by black_trees

Contents

1. Tools

1.1 vimrc

```
syntax on
color xterm16

set ts=4 sts=4 sw=4 ai
set wildmenu nocompatible nu rnu ruler mouse=a
set timeoutlen=666 ttimeoutlen=0 backup swapfile undofile
set ar acd backspace=indent,eol,start foldmethod=marker
set encoding=utf-8

nmap F :e ./<CR>
nmap <leader>n :tabnew<CR>
nmap <leader>tc :term<CR>

inoremap [ []<Esc>i
inoremap {<CR> {}<ESC>i<CR><ESC>O
inoremap ( ()<Esc>i
inoremap ' ''<Esc>i
inoremap " ""<Esc>i
```

1.2 对拍

```
g++ gen.cpp -o gen -std=c++17 -Wall -O2
g++ ans.cpp -o ans -std=c++17 -Wall -O2
g++ usr.cpp -o ans -std=c++17 -Wall -O2
while true; do
    ./gen > 1
    ./ans < 1 > 2
    ./usr < 1 > 3
    diff 2 3
    if [ $? -ne 0 ] ; then break; fi
done
```

1.3 注意事项

- Linux下开大栈空间:如果 ulimit -a 是 unlimited,那么写入 ulimit -s 65536; ulimit -m 1048576 即可。

2. Ds

2.1 链表

```
struct node {
  int value;
  node *next, *prev;
}, head, tail;
void init() {
    head = new node(), tail = new node();
    head -> next = tail, tail -> prev = head;
}
void insert(node *p, int val) {
    node *q; q = new node();
    q -> val = val, p -> next -> prev = q;
    q -> next = p -> next, q -> prev = p, p -> next = q;
}
void remove(node *p) {
    p -> prev -> next = p-> next,
  p -> next -> prev = p-> pre; delete p;
}
```

2.2 哈希表

```
const int si = 1e5 + 10;
int n, a[si], tot = 0;
int head[si], val[si], cnt[si], Next[si];

int H(int x) { return (x % p) + 1; }
bool insert(int x) {
    bool exist = false;
    int u = H(x);
    for(int i = head[u]; ~i; i = Next[i]) {
        if(val[i] == x) {
            cnt[i]++, exist = true;
            break;
        }
    }
    if(exist) return true;
    ++tot, Next[tot] = head[u], val[tot] = x, cnt[tot] = 1, head[u] = tot;
    return false;
}
int query(int x) {
    int u = H(x);
    for(int i = head[u]; ~i; i = Next[i])
        if(val[i] == x) return cnt[i];
    return 0;
}
```

2.3 并查集

```
int root(int x) {
    if(pa[x] != x) pa[x] = root(pa[x]);
    return pa[x];
}
void Merge(int x, int y) {
    int rx = root(x), ry = rooot(y);
    if(rx == ry) return;
    if(siz[rx] < siz[ry])
        pa[rx] = ry, siz[rx] += siz[ry];
    else pa[ry] = rx, siz[ry] += siz[rx];
}
// remember to init!
```

2.4 树状数组

```
int lowbit(int x) { return x & -x; }
class Fenwick {
  private:
    int t[si], V;
  public:
    void init(int n) { V = n + 1; memset(t, 0, sizeof t); }
    void add(int x, int v) { while(x <= V) t[x] += v, x += lowbit(x); }
    int que(int x) { int ret = 0; while(x > 0) ret += t[x], x -= lowbit(x); return ret; }
} tr;
```

2.5 线段树

```
// {Lazytag} = {+}
class Segment_Tree {
    private :
        struct Node {
            int l, r;
            i64 dat, tag;
        }t[si << 2];
        inline void pushup(int p) {
            t[p].dat = t[p << 1].dat + t[p << 1 | 1].dat;
        }
        inline void pushdown(int p) {
            if(!t[p].tag) return;
            t[p << 1].dat += 1ll * t[p].tag * (t[p << 1].r - t[p << 1].l + 1);
            t[p << 1 | 1].dat += 1ll * t[p].tag * (t[p << 1 | 1].r - t[p << 1 | 1].l + 1);
            t[p << 1].tag += t[p].tag, t[p << 1 | 1].tag += t[p].tag, t[p].tag = 0;
        }
    public :
        void build(int p, int l, int r) {
            t[p].l = l, t[p].r = r, t[p].tag = 0;
            if(l == r) {
                t[p].dat = a[l];
                return;
            }
            int mid = (l + r) >> 1;
            build(p << 1, l, mid), build(p << 1 | 1, mid + 1, r);
            pushup(p); return;
        }
        void update(int p, int l, int r, int v) {
            if(l <= t[p].l && t[p].r <= r) {
                t[p].dat += v * (t[p].r - t[p].l + 1);
                t[p].tag += v; return;
            }
            pushdown(p); // 没到可以直接返回的时候，马上要递归子树了，也要 pushdown。
            int mid = (t[p].l + t[p].r) >> 1;
            if(l <= mid)
                update(p << 1, l, r, v);
            if(r > mid)
                update(p << 1 | 1, l, r, v);
            pushup(p); return;
```

```
        }
        i64 query(int p, int l, int r) {
            i64 res = 0;
            if(l <= t[p].l && t[p].r <= r)
                return t[p].dat;
            pushdown(p); // 查询要查值，需要子树信息，必然要pushdown。
            int mid = (t[p].l + t[p].r) >> 1;
            if(l <= mid)
                res += query(p << 1, l, r);
            if(r > mid)
                res += query(p << 1 | 1, l, r);
            return res;
        }
};

// {Lazytag} = {+, *}
class Segment_Tree {
    private :
        struct Node {
            int l, r;
            i64 dat, add, mul;
        }t[si << 2];
        inline void pushup(int p) {
            t[p].dat = (t[p << 1].dat + t[p << 1 | 1].dat) % mod;
        }
        inline void pushdown(int p) {
            if(!t[p].add && t[p].mul == 1) return;
            t[p << 1].dat = (t[p << 1].dat * t[p].mul + t[p].add * (t[p << 1].r - t[p << 1].l + 1)) % mod   ;
            t[p << 1 | 1].dat = (t[p << 1 | 1].dat * t[p].mul + t[p].add * (t[p << 1 | 1].r - t[p << 1 | 1].l + 1)) % mod;

            t[p << 1].mul = (t[p << 1].mul * t[p].mul) % mod;
            t[p << 1 | 1].mul = (t[p << 1 | 1].mul * t[p].mul) % mod;

            t[p << 1].add = (t[p << 1].add * t[p].mul + t[p].add) % mod;
            t[p << 1 | 1].add = (t[p << 1 | 1].add * t[p].mul + t[p].add) % mod;

            t[p].add = 0, t[p].mul = 1;
        }
    public :
        void build(int p, int l, int r) {
            t[p].l = l, t[p].r = r, t[p].mul = 1ll, t[p].add = 0ll;
            if(l == r) {
                t[p].dat = a[l] % mod;
                return;
            }
            int mid = (l + r) >> 1;
            build(p << 1, l, mid), build(p << 1 | 1, mid + 1, r);
            pushup(p); return;
        }
        void update_add(int p, int l, int r, i64 v) {
            if(l <= t[p].l && t[p].r <= r) {
                t[p].add = (t[p].add + v) % mod;
                t[p].dat = (t[p].dat + v * (t[p].r - t[p].l + 1)) % mod;
                return;
            }
            pushdown(p);
            int mid = (t[p].l + t[p].r) >> 1;
            if(l <= mid)
                update_add(p << 1, l, r, v);
            if(r > mid)
                update_add(p << 1 | 1, l, r, v);
            pushup(p); return;
        }
        void update_mul(int p, int l, int r, i64 v) {
            if(l <= t[p].l && t[p].r <= r) {
                t[p].add = (t[p].add * v) % mod;
                t[p].mul = (t[p].mul * v) % mod;
                t[p].dat = (t[p].dat * v) % mod;
                return;
            }
            pushdown(p);
            int mid = (t[p].l + t[p].r) >> 1;
            if(l <= mid)
                update_mul(p << 1, l, r, v);
            if(r > mid)
                update_mul(p << 1 | 1, l, r, v);
            pushup(p); return;
        }
        i64 query(int p, int l, int r) {
            i64 res = 0ll;
            if(l <= t[p].l && t[p].r <= r)
                return t[p].dat % mod;
            pushdown(p);
            int mid = (t[p].l + t[p].r) >> 1;
            if(l <= mid)
                res = (res + query(p << 1, l, r)) % mod;
            if(r > mid)
                res = (res + query(p << 1 | 1, l, r)) % mod;
            return res;
        }
};

Segment_Tree tr;
// 不要到主函数里定义，容易爆栈。

// Merge
int merge(int p, int q, int l, int r) {
    if(!p) return q;
    if(!q) return p;
    if(l == r){
        t[p].mx += t[q].mx;
        return p;
    }
    int mid = (l + r) >> 1;
    t[p].ls = merge(t[p].ls, t[q].ls, l, mid);
    t[p].rs = merge(t[p].rs, t[q].rs, mid + 1, r);
    pushup(p); return p;
}

// SweepLine
void change(int p, int l, int r, int v) {
    int nl = t[p].l, nr = t[p].r;
    if(l <= nl && nr <= r) {
        t[p].cnt += v;
        if(t[p].cnt == 0)
            t[p].len = (nl == nr) ? 0 : t[p << 1].len + t[p << 1 | 1].len;
            // 虽然当前区间直接被覆盖的次数等于 0 了，但还是要考虑下面的子树，因为它们有可能没被修改完。
        else t[p].len = raw[nr + 1] - raw[nl];
        return;
    }
    int mid = (nl + nr) >> 1;
    if(l <= mid) change(p << 1, l, r, v);
    if(r > mid) change(p << 1 | 1, l, r, v);
    if(t[p].cnt > 0) t[p].len = raw[nr + 1] - raw[nl];
    else t[p].len = t[p << 1].len + t[p << 1 | 1].len;
}

```

2.6 轻重链剖分

```
// 处理重儿子，父亲，深度，子树大小
void dfs1(int u, int fa) {
    int kot = 0;
    hson[u] = -1, siz[u] = 1;
    fat[u] = fa, dep[u] = dep[fa] + 1;
    for(int i = head[u]; ~i; i = e[i].Next) {
        int v = e[i].ver;
        if(v == fa) continue;
        dfs1(v, u), siz[u] += siz[v];
        if(siz[v] > kot)
            kot = siz[v], hson[u] = v;
    }
}
// 处理 dfn,rnk, 并进行重链剖分。
void dfs2(int u, int tp) {
    top[u] = tp, dfn[u] = ++tim, rnk[tim] = u;
    if(hson[u] == -1) return;
    dfs2(hson[u], tp);
    // 先 dfs 重儿子，保证重链上 dfn 连续，维持重链的性质
    for(int i = head[u]; ~i; i = e[i].Next) {
```

```
        int v = e[i].ver;
        if(v == fat[u] || v == hson[u]) continue;
        dfs2(v, v);
    }
}
void add_subtree(int u, int value) {
    tr.update(1, dfn[u], dfn[u] + siz[u] - 1, value);
    // 子树代表的区间的左右端点分别是 dfn[u], dfn[u] + siz[u] - 1;
}
int query_subtree(int u) {
    return tr.query(1, dfn[u], dfn[u] + siz[u] - 1) % mod;
}
// 类似倍增 LCA 的跳重链过程
void add_path(int u, int v, int value) {
    while(top[u] != top[v]) {
        if(dep[top[u]] < dep[top[v]])
            swap(u, v);
        // 让链顶深度大的来跳

        tr.update(1, dfn[top[u]], dfn[u], value);
        // 把 u 到链顶的权值全部修改。
        u = fat[top[u]];
        // 跳到链顶的父亲节点。
    }

    if(dep[u] > dep[v]) swap(u, v);
    tr.update(1, dfn[u], dfn[v], value);
    // 一条重链上的 dfn 是连续的。
}
int query_path(int u, int v) {
    int ret = 0;
    while(top[u] != top[v]) {
        if(dep[top[u]] < dep[top[v]])
            swap(u, v);
        ret = (ret + tr.query(1, dfn[top[u]], dfn[u])) % mod;
        u = fat[top[u]];
    }
    if(dep[u] > dep[v]) swap(u, v);
    ret = (ret + tr.query(1, dfn[u], dfn[v])) % mod;
    return ret % mod;
}
int lca(int u, int v) {
    while(top[u] != top[v]) {
        if(dep[top[u]] < dep[top[v]])
            swap(u, v);
        u = fat[top[u]];
    }
    if(dep[u] > dep[v]) swap(u, v);
    return u;
}
```

2.7 主席树

```
// 静态区间第 K 大
const int si = 1e5 + 10;
int n, m, len;
int a[si], id[si];
int tot = 0;
int ls[si << 5], rs[si << 5];
int root[si << 5], dat[si << 5];
int build(int l, int r) {
    int p = ++tot;
    if(l == r) return p;
    int mid = (l + r) >> 1;
    ls[p] = build(l, mid), rs[p] = build(mid + 1, r);
    return p;
}
int insert(int last, int l, int r, int val) { // last 是上一个版本的 [l, r] 节点。
    int p = ++tot;
    dat[p] = dat[last] + 1;
    if(l == r) return p;
    int mid = (l + r) >> 1;
    if(val <= mid)
        ls[p] = insert(ls[last], l, mid, val), rs[p] = rs[last];
    else
        rs[p] = insert(rs[last], mid + 1, r, val), ls[p] = ls[last];
    return p;
}
int ask(int p, int q, int l, int r, int kth) {
    if(l == r) return l;
    int mid = (l + r) >> 1;
    int lcnt = dat[ls[q]] - dat[ls[p]];
    if(kth <= lcnt)
        return ask(ls[p], ls[q], l, mid, kth);
    else
        return ask(rs[p], rs[q], mid + 1, r, kth - lcnt);
}
int index(int val) {
    return lower_bound(id + 1, id + 1 + len, val) - id;
}
int main() {
    read(n), read(m);
    for(int i = 1; i <= n; ++i)
        read(a[i]), id[i] = a[i];
    sort(id + 1, id + 1 + n);
    len = unique(id + 1, id + 1 + n) - id - 1;
    root[0] = build(1, len);
    for(int i = 1; i <= n; ++i)
        root[i] = insert(root[i - 1], 1, len, index(a[i]));
    while(m--) {
        int l, r, k; read(l), read(r), read(k);
        write(id[ask(root[l - 1], root[r], 1, len, k)]);
        write(endl);
    }

    return 0;
}

// 单点修改
const int si = 2e5 + 10;

int n, m, len;
int a[si], id[si << 1];

int tot = 0;
int ls[si << 8], rs[si << 8];
int root[si << 8], dat[si << 8];

int cnt1, cnt2;
int tr1[si], tr2[si];

struct Query { char opt; int l, r, x; } q[si];
inline int lowbit(int x) { return x & -x; }
inline int getid(int val) { return lower_bound(id + 1, id + 1 + len, val) - id; }

int build(int l, int r) {
    int p = ++tot;
    if(l == r) return l;
    int mid = (l + r) >> 1;
    ls[p] = build(l, mid), rs[p] = build(mid + 1, r);
    return p;
}
void insert(int &p, int last, int l, int r, int val, int delta) {
    p = ++tot;
    dat[p] = dat[last] + delta, ls[p] = ls[last], rs[p] = rs[last];
    if(l == r) return;
    int mid = (l + r) >> 1;
    if(val <= mid) insert(ls[p], ls[last], l, mid, val, delta);
    else insert(rs[p], rs[last], mid + 1, r, val, delta);
}
int ask(int l, int r, int kth) {
    if(l == r) return l;
    int mid = (l + r) >> 1;
    int lcnt = 0;
    for(int i = 1; i <= cnt2; ++i) lcnt += dat[ls[tr2[i]]];
    for(int i = 1; i <= cnt1; ++i) lcnt -= dat[ls[tr1[i]]];
    if(kth <= lcnt) {
        for(int i = 1; i <= cnt1; ++i) tr1[i] = ls[tr1[i]];
        for(int i = 1; i <= cnt2; ++i) tr2[i] = ls[tr2[i]];
        return ask(l, mid, kth);
    }
    else {
        for(int i = 1; i <= cnt1; ++i) tr1[i] = rs[tr1[i]];
        for(int i = 1; i <= cnt2; ++i) tr2[i] = rs[tr2[i]];
        return ask(mid + 1, r, kth - lcnt);
    }
```

```
}
void change(int x, int v) {
    int y = getid(a[x]);
    while(x <= n) {
        insert(root[x], root[x], 1, len, y, v);
        x += lowbit(x);
    }
}
int query(int l, int r, int kth) {
    l --, cnt1 = cnt2 = 0;
    while(l) tr1[++cnt1] = root[l], l -= lowbit(l);
    while(r) tr2[++cnt2] = root[r], r -= lowbit(r);
    return ask(1, len, kth);
}

int main() {
    cin >> n >> m;
    int cnt = 0;
    for(int i = 1; i <= n; ++i)
        cin >> a[i], id[++cnt] = a[i];
    for(int i = 1; i <= m; ++i) {
        Query &p = q[i];
        cin >> p.opt;
        if(p.opt == 'C')
            cin >> p.l >> p.x, id[++cnt] = p.x;
        if(p.opt == 'Q')
            cin >> p.l >> p.r >> p.x;
    }
    sort(id + 1, id + 1 + cnt);
    len = unique(id + 1, id + 1 + cnt) - id - 1;
    for(int i = 1; i <= n; ++i) change(i, 1);
    for(int i = 1; i <= m; ++i) {
        Query &p = q[i];
        if(p.opt == 'C') change(p.l, -1), a[p.l] = p.x,
  ↪ change(p.l, 1);
        if(p.opt == 'Q') cout << id[query(p.l, p.r, p.x)]
  ↪ << endl;
    }

    return 0;
}
```

2.8 李超树

```
const ldb eps = 1e-9;
const int mod1 = 39989;
const int mod2 = 1e9;
const int si = 1e5 + 10;

int n, tot = 0;
struct Line { double k, b; } a[si];
ldb calc(int idx, int x) { return (a[idx].k * x +
  ↪ a[idx].b); }
void add(int x, int y, int xx, int yy) {
    ++tot;
    if(x == xx) a[tot].k = 0, a[tot].b = max(y, yy);
    else a[tot].k = (ldb)((1.0 * (yy - y)) / (1.0 * (xx -
  ↪ x))), a[tot].b = y - a[tot].k * x;
}
int cmp(ldb x, ldb y) {
    if((x - y) > eps) return 1; // Greater.
    else if((y - x) > eps) return -1; // Less
    return 0;
}
pdi Max(pdi x, pdi y) {
    if(cmp(x.first, y.first) == 1) return x;
    else if(cmp(y.first, x.first) == 1) return y;
    return (x.second < y.second) ? x : y;
}

struct LichaoTree {
    int id[si << 2];
    void modify(int p, int l, int r, int u) {
        int &v = id[p], mid = (l + r) >> 1;
        if(cmp(calc(u, mid), calc(v, mid)) == 1)
            swap(u, v);
        Lich boundl = cmp(calc(u, l), calc(v, l));
        int boundr = cmp(calc(u, r), calc(v, r));
        if(boundl == 1 || (!boundl && u < v))
        Lich modify(p << 1, l, mid, u);
        if(boundr == 1 || (!boundr && u < v))
            modify(p << 1 | 1, mid + 1, r, u);
    } //
    void update(int p, int nl, int nr, int l, int r, int u)
  ↪ {
        if(l <= nl && nr <= r)
            return modify(p, nl, nr, u);
        int mid = (nl + nr) >> 1;
        if(l <= mid)
            update(p << 1, nl, mid, l, r, u);
        if(r > mid)
            update(p << 1 | 1, mid + 1, nr, l, r, u);
    }
    pdi query(int p, int l, int r, int x) {
        if(x < l || r < x)
            return {0.0, 0};
        ldb ret = calc(id[p], x), mid = (l + r) >> 1;
        if(l == r)
            return {ret, id[p]};
        return Max({ret, id[p]}, Max(query(p << 1, l, mid,
  ↪ x), query(p << 1 | 1, mid + 1, r, x)));
    }
} tr;
```

2.9 珂朵莉树

```
struct node {
    int l, r;
    mutable int val;
    node(const int &il, const int &ir, const int &iv) :
  ↪ l(il), r(ir), val(iv) {}
    bool operator < (const node &b) const { return l < b.l;
  ↪ }
}; std::set<node> odt;

std::set<node>::iterator split(int pos) {
    if(pos > n) return odt.end();
    std::set<node>::iterator it =
 |  | --odt.upper_bound((node){pos, 0, 0});
    if(it -> l == pos) return it;
    int l = it -> l, r = it -> r, v = it -> val;
    odt.erase(it), odt.insert((node){l, pos - 1, v});
    return odt.insert((node){pos, r, v}).first;
} // split the node [l,r] to two smaller node [l,pos),
  ↪ [pos,r];
void assign(int l, int r, int v) {
    std::set<node>::iterator itr = split(r + 1), itl =
  ↪ split(l);
    odt.erase(itl, itr), odt.insert((node){l, r, v});
} // change all element in the interval [l,r] to v;
void example(int l, int r, int v) {
    std::set<node>::iterator itr = split(r + 1), itl =
  ↪ split(l);
    for(; itl != itr; ++itl) {
        // blablabla...
    }
 | return;
}
```

3. Dp

3.1 数位 DP

```

def dfs(当前位数 x, 当前状态 y, 前导零限制 st, 上界限制
  ↪ limit):
    if 到达边界 and 符合要求 then
    返回边界的合法答案
  if 到达边界 and 不符合要求 then
    返回边界的不合法答案          # 这个一般不会有，一般枚举填
      ↪ 数的时候如果没有限制就会有（只要会访问到边界不合法情况
      ↪ 就要加上）。

  if 当前的状态已经记忆化过 then
    返回记录的答案

  var result = 0          # 记录答案
    var up = 9            # 当前位填数的上限
    if 有上界限制 then
        up = 当前位在 n 当中的数字        # n 是要求的 F(n) 的
          ↪ 自变量

    for 枚举当前位的填数值 from 0 to up :
        if 当前位填的数不符合限制 then
            continue
```

```
        if 有前导零限制 and 当前填写的是 0 then
            result += dfs(x - 1, 下一个状态, True, 是否触碰上界限制)
        else then
            result += dfs(x - 1, 下一个状态, False, 是否触碰上界限制)

    记录当前状态的答案 f[x][y] = result
    返回答案 result


def solve(要求的 F 的自变量 n):
    存储每一位数字的 vector 清空
    while n != 0 then:
        vector <-- n % base            # base表示是哪一个进制
        n /= base
    清空状态数组
    返回对应状态的答案（调用 dfs）
```

3.2 子集卷积/SOS DP

```cpp
for(int msk = 0; msk < (1 << n); ++msk)
    dp[msk] = a[msk];
for(int i = 0; i < n; ++i) {
    for(int msk = 0; msk < (1 << n); ++msk) {
        if(msk & (1 << i))
            dp[msk] += dp[msk ^ (1 << i)];
    }
}
```

4. Math

4.1 线性求逆元以及组合数

```cpp
int inv[si], fact[si], invf[si];
void init(int n) {
    inv[1] = 1, fact[0] = invf[0] = 1;
    for(int i = 2; i <= n; ++i)
        inv[i] = 1ll * (mod - mod / i) * inv[mod % i] % mod;
    for(int i = 1; i <= n; ++i)
        fact[i] = 1ll * fact[i - 1] * i % mod,
        invf[i] = 1ll * invf[i - 1] * inv[i] % mod;
}
int C(int n, int m) {
    if(m < 0 || n < m) return 0;
    return 1ll * fact[n] * invf[n - m] % mod * invf[m] % mod;
}
int Catalan(int n) {
    return 1ll * C(n * 2, n) % mod * inv[n + 1] % mod;
}
```

4.2 快速幂

```cpp
int Qpow(int a, int b) {
 | int ret = 1 % mod;
 | for(; b; b >>= 1) {
 |   | if(b & 1) ret = ret * a % mod;
 |   | a = a * a % mod;
 | }
 | return ret % mod;
}
```

4.3 矩阵乘法

```cpp
struct Matrix {
    int a[si][si];
    Matrix() { memset(a,0,sizeof a); }
    Matrix operator * (const Matrix &B) const {
        Matrix C, A = *this;
        for(int i = 1; i <= cnt; ++i)
            for(int j = 1; j <= cnt; ++j)
                for(int k = 1; k <= cnt; ++k)
                    C.a[i][j] += A.a[i][k] * B.a[k][j];
        return C;
    }
};
// 循环的时候最好不要用 si。
// 用一个设定好的常数或者题目给的变量会比较好。
// 但是如果乘法不止需要适用于一对 n,m,k，那么就最好用 si - 1。
// 为啥不会有影响呢？因为构造函数里把没有用到的设置成 0 了。
```

4.4 高斯消元

```cpp
using i64 = long long;
using ldb = long double;

const int si = 50 + 10;
const ldb eps = 1e-5;

int n;
ldb c[si][si], d[si], x[si];

int Gauss() {
    for(int i = 1; i <= n; ++i) {
        int l = i;
        for(int j = i + 1; j <= n; ++j)
            if(fabs(c[j][i]) > fabs(c[l][i]))
                l = j; // 找到最大的
        if(l != i) {
            for(int j = 1; j <= n; ++j)
                swap(c[i][j], c[l][j]);
            swap(d[i], d[l]);
        } // 交换
        if(fabs(c[i][i]) >= eps) {
            for(int j = 1; j <= n; ++j) {
                if(j == i) continue;
                ldb rte = c[j][i] / c[i][i];
                for(int k = 1; k <= n; ++k)
                    c[j][k] -= rte * c[i][k];
                d[j] -= rte * d[i];
            }
        } // 消元
    }
    bool nosol = false, infsol = false;
    for(int i = 1; i <= n; ++i) {
        int j = 1;
        while(fabs(c[i][j]) < eps && j <= n)
            j ++;
        j += (fabs(d[i]) < eps);
        if(j > n + 1) infsol = true;
        if(j == n + 1) nosol = true;
    } // 检查自由元
    if(nosol)  return 0;
    if(infsol) return 1;
    for(int i = n; i >= 1; --i) {
        for(int j = i + 1; j <= n; ++j)
            d[i] -= x[j] * c[i][j];
        x[i] = d[i] / c[i][i];
    } // 回代
    for(int i = 1; i <= n; ++i)
        cout << "x" << i << "=" << fixed << setprecision(2)
  ↪ << x[i] << endl;
    return 2;
}
```

4.5 质数

```cpp
bool is_prime(int n) {
    if(n < 2) return false;
    for(int i = 2; i * i <= n; ++i)
        if(n % i == 0) return false;
    return true;
}

int vis[si];
int m, prime[si];
// O(n log log n)
void get_primes(int n) {
    m = 0;
    memset(vis, 0, sizeof vis);
    for(int i = 2; i <= n; ++i) {
        if(!vis[i]) prime[++m] = i;
        for(int j = i * i; j <= n; ++j)
            vis[j] = 1;
    }
}

// Euler O(n)
void get_primes(int n) {
    m = 0;
    memset(vis, 0, sizeof vis);
    for(int i = 2; i <= n; ++i) {
        if(vis[i] == 0) {
```

```
            vis[i] = i;
            prime[++m] = i;
        }
        for(int j = 1; j <= m; ++j) {
            if(prime[j] > vis[i] || prime[j] * i > n)
                break;
            vis[prime[j] * i] = prime[j];
        }
    }
}

int c[si]; // exponential
int m = 0, p[si]; // prime factor
void divide(int n) {
    m = 0;
    for(int i = 2; i * i <= n; ++i) {
        if(n % i == 0) {
            p[++m] = i, c[m] = 0;
            while(n % i == 0) n /= i, c[m]++;
        }
    }
    if(n > 1) p[++m] = n, c[m] = 1;
}
```

4.6 约数

```
int m, div[si];
void get_factors(int n) {
    m = 0;
    for(int i = 1; i * i <= n; ++i)
        if(n % i == 0) {
            div[++m] = i;
            if(i * i != n) div[++m] = n / i;
        }
}

std::vector<int> fact[si];
void get_factors(int n) {
    for(int i = 1; i <= n; ++i)
        for(int j = 1; j <= n / i; ++j)
            fact[i * j].emplace_back(i);
}

int gcd(int a, int b) {
 | return b ? gcd(b, a % b) : a;
}

int phi[si];
int m = 0, prime[si], vis[si];
void calc_euler_func(int n) {
  m = 0, phi[1] = 1;
  memset(vis, 0, sizeof vis);
  for(int i = 2; i <= n; ++i) {
      if(vis[i] == 0)
          vis[i] = i, prime[++m] = i, phi[i] = i - 1;
      for(int j = 1; j <= m; ++i) {
          if(prime[j] > vis[i] || prime[j] * vis[i] > n)
  ↪ break;
          vis[prime[j] * i] = prime[j];
          if(i % prime[j] == 0)
              phi[prime[j] * i] = phi[i] * prime[j];
          else
              phi[prime[j] * i] = phi[i] * (prime[j] - 1);
      }
  }
}
```

4.7 Exgcd

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

4.8 中国剩余定理

```
#define int long long
int crt(std::vector<int> &r, std::vector<int> &m) {
    int n = 1, ans = 0;
    for(int i = 0; i < (int)m.size(); ++i)
        n = n * m[i];
    for(int i = 0; i < (int)m.size(); ++i) {
        int mi = n / m[i], b, y;
        exgcd(mi, m[i], b, y);
        ans = (ans + r[i] * mi * b % n) % n;
    }
    return (ans % n + n) % n;
}
```

5. Graph

5.1 SPFA 以及负环

```
std::queue<int> q;
void spfa(int s) {
 | memset(dis, 0x3f, sizeof dis);
    memset(vis, false, sizeof vis);
    dis[s] = 0, q.push(s), vis[s] = true;
    while(!q.empty()) {
        int u = q.front();
 |  | q.pop(), vis[u] = false;
        for(int i = head[u]; ~i; i = e[i].Next) {
            int v = e[i].ver, w = e[i].w;
            if(dis[v] > dis[u] + w){
                dis[v] = dis[u] + w;
                if(!vis[v]) q.push(v), vis[v] = true;
            }
        }
    }
}

// Minus Ring Check
bool vis[si];
std::queue<int> Q;
int dis[si], cnt[si];
bool spfa(int s) {
    memset(dis, 0, sizeof dis);
 | memset(cnt, 0, sizeof cnt);
 | memset(vis, false, sizeof vis);
    for(int i = 1; i <= n; ++i)
        Q.push(i), vis[i] = true;
 | cnt[s] = 0; // 全部入队，相当于建立一个超级源点。
    while(!Q.empty()) {
        int u = Q.front();
 |  | Q.pop(), vis[u] = false;
        for(int i = head[u]; ~i; i = e[i].Next) {
            int v = e[i].ver, w = e[i].w;
            if(dis[v] > dis[u] + w) {
                dis[v] = dis[u] + w, cnt[v] = cnt[u] + 1;
                if(cnt[v] >= n) return true;
                if(!vis[v]) Q.push(v), vis[v] = true;
            }
        }
    }
 | return false;
}

// SLF + Swap Optimize

struct Slfswap {
    std::deque<int> dq;
    Slfswap() { dq.clear(); }
    void push(int x) {
        if(!dq.empty()) {
            if(dis[x] < dis[dq.front()])
 |  |  |  | dq.push_front(x);
            else dq.push_back(x);
            if(dis[dq.front()] > dis[dq.back()])
 |  |  |  | swap(dq.front(), dq.back());
            // 这里的两重 if 可以保证只会在至少有两个元素的时
  ↪ 候才交换。
        } else dq.push_back(x);
    }
    void pop() {
        dq.pop_front();
        if(!dq.empty() && dis[dq.front()] > dis[dq.back()])
  ↪
 |  |  | swap(dq.front(), dq.back());
    }
    int size() { return dq.size(); }
    int front() { return dq.front(); }
    bool empty() { return !dq.size(); }
} q;
```

5.2 Dijkstra

```
std::priority_queue<std::pair<int, int> > q;
void dijkstra(int s) {
 | memset(dis, 0x3f, sizeof dis);
    memset(vis, false ,sizeof vis);
    dis[s] = 0, q.push({dis[s], s});
    while(!q.empty()) {
        int u = q.top().second;
 |  | q.pop();
        if(vis[u]) continue;
 |  | vis[u] = true;
        for(int i = head[u]; ~i; i = e[i].Next) {
            int v = e[i].ver, w = e[i].w;
            if(dis[v] > dis[u] + w)
 |  |  |  | dis[v] = dis[u] + w, q.push({-dis[v], v}); //利用相反数把大根堆-> 小根堆
            // 一定要先更新 dis[v] 再 q.push
        }
    }
}
```

5.3 Floyd 以及最小环

```
for(int k = 1; k <= n; ++k)
    for(int i = 1;i <= n; ++i)
        for(int j = 1; j <= n; ++j)
            dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j]);
// 不要忘记初始化.

// 最小环:
std::vector<int> ans_path;
void gopath(int u, int v) {
    if(pos[u][v] == 0)
        return;
    gopath(u, pos[u][v]), ans_path.push_back(pos[u][v]), gopath(pos[u][v], v);
}

signed main() {
    cin >> n >> m;
    memset(a, 0x3f, sizeof a);
    for(int i = 1; i <= n; ++i)
        a[i][i] = 0;
    for(int i = 1; i <= m; ++i) {
        int u, v, w;
        cin >> u >> v >> w;
        a[u][v] = min(a[u][v], w), a[v][u] = a[u][v];
    }
    memcpy(dis, a, sizeof a);
    int ans = 0x3f3f3f3f3f3f3f3f, tmp = ans;
    for(int k = 1; k <= n; ++k)
        for(int i = 1; i < k; ++i) // 注意是dp之前，此时 dis 还是 k-1 的时候的状态。
            for(int j = i + 1; j < k; ++j)
                if(a[j][k] < tmp / 2 && a[k][i] < tmp / 2 && ans > dis[i][j] + a[j][k] + a[k][i])
                    ans = dis[i][j] + a[j][k] + a[k][i],
                    ans_path.clear(),
                    ans_path.push_back(i), gopath(i, j),
                    ans_path.push_back(j),
                    ans_path.push_back(k);
                // 不判的话 a[j][k]+a[k][i] 有可能爆，导致答案出错。
        // 更新最小环取min的过程
        for(int i = 1; i <= n; ++i)
            for(int j = 1; j <= n; ++j)
                if(dis[i][j] > dis[i][k] + dis[k][j])
                    pos[i][j] = k, dis[i][j] = dis[i][k] + dis[k][j];
        // 正常的 Floyd
    }
    if(ans == 0x3f3f3f3f3f3f3f3f)
        return puts("No solution."), 0;
    for(auto x : ans_path)
        cout << x << " ";
    return puts(""), 0;
}
```

5.4 Kruskal

```
struct Edge {
    int x, y, z;
    bool operator < (const Edge &b)const {
        return z < b.z;
    }
} a[si_m];

for(int i = 1; i <= m; ++i)
 | cin >> a[i].x >> a[i].y >> a[i].z;
sort(a + 1, a + 1 + m);
int ans = 0;
for(int i = 1; i <= m; ++i) {
 | if(dsu.same(a[i].x, a[i].y))
 |  | continue;
 | dsu.Union(a[i].x, a[i].y), ans += a[i].z;
}
```

Prim

```
void Prim() {
 | memset(dis, 0x3f, sizeof dis);
    memset(vis, false, sizeof vis), dis[1] = 0;
    for(int i = 1; i < n; ++i) {
        int x = 0;
        for(int j = 1; j <= n; ++j)
            if(!vis[j] && (x == 0 || dis[j] < dis[x]))
                x = j;
        vis[x] = true;
        for(int y = 1; y <= n; ++y)
            if(!vis[y]) dis[y] = min(dis[y], a[x][y]);
    }
}

memset(a, 0x3f, sizeof a);
for(int i = 1; i < n; ++i) {
 | a[i][i] = 0;
 | for(int j = 1; j <= n; ++j) {
 |  | int value;
 |  | cin >> value;
 |  | a[i][j] = a[j][i] = min(a[i][j], value);
 | }
}
Prim();
int ans = 0;
for(int i = 2; i <= n; ++i)
 | ans += dis[i];
```

5.5 倍增 LCA

```
int dep[si_n],f[si_n][20];
void dfs(int u, int fa) {
    dep[u] = dep[fa] + 1, f[u][0] = fa;
    for(int i = 1; i <= 19; ++i)
        f[u][i] = f[f[u][i - 1]][i - 1];
    for(int i = head[u]; ~i; i = e[i].Next) {
        int v = e[i].ver;
        if(v == fa) continue;
        dfs(v, u);
    }
}
int lca(int x, int y) {
    if(dep[x] < dep[y]) swap(x,y);
    for(int i = 19; i >= 0; --i)
        if(dep[f[x][i]] >= dep[y]) x = f[x][i];
    if(x == y) return x;
    for(int i = 19; i >= 0; --i)
        if(f[x][i] != f[y][i]) x = f[x][i], y = f[y][i];
    return f[x][0];
}
```

5.6 Tarjan LCA

```
int pa[si];
int root(int x) {
    if(pa[x] != x)
        return pa[x] = root(pa[x]);
    return pa[x];
}

int n, q, s;
int lca[si];
bool vis[si];
std::vector<int> que[si], pos[si];
```

```
void tarjan(int u) {
    vis[u] = true;
    for(int i = head[u]; ~i; i = e[i].Next) {
        int v = e[i].ver;
        if(vis[v] == true) continue;
        tarjan(v), pa[v] = root(u);
    }
    for(int i = 0; i < (int)que[u].size(); ++i) {
        int v = que[u][i], po = pos[u][i];
        if(vis[v] == true) lca[po] = root(v);
    }
}

int main(){
 | cin >> n >> q >> s;
    for(int i = 1; i <= n; ++i)
        pa[i] = i, vis[i] = false,
        que[i].clear(), pos[i].clear();
    for(int i = 1; i < n; ++i) {
        int u, v;
 |  | cin >> u >> v;
        add(u, v), add(v, u);
    }
    for(int i = 1; i <= q; ++i) {
        int u, v;
 |  | cin >> u >> v;
        if(u == v) lca[i] = u;
        else {
            que[u].pb(v), que[v].pb(u);
            pos[u].pb(i), pos[v].pb(i);
        }
    }
    tarjan(s);
    for(int i = 1; i <= q; ++i)
        cout << lca[i] << endl;
    return 0;
}
```

5.7 拓扑排序

```
int cnt = 0;
std::queue<int> q;
for(int i = 1; i <= n; ++i)
    if(!ind[i]) q.push(i);
while(!q.empty()) {
    int u = q.front(); q.pop();
    ord[u] = ++cnt; // topo 序
    for(auto v : G[u]) if(!(--ind[v])) q.push(v);
    // 删掉边，顺便判一下要不要入队。
}
```

5.8 欧拉回路

```
std::stack<int> s;
void dfs(int u) {
    for(int i = head[u]; ~i ; i = e[i].Next) {
        int v = e[i].ver;
        if(!vis[i]){ // 当前边没有访问过
            vis[i] = true; // 注意一定要访问到就直接标记，不然复杂度会假。
            dfs(v), s.push(v);
        }
    }
}

dfs(1); // 因为有欧拉回路，所以其实从哪个点开始都一样。
std::vector<int>ans;
while(!s.empty())
 | ans.push_back(s.top()), s.pop();
reverse(ans.begin(), ans.end());
for(auto x : ans) cout << x << " "; // 倒序输出。
```

5.9 强连通分量

```

bool ins[si];
std::stack<int> s;
std::vector<int> scc[si];
int n, m, cnt_t = 0, tot = 0;
int dfn[si], low[si], c[si];

void tarjan(int u) {
    dfn[u] = low[u] = ++cnt_t;
    s.push(u), ins[u] = true;
    for(int i = head[u]; ~i; i = e[i].Next) {
        int v = e[i].ver;
        if(!dfn[v])
 |  |  | tarjan(v), low[u] = min(low[u], low[v]);
        // 没有访问过，递归搜索然后更新 low。
        else if(ins[v]) low[u] = min(low[u], dfn[v]);
        // 已经在栈中了，用 dfn[v] 来更新 low[u]。
    }
    if(dfn[u] == low[u]) {
        ++tot; int x;
        do {
            x = s.top(), s.pop(), ins[x] = false;
            c[x] = tot, scc[tot].pb(x);
        } while(u! = x);
    } // 出现了一个 SCC。
}

Edge edag[si << 1];
void contract() {
    for(int u = 1; u <= n; ++u) {
        for(int i = head[u]; ~i; i = e[i].Next) {
            int v = e[i].ver;
            if(c[u] == c[v]) continue;
            add_n(c[u], c[v]);
        }
    } // 缩点。
}

for(int i = 1; i <= n; ++i)
 | if(!dfn[i]) tarjan(i);
```

5.10 边双连通分量

```
int n, m, q;
// 原图
int head[si], tot1 = 0;
struct Edge { int ver, Next; }e[si << 2];
inline void add1(int u, int v) { e[tot1] = (Edge){v, head[u]}, head[u] = tot1++; }

// 缩完点之后的图
// 如果原来的图是连通图的话
// 可以证明缩完点之后必然是一棵树。
int Head[si], tot2 = 0;
struct Tree { int ver, Next; }t[si << 2];
inline void add2(int u, int v) { t[tot2] = (Tree){v, Head[u]}, Head[u] = tot2++; }

// E-dcc 的个数.
int cnt = 0;
int dfn[si], low[si], tim = 0, c[si];
bool bridge[si << 2]; // 是否是桥

// in_edge 是用来消除重边的影响的。
// 表示当前状态是从哪一条边过来的。
void tarjan(int u, int in_edge) {
    dfn[u] = low[u] = ++tim;
    for(int i = head[u]; ~i; i = e[i].Next) {
        int v = e[i].ver;
        if(!dfn[v]) {
            tarjan(v, i);
            low[u] = min(low[u], low[v]);
            if(dfn[u] < low[v]) bridge[i] = bridge[i ^ 1] = true;
        }
        else if((i ^ 1) != in_edge) low[u] = min(low[u], dfn[v]);
    }
}

// 去掉桥边的连通块染色
void dfs(int u, int col) {
    c[u] = col;
    for(int i = head[u]; ~i; i = e[i].Next) {
        int v = e[i].ver;
        if(c[v] || bridge[i]) continue;
        dfs(v, col);
    }
}
void Construct() {
    for(int i = 1; i <= n; ++i){
```

```
        for(int j = head[i]; ~j; j = e[j].Next) {
            int v = e[j].ver;
            if(c[i] == c[v]) continue;
            // 只需要加一次，遍历到反向边的时候会自动补全成无向边
            add2(c[i], c[v]);
        }
    }
}

int main() {
    memset(head, -1, sizeof head);
    memset(Head, -1, sizeof Head);
    memset(bridge, false, sizeof bridge);
    cin >> n >> m;
    for(int i = 1; i <= m; ++i) {
        int u, v;
        cin >> u >> v;
        add1(u, v), add1(v, u);
    }
    for(int i = 1; i <= n; ++i) if(!dfn[i]) tarjan(i, -1);
    for(int i = 1; i <= n; ++i) if(!c[i]) ++cnt, dfs(i, cnt);
    Construct();
}
```

5.11 点双连通分量

```
int n, m, root;

int head[si], tot1 = 0;
int Head[si], tot2 = 0;
struct Edge { int ver, Next; }e[si << 2], g[si << 2];
void add1(int u, int v) { e[tot1] = (Edge){v, head[u]}, head[u] = tot1++; }
void add2(int u, int v) { g[tot2] = (Edge){v, Head[u]}, Head[u] = tot2++; }

// Vdcc 的个数
int cnt = 0;
int dfn[si], low[si], c[si], tim;
int new_id[si]; // 割点的新编号
bool cut[si]; // 是否是割点
std::stack<int> s;
std::vector<int> vdcc[si];

void tarjan(int u) {
    dfn[u] = low[u] = ++tim;
    s.push(u);
    // 孤立点
    if(u == root && head[u] == -1) {
        vdcc[++cnt].emplace_back(u);
        return;
    }
    int flag = 0;
    for(int i = head[u]; ~i; i = e[i].Next) {
        int v = e[i].ver;
        if(!dfn[v]) {
            tarjan(v), low[u] = min(low[u], low[v]);
            if(dfn[u] <= low[v]) {
                ++flag;
                // 根节点特判
                if(u != root || flag > 1) { // 注意这里是短路运算符，不要打反了。
                    cut[u] = true;
                }
                int x; ++cnt;
                do {
                    x = s.top(), s.pop();
                    vdcc[cnt].emplace_back(x);
                } while(v != x);
                // 注意这里要是 v 不是 u
                // 如果 u 被弹出了，之后的连通块就会少 u。
                vdcc[cnt].emplace_back(u);
            }
        }
        else low[u] = min(low[u], dfn[v]);
    }
}

int num;
void Construct() {
    num = cnt;
    for(int u = 1; u <= n; ++u)
        if(cut[u]) new_id[u] = ++num;
    for(int i = 1; i <= cnt; ++i) {
        for(int j : vdcc[i]) {
            if(cut[j]) add2(i, new_id[j]), add2(new_id[j], i);
            else c[j] = i;
        }
        // 如果是割点，就和这个割点所在的 v-Dcc 连边
        // 反之染色。
    }
    // 编号 1~cnt 的是 v-Dcc，编号 > cnt 的是原图割点
}

int main() {
    memset(head, -1, sizeof head);
    memset(Head, -1, sizeof Head);

    cin >> n >> m;
    for(int i = 1; i <= m; ++i) {
        int u, v;
        cin >> u >> v;
        // 判重边
        if(u == v) continue;
        add1(u, v), add1(v, u);
    }
    for(int i = 1; i <= n; ++i)
        if(!dfn[i]) root = i, tarjan(i);
    Construct();
    return 0;
}
```

5.12 虚树

```
int k, a[si];
int stk[si], top = 0;
bool cmp(int x, int y) { return dfn[x] < dfn[y]; }
inline void ADD(int u, int v, int w) { E[Tot] = (Edge){v, Head[u], w}, Head[u] = Tot++; }
inline void Add(int u, int v) { int w = dist(u, v); ADD(u, v, w), ADD(v, u, w); }
void build() {
    sort(a + 1, a + 1 + k, cmp);
    stk[top = 1] = 1, Tot = 0, Head[1] = -1; // 这样清空复杂度才是对的。
    for(int i = 1, Lca; i <= k; ++i) {
        if(a[i] == 1) continue;
        Lca = lca(a[i], stk[top]);
        if(Lca != stk[top]) {
            while(dfn[Lca] < dfn[stk[top - 1]])
                Add(stk[top - 1], stk[top]), --top;
            if(dfn[Lca] > dfn[stk[top - 1]])
                Head[Lca] = -1, Add(Lca, stk[top]), stk[top] = Lca;
            else Add(Lca, stk[top--]); // Lca = stk[top - 1].
        }
        Head[a[i]] = -1, stk[++top] = a[i];
    }
    for(int i = 1; i < top; ++i)
        Add(stk[i], stk[i + 1]);
    return;
}
```

6. String

6.1 Kmp

```
Next[1] = 0;
for(int i = 2, j = 0; i <= n; ++i) {
    while(j > 0 && s[i] != s[j + 1]) j = Next[j];
    if(s[i] == s[j + 1]) j ++;
    Next[i] = j;
}
for(int i = 1, j = 0; i <= m; ++i) {
    while(j > 0 && (j == n || s[i] != s[j + 1])) j = Next[j];
    if(t[i] == s[j + 1]) ++j;
    f[i] = j;
    if(f[i] == n) orc[++cnt] = i - n + 1;
}
```

6.2 Trie

```cpp
// 定义 NULL 为 0，字符集为 a~z。
int tr[si][27];
bool exist[si];
int tot, root;

void init() {
    memset(tr, 0, sizeof tr);
    memset(exist, false, sizeof exist);
    tot = 0, root = ++tot;
}
void insert(string s) {
    int p = root;
    for(int i = 0; i < (int)s.size(); ++i) {
        int ch = (int) (s[i] - 'a') + 1;
        if(!tr[p][ch])
            tr[p][ch] = ++tot;
        p = tr[p][ch];
    }
    exist[p] = true;
}
bool query(string s) {
    int p = root;
    for(int i = 0; i < (int)s.size(); ++i) {
        int ch = (int) (s[i] - 'a') + 1;
        if(!tr[p][ch])
            return false;
        p = tr[p][ch];
    }
    return exist[p];
}
```

6.3 01Trie

```cpp
using i64 = long long;

const int si = 1e5 + 10;
const int k = 32;
int tr[k * si][2];
i64 value[k * si];
int tot = 0, root = ++tot;

int newnode() {
    tr[++tot][0] = tr[tot][1] = value[tot] = 0;
    return tot;
}
int cacid(int num, int pos) {
    return (num >> pos) & 1;
}
void insert(int num) {
    int p = root;
    for(int i = 32; i >= 0; --i) {
        int ch = cacid(num, i);
        if(!tr[p][ch])
            tr[p][ch] = newnode();
        p = tr[p][ch];
    }
    value[p] = num;
}
// 查询异或 x 最大的一个。
i64 query(i64 num) {
    int p = root;
    for(int i = 32; i >= 0; --i) {
        int ch = cacid(num, i);
        if(tr[p][ch ^ 1])
            p = tr[p][ch ^ 1];
        else
            p = tr[p][ch];
    }
    return value[p];
}

// 维护异或和，全局加一。
const int si = 1e4 + 10;
const int MaxDepth = 21;

int tr[si * (MaxDepth + 1)][2];
int wei[si * (MaxDepth + 1)], xorv[si * (MaxDepth + 1)];
int tot = 0, root = ++tot;
// 其实这里 root 可以不用赋值，递归开点的时候会自动给编号的。

int newnode() {
    tr[++tot][0] = tr[tot][1] = wei[tot] = xorv[tot] = 0;
    return tot;
}
void maintain(int p) {
    wei[p] = xorv[p] = 0;
    // 为了应对不断的删除和插入，每次维护 p 的时候都令 wei, xorv = 0。
    // 也就是每次都**重新收集一次信息**，而不是从原来的基础上修改。
    if(tr[p][0]) {
        wei[p] += wei[tr[p][0]];
        xorv[p] ^= (xorv[tr[p][0]] << 1);
        // 因为儿子所维护的异或和实际上比 p 少一位，
        // 如果要按位异或就要让儿子的异或和左移一位，和 p 对齐。
    }
    if(tr[p][1]) {
        wei[p] += wei[tr[p][1]];
        xorv[p] ^= (xorv[tr[p][1]] << 1) | (wei[tr[p][1]] & 1);
        // 利用奇偶性计算。
    }
    wei[p] = wei[p] & 1;
    // 每插入一次或者删除一次，奇偶性都会变化。
}
// 类似线段树的 pushup，从底向上收集信息。
// 换种说法，是更新节点 p 的信息。
void insert(int &p, int x, int depth) {
    if(!p)
        p = newnode();
    if(depth > MaxDepth) {
        wei[p] += 1;
        return;
    }
    insert(tr[p][x & 1], x >> 1, depth + 1);
    // 从低到高位插入，所以是 x >> 1。
    maintain(p);
}
// 插入元素 x。
void remove(int p, int x, int depth) {
    // 不知道是不是应该写 > MaxDepth - 1 还是 > MaxDepth ?
    if(depth == MaxDepth) {
        wei[p] -= 1;
        return;
    }
    remove(tr[p][x & 1], x >> 1, depth + 1);
    maintain(p);
}
// 删除元素 x，但是 x 不能是不存在的元素。
// 否则会访问空节点 0 然后继续往下，会出错。
void addall(int p) {
    swap(tr[p][0], tr[p][1]);
    if(tr[p][0])
        addall(tr[p][0]);
    maintain(p);
    // 交换后下面都被更改了，需要再次 maintain。
}
// 全部加一

int main() {
    int n;
    cin >> n;
    std::vector<int> v(n + 1);
    for(int i = 1; i <= n; ++i) {
        cin >> v[i],
        insert(root, v[i], 0);
    }
    cout << xorv[root] << endl;
    // 查询总异或和
    int m;
    cin >> m;
    for(int i = 1; i <= m; ++i) {
        int x, y;
        cin >> y >> x;
        if(y == 0)
            remove(root, x, 0);
            // remove 和 addall 混用时小心 remove 掉不存在的元素！
        else
            addall(root);
        cout << xorv[root] << endl;
    }
```

```
}

// merge
int merge(int p, int q) {
    if(!p)
        return q;
    if(!q)
        return p;
    wei[p] += wei[q], xorv[p] ^= xorv[q];
    tr[p][0] = merge(tr[p][0], tr[q][0]);
    tr[p][1] = merge(tr[p][1], tr[q][1]);
    return p;
}
```

6.4 Ac Automaton

```
// 求有多少个 s 在 t 中出现过。
namespace Ac_Automaton{
  | const int si = 1e6 + 10;
  | int root = 0, tot = 0;
  | int tr[si][27], End[si], fail[si];
  | int cal(char ch) { return (int)(ch - 'a') + 1; }
  | void init() {
  |   | tot = 0;
  |   | memset(tr, 0, sizeof tr);
  |   | memset(End, 0, sizeof End);
  |   | memset(fail, 0, sizeof fail);
  | }
  | void insert(char *s) {
  |   | int u = 0;
  |   | for(int i = 1; s[i]; ++i) {
  |   |   | if(!tr[u][cal(s[i])])
  |   |   |   | tr[u][cal(s[i])] = ++ tot;
  |   |   | u = tr[u][cal(s[i])];
  |   | }
  |   | ++End[u];
  | }
  | std::queue<int>q;
  | void build() {
  |   | for(int i = 1; i <= 26; ++i)
  |   |   | if(tr[root][i]) q.push(tr[root][i]);
  |   | while(!q.empty()) {
  |   |   | int u = q.front();
  |   |   | q.pop();
  |   |   | for(int i = 1; i <= 26; ++i) {
  |   |   |   | if(tr[u][i])
  |   |   |   |   | fail[tr[u][i]] = tr[fail[u]][i],
    ↪ q.push(tr[u][i]);
  |   |   |   | else tr[u][i] = tr[fail[u]][i];
  |   |   | }
  |   | }
  | }
  | int query(char *t) {
  |   | int u = 0, res = 0;
  |   | for(int i = 1; t[i]; ++i) {
  |   |   | u = tr[u][cal(t[i])];
  |   |   | for(int j = u; j && End[j] != -1; j = fail[j])
  |   |   |   | res += End[j], End[j] = -1;
  |   | }
  |   | return res;
  | }
}
```

```
using namespace Ac_Automaton;

// 求次数。
namespace Ac_Automaton {
  | const int si = 2e6 + 10;
  | int root = 0, tot = 0, cnt_f = 0;
  | int tr[si][27], End[si], fail[si], cnt[si];
  | int cal(char ch) { return (int)(ch - 'a') + 1; }
  | void init() {
  |   | tot = 0;
  |   | memset(tr, 0, sizeof tr);
  |   | memset(cnt, 0, sizeof cnt);
  |   | memset(End, 0, sizeof End);
  |   | memset(fail, 0, sizeof fail);
  | }
  | void insert(char *s, int nu) {
  |   | int u = 0;
  |   | for(int i = 1; s[i]; ++i) {
  |   |   | if(!tr[u][cal(s[i])])
  |   |   |   | tr[u][cal(s[i])] = ++ tot;
  |   |   | u = tr[u][cal(s[i])];
  |   | }
  |   | End[nu] = u; // 这里改为记录第 nu 个模式串的结尾的位置。
  | }
  | int head[si];
  | struct Fail_Tree{ int ver, Next; }ft[si << 1];
  | void add(int u, int v) { ft[cnt_f] = (Fail_Tree){v,
    ↪ head[u]}, head[u] = cnt_f++; }
  | std::queue<int>q;
  | void build() {
  |   | for(int i = 1; i <= 26; ++i)
  |   |   | if(tr[root][i]) q.push(tr[root][i]);
  |   | while(!q.empty()) {
  |   |   | int u = q.front();
  |   |   | add(fail[u], u), q.pop();  // 构建 Fail 树
  |   |   | for(int i = 1; i <= 26; ++i) {
  |   |   |   | if(tr[u][i])
  |   |   |   |   | fail[tr[u][i]] = tr[fail[u]][i],
    ↪ q.push(tr[u][i]);
  |   |   |   | else tr[u][i] = tr[fail[u]][i];
  |   |   | }
  |   | }
  | }
  | void dfs(int u, int fa) {
  |   | for(int i = head[u]; ~i; i = ft[i].Next) {
  |   |   | int v = ft[i].ver;
  |   |   | if(v == fa) continue;
  |   |   | dfs(v, u), cnt[u] += cnt[v];
  |   | }  // 统计
  | }
  | void query(char *t) {
  |   | int u = 0;
  |   | for(int i = 1; t[i]; ++i)
  |   |   | u = tr[u][cal(t[i])], ++cnt[u];
  |   | // 记录每个状态被匹配多少次
  |   | dfs(root, -1);
  |   | for(int i = 1; i <= n; ++i)
  |   |   | printf("%d\n", cnt[End[i]]);
  | }
}
using namespace Ac_Automaton;
```

Good Luck && Have Fun!