my library for ICPC

Kimiyuki Onaka

December 7, 2018
repo: git@github.com:kmyk/competitive-programming-library.git
commit: aac1b19c13d77bf4dd4b7b8a008f629645600137

Contents

1 misc

1.1 environment.sh

```
#!/bin/bash

cat <<EOF > ~/.vimrc
syntax on
set smartindent
set tabstop=4
set shiftwidth=4
set expandtab
set relativenumber
EOF

setxkbmap -option ctrl:swapcaps

alias e=vim
alias cxx='$CXX␣-std=c++14␣-Wall␣-O2'
alias cxxo='$CXX␣-std=c++14␣-Wall␣-O3␣-mtune=native␣-march=native'
alias cxxg='$CXX␣-std=c++14␣-Wall␣-g␣-fsanitize=undefined␣-D_GLIBCXX_DEBUG'

judge() {
    for f in test/*.in ; do
        echo $f
        diff <(./a.out < $f) ${f%.in}.out
    done
}
print() {
    for x in {A..Z} ; do
        url=http://255.255.255.255/${x}_ja.html
        if [[ $(curl -s $url -o /dev/null -w '%{http_code}') = 200 ]] ; then
            google-chrome --headless --disable-gpu --print-to-pdf $url
            lpr output.pdf
        fi
    done
}
```

1.2 template.cpp

```
:%! sh -c "`cat`"
#!/bin/sh
cat <<EOF
/**
 * @file .cpp
 * @author `git config user.name`
 * @date `date +'%a. %d, %Y'`
 */
#include <bits/stdc++.h>
#include <algorithm>
#include <array>
#include <cassert>
#include <climits>
#include <cmath>
#include <cstdio>
#include <functional>
#include <iostream>
#include <map>
#include <numeric>
#include <queue>
#include <set>
#include <tuple>
#include <unordered_map>
#include <unordered_set>
#include <vector>
#define REP(i, n) for (int i = 0; (i) < (int)(n); ++ (i))
#define REP3(i, m, n) for (int i = (m); (i) < (int)(n); ++ (i))
#define REP_R(i, n) for (int i = int(n) - 1; (i) >= 0; -- (i))
#define REP3R(i, m, n) for (int i = int(n) - 1; (i) >= (int)(m); -- (i))
#define ALL(x) begin(x), end(x)
#define dump(x) cerr << #x "␣=␣" << x << endl
#define unittest_name_helper(counter) unittest_ ## counter
#define unittest_name(counter) unittest_name_helper(counter)
#define unittest __attribute__((constructor)) void unittest_name(__COUNTER__) ()
using ll = long long;
using namespace std;
template <class T> using reversed_priority_queue = priority_queue<T, vector<T>, greater<T> >;
template <class T, class U> inline void chmax(T & a, U const & b) { a = max<T>(a, b); }
template <class T, class U> inline void chmin(T & a, U const & b) { a = min<T>(a, b); }
template <typename X, typename T> auto vectors(X x, T a) { return vector<T>(x, a); }
template <typename X, typename Y, typename Z, typename... Zs> auto vectors(X x, Y y, Z z, Zs... zs) { auto cont = vectors(y, z, zs...); return vector<decltype(cont)>(x,
      cont); }
template <typename T> ostream & operator << (ostream & out, vector<T> const & xs) { REP (i, int(xs.size()) - 1) out << xs[i] << '␣'; if (not xs.empty()) out << xs.back()
      ; return out; }
void graphviz(vector<vector<int> > const & g, bool is_digraph = false, string const & name = "graph") { ofstream ofs(name + ".dot"); ofs << (is_digraph ? "di" : "") << "
      graph␣graph_name␣{" << endl; ofs << "␣␣␣␣graph␣[␣bgcolor␣=␣\"#00000000\"␣]" << endl; ofs << "␣␣␣␣node␣[␣shape␣=␣circle,␣style␣=␣filled,␣fillcolor␣=␣\"#ffffffff\"␣]
      " << endl; REP (i, g.size()) for (int j : g[i]) if (is_digraph or i <= j) ofs << "␣␣␣␣" << i << (is_digraph ? "␣->␣" : "␣--␣") << j << endl; ofs << "}" << endl;
      ofs.close(); system(("dot␣-T␣png␣" + name + ".dot␣>␣" + name + ".png").c_str()); } /* {{{ }}} */
const int dy[] = { -1, 1, 0, 0 };
const int dx[] = { 0, 0, 1, -1 };
bool is_on_field(int y, int x, int h, int w) { return 0 <= y and y < h and 0 <= x and x < w; }
int main() {
    int n; scanf("%d", &n);
    vector<ll> a(n); REP (i, n) scanf("%lld", &a[i]);
    vector<vector<int> > f = vectors(h, w, int());
    REP (y, h) REP (x, w) scanf("%d", &f[y][x]);
    printf("%lld\n", ans);
    return 0;
}
EOF
```

2 data structure

2.1 data-structure/binary-indexed-tree.inc.cpp

```
template <typename Monoid>
struct binary_indexed_tree {  // on monoid
    typedef typename Monoid::underlying_type underlying_type;
    vector<underlying_type> data;
    Monoid mon;
    binary_indexed_tree(size_t n, Monoid const & a_mon = Monoid()) : mon(a_mon) {
        data.resize(n, mon.unit());
    }
    void point_append(size_t i, underlying_type z) {  // data[i] += z
        for (size_t j = i + 1; j <= data.size(); j += j & -j) data[j - 1] = mon.append(data[j - 1], z);
    }
    underlying_type initial_range_concat(size_t i) {  // sum [0, i)
```

```
            underlying_type acc = mon.unit();
            for (size_t j = i; 0 < j; j -= j & -j) acc = mon.append(data[j - 1], acc);
            return acc;
        }
};
struct plus_monoid {
    typedef int underlying_type;
    int unit() const { return 0; }
    int append(int a, int b) const { return a + b; }
};

unittest {
    binary_indexed_tree<plus_monoid> bit(8);
    bit.point_append(3, 4);
    bit.point_append(4, 3);
    bit.point_append(7, 1);
    assert (bit.initial_range_concat(3) == 0);
    assert (bit.initial_range_concat(5) == 7);
    assert (bit.initial_range_concat(8) == 8);
    bit.point_append(4, 2);
    assert (bit.initial_range_concat(3) == 0);
    assert (bit.initial_range_concat(5) == 9);
    assert (bit.initial_range_concat(8) == 10);
}
```

2.2 data-structure/segment-tree.inc.cpp

```
/**
 * @brief a segment tree, or a fenwick tree
 * @tparam Monoid (commutativity is not required)
 */
template <class Monoid>
struct segment_tree {
    typedef typename Monoid::underlying_type underlying_type;
    int n;
    vector<underlying_type> a;
    const Monoid mon;
    segment_tree() = default;
    segment_tree(int a_n, underlying_type initial_value = Monoid().unit(), Monoid const & a_mon = Monoid()) : mon(a_mon) {
        n = 1; while (n < a_n) n *= 2;
        a.resize(2 * n - 1, mon.unit());
        fill(a.begin() + (n - 1), a.begin() + ((n - 1) + a_n), initial_value); // set initial values
        REP_R (i, n - 1) a[i] = mon.append(a[2 * i + 1], a[2 * i + 2]); // propagate initial values
    }
    void point_set(int i, underlying_type z) { // 0-based
        assert (0 <= i and i <= n);
        a[i + n - 1] = z;
        for (i = (i + n) / 2; i > 0; i /= 2) { // 1-based
            a[i - 1] = mon.append(a[2 * i - 1], a[2 * i]);
        }
    }
    underlying_type range_concat(int l, int r) { // 0-based, [l, r)
        assert (0 <= l and l <= r and r <= n);
        underlying_type lacc = mon.unit(), racc = mon.unit();
        for (l += n, r += n; l < r; l /= 2, r /= 2) { // 1-based loop, 2x faster than recursion
            if (l % 2 == 1) lacc = mon.append(lacc, a[(l ++) - 1]);
            if (r % 2 == 1) racc = mon.append(a[(-- r) - 1], racc);
        }
        return mon.append(lacc, racc);
    }
};
struct plus_monoid {
    typedef int underlying_type;
    int unit() const { return 0; }
    int append(int a, int b) const { return a + b; }
};
template <int mod>
struct modplus_monoid {
    typedef int underlying_type;
    int unit() const { return 0; }
    int append(int a, int b) const { int c = a + b; return c < mod ? c : c - mod; }
};
struct max_monoid {
    typedef int underlying_type;
    int unit() const { return INT_MIN; }
    int append(int a, int b) const { return max(a, b); }
};
struct min_monoid {
    typedef int underlying_type;
    int unit() const { return INT_MAX; }
    int append(int a, int b) const { return min(a, b); }
};
```

2.3 data-structure/dual-segment-tree.inc.cpp

```
template <class OperatorMonoid>
struct dual_segment_tree {
    typedef typename OperatorMonoid::underlying_type operator_type;
    typedef typename OperatorMonoid::target_type underlying_type;
    int n;
    vector<operator_type> f;
    vector<underlying_type> a;
    const OperatorMonoid op;
    dual_segment_tree() = default;
    dual_segment_tree(int a_n, underlying_type initial_value, OperatorMonoid const & a_op = OperatorMonoid()) : op(a_op) {
        n = 1; while (n < a_n) n *= 2;
        a.resize(n, initial_value);
        f.resize(n - 1, op.unit());
    }
    underlying_type point_get(int i) { // 0-based
        underlying_type acc = a[i];
        for (i = (i + n) / 2; i > 0; i /= 2) { // 1-based
            acc = op.apply(f[i - 1], acc);
        }
        return acc;
    }
    void range_apply(int l, int r, operator_type z) { // 0-based, [l, r)
        assert (0 <= l and l <= r and r <= n);
        range_apply(0, 0, n, l, r, z);
    }
    void range_apply(int i, int il, int ir, int l, int r, operator_type z) {
        if (l <= il and ir <= r) { // 0-based
            if (i < f.size()) {
                f[i] = op.append(z, f[i]);
            } else {
                a[i - n + 1] = op.apply(z, a[i - n + 1]);
            }
        } else if (ir <= l or r <= il) {
            // nop
        } else {
            range_apply(2 * i + 1, il, (il + ir) / 2, 0, n, f[i]);
            range_apply(2 * i + 2, (il + ir) / 2, ir, 0, n, f[i]);
            f[i] = op.unit();
```

```cpp
            range_apply(2 * i + 1, il, (il + ir) / 2, l, r, z);
            range_apply(2 * i + 2, (il + ir) / 2, ir, l, r, z);
        }
    }
    void point_set(int i, underlying_type z) {
        range_apply(i, i + 1, op.unit());  // to flush lazed ops
        a[i + n - 1] = z;
    }
/*
    // fast methods
    inline underlying_type point_get(int i) {
        return a[i + n - 1];
    }
    inline void point_set_primitive(int i, underlying_type z) {
        a[i + n - 1] = z;
    }
    void point_set_commit() {
        REP_R (i, n - 1) a[i] = mon.append(a[2 * i + 1], a[2 * i + 2]);
    }
*/
};

struct plus_operator_monoid {
    typedef int underlying_type;
    typedef int target_type;
    int unit() const { return 0; }
    int append(int a, int b) const { return a + b; }
    int apply(int a, int b) const { return a + b; }
};
struct max_operator_monoid {
    typedef int underlying_type;
    typedef int target_type;
    int unit() const { return INT_MIN; }
    int append(int a, int b) const { return max(a, b); }
    int apply(int a, int b) const { return max(a, b); }
};
struct min_operator_monoid {
    typedef int underlying_type;
    typedef int target_type;
    int unit() const { return INT_MAX; }
    int append(int a, int b) const { return min(a, b); }
    int apply(int a, int b) const { return min(a, b); }
};

unittest {
    dual_segment_tree<min_operator_monoid> segtree(12, 100);
    segtree.range_apply(2, 7, 50);
    segtree.range_apply(5, 9, 30);
    segtree.range_apply(1, 11, 80);
    assert (segtree.point_get( 0) == 100);
    assert (segtree.point_get( 1) ==  80);
    assert (segtree.point_get( 2) ==  50);
    assert (segtree.point_get( 3) ==  50);
    assert (segtree.point_get( 4) ==  50);
    assert (segtree.point_get( 5) ==  30);
    assert (segtree.point_get( 6) ==  30);
    assert (segtree.point_get( 7) ==  30);
    assert (segtree.point_get( 8) ==  30);
    assert (segtree.point_get( 9) ==  80);
    assert (segtree.point_get(10) ==  80);
    assert (segtree.point_get(11) == 100);
}

template <int MOD>
struct linear_operator_monoid {
    typedef pair<int, int> underlying_type;
    typedef int target_type;
    linear_operator_monoid() = default;
    underlying_type unit() const {
        return make_pair(1, 0);
    }
    underlying_type append(underlying_type g, underlying_type f) const {
        target_type fst = g.first *(ll) f.first % MOD;
        target_type snd = (g.second + g.first *(ll) f.second) % MOD;
        return make_pair(fst, snd);
    }
    target_type apply(underlying_type f, target_type x) const {
        return (f.first *(ll) x + f.second) % MOD;
    }
};
```

2.4 data-structure/lazy-propagation-segment-tree.inc.cpp

```cpp
/**
 * @note lazy_propagation_segment_tree<max_monoid, plus_operator_monoid> is the starry sky tree
 * @note verified https://www.hackerrank.com/contests/world-codesprint-12/challenges/factorial-array/submissions/code/1304452669
 * @note verified https://www.hackerrank.com/contests/world-codesprint-12/challenges/animal-transport/submissions/code/1304454860
 * @note intersting discussion about range-extension and partial-function-extension: https://github.com/kmyk/competitive-programming-library/issues/3
 */
template <class Monoid, class OperatorMonoid>
struct lazy_propagation_segment_tree { // on monoids
    static_assert (is_same<typename Monoid::underlying_type, typename OperatorMonoid::target_type>::value, "");
    typedef typename Monoid::underlying_type underlying_type;
    typedef typename OperatorMonoid::underlying_type operator_type;
    const Monoid mon;
    const OperatorMonoid op;
    int n;
    vector<underlying_type> a;
    vector<operator_type> f;
    lazy_propagation_segment_tree() = default;
    lazy_propagation_segment_tree(int a_n, underlying_type initial_value = Monoid().unit(), Monoid const & a_mon = Monoid(), OperatorMonoid const & a_op = OperatorMonoid())
            : mon(a_mon), op(a_op) {
        n = 1; while (n <= a_n) n *= 2;
        a.resize(2 * n - 1, mon.unit());
        fill(a.begin() + (n - 1), a.begin() + ((n - 1) + a_n), initial_value); // set initial values
        REP_R (i, n - 1) a[i] = mon.append(a[2 * i + 1], a[2 * i + 2]); // propagate initial values
        f.resize(max(0, (2 * n - 1) - n), op.identity());
    }
    void point_set(int i, underlying_type z) {
        assert (0 <= i and i < n);
        point_set(0, 0, n, i, z);
    }
    void point_set(int i, int il, int ir, int j, underlying_type z) {
        if (i == n + j - 1) { // 0-based
            a[i] = z;
        } else if (ir <= j or j+1 <= il) {
            // nop
        } else {
            range_apply(2 * i + 1, il, (il + ir) / 2, 0, n, f[i]);
            range_apply(2 * i + 2, (il + ir) / 2, ir, 0, n, f[i]);
            f[i] = op.identity();
            point_set(2 * i + 1, il, (il + ir) / 2, j, z);
            point_set(2 * i + 2, (il + ir) / 2, ir, j, z);
            a[i] = mon.append(a[2 * i + 1], a[2 * i + 2]);
```

```
        }
    }
    void range_apply(int l, int r, operator_type z) {
        assert (0 <= l and l <= r and r <= n);
        range_apply(0, 0, n, l, r, z);
    }
    void range_apply(int i, int il, int ir, int l, int r, operator_type z) {
        if (l <= il and ir <= r) { // 0-based
            a[i] = op.apply(z, a[i]);
            if (i < f.size()) f[i] = op.compose(z, f[i]);
        } else if (ir <= l or r <= il) {
            // nop
        } else {
            range_apply(2 * i + 1, il, (il + ir) / 2, 0, n, f[i]);
            range_apply(2 * i + 2, (il + ir) / 2, ir, 0, n, f[i]);
            f[i] = op.identity();  // unnecessary if the oprator monoid is commutative
            range_apply(2 * i + 1, il, (il + ir) / 2, l, r, z);
            range_apply(2 * i + 2, (il + ir) / 2, ir, l, r, z);
            a[i] = mon.append(a[2 * i + 1], a[2 * i + 2]);
        }
    }
    underlying_type range_concat(int l, int r) {
        assert (0 <= l and l <= r and r <= n);
        return range_concat(0, 0, n, l, r);
    }
    underlying_type range_concat(int i, int il, int ir, int l, int r) {
        if (l <= il and ir <= r) { // 0-based
            return a[i];
        } else if (ir <= l or r <= il) {
            return mon.unit();
        } else {
            return op.apply(f[i], mon.append(
                    range_concat(2 * i + 1, il, (il + ir) / 2, l, r),
                    range_concat(2 * i + 2, (il + ir) / 2, ir, l, r)));
        }
    }
};

struct max_monoid {
    typedef int underlying_type;
    int unit() const { return 0; }
    int append(int a, int b) const { return max(a, b); }
};
struct plus_operator_monoid {
    typedef int underlying_type;
    typedef int target_type;
    int identity() const { return 0; }
    int apply(underlying_type a, target_type b) const { return a + b; }
    int compose(underlying_type a, underlying_type b) const { return a + b; }
};
typedef lazy_propagation_segment_tree<max_monoid, plus_operator_monoid> starry_sky_tree;

struct min_monoid {
    typedef int underlying_type;
    int unit() const { return INT_MAX; }
    int append(int a, int b) const { return min(a, b); }
};
struct plus_with_int_max_operator_monoid {
    typedef int underlying_type;
    typedef int target_type;
    int identity() const { return 0; }
    int apply(underlying_type a, target_type b) const { return b == INT_MAX ? INT_MAX : a + b; }
    int compose(underlying_type a, underlying_type b) const { return a + b; }
};

unittest {
    lazy_propagation_segment_tree<min_monoid, plus_with_int_max_operator_monoid> segtree(9);
    segtree.point_set(2, 2);
    segtree.point_set(3, 3);
    segtree.point_set(4, 4);
    segtree.point_set(6, 6);
    assert (segtree.range_concat(2, 3) == 2);
    assert (segtree.range_concat(5, 8) == 6);
    segtree.range_apply(1, 4, 9);
    assert (segtree.range_concat(3, 6) == 4);
    assert (segtree.range_concat(0, 3) == 11);
}

template <int N>
struct count_monoid {
    typedef array<int, N> underlying_type;
    underlying_type unit() const { return underlying_type(); }
    underlying_type append(underlying_type a, underlying_type b) const {
        underlying_type c = {};
        REP (i, N) c[i] = a[i] + b[i];
        return c;
    }
};
template <int N>
struct increment_operator_monoid {
    typedef int underlying_type;
    typedef array<int, N> target_type;
    underlying_type identity() const { return 0; }
    target_type apply(underlying_type a, target_type b) const {
        if (a == 0) return b;
        target_type c = {};
        REP (i, N - a) c[i + a] = b[i];
        return c;
    }
    underlying_type compose(underlying_type a, underlying_type b) const { return a + b; }
};

template <int32_t MOD>
struct plus_monoid {
    typedef mint<MOD> underlying_type;
    underlying_type unit() const { return 0; }
    underlying_type append(underlying_type a, underlying_type b) const { return a + b; }
};
template <int32_t MOD>
struct linear_operator_monoid {
    typedef pair<mint<MOD>, mint<MOD> > underlying_type;
    typedef mint<MOD> target_type;
    static underlying_type make(mint<MOD> a, mint<MOD> b) {
        return make_pair(a, b);
    }
    underlying_type identity() const {
        return make(1, 0);
    }
    target_type apply(underlying_type a, target_type b) const {
        return a.first * b + a.second;
    }
    underlying_type compose(underlying_type a, underlying_type b) const {
        return make(a.first * b.first, a.second + a.first * b.second);
    }
};
```

2.5 data-structure/dynamic-segment-tree.inc.cpp

```cpp
/**
 * @note verified http://arc054.contest.atcoder.jp/submissions/1335245
 * @note verified https://csacademy.com/contest/ceoi-2018-day-2/task/fibonacci-representations-small/
 * @note you can implement this with unordered_map, but the constructor requires the size
 */
template <class Monoid>
struct dynamic_segment_tree { // on monoid
    typedef Monoid monoid_type;
    typedef typename Monoid::underlying_type underlying_type;
    struct node_t {
        int left, right; // indices on pool
        underlying_type value;
    };
    deque<node_t> pool;
    stack<int> bin;
    int root; // index
    ll width; // of the tree
    int size; // the number of leaves
    Monoid mon;
    dynamic_segment_tree(Monoid const & a_mon = Monoid()) : mon(a_mon) {
        node_t node = { -1, -1, mon.unit() };
        pool.push_back(node);
        root = 0;
        width = 1;
        size = 1;
    }
protected:
    int create_node(int parent, bool is_right) {
        // make a new node
        int i;
        if (bin.empty()) {
            i = pool.size();
            node_t node = { -1, -1, mon.unit() };
            pool.push_back(node);
        } else {
            i = bin.top();
            bin.pop();
            pool[i] = { -1, -1, mon.unit() };
        }
        // link from the parent
        assert (parent != -1);
        int & ptr = is_right ? pool[parent].right : pool[parent].left;
        assert (ptr == -1);
        ptr = i;
        return i;
    }
    underlying_type get_value(int i) {
        return i == -1 ? mon.unit() : pool[i].value;
    }
public:
    void point_set(ll i, underlying_type z) {
        assert (0 <= i);
        while (width <= i) {
            node_t node = { root, -1, pool[root].value };
            root = pool.size();
            pool.push_back(node);
            width *= 2;
        }
        point_set(root, -1, false, 0, width, i, z);
    }
    void point_set(int i, int parent, bool is_right, ll il, ll ir, ll j, underlying_type z) {
        if (il == j and ir == j + 1) { // 0-based
            if (i == -1) {
                i = create_node(parent, is_right);
                size += 1;
            }
            pool[i].value = z;
        } else if (ir <= j or j + 1 <= il) {
            // nop
        } else {
            if (i == -1) i = create_node(parent, is_right);
            point_set(pool[i].left,  i, false, il, (il + ir) / 2, j, z);
            point_set(pool[i].right, i, true,  (il + ir) / 2, ir, j, z);
            pool[i].value = mon.append(get_value(pool[i].left), get_value(pool[i].right));
        }
    }
    void point_delete(ll i) {
        assert (0 <= i);
        if (width <= i) return;
        root = point_delete(root, -1, false, 0, width, i);
    }
    int point_delete(int i, int parent, bool is_right, ll il, ll ir, ll j) {
        if (i == -1) {
            return -1;
        } else if (il == j and ir == j + 1) { // 0-based
            bin.push(i);
            size -= 1;
            return -1;
        } else if (ir <= j or j + 1 <= il) {
            return i;
        } else {
            pool[i].left  = point_delete(pool[i].left,  i, false, il, (il + ir) / 2, j);
            pool[i].right = point_delete(pool[i].right, i, true,  (il + ir) / 2, ir, j);
            if (pool[i].left == -1 and pool[i].right == -1 and i != root) {
                bin.push(i);
                size -= 1;
                return -1;
            } else {
                pool[i].value = mon.append(get_value(pool[i].left), get_value(pool[i].right));
                return i;
            }
        }
    }
    underlying_type range_concat(ll l, ll r) {
        assert (0 <= l and l <= r);
        if (width <= l) return mon.unit();
        return range_concat(root, 0, width, l, min(width, r));
    }
    underlying_type range_concat(int i, ll il, ll ir, ll l, ll r) {
        if (i == -1) return mon.unit();
        if (l <= il and ir <= r) { // 0-based
            return pool[i].value;
        } else if (ir <= l or r <= il) {
            return mon.unit();
        } else {
            return mon.append(
                    range_concat(pool[i].left,  il, (il + ir) / 2, l, r),
                    range_concat(pool[i].right, (il + ir) / 2, ir, l, r));
        }
    }
    template <class Func>
    void traverse_leaves(Func func) {
        return traverse_leaves(root, 0, width, func);
    }
    template <class Func>
```

```cpp
    void traverse_leaves(ll i, ll il, ll ir, Func func) {
        if (i == -1) return;
        if (ir - il == 1) {
            func(il, pool[i].value);
        } else {
            traverse_leaves(pool[i].left,  il, (il + ir) / 2, func);
            traverse_leaves(pool[i].right, (il + ir) / 2, ir, func);
        }
    }
};
```

2.6 data-structure/union-find-tree.inc.cpp

```cpp
struct union_find_tree {
    vector<int> data;
    union_find_tree() = default;
    explicit union_find_tree(size_t n) : data(n, -1) {}
    bool is_root(int i) { return data[i] < 0; }
    int find_root(int i) { return is_root(i) ? i : (data[i] = find_root(data[i])); }
    int tree_size(int i) { return - data[find_root(i)]; }
    int unite_trees(int i, int j) {
        i = find_root(i); j = find_root(j);
        if (i != j) {
            if (tree_size(i) < tree_size(j)) swap(i,j);
            data[i] += data[j];
            data[j] = i;
        }
        return i;
    }
    bool is_same(int i, int j) { return find_root(i) == find_root(j); }
};
```

2.7 data-structure/treap.inc.cpp

```cpp
#include <random>
#include <memory>

// https://www.hackerrank.com/contests/zalando-codesprint/challenges/give-me-the-order/submissions/code/6004391
template <typename T>
struct treap {
    typedef T value_type;
    typedef double key_type;
    value_type v;
    key_type k;
    shared_ptr<treap> l, r;
    size_t m_size;
    treap(value_type v)
            : v(v)
            , k(generate())
            , l()
            , r()
            , m_size(1) {
    }
    static size_t size(shared_ptr<treap> const & t) {
        return t ? t->m_size : 0;
    }
    static shared_ptr<treap> merge(shared_ptr<treap> const & a, shared_ptr<treap> const & b) { // destructive
        if (not a) return b;
        if (not b) return a;
        if (a->k > b->k) {
            a->r = merge(a->r, b);
            return update(a);
        } else {
            b->l = merge(a, b->l);
            return update(b);
        }
    }
    static pair<shared_ptr<treap>, shared_ptr<treap> > split(shared_ptr<treap> const & t, size_t i) { // [0, i) [i, n), destructive
        if (not t) return { shared_ptr<treap>(), shared_ptr<treap>() };
        if (i <= size(t->l)) {
            shared_ptr<treap> u; tie(u, t->l) = split(t->l, i);
            return { u, update(t) };
        } else {
            shared_ptr<treap> u; tie(t->r, u) = split(t->r, i - size(t->l) - 1);
            return { update(t), u };
        }
    }
    static shared_ptr<treap> insert(shared_ptr<treap> const & t, size_t i, value_type v) { // destructive
        shared_ptr<treap> l, r; tie(l, r) = split(t, i);
        shared_ptr<treap> u = make_shared<treap>(v);
        return merge(merge(l, u), r);
    }
    static pair<shared_ptr<treap>, shared_ptr<treap> > erase(shared_ptr<treap> const & t, size_t i) { // (t \ t_i, t_i), destructive
        shared_ptr<treap> l, u, r;
        tie(l, r) = split(t, i + 1);
        tie(l, u) = split(l, i);
        return { merge(l, r), u };
    }
private:
    static shared_ptr<treap> update(shared_ptr<treap> const & t) {
        if (t) {
            t->m_size = 1 + size(t->l) + size(t->r);
        }
        return t;
    }
    static key_type generate() {
        static random_device device;
        static default_random_engine engine(device());
        static uniform_real_distribution<double> dist;
        return dist(engine);
    }
};
```

2.8 data-structure/sparse-table.inc.cpp

```cpp
/**
 * @brief sparse table on a semilattice
 * @note a semilattice is a commutative idempotent semigroup
 * @note for convenience, it requires the unit
 * @note space: O(N log N)
 * @note time:  O(N log N) for construction; O(1) for query
 */
template <class Semilattice>
struct sparse_table {
    typedef typename Semilattice::underlying_type underlying_type;
    vector<vector<underlying_type> > table;
    Semilattice lat;
    sparse_table() = default;
    sparse_table(vector<underlying_type> const & data, Semilattice const & a_lat = Semilattice())
            : lat(a_lat) {
```

```
        int n = data.size();
        int log_n = 32 - __builtin_clz(n);
        table.resize(log_n, vector<underlying_type>(n));
        table[0] = data;
        REP (k, log_n - 1) {
            REP (i, n) {
                table[k + 1][i] = i + (1ll << k) < n ?
                    lat.append(table[k][i], table[k][i + (1ll << k)]) :
                    table[k][i];
            }
        }
    }
    underlying_type range_concat(int l, int r) const {
        if (l == r) return lat.unit();  // if there is no unit, remove this line
        assert (0 <= l and l < r and r <= table[0].size());
        int k = 31 - __builtin_clz(r - l);  // log2
        return lat.append(table[k][l], table[k][r - (1ll << k)]);
    }
};
struct max_semilattice {
    typedef int underlying_type;
    int unit() const { return INT_MIN; }
    int append(int a, int b) const { return max(a, b); }
};
struct min_semilattice {
    typedef int underlying_type;
    int unit() const { return INT_MAX; }
    int append(int a, int b) const { return min(a, b); }
};
struct gcd_semilattice {
    typedef int underlying_type;
    int unit() const { return 0; }
    int append(int a, int b) const { return gcd(a, b); }
};

unittest {
    random_device device;
    default_random_engine gen(device());
    int n = uniform_int_distribution<int>(1, 500)(gen);
    vector<int> data(n);
    REP (i, n) data[i] = uniform_int_distribution<int>(- 1000000, 1000000)(gen);
    sparse_table<max_semilattice> table(data);
    auto range_concat = [&](int l, int r) {
        int acc = INT_MIN;
        REP3 (i, l, r) chmax(acc, data[i]);
        return acc;
    };
    REP (iteration, 500) {
        int l = uniform_int_distribution<int>(0, n - 1)(gen);
        int r = uniform_int_distribution<int>(l + 1, n)(gen);
        assert (table.range_concat(l, r) == range_concat(l, r));
    }
}
```

2.9 data-structure/sliding-window.inc.cpp

```
/**
 * @brief the sliding window minimum algorithm
 * @note to get maximums, use greater<T>
 * @note verified http://poj.org/problem?id=2823
 * @note verified http://cf16-tournament-round3-open.contest.atcoder.jp/tasks/asaporo_d
 */
template <typename T, class Compare = less<T> >
class sliding_window {
    deque<pair<int, T> > data;
    Compare compare;
public:
    sliding_window(Compare const & a_compare = Compare())
        : compare(a_compare) {}
    T front() {  // O(1), minimum
        return data.front().second;
    }
    void push_back(int i, T a) {  // O(1) amortized.
        while (not data.empty() and compare(a, data.back().second)) {
            data.pop_back();
        }
        data.emplace_back(i, a);
    }
    void pop_front(int i) {
        if (data.front().first == i) {
            data.pop_front();
        }
    }
    void push_front(int i, T a) {
        if (data.empty() or not compare(data.front().second, a)) {
            data.emplace_front(i, a);
        }
    }
};
```

2.10 data-structure/convex-hull-trick.inc.cpp

```
/**
 * @note y = ax + b
 */
struct line_t { ll a, b; };
bool operator < (line_t lhs, line_t rhs) { return make_pair(- lhs.a, lhs.b) < make_pair(- rhs.a, rhs.b); }
struct rational_t { ll num, den; };
rational_t make_rational(ll num, ll den = 1) {
    if (den < 0) { num *= -1; den *= -1; }
    return { num, den };
}
bool operator < (rational_t lhs, rational_t rhs) {
    if (lhs.num ==   LLONG_MAX or rhs.num == - LLONG_MAX) return false;
    if (lhs.num == - LLONG_MAX or rhs.num ==   LLONG_MAX) return true;
    return lhs.num * rhs.den < rhs.num * lhs.den;
}

/*
 * @sa http://d.hatena.ne.jp/sune2/20140310/1394440369
 * @sa http://techtipshoge.blogspot.jp/2013/06/convex-hull-trickdequepop-back.html
 * @sa http://satanic0258.hatenablog.com/entry/2016/08/16/181331
 * @sa http://wcipeg.com/wiki/Convex_hull_trick
 * @note verified at http://codeforces.com/contest/631/submission/31828502
 */
struct convex_hull_trick {
    convex_hull_trick() {
        lines.insert({ + LLONG_MAX, 0 });  // sentinels
        lines.insert({ - LLONG_MAX, 0 });
        cross.emplace(make_rational(- LLONG_MAX), (line_t) { - LLONG_MAX, 0 });
    }
    /**
     * @note O(log n)
```

```
     */
    void add_line(ll a, ll b) {
        auto it = lines.insert({ a, b }).first;
        if (not is_required(*prev(it), { a, b }, *next(it))) {
            lines.erase(it);
            return;
        }
        cross.erase(cross_point(*prev(it), *next(it)));
        {  // remove right lines
            auto ju = prev(it);
            while (ju != lines.begin() and not is_required(*prev(ju), *ju, { a, b })) -- ju;
            cross_erase(ju, prev(it));
            it = lines.erase(++ ju, it);
        }
        {  // remove left lines
            auto ju = next(it);
            while(next(ju) != lines.end() and not is_required({ a, b }, *ju, *next(ju))) ++ ju;
            cross_erase(++ it, ju);
            it = prev(lines.erase(it, ju));
        }
        cross.emplace(cross_point(*prev(it), *it), *it);
        cross.emplace(cross_point(*it, *next(it)), *next(it));
    }
    /**
     * @note O(log n)
     */
    ll get_min(ll x) const {
        line_t f = prev(cross.lower_bound(make_rational(x)))->second;
        return f.a * x + f.b;
    }
private:
    set<line_t> lines;
    map<rational_t, line_t> cross;
    template <typename Iterator>
    void cross_erase(Iterator first, Iterator last) {
        for (; first != last; ++ first) {
            cross.erase(cross_point(*first, *next(first)));
        }
    }
    rational_t cross_point(line_t f1, line_t f2) const {
        if (f1.a ==   LLONG_MAX) return make_rational(- LLONG_MAX);
        if (f2.a == - LLONG_MAX) return make_rational(  LLONG_MAX);
        return make_rational(f1.b - f2.b, f2.a - f1.a);
    }
    bool is_required(line_t f1, line_t f2, line_t f3) const {
        if (f1.a == f2.a and f1.b <= f2.b) return false;
        if (f1.a == LLONG_MAX or f3.a == - LLONG_MAX) return true;
        return (f2.a - f1.a) * (f3.b - f2.b) < (f2.b - f1.b) * (f3.a - f2.a);
    }
};

unittest {
    default_random_engine gen;
    repeat (iteration, 1000) {
        vector<pair<int, int> > lines;
        convex_hull_trick cht;
        repeat (i, 100) {
            int a = uniform_int_distribution<int>(- 30, 30)(gen);
            int b = uniform_int_distribution<int>(- 30, 30)(gen);
            lines.emplace_back(a, b);
            cht.add_line(a, b);
        }
        repeat (i, 10) {
            int x = uniform_int_distribution<int>(- 100, 100)(gen);
            int y = INT_MAX;
            for (auto line : lines) {
                int a, b; tie(a, b) = line;
                setmin(y, a * x + b);
            }
            assert (cht.get_min(x) == y);
        }
    }
}

struct inverted_convex_hull_trick {
    convex_hull_trick data;
    void add_line(ll a, ll b) { data.add_line(- a, - b); }
    ll get_max(ll x) { return - data.get_min(x); }
};
```

3 graph

3.1 graph/ford-fulkerson.inc.cpp

```
struct edge_t { int to, cap, rev; };
int maximum_flow_destructive(int s, int t, vector<vector<edge_t> > & g) { // ford fulkerson, O(EF)
    int n = g.size();
    vector<bool> used(n);
    function<int (int, int)> dfs = [&](int i, int f) {
        if (i == t) return f;
        used[i] = true;
        for (edge_t & e : g[i]) {
            if (used[e.to] or e.cap <= 0) continue;
            int nf = dfs(e.to, min(f, e.cap));
            if (nf > 0) {
                e.cap -= nf;
                g[e.to][e.rev].cap += nf;
                return nf;
            }
        }
        return 0;
    };
    int result = 0;
    while (true) {
        used.clear(); used.resize(n);
        int f = dfs(s, numeric_limits<int>::max());
        if (f == 0) break;
        result += f;
    }
    return result;
}
void add_edge(vector<vector<edge_t> > & g, int from, int to, int cap) {
    g[from].push_back((edge_t) {   to, cap, int(g[  to].size()    ) });
    g[  to].push_back((edge_t) { from,   0, int(g[from].size() - 1) });
}
int maximum_flow(int s, int t, vector<vector<edge_t> > g /* adjacency list */) { // ford fulkerson, O(FE)
    return maximum_flow_destructive(s, t, g);
}

vector<pair<int,int> > perfect_bipartite_matching(set<int> const & a, set<int> const & b, vector<vector<int> > const & g /* adjacency list */) { // O(V + FE)
    assert (a.size() + b.size() <= g.size());
    int n = g.size();
    int src = n;
```

```
    int dst = n + 1;
    vector<vector<edge_t> > h(n + 2);
    auto add_edge = [&](int from, int to, int cap) {
        h[from].push_back((edge_t) {   to, cap, int(h[  to].size()    ) });
        h[  to].push_back((edge_t) { from,   0, int(h[from].size() - 1) });
    };
    repeat (i,n) {
        if (a.count(i)) {
            add_edge(src, i, 1);
            for (int j : g[i]) if (b.count(j)) {
                add_edge(i, j, 1); // collect edges e : a -> b, from g
            }
        }
        if (b.count(i)) {
            add_edge(i, dst, 1);
        }
    }
    maximum_flow_destructive(src, dst, h);
    vector<pair<int,int> > ans;
    for (int from : a) {
        for (edge_t e : h[from]) if (b.count(e.to) and e.cap == 0) {
            ans.emplace_back(from, e.to);
        }
    }
    return ans;
}
```

3.2 graph/dinic.inc.cpp

```
// https://kimiyuki.net/blog/2016/01/16/arc-031-d/
double maximum_flow(int s, int t, vector<vector<double> > const & capacity /* adjacency matrix */) { // dinic, O(V^2E)
    int n = capacity.size();
    vector<vector<double> > flow(n, vector<double>(n));
    auto residue = [&](int i, int j) { return capacity[i][j] - flow[i][j]; };
    vector<vector<int> > g(n); repeat (i,n) repeat (j,n) if (capacity[i][j] or capacity[j][i]) g[i].push_back(j); // adjacency list
    double result = 0;
    while (true) {
        vector<int> level(n, -1); level[s] = 0;
        queue<int> q; q.push(s);
        for (int d = n; not q.empty() and level[q.front()] < d; ) {
            int i = q.front(); q.pop();
            if (i == t) d = level[i];
            for (int j : g[i]) if (level[j] == -1 and residue(i,j) > 0) {
                level[j] = level[i] + 1;
                q.push(j);
            }
        }
        vector<bool> finished(n);
        function<double (int, double)> augmenting_path = [&](int i, double cur) -> double {
            if (i == t or cur == 0) return cur;
            if (finished[i]) return 0;
            finished[i] = true;
            for (int j : g[i]) if (level[i] < level[j]) {
                double f = augmenting_path(j, min(cur, residue(i,j)));
                if (f > 0) {
                    flow[i][j] += f;
                    flow[j][i] -= f;
                    finished[i] = false;
                    return f;
                }
            }
            return 0;
        };
        bool cont = false;
        while (true) {
            double f = augmenting_path(s, numeric_limits<double>::max());
            if (f == 0) break;
            result += f;
            cont = true;
        }
        if (not cont) break;
    }
    return result;
}

// https://kimiyuki.net/blog/2017/10/22/kupc-2017-h/
uint64_t pack(int i, int j) {
    return (uint64_t(i) << 32) | j;
}
ll maximum_flow(int s, int t, int n, unordered_map<uint64_t, ll> & capacity /* adjacency matrix */) { // dinic, O(V^2E)
    auto residue = [&](int i, int j) { auto key = pack(i, j); return capacity.count(key) ? capacity[key] : 0; };
    vector<vector<int> > g(n); repeat (i,n) repeat (j,n) if (residue(i, j) or residue(j, i)) g[i].push_back(j); // adjacency list
    ll result = 0;
    while (true) {
        vector<int> level(n, -1); level[s] = 0;
        queue<int> q; q.push(s);
        for (int d = n; not q.empty() and level[q.front()] < d; ) {
            int i = q.front(); q.pop();
            if (i == t) d = level[i];
            for (int j : g[i]) if (level[j] == -1 and residue(i,j) > 0) {
                level[j] = level[i] + 1;
                q.push(j);
            }
        }
        vector<bool> finished(n);
        function<ll (int, ll)> augmenting_path = [&](int i, ll cur) -> ll {
            if (i == t or cur == 0) return cur;
            if (finished[i]) return 0;
            finished[i] = true;
            for (int j : g[i]) if (level[i] < level[j]) {
                ll f = augmenting_path(j, min(cur, residue(i,j)));
                if (f > 0) {
                    capacity[pack(i, j)] -= f;
                    capacity[pack(j, i)] += f;
                    finished[i] = false;
                    return f;
                }
            }
            return 0;
        };
        bool cont = false;
        while (true) {
            ll f = augmenting_path(s, numeric_limits<ll>::max());
            if (f == 0) break;
            result += f;
            cont = true;
        }
        if (not cont) break;
    }
    return result;
}
```

3.3 graph/minimum-cost-flow.inc.cpp

```
template <class T>
struct edge { int to; T cap, cost; int rev; };
template <class T>
void add_edge(vector<vector<edge<T> > > & graph, int from, int to, T cap, T cost) {
    graph[from].push_back((edge<T>) {   to, cap,   cost, int(graph[  to].size())     });
    graph[  to].push_back((edge<T>) { from,   0, - cost, int(graph[from].size()) - 1 });
}
/**
 * @brief minimum-cost flow with primal-dual method
 * @note mainly O(V^2UC) for U is the sum of capacities and C is the sum of costs. and additional O(VE) if negative edges exist
 */
template <class T>
T min_cost_flow_destructive(int src, int dst, T flow, vector<vector<edge<T> > > & graph) {
    T result = 0;
    vector<T> potential(graph.size());
    if (0 < flow) { // initialize potential when negative edges exist (slow). you can remove this if unnecessary
        fill(ALL(potential), numeric_limits<T>::max());
        potential[src] = 0;
        while (true) { // Bellman-Ford algorithm
            bool updated = false;
            REP (e_from, graph.size()) for (auto & e : graph[e_from]) if (e.cap) {
                if (potential[e_from] == numeric_limits<T>::max()) continue;
                if (potential[e.to] > potential[e_from] + e.cost) {
                    potential[e.to] = potential[e_from] + e.cost; // min
                    updated = true;
                }
            }
            if (not updated) break;
        }
    }
    while (0 < flow) {
        // update potential using dijkstra
        vector<T> distance(graph.size(), numeric_limits<T>::max()); // minimum distance
        vector<int> prev_v(graph.size()); // constitute a single-linked-list represents the flow-path
        vector<int> prev_e(graph.size());
        { // initialize distance and prev_{v,e}
            reversed_priority_queue<pair<T, int> > que; // distance * vertex
            distance[src] = 0;
            que.emplace(0, src);
            while (not que.empty()) { // Dijkstra's algorithm
                T d; int v; tie(d, v) = que.top(); que.pop();
                if (potential[v] == numeric_limits<T>::max()) continue; // for unreachable nodes
                if (distance[v] < d) continue;
                // look round the vertex
                REP (e_index, graph[v].size()) {
                    // consider updating
                    edge<T> e = graph[v][e_index];
                    int w = e.to;
                    if (potential[w] == numeric_limits<T>::max()) continue;
                    T d1 = distance[v] + e.cost + potential[v] - potential[w]; // updated distance
                    if (0 < e.cap and d1 < distance[e.to]) {
                        distance[w] = d1;
                        prev_v[w] = v;
                        prev_e[w] = e_index;
                        que.emplace(d1, w);
                    }
                }
            }
        }
        if (distance[dst] == numeric_limits<T>::max()) return -1; // no such flow
        REP (v, graph.size()) {
            if (potential[v] == numeric_limits<T>::max()) continue;
            potential[v] += distance[v];
        }
        // finish updating the potential
        // let flow on the src->dst minimum path
        T delta = flow; // capacity of the path
        for (int v = dst; v != src; v = prev_v[v]) {
            chmin(delta, graph[prev_v[v]][prev_e[v]].cap);
        }
        flow -= delta;
        result += delta * potential[dst];
        for (int v = dst; v != src; v = prev_v[v]) {
            edge<T> & e = graph[prev_v[v]][prev_e[v]]; // reference
            e.cap -= delta;
            graph[v][e.rev].cap += delta;
        }
    }
    return result;
}
```

3.4 graph/lowest-common-ancestor.inc.cpp

```
struct indexed_min_monoid {
    typedef pair<int, int> underlying_type;
    underlying_type unit() const { return { INT_MAX, INT_MAX }; }
    underlying_type append(underlying_type a, underlying_type b) const { return min(a, b); }
};
/**
 * @brief lowest common ancestor with \pm 1 RMQ and sparse table
 * @see https://www.slideshare.net/yumainoue965/lca-and-rmq
 * @note verified http://www.utpc.jp/2011/problems/travel.html
 */
struct lowest_common_ancestor {
    sparse_table<indexed_min_monoid> table;
    vector<int> index;
    lowest_common_ancestor() = default;
    /**
     * @note O(N)
     * @param g is an adjacent list of a tree
     */
    lowest_common_ancestor(int root, vector<vector<int> > const & g) {
        vector<pair<int, int> > tour;
        index.assign(g.size(), -1);
        function<void (int, int, int)> go = [&](int i, int parent, int depth) {
            index[i] = tour.size();
            tour.emplace_back(depth, i);
            for (int j : g[i]) if (j != parent) {
                go(j, i, depth + 1);
                tour.emplace_back(depth, i);
            }
        };
        go(root, -1, 0);
        table = sparse_table<indexed_min_monoid>(tour);
    }
    /**
     * @note O(1)
     */
    int operator () (int x, int y) const {
        assert (0 <= x and x < index.size());
        assert (0 <= y and y < index.size());
        x = index[x];
        y = index[y];
        if (x > y) swap(x, y);
        return table.range_concat(x, y + 1).second;
```

```
    }
};
```

3.5 graph/strongly-connected-components.cpp

```
vector<vector<int> > opposite_graph(vector<vector<int> > const & g) {
    int n = g.size();
    vector<vector<int> > h(n);
    REP (i, n) for (int j : g[i]) h[j].push_back(i);
    return h;
}
/**
 * @brief strongly connected components decomposition, Kosaraju's algorithm
 * @return the pair (the number k of components, the function from vertices of g to components)
 * @note O(V + E)
 */
pair<int, vector<int> > decompose_to_strongly_connected_components(vector<vector<int> > const & g, vector<vector<int> > const & g_rev) {
    int n = g.size();
    vector<int> acc(n); {
        vector<bool> used(n);
        function<void (int)> dfs = [&](int i) {
            used[i] = true;
            for (int j : g[i]) if (not used[j]) dfs(j);
            acc.push_back(i);
        };
        REP (i,n) if (not used[i]) dfs(i);
        reverse(ALL(acc));
    }
    int size = 0;
    vector<int> component_of(n); {
        vector<bool> used(n);
        function<void (int)> rdfs = [&](int i) {
            used[i] = true;
            component_of[i] = size;
            for (int j : g_rev[i]) if (not used[j]) rdfs(j);
        };
        for (int i : acc) if (not used[i]) {
            rdfs(i);
            ++ size;
        }
    }
    return { size, move(component_of) };
}

/**
 * @return a tree in many cases
 */
vector<vector<int> > decomposed_graph(int size, vector<int> const & component_of, vector<vector<int> > const & g) {
    int n = g.size();
    vector<vector<int> > h(size);
    REP (i, n) for (int j : g[i]) {
        if (component_of[i] != component_of[j]) {
            h[component_of[i]].push_back(component_of[j]);
        }
    }
    REP (k, size) {
        sort(ALL(h[k]));
        h[k].erase(unique(ALL(h[k])), h[k].end());
    }
    return h;
}

/**
 * @brief memory optimized version
 * @note                   stack overflow
 */
pair<int, vector<int> > decompose_to_strongly_connected_components(vector<vector<int> > const & g, vector<vector<int> > const & g_rev) {
    int n = g.size();
    vector<int> acc(n); {
        auto it = acc.rbegin();
        vector<bool> used(n);
        stack<pair<int, int> > stk;
        REP (k, n) if (not used[k]) {
            stk.emplace(k, 0);
            used[k] = true;
            while (not stk.empty()) { // dfs
                int i = stk.top().first;
                int & e = stk.top().second;
                while (e < g[i].size()) {
                    int j = g[i][e];
                    ++ e;
                    if (not used[j]) {
                        stk.emplace(j, 0);
                        used[j] = true;
                        break;
                    }
                }
                if (stk.top().first == i) {
                    *(it ++) = i;
                    stk.pop();
                }
            }
        }
    }
    int size = 0;
    vector<int> component_of(n); {
        vector<bool> used(n);
        stack<int> stk;
        for (int k : acc) if (not used[k]) {
            stk.push(k);
            used[k] = true;
            while (not stk.empty()) { // dfs
                int i = stk.top(); stk.pop();
                component_of[i] = size;
                for (int j : g_rev[i]) if (not used[j]) {
                    stk.push(j);
                    used[j] = true;
                }
            }
            ++ size;
        }
    }
    return { size, move(component_of) };
}
```

3.6 graph/two-edge-connected-components.inc.cpp

```
/**
 * @brief 2-edge-connected components decomposition
 * @param g an adjacent list of the simple undirected graph
 * @note O(V + E)
 */
pair<int, vector<int> > decompose_to_two_edge_connected_components(vector<vector<int> > const & g) {
```

```cpp
    int n = g.size();
    vector<int> imos(n); { // imos[i] == 0  iff  the edge i -> parent is a bridge
        vector<char> used(n); // 0: unused ; 1: exists on stack ; 2: removed from stack
        function<void (int, int)> go = [&](int i, int parent) {
            used[i] = 1;
            for (int j : g[i]) if (j != parent) {
                if (used[j] == 0) {
                    go(j, i);
                    imos[i] += imos[j];
                } else if (used[j] == 1) {
                    imos[i] += 1;
                    imos[j] -= 1;
                }
            }
            used[i] = 2;
        };
        REP (i, n) if (used[i] == 0) {
            go(i, -1);
        }
    }
    int size = 0;
    vector<int> component_of(n, -1); {
        function<void (int)> go = [&](int i) {
            for (int j : g[i]) if (component_of[j] == -1) {
                component_of[j] = imos[j] == 0 ? size ++ : component_of[i];
                go(j);
            }
        };
        REP (i, n) if (component_of[i] == -1) {
            component_of[i] = size ++;
            go(i);
        }
    }
    return { size, move(component_of) };
}
```

3.7 graph/centroid-decomposition.inc.cpp

```cpp
/**
 * @note O(n) time
 * @note O(n) space on heap
 * @return the size is 1 or 2
 */
vector<int> get_centroids(vector<vector<int> > const & g, int root, set<int> const & forbidden) {
    map<int, int> available; {
        function<void (int, int)> go = [&](int i, int parent) {
            available.emplace(i, available.size());
            for (auto j : g[i]) if (j != parent and not forbidden.count(j)) {
                go(j, i);
            }
        };
        go(root, -1);
    }
    int n = available.size();
    vector<int> result;
    vector<int> size(n, -1);
    function<void (int, int)> go = [&](int x, int parent) {
        bool is_centroid = true;
        int i = available[x];
        size[i] = 1;
        for (auto y : g[x]) if (y != parent and available.count(y)) {
            int j = available[y];
            go(y, x);
            size[i] += size[j];
            if (size[j] > n / 2) is_centroid = false;
        }
        if (n - size[i] > n / 2) is_centroid = false;
        if (is_centroid) result.push_back(x);
    };
    go(root, -1);
    return result;
}
```

3.8 graph/topological-sort.inc.cpp

```cpp
vector<int> topological_sort(int n, vector<vector<int> > const & g_rev) {
    vector<int> order;
    vector<char> used(n);
    function<bool (int)> go = [&](int i) {
        used[i] = 1;  // in stack
        for (int j : g_rev[i]) {
            if (used[j] == 1) return true;
            if (not used[j]) {
                if (go(j)) return true;
            }
        }
        used[i] = 2;  // completely used
        order.push_back(i);
        return false;
    };
    REP (i, n) if (not used[i]) {
        if (go(i)) return vector<int>();
    }
    return order;
}
```

4 modulus

4.1 modulus/factorial.inc.cpp

```cpp
template <int32_t MOD>
mint<MOD> fact(int n) {
    static vector<mint<MOD> > memo(1, 1);
    while (n >= memo.size()) {
        memo.push_back(memo.back() * mint<MOD>(memo.size()));
    }
    return memo[n];
}
template <int32_t PRIME>
mint<PRIME> inv_fact(int n) {
    static vector<mint<PRIME> > memo;
    if (memo.size() <= n) {
        int l = memo.size();
        int r = n * 1.3 + 100;
        memo.resize(r);
        memo[r - 1] = fact<PRIME>(r - 1).inv();
        for (int i = r - 2; i >= l; -- i) {
```

```cpp
            memo[i] = memo[i + 1] * (i + 1);
        }
    }
    return memo[n];
}

unittest {
    constexpr int32_t MOD = 1e9 + 7;
    assert (fact<MOD>(0) == 1);
    assert (fact<MOD>(1) == 1);
    assert (fact<MOD>(2) == 2);
    assert (fact<MOD>(3) == 6);
    assert (fact<MOD>(4) == 24);
    assert (fact<MOD>(5) == 120);
}
```

4.2 modulus/choose.inc.cpp

```cpp
/**
 * @tparam MOD must be a prime
 * @note O(n log n) at first time, otherwise O(1)
 */
template <int32_t MOD>
mint<MOD> choose(int n, int r) {
    assert (0 <= r and r <= n);
    return fact<MOD>(n) * inv_fact<MOD>(n - r) * inv_fact<MOD>(r);
}
template <int32_t MOD>
mint<MOD> permute(int n, int r) {
    assert (0 <= r and r <= n);
    return fact<MOD>(n) * inv_fact<MOD>(n - r);
}
template <int32_t MOD>
mint<MOD> multichoose(int n, int r) {
    assert (0 <= n and 0 <= r);
    if (n == 0 and r == 0) return 1;
    return choose<MOD>(n + r - 1, r);
}
```

5 number

5.1 number/gcd.inc.cpp

```cpp
/**
 * @note if arguments are negative, the result may be negative
 */
template <typename T>
T gcd(T a, T b) {
    while (a) {
        b %= a;
        swap(a, b);
    }
    return b;
}
template <typename T>
T lcm(T a, T b) {
    return a / gcd(a, b) * b;
}

unittest {
    assert (gcd(0, 0) == 0);
    assert (gcd(42, 0) == 42);
    assert (gcd(0, 42) == 42);
    assert (gcd(3, -12) == 3);
    assert (gcd(-3, 12) == -3);
    assert (gcd(7, -12) == -1);
    assert (gcd(-7, 12) == 1);
    assert (gcd(-9, -12) == -3);
    assert (gcd(-1, -1) == -1);
}
```

5.2 number/primes.inc.cpp

```cpp
/**
 * @brief enumerate primes in [2, n) with O(n log log n)
 */
vector<int> list_primes(int n) {
    vector<bool> is_prime(n, true);
    is_prime[0] = is_prime[1] = false;
    for (int i = 2; i *(ll) i < n; ++ i)
        if (is_prime[i])
            for (int k = 2 * i; k < n; k += i)
                is_prime[k] = false;
    vector<int> primes;
    for (int i = 2; i < n; ++ i)
        if (is_prime[i])
            primes.push_back(i);
    return primes;
}

/**
 * @note O(sqrt n)
 */
bool is_prime(ll n, vector<int> const & primes) {
    if (n == 1) return false;
    for (int p : primes) {
        if (n < (ll)p * p) break;
        if (n % p == 0) return true;
    }
    return false;
}

/**
 * @note the last number of primes must be >= sqrt n
 */
map<ll, int> prime_factorize(ll n, vector<int> const & primes) {
    map<ll, int> result;
    for (int p : primes) {
        if (n < p *(ll) p) break;
        while (n % p == 0) {
            result[p] += 1;
            n /= p;
        }
    }
    if (n != 1) result[n] += 1;
    return result;
}
```

```

/**
 * @note if n < 10^9, d(n) < 1200 + a
 */
vector<ll> list_divisors(ll n, vector<int> const & primes) {
    vector<ll> result;
    result.push_back(1);
    for (auto it : prime_factorize(n, primes)) {
        ll p; int k; tie(p, k) = it;
        int size = result.size();
        REP (y, k) {
            REP (x, size) {
                result.push_back(result[y * size + x] * p);
            }
        }
    }
    return result;
}

vector<ll> list_prime_factors(ll n, vector<int> const & primes) {
    vector<ll> result;
    for (int p : primes) {
        if (n < p *(ll) p) break;
        while (n % p == 0) {
            result.push_back(p);
            n /= p;
        }
    }
    if (n != 1) result.push_back(n);
    return result;
}

/**
 * @brief fully factorize all numbers in [0, n) with O(n log log n)
 */
vector<vector<int> > extended_sieve_of_eratosthenes(int n) {
    vector<vector<int> > prime_factors(n + 1);
    REP3 (i, 2, n) {
        if (prime_factors[i].empty()) {
            for (int k = i; k < n; k += i) {
                prime_factors[k].push_back(i);
            }
        }
    }
    return prime_factors;
}

/**
 * @note O(sqrt(n))
 */
map<ll, int> prime_factorize1(ll n) {
    map<ll, int> factors;
    for (int p : { 2, 3, 5 }) {
        while (n % p == 0) {
            n /= p;
            ++ factors[p];
        }
    }
    for (int p = 6; (ll)p * p <= n; p += 6) {
        for (int q : { p + 1, p + 5 }) {
            while (n % q == 0) {
                n /= q;
                ++ factors[q];
            }
        }
    }
    if (n) {
        ++ factors[n];
    }
    return factors;
}

vector<vector<int> > sieve_prime_factors(int n) {
    vector<vector<int> > ps(n);
    REP3 (a, 2, n) {
        if (ps[a].empty()) {
            for (int b = 2 * a; b < n; b += a) {
                for (int b1 = b; b1 % a == 0; b1 /= a) {
                    ps[b1].push_back(a);
                }
            }
        }
    }
    return ps;
}
```

5.3 number/matrix.inc.cpp

```
template <typename T>
vector<vector<T> > operator * (vector<vector<T> > const & a, vector<vector<T> > const & b) {
    int n = a.size();
    vector<vector<T> > c = vectors(n, n, T());
    REP (y, n) REP (z, n) REP (x, n) c[y][x] += a[y][z] * b[z][x];
    return c;
}
template <typename T>
vector<T> operator * (vector<vector<T> > const & a, vector<T> const & b) {
    int n = a.size();
    vector<T> c(n);
    REP (y, n) REP (z, n) c[y] += a[y][z] * b[z];
    return c;
}
template <typename T>
vector<vector<T> > unit_matrix(int n) {
    vector<vector<T> > e = vectors(n, n, T());
    REP (i, n) e[i][i] = 1;
    return e;
}
template <typename T>
vector<vector<T> > zero_matrix(int n) {
    vector<vector<T> > o = vectors(n, n, T());
    return o;
}

template <typename T>
T determinant(vector<vector<T> > a) {
    int n = a.size();
    REP (z, n) { // make A upper trianglar
        if (a[z][z] == 0) { // swap rows to avoid zero-division
            int x = z + 1;
            for (; x < n; ++ x) {
                if (a[x][z] != 0) {
                    a[z].swap(a[x]);
                    break;
                }
```

```
            }
            if (x == n) return 0; // A is singular
        }
        REP3 (y, z + 1, n) {
            T k = a[y][z] / a[z][z];
            REP3 (x, z + 1, n) {
                a[y][x] -= k * a[z][x]; // elim
            }
            a[y][z] = 0;
        }
    }
    T acc = 1;
    REP (z, n) acc *= a[z][z]; // product of the diagonal elems
    return acc;
}

template <class T>
vector<vector<T> > small_matrix(vector<vector<T> > const & a) {
    int n = a.size();
    assert (n >= 1);
    auto b = a;
    b.resize(n - 1);
    REP (y, n - 1) {
        b[y].resize(n - 1);
    }
    return b;
}

template <typename T>
vector<T> gaussian_elimination(vector<vector<T> > f, vector<T> x) {
    int n = x.size();
    REP (y, n) {
        int pivot = y;
        while (pivot < n and abs(f[pivot][y]) < eps) ++ pivot;
        assert (pivot < n);
        swap(f[y], f[pivot]);
        x[y] /= f[y][y];
        REP3 (x, y + 1, n) f[y][x] /= f[y][y];
        f[y][y] = 1;
        REP (ny, n) if (ny != y) {
            x[ny] -= f[ny][y] * x[y];
            REP3 (x, y + 1, n) f[ny][x] -= f[ny][y] * f[y][x];
            f[ny][y] = 0;
        }
    }
    return x;
}

constexpr double eps = 1e-8;
template <typename T>
vector<vector<T> > inverse_matrix(vector<vector<T> > f) {
    int n = f.size();
    vector<vector<T> > g = unit_matrix<T>(n);
    REP (y, n) {
        int pivot = y;
        while (pivot < n and abs(f[pivot][y]) < eps) ++ pivot;
        assert (pivot < n);
        swap(f[y], f[pivot]);
        REP (x, n) g[y][x] /= f[y][y];
        REP3 (x, y + 1, n) f[y][x] /= f[y][y];
        f[y][y] = 1;
        REP (ny, n) if (ny != y) {
            REP (x, n) g[ny][x] -= f[ny][y] * g[y][x];
            REP3 (x, y + 1, n) f[ny][x] -= f[ny][y] * f[y][x];
            f[ny][y] = 0;
        }
    }
    return g;
}
unittest {
    vector<vector<double> > f { { 1, 2 }, { 3, 4 } };
    auto g = f * inverse_matrix(f);
    assert (abs(g[0][0] - 1) < eps);
    assert (abs(g[0][1]    ) < eps);
    assert (abs(g[1][0]    ) < eps);
    assert (abs(g[1][1] - 1) < eps);
}

template <typename T>
vector<vector<T> > powmat(vector<vector<T> > x, ll y) {
    int n = x.size();
    auto z = unit_matrix<T>(n);
    for (ll i = 1; i <= y; i <<= 1) {
        if (y & i) z = z * x;
        x = x * x;
    }
    return z;
}

template <typename T>
vector<vector<T> > concat_matrix_vector(vector<vector<T> > const & f, vector<T> const & x) {
    int h = f.size();
    int w = f.fornt().size();
    vector<vector<T> > fx(h);
    REP (y, h) {
        fx[y].resize(w + 1);
        copy(whole(f[y]), fx[y].begin());
        fx[y][w] = x[y];
    }
    return fx;
}
```

5.4 number/extgcd.inc.cpp

```
/**
 * @brief extended gcd
 * @description for given a and b, find x, y and gcd(a, b) such that ax + by = 1
 * @note O(log n)
 * @see https://topcoder.g.hatena.ne.jp/spaghetti_source/20130126/1359171466
 */
tuple<ll, ll, ll> extgcd(ll a, ll b) {
    ll x = 0, y = 1;
    for (ll u = 1, v = 0; a; ) {
        ll q = b / a;
        x -= q * u; swap(x, u);
        y -= q * v; swap(y, v);
        b -= q * a; swap(b, a);
    }
    return make_tuple(x, y, b);
}
unittest {
    random_device device;
    default_random_engine gen(device());
    REP (iteration, 1000) {
        ll a = uniform_int_distribution<ll>(1, 10000)(gen);
```

```
        ll b = uniform_int_distribution<ll>(1, 10000)(gen);
        ll x, y, d; tie(x, y, d) = extgcd(a, b);
        assert (a * x + b * y == d);
        assert (d == __gcd(a, b));
    }
}

/**
 * @note recursive version (slow)
 */
pair<int, int> extgcd_recursive(int a, int b) {
    if (b == 0) return { 1, 0 };
    int na, nb; tie(na, nb) = extgcd(b, a % b);
    return { nb, na - a/b * nb };
}

/**
 * @note x and n must be relatively prime
 * @note O(log n)
 */
ll modinv(ll x, ll n) {
    assert (1 <= x and x < n);
    ll y, d; tie(y, ignore, d) = extgcd(x, n);
    if (d != 1) return 0;  // no inverse
    assert (x * y % n == 1);
    return (y % n + n) % n;
}

/**
 * @brief chinese remainder theorem
 * @note the unit element is (0, 1)
 */
pair<ll, ll> crt(pair<ll, ll> eqn1, pair<ll, ll> eqn2) {
    ll x1, m1; tie(x1, m1) = eqn1;
    ll x2, m2; tie(x2, m2) = eqn2;
    ll x = x1 + m1 * (x2 - x1) * modinv(m1 % m2, m2);
    ll m = m1 * m2;
    return { (x % m + m) % m, m };
}

ll multmod(ll a, ll b, ll m) {
    a = (a % m + m) % m;
    b = (b % m + m) % m;
    ll c = 0;
    REP (i, 63) {
        if (b & (1ll << i)) {
            c += a;
            if (c > m) c -= m;
        }
        a *= 2;
        if (a > m) a -= m;
    }
    return c;
}

pair<ll, ll> crt(pair<ll, ll> eqn1, pair<ll, ll> eqn2) {
    ll x1, m1; tie(x1, m1) = eqn1;
    ll x2, m2; tie(x2, m2) = eqn2;
    if (m1 == 0 or m2 == 0) return make_pair(0ll, 0ll);
    assert (1 <= m1 and 1 <= m2);
    ll m1_inv, d; tie(m1_inv, ignore, d) = extgcd(m1, m2);
    if ((x1 - x2) % d) return make_pair(0ll, 0ll);
    ll m = m1 * m2 / d;
    // ll x = x1 + (m1 / d) * (x2 - x1) % m * (m1_inv % m) % m;
    ll x = x1 + multmod(multmod(m1 / d, x2 - x1, m), m1_inv, m);
    return make_pair((x % m + m) % m, m);
}
```

6 string

6.1 string/palindrome.inc.cpp

```
// http://snuke.hatenablog.com/entry/2014/12/02/235837
vector<int> manacher(string const & s) { // radiuses of odd palindromes, O(N)
    int n = s.length();
    vector<int> r(n);
    int i = 0, j = 0;
    while (i < n) {
        while (i-j >= 0 and i+j < n and s[i-j] == s[i+j]) ++ j;
        r[i] = j;
        int k = 1;
        while (i-k >= 0 and i+k < n and k+r[i-k] < j) {
            r[i+k] = r[i-k];
            ++ k;
        }
        i += k;
        j -= k;
    }
    return r;
}
vector<int> odd_palindrome_length(string const & s) {
    int n = s.length();
    vector<int> r = manacher(s);
    vector<int> l(n);
    repeat (i,n) l[i-r[i]+1] = 2*r[i]-1;
    repeat (i,n-1) setmax(l[i+1], l[i]-2);
    return l;
}
vector<int> even_palindrome_length(string const & s) {
    int n = s.length();
    string t(2*n+1, '\0');
    repeat (i,n) t[2*i+1] = s[i];
    vector<int> r = manacher(t);
    vector<int> l(n);
    repeat (i,n) if (r[2*i+2] >= 3) l[i-r[2*i+2]/2+1] = r[2*i+2]-1;
    repeat (i,n-1) setmax(l[i+1], l[i]-2);
    return l;
}
```

6.2 string/suffix-array.inc.cpp

```
/**
 * @brief suffix array
 * @note O(N (\log N)^2), Manber & Myers,
 * @note sa[i] is the index of i-th smallest substring of s, s[sa[i], N)
 * @note rank[i] is the rank of substring s[i, N)
 */
void suffix_array(string const & s, vector<int> & sa, vector<int> & rank) {
    int n = s.length();
    sa.resize(n + 1);
```

```
    rank.resize(n + 1);
    REP (i, n + 1) {
        sa[i] = i;
        rank[i] = i < n ? s[i] : -1;
    }
    auto rankf = [&](int i) { return i <= n ? rank[i] : -1; };
    vector<int> nxt(n + 1);
    for (int k = 1; k <= n; k <<= 1) {
        auto cmp = [&](int i, int j) { return make_pair(rank[i], rankf(i + k)) < make_pair(rank[j], rankf(j + k)); };
        sort(sa.begin(), sa.end(), cmp);
        nxt[sa[0]] = 0;
        REP3 (i, 1, n + 1) {
            nxt[sa[i]] = nxt[sa[i - 1]] + (cmp(sa[i - 1], sa[i]) ? 1 : 0);
        }
        rank.swap(nxt);
    }
}

/**
 * @description lcp[i] is the length of the common prefix between i-th and (i+1)-th substring of s
 * @note O(N),
 * @note lcp.size() == n, != n + 1
 */
vector<int> longest_common_prefix_array(string const & s, vector<int> const & sa, vector<int> const & rank) {
    int n = s.length();
    vector<int> lcp(n);
    int h = 0;
    lcp[0] = 0;
    REP (i, n) {
        int j = sa[rank[i] - 1];
        if (h > 0) -- h;
        while (j + h < n and i + h < n and s[j + h] == s[i + h]) ++ h;
        lcp[rank[i] - 1] = h;
    }
    return lcp;
}

unittest {
    constexpr int n = 100;
    default_random_engine gen;
    string s;
    REP (i, n) s += uniform_int_distribution<char>('a', 'z')(gen);
    vector<int> sa, rank; suffix_array(s, sa, rank);
    vector<int> lcp = longest_common_prefix_array(s, sa, rank);
    REP (i, n + 1) {
        assert (sa[rank[i]] == i);
    }
    auto compute_lcp = [](string s, string t) {
        int i = 0;
        while (i < s.length() and i < t.length() and s[i] == t[i]) ++ i;
        return i;
    };
    REP (i, n) {
        assert (lcp[i] == compute_lcp(s.substr(sa[i]), s.substr(sa[i + 1])));
    }
}

int sa_lower_bound(string const & s, vector<int> const & sa, string const & t) { // returns an index on suffix array
    int n = s.size();
    int l = 0, r = n+1; // (l, r]
    while (l + 1 < r) {
        int m = (l + r) / 2;
        (s.compare(sa[m], string::npos, t) < 0 ? l : r) = m;
    }
    return r;
}
int sa_prefix_upper_bound(string const & s, vector<int> const & sa, string const & t) { // returns an index on suffix array
    int n = s.size();
    int l = 0, r = n+1; // (l, r]
    while (l + 1 < r) {
        int m = (l + r) / 2;
        (s.compare(sa[m], t.size(), t) <= 0 ? l : r) = m;
    }
    return r;
}
int sa_match(string const & target, string const & pattern, vector<int> const & sa, segment_tree<int> const & lcp) { // O(m \log n)
    int l = sa_lower_bound(target, sa, pattern);
    int r = sa_prefix_upper_bound(target, sa, pattern);
    return r - l;
}
```

6.3 string/aho-corasick.inc.cpp

```
struct pma_t { // Aho-Corasick
    pma_t* next[26];
    pma_t* fail;
    int accept;
};
pma_t *ac_new() {
    pma_t *pma = new pma_t;
    *pma = {};
    return pma;
}
void ac_add_pattern(pma_t *pma, string const & pattern) { // O(m)
    for (char c : pattern) { // make trie
        if (pma->next[c-'a'] == nullptr) {
            pma->next[c-'a'] = ac_new();
        }
        pma = pma->next[c-'a'];
    }
    pma->accept += 1;
}
void ac_construct_links(pma_t *pma) { // O(m)
    queue<pma_t *> que; // make failure link using bfs
    repeat (c,26) {
        if (pma->next[c]) {
            pma->next[c]->fail = pma;
            que.push(pma->next[c]);
        } else {
            pma->next[c] = pma;
        }
    }
    while (not que.empty()) {
        pma_t *now = que.front(); que.pop();
        repeat (c,26) {
            if (now->next[c]){
                pma_t *nxt = now->fail;
                while (not nxt->next[c]) nxt = nxt->fail;
                now->next[c]->fail = nxt->next[c];
                now->next[c]->accept += nxt->next[c]->accept;
                que.push(now->next[c]);
            }
        }
    }
}
```

```
int ac_match(pma_t *pma, string const & target) { // O(n)
    int result = 0;
    for (char c : target) {
        while (not pma->next[c-'a']) pma = pma->fail;
        pma = pma->next[c-'a'];
        result += pma->accept;
    }
    return result;
}
void ac_delete(pma_t *pma, unordered_set<pma_t *> & deleted) {
    deleted.insert(pma);
    repeat (c,26) if (pma->next[c] and not deleted.count(pma)) {
        ac_delete(pma->next[c], deleted);
    }
    delete pma;
}
void ac_delete(pma_t *pma) {
    unordered_set<pma_t *> deleted;
    ac_delete(pma, deleted);
}
```

7 geometry

7.1 geometry/convex-hull.inc.cpp

```
double dot(complex<double> a, complex<double> b) { return real(a * conj(b)); }
double cross(complex<double> a, complex<double> b) { return imag(conj(a) * b); }
int ccw(complex<double> a, complex<double> b, complex<double> c) {
    b -= a;
    c -= a;
    if (cross(b, c) > 0) return +1;  // counter clockwise
    if (cross(b, c) < 0) return -1;  // clockwise
    if (dot(b, c) < 0)   return +2;  // c--a--b on line
    if (abs(b) < abs(c)) return -2;  // a--b--c on line
    return 0;
}
/**
 * @see http://www.prefield.com/algorithm/geometry/convex_hull.html
 */
vector<complex<double> > convex_hull(vector<complex<double> > ps) {
    int n = ps.size();
    if (n <= 2) return ps;
    int k = 0;
    sort(ps.begin(), ps.end(), [&](complex<double> const a, complex<double> const b) { return make_pair(a.real(), a.imag()) < make_pair(b.real(), b.imag()); });
    vector<complex<double> > ch(2 * n);
    for (int i = 0; i < n; ch[k ++] = ps[i ++]) {  // lower-hull
        while (k >= 2 and ccw(ch[k - 2], ch[k - 1], ps[i]) <= 0) -- k;
    }
    for (int i = n - 2, t = k + 1; i >= 0; ch[k ++] = ps[i --]) {  // upper-hull
        while (k >= t and ccw(ch[k - 2], ch[k - 1], ps[i]) <= 0) -- k;
    }
    ch.resize(k - 1);
    return ch;
}
```

8 utils

8.1 utils/binsearch.inc.cpp

```
/**
 * @brie a flexible binary search
 * @param[in] p  a monotone predicate defined on [l, r)
 * @return  \min \{ x \in [l, r) \mid p(x) \}, or r if it doesn't exist
 */
template <typename UnaryPredicate>
int64_t binsearch(int64_t l, int64_t r, UnaryPredicate p) {
    assert (l <= r);
    -- l;
    while (r - l > 1) {
        int64_t m = l + (r - l) / 2;
        (p(m) ? r : l) = m;
    }
    return r;
}

unittest {
    for (int l : { 0, 1, 2, 3 }) {
        for (int r : { 8, 9, 10, 11 }) {
            assert (binsearch(l, r, [&](int n) { assert (l <= n and n < r); return true;   }) == l);
            assert (binsearch(l, r, [&](int n) { assert (l <= n and n < r); return false;  }) == r);
            REP3 (i, l, r + 1) {
                assert (binsearch(l, r, [&](int n) { assert (l <= n and n < r); return n >= i; }) == i);
            }
        }
    }
}

/**
 * @return  \max \{ x \in (l, r] \mid p(x) \}, or l if it doesn't exist
 */
template <typename UnaryPredicate>
int64_t binsearch_max(int64_t l, int64_t r, UnaryPredicate p) {
    assert (l <= r);
    ++ r;
    while (r - l > 1) {
        int64_t m = l + (r - l) / 2;
        (p(m) ? l : r) = m;
    }
    return l;
}

unittest {
    for (int l : { 0, 1, 2, 3 }) {
        for (int r : { 8, 9, 10, 11 }) {
            assert (binsearch_max(l, r, [&](int n) { assert (l < n and n <= r); return false; }) == l);
            assert (binsearch_max(l, r, [&](int n) { assert (l < n and n <= r); return true;  }) == r);
            REP3 (i, l, r + 1) {
                assert (binsearch_max(l, r, [&](int n) { assert (l < n and n <= r); return n <= i; }) == i);
            }
        }
    }
}
```

8.2 utils/longest-increasing-subsequence.inc.cpp

```cpp
template <typename T>
vector<T> longest_strict_increasing_subsequence(vector<T> const & xs) {
    vector<T> l;  // l[i] is the last element of the increasing subsequence whose length is i + 1
    l.push_back(xs.front());
    for (auto && x : xs) {
        auto it = lower_bound(l.begin(), l.end(), x);
        if (it == l.end()) {
            l.push_back(x);
        } else {
            *it = x;
        }
    }
    return l;
}

template <typename T>
vector<T> longest_weak_increasing_subsequence(vector<T> const & xs) {
    vector<T> l;
    for (auto && x : xs) {
        auto it = upper_bound(l.begin(), l.end(), x);
        if (it == l.end()) {
            l.push_back(x);
        } else {
            *it = x;
        }
    }
    return l;
}
```

8.3 utils/dice.inc.cpp

```cpp
struct dice_t { // regular hexahedron group
    //       ------
    //      \      \      4
    //     / \   C  \    2156
    //    / A \______\    3 ^
    //    \ A /    B /   ^^ |
    //     \ /   B  /    ab bottom
    //      v__B___/
    int a, b; // in [1, 6]
    int c() const {
        static const int table[6][6] = {
            { 0, 3, 5, 2, 4, 0 },
            { 4, 0, 1, 6, 0, 3 },
            { 2, 6, 0, 0, 1, 5 },
            { 5, 1, 0, 0, 6, 2 },
            { 3, 0, 6, 1, 0, 4 },
            { 0, 4, 2, 5, 3, 0 },
        };
        assert (table[a-1][b-1] != 0);
        return  table[a-1][b-1];
    }
};
dice_t rotate_up(   dice_t dice) { return (dice_t) { dice.a, 7 - dice.c() }; }
dice_t rotate_right(dice_t dice) { return (dice_t) { 7 - dice.c(), dice.b }; }
dice_t rotate_down( dice_t dice) { return (dice_t) { dice.a, dice.c() }; }
dice_t rotate_left( dice_t dice) { return (dice_t) { dice.c(), dice.b }; }
bool operator == (dice_t x, dice_t y) { return x.a == y.a and x.b == y.b; }
```

8.4 utils/subset.inc.cpp

```cpp
/**
 * @sa https://kimiyuki.net/blog/2017/07/16/enumerate-sets-with-bit-manipulation/
 */

// for a set z, list y \subseteq z, ascending order
for (int y = 0; ; y = (y - z) & z) {
    ...
    if (y == z) break;
}

// for a set z, list y \subseteq z, descending order
for (int y = z; ; y = (y - 1) & z) {
    ...
    if (y == 0) break;
}

// for a set x and an ordinal n, list y s.t. x \subseteq y \subseteq n
or (int y = x; y < (1 << n); y = (y + 1) | x) {
    ...
}

// for an ordinal n and integer k, list x \subseteq n s.t. \|x\| = k
for (int x = (1 << k) - 1; x < (1 << n); ) {
    ...
    int t = x | (x - 1);
    x = (t + 1) | (((~ t & - ~ t) - 1) >> (__builtin_ctz(x) + 1));
}
```