ACM 模板

dnvtmf

2015

目录

1 数据结构

1.1 RMQ 相关

```
/*区间的rmq问题
 * 在一维数轴上，添加或删除若干区间[l,r]， 询问某区间[ql, qr]内覆盖了多少个完整的区间
 * 做法：离线，按照右端点排序，然后按照左端点建立线段树保存左端点为l的区间个数，
 *       接着按排序结果从小到大依次操作，遇到询问时，查询比ql大的区间数
 * 遇到不能改变查询顺序的题，应该用可持久化线段树
 */
/*数组区间颜色数查询
    问题：给定一个数组,要求查询某段区间内有多少种数字
    解决：将查询离线，按右端点排序；从左到右依次扫描，扫描到第i个位置时，将该位置加1，
    该位置的前驱(上一个出现一样数字的位置)减1，然后查询所有右端点为i的询问的一个区间和[l, r].
*/
```

1.2 树链剖分

```
///树链剖分 Heavy-Light Decomposition
//将一个树划分为若干个不相交的路径，使每个结点仅在一条路径上.
//满足：从结点u到v最多经过log N条路径，以及log N条不在路径上的边.
//采用启发式划分，即某结点选择与子树中结点数最大的儿子划分为一条路径.
//时间复杂度：用其他数据结构来维护每条链，复杂度为所选数据结构乘以log N.
//用split()来进行树链剖分，其中使用bfs进行划分操作. 对于每一个结点v，找到它的size最大的子结点u.
    如果u不存在，那么给v分配一条新的路径，否则v就延续u所属的路径.
//查询两个结点u，v之间的路径是，首先判断它们是否属于同一路径. 如果不是，
    选择所属路径顶端结点h的深度较大的结点，不妨假设是v，查询v到h，并令v = father[h]继续查询，直至u，
    v属于同一路径. 最后在这条路径上查询并返回.
/*
sz[u]: 结点u的子树的结点数量
fa[u]: 结点u的父结点
dep[u]: 结点u在树中的深度
belong[u]: 结点u所在剖分链的编号
id[u]: 结点u在其路径中的编号，  由深入浅编号
start[p]: 链p的第一个结点
len[p]: 链p的长度
total: 划分链的数量
dfn: 树中结点的遍历顺序
*/
struct edge
{
    int next, to, cost;
} e[NUM << 1];
int head[NUM], tot;
void init()
{
    memset(head, -1, sizeof(head));
    tot = 0;
}
void add_edge(int u, int v, int w)
{
    e[tot] = (edge) {head[u], v, w};
    head[u] = tot++;
}
int sz[NUM], fa[NUM], dep[NUM], start[NUM], belong[NUM], id[NUM], len[NUM], total;
int dfn[NUM];
bool vis[NUM];
void split()
{
    int tail = 0,  top = 0;
```

```
    dep[dfn[top++] = 1] = 0;
    fa[1] = 0;
    ne[1] = 0;
    val[0] = 0;
    while(tail < top)
    {
        int u = dfn[tail++];
        for(int i = head[u]; ~i; i = e[i].next)
            if(e[i].to != fa[u])
            {
                dep[dfn[top++] = e[i].to] = dep[u] + 1;
                fa[e[i].to] = u;
            }
    }
    memset(vis, 0, sizeof(vis));
    total = 0;
    while(--top >= 0)
    {
        int u = dfn[top], v = -1;
        sz[u] = 1;
        for(int i = head[u]; ~i; i = e[i].next)
            if(vis[e[i].to])
            {
                sz[u] += sz[e[i].to];
                if(v == -1 || sz[e[i].to] > sz[v])
                    v = e[i].to;
            }
        if(v == -1)
        {
            id[u] = len[++total] = 1;
            belong[start[total] = u] = total;
        }
        else
        {
            id[u] = ++len[belong[u] = belong[v]];
            start[belong[u]] = u;
        }
        vis[u] = true;
    }
}
```

1.3 伸展树 Splay

```
///伸展树Splay
//均摊复杂度O(log n) 最坏复杂度O(n)
/*操作
1. 旋转: 左右旋, 将左（右）孩子变为根
2. spaly: 将结点x旋转至根y处
3. find: 同二叉树查找, 查找成功后splay
4. remove: 查找x, 如果x无孩子或一个孩子, 删除x, splay（x）的根结点; x有两个孩子, 用x的后继y代替x,
    splay（y)
5. join, split合并, 分解数.
6. 区间操作: [a, b], 将a-1 splay至根处, b+1至根的右孩子处, 那根的右孩子的左子树表示区间[a, b]
*/
const int NUM = 1000000 + 10;
struct Splay_node
{
    int ch[2], fa;
    int key;
    int len;
    bool flag;
    LL val, sum;
```

```
    void malloc(int _key)
    {
        ch[0] = ch[1] = fa = 0;
        key = _key;
        len = 1;
        flag = false;
        val = sum = 0;
    }
};
struct Splay
{
    Splay_node *e;
    int tot;
    int root;
    Splay()
    {
        e = new Splay_node[NUM];
        tot = 0;
        root = -1;
    }
    ~Splay()
    {
        delete []e;
    }
    void push_down(int x)//标记下传
    {
        if(e[x].flag)
        {
            int y = e[x].ch[0];
            if(y)
            {
                e[y].flag = true;
                e[y].val = e[x].val;
                e[y].sum = e[y].val * e[y].len;
            }
            y = e[x].ch[1];
            if(y)
            {
                e[y].flag = true;
                e[y].val = e[x].val;
                e[y].sum = e[y].val * e[y].len;
            }
            e[x].flag = false;
        }
    }
    void push_up(int x)//标记上传
    {
        e[x].sum = e[x].val;
        e[x].len = 1;
        if(e[x].ch[0])
        {
            e[x].sum += e[e[x].ch[0]].sum;
            e[x].len += e[e[x].ch[0]].len;
        }
        if(e[x].ch[1])
        {
            e[x].sum += e[e[x].ch[1]].sum;
            e[x].len += e[e[x].ch[1]].len;
        }
    }
    //旋转操作: c = 0: 左旋, 将父节点旋转到结点x的左儿子; c = 1: 右旋, 将父节点旋转到结点x的右儿子
    void rot(int x, int c)
    {
        int y = e[x].fa, z = e[y].fa;
```

```
        push_down(y), push_down(x);
        e[y].ch[!c] = e[x].ch[c];
        if(e[x].ch[c]) e[e[x].ch[c]].fa = y;
        e[y].fa = x;
        if(z) e[z].ch[e[z].ch[1] == y] = x;
        e[x].ch[c] = y;
        e[x].fa = z;
        push_up(y);
        if(y == root) root = x;
    }
    //splay操作: 将结点x旋转到结点fa下面
    void splay(int x, int fa)
    {
        int y, z;
        push_down(x);
        while((y = e[x].fa) != fa)
        {
            z = e[y].fa;
            if(z == fa) rot(x, e[y].ch[0] == x);//单旋
            else
            {
                if(e[z].ch[0] == y)
                {
                    if(e[y].ch[0] == x) rot(y, 1), rot(x, 1);//一字旋
                    else rot(x, 0), rot(x, 1);//之字旋
                }
                else
                {
                    if(e[y].ch[1] == x) rot(y, 0), rot(x, 0);//一字旋
                    else rot(x, 1), rot(x, 0);//之字旋
                }
            }
        }
        push_up(x);
    }
    void insert(int key)
    {
        int x = ++tot;
        e[x].malloc(key);
        int y = root;
        if(y == -1)
        {
            root = x;
            e[x].fa = 0;
            return ;
        }
        while(true)
        {
            if(e[y].key < key)
            {
                if(e[y].ch[1]) y = e[y].ch[1];
                else
                {
                    e[y].ch[1] = x;
                    e[x].fa = y;
                    splay(x, 0);
                    return ;
                }
            }
            else if(e[y].key > key)
            {
                if(e[y].ch[0]) y = e[y].ch[0];
                else
                {
```

```
                    e[y].ch[0] = x;
                    e[x].fa = y;
                    splay(x, 0);
                    return ;
                }
            }
            else
                return ;
        }
    }
    int find(int key)
    {
        int x = root;
        while(e[x].key != key)
        {
            x = e[x].ch[e[x].key < key];
            if(!x)//没有找到
                return -1;
        }
        return x;
    }
    void update(int L, int R, LL val)
    {
        splay(find(L - 1), 0);
        splay(find(R + 1), root);
        int x = e[e[root].ch[1]].ch[0];
        e[x].flag = true;
        e[x].val = val;
        e[x].sum = val * e[x].len;
    }
    LL query(int L, int R)
    {
        splay(find(L - 1), 0);
        splay(find(R + 1), root);
        int x = e[e[root].ch[1]].ch[0];
        return e[x].sum;
    }
};
```

1.4 Treap

```
//随机化二叉树Treap
struct node
{
    int key;
    int ch[2], fix;
};
struct Treap
{
    node *p;
    int size, root;
    Treap()
    {
        p = new node[NUM];
        srand(time(0));
        size = -1;
        root = -1;
    }
    ~Treap() {delete []p;}
    void rot(int &x, int op)//op = 0 左旋, op = 1右旋
    {
        int y = p[x].ch[!op];
```

```
        p[x].ch[!op] = p[y].ch[op];
        p[y].ch[op] = x;
        x = y;
    }
    //如果当前节点的优先级比根大就旋转,
     如果当前节点是根的左儿子就右旋如果当前节点是根的右儿子就左旋.
    void insert(int tkey, int &k = root)
    {
        if(k == -1)
        {
            k = ++size;
            p[k].ch[0] = p[k].ch[1] = -1;
            p[k].key = tkey;
            p[k].fix = rand();
        }
        else if(tkey < p[k].key)
        {
            insert(tkey, p[k].ch[0]);
            if(p[p[k].ch[0]].fix > p[k].fix)
                rot(k, 1);
        }
        else
        {
            insert(tkey, p[k].ch[1]);
            if(p[p[k].ch[1]].fix > p[k].fix)
                rot(k, 0);
        }
    }
    void remove(int tkey, int &k)//把要删除的节点旋转到叶节点上, 然后直接删除
    {
        if(k == -1) return ;
        if(tkey < p[k].key)
            remove(tkey, p[k].ch[0]);
        else if(tkey > p[k].key)
            remove(tkey, p[k].ch[1]);
        else
        {
            if(p[k].ch[0] == -1 && p[k].ch[1] == -1)
                k = -1;
            else if(p[k].ch[0] == -1)
                k = p[k].ch[1];
            else if(p[k].ch[1] == -1)
                k = p[k].ch[0];
            else if(p[p[k].ch[0]].fix < p[p[k].ch[1]].fix)
            {
                rot(k, 0);
                remove(tkey, p[k].ch[0]);
            }
            else
            {
                rot(k, 1);
                remove(tkey, p[k].ch[1]);
            }
        }
    }
};
```

1.5 ST 表

```
///ST表(Sparse Table)
//对静态数组, 查询任意区间[l, r]的最大(小)值
// 预处理O(nlog n), 查询O(1)
```

```cpp
#define MAX 10000
int st[MAX][22];//st表 —— st[i][j]表示从第i个元素起，连续2^j个元素的最大（小）值
int Log2[MAX];//对应于数x中最大的是2的幂的区间长度， k = floor(log2(R - L + 1))
void pre_ST(int n, int ar[])//n 数组长度， ar 数组
{
    int i, j;
    Log2[1] = 0;
    for(i = 2; i <= n; i++) Log2[i] = Log2[i>>1] + 1;
    for(i = n - 1; i >= 0; i--)
    {
        st[i][0] = ar[i];
        for(j = 1; i + (1 << j) <= n; j++)
            st[i][j] = max(st[i][j - 1], st[i + (1 << (j - 1))][j - 1]);
    }
}
int query(int l, int r)
{
    int k = Log2[r - l + 1];
    return max(st[l][k], st[r - (1 << k) + 1][k]);
}
```

1.6 莫队算法分块

```cpp
///莫队算法
//复杂度: O(n√n · 转移的复杂度)
//n个数, q次区间查询
/*使用条件:
已知区间[l, r]的答案, 可在O(1) 或O(log n) 内得到[l + 1, r], [l - 1, r], [l, r + 1], [l, r - 1]的答案
查询可以离线*/
/*做法:
    将查询以l所在的块为第一关键字, 以r为第二关键字排序, 然后从一个初始状态开始转移
注意:   1. 转移的时候, 要先转移使l减小, 和使r增大的部分, 后转移使l增大和使r减小的部分, 避免l > r
        2. 转移的时候注意先加减l, r, 后加减l, r的问题.
*/
int qst;//每块的大小
struct Query
{
    int l, r, id;
} qry[NUM];
LL ans[NUM];
bool operator < (const Query &a, const Query &b)
{
    if((a.l / sqt) == (b.l / sqt)) return a.r < b.r;
    return a.l < b.l;
}
int update(int l, int r);//转移[l, r]到答案的改变
void solve(int n)
{
    sqt = sqrt(1.0 * n);
    sort(qry, qry + Q);
    int l = 0, r = 0;
    int res = a[0];
    for(i = 0; i < Q; i++)
    {
        while(l > qry[i].l) res += update(--l, r);
        while(r < qry[i].r) res += update(l, ++r);
        while(r > qry[i].r) res -= update(l, r--);
        while(l < qry[i].l) res -= update(l++, r);
        ans[qry[i].id] = res;
    }
}
//分块 O(N√N)
```

1.7 并查集 Disjoint Set

```
//并查集 Disjoint Set
int father[MAX], rk[MAX];
void init()
{
    for(int i = 0; i < MAX; i++)
    {
        father[i] = i;
        rk[i] = 0;
    }
}
int find(int x)
{
    return father[x] == x ? x : father[x] = find(father[x]);
}
void gather(int x, int y)
{
    x = find(x);
    y = find(y);
    if(x == y)   return;
    if(rk[x] > rk[y])  father[y] = x;
    else
    {
        if(rk[x] == rk[y]) rk[y]++;
        father[x] = y;
    }
}

bool same(int x, int y)
{
    return find(x) == find(y);
}
```

2 动态规划

2.1 最长上升公共子序列 LIS

```
/*最长上升子序列LIS
    给一个序列, 求满足的严格递增的子序列的最大长度(或者子序列)
    标签: dp
    做法: dp[i]表示长度为i的子序列在第i位的最小值, 每次更新时, 找到最大的k使dp[k] ≤ a_i,
    将dp[k+1]的值更新为a_i.
    可以用pre数组存储第i个数的最长子序列的前一个数.
*/
/*两个互不覆盖的最长上升子序列
    给一个序列, 要求找到两个互不覆盖的最长上升子序列, 使其长度之和最大. (n ≤ 1000)
    标签: 二维dp, 树状数组优化
    做法: dp[l][r]表示第一个串以l结尾, 第二个串以r结尾的最大长度和, 转移方程:
```

$$dp[i][u] = \max\{dp[i][j]\} + 1(1 \le j < u)$$

,

$$dp[u][j] = \max\{dp[i][j]\} + 1(1 \le i < u)$$

```
    .然后用多个树状数组维护区间最大值.
*/
/*二维偏序的LIS
    给一个二维坐标(x,y)的序列, 求满足对任意i<j, 都有x_i < x_j, y_i < y_j的最长子序列
    做法: 二分 + 树状数组 + dp (cdq)
    将序列[l, r]二分, 先处理左边的区间[l, mid],
    再用左边的区间更新右边的区间, 即将区间[l,r]按左端点排序, 然后依次扫描,
    遇到在左半区间的加入树状数组, 遇到在右半区间的查询比当前y值更小的数对数并更新,
    然后再递归处理右边的区间[mid+1, r]
*/
```

3 图论

3.1 拓扑排序

```
///拓扑排序 (Topologicla Sorting)
//有向无环图 O(|E|+|V|)
/*算法:
    1. 选择没有前驱(入度为0)的顶点v, 并输出
    2. 删除从v出发的所有有向边
    3. 重复前两步, 直至没有入度为0的顶点
    4. 如果最后还剩下一些顶点, 这该图不是DAG
*/
const int MAXV = 1000;
int V;//顶点数
int deg[MAXV];//入度
int ans[MAXV];//拓扑排序结果
bool TopoSort()
{
    int top = 0, tail = 0;
    for(int i = 0; i <= V; ++i)
    {
        if(deg[i] == 0)
            ans[top++] = i;
    }
    while(tail < top)
    {
        for(int i = head[tail++]; ~i; i = e[i].next)
        {
            if(--deg[e[i].to] ==  0)
                ans[top++] = e[i].to;
        }
    }
    return top == V;
}
```

3.2 欧拉回路 Euler, 哈密顿回路 Hamilton

```
///欧拉回路 Euler
/*
定义: 寻找一条回路, 经过且仅经过一次所有边, 最后回到出发点
存在的充要条件: 1. 该图是连通的 2. 无向图, 度数为奇数的顶点的个数为0; 有向图, 每个顶点入度等于出度
构造算法: O(|E|)
    1. 深度搜索, 得出一条回路.
    2. 如果该回路不是欧拉回路, 则沿该回路回溯, 找到一个没有搜索过的顶点, 重复步骤1,
     然后将新回路加入答案中.
定义欧拉路径: 寻找一条简单路径, 经且仅经过所有边一次
存在条件: 连通, (无向图)度数为奇数的顶点个数为0或2, (有向图)只有两个顶点入度不等于出度,
     且一个入度比出度大1, 另一个小1.
*/
#define MAXE
int ans[MAXE], ansi;//ansi等于V表示存在欧拉回路
bool vis[2 * MAXE];
void dfs(int u)
{
    for(int i = head[u]; ~i; i = e[i].next)//链式前向星存储
    {
        if(vis[i]) continue;
        vis[i ^ 1] = vis[i] = true;//标记当前边及反向边
        dfs(e[i].to);
        ans[ansi++] = i;//沿回溯道路将经过的每个点加入答案
```

```
    }
}
///有向图欧拉路径
int in[MAX_V], out[MAX_V];//入度和出度
int num;//已经有多少边在欧拉路径中
int pre_edge[MAX_E];//存储每一条边的上一条边
//通过入度和出度判定是否可能存在欧拉路径, 存在返回起点, 否则返回-1
int is_exist_euler()
{
    int sp = -1, tp = -1;
    bool flag = true;
    for(int i = 0; flag && i < V; ++i)
    {
        if(in[i] == out[i]) continue;
        if(in[i] - out[i] == 1)
        {
            if(tp == -1) tp = i;
            else flag = false;
        }
        else if(out[i] - in[i] == 1)
        {
            if(sp == -1) sp = i;
            else flag = false;
        }
        else flag = false;
    }
    if(!flag) return -1;
    if(sp < 0 && tp >= 0) return - 1;
    if(sp >= 0 && tp < 0) return -1;
    if(sp < 0 && tp < 0)
    {
        int i = 0;
        while(sp == -1)
        {
            if(head[i] != -1)
            {
                sp = i;
            }
            i++;
        }
    }
    if(!flag)
        sp = -1;
    return sp;
}
int find_path(int u, int pree)
{
    //寻找以u为起点的一条边
    int i;
    while(head[u] != -1)
    {
        i = head[u];
        pre_edge[i] = pree;//加入改变进欧拉路径
        pree = i;
        head[u] = e[i].next;//将已经访问过的边去掉
        u = e[i].v;//要访问的下一个结点
        num++;
    }
    return pree;//访问当前路径最后一条边
}
//存在欧拉路径返回最后一条边的编号, 否则返回-1
int oula_path()
{
    int st = is_exist_euler();//判断是否可能存在欧拉路径
```

```
    if(st == -1) return -1;
    num = 0;
    int ed = find_path(st, -1);//找到基础路径
    int ei = ed;
    while(ei >= 0)
    {
        if(head[e[ei].u] != -1)
        {
            pre_edge[ei] = find_path(e[ei].u, pre_edge[ei]);//寻找环
        }
        ei = pre_edge[ei];
    }
    if(num != E)//可能存在其他连通块
        return -1;
    return ed;
}
///哈密顿回路(Hamilton)
/*
定义: 寻找一条回路, 经且仅经过每个顶点一次, 最后回到出发点.
存在的充分条件: 任意两个不同顶点的度数和大于等于顶点数V.
构造算法: O(|V|^2)
    1. 任找两个相邻顶点S, T
    2. 分别向两头扩展至无法扩展为止, 称头尾结点为S, T
    3. 若S, T不相邻, 在路径S -> T中找到结点V_i, 其中V_i与T相邻, V_{i+1}与S相邻,
     令S -> T变为S -> V_i -> T -> V_{i+1}.
    4. (S -> T已为回路)若S -> T中顶点个数不为V, 找S -> T中找到顶点V_i, 其中V_i与一个未访问过得顶点相邻,
     从V_i处断开(S为V_i, T为V_{i+1},重复2.
*/
#define MAXV
int V;
int ans[MAXV];
bool G[MAXV][MAXV];//邻接矩阵存储
bool vis[MAXV];

void Hamilton()
{
    int s = 1, t;
    int ansi = 2;
    int i, j;
    memset(vis, false, sizeof(vis));
    for(i = 1; i <= V; i++) if(G[s][i]) break;
    t = i;
    ans[0] = s, ans[1] = t;
    while(1)
    {
        while(1)
        {
            for(i = 1; i <= V; i++)
                if(G[t][i] && !vis[i])
                {
                    ans[ansi++] = i;
                    vis[i] = true;
                    t = i;
                    break;
                }
            if(i > V)break;
        }
        reverse(ans, ans + ansi);
        swap(s, t);
        while(1)
        {
            for(i = 1; i <= V; i++)
                if(G[t][i] && !vis[i])
                {
```

```
                    ans[ansi++] = i;
                    vis[i] = true;
                    t = i;
                    break;
                }
            if(i > V)break;
        }
        if(!G[s][t])
        {
            for(i = 1; i < ansi - 2; i++)if(G[s][ans[i + 1]] && G[ans[i]][t]) break;
            t = ans[++i];
            reverse(ans + i, ans + ansi);
        }
        if(ansi == V)return ;
        for(j = 1; j <= V; j++)
            if(!vis[j])
                for(i = 1; i < ansi - 2; i++)if(G[ans[i]][j])break;
        s = ans[i - 1];
        t = j;
        reverse(ans, ans + i);
        reverse(ans + i, ans + ansi);
        ans[ansi++] = j;
        vis[j] = true;
    }
}
```

3.3 无向图的桥，割点，双连通分量

```
/*
无向连通图的点连通度：使一个无向连通图变成多个连通块要删去的最少顶点数(及相连的边).
无向连通图的边连通度：使一个无向连通图变成多个连通块要删去的最少边数.
当无向图的点连通度或边连通度大于1时，称该图双连通图.
割点(关节点,割顶)：(点连通度为1) 删去割点及其相连的边后，该图由连通变为不连通;
割边(桥)：(边连通度为1)删去割边后，连通图变得不连通.
点双连通分量：点连通度大于1的分量.
边双连通分量：边连通度大于1的分量.
桥的两个端点是割点，有边相连的两割点之间的边不一定是桥(重边).
*/
///Tarjan算法求割点或桥
/*
定义 dfn(u) 为无向图中在深度优先搜索中的遍历次序，low(u)
     为顶点u或u的子树中的顶点经过非父子边追溯到的最早的结点 (dfn序号最小).

割点：该点是根结点且有不止一棵子树，或该结点的任意一个孩子结点，没有到该结点祖先的反向边(存在
    dfn(u) ≤ low(v))

桥：当且仅当该边(u,v)是树枝边，且dfn[u] < low[v]

缩点：缩点后变成一棵k个点k-1条割边连接成的树，而割点可以存在于多个块中.

将有桥的连通图加边变为边双连通图(加边数最少)：将边双连通分量缩点，形成一棵树，
     反复将两个最近公共祖先最远的两个叶子节点之间连一条边，(这样形成一个双连通分量，可缩点)，
     刚好(leaf+1)/2条边.

点双连通分支：建立一个栈，存储当前双连通分支，在搜索图时，每找到一条树枝边或后向边(非横叉边)，
     就把这条边加入栈中. 如果遇到某时满足dfn[u] ≤ low[v]，说明u是一个割点，
     同时把边从栈顶一个个取出，直到遇到了边(u,v)，取出的这些边与其关联的点，组成一个点双连通分支.
     割点可以属于多个点双连通分支，其余点和每条边只属于且属于一个点双连通分支.(树枝边:
     搜索过程中遍历过的边).

边双连通分支：求出所有的桥后，把桥边删除，原图变成了多个连通块 则每个连通块就是一个边双连通分支.
     桥不属于任何一个边双连通分支，其余的边和每个顶点都属于且只属于一个边双连通分支.
```

```
*/

//图
const int MAXV = 100010, MAXE = 100010;
struct edge
{
    int next, to;
} e[MAXE];
int head[MAXV], htot;
int V, E;
void init()
{
    memset(head, -1, sizeof(head));
    htot = 0;
}
void add_edge(int u, int v)
{
    e[htot].to = v;
    e[htot].next = head[u];
    head[u] = htot++;
}

int dfn[MAXV], low[MAXV];
int stk[MAXV], top;//栈 -DCC

//割点 点双连通分量
int cp[MAXV];//记录割点
int id[MAXE], cnt_dcc;//连通分量编号及总数 -DCC
int index;
void tarjan(int u, int root, int pree) //重边对应边的id, 否则对应父亲结点fa
{
    dfn[u] = low[u] = index++;
    int num = 0;
    for(int i = head[u]; i != -1; i = e[i].next)
    {
        int v = e[i].to;
        if(!dfn[v])
        {
            num++;
            stk[top++] = i;//-DCC
            tarjan(v, root, i);
            if(low[u] > low[v])
                low[u] = low[v];
            if((u == root && num >= 2) || (u != root && dfn[u] <= low[v]))
            {
                cp[u] = 1;//是割点
                //-DCC
                cnt_dcc++;
                do
                {
                    id[stk[--top]] = cnt_dcc;
                }
                while(stk[top] != i);
            }
        }
        else if((i ^ 1) != pree) //-DCC
        {
            if(low[u] > dfn[v])
                low[u] = dfn[v];
            //-DCC
            if(dfn[u] > dfn[v])
                stk[top++] = i;
        }
    }
```

```
}
void DCC(int V)
{
    index = top = cnt_dcc = 0;// -DCC
    memset(dfn, 0, sizeof(dfn));
    memset(cp, 0, sizeof(cp));
    for(int i = 1; i <= V; i++)
        tarjan(i, i, -1);
}

//桥 边双连通分量
//注释部分为求边的双连通分量
int ce[MAXE], num; //记录桥
//int id[MAXE << 1], cnt_dcc; //连通分量编号及总数
int dfn[MAXV], low[MAXV];
//int stk[MAXV], top;
void tarjan(int u, int order, int pree)
{
    dfn[u] = low[u] = order++;
//    stk[top++] = u;
    for(int i = head[u]; ~i; i = e[i].next)
    {
        if(!e[i].inpath || (i ^ 1) == pree) continue;
        int v = e[i].to;
        if(!dfn[v])
        {
            tarjan(v, order, i);
            if(low[v] < low[u]) low[u] = low[v];
            if(dfn[u] < low[v])
            {
                ce[num++] = (i >> 1) + 1;
//                ++cnt_dcc;
//                do
//                {
//                    id[stk[--top]] = cnt_dcc;
//                }
//                while(stk[top] != v);
            }
        }
        else if(dfn[v] < low[u])
            low[u] = dfn[v];
    }
}
void DCC()
{
    memset(dfn, 0, sizeof(dfn));
    num = 0;
//    top = cnt_dcc = 0;
    for(int u = 1; u <= V; ++u) if(!dfn[u]) tarjan(u, 1, -1);
}
//缩点
//为连通图
const int MAXV = 10000 + 10, MAXE = 2000000 + 10;
int dfn[MAXV], low[MAXV];
int Belong[MAXV];
struct Graph
{
    struct edge
    {
        int next, to;
        bool isBridge;
    } e[MAXE];
    int head[MAXV], htot;
    int V, E;
```

```
154     void init()
155     {
156         memset(head, -1, sizeof(head));
157         htot = 0;
158     }
159     void add_edge(int u, int v)
160     {
161         e[htot].to = v;
162         e[htot].next = head[u];
163         e[htot].isBridge = false;
164         head[u] = htot++;
165     }
166     void tarjan(int u, int order, int pree)
167     {
168         dfn[u] = low[u] = order++;
169         for(int i = head[u]; ~i; i = e[i].next)
170         {
171             if((i ^ 1) == pree) continue;
172             int v = e[i].to;
173             if(!dfn[v])
174             {
175                 tarjan(v, order, i);
176                 if(low[v] < low[u]) low[u] = low[v];
177                 if(dfn[u] < low[v])
178                 {
179                     e[i].isBridge = e[i ^ 1].isBridge = true;
180                 }
181             }
182             else if(dfn[v] < low[u])
183                 low[u] = dfn[v];
184         }
185     }
186     void ReBuild(Graph &g)
187     {
188         memset(dfn, 0, sizeof(dfn));
189         tarjan(u, 1, -1);
190         queue<int> que;
191         int &n = g.V;
192         Belong[1] = ++n;
193         que.push(1);
194         while(!que.empty())
195         {
196             int u = que.front();
197             que.pop();
198             for(int i = head[u]; ~i; i = e[i].next)
199             {
200                 int &v = e[i].to;
201                 if(Belong[v]) continue;
202                 if(e[i].isBridge)
203                 {
204                     Belong[v] = ++n;
205                     g.add_edge(Belong[u], Belong[v]);
206                     g.add_edge(Belong[v], Belong[u]);
207                 }
208                 else Belong[v] = Belong[u];
209                 que.push(v);
210             }
211         }
212     }
213 } g1, g2;
```

3.4 最短路 shortest path

```
///最短路 Shortest Path
//dijkstra算法 O(|E|log|V|)
struct edge {int next, to, cost;} e[MAXE];
int head[MAXV], tot;
int V, E;//结点数和边数(结点编号开始于1)
void init();
void add_edge(int u, int v, int w);
int dist[MAXV];
void dijstra(int s)
{
    priority_queue<P, vector<P>, greater<P> > que;
    for(int i = 1; i <= V; ++i) dist[i] = INF;
    dist[s] = 0;
    que.push(P(0, s));
    while(!que.empty())
    {
        P p = que.top(); que.pop();
        int u = p.SE;
        if(dist[u] < p.FI) continue;
        for(int i = head[u]; ~i; i = e[i].next)
        {
            if(dist[e[i].to] <= dist[u] + e[i].cost) continue;
            dist[e[i].to] = dist[u] + e[i].cost;
            que.push(P(dist[e[i].to], e[i].to));
        }
    }
}

//dijkstra算法O(|V|^2)
int cost[MAXV][MAXV];
int d[MAXV];
bool vis[MAXV];
void dijkstra(int s)
{
    fill(d, d + V, INF);
    memset(vis, 0, sizeof(vis));
    d[s] = 0;
    while(true)
    {
        int v = -1;
        for(int u = 0; u < V; u++)
            if(!vis[u] && (v == -1 || d[u] < d[v]))
                v = u;
        if(v == -1) break;
        for(int u = 0; u < V; u++)
            d[u] = min(d[u], d[v] + cost[v][u]);
    }
}
//Bellman-Ford算法O(|E|·|V|)
//d[v] = min{d[u] + w[e]}(e =< u, v >∈ E)

const int MAXV = 1000, MAXE = 1000, INF = 1000000007;
struct edge {int u, v, cost;} e[MAXE];
int V, E;
//graph G
int d[MAXV];
void Bellman_Ford(int s)
{
    for(int i = 0; i < V; i++)
        d[i] = INF;
    d[s] = 0;
    while(true)
```

```
    {
        bool update = false;
        for(int i = 0; i < E; i++)
        {
            if(d[e[i].u] != INF && d[e[i].v] > d[e[i].u] + e[i].cost)
            {
                d[e[i].v] = d[e[i].u] + e[i].cost;
                update = true;
            }
        }
    }
}
//判负圈
bool find_negative_loop()
{
    memset(d, 0, sizeof(d));
    for(int i = 0; i < V; i++)
    {
        for(int j = 0; j < E; j++)
        {
            if(d[e[j].v] > d[e[i].u] + e[j].cost)
            {
                d[e[j].v] = d[e[j].u] + e[j].cost;
                if(i == V - 1)
                    return true;
                //循环了V次后还不能收敛，即存在负圈
            }
        }
    }
    return false;
}

//spfa算法 O(|E|log|V|)
//适用于负权图和稀疏图，稳定性不如dijstra
//存在负环返回false
int dist[MAXV];
int outque[MAXV];//出队次数，如果大于V，证明有负圈
bool vis[MAXV];
bool spfa(int s)
{
    for(int i = 0; i < V; i++)
    {
        vis[i] = false;
        dist[i] = INF;
        outque[i] = 0;
    }
    dist[s] = 0;
    queue<int> que;
    que.push(s);
    vis[s] = true;
    while(!que.empty())
    {
        int u = que.front();
        que.pop();
        vis[u] = false;
        if(++outque[u] > V) return false;
        for(int i = head[u]; i != -1; i = e[i].next)
        {
            int v = e[i].to;
            if(dist[v] <= dist[u] + e[i].cost) continue;
            dist[v] = dist[u] + e[i].cost;
            if(vis[v]) continue;
            vis[v] = true;
            que.push(v);
```

```
        }
    }
    return true;
}

///任意两点间最短路
//Floyd-Warshall算法 O(|V|^3)
int dist[MAXV][MAXV];
void floyd_warshall(int V)
{
    int i, j, k;
    for(k = 0; k < V; k++)
        for(i = 0; i < V; i++)
            for(j = 0; j < V; j++)
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
}

///两点间最短路 —— 一条可行路径还原
/*用prev[u]记录从s到u的最短路上u的前驱结点*/
vector <int> get_path(int t)
{
    vector <int> path;
    for(; t != -1; t = prev[t])
        path.push_back(t);
    reverse(path.begin(), path.end());
    return path;
}

///两点间最短路 —— 所有可行路径还原 O(|E|)
/*
从终点t反向dfs, 将所有满足dist[u] = e.cost + dist[v]的边e(u,v)加入路径中即可
在无向图中直接运行即可, 而在有向图中需要在其逆图中运行.
*/
int InPath[MAXE << 1];
void GetPath()
{
    memset(vis, 0, sizeof(vis));
    //memset(InPath, 0, sizeof(InPath));
    queue<int> que;
    que.push(n);
    vis[n] = true;
    while(!que.empty())
    {
        int u = que.front(); que.pop();
        for(int i = head[u]; ~i; i = e[i].next)
        {
            if(dist[u] != dist[e[i].to] + e[i].cost) continue;
            InPath[i] = InPath[i ^ 1] = true;
            if(vis[e[i].to]) continue;
            vis[e[i].to] = true;
            que.push(e[i].to);
        }
    }
}
///求最短路网络上的桥
//方法1: 将最短路网络新建一个图, 跑Tarjan算法.
//方法2: 最短路径还原时, 当且仅当队列为空, 且当前结点只有一条边指向s时, 该边为桥
//求割点类似
int vis[MAXV];
int ce[MAXE], num;
void get_bridge(int s, int t)//在逆图中运行
{
    priority_queue<P> que;
    que.push(P(dist[t], t));//按到t的距离远近出队, 保证割点一定后出队
```

```
    memset(vis, 0, sizeof(vis));
    vis[t] = 1;
    num = 0;
    while(!que.empty())
    {
        int u = que.top().SE;
        que.pop();
        int cnt = 0;
        if(que.empty())
        {
            for(int i = rhead[u]; i != -1; i = re[i].next)
            {
                int v = re[i].to, w = re[i].cost;
                if(dist[v] + w == dist[u])
                {
                    cnt++;
                    if(cnt >= 2) break;
                }
            }
        }
        bool f = que.empty() && cnt == 1;//当且仅当, 队列为空且只有一条路时, 找到桥
        for(int i = rhead[u]; i != -1; i = re[i].next)
        {
            int v = re[i].to, w = re[i].cost;
            if(dist[v] + w == dist[u])
            {
                if(f) ce[num++] = i;
                if(!vis[v])
                {
                    vis[v] = 1;
                    que.push(P(dist[v], v));
                }
            }
        }
    }
}
```

3.5 最大流 maximum flow

```
///最大流 maximum flow
//最大流最小割定理 : 最大流 = 最小割
///FF算法 Ford–Fulkerson算法 O(F|E|) F为最大流量
//1. 初始化: 原边容量不变, 回退边容量为0, max_flow = 0
//2. 在残留网络中找到一条从源S到汇T的增广路, 找不到得到最大流max_flow
//3. 增广路中找到瓶颈边, max_flow加上其容量
//4. 增广路中每条边减去瓶颈边容量, 对应回退边加上其容量
struct edge
{
    int to, cap, rev;
};

vector <edge> G[MAXV];
bool used[MAXV];

void add_edge(int from, int to, int cap)
{
    G[from].push_back((edge) {to, cap, G[to].size()});
    G[to].push_back((edge) {from, 0, G[from].size() - 1});
}

//dfs寻找增广路
int dfs(int v, int t, int f)
```

```
{
    if(v == t)
        return f;
    used[v] = true;
    for(int i = 0; i < G[v].size(); i++)
    {
        edge &e = G[v][i];
        if(!used[e.to] && e.cap > 0)
        {
            int d = dfs(e.to, t , min(f, e.cap));
            if(d > 0)
            {
                e.cap -= d;
                G[e.to][e.rev].cap += d;
                return d;
            }
        }
    }
    return 0;
}

//求解从s到t的最大流
int max_flow(int s, int t)
{
    int flow = 0;
    for(;;)
    {
        memset(used, 0, sizeof(used));
        int f = dfs(s, t, INF);
        if(f == 0)
            return flow;
        flow += f;
    }
}
///Dinic算法 O(|E|·|V|^2)
//似乎比链式前向星快
struct edge {int to, cap, rev;};
vector <edge> G[MAXV];
int level[MAXV];
int iter[MAXV];
void init()
{
    for(int i = 0; i < MAXV; i++)
        G[i].clear();
}
void add_edge(int from, int to, int cap)
{
    G[from].push_back((edge) {to, cap, G[to].size()});
    G[to].push_back((edge) {from, 0, G[from].size() - 1 });
}
bool bfs(int s, int t)
{
    memset(level, -1, sizeof(level));
    queue <int> que;
    level[s] = 0;
    que.push(s);
    while(!que.empty())
    {
        int v = que.front();
        que.pop();
        for(int i = 0; i < (int)G[v].size(); i++)
        {
            edge &e = G[v][i];
            if(e.cap > 0 && level[e.to] < 0)
```

```
            {
                level[e.to] = level[v] + 1;
                que.push(e.to);
            }
        }
    }
    return level[t] != -1;
}

int dfs(int v, int t, int f)
{
    if(v == t) return f;
    for(int &i = iter[v]; i < (int)G[v].size(); i++)
    {
        edge &e = G[v][i];
        if(e.cap > 0 && level[v] < level[e.to])
        {
            int d = dfs(e.to, t, min(f, e.cap));
            if(d > 0)
            {
                e.cap -= d;
                G[e.to][e.rev].cap += d;
                return d;
            }
        }
    }
    return 0;
}

int max_flow(int s, int t)
{
    int flow = 0, cur_flow;
    while(bfs(s, t))
    {
        memset(iter, 0, sizeof(iter));
        while((cur_flow = dfs(s, t, INF)) > 0) flow += cur_flow;
    }
    return flow;
}
///SAP算法 O(|E| · |V|^2)
#define MAXV 1000
#define MAXE 10000
struct edge
{
    int cap, next, to;
} e[MAXE * 2];
int head[MAXV], tot_edge;
void init()
{
    memset(head, -1, sizeof(head));
    tot_edge = 0;
}
void add_edge(int u, int v, int cap)
{
    e[tot_edge] = (edge) {cap, head[u], v};
    head[u] = tot_edge++;
}
int V;
int numh[MAXV];//用于GAP优化的统计高度数量数组
int h[MAXV];//距离标号数组
int pree[MAXV], prev[MAXV];//前驱边与结点
int SAP_max_flow(int s, int t)
{
    int i, flow = 0, u, cur_flow, neck = 0, tmp;
```

```
    memset(h, 0, sizeof(h));
    memset(numh, 0, sizeof(numh));
    memset(prev, -1, sizeof(prev));
    for(i = 1; i <= V; i++)//从1开始的图，初识化为当前弧的第一条临接边
        pree[i] = head[i];
    numh[0] = V;
    u = s;
    while(h[s] < V)
    {
        if(u == t)
        {
            cur_flow = INT_MAX;
            for(i = s; i != t; i = e[pree[i]].to)
            {
                if(cur_flow > e[pree[i]].cap)
                {
                    neck = i;
                    cur_flow = e[pree[i]].cap;
                }
            }//增广成功，寻找"瓶颈"边
            for(i = s; i != t; i = e[pree[i]].to)
            {
                tmp = pree[i];
                e[tmp].cap -= cur_flow;
                e[tmp ^ 1].cap += cur_flow;
            }//修改路径上的边容量
            flow += cur_flow;
            u = neck;//下次增广从瓶颈边开始
        }
        for(i = pree[u]; i != -1; i = e[i].next)
            if(e[i].cap && h[u] == h[e[i].to] + 1)
                break;//寻找可行弧
        if(i != -1)
        {
            pree[u] = i;
            prev[e[i].to] = u;
            u = e[i].to;
        }
        else
        {
            if(0 == --numh[h[u]])break;//GAP优化
            pree[u] = head[u];
            for(tmp = V, i = head[u]; i != -1; i = e[i].next)
                if(e[i].cap)
                    tmp = min(tmp, h[e[i].to]);
            h[u] = tmp + 1;
            ++numh[h[u]];
            if(u != s) u = prev[u];//从标号并且从当前结点的前驱重新增广
        }
    }
    return flow;
}

///EK算法 O(|V|·|E|^2)
//bfs寻找增广路
const int MAXV = 210;
int g[MAXV][MAXV], pre[MAXV];
int n;
bool vis[MAXV];
bool bfs(int s, int t)
{
    queue <int> que;
    memset(pre, -1, sizeof(pre));
    memset(vis, 0, sizeof(vis));
```

```
    que.push(s);
    vis[s] = true;
    while(!que.empty())
    {
        int u = que.front();
        if(u == t) return true;
        que.pop();
        for(int i = 1; i <= n; i++)
            if(g[u][i] && !vis[i])
            {
                vis[i] = true;
                pre[i] = u;
                que.push(i);
            }
    }
    return false;
}
int EK_max_flow(int s, int t)
{
    int u, max_flow = 0, minv;
    while(bfs(s, t))
    {
        minv = INF;
        u = t;
        while(pre[u] != -1)
        {
            minv = min(minv, g[pre[u]][u]);
            u = pre[u];
        }
        ans += minv;
        u = t;
        while(pre[u] != -1)
        {
            g[pre[u]][u] -= minv;
            g[u][pre[u]] += minv;
            u = pre[u];
        }
    }
    return max_flow;
}
```

3.6 最小割 minimum cut

```
///最小割Minimum Cut
/*
定义:
割: 网络(V,E)的割(cut)[S,T]将点集V划分为S和T(T=V-S)两个部分, 使得源s ∈ S, 汇t ∈ T.
    符号[S,T]代表一个边集合{< u,v > | < u,v >∈ E,u ∈ S,v ∈ T}. 穿过割[S,T]的净流(net
    flow)定义为f(S,T), 容量(capacity)定义为c(S,T).
最小割: 该网络中容量最小的割
(割与流的关系) 在一个流网络G(V, E)中, 设其任意一个流为f, 且[S, T]为G一个割. 则通过割的净流为f(S, T)
    = |f|.
(对偶问题的性质) 在一个流网络G(V, E)中, 设其任意一个流为f, 任意一个割为[S, T], 必有|f| ≤ c[S,T].
(最大流最小割定理) 如果f是具有源s和汇t的流网络G(V,E)中的一个流,则下列条件是等价的:
        (1) f是G的一个最大流
        (2) 残留网络G_f不包含增广路径
        (3) 对G的某个割[S,T], 有|f|=c[S,T]
        即最大流的流值等于最小割的容量
*/
/*
性质:
    性质1(不连通): 在给定的流网络中, 去掉割的边集, 则不存在任何一条从源到汇的路径.
```

```
    性质2(两类点): 在给定的流网络中, 任意一个割将点集划分成两部分. 割为两部分点之间的"桥梁".
技巧:
    技巧1 用正无限容量排除不参与决策的边.
    技巧2 使用割的定义式来分析最优性.
    技巧3 利用与源或汇关联的边容量处理点权.
*/
/*
最小割的求法:
    1. 先求的最大流
    2. 在得到最大流f后的残留网络G_f中,从源s开始深度优先遍历(DFS),所有被遍历的点, 即构成点集S
    注意: 虽然最小割中的边都是满流边, 但满流边不一定都是最小割的边.
*/
int max_flow(int s, int t) {}
int getST(int s, int t, int vis[])
{
    int mincap = max_flow(s, t);
    memset(nd, 0, sizeof(nd));
    queue<int> que;
    que.push(s);
    vis[s] = 1;
    while(!que.empty())
    {
        int u = que.front(); que.pop();
        for(int i = 0; i < (int)g[u].size(); i++)//travel v
            if(g[u][i].cap > 0 && !vis[g[u][i].to])
            {
                vis[g[u][i].to] = 1;
                que.push(g[u][i].to);
            }
    }
    return mincap;
}
///无向图全局最小割 Stoer—Wagner算法
/*定理: 对于图中任意两点s和t来说, 无向图G的最小割要么为s到t的割, 要么是生成图G/{s,t}的割(把s和t合并)
算法的主要部分就是求出当前图中某两点的最小割, 并将这两点合并
快速求当前图某两点的最小割:
    1. 维护一个集合A, 初始里面只有v_1(可以任意)这个点
    2. 区一个最大的w(A, y)的点y放入集合A(集合到点的权值为集合内所有点到该点的权值和)
    3. 反复2,直至A集合G集相等
    4. 设后两个添加的点为s和t, 那么w(G—{t}, t)的值, 就是s到t的cut值
*/
//O(|V|^3)
const int MAXV = 510;
int n;
int g[MAXV][MAXV];//g[u][v]表示u,v两点间的最大流量
int dist[MAXV];//集合A到其他点的距离
int vis[MAXV];
int min_cut_phase(int &s, int &t, int mark) //求某两点间的最小割
{
    vis[t] = mark;
    while(true)
    {
        int u = -1;
        for(int i = 1; i <= n; i++)
            if(vis[i] < mark && (u == -1 || dist[i] > dist[u])) u = i;
        if(u == -1) return dist[t];
        s = t, t = u;
        vis[t] = mark;
        for(int i = 1; i <= n; i++) if(vis[i] < mark) dist[i] += g[t][i];
    }
}

int min_cut()
{
```

```
    int i, j, res = INF, x, y = 1;
    for(i = 1; i <= n; i++)
        dist[i] = g[1][i], vis[i] = 0;
    for(i = 1; i < n; i++)
    {
        res = min(res, min_cut_phase(x, y, i));
        if(res == 0) return res;
        //merge x, y
        for(j = 1; j <= n; j++) if(vis[j] < n) dist[j] = g[j][y] = g[y][j] = g[y][j] + g[x][j];
        vis[x] = n;
    }
    return res;
}
```

3.7 分数规划 Fractional Programming

```
///分数规划  Fractional Programming
//source: <<最小割模型在信息学竞赛中的应用>>
/*
一般形式: min{λ = f(x⃗) = a(x⃗)/b(x⃗)}(x⃗ ∈ S, ∀x⃗ ∈ S, b(x⃗) > 0)
    其中解向量x⃗在解空间S内, a(x⃗)与b(x⃗)都是连续的实值函数.
    解决分数规划问题的一般方法是分析其对偶问题, 还可进行参数搜索(parametric search),
    即对答案进行猜测, 在验证该猜测值的最优性, 将优化问题转化为判定性问题或者其他优化问题.
    构造新函数: g(λ) = min{a(x⃗) − λ · b(x⃗)}(x⃗ ∈ S)
    函数性质: (单调性) g(λ)是一个严格递减函数, 即对于λ1 < λ2, 一定有g(λ1) > g(λ2).
    (Dinkelbach定理) 设λ*为原规划的最优解, 则g(λ) = 0当且仅当λ = λ*.
    设λ*为该优化的最优解, 则:
                                  { g(λ) = 0 ⇔ λ = λ*
                                  { g(λ) < 0 ⇔ λ > λ*
                                  { g(λ) > 0 ⇔ λ < λ*
    由该性质, 就可以对最优解λ*进行二分查找.
    上述是针对最小化目标函数的分数规划, 实际上对于最大化目标函数也一样适用.
*/
///0-1分数规划 0-1 fractional programming
/*是分数规划的解向量x⃗满足∀x_i ∈ {0,1}, 即一下形式:
                                  min{λ = f(x) = (Σ_{e∈E} w_e x_e)/(Σ_{e∈E} 1 · x_e) = (w⃗ · x⃗)/(c⃗ · x⃗)}
    其中, x⃗表示一个解向量, x_e ∈ {0,1}, 即对与每条边都选与不选两种决策,
    并且选出的边集组成一个s-t边割集. 形式化的, 若x_e = 1, 则e ∈ C, x_e = 0, 则e ∉ C.
*/
```

3.8 最大闭权图 maximum weight closure of a graph

```
///最大权闭合图 Maximum Weight Closure of a Graph
/*定义:
定义一个有向图G(V, E)的闭合图(closure)是该有向图的一个点集, 且该点集的所有出边都还指向该点集.
    即闭合图内的任意点的任意后继也一定在闭合图中. 更形式化地说,
    闭合图是这样的一个点集V′ ∈ V, 满足对于∀u ∈ V′引出的∀ < u, v >∈ E, 必有v ∈ V′成立.
    还有一种等价定义为: 满足对于∀ < u, v >∈ E , 若有u ∈ V′成立, 必有v ∈ V′成立,
    在布尔代数上这是一个"蕴含(imply)"的运算.
    按照上面的定义, 闭合图是允许超过一个连通块的.给每个点v分配一个点权w_v(任意实数, 可正可负).
    最大权闭合图(maximum weight closure), 是一个点权之和最大的闭合图, 即最大化 Σ_{v∈V′} w_v.
*/
/*转化为最小割模型G_N(V_N, E_N)
在原图点集的基础上增加源s和汇t;
    将原图每条有向边< u, v >∈ E >替换为容量为c(u, v) = ∞的有向边< u, v >∈ E_N;
    增加连接源s到原图每个正权点v(w_v > 0)的有向边< s, v >∈ E_N, 容量为c(s, v) = w_v,
```

增加连接原图每个负权点$v(w_v<0)$到汇t的有向边$<v,t>\in E_N$，容量为$c(v,t)=-w_v$. 其中，正无限∞定义为任意一个大于$\sum_{v\in V}|w_v|$的整数。更形式化地，有：

$$V_N = V \cup \{s,t\}$$

$$E_N = E \cup \{<s,v>|v\in V, w_v>0\} \cup \{<v,t>|v\in V, w_v<0\}$$

$$\begin{cases} c(u,v)=\infty & <s,v>\in E \\ c(s,v)=w_v & w_v>0 \\ c(v,t)=-w_v & w_v<0 \end{cases}$$

当网络N的取到最小割时，其对应的图G的闭合图将达到最大权.
*/

3.9 最大密度子图 Maximum Density Subgraph

```
///最大密度子图 a Maximum Density Subgraph
/*
定义:
    定义无向图G=(V,E)的密度(density)D 为该图的边数|E|与该图的点数|V|的比值D=|E|/|V|.
    给一个无向图G=(V,E)，其中具有最大密度的子图G'=(V',E')，称为最大密度子图(maximum density
    subgraph)，即最大化D'=|E'|/|V'|.
*/
/*
做法:
    先转化为分数规划，在转化为最小割即可.
*/
```

3.10 二分图的最小点权覆盖集与最大点权独立集

```
///二分图的最小点权覆盖集与最大点权独立集 Minimum Weight Vertex Covering Set and Maximum Weight
    Vertex Independent Set in a Bipartite Graph
/*定义:
点覆盖集 (vertex covering set, VCS) 是无向图的G的一个点集，
    使得该图中所有边都至少有一个端点在该集合内. 形式化的定义: 点覆盖集为 V'∈V，
    满足对于∀(u,v)∈E，都有 u∈V' 或 v∈V'.
点独立集 (vertex independent set, VIS) 是无向图的G的一个点集，
    使得任意两个在该集合中的点在原图中都不相邻，即导出子图为零图(没有边)的点集. 形式化的定义:
    点独立集是V'∈V，满足∀u,v∈V'，都有(u,v)∉E成立. 等价的定义: 点独立集为V'∈V，
    满足∀(u,v)∈E，都有u∈V'与v∈V'不同时成立.
最小点覆盖集 (minimum vertex covering set, MinVCS) 是在无向图G中，点数最少的点覆盖集.
最大点独立集 (maximum vertex independent set, MaxVIS) 是在无向图G中，点数最多的点独立集.
一个带点权无向图G=(V,E)，对于∀v∈V，都被分配一个非负点权w_v.
最小点权覆盖集 (minimum weight vertex covering set, MinWVCS) 是在带点权无向图G中，
    点权之和最小的点覆盖集.
最大点权独立集 (maximum weight vertex independent set, MaxWVIS) 是在带点权无向图G中，
    点权之和最大的点独立集.
带权二分图G=(V,E)中，其中V=X∪Y,X∩Y=∅，且对于∀v∈V，都被分配了一个非负的权值w_v(w_v≥0).
*/
/*二分图的最小点权覆盖集算法 Algorithm for MinWVCS in a Bipartite Graph
考虑二分图的网络流解法，它加入了额外的源s和汇t，将匹配以一条条s-u-v-t形式的流路径"串联"起来.
同样，如上建图，建立一个源s，向X部每个点连边；建立一个汇t，从Y部每个点向汇t连边.
    把二分图看做有向的，则任意一条从s到t的路径，一定具有s-u-v-t的形式.
    割的性质是不存在一条从s到t的路径，故路径上的三条边(s,u),(u,v),(v,t)中至少有一条在割中.
    若人为的令边(u,v)不在最小割中，则令其容量为正无限c(u,v)=+∞，则条件简化为 (s,u),(v,t)
    至少有一条边在最小割中，正好和点覆盖集的形式相对应(u∈V'或v∈V').
将二分图G的最小点权覆盖向网络N=(V_N,E_N)的最小割模型的转化:
    在图G的基础上添加源s和汇t；将每条二分图的边替换为容量为c(u,v)=∞的有向边(u,v)∈E_N;
    增加源s到X部的点u的有向边(s,u)∈E_N，容量即为改点的权值c(s,u)=w_u;
    增加Y部的点v到汇t有向边的(v,t)∈E_N，同样容量为该点的权c(v,t)=w_v.
```

引理: 网络N的简单割[S, T]与图G的点覆盖集$V' = X' \cup Y'$存在一一对应关系:
点覆盖集中的点在网络N中相应的带权边组成一个简单割; 反之亦然, 即:

$$[S, T] = [s, X'] \cup [Y', t]$$

由于最小点覆盖和最小割的优化方向一致, 故带权二分图的最小点覆盖转化为最小割模型.
复杂度: $O(MaxFlow(N))$
*/
/*二分图的最大点权独立集算法 *Algorithm for MaxWVIS in a Bipartite Graph*
点独立集定义重写, 可以得到其补图的点覆盖集定义. 即

$$\overline{u \in V' v \in V'} = u \in \overline{V'} \ or \ v \in \overline{V'}$$

定理 (覆盖集与独立集互补定理): 若$\overline{V'}$为不含孤立点的任意图的一个点覆盖集当且仅当V'是该图的一个点独立集.
推论 (最优性)若V'为不含孤立点的任意图的一个最小点权覆盖集, 则$\overline{V'}$就是该图的一个最大点权独立集.
求出最大点权独立点集后还有加上反图中的孤立点才是答案.
复杂度: $O(MaxFlow(N))$
*/

3.11 最小费用最大流 minimum cost flow

```
///最小费用最大流 miniunm cost flow
//不断寻找最短路增广即可
//复杂度: O(F · MaxFlow(G))
//对于稀疏图的效率较高, 对于稠密图的的效率低
///dijkstra实现 基于0开始的图
const int MAXV = 11000, MAXE = 41000;
struct edge {int next, to, cap, cost;} e[MAXE << 1];
int head[MAXV], htot;
int V;
void init()
{
    memset(head, -1, sizeof(head));
    htot = 0;
}
void add_edge(int u, int v, int cap, int cost)
{
    e[htot] = (edge) {head[u], v, cap, cost};
    head[u] = htot++;
    e[htot] = (edge) {head[v], u, 0, -cost};
    head[v] = htot++;
}
int dist[MAXV];
int prev[MAXV], pree[MAXV];
int h[MAXV];
void dijkstra(int s)
{
    priority_queue<P, vector<P>, greater<P> > que;
    fill(dist, dist + V, INF);
    que.push(P(0, s));
    dist[s] = 0;
    while(!que.empty())
    {
        P p = que.top(); que.pop();
        int u = p.SE;
        if(dist[u] < p.FI) continue;
        for(int i = head[u]; ~i; i = e[i].next)
            if(e[i].cap > 0 && dist[e[i].to] > dist[u] + e[i].cost + h[u] - h[e[i].to])
            {
                dist[e[i].to] = dist[u] + e[i].cost + h[u] - h[e[i].to];
                prev[e[i].to] = u;
                pree[e[i].to] = i;
                que.push(P(dist[e[i].to], e[i].to));
            }
```

```
    }
}
int min_cost_flow(int s, int t, int flow)
{
    int min_cost = 0;
    memset(h, 0, sizeof(h));
    while(flow > 0)
    {
        dijkstra(s);
        if(dist[t] == INF) return -1;
        for(int i = 0; i < V; i++) h[i] += dist[t];
        int now_flow = flow;
        for(int u = t; u != s; u = prev[u])//寻找瓶颈边
            now_flow = min(now_flow, e[pree[u]].cap);
        flow -= now_flow;
        min_cost += now_flow * dist[t];
        for(int u = t; u != s; u = prev[u])
        {
            e[pree[u]].cap -= now_flow;
            e[pree[u] ^ 1].cap += now_flow;
        }
    }
    return min_cost;
}
///spfa实现 基于0开始的图
struct edge {int next, to, cap, cost;} e[MAXE << 1];
int head[MAXV], htot;
int V;
void init()
{
    memset(head, -1, sizeof(head));
    htot = 0;
}
void add_edge(int u, int v, int cap, int cost)
{
    e[htot] = (edge) {head[u], v, cap, cost};
    head[u] = htot++;
    e[htot] = (edge) {head[v], u, 0, -cost};
    head[v] = htot++;
}
int dist[MAXV];
int prev[MAXV], pree[MAXV];
void spfa(int s)
{
    fill(dist, dist + V, INF);
    dist[s] = 0;
    bool update = true;
    while(update)
    {
        update = false;
        for(int v = 0; v < V; v++)
        {
            if(dist[v] == INF) continue;
            for(int i = head[v]; i != -1; i = e[i].next)
            {
                //edge &e = G[v][i];
                if(e[i].cap > 0 && dist[e[i].to] > dist[v] + e[i].cost)
                {
                    dist[e[i].to] = dist[v] + e[i].cost;
                    prev[e[i].to] = v;
                    pree[e[i].to] = i;
                    update = true;
                }
            }
```

```
        }
    }
}
int h[MAXV];
void dijkstra(int s)
{
    priority_queue<P, vector<P>, greater<P> > que;
    fill(dist, dist + V, INF);
    que.push(P(0, s));
    dist[0] = 0;
    while(!que.empty())
    {
        P p = que.top(); que.pop();
        int u = p.SE;
        if(dist[u] < p.FI) continue;
        for(int i = head[u]; ~i; i = e[i].next)
            if(e[i].cap > 0 && dist[e[i].to] > dist[u] + e[i].cost + h[u] - h[e[i].to])
            {
                dist[e[i].to] = dist[u] + e[i].cost + h[u] - h[e[i].to];
                prev[e[i].to] = u;
                pree[e[i].to] = i;
                que.push(P(dist[e[i].to], e[i].to));
            }
    }
}
int min_cost_flow(int s, int t, int flow)
{
    int min_cost = 0;
    while(flow > 0)
    {
        spfa(s);
        if(dist[t] == INF) return -1;
        int now_flow = flow;
        for(int u = t; u != s; u = prev[u])//寻找瓶颈边
            now_flow = min(now_flow, e[pree[u]].cap);
        flow -= now_flow;
        min_cost += now_flow * dist[t];
        for(int u = t; u != s; u = prev[u])
        {
            e[pree[u]].cap -= now_flow;
            e[pree[u] ^ 1].cap += now_flow;
        }
    }
    return min_cost;
}

///zkw最小费用流，在稠密图上很快
const int MAXV = 11000, MAXE = 41000;
struct edge {int next, to, cap, cost;} e[MAXE << 1];
int head[MAXV], htot;
int V;
void init()
{
    memset(head, -1, sizeof(head));
    htot = 0;
}
void add_edge(int u, int v, int cap, int cost)
{
    e[htot] = (edge) {head[u], v, cap, cost};
    head[u] = htot++;
    e[htot] = (edge) {head[v], u, 0, -cost};
    head[v] = htot++;
}
int dist[MAXV];
```

```cpp
int slk[MAXV];
int src, sink;//源和汇
bool vis[MAXV];
int min_cost;//最小费用
int aug(int u, int f)
{
    int left = f;
    if(u == sink)
    {
        min_cost += f * dist[src];
        return f;
    }
    vis[u] = true;
    for(int i = head[u]; ~i; i = e[i].next)
    {
        int v = e[i].to;
        if(e[i].cap > 0 && !vis[v])
        {
            int t = dist[v] + e[i].cost - dist[u];
            if(t == 0)
            {
                int delta = aug(v, min(e[i].cap, left));
                if(delta > 0) e[i].cap -= delta, e[i ^ 1].cap += delta, left -= delta;
                if(left == 0) return f;
            }
            else
                slk[v] = min(t, slk[v]);
        }
    }
    return f - left;
}
bool modlabel()
{
    int delta = INF;
    for(int i = 0; i < V; i++)
        if(!vis[i]) delta = min(delta, slk[i]), slk[i] = INF;
    if(delta == INF) return false;
    for(int i = 0; i < V; i++)
        if(vis[i]) dist[i] += delta;
    return true;
}
int zkw_min_cost_flow(int s, int t)
{
    src = s, sink = t;
    min_cost = 0;
    int flow = 0;
    memset(dist, 0, sizeof(dist));
    memset(slk, 0x3f, sizeof(slk));
    int tmp = 0;
    do
    {
        do
        {
            memset(vis, false, sizeof(vis));
            flow += tmp;
        }
        while((tmp = aug(src, INF)));
    }
    while(modlabel());
    return min_cost;
}
```

3.12 有上下界的网络流

```
///有上下界的网络流
//1. 建图——消除上下界
/* 设原来的源点为src, 汇点为sink. 新建一个超级源S和超级汇T, 对于原网络中的每一条边<u, v>, 上界U,
     下界L, 拆分为三条边
    1). <u, T> 容量L  2). <S, v> 容量L 3). <u, v> 容量U - L
    最后添加边<sink, src>, 容量+∞.
    在新建的网络上, 计算从S到T的最大流, 如果从S出发的每条边都是满流, 说明存在可行流,
     否则不存在可行流.
    求出可行流后, 要继续求最大流, 将该可行流还原到原网络中, 从src到sink不断增广, 直至找不到增广路.
    要求最小流: 先不连<sink, src>, 计算S到T的最大流, 然后连<sink, src>容量+∞,
     并不断从S寻找到T的增广路, 这进一步增广的流量就是最小流
    实现的时候, 要将从S连向同一结点, 同一结点连向T的多条边合并成一条(容量增加).
*/
```

3.13 树的直径

```
///树的直径
//树的直径是指树的最长简单路
/*方法一: O(|E|)
两遍BFS, 先任选一个起点找到最长路的终点, 再从终点进行BFS, 第二次找到的最长路即为树的直径.
*/
int dist[MAXV];
int bfs(int s)
{
    memset(dist, -1, sizeof(dist));
    queue<int> que;
    dist[s] = 0;
    que.push(s);
    int mx = 0, ed = s;
    while(!que.empty())
    {
        int u = que.front(); que.pop();
        for(int i = head[u]; ~i; i = e[i].next)
        {
            if(dist[e[i].to] != -1) continue;
            dist[e[i].to] = dist[u] + 1;
            if(dist[e[i].to] > mx) mx = dist[ed = e[i].to];
            que.push(e[i].to);
        }
    }
    return ed;
}

/*方法二: O(|E|)
树的直径为某个点的最长距离与次长距离之和
*/
```

3.14 最近公共祖先 LCA

```
///最近公共祖先LCA Least Common Ancestors
//较为暴力的做法:
/*
预处理: dfs深度搜索, 求出每个结点的深度.
单个查询: 查询(u, v)的LCA, 不断寻找深度较大的那个结点的父亲结点, 直至到达同一结点为止.
时间复杂度: 预处理: O(|V|), 单次查询: O(n)
*/
//Tarjian的离线算法 O(n + q)
```

```
struct edge {int next, to, lca;};
//由要查询的<u,v>构成的图
edge qe[MAXE * 2];
int qh[MAXV], qtot;
//原图
edge e[MAXE * 2]
int head[MAXV], tot;
//并查集
int fa[MAXV];
inline int find(int x)
{
    if(fa[x] != x) fa[x] = find(fa[x]);
    return fa[x];
}
bool vis[MAXV];
void LCA(int u)
{
    vis[u] = true;
    fa[u] = u;
    for(int i = head[u]; i != -1; i = e[i].next)
        if(!vis[e[i].to])
        {
            LCA(e[i].to);
            fa[e[i].to] = u;
        }
    for(int i = qh[u]; i != -1; i = qe[i].next)
        if(vis[qe[i].to])
        {
            qe[i].lca = find(eq[i].to);
            eq[i ^ 1].lca = qe[i].lca;//无向图, 入边两次
        }
}

//RMQ的在线算法 O(n log n)
/*算法描述:
    dfs扫描一遍整棵树,
     记录下经过的每一个结点(每一条边的两个端点)和结点的深度(到根节点的距离),一共2n-1次记录
    再记录下第一次扫描到结点u时的序号
    RMQ: 得到dfs中从u到v路径上深度最小的结点, 那就是LCA[u][v].
*/
struct node
{
    int u;//记录经过的结点
    int depth;//记录当前结点的深度
} vs[2 * MAXV];
bool operator < (node a, node b) {return a.depth < b.depth;}
int id[MAXV];//记录第一次经过点u时的dfn序号
void dfs(int u, int fa, int dep, int &k)
{
    vs[k] = (node) {u, dep};
    id[u] = k++;
    for(int i = head[u]; i != -1; i = e[i].next)
        if(e[i].to != fa)
        {
            dfs(e[i].to, u, dep + 1, k);
            vs[k++] = (node) {u, dep};
        }
}
//RMQ
//动态查询id[u] 到 id[v] 之间的depth最小的结点
//ST表
int Log2[MAXV * 2];
node st[MAXV * 2][32];
void pre_st(int n, node ar[])
```

```
{
    Log2[1] = 0;
    for(int i = 2; i <= n; i++) Log2[i] = Log2[i >> 1] + 1;
    for(int i = n - 1; i >= 0; i--)
    {
        st[i][0] = ar[i];
        for(int j = 1; i + (1 << j) <= n; j++)
            st[i][j] = min(st[i][j - 1], st[i + (1 << (j - 1))][j - 1]);
    }
}
int query(int l, int r)
{
    int k = Log2[r - l + 1];
    return min(st[l][k], st[r - (1 << k) + 1][k]).u;
}

void lca_init()
{
    int k = 0;
    dfs(1, -1, 0, k);
    pre_st(k, vs);
}

int LCA(int u, int v)
{
    u = id[u], v = id[v];
    if(u > v) swap(u, v);
    return query(u, v);
}
```

4 数学专题

4.1 素数 Prime

```
///素数 Prime
/*素数定理
    设素数分布函数π(n)为小于等于n的素数个数, 则有以下近似:
                                    lim_{n→∞} π(n) / (n/ln n) = 1
    .
*/
/*伪素数
    如果n是一个合数, 并且a^{n-1} ≡ 1(mod n), 则说n是一个基为a的伪素数.
*/
///涮素数
int prim[NUM], prim_num;
//O(n log n)
void pre_prime()
{
    prim_num = 0;
    for(int i = 2; i < NUM; ++i)
    {
        if(prim[i]) continue;
        prim[prim_num++] = i;
        for(int j = i + i; j < NUM; j += i) prim[j] = 1;
    }
}

//O(n)
void pre_prime()
{
    prim_num = 0;
    for(int i = 2; i < NUM; ++i)
    {
        if(!prim[i]) prim[prim_num++] = i;
        for(int j = 0; j < prim_num && i * prim[j] < NUM; ++j)
        {
            prim[i * prim[j]] = 1;
            if(i % prim[j] == 0) break;
        }
    }
}

//区间素数
/*要获得区间[L, U]内的素数, L和U很大, 但U - L不大, 那么,
    先线性涮出1到√2147483647 ≤ 46341之间所有的素数, 然后在通过已经涮好的素数涮出给定区间的素数
*/

///素数判定
//试除法: 略过偶数, 试除2到√n间的所有数O(√n)
bool isPrime(int n)
{
    if(n % 2 == 0)return n == 2;
    for(int i = 3; i * i <= n; i += 2)
        if(n % i == 0)
            return false;
    return true;
}

//Miller_Rabin O(test_num · log n)
int qpow(int x, int k, int mod) {}
//以a为基, n-1 = 2^t u,(t ≥ 1 and u is odd), 通过a^{n-1} ≡ 1(mod n)验证n是不是合数
//一定是合数返回true, 不一定返回false
```

```cpp
bool witness(LL a, LL n, LL u, LL t)
{
    LL res = qpow(a, u, n);//a^u(mod n)
    LL last = res;
    while(t--)
    {
        //res = qmult(res, res, n);
        res = res * res % n;
        if(res == 1 && last != 1 && last != n - 1) return true; //合数
        last = res;
    }
    return res != 1;
}

//是素数返回true(可能是伪素数，但概率极小，至多为2^-testnum)，合数返回false.
bool Miller_Rabin(LL n, int test_num = 50)
{
    if(n < 2)return false;
    if(n == 2)return true;
    if((n & 1) == 0) return false; //偶数
    LL u = n - 1;
    LL t = 0;
    while((u & 1) == 0) {u >>= 1; ++t;}
    while(test_num--)
    {
        LL a = rand() % (n - 1) + 1;//产生1~n-1之间的随机数
        if(check(a, n, u, t))
            return false;//合数
    }
    return true;
}
```

4.2 因式分解 Factorization 和约数

```cpp
///因式分解Factorization
///唯一分解理论
/*
```

所有正整数N皆可表示为素数之积，即 $N=\prod_{i=1}^{m} p_i^{k_i}, p_i$ 是素数.

因子的性质:

1. 因子个数函数 τ 定义为正整数n的所有正因子个数，记为 $\tau(n)$，则 $\tau(n)=\prod_{i=0}^{m}(k_i+1)$
2. 因子和函数 σ 定义为整数n的所有正因子之和，记为 $\sigma(n)$，则 $\sigma(n)=\prod_{i=1}^{m}\frac{p_i^{k_i+1}-1}{p_i-1}$
3. 因子和函数 σ 和因子个数函数 τ 是乘性函数.
4. 设 $a=\prod_{i=1}^{m} p_i^{x_i}, b=\prod_{i=1}^{m} p_i^{y_i}$，则

$$\gcd(a,b)=\prod_{i=1}^{m}\min(x_i,y_i)$$

，

$$lcm(a,b)=\prod_{i=1}^{m}\max(x_i,y_i)$$

5. 质因数个数: 10^5以内的数质因数至多7个，10^6以内的数的质因数至多8个，10^9以内的数的质因数至多10个

```cpp
*/
///分解质因数
//暴力试除法，O(√N)
int prim[NUM], tot;//素数表
vector<P> FAC(int n)
{
    vector<P> fac;
    fac.clear();
```

```
    for(int i = 0; i < tot && prim[i] * prim[i] <= n; ++i)
    {
        if(n % prim[i]) continue;
        int cnt = 0;
        while(n % prim[i] == 0)
        {
            ++cnt;
            n /= prim[i];
        }
        fac.push_back(P(prim[i], cnt));
    }
    if(n > 1) fac.push_back(P(n, 1));
    return fac;
}

int facs[NUM];
int find_fac(int n)
{
    int cnt = 0;
    for(int i = 2; i * i <= n; i += 2)
    {
        while(!(n % i))
        {
            n /= i;
            facs[cnt++] = i;
        }
        if(i == 2) --i;
    }
    if(n > 1) facs[cnt++] = n;
    return cnt;
}
///预处理1~n间所有数的约数 O(n log n)
vector<int> facs[NUM];
int prim[NUM], tot;
void pre_fac()
{
    for(int i = 2; i < NUM; ++i)
    {
        if(prim[i]) continue;
        prim[tot++] = i;
        facs[i].push_back(i);
        for(int j = i + i; j < NUM; j += i)
        {
            prim[j] = 1;
            facs[j].push_back(i);
        }
    }
}
///pollard_rho启发式方法 O(∜n)
//对较大的数整数进行分解
/*
选取[2, n - 1]间的随机数, 通过随机函数x = (x^2 + c)%n, 如果序列出现循环, 则退出;
     否则计算p = gcd(x_{i-1}, x_i), 如果p = 1, 则继续直到p > 1为止, 若p = n, 则n是素数,
     否则p是n的一个约数, 并递归分解p和n/p.
当p为素数用pollard_rho较为耗时, 所以可以先用miller_rabin判断p是否为素数.

*/
LL factor[100];//质因数分解结果(刚返回时是无序的)
int tol;//质因数的个数. 数组小标从0开始
LL qmult(LL a, LL b, LL mod)
{
    a %= mod;
    b %= mod;
    LL res = 0;
```

```
    while(b)
    {
        if(b & 1) if((res += a) >= mod) res -= mod;
        if((a <<= 1) >= mod) a -= mod;
        b >>= 1;
    }
    return res;
}
inline LL gcd(LL a, LL b)
{
    while(b) {swap(b, a = a % b);}
    if(a >= 0) return a;
    else return -a;
}

LL Pollard_rho(LL n, LL c)
{
    LL i = 1, k = 2;
    LL x = rand() % n;//0 ~ n - 1
    LL y = x;
    while(1)
    {
        //x = (qmult(x, x, n) + n - 1) % n;
        x = (x * x + c) % n;
        LL d = gcd(y - x, n);
        if(d != 1 && d != n) return d;
        if(y == x) return n;
        if(++i == k) y = x, k += k;
    }
}
//对n进行素因子分解
long long factor[110];//乱序返回
int factor_num;
void FindFactor(LL n)
{
    if(n == 0) return ;
    if(Miller_Rabin(n))//测试n是否是素数
    {
        factor[factor_num++] = n;
        return ;
    }
    LL p = n;
    int c = 107;//一般取107左右
    while(p >= n) p = Pollard_rho(p, c--);//值变化, 防止死循环
    FindFactor(p);
    FindFactor(n / p);
}


//枚举约数 O(√n)
void DIV(int n)
{
    vector<int> d;
    int i;
    for(i = 1; i * i < n; ++i)
        if(n % i == 0)
        {
            d.push_back(i);
            d.push_back(n / i);
        }
    if(i * i == n) d.push_back(i);
}
```

4.3 欧拉函数 Euler

```
///欧拉函数（Euler's totient function）
/*
对正整数n，欧拉函数是小于或等于n的数中与n互质的数的数目
通式：φ(n) = n(1 − 1/p1)(1 − 1/p2)···(1 − 1/pk)，其中p1, p2, ···, pk 为n的所有质因数，n是不为0的整数.
φ(1) = 1 （唯一和1互质的数(小于等于1)就是1本身）;
φ(p) = p − 1, p为素数
若n是质数p的k次幂，φ(n) = p^k − p^(k−1) = (p − 1)p^(k−1)，因为除了p的倍数外，其他数都跟n互质。
欧拉函数是积性函数——若m，n互质，φ(mn) = φ(m)φ(n).
当n为奇数时，φ(2n) = φ(n).
*/
///欧拉定理：对任何两个互质的正整数a, m, m ≥ 2有 a^φ(m) ≡ 1(%m)
///费马小定理：当p是质数时，为：a^(p−1) ≡ 1(%p)
///降幂公式：A^x = A^(x%φ(C)+φ(C))%C，(x > φ(C))

//求欧拉函数
int Euler(int n)
{
    int euler = n;
    for(int i = 1; i < primen; i++)
        if(n % prime[i])
        {
            euler = euler / prime[i] * (prime[i] − 1);
        }
    return euler;
}
//预处理
int euler[NUM];
bool notPrim[NUM];
int Euler()
{
    int i;
    for(i = 0; i < NUM; i++)
        euler[i] = i;
    for(i = 2; i < NUM; i++)
        if(!notPrim[i])
        {
            euler[i] = i − 1;
            for(int j = i + i; j < NUM; j += i)
            {
                notPrim[i] = true;
                euler[j] = euler[j] / i * (i − 1);
            }
        }
}
```

4.4 快速幂快速乘

```
//快速幂，快速模幂
LL qpow(LL x, int k, LL mod)
{
    LL ans = 1;
    while(k)
    {
        if(k & 1) ans = ans * x % mod;
        x = x * x  % mod;
        k >>= 1;
    }
    return ans;
}

```

```cpp
//快速模乘a × b%mod
//用于a × b可能爆LL
LL mod_mult(LL a, LL b, LL mod)
{
    LL res = 0;
    if(a >= mod) a %= mod;
    while(b)
    {
        if(b & 1)
        {
            res += a;
            if(res += mod) res -= mod;
        }
        a <<= 1;
        if(a >= mod) a -= mod;
        b >>= 1;
    }
    return res;
}
```

4.5 最大公约数 GCD

```cpp
///最大公约数gcd
/*gcd(a, b)的性质
    gcd(0, 0) = 0, gcd(a, b) = gcd(b, a), gcd(a, b) = gcd(−a, b)
    gcd(a, b) = gcd(|a|, |b|), gcd(a, 0) = |a|, gcd(a, ka) = |a|, (k ∈ Z)
    gcd(a, b) = n gcd(a, b), gcd(a, gcd(b, c)) = gcd(gcd(a, b), c)
    1. 如果a, b都是偶数, 则gcd(a, b) = 2 · gcd(a/2, b/2).
    2. 如果a是奇数, b是偶数, 则gcd(a, b) = gcd(a, b/2).
    3. 如果a, b都是奇数, 则gcd(a, b) = gcd((a − b)/2, b).
    gcd递归定理: gcd(a, b) = gcd(b, a%b)
    最大公倍数lcm(a, b) = ab/gcd(a,b)
    n个数的gcd和lcm, 记第i个数
```

$$a_i = \prod_{k=1}^{l} p_k^{g_{ik}}$$

```
    , 则
```

$$\gcd(a_1, a_2, \cdots, a_n) = \prod_{k=1}^{l} p_k^{\min\{g_{1k}, g_{2k}, \cdots, g_{nk}\}}$$

```
    ,
```

$$\mathrm{lcm}(a_1, a_2, \cdots, a_n) = \prod_{k=1}^{l} p_k^{\max\{g_{1k}, g_{2k}, \cdots, g_{nk}\}}$$

```
    .
    一段区间[l, r](r = l → n)的gcd最多变化log次
    1, 2, · · · , n的lcm为, 如果n是某质数p的幂, 则lcm(n) = lcm(n − 1) × p, 否则lcm(n) = lcm(n − 1).
*/
//欧几里得算法O(log n)
//递归
int gcd(int a, int b) {return b ? gcd(b, a % b) : a;}
//循环
int gcd(int a, int b) {while(b) swap(b, a = a % b); return a;}
//二进制GCD
int gcd(int a, int b)
{
    if(a == 0)  return b;
    if(b == 0)  return a;
    if(!(a & 1) && !(b & 1))    return gcd(a >> 1, b >> 1) << 1;
    else if(!(b & 1))   return gcd(a, b >> 1);
    else if(!(a & 1))   return gcd(a >> 1, b);
    else return gcd(abs(a − b) >> 1, min(a, b));
}
//小数的gcd
//EPS控制精度
double fgcd(double a, double b)
```

```
{
    if(-EPS < b && b < EPS) return a;
    return fgcd(b, fmod(a, b));
}
///扩展欧几里得算法
void exgcd(int a, int b, int &g, int &x, int &y)
{
    if(b) exgcd(b, a % b, g, y, x), y -= a / b * x;
    else x = 1, y = 0, g = a;
}
//应用
//1. 求解ax + by = c的x的最小正整数解
//x的通解为x0 + b/gcd(a, b) * k
int solve(int a, int b, int c)
{
    int g = exgcd(a, b, x, y), t = b / g;
    if(c % g) return -1;//c % gcd(a, b) != 0无解
    int x0 = x * c / g;
    x0 = ((x0 % t) + t) % t;
    int y0 = (c - a * x0) / b;
    return x0;
}
//2. 求解a关于p的逆元
//

/*题目:
1.  给n个数, q个询问, 每个询问查询区间[l, r]中每个子区间的区间gcd之和,
    即查询$\sum_{i=l}^{r}\sum_{j=i}^{r}\gcd(a_i,a_{i+1},\cdots,a_j).(1\le n,q\le 10^4)$
    来源: 2015年多校第八场1002, hdu5381  The sum of gcd
    标签: gcd, 莫队, ST, 二分
    做法: 由于以l为左端点的所有区间中, 它们的gcd最多变化$\log a_l$次, 并且gcd是递减的. 因此用ST表预处理,
    可以在$O(1)$时间内求出任意区间的gcd, 然后用二分求出对于每个左端点l和每个右端点r,
    找出每种gcd的范围, 这样就可以在$\log N$时间内求出任意以l为左端点, 或以r为右端点,
    到某位置的所有gcd之和, 即$\sum_{i=l}^{r}\gcd(a_l,a_{l+1},\cdots,a_i)$, 最后离线询问, 用莫队求解. 时间复杂度:
    $O(N\log^2 N)$.
*/
```

4.6 莫比乌斯反演 Mobius

```
///莫比乌斯反演 Mobius
//mobius函数
/*
```

$$\mu(x)=\begin{cases}1 & x=1\\ (-1)^r & x=p_1\cdot p_2\cdots p_r,\text{其中}p_i(i=1,2,\cdots r)\text{是素数}\\ 0 & \text{其他}\end{cases}$$

```
*/
//莫比乌斯反演
//
```

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

```
//
```

$$\sum_{d|n}\mu(d)=\begin{cases}1 & n=1\\ 0 & n\neq 1\end{cases}$$

```
//$\sum_{d|n}\phi(d)=n,\phi(d)$为欧拉函数
//$\phi(n)=n\sum_{d|n}\mu(d)/d$

//使用1
/*
```

```
    ∑gcd(i,j) == D(1 ≤ i ≤ a, 1 ≤ j ≤ b, a ≤ b), 即求gcd(i, j)等于d的对数, ⌊x⌋表示下取整
    ∑_{i=1}^{a} ∑_{j=1}^{b} gcd(i,j) == D
    ⇒ ∑_{i=1}^{⌊a/D⌋} ∑_{j=1}^{⌊b/D⌋} gcd(i,j) == 1
    ⇒ ∑_{i=1}^{⌊a/D⌋} ∑_{d|gcd(i,j)} μ(d), 使用mobius函数和的性质替换gcd(i, j)==1
    ⇒ ∑_{d=1}^{⌊a/D⌋} μ(d)⌊⌊a/D⌋/d⌋ · ⌊⌊b/D⌋/d⌋, d|gcd(i,j) ⇔ d|i ∪ d|j
    D == 1, ∑_{d=1}^{a} μ(d) · ⌊a/d⌋ · ⌊b/d⌋
*/

//使用2
/*
    ∑_{i=1}^{a} ∑_{j=1}^{b} gcd(i,j), a ≤ b
    ⇒ ∑_{d=1}^{a} ∑_{i=1}^{⌊a/d⌋} ∑_{j=1}^{⌊b/d⌋} d · (gcd(i,j) == 1)
    ⇒ ∑_{d=1}^{a} ∑_{d'=1}^{⌊a/d⌋} d · μ(d') · ⌊a/(dd')⌋ · ⌊b/(dd')⌋, 使用1
    ⇒ ∑_{d=1}^{a} ∑_{d|D} d · μ(D/d) · ⌊a/D⌋ · ⌊b/D⌋, D = dd'
    ⇒ ∑_{D=1}^{a} ⌊a/D⌋ · ⌊b/D⌋ · (id · μ)(D)
    ⇒ ∑_{D=1}^{a} ⌊a/D⌋ · ⌊b/D⌋ · φ(D), id · μ = φ
*/

///积性函数
//定义在正整数集上的函数f(n)(称为算术函数), 若gcd(a,b) = 1时有f(a) · f(b) = f(a · b), 则称f(x)为积性函数。
//一个显然的性质: (非恒等于零的) 积性函数f(n)必然满足f(1) = 1.
//定义逐点加法(f + g)(n) = f(x) + g(x), f(x · g) = f(x) · g(x).
//一个比较显然的性质: 若f, g均为积性函数, 则f · g也是积性函数。
//积性函数的求值: n = ∏p_i^{a_i} 则f(n) = ∏f(p_i^{a_i}), 所以只要解决n = p^a时f(n)的值即可.

//常见积性函数有:
//恒为1的常函数1(n) = 1,
//恒等函数id(n) = n,
//单位函数ε(n) = (n == 1), (这三个都是显然为积性)
//欧拉函数φ(n)(只要证两个集合相等就能证明积性)
//莫比乌斯函数μ(n) (由定义也是显然的)
//因子个数函数τ
//因子和函数σ
//μ · id = φ
void pre_mobius()
{
    mu[1] = 1;
    for(int i = 2; i < NUM; i++)
        if(!mu[i])
        {
            mu[i] = -1;
            for(int j = i + i; j < NUM; j += i)
                if((j / i) % i == 0)
                    mu[j] = 2;
                else
                {
                    if(mu[j] == 0) mu[j] = -1;
                    else mu[j] = -mu[j];
                }
        }
        else if(mu[i] == 2 || mu[i] == -2) mu[i] = 0;
```

```
}
```

4.7 逆元 Inverse

```
///逆元inverse
//定义：如果a·b≡1(%MOD)，则b是a的逆元(模逆元，乘法逆元)
//a的逆元存在条件：gcd(a, MOD) == 1
//性质：逆元是积性函数，如果c=a·b，则 inv[c]=inv[a]·inv[b]%MOD
//方法一：  循环找解法(暴力)
//O(n)  预处理inv[1-n]：O(n^2)
LL getInv(LL x, LL MOD)
{
    for(LL i = 1; i < MOD; i++)
        if(x * i % MOD == 1)
            return i;
    return -1;
}

//方法二：  费马小定理和欧拉定理
//费马小定理:a^(p-1)≡1(%p)，其中p是质数，所以a的逆元是a^(p-2)%p
//欧拉定理:x^φ(m)≡1(%m) x与m互素，m是任意整数
//O(logn)(配合快速幂)，预处理inv[1-n]：O(nlogn)
LL qpow(LL x, LL k, LL MOD) {....}
LL getInv(LL x, LL MOD)
{
    //return qpow(x, euler_phi(MOD) - 1, MOD);
    return qpow(x, MOD - 2, MOD);//MOD是质数
}

//方法三：扩展欧几里得算法
//扩展欧几里得算法可解决  a·x+b·y=gcd(a,b)
//所以a·x%MOD=gcd(a,b)%MOD(b=MOD)
//O(logn)，预处理inv[1-n]：O(nlogn)
inline void exgcd(LL a, LL b, LL &g, LL &x, LL &y)
{
    if(b) exgcd(b, a % b, g, y, x), y -= (a / b) * x;
    else g = a, x = 1, y = 0;
}

LL getInv(LL x, LL mod)
{
    LL g, inv, tmp;
    exgcd(x, mod, g, inv, tmp);
    return g != 1 ? -1 : (inv % mod + mod) % mod;
}

//方法四：积性函数
//已处理inv[1] —— inv[n - 1]，求inv[n]，(MOD > n) (MOD为质数,不存在逆元的i干扰结果)
//MOD=x·n-y(0≤y<n)⇒x·n=y(%MOD)⇒x·n·inv[y]=y·inv[y]=1(%MOD)
//所以inv[n]=x·inv[y](x=MOD-MOD/n,y=MOD%n)
//O(logn) 预处理inv[1-n]：O(n)
LL inv[NUM];
void inv_pre(LL mod)
{
    inv[0] = inv[1] = 1LL;
    for(int i = 2; i < NUM; i++)
        inv[i] = (mod - mod / i) * inv[mod % i] % mod;
}
LL getInv(LL x, LL mod)
{
    LL res = 1LL;
    while(x > 1)
```

```
    {
        res = res * (mod - mod / x) % mod;
        x   = mod % x;
    }
    return res;
}
//方法五：积性函数+因式分解
//预处理出所有质数的的逆元，采用exgcd来实现素数O(log n)求逆
//采用质因数分解，  可在O(log n)求出任意一个数的逆元
//预处理O(n log n)，单个O(log n)
```

4.8 模运算 Module

```
/*
 模(Module)
 1. 基本运算
    Add: (a + b) % p = (a % p + b % p) % p
    Subtract:(a - b) % p = ((a%p - b%p)%p + p)%p
    Multiply:(a * b) % p = ((a % p) * (b % p)) % p
    Dvidive: (a / b) % p = (a * b^-1) % p, b^-1是b关于p的逆元
    Power: (a^b)%p = ((a%p)^b)%p

    对一个数连续取模，有效的取模次数小于O(log n)
 2. 推论
    若a ≡ b(%p), c ≡ d(%p)，则(a + c) ≡ (b + d)(%p), (a - c) ≡ (b - d)(%p), (a * c) ≡ (b * d)(%p), (a/c) ≡ (b/d)(%p)

 3. 费马小定理
    若p是素数，对任意正整数x，有 x^p ≡ x(%p).
 4. 欧拉定理
    若p与x互素，则有x^φ(p) ≡ 1(%p).
 5. n! = ap^e, gcd(a, p) == 1, p是素数
    e = (n/p + n/p^2 + n/p^3 + ···)(a不能被p整除)
    威尔逊定理：(p - 1)! ≡ -1(%p)(当且仅当p是素数)
    n!中不能被p整除的数的积:n! = (p - 1)!^(n/p) × (n  mod  p)!
    n!中能被p整除的项为:p, 2p, 3p, ... , (n/p)p, 除以p得到1,2,3,...,n/p(问题从缩减到n/p)
    在O(p)时间内预处理除0 ≤ n < p范围内中的n! mod p的表
    可在O(log_p n)时间内算出答案
    若不预处理，复杂度为O(p log_p n)
 */
int fact[MAX_P];//预处理n! mod p的表.O(p)
//分解n! = a  p^e.返回a % p.  O(log_p n)
int mod_fact(int n, int p, int &e)
{
    e = 0;
    if(n == 0) return 1;
    //计算p的倍数的部分
    int res = mod_fact(n / p, p, e);
    e += n / p;
    //由于(p - 1)! ≡ -1,因此只需知n/p的奇偶性
    if(n / p % 2) return res * (p - fact[n % p]) % p;
    return res * fact[n % p] % p;
}

/*
 6. n! = t(p^c)^u, gcd(t, p^c) == 1, p是素数
    1 ~ n中不能被p整除的项模p^c，以p^c为循环节，预处理出n!%p^c的表
    1 ~ n中能被p整除的项，提取 n/p 个p出来，剩下阶乘(n/p)!，递归处理
    最后，t还要乘上p^u
 */
LL fact[NUM];
LL qpow(LL x, LL k, LL mod);
inline void pre_fact(LL p, LL pc)//预处理n!%p^c, O(p^c)
```

```
{
    fact[0] = fact[1] = 1;
    for(int i = 2; i < pc; i++)
    {
        if(i % p) fact[i] = fact[i - 1] * i % pc;
        else fact[i] = fact[i - 1];
    }
}
//分解n! = t(p^c)^u, n!%pc = t · p^u%pc)
inline void mod_factorial(LL n, LL p, LL pc, LL &t, LL &u)
{
    for(t = 1, u = 0; n; u += (n /= p))
        t = t * fact[n % pc] % pc * qpow(fact[pc - 1], n / pc, pc) % pc;
}
/*
 7. 大组合数求模, mod不是质数
    求C_n^m%mod
    1) 因式分解:mod = p1^c1 p2^c2 ··· pk^ck
    2) 对每个因子p^c, 求C_n^m%p^c = n!%p^c / (m!%p^c (n-m)!%p^c) %p^c
    3) 根据中国剩余定理求解答案(注: 逆元采用扩展欧几里得求法)
 */
LL fact[NUM];
LL prim[NUM], prim_num;
LL pre_prim();
LL pre_fact(LL p, LL pc);
LL mod_factorial(LL n, LL p, LL pc, LL &t, LL &u);
LL qpow(LL x, LL k, LL mod);
LL getInv(LL x, LL mod);

LL C(LL n, LL m, LL mod)
{
    if(n < m) return 0;
    LL p, pc, tmpmod = mod;
    LL Mi, tmpans, t, u, tot;
    LL ans = 0;
    int i, j;
    //将mod因式分解,mod = p1^c1 p2^c2 ··· pk^ck
    for(i = 0; prim[i] <= tmpmod; i++)
        if(tmpmod % prim[i] == 0)
        {
            for(p = prim[i], pc = 1; tmpmod % p == 0; tmpmod /= p)
                pc *= p;
            //求C_n^k%p^c
            pre_fact(p, pc);
            mod_factorial(n, p, pc, t, u);//n!
            tmpans = t;
            tot = u;
            mod_factorial(m, p, pc, t, u);//m!
            tmpans = tmpans * getInv(t, pc) % pc;//求逆元: 采用扩展欧几里得定律
            tot -= u;
            mod_factorial(n - m, p, pc, t, u);//(n - m)!
            tmpans = tmpans * getInv(t, pc) % pc;
            tot -= u;
            tmpans = tmpans * qpow(p, tot, pc) % pc;
            //中国剩余定理
            Mi = mod / pc;
            ans = (ans + tmpans * Mi % mod * getInv(Mi, pc) % mod) % mod;
        }
    return ans;
}

/*
 8. 大组合数求模, mod是素数, Lucas定理
    Lucas定理: C_n^m%mod = C_{n/mod}^{m/mod} · C_{n%mod}^{m%mod} %mod
```

```
    采用O(n)方法预处理0~n-1的n!%mod和每个数的逆元，则可在O(log n)时间求出C_n^k%mod
*/
LL fact[NUM], inv[NUM];
void Lucas_init(LL mod);//预处理
LL Lucas(LL n, LL m, LL mod) //mod是质数
{
    LL a, b, res = 1LL;
    while(n && m)
    {
        a = n % mod, b = m % mod;
        if(a < b) return 0LL;
        res = res * fact[a] % mod * inv[fact[b] * fact[a - b] % mod, mod] % mod;
        n /= mod, m /= mod;
    }
    return res;
}
```

4.9 幂 p 的原根

```
///幂模p与原根primitive root
//a模p的次数(或阶)指e_p(a) = (使得a^e ≡ 1(%p)的最小指数e ≥ 1)
//(次数整除性质)，设a是不被素数p整除的整数，假设a^n ≡ 1(%p)，则次数e_p(a)整除n，特别地，
    次数e_p(a)总整除p-1.
//原根：具有最高次数e_p(g) = p - 1的数g被称为模p的原根
//原根定理：每个素数p都有恰好φ(p - 1)个原根
//求最小原根：原根分布较广，而且最小的原根通常也比较小，
    可以从小到大通过枚举每个正整数来快速的寻找一个原根. 对于一个待检查的p，对于p-1的每一个素因子a，
    检查g^((p-1)/a) ≡ 1(%p)是否成立，如果成立则不是.
//1e9 + 7的最小原根为5, 1e9 + 7的最小原根为13
LL qpow(LL x, LL k, LL mod);
bool g_test(LL g, LL p, vector<LL> &factor)
{
    for(int i = 0; i < (int)factor.size(); i++)
        if(qpow(g, (p - 1) / factor[i], p) == 1)
            return false;
    return true;
}
LL primitive_root(LL p)
{
    vector<LL> factor;
    //求p-1的素因子
    LL tmp = p - 1;
    for(LL i = 2; i * i <= tmp; i++)
        if(tmp % i == 0)
        {
            factor.push_back(i);
            while(tmp % i == 0) tmp /= i;
        }
    if(tmp != 1) factor.push_back(tmp);
    LL g = 1;
    while(true)
    {
        if(g_test(g, p, factor))
            return g;
        g++;
    }
}
```

4.10 中国剩余定理和线性同余方程组

```
/*线性同余方程
 a_i × x ≡ b_i(% m_i) (1 ≤ i ≤ n)
 如果方程组有解，那么一定有无穷有无穷多解，解的全集可写为x ≡ b(% m)的形式.
 对方程逐一求解.令b = 0, m = 1;
 1.x ≡ b(% m)可写为x = b + m · t;
 2.带入第i个式子: a_i(b + m · t) ≡ b_i(% m_i),即a_i · m · t ≡ b_i − a_i · b(% m_i)
 3.当gcd(m_i, a_i · m)无法整除b_i − a_i · b时原方程组无解,否则用exgcd，求出满组条件的最小非负整数t,

中国剩余定理:
    对x ≡ a_i(% m_i)(1 ≤ i ≤ n),其中m_1, m_2, ···, m_n两两互素， a_1, a_2, ···, a_n是任意整数，则有解:
    M = ∏m_i, b = ∑_i^n a_i M_i^{-1} M_i(M_i = M/m_i)
*/
int gcd(int a, int b);
int getInv(int x, int mod);
pair<int, int> linear_congruence(const vector<int> &A, const vector<int> &B, const vector<int> &M)
{
    //初始解设为表示所有整数的x ≡ 0(% 1)
    int x = 0, m = 1;
    for(int i = 0; i < A.size(); i++)
    {
        int a = A[i]*m, b = B[i] − A[i] * x, d = gcd(M[i], a);
        if(b % d == 0) return make_pair(0, −1);//无解
        int t = b/d * getInv(a / d, M[i] / d) % (M[i] / d);
        x = x + m * t;
        m *= M[i] / d;
    }
    return make_pair(x % m, m);
}
```

4.11 伪随机数的生成–梅森旋转算法

```
//伪随机数生成——梅森旋转算法（Mersenne twister）
/*是一个伪随机数发生算法. 对于一个k位的长度，Mersenne Twister会在[0,2^k − 1](1 <= k <= 623)
    的区间之间生成离散型均匀分布的随机数.梅森旋转算法的周期为梅森素数2^19937 − 1*/
//32位算法
int mtrand_init = 0;
int mtrand_index;
int mtrand_MT[624];
void mt_srand(int seed)
{
    mtrand_index = 0;
    mtrand_init = 1;
    mtrand_MT[0] = seed;
    for(int i = 1; i < 624; i++)
    {
        int t = 1812433253 * (mtrand_MT[i − 1] ^ (mtrand_MT[i − 1] >> 30)) + i;//0x6c078965
        mtrand_MT[i] = t & 0xffffffff;   //取最后的32位赋给MT[i]
    }
}

int mt_rand()
{
    if(!mtrand_init)
        srand((int)time(NULL));
    int y;
    if(mtrand_index == 0)
    {
        for(int i = 0; i < 624; i++)
        {
            //2^31 −1 = 0x7fff ffff  2^31 = 0x8000 0000
            int y = (mtrand_MT[i] & 0x80000000) + (mtrand_MT[(i + 1) % 624] & 0x7fffffff);
            mtrand_MT[i] = mtrand_MT[(i + 397) % 624] ^ (y >> 1);
```

```
            if(y & 1) mtrand_MT[i] ^= 2567483615; // 0x9908b0df
        }
    }
    y = mtrand_MT[mtrand_index];
    y = y ^ (y >> 11);
    y = y ^ ((y << 7) & 2636928640);    //0x9d2c5680
    y = y ^ ((y << 15) & 4022730752);   // 0xefc60000
    y = y ^ (y >> 18);
    mtrand_index = (mtrand_index + 1) % 624;
    return y;
}
```

4.12 位运算

```
///异或Xor
/*
性质:   1. 0⊕1=0,1⊕0=1,0⊕0=0,1⊕1=1
        2. (交换律) a⊕b=b⊕a
        3. (结合律) (a⊕b)⊕c=a⊕(b⊕c)
        4. a⊕a=0
        5. 0⊕a=a
        6. (二进制分解) a⊕b=∑_{i=0}^{∞} a_i⊕b_i, 其中 a_i,b_i是数a, b的二进制表达的第i位
            不同位置上运算互不影响
        7. 若a为偶数, 则 a⊕(a+1)=1,a⊕1=(a+1),(a+1)⊕1=a
*/

///求所有子集
//如: 101的子集有 101, 100, 001, 000
void SubSet(int x)
{
    for(int i = x; ; i = (i - 1) & x)
    {
        //do someting
        if(!i) break;
    }
}
///求最后一个1所在的位置
//如lowbit(10010) = 10
void lowbit(int x)
{
    return x & (-x);
}
///枚举1,2,···,(1 << n) - 1中1的个数为k的所有元素(k ≠ 0)
void kSubSet(int n, int k)
{
    int x = (1 << k) - 1;
    while(x < (1 << n))
    {
        //do something
        int tx = x & -x, ty = tx + x;
        x = (((x ^ ty) >> 2) / tx) | ty;
    }
}
```

4.13 博弈论 Game Theory

```
///博弈论 Game Theory
/*
```

SG组合游戏

SG组合游戏定义: 游戏两人参与, 轮流决策, 最优决策;
当有人无法决策时, 游戏结束, 无法决策的人输;
游戏可在有限步内结束, 不会多次抵达同一状态, 没有平局;
游戏某一状态可以到达的状态集合存在且确定.

将游戏的每一个状态看做结点, 状态间的转移看做是有向边, 则SG组合游戏是一个无环图.

必胜态和必败态: 必胜态(N-position): 当前玩家有策略使得对手无论做什么操作, 都能保证自己胜利.
必败态(P-position): 对手的必胜态.
组合游戏中某一状态不是必胜态就是必败态.
对任意的必胜态, 总存在一种方式转移到必败态.
对任意的必败态, 只能转移到必胜态.

找出必败态和必胜态: 1. 按照规则, 终止状态设为必败(胜)态.
2. 将所有能到达必败态的状态标为必胜态.
3. 将只能到达必胜态的状态标为必败态.
4. 重复2-3, 直到不再产生必败(胜)态.

游戏的和: 考虑任意多个同时进行的SG组合游戏, 这些SG组合游戏的和是这样一个SG组合游戏,
在它进行的过程中, 游戏者可以任意挑选其中的一个单一游戏进行决策, 最终, 没有办法进行决策的人输.

//

SG函数(the Sprague-Grundy function)

定义: 游戏状态为x, sg(x)表示状态x的sg函数值, $sg(x) = \min\{n|n \in N \cap n \notin F(x)\}$,
F(x)表示x能够达到的所有状态. 一个状态为必败态则sg(x)=0.

SG定理: 如果游戏G由n个子游戏组成, $G = G_1 + G_2 + G_3 + \cdots + G_n$, 并且第i个游戏sg函数值为$sg_i$,
则游戏G的sg函数值为$g = sg_1 \oplus sg_2 \oplus \cdots \oplus sg_n$

SG函数与组合游戏: sg(x)为0, 则状态x为必败态; 否则x为必胜态.

//

Nim游戏: 两名游戏者从N堆石子中轮流取石子, 每次任选一堆石子从中取走至少1个石子,
取走最后一个石子的人胜利.

$SG(i) = X(i = 1, 2, \cdots N)$, 当且仅当$SG(tot) = SG(1) \oplus SG(2) \oplus \cdots SG(N)$

定理: 在我们每次只能进行一步操作的情况下, 对于任何的游戏的和,
我们若将其中的任一单一SG组合游戏换成数目为它的SG值的一堆石子, 该单一
SG组合游戏的规则变成取石子游戏的规则(可以任意取, 甚至取完), 则游戏的和的胜负情况不变. 这样,
所有的游戏的和都等价成nim游戏.

anti-nim游戏: 游戏规则和nim游戏一样, 除了最后一个取走石子的人输.

SJ定理: 对于任意一个Anti-SG游戏, 如果我们规定当局面中所有的单一游戏的SG值为0时, 游戏结束,
则先手必胜当且仅当: (1) 游戏的SG函数不为0且游戏中某个单一游戏的SG函数大于1; (2) 游戏的
SG函数为0且游戏中没有单一游戏的SG函数大于1.

Every-SG游戏: 对任何没有结束的单一游戏, 决策者都必须对该游戏进行一步决策, 其他规则与普通SG游戏一样.

定义step函数:

$$step(v) = \begin{cases} 0 & v\text{为终止状态} \\ \max((step(u)) + 1 & SG(v) > 0 \cap \mathtt{u}\ \text{为}\ \mathtt{v}\ \text{的后继状态} \cap SG(u) = 0 \\ \min(step(u)) + 1 & SG(v) = 0 \cap \mathtt{u}\ \text{为}\ \mathtt{v}\ \text{的后继状态} \end{cases}$$

定理: 对于Every-SG游戏先手必胜当且仅当单一游戏的中最大的step为奇数.

翻硬币游戏: N枚硬币排成一排, 有的正面朝上, 有的反面朝上. 我们从左开始对硬币按1到N编号.
游戏者根据某些约束翻硬币(如: 每次只能翻一或两枚, 或者每次只能翻连续的几枚), 但他所翻动的硬币中,
最右边的必须是从正面翻到反面. 谁不能翻谁输.

结论: 局面的SG值为局面中每个正面朝上的棋子单一存在时的SG值的异或和.

树的删边游戏 给出一个有N个点的树, 有一个点作为树的根节点. 游戏者轮流从树中删去边, 删去一条边后,
不与根节点相连的部分将被移走. 谁无路可走谁输.

定理: 叶子节点的SG值为0; 中间节点的SG值为它的所有子节点的SG值加1后的异或和.

无向图的删边游戏: 一个无相联通图, 有一个点作为图的根. 游戏者轮流从图中删去边, 删去一条边后,
不与根节点相连的部分将被移走. 谁无路可走谁输.

著名的定理——Fusion Principle: 我们可以对无向图做如下改动: 将图中的任意一个偶环缩成一个新点,
任意一个奇环缩成一个新点加一个新边; 所有连到原先环上的边全部改为与新点相连.
这样的改动不会影响图的SG值.

Crazy Nim: (Gym 100338D)

有三堆石子, 分别有a, b, c个石子($a \neq b, a \neq c, b \neq c, 0 < a, b, c \leq 10^9$),
两人轮流从任意一堆石子中拿任意数量的石子, 要求如果拿完后还有石子,
则石子数不能与另外两堆的数量相同. 谁不能拿谁输.

后手必赢当且仅当$(a+1)\oplus(b+1)\oplus(c+1)$
*/

4.14 快速傅里叶变换和数论变换 (FFT 和 NTT)

///快速傅里叶变换FFT(Fast Fourier Transformation)
/*原理:
DFT(离散傅里叶变换), 变换公式:

$$\begin{cases} X(k)=\sum_{i=0}^{N-1}x(i)W_N^{ik} & k=0,1,\cdots,N-1 \\ W_N=e^{-j\frac{2\pi}{N}} \end{cases}$$

W_N被称为旋转因子, 有如下性质:
1. 对称性: $(W_N^{ik})^*=W_N^{-ik}$
2. 周期性: $W_N^{ik}=W_N^{(i+N)k}=W_N^{i(N+k)}$
3. 可约性: $W_N^{ik}=W_{mN}^{mik},W_N^{ik}=W_{\frac{N}{m}}^{\frac{ik}{m}}$
所以: $W_N^{i(N-k)}=W_N^{(N-i)k}=W_N^{-ik},W_N^{N/2}=-1,W_N^{k+N/2}=-W_N^k$
IDFT(DFT逆变换), 变换公式:

$$x(k)=\frac{1}{N}\sum_{i=1}^{N-1}X(i)W_N^{-ik}$$

(基2)FFT推导:

$$\begin{aligned} X(k)=&\sum_{i=0}^{N-1}x(i)W_N^{ik} \\ =&\sum_{r=0}^{N/2-1}x(2r)W_N^{2rk}+\sum_{r=0}^{N/2-1}x(2r+1)W_N^{(2r+1)k} \\ =&\sum_{r=0}^{N/2-1}x(2r)W_{N/2}^{rk}+W_N^k\sum_{r=0}^{N/2-1}x(2r+1)W_{N/2}^{rk} \\ =&\begin{cases}\sum_{r=0}^{N/2-1}x(2r)W_{N/2}^{rk}+W_N^k\sum_{r=0}^{N/2-1}x(2r+1)W_{N/2}^{rk} & k<N/2 \\ \sum_{r=0}^{N/2-1}x(2r)W_{N/2}^{rk'}-W_N^{k'}\sum_{r=0}^{N/2-1}x(2r+1)W_{N/2}^{rk'} & k\geq N/2,k'=k-N/2\end{cases}\end{aligned}$$

所以通过计算

$$X_1(k)=\sum_{r=0}^{N/2-1}x(2r)W_{N/2}^{rk},X_2(k)=\sum_{r=0}^{N/2-1}x(2r+1)W_{N/2}^{rk},k<N/2$$

可以计算得

$$\begin{cases} X(k) & =X_1(k)+W_N^kX_2(k) \\ X(k+N/2) & =X_1(k)-W_N^kX_2(k)\end{cases}\quad k<N/2$$

DFT变换满足cyclic convolution的性质, 即
定义循环卷积c=a(*)b:

$$c_r=\sum_{x+y=r(\%N)}a_xb_y$$

则有: DFT(a(*)b) = DFT(a) · DFT(b), 所以a(*)b = DFT^{-1}(DFT(a) · DFT(b))
注意: FFT是cyclic的, 需要保证高位有足够多的0
FFT算法限制, n必须是2的幂
FFT是定义在复数上的, 因此与整数变换可能有精度误差
*/
//FFT常被用来为多项式乘法加速, 即在$O(n\log n)$复杂度内完成多项式乘法
//也需要用FFT算法来解决需要构造多项式相乘来进行计数的问题
//#incldue <complex>
//typedef std::complex<double> Complex;
struct Complex//复数类, 可以直接用STL库中的complex<double>
{
    double r, i;
    Complex(double _r = 0.0, double _i = 0.0) {r = _r, i = _i;}
    Complex operator + (const Complex &b) const {return Complex(r + b.r, i + b.i);}
    Complex operator − (const Complex &b) const {return Complex(r − b.r, i − b.i);}
    Complex operator * (const Complex &b) const
    {

```
        return Complex(r * b.r - i * b.i, r * b.i + i * b.r);
    }
    double real() {return r;}
    double image() {return i;}
};
void brc(vector<Complex> &p, int N)//蝶形变换, 交换位置i与逆序i, 如N=2^3, 交换p[011=3]与p[110=6]
{
    int i, j, k;
    for(i = 1, j = N >> 1; i < N - 2; i++)
    {
        if(i < j) swap(p[i], p[j]);
        for(k = N >> 1; j >= k; k >>= 1) j -= k;
        if(j < k) j += k;
    }
}
void FFT(vector<Complex> &p, int N, int op)//op = 1表示DFT傅里叶变换, op=-1表示傅里叶逆变换
{
    brc(p, N);
    double p0 = PI * op;
    for(int h = 2; h <= N; h <<= 1, p0 *= 0.5)
    {
        int hf = h >> 1;
        Complex unit(cos(p0), sin(p0));
        for(int i = 0; i < N; i += h)
        {
            Complex w(1.0, 0.0);
            for(int j = i; j < i + hf; j++)
            {
                Complex u = p[j], t = w * p[j + hf];
                p[j] = u + t;
                p[j + hf] = u - t;
                w = w * unit;
            }
        }
    }
}
//Polynomial multiplication多项式相乘 $\vec{X} \times \vec{Y} = \vec{Z}$
void polynomial_multi(const vector<int> &a, const vector<int> &b, vector<int> &res, int n)
{
    int N = 1;
    int i = 0;
    while(N < n + n) N <<= 1;//FFT的项数必须是2的幂
    vector<Complex> A(N, Complex(0.0)), B(N, Complex(0.0)), D(N);
    for(i = 0; i < (int)a.size(); i++)A[i] = Complex(a[i], 0.0);
    for(i = 0; i < (int)b.size(); i++) B[i] = Complex(b[i], 0.0);
    FFT(A, N, 1);
    FFT(B, N, 1);
    for(i = 0; i < N; i++) D[i] = A[i] * B[i];
    FFT(D, N, -1);
    for(i = 0, res.clear(); i < N; i++) res.PB(round(D[i].real() / N));
}

/*
应用1: 给一个01串S, 求有多少对(i, j, k)($i < j < k$)使$S_i = S_j = S_k = 1$, 且j - i = k - j
*/


//数论变换(Number Theory Transformation, NTT)
/*NTT的推导:
0. DFT变换公式: $A(k) = \sum_{i=0}^{N-1} a(i)\varpi^{ik}$
    IDFT变换公式: $a(k) = N^{-1} \sum_{i=0}^{N-1} A(i)\varpi^{-ik}$
```

1. 周期性：由于

$$
\begin{aligned}
& & A(k)\cdot B(k) &= C(k)\\
\Rightarrow & & [\sum_{i=0}^{N-1} a(i)\varpi^{ik}]\cdot[\sum_{j=0}^{N-1} b(j)\varpi^{jk}] &= \sum_{i=0}^{N-1} c(i)\varpi^{ik}\\
\Rightarrow & & \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} a(i)b(j)\varpi^{(i+j)k} &= \sum_{i=0}^{N-1}(\sum_{x+y=i(\%N)} a(x)b(y))\varpi^{ik}\\
\Rightarrow & \sum_{j=0}^{i} a(i)b(i-j)\varpi^{ik}+\sum_{j=i+1}^{N-1} a(i)b(N+i-j)\varpi^{(i+N)k} &= \sum_{x+y=i(\%N)} a(x)b(y)\varpi^{ik},\ (\forall i\in[0,N-1])\\
\Rightarrow & \forall i\in[0,N-1], k\in[0,N-1], \varpi^{(i+N)k} &= \varpi^{ik}\\
\Rightarrow & \varpi\text{具有周期为 N 的周期性，即}\varpi^{N}=1
\end{aligned}
$$

2. 求和引理：若要实现逆变换，则有：

$$
\begin{aligned}
A(k) &= \sum_{i=0}^{N-1} a(i)\varpi^{ik}\\
a(k) &= N^{-1}\sum_{i=0}^{N-1}[\sum_{j=0}^{N-1} a(j)\varpi^{ij}]\varpi^{-ik}\\
&= N^{-1}\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} a(j)\varpi^{i(j-k)}\\
&= N^{-1}\sum_{i=0}^{N-1} a(k)+N^{-1}\sum_{j\neq k} a(j)[\sum_{i=0}^{N-1}(\varpi^{j-k})^{i}]
\end{aligned}
$$

所以，有$\forall i\neq 0, \sum_{j=0}^{N-1}(\varpi^{i})^{j}=\frac{(\varpi^{i})^{N}-1}{(\varpi^{i})-1}=\frac{(\varpi^{N})^{i}-1}{(\varpi^{i})-1}=0$

即$\varpi^{N}=1$，且$\varpi^{i}\neq 1(i\neq 0)$

3. FFT的分治计算：

由$N=\prod_{i=1}^{m} p_i^{k_i}$，令$n=p_i(i=1,2,\cdots,m), N'=\frac{N}{n}$

则

$$
\begin{aligned}
& A(k+pN')\\
= & \sum_{i=0}^{N-1} a(i)\varpi^{i(k+pN')}\\
= & \sum_{i=0}^{n-1}[\sum_{j=0}^{N'-1} a(i+jn)\varpi^{(i+jn)(k+pN')}]\\
= & \sum_{i=0}^{n-1}[\sum_{j=0}^{N'-1} a(i+jn)\varpi^{ik+jnk+ipN'+jpN}]\\
= & \sum_{i=0}^{n-1}[\varpi^{i(k+pN')}\sum_{j=0}^{N'-1} a(i+jn)\varpi^{jnk}]
\end{aligned}
$$

其中，$0\leq k<N, 0\leq k+pN'<N$

现将规模为N的问题分解为n个规模为N'的子问题，如此分治，有:$T(N)=nT(\frac{N}{n})+O(Nn)$，其中n | N

于是，总的时间复杂度为：$T(N)=O(N\cdot\sum_{i=1}^{m}(p_ik_i))$，若$N=2^m, T(N)=O(N\log N)$

总结一下，现在对某一整数N，如果要进行再整数域上的FFT，必须满足存在旋转因子ϖ，使

$$
\varpi^{i}\begin{cases}\neq 1, & i=1,2,\cdots,N-1\\ =1, & i=N\end{cases}
$$

要在整数域了满足上述条件的，可以是关于素数p的取模运算，若a是在模p意义下的原根，旋转因子为$a^{\frac{p-1}{N}}$，
要求N整除p - 1,最后的结果为对p取模后的答案(如果要求准确答案，
需要满足$p>\max\{a(i),b(i),c(i)\}(i=0,1,2,\cdots,N-1)$，或者对不同素数p进行多次计算，
然后用中国剩余定理求解)

适合$p=998244353=119\times 2^{23}+1(2^{23}>8.3e6)$，3是p的原根

p = 985661441, 3是p的原根，$(p-1)=2^{20}*i+1$

适合的p有很多，枚举i，判断(1<<K)*i + 1是否为素数即可

*/

```
//来源: 2015多校第三场, 1007标程
LL qpow(LL x, LL k, LL mod);
const LL mod = 998244353, wroot = 3;
int wi[NUM << 2];
int ntt_init(int n)
{
    int N = 1;
    while(N < n + n) N <<= 1;
```

```
    wi[0] = 1, wi[1] = qpow(wroot, (mod - 1) / N, mod);
    for(int i = 2; i < N; i++)
        wi[i] = 1LL * wi[i - 1] * wi[1] % mod;
    return N;
}

void brc(vector<int> &p, int N) //蝶形变换, 交换位置i与逆序i, 如N=2^3, 交换p[011=3]与p[110=6]
{
    int i, j, k;
    for(i = 1, j = N >> 1; i < N - 2; i++)
    {
        if(i < j) swap(p[i], p[j]);
        for(k = N >> 1; j >= k; k >>= 1) j -= k;
        if(j < k) j += k;
    }
}
void NTT(vector<int> &a, int N, int op)
{
    brc(a, N);
    for(int h = 2; h <= N; h <<= 1)
    {
        int unit = op == -1 ? N - N / h : N / h;
        int hf = h >> 1;
        for(int i = 0; i < N; i += h)
        {
            int w = 0;
            for(int j = i; j < i + hf; j++)
            {
                int u = a[j], t = 1LL * wi[w] * a[j + hf] % mod;
                if((a[j] = u + t) >= mod) a[j] -= mod;
                if((a[j + hf] = u - t) < 0) a[j + hf] += mod;
                if((w += unit) >= N) w -= N;
            }
        }
    }
    if(op == -1)
    {
        int inv = qpow(N, mod - 2, mod);
        for(int i = 0; i < N; i++) a[i] = 1LL * a[i] * inv % mod;
    }
}
```

4.15 一些数学知识

```
//1. 格雷码: (相邻码之间二进制只有一位不同), 构造方法: a_i = i xor (i >> 1)(a_i为求第i个格雷码)
/*2. 多边形数: 可以排成正多边形的整数
    第n个s边形数的公式是: n[(s-2)n-(s-4)]/2
    费马多边形定理:每一个正整数最多可以表示成n个n—边形数之和
*/
//3. 四平方和定理: 每个正整数均可表示为4个整数的平方和. 它是费马多边形数定理和华林问题的特例.
//4. 即对任意奇素数 p, 同余方程x^2 + y^2 + 1 ≡ 0 (mod p)必有一组整数解x, y满足0 ≤ x < p/2, 0 ≤ y < p/2
///勾股数
/*
勾股数: 对正整数a, b, c, 如果有a^2 + b^2 = c^2, 称(a, b, c)为勾股数
性质: a, b中一个为奇数, 一个为偶数, c一定为奇数.
本原勾股数: 满足gcd(a, b, c) = 1的勾股数
本原勾股数定理: 如果对奇数s > t ≥ 1, 且gcd(s, t) = 1, 则有: a = s × t, b = (s^2 - t^2)/2, c = (s^2 + t^2)/2, (a, b,
    c)是本原勾股数
*/
/*数组映射循环同构
    对于一个数组, 我们假设不关心它的每个数值得大小, 只关心它们是否相同, 而且它是循环的,
    即从任意位置都可以看做是数组的起点. You are given a set of N vectors, each vector consists of K
```

```
    integers. Vector X is equivalent to Y (denoted X ≡ Y). if there exist a bijection f : Z → Z
    and an integer r, such that X[i] = f(Y [(i + r) mod K]) for each i in the range [0..K − 1]. For
    example, (1, 2, 2, 3) ≡ (22, 3, 4, 22), with r = 2 and f(22) = 2, f(3) = 3 and f(4) = 1. But (22, 3,
    22, 4) is not equivalent to (1, 2, 2, 3).
    我们将对于每个位置，我们找出下一个相同数字的位置，用它们的位置差表示该数组，然后用最小表示法，
    即可唯一的表示出该等价关系.
    例: 对(22, 3, 4, 22)，用位置差代替该数字，得: (3, 4, 4, 1)，最小表示法得: (1, 3, 4,
    4)这样所有等价的数组有且仅有这么一种表示方法.
*/

/*关于浮点运算
1. 精度问题: 精度相差过大的运算会带来较大的精度误差
2. 比较: 不能直接比较，用EPS修正精度误差
3. 越界: 在sqrt(), asin(), acos()等处注意有精度误差带来的越界，如a = 0，但是可能表示为−1e−12,
    那样sqrt, asin, acos等会RE，a = 1时，asin, acos也可能出错.
4. 四舍五入: 三种常见的方法:
    printf("%.3lf", a);  //保留a的三位小数，按照第四位四舍五入
    (int)a;  //将a靠进0取整
    ceil(a); floor(a);   //顾名思义，向上取证，向下取整. 需要注意的是，这两个函数都返回double,
    而非int
    由于精度误差，可能四舍五入后与正确答案有差异. 如:  题目要求输出保留两位小数.
    正确答案的精确值是0.005，按理应该输出0.01，但你的结果可能是0.005000000001(恭喜),
    也有可能是0.004999999999(悲剧)，如果按照printf("%.2lf", a)输出，那你的遭遇将和括号里的字相同.
    解决办法是，如果a为正，则输出a+eps，否则输出a−eps
5. 输出 −0.000: 解决: 先判断最后结果是否为0，如果是0，直接输出0.000.
6. 关于set<double>: 由于精度误差，本来相等的两个数判为不等; 解决办法: 将double封装，然后重载小于符:
    bool operator < (const Date &b) const {return val < d.val − EPS;}
7. EPS的取值: 选取适合的EPS值，一般取EPS = 1e−8
8. 容易产生较大浮点误差的函数有asin, acos. 欢迎尽量使用atan2().
9. 尽量不使用浮点数: 如果数据明确说明是整数，而且范围不大的话，使用int或者long
    long代替double都是极佳选择，这样就不存在浮点误差了.
10. 当相加减乘的两个浮点数相差过大时，可以略去这些计算，从而减少计算量，甚至降低复杂度. 反之,
    为避免由此带来的精度误差，尽量使相运算的两个浮点数的相差尽量小.
*/
int sgn(double a) {return a < −EPS ? −1 : a < EPS ? 0 : 1;}
int mysqrt(double a) {return sqrt(max(0.0, a));}
```

5 线性代数 Linear Algebra

5.1 矩阵 Matrix

```
///矩阵 Matrix
/*定义:
    矩阵: m行n列.
    n阶方阵: n行n列的矩阵.
    行矩阵: 1 × n的矩阵.
    列矩阵: n × 1的矩阵.
    零矩阵O_{m×n}: 全为0的矩阵.
    单位矩阵I_n: 对角线全为1的n阶方阵.
    同型矩阵A, B: A, B的行数和列数都相同.
*/
/*操作和性质:
线性运算:
    两矩阵A, B相等: A, B是同型矩阵，且对应元相等.
    矩阵的加法和减法: 当A, B是同型矩阵时可加减，结果是对应元相加减的同型矩阵.
    A的负矩阵−A: A的每个元取反.
    矩阵的数乘kA: A的每个元乘以k.
线性运算的性质:
    1. (交换律) A + B = B + A
    2. (结合律) (A + B) + C = A + (B + C)
```

```
    3. A + O = A, A + (-A) = O
    4. 1A = A
    5. k(lA) = (kl)A
    6. k(A + B) = kA + kB
    7. (k + l)A = kA + lA
矩阵的乘法:
    m × p矩阵A = (a_ij)_{m×p}, p × n矩阵B = (b_ij)_{p×n}的乘积为m × n矩阵C = (c_ij)_{m×n}, 其中
```

$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}$$

```
    .
    要求A的列数等于B的行数时才能相乘
矩阵的乘法的性质:
    1. (结合律) (AB)C = A(BC)
    2. (数乘结合律) k(AB) = (kA)B = A(kB)
    3. (分配率) A(B + C) = AB + AC, (B + C)A = BA + CA
    4. 矩阵的乘法一般不满足交换律, 当AB = BA时, 称A与B可交换.
    5. 不满足消去率, 即由AB - AC = A(B - C) = O不能推出B - C = O
    6. I_m A_{m×n} = A_{m×n} I_m = A_{m×n}
    7. n阶单位矩阵与任意n阶矩阵A是可交换的, 即IA = AI = A.
方阵的幂:
    设A是n阶方阵, k是正整数, 定义:
```

$$\begin{cases} A^0 = I \\ A^1 = A \\ A^{k+1} = A^k A, \quad k = 1, 2, \cdots \end{cases}$$

```
方阵的幂的性质:
    1. A^m A^k = A^{m+k}
    2. (A^m)^k = A^{mk}
    3. 一般(AB)^k ≠ A^k B^k, 当AB = BA时, (AB)^k = A^k B^k = B^k A^k, 其逆不真.
矩阵的转置:
    将矩阵A的行列互换, 所得的矩阵称为A的转置, 即为A_T.
    m × n矩阵的转置是n × m矩阵
    对称矩阵A: A^T = A
    反对称矩阵A: A^T = -A
矩阵的转置的性质:
    1. (A^T)^T = A
    2. (A + B)^T = A^T + B^T
    3. (kA)^T = kA^T, k为数
    4. (AB)^T = B^T A^T
分块矩阵:
    将m × n矩阵分为r × c个子矩阵, 分块矩阵的乘法相当于将把各个子矩阵当作数是的乘法.
*/
const int Matrix_N = 55, Matrix_M = 55;
struct Matrix
{
    int a[Matrix_N][Matrix_M];
    int n, int m;
    Matrix(int _n = 0, int _m = 0, bool I = false) {init(_n, _m, I);}
    void init(int _n = 0, int _m = 0, bool I = false)
    {
        memset(a, 0, sizeof(a));
        n = _n;
        m = _m;
        if(I) for(int i = 1; i <= n; ++i) a[i][i] = 1;
    }
    //C = A + B
    Matrix operator + (const Matrix &B) const
    {
        Matrix res(n, m);
        for(int i = 1; i <= n; ++i)
            for(int j = 1; j < m; ++j)
                res.a[i][j] = a[i][j] + B.a[i][j];
        return Matrix;
    }
    //C = A - B
```

```
    Matrix operator - (const Matrix &B) const
    {
        Matrix res(n, m);
        for(int i = 1; i <= n; ++i)
            for(int j = 1; j < m; ++j)
                res.a[i][j] = a[i][j] - B.a[i][j];
        return Matrix;
    }
    //A += B
    Matrix operator += (const Matrix &B) const
    {
        for(int i = 1; i <= n; ++i)
            for(int j = 1; j < m; ++j)
                a[i][j] += B.a[i][j];
        return *this;
    }
    //C = AB
    Matrix operator * (const Matrix & B) const
    {
        Matrix res(n, B.m);
        int i, j, k;
        for(i = 1; i <= n; ++i)
            for(j = 1, res.a[i][j] = 0; j <= B.m; ++j)
                for(k = 1; k <= m; ++k)
                    res.a[i][j] = a[i][k] * B.a[k][j];
//                    if((res.a[i][j] += a[i][k] * B.a[k][j] % mod) >= mod)
//                        res.a[i][j] -= mod;
        return res;
    }
    //方阵的k次幂
    Matrix operator ^(int k)
    {
        Matrix x = *this;
        Matrix res(x.n, x.n, true);
        while(k)
        {
            if(k & 1) res = res * x;
            x = x * x;
            k >>= 1;
        }
        return res;
    }
    void out()
    {
        printf("N = %d, M = %d\n", n, m);
        for(int i = 1; i <= n; ++i)
            for(int j = 1; j <= m; ++j)
                printf("%d%c", a[i][j], j == n ? '\n' : ' ');
    }
};

/*应用:
1. A是1 × n的行矩阵, B是n阶方阵, 则
```

$$C_{1\times n} = ABB\cdots B = A(B^m)$$

```
2. 设A是n × 1的列矩阵, B是n阶方阵, 求
```

$$\sum_{k=0}^{m-1} AB^k$$

```
    做法: 令C为分块矩阵
```

$$\begin{bmatrix} B & A \\ O & I \end{bmatrix}$$

```
    , 则
```

$$C^m = \begin{bmatrix} B^m & \sum_{k=1}^{m-1} AB^k \\ O & I \end{bmatrix}$$

```
*/
```

5.2 矩阵的初等变换和矩阵的逆

```
///矩阵的初等变换和矩阵的逆
const int Matrix_N = 1010, Matrix_M = 1010;
//矩阵类 适用与求矩阵的逆与高斯消元等场合
//行的初等变换
typedef vector<double> VD;
VD operator * (const VD &a, const double b)
{
    int _n = a.size();
    VD c(_n);
    for(int i = 0; i < _n; i++)
        c[i] = a[i] * b;
    return c;
}
VD operator - (const VD &a, const VD &b)
{
    int _n = a.size();
    VD c(_n);
    for(int i = 0; i < _n; i++)
        c[i] = a[i] - b[i];
    return c;
}
VD operator + (const VD &a, const VD &b)
{
    int _n = a.size();
    VD c(_n);
    for(int i = 0; i < _n; i++)
        c[i] = a[i] + b[i];
    return c;
}
struct Matrix
{
    int n, m;
    VD *a;
    void Matrix(int _n = Matrix_N, int _m = Matrix_M)
    {
        n = _n, m = _m;
        a = new VD[n];
        for(int i = 0; i < n; i++)
            a[i].resize(m, 0);
    }
    void ~Matrix()
    {
        delete []a;
    }
    void clear()//0矩阵
    {
        for(int i = 0; i < n; i++)
            a[i].assign(0);
    }
    void I()//单位矩阵
    {
        clear();
        for(int i = 0; i < n; i++)
            a[i][i] = 1;
    }
    //矩阵加法, 同上
    Matrix operator + (const Matrix &b) const
    {
        Matrix c(n, m);
        for(int i = 0; i < n; i++)
            c.a[i] = a[i] + b.a[i];
        return c;
```

```
    }
    Matrix operator - (const Matrix &b) const //矩阵减法
    {
        Matrix c(n, m);
        for(int i = 0; i < n; i++)
            c.a[i] = a[i] - b.a[i];
        return c;
    }
    Matrix operator * (const Matrix &b) const //矩阵乘
    {
        Matrix c(n, b.m);
        for(int i = 0; i < n; i++)
            for(int j = 0; j < b.m; j++)
            {
                c[i][j] = 0;
                for(int k = 0; k < m; k++)
                    c.a[i][j] += a[i][j] * b.a[i][j];
            }
        return c;
    }

    //实现求矩阵的逆O(n^3)
    //将原矩阵A和一个单位矩阵I做一个大矩阵(A,I), 用行的初等变换将大矩阵中的A变为I,
     将会的到(I,A^-1)的形式
    //注意:
    Matrix inverse()
    {
        Matrix c;
        c.I();
        for(in i = 0; i < n; i++)
        {
            for(int j = i; j < n; j++)
                if(fabs(a[j][i]) > 0)
                {
                    swap(a[i], a[j]);
                    swap(c[i], c[j]);
                    break;
                }
            c[i] = c[i] * (1.0 / a[i][i]);
            a[i] = a[i] * (1.0 / a[i][i]);
            for(int j = 0; j < n; j++)
                if(j != i && fabs(a[j][i]) > 0)
                {
                    c[j] = c[j] - a[i] * a[j][i];
                    a[j] = c[j] - a[i] * a[j][i];
                }
        }
    }
};
//Guass消元
int Guass(double a[][MAXN], bool l[], double ans[], int n)
{//l, ans储存解, l[]表示是否是自由元
    int res = 0, r = 0;
    for(int i = 0; i < n; i++) l[i] = false;
    for(int i = 0; i < n; i++)
    {
        for(int j = r; j < n; j++)
            if(fabs(a[j][i]) > EPS)
            {
                for(int k = i; k <= n; k++)
                    swap(a[j][k], a[r][k]);
                break;
            }
        if(fabs(a[r][i]) < EPS)
```

```
        {
            ++res;
            continue;
        }
        for(int j = 0; j < n; j++)
            if(j != r && fabs(a[j][i]) > EPS)
            {
                double tmp = a[j][i] / a[r][i];
                for(int k = i; k <= n; k++)
                    a[j][k] -= tmp * a[r][k];
            }
        l[i] = true;
        ++r;
    }
    for(int i = 0; i < n; i++)
        if(l[i])
            for(int j = 0; j < n; j++)
                if(fabs(a[j][i]) > 0)
                    ans[i] = a[j][n] / a[j][i];
    return res;//返回解空间的维数
}
//常系数线性齐次递推
/*已知f_x = a_0 f_{x-1} + a_1 f_{x-2} + ... a_{n-1} f_{x-n} 和 f_0, f_1, ..., f_{n-1}, 给定t, 求f_t
f的递推可以看做是一个n × n的矩阵A乘以一个n维列向量β, 即
```

$$A = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ a_{n-1} & a_{n-2} & a_{n-3} & \cdots & a_0 \end{bmatrix}, \beta_n = \begin{bmatrix} f_{x-n} \\ f_{x-n+1} \\ \vdots \\ f_{x-2} \\ f_{x-1} \end{bmatrix}$$

```
则β_t = A^{t-n+1} β_0 (t ≥ n)
*/
```

6 组合数学 Combinatorial Mathematics

6.1 排列 Permutation

```
/*排列
 *排列: 从集合A={a_1, a_2, ..., a_n}的n个元素中取r个按照一定的次序排列起来, 称为集合A的r—排列。
 * 记其排列数:
```

$$P_n^r = \begin{cases} 0, & n < r \\ 1, & n \geq r = 0 \\ n(n-1)\cdots(n-r+1) = \dfrac{n!}{(n-r)!}, & r \leq n \end{cases}$$

```
* 推论: 当n ≥ r ≥ 2时, 有P_n^r = nP_{n-1}^{r-1}
*       当n ≥ r ≥ 2是, 有P_n^r = rP_{n-1}^{r-1} + P_{n-1}^r
*
*圆排列: 从集合A={a_1, a_2, ..., a_n}的n个元素中取出r个元素按照某种顺序排成一个圆圈, 称这样的排列为圆排列。
* 集合A中n个元素的r圆排列的个数为:
```

$$\frac{P_n^r}{r} = \frac{n!}{r(n-r)!}$$

```
*
*重排列: 从重集B={k_1 · b_1, k_2 · b_2, ..., k_n · b_n}中选取r个元素按照一定的顺序排列起来, 称这种r—排列为重排列。
* 重集B={∞ · b_1, ∞ · b_2, ..., ∞ · b_n}的r—排列的个数为n^r。
* 重集B={n_1 · b_1, n_2 · b_2, ..., n_k · b_k}的全排列的个数为
```

$$\frac{n!}{n_1! \cdot n_2! \cdots n_k!}, n = \sum_{i=1}^{k} n_i$$

```
*
```

*错排：$\{1,2,\cdots,n\}$的全排列，使得所有的i都有$a_i \neq i$，$a_1a_2\cdots a_n$是其的一个排列
* 错排数

$$D_n = n!(1-\frac{1}{1!}+\frac{1}{2!}-\frac{1}{3!}+\cdots+(-1)^n\frac{1}{n!})$$

* 递归关系式：

$$\begin{cases} D_n = (n-1)(D_{n-1}+D_{n-2}), & n>2 \\ D_0 = 1, D_1 = 0 \end{cases}$$

* 性质：

$$\lim_{n\to\infty}\frac{D_n}{n!} = e^{-1}$$

* 前17个错排值

n	0	1	2	3	4	5
D_n	1	0	1	2	9	44
n	6	7	8	9	10	11
D_n	265	1845	14833	133496	1334961	14684570
n	12	13	14	15	16	17
D_n	176214841	2290792932	32071101049	481066515734	7697064251745	130850092279664

*相对位置上有限制的排列的问题：
*　求集合$\{1,2,3,\cdots,n\}$的不允许出现12,23,34, ... , (n–1)n的全排列数为

$$Q_n = n! - C_{n-1}^1(n-1)! + C_{n-1}^2(n-2)! - \cdots + (-1)^{n-1}C_{n-1}^{n-1}\cdot 1!$$

*　当$n \geq 2$时，有$Q_n = D_n + D_{n-1}$
*　求集合$\{1,2,3,\cdots,n\}$的圆排列中不出现$12,23,34,\cdots,(n-1)n,n1$的圆排列个数为：

$$(n-1)! - C_n^1(n-2)! + \cdots + (-1)^{n-1}C_n^{n-1}0! + (-1)^nC_n^n\cdot 1$$

*一般限制的排列：
*　棋盘：设n是一个正整数，$n\times n$的格子去掉某些格后剩下的部分称为棋盘（可能不去掉）
*　棋子问题：在给定棋盘C中放入k个无区别的棋子，要求每个棋子只能放一格，且各子不同行不同列，
　　求不同的放法数$r_k(C)$
*　棋子多项式：给定棋盘C，令$r_0(C)=1$，n为C的格子数，则称

$$R(C) = \sum_{k=0}^{n} r_k(C)x^k$$

　　为棋盘C的棋子多项式
*　定理1：给定棋盘C，指定C中某格A，令C_i为C中删去A所在列与行所剩的棋盘，C_e为C中删去格A所剩的棋盘，则
*

$$R(C) = xR(C_i) + R(C_e)$$

*　设C_1和C_2是两个棋盘，若C_1的所有格都不与C_2的所有格同行同列，则称两个棋盘是独立的.
*　定理2：若棋盘C可分解为两个独立的棋盘C_1和C_2，则

$$R(C) = R(C_1)R(C_2)$$

*　n元有禁位的排列问题：求集合$\{1,2,\cdots,n\}$的所有满足$i(i=1,2,\cdots,n)$不排在某些已知位的全排列数。
*　n元有禁位的排列数为

$$n! - r_1(n-1)! + r_2(n-2)! - \cdots + (-1)^nr_n$$

　　其中r_i为将i个棋子放入禁区棋盘的方式数，$i=1,2,\cdots,n$
*/

6.2 全排列 Full Permutation

```
//全排列Full permutation
//排列数为n!
//求全排列
//next_permutation求下一个排列，如果是最后一个排列则返回false
void FullPermutation()
{
    //求所有全排列需排好序
    do
    {
```

```
        //do someting
    }
    while(next_permutation(perm2, perm2 + n));
}

///康托展开
//把一个整数X展开成如下形式: X = a_n(n-1)! + a_{n-1}(n-2)! + ··· + a_i(i-1)! + ··· + a_2·1! + a_1·0!
//其中, a为整数, 并且0 ≤ a_i < i(1 ≤ i ≤ n). 这就是康托展开.
//应用
//1~n的全排列中某一排列在所有排列中的位置
//字典序比排列P小的排列个数为: X_P = sum_{i=1}^{n} count_{j=i+1}^{n} a[j] < a[i] × (n-i-1)!
//例: {1, 3, 2, 4}在1~4的所有全排列的位置为(X_P = 0×3! + 1×2! + 0×1! + 0×1!) + 1 = 3
const int PermSize = 12;
long long fac[PermSize] = {1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 39916800};//n!
long long Cantor(int s[], int n)
{
    int i, j, temp;
    long long num = 0;
    for(i = 0; i < n; i++)
    {
        temp = 0;
        for(int j = i + 1; j < n; j++)
        {
            if(s[j] < s[i])
                temp++; //判断几个数小于它
        }
        num += fac[n - i - 1] * temp; //(或num=num+fac[n-i-1]*temp;)
    }
    return num + 1;
}


//逆运算 逆康托展开
//已知1~n的全排列中的某一排列permutation在所以全排列中的位置, 求该排列
//第pos个排列, a[i]为排除a[1···i-1]后第((pos - 1)%(n - i)!)/(n - i - 1)! + 1小的数字
//例: 1~4中第3大的数为:2/3!=0...2, p[0] = 1(1, 2, 3, 4第1小的数); 2/2!=1...0, p[1]=3(2, 3,
    4第2小的数; 0/1!=0...0, p[2]=2{2,4第1小的数}; 0/0!=0...0,
    p[3]=4{4第1小的数};  故第3个的排列为{1,3,2,4}
int fac[] = {1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880};
int perm[9];
void unContor(int pos, int n)
{
    int i, j;
    bool vis[12] = {0};
    pos--;
    for(i = 0; i < n; i++)
    {
        int t = sum / fac[n - i - 1];
        for(j = 1; t && j <= n; j++)
            if(!vis[j]) t--;
        vis[perm[i] = j] = 1;
        sum %= fac[n - i - 1];
    }
}
```

6.3 组合与组合恒等式

```
/*1. 组合: 从n个不同的元素中取r个的方案数C_n^r:
```

$$C_n^r = \begin{cases} \dfrac{n!}{r!(n-r)!}, & n \geq r \\ 1, & n \geq r = 0 \\ 0, & n < r \end{cases}$$

推论1：$C_n^r = C_n^{n-r}$

推论2(Pascal公式)：$C_n^r = C_{n-1}^r + C_{n-1}^{r-1}$

推论3：$\sum_{k=r-1}^{n-1} C_k^{r-1} = C_{n-1}^{r-1} + C_{n-2}^{r-2} + \ldots + C_{r-1}^{r-1} = C_n^r$

2. 从重集$B = \{\infty \cdot b_1, \infty \cdot b_2, \ldots, \infty \cdot b_n\}$的r-组合数F(n,r)为$F(n,r) = C_{n+r-1}^r$

3. 二项式定义

当n是一个正整数时，对任何x和y有：

$$(x+y)^n = \sum_{k=0}^{n} C_n^k x^k y^{n-k}$$

令y=1，有：

$(1+x)^n = \sum_{k=0}^{n} C_n^k x^k = \sum_{k=0}^{n} C_n^{n-k} x^k$

广义二项式定理：

广义二项式系数：对于任何实数α和整数k,有

$$C_\alpha^K = \begin{cases} \frac{\alpha(\alpha-1)\ldots(\alpha-k+1)}{k!} & k > 0 \\ 1 & k = 0 \\ 0 & k < 0 \end{cases}$$

设α是一个任意实数，则对满足$|\frac{x}{y}| < 1$的所有x和y，有

$$(x+y)^\alpha = \sum_{k=0}^{\infty} C_\alpha^k x^k y^{\alpha-k}$$

推论：令$z = \frac{x}{y}$,则有

$$(1+z)^\alpha = \sum_{k=0}^{\infty} C_\alpha^k z^k, |z| < 1$$

令$\alpha = -n$(n是正整数),有

$$(1+z)^{-n} = \frac{1}{(1+z)^n} = \sum_{k=0}^{\infty} (-1)^k C_{n+k-1}^k z^k$$

又令z = −rz, (r为非零常数),有

又令n=1,有

$$\frac{1}{1+z} = \sum_{k=0}^{\infty} (-1)^k z^k$$

令z = −z，有

$$\frac{1}{1-z} = \sum_{k=0}^{\infty} z^k$$

令$\alpha = \frac{1}{2}$，有

$$\sqrt{1+z} = 1 + \sum_{k=1}^{\infty} \frac{(-1)^{k-1}}{k \cdot 2^{2k-1}} C_{2k-2}^{k-1} z^k$$

4. 组合恒等式

1. $\sum_{k=0}^{n} C_n^k = 2^n$
2. $\sum_{k=0}^{n} (-1)^k C_n^k = 0$
3. 对于正整数 n 和 k,

$$C_n^k = \frac{n}{k} C_{n-1}^{k-1}$$

4. 对于正整数 n,

$$\sum_{k=0}^{n} k C_n^k = \sum_{k=1}^{n} k C_n^k = n \cdot 2^{n-1}$$

5. 对于正整数 n,

$$\sum_{k=0}^{n} (-1)^k k C_n^k = 0$$

6. 对于正整数 n,

$$\sum_{k=0}^{n} k^2 C_n^k = n(n+1)2^{n-2}$$

7. 对于正整数 n,

$$\sum_{k=0}^{n}\frac{1}{k+1}C_n^k=\frac{2^{n+1}-1}{n+1}$$

8. (Vandermonde 恒等式) 对于正整数 n,m 和 p, 有 $p\leq minm,n$,

$$\sum_{k=0}^{p}C_n^kC_m^{p-k}=C_{m+n}^p$$

9. (令 p=m) 对于任何正整数 n,m,

$$\sum_{k=0}^{m}C_m^kC_n^k=C_{m+n}^m$$

10. (又令 m=n) 对于任何正整数 n,

$$\sum_{k=0}^{n}(C_n^k)^2=C_{2n}^n$$

11. 对于非负整数 p,q 和 n,

$$\sum_{k=0}^{p}C_p^kC_q^kC_{n+k}^{p+q}=C_n^pC_n^q$$

12. 对于非负整数 p,q 和 n,

$$\sum_{k=0}^{p}C_p^kC_q^kC_{n+p+q-k}^{p+q}=C_{n+p}^pC_{n+q}^q$$

13. 对于非负整数 n,k,

$$\sum_{i=0}^{n}C_i^k=C_{n+1}^{k+1}$$

14. 对于所有实数 α 和非负整数 k,

$$\sum_{j=0}^{k}C_{\alpha+j}^j=C_{\alpha+k+1}^k$$

15.

$$\sum_{k=0}^{n}\frac{2^{k+1}}{k+1}C_n^K=\frac{3^{n+1}-1}{n+1}$$

16.

$$\sum_{k=0}^{m}C_{n-k}^{m-k}=C_{n+1}^m$$

17.

$$\sum_{k=m}^{n}C_k^mC_n^k=C_n^m2^{n-m}$$

18.

$$\sum_{k=0}^{m}(-1)^kC_n^k=(-1)^mC_{n-1}^m$$

```
*/
```

6.4 鸽笼原理与 Ramsey 数

```
/*鸽笼原理:
 * 简单形式: 如果把n+1个物体放到n个盒子中去, 则至少有一个盒子中放有两个或更多的物体.
 * 一般形式: 设q_i是正整数(i = 1,2,⋯,n), q ≥ q_1 + q_2 + ⋯ + q_n − n + 1,
    如果把q个物体放入n个盒子中去, 则存在一个i使得第i个盒子中至少有q_i个物体.
 * 推论1: 如果把n(r−1)+1个物体放入n个盒子中, 则至少存在一个盒子放有不少于r个物体。
 * 推论2: 对于正整数m_i(i = 1,2,⋯,n), 如果 \frac{\sum_{i=1}^{n}{m_i}}{n} > r −
    1,则至少存在一个i, 使得m_i ≥ r.
 * 例: 在给定的n个整数a_1, a_2, ⋯, a_n中, 存在k和l(0 ≤ k < l ≤ n), 使得a_{k+1} + a_{k+2} + ⋯ + a_l能被n整除
 */
/*Ramsey定理和Ramsey数
在人数为6的一群人中, 一定有三个人彼此相识, 或者彼此不相识.
在人数为10的一群人中, 一定有3个人彼此不相识或者4个人彼此相识。
在人数为10的一群人中, 一定有3个人彼此相识或者4个人彼此不相识。
在人数为20的一群人中, 一定有4个人彼此相识或者4个人彼此不相识。

设a,b为正整数,令N(a,b)是保证有a个人彼此相识或者有b个人彼此不相识所需的最少人数,则称N(a,b)为Ramsey数.
Ramsey数的性质:
N(a,b) = N(b,a)
N(a,2) = a
当a, b ≥ 2时, N(a,b)是一个有限数, 并且有N(a,b) ≤ N(a − 1,b) + N(a,b − 1)
```

当N(a-1,b)和N(a,b-1)都是偶数时，则有$N(a,b) \leq N(a-1,b)+N(a,b-1)-1$

N(a,b)	2	3	4	5	6	7	8	9
2	2	3	4	5	6	7	8	9
3		6	9	14	18	23	28	36
4			18	24	44	66		
5				55	94	156		
6					178	322		
7						626		

```
如果把一个完全n角形，用r中颜色c_1,c_2,...,c_r对其边任意着色。
    设N(a_1,a_2,...,a_r)是保证下列情况之一出现的最小正整数:
        c_1颜色着色的一个完全a_1角形
        用c_2颜色着色的一个完全a_2角形
        .........
        或用颜色c_r着色的一个完全a_r角形
        则称数N(a_1,a_2,...,a_r)为Ramsey数。
    对与所有大于1的整数a_1,a_2,a_3,数N(a_1,a_2,a_3)是存在的。
    对于任意正整数m和a_1,a_2,...,a_m >= 2,Ramsey数N(a_1,a_2,...,a_m)是存在的。
 .....
*/
```

6.5 容斥原理

```
/*容斥原理
 * 集合S中具有性质p_i(i=1,2,...,m)的元素所组成的集合为A_i,则S中不具有性质p_1,p_2,...,p_m的元素个数为
 * |¬A_1 ∩ ¬A_2 ∩ ... ∩ ¬A_m| = |S| - Σ_{i=1}^{m}|A_i| + Σ_{i≠j} A_i∩A_j - Σ_{i≠j≠k}|A_i∩A_j∩A_k + ... + (-1)^m|A_1∩A_2∩...∩A_m|
 */
/*重集的r-组合
 * 重集B={k_1·a_1, k_2·a_2, ..., k_n·a_n}的r-组合数:
 * 利用容斥原理，求出重集B'={∞·a_1, ∞·a_2, ..., ∞·a_n}的r-组合数F(n,r)
 * 在求出满足自少含k_i+1个a_i(1<=i<=n)的r-组合数，等同于重集B'的r-k_i-1-组合数
 * ....
 * 右容斥原理得: 重集B的r-组合数为:
 *
```

$$F(n,r)-\sum_{i=1}^{n}F(n,r-k_i-1)+\sum_{i\neq j}F(n,r-k_i-k_j-2)+\cdots+(-1)^nF(n,r-k_1-k_2-\cdots-k_n-n)$$

```
 */
```

6.6 母函数 Generating Function

```
/*母函数
 * 普通母函数:
 * 定义: 给定一个无穷序列(a_0,a_1,a_2,...,a_n,...)(简记为{a_n}),称函数
 *
```

$$f(x)=a_0+a_1x^1+a_2x^2+\cdots+a_nx^n+\cdots=\sum_{i=1}^{\infty}a_ix^i$$

```
 * 为序列{a_n}的普通母函数
 * 常见普通母函数:
 * 序列(C_n^0, C_n^1, C_n^2, ..., C_n^n)的普通母函数为f(x)=(1+x)^n
 * 序列(1,1,...,1,...)的普通母函数为f(x)=1/(1-x)
 * 序列(C_{n-1}^0, -C_n^1, C_{n+1}^2, ..., (-1)^k C_{n+k-1}^k, ...)的普通母函数为f(x)=(1+x)^{-n}
 * 序列(C_0^0, C_2^1, C_4^2, ..., C_{2n}^n, ...,)的普通母函数为f(x)=(1-4x)^{-1/2}
 * 序列(0, 1×2×3, 2×3×4, ..., n×(n+1)×(n+2), ...)的普通母函数为6/(1-x)^4
 *
 * 指数母函数
 * 定义: 称函数
 *
```

$$f_e(x)=a_0+a_1\frac{x^1}{1!}+a_2\frac{x^2}{2!}+\cdots+a_n\frac{x^n}{n!}+\cdots=\sum_{n=0}^{\infty}a_n\frac{x^n}{n!}$$

```
    为序列(a_0,a_1,...,a_n,...)的指数母函数。
```

```
 *  常见指数母函数为
 *  序列(1,1,⋯,1,⋯)的指数母函数为f_e(x)=e^x
 *  n是整数, 序列(P_n^0,P_n^1,⋯,P_n^n)的指数母函数为f_e(x)=(1+x)^n
 *  序列(P_0^0,P_2^1,P_4^2,⋯,P_{2n}^n,⋯)的指数母函数为f_e(x)=(1-4x)^{-1/2}
 *  序列(1,α,α^2,⋯,α^n,⋯)的指数母函数为f_e(x)=e^{αx}
 *
 * 指数母函数和普通母函数的关系: 对同一序列的{a_n}的普通母函数f(x)和指数母函数f_e(x)有:
```

$$f(x)=\int_0^\infty e^{-s}f_e(sx)\mathrm{d}s$$

```
 *
 * 母函数的基本运算:
 *  设A(x), B(x) ,
    C(x)分别是序列(a_0,a_1,⋯,a_r,⋯),(b_0,b_1,⋯,b_r,⋯),(c_0,c_1,⋯,c_r,⋯)的普通(指数)母函数,则有:
 *  C(x) = A(x) + B(x) 当且仅当对所有的i, 都有c_i=a_i+b_i(i=0,1,2,⋯,r,⋯).
 *  C(x) = A(x)B(x) 当且仅当对所有的i, 都有c_i=∑_{k=0}^i a_k b_{i-k}(i=0,1,2,⋯,r,⋯).
 */
/*母函数在组合排列上的应用
    从n个不同的物体中允许重复地选取r个物体, 但是每个物体出现偶数次的方式数。
```

$$f(x)=(1+x^2+x^4+/cdots)^n=(\frac{1}{1-x^2})^n=\sum_{r=0}^{\infty}C_{n+r-1}^r x^{2r}$$

```
    故答案为a_r=C_{n+r-1}^r
*/
```

6.7 整数拆分和 Ferrers 图

```
/*整数拆分
问题: 将正整数n拆分为若干个正整数部分, 或将n个无区别的球放入一些无区别的盒子中(非空).
性质: 母函数应用
定义:   1. 用P(n)表示n拆分种类数。
        2. 用P_k(n)表示n拆分成1,2,⋯,k的允许重复的方法数。
        3. 用P_o(n)表示n拆分成奇整数的方法数。
        4. 用P_d(n)表示n拆分成不同的整数的方法数。
        5. 用P_t(n)表示n拆分成2的不同幂的方法数。
定理1:  设a,b,c,⋯是大于0 的正整数, 则
```

$$\frac{1}{(1-x^a)(1-x^b)(1-x^c)\cdots}$$

```
        的级数展开式中的x^n的系数等于把正整数n拆分成a,b,c,⋯的方法数P(n).
推论1:  {P_k(n)}的普通母函数为
```

$$\frac{1}{(1-x)(1-x^2)\cdots(1-x^k)}$$

```
推论2:  {P(n)}的普通母函数为
```

$$\frac{1}{(1-x)(1-x^2)(1-x^3)\cdots}$$

```
推论3:  {P_o(n)}的普通母函数为
```

$$\frac{1}{(1-x)(1-x^3)(1-x^5)(1-x^7)\cdots}$$

```
定理2:  设a,b,c,⋯都是大于0的正整数,则
```

$$(1+x^a)(1+x^b)(1+x^c)\cdots$$

```
        的级数展开式中x^n项的系数就是把n拆分成a,b,c,⋯的和, 且a,b,c,⋯最多只出现一次的方法数。
推论1:  {P_d(n)}的普通母函数是
```

$$(1+x)(1+x^2)(1+x^3)(1+x^4)\cdots$$

```
推论2:  {P_t(n)}的普通母函数是
```

$$(1+x)(1+x^2)(1+x^4)(1+x^8)\cdots$$

定理3: (Euler) 对于正整数n都有

$$P_o(n) = P_d(n)$$

定理4: (Sylvester) 对于正整数n, 有

$$P_t(n) = 1$$

定理5: 对于正整数n, 有

$$P(n) < e^{3\sqrt{n}}$$

6. 将p(n)生成函数配合五边形数定理, 可以得到以下的递归关系式
$p(n) = \sum_i (-1)^{i-1} p(n - q_i)$
其中q_i是第i个广义五边形数.

Ferrers图: 设n的一个拆分为

$$n = a_1 + a_2 + \cdots + a_k$$

并假设$a_1 \geq a_2 \geq a_3 \geq \cdots \geq a_k \geq 1$

画一个由一行行的点所组成的图, 第一行有@a_1的点, 第二行有a_2个点, ·····, 第 k 行有a_k个点, 则称此图为 Ferrers 图。共轭图: 将一个 Ferrers 图的行列互换后仍是 Ferrers 图, 称互换后的 Ferrers 图为原图的共轭图。定理 6: 正整数 n 拆分成 m 项的和的方式数等于 n 拆分成最大数为 m 的方式数定理 7: 正整数 n 拆分成最多不超过 m 个项的方式数等于 n 拆分成最大的数不超过 m 的方式数。*//* 应用: 将一个大小为 n 的集合划分为 m 个子集 (可以为空), 对任意的 k 可以知道该划分中至少有 k 个元素的子集的数目 d, 问最坏情况下可以确定该集合至少有多少元素?
$(1 \leq n \leq 10^18, 1 \leq m \leq 20000)$(即为将整数 n 拆分为不超过 m 个整数后, 最坏情况下某一 Ferrers 图中最大矩形的点数)source: Gym 100490B(ASC38)solve: 二分枚举划分成的最大元素数 p, 使得任意的 $kd \leq p$, 那么第 2 个子集的元素数为 p/2, ..., 第 m 个子集的元素数为 p/m, 如果方案可行, 则 $p/1 + p/2 + \cdots + p/m >= n$*/

6.8 线性规划 Linear Programming

```
///线性规划
/*标准形式
    有n个变量, m个线性不等式
```

$$\max y = \sum_{j=1}^{n} c_j x_j$$

约束条件:

$$\sum_{j=1}^{n} a_{ij} x_j = b_i, i = 1, 2, \cdots, m$$

$$x_j \geq 0, j = 1, 2, \cdots, m$$

用矩阵形式表示为:
最大化$C^T X$, 满足约束$AX = B$, $X \geq 0$, 其中C, X为n为行向量, B为m维列向量, A为$m \times n$的矩阵

转化为标准形式:
1. 最小化目标函数: 目标函数中的系数取负, 即令C等于−C.
3. 有不等式线性约束: 大于等于线性约束减去一个新的松弛变量, 小于等于约束加上一个新的松弛变量.
4. 没有非负约束: $x_i \in (-\infty, +\infty)$, 则用$x_i' - x_i''$代替原变量$x_i$
$x_i \in (-\infty, 0]$, 同样用$x_i' - x_i''$代替原变量x_i, 并添加一个不等式线性约束$x_i' - x_i'' \leq 0$
注: 约束条件不能是小于或大于形式的不等式.
例: $\max y = 3x_1 - x_2 - x_3$

$$\begin{aligned} s.t. \quad & x_1 - 2x_2 + x_3 \leq 11 \\ & -4x_1 + x_2 + 2x_3 \geq 3 \\ & -2x_1 + x_3 = 1 \\ & x_i \geq 0, i = 1, 2, 3 \end{aligned}$$

转化为标准形式为:
$\max y = 3x_1 - x_2 - x_3$

$$\begin{aligned} s.t. \quad & x_1 - 2x_2 + x_3 + x_4 = 11 \\ & -4x_1 + x_2 + 2x_3 - x_5 = 3 \\ & -2x_1 + x_3 = 1 \\ & x_i \geq 0, i = 1, 2, 3, 4, 5 \end{aligned}$$

```
*/
/*单纯形方法
```

1. 把一般的线性规划问题表示为标准形式
2. 任选一初始可行解
3. 用这个基本可行解中的非基本变量表示出基本变量和目标函数
4. 根据目标函数表达式中的非基本变量的系数的符号，选择一个有负系数的非基本变量来变成基本变量，增加这个非基本变量的值，直到基本变量之一变成零.
5. 重复第3, 4步，直到目标函数的表达式中的非基本变量的系数全部为正为止.

单纯形方法之表格法:
以基本变量为行，非基本变量和解为列. 即用非基本变量表示基本变量的表达式中，将常数写在右边，带变量的写在左边，其系数和常数构成的一个$m \times n$的表格.
表格之间的变换(单纯形方法第4步):
如果选择的第i个基本变量和第j个非基本变量交换，则称系数a_{ij}为枢纽，包含枢纽的行为枢行，包含枢纽的列为枢列.
1. 把枢纽换为它的倒数
2. 把枢行的所有元都除以枢纽
3. 把枢列的所有元都除以枢纽且反号.
4. 对于表格中的其他元$a_{pq}(p \neq i, q \neq j)$换为$a_{pq} - (a_{pj}\frac{a_{iq}}{a_{ij}})$，将$w_p$换为$w_p - a_{pj}\frac{w_i}{a_{ij}}$

二阶段法:
第一阶段: 构造新的线性规划问题:

最大化: $z = -\sum_{i=1}^{m} y_i$

约束条件: $\sum_{j=1}^{n} a_{ij}x_j + y_i = b_i, (i = 1, 2, \cdots, m)$

$x_j \geq 0, (j = 1, 2, \cdots, n), y_j \geq 0, (j = 1, 2, \cdots, m)$

然后从基本可行解，$(0, 0, \cdots, 0, b_1, b_2, \cdots, b_m)$求得最优基本可行解，如果该最优基本可行解对应的目标函数z的值小于0，则说明原线性规划无可行解;否则$(x_1, x_2, \cdots, x_n)$是原线性规划的基本可行解.

第二阶段: 用第一阶段求出的基本可行解和单纯形方法求解最优基本可行解.

特殊情况:
1. 有无穷多个最优解: 第二阶段结束时目标函数所在行的系数至少有一个为0.
2. 目标函数值无解(无最优可行解): 第二阶段结束时有以非基本变量所对应的列的元全部为负.
3. 无可行解: 第一阶段结束时，在最后的单纯形表格中，目标函数所在的行的所有系数非负，目标函数的值小于零时，原问题不存在可行解.

*/
/*线性规划问题的对偶问题

最大化: $y = \sum_{i=1}^{n} c_i x_i$

约束条件: $\sum_{j=1}^{n} a_{ij}x_j \leq b_i, (i = 1, 2, \cdots, m)$

$x_j \geq 0, (j = 1, 2, \cdots, n)$

与

最小化: $z = \sum_{i=1}^{m} b_i y_i$

约束条件: $\sum_{j=1}^{m} a_{ji}y_j \geq c_i, (i = 1, 2, \cdots, n)$

$y_j \geq 0, (j = 1, 2, \cdots, m)$

称为本原问题与与其对偶问题.
定理: 线性规划问题的对偶问题的对偶问题是本原问题.
定理: 设$(x_1, x_2, \cdots, x_n)$与$(y_1, y_2, \cdots, y_m)$是一对互为对偶问题的线性规划问题的可行解，恒有:

$$\sum_{i=1}^{n} c_i x_i \leq \sum_{i=1}^{m} b_i y_i$$

定理: 设$(x_1, x_2, \cdots, x_n)$与$(y_1, y_2, \cdots, y_m)$是一对互为对偶问题的线性规划问题的可行解，且有:
$\sum_{i=1}^{n} c_i x_i = \sum_{i=1}^{m} b_i y_i$，则$(x_1, x_2, \cdots, x_n)$和$(y_1, y_2, \cdots, y_m)$是对应线性规划问题的最优可行解.
定理: 如果本原问题与对偶问题之一有最优可行解，则另一个问题也有最优可行解，且它们的目标函数值相等.
推论: 在本原问题的最优单纯形表中，目标函数的松弛变量的的系数就是对偶问题的最优基本可行解.
*/

```
const double EPS = 1e-6, dINF = 1e15;
//最优解
#define OPTIMAL -1
```

```
//无最优解
#define UNBOUNED -2
//有无穷多个最优解
#define INFINITELY -3
//无可行解
#define INFEASIBLE -4
//找到枢纽
#define PIVOT_OK 1
inline int sgn(double x)
{
    return x < -EPS ? -1 : x > EPS ? 1 : 0;
}
const int maxn = 100;
struct LinearProgramming
{
    double a[maxn][maxn], w[maxn], goal[maxn], ans;
    int row[maxn], col[maxn];
    int N, m, O, n;//原变量数, 线性约束数, 松弛变量数, 非基本变量数
    //得到枢纽
    int Pivot(int &x, int &y)
    {
        x = -1, y = 0;
        for(int j = 0; j < n; ++j) if(sgn(a[m][j] - a[m][y]) < 0) y = j;
        if(sgn(a[m][y]) >= 0) return OPTIMAL;//已经得到最优解
        double minv = dINF, val;
        for(int i = 0; i < m; ++i)
        {
            val = w[i] / a[i][y];
            if(sgn(val) <= 0) continue;
            if(sgn(val - minv) < 0)
            {
                minv = val;
                x = i;
            }
        }
        if(x < 0) return UNBOUNED;//有可行解, 但无最优解
        return PIVOT_OK;
    }
    //单纯形方法, 第m+1行为目标函数
    int Simplex()
    {
        int k, x, y;
        while((k = Pivot(x, y)) == PIVOT_OK)
        {
            swap(row[x], col[y]);
            double tmp = a[x][y], temp;
            for(int i = 0; i <= m; ++i)
            {
                if(i == x) continue;
                temp = a[i][y] / tmp;
                for(int j = 0; j < n; ++j)
                {
                    if(j == y) continue;
                    a[i][j] -= a[x][j] * temp;
                }
                w[i] -= w[x] * temp;
            }
            for(int j = 0; j < n; ++j) a[x][j] /= tmp;
            w[x] /= tmp;
            tmp = -tmp;
            for(int i = 0; i <= m; ++i) a[i][y] /= tmp;
            a[x][y] = -a[x][y];
        }
        return k;
```

```
    }
    //二阶段法解线性规划
    int solve()
    {
        w[m] = 0;
        for(int j = 0; j < n; ++j) col[j] = j, a[m][j] = 0;
        for(int i = 0; i < m; ++i)
        {
            row[i] = i + n;
            w[m] -= w[i];
            for(int j = 0; j < n; ++j) a[m][j] -= a[i][j];
        }
        int result = Simplex();
        if(result == INFEASIBLE || sgn(w[m]) < 0) return INFEASIBLE;//无可行解
        for(int j = 0, k = 0; j < n; ++j)
        {
            if(col[j] >= n) continue;
            col[k] = col[j];
            for(int i = 0; i < m; ++i) a[i][k] = a[i][j];
            ++k;
        }
        w[m] = 0;
        n -= m;
        for(int j = 0; j < n; ++j) a[m][j] = 0;
        for(int i = 0; i < m; ++i)
        {
            if(row[i] >= N) continue;
            for(int j = 0; j < n; ++j)
            {
                a[m][j] += goal[row[i]] * a[i][j];
            }
            w[m] += goal[row[i]] * w[i];
        }
        for(int j = 0; j < n; ++j)
        {
            if(col[j] < N) a[m][j] -= goal[col[j]];
        }
        result = Simplex();
        if(result == OPTIMAL)
        {
            for(int j = 0; j < n; ++j)
                if(sgn(a[m][j]) == 0)
                {
                    result = INFINITELY;//无穷多个最优解
                    break;
                }
        }
        ans = w[m];
        return result;
    }
} lp;
```

7 字符串

7.1 KMP 以及扩展 KMP

```
///KMP
//O(n + m)
/*
next[]的含义: x[i - next[i]···i] = x[0···next[i] - 1]
next[i] 为满足x[i - z···i - 1] = x[0···z - 1]的最大z值 (就是x的自身匹配)
*/
void pre_kmp(char x[], int m, int kmpNext[])
{
    int i, j;
    j = kmpNext[i = 0] = -1;
    while(i < m)
    {
        while(j != -1 && x[i] != x[j]) j = kmpNext[j];
        if(x[++i] == x[++j]) kmpNext[i] = kmpNext[j];
        else kmpNext[i] = j;
    }
}

//返回x在y中出现的次数, 可以重叠
//x是模式串, y是文本串
int KMP_Count(char x[], int m, char y[], int n, int next[])
{
    int i = 0, j = 0, ans = 0;
    pre_kmp(x, m, next);
    while(i < n)
    {
        while(j != -1 && y[i] != x[j]) j = next[j];
        ++i, ++j;
        if(j >= m)
        {
            ans++;
            j = next[j];
        }
    }
    return ans;
}

///扩展KMP
/*
复杂度: O(n + m)
next[i]: x[i···m - 1]与x[0···m - 1]的最长公共前缀
extend[i]: y[i···n - 1]与x[0···m - 1]的最长公共前缀
*/
void pre_exkmp(const char x[], int m, int next[])
{
    for(int i = 0, j = -1, k, p; i < m; i++, j--)
        if(j < 0 || i + next[i - k] >= p)
        {
            if(j < 0) j = 0, p = i;
            while(p < m && x[p] == x[j]) j++, p++;
            next[k = i] = j;
        }
        else
            next[i] = next[i - k];
}
//x是模式串, y是文本串
void exkmp(char x[], int m, char y[], int n, int next[], int extend[])
{
    pre_exkmp(x, m, next);
```

```
    for(int i = 0, j = -1, k, p; i < n; i++, j--)
        if(j < 0 || i + next[i - k] >= p)
        {
            if(j < 0) j = 0, p = i;
            while(p < n && j < m && x[p] == y[j]) j++, p++;
            extend[k = i] = j;
        }
        else
            extend[i] = next[i - k];
}
```

7.2 回文串 palindrome

```
//manacher算法 O(n)
/*写法一
预处理: 在字符串中加入一个分隔符（不在字符串中的符号），将奇数长度的回文串和偶数长度的回文串统一;
    在字符串之前再加一个分界符（如'&'），防止比较时越界*/

void manacher(char *s, int len, int p[])
{//s = &s[0]#s[1]#...#s[len]\0
    int i, mx = 0, id;
    for(i = 1; i <= len; i++)
    {
        p[i] = mx > i ? min(p[2*id - i], mx - i) : 1;
        while(s[i + p[i]] == s[i - p[i]]) ++p[i];
        if(p[i] + i > mx) mx = p[i] + (id = i);
        p[i] -= (i & 1) != (p[i] & 1);//去掉分隔符带来的影响
    }
    //此时, p[(2<<i) + 1]为以s[i]为中心的奇数长度的回文串的长度
    //p[(2<<i)]为以s[i]和s[i+1]为中心的偶数长度的回文串的长度
}

/*写法二
将位置在[i,j]的回文串的长度信息存储在p[i+j]上
*/
void manacher2(char *s, int len, int p[])
{
    p[0] = 1;
    for(int i = 1, j = 0; i < (len<<1) - 1; ++i)
    {
        int u = i >> 1, v = i - u, r = ((j + 1) >> 1) + p[j] - 1;
        int u = i >> 1, v = i - u, r = ((j + 1) >> 1) + p[j] - 1;
        p[i] = r < v ? 0 : min(r - v + 1, p[(j<<1) - 1]);
        while(u > p[i] - 1 && v + p[i] < len && s[u - p[i]] == s[u + p[i]]) ++p[i];
        if(u + p[i] - 1 > r) j = i;
    }
}
```

7.3 哈希算法 Hash

```
///滚动哈希算法 O(n+m)
//使用于求字符串s在字符串t中出现的位置或次数 可以简单的推到二维的情况
//哈希函数H(S) = (s_1 B^{m-1} + s_2 B^{m-2} + ... + s_m B^0)%h, 其中字符串S = s_1 s_2 ... c_m, m = |S|, B为基数, 1 < B < h
    且h与b互素
//H(s_{k-1} s_k ... s_{k+m-1}) = (H(s_k s_{k+1} ... s_{k+m}) - s_k B^m + s_{k+m})%h
//单hash一般足够, 也可以使用双hash
//常用h: 10^9 + 7, 10^9 + 9, B取比字符集大的一个素数.
//注意: 将要hash的每一个字符应该至少从1开始编号, 即不能为0.

```

```
//应用1: 在许多字符串中寻找与目标串相同的字符串的个数
LL B = 71, mod = 1000000007;
int cal(string s, string t)//查询s的子串是t的个数
{
    int lens = s.length(), lent = t.length();
    if(lent > lens) return 0;
    LL BN = 1, hasht = 0, hashs = 0;
    for(int i = 0; i < lent; i++) hasht = (hasht * B + (t[i] - 'a' + 1)) % mod, BN = BN * B % mod;
    for(int i = 0; i < lent; i++) hashs = (hashs * B + (t[i] - 'a' + 1)) % mod;
    int cnt = (hasht == hashs);
    for(int i = lent; i < lens; i++)
    {
        hashs = (hashs * B + (s[i] - 'a' + 1)) % mod - BN * (s[i - lent] - 'a' + 1) % mod;
        hashs = (hashs % mod + mod) % mod;
        if(hashs == hasht) cnt++;
    }
    return hashs;
}
```

7.4 后缀数组 Suffix Array

```
///后缀数组(Suffix Array)
/*
后缀数组是指将某个字符串的所有后缀按字典序排序后得到的数组
*/
//计算后缀数组
//朴素做法 将所有后缀进行排序O(n^2 log n)采用快排 适用于m比较大的时候
///Manber–Myers O(n log^2 n)
int rk;
int sa[NUM], rk[NUM], height[NUM];
int cmp(int i, int j)
{
    if(rk[i] != rk[j])
        return rk[i] < rk[j];
    else
    {
        int ri = i + rk <= n ? rk[i + rk] : -1;
        int rj = i + rk <= n ? rk[j + rk] : -1;
        return ri < rj;
    }
}
void da(int *a, int n)
{
    int i;
    a[n] = -1;
    for(i = 0; i <= n; i++)
    {
        sa[i] = i;
        rk[i] = a[i];
    }
    for(m = 1; rk[n] < n; m <<= 1)
    {
        sort(sa, sa + n + 1, cmp);
        tmp[sa[0]] = 0;
        for(i = 1; i <= n; i++)
            tmp[sa[i]] = tmp[sa[i - 1]] + (cmp(sa[i], sa[i - 1] || cmp(sa[i - 1], sa[i])));
        for(i = 0; i <= n; i++)
            rk[i] = tmp[i];
    }
}

//应用
```

```
//基于后缀数组的字符串匹配
bool contain(string s, int *sa, string t)
{
    int a = 0, b = s.length();
    while(b - a > 1)
    {
        int c = (a + b) / 2;
        if(s.compare(sa[c], t.length(), t) < 0)a = c;
        else
            b = c;
    }
    return s.compare(sa[b], t.length(), t) == 0;
}

///倍增法模板: O(n log n)采用基数排数
//n为字符个数 r[n - 1] 要比所有a[0, n - 2]要小
//r 字符串对应的数组
//m为最大字符值+1
int sa[NUM];
int rk[NUM], height[NUM], sv[NUM], sn[NUM];
void da(char r[], int n, int m)
{
    int i, j, p, *x = rk, *y = height;
    for(i = 0; i < m; i++) sn[i] = 0;
    for(i = 0; i < n; i++) sn[x[i] = r[i]]++;
    for(i = 1; i < m; i++) sn[i] += sn[i - 1];
    for(i = n - 1; i >= 0; i--) sa[--sn[x[i]]] = i;
    for(j = p = 1; p < n; j <<= 1, m = p)
    {
        for(p = 0, i = n - j; i < n; i++) y[p++] = i;
        for(i = 0; i < n; i++) if(sa[i] >= j) y[p++] = sa[i] - j;
        for(i = 0; i < n; i++) sv[i] = x[y[i]];
        for(i = 0; i < m; i++) sn[i] = 0;
        for(i = 0; i < n; i++) sn[sv[i]]++;
        for(i = 1; i < m; i++) sn[i] += sn[i - 1];
        for(i = n - 1; i >= 0; i--) sa[--sn[sv[i]]] = y[i];
        for(swap(x, y), x[sa[0]] = 0, i = 1, p = 1; i < n; i++)
            x[sa[i]] = (y[sa[i]] == y[sa[i - 1]] && y[sa[i] + j] == y[sa[i - 1] + j]) ? p - 1 : p++;
    }
}

///DC3模板: O(3n)
int sa[NUM * 3], r[NUM * 3]; //sa数组和r数组要开三倍大小的空间
int rk[NUM], height[NUM], sn[NUM], sv[NUM];
#define F(x) ((x) / 3 + ((x) % 3 == 1 ? 0 : tb))
#define G(x) ((x) < tb ? (x) * 3 + 1 : ((x) - tb) * 3 + 2)
int cmp0(int r[], int a, int b)
{return r[a] == r[b] && r[a + 1] == r[b + 1] && r[a + 2] == r[b + 2];}
int cmp12(int r[], int a, int b, int k)
{
    if(k == 2) return r[a] < r[b] || (r[a] == r[b] && cmp12(r, a + 1, b + 1, 1));
    else return r[a] < r[b] || (r[a] == r[b] && sv[a + 1] < sv[b + 1]);
}
void sort(int r[], int a[], int b[], int n, int m)//基数排序
{
    int i;
    for(i = 0; i < m; i++) sn[i] = 0;
    for(i = 0; i < n; i++) sn[sv[i] = r[a[i]]]++;
    for(i = 1; i < m; i++) sn[i] += sn[i - 1];
    for(i = n - 1; i >= 0; i--) b[--sn[sv[i]]] = a[i];
}
void dc3(int r[], int sa[], int n, int m)
{
    int *rn = r + n, *san = sa + n, *wa = height, *wb = rk;
```

```
    int i, j, p, ta = 0, tb = (n + 1) / 3, tbc = 0;
    r[n] = r[n + 1] = 0;
    for(i = 0; i < n; i++) if(i % 3 != 0) wa[tbc++] = i;
    sort(r + 2, wa, wb, tbc, m);
    sort(r + 1, wb, wa, tbc, m);
    sort(r, wa, wb, tbc, m);
    for(p = 1, rn[F(wb[0])] = 0, i = 1; i < tbc; i++)
        rn[F(wb[i])] = cmp0(r, wb[i - 1], wb[i]) ? p - 1 : p++;
    if(p < tbc) dc3(rn, san, tbc, p);
    else for(i = 0; i < tbc; i++) san[rn[i]] = i;
    for(i = 0; i < tbc; i++) if(san[i] < tb) wb[ta++] = san[i] * 3;
    if(n % 3 == 1) wb[ta++] = n - 1;
    sort(r, wb, wa, ta, m);
    for(i = 0; i < tbc; i++) sv[wb[i] = G(san[i])] = i;
    for(i = 0, j = 0, p = 0; i < ta && j < tbc; p++)
        sa[p] = cmp12(r, wa[i], wb[j], wb[j] % 3) ? wa[i++] : wb[j++];
    for(; i < ta; p++) sa[p] = wa[i++];
    for(; j < tbc; p++) sa[p] = wb[j++];
}

///高度数组longest commest prefix
//height[i] = suffix(sa[i])和suffix(sa[i - 1])的最长公共前缀lcp(sa[i],sa[i-1])
//rk[0..n-1]:rk[i]保存的是原串中suffix[i]的名次
//height数组性质:
//任意两个suffix(j)和suffix(k)(rank[j] < rank[k])的最长公共前缀: min_{i=j+1->k} {height[rank[i]]}
//height[rank[i]] ≥ height[rank[i - 1]] - 1
int rk[maxn], height[maxn];
void cal_height(char *r, int *sa, int n)
{
    int i, j, k = 0;
    for(i = 0; i < n; i++)rk[sa[i]] = i;
    for(i = 0; i < n; height[rk[i++]] = k)
        for(k ? k-- : 0, j = sa[rk[i] - 1]; r[i + k] == r[j + k]; k++);
}
///后缀数组应用
//询问任意两个后缀的最长公共前缀: RMQ问题, min(i=j+1-->k){height[rk[i]]}
//重复子串: 字符串R在字符串L中至少出现2次, 称R是L的重复子串
//可重叠最长重复子串: O(n) height数组中的最大值
//不可重叠最长重复子串: O(n log n)变为二分答案, 判断是否存在两个长度为k的子串是相同且不重叠的.
    将排序后后缀分为若干组, 其中每组的后缀的height值都不小于k,
    然后有希望成为最长公共前缀不小于k的两个后缀一定在同一组, 然后对于每组后缀,
    判断sa的最大值和最小值之差不小于k, 如果一组满足, 则存在, 否则不存在.
//可重叠的k次最长重复子串: O(n log n) 二分答案, 将后缀分为若干组, 判断有没有一个组的后缀个数不小于k.
//不相同的子串个数: 等价于所有后缀之间不相同的前缀的个数O(n): 后缀按suffix(sa[1]), suffix(sa[2]), ···
    , suffix(n)的顺序计算, 新进一个后缀suffix(sa[k]), 将产生n - sa[k] + 1的新的前缀,
    其中height[k]的和前面是相同的, 所以suffix(sa[k])贡献n - sa[k] + 1 - height[k]个不同的子串.
    故答案是∑_{k=1}^{n} n - sa[k] - 1 - height[k].
//最长回文子串: 字符串S(长度n)变为字符串+特殊字符+反写的字符串,
    求以某字符(位置k)为中心的最长回文子串(长度为奇数或偶数), 长度为: 奇数lcp(suffix(k), suffix(2*n
    + 2 - k)); 偶数lcp(suffix(k), suffix(2*n + 3 - k)) O(n log n) RMQ:O(n)
//连续重复子串: 字符串L是有字符串S重复R次得到的.
//给定L, 求R的最大值: O(n), 枚举S的长度k, 先判断L的长度是否能被k整除, 在看lcp(suffix(1),
    suffix(k+1))是否等于n - k. 求解时只需预处理height数组中的每一个数到height[rk[1]]的最小值即可
//给定字符串, 求重复次数最多的连续重复子串O(n log n): 先穷举长度L,
    然后求长度为L的子串最多能连续出现几次. 首先连续出现1次是肯定可以的,
    所以这里只考虑至少2次的情况. 假设在原字符串中连续出现2次, 记这个子字符串为S,
    那么S肯定包括了字符r[0], r[L], r[L*2], r[L*3], ···中的某相邻的两个.
    所以只须看字符r[L*i]和r[L*(i+1)]往前和往后各能匹配到多远, 记这个总长度为K,
    那么这里连续出现了K/L+1次. 最后看最大值是多少.
//字符串A和B最长公共前缀O(|A|+|B|): 新串: A+特殊字符+B, height//k : A++B,
    对后缀数组分组(每组height值都不小于k), 每组中扫描到B时,
    统计与前面的A的后缀能产生多少个长度不小于k的公共子串, 统计得结果.

```

```
//给定n个字符串，求出现在不小于k个字符串中的最长子串O(n log n)：连接所有字符串，二分答案，然后分组，
    判断每组后缀是否出现在至少k个不同的原串中.
//给定n个字符串，求在每个字符串中至少出现两次且不重叠的最长子串O(n log n)：做法同上，
    也是先将n个字符串连起来，中间用不相同的且没有出现在字符串中的字符隔开，求后缀数组.
    然后二分答案，再将后缀分组.判断的时候，要看是否有一组后缀在每个原来的字符串中至少出现两次，
    并且在每个原来的字符串中，
    后缀的起始位置的最大值与最小值之差是否不小于当前答案(判断能否做到不重叠，
    如果题目中没有不重叠的要求，那么不用做此判断).
//给定n个字符串，求出现或反转后出现在每个字符串中的最长子串：只需要先将每个字符串都反过来写一遍，
    中间用一个互不相同的且没有出现在字符串中的字符隔开，再将n个字符串全部连起来，
    中间也是用一个互不相同的且没有出现在字符串中的字符隔开，求后缀数组.然后二分答案，再将后缀分组.
    判断的时候，要看是否有一组后缀在每个原来的字符串或反转后的字符串中出现.
    这个做法的时间复杂度为O(n log n).
```

7.5 字典树 Trie

```
///字典数 Trie
struct Trie
{
    Trie *next[26];//根据字符集的大小变化
    int cnt;//字符串个数，根据应用变化
    Trie()
    {
        memset(next, 0, sizeof(next));
        cnt = 0;
    }
};
Trie *root = NULL;
void clearTrie(Trie *p)//清空整棵字典树
{
    if(p)
    {
        for(int i = 0; i < 26; i++)
            clearTrie(p->next[i]);
        delete p;
    }
}
void initTrie()
{
    clearTrie(root);
    root = new root;
}
//将字符串str加入字符串中
void createTrie(char str[])
{
    int i = 0, id;
    Trie *p = root;
    while((id = str[i] - 'a') >= 0)//得到该字符的编号
    {
        if(p->next[id] == NULL)
        {
            p->next[id] = new Trie;
        }
        p = p->next[id];
    }
    p->cnt++;
}
//查询某字符串是否在字典树中
bool findTrie(char str[])
{
    int i = 0, id;
    Trie *p = root;
```

```
    while((id = str[i++] - 'a') >= 0)
    {
        p = p->next[id];
        if(p == NULL) return false;
    }
    return p->cnt > 0;
}
//从字典树中删除一个字符串
void deleteTrie(char str[])
{
    int i = 0, id;
    Trie *p = root;
    while((id = str[i++] - 'a') >= 0)
    {
        p = p->next[id];
        if(p == NULL) return ;
    }
    if(p->cnt) p->cnt--;
}
```

7.6 字符串循环同构的最小表示法

```
///最小表示法 Minimum Representation
//对于一个字符串S, 求S的循环的同构字符串S' 中字典序最小的一个
//O(|S|)
int MinimumRepresentation(char *s, int len)
{
    int i = 0, j = 1, k = 0, t;
    while(i < len && j < len && k < len)
    {
        t = s[(i + k) >= len ? i + k - len : i + k] - s[(j + k) >= len ? j + k - len : j + k];
        if(!t) k++;
        else
        {
            if(t > 0) i = i + k + 1;
            else j = j + k + 1;
            if(i == j) ++j;
            k = 0;
        }
    }
    return (i < j ? i : j);
}
```

7.7 字符串问题汇总

1. 字符串A的任意子串x和字符串B的任意子串y(x, y可以为空)形成一个新的字符串xy, 能构成多少种字符串?
   来源: hdu5344, 2015多校第5场1001
   做法: 问题关键在于去重. 答案为: A中不同子串的个数 × B中不同子串的个数 − $\sum_{c='a'}^{'z'}$
   (A中以字符c结尾的不同子串数 * B中以字符c开头的不同子串数). 将A反转, 然后用后缀数组维护A,
   B中不同子串的个数, 和以某字符开头的不同子串数即可. 答案会爆long long, 要用unsigned long long

2. 求后缀数组中, 某一子串S[l, r]出现的首次出现的位置, 或者最后出现的位置,
   或者求某一公共子串出现的次数
   标签: ST表, 二分
   做法: 用ST表预处理高度数组, 那么可以在O(1)时间内求出任意两个后缀之间的最长公共前缀;
   在某后缀后面(或前面)的后缀与该后缀的lcp是非增(非减)的,
   因此可以用二分求出距该后缀最远的一个与该后缀有最长公共前缀长度大于等于len的后缀.

3. 询问一个字符串S中, 比某子串S[L, R]字典序小的子串数.

来源：**Gym** 100418**C**

标签：后缀数组，二分，区间更新查询

做法：将查询离线处理，用二分得到每个被询问的子串在后缀数组中第一次出现的位置，
然后按照后缀的字典序遍历，（即遍历后缀数组）得到比每个后缀小的子串数，在此过程中，
处理询问的答案．比某个后缀小的子串数，为不是该后缀的前缀，但比该后缀小的子串数 +
是该后缀的前缀的子串数．处理时，随高度数组处理，有前缀长度为**i**(**i**= 0, 1, 2, ···, **height**[**i** +1]),
比该后缀小的子串数．直接处理会超时，要用线段树更新和查询区间．

4. 求有多少个**z**，使得由$z = (z \cdot a + c)/k(\%m)$产生的一个长度为**n**的数字序列，将小于**m**/2的数字标记为**0**，
其他的标记为**1**后与目标串**b**相同．

来源：**Gym** 100523**G**

标签：**hash**，倍增

做法：分析生成函数，可以发现这是一个有**m**个状态的有限状态自动机，
然后可以在$O(m)$时间内求出每个状态将会转移到的下一个状态．类似于**ST**表的思想，
我们预处理出从每个状态出发，长度为**2**的幂的生成字符串的**hash**值，并保存下一个会转移到的位置．
就有转移方程：
$hash[i][j] = hash[i][j-1] \times B[1 << (j-1)] + hash[pos[i][j-1]][j-1], pos[i][j] = pos[pos[i][j-1]][j-1]$,
然后我们就可以在$O(m \log n)$时间内求出每个**z**出发，长度为**n**的生成字符串的**hash**值，
然后与目标串**b**的**hash**值比较，然后统计即可．然而，由于内存限制，我们要用滚动数组预处理，
并在此过程中求出长度为**n**的生成字符串的**hash**值．

时间复杂度：$Om \log n$

8 计算几何

8.1 计算几何基础

```
//精度设置
const double EPS = 1e-6;
//点(向量)的定义和基本运算
/*
向量点积$a \cdot b = |a||b| \cos\theta$
点积>0, 表示两向量夹角为锐角
点积=0, 表示两向量垂直
点积<0, 表示两向量夹角为钝角
*/
/*
向量叉积 $a \times b = |a||b| \sin\theta$
叉积>0, 表示向量b在当前向量逆时针方向
叉积=0, 表示两向量平行
叉积<0, 表示向量b在当前向量顺时针方向
*/
inline int sgn(double x) {if(x < -EPS)return -1; return x > EPS ? 1 : 0;}
struct Point
{
    double x, y;
    Point(double _x = 0.0, double _y = 0.0): x(_x), y(_y) {}
    Point operator + (const Point &b)const {return Point(x + b.x, y + b.y);}//向量加法
    Point operator - (const Point &b)const {return Point(x - b.x, y - b.y);}//向量减法
    double operator * (const Point &b)const {return x * b.x + y * b.y;}//向量点积
    double operator ^ (const Point &b)const {return x * b.y - y * b.x;} //向量叉积
    Point operator * (double b) {return Point(x * b, y * b);}//标量乘法
    Point rot(double ang) {return Point(x * cos(ang) - y * sin(ang), x * sin(ang) + y *
     cos(ang));}//旋转
    double norm() {return sqrt(x * x + y * y);}//向量的模
};
//直线 线段定义
//直线方程: 两点式: $(x_2 - x_1)(y - y_1) = (y_2 - y_1)(x - x_1)$
struct Line
{
    Point s, e;
    //double k;
    Line() {}
    Line(Point _s, Point _e)
    {
        s = _s, e = _e;
        //k = atan2(e.y - s.y, e.x - s.x);
    }
    //求两直线交点
    //返回-1两直线重合, 0 相交, 1 平行
    pair<int, Point> operator &(Line b)
    {
        if(sgn((s - e) ^ (b.s - b.e)) == 0)
        {
            if(sgn((s - b.e) ^ (b.s - b.e)) == 0) return make_pair(-1, s);//重合
            else return make_pair(1, s);//平行
        }
        double t = ((s - b.s) ^ (b.s - b.e)) / ((s - e) ^ (b.s - b.e));
        return make_pair(0, Point(s.x + (e.x - s.x) * t, s.y + (e.y - s.y) * t));
    }
};

//两点间距离
double dist(Point &a, Point &b) {return (a - b).norm();}

/*判断点p在线段l上
```

```
 * (p - l.s) ^ (l.s - l.e) = 0; 保证点p在直线L上
 * p在线段l的两个端点l.s,l.e为对角定点的矩形内
 */
bool Point_on_Segment(Point &p, Line &l)
{
    return sgn((p - l.s) ^ (l.s - l.e)) == 0 &&
           sgn((p.x - l.s.x) * (p.x - l.e.x)) <= 0 &&
           sgn((p.y - l.s.y) * (p.y - l.e.y)) <= 0;
}
//判断点p在直线l上
bool Point_on_Line(Point &p, Line &l)
{
    return sgn((p - l.s) ^ (l.s - l.e)) == 0;
}

/*判断两线段l1, l2相交
 * 1. 快速排斥实验: 判断以l1为对角线的矩形是否与以l2为对角线的矩形是否相交
 * 2. 跨立实验: l2的两个端点是否在线段l1的两端
 */
bool seg_seg_inter(Line seg1, Line seg2)
{
    return
        sgn(max(seg1.s.x, seg1.e.x) - min(seg2.s.x, seg2.e.x)) >= 0 &&
        sgn(max(seg2.s.x, seg2.e.x) - min(seg1.s.x, seg1.e.x)) >= 0 &&
        sgn(max(seg1.s.y, seg1.e.y) - min(seg2.s.y, seg2.e.y)) >= 0 &&
        sgn(max(seg2.s.y, seg2.e.y) - min(seg1.s.y, seg1.e.y)) >= 0 &&
        sgn((seg2.s - seg1.e) ^ (seg1.s - seg1.e)) * sgn((seg2.e - seg1.e) ^ (seg1.s - seg1.e)) <=
    0 &&
        sgn((seg1.s - seg2.e) ^ (seg2.s - seg2.e)) * sgn((seg1.e - seg2.e) ^ (seg2.s - seg2.e)) <=
    0;
}

//判断直线与线段相交
bool seg_line_inter(Line &line, Line &seg)
{
    return sgn((seg.s - line.e) ^ (line.s - line.e)) * sgn((seg.e - line.e) ^ (line.s - line.e)) <=
    0;
}

//点到直线的距离, 返回垂足
Point Point_to_Line(Point p, Point l)
{
    double t = ((p - l.s) * (l.e - l.s)) / ((l.e - l.s) * (l.e - l.s));
    return Point(l.s.x + (l.e.x - l.s.x) * t, l.s.y + (l.e.y - l.s.y) * t);
}
//点到线段的距离
//返回点到线段最近的点
Point Point_to_Segment(Point p, Line seg)
{
    double t = ((p - seg.s) * (seg.e - seg.s)) / ((seg.e - seg.s) * (seg.e - seg.s));
    if(t >= 0 && t <= 1)
        return Point(seg.s.x + (seg.e.x - seg.s.x) * t, seg.s.y + (seg.e.y - seg.s.y) * t);
    else if(sgn(dist(p, seg.s) - dist(p, seg.e) <= 0))
        return seg.s;
    else
        return seg.e;
}
```

8.2 三角形 Triangle

```
///三角形
/*
```

设三角形$\triangle ABC$的三个顶点A, B, C (对应内角角度为A, B, C); 对应的三条边为a, b, c (对应边长为a, b, c).
三点坐标对应为: $A(x_1, y_1)$, $B(x_2, y_2)$, $C(x_3, y_3)$
性质:
    三角形不等式: 三角形两边之和大于第三边, 两边之差的绝对值小于第三边.
        三角形任意一个外角大于不相邻的一个内角.
    勾股定理: 三角形是直角三角形($C = 90°$), 则$a^2 + b^2 = c^2$. 反之亦然.
    正弦定理: (R为三角形外接圆半径)

$$\frac{a}{\sin A} = \frac{b}{\sin B} = \frac{c}{\sin C} = 2R$$

    余弦定理:
        $a^2 = b^2 + c^2 - 2bc \cdot \cos A$
        $b^2 = a^2 + c^2 - 2ac \cdot \cos B$
        $c^2 = a^2 + b^2 - 2ab \cdot \cos C$
角度:
    三角形两内角之和, 等于第三角的外角.
    三角形的内角和为$180°$.
    三角形分类: 钝角三角形(其中一角是钝角($> 90°$)的三角形),
    直角三角形(其中一角是直角($= 90°$)的三角形), 和锐角三角形(三个角都是锐角($< 90°$)的三角形.
    等边三角形的内角均为$60°$.
    等腰三角形两底角相等.
面积:
    设a, b为已知的两边, C为其夹角, 三角形面积$\triangle = \frac{1}{2}ab\sin C$
    已知三角形的三边, 有:
        海伦公式: 令$p = \frac{a+b+c}{2}$, 则$\triangle = \sqrt{p(p-a)(p-b)(p-c)}$
        秦九韶的三斜求积法:$\triangle = \sqrt{\frac{1}{4}\left[c^2a^2 - \left(\frac{c^2+a^2-b^2}{2}\right)^2\right]}$
        幂和: $\triangle = \frac{1}{4}\sqrt{(a^2+b^2+c^2)^2 - 2(a^4+b^4+c^4)}$
    已知一边a, 和该边上的高h_a, 则$\triangle = \frac{ah_a}{2}$
    已知三点坐标, 面积为下列行列式的绝对值:

$$\frac{1}{2}\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}$$

三角的五心:
    重心(形心 Centroid)
        定义: 三条中线的交点.
        坐标: 三点坐标的算术平均, 即$\left(\frac{x_1+x_2+x_3}{2}, \frac{y_1+y_2+y_3}{2}\right)$.
        三角形的重心与三顶点连线, 所形成的六个三角形面积相等.
        顶点到重心的距离是中线的$\frac{2}{3}$.
        重心到三角形3个顶点距离的平方和最小.
        以重心为起点, 以三角形三定点为终点的三条向量之和等于零向量.
*/
```
Point MassCenter(Point A, Point B, Point C)
{
    return (A + B + C) * (1.0 / 3.0);
}
```
/*
    内心I(Inner Center)
        定义: 三个内角的角平分线的交点.
        坐标: 三点坐标的面积加权平均, 即$\left(\frac{ax_1+bx_2+cx_3}{a+b+c}, \frac{ay_1+by_2+cy_3}{a+b+c}\right)$.
        内心到三角形三边的距离相等, 等于内切圆半径r.
        内心是三角形内切圆的圆心, 内切圆半径r, 有$\triangle = \frac{1}{2}(a+b+c)r$.
        直角三角形两股和等于斜边长加上该三角形内切圆直径, 即a + b = c + 2r,
        由此性质再加上勾股定理$a^2 + b^2 = c^2$, 可推得: $\triangle = r(r+c)$
*/
```
Point InnerCenter(Point A, Point B, Point C)
{
    double a = dist(B, C), b = dist(A, C), c = dist(A, B);
    return (A * a + B * b + C * c) * (1.0 / (a + b + c));
}
```
/*

外心*(Circum Center)*

定义: 三条边垂直平分线的交点.

坐标:

$$\left(\frac{\begin{vmatrix} x_1^2+y_1^2 & 1 & y_1 \\ x_2^2+y_2^2 & 1 & y_2 \\ x_3^2+y_3^2 & 1 & y_3 \end{vmatrix}}{2\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}}, \frac{\begin{vmatrix} x_1^2+y_1^2 & x_1 & 1 \\ x_2^2+y_2^2 & x_2 & 1 \\ x_3^2+y_3^2 & x_3 & 1 \end{vmatrix}}{2\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}}\right)$$

外心到三个顶点的距离都相等, 等于外接圆的半径*R*.

直角三角形的外心是斜边的中点, 外接圆半径*R*为斜边的一半; 钝角三角形的外心在三角形外, 靠近最长边; 锐角三角形的外心在三角形内.

外心是三角形外接圆的圆心, 外接圆半径*R*, 有$\triangle = \frac{abc}{4R}$

```
*/
//采用求垂直平分线相交的方法
Point CircumCenter(Point A, Point B, Point C)
{
    return  (Line((A + B) * 0.5, (A - B).rot(PI * 0.5) + ((A + B) * 0.5)) &
             Line((B + C) * 0.5, (B - C).rot(PI * 0.5) + ((B + C) * 0.5))).second;
}
//公式法
Poiny CircumCenter(Point A, Point B, Point C)
{
    Point t1 = B - A, t2 = C - A, t3((t1 * t1) * 0.5, (t2 * t2) * 0.5);
    swap(t1.y, t2.x);
    return A + Point(t3 ^ t2, t1 ^ t3) * (1.0 / (t1 ^ t2));
}
/*
```

垂心*H(Ortho Center)*

定义: 三条高的交点.

坐标:

$$\left(\frac{\begin{vmatrix} x_2x_3+y_2y_3 & 1 & y_1 \\ x_3x_1+y_3y_1 & 1 & y_2 \\ x_1x_2+y_1y_2 & 1 & y_3 \end{vmatrix}}{\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}}, \frac{\begin{vmatrix} x_2x_3+y_2y_3 & x_1 & 1 \\ x_3x_1+y_3y_1 & x_2 & 1 \\ x_1x_2+y_1y_2 & x_3 & 1 \end{vmatrix}}{\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}}\right)$$

垂心分每条高线的两部分乘积相等, 即$AH \times HH_A = BH \times HH_B = CH \times HH_C$.

直角三角形垂心为*C*($\angle C = 90^\circ$), 锐角三角形的垂心在三角形内部, 钝角三角形的垂心在三角形外部.

垂心到三角形一顶点距离为此三角形外心到此顶点对边距离的*2*倍.

一个三角形*ABC*的三个顶点*A*, *B*, *C*和它的垂心*H*构成一个垂心组: *A*, *B*, *C*, *H*. 也就是说, 这四点中任意的三点的垂心都是第四点.

由海伦公式和三角形面积公式, 可以推出各边上的高的长度.

反海伦公式

如果设$h_s = \frac{h_a^{-1} + h_b^{-1} + h_c^{-1})}{2}$(, 那么有以下类似于海伦公式的三角形面积公式

$$S^{-1} = 4\sqrt{h_s(h_s - h_a^{-1})(h_s - h_b^{-1})(h_s - h_c^{-1})}$$

.

```
*/
Point OrthoCenter(Point A, Point B, Point C)
{
    return MassCenter(A, B, C) * 3.0 - CircumCenter(A, B, C) * 2.0;
}
/*
```

旁心*J*

定义: 三角形一内角平分线和另外两顶点处的外角平分线的点.

旁心是三角形旁切圆(与三角形的一边和其他两边的延长线相切的圆)的圆心, 每个三角形有三个旁心, 而且一定在三角形外.

旁心到三边的距离相等.

旁切圆与三角形的边(或其延长线)相切的点称为旁切点.

某顶点和其对面的旁切点将三角形的圆周等分为两半.

三个旁心与内心组成一个垂心组，也就是说内心是三个旁心所组成的三角形的垂心，而相应的三个垂足则是旁心所对的顶点.

旁心坐标和旁切圆半径:

$$J_A=\left(\frac{bx_2+cx_3-ax_1}{b+c-a},\frac{by_2+cy_3-ay_1}{b+c-a}\right),r_A=\frac{2\triangle}{b+c-a}$$

,

$$J_B=\left(\frac{ax_1+cx_3-bx_2}{a+c-b},\frac{ay_1+cy_3-by_2}{a+c-b}\right),r_B=\frac{2\triangle}{a+c-b}$$

,

$$J_C=\left(\frac{bx_2+ax_1-cx_3}{a+b-c},\frac{by_2+ay_1-cy_3}{a+b-c}\right),r_C=\frac{2\triangle}{a+b-c}$$

关系:

等边三角形四心(除旁心)重合.

等腰三角形重心，中心和垂心都位于顶点向底边的垂线.

欧拉线: 三角形的垂心，外心，重心和九点圆圆心的一条直线.

欧拉线上的四点中九点圆圆心到垂心和外心的距离相等，而且重心到外心的距离是重心到垂心距离的一半.

注意内心一般不在欧拉线上，除了等腰三角形外.

重心，内心，奈格尔点，类似重心四点共线.

三角形的外接圆半径R，内切圆半径r 以及内外心间距OI 之间有如下关系: $R^2-OI^2=2Rr$.

内切圆在一边上的切点与旁切圆在该边的切点之间的距离恰好是另外两边的差(绝对值).

对于一个顶点 (比如A) 所对的旁切圆，三角形ABC的外接圆半径R，A所对旁切圆半径r_A以及内外心间距OJ_A之间关系: $OJ_A^2-R^2=2Rr_A$

内心I，B，C，J_A四点共圆，其中IJ_A是这个圆的直径，而圆心P_A在三角形ABC的外接圆上，并且过BC的中垂线，即等分劣弧BC. 对其它两边也有同样的结果.

三角形内切圆的半径r与三个顶点上的高h_a,h_b,h_c有如下的关系:

$$\frac{1}{r}=\frac{1}{h_a}+\frac{1}{h_b}+\frac{1}{h_c}$$

三个旁切圆的半径也和高有类关系:

$$\frac{1}{r_A}=-\frac{1}{h_a}+\frac{1}{h_b}+\frac{1}{h_c}.$$

$$\frac{1}{r_B}=\frac{1}{h_a}-\frac{1}{h_b}+\frac{1}{h_c}.$$

$$\frac{1}{r_C}=\frac{1}{h_a}+\frac{1}{h_b}-\frac{1}{h_c}.$$

费马点:

三角形内到三角形的三个顶点A，B，C的距离之和PA+PB+PC最小的点P.

每个三角形只有一个费马点.

费马点求法:

当有一个内角不小于120°时，费马点为此角对应顶点.

当三角形的内角都小于120°时，以三角形的每一边为底边，向外做三个正三角形$\triangle ABC'$，$\triangle BCA'$，$\triangle CAB'$，连接CC'，BB'，AA'，则三条线段的交点就费马点. (此时$\angle APB=\angle APC=\angle BPC=120°$)

九点圆(又称欧拉圆，费尔巴哈圆)，在平面几何中，对任何三角形，九点圆通过三角形三边的中点，三高的垂足，和顶点到垂心的三条线段的中点.

九点圆定理指出对任何三角形，这九点必定共圆;

九点圆的半径是外接圆的一半. 且九点圆平分垂心与外接圆上的任一点的连线;

圆心在欧拉线上. 且在垂心到外心的线段的中点;

九点圆和三角形的内切圆和旁切圆相切(费尔巴哈定理)，切点称为费尔巴哈点;

圆周上四点任取三点做三角形，四个三角形的九点圆圆心共圆(库利奇－大上定理).

奈格尔点:

旁切点J_A,J_B,J_C分别是三旁切圆和三条边的切点，直线AJ_A,BJ_B,CJ_C共点交于三角形ABC的奈格尔点N.

另一种方法构造J_A，从点A出发沿着三角形ABC的边走到半周长位置，类似的得到J_B和J_C.

因为这个构造，奈格尔点有时也被称为平分周长点(或译界心).

半角定理: 三角形的三个角的半角的的正切和三边有如下关系:

$$\tan\frac{A}{2}=\frac{p}{b+c-a},\ \tan\frac{B}{2}=\frac{p}{a+c-b},\ \tan\frac{C}{2}=\frac{p}{a+b-c}$$

其中，$p=\sqrt{\frac{(b+c-a)(a+c-b)(a+b-c)}{a+b+c}}$.

角平分线长度

设在三角形ABC中，已知三边a，b，c，若三个角A，B，C的角平分线分别为t_a,t_b,t_c

则用三边表示三条内角平分线长度公式为

$$t_a=\frac{1}{b+c}\sqrt{(b+c+a)(b+c-a)bc}$$

$$t_b=\frac{1}{a+c}\sqrt{(a+c+b)(a+c-b)ac}$$

$$t_c=\frac{1}{a+b}\sqrt{(a+b+c)(a+b-c)ab}$$

等角共轭

```
    几何学中，设点 P 是三角形 ABC 平面上一点，作直线 PA, PB 和 PC 分别关于角 A, B 和 C
    的平分线的反射，这三条反射线必然交于一点，称此点为 P 关于三角形 ABC 的等角共轭.
    (这个定义只对点，不是对三角形 ABC 的边.)
    内心I的等角共轭点是自身. 垂心H的等角共轭点是外心O. 重心的等角共轭点是类似重心K.
*/
```

8.3 多边形 Polygon

```
/// 多边形
//*判断点在凸多边形内
//点形成一个凸包，而且按逆时针排序 (如果是顺时针把里面的 < 0改为 > 0)
//点的编号: 0~n-1
//返回值: -1: 点在凸多边形外; 0: 点在凸多边形边界上; 1: 点在凸多边形内
int inConvexPoly(Point a, Point p[], int n)
{
    p[n] = p[0];
    for(int i = 0; i < n; i++)
    {
        if(sgn((p[i] - a) ^ (p[i + 1] - a)) < 0)return -1;
        else if(OnSeg(a, Line(p[i], p[i + 1])))return 0;
    }
    return 1;
}
//*判断点在任意多边形内
//射线法, poly[]的顶点数要大于等于3,点的编号0~n-1
//返回值
//-1:点在凸多边形外
//0:点在凸多边形边界上
//1:点在凸多边形内
int inPoly(Point p, Point poly[], int n)
{
    int cnt;
    Line ray, side;
    cnt = 0;
    ray.s = p;
    ray.e.y = p.y;
    ray.e.x = -100000000000.0;//-INF,注意取值防止越界
    for(int i = 0; i < n; i++)
    {
        side.s = poly[i];
        side.e = poly[(i + 1) % n]; //*判断点在任意多边形内
        if(OnSeg(p, side))return 0;
        //如果平行轴则不考虑
        if(sgn(side.s.y - side.e.y) == 0)
            continue;
        if(OnSeg(side.s, ray))
        {
            if(sgn(side.s.y - side.e.y) > 0)cnt++;
        }
        else if(OnSeg(side.e, ray))
        {
            if(sgn(side.e.y - side.s.y) > 0)cnt++;
        }
        else if(inter(ray, side))
            cnt++;
    }
    if(cnt % 2 == 1)return 1;
    else return -1;
}

//判断凸多边形
//允许共线边
```

```
//点可以是顺时针给出也可以是逆时针给出
//点的编号1~n-1
bool isconvex(Point poly[], int n)
{
    bool s[3];
    memset(s, false, sizeof(s));
    for(int i = 0; i < n; i++)
    {
        s[sgn((poly[(i + 1) % n] - poly[i]) ^ (poly[(i + 2) % n] - poly[i])) + 1] = true;
        if(s[0] && s[2])return false;
    }
    return true;
}
//多边形的面积: 分解为多个三角形的面积和
double CalArea(vector<Point> p)
{
    double res = 0;
    p.push_back(p[0]);
    for(int i = 1; i < (int)p.size(); i++)
        res += (p[i] ^ p[i - 1]);//计算由原点, p[i], p[i-1]构成的三角形的有向面积
    return fabs(res * 0.5);//三角面积需乘以0.5, 以及取正
}
//多边形的质心: 三角质心坐标的的面积加权和
Point Centroid(vector<Point> p)//按顺时针方向或逆时针方向排列
{
    Point c = Point(0.0, 0.0);
    double S = 0.0, s;
    for(int i = 2; i < (int)p.size(); i++)
    {
        s = ((p[i] - p[0]) ^ (p[i] - p[i - 1]));//求三角形面积(*0.5)
        S += s;
        c = c + (p[0] + p[i - 1] + p[i]) * s;//求三角形质心(/3.0)
    }
    c = c * (1.0 / (3.0 * S));
    return c;
}
```

8.4 圆 Circle

```
/// 圆
/*
圆心(x, y), 半径r
面积: πr^2, 周长: 2πr
弧长: θr
三点圆方程:
```

$$\begin{vmatrix} x^2+y^2 & x & y & 1 \\ x_1^2+y_1^2 & x_1 & y_1 & 1 \\ x_2^2+y_2^2 & x_2 & y_2 & 1 \\ x_3^2+y_3^2 & x_3 & y_3 & 1 \end{vmatrix} = 0$$

```
四边形的内切圆和外切圆:
    不是所有的四边形都有内切圆, 拥有内切圆的四边形称为圆外切四边形.
    凸四边形ABCD有内切圆当且仅当两对对边之和相等: AB + CD = AD + BC.
    圆外切四边形的面积和内切圆半径的关系为: S_ABCD = rs, 其中s 为半周长.
    同时拥有内切圆和外接圆的四边形称为双心四边形. 这样的四边形有无限多个. 若一个四边形为双心四边形,
    那么其内切圆在两对对边的切点的连线相互垂直. 而只要在一个圆上选取两条相互垂直的弦,
    并过相应的顶点做切线, 就能得到一个双心四边形.
    正多边形必然有内切圆, 而且其内切圆的圆心和外接圆的圆心重合, 都在正多边形的中心. 边长为a
    的正多边形的内切圆半径为:
```

$$r_n = \frac{a}{2}\cot\left(\frac{\pi}{n}\right)$$

```
其内切圆的面积为:
```

$$s_n = \pi r_n^2 = \frac{\pi a^2}{4}\cot^2\left(\frac{\pi}{n}\right)$$

内切圆面积s_n与正多边形的面积S_n之比为：

$\varphi_n = \frac{s_n}{S_n} = \frac{\frac{\pi a^2}{4}\cot^2(\frac{\pi}{n})}{\frac{na}{2}[\frac{a}{2}\cot(\frac{\pi}{n})]} = \frac{\pi}{n}\cot(\frac{\pi}{n})$

故此，当正多边形的边数n趋向无穷时，

$$\lim_{n\to\infty}\varphi_n = \lim_{n\to\infty}\frac{\pi}{n}\cot\left(\frac{\pi}{n}\right) = \lim_{n\to\infty}\cos^2\left(\frac{\pi}{n}\right) = 1$$

(婆罗摩笈多公式)若圆内接四边形的四边边长分别是a, b, c, d, 则其面积为$S = \sqrt{(p-a)(p-b)(p-c)(p-d)}$,
其中p为半周长:$p = \frac{a+b+c+d}{2}$. 在所有周长为定值2p的圆内接四边形中，面积最大的是正方形.

四边形外接圆的半径为，$R = \frac{\sqrt{(ac+bd)(ad+bc)(ab+cd)}}{4S}$

(泰勒斯定理)若A, B, C是圆周上的三点，且AC是该圆的直径，那么∠ABC必然为直角. 或者说，
直径所对的圆周角是直角.

泰勒斯定理的逆定理同样成立，即：直角三角形中，直角的顶点在以斜边为直径的圆上.

所有的正多边形都有外接圆，外接圆的圆心和正多边形的中心重合. 边长为a 的n 边正多边形外接圆的半径为：

$$R_n = \frac{a}{2\sin(\frac{\pi}{n})} = \frac{a}{2}\csc\left(\frac{\pi}{n}\right)$$

面积为：

$$A_n = \pi R_n^2 = \frac{\pi a^2}{4\sin^2(\frac{\pi}{n})} = \frac{\pi a^2}{4}\csc^2(\frac{\pi}{n})$$

正n 边形的面积s_n与其外接圆的面积A_n之比为

$$\rho_n = \frac{S_n}{A_n} = \frac{\frac{na^2}{4}\cot(\frac{\pi}{n})}{\frac{\pi a^2}{4}\csc^2(\frac{\pi}{n})} = \frac{n}{\pi}\cos\left(\frac{\pi}{n}\right)\sin\left(\frac{\pi}{n}\right) = \frac{n}{2\pi}\sin\left(\frac{2\pi}{n}\right)$$

故此，当n趋向无穷时，

$$\lim_{n\to\infty}\rho_n = \lim_{n\to\infty}\frac{n}{2\pi}\sin\left(\frac{2\pi}{n}\right) = 1$$

另外，其内切圆的面积S_n与其外接圆的面积A_n之比为：

$$\tau_n = \frac{s_n}{A_n} = \frac{s_n}{S_n}\cdot\frac{S_n}{A_n} = \varphi_n\rho_n = \left[\frac{\pi}{n}\cot\left(\frac{\pi}{n}\right)\right]\left[\frac{n}{\pi}\cos\left(\frac{\pi}{n}\right)\sin\left(\frac{\pi}{n}\right)\right] = \cos^2\left(\frac{\pi}{n}\right)$$

*/

```
//圆与(直线)线段的相交
//num表示圆O(o, r)与线段(s, e)的交点数, res里存储的是交点
void Circle_cross_Segment(Point s, Point e, Point o, double r, Point res[], int &num)
{
    double dx = e.x - s.x, dy = e.y - s.y;
    double A = dx * dx + dy * dy;
    double B = 2.0 * dx * (s.x - o.x) + 2.0 * dy * (s.y - o.y);
    double C = sqr(s.x - o.x) + sqr(s.y - o.y) - r * r;
    double delta = B * B - 4.0 * A * C;
    num = 0;
    if(sgn(delta) < 0) return ;
    delta = sqrt(max(0.0, delta));
    double k1 = (-B - delta) / (2.0 * A);
    double k2 = (-B + delta) / (2.0 * A);
    //if(sgn(k1 - 1.0) <= 0 && sgn(k1) >= 0) //圆与线段相交条件判断
    res[num++] = Point(s.x + k1 * dx, s.y + k1 * dy);
    //if(sgn(k2 - 1.0) <= 0 && sgn(k2) >= 0)
    res[num++] = Point(s.x + k2 * dx, s.y + k2 * dy);
}
```

8.5 凸包 ConvexHull

```
//凸包Convex Hull
//

//Graham算法O(n log n)
//写法一：按直角坐标排序
//直角坐标序比较 (水平序)
bool cmp(const Point &a, const Point &b)//先比较x, 后比较x均可
```

```
{
    return a.x < b.x || (a.x == b.x && a.y < b.y);
}
vector<Point> Graham(vector<Point> p)
{
    int n = p.size();
    sort(p.begin(), p.end(), cmp);
    vector<Point> res(n + n + 5);
    int top = 0;
    for(int i = 0; i < n; i++)//扫描下凸壳
    {
        while(top > 1 && sgn((res[top - 1] - res[top - 2]) ^ (p[i] - res[top - 2])) <= 0) top--;
        res[top++] = p[i];
    }
    int k = top;
    for(int i = n - 2; i >= 0; i--)//扫描上凸壳
    {
        while(top > k && sgn((res[top - 1] - res[top - 2]) ^ (p[i] - res[top - 2])) <= 0) top--;
        res[top++] = p[i];
    }
    if(top > 1) top--;//最后一个点和第一个点一样，可以不去掉，某些计算时方便一些
    res.resize(top);
    return res;
}

//写法二：按极坐标排序
//可能由于精度问题出现RE
Point p0;//p0 原点集中最左下方的点
bool cmp(const Point &p1, const Point &p2) //极角排序函数 ， 角度相同则距离小的在前面
{
    double tmp = (p0 - p2) ^ (p1 - p2);
    if(sgn(tmp) > 0) return true;
    else if(sgn(tmp) == 0 && sgn(((p0 - p1) * (p0 - p1)) - ((p0 - p2) * (p0 - p2))) < 0)  return
     true;
    else return false;
}

vector<Point> Graham(vector<Point> p)
{
    //p0
    int pn = p.size();
    for(int i = 1; i < pn; i++)
        if(p[i].x < p[0].x || (p[i].x == p[0].x && p[i].y < p[0].y))
            swap(p[i], p[0]);
    p0 = p[0];
    //sort
    sort(p.begin() + 1, p.end(), cmp);
    vector<Point> stk(pn * 2 + 5);
    int top = 0;
    for(int i = 0; i < pn; i++)
    {
        while(top > 1 && sgn((stk[top - 1] - stk[top - 2]) ^ (p[i] - stk[top - 2])) <= 0) top--;
        stk[top++] = p[i];
    }
    stk.resize(top);
    return stk;
}
```

8.6 半平面交 Half-plane Cross

```
//半平面交
inline int sgn(double x)
```

```
{
    if(x < -EPS) return -1;
    return x > EPS ? 1 : 0;
}
struct Point
{
    double x, y;
    Point(double xx = 0.0, double yy = 0.0): x(xx), y(yy) {}
    void in()
    {
        scanf("%lf%lf", &x, &y);
    }
    Point operator + (Point b)
    {
        return Point(x + b.x, y + b.y);
    }
    Point operator - (Point b)
    {
        return Point(x - b.x, y - b.y);
    }
    Point operator *(double b)
    {
        return Point(x * b, y * b);
    }
    double operator *(Point b)
    {
        return x * b.x + y * b.y;
    }
    double operator ^(Point b)
    {
        return x * b.y - y * b.x;
    }
} p[NUM], ans[NUM];
struct Line
{
    Point s, e;
    double k;
    Line() {}
    Line(Point _s, Point _e): s(_s), e(_e) {k = atan2(e.y - s.y, e.x - s.x);}
    void K()
    {
        k = atan2(e.y - s.y, e.x - s.x);
    }
    Point operator &(Line &b)
    {
        return s + (e - s) * (((s - b.s) ^ (b.s - b.e)) / ((s - e) ^ (b.s - b.e)));
    }
} L[NUM], que[NUM];
int ln;

inline bool HPIcmp(Line a, Line b)
{
    if(sgn(a.k - b.k) != 0)return sgn(a.k - b.k) < 0;
    return ((a.s - b.s) ^ (b.e - b.s)) < 0;
}
void HPI(Line line[], int n, Point res[], int &resn)
{
    int tot = n;
    sort(line, line + n, HPIcmp);
    tot = 1;
    for(int i = 1; i < n; i++)
    {
        if(sgn(abs(line[i].k - line[i - 1].k)) != 0)
            line[tot++] = line[i];
```

```
    }
    int head = 0, tail = 1;
    que[0] = line[0];
    que[1] = line[1];
    resn = 0;
    for(int i = 2; i < tot; i++)
    {
        if(sgn((que[tail].e - que[tail].s) ^ (que[tail - 1].e - que[tail - 1].s)) == 0 ||
                sgn((que[head].e - que[head].s) ^ (que[head + 1].e - que[head + 1].s)) == 0)
            return ;
        while(head < tail && sgn(((que[tail]&que[tail - 1]) - line[i].s) ^ (line[i].e - line[i].s))
    > 0)
            tail--;
        while(head < tail && sgn(((que[head]&que[head + 1]) - line[i].s) ^ (line[i].e - line[i].s))
    > 0)
            head++;
        que[++tail] = line[i];
    }
    while(head < tail && sgn(((que[tail]&que[tail - 1]) - que[head].s) ^ (que[head].e -
     que[head].s)) > 0)
        tail--;
    while(head < tail && sgn(((que[head]&que[head + 1]) - que[tail].s) ^ (que[tail].e -
     que[tail].s)) > 0)
        head++;
    if(tail <= head + 1) return ;
    for(int i = head; i < tail; i++)
        res[resn++] = que[i] & que[i + 1];
    if(head < tail - 1)
        res[resn++] = que[head] & que[tail];
}
```

8.7 立体几何

```
///三维几何
///点
inline int sgn(double x) {if(x < -EPS)return -1; return x > EPS ? 1 : 0;}
struct Point
{
    double x, y, z;
    Point(double _x = 0.0, double _y = 0.0, double _z = 0.0) {x = _x, y = _y, z = _z;}
    void in() {scanf("%lf%lf%lf", &x, &y, &z);}
    double operator *(const Point b)const {return x * b.x + y * b.y + z * b.z;} //点积
    Point operator ^(const Point b)const {return Point(y * b.z - z * b.y, z * b.x - x * b.z, x *
     b.y - y * b.x);}//叉积
    Point operator *(const double b)const {return Point(x * b, y * b, z * b);}//标量乘
    Point operator + (const Point b)const {return Point(x + b.x, y + b.y, z + b.z);}//向量加
    Point operator - (const Point b)const {return Point(x - b.x, y - b.y, z - b.z);}//向量减
    Point rot(double ang){}
};
//判断四点共面(coplanar)
bool is_coplanar(Point p[4])
{
    Point normal = (p[0] - p[1]) ^ (p[1] - p[2]); //求前三点的法向量
    return sgn(normal * (p[3] - p[0])) == 0; //判断与第四点的连线是否与法向量垂直
}

///线
///面Plane
//三点确定一个平面，其法向量(normal vector)为:
Point getPlaneNormal(Point a, Point b, Point c)
{
    return (a - b) ^ (a - c);
```

```
}
///体
//四面体体积
double V(Point a, Point b, Point c, Point d)
{
    return ((a - b) ^ (a - c)) * (a - d) * (1.0 / 6.0);
}
//四面体重心
Point zhongxin(Point a, Point b, Point c, Point d)
{
    //
//  Point res;
//  res.x = (a.x + b.x + c.x + d.x)/4.0;
//  res.y = (a.y + b.y + c.y + d.y)/4.0;
//  res.z = (a.z + b.z + c.z + d.z)/4.0;
//  return res;
    return (a + b + c + d) * 0.25;
}
//多面体的重心
//将多面体分解为若干个四面体, 重心为体积的加权和 zhongxin = (v1*zx1 + v2*zx2 + ... + vn*zxn)/(v1 +
    v2 + ... + v3)

//空间三角形的面积
double Area(Point a, Point b, Point c)
{
    return (a - b) * (a - c);
}
//点到平面的距离 h = 3.0*V/S
Point dis(Point p, Point a, Point b, Point c)
{
    return fabs(3.0 * V(p, a, b, c) / S(a, b, c));
}
```

8.8 格点 Lattice Point

```
//格点(Lattice Point)上的几何
//定义: 直角坐标系中横纵坐标均为整数的点称为格点(或整点).
/*性质:
1. 格点多边形的面积必为整数或半整数(奇数的一半)
2. 格点关于格点的对称点为格点
3. 格点多边形面积公式(pick公式):
    格点多边形的面积S = 多边形内部格点数a + 边上的格点数b/2 - 1
4. 格点正多边形只能是正方形
5. 格点三角形边界上无其他格点, 内部有一个格点, 则该点为此三角形的重心.
*/
```

9 搜索等

```
///二分搜索
//对于某些满足单调性质的数列,或函数,可以二分搜索答案,在O(log n)时间内求解
//如f(x) = 1 (x <= y) = 0 (x > y), 可以二分搜索出分界值y
//注意: l%2 == 0, r = l + 1时, (l + r)/2 == l 此处易出现死循环
int binary_search(int l, int r)
{
    int mid;
    int ans = l;
    while(l <= r)
    {
        mid = (l + r) >> 1;
        if(f(mid))
        {
            r = mid - 1;//视情况定
            ans = mid;
        }
        else
            l = mid + 1;
    }
    return ans;
}
///三分搜索
//对于满足抛物线性质的数列或函数,可以三分答案,在O(log n)时间内求解
//方便于求(抛物线)的最值
//注意: l % 3 == 0, r = l + 1 | l + 2时, (l + l + r) / 3 == l 容易出现死循环
int three_search(int l, int r)
{
    int ll, rr;
    while(l + 2 < r)
    {
        ll = (l + l + r) / 3;
        rr = (l + r + r) / 3;
        if(f(ll) < f(rr))
            r = rr;
        else
            l = ll;
    }
    return min(f(l), f(r), f(l + 1));
}
```

10 分治

```
///分治
//对于某些统计类问题, 可以将问题分为两半, 然后统计跨过两区间的符合条件的数目即可
//应用1: 二维偏序求LIS
```

11 Java

```
import java.io.*;
import java.util.*;
import java.math.*;
import java.BigInteger;
import java.text.DecimalFormat
public class Main{
    public static void main(String arg[]) throws Exception{
        Scanner cin = new Scnner(System.in);

        BigInteger a, b;
        a = new BigInteger(”123”);
        a = cin.nextBigInteger();
        a.add(b);//a + b
        a.subtract(b);// a − b
        a.multiply(b);// a * b
        a.divide(b);// a / b
        a.negate();// −a
        a.remainder(b);//a%b
        a.abs();//|a|
        a.pow(b);//a^b
        //.... and other math fuction, like log();
        a.toString();
        a.compareTo(b);//
        //四舍五入
        double c = 2.3659874;
        //小数格式化，引号中的0.000表示保留小数点后三位（第四位四舍五入）
        DecimalFormat df = new DecimalFormat(”0.000”);
        String num = df.format(a);
        System.out.println(num);

    }
}
```