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. Basic

2.1 二分

注意区分写法的不同!

```
int l = 1, r = n; // 判无解则令 r = n + 1，这种写法 mid 永远不会取到 r
while(l < r) {
    int mid = (l + r) >> 1;
    if(a[mid] >= x) r = mid; // mid 也可能是答案，也要取。
    else l = mid + 1;
}
ans = a[l];

int l = 1, r = n; // 判无解则令 l = 0，这种写法 mid 永远不会取到 l
while(l < r) {
    int mid =(l + r + 1) >> 1;
    if(a[mid] <= x) l = mid; // mid 也可能是答案，也要取。
    else r = mid - 1;
}
ans = a[l];

double l = 0.0, r = (double)(1e9 + 7), ans;
while(l + eps < r){
    double mid = (l + r) / 2;
    if(valid(mid)) r = mid, ans = mid;
    else l = mid;
}
```

2.2 时间复杂度分析

一些记号:

- Θ：对于两个函数 $f(n), g(n)$，如果存在 $c_1, c_2, n_0 > 0$，使得 $\forall n \geq n_0, 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$，则记为 $f(n) = \Theta(g(n))$，用于描述算法的上下界。
- O：对于两个函数 $f(n), g(n)$，如果存在 $c, n_0 > 0$，使得 $\forall n \geq n_0, 0 \leq f(n) \leq c \cdot g(n)$，则记为 $f(n) = O(g(n))$，用于描述算法的上界，大部分时候都用这个描述，不过算法使用的时候的最坏复杂度不是大 O 记号，用 Θ 表示最坏复杂度是完全可以的，只不过大部分时候都只比较方便证明出上界，所以用大 O 用的多，就是说，在一定情况下 O 可以表示最坏复杂度。
- o：就是 O 去掉等号变成 $<$。
- Ω：$\geq$。
- ω：$>$。

一些性质:

- $f1(n) + f2(n) = O(\max(f1(n), f2(n)))$（两个函数之和的上界是他们当中在**渐进意义上**较大的函数）。
- $f1(n) \cdot f2(n) = O(f1(n) \times f2(n))$。
- $\forall a \neq 1, \log_a(n) = O(\log_2(n))$，所以渐进时间复杂度的 log 都表示 $\log_2$，因为对于所有对数函数，不管底数如何，增长率 (Θ) 都是相同的，为了讨论方便，都换成底数为 2 的对数。

对于一个递归式算法 $T(n) = aT(\frac{n}{b}) + f(n)$，

其中 n 是问题规模大小，a 是子问题的个数，$\frac{n}{b}$ 是每个子问题的大小（规模），$f(n)$ 是将原问题分成子问题和将子问题的解合并的时间。
有以下结论（Master Theorem）

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with[1] $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ with $\epsilon > 0$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$.
   Regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.

3. Ds

3.1 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;
}
```

3.2 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;
}
```

3.3 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!
```

3.4 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;
```

3.5 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;
        }
```

```cpp
        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;
}

```

3.6 轻重链剖分

```cpp
// 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[i] - 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;
}
```

3.7 主席树

```cpp
// 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
```

```cpp
        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;
}
```

3.8 李超树

```cpp
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);
```

```cpp
        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;
```

3.9 珂朵莉树

```cpp
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;
}
```

4. Dp

4.1 一些注意事项

其实动态规划的本质是，在整个状态空间中，建立了一张 DAG（有向无环图），这保证了动态规划的 **无后效性**（不会成环）。其中，每一个节点对应一个 **状态**，同层的一系列节点称之为 **阶段**，它对应的就是一个 **子问题**。
而 **最优子结构**则是，对于一个问题的最优解，它一定包含了子问题的最优解，或者说，一个问题的最优解应当由子问题的最优解导出，而 **决策**则是这张图上的有向边。
贪心算法其实也是满足了最优子结构的，只是它和动态规划的转移不同，它每次只 **浅显**的取了 **当前局面**下的最优解。
动态规划则是，从先前的（多个）子问题当中，找到当前局面的最优解。
换句话来说，动态规划就是，对于一些满足最优子结构性质的信息，通过分割子问题，在子问题间无后效性转移，求得最终解的算法。

- dp 数组当中应当记录两个东西，一个是阶段，一个是状态，阶段是一定要确定清楚的，状态可以先暂时不管，慢慢根据转移的需求来确定。
- 当我们发现 dp 的状态有点多，复杂度高的时候，不妨考虑精简状态，看看哪些状态是一定不可能转移的，以此达到排除冗杂状态的目的。
- 设计 dp 阶段时一定不要拘泥于基本情况，要思考更深入的情况。
- 考虑 dp 的时候，一定不要考虑以后的阶段怎么处理，我们只关心怎么分割当前的子问题，只关心怎么样覆盖完状态空间。
- 对于一类计数 dp 问题，有一个很重要的前提条件：**一种合法的基本情况和一个合法的转移序列是唯一对应的**，这样，我们处理的信息天然就满足不重不漏性质，只需要保证转移合法，就能覆盖整个状态空间。
- 如果大步大步的转移比较困难，类似本题中的成段转移，不如考虑分割一下，变成小步小步的加入和新建，这样能大幅降低思考难度。

4.2 背包 DP

```cpp

// 01
int dp[si];

memset(dp, 0, sizeof dp);
for(int i = 1; i <= n; ++i)
  for(int j = m; j >= v[i]; --j)
    dp[j] = max(dp[j], dp[j - v[i]] + w[i]);
  // 因为直接共用一个数组了，所以不用手动继承上一个阶段了。
cout << dp[m] << endl;

// Complete
memset(dp, 0, sizeof dp);
for(int i = 1; i <= n; ++i)
    for(int j = v[i]; j <= m; ++j)
        dp[j] = max(dp[j], dp[j - v[i]] + w[i]);
cout << dp[m] << endl;

// Multi
for(int i = 1; i <= n; ++i)
    scanf("%d%d%d", &v[i], &w[i], &c[i]);
auto calc = [&](int u, int i, int k) - >int {
    return f[u + k * v[i]] - k * w[i]; };
for(int i = 1; i <= n; ++i) {
    for(int u = 0; u < v[i]; ++u) {
        int head = 1, tail = 0;
        memset(q, 0, sizeof q);
        int mxp = (m - u) / v[i];
        for(int k = mxp - 1; k >= max(mxp - c[i], 0); --k) {
            while(head <= tail && calc(u, i, q[tail]) <= calc(u, i, k)) tail -- ;
            q[++tail] = k;
        }
        for(int p = mxp; p >= 0; --p) {
            while(head <= tail && q[head]>p - 1) head++;
            if(head <= tail)
                f[u + p * v[i]] = max(f[u + p * v[i]], calc(u, i, q[head]) + p * w[i]);
            if(p - c[i] - 1 >= 0) {
                while(head <= tail && calc(u, i, q[tail]) <= calc(u, i, p - c[i] - 1)) tail -- ;
                q[++tail] = p - c[i] - 1;
            }
        }
    }
}
// Group
for(int i = 1; i <= n; ++i) // 枚举组
    for(int j = m; j >= 0; --j) // 枚举给每一个组分多少空间
        // 需要保证每组只选一个，所以要像 01 背包一样倒序循环。
        for(int k = 1; k <= c[i]; ++k) // 枚举这组选哪一个
            if(j >= v[i][k])
                dp[j] = max(dp[j], dp[j - v[i][k]] + w[i][k]);
// Tree
void dfs(int u, int fa) {
    for(int i = 0; i < (int)g[u].size(); ++i) {
        int ver = g[u][i];
        if(ver == fa) continue;
        dfs(ver, u);
    }
    for(int i = 0; i < (int)g[u].size(); ++i) {
        int ver = g[u][i];
        if(ver == fa) continue;
        for(int j = m - v[u]; j >= 0; --j) // 枚举排除 u 自己的，总共可以往下分配的空间
            for(int k = 0; k <= j; ++k) // 可以给当前子树分配的空间
                dp[u][j] = max(dp[u][j], dp[u][j - k] + dp[ver][k]);
    }
    // 上面的循环也可以放到 dfs 后面去
    for(int i = m; i >= v[u]; --i) // 强制选 u 的转移。
        dp[u][i] = dp[u][i - v[u]] + w[u];
    for(int i = 0; i < v[u]; ++i) // 连 u 的空间都无法满足，0
        dp[u][i] = 0;
}
```

4.3 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）
```

4.4 子集卷积/SOS DP

```
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)];
    }
}
```

4.5 斜率优化

可以斜率优化的方程通常具有以下形式：

$dp(i) = \min_{j=L(i)}^{R(i)} \{dp(j) + val(i,j)\}$，其中 $val(i,j)$ 为一个关于 i,j 的多项式，$L(i), R(i)$ 为一个关于 i 的函数，用于限制 j 的范围。

并且 $val(i,j)$ 存在形如 $i \times j$ 的项，与单调队列优化的仅有 i,j 项不同。

斜率优化的思想是，先拆掉 $L(i), R(i)$ 的限制，将所有决策点转化为二维平面上的点，将方程转化为一个一次函数来进行决策，在决策时再加上 $L(i), R(i)$ 的限制，具体来说，我们建立以下映射：

- 将仅和 j 相关的项看作 y，记这些项组成的多项式为 y_i，形如 $dp(j) + v(j) + \ldots$。- 将和 i,j 同时相关的项看作 k, x，其中 i 这一部分作为 k，记为 k_i，j 这一部分作为 x，记为 x_j，式子形如 $C_1 \times (C_2 - v(i)) \times w(j)$ (其中 C_1, C_2 为常量)，那么 $k_i = C_1 \times (C_2 - v(i)), x_j = w(j)$ - 将仅和 i 相关的项看作 b，记为 b_i，为了方便我们把常量也算进这一部分，式子形如 $dp(i) + v(i) \times w(i) + C$，我们要最小化的就是这一部分（本质是最小化 $dp(i)$，其它的是常量所以无所谓。）

(以上的式子只是做一个参考理解，需要根据实际情况来改变。)

然后，问题就转化为，给定一堆平面上的点 (x_j, y_j)，对于一条直线 $y = k_i x + b_i$，我们需要选择一个满足 $L(i) \le j \le R(i)$ 限制的 (x_j, y_j) 代入直线，使得 b_i 最小。

这部分的做法就是维护凸壳了。

对于 $L(i), R(i)$ 的下标限制：

- 如果是类似本题的 $0 \le j < i$，说明不需要删除决策点，而且每次只会在尾部插入决策，我们枚举就好了
- 如果 $L(i), R(i)$ 是随 i 单调变化的，我们就需要使用单调队列来排除冗杂
- 如果是没啥单调性的，也就是说一般要支持在任意位置插入删除决策点，就需要使用平衡树或者 CDQ 分治。

对于斜率的限制，只需要看斜率是否单调递增即可：

- 如果斜率随 i 单调递增，那么可以直接使用单调队列取队头转移。
- 如果斜率不随 i 单调递增，我们就需要在凸壳上二分答案找到最优决策点。

对于 x_j 的限制，只需要看它是否随 j 单调递增即可。

- 如果它随 j 单调递增，那么我们就只需要一个一个插入决策就行。
- 如果它不随 j 单调递增，那么我们就需要使用平衡树 / CDQ 分治来支持插入决策点的操作，注意 CDQ 维护的时候还要对前一半排序。

Last but not least: 如果使用交叉相乘来避免精度问题，要小心数据范围，如果直接使用浮点数，要记得，eps 不要开太小了，要视情况而定。

5. Math

5.1 线性求逆元以及组合数

```
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;
}
```

5.2 组合数性质

二项式定理：

$$(a+b)^n = \sum_{i=0}^{n} \binom{n}{i} a^{n-i} b^i$$

一些组合数性质：

I.

$$\binom{n}{m} = \binom{n}{n-m}$$

这个是显然的，因为你选 m 个和选 $n-m$ 个的情况是捆绑起来的。

II.

$$\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$$

这个也是显然的，根据定义展开就可以得到。

也可以写作

$$k\binom{n}{k} = n\binom{n-1}{k-1}$$

III.

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$$

这个就是组合数的递推式，也可以看作是杨辉三角。

IV.

$$\binom{n}{0} + \binom{n}{1} + \cdots + \binom{n}{n} = \sum_{i=0}^{n} \binom{n}{i} = 2^n$$

这是二项式定理的特殊情况。取 $a = b = 1$ 就可以了。

V.

$$\sum_{i=0}^{n} (-1)^i \binom{n}{i} = [n = 0]$$

二项式定理的另一种特殊情况，可取 $a=1,b=-1$。式子的特殊情况是取 $n=0$ 时答案为 1。
后面那个是 Iverson Bracket.
VI.

$$\sum_{i=0}^{m}\binom{n}{i}\binom{m}{m-i}=\binom{m+n}{m}\quad(n\geq m)$$

这个就是范德蒙德卷积的推论。
VII.

$$\sum_{i=0}^{n}\binom{n}{i}^2=\binom{2n}{n}$$

仍旧是范德蒙德卷积的推论.
VIII.

$$\sum_{l=0}^{n}\binom{l}{k}=\binom{n+1}{k+1}$$

通过组合分析——考虑 $S=a_1,a_2,\cdots,a_{n+1}$ 的 $k+1$ 子集数可以得证，在恒等式证明中比较常用。
IX.

$$\binom{n}{r}\binom{r}{k}=\binom{n}{k}\binom{n-k}{r-k}$$

用定义展开一下就可以证明了，式子形式很好记。
X.

$$\sum_{i=0}^{n}\binom{n-i}{i}=F_{n+1}$$

其中 F 是斐波那契数列。

5.3 范德蒙德卷积

范德蒙德卷积

$$\sum_{i=0}^{k}\binom{n}{i}\binom{m}{k-i}=\binom{n+m}{k}$$

也可以写成：

$$\sum_{i=-r}^{s}\binom{n}{r+i}\binom{m}{s-i}=\binom{n+m}{r+s}$$

5.4 快速幂

```
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;
}
```

5.5 矩阵乘法

```
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。
```

5.6 高斯消元

```
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;
}
```

5.7 质数

```
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];
        }
```

```cpp
    }
}

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;
}
```

5.8 约数

```cpp
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; ++j) {
          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);
      }
  }
}
```

5.9 Exgcd

```cpp
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);
}
```

5.10 中国剩余定理

```cpp
#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;
}
```

5.11 ExCRT

考虑两个方程怎么做。
假设 $x \equiv a_1 \pmod{m_1}, x \equiv a_2 \pmod{m_2}$。
按照类似 gcd 那边的套路：
$x = m_1 p + a_1 = m_2 q + a_2, p, q \in \mathbb{Z}$。
然后可以知道 $m_1 p - m_2 q = a_2 - a_1$，然后这东西就是类似线性同余的东西。有解当且仅当 $\gcd(m_1, m_2) \mid a_2 - a_1$。
然后就 exgcd 解一下，显然这两个方程的解应该是 $m_2 q + a_2 \pmod{\operatorname{lcm}(m_1, m_2)}$。
然后我们就直接合并多个方程就可以了。

5.12 Lucas 定理

对于任意质数 p，存在如下定理：

$$\binom{n}{m} \equiv \binom{\lfloor \frac{n}{p} \rfloor}{\lfloor \frac{m}{p} \rfloor} \cdot \binom{n \bmod p}{m \bmod p} \pmod{p}$$

5.13 Ex Lucas

考虑类似 exCRT 的经典套路，我们分解质因数，用 CRT 构造一个方程然后合并，这样每个方程里面都是一个 Lucas。
也就是，令 $p = \prod_{i=1}^{r} a_r^{c_r}$。
然后因为任意 $a_i^{c_i}, a_j^{c_j}$ 互质，所以我们把他们当作模数。
由 CRT，令 $\binom{n}{m}$ 为未知数，有：

$$\begin{cases} c_1 & \equiv \binom{n}{m} \pmod{a_1^{c_1}} \\ c_2 & \equiv \binom{n}{m} \pmod{a_2^{c_2}} \\ & \dots \\ c_r & \equiv \binom{n}{m} \pmod{a_r^{c_r}} \end{cases}$$

可以由此解出未知数在模 p 意义下的唯一解 $\binom{n}{m} \bmod p$

5.14 数论相关结论

算术基本定理：任何一个大于 1 的正整数都可以唯一分解为有限个质数的乘积。
也叫唯一分解定理，可以写成 $N = p_1^{c1} \times p_2^{c2} \times p_3^{c3} \times \dots p_m^{cm}, c_i \in \mathbb{N}^*, p_i < p_{i+1}, \mathrm{PRIME}(p_i)$。
唯一分解定理的三个推论：
1. 若 $n \in \mathbb{N}^*$，则 n 的正约数集合为 $\{x | x = p_1^{b_1} p_2^{b_2} \dots p_m^{b_m}, b_i \le c_i\}$。
2. n 的正约数个数为 $\prod_{i=1}^{m}(c_i + 1)$ 3. n 的正约数之和为 $(1 + p_1 + p_1^2 + p_1^3 + \dots + p_1^{c_1}) \times \dots (1 + p_m + p_m^2 + p_m^3 + \dots + p_m^{c_m}) = \prod_{i=1}^{m} (\sum_{j=1}^{c_i} p_i^j)$
约数个数和结论：$1 \sim n$ 所有数的约数个数之和约为 $n \log n$ 个
更相减损术:

- $\forall a, b \in \mathbb{N}, a \ge b$ 有 $\gcd(a, b) = \gcd(b, a - b) = \gcd(a, a - b)$
- $\forall a, b \in \mathbb{N}$，有 $\gcd(2a, 2b) = 2\gcd(a, b)$。

欧拉函数的性质
性质1：$\forall n > 1, 1 \sim n$ 中与 n 互质的数的和为 $n \cdot \varphi(n)/2$。
由更相减损术，和 n 互质的数必然成对出现，且均值为 $n/2$，证毕。
性质2：欧拉函数为积性函数，且有：$\gcd(a, b) = 1 \Rightarrow \varphi(ab) = \varphi(a) \cdot \varphi(b)$。
展开计算式就行了。
性质3：(积性函数的性质)：在唯一分解定理背景下，若 f 为积性函数，则有：$f(n) = \prod_{i=1}^{m} f(p_i^{c_i})$
显然任意的 $p_i^{c_i}$ 和 $p_j^{c_j}$ 必然互质，由积性函数的性质，对整体应用结论，可以得到原式。
性质4：若 p 为质数，若 $p | n$ 且 $p^2 \not| n$，则 $\varphi(n) = \varphi(n/p)\varphi(p) = \varphi(n/p) \cdot (p - 1)$。
积性函数的性质，显然，常用于递推。
性质5：若 p 为质数，若 $p | n$ 且 $p^2 | n$，则 $\varphi(n) = \varphi(n/p) \times p$
因为 n/p 和 p 不互质，所以只能展开计算式得到，常用于递推。
性质6：$\sum_{d|n} \varphi(d) = n$。
很有意思的性质，先对 n 分解质因数，令 $f(x) = \sum_{d|x} \varphi(d)$。
显然 $f(p_i^{c_i}) = \varphi(1) + \varphi(p_i) + \varphi(p_i^2) + \dots + \varphi(p_i^{c_i}) = p_i^{c_i}$（由性质 5 可以发现是一个等比数列求和）。

然后发现若 $\gcd(n,m)=1$，$f(nm)=(\sum_{d|n}\varphi(d))\cdot(\sum_{d|m}\varphi(d))=f(n)f(m)$。
所以 f 是积性函数，由积性函数性质可以得到原式成立。
结论 1：对于足够大的 $n\in\mathbb{N}^*$，$[1,n]$ 中的素数个数约为 $\frac{n}{\ln n}$ 个（**对数结论**）
结论 2：若 $\neg\mathrm{PRIME}(n)$，则 $\exists T\in[2,\sqrt{n}]$，使得 $T|n$。（**根号结论**）

5.15 容斥原理

若有 n 个集合 $S_1\dots S_n$，并且集合之间可能有交集。
那么 $|\bigcup S_i|$ 就等于 $\sum_i |S_i| - \sum_{i,j}|S_i\cap S_j| + \sum_{i,j,k}|S_i\cap S_j\cap S_k|\cdots + (-1)^{n+1}\sum_{a_1,\dots a_n}|\bigcap_j S_{a_j}|$。
$a_1,\dots a_n$ 是用来枚举集合的。
这个柿子也可以简述为，多个集合的并集大小等于奇数个集合的交集的大小之和减去偶数个集合的交集大小之和。
或者描述为:
$\sum$ 在任意一个集合内的元素个数总和 $\sum$ 在任意两个集合交内的元素个数总和 $+\sum$ 在任意三个集合交内的元素个数总和...
注意这里"在任意两个集合交集内的元素个数总和"是要算重的，也就是说如果 x 在 $A\cap B\cap C$ 当中，那么在算任意两个集合交内的元素个数总和时，x 的贡献就是 3。
这么做其实就是为了方便计数，因为有多个条件但是只是"至少"满足一个或者几个的时候，无法比较方便的知道哪些条件满足，哪些条件不满足。
所以我们直接只考虑某些特定的条件一定被满足的时候方案数，其它的直接不管怎么搞，反正不合法或者重复的肯定会被容斥掉。
这就是一种"至少转强制"的思想。

5.16 二项式反演

用于解决"某个物品恰好若干个"的一类问题。
设 $g(n)$ 表示至多 n 种的方案数，$f(n)$ 表示恰好 n 种的方案数，则有

$$g(n)=\sum_{i=0}^{n}\binom{n}{i}f(i)\iff f(n)=\sum_{i=0}^{n}(-1)^{n-i}\binom{n}{i}g(i)$$

设 $g(n)$ 表示至少 n 种的方案数，$f(n)$ 表示恰好 n 种的方案数，则有

$$g(n)=\sum_{i=k}^{n}\binom{i}{k}f(i)\iff f(k)=\sum_{i=k}^{n}(-1)^{i-k}\binom{i}{k}g(i)$$

6. Graph

6.1 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;
```

6.2 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
        }
    }
}
```

6.3 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;
```

```cpp
    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;
}
```

6.4 Kruskal

```cpp
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

```cpp
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];
```

6.5 倍增 LCA

```cpp
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];
}
```

6.6 Tarjan LCA

```cpp
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;
}
```

6.7 拓扑排序

```cpp
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。
}
```

6.8 欧拉回路

```cpp
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。
```

6.9 强连通分量

```cpp

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);
```

6.10 边双连通分量

```cpp
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();
}
```

6.11 点双连通分量

```cpp
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) {
```

```cpp
        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;
}
```

6.12 虚树

```cpp
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;
}
```

7. String

7.1 Kmp

```cpp
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;
}
```

7.2 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];
}
```

7.3 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) {
```

```cpp
    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;
}
```

7.4 Ac Automaton

```cpp
// 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];
            }
        }
```

```cpp
 | }
 | 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;
```

8. Misc

8.1 莫队

```cpp
int n, Q, unit;
int a[si];
struct Query {
    int l, r, id;
    bool operator < (const Query &b) const {
        if((l / unit) != (b.l / unit))
            return l < b.l;
        if((l / unit) & 1)
            return r < b.r;
        return r > b.r;
    }
}ask[si];
void add(int pos) { }
void sub(int pos) { }

int main() {
    std::vector<int> v; v.clear();
    cin >> n, unit = sqrt(n);
    for(int i = 1; i <= n; ++i)
        cin >> a[i], v.emplace_back(a[i]);
    sort(v.begin(), v.end()), v.erase(unique(v.begin(), v.end()), v.end());
    for(int i = 1; i <= n; ++i)
        a[i] = lower_bound(v.begin(), v.end(), a[i]) - v.begin();

    cin >> Q;
    for(int i = 1; i <= Q; ++i)
        cin >> ask[i].l >> ask[i].r,
        ask[i].id = i;
    sort(ask + 1, ask + 1 + Q);

    int l = 1, r = 0;
    for(int i = 1; i <= Q; ++i) {
        Query &q = ask[i];
        while(l > q.l) add(--l);
        while(r < q.r) add(++r);
        while(l < q.l) sub(l++);
        while(r > q.r) sub(r--);
        res[q.id] = ans;
    }

    for(int i = 1; i <= Q; ++i)
        cout << res[i] << endl;
    return 0;
}
```

Good Luck && Have Fun!