目次

1 data-structure

1.1 Sparse Table

Description

Sparse table は，冪等モノイド $(T, \cdot, e)$ の静的な列の区間積を高速に計算するデータ構造である．

冪等な二項演算とは， $\forall a \in T, a \cdot a = a$ が成り立つような写像 $\cdot : T \times T \to T$ である．冪等二項演算には，max, min, gcd, bitwise and, bitwise or などがある．

空間計算量: $O(n \log n)$

Operations

- `SparseTable(vector<T> v)`
  - v の要素から sparse table を構築する
  - 時間計算量: $O(n \log n)$
- `T fold(int l, int r)`
  - 区間 $[l, r)$ の値を fold する
  - 時間計算量: $O(1)$

```cpp
template <typename M>
class SparseTable {
    using T = typename M::T;

   public:
    SparseTable() = default;
    explicit SparseTable(const std::vector<T>& v) {
        int n = v.size(), b = 0;
        while ((1 << b) <= n) ++b;
        lookup.resize(b, std::vector<T>(n));
        std::copy(v.begin(), v.end(), lookup[0].begin());
        for (int i = 1; i < b; ++i) {
            for (int j = 0; j + (1 << i) <= n; ++j) {
                lookup[i][j] =
                    M::op(lookup[i - 1][j], lookup[i - 1][j + (1 << (i - 1))]);
            }
        }
    }

    T fold(int l, int r) const {
        if (l == r) return M::id();
        int i = 31 - __builtin_clz(r - l);
        return M::op(lookup[i][l], lookup[i][r - (1 << i)]);
    }

   private:
    std::vector<std::vector<T>> lookup;
};
```

1.2 2D Segment Tree

Description

2D セグメント木は，モノイド $(T, \cdot, e)$ の重みを持つ 2 次元平面上の点集合に対する一点更新と矩形領域積取得を提供するデータ構造である．

この実装では，重みをもたせる点を先読みして初期化時に渡す必要がある．

空間計算量: $O(n)$

Operations

- `SegmentTree2D(vector<pair<X, Y>> pts)`
  - pts の点に対する 2D セグメント木を初期化する
  - 時間計算量: $O(n \log n)$
- `void update(X x, Y y, T val)`
  - 点 (x, y) の重みを val に更新する
  - 時間計算量: $O((\log n)^2)$
- `T fold(X sx, X tx, Y sy, Y ty)`
  - 矩形領域 $[sx, tx) \times [sy, ty)$ 内の点の重みの積を取得する
  - 時間計算量: $O((\log n)^2)$

```cpp
#include "segment_tree.cpp"

template <typename X, typename Y, typename M>
class SegmentTree2D {
    using T = typename M::T;

public:
    SegmentTree2D() = default;
    explicit SegmentTree2D(const std::vector<std::pair<X, Y>>& pts) {
        for (auto& [x, y] : pts) {
            xs.push_back(x);
        }
        std::sort(xs.begin(), xs.end());
        xs.erase(std::unique(xs.begin(), xs.end()), xs.end());

        const int n = xs.size();
        size = 1;
        while (size < n) size <<= 1;
        ys.resize(2 * size);
        seg.resize(2 * size);

        for (auto& [x, y] : pts) {
            ys[size + getx(x)].push_back(y);
        }

        for (int i = 0; i < n; ++i) {
            std::sort(ys[size + i].begin(), ys[size + i].end());
            ys[size + i].erase(std::unique(ys[size + i].begin(), ys[size + i].end()), ys[
                size + i].end());
        }
        for (int i = size - 1; i > 0; --i) {
            std::merge(ys[2*i].begin(), ys[2*i].end(), ys[2*i+1].begin(), ys[2*i+1].end(),
                std::back_inserter(ys[i]));
            ys[i].erase(std::unique(ys[i].begin(), ys[i].end()), ys[i].end());
        }
        for (int i = 0; i < size + n; ++i) {
            seg[i] = SegmentTree<M>(ys[i].size());
        }
    }

    T get(X x, Y y) const {
        int kx = getx(x);
        assert(kx < (int) xs.size() && xs[kx] == x);
        kx += size;
        int ky = gety(kx, y);
```

```
        assert(ky < (int) ys[kx].size() && ys[kx][ky] == y);
        return seg[kx][ky];
    }

    void update(X x, Y y, T val) {
        int kx = getx(x);
        assert(kx < (int) xs.size() && xs[kx] == x);
        kx += size;
        int ky = gety(kx, y);
        assert(ky < (int) ys[kx].size() && ys[kx][ky] == y);
        seg[kx].update(ky, val);
        while (kx >>= 1) {
            ky = gety(kx, y);
            int kl = gety(2*kx, y), kr = gety(2*kx+1, y);
            T vl = (kl < (int) ys[2*kx].size() && ys[2*kx][kl] == y ? seg[2*kx][kl] : M::
                id());
            T vr = (kr < (int) ys[2*kx+1].size() && ys[2*kx+1][kr] == y ? seg[2*kx+1][kr]
                : M::id());
            seg[kx].update(ky, M::op(vl, vr));
        }
    }

    T fold(X sx, X tx, Y sy, Y ty) const {
        T ret = M::id();
        for (int l = size + getx(sx), r = size + getx(tx); l < r; l >>= 1, r >>= 1) {
            if (l & 1) {
                ret = M::op(ret, seg[l].fold(gety(l, sy), gety(l, ty)));
                ++l;
            }
            if (r & 1) {
                --r;
                ret = M::op(ret, seg[r].fold(gety(r, sy), gety(r, ty)));
            }
        }
        return ret;
    }

private:
    int size;
    std::vector<X> xs;
    std::vector<std::vector<Y>> ys;
    std::vector<SegmentTree<M>> seg;

    int getx(X x) const { return std::lower_bound(xs.begin(), xs.end(), x) - xs.begin(); }
    int gety(int k, Y y) const { return std::lower_bound(ys[k].begin(), ys[k].end(), y) -
        ys[k].begin(); }
};
```

2 math

2.1 Bostan-Mori Algorithm

Description

Bostan-Mori algorithm は，d 階線形漸化式の第 n 項を高速に求めるアルゴリズムである．

Operations

- `T bostan\_mori\_division(Polynomial<T> p, Polynomial<T> q, long long n)`
  - $p(x)/q(x)$ の第 n 項を求める．
  - 時間計算量: $O(\mathsf{M}(k)\log n)$, $\mathsf{M}(k)$ は k 次多項式乗算の計算量 (FFT なら $O(k\log k)$)
- `T bostan\_mori\_recurrence(vector<T> a, vector<T> c, long long n)`
  - 初めの k 項 $a_0,\dots,a_{k-1}$ と漸化式 $a_n = c_0a_{n-1}+\cdots+c_{k-1}a_{n-k}$ によって定まる数列 (a_n) の第 n 項を求める．
  - 時間計算量: 同上

```
#include "../convolution/ntt.hpp"
#include "../math/polynomial.cpp"

template <typename T>
T bostan_mori_division(Polynomial<T> p, Polynomial<T> q, long long n) {
    auto take = [&](const std::vector<T>& p, int s) {
        std::vector<T> r((p.size() + 1) / 2);
        for (int i = 0; i < (int) r.size(); ++i) {
            if (2 * i + s < (int) p.size()) {
                r[i] = p[2 * i + s];
            }
        }
        return r;
    };

    while (n > 0) {
        auto qm = q;
        for (int i = 1; i < (int) qm.size(); i += 2) qm[i] = -qm[i];
        p = take(p * qm, n & 1);
        q = take(q * qm, 0);
        n >>= 1;
    }

    return p[0] / q[0];
}

template <typename T>
T bostan_mori_recurrence(const std::vector<T>& a, const std::vector<T>& c, long long n) {
    const int d = c.size();
    if (n < d) return a[n];

    std::vector<T> q(d + 1);
    q[0] = 1;
    for (int i = 0; i < d; ++i) q[i + 1] = -c[i];
    auto p = convolution(a, q);
    p.resize(d);

    auto take = [&](const std::vector<T>& p, int s) {
        std::vector<T> r((p.size() + 1) / 2);
        for (int i = 0; i < (int) r.size(); ++i) {
            r[i] = p[2 * i + s];
        }
        return r;
    };

    while (n > 0) {
        auto qm = q;
        for (int i = 1; i < (int) qm.size(); i += 2) qm[i] = -qm[i];
        p = take(convolution(p, qm), n & 1);
        q = take(convolution(q, qm), 0);
        n >>= 1;
    }

```

```
    return p[0] / q[0];
}
```

3 string

3.1 Manacher's Algorithm

Description

Manacher のアルゴリズムは，文字列中の回文である部分文字列を求めるアルゴリズムである．

返り値を A とする．S_i を中心とする最大の回文の長さを x とすると，$A[2i] = (x+1)/2$．S_iS_{i+1} を中心とする最大の回文の長さを x とすると，$A[2i+1] = x/2$.

- `vector<int> manacher(string s)`
  - Manacher のアルゴリズムを実行する
  - 時間計算量: $O(n)$

```
std::vector<int> manacher(const std::string& s) {
    int n = s.size();
    std::vector<int> vs(2 * n - 1);
    // odd length
    for (int i = 0, l = 0, r = -1; i < n; ++i) {
        int k = (i > r) ? 1 : std::min(vs[2 * (l + r - i)], r - i + 1);
        while (0 <= i - k && i + k < n && s[i - k] == s[i + k]) ++k;
        vs[2 * i] = k;
        --k;
        if (i + k > r) {
            l = i - k;
            r = i + k;
        }
    }
    // even length
    for (int i = 1, l = 0, r = -1; i < n; ++i) {
        int k = (i > r) ? 0 : std::min(vs[2 * (l + r - i + 1) - 1], r - i + 1);
        while (0 <= i - k - 1 && i + k < n && s[i - k - 1] == s[i + k]) ++k;
        vs[2 * i - 1] = k;
        --k;
        if (i + k > r) {
            l = i - k - 1;
            r = i + k;
        }
    }
    return vs;
}
```

3.2 min_cyclic_shift.hpp

```
/**
 * @brief Minimum Cyclic String
 */

template <typename T>
std::vector<T> min_cyclic_string(const std::vector<T>& s) {
    int n = s.size();
    auto ss = s;
    ss.insert(ss.end(), s.begin(), s.end());
    int i = 0, ans = 0;
    while (i < n) {
        ans = i;
        int j = i + 1, k = i;
        while (j < 2 * n && ss[k] <= ss[j]) {
            if (ss[k] < ss[j])
                k = i;
            else
                k++;
            j++;
        }
        while (i <= k) i += j - k;
    }
    return std::vector<T>(ss.begin() + ans, ss.begin() + ans + n);
}

std::string min_cyclic_string(const std::string& s) {
    auto res = min_cyclic_string(std::vector<char>(s.begin(), s.end()));
    return std::string(res.begin(), res.end());
}
```

4 tree

4.1 Link/Cut Tree

Description

Link/cut tree は，森を管理するデータ構造である．以下の機能を提供する: - 辺の追加 - 辺の削除 - 根の変更 - 頂点の値の更新 - パス上の頂点の値 (モノイド) の総和

木をパスに分解し，それぞれのパスを splay tree で管理することでこれらの操作を実現する．

空間計算量: $O(n)$

Operations

- `LinkCutTree(int n)`
  - 頂点数 n で初期化する
  - 時間計算量: $O(n)$
- `void link(int u, int v)`
  - 辺 uv を追加する
  - 時間計算量: amortized $O(\log n)$
- `void cut(int v)`
  - 頂点 v とその親を結ぶ辺を削除する
  - 時間計算量: amortized $O(\log n)$
- `void evert(int v)`
  - 頂点 v を木の根にする
  - 時間計算量: amortized $O(\log n)$
- `void get(int v)`
  - 頂点 v の値を取得する
  - 時間計算量: $O(1)$
- `void set(int v, T x)`
  - 頂点 v の値を x に変更する
  - 時間計算量: amortized $O(\log n)$
- `T fold(int u, int v)`

- uv パス上の頂点の値を fold する
- 時間計算量: amortized $O(\log n)$

```cpp
template <typename M, typename M::T (*flip)(typename M::T)>
class LinkCutTree {
    using T = typename M::T;

public:
    LinkCutTree() = default;
    explicit LinkCutTree(int n) {
        for (int i = 0; i < n; ++i) {
            vertex.push_back(std::make_shared<Node>(M::id));
        }
    }

    void link(int v, int p) {
        evert(v);
        expose(vertex[p]);
        vertex[v]->par = vertex[p];
        vertex[p]->right = vertex[v];
        recalc(vertex[p]);
    }

    void cut(int v) {
        expose(vertex[v]);
        auto p = vertex[v]->left;
        vertex[v]->left = p->par = nullptr;
        recalc(vertex[v]);
    }

    void evert(int v) {
        expose(vertex[v]);
        reverse(vertex[v]);
    }

    T get(int v) const {
        return vertex[v]->val;
    }

    void set(int v, const T& x) {
        expose(vertex[v]);
        vertex[v]->val = x;
        recalc(vertex[v]);
    }

    T fold(int u, int v) {
        evert(u);
        expose(vertex[v]);
        return vertex[v]->sum;
    }

private:
    struct Node;
    using node_ptr = std::shared_ptr<Node>;

    struct Node {
        node_ptr left, right, par;
        T val, sum;
        int sz;
        bool rev;

        Node(const T& x)
            : left(nullptr), right(nullptr), par(nullptr),
              val(x), sum(x), sz(1), rev(false) {}
    };

    std::vector<node_ptr> vertex;

    static void expose(node_ptr v) {
        node_ptr prev = nullptr;
        for (auto cur = v; cur; cur = cur->par) {
            splay(cur);
            cur->right = prev;
            recalc(cur);
            prev = cur;
        }
        splay(v);
    }

    // splay tree

    static int size(const node_ptr& t) {
        return t ? t->sz : 0;
    }

    static void recalc(const node_ptr& t) {
        if (!t) return;
        t->sz = size(t->left) + 1 + size(t->right);
        t->sum = t->val;
        if (t->left) t->sum = M::op(t->left->sum, t->sum);
        if (t->right) t->sum = M::op(t->sum, t->right->sum);
    }

    static void push(const node_ptr& t) {
        if (t->rev) {
            if (t->left) reverse(t->left);
            if (t->right) reverse(t->right);
            t->rev = false;
        }
    }

    static void reverse(const node_ptr& t) {
        std::swap(t->left, t->right);
        t->sum = flip(t->sum);
        t->rev ^= true;
    }

    static void rotate_left(node_ptr t) {
        node_ptr s = t->right;
        t->right = s->left;
        if (s->left) s->left->par = t;
        s->par = t->par;
        if (t->par) {
            if (t->par->left == t) {
                t->par->left = s;
            }
            if (t->par->right == t) {
                t->par->right = s;
            }
        }
```

```
        s->left = t;
        t->par = s;
        recalc(t);
        recalc(s);
    }

    static void rotate_right(node_ptr t) {
        node_ptr s = t->left;
        t->left = s->right;
        if (s->right) s->right->par = t;
        s->par = t->par;
        if (t->par) {
            if (t->par->left == t) {
                t->par->left = s;
            }
            if (t->par->right == t) {
                t->par->right = s;
            }
        }
        s->right = t;
        t->par = s;
        recalc(t);
        recalc(s);
    }

    static bool is_root(const node_ptr& t) {
        return !t->par || (t->par->left != t && t->par->right != t);
    }

    static void splay(node_ptr t) {
        push(t);
        while (!is_root(t)) {
            auto p = t->par;
            if (is_root(p)) {
                push(p);
                push(t);
                if (t == p->left) rotate_right(p);
                else rotate_left(p);
            } else {
                auto g = p->par;
                push(g);
                push(p);
                push(t);
                if (t == p->left) {
                    if (p == g->left) {
                        rotate_right(g);
                        rotate_right(p);
                    } else {
                        rotate_right(p);
                        rotate_left(g);
                    }
                } else {
                    if (p == g->left) {
                        rotate_left(p);
                        rotate_right(g);
                    } else {
                        rotate_left(g);
                        rotate_left(p);
                    }
                }
            }
        }
    }
};
```