数据结构A

笛卡尔树

小根笛卡尔树

```
cin >> n;
for (int i = 0;i < n;++i)cin >> nums[i];
for (int i = 0;i < n;++i)rs[i] = -1;
for (int i = 0;i < n;++i)ls[i] = -1;
top = 0;
for (int i = 0; i < n; i++) {
    int k = top;
    while (k > 0 && nums[stk[k - 1]] > nums[i]) k--;
    if (k) rs[stk[k - 1]] = i;  // rs代表笛卡尔树每个节点的右儿子
    if (k < top) ls[i] = stk[k];  // ls代表笛卡尔树每个节点的左儿子
    stk[k++] = i;
    top = k;
}
```

dsu并查集

路径优化(普遍)

```
struct dsu {
    std::vector<int> d;
    dsu(int n) { d.resize(n + 1); iota(d.begin(), d.end(), 0); }
    int get_root(int x) { return d[x] = (x == d[x] ? x : get_root(d[x]));
};
    bool merge(int u, int v) {
        if (get_root(u) != get_root(v)) {
            d[get_root(u)] = get_root(v);
            return true;
        }
        else return false;
    }
};
```

根据集合的大小优化

```
//左移位数根据节点个数定
#define UFLIMIT (2<<17)
int unicnt[UFLIMIT];
void ufinit(int n) {
    for (int i = 0;i < n;i++)unicnt[i] = 1;
}
int ufroot(int x) { return unicnt[x] <= 0 ? -(unicnt[x] = -ufroot(-
unicnt[x])) : x; }
int ufsame(int x, int y) { return ufroot(x) == ufroot(y); }
void uni(int x, int y) {
    if ((x = ufroot(x)) == (y = ufroot(y)))return;
    if (unicnt[x] < unicnt[y])std::swap(x, y);
```

```
    unicnt[x] += unicnt[y];
    unicnt[y] = -x;
}
```

按秩合并优化

```
class UnionFind {
private:
    std::vector<int> parent;
    std::vector<int> rank;
public:
    UnionFind(int n) {
        parent.resize(n, 0);
        rank.resize(n, 0);
        iota(parent.begin(), parent.end(), 0);
    }
    int find(int x) {
        if (parent[x] == x)
            return x;
        return parent[x] = find(parent[x]);
    }
    void merge(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX == rootY) return;
        if (rank[rootX] > rank[rootY])
            std::swap(rootX, rootY);
        parent[rootX] = rootY;
        if (rank[rootX] == rank[rootY]) {
            rank[rootY]++;
        }
    }
    bool isConnect(int x, int y) {
        return find(x) == find(y);
    }
};
```

常用操作

```
struct DSU {
    vector<int> fa, p, e, f;

    DSU(int n) {
        fa.resize(n + 1);
        iota(fa.begin(), fa.end(), 0);
        p.resize(n + 1, 1);
        e.resize(n + 1);
        f.resize(n + 1);
    }
    int get(int x) {
        while (x != fa[x]) {
            x = fa[x] = fa[fa[x]];
        }
        return x;
```

```
    }
    bool merge(int x, int y) { // 设x是y的祖先
        if (x == y) f[get(x)] = 1;
        x = get(x), y = get(y);
        e[x]++;
        if (x == y) return false;
        if (x < y) swap(x, y); // 将编号小的合并到大的上
        fa[y] = x;
        f[x] |= f[y], p[x] += p[y], e[x] += e[y];
        return true;
    }
    bool same(int x, int y) {
        return get(x) == get(y);
    }
    bool F(int x) { // 判断连通块内是否存在自环
        return f[get(x)];
    }
    int size(int x) { // 输出连通块中点的数量
        return p[get(x)];
    }
    int E(int x) { // 输出连通块中边的数量
        return e[get(x)];
    }
};
```

ST 表

用于解决区间可重复贡献问题，需要满足 x 运算符 $x = x$ （如区间最大值：$\max(x, x) = x$ 、区间 gcd：$\gcd(x, x) = x$ 等），但是不支持修改操作。$\mathcal{O}(N \log N)$ 预处理，$\mathcal{O}(1)$ 查询。

```
struct ST {
    const int n, k;
    vector<int> in1, in2;
    vector<vector<int>> Max, Min;
    ST(int n) : n(n), in1(n + 1), in2(n + 1), k(__lg(n)) {
        Max.resize(k + 1, vector<int>(n + 1));
        Min.resize(k + 1, vector<int>(n + 1));
    }
    void init() {
        for (int i = 1; i <= n; i++) {
            Max[0][i] = in1[i];
            Min[0][i] = in2[i];
        }
        for (int i = 0, t = 1; i < k; i++, t <<= 1) {
            const int T = n - (t << 1) + 1;
            for (int j = 1; j <= T; j++) {
                Max[i + 1][j] = max(Max[i][j], Max[i][j + t]);
                Min[i + 1][j] = min(Min[i][j], Min[i][j + t]);
            }
        }
    }
    int getMax(int l, int r) {
        if (l > r) {
            swap(l, r);
        }
```

```
        int k = __lg(r - l + 1);
        return max(Max[k][l], Max[k][r - (1 << k) + 1]);
    }
    int getMin(int l, int r) {
        if (l > r) {
            swap(l, r);
        }
        int k = __lg(r - l + 1);
        return min(Min[k][l], Min[k][r - (1 << k) + 1]);
    }
};
```

Fenwick Tree 树状数组

```
template<class T> struct BIT {
    int n;
    vector<T> w;
    BIT(int n, auto &in) : n(n), w(n + 1) { // 预处理填值
        for (int i = 1; i <= n; i++) {
            add(i, in[i]);
        }
    }
    void add(int x, T v) {
        for (; x <= n; x += x & -x) {
            w[x] += v;
        }
    }
    T ask(int x) { // 前缀和查询
        T ans = 0;
        for (; x; x -= x & -x) {
            ans += w[x];
        }
        return ans;
    }
    T ask(int l, int r) { // 差分实现区间和查询
        return ask(r) - ask(l - 1);
    }
};
```

逆序对扩展

```
struct BIT {
    int n;
    vector<int> w, chk; // chk 为传入的待处理数组
    BIT(int n, auto &in) : n(n), w(n + 1), chk(in) {}
    /* 需要全部常规封装 */
    int get() {
        vector<array<int, 2>> alls;
        for (int i = 1; i <= n; i++) {
            alls.push_back({chk[i], i});
        }
        sort(alls.begin(), alls.end());
        int ans = 0;
        for (auto [val, idx] : alls) {
```

```
            ans += ask(idx + 1, n);
            add(idx, 1);
        }
        return ans;
    }
};
```

前驱后继扩展（常规+第 k 小值查询+元素排名查询+元素前驱后继查询）

注意，被查询的值都应该小于等于 N ，否则会越界；如果离散化不可使用，则需要使用平衡树替代。

```
struct BIT {
    int n;
    vector<int> w;
    BIT(int n) : n(n), w(n + 1) {}
    void add(int x, int v) {
        for (; x <= n; x += x & -x) {
            w[x] += v;
        }
    }
    int kth(int x) { // 查找第 k 小的值
        int ans = 0;
        for (int i = __lg(n); i >= 0; i--) {
            int val = ans + (1 << i);
            if (val < n && w[val] < x) {
                x -= w[val];
                ans = val;
            }
        }
        return ans + 1;
    }
    int get(int x) { // 查找 x 的排名
        int ans = 1;
        for (x--; x; x -= x & -x) {
            ans += w[x];
        }
        return ans;
    }
    int pre(int x) { return kth(get(x) - 1); } // 查找 x 的前驱
    int suf(int x) { return kth(get(x + 1)); } // 查找 x 的后继
};
const int N = 10000000; // 可以用于在线处理平衡二叉树的全部要求
signed main() {
    BIT bit(N + 1); // 在线处理不能够离散化，一定要开到比最大值更大
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++) {
        int op, x;
        cin >> op >> x;
        if (op == 1) bit.add(x, 1); // 插入 x
        else if (op == 2) bit.add(x, -1); // 删除任意一个 x
        else if (op == 3) cout << bit.get(x) << "\n"; // 查询 x 的排名
        else if (op == 4) cout << bit.kth(x) << "\n"; // 查询排名为 x 的数
        else if (op == 5) cout << bit.pre(x) << "\n"; // 求小于 x 的最大值（前
驱）
```

```
        else if (op == 6) cout << bit.suf(x) << "\n"; // 求大于 x 的最小值（后继）
    }
}
```

最值查询扩展（常规+区间最值查询+单点赋值）

以 $\mathcal{O}(\log \log N)$ 的复杂度运行，但是即便如此依然略优于线段树（后者常数较大）。

```
template<class T> struct BIT {
    int n;
    vector<T> w, base;
    #define low(x) (x & -x)
    BIT(int n, auto &in) : n(n), w(n + 1), base(n + 1) {
        for (int i = 1; i <= n; i++) {
            update(i, in[i]);
        }
    } /* 可以增加并使用常规封装中的几个函数 */
    void update(int x, int v) { // 单点赋值
        base[x] = max(base[x], v);
        for (; x <= n; x += low(x)) {
            w[x] = max(w[x], v);
        }
    }
    T getMax(int l, int r) { // 最值查询
        T ans = T();
        while (r >= l) {
            ans = max(base[r], ans);
            for (r--; r - low(r) >= l; r -= low(r)) {
                ans = max(w[r], ans);
            }
        }
        return ans;
    }
};
```

二维树状数组

封装一：该版本不能同时进行区间修改+区间查询。无离散化版本的空间占用为 $\mathcal{O}(NM)$、建树复杂度为 $\mathcal{O}(NM)$、单次查询复杂度为 $\mathcal{O}(\log N \cdot \log M)$。

```
struct BIT_2D {
    int n, m;
    vector<vector<int>> w;

    BIT_2D(int n, int m) : n(n), m(m) {
        w.resize(n + 1, vector<int>(m + 1));
    }
    void add(int x, int y, int k) {
        for (int i = x; i <= n; i += i & -i) {
            for (int j = y; j <= m; j += j & -j) {
                w[i][j] += k;
            }
        }
```

```
    }
    void add(int x, int y, int X, int Y, int k) { // 区块修改：二维差分
        X++, Y++;
        add(x, y, k), add(X, y, -k);
        add(X, Y, k), add(x, Y, -k);
    }
    int ask(int x, int y) { // 单点查询
        int ans = 0;
        for (int i = x; i; i -= i & -i) {
            for (int j = y; j; j -= j & -j) {
                ans += w[i][j];
            }
        }
        return ans;
    }
    int ask(int x, int y, int X, int Y) { // 区块查询：二维前缀和
        x--, y--;
        return ask(X, Y) - ask(x, Y) - ask(X, y) + ask(x, y);
    }
};
```

封装二：该版本支持全部操作。但是时空复杂度均比上一个版本多 4 倍。

```
struct BIT_2D {
    int n, m;
    vector<vector<int>> b1, b2, b3, b4;

    BIT_2D(int n, int m) : n(n), m(m) {
        b1.resize(n + 1, vector<int>(m + 1));
        b2.resize(n + 1, vector<int>(m + 1));
        b3.resize(n + 1, vector<int>(m + 1));
        b4.resize(n + 1, vector<int>(m + 1));
    }
    void add(auto &w, int x, int y, int k) { // 单点修改
        for (int i = x; i <= n; i += i & -i) {
            for (int j = y; j <= m; j += j & -j) {
                w[i][j] += k;
            }
        }
    }
    void add(int x, int y, int k) { // 多了一步计算
        add(b1, x, y, k);
        add(b2, x, y, k * (x - 1));
        add(b3, x, y, k * (y - 1));
        add(b4, x, y, k * (x - 1) * (y - 1));
    }
    void add(int x, int y, int X, int Y, int k) { // 区块修改：二维差分
        X++, Y++;
        add(x, y, k), add(X, y, -k);
        add(X, Y, k), add(x, Y, -k);
    }
    int ask(auto &w, int x, int y) { // 单点查询
        int ans = 0;
        for (int i = x; i; i -= i & -i) {
            for (int j = y; j; j -= j & -j) {
```

```
                ans += w[i][j];
            }
        }
        return ans;
    }
    int ask(int x, int y) { // 多了一步计算
        int ans = 0;
        ans += x * y * ask(b1, x, y);
        ans -= y * ask(b2, x, y);
        ans -= x * ask(b3, x, y);
        ans += ask(b4, x, y);
        return ans;
    }
    int ask(int x, int y, int X, int Y) { // 区块查询：二维前缀和
        x--, y--;
        return ask(X, Y) - ask(x, Y) - ask(X, y) + ask(x, y);
    }
};
```

线段树

快速线段树（单点修改+区间最值）

```
struct Segt {
    vector<int> w;
    int n;
    Segt(int n) : w(2 * n, (int)-2E9), n(n) {}

    void modify(int pos, int val) {
        for (w[pos += n] = val; pos > 1; pos /= 2) {
            w[pos / 2] = max(w[pos], w[pos ^ 1]);
        }
    }

    int ask(int l, int r) {
        int res = -2E9;
        for (l += n, r += n; l < r; l /= 2, r /= 2) {
            if (l % 2) res = max(res, w[l++]);
            if (r % 2) res = max(res, w[--r]);
        }
        return res;
    }
};
```

区间加法修改、区间最小值查询 已修正 2025.7.19

```
template<class T> struct Segt {
    struct node {
        int l, r;
        T w, rmq, lazy;
    };
    std::vector<T> w;
    std::vector<node> t;

```

```
    Segt() {}
    Segt(int n) { init(n); }
    Segt(std::vector<int> in) {
        int n = in.size() - 1;
        w.resize(n + 1);
        for (int i = 1; i <= n; i++) {
            w[i] = in[i];
        }
        init(in.size() - 1);
    }

#define GL (k << 1)
#define GR (k << 1 | 1)

    void init(int n) {
        w.resize(n + 1);
        t.resize(n * 4 + 1);
        auto build = [&](auto self, int l, int r, int k = 1) {
            if (l == r) {
                t[k] = { l, r, w[l], w[l], 0 }; // 如果有赋值为 0 的操作，则懒标记必须要 -1
                return;
            }
            t[k] = { l, r, 0, 0, 0 };
            int mid = (l + r) / 2;
            self(self, l, mid, GL);
            self(self, mid + 1, r, GR);
            pushup(k);
            };
        build(build, 1, n);
    }
    void pushdown(node& p, T lazy) { /* 【在此更新下递函数】 */
        p.w += (p.r - p.l + 1) * lazy;
        p.rmq += lazy;
        p.lazy += lazy;
    }
    void pushdown(int k) {
        if (t[k].lazy == 0) return;
        pushdown(t[GL], t[k].lazy);
        pushdown(t[GR], t[k].lazy);
        t[k].lazy = 0;
    }
    void pushup(int k) {
        auto pushup = [&](node& p, node& l, node& r) { /* 【在此更新上传函数】 */
            p.w = l.w + r.w;
            p.rmq = std::min(l.rmq, r.rmq); // RMQ -> min/max
            };
        pushup(t[k], t[GL], t[GR]);
    }
    void modify(int l, int r, T val, int k = 1) { // 区间修改
        if (l <= t[k].l && t[k].r <= r) {
            pushdown(t[k], val);
            return;
        }
        pushdown(k);
```

```
        int mid = (t[k].l + t[k].r) / 2;
        if (l <= mid) modify(l, r, val, GL);
        if (mid < r) modify(l, r, val, GR);
        pushup(k);
    }
    T rmq(int l, int r, int k = 1) { // 区间询问最小值
        if (l <= t[k].l && t[k].r <= r) {
            return t[k].rmq;
        }
        pushdown(k);
        int mid = (t[k].l + t[k].r) / 2;
        T ans = std::numeric_limits<T>::max(); // RMQ -> 为 max 时需要修改为
::lowest()
        if (l <= mid) ans = std::min(ans, rmq(l, r, GL)); // RMQ -> min/max
        if (mid < r) ans = std::min(ans, rmq(l, r, GR)); // RMQ -> min/max
        return ans;
    }
    T ask(int l, int r, int k = 1) { // 区间询问
        if (l <= t[k].l && t[k].r <= r) {
            return t[k].w;
        }
        pushdown(k);
        int mid = (t[k].l + t[k].r) / 2;
        T ans = 0;
        if (l <= mid) ans += ask(l, r, GL);
        if (mid < r) ans += ask(l, r, GR);
        return ans;
    }
    void debug(int k = 1) {
        std::cout << "[" << t[k].l << ", " << t[k].r << "]: ";
        std::cout << "w = " << t[k].w << ", ";
        std::cout << "Min = " << t[k].rmq << ", ";
        std::cout << "lazy = " << t[k].lazy << ", ";
        std::cout << std::endl;
        if (t[k].l == t[k].r) return;
        debug(GL), debug(GR);
    }
};
```

同时需要处理区间加法与乘法修改

```
template <class T> struct Segt_ {
    struct node {
        int l, r;
        T w, add, mul = 1; // 注意初始赋值
    };
    vector<T> w;
    vector<node> t;

    Segt_(int n) {
        w.resize(n + 1);
        t.resize((n << 2) + 1);
        build(1, n);
    }
    Segt_(vector<int> in) {
```

```
        int n = in.size() - 1;
        w.resize(n + 1);
        for (int i = 1; i <= n; i++) {
            w[i] = in[i];
        }
        t.resize((n << 2) + 1);
        build(1, n);
    }
    void pushdown(node &p, T add, T mul) { // 在此更新下递函数
        p.w = p.w * mul + (p.r - p.l + 1) * add;
        p.add = p.add * mul + add;
        p.mul *= mul;
    }
    void pushup(node &p, node &l, node &r) { // 在此更新上传函数
        p.w = l.w + r.w;
    }
#define GL (k << 1)
#define GR (k << 1 | 1)
    void pushdown(int k) { // 不需要动
        pushdown(t[GL], t[k].add, t[k].mul);
        pushdown(t[GR], t[k].add, t[k].mul);
        t[k].add = 0, t[k].mul = 1;
    }
    void pushup(int k) { // 不需要动
        pushup(t[k], t[GL], t[GR]);
    }
    void build(int l, int r, int k = 1) {
        if (l == r) {
            t[k] = {l, r, w[l]};
            return;
        }
        t[k] = {l, r};
        int mid = (l + r) / 2;
        build(l, mid, GL);
        build(mid + 1, r, GR);
        pushup(k);
    }
    void modify(int l, int r, T val, int k = 1) { // 区间修改
        if (l <= t[k].l && t[k].r <= r) {
            t[k].w += (t[k].r - t[k].l + 1) * val;
            t[k].add += val;
            return;
        }
        pushdown(k);
        int mid = (t[k].l + t[k].r) / 2;
        if (l <= mid) modify(l, r, val, GL);
        if (mid < r) modify(l, r, val, GR);
        pushup(k);
    }
    void modify2(int l, int r, T val, int k = 1) { // 区间修改
        if (l <= t[k].l && t[k].r <= r) {
            t[k].w *= val;
            t[k].add *= val;
            t[k].mul *= val;
            return;
        }
```

```
        pushdown(k);
        int mid = (t[k].l + t[k].r) / 2;
        if (l <= mid) modify2(l, r, val, GL);
        if (mid < r) modify2(l, r, val, GR);
        pushup(k);
    }
    T ask(int l, int r, int k = 1) { // 区间询问，不合并
        if (l <= t[k].l && t[k].r <= r) {
            return t[k].w;
        }
        pushdown(k);
        int mid = (t[k].l + t[k].r) / 2;
        T ans = 0;
        if (l <= mid) ans += ask(l, r, GL);
        if (mid < r) ans += ask(l, r, GR);
        return ans;
    }
    void debug(int k = 1) {
        cout << "[" << t[k].l << ", " << t[k].r << "]: ";
        cout << "w = " << t[k].w << ", ";
        cout << "add = " << t[k].add << ", ";
        cout << "mul = " << t[k].mul << ", ";
        cout << endl;
        if (t[k].l == t[k].r) return;
        debug(GL), debug(GR);
    }
};
```

区间赋值/推平

如果存在推平为 0 的操作，那么需要将 `lazy` 初始赋值为 -1 。

```
void pushdown(node &p, T lazy) { /* 【在此更新下递函数】 */
    p.w = (p.r - p.l + 1) * lazy;
    p.lazy = lazy;
}
void modify(int l, int r, T val, int k = 1) {
    if (l <= t[k].l && t[k].r <= r) {
        t[k].w = val;
        t[k].lazy = val;
        return;
    }
    // 剩余部分不变
}
```

区间取模

原题需要进行“单点赋值+区间取模+区间求和” See 。该操作不需要懒标记。

需要额外维护一个区间最大值，当模数大于区间最大值时剪枝，否则进行单点取模。由于单点 $\texttt{MOD} < x$ 时 $x \bmod \texttt{MOD} < \frac{x}{2}$ ，故单点取模至 0 最劣只需要 $\log x$ 次 。

```
void modifyMod(int l, int r, T val, int k = 1) {
    if (l <= t[k].l && t[k].r <= r) {
        if (t[k].rmq < val) return; // 重要剪枝
```

```
    }
    if (t[k].l == t[k].r) {
        t[k].w %= val;
        t[k].rmq %= val;
        return;
    }
    int mid = (t[k].l + t[k].r) / 2;
    if (l <= mid) modifyMod(l, r, val, GL);
    if (mid < r) modifyMod(l, r, val, GR);
    pushup(k);
}
```

区间异或修改

原题需要维护"区间异或修改+区间求和" See 。

```
struct Segt { // #define GL (k << 1) // #define GR (k << 1 | 1)
    struct node {
        int l, r;
        int w[N], lazy; // 注意这里为了方便计算，w 只需要存位
    };
    vector<int> base;
    vector<node> t;

    Segt(vector<int> in) : base(in) {
        int n = in.size() - 1;
        t.resize(n * 4 + 1);
        auto build = [&](auto self, int l, int r, int k = 1) {
            t[k] = {l, r}; // 前置赋值
            if (l == r) {
                for (int i = 0; i < N; i++) {
                    t[k].w[i] = base[l] >> i & 1;
                }
                return;
            }
            int mid = (l + r) / 2;
            self(self, l, mid, GL);
            self(self, mid + 1, r, GR);
            pushup(k);
        };
        build(build, 1, n);
    }
    void pushdown(node &p, int lazy) { /* 【在此更新下递函数】 */
        int len = p.r - p.l + 1;
        for (int i = 0; i < N; i++) {
            if (lazy >> i & 1) { // 即 p.w = (p.r - p.l + 1) - p.w;
                p.w[i] = len - p.w[i];
            }
        }
        p.lazy ^= lazy;
    }
    void pushdown(int k) { // 【不需要动】
        if (t[k].lazy == 0) return;
        pushdown(t[GL], t[k].lazy);
        pushdown(t[GR], t[k].lazy);
```

```
        t[k].lazy = 0;
    }
    void pushup(int k) {
        auto pushup = [&](node &p, node &l, node &r) { /* 【在此更新上传函数】
*/
            for (int i = 0; i < N; i++) {
                p.w[i] = l.w[i] + r.w[i]; // 即 p.w = l.w + r.w;
            }
        };
        pushup(t[k], t[GL], t[GR]);
    }
    void modify(int l, int r, int val, int k = 1) { // 区间修改
        if (l <= t[k].l && t[k].r <= r) {
            pushdown(t[k], val);
            return;
        }
        pushdown(k);
        int mid = (t[k].l + t[k].r) / 2;
        if (l <= mid) modify(l, r, val, GL);
        if (mid < r) modify(l, r, val, GR);
        pushup(k);
    }
    i64 ask(int l, int r, int k = 1) { // 区间求和
        if (l <= t[k].l && t[k].r <= r) {
            i64 ans = 0;
            for (int i = 0; i < N; i++) {
                ans += t[k].w[i] * (1LL << i);
            }
            return ans;
        }
        pushdown(k);
        int mid = (t[k].l + t[k].r) / 2;
        i64 ans = 0;
        if (l <= mid) ans += ask(l, r, GL);
        if (mid < r) ans += ask(l, r, GR);
        return ans;
    }
};
```

拆位运算

原题同上。使用若干棵线段树维护每一位的值，区间异或转变为区间翻转。

```
template<class T> struct Segt_ { // GL 为 (k << 1)，GR 为 (k << 1 | 1)
    struct node {
        int l, r;
        T w;
        bool lazy; // 注意懒标记用布尔型足以
    };
    vector<T> w;
    vector<node> t;

    Segt_() {}
    void init(vector<int> in) {
        int n = in.size() - 1;
```

```
        w.resize(n * 4 + 1);
        for (int i = 0; i <= n; i++) { w[i] = in[i]; }
        t.resize(n * 4 + 1);
        build(1, n);
    }
    void pushdown(node &p, bool lazy = 1) { // 【在此更新下递函数】
        p.w = (p.r - p.l + 1) - p.w;
        p.lazy ^= lazy;
    }
    void pushup(node &p, node &l, node &r) { // 【在此更新上传函数】
        p.w = l.w + r.w;
    }
    void pushdown(int k) { // 【不需要动】
        if (t[k].lazy == 0) return;
        pushdown(t[GL]), pushdown(t[GR]); // 注意这里不再需要传入第二个参数
        t[k].lazy = 0;
    }
    void pushup(int k) { pushup(t[k], t[GL], t[GR]); } // 【不需要动】
    void build(int l, int r, int k = 1) {
        if (l == r) {
            t[k] = {l, r, w[l], 0}; // 注意懒标记初始为 0
            return;
        }
        t[k] = {l, r};
        int mid = (l + r) / 2;
        build(l, mid, GL);
        build(mid + 1, r, GR);
        pushup(k);
    }
    void reverse(int l, int r, int k = 1) { // 区间翻转
        if (l <= t[k].l && t[k].r <= r) {
            pushdown(t[k], 1);
            return;
        }
        pushdown(k);
        int mid = (t[k].l + t[k].r) / 2;
        if (l <= mid) reverse(l, r, GL);
        if (mid < r) reverse(l, r, GR);
        pushup(k);
    }
    T ask(int l, int r, int k = 1) { // 区间求和
        if (l <= t[k].l && t[k].r <= r) {
            return t[k].w;
        }
        pushdown(k);
        int mid = (t[k].l + t[k].r) / 2;
        T ans = 0;
        if (l <= mid) ans += ask(l, r, GL);
        if (mid < r) ans += ask(l, r, GR);
        return ans;
    }
};
signed main() {
    int n; cin >> n;
    vector in(20, vector<int>(n + 1));
    Segt_<i64> segt[20]; // 拆位建线段树
```

```
    for (int i = 1, x; i <= n; i++) { cin >> x;
        for (int bit = 0; bit < 20; bit++) {
            in[bit][i] = x >> bit & 1;
        }
    }
    for (int i = 0; i < 20; i++) {
        segt[i].init(in[i]);
    }

    int m, op;
    for (cin >> m; m; m--) { cin >> op;
        if (op == 1) {
            int l, r; i64 ans = 0; cin >> l >> r;
            for (int i = 0; i < 20; i++) {
                ans += segt[i].ask(l, r) * (1LL << i);
            }
            cout << ans << "\n";
        } else {
            int l, r, val; cin >> l >> r >> val;
            for (int i = 0; i < 20; i++) {
                if (val >> i & 1) { segt[i].reverse(l, r); }
            }
        }
    }
}
```

树套树

线段树套平衡树

```
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
using pii = std::pair<int, int>;
const int inf = 2147483647;

tree<pii, null_type, std::less<pii>, rb_tree_tag,
tree_order_statistics_node_update> ver;

template<class Info>
struct SegmentTree {
    int n;
    std::vector<Info> info;

    SegmentTree() : n(0) {}

    void init(int n_) {
        n = n_;
        info.assign(4 << std::__lg(n), Info());
    }

public:
    // --- 初始化与构建 ---
    void build(const std::vector<int>& a) {
        _build(1, 0, n, a);
```

```
    }

private:
    void _build(int p, int l, int r, const std::vector<int>& a) {
        // 从叶子节点开始，向上启发式合并来构建整棵树
        if (r - l == 1) {
            if (l < a.size()) info[p].ver.insert({ a[l], l });
            return;
        }
        int m = (l + r) / 2;
        _build(2 * p, l, m, a);
        _build(2 * p + 1, m, r, a);

        // 启发式合并 (小树合并到大树)
        if (info[2 * p].ver.size() > info[2 * p + 1].ver.size()) {
            info[p].ver = info[2 * p].ver;
            for (const auto& item : info[2 * p + 1].ver)
info[p].ver.insert(item);
        }
        else {
            info[p].ver = info[2 * p + 1].ver;
            for (const auto& item : info[2 * p].ver)
info[p].ver.insert(item);
        }
    }

public:
    // 单点修改
    void update(int pos, int old_val, int new_val) {
        _update(1, 0, n, pos, old_val, new_val);
    }

    // 查询排名 (返回比 k 小的数的个数)
    int query_rank_count(int l, int r, int k) {
        return _query_rank_count(1, 0, n, l, r, k);
    }

    // 查询前驱
    int query_pred(int l, int r, int k) {
        return _query_pred(1, 0, n, l, r, k);
    }

    // 查询后继
    int query_succ(int l, int r, int k) {
        return _query_succ(1, 0, n, l, r, k);
    }

private:
    void _update(int p, int l, int r, int pos, int old_val, int new_val) {
        info[p].ver.erase({ old_val, pos });
        info[p].ver.insert({ new_val, pos });
        if (r - l == 1) return;
        int m = (l + r) / 2;
        if (pos < m) _update(2 * p, l, m, pos, old_val, new_val);
        else _update(2 * p + 1, m, r, pos, old_val, new_val);
    }
```

```
    int _query_rank_count(int p, int l, int r, int x, int y, int k) {
        if (l >= y || r <= x) return 0;
        if (l >= x && r <= y) {
            return info[p].ver.order_of_key({ k, -1 });
        }
        int m = (l + r) / 2;
        return _query_rank_count(2 * p, l, m, x, y, k) +
_query_rank_count(2 * p + 1, m, r, x, y, k);
    }

    int _query_pred(int p, int l, int r, int x, int y, int k) {
        if (l >= y || r <= x) return -inf;
        if (l >= x && r <= y) {
            auto it = info[p].ver.lower_bound({ k, -1 });
            if (it == info[p].ver.begin()) return -inf;
            return (--it)->first;
        }
        int m = (l + r) / 2;
        return std::max(_query_pred(2 * p, l, m, x, y, k), _query_pred(2 *
p + 1, m, r, x, y, k));
    }

    int _query_succ(int p, int l, int r, int x, int y, int k) {
        if (l >= y || r <= x) return inf;
        if (l >= x && r <= y) {
            auto it = info[p].ver.upper_bound({ k, inf });
            if (it == info[p].ver.end()) return inf;
            return it->first;
        }
        int m = (l + r) / 2;
        return std::min(_query_succ(2 * p, l, m, x, y, k), _query_succ(2 *
p + 1, m, r, x, y, k));
    }
};

struct Info {
    tree<pii, null_type, std::less<pii>, rb_tree_tag,
tree_order_statistics_node_update> ver;
};
```

坐标压缩与离散化

简单版本

```
sort(alls.begin(), alls.end());
alls.erase(unique(alls.begin(), alls.end()), alls.end());
auto get = [&](int x) {
    return lower_bound(alls.begin(), alls.end(), x) - alls.begin();
};
```

离散化

```cpp
template <typename T> struct Compress_ {
    int n, shift = 0; // shift 用于标记下标偏移量
    vector<T> alls;

    Compress_() {}
    Compress_(auto in) : alls(in) {
        init();
    }
    void add(T x) {
        alls.emplace_back(x);
    }
    template <typename... Args> void add(T x, Args... args) {
        add(x), add(args...);
    }
    void init() {
        alls.emplace_back(numeric_limits<T>::max());
        sort(alls.begin(), alls.end());
        alls.erase(unique(alls.begin(), alls.end()), alls.end());
        this->n = alls.size();
    }
    int size() {
        return n;
    }
    int operator[](T x) { // 返回 x 元素的新下标
        return upper_bound(alls.begin(), alls.end(), x) - alls.begin() +
shift;
    }
    T Get(int x) { // 根据新下标返回原来元素
        assert(x - shift < n);
        return x - shift < n ? alls[x - shift] : -1;
    }
    bool count(T x) { // 查找元素 x 是否存在
        return binary_search(alls.begin(), alls.end(), x);
    }
    friend auto &operator<< (ostream &o, const auto &j) {
        cout << "{";
        for (auto it : j.alls) {
            o << it << " ";
        }
        return o << "}";
    }
};
using Compress = Compress_<int>;
```

轻重链剖分/树链剖分

将线段树处理的部分分离，方便修改。支持链上查询/修改、子树查询/修改，建树时间复杂度 $\mathcal{O}(N \log N)$，单次查询时间复杂度 $\mathcal{O}(\log^2 N)$。

```cpp
struct Segt {
    struct node {
        int l, r, w, lazy;
```

```
    };
    vector<int> w;
    vector<node> t;

    Segt() {}
    #define GL (k << 1)
    #define GR (k << 1 | 1)

    void init(vector<int> in) {
        int n = in.size() - 1;
        w.resize(n + 1);
        for (int i = 1; i <= n; i++) {
            w[i] = in[i];
        }
        t.resize(n * 4 + 1);
        auto build = [&](auto self, int l, int r, int k = 1) {
            if (l == r) {
                t[k] = {l, r, w[l], 0}; // 如果有赋值为 0 的操作，则懒标记必须要 -1
                return;
            }
            t[k] = {l, r};
            int mid = (l + r) / 2;
            self(self, l, mid, GL);
            self(self, mid + 1, r, GR);
            pushup(k);
        };
        build(build, 1, n);
    }
    void pushdown(node &p, int lazy) { /* 【在此更新下递函数】 */
        p.w += (p.r - p.l + 1) * lazy;
        p.lazy += lazy;
    }
    void pushdown(int k) { // 不需要动
        if (t[k].lazy == 0) return;
        pushdown(t[GL], t[k].lazy);
        pushdown(t[GR], t[k].lazy);
        t[k].lazy = 0;
    }
    void pushup(int k) { // 不需要动
        auto pushup = [&](node &p, node &l, node &r) { /* 【在此更新上传函数】 */
            p.w = l.w + r.w;
        };
        pushup(t[k], t[GL], t[GR]);
    }
    void modify(int l, int r, int val, int k = 1) {
        if (l <= t[k].l && t[k].r <= r) {
            pushdown(t[k], val);
            return;
        }
        pushdown(k);
        int mid = (t[k].l + t[k].r) / 2;
        if (l <= mid) modify(l, r, val, GL);
        if (mid < r) modify(l, r, val, GR);
        pushup(k);
```

```
    }
    int ask(int l, int r, int k = 1) {
        if (l <= t[k].l && t[k].r <= r) {
            return t[k].w;
        }
        pushdown(k);
        int mid = (t[k].l + t[k].r) / 2;
        int ans = 0;
        if (l <= mid) ans += ask(l, r, GL);
        if (mid < r) ans += ask(l, r, GR);
        return ans;
    }
};

struct HLD {
    int n, idx;
    vector<vector<int>> ver;
    vector<int> siz, dep;
    vector<int> top, son, parent;
    vector<int> in, id, val;
    Segt segt;

    HLD(int n) {
        this->n = n;
        ver.resize(n + 1);
        siz.resize(n + 1);
        dep.resize(n + 1);

        top.resize(n + 1);
        son.resize(n + 1);
        parent.resize(n + 1);

        idx = 0;
        in.resize(n + 1);
        id.resize(n + 1);
        val.resize(n + 1);
    }
    void add(int x, int y) { // 建立双向边
        ver[x].push_back(y);
        ver[y].push_back(x);
    }
    void dfs1(int x) {
        siz[x] = 1;
        dep[x] = dep[parent[x]] + 1;
        for (auto y : ver[x]) {
            if (y == parent[x]) continue;
            parent[y] = x;
            dfs1(y);
            siz[x] += siz[y];
            if (siz[y] > siz[son[x]]) {
                son[x] = y;
            }
        }
    }
    void dfs2(int x, int up) {
        id[x] = ++idx;
```

```
        val[idx] = in[x]; // 建立编号
        top[x] = up;
        if (son[x]) dfs2(son[x], up);
        for (auto y : ver[x]) {
            if (y == parent[x] || y == son[x]) continue;
            dfs2(y, y);
        }
    }
    void modify(int l, int r, int val) { // 链上修改
        while (top[l] != top[r]) {
            if (dep[top[l]] < dep[top[r]]) {
                swap(l, r);
            }
            segt.modify(id[top[l]], id[l], val);
            l = parent[top[l]];
        }
        if (dep[l] > dep[r]) {
            swap(l, r);
        }
        segt.modify(id[l], id[r], val);
    }
    void modify(int root, int val) { // 子树修改
        segt.modify(id[root], id[root] + siz[root] - 1, val);
    }
    int ask(int l, int r) { // 链上查询
        int ans = 0;
        while (top[l] != top[r]) {
            if (dep[top[l]] < dep[top[r]]) {
                swap(l, r);
            }
            ans += segt.ask(id[top[l]], id[l]);
            l = parent[top[l]];
        }
        if (dep[l] > dep[r]) {
            swap(l, r);
        }
        return ans + segt.ask(id[l], id[r]);
    }
    int ask(int root) { // 子树查询
        return segt.ask(id[root], id[root] + siz[root] - 1);
    }
    void work(auto in, int root = 1) { // 在此初始化
        assert(in.size() == n + 1);
        this->in = in;
        dfs1(root);
        dfs2(root, root);
        segt.init(val); // 建立线段树
    }
    void work(int root = 1) { // 在此初始化
        dfs1(root);
        dfs2(root, root);
        segt.init(val); // 建立线段树
    }
};
```

小波矩阵树：高效静态区间第 K 大查询

手写 `bitset` 压位，以 $\mathcal{O}(N \log N)$ 的时间复杂度和 $\mathcal{O}(N + \frac{N \log N}{64})$ 的空间建树后，实现单次 $\mathcal{O}(\log N)$ 复杂度的区间第 k 大值询问。建议使用 `0-idx` 计数法，但是经测试 `1-idx` 也有效，但需要更多的检验。

```cpp
#define __count(x) __builtin_popcountll(x)
struct Wavelet {
    vector<int> val, sum;
    vector<u64> bit;
    int t, n;

    int getSum(int i) {
        return sum[i >> 6] + __count(bit[i >> 6] & ((1ULL << (i & 63)) - 1));
    }

    Wavelet(vector<int> v) : val(v), n(v.size()) {
        sort(val.begin(), val.end());
        val.erase(unique(val.begin(), val.end()), val.end());

        int n_ = val.size();
        t = __lg(2 * n_ - 1);
        bit.resize((t * n + 64) >> 6);
        sum.resize(bit.size());
        vector<int> cnt(n_ + 1);

        for (int &x : v) {
            x = lower_bound(val.begin(), val.end(), x) - val.begin();
            cnt[x + 1]++;
        }
        for (int i = 1; i < n_; ++i) {
            cnt[i] += cnt[i - 1];
        }
        for (int j = 0; j < t; ++j) {
            for (int i : v) {
                int tmp = i >> (t - 1 - j);
                int pos = (tmp >> 1) << (t - j);
                auto setBit = [&](int i, u64 v) {
                    bit[i >> 6] |= (v << (i & 63));
                };
                setBit(j * n + cnt[pos], tmp & 1);
                cnt[pos]++;
            }
            for (int i : v) {
                cnt[(i >> (t - j)) << (t - j)]--;
            }
        }
        for (int i = 1; i < sum.size(); ++i) {
            sum[i] = sum[i - 1] + __count(bit[i - 1]);
        }
    }

    int small(int l, int r, int k) {
        r++;
```

```
        for (int j = 0, x = 0, y = n, res = 0;; ++j) {
            if (j == t) return val[res];
            int A = getSum(n * j + x), B = getSum(n * j + l);
            int C = getSum(n * j + r), D = getSum(n * j + y);
            int ab_zeros = r - l - C + B;
            if (ab_zeros > k) {
                res = res << 1;
                y -= D - A;
                l -= B - A;
                r -= C - A;
            } else {
                res = (res << 1) | 1;
                k -= ab_zeros;
                x += y - x - D + A;
                l += y - l - D + B;
                r += y - r - D + C;
            }
        }
    }
    int large(int l, int r, int k) {
        return small(l, r, r - l - k);
    }
};
```

主席树（可持久化线段树）

以 $\mathcal{O}(N \log N)$ 的时间复杂度建树、查询、修改。

```
struct PreesidentTree {
    static constexpr int N = 2e5 + 10;
    int cntNodes, root[N];

    struct node {
        int l, r;
        int cnt;
    }tr[4 * N + 17 * N];

    //u 是新节点，v 是旧节点
    void modify(int& u, int v, int l, int r, int x) {
        u = ++cntNodes;
        tr[u] = tr[v];
        tr[u].cnt++;
        if (l == r) return;
        int mid = (l + r) / 2;
        if (x <= mid) modify(tr[u].l, tr[v].l, l, mid, x);
        else modify(tr[u].r, tr[v].r, mid + 1, r, x);
    }

    //u 是新节点，v 是旧节点
    int kth(int u, int v, int l, int r, int k) {
        if (l == r) return l;
        int res = tr[tr[u].l].cnt - tr[tr[v].l].cnt;
        int mid = (l + r) / 2;
        if (k <= res) return kth(tr[u].l, tr[v].l, l, mid, k);
        else return kth(tr[u].r, tr[v].r, mid + 1, r, k - res);
```

```
    }
};
```

普通莫队

以 $\mathcal{O}(N\sqrt{N})$ 的复杂度完成 Q 次询问的离线查询，其中每个分块的大小取 $\sqrt{N}=\sqrt{10^5}=317$，也可以使用 `n / min<int>(n, sqrt(q))`、`ceil((double)n / (int)sqrt(n))` 或者 `sqrt(n)` 划分。

```
signed main() {
    int n;
    cin >> n;
    vector<int> w(n + 1);
    for (int i = 1; i <= n; i++) {
        cin >> w[i];
    }

    int q;
    cin >> q;
    vector<array<int, 3>> query(q + 1);
    for (int i = 1; i <= q; i++) {
        int l, r;
        cin >> l >> r;
        query[i] = {l, r, i};
    }

    int Knum = n / min<int>(n, sqrt(q)); // 计算块长
    vector<int> K(n + 1);
    for (int i = 1; i <= n; i++) { // 固定块长
        K[i] = (i - 1) / Knum + 1;
    }
    sort(query.begin() + 1, query.end(), [&](auto x, auto y) {
        if (K[x[0]] != K[y[0]]) return x[0] < y[0];
        if (K[x[0]] & 1) return x[1] < y[1];
        return x[1] > y[1];
    });

    int l = 1, r = 0, val = 0;
    vector<int> ans(q + 1);
    for (int i = 1; i <= q; i++) {
        auto [ql, qr, id] = query[i];
        auto add = [&](int x) -> void {};
        auto del = [&](int x) -> void {};
        while (l > ql) add(w[--l]);
        while (r < qr) add(w[++r]);
        while (l < ql) del(w[l++]);
        while (r > qr) del(w[r--]);
        ans[id] = val;
    }
    for (int i = 1; i <= q; i++) {
        cout << ans[i] << endl;
    }
}
```

需要注意的是，在普通莫队中，`K` 数组的作用是根据左边界的值进行排序，当询问次数很少时（$q \ll n$），可以直接合并到 `query` 数组中。

```
vector<array<int, 4>> query(q);
for (int i = 1; i <= q; i++) {
    int l, r;
    cin >> l >> r;
    query[i] = {l, r, i, (l - 1) / Knum + 1}; // 合并
}
sort(query.begin() + 1, query.end(), [&](auto x, auto y) {
    if (x[3] != y[3]) return x[3] < y[3];
    if (x[3] & 1) return x[1] < y[1];
    return x[1] > y[1];
});
```

带修改的莫队（带时间维度的莫队）

以 $\mathcal{O}(N^{\frac{5}{3}})$ 的复杂度完成 Q 次询问的离线查询，其中每个分块的大小取 $N^{\frac{2}{3}} = \sqrt[3]{100000^2} = 2154$（直接取会略快），也可以使用 `pow(n, 0.6666)` 划分。

```
signed main() {
    int n, q;
    cin >> n >> q;
    vector<int> w(n + 1);
    for (int i = 1; i <= n; i++) {
        cin >> w[i];
    }

    vector<array<int, 4>> query = {{}}; // {左区间，右区间，累计修改次数，下标}
    vector<array<int, 2>> modify = {{}}; // {修改的值，修改的元素下标}
    for (int i = 1; i <= q; i++) {
        char op;
        cin >> op;
        if (op == 'Q') {
            int l, r;
            cin >> l >> r;
            query.push_back({l, r, (int)modify.size() - 1,
(int)query.size()});
        } else {
            int idx, w;
            cin >> idx >> w;
            modify.push_back({w, idx});
        }
    }

    int Knum = 2154; // 计算块长
    vector<int> K(n + 1);
    for (int i = 1; i <= n; i++) { // 固定块长
        K[i] = (i - 1) / Knum + 1;
    }
    sort(query.begin() + 1, query.end(), [&](auto x, auto y) {
        if (K[x[0]] != K[y[0]]) return x[0] < y[0];
        if (K[x[1]] != K[y[1]]) return x[1] < y[1];
        return x[3] < y[3];
```

```
    });

    int l = 1, r = 0, val = 0;
    int t = 0; // 累计修改次数
    vector<int> ans(query.size());
    for (int i = 1; i < query.size(); i++) {
        auto [ql, qr, qt, id] = query[i];
        auto add = [&](int x) -> void {};
        auto del = [&](int x) -> void {};
        auto time = [&](int x, int l, int r) -> void {};
        while (l > ql) add(w[--l]);
        while (r < qr) add(w[++r]);
        while (l < ql) del(w[l++]);
        while (r > qr) del(w[r--]);
        while (t < qt) time(++t, ql, qr);
        while (t > qt) time(t--, ql, qr);
        ans[id] = val;
    }
    for (int i = 1; i < ans.size(); i++) {
        cout << ans[i] << endl;
    }
}
```

对顶堆

```
namespace Set {
    const int kInf = 1e9 + 2077;
    std::multiset<int> less, greater;
    void init() {
        less.clear(), greater.clear();
        less.insert(-kInf), greater.insert(kInf);
    }
    void adjust() {
        while (less.size() > greater.size() + 1) {
            std::multiset<int>::iterator it = (--less.end());
            greater.insert(*it);
            less.erase(it);
        }
        while (greater.size() > less.size()) {
            std::multiset<int>::iterator it = greater.begin();
            less.insert(*it);
            greater.erase(it);
        }
    }
    void add(int val_) {
        if (val_ <= *greater.begin()) less.insert(val_);
        else greater.insert(val_);
        adjust();
    }
    void del(int val_) {
        std::multiset<int>::iterator it = less.lower_bound(val_);
        if (it != less.end()) {
            less.erase(it);
        }
        else {
```

```
            it = greater.lower_bound(val_);
            greater.erase(it);
        }
        adjust();
    }
    int get_middle() {
        return *less.rbegin();
    }
}
```

主席树（可持久化线段树）

以 $\mathcal{O}(N \log N)$ 的时间复杂度建树、查询、修改。

```
struct PresidentTree {
    static constexpr int N = 2e5 + 10;
    int cntNodes, root[N];
    struct node {
        int l, r;
        int cnt;
    } tr[4 * N + 17 * N];
    void modify(int &u, int v, int l, int r, int x) {
        u = ++cntNodes;
        tr[u] = tr[v];
        tr[u].cnt++;
        if (l == r) return;
        int mid = (l + r) / 2;
        if (x <= mid)
            modify(tr[u].l, tr[v].l, l, mid, x);
        else
            modify(tr[u].r, tr[v].r, mid + 1, r, x);
    }
    int kth(int u, int v, int l, int r, int k) {
        if (l == r) return l;
        int res = tr[tr[v].l].cnt - tr[tr[u].l].cnt;
        int mid = (l + r) / 2;
        if (k <= res)
            return kth(tr[u].l, tr[v].l, l, mid, k);
        else
            return kth(tr[u].r, tr[v].r, mid + 1, r, k - res);
    }
};
```

KD Tree

在第 k 维上的单次查询复杂度最坏为 $\mathcal{O}(n^{1-k^{-1}})$。

```
struct KDT {
    constexpr static int N = 1e5 + 10, K = 2;
    double alpha = 0.725;
    struct node {
        int info[K];
        int mn[K], mx[K];
    } tr[N];
```

```
    int ls[N], rs[N], siz[N], id[N], d[N];
    int idx, rt, cur;
    int ans;
    KDT() {
        rt = 0;
        cur = 0;
        memset(ls, 0, sizeof ls);
        memset(rs, 0, sizeof rs);
        memset(d, 0, sizeof d);
    }
    void apply(int p, int son) {
        if (son) {
            for (int i = 0; i < K; i++) {
                tr[p].mn[i] = min(tr[p].mn[i], tr[son].mn[i]);
                tr[p].mx[i] = max(tr[p].mx[i], tr[son].mx[i]);
            }
            siz[p] += siz[son];
        }
    }
    void maintain(int p) {
        for (int i = 0; i < K; i++) {
            tr[p].mn[i] = tr[p].info[i];
            tr[p].mx[i] = tr[p].info[i];
        }
        siz[p] = 1;
        apply(p, ls[p]);
        apply(p, rs[p]);
    }
    int build(int l, int r) {
        if (l > r) return 0;
        vector<double> avg(K);
        for (int i = 0; i < K; i++) {
            for (int j = l; j <= r; j++) {
                avg[i] += tr[id[j]].info[i];
            }
            avg[i] /= (r - l + 1);
        }
        vector<double> var(K);
        for (int i = 0; i < K; i++) {
            for (int j = l; j <= r; j++) {
                var[i] += (tr[id[j]].info[i] - avg[i]) *
(tr[id[j]].info[i] - avg[i]);
            }
        }
        int mid = (l + r) / 2;
        int x = max_element(var.begin(), var.end()) - var.begin();
        nth_element(id + l, id + mid, id + r + 1, [&](int a, int b) {
            return tr[a].info[x] < tr[b].info[x];
        });
        d[id[mid]] = x;
        ls[id[mid]] = build(l, mid - 1);
        rs[id[mid]] = build(mid + 1, r);
        maintain(id[mid]);
        return id[mid];
    }
    void print(int p) {
```

```
        if (!p) return;
        print(ls[p]);
        id[++idx] = p;
        print(rs[p]);
    }
    void rebuild(int &p) {
        idx = 0;
        print(p);
        p = build(1, idx);
    }
    bool bad(int p) {
        return alpha * siz[p] <= max(siz[ls[p]], siz[rs[p]]);
    }
    void insert(int &p, int cur) {
        if (!p) {
            p = cur;
            maintain(p);
            return;
        }
        if (tr[p].info[d[p]] > tr[cur].info[d[p]]) insert(ls[p], cur);
        else insert(rs[p], cur);
        maintain(p);
        if (bad(p)) rebuild(p);
    }
    void insert(vector<int> &a) {
        cur++;
        for (int i = 0; i < K; i++) {
            tr[cur].info[i] = a[i];
        }
        insert(rt, cur);
    }
    bool out(int p, vector<int> &a) {
        for (int i = 0; i < K; i++) {
            if (a[i] < tr[p].mn[i]) {
                return true;
            }
        }
        return false;
    }
    bool in(int p, vector<int> &a) {
        for (int i = 0; i < K; i++) {
            if (a[i] < tr[p].info[i]) {
                return false;
            }
        }
        return true;
    }
    bool all(int p, vector<int> &a) {
        for (int i = 0; i < K; i++) {
            if (a[i] < tr[p].mx[i]) {
                return false;
            }
        }
        return true;
    }
    void query(int p, vector<int> &a) {
```

```
        if (!p) return;
        if (out(p, a)) return;
        if (all(p, a)) {
            ans += siz[p];
            return;
        }
        if (in(p, a)) ans++;
        query(ls[p], a);
        query(rs[p], a);
    }
    int query(vector<int> &a) {
        ans = 0;
        query(rt, a);
        return ans;
    }
};
```

/END/