目次

1 テンプレート

```cpp
#include <bits/stdc++.h>
using namespace std;

using ll = long long;
using P = pair<int, int>;
constexpr ll MOD = 1000000007;
constexpr int INF = 1 << 30;
#define REP(i, n) for (int i = 0, i_len = (n); i < i_len; i++)
#define ALL(v) (v).begin(), (v).end()
```

2 グラフ

2.1 Lowest Common Ancestor

木の最近共通祖先 (Lowest Common Ancestor: LCA) をダブリングにより求める。前計算：時間・空間ともに $O(V \log V)$、クエリあたり：$O(\log V)$ である。

- LCA(G, r)：木 G と 根 r から、前計算する。
- int query(u, v)：LCA(u, v) を求める.
- bool is_on_path(u, v, a)：頂点 a が 頂点 u, v を結ぶパス上に存在するかどうか

```cpp
struct LCA {
    vector<vector<int>> parent;
    vector<int> depth;
    LCA() {}
    LCA(const vector<vector<int>>& G, int r = 0) { init(G, r); }

    void init(const vector<vector<int>>& G, int r = 0) {
        int V = (int)G.size();
        int h = 1;
        while ((1 << h) < V) ++h;
        parent.assign(h, vector<int>(V, -1));
        depth.assign(V, -1);
        dfs(G, r, -1, 0);
        for (int i = 0; i + 1 < (int)parent.size(); ++i) {
            for (int v = 0; v < V; ++v) {
                if (parent[i][v] != -1) {
                    parent[i + 1][v] = parent[i][parent[i][v]];
                }
            }
        }
    }

    void dfs(const vector<vector<int>>& G, int v, int p, int d) {
        parent[0][v] = p;
        depth[v] = d;
        for (auto e : G[v])
            if (e != p) dfs(G, e, v, d + 1);
    }

    int query(int u, int v) {
        if (depth[u] > depth[v]) swap(u, v);
        for (int i = 0; i < (int)parent.size(); ++i) {
            if ((depth[v] - depth[u]) & (1 << i)) v = parent[i][v];
        }
        if (u == v) return u;
        for (int i = (int)parent.size() - 1; i >= 0; --i) {
            if (parent[i][u] != parent[i][v]) {
                u = parent[i][u];
                v = parent[i][v];
            }
        }
        return parent[0][u];
    }

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

    bool is_on_path(int u, int v, int x) {
        return dist(u, x) + dist(x, v) == dist(u, v);
    }
};
```

2.2 Strongly Connected Components

有向グラフを強連結成分分解する。計算量は $O(V + E)$

- SCC(int V)：コンストラクタ. V 頂点 E 辺の有向グラフを作る.
- void add_edge(int from, int to)：頂点 from から 頂点 to へ有向辺を足す.
- pair<int, vector<int>> scc_ids()：
  (SCC の個数, SCC の id) を返す. id[v] := 頂点 v が属する連結成分の番号
- vector<vector<int>> graph.scc()：
  次の条件を満たす「頂点のリスト」のリストを返す.
  - 全ての頂点がちょうど 1 つずつ、どれかのリストに含まれる.
  - 内側のリストと強連結成分が一対一に対応する. リスト内の順序は未定義.
  - リストはトポロジカルソートされている.

```cpp
struct SCC {
    int _n;
    struct edge {
        int to;
    };
    vector<pair<int, edge>> edges;

    template <class E>
    struct csr {
        vector<int> start;
        vector<E> elist;
        csr(int n, const vector<pair<int, E>>& edges)
            : start(n + 1), elist(edges.size()) {
            for (auto e : edges) start[e.first + 1]++;
            for (int i = 1; i <= n; i++) start[i] += start[i - 1];
            auto counter = start;
            for (auto e : edges) elist[counter[e.first]++] = e.second;
        }
```

```cpp
    };

    SCC(int n) : _n(n) {}
    SCC() : _n(0) {}

    int num_vertices() { return _n; }

    void add_edge(int from, int to) {
        edges.push_back({from, {to}});
    }

    // return pair of (# of scc, scc id)
    pair<int, vector<int>> scc_ids() {
        auto g = csr<edge>(_n, edges);
        int now_ord = 0, group_num = 0;
        vector<int> visited, low(_n), ord(_n, -1), ids(_n);
        visited.reserve(_n);
        auto dfs = [&](auto self, int v) -> void {
            low[v] = ord[v] = now_ord++;
            visited.push_back(v);
            for (int i = g.start[v]; i < g.start[v + 1]; i++) {
                auto to = g.elist[i].to;
                if (ord[to] == -1) {
                    self(self, to);
                    low[v] = min(low[v], low[to]);
                } else {
                    low[v] = min(low[v], ord[to]);
                }
            }
            if (low[v] == ord[v]) {
                while (true) {
                    int u = visited.back();
                    visited.pop_back();
                    ord[u] = _n;
                    ids[u] = group_num;
                    if (u == v) break;
                }
                group_num++;
            }
        };
        for (int i = 0; i < _n; i++)
            if (ord[i] == -1) dfs(dfs, i);
        for (auto& x : ids) x = group_num - 1 - x;
        return {group_num, ids};
    }

    vector<vector<int>> scc() {
        auto ids = scc_ids();
        int group_num = ids.first;
        vector<int> counts(group_num);
        for (auto x : ids.second) counts[x]++;
        vector<vector<int>> groups(ids.first);
        for (int i = 0; i < group_num; i++) groups[i].reserve(counts[i]);
        for (int i = 0; i < _n; i++) groups[ids.second[i]].push_back(i);
        return groups;
    }
};
```

2.3 2-SAT

n 変数 $x_0, x_1, \ldots, x_{n-1}$ に関して、

$$(x_i = f) \vee (x_j = g)$$

というクローズを足し、これを全て満たす変数の割り当てがあるか、という問題を解く。

- two_sat(n) : n 変数の 2-SAT を作る。$O(n)$
- void add_clause(i, f, j, g) : クローズ $(x_i = f) \vee (x_j = g)$ を足す。ならし $O(1)$
- bool satisfiable() :
  (割り当てが存在する ? true : false). クローズの個数を m として $O(n + m)$
- vector<bool> answer() :
  最後に呼んだ satisfiable のクローズを満たす割り当てを返す。satisfiable を呼ぶ前や、割り当てがない場合、中身が未定義の長さ n の vector を返す。$O(n)$

```cpp
struct TwoSAT {
  public:
    TwoSAT() : _n(0), scc(0) {}
    TwoSAT(int n) : _n(n), _answer(n), scc(2 * n) {}

    void add_clause(int i, bool f, int j, bool g) {
        scc.add_edge(2 * i + (f ? 0 : 1), 2 * j + (g ? 1 : 0));
        scc.add_edge(2 * j + (g ? 0 : 1), 2 * i + (f ? 1 : 0));
    }

    bool satisfiable() {
        auto id = scc.scc_ids().second;
        for (int i = 0; i < _n; i++) {
            if (id[2 * i] == id[2 * i + 1]) return false;
            _answer[i] = id[2 * i] < id[2 * i + 1];
        }
        return true;
    }

    vector<bool> answer() { return _answer; }

  private:
    int _n;
    vector<bool> _answer;
    SCC scc;  // 強連結成分分解を用いる
};
```

3 フロー

3.1 dinic

最大流問題を解くアルゴリズム。計算量は $O(VE^2)$ だが実用上かなり高速なことが多い。

- Dinic flow(V): 構造体の宣言。V は頂点数。
- flow.add_edge(u, v, c): u → v に容量 c の辺を追加する

- flow.max_flow(s, t): s → t の最大流を返す

```cpp
// Dinic法 O(V^2E)
struct Dinic {
    int V;                                  // 頂点数
    vector<vector<vector<long long>>> graph;  // グラフ
    vector<int> dis;                        // 始点からの距離
    vector<int> next;                       // 次に処理する頂点のメモ
    Dinic(int v) : V(v) { graph.resize(V); }
    void add_edge(int from, int to, long long capacity) {
        graph[from].push_back({to, capacity, (int)graph[to].size()});
        graph[to].push_back({from, 0, (int)graph[from].size() - 1});
    }
    void bfs(int s) {
        dis.assign(V, -1);
        dis[s] = 0;
        deque<int> pos = {s};
        while (pos.size()) {
            int now = pos[0];
            pos.pop_front();
            for (auto& to : graph[now]) {
                if (dis[to[0]] < 0 and to[1] > 0) {
                    dis[to[0]] = dis[now] + 1;
                    pos.emplace_back(to[0]);
                }
            }
        }
    }
    long long dfs(int v, int t, long long f) {
        if (v == t) return f;
        for (int& i = next[v]; i < graph[v].size(); i++) {
            int to = graph[v][i][0];
            long long& cap = graph[v][i][1];
            int rev = graph[v][i][2];
            if (cap > 0 and dis[v] < dis[to]) {
                long long d = dfs(to, t, min(f, cap));
                if (d > 0) {
                    cap -= d;
                    graph[to][rev][1] += d;
                    return d;
                }
            }
        }
        return 0;
    }
    long long max_flow(int s, int t) {
        long long flow = 0;
        while (1) {
            bfs(s);
            if (dis[t] < 0) return flow;
            next.assign(V, 0);
            long long f;
            while ((f = dfs(s, t, LLONG_MAX)) > 0) flow += f;
        }
    }
};
```

3.2 最小費用流

最小費用流問題を解くアルゴリズム。計算量は $O(FE\log V)$

- MinCostFlow flow(V): 構造体の宣言。V は頂点数。
- flow.add_edge(u, v, c, d): u → v に容量 c, コスト d の辺を追加する
- flow.min_cost_flow(s, t, F): s → t に流量 F を流すときの最小コストを返す。流せない場合は-1 を返す。

```cpp
// 最小費用流 O(FElogV)
struct MinCostFlow {
    int V;
    vector<vector<vector<long long>>> g;  // g[from] = {{to, 容量, コスト, 逆辺のindex}
        ... }
    vector<long long> h, dis;             // ポテンシャル, 最短距離
    vector<int> prevv, preve;             // 直前の頂点, 辺

    MinCostFlow(int v) : V(v), g(v), dis(v), prevv(v), preve(v) {
    }

    void add_edge(int u, int v, long long c, long long d) {
        g[u].push_back({v, c, d, (int)g[v].size()});
        g[v].push_back({u, 0, -d, (int)g[u].size() - 1});
    }

    long long min_cost_flow(int s, int t, long long f) {
        long long res = 0;
        h.assign(V, 0);
        using Q = pair<long long, int>;
        while (f > 0) {
            priority_queue<Q, vector<Q>, greater<Q>> que;
            dis.assign(V, LLONG_MAX);
            dis[s] = 0;
            que.push({0, s});
            while (que.size()) {
                Q q = que.top();
                int v = q.second;
                que.pop();
                if (dis[v] < q.first) continue;
                for (int i = 0; i < g[v].size(); i++) {
                    auto edge = g[v][i];
                    int to = edge[0];
                    long long cap = edge[1], cost = edge[2];
                    if (cap > 0 and dis[to] > dis[v] + cost + h[v] - h[to]) {
                        dis[to] = dis[v] + cost + h[v] - h[to];
                        prevv[to] = v;
                        preve[to] = i;
                        que.push({dis[to], to});
                    }
                }
            }
            if (dis[t] == LLONG_MAX) return -1;
            for (int i = 0; i < V; i++) h[i] += dis[i];
            long long d = f;
            for (int i = t; i != s; i = prevv[i]) d = min(d, g[prevv[i]][preve[i]][1]);
            f -= d;
            res += d * h[t];
            for (int i = t; i != s; i = prevv[i]) {
                auto& edge = g[prevv[i]][preve[i]];
                edge[1] -= d;
                g[i][edge[3]][1] += d;
            }
```

```
        }
        return res;
    }
};
```

4 数学

4.1 拡張ユークリッド互除法

2 つの整数 a, b について $ax + by = \gcd(a, b)$ の整数解 (x, y) を求めるアルゴリズム。計算量は $O(\log \min(a, b))$。また、追加で以下の条件を満たす。

- すべての整数解 (x, y) のうち、$|x| + |y|$ が最小である解を求める。
- $\gcd(a, b) \neq \min(a, b)$ のとき $|x| \leq \left|\frac{b}{2\gcd(a,b)}\right|, |y| \leq \left|\frac{a}{2\gcd(a,b)}\right|$

使用方法

- extgcd(a, b, x, y) : x, y に解を格納する。返り値として $\gcd(a, b)$ を返す。

```cpp
template <class T>
T extgcd(T a, T b, T& x, T& y) {
    if (b != 0) {
        T d = extgcd(b, a % b, y, x);
        y -= (a / b) * x;
        return d;
    } else {
        x = 1;
        y = 0;
        return a;
    }
}
```

4.2 modint

自動的に mod をとる構造体。mod が問題で固定かつ素数であるとき使用できる。

using mint = modint<1000000007>; 等のように定義して使用するのが推奨。

```cpp
template <int64_t Modulus>
struct Modint {
    using mint = Modint;
    long long v;
    Modint() : v(0) {}
    Modint(long long x) {
        x %= Modulus;
        if (x < 0) x += Modulus;
        v = x;
    }
    const long long& val() const { return v; }
    // 代入演算子
    mint& operator+=(const mint rhs) {
        v += rhs.v;
        if (v >= Modulus) v -= Modulus;
        return *this;
    }
    mint& operator-=(const mint rhs) {
        if (v < rhs.v) v += Modulus;
        v -= rhs.v;
        return *this;
    }
    mint& operator*=(const mint rhs) {
        v = v * rhs.v % Modulus;
        return *this;
    }
    mint& operator/=(mint rhs) { return *this = *this * rhs.inv(); }
    // 累乗, 逆元
    mint pow(long long n) const {
        mint x = *this, res = 1;
        while (n) {
            if (n & 1) res *= x;
            x *= x;
            n >>= 1;
        }
        return res;
    }
    mint inv() const { return pow(Modulus - 2); }
    // 算術演算子
    mint operator+(const mint rhs) const { return mint(*this) += rhs; }
    mint operator-(const mint rhs) const { return mint(*this) -= rhs; }
    mint operator*(const mint rhs) const { return mint(*this) *= rhs; }
    mint operator/(const mint rhs) const { return mint(*this) /= rhs; }
    mint operator-() const { return mint() - *this; }  // 単項
    // 入出力ストリーム
    friend ostream& operator<<(ostream& os, const mint& p) { return os << p.v; }
    friend istream& operator>>(istream& is, mint& p) {
        long long t;
        is >> t;
        p = mint(t);
        return (is);
    }
};
```

4.3 FFT

- encode(a): 整数型の配列 a を std::complex 型に変換。
- decode(a):std::complex 型の配列 a を 64bit 整数型に変換。配列の要素毎に実部を丸めて整数に変換している。
- FFT(a):std::complex 型で長さ n の配列 a をフーリエ変換する。整数型の配列は引数にとれないため、encode(a),decode(a) 等で適宜変換を行うこと。
- convolution(a,b): 長さ n の整数列 a, 長さ m の整数列 b の畳み込みを $O((n+m)\log(n+m))$ で計算する。畳み込み後の配列の要素がすべて double に収まる必要がある。

```cpp
// 整数配列を複素数へ

vector<complex<double>> encode(vector<long long>& a) {
    int N = a.size();
    vector<complex<double>> ret(N);
    for (int i = 0; i < N; i++) {
        ret[i] = complex<double>(a[i], 0);
    }
    return ret;
```

```
}

// 複素数配列を整数へ
vector<long long> decode(vector<complex<double>>& a) {
    int N = a.size();
    vector<long long> ret(N);
    for (int i = 0; i < N; i++) {
        ret[i] = (long long)round(a[i].real());
    }
    return ret;
}

// 非再帰
void FFT(vector<complex<double>>& a, int reverse = false) {
    int n = a.size();
    int h = 0;
    for (int i = 0; 1 << i < n; i++) h++;
    for (int i = 0; i < n; i++) {
        int j = 0;
        for (int k = 0; k < h; k++) j |= (i >> k & 1) << (h - 1 - k);
        if (i < j) swap(a[i], a[j]);
    }
    for (int b = 1; b < n; b *= 2) {
        for (int j = 0; j < b; j++) {
            double p2 = 2 * acos(-1);
            if (reverse) p2 *= -1;
            complex<double> w = exp(complex<double>(0, p2 * j / (double)(2 * b)));
            for (int k = 0; k < n; k += b * 2) {
                complex<double> s = a[j + k];
                complex<double> t = a[j + k + b] * w;
                a[j + k] = s + t;
                a[j + k + b] = s - t;
            }
        }
    }
    if (reverse)
        for (int i = 0; i < n; i++) a[i] /= (double)n;
    return;
}

vector<long long> convolution(vector<long long>& a, vector<long long>& b) {
    vector<complex<double>> A = encode(a), B = encode(b);
    int s = (int)a.size() + (int)b.size() - 1;
    int t = 1;
    while (t < s) t *= 2;
    A.resize(t);
    B.resize(t);
    FFT(A, 0);
    FFT(B, 0);
    for (int i = 0; i < t; i++) {
        A[i] *= B[i];
    }
    FFT(A, 1);
    A.resize(s);
    return decode(A);
}
```

4.4 高速ゼータ変換・メビウス変換

部分集合に対するゼータ変換・メビウス変換を集合の要素数を n として $O(n2^n)$ で行うアルゴリズム。

bitwise AND 畳み込みや bitwise OR 畳み込みを高速化できる。

- subset_zeta(f, n, inv)：長さ 2^n の配列 f の下位集合ゼータ変換 $F[S] = \sum_{X \subseteq S} f(X)$ を求める。
- supset_zeta(f, n, inv)：長さ 2^n の配列 f の上位集合ゼータ変換 $F[S] = \sum_{X \supseteq S} f(X)$ を求める。

inv=true のとき逆変換としてメビウス変換を行い、F から f を求める。

```
template <class T>
vector<T> subset_zeta(vector<T> f, int n, bool inv = false) {
    for (int i = 0; i < n; i++) {
        for (int S = 0; S < (1 << n); S++) {
            if ((S & (1 << i)) != 0) {  // if i in S
                if (!inv) {
                    f[S] += f[S ^ (1 << i)];
                } else {
                    f[S] -= f[S ^ (1 << i)];
                }
            }
        }
    }
    return f;
}

template <class T>
vector<T> supset_zeta(vector<T> f, int n, bool inv = false) {
    for (int i = 0; i < n; i++) {
        for (int S = 0; S < (1 << n); S++) {
            if ((S & (1 << i)) == 0) {  // if i not in S
                if (!inv) {
                    f[S] += f[S ^ (1 << i)];
                } else {
                    f[S] -= f[S ^ (1 << i)];
                }
            }
        }
    }
    return f;
}
```

4.5 高速アダマール変換

クロネッカー冪行列をベクトルに掛ける計算を高速に行うアルゴリズム。

配列の長さは 2^n であるとする。計算量は $O(n2^n)$ である。

- fwht(a, inv)：配列 a を高速にアダマール変換する。inv=true のとき逆変換する。
- xor_convolution(a, b)：bitwise XOR 畳み込み後の配列 c を返す。
- and_convolution(a, b)：bitwise AND 畳み込み後の配列 c を返す。

- or_convolution(a, b)：bitwise OR 畳み込み後の配列 c を返す。

```
namespace Kronecker {

template <class T>
void mul(vector<T>& x, T a, T b, T c, T d) {
    int n = x.size();
    for (int j = 1; j < n; j <<= 1) {
        for (int i = 0; i < n; i++) {
            if ((i & j) == 0) {
                T s = a * x[i] + b * x[i + j];
                T t = c * x[i] + d * x[i + j];
                x[i] = s;
                x[i + j] = t;
            }
        }
    }
}

template <class T>
void fwht(vector<T>& a, bool inv) {
    mul(a, T(1), T(1), T(1), T(-1));
    if (inv) {
        for (T& x : a) x /= T(a.size());
    }
}

template <class T>
vector<T> xor_convolution(vector<T>& a, vector<T>& b) {
    fwht(a, false);
    fwht(b, false);
    int n = a.size();
    vector<T> c(n);
    for (int i = 0; i < n; i++) c[i] = a[i] * b[i];
    fwht(c, true);
    return c;
}

template <class T>
vector<T> and_convolution(vector<T>& a, vector<T>& b) {
    mul(a, T(1), T(1), T(0), T(1));
    mul(b, T(1), T(1), T(0), T(1));
    int n = a.size();
    vector<T> c(n);
    for (int i = 0; i < n; i++) c[i] = a[i] * b[i];
    mul(c, T(1), T(-1), T(0), T(1));
    return c;
}

template <class T>
vector<T> or_convolution(vector<T>& a, vector<T>& b) {
    mul(a, T(1), T(0), T(1), T(1));
    mul(b, T(1), T(0), T(1), T(1));
    int n = a.size();
    vector<T> c(n);
    for (int i = 0; i < n; i++) c[i] = a[i] * b[i];
    mul(c, T(1), T(0), T(-1), T(1));
    return c;
}

}  // namespace Kronecker
```

5 データ構造

5.1 Fenwick Tree

一点更新・区間和取得のクエリを $O(\log N)$ で処理するデータ構造。

- add(i, x)：点更新 $a[i] \leftarrow a[i] + x$
- sum(l, r)：区間和取得 $a[l] + a[l+1] + \cdots + a[r-1]$

```
template <class T> struct FenwickTree {
    int n;
    vector<T> data;
    FenwickTree() : n(0) {}
    FenwickTree(int n) : n(n), data(n, 0) {}

    // a[i] += x
    void add(int i, T x) {
        for (int p = i + 1; p <= n; p += p & -p) data[p - 1] += x;
    }
    // [0, r)
    T sum(int r) {
        T s(0);
        for (int p = r; p > 0; p -= p & -p) s += data[p - 1];
        return s;
    }
    // [l, r)
    T sum(int l, int r) {
        return sum(r) - sum(l);
    }
};
```

5.2 セグメント木

モノイドを満たすデータ S に対し使用できるデータ構造。

長さ N の S の配列に対し、要素の 1 点更新、区間クエリを $O(\log N)$ で行える。モノイド S 同士の演算の計算量が $O(f(n))$ とき、すべての計算量が $O(f(n))$ 倍になる。

```
template <class S, S (*op)(S, S), S (*e)()>
struct SegmentTree {
  private:
    int _n, size, log;
    vector<S> dat;
    void update(int k) { dat[k] = op(dat[2 * k], dat[2 * k + 1]); }

  public:
    SegmentTree() : SegmentTree(0) {}
    SegmentTree(int n) : SegmentTree(vector<S>(n, e())) {}
    SegmentTree(const vector<S>& v) : _n(int(v.size())) {
        log = 0;
        while ((1 << log) < _n) log++;
        size = 1 << log;
        dat = vector<S>(2 * size, e());
        for (int i = 0; i < _n; i++) dat[size + i] = v[i];
```

```cpp
        for (int i = size - 1; i >= 1; i--) {
            update(i);
        }
    }
    // a[p] = x
    void set(int p, S x) {
        p += size;
        dat[p] = x;
        for (int i = 1; i <= log; i++) update(p >> i);
    }
    // return a[p]
    S get(int p) const { return dat[p + size]; }
    // return op(a[l], ..., a[r-1])
    S prod(int l, int r) const {
        S sml = e(), smr = e();
        l += size;
        r += size;
        while (l < r) {
            if (l & 1) sml = op(sml, dat[l++]);
            if (r & 1) smr = op(dat[--r], smr);
            l >>= 1;
            r >>= 1;
        }
        return op(sml, smr);
    }
    S all_prod() const { return dat[1]; }

    // SegmentTree上の二分探索 (必要な場合)
    // return r, f(op(a[l], ..., a[r-1])) == true
    template <bool (*f)(S)>
    int max_right(int l) const {
        return max_right(l, [](S x) { return f(x); });
    }
    template <class F>
    int max_right(int l, F f) const {
        assert(f(e()));
        if (l == _n) return _n;
        l += size;
        S sm = e();
        do {
            while (l % 2 == 0) l >>= 1;
            if (!f(op(sm, dat[l]))) {
                while (l < size) {
                    l = (2 * l);
                    if (f(op(sm, dat[l]))) {
                        sm = op(sm, dat[l]);
                        l++;
                    }
                }
                return l - size;
            }
            sm = op(sm, dat[l]);
            l++;
        } while ((l & -l) != l);
        return _n;
    }
    // return l, f(op(a[l], ..., a[r-1])) == true
    template <bool (*f)(S)>
    int min_left(int r) const {
        return min_left(r, [](S x) { return f(x); });
    }
    template <class F>
    int min_left(int r, F f) const {
        assert(f(e()));
        if (r == 0) return 0;
        r += size;
        S sm = e();
        do {
            r--;
            while (r > 1 && (r % 2)) r >>= 1;
            if (!f(op(dat[r], sm))) {
                while (r < size) {
                    r = (2 * r + 1);
                    if (f(op(dat[r], sm))) {
                        sm = op(dat[r], sm);
                        r--;
                    }
                }
                return r + 1 - size;
            }
            sm = op(dat[r], sm);
        } while ((r & -r) != r);
        return 0;
    }
};
```

使用例

Range Minimum Query (RMQ)

```cpp
int op(int a, int b) { return min(a, b); }
int e() { return INT32_MAX; }

int n;
SegmentTree<int, op, e> seg(n);
```

5.3 遅延評価セグメント木

モノイド S と、S に対する作用素 $f : S \to S$ に対し利用できるデータ構造。

長さ N の S の配列に対し、

- 区間 $[l, r)$ の要素に一括で f を作用 $(a_i \leftarrow f(a_i), l \leq i < r)$
- 区間 $[l, r)$ の要素の総積の取得

を $O(\log N)$ で行うことができる。

```cpp
template <class S,
          S (*op)(S, S),
          S (*e)(),
          class F,
          S (*mapping)(F, S),
          F (*composition)(F, F),
          F (*id)()>
struct LazySegmentTree {
  private:
    int _n, size, log;
    vector<S> dat;
    vector<F> lz;
```

```cpp
    void update(int k) { dat[k] = op(dat[2 * k], dat[2 * k + 1]); }
    void all_apply(int k, F f) {
        dat[k] = mapping(f, dat[k]);
        if (k < size) lz[k] = composition(f, lz[k]);
    }
    void push(int k) {
        all_apply(2 * k, lz[k]);
        all_apply(2 * k + 1, lz[k]);
        lz[k] = id();
    }
    int lower_bits(int x, int k) { return x & ((1 << k) - 1); }

  public:
    LazySegmentTree() : LazySegmentTree(0) {}
    LazySegmentTree(int n) : LazySegmentTree(vector<S>(n, e())) {}
    LazySegmentTree(const vector<S>& v) : _n(int(v.size())) {
        log = 0;
        while ((1 << log) < _n) log++;
        size = 1 << log;
        dat = vector<S>(2 * size, e());
        lz = vector<F>(size, id());
        for (int i = 0; i < _n; i++) dat[size + i] = v[i];
        for (int i = size - 1; i >= 1; i--) update(i);
    }
    // a[p] = x
    void set(int p, S x) {
        p += size;
        for (int i = log; i >= 1; i--) push(p >> i);
        dat[p] = x;
        for (int i = 1; i <= log; i++) update(p >> i);
    }
    // return a[p]
    S get(int p) {
        p += size;
        for (int i = log; i >= 1; i--) push(p >> i);
        return dat[p];
    }
    // return op(a[l], ..., a[r-1])
    S prod(int l, int r) {
        if (l == r) return e();
        l += size;
        r += size;
        for (int i = log; i >= 1; i--) {
            if (lower_bits(l, i) > 0) push(l >> i);
            if (lower_bits(r, i) > 0) push((r - 1) >> i);
        }
        S sml = e(), smr = e();
        while (l < r) {
            if (l & 1) sml = op(sml, dat[l++]);
            if (r & 1) smr = op(dat[--r], smr);
            l >>= 1;
            r >>= 1;
        }
        return op(sml, smr);
    }
    S all_prod() { return dat[1]; }
    // a[p] = f(a[p])
    void apply(int p, F f) {
        p += size;
        for (int i = log; i >= 1; i--) push(p >> i);
        dat[p] = mapping(f, dat[p]);
        for (int i = 1; i <= log; i++) update(p >> i);
    }
    // i = l...r-1 について a[i] = f(a[i])
    void apply(int l, int r, F f) {
        if (l == r) return;
        l += size;
        r += size;
        for (int i = log; i >= 1; i--) {
            if (lower_bits(l, i) > 0) push(l >> i);
            if (lower_bits(r, i) > 0) push((r - 1) >> i);
        }
        int l2 = l, r2 = r;
        while (l < r) {
            if (l & 1) all_apply(l++, f);
            if (r & 1) all_apply(--r, f);
            l >>= 1;
            r >>= 1;
        }
        l = l2;
        r = r2;
        for (int i = 1; i <= log; i++) {
            if (lower_bits(l, i) > 0) update(l >> i);
            if (lower_bits(r, i) > 0) update((r - 1) >> i);
        }
    }
    // SegmentTree上の二分探索 (必要な場合)
    // return r, f(op(a[l], ..., a[r-1])) == true
    template <bool (*g)(S)>
    int max_right(int l) {
        return max_right(l, [](S x) { return g(x); });
    }
    template <class G>
    int max_right(int l, G g) {
        assert(g(e()));
        if (l == _n) return _n;
        l += size;
        for (int i = log; i >= 1; i--) push(l >> i);
        S sm = e();
        do {
            while (l % 2 == 0) l >>= 1;
            if (!g(op(sm, dat[l]))) {
                while (l < size) {
                    push(l);
                    l = (2 * l);
                    if (g(op(sm, dat[l]))) {
                        sm = op(sm, dat[l]);
                        l++;
                    }
                }
                return l - size;
            }
            sm = op(sm, dat[l]);
            l++;
        } while ((l & -l) != l);
        return _n;
    }
    // return l, f(op(a[l], ..., a[r-1])) == true
    template <bool (*g)(S)>
    int min_left(int r) {
```

```cpp
        return min_left(r, [](S x) { return g(x); });
    }
    template <class G>
    int min_left(int r, G g) {
        assert(g(e()));
        if (r == 0) return 0;
        r += size;
        for (int i = log; i >= 1; i--) push((r - 1) >> i);
        S sm = e();
        do {
            r--;
            while (r > 1 && (r % 2)) r >>= 1;
            if (!g(op(dat[r], sm))) {
                while (r < size) {
                    push(r);
                    r = (2 * r + 1);
                    if (g(op(dat[r], sm))) {
                        sm = op(dat[r], sm);
                        r--;
                    }
                }
                return r + 1 - size;
            }
            sm = op(dat[r], sm);
        } while ((r & -r) != r);
        return 0;
    }
};
```

5.3.1 使用例

Range Update & Range Minimum Query

```cpp
constexpr int INF = INT32_MAX;
constexpr int ID = INT32_MAX;

int op(int a, int b) { return min(a, b); }
int e() { return INF; }
int mapping(int f, int a) { return (f == ID ? a : f); }
int composition(int f, int g) { return (f == ID ? g : f); }
int id() { return ID; }

int n;
LazySegmentTree<int, op, e, int, mapping, composition, id> seg(n);
```

Range Add & Range Sum Query

```cpp
using S = pair<ll, ll>;

S op(S a, S b) { return S(a.first + b.first, a.second + b.second); }
S e() { return P(0, 0); }
S mapping(ll f, S x) { return S(x.first + f * x.second, x.second); }
ll composition(ll f, ll g) { return f + g; }
ll id() { return 0; }

int n;
vector<S> a(n, S(0, 1));
LazySegmentTree<S, op, e, ll, mapping, composition, id> seg(a);
```

Range Add & Range Minimum Query

```cpp
int op(int a, int b) { return min(a, b); }
int e() { return INT32_MAX; }
int mapping(int f, int x) { return x + f; }
int composition(int f, int g) { return f + g; }
int id() { return 0; }

vector<int> a(n, 0);
LazySegmentTree<int, op, e, int, mapping, composition, id> seg(a);
```

Range Update & Range Sum Query

```cpp
using S = pair<ll, ll>;
constexpr int ID = INT32_MAX;

S op(S a, S b) { return S(a.first + b.first, a.second + b.second); }
S e() { return S(0, 0); }
S mapping(int f, S x) { return (f == ID ? x : S(f * x.second, x.second)); }
int composition(int f, int g) { return (f == ID ? g : f); }
int id() { return ID; }

int n;
vector<S> a(n, S(0, 1));
LazySegmentTree<S, op, e, int, mapping, composition, id> seg(a);
```

5.4 Undo つき UnionFind

経路圧縮を行わないことで undo 可能にした UnionFind。

- RollbackUnionFind(n) : 大きさ n の UnionFind を生成する。
- find(x) : x の根を返す。 $O(\log n)$
- unite(x,y) : x と y のマージに成功したら true 失敗したら false を返す。 $O(\log n)$
- undo() : 直前の unite 操作を取り消す。 $O(1)$
- time() : 現在までに unite() が呼ばれた回数を返す。 $O(1)$
- snapshot() : 現在の UnionFind の状態を保存する。$O(1)$
- rollback(t) :
  - $t = -1$ のとき : snapshot() で保存した状態まで巻き戻す。
  - $t \neq -1$ のとき : unite() が t 回 呼び出された時の状態まで巻き戻す。

```cpp
struct RollbackUnionFind {
    vector<int> data;
    stack<pair<int, int>> history;
    int inner_snap = 0;
    RollbackUnionFind(int n) { data.resize(n, -1); }
    int find(int x) { return data[x] < 0 ? x : find(data[x]); }
    bool unite(int x, int y) {
        x = find(x), y = find(y);
        history.push({x, data[x]});
        history.push({y, data[y]});
        if (x == y) return false;
        if (-data[x] < -data[y]) swap(x, y);
        data[x] += data[y];
        data[y] = x;
```

```cpp
        return true;
    }
    int same(int x, int y) { return find(x) == find(y); }
    int size(int x) { return (-data[find(x)]); }
    void undo() {
        data[history.top().first] = history.top().second;
        history.pop();
        data[history.top().first] = history.top().second;
        history.pop();
    }
    int time() { return int(history.size() >> 1); }
    void snapshot() { inner_snap = time(); }
    void rollback(int t = -1) {
        if (t == -1) t = inner_snap;
        while (t < time()) undo();
    }
};
```

6 文字列

文字列 s の l 番目から $r-1$ 番目の要素から成る部分文字列を $s[l,r)$ と表記する

6.1 Rolling Hash

文字列（または数列）を Hash 値に変換することで、部分文字列の一致判定を $O(1)$ で行うアルゴリズム

- RollingHash(string str)：コンストラクタ。init(str) を実行する。
- void init(string str)：長さ n の文字列 str のハッシュ値を求める。計算量 $O(n)$
- bool match(rh1, l1, r1, rh2, l2, r2)：文字列 s_1, s_2 の Rolling Hash を rh1, rh2 として、$s_1[l_1, r_1), s_2[l_2, r_2)$ が一致しているか判定する

```cpp
struct RollingHash {
    static constexpr int M = 2;
    static constexpr long long MODS[M] = {999999937, 1000000007};
    static constexpr long long BASE = 9973;
    vector<long long> powb[M], hash[M];
    int n;
    RollingHash() {}
    RollingHash(const string& str) { init(str); }
    void init(const string& str) {
        n = str.size();
        for (int k = 0; k < M; k++) {
            powb[k].resize(n + 1, 1);
            hash[k].resize(n + 1, 0);
            for (int i = 0; i < n; i++) {
                hash[k][i + 1] = (hash[k][i] * BASE + str[i]) % MODS[k];
                powb[k][i + 1] = powb[k][i] * BASE % MODS[k];
            }
        }
    }
    // get hash str[l,r)
    long long get(int l, int r, int k) {
        long long res = hash[k][r] - hash[k][l] * powb[k][r - l] % MODS[k];
        if (res < 0) res += MODS[k];
        return res;
    }
};

bool match(RollingHash& rh1, int l1, int r1, RollingHash& rh2, int l2, int r2) {
    bool res = true;
    for (int k = 0; k < RollingHash::M; k++) {
        res &= rh1.get(l1, r1, k) == rh2.get(l2, r2, k);
    }
    return res;
}
```

6.2 Trie 木

文字列の集合 (辞書)$\{s_1, s_2, \ldots, s_m\}$ 内で、文字列 t、または t の prefix と一致する文字列を $O(|t|)$ で検索できる木構造。空間計算量 $O(k \cdot \sum s_i)$、k は文字の種類数。

- insert(s)：文字列 s を Trie 木に追加する。計算量 $O(|s|)$
- next(idx)：頂点 idx から文字 c で遷移したときの頂点番号を返す。計算量 $O(1)$
- find(t)：文字列 t に対応する頂点番号を求める。存在しない場合、-1 を返す。計算量 $O(|t|)$

```cpp
struct Trie {
    static constexpr int C_SIZE = 26;    // C_SIZE  : 文字の種類数
    static constexpr int C_BEGIN = 'a';  // C_BEGIN : 開始文字
    static constexpr int ROOT = 0;
    struct Node {
        array<int, C_SIZE> to = {};  // 子ノードの番号，存在しなければ-1
        vector<int> ids;             // そのノードが終端である文字列のIDリスト
        Node() { to.fill(-1); }
    };
    vector<Node> nodes;
    int cnt = 0;  // 追加した文字列の個数
    Trie() : nodes(1) {}
    // nodes[idx]から文字cで遷移したときの頂点のindex
    int next(int idx, char c) { return nodes[idx].to[c - C_BEGIN]; }
    // 文字列の追加
    int insert(const string& s) {
        int now = ROOT;
        for (char c : s) {
            int k = c - C_BEGIN;
            if (nodes[now].to[k] == -1) {
                nodes[now].to[k] = nodes.size();
                nodes.push_back(Node());
            }
            now = nodes[now].to[k];
        }
        nodes[now].ids.push_back(cnt++);
        return now;
    }
    // 文字列に対応するnodeのindexを検索，存在しなければ-1
    int find(const string& s) {
        int now = ROOT;
        for (char c : s) {
            now = next(now, c);
            if (now == -1) return -1;
        }
        return now;
```

```
    }
};
```

6.3 Aho-Corasick 法

複数文字列検索アルゴリズム。入力文字列 t 内の文字列パターン $\{s_1, s_2, \ldots, s_m\}$ の要素と一致する箇所を全て検索する。Trie 木を拡張したオートマトンを構築する。

- insert(s)：パターンとして文字列 s を追加する。計算量 $O(|s|)$
- build()：前処理としてオートマトンの状態遷移を構築する。計算量 $O(k \cdot \sum |s_i|)$。k は文字の種類数。これ以下の関数は build() 後に正常に動作する。
- next(idx)：頂点 idx から文字 c で遷移したときの頂点番号を返す。計算量 $O(1)$
- accept(idx)：頂点 idx にマッチするパターンが存在するか。
- match(idx)：頂点 idx にマッチするすべてのパターンの ID を返す。
- search(t)：文字列 t にマッチするパターンを全て求める。`res[i]` には $t[i]$ を末尾としてマッチするパターンの ID リストが格納される。計算量 $O(|t| + \text{マッチ数})$

```cpp
struct AhoCorasick {
    static constexpr int C_SIZE = 26;    // C_SIZE  : 文字の種類数
    static constexpr int C_BEGIN = 'a';  // C_BEGIN : 開始文字
    static constexpr int ROOT = 0;
    struct Node {
        array<int, C_SIZE> to = {};    // to[c]=文字cによる遷移先ノード，存在しなければ-1
        vector<int> ids;               // そのノードでマッチする文字列のIDリスト
        int fail = ROOT, drct = ROOT;  // 失敗時の遷移先，suffixでマッチする遷移先
        Node() { to.fill(-1); }
    };
    vector<Node> nodes;
    vector<string> patterns;
    AhoCorasick() : nodes(1) {}

    int insert(const string& s) {
        int now = ROOT;
        for (char c : s) {
            int k = c - C_BEGIN;
            if (nodes[now].to[k] == -1) {
                nodes[now].to[k] = nodes.size();
                nodes.push_back(Node());
            }
            now = nodes[now].to[k];
        }
        nodes[now].ids.push_back(patterns.size());
        patterns.push_back(s);
        return now;
    }

    void build() {
        queue<int> que;
        for (int& x : nodes[ROOT].to) {
            if (x == -1) {
                x = ROOT;
            } else {
                que.push(x);
            }
        }
        while (!que.empty()) {
            int now = que.front();
            que.pop();
            int fail = nodes[now].fail;
            for (int k = 0; k < C_SIZE; k++) {
                int& nxt = nodes[now].to[k];
                if (nxt == -1) {
                    nxt = nodes[fail].to[k];  // failure link で遷移してから遷移
                } else {
                    nodes[nxt].fail = nodes[fail].to[k];  // 遷移先のfailure linkを求める
                    que.push(nxt);
                }
            }
            nodes[now].drct = (nodes[fail].ids.empty() ? nodes[fail].drct : fail);
        }
    }

    int next(int idx, char c) { return nodes[idx].to[c - C_BEGIN]; }

    bool accept(int idx) { return nodes[idx].drct != ROOT || !nodes[idx].ids.empty(); }

    vector<int> match(int idx) {
        vector<int> res;
        int now = idx;
        while (now != ROOT) {
            for (int id : nodes[now].ids) res.push_back(id);
            now = nodes[now].drct;
        }
        return res;
    }

    vector<vector<int>> search(const string& s) {
        vector<vector<int>> res;
        int now = ROOT;
        for (char c : s) {
            now = next(now, c);
            res.emplace_back(match(now));
        }
        return res;
    }
};
```

6.4 Suffix Array

文字列の suffix(接尾辞) の開始位置の配列を suffix の辞書順でソートした配列を求めるアルゴリズム

- suffix_array(str)：長さ n の文字列 str の suffix array を求める。計算量 $O(n \log^2 n)$
- contain(s, t, sa)：文字列 s, t と s の suffix array sa より s に t が含まれているかを判定する。$O(|t| \log |s|)$

```cpp
vector<int> suffix_array(const string& str) {
    int n = str.size();
    vector<int> sa(n + 1), rank(n + 1, -1);  // sa[i] = 辞書順で i番目であるsuffixの開始位置
    iota(sa.begin(), sa.end(), 0);
```

```cpp
    for (int i = 0; i < n; i++) rank[i] = str[i];
    int k;
    auto comp = [&](const int& i, const int& j) {
        if (rank[i] != rank[j]) {
            return rank[i] < rank[j];
        } else {
            int ri = i + k <= n ? rank[i + k] : -1;
            int rj = j + k <= n ? rank[j + k] : -1;
            return ri < rj;
        }
    };
    for (k = 1; k <= n; k <<= 1) {
        sort(sa.begin(), sa.end(), comp);
        vector<int> tmp(n + 1, 0);
        for (int i = 0; i < n; i++) {
            tmp[sa[i + 1]] = tmp[sa[i]];
            if (comp(sa[i], sa[i + 1])) tmp[sa[i + 1]]++;
        }
        rank = tmp;
    }
    return sa;
}

// 文字列sに文字列tに含まれているか判定する
bool contain(const string& s, const string& t, vector<int>& sa) {
    int l = 0, r = int(s.size());
    while (r - l > 1) {
        int mid = (l + r) / 2;
        if (s.substr(sa[mid], t.size()) < t) {
            l = mid;
        } else {
            r = mid;
        }
    }
    return s.substr(sa[r], t.size()) == t;
}
```

6.5 Longest Common Prefix Array

Suffix Array 上で隣接する suffix の最長共通接頭辞 (LCP) の長さを求めるアルゴリズム。

- lcp_array(s, sa) : 長さ n の文字列 s とその Suffix Array sa から LCP Array を求める。計算量 $O(n)$

```cpp
// lcp[i] = sa[i]とsa[i+1]の common prefix の長さ
vector<int> lcp_array(const string& s, const vector<int>& sa) {
    int n = sa.size();
    vector<int> lcp(n), rank(n);  // rank[i] = s[i,n)の辞書順位
    for (int i = 0; i < n; i++) rank[sa[i]] = i;
    for (int i = 0, len = 0; i < n; i++) {
        if (len > 0) len--;
        if (rank[i] == n - 1) continue;
        int j = sa[rank[i] + 1];  // s[i,n)より辞書順で1つ大きいsuffixの先頭位置
        while (max(i, j) + len < int(s.size()) && s[i + len] == s[j + len]) len++;
        lcp[rank[i]] = len;
    }
    return lcp;
}
```

6.6 Z algorithm

長さ n の文字列 s に対して、$s[0,n)$ と $s[i,n)$ の最長共通接頭辞 (LCP : Longest Common Prefix) の長さ z[i] を全ての i について求めるアルゴリズム。

- z_algorithm(s) : 長さ n の配列 z を返す。計算量 $O(n)$

```cpp
vector<int> z_algorithm(string& s) {
    int n = int(s.size());
    vector<int> z(n);
    z[0] = n;
    for (int i = 1, l = 0, r = 0; i < n; i++) {
        int& k = z[i];
        k = (r <= i ? 0 : min(r - i, z[i - l]));
        while (i + k < n && s[k] == s[i + k]) k++;
        if (r < i + k) l = i, r = i + k;
    }
    return z;
}
```

7 幾何

7.1 2D Geometry Template

二次元幾何のライブラリを利用するために必要となる構造体や関数などをまとめたものです。

```cpp
using Real = double;
using Point = complex<Real>;
using Polygon = vector<Point>;
const Real EPS = 1e-8, PI = acos(-1);

Point operator*(const Point& p, const Real& d) {
    return Point(p.real() * d, p.imag() * d);
}

Point operator/(const Point& p, const Real& d) {
    return Point(p.real() / d, p.imag() / d);
}

istream& operator>>(istream& is, Point& p) {
    Real a, b;
    is >> a >> b;
    p = Point(a, b);
    return is;
}

ostream& operator<<(ostream& os, const Point& p) {
    return os << fixed << setprecision(20) << p.real() << " " << p.imag();
}

int sign(const Real& r) {
    if (r <= -EPS) return -1;
    if (r >= +EPS) return +1;
    return 0;
}

bool equals(const Real& a, const Real& b) {
```

```
    return sign(a - b) == 0;
}

namespace std {
bool operator<(const Point& a, const Point& b) {
    if (equals(a.real(), b.real())) return a.imag() < b.imag();
    return a.real() < b.real();
}
}  // namespace std

Real dot(const Point& a, const Point& b) {
    return (conj(a) * b).real();
}

Real cross(const Point& a, const Point& b) {
    return (conj(a) * b).imag();
}

struct Line {
    Point a, b;
    Line() = default;
    Line(Point a, Point b) : a(a), b(b) {}
};
using Segment = Line;
```

7.2 2D Points and Vectors

- projection(l, p) : 直線 l に対する点 p の射影を求める
- reflection(l, p) : 直線 l に対する点 p の反射を求める
- ccw(a, b, c) : 点 a から見た、点 b, c の位置関係を求める
  - $+1$: a, b, c が反時計回りになる
  - -1 : a, b, c が時計回りになる
  - $+2$: c, a, b がこの順で同一直線上にある
  - -2 : a, b, c がこの順で同一直線上にある
  - 0 : a, c, b がこの順で同一直線上にある

```
Point projection(const Line& l, const Point& p) {
    return l.a + (l.a - l.b) * dot(p - l.a, l.a - l.b) / norm(l.a - l.b);
}

Point reflection(const Line& l, const Point& p) {
    return p + (projection(l, p) - p) * 2.0;
}

int ccw(const Point& a, Point b, Point c) {
    b -= a, c -= a;
    if (sign(cross(b, c)) == +1) return +1;
    if (sign(cross(b, c)) == -1) return -1;
    if (sign(dot(b, c)) == -1) return +2;
    if (norm(b) < norm(c)) return -2;
    return 0;
}
```

7.3 2D Segments and Lines

- is_orthogonal(a, b) : 直線 a, b が直交するか判定する
- is_parallel(a, b) : 直線 a, b が平行か判定する
- is_intersect_ss(s, t) : 線分 s, t が交差するか判定する
- cross_point_ll(l, m) : 直線 l, m の交点を求める
- distance_sp(s, p) : 線分 s と 点 p の距離を求める
- distance_ss(a, b) : 線分 a, b の距離を求める

```
bool is_orthogonal(const Line& a, const Line& b) {
    return equals(dot(a.b - a.a, b.b - b.a), 0);
}

bool is_parallel(const Line& a, const Line& b) {
    return equals(cross(a.b - a.a, b.b - b.a), 0);
}

bool is_intersect_ss(const Segment& s, const Segment& t) {
    return ccw(s.a, s.b, t.a) * ccw(s.a, s.b, t.b) <= 0 &&
           ccw(t.a, t.b, s.a) * ccw(t.a, t.b, s.b) <= 0;
}

Point cross_point_ll(const Line& l, const Line& m) {
    Real A = cross(l.b - l.a, m.b - m.a);
    Real B = cross(l.b - l.a, l.b - m.a);
    if (equals(abs(A), 0) && equals(abs(B), 0)) return m.a;
    return m.a + (m.b - m.a) * B / A;
}

Real distance_sp(const Segment& s, const Point& p) {
    Point r = projection(s, p);
    if (ccw(s.a, s.b, r) == 0) return abs(r - p);
    return min(abs(s.a - p), abs(s.b - p));
}

Real distance_ss(const Segment& a, const Segment& b) {
    if (is_intersect_ss(a, b)) return 0;
    return min({distance_sp(a, b.a), distance_sp(a, b.b), distance_sp(b, a.a), distance_sp
        (b, a.b)});
}
```

7.4 2D Polygon

- area(p) : 多角形 p の面積を求める
- is_convex(p) : p が凸多角形か判定する
- contains(Q, p) : 多角形 Q が 点 p を含んでいる ? 2 : 辺上にある ? 1 : 0
- convex_hull(p) : 点集合 p の凸包を求める
- convex_diameter(p) : 凸多角形 p の直径を求める
- convex_cut(p, l) : 凸多角形 p を直線 l で切断し、左にできた図形の面積を求める

```
Real area(const Polygon& p) {
    Real S = 0;
```

```cpp
    int n = p.size();
    for (int i = 0; i < n; i++) {
        S += cross(p[i], p[(i + 1) % n]);
    }
    return S * 0.5;
}

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

int contains(const Polygon& Q, const Point& p) {
    int n = Q.size();
    bool in = false;
    for (int i = 0; i < n; i++) {
        Point a = Q[i] - p, b = Q[(i + 1) % n] - p;
        if (sign(a.imag() - b.imag()) == +1) swap(a, b);
        if (sign(a.imag()) <= 0 && sign(b.imag()) == +1 && sign(cross(a, b)) == -1) in = !in;
        if (sign(cross(a, b)) == 0 && sign(dot(a, b)) <= 0) return 1;
    }
    return in ? 2 : 0;
}

Polygon convex_hull(Polygon p) {
    int n = p.size(), k = 0;
    if (n <= 2) return p;
    sort(p.begin(), p.end());
    Polygon ch(2 * n);
    for (int i = 0; i < n; ch[k++] = p[i++]) {
        while (k >= 2 && sign(cross(ch[k - 1] - ch[k - 2], p[i] - ch[k - 1])) == -1) --k;
    }
    for (int i = n - 2, t = k + 1; i >= 0; ch[k++] = p[i--]) {
        while (k >= t && sign(cross(ch[k - 1] - ch[k - 2], p[i] - ch[k - 1])) == -1) --k;
    }
    ch.resize(k - 1);
    return ch;
}

Real convex_diameter(const Polygon& p) {
    int n = p.size(), is = 0, js = 0;
    for (int i = 1; i < n; i++) {
        if (sign(p[i].imag() - p[is].imag()) == +1) is = i;
        if (sign(p[i].imag() - p[js].imag()) == -1) js = i;
    }
    Real maxdis = norm(p[is] - p[js]);
    int maxi, maxj, i, j;
    i = maxi = is;
    j = maxj = js;
    do {
        if (sign(cross(p[(i + 1) % n] - p[i], p[(j + 1) % n] - p[j])) >= 0) {
            j = (j + 1) % n;
        } else {
            i = (i + 1) % n;
        }
        if (norm(p[i] - p[j]) > maxdis) {
            maxdis = norm(p[i] - p[j]);
            maxi = i;
            maxj = j;
        }
    } while (i != is || j != js);
    return sqrt(maxdis);
}

Polygon convex_cut(const Polygon& p, const Line& l) {
    int n = p.size();
    Polygon ret;
    for (int i = 0; i < n; i++) {
        Point now = p[i], nxt = p[(i + 1) % n];
        if (ccw(l.a, l.b, now) != -1) ret.push_back(now);
        if (ccw(l.a, l.b, now) * ccw(l.a, l.b, nxt) < 0) {
            ret.push_back(cross_point_ll(Line(now, nxt), l));
        }
    }
    return ret;
}
```

7.5 2D Closest Pair

- closest_pair(ps)：点集合 ps について、最も近い 2 点の距離を求める

```cpp
Real closest_pair(Polygon ps) {
    sort(ps.begin(), ps.end());
    Polygon a(ps.size());

    function<Real(int, int)> rec = [&](int left, int right) -> Real {
        if (right - left <= 1) return 1e18;
        int mid = (left + right) / 2;
        Real x = ps[mid].real();
        Real ret = min(rec(left, mid), rec(mid, right));
        inplace_merge(ps.begin() + left, ps.begin() + mid, ps.begin() + right,
                      [&](const Point& a, const Point& b) { return a.imag() < b.imag(); });
        int pos = 0;
        for (int i = left; i < right; i++) {
            if (fabs((ps[i].real()) - x) >= ret) continue;
            for (int j = 0; j < pos; j++) {
                auto tar = ps[i] - a[pos - j - 1];
                if (tar.imag() >= ret) break;
                ret = min(ret, abs(tar));
            }
            a[pos++] = ps[i];
        }
        return ret;
    };
    return rec(0, (int)ps.size());
}
```

7.6 2D Circle

- intersection_cc(c1,c2)：円 $c1, c2$ の共通接線の個数を求める
- incircle(a, b, c)：三角形 abc の内接円を求める
- circumscribed_circle(a, b, c)：三角形 abc の外接円を求める

- cross_point_cl(c, l)：円 c と 直線 l の交点を求める
- cross_point_cc(c1, c2)：円 $c1, c2$ の交点を求める
- tangent_cp(c, p)：点 p を通る円 c の接線を求める
- tangent_cc(c1, c2)：円 $c1, c2$ の共通接線を求める
- area_poly_c(p, c)：多角形 p と円 c の共通部分の面積を求める
- area_cc(c1, c2)：円 $c1, c2$ の共通部分の面積を求める

```cpp
struct Circle {
    Point p;
    Real r;
    Circle() = default;
    Circle(Point p, Real r) : p(p), r(r) {}
};

int intersection_cc(Circle c1, Circle c2) {
    if (c1.r < c2.r) swap(c1, c2);
    Real d = abs(c1.p - c2.p);
    if (sign(c1.r + c2.r - d) == -1) return 4;
    if (equals(c1.r + c2.r, d)) return 3;
    if (sign(c1.r - c2.r - d) == -1) return 2;
    if (equals(c1.r - c2.r, d)) return 1;
    return 0;
}

Circle incircle(const Point& a, const Point& b, const Point& c) {
    Real A = abs(b - c), B = abs(c - a), C = abs(a - b);
    Point x = Point((a * A + b * B + c * C) / (A + B + C));
    Real r = distance_sp(Segment(a, b), x);
    return Circle(x, r);
}

Circle circumscribed_circle(const Point& a, const Point& b, const Point& c) {
    Point m1((a + b) / 2.0), m2((b + c) / 2.0);
    Point v((b - a).imag(), (a - b).real()), w((b - c).imag(), (c - b).real());
    Line s(m1, Point(m1 + v)), t(m2, Point(m2 + w));
    Point x = cross_point_ll(s, t);
    return Circle(x, abs(a - x));
}

pair<Point, Point> cross_point_cl(const Circle& c, const Line& l) {
    Point pr = projection(l, c.p);
    if (equals(abs(pr - c.p), c.r)) return {pr, pr};
    Point e = (l.b - l.a) / abs(l.b - l.a);
    Real k = sqrt(norm(c.r) - norm(pr - c.p));
    return {pr - e * k, pr + e * k};
}

pair<Point, Point> cross_point_cc(const Circle& c1, const Circle& c2) {
    Real d = abs(c1.p - c2.p);
    Real a = acos((c1.r * c1.r + d * d - c2.r * c2.r) / (2 * c1.r * d));
    Real t = atan2(c2.p.imag() - c1.p.imag(), c2.p.real() - c1.p.real());
    Point p1 = c1.p + Point(cos(t + a), sin(t + a)) * c1.r;
    Point p2 = c1.p + Point(cos(t - a), sin(t - a)) * c1.r;
    return {p1, p2};
}

pair<Point, Point> tangent_cp(const Circle& c, const Point& p) {
    return cross_point_cc(c, Circle(p, sqrt(norm(c.p - p) - c.r * c.r)));
}

vector<Line> tangent_cc(Circle c1, Circle c2) {
    vector<Line> ret;
    if (c1.r < c2.r) swap(c1, c2);
    Real g = norm(c1.p - c2.p);
    if (equals(g, 0.0)) return ret;
    Point u = (c2.p - c1.p) / sqrt(g);
    Point v = u * Point(cos(PI * 0.5), sin(PI * 0.5));
    for (int s : {-1, 1}) {
        Real h = (c1.r + s * c2.r) / sqrt(g);
        if (equals(1 - h * h, 0.0)) {
            ret.emplace_back(c1.p + u * c1.r, c1.p + (u + v) * c1.r);
        } else if (sign(1 - h * h) == +1) {
            Point uu = u * h, vv = v * sqrt(1 - h * h);
            ret.emplace_back(c1.p + (uu + vv) * c1.r, c2.p - (uu + vv) * c2.r * s);
            ret.emplace_back(c1.p + (uu - vv) * c1.r, c2.p - (uu - vv) * c2.r * s);
        }
    }
    return ret;
}

Real area_poly_c(const Polygon& p, const Circle& c) {
    int n = p.size();
    if (n < 3) return 0.0;
    function<Real(Circle, Point, Point)> cross_area = [&](const Circle& c, const Point& a,
          const Point& b) {
        Point va = c.p - a, vb = c.p - b;
        Real f = cross(va, vb), ret = 0.0;
        if (equals(f, 0.0)) return ret;
        if (max(abs(va), abs(vb)) < c.r + EPS) return f;
        if (distance_sp(Segment(a, b), c.p) > c.r - EPS) return c.r * c.r * arg(vb * conj(
              va));
        auto u = cross_point_cl(c, Segment(a, b));
        vector<Point> tot{a, u.first, u.second, b};
        for (int i = 0; i + 1 < (int)tot.size(); i++) {
            ret += cross_area(c, tot[i], tot[i + 1]);
        }
        return ret;
    };
    Real S = 0;
    for (int i = 0; i < n; i++) {
        S += cross_area(c, p[i], p[(i + 1) % n]);
    }
    return S * 0.5;
}

Real area_cc(const Circle& c1, const Circle& c2) {
    Real d = abs(c1.p - c2.p);
    if (c1.r + c2.r <= d + EPS) return 0.0;
    if (d <= fabs(c1.r - c2.r) + EPS) {
        Real r = min(c1.r, c2.r);
        return r * r * PI;
    }
    Real rc = (d * d + c1.r * c1.r - c2.r * c2.r) / (2.0 * d);
    Real theta = acos(rc / c1.r);
    Real phi = acos((d - rc) / c2.r);
    return c1.r * c1.r * theta + c2.r * c2.r * phi - d * c1.r * sin(theta);
```

```
}
```

7.7 3D Geometry Template

三次元幾何のライブラリを利用するために必要となるクラスや関数などをまとめたものです。

```
#define EPS (1e-7)
#define equals(a, b) (fabs((a) - (b)) < EPS)

class Point3d {
  public:
    double x, y, z;

    Point3d(double x = 0, double y = 0, double z = 0) : x(x), y(y), z(z) {}

    Point3d operator+(const Point3d& a) {
        return Point3d(x + a.x, y + a.y, z + a.z);
    }
    Point3d operator-(const Point3d& a) {
        return Point3d(x - a.x, y - a.y, z - a.z);
    }
    Point3d operator*(const double& d) {
        return Point3d(x * d, y * d, z * d);
    }
    Point3d operator/(const double& d) {
        return Point3d(x / d, y / d, z / d);
    }

    bool operator<(const Point3d& p) const {
        if (!equals(x, p.x)) return x < p.x;
        if (!equals(y, p.y)) return y < p.y;
        if (!equals(z, p.z)) return z < p.z;
        return false;
    }

    bool operator==(const Point3d& p) const {
        return equals(x, p.x) && equals(y, p.y) && equals(z, p.z);
    }
};

struct Segment3d {
    Point3d p[2];
    Segment3d(Point3d p1 = Point3d(), Point3d p2 = Point3d()) {
        p[0] = p1, p[1] = p2;
    }
    bool operator==(const Segment3d& seg) const {
        return (p[0] == seg.p[0] && p[1] == seg.p[1]) || (p[0] == seg.p[1] && p[1] == seg.p[0]);
    }
};

using Line3d = Segment3d;
using Vector3d = Point3d;

ostream& operator<<(ostream& os, const Point3d& p) {
    return os << "(" << p.x << "," << p.y << "," << p.z << ")";
}

ostream& operator<<(ostream& os, const Segment3d& seg) {
    return os << "(" << seg.p[0] << "," << seg.p[1] << ")";
}

double dot(const Point3d& a, const Point3d& b) {
    return a.x * b.x + a.y * b.y + a.z * b.z;
}

Vector3d cross(const Point3d& a, const Point3d& b) {
    return Vector3d(a.y * b.z - a.z * b.y, a.z * b.x - a.x * b.z, a.x * b.y - a.y * b.x);
}

inline double norm(const Point3d& p) {
    return p.x * p.x + p.y * p.y + p.z * p.z;
}

inline double abs(const Point3d& p) {
    return sqrt(norm(p));
}

inline double toRad(double theta) {
    return theta * M_PI / 180.0;
}

double distanceLP(Line3d line, Point3d p) {
    return abs(cross(line.p[1] - line.p[0], p - line.p[0])) / abs(line.p[1] - line.p[0]);
}

Point3d project(Segment3d seg, Point3d p) {
    Vector3d base = seg.p[1] - seg.p[0];
    double t = dot(p - seg.p[0], base) / norm(base);
    return seg.p[0] + base * t;
}

Point3d reflect(Segment3d seg, Point3d p) {
    return p + (project(seg, p) - p) * 2.0;
}

bool on_line3d(Line3d line, Point3d p) {
    return equals(abs(cross(line.p[1] - p, line.p[0] - p)), 0);
}

bool on_segment3d(Segment3d seg, Point3d p) {
    if (!on_line3d(seg, p)) return false;
    double dist[3] = {abs(seg.p[1] - seg.p[0]), abs(p - seg.p[0]), abs(p - seg.p[1])};
    return on_line3d(seg, p) && equals(dist[0], dist[1] + dist[2]);
}

double distanceSP(Segment3d seg, Point3d p) {
    Point3d r = project(seg, p);
    if (on_segment3d(seg, r)) return abs(p - r);
    return min(abs(seg.p[0] - p), abs(seg.p[1] - p));
}
```

7.8 3D Plane

平面に対する操作をまとめたクラスファイルです。

```
class Plane3d {
  public:
    Point3d normal_vector;  // 法線ベクトル
    double d;               // 平面方程式 normal_vector = (a,b,c), a*x + b*y + c*z + d = 0
```

```cpp

    Plane3d(Point3d normal_vector = Point3d(), double d = 0) : normal_vector(normal_vector), d(d) {}
    Plane3d(Vector3d a, Vector3d b, Vector3d c) {
        Vector3d v1 = b - a;
        Vector3d v2 = c - a;
        Vector3d tmp = cross(v1, v2);
        normal_vector = tmp / abs(tmp);
        set_d(a);
    }

    // 法線ベクトルnormal_vectorと平面上の１点からdを計算する
    void set_d(Point3d p) {
        d = dot(normal_vector, p);
    }

    // 平面と点pの距離を求める
    double distanceP(Point3d p) {
        Point3d a = normal_vector * d;  // 平面上の適当な点をつくる
        return abs(dot(p - a, normal_vector));
    }

    // 平面上でもっとも点pと近い点を求める
    Point3d nearest_point(Point3d p) {
        Point3d a = normal_vector * d;
        return p - (normal_vector * dot(p - a, normal_vector));
    }

    // 平面と線分が交差するか
    bool intersectS(Segment3d seg) {
        Point3d a = normal_vector * d;
        double res1 = dot(a - seg.p[0], normal_vector);
        double res2 = dot(a - seg.p[1], normal_vector);
        if (res1 > res2) swap(res1, res2);
        if ((equals(res1, 0.0) || res1 < 0) && (equals(res2, 0.0) || res2 > 0)) return true;
        return false;
    }

    // 平面と線分の交点を求める
    Point3d crosspointS(Segment3d seg) {
        Point3d a = normal_vector * d;
        double dot_p0a = fabs(dot(seg.p[0] - a, normal_vector));
        double dot_p1a = fabs(dot(seg.p[1] - a, normal_vector));
        if (equals(dot_p0a + dot_p1a, 0)) return seg.p[0];
        return seg.p[0] + (seg.p[1] - seg.p[0]) * (dot_p0a / (dot_p0a + dot_p1a));
    }
};
```

7.9 3D Point on the Triangle

平面上の三角形 (tri1, tri2, tri3) と点 (p) を入力として受け取り、その点が三角形上に存在するかどうか判定します。

```cpp
bool point_on_the_triangle3d(Point3d tri1, Point3d tri2, Point3d tri3, Point3d p) {
    // 線分上にpがあった場合、三角形内とみなす場合は以下のコメントアウトを外す
    /*
    if( on_segment3d(Segment3d(tri1,tri2),p) ) return true;
    if( on_segment3d(Segment3d(tri2,tri3),p) ) return true;
    if( on_segment3d(Segment3d(tri3,tri1),p) ) return true;
    */

    vector<Point3d> vec(3);
    vec[0] = tri1, vec[1] = tri2, vec[2] = tri3;
    double area = 0;
    {
        double a = abs(vec[0] - vec[1]), b = abs(vec[1] - vec[2]), c = abs(vec[2] - vec[0]);
        double s = (a + b + c) / 2;
        area = sqrt(s * (s - a) * (s - b) * (s - c));
    }
    double sum = 0;
    for (int i = 0; i < 3; ++i) {
        double a = abs(vec[i] - vec[(i + 1) % 3]), b = abs(vec[(i + 1) % 3] - p), c = abs(p - vec[i]);
        double s = (a + b + c) / 2;
        sum += sqrt(s * (s - a) * (s - b) * (s - c));
    }
    return equals(sum, area);
}
```

7.10 3D Libraries for Lines and Segments

平面上の直線と線分に関数するライブラリです。ライブラリ中では直線は Line3d、線分は Segment3d として表記されます。

```cpp
// 直線 l1 と l2 は平行か？
bool isParallel(Line3d l1, Line3d l2) {
    Vector3d A = l1.p[0], B = l1.p[1], C = l2.p[0], D = l2.p[1];
    Vector3d AB = B - A, CD = D - C;
    Vector3d n1 = AB / abs(AB), n2 = CD / abs(CD);
    double tmp = dot(n1, n2);
    tmp = 1 - tmp * tmp;
    return equals(tmp, 0.0);
}

// 直線 l1 と l2 を結ぶような線分であって最も距離が短いものを返す
// Note: l1 と l2 が平行な時には使用できないので注意
Segment3d nearest_segmentLL(Line3d l1, Line3d l2) {
    assert(!isParallel(l1, l2));  // 平行な場合は使用不可
    // l1.p[0] = A, l1.p[1] = B, l2.p[0] = C, l2.p[1] = D
    Vector3d AB = l1.p[1] - l1.p[0];
    Vector3d CD = l2.p[1] - l2.p[0];
    Vector3d AC = l2.p[0] - l1.p[0];
    Vector3d n1 = AB / abs(AB), n2 = CD / abs(CD);
    double d1 = (dot(n1, AC) - dot(n1, n2) * dot(n2, AC)) / (1.0 - pow(dot(n1, n2), 2));
    double d2 = (dot(n1, n2) * dot(n1, AC) - dot(n2, AC)) / (1.0 - pow(dot(n1, n2), 2));
    return Segment3d(l1.p[0] + n1 * d1, l2.p[0] + n2 * d2);
}

// 直線 l1 と l2 は交差するか？
bool intersectLL(Line3d l1, Line3d l2) {
    Vector3d A = l1.p[0], B = l1.p[1], C = l2.p[0], D = l2.p[1];

    // そもそもl1,l2が直線じゃない
    if (equals(abs(B - A), 0.0) || equals(abs(D - C), 0.0)) {
        // この場合は注意
        // そもそも与えられた線分が線分になっていないので、交差するかどうかは判定できない
        return false;
    }
```

```
    Vector3d AB = B - A, CD = D - C;
    Vector3d n1 = AB / abs(AB), n2 = CD / abs(CD);
    double tmp = dot(n1, n2);
    tmp = 1 - tmp * tmp;

    if (equals(tmp, 0.0)) return 0;  // 直線が平行

    Segment3d ns = nearest_segmentLL(l1, l2);
    if (ns.p[0] == ns.p[1]) return true;
    return false;
}

// 線分 seg1 と seg2 は交差しているか？
bool intersectSS(Segment3d seg1, Segment3d seg2) {
    if (isParallel(seg1, seg2)) return false;
    Segment3d seg = nearest_segmentLL(seg1, seg2);
    if (!(seg.p[0] == seg.p[1])) return false;
    Point3d cp = seg.p[1];
    return on_segment3d(seg1, cp) && on_segment3d(seg2, cp);
}
```

7.11 3D Intersection of Planes

２つの平面の交差判定等を行うライブラリです。

```
using P3db = pair<Point3d, bool>;

/*
[*] Input:
    ２つの平面 pl1, pl2
[*] Output:
    ２つの平面が交線をもつ場合 -> first:交線上の任意の１点, second:true
           交線を持たない場合 -> first:empty            , second:false
*/
P3db intersectPlPl(const Plane3d& pl1, const Plane3d& pl2) {
    Vector3d v = cross(pl1.normal_vector, pl2.normal_vector);
    if (!equals(v.x, 0.0)) {
        Point3d p(0,
                  (pl1.d * pl2.normal_vector.z - pl2.d * pl1.normal_vector.z) / v.x,
                  (pl1.d * pl2.normal_vector.y - pl2.d * pl1.normal_vector.y) / (-v.x));
        return P3db(p, true);
    }
    if (!equals(v.y, 0.0)) {
        Point3d p((pl1.d * pl2.normal_vector.z - pl2.d * pl1.normal_vector.z) / (-v.y),
                  0,
                  (pl1.d * pl2.normal_vector.x - pl2.d * pl1.normal_vector.x) / v.y);
        return P3db(p, true);
    }
    if (!equals(v.z, 0.0)) {
        Point3d p((pl1.d * pl2.normal_vector.y - pl2.d * pl1.normal_vector.y) / v.z,
                  (pl1.d * pl2.normal_vector.x - pl2.d * pl1.normal_vector.x) / (-v.z),
                  0);
        return P3db(p, true);
    }
    return P3db(Point3d(), false);  //平行なのでそのような交線は存在しない
}

/*
[*] Input:
    ２つの平面 plane, plane2 とその交線上の任意の１点
[*] Output:
    ２つの平面の交線

説明:
2つの平面の外積から交線の方向ベクトルを得る
あとは任意の１点に拡張した方向ベクトルを加えてセグメント化する
交線上の任意の１点はintersectPlPlで取得できる
面倒な仕様になってしまった
*/
Line3d intersectPlPl_converter(Plane3d plane, Plane3d plane2, Point3d tmp) {
    Vector3d ve = cross(plane.normal_vector, plane2.normal_vector);
    return Line3d(tmp, tmp + (ve * 10));  // 任意の倍数で拡張、ここでは10
}
```