.

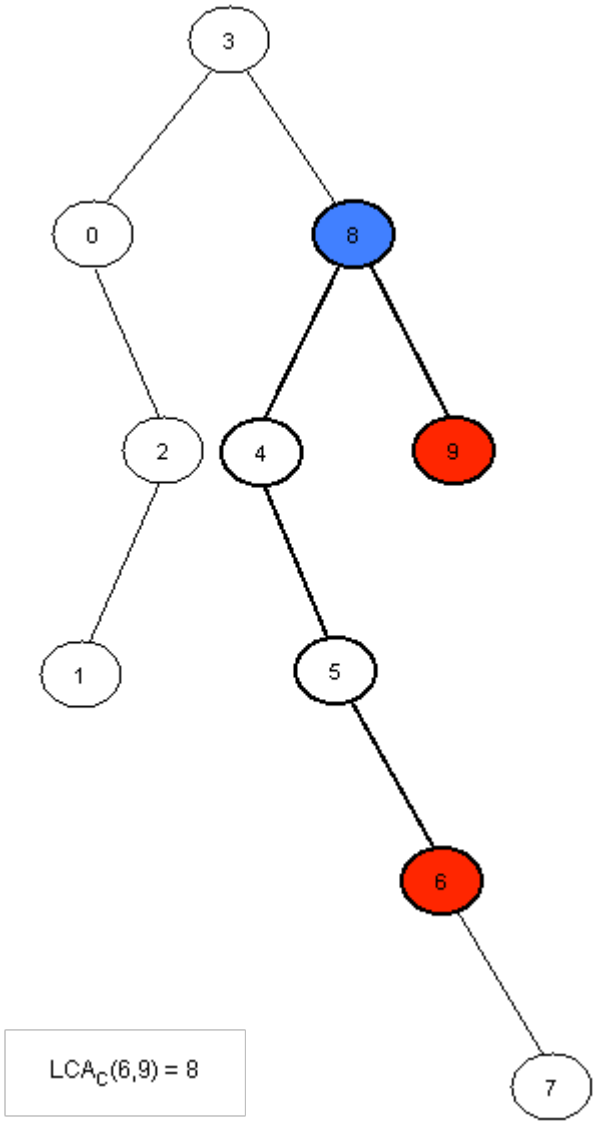
Now, we must find the node situated on the smallest level. For this, we can use **RMQ**. So, **$LCA_T(u, v) = E[RMQ_L(H[u], H[v])]$** (remember that RMQ returns the index). Here is how **E**, **L** and **H** should look for the example:



5.1.4 From RMQ to LCA

A Cartesian Tree of an array **A[0, N - 1]** is a binary tree **C(A)** whose root is a minimum element of **A**, labeled with the position **i** of this minimum. The left child of the root is the Cartesian Tree of **A[0, i - 1]** if **i > 0**, otherwise there's no child. The right child is defined similary for **A[i + 1, N - 1]**. Note that the Cartesian Tree is not necessarily unique if **A** contains equal elements. In this tutorial the first appearance of the minimum value will be used, thus the Cartesian Tree will be unique. It's easy to see now that **$RMQ_A(i, j) = LCA_C(i, j)$**.
Here is an example:





Now we only have to compute **C(A)** in linear time. This can be done using a stack. At the beginning the stack is empty. We will then insert the elements of **A** in the stack. At the **i-th** step **A[i]** will be added next to the last element in the stack that has a smaller or equal value to **A[i]**, and all the greater elements will be removed. The element that was in the stack on the position of **A[i]** before the insertion was done will become the left son of **i**, and **A[i]** will become the right son of the smaller element behind him. At every step the first element in the stack is the root of the cartesian tree. It's easier to build the tree if the stack will hold the indexes of the elements, and not their value.
Here is how the stack will look at each step for the example above:

Step	Stack	Modifications made in the tree
0	0	0 is the only node in the tree.

1	0 1	1 is added at the end of the stack. Now, 1 is the right son of 0.
2	0 2	2 is added next to 0, and 1 is removed (A[2] < A[1]). Now, 2 is the right son of 0 and the left son of 2 is 1.
3	3	A[3] is the smallest element in the vector so far, so all elements in the stack will be removed and 3 will become the root of the tree. The left child of 3 is 0.
4	3 4	4 is added next to 3, and the right son of 3 is 4.
5	3 4 5	5 is added next to 4, and the right son of 4 is 5.
6	3 4 5 6	6 is added next to 5, and the right son of 5 is 6.
7	3 4 5 6 7	7 is added next to 6, and the right son of 6 is 7.
8	3 8	8 is added next to 3, and all greater elements are removed. 8 is now the right child of 3 and the left child of 8 is 4.
9	3 8 9	9 is added next to 8, and the right son of 8 is 9.

Note that every element in **A** is only added once and removed at most once, so the complexity of this algorithm is **O(N)**. Here is how the tree-processing function will look:

```
void computeTree(int A[MAXN], int N, int T[MAXN])
{
    int st[MAXN], i, k, top = -1;
  //we start with an empty stack
  //at step i we insert A[i] in the stack
```

```
    for (i = 0; i < N; i++)
    {
  //compute the position of the first element that is
  //equal or smaller than A[i]
        k = top;
        while (k >= 0 && A[st[k]] > A[i]) k--;
  //we modify the tree as explained above
        if (k != -1)      [i] = st[k];
        if (k < top)      T[st[k + 1]] = i;
  //we insert A[i] in the stack and remove
  //any bigger elements
        st[++k] = i;
        top = k;
    }
  //the first element in the stack is the root of
  //the tree, so it has no father
    T[st[0]] = -1;
}
```

5.1.5 An<O(N), O(1)> algorithm for the restricted RMQ

Now we know that the general RMQ problem can be reduced to the restricted version using LCA. Here, consecutive elements in the array differ by exactly 1. We can use this and give a fast **<O(N), O(1)>** algorithm. From now we will solve the RMQ problem for an array **A[0, N - 1]** where **|A[i] - A[i + 1]| = 1**, **i = [1, N - 1]**. We transform **A** in a binary array with **N-1** elements, where **A[i] = A[i] - A[i + 1]**. It's obvious that elements in **A** can be just **+1** or **-1**. Notice that the old value of **A[i]** is now the sum of **A[1]**, **A[2]** .. **A[i]** plus the old **A[0]**. However, we won't need the old values from now on.

To solve this restricted version of the problem we need to partition **A** into blocks of size **l = [(log N) / 2]**. Let **A'[i]** be the minimum value for the **i-th** block in **A** and **B[i]** be the position of this minimum value in **A**. Both **A** and **B** are **N/l** long. Now, we preprocess **A'** using the ST algorithm described in Section1. This will take **O(N/l * log(N/l)) = O(N)** time and space. After this preprocessing we can make queries that span over several blocks in **O(1)**. It remains now to show how the in-block queries can be made. Note that the length of a block is **l = [(log N) / 2]**, which is quite small. Also, note that **A** is a binary array. The total number of binary arrays of size **l** is 2^l**=sqrt(N)**. So, for each binary block of size **l** we need to lock up in a table **P** the value for RMQ between every pair of indices. This can be trivially computed in **O(sqrt(N)*l^2)=O(N**) time and space. To index table **P**, preprocess the type of each block in **A** and store it in array **T[1, N/l]**. The block type is a binary number obtained by replacing -1 with 0 and +1 with 1.

Now, to answer $\mathbf{RMQ_A(i, j)}$ we have two cases:

i and **j** are in the same block, so we use the value computed in **P** and **T**

i and **j** are in different blocks, so we compute three values: the minimum from **i** to the end of **i's** block using **P** and **T**, the minimum of all blocks between **i's** and j**'s** block using precomputed queries on **A'** and the minimum from the begining of **j's** block to **j**, again using **T** and **P**; finally return the position where the overall minimum is using the three values you just computed.

5.1.6 An AC programme

```
const int MAXN = 2001;
struct nod
{
    int i;
    struct nod * next;
}*adjlist[MAXN], *p, *q;
int L[MAXN], E[MAXN], H[MAXN], RMQ[MAXN][20];
int f[MAXN], LOG2[MAXN], flag[MAXN], co[MAXN];
int a, b, c, n, m, nn;

void input()
{
    int i, j, k, kk, p, qq;
    for(i = 0; i < nn; ++i)     f[i] = 0;
    for(i = 0; i < nn; ++i)     adjlist[i] = NULL;
    for(i = 1; i <= nn; ++i)
    {
        scanf("%d:(%d)", &j, &kk);
        for(p = 1;  p <= kk; ++p)
        {
            scanf("%d", &qq);
            f[qq-1] = 1;
            q = (struct nod *)malloc(sizeof(struct nod));
            q->i = j-1;
            q->next = adjlist[qq-1];
            adjlist[qq-1] = q;

            q = (struct nod *)malloc(sizeof(struct nod));
            q->i = qq-1;
            q->next = adjlist[j-1];
            adjlist[j-1] = q;
        }
    }
}

void dfs(int node, int l)
```

```
{
    struct nod *p;
    flag[node] = 1;
    E[n] = node;
    L[n] = l;
    H[node] = n;
    ++n;
    p = adjlist[node];
    while(p)
    {
        if(!flag[p->i])
        {
            dfs(p->i, l+1);
            E[n] = node;
            L[n] = l;
            ++n;
        }
        p = p->next;
    }
}

inline void computeLog()
{
    int i, j, down = 2, up = 4, uper;
    LOG2[1] = 0;
    LOG2[2] = 1;
    for(i = 1; up < MAXN; ++i)
    {
        up = 1<<(i+1);
        uper = MAXN>up?up:MAXN;
        for(j = down; j < uper; ++j)    LOG2[j] = i;
        down = up;
    }
}

void process2()
{
    int i, j, kk;
    for (i = 0; i < n; i++)   RMQ[i][1] = i;
    for (j = 2; (1 << (j-1)) < n; ++j)
        for (i = 0; i + (1 << (j-2)) < n; ++i)
        {
            kk = i + (1 << (j - 2));
            if (L[RMQ[i][j - 1]] < L[RMQ[kk][j - 1]]) RMQ[i][j] = RMQ[i][j - 1];
```

```
                else    RMQ[i][j] = RMQ[kk][j - 1];
            }
}

inline int query(int p, int q)
{
    int k, qq;
    if(q < p)
    {
        int tmp = q;
        q = p;
        p = tmp;
    }
    k = LOG2[q-p+1];
    qq = q-(1<<k)+1;
    ++k;
    if(L[RMQ[p][k]] < L[RMQ[qq][k]])    return RMQ[p][k];
    return    RMQ[qq][k];
}

void rel()
{
    int i;
    for(i = 0; i < n; ++i)
    {
        while(adjlist[i])
        {
            p = adjlist[i]->next;
            free(adjlist[i]);
            adjlist[i] = p;
        }
    }
}

int main()
{
    int i;
    computeLog();
    while(scanf("%d", &nn) == 1)
    {
        input();
        for(i = 0; i < nn; ++i)    flag[i] = 0;
        for(i = 0; i < nn; ++i)    if(!f[i]) break;
        n = 0;
```

```
        dfs(i, 0);
        process2();
        for(i = 1; i <= nn; ++i)    co[i] = 0;
        scanf("%d", &m);
        for(i = 1; i <= m; ++i)
        {
            scanf(" (%d,%d)", &a, &b);
            co[E[query(H[a-1], H[b-1])]+1]++;
        }
        for(i = 1; i <= nn; ++i)
            if(co[i])   printf("%d:%d\n", i, co[i]);
        rel();
    }
    return 0;
}
```

5.2 最长公共子序列 LCS

```
char a[201], b[201];//a b 串
char com[402], d[101][101];//d 存储解
int c[101][101];
int la, lb, f, l;

void lcsl()
{
    int i, j;
    for(i = 1; i <= la; ++i)    c[i][0] = 0;
    for(j = 1; j <= lb; ++j)    c[0][j] = 0;
    for(i = 1; i <= la; ++i)
        for(j = 1;   j <= lb; ++j)
            if(a[i] == b[j])        c[i][j] = c[i-1][j-1] + 1,d[i][j] = '1';
            else if(c[i-1][j] >= c[i][j-1])    c[i][j] = c[i-1][j],d[i][j] = '2';
            else  c[i][j] = c[i][j-1],d[i][j] = '3';
}

void LCS(int i, int j)
{
    int k;
    if(i == 0 || j == 0)    return;
    if(d[i][j] == '1')
    {
        LCS(i-1, j-1);
        for(k = f; k < i; ++k)      printf("%c", a[k]);
        for(k = l; k < j; ++k)      printf("%c", b[k]);
```

```
            f = i+1,l = j+1;
            printf("%c", a[i]);
        }
        else if(d[i][j] == '2')        LCS(i-1, j);
        else         LCS(i, j-1);
}

int main()
{
        int i, j, k, kk;
        while(scanf("%s %s", a+1, b+1) !=EOF)
        {
            la = strlen(a+1);
            lb = strlen(b+1);
            lcsl();
            f = 1, l = 1;
            LCS(la, lb);
            for(k = f; k <= la; ++k)   printf("%c", a[k]);
            for(k = l; k <= lb; ++k)   printf("%c", b[k]);
            printf("\n");
        }
        return 0;
}
```

5.3 最长上升子序列/最长不下降子序列(LIS)

5.3.1 O(n^2)

```
int pint[1001];
int sub[1001];//以 pint[i]结尾的 LIS

int main()
{
        int n,i,j;
        scanf("%d",&n);
        for(i=1;i<=n;i++)    scanf("%d",&pint[i]);
        sub[1]=0;
        for(i=2;i<=n;i++)
        {
            sub[i]=0;
            for(j=1;j<i;j++)
                if(pint[j]<pint[i])    if(sub[i]<sub[j]+1)  sub[i]=sub[j]+1;
        }
```

```
    j=sub[1];
    for(i=2;i<=n;i++)    if(j<sub[i]) j=sub[i];
    printf("%d\n",j+1);
    return 0;
}
```

5.3.2 O(nlogn)

设 A[t]表示序列中的第 t 个数，F[t]表示从 1 到 t 这一段中以 t 结尾的最长上升子序列的长度，初始时设 F[t] = 0(t = 1, 2, ..., len(A))。则有动态规划方程：F[t] = max{1, F[j] + 1} (j = 1, 2, ..., t - 1, 且 A[j] < A[t])。

现在，我们仔细考虑计算 F[t]时的情况。假设有两个元素 A[x]和 A[y]，满足

(1)x < y < t (2)A[x] < A[y] < A[t] (3)F[x] = F[y]

此时，选择 F[x]和选择 F[y]都可以得到同样的 F[t]值，那么，在最长上升子序列的这个位置中，应该选择 A[x]还是应该选择 A[y]呢？

很明显，选择 A[x]比选择 A[y]要好。因为由于条件(2)，在 A[x+1] ... A[t-1]这一段中，如果存在 A[z]，A[x] < A[z] < a[y]，则与选择 A[y]相比，将会得到更长的上升子序列。

再根据条件(3)，我们会得到一个启示：根据 F[]的值进行分类。对于 F[]的每一个取值 k，我们只需要保留满足 F[t] = k 的所有 A[t]中的最小值。设 D[k]记录这个值，即 D[k] = min{A[t]} (F[t] = k)。

注意到 D[]的两个特点：

(1) D[k]的值是在整个计算过程中是单调不上升的。

(2) D[]的值是有序的，即 D[1] < D[2] < D[3] < ... < D[n]。

利用 D[]，我们可以得到另外一种计算最长上升子序列长度的方法。设当前已经求出的最长上升子序列长度为 len。先判断 A[t]与 D[len]。若 A[t] > D[len]，则将 A[t]接在 D[len]后将得到一个更长的上升子序列，len = len + 1， D[len] = A[t]；否则，在 D[1]..D[len]中，找到最大的 j，满足 D[j] < A[t]。令 k = j + 1，则有 D[j] < A[t] <= D[k]，将 A[t]接在 D[j]后将得到一个更长的上升子序列，同时更新 D[k] = A[t]。最后，len 即为所要求的最长上升子序列的长度。

在上述算法中，若使用朴素的顺序查找在 D[1]..D[len]查找，由于共有 O(n)个元素需要计算，每次计算时的复杂度是 O(n)，则整个算法的时间复杂度为 O(n^2)，与原来的算法相比没有任何进步。但是由于 D[]的特点(2)，我们在 D[]中查找时，可以使用二分查找高效地完成，则整个算法的时间复杂度下降为 O(nlogn)，有了非常显著的提高。需要注意的是，D[]在算法结束后记录的并不是一个符合题意的最长上升子序列！

```
const int MAX = 100;                    // LIS(n*logn)
int a[MAX],pos[MAX],n;//序列 a（0 开始），
int binsearch(int a,int len)
{
    int lt,rt,mid;
    lt = 0; rt = len;
    while (lt <= rt)
    {
        mid = (lt + rt) / 2;
        if (a < pos[mid]) rt = mid - 1;
```

```
            else if (a > pos[mid]) lt = mid + 1;
            else return mid;
        }
        if (pos[mid] < a) ++mid;        //can be removed
        return mid;
}
int LIS()
{
        int i,len;
        pos[0] = a[0];
        len = 0;
        for (i = 1 ; i < n ; i++)
        {
            if (a[i] > pos[len])   pos[++len] = a[i];
            else  pos[binsearch(a[i],len)] = a[i];
        }
        return len;
}
int main(){printf("%d\n",LIS()+1);}
```

5.4 Joseph 问题

问题描述：n 个人（编号 0~(n-1)），从 0 开始报数，报到(m-1)的退出，剩下的人继续从 0 开始报数。求胜利者的编号。

我们知道第一个人(编号一定是 m%n-1) 出列之后，剩下的 n-1 个人组成了一个新的约瑟夫环（以编号为 k=m%n 的人开始）：

k k+1 k+2 ... n-2, n-1, 0, 1, 2, ... k-2

并且从 k 开始报 0。

现在我们把他们的编号做一下转换：

k --> 0

k+1 --> 1

k+2 --> 2

...

...

k-2 --> n-2

k-1 --> n-1

变换后就完完全全成为了(n-1)个人报数的子问题，假如我们知道这个子问题的解：例如 x 是最终的胜利者，那么根据上面这个表把这个 x 变回去不刚好就是 n 个人情况的解吗？!!

变回去的公式很简单，相信大家都可以推出来：x' =(x+k)%n

如何知道(n-1)个人报数的问题的解？对，只要知道(n-2)个人的解就行了。(n-2)个人的解呢？当然是先求(n-3)的情况 ---- 这显然就是一个倒推问题！好了，思路出来了，下面写递推公式：

令 f[i]表示 i 个人玩游戏报 m 退出最后胜利者的编号，最后的结果自然是 f[n]

递推公式
f[1]=0;
f[i]=(f[i-1]+m)%i; (i>1)
有了这个公式，我们要做的就是从 1-n 顺序算出 f[i]的数值，最后结果是 f[n]。因为实际生活中编号总是从 1 开始，我们输出 f[n]+1
由于是逐级递推，不需要保存每个 f[i]，程序也是异常简单：

```
int main()
{
    int n, m, i, s=0;
    printf ("N M = "); scanf("%d%d", &n, &m);
    for (i=2; i<=n; i++) s=(s+m)%i;
    printf ("The winner is %d\n", s+1);
}
```

5.5 0/1 背包问题

```
const int Max = 100001;
struct NOD
{int A, C;};
NOD nod[110];
int n, m;
int dp[Max], num[Max]/*所用的当前硬币数*/, id[Max]/*硬币号*/;

bool cmp(NOD p,NOD q)
{
    if(p.A == q.A) return p.C <= q.C;
    return p.A < q.A;
}

int main()
{
    while(scanf("%d%d",&n,&m) != EOF&&n)
    {
        int i, j;
        for(i = 0;i < n;i ++) scanf("%d",&nod[i].A);
        for(i = 0;i < n;i ++) scanf("%d",&nod[i].C);
        std::sort(nod,nod+n,cmp);
        memset(dp,0,(m+1)*sizeof(int));
        memset(id,-1,(m+1)*sizeof(int));
        dp[0] = 1;
        int max = 0;
        for(i = 0;i < n;i ++)
        {
            max += nod[i].A*nod[i].C;
```

```
                if(max > m) max = m;
                for(j = nod[i].A;j <= max;j ++)
                {
                    if(dp[j]) continue;
                    int tmp = j - nod[i].A;
                    if(!dp[tmp]) continue;
                    if(id[tmp] == i)
                    {
                        if(num[tmp] >= nod[i].C) continue;
                        num[j] = num[tmp] + 1;
                    }
                    else num[j] = 1;
                    id[j] = i;
                    dp[j] = 1;
                }
            }
            int ans = 0;
            for(i = 1;i <= m;i ++)
                if(dp[i])   ans ++;
            printf("%d\n",ans);
        }
        return 0;
}
```

6 组合数学相关

6.1 The Number of the Same BST

Let x be a node in a binary search tree. If y is a node in the left subtree of x, then key[y] <= key[x]. If y is a node in the right subtree of x, then key[y] > key[x];
Now given a vector X, then you may get a binary search tree from X. Your job is to calculate how many different vectors can build the same binary search tree. To make it easy, you should just output the number of different vectors mod 9901.
假设树中总的节点数是 n，以每一个节点为根的子树中包含的节点个数分别是 sum[1], sum[2], ..., sum[n]。则不同 BST 的个数是 n!/(sum[1] * sum[2] * ... * sum[n])。

```
#define MAX 300
int a[101],b[101];
int l[101], r[101];
int n, c;

int f(int i)
{
```

```
    int re;
    if(l[i] == -1 && r[i] == -1)    re = 1;
    else if(l[i] == -1)    re = f(r[i])+1;
    else if(r[i] == -1)    re = f(l[i])+1;
    else  re = f(l[i])+f(r[i])+1;
    int j, k;
    int m = re;
    while(m > 1)
    {
        for(j = 2; j < m; ++j)      if(m % j == 0) break;
        for(k = 1; k <= n; ++k)
            if(a[k] % j == 0)
            {
                a[k] /= j;
                break;
            }
        m /= j;
    }
    return re;
}

int main()
{
    while(scanf("%d ", &n) && n)
    {
        int i;
        for(i = 1; i <= n; ++i)     b[i] = l[i] = r[i] = -1;
        int node;
        scanf("%d", &b[1]);
        c = 1;
        for(i = 2; i <= n; ++i)
        {
            scanf("%d", &b[i]);
            int k = 1, kk;
            int lr;
            while(1)
            {
                if(b[i] <= b[k])
                {
                    kk = k;
                    k = l[k];
                    lr = 1;
                }
                else
```

```
                {
                    kk = k;
                    k = r[k];
                    lr = 2;
                }
                if(-1 == k)
                {
                    if(lr == 1)     l[kk] = i;
                    else  r[kk] = i;
                    break;
                }
            }
        }
        for(i = 1; i <= n; ++i)     a[i] = n+1-i;
        f(1);
        int ans = 1;
        for(i = 1; i <= n; ++i)     ans = (ans*a[i]) % 9901;
        printf("%d\n", ans);
    }
    return 0;
}
```

6.2 排列生成

以 12…n 为第一个排列，排序的规则，是由一个排列 $(p)=(p_1p_2\llcorner p_n)$ 直接生成下一个排列：

(1)求满足 $p_{k-1}<p_k$ 的k的最大值为i,

$$i=\max\{k\,|\,p_{k-1}<p_k\}$$

(2)求满足 $p_{i-1}<p_k$ 的k的最大值为j,

$$j=\max\{k\,|\,p_{i-1}<p_k\}$$

(3) p_{i-1}与p_i互位置得

$$(q)=(q_1q_2\llcorner\ q_n)$$

(4) $(q)=(q_1q_2\llcorner\ q_n)$中$q_iq_{i+1}\llcorner\ q_n$ 部分的顺序逆转

$$q_1q_2\llcorner\ q_{i-1}q_n\llcorner\ q_{i+1}q_i$$

6.3 逆序

6.3.1 归并排序求逆序

```
void mergeSort(long first, long last)
{
    if(first < last)
    {
        long mid = (first + last) / 2;
        mergeSort(first, mid);
        mergeSort(mid+1, last);
        merge(first, mid, last);
    }
}
void merge(long p, long q, long r)
{
    long i, j = 0;
    long beginA = p, endA = q, beginB = q+1, endB = r;
    while(beginA <= endA && beginB <= endB)
    {
        if(a[beginA] <= a[beginB])    b[j++] = a[beginA++];
        else
        {
            b[j++] = a[beginB++];
            change += q - beginA + 1;
        }
    }
    while(beginA <= endA)        b[j++] = a[beginA++];
    while(beginB <= endB)        b[j++] = a[beginB++];
    for(i = 0; i < j; i++)       a[p+i] = b[i];
}
```

7 数值分析

7.1 二分法

(1) f(a)*f(b)<0;
(2) r=(a+b)/2,f(r);
(3) f(a)*f(b)>0, 则 a=r,否则 b=r;
(4) b-a<EPS,结束.

```
#define d 10e-4
```

```
int main()
{
    double x,y,c,l,a,b,h;
    while(scanf("%lf %lf %lf",&x,&y,&c)!=EOF)
    {
        h=(y*y-x*x)/(c*c);
        a=b=2;
        while(a*a-a*a/((a-1)*(a-1))-h>0)
        {b=a;a-=d;}
        if(b<2) ;
        else while(b*b-b*b/((b-1)*(b-1))-h<0)  b+=2;
        l=(a+b)/2;
        while(fabs(l*l-l*l/((l-1)*(l-1))-h)>10e-6)
        {
            if((a*a-a*a/((a-1)*(a-1))-h)*(l*l-l*l/((l-1)*(l-1))-h)>0) a=l;
            else b=l;
            l=(a+b)/2;
        }
        l=sqrt(y*y-c*c*l*l);
        printf("%.3f\n",l);
    }
    return 0;
}
```

7.2 迭代法(x=f(x))

f(x)在[a,b]上可微，满足：|f'(x)|<1,对 x0∈[a,b]，x0=f(x0)迭代收敛。

```
const double PI = acos(-1.0);
const double EPS = 5e-9;

int main()
{
    double l, n, c, l2;
    while (scanf("%lf%lf%lf", &l, &n, &c)!=EOF)
    {
        if (l==-1 && n==-1 && c==-1) break;
        l2 = l*(1+n*c);
        if(l==0 || fabs(l2-l) < EPS) {
            printf("0.000\n");
            continue;
        }
        double alpha =sqrt(6-6*l/l2), alpha2, h;
        do {
```

```
            alpha2 = l2*sin(alpha)/l;
            alpha = alpha2;
//          printf("d: %.16lf\n", fabs(l*alpha2-l2*sin(alpha)));
        } while ( fabs(l*alpha2-l2*sin(alpha)) > EPS );
        if (fabs(alpha) < EPS) h = 0.0;
        else h = 0.5*l2*(1-cos(alpha))/alpha;
        printf("%.3lf\n", h);
    }
}
```

7.3 牛顿迭代

x=x-f(x)/f’(x);
f(x)在[a,b]上二阶可导，满足：(1)f(a)f(b)<0;(2)f’(x)!=0(3) f’’(x)不变号；(4)初值 x∈[a,b]，f(x) f’’(x)>0, 迭代收敛。

```
int main()
{
    double l,n,c,h,a,b;

    while(scanf("%lf %lf %lf",&l,&n,&c)!=EOF)
    {
        //if(l==-1&&n==-1&&c==-1) break;
        if(l<0&&n<0&&c<0) break;
        b=(1+n*c)*l;
        if(n==0||fabs(b-l)<10e-11||l==0) printf("0.000\n");
        else
        {
            c=1+n*c;
            a=1;
            while(fabs((c*sin(a)-a)/(c*cos(a)-1))>10e-11)
                a-=(c*sin(a)-a)/(c*cos(a)-1);
            //if(fabs(a)<10e-12) h=0.0;       可要可不要
            //else
                h=l*c*(1-cos(a))/a/2;
            printf("%.3f\n",h);
        }
    }
    return 0;
}
```

7.4 数值积分

```
//利用辛普森变步长公式求积分
#define pi 3.14159265358979323846264338327950
```

```
const double eps=1e-6;
int a,b,h,t;

double fun(double x)
{return sqrt(a*sin(x)*sin(x)+b*cos(x)*cos(x));}

int main()
{
	long n;
	double start , end,height,s,tt;
	start=0;
	end=pi/2;
	while(scanf("%d %d %d",&h,&a,&b)!=EOF)
	{
		if(a<b) {t=a;a=b;b=t;}
		a*=a,b*=b;
		height=end-start;
		n=1;
		s=height*(fun(start)+fun(end))/2;
		tt=s+eps+1;
		while(fabs(s-tt)>=eps)
		{
			tt=s;
			s=0;
			for(int k=0;k<n;k++)	s+=fun((k+0.5)*height);
			s=tt/2+height*s/2;
			n*=2;
			height/=2;
		}
		printf("%.2f\n",4*s*h);
	}
	return 0;
}
```

7.5 高斯消元

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <math.h>
int GS(int,double **,double *,double);
double **TwoArrayAlloc(int ,int);
void TwoArrayFree(double **);
```

```
void main()
{
    int i,n;
    double ep,**a,*b;
    n=3;
    ep=1e-4;
    a=TwoArrayAlloc(n,n);
    b=(double *)calloc(n,sizeof(double));
    if(b==NULL)
    {
        printf("内存分配失败\n");
        exit(1);
    }
    a[0][0]=3;a[0][1]=2;a[0][2]=1;
    a[1][0]=1;a[1][1]=5;a[1][2]=7;
    a[2][0]=5;a[2][1]=4;a[2][2]=3;
    b[0]=6;b[1]=12;b[2]=12;
    if(!GS(n,a,b,ep))
    {
        printf("不可以用高斯消元法求解\n");
        exit(0);
    }
    printf("该方程组的解为: \n");
    for(i=0;i<3;i++)    printf("x%d = %.2f\n",i,b[i]);
    TwoArrayFree(a);
    free(b);
}
int GS(int n,double **a,double *b,double ep)
 {
    int i,j,k,l;
    double t;
    for(k=1;k<=n;k++)
    {
        for(l=k;l<=n;l++)
            if(fabs(a[l-1][k-1])>ep)
                break;
            else if(l==n)
                return 0;
            if(l!=k)
            {
                for(j=k;j<=n;j++)
                {
                    t=a[k-1][j-1];
                    a[k-1][j-1]=a[l-1][j-1];
```

```
                    a[l-1][j-1]=t;
                }
                t=b[k-1];
                b[k-1]=b[l-1];
                b[l-1]=t;
            }
            t=1/a[k-1][k-1];
            for(j=k+1;j<=n;j++)
                a[k-1][j-1]=t*a[k-1][j-1];
            b[k-1]*=t;
            for(i=k+1;i<=n;i++)
            {
                for(j=k+1;j<=n;j++)
                    a[i-1][j-1]-=a[i-1][k-1]*a[k-1][j-1];
                b[i-1]-=a[i-1][k-1]*b[k-1];
            }
    }
    for(i=n-1;i>=1;i--)
        for(j=i+1;j<=n;j++)
            b[i-1]-=a[i-1][j-1]*b[j-1];
    return(1);
}
double **TwoArrayAlloc(int r,int c)
{
    double *x,**y;
    int n;
    x=(double *)calloc(r*c,sizeof(double));
    y=(double **)calloc(r,sizeof(double*));
    for(n=0;n<=r-1;++n)
    y[n]=&x[c*n];
    return(y);
}
void TwoArrayFree(double **x)
{
    free(x[0]);free(x);
}
```

8 其它

递归式：

公式一: T(1)=1, T(n)=T(n-1)+n, 则 T(n)为 n(n+1)/2

公式二: T(1)=1, T(n)=T(n/2)+1, 则 T(n)约为 lgN

公式三: T(1)=0, T(n)=T(n/2)+n, 则 T(n)约为 2n

公式四: T(1)=0, T(n)=2T(n/2)+n, 则 T(n)约为 nlgN
公式五: T(1)=1, T(n)=2T(n/2)+1, 则 T(n)约为 2n

欧拉常数 γ = 0.57721566490153286060651209
调和级数 S=1+1/2+1/3+……Sn=ln(n)+ γ

斐波那契数列：(1/√5)*{[(1+√5)/2]^n - [(1-√5)/2]^n}【√5 表示根号 5】

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \underbrace{\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdots \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}}_{n \text{ times}}.$$

史特林公式：
n!约等于(n/e)^n * sqrt(2*pi*n)
n！=（2*pi*n)^1/2 *(n/e)^n *e^(a/12*n)

```
int m;
char s[mm][61];

int cmp(const void *c, const void *d)
{
    char *e = (char *)c, *f = (char *)d;
    return strcmp(e, f);
}
qsort(s, m, 61, cmp);
```