Namomo Camp 2024.7

「*Namomo Camp 2024.7*」学习笔记

CCA

Shenzhen MSU-BIT University

最初写作于: **2024 年 07 月 27 日**

最后更新于: **2024 年 08 月 01 日**

目录

I. 数据结构选讲

1.1 静态 RMQ

e.g. 一个静态 RMQ 问题 例 1.1.1

给定长度为 n 的序列 $a_1, a_2, \cdots, a_n$. 进行 q 次询问, 每次询问给定 l, r, 需要求出 $\min\{a_l, a_{l+1}, \cdots, a_r\}$.

1.1.1 $\mathcal{O}(n \log n) - \mathcal{O}(1)$ 稀疏表

该问题一个广为人知的解法是使用稀疏表.

考虑设 $f_{i,k} = \min\{a_i, a_{i+1}, \cdots, a_{i+2^k-1}\}, k = \lfloor \log_2(r-l+1) \rfloor$, 则有

$$\min\{a_l, a_{l+1}, \cdots, a_r\} = \min\{f_{l,k}, f_{r-2^k+1,k}\} \tag{1.1}$$

f 的预处理可以做到 $\mathcal{O}(n \log n)$ 的复杂度, 单次询问可以通过 $\mathcal{O}(1)$ 调用 f 的值回答. 能做到这一点本质上是利用了 min 操作的可重性, 也即若存在集合 S, T, 其中 T 是由 S 中的每个元素复制不小于 1 次得到, 那么 $\min S = \min T$.

```cpp
template <typename type>
class SparseTable {
private:
    int n, lg;
    std::function<type(type, type)> merge;
    std::vector<std::vector<type>> f;

public:
    SparseTable(std::vector<type> &a,
      std::function<type(type, type)> merge): merge(merge) {
        n = a.size(), lg = std::__lg(n);
        f = std::vector<std::vector<type>>
          (n, std::vector<type>(lg + 1));
        for (size_t i = 0; i < n; i++) f[i][0] = a[i];
        for (size_t j = 1; j <= lg; j++)
            for (size_t i = 0; i < n; i++)
                if (i + (1 << j - 1) < n)
                    f[i][j] = merge(f[i][j - 1],
                      f[i + (1 << j - 1)][j - 1]);
                else f[i][j] = f[i][j - 1];
    }

    type qry(int l, int r) {
        int t = std::__lg(r - l + 1);
        return merge(f[l][t], f[r - (1 << t) + 1][t]);
```

```
26      }
27  };
```

1.1.2 $\mathcal{O}(n+q\log n)$ 离线单调栈上二分

考虑将询问离线下来, 按 r 排序, 从左到右遍历每个元素并维护一个上升的单调栈(栈中存储元素的下标). 假设当前遍历到 a_i, 要处理 $r=i$ 的询问, 那么答案就是单调栈中最小的不小于 l 的下标对应的元素.

关于使用单调栈的正确性: 假设单调栈中存在 j, 现在要加入其中的元素是 i, 如果满足 $a_j \geqslant a_i$, 那 j 就没有了继续存在于单调栈中的必要, 因为对于接下来的所有询问, 如果 $j\in[l,r]$, 那么 $i\in[l,r]$, 而 $a_i\leqslant a_j$, 所以 a_j 不可能成为答案. 于是每次加入的 a_i 都大于栈中的所有元素, 也就维护了其单调上升的性质.

在整个流程中, 每个元素进出单调栈 $\mathcal{O}(1)$ 次, 一共要进行 q 次二分, 于是总时间复杂度为 $\mathcal{O}(n+q\log n)$.

1.1.3 $\mathcal{O}\left(n^{\frac{2}{3}}q^{\frac{2}{3}}\right)$ 分块

考虑将序列分为 B 块, 求出

- 每块中的最小值, 这部分时间复杂度为 $\mathcal{O}(n)$.
- 每块中最小值形成的序列中每个区间的最小值, 这部分时间复杂度为 $\mathcal{O}\left(\left(\frac{n}{B}\right)^2\right)$.
- 每块的前缀, 后缀最小值, 这部分时间复杂度为 $\mathcal{O}(n)$.

对于跨至少两个块的询问, 可以 $\mathcal{O}(1)$ 求出答案, 对于块内的询问, 可以 $\mathcal{O}(B)$ 求出答案. 于是这部分时间复杂度为 $\mathcal{O}(qB)$.

总时间复杂度为 $\mathcal{O}\left(\frac{n^2}{B^2}+qB\right)$, 取 $B=n^{\frac{2}{3}}q^{-\frac{1}{3}}$, 时间复杂度取到最小值 $\mathcal{O}\left(n^{\frac{2}{3}}q^{\frac{2}{3}}\right)$.

1.1.4 $\mathcal{O}(n)-\mathcal{O}(1)$ 稀疏表 + 分块 + 单调栈

考虑分块, 块大小设为 ω.

> **整数位运算的复杂度** 引用 **1.1.1**
>
> 在 Word-RAM 模型中 $\omega=\Omega(n)$, ω 位整数位运算的时间复杂度为 $\mathcal{O}(1)$. 现代计算机中常用的 ω 取值为 64 或 32.

进行以下预处理:

- 对于块结果构成的序列, 构建稀疏表, 这部分复杂度为 $\mathcal{O}\left(\frac{n}{\omega}\log\frac{n}{\omega}\right)\leqslant\mathcal{O}(n)$.
- 对于每个块计算前缀, 后缀最小值, 这部分时间复杂度为 $\mathcal{O}(n)$.

- 对每个元素计算以其结尾的块内单调栈的二进制表示，这部分时间复杂度为 $\mathcal{O}(n)$.

对于跨至少两个块的询问，可以 $\mathcal{O}(1)$ 求出答案. 对于块内的询问，设以 r 结尾的块内单调栈的二进制表示为 S_r，则答案为 S_r 右移「l 在块内的下标」位后第一个 1 对应的值，可以使用一些二进制操作求出，时间复杂度为 $\mathcal{O}(1)$.

以上我们均以 min 操作为例，但不难发现上述结论和算法对 max 操作也都是成立的. 于是静态 RMQ 问题可以做到 $\mathcal{O}(n)-\mathcal{O}(1)$，是理论最优复杂度.

```cpp
template <typename type>
class RMQ {
private:
    static constexpr unsigned w = 64;
    using u64 = unsigned long long;
    int n, m;
    std::function<type(type, type)> merge;
    std::vector<type> a, pre, suf;
    std::vector<u64> S;
    SparseTable<type> *ST;

public:
    RMQ(std::vector<type> a, std::function<type(type, type)> merge):
      merge(merge), a(a) {
        while (a.size() % w != 0) a.push_back(type());
        n = a.size(), m = (n + 1) / w;

        std::vector<type> arr(m);
        for (size_t i = 0; i < m; i++) {
            arr[i] = a[i * w];
            for (size_t j = 1; j < w; j++)
                arr[i] = merge(arr[i], a[i * w + j]);
        }
        ST = new SparseTable<type>(arr, merge);

        pre.resize(n), suf.resize(n);
        for (size_t i = 0; i < m; i++) {
            pre[i * w] = a[i * w];
            for (size_t j = 1; j < w; j++)
                pre[i * w + j] =
                  merge(pre[i * w + j - 1], a[i * w + j]);
            suf[(i + 1) * w - 1] = a[(i + 1) * w - 1];
            for (int j = w - 2; j >= 0; j--)
                suf[i * w + j] =
                  merge(suf[i * w + j + 1], a[i * w + j]);
        }

        S.resize(n);
        for (size_t i = 0; i < m; i++) {
            std::stack<size_t> sta;
            u64 st = 0;
```

```
            for (size_t j = 0; j < w; j++) {
                while (not sta.empty() and merge(a[i * w + sta.top()],
                  a[i * w + j]) == a[i * w + j])
                    st ^= (1ull << sta.top()), sta.pop();
                st ^= (1ull << j), sta.push(j);
                S[i * w + j] = st;
            }
        }
    }

    type qry(int l, int r) {
        int idl = l / w, idr = r / w;
        if (idl != idr) {
            type ans = suf[l];
            if (idr - idl >= 2)
                ans = merge(ans, (*ST).qry(idl + 1, idr - 1));
            ans = merge(ans, pre[r]);
            return ans;
        } else {
            u64 st = S[r] >> (l - idl * w);
            int pos = __builtin_ffsll(st) - 1;
            return a[l + pos];
        }
    }
};
```

1.2 LCA

Def LCA 问题 定义 1.2.1

给定一棵 n 个点的有根树，有 q 次询问，每次询问给定两个结点 u, v. 需要求出 u, v 的最近公共祖先，也即最深的结点，满足同时是 u 和 v 的祖先结点.

1.2.1 $\mathcal{O}(n \log n) - \mathcal{O}(\log n)$ 倍增

最符合直觉的做法.

首先考虑一个暴力，假设 $\mathrm{dep}_u > \mathrm{dep}_v$，将 u 跳到与 v 相同的高度. 显然 u 在此过程中是不可能与 v 重合的，如果 u 在与 v 深度相同的时候与 v 重合了，那么它们的 LCA 就是 v. 否则两个点一起一步步往上跳，容易发现它们第一次重合的位置就是两者的 LCA.

考虑优化上述过程，本质都是让点往上跳，直到第一次满足某个条件. 设 $f_{u,k}$ 表示 u 向上跳 2^k 步到达的结点，特别的，定义根节点和 0 的父结点都是 0. 如果枚举 $k : \log_2 n \to 0$，$f_{u,k}$ 满足条件就不跳，否则跳$(u \to f_{u,k})$. 容易证明枚举结束后 u 的父结点即为所求.

优化后时间复杂度为 $\mathcal{O}(n \log n) - \mathcal{O}(\log n)$.

1.2.2 $\mathcal{O}(n) - \mathcal{O}(\log n)$ 树链剖分

最常用的做法，因为常数极小.

Def 关于「重」 定义 1.2.2

我们称 v 是 u 的「重儿子」，当且仅当 v 是 u 的所有儿子中子树大小最大的. 显然每个非叶子结点都有恰好一个重儿子. 我们称父结点与重儿子之间的连边为「重边」，一条仅包含重边的链为「重链」.

注意到从任意节点到其先祖的链上最多经过 $\log_2 n$ 条非重边. (因为从上到下每经过一条非重边子树大小一定不大于原本的 $\frac{1}{2}$). 同样考虑每次将结点往上跳，不妨假设 $\mathrm{dep}_u > \mathrm{dep}_v$，将 u 跳至当前重链顶端的父结点. 直到 u 和 v 在同一条重链上，较深的点即为两者的 LCA. 该做法的正确性易证，不再赘述.

预处理只需要将树遍历两遍，故复杂度优于倍增，为 $\mathcal{O}(n) - \mathcal{O}(\log n)$.

```
1  class HeavyLight_Decomposition_LCA {
2  private:
3      int n, rt;
4      std::vector<std::vector<int>> E;
5      std::vector<int> fa, dep, siz, top, son;
6
```

```cpp
    void dfs1(int u, int last) {
        fa[u] = last, dep[u] = dep[last] + 1, siz[u] = 1;
        for (int v : E[u]) if (v ≠ last) {
            dfs1(v, u), siz[u] += siz[v];
            if (siz[v] > siz[son[u]]) son[u] = v;
        }
    }

    void dfs2(int u, int t) {
        top[u] = t;
        if (son[u]) dfs2(son[u], t);
        for (int v : E[u])
            if (v ≠ fa[u] and v ≠ son[u]) dfs2(v, v);
    }

public:
    HeavyLight_Decomposition_LCA
      (std::vector<std::vector<int>> &E, int rt): E(E), rt(rt) {
        n = E.size();
        fa.resize(n), dep.resize(n), siz.resize(n),
        top.resize(n), son.resize(n);
        dfs1(rt, 0), dfs2(rt, rt);
    }

    int LCA(int u, int v) {
        while (top[u] ≠ top[v]) {
            if (dep[top[u]] < dep[top[v]]) std::swap(u, v);
            u = fa[top[u]];
        }
        return dep[u] < dep[v] ? u : v;
    }
};
```

1.2.3 $\mathcal{O}((n+q)\alpha(n))$ 离线并查集

考虑将整棵树遍历一遍，假设遍历完了以 r 为根的子树，则将 r 与其父亲所在集合合并. 设当前遍历到的结点为 u，在它上面挂了一个询问 (u,v)，若 v 已经被遍历过了，则答案为 v 所在集合中深度最小的结点.

⚠ u 与 v 满足祖孙关系 注意 1.2.1

若 u,v 其中一个是另一个的祖先，那么需要特别判断. 具体来说，设 dfni_u 和 dfno_u 分别表示结点 u dfs 时进入和退出的时间戳，那么 u 为 v 的祖先当且仅当 $\text{dfni}_u < \text{dfni}_v < \text{dfno}_u$，此时 $\text{LCA}(u,v) = u$. 这些询问只需要 dfs 一遍即可计算出答案.

上述做法的正确性易证，不再赘述. 设并查集单次合并和查询的均摊复杂度为 $\alpha(n)$，则该算法的时间复杂度为 $\mathcal{O}((n+q)\alpha(n))$.

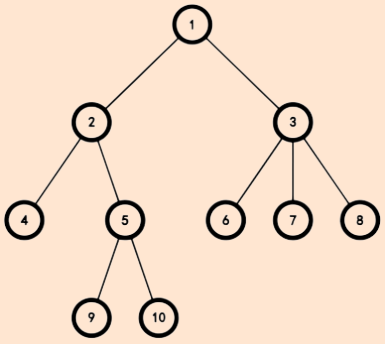
```cpp
class TarjanLCA {
private:
    class DSU {
    private:
        int n;
        std::vector<int> fa, pnum, top;

        int find(int x) {
            if (x ≠ fa[x]) fa[x] = find(fa[x]);
            return fa[x];
        }

    public:
        DSU(int n): n(n) {
            for (int i = 0; i ⩽ n; i++)
                fa.push_back(i), top.push_back(i);
            pnum = std::vector<int>(n + 1, 1);
        }

        bool merge(int x, int y) {
            int Rx = find(x), Ry = find(y);
            if (x ≠ y) {
                if (pnum[Rx] < pnum[Ry])
                    fa[Rx] = Ry, pnum[Ry] += pnum[Rx];
                else fa[Ry] = Rx, pnum[Rx] += pnum[Ry];
                top[Rx] = top[Ry];
            }
            return Rx ≠ Ry;
        }

        int shallow(int x) {
            return top[find(x)];
        }
    };

    int n, m, rt, cnt;
    std::vector<std::vector<int>> E, q;
    std::vector<bool> vis;
    std::vector<int> dfn;
    std::vector<std::pair<int, int>> p;
    DSU *dsu;

    void calc(int u, int last) {
        dfn[u] = ++cnt;
        for (int v : E[u]) if (v ≠ last) calc(v, u);
        cnt++;
        for (int i : q[u]) {
            int v = v = p[i].first + p[i].second - u;
            if (dfn[v] ⩾ dfn[u] and dfn[v] ⩽ cnt) ans[i] = u;
        }
    }
```

```cpp

    void dfs(int u, int last) {
        vis[u] = true;
        for (int i : q[u]) {
            int v = p[i].first + p[i].second - u;
            if (not ans[i] and vis[v]) ans[i] = dsu->shallow(v);
        }
        for (int v : E[u]) if (v != last) dfs(v, u);
        if (u != rt) dsu->merge(u, last);
    }

public:
    std::vector<int> ans;

    TarjanLCA(std::vector<std::vector<int>> &E, int rt,
      std::vector<std::pair<int, int>> &p): E(E), rt(rt), p(p) {
        n = E.size(), m = p.size(), cnt = 0;
        q.resize(n + 1), vis.resize(n + 1),
        dfn.resize(n + 1), ans.resize(m);
        for (int i = 0; i < p.size(); i++)
            q[p[i].first].push_back(i),
            q[p[i].second].push_back(i);
        dsu = new DSU(n);
        calc(rt, 0), dfs(rt, 0);
    }
};
```

1.2.4 $\mathcal{O}(n)-\mathcal{O}(1)$ 欧拉序转化为静态 RMQ

Def 欧拉序列 定义 1.2.3

定义一棵树的欧拉序列为 dfs 过程中每次进入一个点时记下其编号构成的序列.



以该树为例, 其欧拉序列为:

[1, 2, 4, 2, 5, 9, 5, 10, 5, 2, 1, 3, 6, 3, 7, 3, 8, 3, 1]

容易发现欧拉序列长度为 $2n-1$, 因为根节点在开头出现一次, 后续每条边会生成两个序列中的元素. 注意到 u,v 的 LCA 即为 u 和 v 在序列中任意一次出现的位置之间, 所有结点中深度最小的那个. 设 p_u 表示 u 在序列中第一次出现的位置, 查询 $[p_u,p_v]$ 之间深度最小的结点, 这个可以用静态 RMQ 维护, 可以做到 $\mathcal{O}(n)-\mathcal{O}(1)$.

```cpp
class LCAtoRMQ {
private:
    int n, rt;
    std::vector<std::vector<int>> E;
    std::vector<int> dep, pos;
    std::vector<std::pair<int, int>> eul;
    RMQ<std::pair<int, int>> *seq;

    void dfs(int u, int last) {
        dep[u] = dep[last] + 1;
        eul.push_back(std::make_pair(dep[u], u));
        if (not pos[u]) pos[u] = eul.size() - 1;
        for (int v : E[u]) if (v ≠ last)
            dfs(v, u), eul.push_back(std::make_pair(dep[u], u));
    }

public:
    LCAtoRMQ(std::vector<std::vector<int>> E, int rt): E(E), rt(rt) {
        n = E.size() - 1;
        dep.resize(n + 1), pos.resize(n + 1);
        dfs(rt, 0);
        seq = new RMQ<std::pair<int, int>>(eul,
          [](std::pair<int, int> x, std::pair<int, int> y) {
            return std::min(x, y);
        });
    }

    int LCA(int u, int v) {
        if (pos[u] > pos[v]) std::swap(u, v);
        return seq→qry(pos[u], pos[v]).second;
    }
};
```

1.3 线段树

Def 线段树 **定义 1.3.1**

线段树是一种常用数据结构，其用于维护可并的序列信息. 线段树的结构为一棵完全二叉树，叶节点维护序列上的单点信息，非叶结点维护其儿子信息的并. 容易发现，线段树的高度为 $\log_2 n$，结点个数约为 $2n$.

1.3.1 半群单点修改，区间求积

Def 半群 **定义 1.3.2**

若集合 $\mathbb{S}$ 和二元运算 $\xi : \mathbb{S} \times \mathbb{S} \to \mathbb{S}$ 对任意 $x, y, z \in \mathbb{S}$，均满足 $\xi(\xi(x, y), z) = \xi(x, \xi(y, z))$，则称 $(\mathbb{S}, \xi)$ 为半群.

e.g. 一次函数单点修改，区间复合 **例 1.3.1**

给定 n 个一次函数形成的序列 $f_1, f_2, \cdots, f_n$，按顺序执行 q 个操作，形如：

- p, a, b: 将 f_p 修改为 $f_p(x) = ax + b$.
- l, r, x: 求 $f_r(f_{r-1}(\cdots f_l(x))) \bmod 998244353$.

Point Set Range Composite – Library Checker

建立线段树，根节点维护区间 $[1, n]$，对任意非叶结点 $[l, r]$，其左右儿子分别维护区间 $\left[l, \frac{l+r}{2}\right]$ 和 $\left[\frac{l+r}{2} + 1, r\right]$，直到叶结点维护区间 $[l, r]$ 满足 $l = r$.

叶节点的信息来源于其对应函数 f_l，非叶结点维护的区间半群积可以 $\mathcal{O}(1)$ 从其左右儿子处算得，即 $a_r(a_l x + b_l) + b_r = (a_l a_r)x + (a_r b_l + b_r)$.

注意到预处理时线段树上每个结点消耗的计算次数为 $\mathcal{O}(1)$，修改时最多影响到线段树中的 $\log_2 n$ 个结点，查询时答案可用不超过 $2\log_2 n$ 个区间合并得到. 于是支持半群单点修改，区间求积的线段树复杂度为 $\mathcal{O}(n)$ – $\mathcal{O}(\log n)$.

1.3.2 较复杂的合并规则

e.g. 单点修改，区间最大子段和 例 **1.3.2**

给定长度为 n 的整数序列 $a_1, a_2, \cdots, a_n$，按顺序执行 q 个操作，形如：

- p, x：将 a_p 修改为 x.
- l, r：输出序列 $a_l, a_{l+1}, \cdots, a_r$ 的最大子段和.

Can you answer these queries III – SPOJ

与上文所述类似的，考虑建立线段树，但我们发现合并操作并不那么简单，因为区间的最大子段和并不能直接通过左右区间的最大子段和得到.

我们称左子树维护的区间为「左区间」，右子树维护的区间为「右区间」，注意到区间最大子段和只有三种情况：

- 全在左区间内，可以通过左子树维护的最大子段和得到.
- 全在右区间内，可以通过右子树维护的最大子段和得到.
- 跨两个区间，这时其一定能被刻画为左区间的最大后缀和加右区间的最大前缀和.

于是对每个区间多维护「最大前缀和」和「最大后缀和即可」，而对于它们的更新：

- 最大前缀和等于左区间的最大前缀和或左区间的和加右区间的最大前缀和.
- 最大后缀和等于右区间的最大后缀和或右区间的和加左区间的最大后缀和.

再维护一个熟知的区间和即可 $\mathcal{O}(1)$ 合并子区间信息，总时间复杂度 $\mathcal{O}(n)$ – $\mathcal{O}(\log n)$.

1.3.3 半群区间自同态作用，区间求积

Def 自同态 定义 **1.3.3**

若 $f : \mathbb{S} \to \mathbb{S}$ 满足 $\forall x, y \in \mathbb{S}, f(\xi(x, y)) = \xi(f(x), f(y))$，则称 f 为 $(\mathbb{S}, \xi)$ 的自同态. $(\mathbb{S}, \xi)$ 的所有自同态构成的集合被称作 $\mathrm{End}(\mathbb{S}, \xi)$.

e.g. 区间仿射，区间求和 例 **1.3.3**

给定长度为 n 的整数序列 $a_1, a_2, \cdots, a_n$，按顺序执行 q 个操作，形如：

- l, r, b, c：对于所有 $i \in [l, r]$，将 a_i 修改为 $a_i \times b + c$.
- l, r：求出 $\sum_{i=l}^{r} a_i \bmod 998244353$.

Range Affine Range Sum – Library Checker

在区间修改的情况下，每次自同态无法立即修改它作用到的所有结点. 于是对于每个结点，考虑不仅维护其半群积，还要维护其自同态. 于是结点的实际半群积是其维护的半群积按由近到远的顺序经过其所有祖先的自同态作用后得到的元素.

考虑修改操作，在其作用于某个结点对应的区间时，只需要修改该结点. 考虑查询操作，在访问任意结点前，其所有祖先的自同态都需要通过分别作用于其左右子节点，修改为恒等映射. 容易发现，修改单个结点和下传单个结点的自同态是 $\mathcal{O}(1)$ 的. 于是该算法，也即「懒标记线段树」的时间复杂度仍为 $\mathcal{O}(n) - \mathcal{O}(\log n)$.

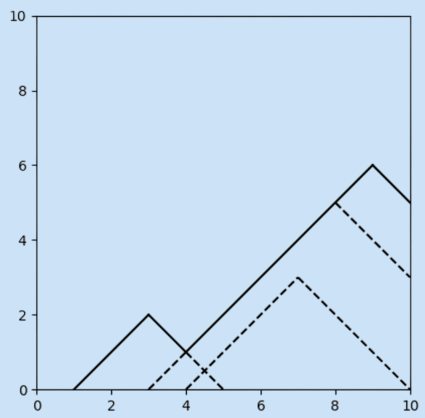
1.3.4 动态开点

考虑一些维护区间的长度很大的线段树，我们发现其中很多点是无用的，也即不论是修改还是查询都不会访问到. 于是考虑让线段树一开始只有根节点，在需要用到某个结点时再动态构建它. 以下给出一个动态开点线段树的实现框架：

```
class SegmentTree {
private:
    class Node {
    public:
        Node *lc, *rc;
        // 定义需要维护的变量
        Node (int l, int r) : lc(nullptr), rc(nullptr) /*初始化需要维护的变量*/ {}
    } *root = nullptr;
    int limL, limR;

    void pushdown (Node *p, int L, int R) {
        // if (p->lazy == 0) return;
        int mid = L + R >> 1;
        if (p->lc == nullptr) p->lc = new Node(L, mid);
        // 下传左子树
        if (p->rc == nullptr) p->rc = new Node(mid + 1, R);
        // 下传右子树
        // p->lazy = 0;
    }

    void pushup (Node *p, int L, int R) {
        int mid = L + R >> 1;
        if (p->lc == nullptr) p->lc = new Node(L, mid);
        if (p->rc == nullptr) p->rc = new Node(mid + 1, R);
        // 子树信息合并得到结点 p 的信息
    }

    void upd (int l, int r,/*需要做修改的参数*/ Node *&p, int L, int R) {
        if (p == nullptr) p = new Node(L, R);
```

```
30          if (l ⩽ L and R ⩽ r) { /*对合法区间做修改*/ return; }
31          pushdown(p, L, R);
32          int mid = L + R >> 1;
33          if (l ⩽ mid) upd(l, r,/*需要做修改的参数*/ p→lc, L, mid);
34           if (r ⩾ mid + 1) upd(l, r,/*需要做修改的参数*/ p→rc, mid +
    1, R);
35          pushup(p, L, R);
36      }
37
38      /*需要查询的值类型*/ qry (int l, int r, Node* p, int L, int R) {
39          if (p == nullptr) return /*空结点的信息*/;
40          if (l ⩽ L and R ⩽ r) return /*结点 p 的信息*/;
41          pushdown(p, L, R);
42          int mid = L + R >> 1;
43          auto res = /*需要查询的值类型的默认值*/;
44          if (l ⩽ mid) res = /*res 与左子树查询到的信息合并*/;
45          if (r ⩾ mid + 1) res = /*res 与右子树查询到的信息合并*/;
46          return res;
47      }
48
49      void del (Node* u) {
50          if (u→lc ≠ nullptr) del(u→lc);
51          if (u→rc ≠ nullptr) del(u→rc);
52          delete u;
53          u = nullptr;
54      }
55
56  public:
57      SegmentTree (int l, int r) : limL(l), limR(r), root(nullptr) {}
58
59      // 提供给外界调用的接口
60
61      ~SegmentTree () { del(root); }
62  };
```

e.g. 区间修改，区间查询最小值及其出现次数 例 **1.3.4**



如图所示，每座山都是三角形，用其顶点坐标代表.

有 q 次操作，每次操作形如：

- x, y：如果不存在顶点位于 (x, y) 的山，则将其删除，否则将这座山加入图中.

每次操作后求出从上往下看能看到的线条长度.

[NAC2024] H. MountainCraft – UCUP

容易发现此题本质上是每次给某个区间 $+1$ 或 -1, 需要动态维护序列上 0 的数量.

为了做到这件事情, 我们考虑维护区间最小值和其出现次数, 合并时如果子区间的最小值相同, 则该区间的最小值数量为左右区间的最小值数量之和, 否则继承其中一个子区间的信息. 合并可以做到 $\mathcal{O}(1)$, 于是整体是复杂度为 $\mathcal{O}(n)-\mathcal{O}(\log n)$.

```
class SegmentTree {
private:
    class Node {
    public:
        Node *lc, *rc;
        int min, cnt, lazy;
        Node (int l, int r) : lc(nullptr), rc(nullptr), min(0), cnt(r - l + 1), lazy(0) {}
    } *root = nullptr;
    int limL, limR;

    void pushdown (Node *p, int L, int R) {
        if (p->lazy == 0) return;
        int mid = L + R >> 1;
        if (p->lc == nullptr) p->lc = new Node(L, mid);
        p->lc->min += p->lazy, p->lc->lazy += p->lazy;
        if (p->rc == nullptr) p->rc = new Node(mid + 1, R);
        p->rc->min += p->lazy, p->rc->lazy += p->lazy;
        p->lazy = 0;
    }

    std::pair<int, int> merge (std::pair<int, int> a, std::pair<int, int> b) {
        int min = std::min(a.first, b.first), cnt = 0;
        if (a.first == min) cnt += a.second;
        if (b.first == min) cnt += b.second;
        return std::make_pair(min, cnt);
    }

    void upd (int l, int r, int k, Node *&p, int L, int R) {
        if (p == nullptr) p = new Node(L, R);
        if (l <= L and R <= r) { p->min += k, p->lazy += k; return; }
        pushdown(p, L, R);
        int mid = L + R >> 1;
        if (l <= mid) upd(l, r, k, p->lc, L, mid);
        if (r >= mid + 1) upd(l, r, k, p->rc, mid + 1, R);
        if (p->lc == nullptr and p->rc == nullptr) std::tie(p->min, p->cnt) = std::make_pair(0, R - L + 1);
            else if (p->lc == nullptr) std::tie(p->min, p->cnt) = std::make_pair(0, mid - L + 1 + p->rc->cnt * (p->rc->min == 0));
            else if (p->rc == nullptr) std::tie(p->min, p->cnt) = std::make_pair(0, R - mid + p->lc->cnt * (p->lc->min == 0));
```

```
38          else std::tie(p->min, p->cnt) = merge(std::make_pair(p->lc->min, p->lc->cnt), std::make_pair(p->rc->min, p->rc->cnt));
39      }
40
41      std::pair<int, int> qry (int l, int r, Node *p, int L, int R) {
42          if (p == nullptr) return std::make_pair(0, std::min(r, R) - std::max(l, L) + 1);
43          if (l <= L and R <= r) return std::make_pair(p->min, p->cnt);
44          pushdown(p, L, R);
45          int mid = L + R >> 1;
46          auto res = std::make_pair(INT_MAX, 0);
47          if (l <= mid) res = merge(res, qry(l, r, p->lc, L, mid));
48          if (r >= mid + 1) res = merge(res, qry(l, r, p->rc, mid + 1, R));
49          return res;
50      }
51
52  public:
53      SegmentTree (int l, int r) : limL(l), limR(r), root(nullptr) {}
54
55      void upd (int l, int r, int k) {
56          upd(l, r, k, root, limL, limR);
57      }
58
59      std::pair<int, int> qry (int l, int r) {
60          return qry(l, r, root, limL, limR);
61      }
62  };
```

1.3.5 树上半群路径修改，路径求积

e.g. 树上路径/子树修改，路径/子树求和 例 **1.3.5**

给定一棵 n 个结点的树，每个点均有点权，按顺序进行以下 q 个操作，形如：

- $1, x, y, z$：将树上从 x 到 y 最短路径上的所有结点权值加 z.
- $2, x, y$：求树上从 x 到 y 最短路径上的所有结点权值之和.
- $3, x, y$：将树上以 x 为根的子树内所有结点权值加 y.
- $4, x$：求树上以 x 为根的子树内所有结点权值之和.

「模板」重链剖分/树链剖分 – 洛谷

考虑之前提到的树链剖分技术. 假设我们以优先重儿子的顺序遍历整棵树，那么重链上结点的 dfs 序都是连续的，且在任意先序遍历中子树内结点的 dfs 序也是连续的.

因此我们考虑将对树上路径的修改和查询拆分为对不超过 $2\log_2 n$ 条重链的修改和查询，每次修改和查询都是对线段树上一段连续的区间进行的，可以做到单次 $\mathcal{O}(\log n)$ 的时间复杂度. 于是应用这种技巧，总时间复杂度能做到 $\mathcal{O}(n)$ – $\mathcal{O}(\log^2 n)$.

1.3.6 李超线段树

e.g. 插入线段，询问 $f(x_0)$ 的最小值 **例 1.3.6**

平面上一开始什么都没有，按顺序执行 n 个操作，形如：

- $0, k$: 查询平面上与直线 $x = k$ 相交的线段中，交点纵坐标最大的线段编号.
- $1, x_0, y_0, x_1, y_1$: 插入线段 $(x_0, y_0)(x_1, y_1)$，其编号为被插入的顺序.

[HEOI2013] Segment – 洛谷

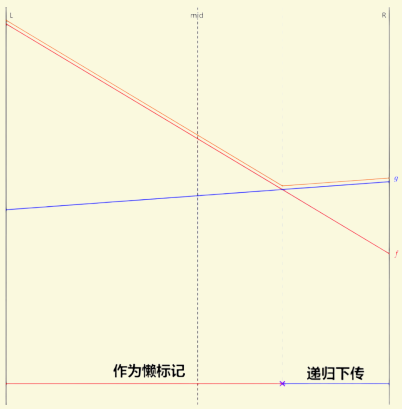
考虑将加入的线段转化为 $y = ax + b$ $(x \in [l, r])$ 的形式.

⚠ 加入线段 $(x_0, y_0)(x_0, y_1)$，其中 $y_0 \leqslant y_1$ **注意 1.3.1**

这种情况下求出直线的斜率为 ∞，可以将其看成 $y = 0 \cdot x + y_1$ $(x \in [x_0, x_0])$.

进行修改的时候，我们很自然地想到记录每个区间的「绝对优势线段」，也即对任意横坐标，该线段上的纵坐标都大于其它所有线段在此处的取值.

红线为我们需要插入的线段，考虑某个被其完整覆盖的线段树区间：



- 若该区间无绝对优势线段，则将其赋值为该线段.
- 否则，注意到当前区间的更新有三种情况：
  - 新线段完全优于之前的线段，则直接更新.
  - 新线段完全劣于之前的线段，则不进行操作.
  - 新线段与之前的线段在该区间内有交点，则一定有某条线段在某个子区间上严格优于另一条线段，将其作为该区间的「绝对优势线段」，而另一条线段则往其可能成为「绝对优势线段」的子区间递归下去.

注意到每次修改可能涉及到 $2\log_2 n$ 个区间的修改，每个区间内最多递归 $\log_2 n$ 次，于是添加一条线段的时间复杂度为 $\mathcal{O}(\log^2 n)$.

对于查询，由于我们在某条线段一定能成为整个区间的「绝对优势线段」时直接将原线段覆盖了，所以查询答案时需要借用标记永久化的思想，记录从根开始整条路径上的最优值. 但这并不会使复杂度增加，于是查询操作的时间复杂度为 $\mathcal{O}(\log n)$.

1.4 并查集

并查集用于维护一些有传递性的关系，比如连通性.

在某些时候我们还需要维护结点与其父亲间的权值，而该信息在路径压缩时会被破坏，所以需要特别处理.

关于并查集的复杂度，朴素的并查集是 $\mathcal{O}(n^2)$ 的. 带路径压缩或按秩合并优化可以做到 $\mathcal{O}(n\log n)$，两个优化一起使用可以做到 $\mathcal{O}(n\alpha(n))$. 其中 α 是反阿克曼函数，在一般的数据范围下可以看作常数 4.

```cpp
class DSU {
private:
    bool type;
    int n;
    std::vector<int> fa, pnum, val;

    int _find(int x) {
        if (x != fa[x]) fa[x] = _find(fa[x]);
        return fa[x];
    }

    bool _merge(int x, int y) {
        int Rx = _find(x), Ry = _find(y);
        if (x != y) {
            if (pnum[Rx] < pnum[Ry])
                fa[Rx] = Ry, pnum[Ry] += pnum[Rx];
            else fa[Ry] = Rx, pnum[Rx] += pnum[Ry];
        }
        return Rx != Ry;
    }

    int find_(int x) {
        if (x != fa[x]) {
            int mark = fa[x];
            fa[x] = find_(fa[x]), val[x] += val[mark];
        }
        return fa[x];
    }

    bool merge_(int x, int y, int w) {
        int Rx = find_(x), Ry = find_(y);
        if (Rx == Ry) return (val[x] - val[y] == w);
        if (pnum[Rx] < pnum[Ry])
            fa[Rx] = Ry,
            pnum[Ry] += pnum[Rx],
            val[Rx] = val[y] + w - val[x];
        else fa[Ry] = Rx,
             pnum[Rx] += pnum[Ry],
```

```
39                 val[Ry] = val[x] - w - val[y];
40             return true;
41         }
42
43     public:
44         DSU(int n, bool type): n(n), type(type) {
45             for (int i = 0; i <= n; i++) fa.push_back(i);
46             pnum = std::vector<int>(n + 1, 1);
47             if (type) val.resize(n + 1);
48         }
49
50         int find(int u) {
51             return type ? find_(u) : _find(u);
52         }
53
54         bool merge(int u, int v) {
55             return _merge(u, v);
56         }
57
58         bool merge(int u, int v, int w) {
59             return merge_(u, v, w);
60         }
61     };
```

1.5 分治

1.5.1 维护凸包

e.g. 插入直线，询问最小值 **例 1.5.1**

给定 n 条直线，按顺序执行 q 次操作，形如：

- $0, a, b$：插入直线 $y = ax + b$.
- $1, p$：求 $\min y_{x=p}$.

Line Add Get Min – Library Checker

首先问题可以转化为插入点 (a, b)，求 $(-p, -1)$ 与其中某个点的最大点积.

如果没有插入操作：答案一定是凸包上的点. 将凸包分为上下两个凸壳后，注意到答案是凸的，于是直接二分即可.

对于原问题考虑将点分组，设 $n = 2^{k_0} + 2^{k_1} + \cdots$，分别维护大小为 2^{k_i} 的凸包. 插入一个点的时候其形成一个新的包含一个点的凸包. 类似二进制意义下的 $+1$，每当存在两个大小为 2^k 的凸包就将它们合并为一个大小为 2^{k+1} 的凸包，直到没有相同大小的凸包.

注意到每个点最多被合并 $\log_2 n$ 次，在每次合并中它对复杂度的贡献都是 $\mathcal{O}(1)$ 的，于是插入操作的时间复杂度为均摊 $\mathcal{O}(\log n)$. 对于询问，在不超过 $\log_2 n$ 个凸包上分别二分即可，时间复杂度为 $\mathcal{O}(\log^2 n)$.

1.5.2 可持久化

e.g. 区间第 k 小 **例 1.5.2**

给定长度为 n 的序列 $a_1, a_2, \cdots, a_n$，按顺序执行以下 q 个询问：

- l, r, k：求出集合 $\{a_l, a_{l+1}, \cdots, a_r\}$ 中第 k 小的值.

「模板」可持久化线段树 2 – 洛谷

如果是全局第 k 小问题，可以在权值线段树上二分查找第 k 小的值，复杂度 $\mathcal{O}(\log n)$.

注意到加入一个权值最多对 $\log_2 n$ 个结点造成影响，于是将该序列的元素从左到右依次加入线段树中，将每次造成影响的点单独取出来建新点，如果其儿子与原树相同就直接连向原树. 这样总共会产生 $\mathcal{O}(n \log n)$ 个结点，故时间与空间复杂度都是 $\mathcal{O}(n \log n)$.

对于查询，注意到从第 k 次插入操作产生的根节点向下走即为仅考虑前 k 个元素构成的权值线段树，从 rt_{l-1} 和 rt_r 同时向下走，取其差作为实际值即可转化为全局第 k 小问题.

```cpp
class PersistentVT {
private:
    int n, w, cnt;
    std::vector<int> val, rt, num, lc, rc;

    void build(int &x, int y, int l, int r, int k) {
        x = ++cnt, num[x] = num[y] + 1;
        lc[x] = lc[y], rc[x] = rc[y];
        if (l == r) return;
        int mid = l + r >> 1;
        if (k <= mid) build(lc[x], lc[y], l, mid, k);
        else build(rc[x], rc[y], mid + 1, r, k);
    }

    int query(int x, int y, int l, int r, int k) {
        if (l == r) return val[l];
        int mid = l + r >> 1;
        int sum = num[lc[x]] - num[lc[y]];
        if (k <= sum) return query(lc[x], lc[y], l, mid, k);
        else return query(rc[x], rc[y], mid + 1, r, k - sum);
    }

public:
    PersistentVT(std::vector<int> a): cnt(0) {
        n = a.size(), rt.resize(n + 1);
        val = discretize({&a}), w = val.size();
        size_t maxNum = 2 * (n + 1) * (log2(w) + 1);
        num.resize(maxNum), lc.resize(maxNum), rc.resize(maxNum);
        for (size_t i = 1; i <= n; i++)
            build(rt[i], rt[i - 1], 0, w, a[i - 1]);
    }

    int qry(int l, int r, int k) {
        return query(rt[r], rt[l - 1], 0, w, k);
    }
};
```

1.5.3 点分树

对于一棵树, 假设它的重心是 rt, 那么以 rt 为根会产生若干棵与它直接相连的子树 T_{u_i}. 将 rt 与 u_i 的连边断掉, 添加 rt 到 T_{u_i} 重心的连边. 对 T_{u_i} 的重心递归做相同的操作. 全部递归结束以后我们发现现在的树变成了原树的一棵重构树, 且有以下性质:

- 重构树中包含原树上所有结点恰好一次.
- 重构树树高不大于 $\log_2 n$, n 表示原树结点数量.
- 重构树上任意子树在原树中也连通.

我们称这棵重构树为「点分树」, 由于上述性质的存在, 原本一些复杂度很不对劲的暴力做法在点分树上会变得正确起来, 比如一边暴力向上跳一边维护一些信息做容斥.

```cpp
class DivideTree {
private:
    int n;
    std::vector<std::vector<int>> E;
    std::vector<int> max, siz;
    std::vector<bool> vis;

    void updsiz(int u, int last) {
        siz[u] = 1;
        for (int v : E[u]) if (v ≠ last and not vis[v])
            updsiz(v, u), siz[u] += siz[v];
    }

    void find(int u, int last, int m, int &rt) {
        siz[u] = 1, max[u] = 0;
        for (int v : E[u]) if (v ≠ last and not vis[v]) {
            find(v, u, m, rt), siz[u] += siz[v];
            max[u] = std::max(max[u], siz[v]);
        }
        max[u] = std::max(max[u], m - siz[u]);
        if (max[u] < max[rt]) rt = u;
    }

    void build(int u, int last) {
        vis[u] = true;
        for (int v : E[u]) if (v ≠ last and not vis[v]) {
            int rt = 0;
            find(v, u, siz[v], rt), updsiz(rt, u);
            T[u].push_back(rt), T[rt].push_back(u);
            build(rt, u);
        }
    }

public:
    int root;
    std::vector<std::vector<int>> T;

    DivideTree(std::vector<std::vector<int>> E): E(E), root(0) {
        n = E.size();
        T.resize(n), max.resize(n), siz.resize(n), vis.resize(n);
        max[0] = n;
        find(1, 0, n - 1, root), updsiz(root, 0);
        build(root, 0);
    }
};
```

1.6 分块

分块是一种思想，一般伴随着复杂度分析技巧使用. 与线段树不同的是，分块一般用于处理不太好合并的信息，为了加快修改和查询，将它们「打包」成簇，有时还会伴随重构.

1.6.1 分块构造

e.g. Manhattan 旅行商问题 例 1.6.1

给定平面上值域为 $[0,m]$ 的 n 个整点 (x_i,y_i)，求排列 $\{p_n\}$ 使

$$\sum_{i=1}^{n-1}\left|x_{p_{i+1}}-x_{p_i}\right|+\left|y_{p_{i+1}}-y_{p_i}\right| \tag{1.2}$$

尽可能小.

我们总能构造出 $\mathcal{O}(m\sqrt{n})$ 的解.

将点按 $\left(\left\lfloor\frac{x_i}{B}\right\rfloor,y_i\right)$ 排序. 对于块内移动，x 方向上的总移动距离不大于 nB，y 方向上的总移动距离不大于 $\frac{m^2}{B}$. 对于块间移动，最多进行 $\frac{m}{B}$ 次，每次移动距离不超过 m.

取 $B=mn^{-\frac{1}{2}}$ 得总路径长度不超过 $m\sqrt{n}$.

1.6.2 莫队算法

e.g. 静态区间不同值个数 例 1.6.2

给定长度为 n 的整数序列 $a_1,a_2,\cdots,a_n$，依次进行 q 个询问，形如：

- l,r：求出集合 $\{a_l,a_{l+1},\cdots,a_r\}$ 中不同值的数量.

[SDOI2009] HH 的项链 – 洛谷

1.6.2.1 $\mathcal{O}(n\sqrt{q})$ 莫队

考虑按顺序枚举所有询问，假设当前枚举到 $[l_i,r_i]$，每种值的出现次数记为 cnt_x，当前答案为 ans. 那么接着枚举 $[l_{i+1},r_{i+1}]$ 的时候，l_i 要向着 l_{i+1} 移动，r_i 同理. 而在移动的过程中会有一些元素被加入或删除，我们及时维护 cnt 和 ans 的变化即可动态维护每个询问的答案. 容易发现加入或删除元素都是 $\mathcal{O}(1)$ 的，于是这个做法的时间复杂度为 $\mathcal{O}(nq)$.

面对这个朴素的 brute，我们第一时间想到的肯定是用某数据结构维护一段元素的插入和删除，而湖南选手莫涛给出了一种仅通过更改询问的枚举顺序就能优化复杂度的想法.

具体来说，我们将序列分为 B 块，每块长度为 $\frac{n}{B}$. 将询问按左端点所在块的编号为第一关键字，右端点为第二关键字从小到大排序.

进行如下复杂度分析: 对于左端点在相同块内移动的询问, 左端点总共移动不超过 $\frac{qn}{B}$ 次. 右端点总共移动不超过 nB 次. 对于左端点在相邻块之间移动的询问, 最多发生 B 次, 每次左右端点总移动次数不超过 $2n$. 于是总移动次数不超过 $q\frac{n}{B}+3nB$, 取 $B=\sqrt{q}$ 可以取到时间复杂度的最小值 $\mathcal{O}(n\sqrt{q})$.

考虑一个常数优化, 注意到如果按上述方法排序, 每次进行相邻块之间的移动时, 右端点几乎都要从最右边回到最左边. 而如果我们将左端点处于奇数块的询问按右端点从小到大排序, 将左端点处于偶数块的询问按右端点从大到小排序就可以避免这一点.

```cpp
class MoCaptainship {
private:
    class DifferentColors {
        friend class MoCaptainship;
        int m, num;
        std::vector<int> a, cnt;

        void add(int x) { num += not cnt[x]++; }
        void del(int x) { num -= not --cnt[x]; }

    public:
        DifferentColors(int m): m(m), num(0) {
            cnt.resize(m + 1);
        }
    };

    class Question {
        friend class MoCaptainship;
        int id, l, r, ans;
    public:
        Question(int id, int l, int r): id(id), l(l), r(r), ans(0) {}
    };

    int n, q;
    std::vector<int> a, bel;
    std::vector<Question> ask;

public:
    MoCaptainship(int n, std::vector<int> &a,
      std::vector<std::pair<int, int>> &seg): n(n), a(a) {
        q = seg.size(), bel.resize(n + 1);
        int B = sqrt(q), L = ceil(1.0 * n / B);
        for (size_t i = 1; i <= B; i++)
            for (size_t j = (i - 1) * L + 1; j <= i * L; j++)
                if (j <= n) bel[j] = i;

        for (int i = 0; i < q; i++)
            ask.push_back(Question(i, seg[i].first, seg[i].second));
        std::sort(ask.begin(), ask.end(),
          [this](const Question &a, const Question &b) {
```

```
41              return (bel[a.l] ^ bel[b.l]) ? bel[a.l] < bel[b.l] :
42                ((bel[a.l] & 1) ? a.r < b.r : a.r > b.r);
43          });
44
45          DifferentColors diff(1000000);
46          int l = 1, r = 0;
47          for (auto &qs : ask) {
48              while (l < qs.l) diff.del(a[l++]);
49              while (l > qs.l) diff.add(a[--l]);
50              while (r < qs.r) diff.add(a[++r]);
51              while (r > qs.r) diff.del(a[r--]);
52              qs.ans = diff.num;
53          }
54
55          std::sort(ask.begin(), ask.end(),
56            [this](const Question &a, const Question &b) {
57              return a.id < b.id;
58          });
59      }
60
61      int ans(int id) { return ask[id].ans; }
62  };
```

1.6.2.2 $\mathcal{O}((n+q)\log n)$ 树状数组

事实上此题数据经过加强以后朴素的莫队算法已经无法通过了，所以哪怕正解和莫队算法没有什么关系我们仍然提一下，以求严谨.

注意到在右端点确定的情况下，如果存在若干个相同的数字，只有最右边那个有用. 于是将询问按右端点排序，设当前考虑到的右端点为 i. 设序列 $\{b_n\}(b_i \in \{0,1\})$ 表示目前 a 序列中的每个点是否有用，那么加入 a_i 时 b_i 会变成 1，而 a_i 如果存在上一次出现的位置，其对应的 b 值会变为 0. 对于一次左端点在 l 处的查询，只需要求出 $\sum_{j=l}^{i} b_j$ 即可. 维护某个值上一次出现的位置直接用数组记录，维护单点修改和区间求和用树状数组即可. 于是总时间复杂度为 $\mathcal{O}((n+q)\log n)$.