目次

1 初期設定

1.1 テンプレート

```cpp
#pragma once
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
#define rep(i,s,t) for (int i = (int)(s); i < (int)(t); ++i)
#define all(x) (x).begin(), (x).end()

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);

}
```

1.2 コンパイルの alias

```
alias gxx='g++ -std=c++17 -Wall -Wextra -fsanitize=undefined -D_GLIBCXX_DEBUG'
```

2 蟻本

3 convolution

畳み込みライブラリ．演算の種類と用いる変換の対応は以下の通り．

- MAX conv: 累積和 (右)
- MIN conv: 累積和 (左)
- XOR conv: fwht
- OR conv: fzt(true)
- AND conv: fzt(false)
- GCD conv: divisor_fzt(false)
- LCM conv: divisor_fzt(true)
- $+$ conv: fft

計算量はだいたい $O(n \log n)$, 約数 FZT/FMT のみ $O(n \log \log n)$

3.1 Bitwise XOR Convolution

```cpp
template <typename T>
void fwht(std::vector<T>& a) {
    int n = a.size();
    for (int h = 1; h < n; h <<= 1) {
        for (int i = 0; i < n; i += h << 1) {
            for (int j = i; j < i + h; ++j) {
                T x = a[j];
                T y = a[j | h];
                a[j] = x + y;
                a[j | h] = x - y;
            }
        }
    }
}

template <typename T>
void ifwht(std::vector<T>& a) {
    int n = a.size();
    for (int h = 1; h < n; h <<= 1) {
        for (int i = 0; i < n; i += h << 1) {
            for (int j = i; j < i + h; ++j) {
                T x = a[j];
                T y = a[j | h];
                a[j] = (x + y) / 2;
                a[j | h] = (x - y) / 2;
            }
        }
    }
}
```

3.2 Bitwise AND/OR Convolution

```cpp
template <typename T>
void fzt(std::vector<T>& a, bool subset) {
    int k = 31 - __builtin_clz(a.size());
    for (int i = 0; i < k; ++i) {
        for (int j = 0; j < (1 << k); ++j) {
            if ((j >> i & 1) == subset) a[j] += a[j ^ (1 << i)];
        }
    }
}

template <typename T>
void fmt(std::vector<T>& a, bool subset) {
    int k = 31 - __builtin_clz(a.size());
```

```
    for (int i = 0; i < k; ++i) {
        for (int j = 0; j < (1 << k); ++j) {
            if ((j >> i & 1) == subset) a[j] -= a[j ^ (1 << i)];
        }
    }
}
```

3.3 GCD/LCM Convolution

```
template <typename T>
void divisor_fzt(std::vector<T>& a, bool subset) {
    int n = a.size();
    std::vector<bool> sieve(n, true);
    for (int p = 2; p < n; ++p) {
        if (!sieve[p]) continue;
        if (subset) {
            for (int k = 1; k * p < n; ++k) {
                sieve[k * p] = false;
                a[k * p] += a[k];
            }
        } else {
            for (int k = (n - 1) / p; k > 0; --k) {
                sieve[k * p] = false;
                a[k] += a[k * p];
            }
        }
    }
}

template <typename T>
void divisor_fmt(std::vector<T>& a, bool subset) {
    int n = a.size();
    std::vector<bool> sieve(n, true);
    for (int p = 2; p < n; ++p) {
        if (!sieve[p]) continue;
        if (subset) {
            for (int k = (n - 1) / p; k > 0; --k) {
                sieve[k * p] = false;
                a[k * p] -= a[k];
            }
        } else {
            for (int k = 1; k * p < n; ++k) {
                sieve[k * p] = false;
                a[k] -= a[k * p];
            }
        }
    }
}
```

3.4 Fast Fourier Transform

NTT では， omega を primitive_root.pow((mod - 1) / m) にする．mod と原子根の組は (167772161, 3), (469762049, 3), (754974721, 11), (998244353, 3), (1224736769, 3)

```
const double PI = acos(-1);

void fft(std::vector<std::complex<double>>& a) {
    int n = a.size();
    for (int m = n; m > 1; m >>= 1) {
        double ang = 2.0 * PI / m;
        std::complex<double> omega(cos(ang), sin(ang));
        for (int s = 0; s < n / m; ++s) {
            std::complex<double> w(1, 0);
            for (int i = 0; i < m / 2; ++i) {
                auto l = a[s * m + i];
                auto r = a[s * m + i + m / 2];
                a[s * m + i] = l + r;
                a[s * m + i + m / 2] = (l - r) * w;
                w *= omega;
            }
        }
    }
}

void ifft(std::vector<std::complex<double>>& a) {
    int n = a.size();
    for (int m = 2; m <= n; m <<= 1) {
        double ang = -2.0 * PI / m;
        std::complex<double> omega(cos(ang), sin(ang));
        for (int s = 0; s < n / m; ++s) {
            std::complex<double> w(1, 0);
            for (int i = 0; i < m / 2; ++i) {
                auto l = a[s * m + i];
                auto r = a[s * m + i + m / 2] * w;
                a[s * m + i] = l + r;
                a[s * m + i + m / 2] = l - r;
                w *= omega;
            }
        }
    }
}

template <typename T>
std::vector<double> convolution(const std::vector<T>& a, const std::vector<T>& b) {
    int size = a.size() + b.size() - 1;
    int n = 1;
    while (n < size) n <<= 1;
    std::vector<std::complex<double>> na(a.begin(), a.end()), nb(b.begin(), b.end());
    na.resize(n);
    nb.resize(n);
    fft(na, false);
    fft(nb, false);
    for (int i = 0; i < n; ++i) na[i] *= nb[i];
    ifft(na, false);
    std::vector<double> ret(size);
    for (int i = 0; i < size; ++i) ret[i] = na[i].real() / n;
    return ret;
}
```

4 data-structure

4.1 Segment Tree

- `void update(int k, T x)`
  - k 番目の要素を x に更新する
  - 時間計算量: $O(\log n)$
- `T fold(int l, int r)`

  - 区間 $[l, r)$ の値を fold する
  - 時間計算量: $O(\log n)$
- `int find\_first(int l, F cond)`
  - `fold(l, r)` が条件 `cond` を満たすような最小の $r(> l)$ 返す．列の単調性を仮定する．そのような r が存在しない場合は `-1` を返す
  - 時間計算量: $O(\log n)$
- `int find\_last(int r, F cond)`
  - `fold(l, r)` が条件 `cond` を満たすような最大の $l(< r)$ 返す．列の単調性を仮定する．そのような l が存在しない場合は `-1` を返す
  - 時間計算量: $O(\log n)$

モノイドの渡し方

```
struct Monoid {
    using T = int;  // 型
    static T id() { return 1e9; };  // 単位元
    static T op(T a, T b) { return min(a, b); }  // 二項演算（結合的）
};
```

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

public:
    SegmentTree() = default;
    explicit SegmentTree(int n): SegmentTree(std::vector<T>(n, M::id())) {}
    explicit SegmentTree(const std::vector<T>& v) {
        size = 1;
        while (size < (int) v.size()) size <<= 1;
        node.resize(2 * size, M::id());
        std::copy(v.begin(), v.end(), node.begin() + size);
        for (int i = size - 1; i > 0; --i) node[i] = M::op(node[2 * i], node[2 * i + 1]);
    }

    T operator[](int k) const {
        return node[k + size];
    }

    void update(int k, const T& x) {
        k += size;
        node[k] = x;
        while (k >>= 1) node[k] = M::op(node[2 * k], node[2 * k + 1]);
    }

    T fold(int l, int r) const {
        T vl = M::id(), vr = M::id();
        for (l += size, r += size; l < r; l >>= 1, r >>= 1) {
            if (l & 1) vl = M::op(vl, node[l++]);
            if (r & 1) vr = M::op(node[--r], vr);
        }
        return M::op(vl, vr);
    }

    template <typename F>
    int find_first(int l, F cond) const {
        T vl = M::id();
        int r = size;
        for (l += size, r += size; l < r; l >>= 1, r >>= 1) {
            if (l & 1) {
                T nxt = M::op(vl, node[l]);
                if (cond(nxt)) {
                    while (l < size) {
                        nxt = M::op(vl, node[2 * l]);
                        if (cond(nxt)) l = 2 * l;
                        else vl = nxt, l = 2 * l + 1;
                    }
                    return l - size;
                }
                vl = nxt;
                ++l;
            }
        }
        return -1;
    }

    template <typename F>
    int find_last(int r, F cond) const {
        T vr = M::id();
        int l = 0;
        for (l += size, r += size; l < r; l >>= 1, r >>= 1) {
            if (r & 1) {
                --r;
                T nxt = M::op(node[r], vr);
                if (cond(nxt)) {
                    while (r < size) {
                        nxt = M::op(node[2 * r + 1], vr);
                        if (cond(nxt)) r = 2 * r + 1;
                        else vr = nxt, r = 2 * r;
                    }
                    return r - size;
                }
                vr = nxt;
            }
        }
        return -1;
    }

private:
    int size;
    std::vector<T> node;
};
```

4.2 Segment Tree with Lazy Propagation

モノイドと作用の渡し方

```
// 作用されるモノイド
struct M {
    using T = pair<ll, int>;  // 型
    static T id() { return {0, 1}; }  // 単位元
    static T op(T a, T b) {
        return {a.first + b.first, a.second + b.second};  // 二項演算（結合的）
    }
};

// 作用するモノイド
struct O {
```

```
    using T = ll;  // 型
    static T id() { return 0; }  // 単位元
    static T op(T a, T b) {
        return a + b;  // 二項演算（結合的）
    }
};

// 作用（分配的）
M::T act(M::T a, O::T b) {
    return {a.first + a.second * b, a.second};
}
```

```
template <typename M, typename O, typename M::T (*act)(typename M::T, typename O::T)>
class LazySegmentTree {
    using T = typename M::T;
    using E = typename O::T;

public:
    LazySegmentTree() = default;
    explicit LazySegmentTree(int n) : LazySegmentTree(std::vector<T>(n, M::id())) {}
    explicit LazySegmentTree(const std::vector<T>& v) {
        size = 1;
        while (size < (int) v.size()) size <<= 1;
        node.resize(2 * size, M::id());
        lazy.resize(2 * size, O::id());
        std::copy(v.begin(), v.end(), node.begin() + size);
        for (int i = size - 1; i > 0; --i) node[i] = M::op(node[2 * i], node[2 * i + 1]);
    }

    T operator[](int k) {
        return fold(k, k + 1);
    }

    void update(int l, int r, const E& x) { update(l, r, x, 1, 0, size); }

    T fold(int l, int r) { return fold(l, r, 1, 0, size); }

    template <typename F>
    int find_first(int l, F cond) {
        T v = M::id();
        return find_first(l, size, 1, 0, size, v, cond);
    }

    template <typename F>
    int find_last(int r, F cond) {
        T v = M::id();
        return find_last(0, r, 1, 0, size, v, cond);
    }

private:
    int size;
    std::vector<T> node;
    std::vector<E> lazy;

    void push(int k) {
        if (lazy[k] == O::id()) return;
        if (k < size) {
            lazy[2 * k] = O::op(lazy[2 * k], lazy[k]);
            lazy[2 * k + 1] = O::op(lazy[2 * k + 1], lazy[k]);
        }
        node[k] = act(node[k], lazy[k]);
        lazy[k] = O::id();
    }

    void update(int a, int b, const E& x, int k, int l, int r) {
        push(k);
        if (r <= a || b <= l) return;
        if (a <= l && r <= b) {
            lazy[k] = O::op(lazy[k], x);
            push(k);
            return;
        }
        int m = (l + r) / 2;
        update(a, b, x, 2 * k, l, m);
        update(a, b, x, 2 * k + 1, m, r);
        node[k] = M::op(node[2 * k], node[2 * k + 1]);
    }

    T fold(int a, int b, int k, int l, int r) {
        push(k);
        if (r <= a || b <= l) return M::id();
        if (a <= l && r <= b) return node[k];
        int m = (l + r) / 2;
        return M::op(fold(a, b, 2 * k, l, m),
                     fold(a, b, 2 * k + 1, m, r));
    }

    template <typename F>
    int find_first(int a, int b, int k, int l, int r, T& v, F cond) {
        push(k);
        if (r <= a) return -1;
        if (b <= l) return l;
        if (a <= l && r <= b && !cond(M::op(v, node[k]))) {
            v = M::op(v, node[k]);
            return -1;
        }
        if (r - l == 1) return r;
        int m = (l + r) / 2;
        int res = find_first(a, b, 2 * k, l, m, v, cond);
        if (res != -1) return res;
        return find_first(a, b, 2 * k + 1, m, r, v, cond);
    }

    template <typename F>
    int find_last(int a, int b, int k, int l, int r, T& v, F cond) {
        push(k);
        if (b <= l) return -1;
        if (r <= a) return r;
        if (a <= l && r <= b && !cond(M::op(node[k], v))) {
            v = M::op(node[k], v);
            return -1;
        }
        if (r - l == 1) return l;
        int m = (l + r) / 2;
        int res = find_last(a, b, 2 * k + 1, m, r, v, cond);
        if (res != -1) return res;
        return find_last(a, b, 2 * k, l, m, v, cond);
    }
};
```

4.3 Fenwick Tree

- `T prefix\_fold(int i)`
  - 区間 $[0, i)$ の値を fold する
  - 時間計算量: $O(\log n)$
- `void update(int i, T x)`
  - i 番目の要素を x と演算した値に更新する
  - 時間計算量: $O(\log n)$
- `int lower\_bound(T x)`
  - `prefix\_fold(i) \geq x` となる最初の i を返す．そのような i が存在しない場合は n を返す．`cmp` を指定しない場合は `<` で比較される．列の単調性を仮定する．
  - 時間計算量: $O(\log n)$

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

public:
    FenwickTree() = default;
    explicit FenwickTree(int n) : n(n), data(n + 1, M::id()) {}

    T prefix_fold(int i) const {
        T ret = M::id();
        for (; i > 0; i -= i & -i) ret = M::op(ret, data[i]);
        return ret;
    }

    void update(int i, const T& x) {
        for (++i; i <= n; i += i & -i) data[i] = M::op(data[i], x);
    }

    int lower_bound(const T& x) const {
        return lower_bound(x, std::less<>());
    }

    int lower_bound(const T& x) const {
        if (x <= M::id()) return 0;
        int k = 1;
        while (k * 2 <= n) k <<= 1;
        int i = 0;
        T v = M::id();
        for (; k > 0; k >>= 1) {
            if (i + k > n) continue;
            T nv = M::op(v, data[i + k]);
            if (nv < x) {
                v = nv;
                i += k;
            }
        }
        return i;
    }

private:
    int n;
    std::vector<T> data;
};
```

4.4 Range Tree

- `RangeTree(vector<tuple<int, Y, T>>> pts)`
  - `pts` の点から領域木を構築する．点は (x, y, v) の形式で与えられる．
  - 時間計算量: $O(n \log n)$
- `T fold(X sx, X tx, Y sy, Y ty)`
  - 長方形領域 $[sx, tx) \times [sy, ty)$ 内の点に対する値の総和を計算する
  - 時間計算量: $O((\log n)^2)$

```cpp
template <typename X, typename Y, typename T>
class RangeTree {
public:
    RangeTree() = default;
    explicit RangeTree(const std::vector<std::tuple<X, Y, T>>& pts) {
        for (auto& [x, y, v] : pts) xs.push_back(x);
        std::sort(xs.begin(), xs.end());
        xs.erase(std::unique(xs.begin(), xs.end()), xs.end());

        int n = xs.size();
        size = 1;
        while (size < n) size <<= 1;
        node.resize(2 * size);
        sum.resize(2 * size);

        for (auto& [x, y, v] : pts) {
            int i = std::lower_bound(xs.begin(), xs.end(), x) - xs.begin();
            node[size + i].emplace_back(y, v);
        }

        for (int i = 0; i < n; ++i) {
            std::sort(node[size + i].begin(), node[size + i].end());
        }
        for (int i = size - 1; i > 0; --i) {
            std::merge(node[2*i].begin(), node[2*i].end(), node[2*i+1].begin(), node[2*i
                +1].end(), std::back_inserter(node[i]));
        }
        for (int i = 0; i < size + n; ++i) {
            sum[i].resize(node[i].size() + 1);
            for (int j = 0; j < (int) node[i].size(); ++j) {
                sum[i][j + 1] = sum[i][j] + node[i][j].second;
            }
        }
    }

    T fold(X sx, X tx, Y sy, Y ty) const {
        int l = std::lower_bound(xs.begin(), xs.end(), sx) - xs.begin();
        int r = std::lower_bound(xs.begin(), xs.end(), tx) - xs.begin();
        T ret = 0;
        auto cmp = [&](const std::pair<Y, T>& p, Y y) { return p.first < y; };
        for (l += size, r += size; l < r; l >>= 1, r >>= 1) {
            if (l & 1) {
                int hi = std::lower_bound(node[l].begin(), node[l].end(), ty, cmp) - node[
                    l].begin();
                int lo = std::lower_bound(node[l].begin(), node[l].end(), sy, cmp) - node[
                    l].begin();
                ret += sum[l][hi] - sum[l][lo];
                ++l;
            }
            if (r & 1) {
```

```
                --r;
                int hi = std::lower_bound(node[r].begin(), node[r].end(), ty, cmp) - node[r].begin();
                int lo = std::lower_bound(node[r].begin(), node[r].end(), sy, cmp) - node[r].begin();
                ret += sum[r][hi] - sum[r][lo];
            }
        }
        return ret;
    }


private:
    int size;
    std::vector<X> xs;
    std::vector<std::vector<std::pair<Y, T>>> node;
    std::vector<std::vector<T>> sum;
};
```

4.5 Union Find

```
class UnionFind {
public:
    UnionFind() = default;
    explicit UnionFind(int n) : data(n, -1) {}

    int find(int x) {
        if (data[x] < 0) return x;
        return data[x] = find(data[x]);
    }

    void unite(int x, int y) {
        x = find(x);
        y = find(y);
        if (x == y) return;
        if (data[x] > data[y]) std::swap(x, y);
        data[x] += data[y];
        data[y] = x;
    }

    bool same(int x, int y) {
        return find(x) == find(y);
    }

    int size(int x) {
        return -data[find(x)];
    }

private:
    std::vector<int> data;
};
```

4.6 Weighted Union Find

- `int find(int x)`
  - x が属する木の根を返す
  - 時間計算量: amortized $O(\alpha(n))$
- `T weight(int x)`
  - 木の根に対する x の重みを返す
  - 時間計算量: amortized $O(\alpha(n))$
- `void unite(int x, int y, T w)`
  - x が属する集合と y が属する集合を $weight(y) - weight(x) = w$ となるように連結する
  - 時間計算量: amortized $O(\alpha(n))$
- `bool same(int x, int y)`
  - x と y が同じ集合に属するかを判定する
  - 時間計算量: amortized $O(\alpha(n))$
- `T diff(int x, int y)`
  - x に対する y の重み，すなわち $weight(y) - weight(x)$ を返す
  - 時間計算量: amortized $O(\alpha(n))$
- `int size(int x)`
  - x が属する集合の大きさを返す
  - 時間計算量: amortized $O(\alpha(n))$

```
template <typename T>
class WeightedUnionFind {
public:
    WeightedUnionFind() = default;
    explicit WeightedUnionFind(int n) : data(n, -1), ws(n) {}

    int find(int x) {
        if (data[x] < 0) return x;
        int r = find(data[x]);
        ws[x] += ws[data[x]];
        return data[x] = r;
    }

    T weight(int x) {
        find(x);
        return ws[x];
    }

    bool unite(int x, int y, T w) {
        w += weight(x);
        w -= weight(y);
        x = find(x);
        y = find(y);
        if (x == y) return false;
        if (data[x] > data[y]) {
            std::swap(x, y);
            w = -w;
        }
        data[x] += data[y];
        data[y] = x;
        ws[y] = w;
        return true;
    }

    bool same(int x, int y) {
        return find(x) == find(y);
    }

    T diff(int x, int y) {
        return weight(y) - weight(x);
```

```
    }

    int size(int x) {
        return -data[find(x)];
    }

private:
    std::vector<int> data;
    std::vector<T> ws;
};
```

4.7 Convex Hull Trick

1 次関数の最小値．傾きは単調非増加

- `ConvexHullTrick(bool monotone\_query)`
  - 最小値クエリの x が単調非減少ならば，`monotone\_query = true` とすれば計算量が改善する．デフォルトは `monotone\_query = false`
  - 時間計算量: $O(1)$
- `T add(T a, T b)`
  - 直線 $ax+b$ を L に追加する
  - 時間計算量: amortized $O(1)$
- `T get(T x)`
  - 与えられた x に対し，L の中で最小値を取る直線の値を求める
  - 時間計算量: `monotone\_query = true` なら amortized $O(1)$, `false` なら $O(\log n)$

```
template <typename T>
class ConvexHullTrick {
public:
    explicit ConvexHullTrick(bool monotone_query = false) : monotone_query(monotone_query)
        {}

    void add(T a, T b) {
        Line line(a, b);
        while (lines.size() >= 2 && check(*(lines.end() - 2), lines.back(), line)) {
            lines.pop_back();
        }
        lines.push_back(line);
    }

    T get(T x) {
        if (monotone_query) {
            while (lines.size() - head >= 2 && lines[head](x) > lines[head + 1](x)) {
                ++head;
            }
            return lines[head](x);
        } else {
            int lb = -1, ub = lines.size() - 1;
            while (ub - lb > 1) {
                int m = (lb + ub) / 2;
                if (lines[m](x) > lines[m + 1](x)) {
                    lb = m;
                } else {
                    ub = m;
                }
            }
            return lines[ub](x);
        }
    }

private:
    struct Line {
        T a, b;
        Line(T a, T b) : a(a), b(b) {}
        T operator()(T x) const { return a * x + b; }
    };

    std::vector<Line> lines;
    bool monotone_query;
    int head = 0;

    static bool check(Line l1, Line l2, Line l3) {
        if (l2.a == l3.a) return l2.b >= l3.b;
        return 1.0 * (l2.b - l1.b) / (l2.a - l1.a) <= 1.0 * (l3.b - l2.b) / (l3.a - l2.a);
    }
};
```

5 flow

5.1 Ford-Fulkerson Algorithm

時間計算量: $O(Ef)$

```
template <typename T>
class FordFulkerson {
public:
    FordFulkerson() = default;
    explicit FordFulkerson(int n) : G(n), used(n) {}

    void add_edge(int u, int v, T cap) {
        G[u].push_back({v, (int) G[v].size(), cap});
        G[v].push_back({u, (int) G[u].size() - 1, 0});
    }

    T max_flow(int s, int t) {
        T flow = 0;
        while (true) {
            std::fill(used.begin(), used.end(), false);
            T f = dfs(s, t, INF);
            if (f == 0) return flow;
            flow += f;
        }
    }

    std::set<int> min_cut(int s) {
        std::stack<int> st;
        std::set<int> visited;
        st.push(s);
        visited.insert(s);
        while (!st.empty()) {
            int v = st.top();
            st.pop();
            for (auto& e : G[v]) {
                if (e.cap > 0 && !visited.count(e.to)) {
                    visited.insert(e.to);
                    st.push(e.to);
```

```
                }
            }
        }
        return visited;
    }

private:
    struct Edge {
        int to, rev;
        T cap;
    };

    const T INF = std::numeric_limits<T>::max() / 2;

    std::vector<std::vector<Edge>> G;
    std::vector<bool> used;

    T dfs(int v, int t, T f) {
        if (v == t) return f;
        used[v] = true;
        for (auto& e : G[v]) {
            if (!used[e.to] && e.cap > 0) {
                T d = dfs(e.to, t, std::min(f, e.cap));
                if (d > 0) {
                    e.cap -= d;
                    G[e.to][e.rev].cap += d;
                    return d;
                }
            }
        }
        return 0;
    }
};
```

5.2 Dinic's Algorithm

時間計算量: $O(V^2E)$

```
template <typename T>
class Dinic {
public:
    Dinic() = default;
    explicit Dinic(int V) : G(V), level(V), iter(V) {}

    void add_edge(int u, int v, T cap) {
        G[u].push_back({v, (int) G[v].size(), cap});
        G[v].push_back({u, (int) G[u].size() - 1, 0});
    }

    T max_flow(int s, int t) {
        T flow = 0;
        while (bfs(s, t)) {
            std::fill(iter.begin(), iter.end(), 0);
            T f = 0;
            while ((f = dfs(s, t, INF)) > 0) flow += f;
        }
        return flow;
    }

    std::set<int> min_cut(int s) {
        std::stack<int> st;
        std::set<int> visited;
        st.push(s);
        visited.insert(s);
        while (!st.empty()) {
            int v = st.top();
            st.pop();
            for (auto& e : G[v]) {
                if (e.cap > 0 && !visited.count(e.to)) {
                    visited.insert(e.to);
                    st.push(e.to);
                }
            }
        }
        return visited;
    }

private:
    struct Edge {
        int to, rev;
        T cap;
    };

    static constexpr T INF = std::numeric_limits<T>::max() / 2;

    std::vector<std::vector<Edge>> G;
    std::vector<int> level, iter;

    bool bfs(int s, int t) {
        std::fill(level.begin(), level.end(), -1);
        level[s] = 0;
        std::queue<int> q;
        q.push(s);
        while (!q.empty() && level[t] == -1) {
            int v = q.front();
            q.pop();
            for (auto& e : G[v]) {
                if (e.cap > 0 && level[e.to] == -1) {
                    level[e.to] = level[v] + 1;
                    q.push(e.to);
                }
            }
        }
        return level[t] != -1;
    }

    T dfs(int v, int t, T f) {
        if (v == t) return f;
        for (int& i = iter[v]; i < (int) G[v].size(); ++i) {
            Edge& e = G[v][i];
            if (e.cap > 0 && level[v] < level[e.to]) {
                T d = dfs(e.to, t, std::min(f, e.cap));
                if (d > 0) {
                    e.cap -= d;
                    G[e.to][e.rev].cap += d;
                    return d;
                }
            }
        }
        return 0;
```

```
    }
};
```

5.3 Minimum Cost Flow

- `void add_edge(int u, int v, Cap cap, Cost cost)`
  - 容量 cap，コスト $cost$ の辺 (u, v) を追加する
  - 時間計算量: $O(1)$
- `void add_edge(int u, int v, Cap lb, Cap ub, Cost cost)`
  - 最小流量 lb, 容量 ub，コスト $cost$ の辺 (u, v) を追加する
  - 時間計算量: $O(1)$
- `Cost min_cost_flow(int s, int t, Cap f, bool arbitrary)`
  - 始点 s から終点 t への流量 f の最小費用流を求める．`arbitrary == true` の場合，流量は f 以下の任意の値とする．
  - 時間計算量: $O(fE \log V)$

Note

このライブラリがそのまま使える場合は，すべての辺のコストが非負である普通の最小費用流のとき．以下，いろいろな状況での使い方を説明する．

- 負辺がある場合
  - ポテンシャルの初期値の計算に，負辺があっても動作する最短路アルゴリズムを用いる必要がある．Bellman-Ford algorithm を用いることができる．また，グラフが DAG である場合は，トポロジカルソートして DP することができる．`calculate_initial_potential()` という private メソッドを用意しているのでその中を自分で書き換える．
  - 蟻本に載っているテク（超頂点を作って頑張る）を用いて負辺を除去することもできる．
- 負閉路がある場合
  - Bellman-Ford algorithm で負閉路を見つけてそこに流せるだけ流しておけば良い．書いたことがないのでピンときていない．
- 流量が任意の場合
  - 負辺がある場合に任意の流量を流して最小費用を求めたい場合がある．これは s-t 最短路が負である限り流せば良い．ここの処理がコメントアウトしてあるので，適宜外す．引数の `f` には適当に大きな値を設定しておけば良い．

```
template <typename Cap, typename Cost>
class MinCostFlow {
public:
    MinCostFlow() = default;
    explicit MinCostFlow(int V) : V(V), G(V), add(0) {}

    void add_edge(int u, int v, Cap cap, Cost cost) {
        G[u].emplace_back(v, cap, cost, (int) G[v].size());
        G[v].emplace_back(u, 0, -cost, (int) G[u].size() - 1);
    }

    void add_edge(int u, int v, Cap lb, Cap ub, Cost cost) {
        add_edge(u, v, ub - lb, cost);
        add_edge(u, v, lb, cost - M);
        add += M * lb;
    }

    Cost min_cost_flow(int s, int t, Cap f, bool arbitrary = false) {
        Cost ret = add;
        std::vector<Cost> dist(V);
        std::vector<int> prevv(V), preve(V);
        using P = std::pair<Cost, int>;
        std::priority_queue<P, std::vector<P>, std::greater<P>> pq;

        auto h = calculate_initial_potential(s);

        while (f > 0) {
            // update h using dijkstra
            std::fill(dist.begin(), dist.end(), INF);
            dist[s] = 0;
            pq.emplace(0, s);
            while (!pq.empty()) {
                Cost d;
                int v;
                std::tie(d, v) = pq.top();
                pq.pop();
                if (dist[v] < d) continue;
                for (int i = 0; i < (int) G[v].size(); ++i) {
                    Edge& e = G[v][i];
                    Cost ndist = dist[v] + e.cost + h[v] - h[e.to];
                    if (e.cap > 0 && dist[e.to] > ndist) {
                        dist[e.to] = ndist;
                        prevv[e.to] = v;
                        preve[e.to] = i;
                        pq.emplace(dist[e.to], e.to);
                    }
                }
            }

            if (!arbitrary && dist[t] == INF) return -1;
            for (int v = 0; v < V; ++v) h[v] += dist[v];

            if (arbitrary && h[t] >= 0) break;

            Cap d = f;
            for (int v = t; v != s; v = prevv[v]) {
                d = std::min(d, G[prevv[v]][preve[v]].cap);
            }
            f -= d;
            ret += d * h[t];
            for (int v = t; v != s; v = prevv[v]) {
                Edge& e = G[prevv[v]][preve[v]];
                e.cap -= d;
                G[v][e.rev].cap += d;
            }
        }
        return ret;
    }

private:
    struct Edge {
        int to;
        Cap cap;
        Cost cost;
        int rev;
        Edge(int to, Cap cap, Cost cost, int rev) : to(to), cap(cap), cost(cost), rev(rev)
            {}
```

```
    };

    static constexpr Cost INF = std::numeric_limits<Cost>::max() / 2;
    static constexpr Cost M = INF / 1e9;  // large constant used for minimum flow requirement for edges

    int V;
    std::vector<std::vector<Edge>> G;
    Cost add;


    std::vector<Cost> calculate_initial_potential(int s) {
        std::vector<Cost> h(V);
        // if all costs are nonnegative, then do nothing
        return h;

        // if there is a negative edge,
        // use Bellman-Ford or topological sort and a DP (for DAG)
        // std::fill(h.begin(), h.end(), INF);
        // h[s] = 0;
        // for (int i = 0; i < V - 1; ++i) {
        //     for (int v = 0; v < V; ++v) {
        //         for (auto& e : G[v]) {
        //             if (e.cap > 0 && h[v] != INF && h[e.to] > h[v] + e.cost) {
        //                 h[e.to] = h[v] + e.cost;
        //             }
        //         }
        //     }
        // }

        // return h;
    }
};
```

6 geometry

6.1 Geometry

Vec は std::complex<T> のエイリアスである.

- geometry.hpp
  - 基本関数群
  - T dot(Vec a, Vec b)
    - 内積を計算する
  - T cross(Vec a, Vec b)
    - 外積の z 座標を計算する
  - Vec rot(Vec a, T ang)
    - a を角 ang だけ回転させる
  - Vec perp(Vec a)
    - a を角 $\pi/2$ だけ回転させる
  - Vec projection(Line l, Vec p)
    - 点 p の直線 l 上の射影を求める
  - Vec reflection(Line l, Vec p)
    - 点 p の直線 l に関して対称な点を求める
  - int ccw(Vec a, Vec b, Vec c)
    - a, b, c が同一直線上にあるなら 0, $a \to b \to c$ が反時計回りなら 1，そうでなければ-1 を返す
  - void sort_by_arg(vector<Vec> pts)
    - 与えられた点を偏角ソートする (ソート順はこの問題[*1]に準拠)
    - 時間計算量: $O(n \log n)$
- intersect.hpp
  - 交差判定
- dist.hpp
  - 距離
- intersection.hpp
  - 交点
- bisector.hpp
  - 二等分線
- triangle.hpp
  - 重心，内心，外心
- tangent.hpp
  - 円の接線
- polygon.hpp
  - 多角形周り．計算量は明示しない限り $O(n)$
  - T area(Polygon poly)
    - 多角形 $poly$ の面積を求める
  - T is_convex(Polygon poly)
    - 多角形 $poly$ が凸か判定する．poly は反時計回りに与えられる必要がある
  - Polygon convex_cut(Polygon poly, Line l)
    - 多角形 $poly$ を直線 l で切断する．詳細な仕様は 凸多角形の切断[*2] を参照.
  - Polygon halfplane_intersection(vector<pair<Vec, Vec>> hps)
    - 半平面の集合が与えられたとき，それらの共通部分 (凸多角形になる) を返す．半平面は，$\boldsymbol{x} \mid (\boldsymbol{x} - \boldsymbol{p}) \cdot \boldsymbol{n} \geq 0$ で表したときに $(\boldsymbol{n}, \boldsymbol{p})$ の形で与える.
    - 時間計算量: $O(n \log n)$

```
using T = double;
using Vec = std::complex<T>;

const T PI = std::acos(-1);

constexpr T eps = 1e-10;
inline bool eq(T a, T b) { return std::abs(a - b) <= eps; }
inline bool eq(Vec a, Vec b) { return std::abs(a - b) <= eps; }
inline bool lt(T a, T b) { return a < b - eps; }
inline bool leq(T a, T b) { return a <= b + eps; }

```

[*1] <https://judge.yosupo.jp/problem/sort_points_by_argument>

[*2] <https://onlinejudge.u-aizu.ac.jp/courses/library/4/CGL/4/CGL_4_C>

```cpp
std::istream& operator>>(std::istream& is, Vec& p) {
    T x, y;
    is >> x >> y;
    p = {x, y};
    return is;
}

struct Line {
    Vec p1, p2;
    Line() = default;
    Line(const Vec& p1, const Vec& p2) : p1(p1), p2(p2) {}
    Vec dir() const { return p2 - p1; }
};

struct Segment : Line {
    using Line::Line;
};

struct Circle {
    Vec c;
    T r;
    Circle() = default;
    Circle(const Vec& c, T r) : c(c), r(r) {}
};

using Polygon = std::vector<Vec>;

T dot(const Vec& a, const Vec& b) {
    return (std::conj(a) * b).real();
}

T cross(const Vec& a, const Vec& b) {
    return (std::conj(a) * b).imag();
}

Vec rot(const Vec& a, T ang) {
    return a * Vec(std::cos(ang), std::sin(ang));
}

Vec perp(const Vec& a) {
    return Vec(-a.imag(), a.real());
}

Vec projection(const Line& l, const Vec& p) {
    return l.p1 + dot(p - l.p1, l.dir()) * l.dir() / std::norm(l.dir());
}

Vec reflection(const Line& l, const Vec& p) {
    return T(2) * projection(l, p) - p;
}

// 0: collinear
// 1: counter-clockwise
// -1: clockwise
int ccw(const Vec& a, const Vec& b, const Vec& c) {
    if (eq(cross(b - a, c - a), 0)) return 0;
    if (lt(cross(b - a, c - a), 0)) return -1;
    return 1;
}

void sort_by_arg(std::vector<Vec>& pts) {
    std::sort(pts.begin(), pts.end(), [&](auto& p, auto& q) {
        if ((p.imag() < 0) != (q.imag() < 0)) return (p.imag() < 0);
        if (cross(p, q) == 0) {
            if (p == Vec(0, 0)) return !(q.imag() < 0 || (q.imag() == 0 && q.real() > 0));
            if (q == Vec(0, 0)) return  (p.imag() < 0 || (p.imag() == 0 && p.real() > 0));
            return (p.real() > q.real());
        }
        return (cross(p, q) > 0);
    });
}
```

6.2 intersection.hpp

```cpp
#include "geometry.hpp"
#include "dist.hpp"

Vec intersection(const Line& l, const Line& m) {
    Vec r = m.p1 - l.p1;
    assert(!eq(cross(l.dir(), m.dir()), 0)); // not parallel
    return l.p1 + cross(m.dir(), r) / cross(m.dir(), l.dir()) * l.dir();
}

std::vector<Vec> intersection(const Circle& c, const Line& l) {
    T d = dist(l, c.c);
    if (lt(c.r, d)) return {};  // no intersection
    Vec e1 = l.dir() / std::abs(l.dir());
    Vec e2 = perp(e1);
    if (ccw(c.c, l.p1, l.p2) == 1) e2 *= -1;
    if (eq(c.r, d)) return {c.c + d*e2};  // tangent
    T t = std::sqrt(c.r*c.r - d*d);
    return {c.c + d*e2 + t*e1, c.c + d*e2 - t*e1};
}

std::vector<Vec> intersection(const Circle& c1, const Circle& c2) {
    T d = std::abs(c1.c - c2.c);
    if (lt(c1.r + c2.r, d)) return {};  // outside
    Vec e1 = (c2.c - c1.c) / std::abs(c2.c - c1.c);
    Vec e2 = perp(e1);
    if (lt(d, std::abs(c2.r - c1.r))) return {};  // contain
    if (eq(d, std::abs(c2.r - c1.r))) return {c1.c + c1.r*e1};  // tangent
    T x = (c1.r*c1.r - c2.r*c2.r + d*d) / (2*d);
    T y = std::sqrt(c1.r*c1.r - x*x);
    return {c1.c + x*e1 + y*e2, c1.c + x*e1 - y*e2};
}

T area_intersection(const Circle& c1, const Circle& c2) {
    T d = std::abs(c2.c - c1.c);
    if (leq(c1.r + c2.r, d)) return 0;  // outside
    if (leq(d, std::abs(c2.r - c1.r))) {  // inside
        T r = std::min(c1.r, c2.r);
        return PI * r * r;
    }
    T ans = 0;
    T a;
    a = std::acos((c1.r*c1.r+d*d-c2.r*c2.r)/(2*c1.r*d));
    ans += c1.r*c1.r*(a - std::sin(a)*std::cos(a));
    a = std::acos((c2.r*c2.r+d*d-c1.r*c1.r)/(2*c2.r*d));
    ans += c2.r*c2.r*(a - std::sin(a)*std::cos(a));
    return ans;
```

```
}
```

6.3 dist.hpp

```
#include "geometry.hpp"
#include "intersect.hpp"

T dist(const Line& l, const Vec& p) {
    return std::abs(cross(p - l.p1, l.dir())) / std::abs(l.dir());
}

T dist(const Segment& s, const Vec& p) {
    if (lt(dot(p - s.p1, s.dir()), 0)) return std::abs(p - s.p1);
    if (lt(dot(p - s.p2, -s.dir()), 0)) return std::abs(p - s.p2);
    return std::abs(cross(p - s.p1, s.dir())) / std::abs(s.dir());
}

T dist(const Segment& s, const Segment& t) {
    if (intersect(s, t)) return T(0);
    return std::min({dist(s, t.p1), dist(s, t.p2), dist(t, s.p1), dist(t, s.p2)});
}
```

6.4 intersect.hpp

```
#include "geometry.hpp"

bool intersect(const Segment& s, const Vec& p) {
    Vec u = s.p1 - p, v = s.p2 - p;
    return eq(cross(u, v), 0) && leq(dot(u, v), 0);
}

// 0: outside
// 1: on the border
// 2: inside
int intersect(const Polygon& poly, const Vec& p) {
    const int n = poly.size();
    bool in = 0;
    for (int i = 0; i < n; ++i) {
        auto a = poly[i] - p, b = poly[(i+1)%n] - p;
        if (eq(cross(a, b), 0) && (lt(dot(a, b), 0) || eq(dot(a, b), 0))) return 1;
        if (a.imag() > b.imag()) std::swap(a, b);
        if (leq(a.imag(), 0) && lt(0, b.imag()) && lt(cross(a, b), 0)) in ^= 1;
    }
    return in ? 2 : 0;
}

int intersect(const Segment& s, const Segment& t) {
    auto a = s.p1, b = s.p2;
    auto c = t.p1, d = t.p2;
    if (ccw(a, b, c) != ccw(a, b, d) && ccw(c, d, a) != ccw(c, d, b)) return 2;
    if (intersect(s, c) || intersect(s, d) || intersect(t, a) || intersect(t, b)) return
        1;
    return 0;
}

// true if they have positive area in common or touch on the border
bool intersect(const Polygon& poly1, const Polygon& poly2) {
    const int n = poly1.size();
    const int m = poly2.size();
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            if (intersect(Segment(poly1[i], poly1[(i+1)%n]), Segment(poly2[j], poly2[(j
                +1)%m]))) {
                return true;
            }
        }
    }
    return intersect(poly1, poly2[0]) || intersect(poly2, poly1[0]);
}

// 0: inside
// 1: inscribe
// 2: intersect
// 3: circumscribe
// 4: outside
int intersect(const Circle& c1, const Circle& c2) {
    T d = std::abs(c1.c - c2.c);
    if (lt(d, std::abs(c2.r - c1.r))) return 0;
    if (eq(d, std::abs(c2.r - c1.r))) return 1;
    if (eq(c1.r + c2.r, d)) return 3;
    if (lt(c1.r + c2.r, d)) return 4;
    return 2;
}
```

6.5 tangent.hpp

```
#include "geometry.hpp"
#include "intersect.hpp"
#include "intersection.hpp"

std::pair<Vec, Vec> tangent_points(const Circle& c, const Vec& p) {
    auto m = (p + c.c) / T(2);
    auto is = intersection(c, Circle(m, std::abs(p - m)));
    return {is[0], is[1]};
}

// for each l, l.p1 is a tangent point of c1
std::vector<Line> common_tangents(Circle c1, Circle c2) {
    assert(!eq(c1.c, c2.c) || !eq(c1.r, c2.r));
    int cnt = intersect(c1, c2);  // number of common tangents
    std::vector<Line> ret;
    if (cnt == 0) {
        return ret;
    }

    // external
    if (eq(c1.r, c2.r)) {
        auto d = c2.c - c1.c;
        Vec e(-d.imag(), d.real());
        e = e / std::abs(e) * c1.r;
        ret.push_back(Line(c1.c + e, c1.c + e + d));
        ret.push_back(Line(c1.c - e, c1.c - e + d));
    } else {
        auto p = (-c2.r*c1.c + c1.r*c2.c) / (c1.r - c2.r);
        if (cnt == 1) {
            Vec q(-p.imag(), p.real());
            return {Line(p, q)};
        } else {
            auto [a, b] = tangent_points(c1, p);
            ret.push_back(Line(a, p));
```

```
            ret.push_back(Line(b, p));
        }
    }

    // internal
    auto p = (c2.r*c1.c + c1.r*c2.c) / (c1.r + c2.r);
    if (cnt == 3) {
        Vec q(-p.imag(), p.real());
        ret.push_back(Line(p, q));
    } else if (cnt == 4) {
        auto [a, b] = tangent_points(c1, p);
        ret.push_back(Line(a, p));
        ret.push_back(Line(b, p));
    }

    return ret;
}
```

6.6 polygon.hpp

```
#pragma once
#include <algorithm>
#include <deque>
#include <utility>
#include <vector>
#include "geometry.hpp"
#include "intersection.hpp"

T area(const Polygon& poly) {
    const int n = poly.size();
    T res = 0;
    for (int i = 0; i < n; ++i) {
        res += cross(poly[i], poly[(i + 1) % n]);
    }
    return std::abs(res) / T(2);
}

bool is_convex(const Polygon& poly) {
    int n = poly.size();
    for (int i = 0; i < n; ++i) {
        if (lt(cross(poly[(i+1)%n] - poly[i], poly[(i+2)%n] - poly[(i+1)%n]), 0)) {
            return false;
        }
    }
    return true;
}

Polygon convex_cut(const Polygon& poly, const Line& l) {
    const int n = poly.size();
    Polygon res;
    for (int i = 0; i < n; ++i) {
        auto p = poly[i], q = poly[(i+1)%n];
        if (ccw(l.p1, l.p2, p) != -1) {
            if (res.empty() || !eq(res.back(), p)) {
                res.push_back(p);
            }
        }
        if (ccw(l.p1, l.p2, p) * ccw(l.p1, l.p2, q) < 0) {
            auto c = intersection(Line(p, q), l);
            if (res.empty() || !eq(res.back(), c)) {
                res.push_back(c);
            }
        }
    }
    return res;
}

Polygon halfplane_intersection(std::vector<std::pair<Vec, Vec>> hps) {
    using Hp = std::pair<Vec, Vec>;  // (normal vector, a point on the border)

    auto intersection = [&](const Hp& l1, const Hp& l2) -> Vec {
        auto d = l2.second - l1.second;
        return l1.second + (dot(d, l2.first) / cross(l1.first, l2.first)) * perp(l1.first
            );
    };

    // check if the halfplane h contains the point p
    auto contains = [&](const Hp& h, const Vec& p) -> bool {
        return dot(p - h.second, h.first) > 0;
    };

    constexpr T INF = 1e15;
    hps.emplace_back(Vec(1, 0), Vec(-INF, 0));  // -INF <= x
    hps.emplace_back(Vec(-1, 0), Vec(INF, 0));  // x <= INF
    hps.emplace_back(Vec(0, 1), Vec(0, -INF));  // -INF <= y
    hps.emplace_back(Vec(0, -1), Vec(0, INF));  // y <= INF

    std::sort(hps.begin(), hps.end(), [&](const auto& h1, const auto& h2) {
        return std::arg(h1.first) < std::arg(h2.first);
    });

    std::deque<Hp> dq;
    int len = 0;
    for (auto& hp : hps) {
        while (len > 1 && !contains(hp, intersection(dq[len-1], dq[len-2]))) {
            dq.pop_back();
            --len;
        }

        while (len > 1 && !contains(hp, intersection(dq[0], dq[1]))) {
            dq.pop_front();
            --len;
        }

        // parallel
        if (len > 0 && eq(cross(dq[len-1].first, hp.first), 0)) {
            // opposite
            if (lt(dot(dq[len-1].first, hp.first), 0)) {
                return {};
            }
            // same
            if (!contains(hp, dq[len-1].second)) {
                dq.pop_back();
                --len;
            } else continue;
        }

        dq.push_back(hp);
        ++len;
    }
```

```
    while (len > 2 && !contains(dq[0], intersection(dq[len-1], dq[len-2]))) {
        dq.pop_back();
        --len;
    }

    while (len > 2 && !contains(dq[len-1], intersection(dq[0], dq[1]))) {
        dq.pop_front();
        --len;
    }

    if (len < 3) return {};

    std::vector<Vec> poly(len);
    for (int i = 0; i < len - 1; ++i) {
        poly[i] = intersection(dq[i], dq[i+1]);
    }
    poly[len-1] = intersection(dq[len-1], dq[0]);
    return poly;
}
```

6.7 triangle.hpp

```
#include "geometry.hpp"
#include "intersection.hpp"
#include "bisector.hpp"

Vec centroid(const Vec& A, const Vec& B, const Vec& C) {
    assert(ccw(A, B, C) != 0);
    return (A + B + C) / T(3);
}

Vec incenter(const Vec& A, const Vec& B, const Vec& C) {
    assert(ccw(A, B, C) != 0);
    T a = std::abs(B - C);
    T b = std::abs(C - A);
    T c = std::abs(A - B);
    return (a*A + b*B + c*C) / (a + b + c);
}

Vec circumcenter(const Vec& A, const Vec& B, const Vec& C) {
    assert(ccw(A, B, C) != 0);
    return intersection(bisector(Segment(A, B)), bisector(Segment(A, C)));
}
```

6.8 Convex Hull

時間計算量: $O(n \log n)$

```
#include "geometry.hpp"

std::vector<Vec> convex_hull(std::vector<Vec>& pts) {
    int n = pts.size();
    if (n == 1) return pts;
    std::sort(pts.begin(), pts.end(), [](const Vec& v1, const Vec& v2) {
        return (v1.imag() != v2.imag()) ? (v1.imag() < v2.imag()) : (v1.real() < v2.real
            ());
    });
    int k = 0; // the number of vertices in the convex hull
    std::vector<Vec> ch(2 * n);
    // right
    for (int i = 0; i < n; ++i) {
        while (k > 1 && lt(cross(ch[k-1] - ch[k-2], pts[i] - ch[k-1]), 0)) --k;
        ch[k++] = pts[i];
    }
    int t = k;
    // left
    for (int i = n - 2; i >= 0; --i) {
        while (k > t && lt(cross(ch[k-1] - ch[k-2], pts[i] - ch[k-1]), 0)) --k;
        ch[k++] = pts[i];
    }
    ch.resize(k - 1);
    return ch;
}
```

6.9 bisector.hpp

```
#include "geometry.hpp"
#include "intersection.hpp"

Line bisector(const Segment& s) {
    auto m = (s.p1 + s.p2) / T(2);
    return Line(m, m + Vec(-s.dir().imag(), s.dir().real()));
}

std::pair<Line, Line> bisector(const Line& l, const Line& m) {
    // parallel
    if (eq(cross(l.dir(), m.dir()), 0)) {
        auto n = Line(l.p1, l.p1 + perp(l.dir()));
        auto p = intersection(n, m);
        auto m = (l.p1 + p) / T(2);
        return {Line(m, m + l.dir()), Line()};
    }
    auto p = intersection(l, m);
    T ang = (std::arg(l.dir()) + std::arg(m.dir())) / T(2);
    auto b1 = Line(p, p + std::polar(T(1), ang));
    auto b2 = Line(p, p + std::polar(T(1), ang + PI / T(2)));
    return {b1, b2};
}
```

7 graph

7.1 Strongly Connected Components

強連結成分のラベルはトポロジカル順序になっている.

```
std::vector<int> scc(const std::vector<std::vector<int>>& G) {
    const int n = G.size();
    std::vector<std::vector<int>> G_rev(n);
    for (int u = 0; u < n; ++u) {
        for (int v : G[u]) G_rev[v].push_back(u);
    }
    std::vector<int> comp(n, -1), order(n);
    std::vector<bool> visited(n);

    auto dfs = [&](const auto& self, int u) -> void {
        if (visited[u]) return;
        visited[u] = true;
        for (int v : G[u]) self(self, v);
```

```
        order.push_back(u);
    };

    for (int v = 0; v < n; ++v) dfs(dfs, v);
    std::reverse(order.begin(), order.end());
    int c = 0;

    auto rdfs = [&](const auto& self, int u, int c) -> void {
        if (comp[u] != -1) return;
        comp[u] = c;
        for (int v : G_rev[u]) self(self, v, c);
    };

    for (int v : order) if (comp[v] == -1) rdfs(rdfs, v, c++);
    return comp;
}
```

7.2 Lowlink

橋，関節点を求めるのに使う

グラフの DFS tree において，頂点 v の訪問時刻を `ord[v]` としたとき，v から後退辺 (DFS tree に含まれない辺) を高々 1 回用いて到達することができる頂点の `ord` の最小値 `low[v]` を lowlink という．

```
class Lowlink {
public:
    std::vector<int> ord, low;
    std::vector<std::pair<int, int>> bridge;
    std::vector<int> articulation;

    Lowlink() = default;
    explicit Lowlink(const std::vector<std::vector<int>>& G) : ord(G.size(), -1), low(G.
        size()), G(G) {
        for (int i = 0; i < (int) G.size(); ++i) {
            if (ord[i] == -1) dfs(i, -1);
        }
    }

    bool is_bridge(int u, int v) const {
        if (ord[u] > ord[v]) std::swap(u, v);
        return ord[u] < low[v];
    }

private:
    std::vector<std::vector<int>> G;
    int k = 0;

    void dfs(int v, int p) {
        ord[v] = k++;
        low[v] = ord[v];
        bool is_articulation = false, checked = false;
        int cnt = 0;
        for (int c : G[v]) {
            if (c == p && !checked) {
                checked = true;
                continue;
            }
            if (ord[c] == -1) {
                ++cnt;
                dfs(c, v);
                low[v] = std::min(low[v], low[c]);
                if (p != -1 && ord[v] <= low[c]) is_articulation = true;
                if (ord[v] < low[c]) bridge.push_back(std::minmax(v, c));
            } else {
                low[v] = std::min(low[v], ord[c]);
            }
        }
        if (p == -1 && cnt > 1) is_articulation = true;
        if (is_articulation) articulation.push_back(v);
    }
};
```

8 math

8.1 知識

- Euler のトーシェント関数: n の相異なる素因数を $p_1, p_2, \dots$ として，

$$\varphi(n) = n \prod_{p_i} \frac{p_i - 1}{p_i}$$

- Möbius 関数:

$$\mu(n) = \begin{cases} 0 & \text{if } n \text{ has a square divisor} \\ (-1)^k & \text{if } n \text{ has } k \text{ prime factors} \end{cases}$$

- Monmort 数 (撹乱順列の個数):

$$W_1 = 0,\ W_2 = 1,\ W_k = (k-1)(W_{k-1} + W_{k-2})$$

- 第 1 種 Stirling 数 $s(n, k)$
  - 定義:

$$x(x-1)\cdots(x-(n-1)) = \sum_{k=0}^{n} s(n,k) x^k$$

  - $s(n, k)$ の絶対値 ($\begin{bmatrix} n \\ k \end{bmatrix}$ と書く) は，n 要素の置換のうち，k 個のサイクルに分解されるものの個数である．
  - 漸化式:

$$\begin{bmatrix} n \\ k \end{bmatrix} = \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} + (n-1) \begin{bmatrix} n-1 \\ k \end{bmatrix}$$

- 第 2 種 Stirling 数 $\left\{ {n \atop k} \right\}$
  - 定義:

$$\left\{ {n \atop k} \right\} = \frac{1}{k!} \sum_{i=0}^{n} (-1)^{k-i} \binom{k}{i} i^n$$

  - $\left\{ {n \atop k} \right\}$ は，n 個の区別できるボールを， k 個の区別できない箱に，すべての箱に 1 つ以上のボールが入るように分配する方法の数である．
  - 漸化式:

$$\left\{ {n \atop k} \right\} = \left\{ {n-1 \atop k-1} \right\} + k \left\{ {n-1 \atop k} \right\}$$

- Lagrange 補間: $(x_1, y_1), \ldots, (x_n, y_n)$ を通る $n+1$ 次多項式は,

$$\delta_i(x) = \frac{(x-x_1)\cdots(x-x_{i-1})(x-x_{i+1})\cdot(x_i-x_n)}{(x_i-x_1)\cdots(x_i-x_{i-1})(x_i-x_{i+1})\cdot(x_i-x_n)}$$

として,

$$\sum_i y_i \delta_i(x)$$

8.2 行列

- 掃き出し法: 以下のコードを参照
- 行列式: 掃き出して対角要素の積. swap のたびに -1 をかけることに注意
- 逆行列: $\begin{pmatrix} A & I \end{pmatrix}$ を掃き出す
- 連立一次方程式: $\begin{pmatrix} A & \boldsymbol{b} \end{pmatrix}$ を掃き出す

```
int pivot = 0;
for (int j = 0; j < n; ++j) {
    int i = pivot;
    while (i < m && eq(A[i][j], T(0))) ++i;
    if (i == m) continue;

    if (i != pivot) A[i].swap(A[pivot]);

    T p = A[pivot][j];
    for (int l = j; l < n; ++l) A[pivot][l] /= p;

    for (int k = 0; k < m; ++k) {
        if (k == pivot) continue;
        T v = A[k][j];
        for (int l = j; l < n; ++l) {
            A[k][l] -= A[pivot][l] * v;
        }
    }

    ++pivot;
}
return A;
```

8.3 商が一定の区間の列挙

```
ll i = 1;
while (i <= n) {
    ll q = n / i;
    ll j = n / q + 1;
    // [i, j) では n/k の値が一定
    i = j;
}
```

8.4 Extended Euclidean Algorithm

$ax + by = \gcd(a, b)$ の解 (x, y) を 1 組求める. これを利用して, 与えられた整数 a の法 mod での逆元を求めることができる.

```
std::pair<long long, long long> extgcd(long long a, long long b) {
    long long s = a, sx = 1, sy = 0, t = b, tx = 0, ty = 1;
    while (t) {
        long long q = s / t;
        std::swap(s -= t * q, t);
        std::swap(sx -= tx * q, tx);
        std::swap(sy -= ty * q, ty);
    }
    return {sx, sy};
}

long long mod_inv(long long a, long long mod) {
    long long inv = extgcd(a, mod).first;
    return (inv % mod + mod) % mod;
}
```

8.5 Garner's Algorithm

連立合同式 $x \equiv b_i \mod m_i \quad (i = 1, \ldots, n)$ の解を求める. m_i が pairwise coprime であるとき, この連立合同式には法 $m = m_1 \ldots m_n$ のもとでただ一つの解が存在することが中国の剰余定理によって保証される.

- `long long garner(vector<long long> b, vector<long long> m, long long mod)`
  - 連立合同式を満たす最小の非負整数を法 mod で求める.
  - 時間計算量: $O(n^2)$

```
#include "extgcd.cpp"

long long garner(const std::vector<long long>& b, std::vector<long long> m, long long mod)
    {
    m.push_back(mod);
    int n = m.size();
    std::vector<long long> coeffs(n, 1);
    std::vector<long long> consts(n, 0);
    for (int k = 0; k < n - 1; ++k) {
        long long t = (b[k] - consts[k]) * mod_inv(coeffs[k], m[k]) % m[k];
        if (t < 0) t += m[k];
        for (int i = k + 1; i < n; ++i) {
            consts[i] = (consts[i] + t * coeffs[i]) % m[i];
            coeffs[i] = coeffs[i] * m[k] % m[i];
        }
    }
    return consts.back();
}
```

8.6 Modular Arithmetic

Modular Exponentiation

$a^e \mod mod$

- `long long mod_pow(long long a, long long e, int mod)`
  - $a^e \mod mod$ を計算する

Discrete Logarithm

$a^x \equiv b \mod mod$ を満たす x を求める.

- `int mod_log(long long a, long long b, int mod)`
  - $a^x \equiv b \mod mod$ を満たす x を求める. 存在しない場合は -1 を返す.
  - 時間計算量: $O(\sqrt{mod})$

Quadratic Residue

$r^2 \equiv n \mod mod$ を満たす r を求める.

- `long long mod\_sqrt(long long n, int mod)`
  - $r^2 \equiv n \mod mod$ を満たす r を求める．$n = 0$ のときは 0 を返す．n と mod が互いに素でないときと r が存在しないときは -1 を返す．

```
long long mod_pow(long long a, long long e, int mod) {
    long long ret = 1;
    while (e > 0) {
        if (e & 1) ret = ret * a % mod;
        a = a * a % mod;
        e >>= 1;
    }
    return ret;
}

long long mod_inv(long long a, int mod) {
    return mod_pow(a, mod - 2, mod);
}

int mod_log(long long a, long long b, int mod) {
    // make a and mod coprime
    a %= mod;
    b %= mod;
    long long k = 1, add = 0, g;
    while ((g = std::gcd(a, mod)) > 1) {
        if (b == k) return add;
        if (b % g) return -1;
        b /= g;
        mod /= g;
        ++add;
        k = k * a / g % mod;
    }

    // baby-step
    const int m = sqrt(mod) + 1;
    std::unordered_map<long long, int> baby_index;
    long long baby = b;
    for (int i = 0; i <= m; ++i) {
        baby_index[baby] = i;
        baby = baby * a % mod;
    }

    // giant-step
    long long am = 1;
    for (int i = 0; i < m; ++i) am = am * a % mod;
    long long giant = k;
    for (int i = 1; i <= m; ++i) {
        giant = giant * am % mod;
        if (baby_index.count(giant)) {
            return i * m - baby_index[giant] + add;
        }
    }
    return -1;
}

long long mod_sqrt(long long n, int mod) {
    if (n == 0) return 0;
    if (mod == 2) return 1;
    if (std::gcd(n, mod) != 1) return -1;
    if (mod_pow(n, (mod - 1) / 2, mod) == mod - 1) return -1;

    int Q = mod - 1, S = 0;
    while (!(Q & 1)) Q >>= 1, ++S;
    long long z = 2;
    while (true) {
        if (mod_pow(z, (mod - 1) / 2, mod) == mod - 1) break;
        ++z;
    }
    int M = S;
    long long c = mod_pow(z, Q, mod);
    long long t = mod_pow(n, Q, mod);
    long long R = mod_pow(n, (Q + 1) / 2, mod);
    while (t != 1) {
        int i = 0;
        long long s = t;
        while (s != 1) {
            s = s * s % mod;
            ++i;
        }
        long long b = mod_pow(c, 1 << (M - i - 1), mod);
        M = i;
        c = b * b % mod;
        t = t * c % mod;
        R = R * b % mod;
    }
    return R;
}
```

8.7 Sum of Floor of Linear

一次関数の床関数の和 $\sum_{i=0}^{N-1} \left\lfloor \frac{Ai+B}{M} \right\rfloor$ を再帰的に計算する．

```
long long floor_sum(long long n, long long m, long long a, long long b) {
    long long sum = 0;
    if (a >= m) {
        sum += (a / m) * n * (n - 1) / 2;
        a %= m;
    }
    if (b >= m) {
        sum += (b / m) * n;
        b %= m;
    }
    long long y = (a * n + b) / m;
    if (y == 0) return sum;
    long long x = (m * y - b + a - 1) / a;
    sum += (n - x) * y + floor_sum(y, a, m, a * x - m * y + b);
    return sum;
}
```

8.8 Polynomial

```
template <typename mint>
class Polynomial : public std::vector<mint> {
    using Poly = Polynomial;

public:
    using std::vector<mint>::vector;
    using std::vector<mint>::operator=;

    Poly pre(int size) const { return Poly(this->begin(), this->begin() + std::min((int)
        this->size(), size)); }
```

```
    Poly rev(int deg = -1) const {
        Poly ret(*this);
        if (deg != -1) ret.resize(deg, 0);
        return Poly(ret.rbegin(), ret.rend());
    }

    Poly& operator+=(const Poly& rhs) {
        if (this->size() < rhs.size()) this->resize(rhs.size());
        for (int i = 0; i < (int) rhs.size(); ++i) (*this)[i] += rhs[i];
        return *this;
    }

    Poly& operator+=(const mint& rhs) {
        if (this->empty()) this->resize(1);
        (*this)[0] += rhs;
        return *this;
    }

    Poly& operator-=(const Poly& rhs) {
        if (this->size() < rhs.size()) this->resize(rhs.size());
        for (int i = 0; i < (int) rhs.size(); ++i) (*this)[i] -= rhs[i];
        return *this;
    }

    Poly& operator-=(const mint& rhs) {
        if (this->empty()) this->resize(1);
        (*this)[0] -= rhs;
        return *this;
    }

    Poly& operator*=(const Poly& rhs) {
        *this = convolution(*this, rhs);
        // // naive convolution O(N^2)
        // std::vector<mint> res(this->size() + rhs.size() - 1);
        // for (int i = 0; i < (int) this->size(); ++i) {
        //     for (int j = 0; j < (int) rhs.size(); ++j) {
        //         res[i + j] += (*this)[i] * rhs[j];
        //     }
        // }
        // *this = res;
        return *this;
    }

    Poly& operator*=(const mint& rhs) {
        for (int i = 0; i < (int) this->size(); ++i) (*this)[i] *= rhs;
        return *this;
    }

    Poly& operator/=(const Poly& rhs) {
        if(this->size() < rhs.size()) {
            this->clear();
            return *this;
        }
        int n = this->size() - rhs.size() + 1;
        return *this = (rev().pre(n) * rhs.rev().inv(n)).pre(n).rev(n);
    }

    Poly& operator%=(const Poly& rhs) {
        *this -= *this / rhs * rhs;
        while (!this->empty() && this->back() == 0) this->pop_back();
        return *this;
    }

    Poly& operator-() const {
        Poly ret(this->size());
        for (int i = 0; i < (int) this->size(); ++i) ret[i] = -(*this)[i];
        return ret;
    }

    Poly operator+(const Poly& rhs) const { return Poly(*this) += rhs; }
    Poly operator+(const mint& rhs) const { return Poly(*this) += rhs; }
    Poly operator-(const Poly& rhs) const { return Poly(*this) -= rhs; }
    Poly operator-(const mint& rhs) const { return Poly(*this) -= rhs; }
    Poly operator*(const Poly& rhs) const { return Poly(*this) *= rhs; }
    Poly operator*(const mint& rhs) const { return Poly(*this) *= rhs; }
    Poly operator/(const Poly& rhs) const { return Poly(*this) /= rhs; }
    Poly operator%(const Poly& rhs) const { return Poly(*this) %= rhs; }

    Poly operator<<(int n) const {
        Poly ret(*this);
        ret.insert(ret.begin(), n, 0);
        return ret;
    }

    Poly operator>>(int n) const {
        if (this->size() <= n) return {};
        Poly ret(*this);
        ret.erase(ret.begin(), ret.begin() + n);
        return ret;
    }


    mint operator()(const mint& x) {
        mint y = 0, powx = 1;
        for (int i = 0; i < (int) this->size(); ++i) {
            y += (*this)[i] * powx;
            powx *= x;
        }
        return y;
    }

    Poly inv(int deg = -1) const {
        assert((*this)[0] != mint(0));
        if (deg == -1) deg = this->size();
        Poly ret({mint(1) / (*this)[0]});
        for (int i = 1; i < deg; i <<= 1) {
            ret = (ret * mint(2) - ret * ret * this->pre(i << 1)).pre(i << 1);
        }
        return ret;
    }

    Poly exp(int deg = -1) const {
        assert((*this)[0] == mint(0));
        if (deg == -1) deg = this->size();
        Poly ret({mint(1)});
        for (int i = 1; i < deg; i <<= 1) {
            ret = (ret * (this->pre(i << 1) + mint(1) - ret.log(i << 1))).pre(i << 1);
        }
        return ret;
```

```
    }

    Poly log(int deg = -1) const {
        assert((*this)[0] == mint(1));
        if (deg == -1) deg = this->size();
        return (this->diff() * this->inv(deg)).pre(deg - 1).integral();
    }

    Poly pow(long long k, int deg = -1) const {
        if (k == 0) return {1};
        if (deg == -1) deg = this->size();
        Poly ret(*this);
        int cnt = 0;
        while (cnt < (int) ret.size() && ret[cnt] == mint(0)) ++cnt;
        if (cnt * k >= deg) return Poly(deg, mint(0));
        ret.erase(ret.begin(), ret.begin() + cnt);
        deg -= cnt * k;
        ret = ((ret * mint(ret[0]).inv()).log(deg) * mint(k)).pre(deg).exp(deg) * mint(ret
            [0]).pow(k);
        ret.insert(ret.begin(), cnt * k, mint(0));
        return ret;
    }

    Poly diff() const {
        Poly ret(std::max(0, (int) this->size() - 1));
        for (int i = 1; i <= (int) ret.size(); ++i) ret[i - 1] = (*this)[i] * mint(i);
        return ret;
    }

    Poly integral() const {
        Poly ret(this->size() + 1);
        ret[0] = mint(0);
        for (int i = 0; i < (int) ret.size() - 1; ++i) ret[i + 1] = (*this)[i] / mint(i +
            1);
        return ret;
    }

    Poly taylor_shift(long long c) const {
        const int n = this->size();
        std::vector<mint> fact(n, 1), fact_inv(n, 1);
        for (int i = 1; i < n; ++i) fact[i] = fact[i-1] * i;
        fact_inv[n-1] = mint(1) / fact[n-1];
        for (int i = n - 1; i > 0; --i) fact_inv[i-1] = fact_inv[i] * i;

        auto ret = *this;
        Poly e(n+1);
        e[0] = 1;
        mint p = c;
        for (int i = 1; i < n; ++i) {
            ret[i] *= fact[i];
            e[i] = p * fact_inv[i];
            p *= c;
        }
        ret = (ret.rev() * e).pre(n).rev();
        for (int i = n - 1; i >= 0; --i) {
            ret[i] *= fact_inv[i];
        }
        return ret;
    }
};
```

9 misc

9.1 Mo's Algorithm

- `Mo(int n)`
  - 長さ n の列に対するクエリを処理する
  - 時間計算量: $O(1)$
- `void query(int l, int r)`
  - 区間 $[l, r)$ に対してクエリをする
  - 時間計算量: $O(1)$
- `void run(ExL exl, ShL shl, ExR exr, ShR shr, Out out)`
  - 以下の関数を引数に取り，クエリを実行する
    - `exl`: 区間を左に 1 マス伸ばしたときの状態を更新する
    - `shl`: 区間を左に 1 マス縮めたときの状態を更新する
    - `exr`: 区間を右に 1 マス伸ばしたときの状態を更新する
    - `shr`: 区間を右に 1 マス縮めたときの状態を更新する
    - `out`: i 番目のクエリの結果を計算する
  - 時間計算量: $O(f(n)n\sqrt{n})$, $f(n)$ は状態の更新にかかる計算量

```
class Mo {
public:
    Mo() = default;
    explicit Mo(int n) : n(n), cnt(0) {}

    void query(int l, int r) {
        queries.emplace_back(cnt++, l, r);
    }

    template <typename ExL, typename ShL, typename ExR, typename ShR, typename Out>
    void run(ExL exl, ShL shl, ExR exr, ShR shr, Out out) {
        int s = sqrt(n);
        std::sort(queries.begin(), queries.end(), [&](const auto& a, const auto& b) {
            if (a.l / s != b.l / s) return a.l < b.l;
            return a.r < b.r;
        });
        int curL = 0, curR = 0;
        for (auto [id, l, r] : queries) {
            while (curL > l) exl(--curL);
            while (curR < r) exr(curR++);
            while (curL < l) shl(curL++);
            while (curR > r) shr(--curR);
            out(id);
        }
    }

private:
    struct Query {
        int id, l, r;
        Query(int id, int l, int r) : id(id), l(l), r(r) {}
    };

    int n, cnt;
    std::vector<Query> queries;
};
```

9.2 Interval Set

Description

整数の閉区間の集合を管理する.

Operations

- `bool covered(T x)`
- `bool covered(T l, T r)`
  - 区間 $[l, r]$ が含まれているか判定する
  - 時間計算量: $O(\log n)$
- `pair<T, T> covered_by(T x)`
- `pair<T, T> covered_by(T l, T r)`
  - 区間 $[l, r]$ を含む区間を返す．そのような区間がない場合は $(-\infty, \infty)$ を返す
  - 時間計算量: $O(\log n)$
- `void insert(T x)`
- `void insert(T l, T r)`
  - 区間 $[l, r]$ を集合に追加する
  - 時間計算量: amortized $O(\log n)$
- `void erase(T x)`
- `void erase(T l, T r)`
  - 区間 $[l, r]$ を集合から削除する
  - 時間計算量: amortized $O(\log n)$
- `T mex(T x)`
  - x 以上の整数のうち，集合に含まれない最小のものを返す
  - 時間計算量: $O(\log n)$

```cpp
template <typename T>
class IntervalSet {
public:
    static constexpr T INF = std::numeric_limits<T>::max() / 2;

    IntervalSet() {
        st.emplace(INF, INF);
        st.emplace(-INF, -INF);
    }

    bool covered(T x) const { return covered(x, x); }
    bool covered(T l, T r) const {
        assert(l <= r);
        auto it = --(st.lower_bound({l + 1, l + 1}));
        return it->first <= l && r <= it->second;
    }

    std::pair<T, T> covered_by(T x) const { return covered_by(x, x); }
    std::pair<T, T> covered_by(T l, T r) const {
        assert(l <= r);
        auto it = --(st.lower_bound({l + 1, l + 1}));
        if (it->first <= l && r <= it->second) return *it;
        return {-INF, -INF};
    }

    void insert(T x) { insert(x, x); }
    void insert(T l, T r) {
        assert(l <= r);
        auto it = --(st.lower_bound({l + 1, l + 1}));
        if (it->first <= l && r <= it->second) return;
        if (it->first <= l && l <= it->second + 1) {
            l = it->first;
            it = st.erase(it);
        } else {
            ++it;
        }
        while (it->second < r) {
            it = st.erase(it);
        }
        if (it->first - 1 <= r && r <= it->second) {
            r = it->second;
            st.erase(it);
        }
        st.emplace(l, r);
    }

    void erase(T x) { erase(x, x); }
    void erase(T l, T r) {
        assert(l <= r);
        auto it = --(st.lower_bound({l + 1, l + 1}));
        if (it->first <= l && r <= it->second) {
            if (it->first < l) st.emplace(it->first, l - 1);
            if (r < it->second) st.emplace(r + 1, it->second);
            st.erase(it);
            return;
        }
        if (it->first <= l && l <= it->second) {
            if (it->first < l) st.emplace(it->first, l - 1);
            it = st.erase(it);
        } else {
            ++it;
        }
        while (it->second <= r) {
            it = st.erase(it);
        }
        if (it->first <= r && r <= it->second) {
            if (r < it->second) st.emplace(r + 1, it->second);
            st.erase(it);
        }
    }

    std::set<std::pair<T, T>> ranges() const { return st; }

    T mex(T x) const {
        auto it = --(st.lower_bound({x + 1, x + 1}));
        if (it->first <= x && x <= it->second) return it->second + 1;
        return x;
    }

private:
    std::set<std::pair<T, T>> st;
};
```

9.3 2-SAT

- `vector<bool> two\_sat(int n, vector<tuple<int, bool, int, bool>> clauses)`
  - n リテラルを含む節のリストが与えられた時，すべての節を充足するリテラルの真偽値の組み合わせを一つ返す．節は {`i, f, j, g`} の形で与え，$((x_i = f) \vee (x_j = g))$ を追加する．問題が充足可能でない場合，空リストを返す．
  - 時間計算量: $O(n)$

```cpp
#include "../graph/scc.cpp"

std::vector<bool> two_sat(int n, const std::vector<std::tuple<int, bool, int, bool>>&
    clauses) {
    std::vector<std::vector<int>> G(2 * n);
    std::vector<bool> val(n);

    for (auto& [i, f, j, g] : clauses) {
        G[n * f + i].push_back(n * (!g) + j);
        G[n * g + j].push_back(n * (!f) + i);
    }

    auto comp = scc(G);
    for (int i = 0; i < n; ++i) {
        if (comp[i] == comp[n + i]) {
            // not satisfiable
            return {};
        }
        val[i] = comp[i] > comp[n + i];
    }
    return val;
}
```

9.4 Horn-SAT

Horn-SAT は，節内の肯定リテラル数が高々 1 つであるような乗法標準形の論理式に対する充足可能性問題 (SAT) である．

$(a_1 \wedge a_2 \wedge \cdots \wedge a_n \to b)$ や $(a_1 \wedge a_2 \wedge \cdots \wedge a_n \to \neg b)$ といった論理式は，高々 1 つの肯定リテラルからなる節 (Horn 節) に変換できる．

この問題は unit-propagation によって解ける．肯定リテラルのみからなる節がない場合はすべてのリテラルを ‘false‘ にし，そのような節があればそのリテラルの値を ‘true‘ に確定し残りのリテラルについて同じ問題を解けば良い．

10 string

10.1 Rolling Hash

mod $2^{61} - 1$

- `RollingHash(string s, long long base)`
- `RollingHash(vector<T> s, long long base)`
  - `s` のハッシュ値を計算する
  - 時間計算量: $O(n)$
- `static long long generate\_base()`
  - ランダムな基数を返す
  - 時間計算量: $O(1)$
- `long long query(int l, int r)`
  - 区間 $[l, r)$ のハッシュ値を返す
  - 時間計算量: $O(1)$
- `long long combine(long long h1, long long h2, int len2)`
  - ハッシュ値 $h1$ と $h2$ を結合する．$h2$ の長さを $len2$ である
  - 時間計算量: $O(1)$
- `void push\_back(char c)`
  - 文字 c を末尾に結合する
  - 時間計算量: $O(1)$

```cpp
class RollingHash {
public:
    static long long generate_base() {
        std::random_device rd;
        std::mt19937_64 rng(rd());
        std::uniform_int_distribution<long long> rand(1, mod - 1);
        return rand(rng);
    }

    RollingHash() = default;
    RollingHash(const std::string& s, long long base) : RollingHash(std::vector<char>(s.
        begin(), s.end()), base) {}
    template <typename T>
    RollingHash(const std::vector<T>& s, long long base)
        : base(base), hashed(s.size() + 1), power(s.size() + 1) {
        power[0] = 1;
        for (int i = 0; i < (int) s.size(); ++i) {
            power[i + 1] = mul(power[i], base);
            hashed[i + 1] = add(mul(hashed[i], base), s[i]);
        }
    }

    long long query(int l, int r) const {
        return add(hashed[r], mod - mul(hashed[l], power[r - l]));
    }

    long long combine(long long h1, long long h2, int len2) const {
        return add(mul(h1, power[len2]), h2);
    }

    void push_back(char c) {
        power.push_back(mul(power.back(), base));
        hashed.push_back(add(mul(hashed.back(), base), c));
    }

private:
    static constexpr long long mod = (1LL << 61) - 1;

    static inline long long add(long long a, long long b) {
        if ((a += b) >= mod) a -= mod;
        return a;
    }

    static inline long long mul(long long a, long long b) {
        __int128_t c = (__int128_t) a * b;
```

```
        return add(c >> 61, c & mod);
    }

    const long long base;
    std::vector<long long> hashed, power;
};
```

10.2 Suffix Array

時間計算量: $O(n \log n)$

```
std::vector<int> suffix_array(const std::string& s) {
    int n = s.size();
    std::vector<int> sa(n);
    std::iota(sa.begin(), sa.end(), 0);
    std::sort(sa.begin(), sa.end(), [&](int i, int j) {
        return s[i] < s[j];
    });
    int cl = 0;
    std::vector<int> rank(n);
    for (int i = 1; i < n; ++i) {
        if (s[sa[i - 1]] != s[sa[i]]) ++cl;
        rank[sa[i]] = cl;
    }
    std::vector<int> tmp(n), nrank(n), cnt(n);
    for (int k = 1; k < n; k <<= 1) {
        // sort by second half
        int cnt1 = 0, cnt2 = k;
        for (int i = 0; i < n; ++i) {
            int j = sa[i] - k;
            if (j >= 0) tmp[cnt2++] = j;
            else tmp[cnt1++] = j + n;
        }

        // sort by first half
        std::fill(cnt.begin(), cnt.end(), 0);
        for (int i = 0; i < n; ++i) ++cnt[rank[tmp[i]]];
        for (int i = 1; i < n; ++i) cnt[i] += cnt[i - 1];
        for (int i = n - 1; i >= 0; --i) sa[--cnt[rank[tmp[i]]]] = tmp[i];

        // assign new rank
        nrank[sa[0]] = 0;
        cl = 0;
        for (int i = 1; i < n; ++i) {
            if (rank[sa[i - 1]] != rank[sa[i]]
                || (sa[i - 1] + k < n ? rank[sa[i - 1] + k] : -1) != (sa[i] + k < n ? rank
                    [sa[i] + k] : -1)) {
                ++cl;
            }
            nrank[sa[i]] = cl;
        }
        std::swap(rank, nrank);
    }
    return sa;
}

/*
// comparator for substrings
// used for string matching with the suffix array

auto cmp = [&](int si, const string& t) {
    int sn = S.size(), tn = t.size();
    int ti = 0;
    for (; si < sn && ti < tn; ++si, ++ti) {
        if (T[si] < t[ti]) return true;
        if (T[si] > t[ti]) return false;
    }
    return si == sn && ti < tn;
};
*/
```

10.3 Longest Common Prefix Array

高さ配列 (LCP array) は，接尾辞配列における隣同士の接尾辞で，先頭何文字が共通しているかを表す配列である．`lcp[i]` は接尾辞 `s[sa[i]..]` と接尾辞 `s[sa[i + 1]..]` の先頭で共通している文字数になる．

時間計算量: $O(n)$

```
std::vector<int> lcp_array(const std::string& s, const std::vector<int>& sa) {
    int n = s.size();
    std::vector<int> rank(n);
    for (int i = 0; i < n; ++i) rank[sa[i]] = i;
    int h = 0;
    std::vector<int> lcp(n - 1);
    for (int i = 0; i < n; ++i) {
        if (h > 0) --h;
        if (rank[i] == 0) continue;
        int j = sa[rank[i] - 1];
        while (j + h < n && i + h < n && s[j + h] == s[i + h]) ++h;
        lcp[rank[i] - 1] = h;
    }
    return lcp;
}
```

10.4 Z Array

Z array は，文字列 `S` と `S[i:]` の最長共通接頭辞の長さを表す配列である

Z array を用いると文字列中のパターンをすべて検索することができる．文字列を `S`，パターンを `P` とし，`P$S` (`$` は `S` にも `P` にも含まれない文字) の Z array を構築すると，値が `P` の長さと一致するところで `P` が現れる．

時間計算量: $O(n)$

```
std::vector<int> z_array(const std::string& s) {
    int n = s.size();
    std::vector<int> z(n);
    z[0] = n;
    int l = 0, r = 0;
    for (int i = 1; i < n; ++i) {
        int k = i - l;
        if (i <= r && z[k] < r - i + 1) {
            z[i] = z[k];
        } else {
            l = i;
            if (i > r) r = i;
            while (r < n && s[r - l] == s[r]) ++r;
            --r;
            z[i] = r - l + 1;
```

```
        }
    }
    return z;
}
```

10.5 Trie

トライ木

- `void insert(string s, int id)`
  - 文字列 s を挿入する
  - 時間計算量: $O(|s|)$
- `void compress()`
  - トライ木を圧縮して Patricia trie を構築する．これにより，木の木の深さが $O(\sqrt{\sum |s|})$ になる．
  - 時間計算量: $O(\sum |s|)$

```
class Trie {
public:
    Trie() : root(std::make_shared<Node>()) {}

    void insert(const std::string& s, int id) { insert(root, s, id, 0); }

    void compress() { compress(root); }

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

    struct Node {
        std::map<char, node_ptr> ch;
        std::vector<int> accept;
        int sz = 0;
        node_ptr par;
        std::string str;

        Node() = default;
    };

    const node_ptr root;

    void insert(const node_ptr& t, const std::string& s, int id, int k) {
        ++t->sz;
        if (k == (int) s.size()) {
            t->accept.push_back(id);
            return;
        }
        int c = s[k];
        if (!t->ch.count(c)) {
            t->ch[c] = std::make_shared<Node>();
            t->ch[c]->par = t;
            t->ch[c]->str = c;
        }
        insert(t->ch[c], s, id, k + 1);
    }

    void compress(node_ptr t) {
        while (t->accept.empty() && t->ch.size() == 1) {
            auto u = t->ch.begin()->second;
            t->ch = u->ch;
            t->accept = u->accept;
            t->str += u->str;
            for (auto [c, w] : t->ch) w->par = t;
            compress(t);
        }
        for (auto [c, u] : t->ch) {
            compress(u);
        }
    }
};
```

10.6 Prefix Function

`P[:i]` の接尾辞でもある最長の proper prefix

```
template <typename T>
std::vector<int> prefix_function(const std::vector<T>& s) {
    const int n = s.size();
    std::vector<int> ret(n);
    int len = 0;
    for (int i = 1; i < n; ++i) {
        if (s[i] == s[len]) {
            ++len;
            ret[i] = len;
        } else {
            if (len != 0) {
                len = ret[len - 1];
                --i;
            } else {
                ret[i] = 0;
            }
        }
    }
    return ret;
}
```

10.7 Aho-Corasick Algorithm

Aho-Corasick 法は，入力文字列に対して複数のパターンを高速にマッチするアルゴリズムである．

- `void insert(string p)`
  - パターン p を挿入する
  - 時間計算量: $O(|p|)$
- `void build()`
  - オートマトンを構築する
  - 時間計算量: $O(\sum |p|)$
- `int get\_next(int i, char c)`
  - 状態 i にいるときに文字 c が出現したときの遷移先の状態を返す
- `long long count(string s)`
  - 文字列 s に対する各パターンのマッチ回数の合計を返す
  - 時間計算量: $O(|s| + \sum |p|)$
- `vector<pair<int, int>> match(string s)`
  - 文字列 s に対する各パターンのマッチ位置を返す

- 時間計算量: $O(|s| + \sum |p|)$

```cpp
class AhoCorasick {
public:
    struct Node {
        std::map<char, int> ch;
        std::vector<int> accept;
        int link = -1;
        int cnt = 0;

        Node() = default;
    };

    std::vector<Node> states;
    std::map<int, int> accept_state;

    explicit AhoCorasick() : states(1) {}

    void insert(const std::string& s, int id = -1) {
        int i = 0;
        for (char c : s) {
            if (!states[i].ch.count(c)) {
                states[i].ch[c] = states.size();
                states.emplace_back();
            }
            i = states[i].ch[c];
        }
        ++states[i].cnt;
        states[i].accept.push_back(id);
        accept_state[id] = i;
    }

    void clear() {
        states.clear();
        states.emplace_back();
    }

    int get_next(int i, char c) const {
        while (i != -1 && !states[i].ch.count(c)) i = states[i].link;
        return i != -1 ? states[i].ch.at(c) : 0;
    }

    void build() {
        std::queue<int> que;
        que.push(0);
        while (!que.empty()) {
            int i = que.front();
            que.pop();

            for (auto [c, j] : states[i].ch) {
                states[j].link = get_next(states[i].link, c);
                states[j].cnt += states[states[j].link].cnt;

                auto& a = states[j].accept;
                auto& b = states[states[j].link].accept;
                std::vector<int> accept;
                std::set_union(a.begin(), a.end(), b.begin(), b.end(), std::back_inserter(
                    accept));
                a = accept;

                que.push(j);
            }
        }
    }

    long long count(const std::string& str) const {
        long long ret = 0;
        int i = 0;
        for (auto c : str) {
            i = get_next(i, c);
            ret += states[i].cnt;
        }
        return ret;
    }

    // list of (id, index)
    std::vector<std::pair<int, int>> match(const std::string& str) const {
        std::vector<std::pair<int, int>> ret;
        int i = 0;
        for (int k = 0; k < (int) str.size(); ++k) {
            char c = str[k];
            i = get_next(i, c);
            for (auto id : states[i].accept) {
                ret.emplace_back(id, k);
            }
        }
        return ret;
    }
};
```

11 tree

11.1 Lowest Common Ancestor

- `LCA(vector<vector<int>> G, int root)`
  - 前計算をする
  - 時間計算量: $O(n \log n)$
- `int query(int u, int v)`
  - 頂点 u と頂点 v の最小共通祖先を返す
  - 時間計算量: $O(\log n)$
- `int dist(int u, int v)`
  - uv 間の距離を計算する
  - 時間計算量: $O(\log n)$
- `int parent(int v, int k)`
  - 頂点 v の k 個上の頂点を求める
- `int jump(int u, int v, int k)`
  - uv パス上の k 番目の頂点を返す．$k = 0$ のとき u を，$k > dist(u, v)$ のとき -1 を返す．
  - 時間計算量: $O(\log n)$

```cpp
class LCA {
public:
    LCA() = default;
    LCA(const std::vector<std::vector<int>>& G, int root) : G(G), LOG(32 - __builtin_clz(G
        .size())), depth(G.size()) {
```

```
        int V = G.size();
        table.assign(LOG, std::vector<int>(V, -1));

        dfs(root, -1, 0);

        for (int k = 0; k < LOG - 1; ++k) {
            for (int v = 0; v < V; ++v) {
                if (table[k][v] >= 0) {
                    table[k + 1][v] = table[k][table[k][v]];
                }
            }
        }
    }

    int query(int u, int v) const {
        if (depth[u] > depth[v]) std::swap(u, v);

        // go up to the same depth
        for (int k = 0; k < LOG; ++k) {
            if ((depth[v] - depth[u]) >> k & 1) {
                v = table[k][v];
            }
        }
        if (u == v) return u;

        for (int k = LOG - 1; k >= 0; --k) {
            if (table[k][u] != table[k][v]) {
                u = table[k][u];
                v = table[k][v];
            }
        }
        return table[0][u];
    }

    int dist(int u, int v) const {
        return depth[u] + depth[v] - 2 * depth[query(u, v)];
    }

    int parent(int v, int k) const {
        for (int i = LOG - 1; i >= 0; --i) {
            if (k >= (1 << i)) {
                v = table[i][v];
                k -= 1 << i;
            }
        }
        return v;
    }

    int jump(int u, int v, int k) const {
        int l = query(u, v);
        int du = depth[u] - depth[l];
        int dv = depth[v] - depth[l];
        if (du + dv < k) return -1;
        if (k < du) return parent(u, k);
        return parent(v, du + dv - k);
    }


protected:
    const std::vector<std::vector<int>>& G;
    const int LOG;
    std::vector<std::vector<int>> table;
    std::vector<int> depth;

    void dfs(int v, int p, int d) {
        table[0][v] = p;
        depth[v] = d;
        for (int c : G[v]) {
            if (c != p) dfs(c, v, d + 1);
        }
    }
};
```

11.2 Tree Isomorphism

やり方だけ説明する

- ハッシュ
  - 確率的 $O(n)$
  - 部分木 v の深さを d とし，d に対応する乱数 R_d を取る．v の子のハッシュを $h_1, \ldots, h_k$ として，v のハッシュ h を $h = \prod(h_i + R)$ とする．
- AHU algorithm
  - 決定的 $O(n \log n)$
  - 子のラベルの列をソートして，その列に対して新しいラベルを付与することを繰り返す
  - 実装が楽だし決定的で嬉しいので TL が厳しくないならこっちが良さそう

根付き木でない場合は，中心を根にしてやればよい

11.3 Centroid Decomposition

- `tuple<vector<int>, vector<int>, vector<int>> centroid\_decomposition(vector<vector<i G)`
- 木 G の隣接リストが与えられたとき，3 つ組 (`level`, `sz`, `par`) を返す．
  - `level`: G を重心分解したときの各頂点のレベル (何回目の分割でそれが重心となるか)
  - `sz`: 各頂点が重心となるときにそれが含まれる部分木のサイズ
  - `par`: 各頂点が重心となる直前に属していた部分木の重心
- 時間計算量: $O(n \log n)$

```
std::tuple<std::vector<int>, std::vector<int>, std::vector<int>> centroid_decomposition(
    const std::vector<std::vector<int>>& G) {
    int N = G.size();
    std::vector<int> sz(N), level(N, -1), sz_comp(N), par(N);

    auto dfs_size = [&](auto& dfs_size, int v, int p) -> int {
        sz[v] = 1;
        for (int c : G[v]) {
            if (c != p && level[c] == -1) sz[v] += dfs_size(dfs_size, c, v);
        }
        return sz[v];
    };

    auto dfs_centroid = [&](auto& dfs_centroid, int v, int p, int n) -> int {
        for (int c : G[v]) {
            if (c != p && level[c] == -1 && sz[c] > n / 2) return dfs_centroid(
                dfs_centroid, c, v, n);
        }
```

```
        return v;
    };

    auto decompose = [&](auto& decompose, int v, int k, int p) -> void {
        int n = dfs_size(dfs_size, v, -1);
        int s = dfs_centroid(dfs_centroid, v, -1, n);
        level[s] = k;
        sz_comp[s] = n;
        par[s] = p;
        for (int c : G[s]) {
            if (level[c] == -1) decompose(decompose, c, k + 1, s);
        }
    };

    decompose(decompose, 0, 0, -1);
    return {level, sz_comp, par};
}
```

11.4 Heavy-Light Decomposition

木上のパスクエリ

update および fold の時間計算量を $f(n)$ とする

- HLD(vector<vector<int>> G, bool edge)
  - 木 G を HL 分解する．edge == true ならクエリは辺に対して実行される．
  - 時間計算量: $O(n)$
- void update(int v, T x, F update)
  - 頂点 v に対して update(x) を実行する
  - 時間計算量: $O(f(n)\log n)$
- void update(int u, int v, T x, F update)
  - uv パス上の頂点/辺に対して update(x) を実行する．
  - 時間計算量: $O(f(n)\log n)$
- T path_fold(int u, int v, F fold)
  - uv パス上の頂点/辺に対して fold() を実行する．
  - 時間計算量: $O(f(n)\log n)$
- T path_fold(int u, int v, F fold, Flip flip)
  - uv パス上の頂点/辺に対して fold() を実行する．値が非可換なら，左から積をとったときと右から積をとったときの値を入れ替える flip 関数を与える必要がある．
  - 時間計算量: $O(f(n)\log n)$
- T subtree_fold(int v, F fold)
  - 頂点 v を根とする部分木の頂点/辺に対して fold() を実行する．
  - 時間計算量: $O(f(n))$
- int lca(int u, int v)
  - 頂点 u と頂点 v の最小共通祖先を返す
  - 時間計算量: $O(\log n)$
- int dist(int u, int v)
  - uv 間の距離を計算する
  - 時間計算量: $O(\log n)$

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

public:
    HLD() = default;
    HLD(const std::vector<std::vector<int>>& G, bool edge)
        : G(G), size(G.size()), depth(G.size()), par(G.size(), -1),
          in(G.size()), out(G.size()), head(G.size()), heavy(G.size(), -1), edge(edge) {
        dfs(0);
        decompose(0, 0);
    }

    template <typename F>
    void update(int v, const T& x, const F& f) const {
        f(in[v], x);
    }

    template <typename F>
    void update_edge(int u, int v, const T& x, const F& f) const {
        if (in[u] > in[v]) std::swap(u, v);
        f(in[v], x);
    }

    template <typename E, typename F>
    void update(int u, int v, const E& x, const F& f) const {
        while (head[u] != head[v]) {
            if (in[head[u]] > in[head[v]]) std::swap(u, v);
            f(in[head[v]], in[v] + 1, x);
            v = par[head[v]];
        }
        if (in[u] > in[v]) std::swap(u, v);
        f(in[u] + edge, in[v] + 1, x);
    }

    template <typename F, typename Flip>
    T path_fold(int u, int v, const F& f, const Flip& flip) const {
        bool flipped = false;
        T resu = M::id(), resv = M::id();
        while (head[u] != head[v]) {
            if (in[head[u]] > in[head[v]]) {
                std::swap(u, v);
                std::swap(resu, resv);
                flipped ^= true;
            }
            T val = f(in[head[v]], in[v] + 1);
            resv = M::op(val, resv);
            v = par[head[v]];
        }
        if (in[u] > in[v]) {
            std::swap(u, v);
            std::swap(resu, resv);
            flipped ^= true;
        }
        T val = f(in[u] + edge, in[v] + 1);
        resv = M::op(val, resv);
        resv = M::op(flip(resu), resv);
        if (flipped) {
            resv = flip(resv);
        }
        return resv;
```

```
    }

    template <typename F>
    T path_fold(int u, int v, const F& f) const {
        path_fold(u, v, f, [&](auto& v) { return v; });
    }

    template <typename F>
    T subtree_fold(int v, const F& f) const {
        return f(in[v] + edge, out[v]);
    }

    int lca(int u, int v) const {
        while (true) {
            if (in[u] > in[v]) std::swap(u, v);
            if (head[u] == head[v]) return u;
            v = par[head[v]];
        }
    }

    int dist(int u, int v) const {
        return depth[u] + depth[v] - 2 * depth[lca(u, v)];
    }

private:
    std::vector<std::vector<int>> G;
    std::vector<int> size, depth, par, in, out, head, heavy;
    bool edge;
    int cur_pos = 0;

    void dfs(int v) {
        size[v] = 1;
        int max_size = 0;
        for (int c : G[v]) {
            if (c == par[v]) continue;
            par[c] = v;
            depth[c] = depth[v] + 1;
            dfs(c);
            size[v] += size[c];
            if (size[c] > max_size) {
                max_size = size[c];
                heavy[v] = c;
            }
        }
    }

    void decompose(int v, int h) {
        head[v] = h;
        in[v] = cur_pos++;
        if (heavy[v] != -1) decompose(heavy[v], h);
        for (int c : G[v]) {
            if (c != par[v] && c != heavy[v]) decompose(c, c);
        }
        out[v] = cur_pos;
    }
};
```

11.5 Rerooting

全方位木 DP

DP は $dp_v = g(f(dp_{c_1}, e_1) * \cdots * f(dp_{c_k}, e_k), v)$ という形の遷移で表されるとする.

Template Parameters

- `M`
  - 可換モノイド
- `Cost`
  - 辺のコストの型
- `T apply\_edge(T a, int s, int t, Cost c)`
  - 遷移の f
- `T apply\_vertex(T x, int v)`
  - 遷移の g

Operations

- `Rerooting(int n)`
  - 頂点数 n で木を初期化する
  - 時間計算量: $O(n)$
- `void add\_edge(int u, int v, Cost c)`
  - 頂点 uv 間にコスト c の辺を張る
  - 時間計算量: $O(1)$
- `vector<T> run()`
  - 各頂点を根としたときの木 DP の値を求める
  - 時間計算量: $O(n)$

```
template <typename M,
          typename Cost,
          typename M::T (*apply_edge)(typename M::T, int, int, Cost),
          typename M::T (*apply_vertex)(typename M::T, int)>
class Rerooting {
    using T = typename M::T;

public:
    explicit Rerooting(int n) : G(n) {}

    void add_edge(int u, int v, Cost c) {
        G[u].emplace_back(v, c);
        G[v].emplace_back(u, c);
    }

    std::vector<T> run() {
        dp_sub.resize(G.size(), M::id());
        dp_all.resize(G.size());
        dfs_sub(0, -1);
        dfs_all(0, -1, M::id());
        return dp_all;
    }

private:
    std::vector<std::vector<std::pair<int, Cost>>> G;
    std::vector<T> dp_sub, dp_all;

    void dfs_sub(int v, int p) {
        for (auto [c, cost] : G[v]) {
            if (c == p) continue;
            dfs_sub(c, v);
```

```
            dp_sub[v] = M::op(dp_sub[v], apply_edge(dp_sub[c], v, c, cost));
        }
        dp_sub[v] = apply_vertex(dp_sub[v], v);
    }

    void dfs_all(int v, int p, const T& val) {
        std::vector<T> ds = {val};
        for (auto [c, cost] : G[v]) {
            if (c == p) continue;
            ds.push_back(apply_edge(dp_sub[c], v, c, cost));
        }
        int n = ds.size();
        std::vector<T> head(n + 1, M::id()), tail(n + 1, M::id());
        for (int i = 0; i < n; ++i) head[i+1] = M::op(head[i], ds[i]);
        for (int i = n - 1; i >= 0; --i) tail[i] = M::op(ds[i], tail[i+1]);
        dp_all[v] = apply_vertex(head[n], v);
        int k = 1;
        for (auto [c, cost] : G[v]) {
            if (c == p) continue;
            dfs_all(c, v, apply_edge(apply_vertex(M::op(head[k], tail[k+1]), v), c, v,
                cost));
            ++k;
        }
    }
};
```

11.6 Link/Cut Tree

森の辺の追加，削除，根の変更，頂点の値の更新，パスクエリ

- `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 $O(\log n)$

```
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);
                    }
                }
            }
        }
    }
```

```
181 };
```