

Algorithm Template Library

ytz123

October 14, 2019

Contents

1 注意事项	4
1.1 需要想到的方法	4
1.2 需要注意的问题	6
2 图论	7
2.1 线段树维护树直径	7
2.2 有向图判断两个点能否到达	9
2.3 dsu	11
2.4 长链剖分	12
2.5 树链剖分	14
2.6 DAG 删去无用边	14
2.7 树分治	14
2.8 支配树	15
2.8.1 DAG 支配树 (含倍增 LCA)	15
2.8.2 一般有向图支配树	16
2.9 一般图最大匹配	18
2.9.1 随机做法	18
2.9.2 正经带花树做法	19
2.10 动态维护图的连通性	20
2.10.1 cdq(含可撤销按秩合并的并查集)	20
2.10.2 LCT	21
2.11 网络流	23
2.11.1 hungary	23
2.11.2 dinic 最大流	23
2.11.3 spfa 费用流	24
3 数据结构	26
3.1 笛卡尔树	26
3.2 莫队	26
3.2.1 带修改莫队	26
3.3 平衡树	28
3.3.1 splay	28
3.3.2 treap	29
3.4 主席树	30
3.4.1 主席树	30
3.4.2 树状数组套主席树	31
3.5 线段树	32
3.5.1 zkw 线段树	32
3.5.2 李超线段树	32
3.5.3 吉老师线段树	34
3.5.4 线段树维护凸包	36
3.5.5 线段树的分裂和合并	37
3.5.6 线段树维护单调序列长度	38
3.5.7 区间除区间加	38
3.6 KDTree	39
3.6.1 3 维 KDtree	39
3.6.2 KDtree 二维空间区间覆盖单点查询	40
3.6.3 KDtree 二维空间单点修改区间查询	41
3.6.4 KDtree 找最近点	43

4	构造	46
4.1	若干排列使所有数对都出现一次	46
4.2	rec-free	46
4.3	点边均整数多边形	47
5	数学	48
5.1	线性基	48
5.1.1	线性基总结	48
5.1.2	线性基模板	48
5.1.3	实数线性基	49
5.2	BSGS	50
5.3	勾股数分解	51
5.4	CRT	51
5.5	蔡勒公式	52
5.6	拓展欧拉定理	52
5.7	素数判定 + 大整数质因数分解	52
5.8	删除 k 个数使 gcd 最大	54
6	字符串	56
6.1	KMP	56
7	其他	57
7.1	cdq 套路	57
7.2	最小表示	57
7.3	十进制快速幂	57
7.4	数字哈希	58
7.5	海岛分金币	58
7.5.1	海岛分金币 1	58
7.5.2	海岛分金币 2	59
7.6	根号枚举	61
7.7	读入输出外挂	61
7.8	给定小数化成分数	62
8	计算几何	63
8.1	全家桶	63
8.2	单位圆覆盖	66

1 注意事项

1.1 需要想到的方法

数据结构:

- 能用 `splay` 吗? 能用线段树代替吗? 能用树状数组代替吗?
能用 `treap` 吗? 能用 `set` 代替吗? 能用权值线段树代替吗?
- 势能线段树 (先确定势能, 然后注意收敛情况不能有遗漏):
区间开根有次数限制, 维护最大值, 直接递归到叶节点
区间取模, 如果 $x > p$, 那么 x 减少至少一半, 维护最大值, 递归到叶节点
区间除, 单点加, 跟上面一样脑瘫问题
区间除, 区间加, 会导致 $\max\text{-}\min$ 减小
- 主席树维护区间覆盖? 如果覆盖值的选择不多那么是可以维护的
一开始建若干棵叶节点权值相同的树, 区间覆盖就把区间指到这里来就可以了
- 主席树上树? 子树问题就在 `dfs` 序上做, 路径问题就 `u` 从 `fa[u]` 修改过来,
→ $u+v-\text{lca}(u,v)-\text{fa}[\text{lca}(u,v)]$ 即可
带修改就套树状数组嘛
路径问题有时候也可用欧拉序来做, 欧拉序可以通过在 `u` 第一次出现的位置 `+1`, 第二次 `-1`
区间 `[st[u], st[v]]` 的和就可以定位到一条路径

分块方法的应用:

- 图的所有点, 按照度数与 \sqrt{m} 的关系分为大点小点
- 区间加单点查, 可以做 o1 修改根号查询, 也能根号修改 o1 查询

分块的实现:

实现上一般不需要 `block` 结构体, 分块的元素其实还是在同一个数组里的

数论:

欧拉定理:

若 a, n 互质, 则 $a^{\phi(n)} \equiv 1 \pmod{n}$, 费马小定理是 n 为质数的特例

所以 $a^i \pmod{n}$ 的循环节长度也就是 $\phi(n)$, 在 a, n 互质的前提下

拓展欧拉定理:

$a^{x \pmod{\phi(m)} + \phi(m)} \equiv a^x \pmod{m}$, a 为任意整数, x, m 为任意正整数, 并且 $x \geq \phi(m)$

威尔逊定理:

$(n-2)! \equiv n \pmod{n}$ n 为素数为 1, 否则 n 为 4 是答案为 2, 其他为 0

鸽巢原理:

$x^k \pmod{p}$ 的循环节不会超过 p

线性筛欧拉函数:

$\phi(i \cdot p) = \phi(i) \cdot \phi(p)$, p 为质数

绝对值相加:

函数 $y = \sum(|x - a_i|)$ 一定是个凸的函数, 可以三分

因子个数:

11 以内因子个数最多 $20w$ 个

计数问题:

- 计算多种情况的答案之和, 转变为枚举某个量, 计算这个量对多少种情况做贡献
- 给定数列求多少个区间的 \max/\min 满足某种限制 (与区间长度挂钩), 先考虑枚举 \max/\min , 再考虑
→ 枚举左/右端点
如果再限制区间不能有相同的数, 可以来回扫描得到以某个点为左/右端点, 右/左端点最远延伸到哪里
这样可以继续使用上述做法, 扫描时扫描短的一边, 加一个 `lower_bound`, 总复杂度就是 $O(n \log^2 n)$
- 数子树不同颜色数问题:
不带修改的离线: `dsu`, $O(n \log n)$
支持修改单点颜色的在线: 对于同一种颜色, 按照 `dfs` 序排序

然后在每个点位置 +1, 对相邻两个点 uv , 在 $lca(u,v)$ 的位置标记-1

求子树就直接 dfs 序求区间和即可

限制子树深度: 上面做法的线段树换成主席树即可, rt 按深度排就行了

随机:

随机排列的 LIS 长度是 \sqrt{n}

对于验证比较耗时, 但是理论上满足条件的比较少

为了防止被构造数据卡掉, 可以随机一下 (前提是顺序不影响意义), 另外还要注意能去重就去重

判定问题:

随机赋值: 某些判定可以考虑随机赋值 val, val 的运算看情况可以考虑异或 or 乘法

判定两个数相等的一种思路: 取若干个质数, 模数都相等就默认两个数相同, 要保证质数乘积 $> MAX_X$

快速幂:

快速幂不只有 2 进制, 一般 2 进制最快

使用 k 进制快速幂实际复杂度是 $k \log(k, n)$

当指数是个很长的数字时, 根据给定数字的进制选择快速幂的进制才是缀吼的

经典问题:

最大和子矩阵, 如果 0 很多导致非 0 只有 $O(n)$ 级别, 可以转成最大子段和 $O(n^2 \log n)$ 来做

图论:

1. 对于边权为 1 的图, 保证起始点 (一开始放入 queue 的点) dis 最小 +SLF 优化 $= O(n)$ 复杂度

2. 长链剖分解决的问题: 满足某条件的树上路径的存在性判定/计数

3. 无向图若干询问, 每次给两个点 uv , 求出满足某条件的路径的 xx 值:

(1) 生成树能不能做

4. 判断普通无向图是不是二分图: 有没有奇环

5. 克鲁斯卡尔重构树, 边化点的思想, 优雅性质: 祖先权值不小于子集的权值

6. 树上某个点集的生成树的边数: 把点按 dfs 序排序, 计算 (相邻两点间距离之和 + 首尾两点距离) / 2

↪ 即可

树:

1. 选择 a 的子树里距离 a 不超过 k 的所有点, $dep + dfn$ 转到二维平面

2. 选择 a 的子树里距离 a 满足特殊限制的一些点 \rightarrow bfs 序 + $L_{most} + R_{most}$, 后者可使用树链剖分维护

3. LCT 的权在边上 \rightarrow 如果可以就把边化点即可

并查集:

1. 维护任意两点距离的奇偶性:

令 $d[x]$ 为 x 到所在根的路径长度的奇偶, 新加一条边 (x, y) , 且 xy 不在一个集合

把 x 的根 fx 合并到 y 的根 fy 下, 令 $new_d[fx] = d[x] \oplus d[y] \oplus 1$ 即可

2. 按秩合并并查集的优点: 可以借栈来撤销操作

字符串:

这能不能哈希

几何:

1. 几何体的欧拉公式: 点数 V - 边数 E + 面数 (区域数) $F = 2$

使用前提:

边不能相交于非顶点的位置, 如果相交, 那么相交点也要算进 V 里

V 不能包含孤立点, 显然如果允许包含孤立点, 我们可以使 V 增加, EF 不变, 等式不再成立

2. 旋转 45 度就 (x, y) 变成 $(x+y, x-y)$, 直角菱形就变矩形了

1.2 需要注意的问题

通用：

错误率较高的简单题，考虑各种可能的特殊情况，以及对特殊情况是否会引起一般情况的错误

检查代码觉得没有问题时，再多读两遍题，是否有漏掉的东西或理解有问题

读入是否有强制在线的加密，确认一下这个加密有没有强制在线

端点是否有特殊情况

数组要开够

不要交错题

变量打错

语言特性：

vector 的 `unique` 一定要跟上 `resize`，即 `v.resize(unique(v.begin(),v.end())-v.begin())`

小心 `memcpy`，最好写成 `sizeof(type)*length` 的写法，毕竟指针的 `sizeof` 并不可信

`lambda` 表达式里的数据类型要和实际数据类型保持一致，比如不要 `ll` 写成 `int`

手动 `or` 用 `unique` 对 `vector` 重排列后请立即 `resize`，否则容易引起错误

输出浮点数，需要判断如果 `fabs` 小于 `eps` 则置 0，避免输出 `-0.00`

浮点型请不要 `cout`，小心输出科学计数法

分块的块大小：

具体问题具体分析！分析修改和查询的次数以及复杂度，再决定块大小！

多组数据的初始化：

网络流 `len = 1`

主席树 `tot = 0`

数据结构的选择：

`splay` 如果没有翻转操作，基本都可以用线段树代替

树链剖分 + 线段树，请注意树上点和线段树上点的映射，`pos` 和 `dfn`

矩阵乘法满足分配律和结合律，但不满足交换律

用 `map` 来存无向边，一定要先处理令 `u < v`，否则存了边 `pair(u,v)` 查 `pair(v,u)` 会查不到

单调队列是 `while ! while !!!`

2 图论

2.1 线段树维护树直径

```

/* 无法处理修改边权的情况,LCA 用 ST 表, 总复杂度  $O(n\log)$  */
/* 边带权的树每次询问删去两条边后, 剩下 3 棵树的直径长度 */
int T, n, m;
int len, head[N], ST[20][N];
struct edge{int u, v, w;}ee[N];
int cnt, fa[N], log_2[N], st[N], en[N], dfn[N], dis[N], dep[N], pos[N];
struct edges{int to, next, cost;}e[N];
void add(int u, int v, int w) {
    e[++ len] = (edges){v, head[u], w}, head[u] = len;
    e[++ len] = (edges){u, head[v], w}, head[v] = len;
}
void dfs1(int u) {
    st[u] = ++ cnt, dfn[cnt] = u;
    for (int v, i = head[u]; i; i = e[i].next) {
        v = e[i].to;
        if (v == fa[u]) continue;
        fa[v] = u, dep[v] = dep[u] + 1;
        dis[v] = dis[u] + e[i].cost, dfs1(v);
    }
    en[u] = cnt;
}
void dfs2(int u) {
    dfn[++ cnt] = u, pos[u] = cnt;
    for (int v, i = head[u]; i; i = e[i].next) {
        v = e[i].to;
        if (v == fa[u]) continue;
        dfs2(v), dfn[++ cnt] = u;
    }
}
int mmin(int x, int y) {
    if (dep[x] < dep[y]) return x;
    return y;
}
int lca(int u, int v) {
    static int w;
    if (pos[u] > pos[v]) swap(u, v);
    w = log_2[pos[v] - pos[u] + 1];
    return mmin(ST[w][pos[u]], ST[w][pos[v] - (1 << w) + 1]);
}
int dist(int u, int v) {
    int Lca = lca(u, v);
    return dis[u] + dis[v] - dis[Lca] * 2;
}
void build() {
    for (int i = 1; i <= cnt; i++)
        ST[0][i] = dfn[i];
    for (int i = 1; i < 20; i++)
        for (int j = 1; j <= cnt; j++)
            if (j + (1 << (i - 1)) > cnt) ST[i][j] = ST[i - 1][j];
            else ST[i][j] = mmin(ST[i - 1][j], ST[i - 1][j + (1 << (i - 1))]);
}

```

```

}
int M;
struct node {
    int l, r, dis;
}tr[N << 1];
void update(int o, int o1, int o2) {
    static int d; static node tmp;
    if (tr[o1].dis == -1) {tr[o] = tr[o2]; return;}
    if (tr[o2].dis == -1) {tr[o] = tr[o1]; return;}
    if (tr[o1].dis > tr[o2].dis) tmp = tr[o1];
    else tmp = tr[o2];
    d = dist(tr[o1].l, tr[o2].l);
    if (d > tmp.dis) tmp.l = tr[o1].l, tmp.r = tr[o2].l, tmp.dis = d;
    d = dist(tr[o1].l, tr[o2].r);
    if (d > tmp.dis) tmp.l = tr[o1].l, tmp.r = tr[o2].r, tmp.dis = d;
    d = dist(tr[o1].r, tr[o2].l);
    if (d > tmp.dis) tmp.l = tr[o1].r, tmp.r = tr[o2].l, tmp.dis = d;
    d = dist(tr[o1].r, tr[o2].r);
    if (d > tmp.dis) tmp.l = tr[o1].r, tmp.r = tr[o2].r, tmp.dis = d;
    tr[o] = tmp;
}
void ask(int s, int t) {
    if (s > t) return;
    for (s += M - 1, t += M + 1; s ^ t ^ 1; s >>= 1, t >>= 1) {
        if (~s&1) update(0, 0, s ^ 1);
        if (t&1) update(0, 0, t ^ 1);
    }
}
int main() {
    int u, v, w, ans; log_2[1] = 0;
    for (int i = 2; i <= 200000; i++)
        if (i == 1 << (log_2[i - 1] + 1))
            log_2[i] = log_2[i - 1] + 1;
        else log_2[i] = log_2[i - 1];
    for (cin >> T; T--; ) {
        cin >> n >> m, cnt = len = 0;
        for (int i = 1; i <= n; i++)
            head[i] = 0;
        for (int i = 1; i < n; i++) {
            cin >> ee[i].u >> ee[i].v >> ee[i].w;
            add(ee[i].u, ee[i].v, ee[i].w);
        }
        dfs1(1);
        for (M = 1; M < n + 2; M <<= 1);
        for (int i = 1; i <= n; i++)
            tr[i + M].l = tr[i + M].r = dfn[i], tr[i + M].dis = 0;
        for (int i = n + M + 1; i <= (M << 1) + 1; i++)
            tr[i].dis = -1;
        cnt = 0, dfs2(1), build();
        for (int i = M; i; i--)
            update(i, i << 1, i << 1 | 1);
        for (int i = 1; i < n; i++)
            if (dep[ee[i].u] > dep[ee[i].v])
                swap(ee[i].u, ee[i].v);
    }
}

```



```

for (int u, v, i = 1; i <= m; i++) {
    cin >> u >> v, ans = 0;
    u = ee[u].v, v = ee[v].v, w = lca(u, v);
    if (w == u || w == v) {
        if (w != u) swap(u, v);
        tr[0].dis = -1, ask(1, st[u] - 1), ask(en[u] + 1, n), ans = max(ans,
            ↪ tr[0].dis);
        tr[0].dis = -1, ask(st[u], st[v] - 1), ask(en[v] + 1, en[u]), ans =
            ↪ max(ans, tr[0].dis);
        tr[0].dis = -1, ask(st[v], en[v]), ans = max(ans, tr[0].dis);
    }
    else {
        if (st[u] > st[v]) swap(u, v);
        tr[0].dis = -1, ask(1, st[u] - 1), ask(en[u] + 1, st[v] - 1), ask(en[v] +
            ↪ 1, n), ans = max(ans, tr[0].dis);
        tr[0].dis = -1, ask(st[u], en[u]), ans = max(ans, tr[0].dis);
        tr[0].dis = -1, ask(st[v], en[v]), ans = max(ans, tr[0].dis);
    }
    printf("%d\n", ans);
}
return 0;
}

```

2.2 有向图判断两个点能否到达

/* 原题: n 个点的有向图, k 对点 (u, v) 满足 u 可达 v
 * p 对点 (u, v) 满足 u 不可达 v , 问这个图是否存在
 * 解法: 按照 p 对可达点直接连边构图, 然后考虑验证
 * $a[i][j]=0/1$ 表示 i 是否可达 j , 我们对 j 分块, *bitset* 加速
 * 时间 $O(n*m/32)$ 空间 $O(n*n/blk)$ blk 随便取
 */

```

const int N = 1e5 + 2;
const int BLK = 5000;
int n, k, p;
int d[N];
vector<int> e[N], f[N], ck[N];
int top, sta[N], in[N];
int cnt, dfn[N], low[N], vis[N];
int sum, bel[N];
bitset<BLK> a[N];
queue<int> q;
int topo[N];
void tarjan(int u) {
    vis[u] = in[u] = 1;
    sta[++top] = u, dfn[u] = low[u] = ++cnt;
    for (int v : e[u])
        if (!vis[v]) {
            tarjan(v);
            low[u] = min(low[v], low[u]);
        }
    else if (in[v])
        low[u] = min(low[v], low[u]);
    if (low[u] == dfn[u]) {

```

```

        sum ++; int i;
        while (1) {
            i = sta[top --];
            in[i] = 0, bel[i] = sum;
            if (i == u) break;
        }
    }
}

int main() {
    cin >> n >> k;
    for (int u, v, i = 1; i <= k; i++) {
        cin >> u >> v;
        e[u].push_back(v);
    }
    cin >> p;
    for (int u, v, i = 1; i <= p; i++) {
        cin >> u >> v;
        f[u].push_back(v);
    }
    for (int i = 1; i <= n; i++)
        if (!vis[i])
            tarjan(i);
    for (int i = 1; i <= n; i++) {
        for (int j : f[i]) {
            if (bel[i] == bel[j]) return puts("NO"), 0;
            ck[bel[i]].push_back(bel[j]); //check
        }
        f[i].clear();
    }
    for (int i = 1; i <= n; i++)
        for (int j : e[i]) {
            if (bel[i] == bel[j]) continue;
            f[bel[i]].push_back(bel[j]);
            d[bel[j]] ++;
        }
    cnt = 0;
    for (int i = 1; i <= sum; i++)
        if (!d[i])
            q.push(i);
    while (!q.empty()) {
        int now = q.front(); q.pop();
        topo[++ cnt] = now;
        for (int j : f[now]) {
            d[j] --;
            if (d[j] == 0) q.push(j);
        }
    }
    for (int i = 1, t = (sum + BLK - 1) / BLK; i <= t; i++) {
        for (int j = sum; j; j--) {
            int u = topo[j];
            a[u].reset();
            if (BLK * (i - 1) < u && u <= BLK * i)
                a[u][u - BLK * (i - 1) - 1] = 1;
            for (int v : f[u])

```

```

        a[u] |= a[v];
    }
    for (int j = 1; j <= sum; j++)
        for (int v : ck[j])
            if (BLK * (i - 1) < v && v <= BLK * i &&
                a[j][v - BLK * (i - 1) - 1] == 1) {
                puts("NO");
                return 0;
            }
    }
    printf("YES\n%d\n", k);
    for (int i = 1; i <= n; i++)
        for (int j : e[i])
            printf("%d %d\n", i, j);
    return 0;
}

```

2.3 dsu

/*DSU

- * 用途: $O(n \log n)$ 解决无修改的子树询问问题, 需要保证操作支持删除
- * 比如子树中不同颜色数, 出现最多的颜色等
- * 解决方法: 对于每个节点, 先对所有轻儿子, *dfs* 下去求一遍, 再消除影响
- * 然后再 *dfs* 自己的重儿子, 然后不消除影响, 再加上所有轻儿子
- * 就得到当前节点为根的子树的答案了

*/

```

int n, c[N];
int cnt[N], maxCnt;
int siz[N], son[N];
vector<int> e[N];
ll ans[N], sum[N];
void dfs1(int u, int fr) {
    siz[u] = 1;
    for (int v : e[u]) {
        if (v == fr) continue;
        dfs1(v, u);
        siz[u] += siz[v];
        if (siz[v] > siz[son[u]]) son[u] = v;
    }
}
void update(int x, int y) {
    sum[cnt[x]] -= x;
    cnt[x] += y;
    sum[cnt[x]] += x;
    if (cnt[x] > maxCnt) maxCnt = cnt[x];
    if (sum[maxCnt] == 0) maxCnt--;
}
void dfs3(int u, int fr, int val) {
    update(c[u], val);
    for (int v : e[u]) {
        if (v == fr) continue;
        dfs3(v, u, val);
    }
}
}

```

```

void dfs2(int u, int fr) {
    for (int v : e[u]) {
        if (v == fr || v == son[u]) continue;
        dfs2(v, u), dfs3(v, u, -1);
    }
    if (son[u]) dfs2(son[u], u);
    for (int v : e[u]) {
        if (v == fr || v == son[u]) continue;
        dfs3(v, u, 1);
    }
    update(c[u], 1);
    ans[u] = sum[maxCnt];
}

int main() {
    ios::sync_with_stdio(false);
    cin >> n;
    for (int i = 1; i <= n; i++)
        cin >> c[i];
    for (int u, v, i = 1; i < n; i++) {
        cin >> u >> v;
        e[u].push_back(v);
        e[v].push_back(u);
    }
    dfs1(1, 1), dfs2(1, 1);
    for (int i = 1; i <= n; i++)
        cout << ans[i] << ' ';
    return 0;
}

```

2.4 长链剖分

/* 长链剖分，选择深度最大的儿子作为重儿子，用于合并以深度为下标的信息
 * 像 *dsu* 一样，直接继承重儿子信息，然后按深度暴力合并其他儿子信息
 * 时间复杂度考虑每个节点作为轻儿子中的节点被合并只会有一次，所以 $O(n)$
 * 另一种用法，可以 $O(n \log n)$ 预处理后， $O(1)$ 找到 k 级祖先
 * *example problem*: 给个树，第 i 个点有两个权值 a_i 和 b_i
 * 现在求一条长度为 m 的路径，使得 $\sum a_i / \sum b_i$ 最小
 * 防爆栈 *trick*: 像重链剖分一样改成 *bfs*
 */

```

int n, m;
double k, a[N], b[N], val[N];
int len[N], son[N];
vector<int> e[N];
double tmp[N], *ptr, *f[N], temp[N];
void dfs(int u, int fr) {
    for (int v : e[u]) {
        if (v == fr) continue;
        dfs(v, u);
        if (len[v] > len[son[u]]) son[u] = v;
    }
    len[u] = len[son[u]] + 1;
}

inline double F(int x, int y) {return y >= len[x] ? 0 : f[x][y];}
bool solve(int u, int fr) {

```

```

/* 为实现  $O(1)$  继承, 采用  $f[u]-f[v]$  来保存  $u \rightarrow v$  路径上的最小权值和 ( $dep[v]>dep[u]$ )
* 即自底向上累加
*/
if (son[u]) {
    f[son[u]] = f[u] + 1;
    if (solve(son[u], u)) return 1;
    f[u][0] = val[u] + f[u][1];
    if (len[u] >= m && f[u][0] - F(u, m) <= 0) return 1;
    for (int v : e[u]) {
        if (v == son[u] || v == fr) continue;
        f[v] = ptr, ptr += len[v];
        if (solve(v, u)) return 1;
        for (int j = 1; j <= len[v] && j <= m; j++) {
            if (len[u] + j < m) continue;
            if (f[v][0] - F(v, j) + f[u][0] - F(u, m - j) <= 0) return 1;
        }
        temp[0] = val[u];
        for (int j = 1; j <= len[v]; j++)
            temp[j] = val[u] + min(f[u][1] - F(u, j + 1), f[v][0] - F(v, j));
        if (len[v] + 1 == len[u]) f[u][0] = temp[len[v]];
        for (int j = 1; j <= len[v]; j++)
            f[u][j] = f[u][0] - temp[j - 1];
        if (len[v] + 1 != len[u]) f[u][len[v] + 1] = f[u][0] - temp[len[v]];
    }
}
else {
    f[u][0] = val[u];
    if (m == 1 && f[u][0] <= 0) return 1;
}
return 0;
}

bool judge(double mid) {
    f[1] = ptr = tmp, ptr += len[1], k = mid;
    for (int i = 1; i <= n; i++) val[i] = a[i] - b[i] * k;
    return solve(1, 1);
}

int main() {
    cin >> n >> m;
    for (int i = 1; i <= n; i++) cin >> a[i];
    for (int i = 1; i <= n; i++) cin >> b[i];
    if (m == -1) {
        double ans = 1e9;
        for (int i = 1; i <= n; i++)
            ans = min(ans, a[i] / b[i]);
        printf("%.2f\n", ans);
        return 0;
    }
    for (int u, v, i = 1; i < n; i++) {
        cin >> u >> v;
        e[u].push_back(v);
        e[v].push_back(u);
    }
    dfs(1, 1);
    int flag = 0;

```

```

double l = 0, r = 2e5, mid, ans;
for (int i = 0; i < 50; i++) {
    mid = (l + r) / 2;
    if (judge(mid)) r = mid - eps, flag = 1, ans = mid;
    else l = mid + eps;
}
if (flag) printf("%.2f\n", ans);
else puts("-1");
return 0;
}

```

2.5 树链剖分

```

int query(int u, int v) { // 查询链上点权和
    int res = 0;
    for (int fu = top[u], fv = top[v]; fu != fv; fu = top[u = fa[fu]]) {
        if (dep[fu] < dep[fv]) swap(fu, fv), swap(u, v);
        res += ask(pos[fu], pos[u]);
    }
    if (dep[u] > dep[v]) swap(u, v);
    return res + ask(pos[u], pos[v]);
}

```

2.6 DAG 删去无用边

/* 无用边定义：对于边 (u, v) 如果存在从 u 到 v 不经过该边的另一条路径，则称该边无用
 * 时间复杂度： $O(n^3)$ */

```

bool f[N][N]; // i 是否可达 j
vector<int> e[N];
int main() {
    rep(i, 1, n)
        for (int j : e[i]) {
            rep(k, 1, n)
                if (i != k && j != k && f[i][k] && f[k][j])
                    no_use_edge;
        }
}

```

2.7 树分治

```

vector<int> e[N];
int n, a[N];
int root, _left, vis[N];
int siz[N], maxv[N];
void find_root(int u, int fr) {
    siz[u] = 1, maxv[u] = 0;
    for (int v : e[u]) {
        if (v == fr || vis[v]) continue;
        find_root(v, u), siz[u] += siz[v];
        maxv[u] = max(maxv[u], siz[v]);
    }
    maxv[u] = max(maxv[u], _left - siz[u]);
    if (!root || maxv[u] < maxv[root]) root = u;
}

```

```

void dfs(int u, int fr) {
    siz[u] = 1;
    for (int v : e[u]) {
        if (v == fr || vis[v]) continue;
        dfs(v, u);
        siz[u] += siz[v];
    }
}

void solve(int u, int w) {
    dfs(u, u), vis[u] = 1;
    for (int v : e[u]) {
        if (vis[v]) continue;
        _left = siz[v];
        root = 0;
        find_root(v, v);
        solve(root, w + 1);
    }
}

int main() {
    _left = n, root = 0, find_root(1, 1);
    solve(root, 0);
}

```

2.8 支配树

2.8.1 DAG 支配树 (含倍增 LCA)

/* 支配树只能维护点的支配关系, 想维护边的话需要把边化成点
 *
 * 对于一个 dag, 假设每个联通块中都只有一个点出度为 0
 * 那么对于一个联通块, 假设这个出度为 0 的点为 s
 * 可以处理出这个块中所有点到 s 的必经点
 * rt 为新建的虚根, 把所有联通块整合在一棵树里
 */

```

int fa[N][19];
vector<int> e[N], f[N], E[N];
int du[N], dep[N];
int lca(int x, int y) {
    if (dep[x] < dep[y]) swap(x, y); // dep[x] > dep[y]
    for (int j = 18; j >= 0; j--)
        if (dep[fa[x][j]] >= dep[y])
            x = fa[x][j];
    if (x == y) return x;
    for (int j = 18; j >= 0; j--)
        if (fa[x][j] != fa[y][j])
            x = fa[x][j], y = fa[y][j];
    return fa[x][0];
}

void topo() {
    static int q[N * 2], l, r; l = 1, r = 0;
    for (int i = 1; i <= n; i++)
        if (du[i] == 0) {
            q[++r] = i;
            E[rt].push_back(i); // 支配树的边
        }
}

```

```

        fa[i][0] = rt;
        dep[i] = 1;
    }
    while (l <= r) {
        int u = q[l ++];
        for (int v : e[u]) {
            du[v] --;
            if (du[v] == 0) {
                int las = -1;
                for (int w : f[v]) {
                    if (las == -1) las = w;
                    else las = lca(las, w);
                }
                E[las].push_back(v);
                fa[v][0] = las;
                dep[v] = dep[las] + 1;
                for (int j = 1; j <= 18; j++)
                    fa[v][j] = fa[fa[v][j - 1]][j - 1];
                q[++ r] = v;
            }
        }
    }
}

int main() {
    int cas, q;
    for (scanf("%d", &cas); cas --; ) {
        scanf("%d %d", &n, &m);
        rt = n + 1;
        for (int i = 0; i <= rt; i++) {
            e[i].clear(); f[i].clear();
            E[i].clear(); du[i] = dep[i] = 0;
            for (int j = 0; j <= 18; j++)
                fa[i][j] = 0;
        }
        for (int u, v, i = 1; i <= m; i++) {
            scanf("%d %d", &u, &v);
            e[v].push_back(u); //反向边
            f[u].push_back(v); //正向边
            du[u] ++;
        }
        topo();
    }
    return 0;
}

```

2.8.2 一般有向图支配树

```

namespace DomTree {
    int n, m, tot;
    int dfn[N], id[N], fa[N]; //fa 为 dfs 树上的父亲
    vector<int> d[N], e[N], f[N], E[N];
    //d 为临时树的边, e 原图边, f 为 dfs 树的反向边, E 为支配树的边
    int fu[N], val[N], semi[N], idom[N]; //fu 为并查集数组
    void dfs(int u) {

```



```

    dfn[u] = ++ tot; id[tot] = u;
    for (int v : e[u]) {
        f[v].push_back(u);
        if (!dfn[v]) {
            fa[v] = u;
            dfs(v);
        }
    }
}

int getFa(int x) {
    if (fu[x] == x) return x;
    int y = getFa(fu[x]);
    if (dfn[semi[val[fu[x]]]] < dfn[semi[val[x]]])
        val[x] = val[fu[x]];
    return fu[x] = y;
}

int smin(int x, int y) {
    return dfn[x] < dfn[y] ? x : y;
}

void solve(int st) { //st 源点
    dfs(st);
    for (int i = tot; i >= 2; i --) {
        int x = id[i];
        if (!f[x].empty()) {
            for (int y : f[x])
                if (dfn[y] < dfn[x])
                    semi[x] = smin(semi[x], y);
            else {
                getFa(y);
                semi[x] = smin(semi[x], semi[val[y]]);
            }
        }
        fu[x] = fa[x]; d[semi[x]].pb(x);
        if (!d[fa[x]].empty()) {
            for (int y : d[fa[x]]) {
                getFa(y);
                int u = val[y];
                idom[y] = (dfn[semi[u]] < dfn[semi[y]]) ? u : fa[x];
            }
        }
    }

    for (int i = 2; i <= tot; i ++ ) {
        int x = id[i];
        if (idom[x] != semi[x]) idom[x] = idom[idom[x]];
    }

    for (int i = 2; i <= tot; i ++ )
        E[idom[id[i]]].push_back(id[i]);
}

void init() {
    in(n), in(m);
    for (int i = 1; i <= n; i ++ )
        d[i].clear(), e[i].clear(), f[i].clear(), E[i].clear();
    for (int u, v, i = 0; i < m; i ++ ) {
        in(u), in(v);

```

```

        e[u].push_back(v); //单向边
    }
    tot = 0;
    for (int i = 1; i <= n; i++) {
        fu[i] = semi[i] = idom[i] = val[i] = i;
        dfn[i] = id[i] = 0;
    }
}
}
int main() {
    ios::sync_with_stdio(false);
    DomTree::init();
    /* 根据题目或者根据图的入度判断源点 st*/
    DomTree::solve(st);
    return 0;
}

```

2.9 一般图最大匹配

2.9.1 随机做法

```

/* 一般图最大匹配，时间复杂度  $O(n^3)$ 
 * 例题： $n$  个人， $m$  个配对关系
 * 输出最大配对数，然后对于每个人输出配对的人
 * 以下是  $O(n^3)$  随机匹配做法，可能会被 hack
 */
const int N = 505, LIM = 3;
int n, m, tim;
int cnt, ans[N], bel[N], vis[N];
vector<int> e[N];
bool dfs(int u) {
    vis[u] = tim; random_shuffle(e[u].begin(), e[u].end());
    for (int v : e[u]) {
        int w = bel[v]; if (vis[w] == tim) continue;
        bel[u] = v, bel[v] = u, bel[w] = 0; if (!w || dfs(w)) return 1;
        bel[u] = 0, bel[v] = w, bel[w] = v;
    }
    return 0;
}
int main() {
    scanf("%d%d", &n, &m);
    for (int u, v; m--; ) {
        scanf("%d %d", &u, &v);
        e[u].push_back(v);
        e[v].push_back(u);
    }
    for (int t = 1; t <= LIM; t++) {
        for (int i = 1; i <= n; i++) {
            if (!bel[i]) tim++, dfs(i);
            int tmp = 0;
            for (int i = 1; i <= n; i++)
                tmp += bel[i] != 0;
            if (tmp > cnt) {
                cnt = tmp;
            }
        }
    }
}

```

```

        for (int i = 1; i <= n; i++)
            ans[i] = bel[i];
    }
}
printf("%d\n", cnt / 2);
for (int i = 1; i <= n; i++)
    printf("%d ", ans[i]);
}

```

2.9.2 正经带花树做法

```

/* 时间复杂度  $O(n^3)$  */
int n, m, ans;
vector<int> e[N];
int tim, pre[N], mate[N];
int l, r, q[N], f[N], vis[N], sta[N];
int fa(int x) {
    return f[x] == x ? x : f[x] = fa(f[x]);
}
int lca(int x, int y) {
    for (++tim, x = fa(x), y = fa(y); ; swap(x, y)) if (x) {
        if (vis[x] == tim) return x;
        vis[x] = tim, x = fa(pre[mate[x]]);
    }
}
int blossom(int x, int y, int g) {
    for (; fa(x) != g; y = mate[x], x = pre[y]) {
        pre[x] = y;
        if (sta[mate[x]] == 1) sta[q[++r] = mate[x]] = 0;
        if (fa(x) == x) f[x] = g;
        if (fa(mate[x]) == mate[x]) f[mate[x]] = g;
    }
}
int match(int s) {
    int y, las;
    memset(sta, -1, sizeof sta);
    memset(pre, 0, sizeof pre);
    for (int i = 1; i <= n; i++) f[i] = i;
    for (q[l = r = 0] = s, sta[s] = 0; l <= r; l++)
        for (int x : e[q[l]])
            if (sta[x] == -1) {
                sta[x] = 1, pre[x] = q[l];
                if (!mate[x]) {
                    for (int j = q[l], i = x; j; j = pre[i = las])
                        las = mate[j], mate[j] = i, mate[i] = j;
                    return 1;
                }
                sta[q[++r] = mate[x]] = 0;
            }
        else if (f[x] != f[q[l]] && sta[x] == 0)
            y = lca(x, q[l]), blossom(x, q[l], y), blossom(q[l], x, y);
    return 0;
}

```

```

int main() {
    scanf("%d %d", &n, &m);
    for (int u, v, i = 0; i < m; i++) {
        scanf("%d %d", &u, &v);
        e[u].push_back(v);
        e[v].push_back(u);
        if (!mate[u] && !mate[v])
            mate[u] = v, mate[v] = u, ans++;
    }
    for (int i = 1; i <= n; i++)
        if (!mate[i] && match(i))
            ans++;
    printf("%d\n", ans);
    for (int i = 1; i <= n; i++)
        printf("%d%c", mate[i], i == n ? '\n' : ' ');
}

```

2.10 动态维护图的连通性

2.10.1 cdq(含可撤销按秩合并的并查集)

/* 例题：给出 m 条边连接的两个点，以及这条边存在的时间区间
 * 求 1 和 n 两个点连通的总时长
 * 解法：cdq 维护图的连通性，对于当前处理的时间区间和边集
 * 将完全包含当前时间段的边并起来，如果连通则该时间区间始终连通
 * 否则将边集分开，分治区间即可。为保证时间复杂度与边集大小为线性关系
 * 我们使用支持撤销的按秩合并并查集，总复杂度 $O(n \log^2 n)$
 */

```

namespace UnionFindSet {
    int f[N], rk[N];
    int top, sta[N * 2]; // n 个点最多 n-1 个有效连接，所以 n*2 足够了
    int fa(int x) {
        while (x != f[x]) x = f[x];
        return x;
    }
    void union_(int x, int y) {
        x = fa(x), y = fa(y);
        if (x == y) return;
        if (rk[x] > rk[y]) swap(x, y);
        if (rk[x] == rk[y]) rk[y]++, sta[++top] = -y;
        f[x] = y, sta[++top] = x;
    }
    void undo(int last) {
        for (; top > last; top--) {
            if (sta[top] < 0) rk[-sta[top]]--;
            else f[sta[top]] = sta[top];
        }
    }
    void init() {
        for (int i = 1; i <= n; i++)
            f[i] = i, rk[i] = 0;
    }
}
using namespace UnionFindSet;

```

```

struct node{int u, v, l, r;}; //每条边连接 (u,v), 且在 [l,r] 时间区间内存活
void solve (int head, int tail, const vector <node> &e) {
    //当前处理的时间区间是 [head,tail], 边集是 e
    if (e.size() == 0) return;
    int last = top, mid = head + tail >> 1;
    vector<node> l, r;
    for (node i : e) {
        if (i.l <= head && tail <= i.r) union_(i.u, i.v);
        else {
            if (i.l <= mid) l.push_back(i);
            if (i.r > mid) r.push_back(i);
        }
    }
    if (fa(1) == fa(n)) {
        ans += a[tail] - a[head - 1]; //把时间区间做了离散化, 使用原来值求和
        undo(last); return;
    }
    if (head == tail) {undo(last);return;} //叶节点也要处理修改操作, 以及记得撤销操作
    solve(head, mid, l), solve(mid + 1, tail, r); undo(last);
}

```

2.10.2 LCT

/* 解决问题:LCT 维护图的连通性

* 实例: 维护每个时刻当前图是否为二分图

* 解决方法: 对于环, 去掉最早消失的边即可

* 对于当前存在的所有边, 用 $on=0/1$ 记录是否在树上

* 对于不在树上的边, $gg[i]$ 代表第 i 条边加到树上是否会形成奇环

* 然后 cnt 记录 $gg[i]=1$ 的边树即可, $cnt=0$ 当前图即为二分图

*/

```

namespace LCT {
    int fa[N], ch[N][2], rev[N], val[N], sum[N], minv[N];
    //val[i] 为每个点消失的时间, sum[i] 记录 i 为根的 splay 子树多少条边
    //minv[i] 记录 i 为根的子树里, 最早消失的是哪条边, 边在权上所以边变点
    int sta[N], top;
    bool isroot(int x) {
        return ch[fa[x]][0] != x && ch[fa[x]][1] != x;
    }
    void reverse(int x) {
        if (!x) return;
        swap(ch[x][0], ch[x][1]);
        rev[x] ^= 1;
    }
    void pushdown(int x) {
        if (!rev[x]) return;
        reverse(ch[x][0]);
        reverse(ch[x][1]);
        rev[x] = 0;
    }
    void pushup(int x) {
        minv[x] = x; sum[x] = (x > n) + sum[ch[x][0]] + sum[ch[x][1]];
        if (ch[x][0] && val[minv[ch[x][0]]] < val[minv[x]]) minv[x] = minv[ch[x][0]];
        if (ch[x][1] && val[minv[ch[x][1]]] < val[minv[x]]) minv[x] = minv[ch[x][1]];
    }
}

```

```

void rot(int x) {
    int y = fa[x], z = fa[y], d = ch[y][1] == x, c = ch[x][!d];
    fa[x] = z; if (!isroot(y)) ch[z][ch[z][1] == y] = x;
    ch[y][d] = c; if (c) fa[c] = y;
    fa[y] = x, ch[x][!d] = y;
    pushup(y), pushup(x);
}

void splay(int x) {
    int u = x, top = 0, y, z;
    while (!isroot(u)) sta[++ top] = u, u = fa[u];
    sta[++ top] = u;
    while (top) pushdown(sta[top --]);
    while (!isroot(x)) {
        y = fa[x], z = fa[y];
        if (!isroot(y)) {
            if ((ch[z][0] == y) ^ (ch[y][0] == x)) rot(x);
            else rot(y);
        }
        rot(x);
    }
}

void access(int x) { //把 x 到根的路径拎出来
    for (int y = 0; x != 0; y = x, x = fa[x]) {
        splay(x), ch[x][1] = y, pushup(x);
    }
}

void makeroot(int x) { //令 x 成为这棵树的根
    access(x), splay(x), reverse(x);
}

int findroot(int x) { //找根
    access(x), splay(x);
    while (ch[x][0]) pushdown(x), x = ch[x][0];
    splay(x); //把根转到顶保证复杂度
    return x;
}

void split(int x, int y) { //拉出 x-y 的路径
    makeroot(x);
    access(y), splay(y); //y 存了这条路径的信息
}

void link(int x, int y) {
    //printf("link %d %d\n", x, y);
    makeroot(x);
    if (findroot(y) != x) fa[x] = y;
}

void cut(int x, int y) {
    makeroot(x);
    if (findroot(y) == x && fa[y] == x && ch[x][1] == y) {
        fa[y] = ch[x][1] = 0;
        pushup(x);
    }
}

/*
usage:
    拉取 x-y 这条链的信息:         split(x, y), printf("%d\n", sum[y]);

```

```

        单点修改 (单点更新完 pushup): splay(x), val[x] = y, pushup(x);
    */
}

```

2.11 网络流

2.11.1 hungary

```

/* 二分图最大匹配, 时间复杂度  $O(nm)$ 
 * 例题: n1 个男, n2 个女, m 个配对关系
 * 输出最大配对数, 然后对于每个男输出配对的女
 */
int n1, n2, m;
vector<int> e[N];
int vis[N], pre[N], ans, tim;
bool dfs(int u) {
    if (vis[u] == tim) return 0;
    vis[u] = tim;
    for (int v : e[u])
        if (!pre[v] || dfs(pre[v]))
            return pre[v] = u, 1;
    return 0;
}
int main() {
    scanf("%d %d %d", &n1, &n2, &m);
    for (int u, v, i = 1; i <= m; i++) {
        scanf("%d %d", &u, &v);
        e[v].push_back(u);
        // 要输出左侧点连接的右侧点, 连边时就由右边点向左边连边
        // 连边 (u->v), 输出的 pre[v] 就是右边的点了
    }
    for (tim = 1; tim <= n2; tim++)
        if (dfs(tim)) ans++;
    printf("%d\n", ans);
    for (int i = 1; i <= n1; i++)
        printf("%d%c", pre[i], i == n1 ? '\n' : ' ');
    return 0;
}

```

2.11.2 dinic 最大流

```

/* 理论复杂度  $O(mn^2)$ , 但其实蛮松的
 * 二分图最大匹配的复杂度是  $O(m\sqrt{n})$ 
 * 如果边权都是 1, 那么是  $O(\min(n^{2/3}, m^{1/2}) * m)$ 
 */
const int inf = 0x3f3f3f3f;
int n, m;
int s, t, len = 1;
int to[M], cap[M], nex[M];
int g[N], p[N], q[N], d[N];
void add(int x, int y, int v) {
    to[++len] = y, cap[len] = v, nex[len] = g[x], g[x] = len;
    to[++len] = x, cap[len] = 0, nex[len] = g[y], g[y] = len;
}
bool bfs() {

```

```

    int l = 1, r = 1, x, i;
    memset (d, 0, sizeof d);
    d[s] = 1, q[1] = s;
    while (l <= r) {
        x = q[l ++];
        for (i = g[x]; i; i = nex[i])
            if (cap[i] && !d[to[i]])
                d[to[i]] = d[x] + 1, q[++ r] = to[i];
    }
    return d[t];
}

int dfs(int x, int y) {
    if (x == t || y == 0) return y;
    int flow = 0;
    for (int &i = p[x]; i; i = nex[i]) {
        if (!cap[i] || d[to[i]] != d[x] + 1) continue;
        int f = dfs(to[i], min(y, cap[i]));
        flow += f, y -= f;
        cap[i] -= f, cap[i ^ 1] += f;
        if (!y) break;
    }
    return flow;
}

int dinic() {
    int maxflow = 0;
    while (bfs()) {
        memcpy(p, g, sizeof g);
        maxflow += dfs(s, inf);
    }
    return maxflow;
}

```

2.11.3 spfa 费用流

```

const int inf = 0x3f3f3f3f;
int s, t, ans, len, maxflow;
int g[N], head[N], incf[N], path[N], pre[N], vis[N], d[N];
struct edge{int to, next, cap, cost;}e[M];
void add(int u, int v, int w, int c) {
    e[++ len] = (edge){v, head[u], w, c}, head[u] = len;
    e[++ len] = (edge){u, head[v], 0, -c}, head[v] = len;
}

bool bfs() {
    static int u, v, w;
    static deque<int> q;
    memset (d, 0x3f, sizeof d);
    d[s] = 0, q.push_back(s);
    while (!q.empty()) {
        vis[u = q.front()] = 0; q.pop_front();
        for (int i = head[u]; i; i = e[i].next) {
            if (e[i].cap && d[v = e[i].to] > (w = d[u] + e[i].cost)) {
                d[v] = w;
                if (!vis[v]) {
                    vis[v] = 1;

```



```
        d[v] < (q.empty() ? 0 : d[q.front()]) ? q.push_front(v) :
        → q.push_back(v);
    }
}
}
return d[t] < inf;
}
int dfs(int x, int y) {
    if (x == t) return y;
    int v, f, flow = 0; vis[x] = 1;
    for (int &i = g[x]; i; i = e[i].next) {
        v = e[i].to;
        if (vis[v] || !e[i].cap || d[x] + e[i].cost != d[v]) continue;
        f = dfs(v, min(y, e[i].cap));
        if (f) ans += f * e[i].cost, e[i].cap -= f, e[i ^ 1].cap += f, flow += f, y -= f;
        if (!y) break;
    }
    return vis[x] = 0, flow;
}
void costflow() {
    maxflow = 0; static int res = 0;
    while (bfs()) {
        memcpy(g, head, sizeof head);
        while ((res = dfs(s, inf)) != 0) maxflow += res;
    }
}
```

3 数据结构

3.1 笛卡尔树

```
/*O(n) 获得 a 数组的 min 笛卡尔树 */
void Tree() {
    for(int i = 1; i <= n; i++) {
        while(top && a[s[top]] > a[i])
            ls[i] = s[top], top--;
        fa[i] = s[top]; fa[ls[i]] = i;
        if(fa[i]) rs[fa[i]] = i;
        s[++top] = i;
    }
}
```

3.2 莫队

3.2.1 带修改莫队

```
/* 原问题：给定长度为 n 的数列，询问操作会询问某段区间内
* 有多少个子区间的异或和不为 0，修改操作会交换两个相邻位置的数字
* 转化：原数列做前缀异或和，询问变成区间内多少对不一样的数字
* 修改变为单点修改，直接带修改莫队来做即可
*
* 带修改莫队复杂度分析：
* 块大小 B 设为  $N^{2/3}$ ，故有  $(N^{1/3})$  块，修改操作  $O(1)$  完成
* 在时间轴上滚动的总复杂度：
* 左右端点所在块不动时，时间单调增加， $O((N^{1/3})^2 * O(N))$ 
* 左右端点所在块有一个变化，时间可能回退  $O(N)$ ，同上
* 无关时间的复杂度：
* 左端点所在块不变，右端点单增， $O(N^{1/3}) * O(N)$ 
* 左端点变化，右端点最多回退  $O(N)$ ， $O(N^{1/3}) * O(N)$ 
* 综上，总复杂度  $O(N^{5/3})$ 
*/
const int N = 1e5 + 5;
const int M = (1 << 20) + 2019;
const int B = 2155;
int n, m, pl, pr, cur, a[N], b[N], d[N], cnt[M];
int idxC, idxQ, tim[N], pos[N], val[N], pre[N];
/*pos[i] 第 i 次修改的位置
*pre[i] 第 i 次修改前，pos[i] 的位置的值
*val[i] 第 i 次修改要将这个位置的值改为 val[i]
*/
ll res, ans[N];
#define bel(x) ((x) / B + 1)
struct query {
    int id, tim, l, r;
    bool operator < (const query &b) const {
        if(bel(l) != bel(b.l)) return l < b.l;
        if(bel(r) != bel(b.r)) return r < b.r;
        return id < b.id;
    }
}q[N];
void add(int p){
```

```

    res += cnt[b[p]];
    cnt[b[p]] ++;
}
void del(int p){
    cnt[b[p]] --;
    res -= cnt[b[p]];
}
void modify(int cur, int dir = 1){
    if(pos[cur] >= pl && pos[cur] <= pr) del(pos[cur]);
    b[pos[cur]] = dir == 1 ? val[cur] : pre[cur];
    if(pos[cur] >= pl && pos[cur] <= pr) add(pos[cur]);
}
void change(int now){
    while(cur < idxC && tim[cur + 1] <= now) modify(++ cur);
    while(cur > 0 && tim[cur] > now) modify(cur --, -1);
}
int main(){
    int op, x, y;
    while (scanf("%d %d", &n, &m) != EOF) {
        for (int i = 1; i <= n; i++) {
            scanf("%d", &a[i]);
            b[i] = b[i - 1] ^ a[i];
            d[i] = a[i];
        }
        idxQ = idxC = 0;
        for (int i = 1; i <= m; i++) {
            scanf("%d", &op);
            if (op == 1) {
                idxQ ++;
                q[idxQ].id = idxQ;
                q[idxQ].tim = i;
                scanf("%d %d", &q[idxQ].l, &q[idxQ].r);
                q[idxQ].l --;
                ans[idxQ] = 1ll * (q[idxQ].r - q[idxQ].l)
                    * (q[idxQ].r - q[idxQ].l + 1) / 2;
            }
            else {
                tim[++ idxC] = i;
                scanf("%d", &x);
                pre[idxC] = b[x];
                b[x] ^= a[x], b[x] ^= a[x + 1];
                swap(a[x], a[x + 1]);
                pos[idxC] = x;
                val[idxC] = b[x];
            }
        }
        //因为要获取 pre[i] 的值, 所以读入时数组要相应改变, 处理完后再恢复为初始值
        //或者这里把 cur 设为最后时刻也可以哦
        for (int i = 1; i <= n; i++)
            b[i] = b[i - 1] ^ d[i];
        pl = 1, pr = 0; //保证初始时 [pl, pr] 是个空区间即可
        cur = res = 0;
        sort(q + 1, q + idxQ + 1);
        for(int i = 1; i <= idxQ; i++){

```

```

        change(q[i].tim);
        while(pl > q[i].l) add(-- pl);
        while(pr < q[i].r) add(++ pr);
        while(pl < q[i].l) del(pl ++);
        while(pr > q[i].r) del(pr --);
        ans[q[i].id] -= res;
    }
    for(int i = 1; i <= idxQ; i++)
        printf("%lld\n", ans[i]);
    for (int i = pl; i <= pr; i ++ )
        cnt[b[i]] --;
}
return 0;
}

```

3.3 平衡树

3.3.1 splay

```

int n, m, a, b, len, tot;
struct node {
    bool rev; //翻转标记
    int v, siz;
    node *c[2];
    node():rev(0),v(0),siz(0),c{NULL, NULL}{}
    node *init(int x);
    void pushdown();
    void mata() {siz = c[0] -> siz + c[1] -> siz + 1;}
    int cmp(int k) {return k<=c[0]->siz?0:(k==c[0]->siz+1?-1:1);}
    void print();
}pool[N], *null = new node();
node *node::init(int x){rev=0, v=x, siz=1, c[0]=c[1]=null;return this;}
void node::pushdown() {
    if (!rev) return;
    if (c[0] != null) c[0] -> rev ^= 1;
    if (c[1] != null) c[1] -> rev ^= 1;
    swap(c[0], c[1]), rev = 0;
}
void node::print() {
    pushdown();
    if (c[0] != null) c[0] -> print();
    if (1 <= v && v <= n) printf("%d ", v);
    if (c[1] != null) c[1] -> print();
}
node *build(int l, int r) {//初始序列为 1-n
    if (l == r) return pool[tot ++].init(r);
    node *tmp = pool[tot ++].init(mid);
    if (l < mid) tmp -> c[0] = build(l, mid - 1);
    if (mid < r) tmp -> c[1] = build(mid + 1, r);
    tmp -> mata(); return tmp;
}
void rot(node *&o, int k) {//把 k 儿子提上来
    o -> pushdown(); node *tmp = o -> c[k];
    tmp -> pushdown(); o -> c[k] = tmp -> c[!k];
}

```

```

    tmp -> c[!k] -> pushdown(); tmp -> c[!k] = o;
    o -> mata(), tmp -> mata(), o = tmp;
}
void splay(node *&o, int k) { //把以 o 为根的 splayTree 中 rk 为 k 的点提到根
    int k1 = o -> cmp(k); o -> pushdown();
    if (k1 == -1) return; o -> c[k1] -> pushdown();
    if (k1) k -= o -> c[0] -> siz + 1;
    int k2 = o -> c[k1] -> cmp(k);
    if (~k2) { //k2 != -1
        if (k2) k -= o -> c[k1] -> c[0] -> siz + 1;
        o -> c[k1] -> c[k2] -> pushdown();
        splay(o -> c[k1] -> c[k2], k);
        if (k2 == k1) rot(o, k1);
        else rot(o -> c[k1], k2);
    }
    rot(o, k1);
}
int main() {
    scanf("%d %d", &n, &m);
    node *root = build(0, n + 1); //方便边界左右处理各多开一个
    for (; m --; ) {
        scanf("%d %d", &a, &b), a ++, b ++, len = b - a + 1;
        splay(root, a - 1), splay(root -> c[1], len + 1);
        root -> c[1] -> c[0] -> rev ^= 1, root -> c[1] -> c[0] -> pushdown();
    }
    root -> print(); return 0;
}

```

3.3.2 treap

```

/* 容易实现的预开内存池 treap, 每次 head 清空即可
* 如果初始要插入 n 个 1, 可改为类似 splay 的 O(n)build 写法
* poolSize 是单组数据的最大节点数, 对于单组数据有很多插入和删除
* 导致使用的节点很多的数据, 无法使用
*/
const int poolSize = 5e5 + 10;
struct node {
    node *c[2];
    int v, r, siz;
    void update();
    void init(int x);
};
node *null = new node(), *root = null;
void node::update() {
    siz = c[0] -> siz + c[1] -> siz + 1;
}
void node::init(int x) {
    v = x, r = rand(), siz = 1;
    c[0] = c[1] = null;
}
node nodesPool[poolSize];
int head; //每次 head=0 要清空!
node *newnode(int x) {
    node *res = &nodesPool[head ++];

```

```

    res -> init(x);
    return res;
}
void rot(node *&o, int d) {
    node *tmp = o -> c[!d];
    o -> c[!d] = tmp -> c[d], tmp -> c[d] = o;
    o -> update(), tmp -> update(), o = tmp;
}
void insert(node *&o, int x) {
    if (o == null) {
        o = newnode(x);
        return;
    }
    int d = x > o -> v ? 0 : 1;
    insert(o -> c[d], x);
    if (o -> c[d] -> r < o -> r) rot(o, !d);
    o -> update();
}
void del(node *&o, int x) {
    if (x == o -> v) {
        if (o -> c[0] == null) {o = o -> c[1]; return;}
        if (o -> c[1] == null) {o = o -> c[0]; return;}
        int d = o -> c[0] -> r < o -> c[1] -> r ? 1 : 0;
        rot(o, d), del(o -> c[d], x);
    }
    else del(o -> c[x <= o -> v], x);
    o -> update();
}
void build(node *&o, int l, int r) {
    o = newnode(1);
    if (l == r) return;
    int mid = l + r >> 1;
    if (l < mid) build(o -> c[0], l, mid - 1);
    if (o -> c[0] != null && o -> c[0] -> r < o -> r) swap(o -> c[0] -> r, o -> r);
    if (mid < r) build(o -> c[1], mid + 1, r);
    if (o -> c[1] != null && o -> c[1] -> r < o -> r) swap(o -> c[1] -> r, o -> r);
    o -> update();
}

```

3.4 主席树

3.4.1 主席树

```

const int N = 130000;
const int M = N * 20; //主席树节点数, 可以直接稳妥选择  $N*(5+\log N)$ 
struct {
    int siz, l, r;
    ll sum, val;
}tr[M];
#define s(x) tr[x].sum
#define v(x) tr[x].val
#define sz(x) tr[x].siz
int tot, root[N];
int build(int l, int r) {
    int x = ++ tot;

```

```

    s(x) = sz(x) = 0;
    if (l < r) {
        l(x) = build(l, mid);
        r(x) = build(mid + 1, r);
    }
    return x;
}

int change(int o, int l, int r, int p, int y) { //在 p 的位置插入一个 y
    int x = ++ tot;
    s(x) = s(o) + y, sz(x) = sz(o) + 1;
    l(x) = l(o), r(x) = r(o), v(x) = y;
    if (l < r) {
        if (p <= mid) l(x) = change(l(o), l, mid, p, y);
        else r(x) = change(r(o), mid + 1, r, p, y);
    }
    return x;
}

ll ask(int o1, int o2, int l, int r, int k) {
    //求 (s, t] 区间前 k 小的数之和, 有 o1=root[s], o2=root[t]
    if (l == r) return v(o2) * k;
    int lsz = sz(l(o2)) - sz(l(o1));
    if (lsz == k) return s(l(o2)) - s(l(o1));
    if (lsz < k) return s(l(o2)) - s(l(o1)) + ask(r(o1), r(o2), mid + 1, r, k - lsz);
    return ask(l(o1), l(o2), l, mid, k);
}

```

3.4.2 树状数组套主席树

```

/* 在线处理, 带单点修改查询区间第 k 小 */
const int MAXN = 1e9; //数字最大值
const int N = 5e4 + 5; //数列长度
int n, m, a[N], rt[N];
int tot, tr[N * 800][3];
//这个 800 大概是 log(MAXN)*log(N), 反正照着内存上限开吧, 不行就只能离线做离散化了
int tmp1[100], tmp2[100];
#define l(x)/r(x)/s(x) tr[x][0/1/2]
int change(int o, int l, int r, int k, int v) {
    int x = ++ tot; s(x) = s(o) + v;
    if (l == r) return x; l(x) = l(o), r(x) = r(o);
    k > mid ? r(x) = change(r(o), mid + 1, r, k, v) : l(x) = change(l(o), l, mid, k, v);
    return x;
}

void modify(int i, int p, int v) {
    while (i <= n) rt[i] = change(rt[i], 1, MAXN, p, v), i += lb(i);
}

int ask(int l, int r, int k) {
    if (l == r) return r; int sum = 0;
    for (int i = 1; i <= tmp1[0]; i++) sum -= s(l(tmp1[i]));
    for (int i = 1; i <= tmp2[0]; i++) sum += s(l(tmp2[i]));
    if (k > sum) {
        for (int i = 1; i <= tmp1[0]; i++) tmp1[i] = r(tmp1[i]);
        for (int i = 1; i <= tmp2[0]; i++) tmp2[i] = r(tmp2[i]);
        return ask(mid + 1, r, k - sum);
    }
}

```

```

    else {
        for (int i = 1; i <= tmp1[0]; i++) tmp1[i] = l(tmp1[i]);
        for (int i = 1; i <= tmp2[0]; i++) tmp2[i] = l(tmp2[i]);
        return ask(l, mid, k);
    }
}
int query(int l, int r, int k) { // 查询区间第 k 小
    tmp1[0] = tmp2[0] = 0;
    for (int i = l - 1; i > 0; i -= lb(i)) tmp1[++ tmp1[0]] = rt[i];
    for (int i = r; i > 0; i -= lb(i)) tmp2[++ tmp2[0]] = rt[i];
    return ask(1, MAXN, k);
}
int main(){
    int t; char op[5];
    for (cin >> t; t--;) {
        cin >> n >> m; tot = 0;
        for (int i = 1; i <= n; i++) cin >> a[i], rt[i] = 0;
        for (int i = 1; i <= n; i++) modify(i, a[i], 1);
        for (int i, j, k; m--;) {
            cin >> op >> i >> j;
            if (op[0] == 'C') modify(i, a[i], -1), modify(i, a[i] = j, 1);
            else cin >> k, printf("%d\n", query(i, j, k));
        }
    }
    return 0;
}

```

3.5 线段树

3.5.1 zkw 线段树

```

if (~s&1)
if ( t&1)

```

3.5.2 李超线段树

*/*problem: 每次在二维平面插入一条线段, 询问所有线段中, 与 $x=k$ 的交点 y 值最大的线段 id*

** 李超线段树, 标记永久化, 线段树每个节点保存的是当前区间最占优势的那条线段*

** 询问某个点, 将根到它的路径上的所有节点保存的优势线段比较一下即可*

** 一个区间裂成 \log 个区间, 每个区间会 *pushdown* 到叶节点, 所以 $O(\log^2 n)$*

** 插入时记得特判斜率不存在即 $x_1=x_2$ 的情况*

** 缺点: 永久化了标记, 所以不支持删除操作*

** 拓展应用: 可以拿来维护斜率优化, 不过多一个 \log , 以及多个离散化*

**/*

```

struct Seg{double k, b;int id;};
struct node {Seg s;int hav;}tr[N << 2];
void pushdown(int o, int l, int r, Seg sg){
    if(!tr[o].hav) return (void) (tr[o].s = sg, tr[o].hav = 1);
    double l1 = sg.k * l + sg.b, r1 = sg.k * r + sg.b;
    double l2 = tr[o].s.k * l + tr[o].s.b, r2 = tr[o].s.k * r + tr[o].s.b;
    if(l2 >= l1 && r2 >= r1) return;
    if(l1 >= l2 && r1 >= r2) return (void) (tr[o].s = sg);
    double pos = (sg.b - tr[o].s.b) / (tr[o].s.k - sg.k);
    if(pos <= mid) pushdown(lc, l, mid, r1 > r2 ? tr[o].s : sg);
    else pushdown(rc, mid + 1, r, l1 > l2 ? tr[o].s : sg);
}

```



```

    if((l1 > l2 && pos >= mid) || (r1 > r2 && pos < mid)) tr[o].s = sg;
}
void add(int o, int l, int r, int s, int t, Seg sg) {
    if (s <= l && r <= t) return (void)pushdown(o, l, r, sg);
    if (s <= mid) add(lc, l, mid, s, t, sg);
    if (mid < t) add(rc, mid + 1, r, s, t, sg);
}
Seg query(int o, int l, int r, int p) {
    if (l == r) return tr[o].hav ? tr[o].s : (Seg){0, 0, 0};
    Seg sg = p <= mid ? query(lc, l, mid, p) : query(rc, mid + 1, r, p);
    if (!tr[o].hav) return sg;
    double p1 = tr[o].s.k * p + tr[o].s.b, p2 = sg.k * p + sg.b;
    if(!sg.id || (p1 > p2 || (fabs(p1 - p2) < eps && tr[o].s.id < sg.id))) sg = tr[o].s;
    return sg;
}
/* 以下为区间查询且维护的最小值，除计算交点外均为整数操作,d[i] 代表 i 的实际 x 坐标 */
void pushup(int o, int l, int r) {
    tr[o].minv = min(tr[lc].minv, tr[rc].minv);
    tr[o].minv = min(tr[o].minv, min(tr[o].s.k * d[l], tr[o].s.k * d[r]) + tr[o].s.b);
}
void pushdown(int o, int l, int r, Seg sg){
    ll l1 = sg.k * d[l] + sg.b, r1 = sg.k * d[r] + sg.b;
    ll l2 = tr[o].s.k * d[l] + tr[o].s.b, r2 = tr[o].s.k * d[r] + tr[o].s.b;
    if(l2 <= l1 && r2 <= r1) return;
    if(l1 <= l2 && r1 <= r2) {tr[o].s = sg;
        if (l == r) {tr[o].minv = l1;return;}}
    else {double pos = 1.0 * (sg.b - tr[o].s.b) / (tr[o].s.k - sg.k);
        if(pos <= d[mid]) pushdown(lc, l, mid, r1 < r2 ? tr[o].s : sg);
        else pushdown(rc, mid + 1, r, l1 < l2 ? tr[o].s : sg);
        if((l1 < l2 && pos > d[mid]) || (r1 < r2 && pos <= d[mid])) tr[o].s = sg;
    }
    pushup(o, l, r);
}
void build(int o, int l, int r) {
    tr[o].s = (Seg){0, tr[o].minv = inf};if (l == r) return;
    build(lc, l, mid);build(rc, mid + 1, r);
}
void add(int o, int l, int r, int s, int t, Seg sg) {
    if (s <= l && r <= t) return (void)pushdown(o, l, r, sg);
    if (s <= mid) add(lc, l, mid, s, t, sg);
    if (mid < t) add(rc, mid + 1, r, s, t, sg);
    pushup(o, l, r);
}
ll ask(int o, int l, int r, int s, int t) {
    if (s <= l && r <= t) return tr[o].minv;
    ll res = min(tr[o].s.k * d[max(s, l)], tr[o].s.k * d[min(r, t)]) + tr[o].s.b;
    if (s <= mid) res = min(res, ask(lc, l, mid, s, t));
    if (mid < t) res = min(res, ask(rc, mid + 1, r, s, t));
    return res;
}

```

3.5.3 吉老师线段树

```

/* 区间每个数变为  $\min(a[i], t) + \text{区间最大} + \text{区间和}$   $O(n \log)$  */
const int N = (1 << 20) + 5;
#define lc (o << 1)
#define rc (lc | 1)
struct node {
    int max1, max2, cnt;
    ll sum;
} tr[N << 1];
int T, n, m;
int op, x, y, t;
int a[N];
void pushup(int o) {
    if (tr[lc].max1 == tr[rc].max1) {
        tr[o].max1 = tr[lc].max1;
        tr[o].cnt = tr[lc].cnt + tr[rc].cnt;
        tr[o].max2 = max(tr[lc].max2, tr[rc].max2);
    }
    else {
        if (tr[lc].max1 > tr[rc].max1) {
            tr[o] = tr[lc];
            tr[o].max2 = max(tr[o].max2, tr[rc].max1);
        }
        else {
            tr[o] = tr[rc];
            tr[o].max2 = max(tr[o].max2, tr[lc].max1);
        }
    }
    tr[o].sum = tr[lc].sum + tr[rc].sum;
}
void pushdown(int o) {
    if (tr[o].max1 < tr[lc].max1) {
        tr[lc].sum += 1ll * (tr[o].max1 - tr[lc].max1) * tr[lc].cnt;
        tr[lc].max1 = tr[o].max1;
    }
    if (tr[o].max1 < tr[rc].max1) {
        tr[rc].sum += 1ll * (tr[o].max1 - tr[rc].max1) * tr[rc].cnt;
        tr[rc].max1 = tr[o].max1;
    }
}
void build(int o, int l, int r) {
    if (l == r) {
        tr[o].max1 = tr[o].sum = a[r];
        tr[o].cnt = 1, tr[o].max2 = 0;
        return;
    }
    int mid = l + r >> 1;
    build(lc, l, mid);
    build(rc, mid + 1, r);
    pushup(o);
}
void update(int o, int l, int r, int s, int t, int v) {
    if (v >= tr[o].max1) return;

```

```

pushdown(o);
if (s <= l && r <= t) {
    if (v > tr[o].max2) {
        tr[o].sum += 1ll * (v - tr[o].max1) * tr[o].cnt;
        tr[o].max1 = v;
        return;
    }
}
int mid = l + r >> 1;
if (s <= mid) update(lc, l, mid, s, t, v);
if (t > mid) update(rc, mid + 1, r, s, t, v);
pushup(o);
}
ll ask_max(int o, int l, int r, int s, int t) {
    if (s <= l && r <= t) return tr[o].max1;
    pushdown(o);
    int mid = l + r >> 1;
    ll res = 0;
    if (s <= mid) res = max(res, ask_max(lc, l, mid, s, t));
    if (t > mid) res = max(res, ask_max(rc, mid + 1, r, s, t));
    return res;
}
ll ask_sum(int o, int l, int r, int s, int t) {
    if (s <= l && r <= t) return tr[o].sum;
    pushdown(o);
    int mid = l + r >> 1;
    ll res = 0;
    if (s <= mid) res += ask_sum(lc, l, mid, s, t);
    if (t > mid) res += ask_sum(rc, mid + 1, r, s, t);
    return res;
}
int main() {
    ios::sync_with_stdio(false);
    for (cin >> T; T--; ) {
        cin >> n >> m;
        for (int i = 1; i <= n; i++)
            cin >> a[i];
        build(1, 1, n);
        while (m--) {
            cin >> op;
            if (op == 0) {
                cin >> x >> y >> t;
                update(1, 1, n, x, y, t);
            }
            else if (op == 1) {
                cin >> x >> y;
                cout << ask_max(1, 1, n, x, y) << '\n';
            }
            else {
                cin >> x >> y;
                cout << ask_sum(1, 1, n, x, y) << '\n';
            }
        }
    }
}

```

```

    return 0;
}

```

3.5.4 线段树维护凸包

/* 线段树维护凸包，单点修改区间查询，原题变换式子形式可得斜率固定，截距最大就在凸包上
 * 所以线段树维护凸包：区间查询所以一个点要修改在 \log 层上
 * 如果区间修改单点查询，即某个点有存活的时间区间，那就把一个区间覆盖在 \log 区间上
 * 然后从根到底查询即可，时间复杂度不变
 */

```

struct point {
    ll x, y;
    point():x(0), y(0){}
    point(ll x, ll y):x(x), y(y){}
    ll operator *(const point &a) const {return x * a.x + y * a.y;}
    ll operator ^(const point &a) const {return x * a.y - y * a.x;}
    point operator -(const point &a) const {return point(x - a.x, y - a.y);}
    bool operator <(const point &a) const {return x == a.x ? y < a.y : x < a.x;}
};

vector<point> tr[N << 2], v1[N << 2], v2[N << 2]; //v1/v2 上/下凸壳

void add(int p, const point &pt, int o = 1, int l = 1, int r = M) {
    tr[o].push_back(pt);
    if (p == r) {
        sort(tr[o].begin(), tr[o].end());
        for (int i = 0; i <= r - 1; i++) {
            while(v1[o].size() > 1 && ((tr[o][i] - v1[o][v1[o].size() - 1]) ^
                (v1[o][v1[o].size() - 2] - v1[o][v1[o].size() - 1])) >= 0)
                v1[o].pop_back();
            v1[o].push_back(tr[o][i]);
            while(v2[o].size() > 1 && ((tr[o][i] - v2[o][v2[o].size() - 1]) ^
                (v2[o][v2[o].size() - 2] - v2[o][v2[o].size() - 1])) <= 0)
                v2[o].pop_back();
            v2[o].push_back(tr[o][i]);
        }
    }
    if (l == r) return;
    p <= mid ? add(p, pt, lc, l, mid) : add(p, pt, rc, mid + 1, r);
}

//对上凸壳查询，查询下凸壳就把 v1 都换成 v2 就有了
ll query1(int s, int t, const point &pt, int o = 1, int l = 1, int r = M) {
    if (s <= l && r <= t) {
        int L = 1, R = v1[o].size() - 1, Mid, Ans = 0;
        while (L <= R) {
            Mid = (L + R) >> 1;
            if (pt * v1[o][Mid] > pt * v1[o][Mid - 1]) Ans = Mid, L = Mid + 1;
            else R = Mid - 1;
        }
        return pt * v1[o][Ans];
    }
    ll ans = -1e18;
    if (s <= mid) ans = max(ans, query1(s, t, pt, lc, l, mid));
    if (mid < t) ans = max(ans, query1(s, t, pt, rc, mid + 1, r));
    return ans;
}

```

3.5.5 线段树的分裂和合并

```

/* 原题：给一个排列，每次对一个子区间排序，0/1 代表正/逆序，最后问某个位置的数字
* 套路解法：二分答案  $k$ ，把原排列化成 01 序列，排序用区间覆盖，可判断  $ans$  与  $k$  的关
  ↳ 系， $O(n \log 2n)$ 
* 巧妙解法：用权值线段树维护已排序的区间，暴力合并。每次切出需要的区间只切 2 次，所以总共合并
  ↳ 次数  $O(n)$ 
* 时空复杂度都是  $O(n \log n)$ 
*/
int tr[N * 60][3]; // 不回回收空间写法，空间足够就这么写，不够就回收一下
#define l(x) tr[x][0]
#define r(x) tr[x][1]
#define s(x) tr[x][2]
#define mid (l + r >> 1)
int n, m, k, o[N];
int tot, rt[N];
set<int> head; // 放所有排好序区间的左端点，帮助定位
void ins(int &o, int l, int r, int p) {
    if (!o) o = ++ tot; s(o)++;
    if (l == r) return;
    p > mid ? ins(r(o), mid + 1, r, p) : ins(l(o), l, mid, p);
}
int ask(int o, int l, int r) {
    return l == r ? r : (l(o) ? ask(l(o), l, mid) : ask(r(o), mid + 1, r));
}
void merge(int &x, int y) { // 把以 y 为根的线段树合并到以 x 为根的线段树里
    if (!(x && y)) {x |= y; return;}
    merge(l(x), l(y)), merge(r(x), r(y)), s(x) += s(y);
}
void split(int &x, int y, int k, int d) { // 把以 y 为根的权值线段树的前 k 个元素 (0/1 代表从
  ↳ 小到大/从大到小) 放到以 x 为根的权值线段树下
    if (s(y) == k) return;
    s(x = ++ tot) = s(y) - k; s(y) = k;
    if (k <= s(tr[y][d])) split(tr[x][d], tr[y][d], k, d), tr[x][!d] = tr[y][!d],
      ↳ tr[y][!d] = 0;
    else split(tr[x][!d], tr[y][!d], k - s(tr[y][d]), d);
}
auto cut(int p) { // 切出以 p 为左端点的区间
    auto it = head.lower_bound(p);
    if (*it == p) return it;
    it--; split(rt[p], rt[*it], p - *it, o[p] = o[*it]);
    return head.insert(p).first;
}
int main() {
    ios::sync_with_stdio(false);
    cin >> n >> m; head.insert(n + 1);
    for (int a, i = 1; i <= n; i++) {
        cin >> a; head.insert(i);
        ins(rt[i], 1, n, a);
    }
    for (int op, l, r, i = 1; i <= m; i++) {
        cin >> op >> l >> r;
        auto li = cut(l), ri = cut(r + 1);
        for (auto j = ++ li; j != ri; j++) merge(rt[l], rt[*j]);
    }
}

```

```

    o[l] = op, head.erase(li, ri);
}
cin >> k;
cut(k), cut(k + 1);
cout << ask(rt[k], 1, n) << endl;
return 0;
}

```

3.5.6 线段树维护单调序列长度

```

/* 询问 1-n 中多少个 i, 满足对任意 j<i 都有 h[j]<h[i]
 * 单点修改, ans=tr[1].len
 * 也可分块做 O(n*(nlogn)^0.5)
 */
struct node {double mx;int len;}tr[N << 2];
int calc(int o, int l, int r, double v) {
    if(l == r) return tr[o].mx > v;
    int mid = (l + r) / 2;
    if(tr[lc].mx <= v) return calc(rc, mid + 1, r, v);
    return tr[o].len - tr[lc].len + calc(lc, l, mid, v);
}
void change(int o, int l, int r, int p, double v) {
    if(r == l) {tr[o] = (node){v, 1};return;}
    int mid = (l + r) >> 1;
    if(p > mid) change(rc, mid + 1, r, p, v);
    else change(lc, l, mid, p, v);
    tr[o].mx = std::max(tr[lc].mx, tr[rc].mx);
    tr[o].len = tr[lc].len + calc(rc, mid + 1, r, tr[lc].mx);
}

```

3.5.7 区间除区间加

```

/* 区间除区间加, 导致区间的 max-min 始终不增
 * 对某个区间做 log 次有效操作后会有 max=min
 * 所以对于 max=min 的区间, 把区间除转成区间加操作即可方便处理
 */
struct node {ll minv, maxv, sum, lazy;}tr[N << 2];
ll div(ll x, int y) { // 向下取整
    ll z = x / y;
    if (z * y > x) z--;
    return z;
}
void div(int o, int l, int r) { // 对 [s,t] 的所有 ai 都除以 z 向下取整
    if (s <= l && r <= t && tr[o].maxv - div(tr[o].maxv, z)
        == tr[o].minv - div(tr[o].minv, z)) {
        ll del = tr[o].maxv - div(tr[o].maxv, z);
        tr[o].maxv -= del, tr[o].minv -= del;
        tr[o].sum -= del * (r - l + 1), tr[o].lazy -= del;
        return;
    }
    pushdown(o, l, r);
    if (s <= mid) div(lc, l, mid);
    if (mid < t) div(rc, mid + 1, r);
    pushup(o);
}

```

```

}
```

3.6 KDTree

3.6.1 3 维 KDtree

```

/*O(n^(2-1/k)), k 为维度, 查询区间最大值 */
int nowD, ans, x[3], y[3]; //x,y 数组存放查询区间各维度的 min 和 max
int n, m, a[N], b[N], c[N], d[N];
struct node {
    int Max[3], Min[3], d[3];
    int val, maxv;
    node *c[2];
    node() {
        c[0] = c[1] = NULL;
        val = maxv = 0;
    }
    void pushup();
    bool operator < (const node &a) const {
        return d[nowD] < a.d[nowD];
    }
} Null, nodes[N];
node *root = &Null;
inline void node::pushup() {
    if (c[0] != &Null) {
        if (c[0] -> Max[1] > Max[1]) Max[1] = c[0] -> Max[1];
        if (c[0] -> Max[2] > Max[2]) Max[2] = c[0] -> Max[2];
        if (c[0] -> Min[0] < Min[0]) Min[0] = c[0] -> Min[0];
        if (c[0] -> Min[2] < Min[2]) Min[2] = c[0] -> Min[2];
        if (c[0] -> maxv > maxv) maxv = c[0] -> maxv;
    }
    if (c[1] != &Null) {
        if (c[1] -> Max[1] > Max[1]) Max[1] = c[1] -> Max[1];
        if (c[1] -> Max[2] > Max[2]) Max[2] = c[1] -> Max[2];
        if (c[1] -> Min[0] < Min[0]) Min[0] = c[1] -> Min[0];
        if (c[1] -> Min[2] < Min[2]) Min[2] = c[1] -> Min[2];
        if (c[1] -> maxv > maxv) maxv = c[1] -> maxv;
    }
}
inline node *build(int l, int r) {
    int mid = l + r >> 1; nowD = rand() % 3;
    nth_element(nodes + l, nodes + mid, nodes + r + 1);
    node *res = &nodes[mid];
    if (l != mid) res -> c[0] = build(l, mid - 1);
    else res -> c[0] = &Null;
    if (r != mid) res -> c[1] = build(mid + 1, r);
    else res -> c[1] = &Null;
    res -> pushup();
    return res;
}
inline int calc(node *o) {
    if (y[0] < o -> Min[0] || x[1] > o -> Max[1] || x[2] > o -> Max[2] || y[2] < o ->
        Min[2]) return -1;
    return o -> maxv;
}
}
```

```

inline void query(node *o) {
    if (o -> val > ans && y[0] >= o -> d[0] && x[1] <= o -> d[1] && x[2] <= o -> d[2] &&
        y[2] >= o -> d[2]) ans = o -> val;
    int dl, dr;
    if (o -> c[0] != &Null) dl = calc(o -> c[0]);
    else dl = -1;
    if (o -> c[1] != &Null) dr = calc(o -> c[1]);
    else dr = -1;
    if (dl > dr) {
        if (dl > ans) query(o -> c[0]);
        if (dr > ans) query(o -> c[1]);
    } else {
        if (dr > ans) query(o -> c[1]);
        if (dl > ans) query(o -> c[0]);
    }
}
}
int main() {
    /* 初始化所有节点 */
    root = build(1, n);
    /* 对每个查询 */ {
        x[] = {}, y[] = {};
        ans = 0, query(root);
    }
}

```

3.6.2 KDtree 二维空间区间覆盖单点查询

```

int nowD, x[2], y[2], z;
struct node {
    int Max[2], Min[2], d[2];
    int val, lazy;
    node *c[2];
    node() {c[0] = c[1] = NULL;}
    void pushup();
    void pushdown();
    bool operator < (const node &a) const {return d[nowD] < a.d[nowD];}
} Null, nodes[N];
node *root = &Null;
inline void node::pushup() {
    if (c[0] != &Null) ...
    if (c[1] != &Null) ...
}
inline void node::pushdown() {
    if (c[0] != &Null) c[0] -> val = c[0] -> lazy = lazy;
    if (c[1] != &Null) c[1] -> val = c[1] -> lazy = lazy;
    lazy = -1;
}
inline node *build(int l, int r, int D) {
    int mid = l + r >> 1; nowD = D;
    nth_element(nodes + l, nodes + mid, nodes + r + 1);
    node *res = &nodes[mid];
    if (l != mid) res -> c[0] = build(l, mid - 1, !D);
    else res -> c[0] = &Null;
    if (r != mid) res -> c[1] = build(mid + 1, r, !D);
}

```



```

    else res -> c[1] = &Null;
    res -> pushup();
    return res;
}
inline int query(node *o) {
    if (o == &Null) return -1;
    if (o -> lazy != -1) o -> pushdown();
    if (x[o] > o -> Max[0] || y[o] > o -> Max[1] || x[o] < o -> Min[0] || y[o] < o ->
        Min[1]) return -1;
    if (x[o] == o -> d[0]) return o -> val;
    return max(query(o -> c[0]), query(o -> c[1]));
}
inline void modify(node *o) {
    if (o == &Null) return;
    if (o -> lazy != -1) o -> pushdown();
    if (x[o] > o -> Max[0] || y[o] > o -> Max[1] || x[o] < o -> Min[0] || y[o] < o ->
        Min[1]) return;
    if (x[o] <= o -> Min[0] && y[o] <= o -> Min[1] && x[o] >= o -> Max[0] && y[o] >= o ->
        Max[1]) {
        o -> val = o -> lazy = z; return;
    }
    if (x[o] <= o -> d[0] && y[o] <= o -> d[1] && x[o] >= o -> d[0] && y[o] >= o -> d[1])
        o -> val = z;
    modify(o -> c[0]), modify(o -> c[1]);
}
int main() {
    /* 初始化所有节点 */
    root = build(1, n, 0);
    /* 对每个操作 */ {
        x[] = {}, y[] = {};
        query(root) / modify(root);
    }
}

```

3.6.3 KDtree 二维空间单点修改区间查询

```

/* 调整重构系数可以影响常数
 * 询问多就让系数接近 0.70-0.75, 询问少就让系数在 0.8-0.90
 */
const int inf = 1e9;
int n, m, tot, nowD;
struct node {
    int Max[2], Min[2], d[2];
    int sum, siz, val;
    node *c[2];
    node() {
        Max[0] = Max[1] = -inf;
        Min[0] = Min[1] = inf;
        sum = val = siz = 0;
        c[0] = c[1] = NULL;
        d[0] = d[1] = 0;
    }
    void update();
} Null, nodes[200010], *temp[200010];

```

```

node *root = &Null;
inline void node::update() {
    siz = c[0] -> siz + c[1] -> siz + 1;
    sum = c[0] -> sum + c[1] -> sum + val;
    if (c[0] != &Null) ...
    if (c[1] != &Null) ...
}
inline bool cmp(const node *a, const node *b) {
    return a -> d[nowD] < b -> d[nowD];
}
inline void traverse(node *o) {
    if (o == &Null) return;
    temp[++ tot] = o;
    traverse(o -> c[0]);
    traverse(o -> c[1]);
}
inline node *build(int l, int r, int D) {
    int mid = l + r >> 1; nowD = D;
    nth_element(temp + l, temp + mid, temp + r + 1, cmp);
    node *res = temp[mid];
    res -> Max[0] = res -> Min[0] = res -> d[0];
    res -> Max[1] = res -> Min[1] = res -> d[1];
    if (l != mid) res -> c[0] = build(l, mid - 1, !D);
    else res -> c[0] = &Null;
    if (r != mid) res -> c[1] = build(mid + 1, r, !D);
    else res -> c[1] = &Null;
    res -> update();
    return res;
}
int x, y, a, b, tmpD;
node **tmp;
inline void rebuild(node *&o, int D) {
    tot = 0, traverse(o), o = build(1, tot, D);
}
inline void insert(node *&o, node *p, int D) {
    if (o == &Null) {o = p; return;}
    if (p -> Max[0] > o -> Max[0]) o -> Max[0] = p -> Max[0];
    if (p -> Max[1] > o -> Max[1]) o -> Max[1] = p -> Max[1];
    if (p -> Min[0] < o -> Min[0]) o -> Min[0] = p -> Min[0];
    if (p -> Min[1] < o -> Min[1]) o -> Min[1] = p -> Min[1];
    o -> siz ++, o -> sum += p -> sum;
    insert(o -> c[p -> c[D] >= o -> c[D]], p, !D);
    if (max(o -> c[0] -> siz, o -> c[1] -> siz) > int(o -> siz * 0.75 + 0.5)) tmpD = D,
        tmp = &o;
}
inline int query(node *o) { // 查询区间为 [x, a] * [y, b] 的权值和
    if (o == &Null) return 0;
    if (x > o -> Max[0] || y > o -> Max[1] || a < o -> Min[0] || b < o -> Min[1]) return
        0;
    if (x <= o -> Min[0] && y <= o -> Min[1] && a >= o -> Max[0] && b >= o -> Max[1])
        return o -> sum;
    return (x <= o -> d[0] && y <= o -> d[1] && a >= o -> d[0] && b >= o -> d[1] ? o ->
        val : 0)
        + query(o -> c[1]) + query(o -> c[0]);
}

```

```

}
int main() {
    node *ttt = &Null;
    /* 对每个操作 */ {
        /* 在 (x,y) 的位置增加权值为 a 的点 */ {
            n ++, nodes[n].sum = nodes[n].val = a, nodes[n].siz = 1;
            nodes[n].Max[0] = nodes[n].Min[0] = nodes[n].d[0] = x;
            nodes[n].Max[1] = nodes[n].Min[1] = nodes[n].d[1] = y;
            nodes[n].c[0] = nodes[n].c[1] = &Null;
            tmp = &(ttt), insert(root, &nodes[n], 0);
            if (*tmp != &Null) rebuild(*tmp, tmpD);
        }
        /* 查询 */ {x, a, y, b = ....; query(root);}
    }
}

```

3.6.4 KDtree 找最近点

```

/* 为了维持树的平衡, 可以一开始把所有点都读进来 build
 * 然后打 flag 标记该点是否被激活, 所有点均参与建树时的更新
 * 需要再额外维护 siz, 记录子树下已经被激活的节点总个数
 */
const int inf = 1 << 30;
int n, m, ql, qr, ans, tot, nowD;
struct point {
    int d[2];
    bool operator < (const point &a) const {
        if (d[nowD] == a.d[nowD]) return d[!nowD] < a.d[!nowD];
        return d[nowD] < a.d[nowD];
    }
}pot[N];
struct node {
    int min[2], max[2], d[2];
    node *c[2];
    node() {
        min[0] = min[1] = max[0] = max[1] = d[0] = d[1] = 0;
        c[0] = c[1] = NULL;
    }
    node(int x, int y);
    void update();
}t[N], Null, *root;
node::node(int x, int y) {
    min[0] = max[0] = d[0] = x;
    min[1] = max[1] = d[1] = y;
    c[0] = c[1] = &Null;
}
inline void node::update() {
    if (c[0] != &Null) ...
    if (c[1] != &Null) ...
}
inline void build(node *&o, int l, int r, int D) {
    int mid = l + r >> 1; nowD = D;
    nth_element(pot + l, pot + mid, pot + r + 1);
}

```

```

    o = new node(pot[mid].d[0], pot[mid].d[1]);
    if (l != mid) build(o -> c[0], l, mid - 1, !D);
    if (r != mid) build(o -> c[1], mid + 1, r, !D);
    o -> update();
}

inline void insert(node *o) {
    node *p = root; int D = 0;
    while (1) {
        if (o -> max[0] > p -> max[0]) p -> max[0] = o -> max[0];
        if (o -> max[1] > p -> max[1]) p -> max[1] = o -> max[1];
        if (o -> min[0] < p -> min[0]) p -> min[0] = o -> min[0];
        if (o -> min[1] < p -> min[1]) p -> min[1] = o -> min[1];
        if (o -> d[D] >= p -> d[D]) {
            if (p -> c[1] == &Null) {
                p -> c[1] = o; return;
            } else p = p -> c[1];
        } else {
            if (p -> c[0] == &Null) {
                p -> c[0] = o; return;
            } else p = p -> c[0];
        }
        D ^= 1;
    }
}

inline int dist(node *o) {
    int dis = 0;
    if (ql < o -> min[0]) dis += o -> min[0] - ql;
    if (ql > o -> max[0]) dis += ql - o -> max[0];
    if (qr < o -> min[1]) dis += o -> min[1] - qr;
    if (qr > o -> max[1]) dis += qr - o -> max[1];
    return dis;
}

inline void query(node *o) {
    int dl, dr, d0;
    d0 = abs(o -> d[0] - ql) + abs(o -> d[1] - qr);
    if (d0 < ans) ans = d0;
    dl = (o -> c[0] != &Null) ? dist(o -> c[0]) : inf;
    dr = (o -> c[1] != &Null) ? dist(o -> c[1]) : inf;
    if (dl < dr) {
        if (dl < ans) query(o -> c[0]);
        if (dr < ans) query(o -> c[1]);
    } else {
        if (dr < ans) query(o -> c[1]);
        if (dl < ans) query(o -> c[0]);
    }
}

int main() {
    ios::sync_with_stdio(false);
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
        cin >> pot[i].d[0] >> pot[i].d[1];
    build(root, 1, n, 0);
    for (int x, y, z; m--; ) {
        cin >> x >> y >> z;

```

```
    if (x == 1) {
        t[tot].max[0] = t[tot].min[0] = t[tot].d[0] = y;
        t[tot].max[1] = t[tot].min[1] = t[tot].d[1] = z;
        t[tot].c[0] = t[tot].c[1] = &Null;
        insert(&t[tot ++]);
    } else {
        ans = inf, ql = y, qr = z;
        query(root), printf("%d\n", ans);
    }
}
return 0;
}
```

4 构造

4.1 若干排列使所有数对都出现一次

```

/*n/2 个排列使得所有数对 (i, j) 且 i < j 都出现一次
 *n 为奇数则首尾相连, 偶数不连
 */
typedef vector<int> vi;
void get_even(int n, vi ans[]) {
    vi a(n);
    for (int i = 0; i < n; i++)
        a[i] = i + 1;
    for (int i = 1; i <= n / 2; i++) {
        ans[i].resize(n + 1);
        for (int j = 0; j < n / 2; j++)
            ans[i][j * 2] = a[j], ans[i][j * 2 + 1] = a[n - 1 - j];
        int t = a[n - 1];
        for (int j = n - 1; j > 0; j--)
            a[j] = a[j - 1];
        a[0] = t;
    }
}
void get_odd(int n, vi ans[]) {
    get_even(n - 1, ans);
    for (int i = 1; i <= n / 2; i++) {
        for (int j = n - 1; j > 0; j--)
            ans[i][j] = ans[i][j - 1] + 1;
        ans[i][0] = 1;
    }
}
int main() {
    vi ans[2019];
    int n; cin >> n;
    if (n & 1) get_odd(n, ans);
    else get_even(n, ans);
    return 0;
}

```

4.2 rec-free

```

/* 不存在四个 1 构成一个矩形, 并使得 1 尽量多, 输出 01 矩阵 */
const int N = 200, n = 150, M = 13; // M 为质数, N > M * M > n
int a[N][N], b[N][N], c[N][N];
void make() {
    for (int i = 1; i <= M; i++)
        for (int j = 1; j <= M; j++)
            a[i][j + 1] = M * (j - 1) + i;
    for (int i = 1; i <= M; i++)
        for (int j = 1; j <= M; j++)
            c[i][a[i][j]] = 1;
    for (int k = 1; k < M; k++) {
        memcpy(b, a, sizeof b);
        for (int i = 1; i <= M; i++)
            for (int j = 1; j <= M; j++)

```

```

        a[i][j] = (b[i + j - 1 - ((i + j - 1) > M ? M : 0)][j]);
    for (int i = 1; i <= M; i++)
        for (int j = 1; j <= M; j++)
            c[k * M + i][a[i][j]] = 1;
}
}

```

4.3 点边均整数多边形

/* 构造满足以下条件的简单多边形 (可以凹但边不能相交)

* 有 k 个点 k 条边, 所有边都为整数, 所有点的坐标都为整数

* 不存在与坐标轴平行的边

*/

```

typedef pair<int, int> P;
P operator * (P a, int b){return P(a.first * b, a.second * b);}
P operator + (P a, P b) {return P(a.first + b.first, a.second + b.second);}
int main() {
    int n; cin >> n;
    if (n == 3) return printf("0 0\n4 3\n-20 21\n"), 0;
    int m = n / 2 - 1;
    vector<P> v;
    v.push_back(P(0, 0));
    v.push_back(P(4, -3) * m);
    v.push_back(P(4, 0) * (2 * m));
    int lim = (n & 1) ? n - 1 : n;
    for (int i = 4; i <= lim; ++ i) {
        P last = v.back();
        if ((i % 2) == 0) v.push_back(last + P(-4, 3));
        else v.push_back(last + P(-4, -3));
    }
    if (n & 1) v.push_back(P(-20, 48));
    for (P p: v) printf("%d %d\n", p.first, p.second);
}

```

5 数学

5.1 线性基

5.1.1 线性基总结

1. 可贪心 (xy 都可放在某个位置时, 放谁都是等价的, 因此可贪心选择)
2. 支持线段树维护, 可以获得区间线性基 (如果支持离线且无修改可以放弃线段树维护减个 \log)
3. 如果有删除某个数的操作:
强制在线: 线段树维护, 把叶子结点的线性基清空, 再更新上来, $O(\log^2)$
允许离线: 维护每个数字的有效区间, 然后从前往后扫即可, $O(\log)$
4. (u, v) 某条路径的异或和 $= (u, v)$ 任选一条路径的异或和, 异或若干个环的结果
5. 对于 n 个数的线性基中有 k 个数, 那么所有子集可以表示出 2^k 个数
每个数的表示方法都有 $2^{(n-k)}$ 种
6. 判断删掉一些边后图是否连通:
先 dfs 出一棵树, 然后对于非树边随机一个权值, 树上点权 = 与该点相连的所有非树边的权值异或和
树上边权 = 这条边连接的子节点为根的子树里所有点权的异或和
删除某些边后图不连通的判定: 这些边的边权能异或组合出 0 来
7. 区间异或某个数 + 区间查询最大异或和
考虑差分令 $b[i] = a[i] \oplus a[i-1]$
那么修改区间就变成了单点修改
区间查询就变成了求 $\{a[l], b[l+1], \dots, b[r]\}$ 的结果
用树状数组维护 a , 线段树维护 b 即可
8. 计算所有满足 uv 之间存在路径异或和为 s 的 (u, v, s) 对里 s 的总和
按位考虑即可, $cnt[i][0/1]$ 为 $d[u]$ 的第 i 位为 0/1 的数量
那么 $d[u]$ 和 $d[v]$ 第 i 位都为 0 的数量就是 $cnt[i][0]^2 + cnt[i][1]^2$
这时看线性基第 i 位如果能填 1 (存在 $a[j] \gg i \& 1$), 那么上面的数量 $\times (2^{(s-1)})$ 就是对答案贡献
 s 为线性基内元素个数, 填 0 同理

5.1.2 线性基模板

```
int n; // 数字总个数
struct Base {
    const static int B = 64;
    ll a[B], s; vector<ll> c;
    Base(){clear();}
    void clear() {
        c.clear(); s = 0;
        for (int i = 0; i < B; i++) a[i] = 0;
    }
    void ins(ll x) {
        for (int i = B - 1; ~i; i--)
            if (x >> i & 1)
                if (a[i] /*x 和 a[i] 可以交换再做后面异或 */x ^= a[i];
                else {a[i] = x; break;}
    }
    void init() { // 化成上三角, 插入结束后就要执行
        for (int i = 0; i < B; i++)
            if (a[i])
                for (int j = i - 1; ~j; j--)
                    if (a[j] && (a[i] >> j & 1))
                        a[i] ^= a[j];
        for (int i = 0; i < B; i++)
            if (a[i])
```



```

        c.push_back(a[i]);
        s = (1ll << (c.size())) - 1; //非空子集可以表示出来的非 0 数字个数
    }
    ll kth(ll k) { //第 k 小, 查询之前要确保已经化成上三角
        ll res = 0;
        if (c.size() != n) -- k; //不等说明可以表示出 0 来
        if (s < k) return -1;
        for (int i = 0, sz = c.size(); i < sz; i++)
            if (k >> i & 1) res ^= c[i];
        return res;
    }
    ll getRank(ll x) { //查询 x 在所有可表示出的正数里的从小到大的 rank
        ll res = 0;
        for (int i = 0, sz = c.size(); i < sz; i++)
            if (x >> i & 1)
                res |= 1ll << i;
        return res;
    }
    void merge(const Base &b) { //合并
        for (int i = 0; i < B; i++)
            if (b.a[i])
                ins(b.a[i]);
    }
    Base intersection(const Base &b) { //求交
        Base na(*this), tmp(*this), res;
        ll cur, d;
        for (int i = 0; i < B; i++) if (b.a[i]) {
            cur = 0, d = b.a[i];
            for (int j = i; ~j; j--) if (d >> j & 1) {
                if (tmp.a[j]) {
                    d ^= tmp.a[j], cur ^= na.a[j];
                    if (d) continue;
                    res.a[i] = cur;
                }
                else tmp.a[j] = d, na.a[j] = cur;
                break;
            }
        }
        return res;
    }
}base;

```

5.1.3 实数线性基

```

int n, m; //向量个数和维度
struct Base {
    double a[N][N]; bool v[N];
    Base() {
        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
                a[i][j] = 0;
        for (int i = 0; i < N; i++)
            v[i] = 0;
    }
}

```

```

bool ins(double *x) { // 高斯消元
    for (int i = 0; i < m; i++) if (fabs(x[i]) > 1e-5) {
        if (v[i]) {
            double t = x[i] / a[i][i];
            for (int j = 0; j < m; j++)
                x[j] -= t * a[i][j];
        }
        else {
            v[i] = 1;
            for (int j = 0; j < m; j++)
                a[i][j] = x[j];
            return 1;
        }
    }
    return 0;
}
}base;

```

5.2 BSGS

*/*bsgs 简单原理: 求 $y^x \equiv z \pmod{p}$*
** 令 $x=km+b$, m 取 \sqrt{p}*
** 先把 b 的 m 个取值存起来, 然后枚举 k 即可*
** 拓展 $\gcd(y, p) \neq 1$*
** 先找到最小的 k 使得 $\gcd(y^k, p) = \gcd(y^{k+1}, p)$*
** 然后约去这个 \gcd 就得到互质了, 最后再加上 k 就行*
**/*

//限制 $\gcd(y, p) = 1$

```

ll bsgs(ll y, ll z, ll p) { //  $y^x \equiv z \pmod{p}$ 
    static map<ll, int> mp; // 可替换为 hash
    z %= p; mp.clear();
    if (z == 1) return 0;
    ll m = sqrt(p) + 1, s = z;
    for (int i = 0; i < m; i++) {
        mp[s] = i;
        s = s * y % p;
    }
    ll yy = qpow(y, m, p); s = 1;
    for (int i = 1; i <= m + 1; i++) {
        s = s * yy % p;
        if (mp.find(s) != mp.end())
            return i * m - mp[s];
    }
    return -1;
}

```

//拓展 bsgs, 不再限制 $\gcd(y, p) = 1$

```

ll exbsgs(ll y, ll z, ll p) {
    static map<ll, int> mp; // 可替换为 hash
    z %= p; mp.clear();
    if (z == 1) return 0;
    ll cnt = 0, t = 1;
    for (ll d = __gcd(y, p); d != 1; d = __gcd(y, p)) {
        if (z % d) return -1;
        cnt++, z /= d, p /= d, t = 1LL * t * y / d % p;
    }
}

```

```

        if(z == t) return cnt;
    }
    ll s = z, m = sqrt(p) + 1;
    for(int i = 0; i < m; i++){
        mp[s] = i;
        s = s * y % p;
    }
    ll x = qpow(y, m, p); s = t;
    for(int i = 1; i <= m; i++){
        if(mp.count(s = s * x % p))
            return i * m - mp[s] + cnt;
    }
    return -1;
}

```

5.3 勾股数分解

```

/*
*1. 判断  $x$  能否分解为  $a^2+b^2, a>0, b>0$ 
* solution: 对  $x$  质因数分解, 有  $4n+1$  的素因子就有分解, 否则没有
*2. 给  $x$  求不同分解数
* solution: 对  $x$  质因数分解, 假设所有形为  $4n+1$  的素因子的幂次组成数组  $[a_1, a_2, \dots, a_k]$ 
* 则有  $ans(x)=f([a_1, a_2, \dots, a_k])=f([a_1, a_2, \dots, a_{k-1}])*(2*a_k+1)+a_k$ 
* 复杂度为  $O(n^{1/4})$ 
*3. 求出所有不同分解组合
* 做法如下, 复杂度最坏为  $O(\sqrt{n} \cdot \log n)$ , 但其实达不到
* 可以默认为  $O(\sqrt{n} \cdot k)$ ,  $k$  介于  $[1, \log n]$ , 期望情况下大概是个不大于 10 的常数
*/
void calc(ll a, ll k, vector<pr> &v) {
    for (ll i = 0, j = int(sqrt(a)) + 5; i * i <= a; i++) {
        while (i * i + j * j > a) j--;
        if (i * i + j * j == a && check(i * j * 2, abs(j * j - i * i), a)) {
            v.push_back(pr(i * j * 2 * k, abs(j * j - i * i) * k));
            v.push_back(pr(abs(j * j - i * i) * k, i * j * 2 * k));
        }
    }
}

void calc(ll a, vector<pr> &v) { // 计算所有有序对  $(x, y)$  使得  $x^2+y^2=a^2$ 
    v.clear();
    for (ll k = 1; k * k <= a; k++)
        if (a % k == 0) {
            calc(a / k, k, v);
            calc(k, a / k, v);
        }
    v.push_back(pr(a, 0)), v.push_back(pr(0, a));
    sort(v.begin(), v.end()); v.resize(unique(v.begin(), v.end()) - v.begin());
}

```

5.4 CRT

```

void exgcd(ll a, ll b, ll &d, ll &x, ll &y) {
    if (!b) {
        d = a, x = 1, y = 0;
        return;
    }
}

```

```

    exgcd(b, a % b, d, y, x);
    y -= a / b * x;
}
ll crt(ll *m, ll *a, int n) { // n 个式子:  $y = mx + a$ , 下标从 1 开始
    ll A = a[1], M = m[1], d, x, y, m2;
    for (int i = 2; i <= n; i++) { //  $k_1 * m_1 - k_2 * m_2 = a_2 - a_1$ 
        exgcd(M, m[i], d, x, y);
        if ((a[i] - A) % d) return -1;
        m2 = M / d * m[i];
        x = (a[i] - A) / d * x % m[i];
        A = (A + x * M % m2) % m2;
        if (A < 0) A += m2; // 保证  $A \geq 0$ 
        M = m2;
    }
    return A; //  $y = Mx + A$ 
}

```

5.5 蔡勒公式

```

/*0-6 对应周日-周六 */
int calc(int yy, int m, int d) {
    if (m < 3) m += 12, yy--;
    int c = yy / 100, y = yy % 100;
    int w = (c / 4 - c * 2 + y + y / 4 + (13 * (m + 1)) / 5 + d - 1) % 7;
    return (w + 7) % 7;
}

```

5.6 拓展欧拉定理

```

/*problem: 求  $(c^{(c^{(...(c^x))})}) \% p$ , 中间嵌套了  $t$  次. 比如  $t=2$  就是求  $(c^{(c^x)}) \% p$ 
*solution: 使用拓展欧拉定理  $c^x \% p = c^{(x \% \phi(p) + \phi(p))} \% p$ , 前提  $x \geq \phi(p)$ 
* 我们不断展开发现当  $t \geq k$  时,  $t$  继续增大结果不变,  $k$  就是对  $p$  一直做  $p = \phi(p)$  直至  $p=1$  的次数
* 所以我们可以预处理出来对  $p$  一直求  $\phi$  的结果, 注意最后因为  $\phi(1)=1$  所以要多加一个
* 然后求结果就可以顺推即可, 该函数做一次复杂度  $O(\log^2 p)$ 
* 其中快速幂的  $\log$  可以通过  $\log(p) * \text{sqrt}(p)$  的预处理做成  $O(1)$  的
*/
ll exEuler(ll x, ll t) {
    if (x >= phi[t]) x = x % phi[t] + phi[t];
    for (int i = t; i > 0; i--) {
        x = qpow(c, x, phi[i - 1]);
        if (__gcd(c, x) != 1) x += phi[i - 1];
    }
    return x % phi[0];
}

int main() {
    for (phi[tot = 0] = p; phi[tot] != 1; )
        phi[tot + 1] = calcPhi(phi[tot]), tot++;
    phi[++tot] = 1;
}

```

5.7 素数判定 + 大整数质因数分解

```

/*miller_rabin 可以判定 ll 以内数字是否为素数
* 时间复杂度  $O(T \log n)$ , 错误率  $(1/4)^T$ ,  $T$  是测试组数

```

```

*
*pollard_rho 算法,  $O(n^{1/4})$  实现大整数的质因数分解
*/
namespace PollardRho {
    const int T = 20; //测试次数
    ll qmul(ll a, ll b, ll p) {
        ll c = 0;
        for (a %= p; b > 0; b >= 1) {
            if (b & 1) c += a;
            if (c >= p) c -= p;
            a <<= 1;
            if (a >= p) a -= p;
        }
        return c;
    }
    ll qpow(ll x, ll k, ll p) {
        ll res = 1;
        for (x %= p; k > 0; k >= 1, x = qmul(x, x, p))
            if (k & 1) res = qmul(res, x, p);
        return res;
    }
    bool check(ll a, ll n, ll x, ll t) {
        ll res = qpow(a, x, n), last = res;
        for (int i = 1; i <= t; i++) {
            res = qmul(res, res, n);
            if (res == 1 && last != 1 && last != n - 1) return 1;
            last = res;
        }
        if (res != 1) return 1;
        return 0;
    }
    //素数判定函数 (ret = 0) -> prime
    bool millerRabin(ll n) {
        if (n < 2) return 1;
        ll x = n - 1, t = 0;
        while (!(x & 1)) x >>= 1, t++;
        bool flag = 1;
        if (t >= 1 && (x & 1)) {
            for (int k = 0; k < T; k++) {
                ll a = rand() % (n - 1) + 1;
                if (check(a, n, x, t)) {
                    flag = 1;
                    break;
                }
            }
            flag = 0;
        }
        if (!flag || n == 2) return 0;
        return 1;
    }
    ll pollardRho(ll x, ll c) {
        ll i = 1, x0 = rand() % x, y = x0, k = 2;
        while (1) {
            i++;

```

```

        x0 = qmul(x0, x0, x) + c % x;
        ll d = abs(__gcd(y - x0, x));
        if (d != 1 && d != x) return d;
        if (y == x0) return x;
        if (i == k) y = x0, k <<= 1;
    }
}
void findFac(ll n, ll *f) {
    if (!millerRabin(n)) {
        f[++ f[0]] = n;
        return;
    }
    ll p = n;
    while (p >= n) p = pollardRho(p, rand() % (n - 1) + 1);
    findFac(p, f), findFac(n / p, f);
}
//质因数分解函数, 因子放在 f 数组, 有重复且无序
void getFac(ll n, ll *f) {
    f[0] = 0;
    if (n <= 1) return;
    findFac(n, f);
}
}
int main() {
    srand(time(NULL));
}

```

5.8 删除 k 个数使 gcd 最大

/* 原题: 删除 k 个数使得剩下 $n-k$ 个数的 gcd 最大, $k \leq n/2$
 * *solution*: 随机, 随机选择一个数 $a[x]$ 假设他在最优解里尝试更新答案即可
 * 须知: ll 范围内数字的因子个数不超过 $20w$ 个, 于是对选择的数字分解因数
 * 然后考虑以每个因数作为答案是否可行即可, 暴力复杂度 $O(20w * n)$
 * 因为有重复计算, 所以考虑 $a[i]$ 只对 $gcd(a[i], a[x])$ 做贡献
 * 然后求前缀和即可, 具体实现参考代码
 * 单次复杂度为 $O(\max(\text{因子个数} * \text{不同素因子个数}, n \log n))$ 约为 $O(n \log n)$
 * 注意 N 取值为 $\max(n, 20w)$
 */

```

map <ll, int> p;
ll n, k, a[N], b[N], c[N];
ll sum, val[N], fac[N], cnt[N];
void dfs(int i, ll x, int id){
    p[x] = id, val[id] = x;
    if (i > b[0]) return;
    for (int j = 0; j < c[i] + 1; j++) {
        dfs(i + 1, x, id);
        x *= b[i], id += fac[i];
    }
}
}
ll solve(ll x) {
    PollardRho::getFac(x, b);
    sort (b + 1, b + b[0] + 1);
    b[0] = unique(b + 1, b + b[0] + 1) - b - 1;
    for (ll y = x, i = 1; i <= b[0]; i++) {

```

```
        c[i] = 0;
        while (y % b[i] == 0) c[i] ++, y /= b[i];
    }
    sum = 1;
    for (int i = b[0]; i > 0; i --)
        fac[i] = sum, sum *= c[i] + 1;
    p.clear(), dfs(1, 1, 0);
    for (int i = 0; i < sum; i ++) cnt[i] = 0;
    for (int i = 1; i <= n; i ++) cnt[p[__gcd(x, a[i])]] ++;
    for (int i = 1; i <= b[0]; i ++)
        for (int j = sum - 1; j >= 0; j --)
            if (j / fac[i] % (c[i] + 1) != c[i])
                cnt[j] += cnt[j + fac[i]];
    ll res = 0;
    for (int i = 0; i < sum; i ++) if (cnt[i] >= n - k) res = max(res, val[i]);
    return res;
}
```

6 字符串

6.1 KMP

```
void kmp(int n, char *a, int m, char *b) {  
    //长度为 m 的 b 中找 a, 下标从 0 开始, 得到的是匹配成功的末尾位置  
    static int nxt[N], i, j;  
    for (nxt[0] = j = -1, i = 1; i < n; nxt[i++] = j) {  
        while (~j && a[j + 1] != a[i]) j = nxt[j];  
        if (a[j + 1] == a[i]) j++;  
    }  
    for (j = -1, i = 0; i < m; i++) {  
        while (~j && a[j + 1] != b[i]) j = nxt[j];  
        if (a[j + 1] == b[i]) j++;  
        if (j == n - 1) {  
            printf("%d ", i);  
            j = nxt[j];  
        }  
    }  
}
```


7 其他

7.1 cdq 套路

cdq 经常解决的问题:

1. 三维偏序

对一维排序, 然后 cdq 处理, 处理到某个区间时, 我们拿出前一半的插入和后一半的查询
 插入是在 (x, y) 插入一个点, 查询是查询一个二维前缀和
 直接把所有点排序然后树状数组做即可

注意的问题:

- (1) 对于询问要乘上该询问对答案的贡献系数!
- (2) 内部排序的第一关键字 x , 第二 y , 第三是修改要放在查询前面!

2. 配合按秩合并的并查集维护图的联通性

具体可参考板子

3. 如果第一个问题中的点可删除怎么办

假设某个点 i 出现和消失时间分别为 $st[i]$ 和 $ed[i]$

那么对于当前处理的操作序列, 对于前一半的插入操作 i (即某个点出现了)

如果 $ed[i] > r$, 那么就说明在后一半的时间中始终存活, 留下

对于留下的插入操作, 考虑对后一半中的询问操作做贡献

然后对于后一半的删除操作 i

如果 $st[i] < l$, 说明它在前一半的时间中适存活, 留下

然后对于留下的这些点, 考虑对前一半的所有询问做贡献

然后递归下去即可

7.2 最小表示

/* 最小表示: $a[0 \dots n-1], a[1 \dots n-1, 0], a[2 \dots n-1, 0, 1]$

* n 个序列中字典序最小的表示, 即为该序列的最小表示

* 该函数返回的是某个最小表示起始位置的下标

*/

```
int minrep(int n) {
    int i = 0, j = 1, k = 0, t;
    while (i < n && j < n && k < n)
        if (t = a[(i + k) % n] - a[(j + k) % n]) {
            if (t > 0) i += k + 1;
            else      j += k + 1;
            if (i == j) j++;
            k = 0;
        }
        else
            k++;
    return i < j ? i : j;
}
```

7.3 十进制快速幂

// $x_i = x_{i-1} * a + x_{i-2} * b$, 求 x_n , n 贼大那种

const int K = 2;

ll mod;

struct matrix {

ll c[K][K];

matrix operator ^ (const char *s) const {

matrix res, x = *this, y = x; res.clear();

```

        for (int i = 0; i < K; i++) res.c[i][i] = 1;
        int len = strlen(s + 1);
        for (int i = len; i >= 1; i--) {
            for (int j = 1; j < 10; j++) {
                if (s[i] == '0' + j) res = res * y;
                y = y * x;
            }
            x = y;
        } return res;
    }
}a, b;
char s[N];
int main(){
    scanf("%lld %lld %lld %lld %s %lld",
        &a.c[0][0], &a.c[0][1],
        &b.c[1][1], &b.c[0][1],
        s + 1, &mod);
    b.c[1][0] = 1;
    printf("%lld\n", (a * (b ^ s)).c[0][0]);
    return 0;
}

```

7.4 数字哈希

```

namespace my_hash {
    const int N = (1 << 19) - 1; //散列大小，一定要取  $2^k-1$ ，不超内存的情况下，N越大碰撞
    ↪ 越少
    struct E {
        int v;
        E *nxt;
    }*g[N + 1], pool[N], *cur = pool, *p;
    int vis[N + 1], T;
    void ins(int v) {
        int u = v & N;
        if (vis[u] < T) vis[u] = T, g[u] = NULL;
        for (p = g[u]; p; p = p -> nxt) if (p -> v == v) return;
        p = cur++; p -> v = v; p -> nxt = g[u]; g[u] = p;
    }
    int ask(int v) {
        int u = v & N;
        if (vis[u] < T) return 0;
        for (p = g[u]; p; p = p -> nxt) if (p -> v == v) return 1;
        return 0;
    }
    void init() {T++; cur = pool;} //应对多组数据使用
}

```

7.5 海岛分金币

7.5.1 海岛分金币 1

/*
非朴素模型，有额外条件：
每个人做决定时如果有多种方案可以使自己获得最大收益

那么他会让决策顺序靠前的人获得的收益尽可能的大！

solution:

贪心模拟

*/

#define v first

#define id second

typedef pair<int, int> pr;

const int N = 1010;

int a[N][N];

pr b[N];

int n, m;

int main() {

cin >> n >> m;

a[1][1] = m;

for (int i = 2; i <= n; i++) {

for (int j = 1; j < i; j++)

b[j] = pr(a[i - 1][j], j);

sort (b + 1, b + i, [&](pr x, pr y){return x.v != y.v ? (x.v < y.v) : (x.id >

↪ y.id);});

//按照是否容易满足来排序，因为容易满足的人消耗掉的金币比较少，也就使得当前的人获利最大

int s = m, nd = (i - 1) / 2;

for (int j = 1; j < i && nd; j++) {

nd--;

s -= (a[i][b[j].id] = a[i - 1][b[j].id] + 1);

}

if (s < 0) {

for (int j = 1; j < i; j++)

a[i][j] = a[i - 1][j];

a[i][i] = -1;

}

else {

a[i][i] = s;

}

}

for (int i = n; i; i--)

printf("%d ", a[n][i]);

return 0;

}

7.5.2 海岛分金币 2

/*

海盗分金币朴素模型：

n 个海盗分 m 个金币，依次做决策，如果不少于半数的人同意则方案通过，否则当前做决策的人会被淘汰

↪ (收益视为 -1)，由下一人做出决策

如果一个海盗有多种方案均为最大收益，那么他会希望淘汰的人越多越好

求出第 x 个做决策的海盗的最大可能受益和最小可能收益

*/

struct node {

int min_v, max_v;

node():min_v(0), max_v(0) {}

node(int min_v, int max_v):min_v(min_v), max_v(max_v) {}

};

node ask(int n, int m, int x) { // n 个人分 m 个金币，第 x 个做决策的人最少/最多分到多少个金币

```

int y = n + 1 - x;
if (n >= (m + 2) * 2) {
    int a = (m + 1) * 2, b = 2, c = 4;
    //前 a 个为 [0,1], 后 b 个为 [0,0], 将持续 c 个
    while (a + b + c <= n) {
        a += b;
        b *= 2;
        c *= 2;
    }
    if (y <= a) return node(0, 1);
    else if (y <= a + b) return node(0, 0);
    else return node(-1, -1);
}
else if (n == m * 2 + 3) {
    if (x == 1) return node(-1, -1);
    else if (y <= m * 2 && y % 2 == 1 || x == 2) return node(0, 0);
    else return node(0, 1);
}
else if (n == m * 2 + 2) {
    if (y <= m * 2 && y % 2 == 1 || x == 1) return node(0, 0);
    else return node(0, 1);
}
else if (n == m * 2 + 1) {
    if (y <= m * 2 && y % 2 == 1) return node(1, 1);
    else return node(0, 0);
}
else {
    if (x & 1) {
        if (x != 1) return node(1, 1);
        else return node(m - (n - 1) / 2, m - (n - 1) / 2);
    }
    else return node(0, 0);
}
}

int main() {
    ios::sync_with_stdio(false);
    int x, n, m, k; node y;
    cin >> n >> m >> k;
    while (k --) {
        cin >> x;
        y = ask(n, m, x);
        printf("%d %d\n", y.min_v, y.max_v);
    }
    return 0;
}

/*
m = 5

1 5
2 0 5
3 1 0 4
4 0 1 0 4

```

```

5 1 0 1 0 3
6 0 1 0 1 0 3
7 1 0 1 0 1 0 2
8 0 1 0 1 0 1 0 2
9 1 0 1 0 1 0 1 0 1
10 0 1 0 1 0 1 0 1 0 1
11 1 0 1 0 1 0 1 0 1 0 0
12 0 _ 0 _ 0 _ 0 _ 0 _ 0
13 0 _ 0 _ 0 _ 0 _ 0 _ 0 -1
14 _ _ _ _ _ _ _ _ _ 0 0
15 _ _ _ _ _ _ _ _ _ 0 0 -1
16 _ _ _ _ _ _ _ _ _ 0 0 -1 -1
17 _ _ _ _ _ _ _ _ _ 0 0 -1 -1 -1
18 _ _ _ _ _ _ _ _ _ 0 0 0 0
19 _ _ _ _ _ _ _ _ _ 0 0 0 0 -1
20 _ _ _ _ _ _ _ _ _ 0 0 0 0 -1 -1
21 _ _ _ _ _ _ _ _ _ 0 0 0 0 -1 -1 -1
22 _ _ _ _ _ _ _ _ _ 0 0 0 0 -1 -1 -1 -1
23 _ _ _ _ _ _ _ _ _ 0 0 0 0 -1 -1 -1 -1 -1
24 _ _ _ _ _ _ _ _ _ 0 0 0 0 -1 -1 -1 -1 -1 -1
25 _ _ _ _ _ _ _ _ _ 0 0 0 0 -1 -1 -1 -1 -1 -1 -1
26 _ _ _ _ _ _ _ _ _ 0 0 0 0 0 0 0 0
*/

```

7.6 根号枚举

```

for (int i = 1, last; i <= n; i = last + 1) {
    last = n / (n / i);
    //当前枚举区间为 [i, last]
}

```

7.7 读入输出外挂

```

namespace IO {//only for int!!!
    static const int SIZE = 1 << 20;
    inline int get_char() {
        static char *S, *T = S, buf[SIZE];
        if (S == T) {
            T = fread(buf, 1, SIZE, stdin) + (S = buf);
            if (S == T) return -1;
        }
        return *S++;
    }
    inline void in(int &x) {//for int
        static int ch;
        while (ch = get_char(), ch < 48); x = ch ^ 48;
        while (ch = get_char(), ch > 47) x = x * 10 + (ch ^ 48);
    }
    char buffer[SIZE];
    char *s = buffer;
    void flush() {//最后需要 flush!!
        fwrite(buffer, 1, s - buffer, stdout);
        s = buffer;
        fflush(stdout);
    }
}

```

```

}
inline void print(const char ch) {
    if(s - buffer > SIZE - 2) flush();
    *s++ = ch;
}
inline void print(char *str) {//for string
    while(*str != 0)
        print(char(*str ++));
}
inline void print(int x) {
    static char buf[25];
    static char *p = buf;
    if (x < 0) print('-'), x = -x;
    if (x == 0) print('0');
    while(x) *(++ p) = x % 10, x /= 10;
    while(p != buf) print(char(*(p --) ^ 48));
}
};

```

7.8 给定小数化成分数

本题答案的分母不超过 $1e9$, 给定小数的小数点位为 18 位

单次 $O(\log 2n)$

```

inf, inff = 10 ** 9, 10 ** 18
for i in range(int(input())):
    n = int(input()[2:])
    if n == 0: print('0 1')
    else:
        lp, lq, rp, rq = 0, 1, 1, 1
        while max(lq, rq) <= inf:
            mp, mq = lp + rp, lq + rq
            if mp * inff <= mq * n:
                l, r, mid, cnt = 1, (inf - lq) // rq + 1, -1, -1
                while l <= r:
                    mid = l + r >> 1
                    if (lp + rp * mid) * inff <= (lq + rq * mid) * n:
                        cnt, l = mid, mid + 1
                    else:
                        r = mid - 1
                lp, lq = lp + rp * cnt, lq + rq * cnt
            else:
                l, r, mid, cnt = 1, (inf - rq) // lq + 1, -1, -1
                while l <= r:
                    mid = l + r >> 1
                    if (rp + lp * mid) * inff > (rq + lq * mid) * n:
                        cnt, l = mid, mid + 1
                    else:
                        r = mid - 1
                rp, rq = rp + lp * cnt, rq + lq * cnt
            if lq <= inf: print(lp, lq)
            else: print(rp, rq)

```

8 计算几何

8.1 全家桶

/* 旋转卡壳:

*1. 最小面积矩形覆盖/最小周长矩形覆盖, 做法相同

*2. 找凸包上距离最远两个点, 可以找距离某条边最远的点, 然后用两个端点到这个点的距离更新答案即

↪ 可

* 圆的反演:

* $OA \cdot OA' = R^2$, O 为反演中心, R 为反演半径, A 为原图形的点, A' 为反形上的对应点

* 过反演中心的圆的反形是一条不过反演中心的直线, 反之亦然

* 不过反演中心的圆反形也为不过反演中心的圆, 且反演中心为两个圆的位似中心

* 推论: 反演中心不在两圆上, 则两圆相对关系不变, 在两个圆上就平行线

* 在一个圆上就一个圆一条线。原来两圆相切就是相切的圆和直线

*/

```
const double eps = 1e-8;
const double pi = acos(-1.0);
//atan2(y,x) 反正切, 返回 (-pi, pi]
int sgn(double x) {return fabs(x) < eps ? 0 : (x < 0 ? -1 : 1);} //判断数字符号
struct point {
    double x, y;
    point():x(0), y(0) {}
    point(double x, double y):x(x), y(y) {}
    bool operator < (const point &a) const {return sgn(y - a.y) == 0 ? x < a.x : y <
        ↪ a.y;}
    point operator + (const point &a) const {return point(x + a.x, y + a.y);}
    point operator - (const point &a) const {return point(x - a.x, y - a.y);}
    point operator * (const double &a) const {return point(x * a, y * a);}
    point operator / (const double &a) const {return point(x / a, y / a);}
};
struct line:public vector<point> {
    line(){}
    line(point a, point b) {push_back(a), push_back(b);}
};
double sqr(double x) {return x * x;}
double abs(point a) {return sqrt(sqr(a.x) + sqr(a.y));} //向量长度
point vec(point a) {return a / abs(a);} //向量 a 对应的单位向量
point vec(const line &a) {return a[1] - a[0];} //直线化为向量
double det(point a, point b) {return a.x * b.y - a.y * b.x;} //叉积
double dot(point a, point b) {return a.x * b.x + a.y * b.y;} //点积
point vertical(point a, point b) {return point(a.y - b.y, -a.x + b.x);} //与 ab 向量垂直的
↪ 向量
point rotate(point u, double a) {
    return point(u.x * cos(a) - u.y * sin(a), u.y * cos(a) + u.x * sin(a));
} //坐标逆时针旋转 a 度
point convertPoint(point u, point v, point c) {
    return point(det(c, v) / det(u, v), det(c, u) / det(v, u));
} //uv 不平行时, 存在 a*u+b*v=c, 返回 (a,b)
int cross(point s, point a, point b) {
    point u = a - s, v = b - s; double res = det(u, v);
    if (res > 0) return 1; //在线段左方
    if (res < 0) return -1; //在线段右方
    if (dot(u, v) <= 0) return 0; //在线段上
```

```

    if (abs(u) > abs(v)) return 2; //在 a 到 b 延长线上
    return -2; //在 b 到 a 的延长线上
} //判断点 s 与线段 ab 的位置关系
bool isInterSS(const line &u, const line &v) {
    if (sgn(det(vec(u), vec(v))) == 0) return 0; //平行
    if (sgn(cross(u[0], u[1], v[0]) * cross(u[0], u[1], v[1]) <= 0) &&
        sgn(cross(v[0], v[1], u[0]) * cross(v[0], v[1], u[1]) <= 0))
        return 1; //相交
    return 0;
} //判断线段交
point interLL(const line &u, const line &v) {
    double a = det(vec(u), vec(v));
    double b = det(vec(u), u[1] - v[0]);
    if (sgn(a) == 0) return v[0];
    return v[0] + vec(v) * b / a;
} //求直线交点
point crossSP(const line &v, const point &u) {
    return interLL(v, line(u, u + vertical(v[0], v[1])));
} //过 u 向直线 v 做垂线返回垂足
struct convexHull {
    vector<point> p; int n;
    convexHull():n(0) {p.clear();}
    convexHull(vector<point> &q):n(0) {
        int m = q.size(), cnt = 0; sort (q.begin(), q.end());
        for (int i = 1; i < m; i++)
            if (sgn(q[i].x - q[cnt].x) != 0 || sgn(q[i].y - q[cnt].y) != 0)
                q[++cnt] = q[i];
        q.resize(m = cnt + 1), p.resize(m + 1);
        for (int i = 1; i < m; i++) if (q[i] < q[0]) swap(q[i], q[0]);
        sort (q.begin() + 1, q.begin() + m, [&](point a, point b){
            double t = det((a - q[0]), (b - q[0]));
            if (sgn(t) == 0) return sgn(abs(q[0] - a) - abs(q[0] - b)) < 0;
            return t > 0;
        }); //以 p[0] 为原点的极角排序
        p[n = 1] = q[0];
        for (int i = 1; i < m; i++) {
            while (n > 1 && det((p[n] - p[n - 1]), (q[i] - p[n])) < eps) n
                --;
            p[++n] = q[i];
        }
        p[0] = p[n], p.resize(n + 1);
    } //得到 n 个点的凸包, 且 p[0]=p[n]
    double getArea() const { //如果求周长需要特判 n=2
        double res = 0;
        for (int i = 2; i < n; i++)
            res += abs(det(p[i] - p[0], p[i - 1] - p[0]));
        return res / 2;
    }
    void getRec() const { //最小矩形覆盖
        double ans = 1e20, L, R, D, H;
        int l = 1, r = 1, t = 1;
        point rc[4];
        for (int i = 0; i < n; i++) { //旋转卡壳
            D = abs(p[i] - p[i + 1]); //以 q[i] 和 q[i+1] 所在直线为底边

```



```

while (sgn(det((p[i + 1] - p[i]), (p[t + 1] - p[i])) -
    det((p[i + 1] - p[i]), (p[t] - p[i]))) > -1) t = (t + 1)
    ↪ % n;
while (sgn(dot((p[i + 1] - p[i]), (p[r + 1] - p[i])) - dot((p[i +
    ↪ 1]
        - p[i]), (p[r] - p[i]))) > -1) r = (r + 1) % n;
if (i == 0) l = r;
while (sgn(dot((p[i + 1] - p[i]), (p[l + 1] - p[i])) - dot((p[i +
    ↪ 1]
        - p[i]), (p[l] - p[i]))) < 1) l = (l + 1) % n;
L = fabs(dot((p[i + 1] - p[i]), (p[l] - p[i])) / D); //直线向左延伸
    ↪ 长度
R = fabs(dot((p[i + 1] - p[i]), (p[r] - p[i])) / D); //向右延伸长度
H = fabs(det((p[i + 1] - p[i]), (p[t] - p[i])) / D); //t 为与底边
    ↪ 垂直距离最大的点
double tmp = (R + L) * H;
if (tmp < ans) {
    ans = tmp;
    rc[0] = p[i] + (p[i + 1] - p[i]) * (R / D); //右下
    rc[1] = rc[0] + vec(vertical(p[i], p[i + 1])) * H; //右上
    rc[2] = rc[1] - (rc[0] - p[i]) * ((R + L)
        / abs(p[i] - rc[0])); //左上
    rc[3] = rc[2] - (rc[1] - rc[0]);
}
} //具体小数位数看题目要求哦
printf("%.5f\n", ans); int fir = 0;
for (int i = 0; i < 4; i++) {
    if (rc[i] < rc[fir]) fir = i;
    if (sgn(rc[i].x) == 0) rc[i].x = 0;
    if (sgn(rc[i].y) == 0) rc[i].y = 0;
} //逆时针输出, 且第一个是 y 坐标最小的点
for (int i = 0; i < 4; i++)
    printf("%.5f %.5f\n", rc[(fir + i) % 4].x, rc[(fir + i) % 4].y);
}
};

bool isInConvexHull(point p, const convexHull &h) { //点是否在凸包内或凸包上
    double s = h.getArea();
    for (int i = 0; i < h.n; i++)
        s -= abs(det(p - h.p[i + 1], p - h.p[i])) / 2;
    return sgn(s) == 0;
} //要求 n>2, 线段上就判断叉积为 0 且坐标在范围里面即可
point getCC(point a, point b, point c) { //三点确定圆心, Center of Circle
    point l1 = (a + b) / 2, l2 = (a + c) / 2;
    return interLL(line(l1, l1 + vertical(a, b)), line(l2, l2 + vertical(a, c)));
}

struct circle {
    point o; double r;
    circle():o(), r(0) {}
    circle(point o, double r):o(o), r(r) {}
    circle(double x, double y, double r):o(x, y), r(r) {}
    circle(int n, vector<point> &p):o(), r(0) { //最小圆覆盖 O(n)
        random_shuffle(p.begin(), p.end());
        for (int i = 0; i < n; i++)

```

```

        if (dot(p[i] - o, p[i] - o) > r) {
            o = p[i], r = 0;
            for (int j = 0; j < i; j++)
                if (dot(p[j] - o, p[j] - o) > r) {
                    o = (p[i] + p[j]) / 2, r = dot(p[j] - o,
                        ↪ p[j] - o);
                    for (int k = 0; k < j; k++)
                        if (dot(p[k] - o, p[k] - o) > r)
                            o = getCC(p[i], p[j],
                                ↪ p[k]), r = dot(p[k] -
                                ↪ o, p[k] - o);
                }
            }
        r = sqrt(r);
    }
};

```

8.2 单位圆覆盖

```

struct node {
    double ang; int in;
    bool operator < (const node &a) const {
        return sgn(ang - a.ang) == 0 ? in > a.in : sgn(ang - a.ang) < 0;
    }
} a[N * N];
void maxCircleCover(int n, vector<point> &p) { //单位圆覆盖  $O(n^2 \log n)$ 
    int ans = 0;
    for (int i = 0; i < n; i++) {
        int c = 0, cnt = 0;
        for (int j = 0; j < n; j++) {
            if (abs(p[i] - p[j]) > 2.0) continue;
            double ang1 = atan2(p[j].y - p[i].y, p[j].x - p[i].x);
            double ang2 = acos(abs(p[i] - p[j]) / 2);
            a[c++] = (node){ang1 - ang2, 1};
            a[c++] = (node){ang1 + ang2, -1};
        }
        sort(a, a + c);
        for (int j = 0; j < c; j++)
            cnt += a[j].in, ans = max(ans, cnt);
    }
    printf("%d\n", ans);
}

```