University of Warsaw

ACM ICPC Team Reference Document

Kamil Dębowski
Marek Sommer
Mateusz Radecki

kamil/1_geo_lib.cpp

```
/* 'const' can be added and for speed create += and similar
INFO 1
In methods 'below()' use '< eps' only if you HAVE TO avoid unnecessary objects,
e.g. if you need the exact size of CH. Using 'eps' may cause discarding
objects that only slightly improve the result, so try to avoid 'eps'.
INFO 2 -- In 'L3::fix()' uncomment scaling by gcd or sqrt, if needed.
INFO 3 -- How to find an upper envelope of lines Ax+By+C=0, where B > 0.
Sort lines by slope increasingly (ties: lower line first). Then a determinant
of three consecutive non-parallel lines is positive iff all three lines
are visible from the above, i.e. they form an upside down 'A' shape. */
template<typename T> T K(T a) { return a * a; }
#define K(a) K(1LL * (a))
typedef long long ll; // can be changed to 'long double'
typedef long double ld;
// const ld PI = 2 * acos(0);
const ld eps = 1e-12;
#pragma GCC diagnostic ignored "-Wnarrowing"
struct P {
  ll x, y; // + ... (trivial operators)
  ll dot(P b) { return x * b.x + y * b.y; }
  ld len() { return sqrt(K(x) + K(y)); }
  P scaleTo(ld to) { return *this * (to / len()); }
  ld dist(P & b) { return (*this - b).len(); }
  P rotate90() { return P{-y, x}; }
  ld angle() { return atan2(y, x); }
  P rotate(ld ang) {
    ld c = cos(ang), s = sin(ang);
    return P{x * c - y * s, x * s + y * c};
  }
  // '<' and 'below()' needed for Convex Hull
  bool operator < (P he) { return make_pair(x, y) < make_pair(he.x, he.y); }
  bool below(P a, P b) { return (b - a) * (*this - a) <= 0/*eps*/; } //INFO 1
  // Internal/External Similitude Center
  P apol_in(P b, ld ratio) { // ratio = dist()/he.dist()
    return (*this + b * ratio) / (1 + ratio);
  }
  P apol_out(P b, ld ratio) {
    return (*this - b * ratio) / (1 - ratio);
  }
};
struct L2 {
  P one, two;
  // P p[2]; P & operator [](int i) { return p[i]; }
  // const P & operator [](int i) const { return p[i]; }
  P dir() { return two - one; }
  P normal() { return dir().rotate90(); }
  ld dist(P he) {
    return abs((he - one) * (he - two)) / one.dist(two);
  }
  ld segDist(P he) { // epsilon not needed, but it would work too
    if((he - two) * normal() < 0 && normal() * (he - one) < 0)
      return dist(he);
    return min(one.dist(he), two.dist(he));
  }
  P inter(L2 he) {
    P A = dir(), B = he.dir();
    ll den = A * B;
    assert(abs(den) > eps); // parallel, maybe equal
    return (A * (he.one * he.two) - B * (one * two)) * (1.0 / den);
    // A = (x1*y2-y1*x2)*(x3-x4)-(x1-x2)*(x3*y4-y3*x4)
    // A'= (x1*y2-y1*x2)*(y3-y4)-(y1-y2)*(x3*y4-y3*x4)
    // B = (x1-x2)*(y3-y4)-(y1-y2)*(x3-x4)
    // return P{A / B, A' / B};
  }
  P project(P he) {
    P unit_normal = normal().scaleTo(1);
    return he + unit_normal * unit_normal.dot(one - he);
  }
  P reflect(P he) { return project(he) * 2 - he; }
// for CH: sort by slope; below() : change to L3 or compare 'x' of intersections
};
L2 toL2(ll a, ll b, ll c) {
  P first;
  if(abs(b) > eps) first = P{0, (ld) -c / b};
  else if(abs(a) > eps) first = P{(ld) -c / a, 0};
  else assert(false);
  return L2{first, first + P{b, -a}};
}
ll det(ll t[3][3]) { // for CH of lines Ax+By+C=0
  ll s = 0;
  for(int i = 0; i < 3; ++i)
    for(int j = i + 1, mul = 1; j != i + 3; ++j, mul -= 2)
      s += t[0][i] * t[1][j%3] * t[2][3-i-j%3] * mul;
  return s;
}
struct L3 {
  // a * x + b * y + c = 0, assert(b > 0 || (b == 0 && a > 0))
  ll a, b, c;
  L3 fix() { // <done>TODO, test it</done>
    assert(abs(b) > eps || abs(a) > eps);
    ll g = (b > eps || (abs(b) < eps && a > eps)) ? 1 : -1;
    // __gcd(x,0) is undef-beh, http://codeforces.com/blog/entry/13410
    // if(is_integral<ll>::value) g *= abs(__gcd(c, __gcd(a?b:a, a?a:b)));
    // if(is_floating_point<ll>::value) g *= sqrt(K(a) + K(b));
    return L3{a / g, b / g, c / g};
  }
  ld dist(P he) {
    return abs(a * he.x + b * he.y + c) / sqrt(K(a) + K(b));
  }
  P dir() { return P{b, -a}; }
  P normal() { return P{a, b}; } // equivalently: dir().rotate90()
  P project(P he) {
    ld den = K(a) + K(b); // non-integer because we need division
    return P{(b * (b * he.x - a * he.y) - a * c) / den,
         (a * (a * he.y - b * he.x) - b * c) / den };
  }
  P reflect(P he) { return project(he) * 2 - he; }
  P inter(L3 he) {
    #define Q(i, j) (i * he.j - j * he.i)
    ll den = Q(a, b);
    assert(abs(den) > 1e-14); // parallel, maybe equal
    return P{Q(b, c), Q(c, a)} * (1.0 / den);
    #undef Q
```


K M R

```
  }
  bool operator < (L3 he) {
    // produces the order for finding an upper envelope
    // assert(b > 0 && he.b > 0);
    // a / b < he.a / he.b, ties: -c/b < ...
    if(abs(a * he.b - b * he.a) < eps) return b * he.c < c * he.b;
              // <done>test it</done>
    return a * he.b < b * he.a;
  }
  bool below(L3 A, L3 C) {
    ll t[3][3] = { {A.a,A.b,A.c}, {a,b,c}, {C.a,C.b,C.c} };
    return det(t) <= 0/*eps*/; // WARN1
  }
};
L3 toL3(P one, P two) {
  ll a = two.y - one.y;
  ll b = one.x - two.x;
  return L3{a, b, -(a * one.x + b * one.y)}.fix();
}
struct Circle {
  P o;
  ld r;
  vector<P> tangency(P he) {
    ld d = o.dist(he);
    if(abs(d - r) < eps) return vector<P>{he};
    if(d < r) return vector<P>{};
    ld alpha = asin(r / d);
    P vec = (o - he) * sqrt(1 - K(r / d)); // *sqrt(d^2-r^2)/d
    // faster: compute 'sin' and 'cos' once
    return vector<P>{he + vec.rotate(alpha), he + vec.rotate(-alpha)};
  }
  vector<P> inter(L3 he) {
    P prim = he.project(o);
    ld d = prim.dist(o);
    if(d >= r + eps) return vector<P>{};
    if(abs(d - r) <= eps) return vector<P>{prim};
    P vec = he.dir().scaleTo(sqrt(K(r) - K(d)));
    return vector<P>{prim + vec, prim - vec};
  }
  vector<P> inter(Circle he) {
    return inter(L3{2*(o.x-he.o.x), 2*(o.y-he.o.y),
        K(r)-K(he.r)-K(o.x)-K(o.y)+K(he.o.x)+K(he.o.y)});
  }
  vector<L2> tangency(Circle he) {
    vector<L2> ret;
    ld ratio = r / he.r;
    auto considerPoint = [&] (P p) {
      vector<P> one = tangency(p), two = he.tangency(p);
      for(int i = 0; i < (int) min(one.size(), two.size()); ++i)
        ret.push_back(L2{one[i], two[i]});
    };
    if(abs(r - he.r < 1e-9)) { // beka z nawiasow XD
      P dir = (he.o - o).rotate90().scaleTo(r);
      for(int tmp : {1, -1})
        ret.push_back(L2{o + dir * tmp, he.o + dir * tmp});
    }
    else considerPoint(o.apol_out(he.o, ratio));
    // the following will produce 2/1/0 pairs
    // for distant/touching/intersecting circles
    considerPoint(o.apol_in(he.o, ratio));
    return ret;
  }
};
Circle apollonius(P a, P b, ld ratio) { // ratio = distA / distB
  assert(ratio >= 0);
  assert(abs(ratio - 1) > 1e-14); // straight line through (a+b)/2
  P in = a.apol_in(b, ratio), out = a.apol_out(b, ratio);
  return Circle{(in + out) / 2, in.dist(out) / 2};
}
Point Bary(Point A, Point B, Point C, LD a, LD b, LD c) {
  return (A * a + B * b + C * c) / (a + b + c);
}
Point Centroid(Point A, Point B, Point C) { return Bary(A, B, C, 1, 1, 1); }
Point Circumcenter(Point A, Point B, Point C) {
  LD a = (B - C).SqNorm(), b = (C - A).SqNorm(), c = (A - B).SqNorm();
  return Bary(A, B, C, a * (b + c - a), b * (c + a - b), c * (a + b - c));
}
Point Incenter(Point A, Point B, Point C) {
  return Bary(A, B, C, (B - C).Norm(), (A - C).Norm(), (A - B).Norm());
}
Point Orthocenter(Point A, Point B, Point C) {
  LD a = (B - C).SqNorm(), b = (C - A).SqNorm(), c = (A - B).SqNorm();
  return Bary(A, B, C, (a+b-c)*(c+a-b), (b+c-a)*(a+b-c), (c+a-b)*(b+c-a));
}
Point Excenter(Point A, Point B, Point C) { // opposite to A
  LD a = (B - C).Norm(), b = (A - C).Norm(), c = (A - B).Norm();
  return Bary(A, B, C, -a, b, c);
}
```

kamil/2_simpson.cpp

```
// Either run integral(A, B) once or split the interval [A, B] into up to ~1000
// smaller intervals -- if the function f behaves oddly or the interval is long.
ld simp(ld low, ld high, const ld * old, vector<ld> & nowe) {
  const int n = 500; // n must be even!!! Try n = 2 and n = 10.
  nowe.resize(n + 1);
  ld total = 0, jump = (high - low) / n;
  for(int i = 0; i <= n; ++i) {
    int mul = i == 0 || i == n ? 1 : 2 + i % 2 * 2; // 1 2 4 2 4 ... 2 1
    nowe[i] = !old || i % 2 ? f(low + i * jump) : old[i/2];
    total += nowe[i] * mul;   // uses a global function ld f(ld x)
  }
  return total * (high - low) / n / 3;
}
ld rec(ld low, ld high, ld prv, const vector<ld> & old) {
  ld mid = (low + high) / 2;
  vector<ld> left, right;
  ld L = simp(low, mid, old.data(), left);
  ld R = simp(mid, high, old.data() + old.size() / 2, right);
  if(abs(L + R - prv) < 1e-12L) return L + R; // eps ~ required abs precision
  return rec(low, mid, L, left) + rec(mid, high, R, right);
}
ld integral(ld low, ld high) {
  vector<ld> old;
  ld prv = simp(low, high, 0, old);
  return rec(low, high, prv, old);    }
```

kamil/3_fft.cpp

```cpp
/* Prec. error max_ans/1e15 (2.5e18) for (long) doubles, so int rounding works
for doubles with answers 0.5e15, e.g. for sizes 2^20 and RANDOM ints in [0,45k],
assuming DBL_MANT_DIG=53 and LDBL_MANT_DIG=64. Consider normalizing and brute.*/
#define REP(i,n) for(int i = 0; i < int(n); ++i)
typedef double ld; // 'long double' is 2.2 times slower
struct C { ld real, imag;
  C operator * (const C & he) const {
    return C{real * he.real - imag * he.imag,
        real * he.imag + imag * he.real};
  }
  void operator += (const C & he) { real += he.real; imag += he.imag; }
};
void dft(vector<C> & a, bool rev) {
  const int n = a.size();
  for(int i = 1, k = 0; i < n; ++i) {
    for(int bit = n / 2; (k ^= bit) < bit; bit /= 2);;;
    if(i < k) swap(a[i], a[k]);
  }
  for(int len = 1, who = 0; len < n; len *= 2, ++who) {
    static vector<C> t[30];
    vector<C> & om = t[who];
    if(om.empty()) {
      om.resize(len);
      const ld ang = 2 * acosl(0) / len;
      REP(i, len) om[i] = i%2 || !who ?
          C{cos(i*ang), sin(i*ang)} : t[who-1][i/2];
    }
    for(int i = 0; i < n; i += 2 * len)
      REP(k, len) {
         const C x = a[i+k], y = a[i+k+len]
            * C{om[k].real, om[k].imag * (rev ? -1 : 1)};
        a[i+k] += y;
        a[i+k+len] = C{x.real - y.real, x.imag - y.imag};
      }
  }
  if(rev) REP(i, n) a[i].real /= n;
}
template<typename T>vector<T> multiply(const vector<T> & a, const vector<T> & b,
    bool split = false, bool normalize = false) {
  if(a.empty() || b.empty()) return {};
  T big = 0; if(normalize) { // [0,B] into [-B/2, B/2]
    assert(a.size() == b.size()); // equal size!!!
    for(T x : a) big = max(big, x);
    for(T x : b) big = max(big, x);
    big /= 2;
  }
  int n = a.size() + b.size();
  vector<T> ans(n - 1);
  /* if(min(a.size(),b.size()) < 190) { // BRUTE FORCE
    REP(i, a.size()) REP(j, b.size()) ans[i+j] += a[i]*b[j];
    return ans; } */
  while(n&(n-1)) ++n;
  auto speed = [&](const vector<C> & w, int i, int k) {
    int j = i ? n - i : 0, r = k ? -1 : 1;
    return C{w[i].real + w[j].real * r, w[i].imag
        - w[j].imag * r} * (k ? C{0, -0.5} : C{0.5, 0});
  };
  if(!split) { // standard fast version
    vector<C> in(n), done(n);
    REP(i, a.size()) in[i].real = a[i] - big;
    REP(i, b.size()) in[i].imag = b[i] - big;
    dft(in, false);
    REP(i, n) done[i] = speed(in, i, 0) * speed(in, i, 1);
    dft(done, true);
    REP(i, ans.size()) ans[i] = is_integral<T>::value ?
        llround(done[i].real) : done[i].real;
  //REP(i,ans.size())err=max(err,abs(done[i].real-ans[i]));
  }
  else { // Split big INTEGERS into pairs a1*M+a2,
    const T M = 1<<15; // where M = sqrt(max_absvalue).
    vector<C> t[2]; // This version is 2.2-2.5 times slower.
    REP(x, 2) {
      t[x].resize(n);
      auto & in = x ? b : a; // below use (in[i]-big) if normalized
      REP(i, in.size()) t[x][i]=C{ld(in[i]%M), ld(in[i]/M)};
      dft(t[x], false);
    }
    T mul = 1;
    for(int s = 0; s < 3; ++s, mul *= M) {
      vector<C> prod(n);
      REP(x, 2) REP(y, 2) if(x + y == s) REP(i, n)
        prod[i] += speed(t[0], i, x) * speed(t[1], i, y);
      dft(prod, true); // remember: llround(prod[i].real)%MOD*mul !!!
      REP(i, ans.size()) ans[i]+= llround(prod[i].real)*mul;
    }
  }
  if(normalize) {
    T so_far = 0;
    REP(i, ans.size()) {
      if(i < (int) a.size()) so_far += a[i] + b[i];
      else so_far -= a[i-a.size()] + b[i-a.size()];
      ans[i] += big * so_far - big * big * min(i + 1, (int) ans.size() - i);
    }
  }
  return ans;
}
// compressing up to 2^17 bits into 2 times smaller vectors
const ll M = 1 << 17; // M can be smaller if vectors are small
vector<ll> compress(const vector<ll> & a) {
  vector<ll> tmp((a.size() + 1) / 2);
  for(int i = 0; 2 * i + 1 < (int) a.size(); ++i)
    tmp[i] += a[2 * i] + a[2 * i + 1] * M;
  if(a.size() % 2) tmp.back() = a.back();
  return tmp; }
vector<ll> my_mul(const vector<ll> & a, const vector<ll> & b) {
  vector<ll> tmp = multiply(compress(a), compress(b), false);
  vector<ll> r(2 * tmp.size() + 1);
  for(int i = 0; i < (int) tmp.size(); ++i) {
    r[2*i] += tmp[i] % M; // can be sped-up with bit shifting
    r[2*i+1] += tmp[i] / M % M; r[2*i+2] += tmp[i] / M / M;
  }
  r.resize(a.size() + b.size() - 1);
  return r;                                  }
```

kamil/4_Rho_Pollarda.cpp

```cpp
//~ vector<ll> witness = {2, 7, 61}; // < 4759123141 = 4e9
vector<ll> witness = {2, 325, 9375, 28178, 450775, 9780504, 1795265022}; // < 2^64

ll mul(ll a, ll b, ll mod) {
  return (__int128) a * b % mod;
}
ll my_pow(ll a, ll b, ll mod) {
  ll res = 1;
  while(b) {
    if(b % 2) res = mul(res, a, mod);
    a = mul(a, a, mod);
    b /= 2;
  }
  return res;
}
bool test(ll n) {
  if(n == 2) return true;
  if(n < 2 || n % 2 == 0) return false;
  ll d = n - 1, s = 0;
  while(d % 2 == 0) {
    d /= 2;
    ++s;
  }
  for(auto i : witness) if(i % n) {
    ll x = my_pow(i, d, n);
    if(x == 1) continue;
    bool zlozona = true;
    for(int j = 0; j < s; ++j) {
      if(x == n - 1) {
        zlozona = false;
        break;
      }
      x = mul(x, x, n);
    }
    if(zlozona) return false;
  }
  return true;
}
ll f(ll x, ll mod, ll c) {
  ll y = mul(x, x, mod) + c;
  if(y > mod) y -= mod;
  return y;
}
void rho(ll n, vector<ll> & w) {
  if(n <= 1) return;
  if(test(n)) {
    w.push_back(n);
    return;
  }
  for(ll c = 1; true; ++c) {
    ll x = 2, y = 2, d = 1;
    while(d == 1) {
      x = f(x, n, c);
      y = f( f(y,n,c), n, c);
      d = __gcd(abs(x - y), n);
    }
    if(d < n) {
      rho(d, w);
      rho(n / d, w);
      return;
    }
  }
}
vector<ll> rozklad(ll n) {
  vector<ll> w;
  for(int i = 2; i <= 100; ++i) while(n % i == 0) {
    n /= i;
    w.push_back(i);
  }
  rho(n, w);
  sort(w.begin(), w.end());
  return w;
}
```

kamil/cf_hull_remove.cpp

```cpp
typedef long long ll;
struct P {
  ll x, y; int id;
  void read(int _id) { id = _id; scanf("%lld%lld", &x, &y); }
  void write() const { printf("(%lld,%lld)", x, y); }
  ll operator * (const P & b) const { return x * b.y - y * b.x; }
  P operator + (const P & b) const { return P{x + b.x, y + b.y}; }
  P operator - (const P & b) const { return P{x - b.x, y - b.y}; }
  bool operator < (const P & b) const {
    return make_pair(x, make_pair(y, id))
      < make_pair(b.x, make_pair(b.y, id));
  }
  bool operator == (const P & b) const {
    return make_pair(x, y) == make_pair(b.x, b.y); }
  bool under(const P & a, const P & b) const {
    return (b - a) * (*this - a) <= 0;
  }
};
struct Node {
  int a_size, b_size;
  P *a, *b, *first;
  Node *L, *R;
  Node() { a_size=b_size=0; a=b=first=0; L=R=0; }
  Node(P *p) { *this = Node(); a_size=1; first=p; }
  Node(Node *l, Node *r) { *this = Node(); L=l; R=r; act(); }
  int size() const { return a_size + b_size; }
  bool empty() const { return !first; }
  bool leaf() const { return !L; }
  void act() {
    if(leaf()) return;
    if(!L->empty()) first = L->first;
    else if(!R->empty()) first = R->first;
    else first = NULL;
    if(L->empty() || R->empty()) {
      a = L->first; // possibly NULL
      a_size = L->size();
      b = R->first;
      b_size = R->size();
```

```
  }
  else {
    a_size = b_size = 1;
    act(L, R); // modifies a_size and b_size
  }
}
private : void act(Node *inter1, Node *inter2) {
  /*if(inter1->leaf() && inter2->leaf()) {*/ //if there is no size()
  if(inter1->size() == 1 && inter2->size() == 1) {
    a = inter1->first;
    b = inter2->first;
    return;
  }
  P *A = inter1->a, *B = inter1->b;
  P *C = inter2->a, *D = inter2->b;
  if(inter1->L && (!B || (A && C && B->under(*A, *C))))
    return act(inter1->L, inter2);
  if(!C || (B && D && C->under(*B, *D)))
    return act(inter1, inter2->R);
  auto x = inter2->first->x;
  long double x1 = B->x - (A ? A->x : 0), x2 = (D ? D->x : 0) - C->x;
  if(!A || (A && D && (A->y - C->y) * x1 * x2 + x2 * (B->y - A->y)
      * (x - A->x) - x1 * (D->y - C->y) * (x - C->x) > 0)) {
    a_size += inter1->size() - inter1->R->size();
    return act(inter1->R, inter2);
  }
  b_size += inter2->size() - inter2->L->size();
  return act(inter1, inter2->L);
  /*long double x1 = B->x - A->x, x2 = D->x - C->x;
  #define remA {a_size += inter1->size() - inter1->R->size(); \
      return act(inter1 -> R, inter2);}
  #define remB return act(inter1 -> L, inter2)
  #define remC return act(inter1, inter2 -> R)
  #define remD {b_size += inter2->size() - inter2->L->size(); \
      return act(inter1, inter2 -> L);}
  if(!A) remA; if(!B) remB; if(!C) remC; if(!D) remD;
  if(B -> underLine(*A, *C)) remB;
  if(C -> underLine(*B, *D)) remC;
  auto x = inter2->first->x;
  long double x1 = B->x - A->x, x2 = D->x - C->x;
  if((A->y - C->y) * x1 * x2 + x2 * (B->y - A->y) * (x - A->x)
    - x1 * (D->y - C->y) * (x - C->x) > 0) remA else remD;*/
} public :
ll query(ll mul) const { // maximize mul*x+y
  auto evaluate = [&] (const P & p) { return mul * p.x + p.y; };
  // if(empty()) return +-INFINITY;
  if(size() == 1) return evaluate(*first);
  if(L -> empty()) return R -> query(mul);
  if(R -> empty()) return L -> query(mul);
  return (evaluate(*a) > evaluate(*b) ? L : R) -> query(mul);
}
// returns true if something was removed
bool remove(ll A, ll B, ll C) { // cut points above Ax+By+C=0
  if(empty()) return false;
  auto evaluate = [&] (const P & p) { return A * p.x + B * p.y; };
  if(leaf()) {
    if(evaluate(*first) >= -C) {
      *this = Node();
      return true;
    }
    return false;
  }
  auto left = a ? evaluate(*a) : 0, right = b ? evaluate(*b) : 0;
  bool modified = 0;
  if(a && (!b || left > right || left >= -C))
    modified |= L->remove(A,B,C);
  if(b && (!a || left < right || right >= -C))
    modified |= R->remove(A,B,C);
  if(modified) act();
  return modified;
}
void getHull(int a_skip, int b_skip, vector<P> & w) const {
  if(a_skip + b_skip >= size()) return;
  if(size() == 1) {
    assert(first);
    w.push_back(*first);
    return;
  }
  assert(L && R);
  L->getHull(a_skip, max(0, b_skip - b_size) + L->size() - a_size, w);
  R->getHull(max(0, a_skip - a_size) + R->size() - b_size, b_skip, w);
}
};
const int UPPER = 0, LOWER = 1;
const int REPS = 2; // 1 means computing UPPER only
long long done[1 << 20];
struct Hull {
  vector<Node> tr[2];
  vector<P> all;
  int pot;
  int getID(const P & a, int type) const {
    int tmp = lower_bound(all.begin(), all.end(), a) - all.begin();
    assert(all[tmp] == a); // I didn't get it in the constructor
    return tmp;
  }
  Hull(vector<P> _all) : all(_all) {
    sort(all.begin(), all.end());
    all.resize(unique(all.begin(), all.end()) - all.begin());
    pot = 1;
    while(pot < (int) all.size()) pot *= 2;
    for(int rep = 0; rep < REPS; ++rep) {
      vector<Node> & t = tr[rep];
      t.clear(); t.resize(2 * pot);
      for(int i = 1; i < pot; ++i) {
        t[i].L = & t[2*i]; t[i].R = & t[2*i+1];
      }
    }
    for(int i = 0; i < (int) all.size(); ++i) todo.push(pot + i);
  }
  priority_queue<int> todo;
  void change(const P & last) {
    for(int type = 0; type < REPS; ++type) {
      int id = getID(last, type);
      P * which_point = & all[id];
```

```
      if(type == LOWER) id = (int) all.size() - 1 - id;
      Node & tmp = tr[type][pot+id];
      if(tmp.empty()) {
        tmp.a_size = 1; tmp.first = which_point;
      }
      else {
        tmp.a_size = 0; tmp.first = NULL;
      }
      todo.push(pot + id);
    }
  }
  vector<P> get() {
    static long long T = 0;
    ++T;
    while(!todo.empty()) {
      int id = todo.top(); todo.pop();
      if(done[id] == T) continue; done[id] = T;
      tr[0][id].act(); tr[1][id].act();
      if(id != 1) todo.push(id / 2);
    }
    todo = priority_queue<int>();
    // ---
    vector<P> w;
    // printf("should = %d\n", (int) tr[0][1].size());
    tr[0][1].getHull(0, 0, w); if((int) w.size() <= 1) return w;
    w.pop_back(); tr[1][1].getHull(0, 0, w);
    w.pop_back(); return w;
  }
};
const int nax = 6e5 + 5;
int type[nax], val[nax]; P p[nax];
int main() {
  int T; scanf("%d", &T);
  while(T--) {
    int n; scanf("%d", &n);
    vector<P> w(n);
    for(int i = 0; i < n; ++i) {
      w[i].read(i+1); //w[i].id = i + 1;
    }
    Hull hull(w); set<P> s;
    for(int i = 0; i < n; ++i) if(!s.count(w[i])) {
      s.insert(w[i]);
      hull.change(w[i]);
    }
    w = hull.get();
  }
}
```

kamil/dewolaj.cpp

```
typedef long long T;
struct P {
  T x, y; int id;
  P operator - (P b) { return P{x - b.x, y - b.y}; }
  T cross(P b) { return x * b.y - y * b.x; }
  T cross(P b, P c) const { return (b - *this).cross(c - *this); }
  T dot(P b) { return x * b.x + y * b.y; }
  bool inTriangle(const P & a, const P & b, const P & c) const {
    #define tmp(a,b) (cross(a,b) > 0)
    return tmp(a,b) == tmp(b,c) && tmp(b,c) == tmp(c,a);
    #undef tmp
  }
  // double angle() const { return atan2(y, x); }
};
int cmpCircle(P a, P b, P c, P d) {
  P v1 = b - a, v2 = d - a;  P v3 = b - c, v4 = d - c;
  long double tmp = (long double) abs(v1.cross(v2)) * v3.dot(v4) +
      (long double) v1.dot(v2) * abs(v3.cross(v4));
  if(abs(tmp) < 1e-8) return 0; if(tmp == 0) return 0;
  if(tmp > 0) return 1; return -1;
}
struct pair_hash {   template <class T1, class T2>
    std::size_t operator () (const std::pair<T1,T2> &p) const {
    return p.first * 10000 + p.second;
    }
};
unordered_map<pair<int,int>, pair<int,int>, pair_hash> mt;
// pair<int,int> t[nax][nax];
set<pair<int,int>> edges;
vector<vector<int>> triangles;
void rec(int a, int c, const vector<P> & points);
void trim(int a, int b) {
  assert(a < b);  auto it = mt.find({a,b});
  if((it != mt.end()) && (it -> second == make_pair(-1, -1)))
    mt.erase(it);
}

void change(int a, int b, int from, int to) {
  if(a > b) swap(a, b);
  if(!mt.count({a,b})) mt[{a,b}] = {-1,-1};
  for(int * x : vector<int *>{&mt[{a,b}].first, &mt[{a,b}].second})
    if(*x == from) {
      *x = to; trim(a, b); return;
    }
  assert(false);
}
void rec(int a, int c, const vector<P> & points) {
  if(a > c) swap(a, c);  if(!mt.count({a,c})) return;
  int b = mt[{a,c}].first; int d = mt[{a,c}].second;
  if(b > d) swap(b, d);
  //if(t[b][d] != make_pair(-1, -1)) return;
  if(b == -1 || d == -1) return;
  for(int rep = 0; rep < 2; ++rep) {
    if(points[a].inTriangle(points[b], points[c], points[d]))
      return;
    swap(a, c);
  }
  debug() << imie(cmpCircle(points[a], points[b], points[c], points[d]));
  if(cmpCircle(points[a], points[b], points[c], points[d]) != 1) {
    debug() << "nie chcę flipnac";
    return;
  }
  debug() << "chcę flipnac";
  assert((!mt.count({b,d})) || (mt[{b,d}] == make_pair(-1, -1)));
  //assert(t[b][d] == make_pair(-1, -1));
  mt[{b,d}] = {a, c};
```


KMR

```
  trim(b, d);
  mt.erase(make_pair(a, c));
  change(a,b,c,d); change(b,c,a,d); change(a,d,c,b); change(c,d,a,b);
  rec(a,b,points); rec(b,c,points); rec(c,d,points); rec(d,a,points);
}
void addTriangle(int a, int b, int c) {
  change(a, b, -1, c); change(a, c, -1, b); change(b, c, -1, a);
}
void anyTriangulation(vector<P> points) {
  sort(points.begin(), points.end(), [](const P & a, const P & b) {
    return make_pair(a.x, a.y) < make_pair(b.x, b.y);
  });
  vector<P> upper, lower;
  for(P C : points) {
    #define backback(w) w[(int)w.size()-2]
    while((int) upper.size() >= 2 && backback(upper).cross(upper.back(), C) > 0
) {
      addTriangle(C.id, backback(upper).id, upper.back().id);
      upper.pop_back();
    }
    upper.push_back(C);
    //if(!lower.empty() && lower[0].x == C.x) continue;
    while((int) lower.size() >= 2 && lower[(int)lower.size()-2].cross(lower.bac
k(), C) < 0) {
      addTriangle(C.id, backback(lower).id, lower.back().id);
      lower.pop_back();
    }
    lower.push_back(C);
    #undef backback
  }
  if(lower.size() == upper.size() && lower.size() == points.size()) {
    cerr << "all points are collinear, assert\n";
    puts("all points are collinear, assert");
    assert(false);
  }
}
const int nax = 1e6 + 5;
int memo_x[nax], memo_y[nax];
long long ans[nax];
long long KK(long long a) { return a * a; }
long long dist(int i, int j) {
  return KK(memo_x[i] - memo_x[j]) + KK(memo_y[i] - memo_y[j]);
}
void consider(int i, int j) {
  assert(i != j); assert(dist(i, j));
  for(int rep = 0; rep < 2; ++rep) {
    if(ans[i] == 0 || ans[i] > dist(i, j))
      ans[i] = dist(i, j);
    swap(i, j);
  }
}
void te() {
  mt.clear(); edges.clear(); triangles.clear();
  int n; scanf("%d", &n);
  for(int i = 0; i <= n; ++i) ans[i] = 0;
  vector<P> points(n);
  for(int i = 0; i < n; ++i) {
scanf("%d%d", &memo_x[i], &memo_y[i]); points[i] = P{memo_x[i], memo_y[i], i};
  }
  mt.reserve(4123123);
  //~ REP(i, points.size()) REP(j, points.size()) t[i][j] = {-1,-1};
  anyTriangulation(points);
  vector<pair<int,int>> init;
  for(auto ppp : mt) init.push_back(ppp.first);
  for(pair<int,int> p : init)
    if(mt.count(p) && mt[p] != make_pair(-1, -1))
      rec(p.first, p.second, points);
  //~ REP(i, points.size()) for(int j = i + 1; j < (int)points.size();++j)rec(i
,j,points);
  for(auto ppp : mt) if(ppp.second != make_pair(-1, -1)) {
  //for(int i = 0; i < (int) points.size(); ++i)
    //for(int j = i + 1; j < (int) points.size(); ++j)
    //  if(mt.count({i,j}) && mt[{i,j}] != make_pair(-1, -1)) {
      //if(t[i][j] != make_pair(-1, -1)) {
        int i = ppp.first.first, j = ppp.first.second;
        assert(i != j); consider(i, j); edges.insert({i, j});
        if(mt[{i,j}].first > j) triangles.push_back(vector<int>{i, j, mt[{i,j}]
.first});
        if(mt[{i,j}].second > j) triangles.push_back(vector<int>{i, j, mt[{i,j}
].second});
      }
  /* for(pair<int,int> edge : edges) printf("%d %d\n", edge.first, edge.second);
  for(auto vec : triangles) { for(int x : vec) printf("%d ", x); puts(""); } */
  //debug() << imie(getHull(points));
  for(int i = 0; i < (int) points.size(); ++i) printf("%lld\n", ans[i]);
}
```

kamil/fast_IO.cpp

```
// uncomment all lines if you need negative numbers
ll fast_read() {
  #define CU c = getchar_unlocked()
  ll x = 0;
  char c;
  while(isspace(CU));;;
  //~ bool is_neg = c == '-';
  //~ if(is_neg) CU;
  while(isdigit(c)) {
    x = 10 * x + c - '0';
    CU;
  }
  //~ if(is_neg) x *= -1;
  return x;
  #undef CU
}
void fast_print(ll x, char after = '\n') {
  static char buf[53];
  int i = 50;
  buf[i+1] = after;
  //~ bool is_neg = x < 0;
  //~ x = abs(x);
  while(x || i == 50) {
    buf[i--] = '0' + x % 10;
    x /= 10;
  }
  //~ if(is_neg) buf[i--] = '-';
```

```
  fputs_unlocked(buf + i + 1, stdout);
}
```

kamil/fast_sort.cpp

```
// about 2-3 times faster than std::sort() for N >= 1e6
void fast_sort(vector<unsigned> & t) {
  int n = t.size(), k = 1 << 16; // t[i] < k * k
  auto tmp = t; // if array: static int/unsigned tmp[nax];
  REP(turn, 2) {
    #define val(x) (turn ? x / k : x % k)
    vector<int> cnt(k+1);
    REP(i, n) cnt[val(t[i]) + 1]++;
    REP(i, k) cnt[i+1] += cnt[i];
    REP(i, n) tmp[ cnt[val(t[i])]++ ] = t[i];
    REP(i, n) t[i] = tmp[i];
    #undef val
  }
}
```

kamil/geo3d.cpp

```
#include "Geo2D.h"
struct Point3 {
  LD x, y, z; // ... + trivial operators
  LD& operator[](int a) {
    if (a == 0) { return x; } if (a == 1) { return y; } return z;
  }
  bool IsZero() { return abs(x) < kEps && abs(y) < kEps && abs(z) < kEps; }
  LD DotProd(Point3 a) { return x * a.x + y * a.y + z * a.z; }
  LD Norm() { return sqrt(x * x + y * y + z * z); }
  LD SqNorm() { return x * x + y * y + z * z; }
  void NormalizeSelf() { *this /= Norm(); }
  Point3 Normalize() {
    Point3 res(*this); res.NormalizeSelf(); return res;
  }
  LD Dis(Point3 a) { return (*this - a).Norm(); }
  pair<LD, LD> SphericalAngles() { return {atan2(z, sqrt(x * x + y * y)), atan2
↳(y, x)}; }
  LD Area(Point3 p) { return Norm() * p.Norm() * sin(Angle(p)) / 2; }
  LD Angle(Point3 p) {
    LD a = Norm(); LD b = p.Norm(); LD c = Dis(p);
    return acos((a * a + b * b - c * c) / (2 * a * b));
  }
  static LD Angle(Point3 p, Point3 q) { return p.Angle(q); }
  Point3 CrossProd(Point3 p) {
    Point3 q(*this);
    return {q[1] * p[2] - q[2] * p[1],
            q[2] * p[0] - q[0] * p[2],
            q[0] * p[1] - q[1] * p[0]};
  }
  static bool LexCmp(Point3& a, const Point3& b) {
    if (abs(a.x - b.x) > kEps) { return a.x < b.x; }
    if (abs(a.y - b.y) > kEps) { return a.y < b.y; }
    return a.z < b.z;
  }
  friend ostream& operator<<(ostream& out, Point3 m);
};
struct Line3 {
  Point3 p[2]; Point3& operator[](int a) { return p[a]; }
  friend ostream& operator<<(ostream& out, Line3 m);
};
struct Plane {
  Point3 p[3];
  Point3& operator[](int a) { return p[a]; }
  Point3 GetNormal() {
    Point3 cross = (p[1] - p[0]).CrossProd(p[2] - p[0]); return cross.Normalize
↳();
  }
  void GetPlaneEq(LD& A, LD& B, LD& C, LD& D) {
    Point3 normal = GetNormal();
    A = normal[0]; B = normal[1]; C = normal[2];
    D = normal.DotProd(p[0]);
    assert(abs(D - normal.DotProd(p[1])) < kEps);
    assert(abs(D - normal.DotProd(p[2])) < kEps);
  }
  vector<Point3> GetOrtonormalBase() {
    Point3 normal = GetNormal();
    Point3 cand = {-normal.y, normal.x, 0};
    if (abs(cand.x) < kEps && abs(cand.y) < kEps) { cand = {0, -normal.z, norma
↳l.y}; }
    cand.NormalizeSelf();
    Point3 third = Plane{Point3{0, 0, 0}, normal, cand}.GetNormal();
    assert(abs(normal.DotProd(cand)) < kEps &&
           abs(normal.DotProd(third)) < kEps && abs(cand.DotProd(third)) < kEps
↳);
    return {normal, cand, third};
  }
};
struct Circle3 {
  Plane pl; Point3 o; LD r;
  friend ostream& operator<<(ostream& out, Circle3 m);
};
struct Sphere {
  Point3 cent; LD r;
};
struct Utils3 {
  static bool Lines3Equal(Line3 p, Line3 l) {
    return Utils3::PtBelongToLine3(p[0], l) && Utils3::PtBelongToLine3(p[1], l);
  }
  //angle PQR
  static LD Angle(Point3 P, Point3 Q, Point3 R) { return (P - Q).Angle(R - Q); }
  static Point3 ProjPtToLine3(Point3 p, Line3 l) { // ok
    Point3 diff = l[1] - l[0]; diff.NormalizeSelf();
    return l[0] + diff * (p - l[0]).DotProd(diff);
  }
  static LD DisPtLine3(Point3 p, Line3 l) { // ok
//  LD area = Area(p, l[0], l[1]); LD dis1 = 2 * area / l[0].Dis(l[1]);
    LD dis2 = p.Dis(ProjPtToLine3(p, l)); // assert(abs(dis1 - dis2) < kEps);
    return dis2;
  }
  static LD DisPtPlane(Point3 p, Plane pl) {
    Point3 normal = pl.GetNormal(); return abs(normal.DotProd(p - pl[0]));
  }
  static Point3 ProjPtToPlane(Point3 p, Plane pl) {
    Point3 normal = pl.GetNormal(); return p - normal * normal.DotProd(p - pl[0
↳]);
  }
```

```
  static bool PtBelongToPlane(Point3 p, Plane pl) { return DisPtPlane(p, pl) < kEps; }
  static Point PlanePtTo2D(Plane pl, Point3 p) { // ok
    assert(PtBelongToPlane(p, pl));
    vector<Point3> base = pl.GetOrtonormalBase();
    Point3 control{0, 0, 0};
    REP (tr, 3) { control += base[tr] * p.DotProd(base[tr]); }
    assert(PtBelongToPlane(pl[0] + base[1], pl));
    assert(PtBelongToPlane(pl[0] + base[2], pl));
    assert((p - control).IsZero());
    return {p.DotProd(base[1]), p.DotProd(base[2])};
  }
  static Line PlaneLineTo2D(Plane pl, Line3 l) {
    return {PlanePtTo2D(pl, l[0]), PlanePtTo2D(pl, l[1])};
  }
  static Point3 PlanePtTo3D(Plane pl, Point p) { // ok
    vector<Point3> base = pl.GetOrtonormalBase();
    return base[0] * base[0].DotProd(pl[0]) + base[1] * p.x + base[2] * p.y;
  }
  static Line3 PlaneLineTo3D(Plane pl, Line l) {
    return {PlanePtTo3D(pl, l[0]), PlanePtTo3D(pl, l[1])};
  }
  static Line3 ProjLineToPlane(Line3 l, Plane pl) { // ok
    return {ProjPtToPlane(l[0], pl), ProjPtToPlane(l[1], pl)};
  }
  static LD DisLineLine(Line3 l, Line3 k) { // ok
    Plane together {l[0], l[1], l[0] + k[1] - k[0]}; // parallel FIXME
    Line3 proj = ProjLineToPlane(k, together);
    Point3 inter = (Utils3::InterLineLine(l, proj))[0];
    Point3 on_k_inter = k[0] + inter - proj[0];
    return inter.Dis(on_k_inter);
  }
  static bool PtBelongToLine3(Point3 p, Line3 l) { return DisPtLine3(p, l) < kEps; }
  static bool Line3BelongToPlane(Line3 l, Plane pl) {
    return PtBelongToPlane(l[0], pl) && PtBelongToPlane(l[1], pl);
  }
  static LD Det(Point3 a, Point3 b, Point3 d) { // ok
    Point3 pts[3] = {a, b, d};
    LD res = 0;
    for (int sign : {-1, 1}) {
      REP (st_col, 3) {
        int c = st_col;
        LD prod = 1;
        REP (r, 3) {
          prod *= pts[r][c];
          c = (c + sign + 3) % 3;
        }
        res += sign * prod;
      }
    }
    return res;
  }
  static LD Area(Point3 p, Point3 q, Point3 r) { q -= p; r -= p; return q.Area(r); }
  static vector<Point3> InterLineLine(Line3 k, Line3 l) {
    if (Lines3Equal(k, l)) { return {k[0], k[1]}; }
    if (PtBelongToLine3(l[0], k)) { return {l[0]}; }
    Plane pl{l[0], k[0], k[1]};
    if (!PtBelongToPlane(l[1], pl)) { return {}; }
    Line k2 = PlaneLineTo2D(pl, k); Line l2 = PlaneLineTo2D(pl, l);
    vector<Point> inter = Utils::InterLineLine(k2, l2);
    vector<Point3> res;
    for (auto P : inter) { res.PB(PlanePtTo3D(pl, P)); }
    return res;
  }
  static Plane ParallelPlane(Plane pl, Point3 A) { // plane parallel to pl going through A
    Point3 diff = A - ProjPtToPlane(A, pl);
    return {pl[0] + diff, pl[1] + diff, pl[2] + diff};
  }
  // image of B in rotation wrt line passing through origin s.t. A1->A2
  // implemented in more general case with similarity instead of rotation
  static Point3 RotateAccordingly(Point3 A1, Point3 A2, Point3 B1) { // ok
    Plane pl{A1, A2, {0, 0, 0}};
    Point A12 = PlanePtTo2D(pl, A1); Point A22 = PlanePtTo2D(pl, A2);
    complex<LD> rat = complex<LD>(A22.x, A22.y) / complex<LD>(A12.x, A12.y);
    Plane plb = ParallelPlane(pl, B1); Point B2 = PlanePtTo2D(plb, B1);
    complex<LD> Brot = rat * complex<LD>(B2.x, B2.y);
    return PlanePtTo3D(plb, {Brot.real(), Brot.imag()});
  }

  static vector<Circle3> InterSpherePlane(Sphere s, Plane pl) { // ok
    Point3 proj = ProjPtToPlane(s.o, pl);
    LD dis = s.o.Dis(proj);
    if (dis > s.r + kEps) { return {}; }
    if (dis > s.r - kEps) { return {{pl, proj, 0}}; } // is it best choice?
    return {{pl, proj, sqrt(s.r * s.r - dis * dis)}};
  }
  static bool PtBelongToSphere(Sphere s, Point3 p) {
    return abs(s.r - s.o.Dis(p)) < kEps;
  }
};
struct PointS { // just for conversion purposes, probably toEucl suffices
  LD lat, lon;
  Point3 toEucl() {
    return Point3{cos(lat) * cos(lon), cos(lat) * sin(lon), sin(lat)};
  }
  PointS(Point3 p) {
    p.NormalizeSelf(); lat = asin(p.z); lon = acos(p.y / cos(lat));
  }
};
LD DistS(Point3 a, Point3 b) {
  return atan2l(b.CrossProd(a).Norm(), a.DotProd(b));
}
struct CircleS {
  Point3 o; // center of circle on sphere
  LD r; // arc len
  LD area() const { return 2 * kPi * (1 - cos(r)); }
};

CircleS From3(Point3 a, Point3 b, Point3 c) { // any three different points
  int tmp = 1;
  if ((a - b).Norm() > (c - b).Norm()) { swap(a, c); tmp = -tmp; }
```

```
  if ((b - c).Norm() > (a - c).Norm()) { swap(a, b); tmp = -tmp; }
  Point3 v = (c - b).CrossProd(b - a); v = v * (tmp / v.Norm());
  return CircleS{v, DistS(a,v)};
}

CircleS From2(Point3 a, Point3 b) { // neither the same nor the opposite
  Point3 mid = (a + b) / 2; mid = mid / mid.Norm(); return From3(a, mid, b);
}

LD SphAngle(Point3 A, Point3 B, Point3 C) { //angle at A, no two points opposite
  LD a = B.DotProd(C); LD b = C.DotProd(A); LD c = A.DotProd(A);
  return acos((b - a * c) / sqrt((1 - Sq(a)) * (1 - Sq(c))));
}

LD TriangleArea(Point3 A, Point3 B, Point3 C) { // no two poins opposite
  LD a = SphAngle(C, A, B); LD b = SphAngle(A, B, C); LD c = SphAngle(B, C, A);
  return a + b + c - kPi;
}
vector<Point3> IntersectionS(CircleS c1, CircleS c2) {
  Point3 n = c2.o.CrossProd(c1.o), w = c2.o * cos(c1.r) - c1.o * cos(c2.r);
  LD d = n.SqNorm();
  if (d < kEps) { return {}; } // parallel circles (can fully overlap)
  LD a = w.SqNorm() / d;
  vector<Point3> res;
  if (a >= 1 + kEps) { return res; }
  Point3 u = n.CrossProd(w) / d;
  if (a > 1 - kEps) { res.PB(u); return res; }
  LD h = sqrt((1 - a) / d);
  res.PB(u + n * h); res.PB(u - n * h);
  return res;
}
bool Eq(LD a, LD b) { return abs(a - b) < kEps; }
vector<Point3> intersect(Sphere a, Sphere b, Sphere c) { // Does not work for 3
 colinear centers
    vector<Point3> res;
    Point3 ex, ey, ez;
    LD r1 = a.r, r2 = b.r, r3 = c.r, d, cnd_x = 0, i, j;
    ex = (b.o - a.o).Normalize();
    i = ex.DotProd(c.o - a.o);
    ey = ((c.o - a.o) - ex * i).Normalize();
    ez = ex.CrossProd(ey);
    d = (b.o - a.o).Norm();
    j = ey.DotProd(c.o - a.o);

    bool cnd = 0;
    if (Eq(r2, d - r1)) { cnd_x = +r1; cnd = 1; }
    if (Eq(r2, d + r1)) { cnd_x = -r1; cnd = 1; }

    if (!cnd && (r2 < d - r1 || r2 > d + r1)) return res;
    if (cnd) {
        if (Eq(Sq(r3), (Sq(cnd_x - i) + Sq(j)))) res.PB(Point3(cnd_x, 0.0, 0.0)
);
    } else {
      LD x = (Sq(r1) - Sq(r2) + Sq(d)) / (2 * d);
      LD y = (Sq(r1) - Sq(r3) + Sq(i) + Sq(j)) / (2 * j) - (i / j) * x;
      LD u = Sq(r1) - Sq(x) - Sq(y);
      if (u >= -kEps) {
        LD z = sqrtl(max(LD(0), u)); res.PB(Pt(x, y, z));
        if (!isZero(z)) res.PB(Pt(x, y, -z));
      }
    }
    for (auto& it : res) { it = a.o + ex * it[0] + ey * it[1] + ez * it[2]; }
    return res;
}
```

kamil/hull_3d.cpp

```
#define REP(i, n) for(int i = 0; i < int(n); ++i)
//algorytm znajduje zasmiecona otoczke (z wszystkimi punktami na scianach)
//jesli chcemy miec tylko wierzcholki otoczki, to odkomentowujemy

//uwaga na ll (nie ma ich w iloczynie wektorowym) i ew. przepelnienie

//jak przerabiamy na double'e, to sciany grupujemy po znormalizowanym
//wektorze normalnym

//klasyfikacja punktow, patrzymy do ilu roznych (pod wzledem wektorow normalny
ch)
//scian nalezy dany punkt:
//0 - nie naleze do wypuklej otoczki
//1 - punkt wewnetrzny sciany
//2 - punkt wewnatrz krawedzi
//3 i wiecej - wierzcholek
const int N=1000;
int n;
struct sciana{
    int t[3]; //numery wierzcholkow sciany
    sciana(){}
    sciana(int a,int b,int c) {t[0]=a; t[1]=b; t[2]=c;}
};
struct P3{
    ll x,y,z;
    void read() { ... }
    bool operator < (const P3 & he) const { // only for map
    return vector<ll>{x, y, z} < vector<ll>{he.x, he.y, he.z};
  }
  ll dot(const P3 & he) const {
    return (ll) x * he.x + (ll) y * he.y + (ll) z * he.z;
  }
  P3 operator - (const P3 & he) const {
    return P3{x - he.x, y - he.y, z - he.z};
  }
  bool operator == (const P3 & he) const {
    return x == he.x && y == he.y && z == he.z;
  }
  P3 operator ^ (const P3 & he) const {
    return P3{y * he.z - z * he.y, z * he.x - x * he.z, x * he.y - y * he.x};
  }
  ll mno(const P3 & b, const P3 & c) const {
    return x * b.y * c.z + y * b.z * c.x + z * b.x * c.y
      - x * b.z * c.y - y * b.x * c.z - z * b.y * c.x;
  }
  double norm() const {
    return sqrt(double(K(x) + K(y) + K(z)));
  }
};
```

```
ostream& operator<<(ostream& out, P3 pp) { ... }
vector<sciana> v;
vector<P3> p;
vector<vector<int>> t; //numer sciany do ktorej nalezy krawedz
void step(int x) {
    //znajdujemy widoczne sciany
    vector<bool> vis(v.size());
    REP(i, v.size()) {
        P3 normal=((p[v[i].t[1]]-p[v[i].t[0]])^(p[v[i].t[2]]-p[v[i].t[0]]));
        ll il=(normal.dot(p[x]-p[v[i].t[0]]));
        vis[i] = false;
        if (il>0) vis[i] = true;
        else if (il==0){
            if ((normal.dot((p[v[i].t[1]]-p[v[i].t[0]])^(p[x]-p[v[i].t[0]]))>=0
                    && normal.dot((p[v[i].t[2]]-p[v[i].t[1]])^(p[x]-p[v[i].t[1]]))>=0
                    && normal.dot((p[v[i].t[0]]-p[v[i].t[2]])^(p[x]-p[v[i].t[2]]))>=0))
                //~ return; /*ODKOMENTUJ*/
            vis[i] = true;
        }
    }
    int ile=v.size();
    vector<pair<int,int>> pom;
    REP(i,ile) if (vis[i])
        REP(j,3) if (!vis[t[v[i].t[(j+1)%3]][v[i].t[j]]])
        pom.push_back(make_pair(v[i].t[j],v[i].t[(j+1)%3]));
    REP(i,ile) if (vis[i]){
        swap(v[i],v.back());
        REP(j,3) t[v[i].t[j]][v[i].t[(j+1)%3]]=i;
        vis[i--]=vis[--ile];
        v.pop_back();
    }
    REP(i, v.size()) assert(!vis[i]);
    for(pair<int,int> pa : pom) {
        t[pa.first][pa.second]=t[pa.second][x]=t[x][pa.first]=v.size();
        v.push_back(sciana(pa.first, pa.second, x));
    }
}
void CH3D(){ //n>=3, oblicza wektor trojkatnych scian v (mozliwe powtorzenia)
    int i=2;
    P3 normal;
    while (i<n && (normal=((p[1]-p[0])^(p[i]-p[0])))==P3{0,0,0}) i++;
    if (i>=n) return; //wspolliniowe
    int x=i++;
    v.push_back(sciana(0,1,x));
    v.push_back(sciana(1,0,x));
    t[0][1]=t[1][x]=t[x][0]=0;
    t[1][0]=t[0][x]=t[x][1]=1;
    while (i<n && normal.dot(p[i]-p[0])==0) i++;
    if (i==n){ //wspolplaszczyznowe, tworzona jest sztuczna sciana dla kazdej krawedzi!!!
    cerr << "deeeebug  wspolplaszczyznowe\n";
        v.clear();
        p[n++] = P3{3123,-3123,954}; //pkt spoza plaszczyzny, uwaga na zakresy!
        CH3D();
        //~ vector<sciana> v2;
        n--;
        return;
    }
    step(i);
    for(int j = 2; j <= n - 1; ++j) if (j!=i && j!=x) step(j);
}
/****************** wszystko ponizej jest opcjonalne ************************/
/**
double area() {
    double res = 0.0;
    debug() << imie(v.size());
    REP(i, v.size()) {
        P3 normal=((p[v[i].t[1]]-p[v[i].t[0]])^(p[v[i].t[2]]-p[v[i].t[0]]));
        res += normal.norm();
    }
    return 0.5*res;
}
double volume() {
    double res = 0.0;
    debug() << imie(v.size());
    REP(i, v.size()) {
    res += p[v[i].t[0]].mno(p[v[i].t[1]], p[v[i].t[2]]);
    debug() << imie((double) p[v[i].t[0]].mno(p[v[i].t[1]], p[v[i].t[2]]));
        //~ P3 normal=((p[v[i].t[1]]-p[v[i].t[0]])^(p[v[i].t[2]]-p[v[i].t[0]]));
        //~ double foo=(double)normal.x*normal.x+(double)normal.y*normal.y+(double)normal.z*normal.z;
        //~ foo=sqrt(foo);
        //~ res+=foo;
    }
    return abs(res) / 6;
}

map<P3, vector<int> > mapa; //sciany
vector<int> klas;
void compute_walls(){ //laczy sciany trojkatne w wielokaty i wyznacza klasyfikacje, najpierw odpal CH3D()
    REP(i, v.size()){
        P3 normal=((p[v[i].t[1]]-p[v[i].t[0]])^(p[v[i].t[2]]-p[v[i].t[0]]));
        int foo=1;//__gcd(abs(normal.x),__gcd(abs(normal.y),abs(normal.z)));
        normal.x/=foo;
        normal.y/=foo;
        normal.z/=foo;
        REP(j,3) if(v[i].t[j] < n) mapa[normal].push_back(v[i].t[j]);
    }

  for(auto & pa : mapa) {
    sort(pa.second.begin(), pa.second.end());
        pa.second.erase(unique(pa.second.begin(), pa.second.end()), pa.second.end());
        for(int j : pa.second) assert(j < n), klas[j]++;
    }
}
/* sortuje wierzcholki na scianach w kolejnosci, tylko dla wersji bez smieci */
int pocz;
#define norm norm_compile
P3 norm;
bool CHcomp(int x,int y){
```

```
    if(x == pocz) return true;
    if(y == pocz) return false;
    return ((p[x]-p[pocz]) ^ (p[y]-p[pocz])).dot(norm) < 0;
}
void sort_walls(){ //najpierw odpal compute_walls()
    for(auto & pa : mapa) {
        vector<int>& w = pa.second;
        if((int) w.size() == 2) continue;
        pocz=w[0];
        norm=(p[w[1]]-p[w[0]])^(p[w[2]]-p[w[0]]);
        sort(w.begin(), w.end(), CHcomp);
    }
}
void show(){
    //~ cout << "Sciany trojkatne:" << endl; for(sciana i : v)
    //~   cout << p[i.t[0]] << " " << ... p[i.t[2]] << endl;
    //~ REP(i,n) cout << p[i] << " " << klas[i] << endl; // klasyfikacja
   //~ for(auto pa : mapa) { //~ cout << pa.first << ": "; // sciany
        //~ for(int x : pa.second) cout << p[x] << " "; cout << endl; }
    printf("%6lf %6lf\n", area() / 1e12, volume() / 1e18);
}
void test_case() {
  cin >> n;
  klas = vector<int>(n);
  p.clear();
  p.resize(n);
  t = vector<vector<int>>(n, vector<int>(n));
    REP(i,n) p[i].read();
    //~ random_shuffle(p,p+n);
    CH3D();
    compute_walls();
    sort_walls();
    show();
    pocz = 0;
    mapa.clear();
    v.clear();
}
```

kamil/hull_online.cpp

```
// adding points online, O(N * log(N))
typedef long long ll; struct P {ll x, y;// trivia: read, write, *, -}
struct cmp_x1 {
  bool operator()(const P & a, const P & b) {
    return make_pair(a.x, b.x) < make_pair(a.y, b.y);
  }
};
struct cmp_x2 { /* to samo co wyżej ale > */ };
typedef function<bool(const P &, const P &)> foo;
typedef set<P, foo> my_set;
struct Hull {
  my_set up, down;
  Hull() {
    up = my_set(cmp_x1());
    down = my_set(cmp_x2());
  }
  void add(const P & last) {
    for(int rep = 0; rep < 2; ++rep) {
      my_set & s = (rep == 0) ? up : down;
      auto belongs = [&] (my_set :: iterator it) {
        assert(it != s.end());
        if(it == s.begin() || next(it, 1) == s.end()) return true;
        P A = *next(it, -1);
        P B = *it;
        P C = *next(it, 1);
        if((B - A) * (C - A) < 0) return true;
        s.erase(it);
        return false;
      };
      auto it = s.insert(last).first;
      if(belongs(it)) {
        while(it != s.begin() && !belongs(next(it, -1)));
        while(next(it, 1) != s.end() && !belongs(next(it, 1)));
      }
    }
  }
  vector<P> get() const {
    vector<P> w;
    for(auto p : up) w.push_back(p);
    if((int) w.size() <= 1) return w;
    w.pop_back();
    for(auto p : down) w.push_back(p);
    w.pop_back();
    return w;
  }
};
```

kamil/lines.cpp

```
const ll is_query = -(1LL << 62);
struct Line {
    ll m, b;
    mutable function<const Line *()> succ;
    bool operator<(const Line &rhs) const {
        if (rhs.b != is_query) return m < rhs.m;
        const Line *s = succ();
        if (!s) return 0;
        ll x = rhs.m;
        return b - s->b < (s->m - m) * x;
    }
};
struct HullDynamic : public multiset<Line> {
    bool bad(iterator y) {
        auto z = next(y);
        if (y == begin()) {
            if (z == end()) return 0;
            return y->m == z->m && y->b <= z->b;
        }
        auto x = prev(y);
        if (z == end()) return y->m == x->m && y->b <= x->b;
        return (x->b - y->b) * (z->m - y->m) >= (y->b - z->b) * (y->m - x->m);
    }
    void insert_line(ll m, ll b) {
        auto y = insert({m, b});
        y->succ = [=] { return next(y) == end() ? 0 : &*next(y); };
        if (bad(y)) {
            erase(y);
            return;
```

```
        }
        while (next(y) != end() && bad(next(y))) erase(next(y));
        while (y != begin() && bad(prev(y))) erase(prev(y));
    }
    ll eval(ll x) {
        auto l = *lower_bound((Line) {x, is_query});
        return l.m * x + l.b;
    }
};
```

kamil/massey.cpp

```
const int mod = ...;
void add_self(int & a, int b); void sub_self(int & a, int b);
int mul(int a, int b); int my_pow(int a, int b); int my_inv(int a);

struct Massey {
  vector<int> start, coef; // 3 optional lines
  vector<vector<int>> powers;
  int memo_inv;
  // Start here and write the next ~25 lines until "STOP"

  int L; // L == coef.size() <= start.size()
  Massey(vector<int> in) { // O(N^2)
    L = 0;
    const int N = in.size();
    vector<int> C{1}, B{1};
    for(int n = 0; n < N; ++n) {
      assert(0 <= in[n] && in[n] < mod); // invalid input
      B.insert(B.begin(), 0);
      int d = 0;
      for(int i = 0; i <= L; ++i)
        add_self(d, mul(C[i], in[n-i]));
      if(d == 0) continue;
      vector<int> T = C;
      C.resize(max(B.size(), C.size()));
      for(int i = 0; i < (int) B.size(); ++i)
        sub_self(C[i], mul(d, B[i]));
      if(2 * L <= n) {
        L = n + 1 - L;
        B = T;
        d = my_inv(d);
        for(int & x : B) x = mul(x, d);
      }
    }
    cerr << "L = " << L << "\n";
    assert(2 * L <= N - 2); // NO RELATION FOUND :(
    // === STOP ===
    for(int i = 1; i < (int) C.size(); ++i)
      coef.push_back((mod - C[i]) % mod);
    assert((int) coef.size() == L);
    for(int i = 0; i < L; ++i)
      start.push_back(in[i]);
    while(!coef.empty() && !coef.back()) { coef.pop_back(); --L; }
    if(!coef.empty()) memo_inv = my_inv(coef.back());
    powers.push_back(coef);
    //~ debug() << imie(coef);
  }
  vector<int> mul_cut(vector<int> a, vector<int> b) {
    vector<int> r(2 * L - 1);
    for(int i = 0; i < L; ++i)
      for(int j = 0; j < L; ++j)
        add_self(r[i+j], mul(a[i], b[j]));
    while((int) r.size() > L) {
      int value = mul(r.back(), memo_inv); // div(r.back(), coef.back());
      const int X = r.size();
      add_self(r[X-L-1], value);
      for(int i = 0; i < L; ++i)
        sub_self(r[X-L+i], mul(value, coef[i]));
      assert(r.back() == 0);
      r.pop_back();
    }
    return r;
  }
  int get(ll k) { // O(L^2 * log(k))
    if(k < (int) start.size()) return start[k];
    if(L == 0) return 0;
    k -= start.size();
    vector<int> vec = coef;
    for(int i = 0; (1LL << i) <= k; ++i) {
      if(i == (int) powers.size())
        powers.push_back(mul_cut(powers.back(), powers.back()));
      if(k & (1LL << i))
        vec = mul_cut(vec, powers[i]);
    }
    int total = 0;
    for(int i = 0; i < L; ++i)
      add_self(total, mul(vec[i], start[(int)start.size()-1-i]));
    return total;
  }
};
int main() {
  // f[n] = 3 * f[n-1] + f[n-3] ---> coef: [3, 0, 1]
  vector<int> in{10, 0, 1, 0, 0, 1, 3, 9, 28, 87};
  Massey massey(in);
  for(int i = 0; i < 30; ++i) printf("%d ", massey.get(i));
  puts(""); // 10 0 ... 951398949 883208606 modulo 1e9+7
}
```

kamil/sztuczki.cpp

```
#include<ext/pb_ds/assoc_container.hpp> // ordered set
#include<ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds; template <typename T> using ordered_set =
    tree<T, null_type, less<T>, rb_tree_tag, tree_order_statistics_node_update>;
ordered_set<int> s; s.insert(1); s.insert(2);
 s.order_of_key(1);   // Out: 0.
*s.find_by_order(1);  // Out: 2.
// unordered_map hash.
struct MyHash { std::size_t operator()(const MojTyp& x) { /* ... */ } };
bool operator==(const MojTyp& a, const MojType& b) { /* ... */ }
std::unordered_set<MojTyp, MyHash> secik;
// Find first / find next on bitset.
for (int pos = bs._Find_first(); pos != bs.size(); pos = bs._Find_next(pos))
  { /* Something with @pos. */ }
// Ratio liczba_elementów / liczba bucketów, domyślnie 1.0 (ustaw na INF, jeśli
secik.max_load_factor(0.25);  // chcesz używać tylko reserve).
```

```
// Maksuje liczbę bucketów do danej liczby, zalecana potęga 2,
secik.reserve(1<<15);  // np. (1 << 22) dla n = 1e6.
int secik.bucket_count(); // Zwraca obecna liczbe bucketow, sluzy do testowania.
// limit_denominator
import fractions
a = fractions.Fraction(3.141592).limit_denominator(7)  // a = Fraction(22, 7)
print(a.numerator, a.denominator)
```

marek/2-Sat.cpp

```
struct Sat {
  int n, ile;
  vector<vector<int>> imp;
  vector<bool> vis;
  vector<int> val, sort;

  void DfsMark(int x) {
    vis[x] = false;
    val[x] = (val[x ^ 1] == -1);
    for (int i : imp[x]) if (vis[i]) DfsMark(i);
  }
  void Dfs(int x) {
    vis[x] = true;
    for (int i : imp[x ^ 1]) if (!vis[i ^ 1]) Dfs(i ^ 1);
    sort[--ile] = x;
  }
  Sat(int m) : n(m * 2), ile(n), imp(n), vis(n), val(n, -1), sort(n) {}
  void Or(int a, int b) {
    imp[a ^ 1].push_back(b);
    imp[b ^ 1].push_back(a);
  }
  bool Run() {
    for (int i = 0; i < n; i++) if (!vis[i]) Dfs(i);
    for (int i : sort) if (vis[i]) DfsMark(i);
    for (int i = 0; i < n; i++)
      if (val[i]) for (int x : imp[i]) if (!val[x]) return false;
    return true;
  }
};
int main() {
  Sat sat(3);
  sat.Or(2 * 0 + 1, 2 * 1 + 0);  // !x_0 or  x_1
  sat.Or(2 * 1 + 0, 2 * 2 + 1);  //  x_1 or !x_2
  debug() << imie(sat.Run());
  debug() << imie(sat.val); // [0, 1, 1, 0, 0, 1] = [x0, !x0, x1, !x1, x2, !x2]
}
```

marek/Avl.cpp

```
constexpr bool persistent = true;
struct Node;
struct N {
  int v;
  N(int v_ = 0) : v(v_) {}
  N(Node* n);
  Node* operator->() const;
  operator int() const { return v; }
};
struct Node {
  N l, r;
  int h = 0;
  bool NeedsTouch() { return false; }
  void Touch() {}
  void Update() { assert(!NeedsTouch()); }
  Node(const Node& node) = default;
  Node* ptr() { return this; }
};
constexpr int kMaxNodes = (450 * 1024 * 1024 /* 450MiB */) / sizeof(Node);
int nast_node = 1 /* 0 is reserved for the null Node */;
Node node_ptr[kMaxNodes];
template <typename ...Args> N New(Args&& ...args) {
  assert(nast_node < kMaxNodes);
  node_ptr[nast_node] = Node(forward<Args>(args)...);
  return nast_node++;
}
N::N(Node* n) : v(n - node_ptr) {}                    // Converts index to ptr.
Node* N::operator->() const { return node_ptr + v; } // Converts ptr to index.
N Touch(N n) {
  if (!n or !n->NeedsTouch()) return n;
  if (persistent) {
    if (n->l) n->l = New(*n->l->ptr());
    if (n->r) n->r = New(*n->r->ptr());
  }
  n->Touch();
  return n;
}
N Make(N l, N v, N r) {
  if (persistent) v = New(*v->ptr());
  v->l = l;
  v->r = r;
  v->h = max(l->h, r->h) + 1;
  assert(abs(l->h - r->h) <= 2);
  v->Update();
  return v;
}
N Bal(N l, N v, N r) {
  assert(abs(l->h - r->h) <= 3);
  Touch(l); Touch(r);
  if (l->h > r->h + 2) {
    N ll = l->l, lr = l->r;
    if (ll->h >= lr->h) return Make(ll, l, Make(lr, v, r));
    return Touch(lr), Make(Make(ll, l, lr->l), lr, Make(lr->r, v, r));
  } else if (r->h > l->h + 2) {
    N rr = r->r, rl = r->l;
    if (rr->h >= rl->h) return Make(Make(l, v, rl), r, rr);
    return Touch(rl), Make(Make(l, v, rl->l), rl, Make(rl->r, r, rr));
  } else {
    return Make(l, v, r);
  }
}
N AddLeft(N n, N v) {
  if (!Touch(n)) return Make(0, v, 0);
  return Bal(AddLeft(n->l, v), n, n->r);
}
N AddRight(N n, N v) {
  if (!Touch(n)) return Make(0, v, 0);
  return Bal(n->l, n, AddRight(n->r, v));
}
```

```
pair<N, N> RemLeft(N n) {
  if (!Touch(n)->l) return {n->r, n};
  auto s = RemLeft(n->l);
  return {Bal(s.first, n, n->r), s.second};
}
// Joins l, r - trees, v - single vertex that will be overridden even
// in the case of a persistent tree.
N Join(N l, N v, N r) {
  if (!Touch(l)) return AddLeft(r, v);
  if (!Touch(r)) return AddRight(l, v);
  if (l->h > r->h + 2) return Bal(l->l, l, Join(l->r, v, r));
  if (r->h > l->h + 2) return Bal(Join(l, v, r->l), r, r->r);
  return Make(l, v, r);
}
N Merge(N l, N r) {
  if (!l or !r) return r + l;
  auto s = RemLeft(r);
  return Join(l, s.second, s.first);
}
pair<N, N> Split(N n, /* Maybe some additional arguments. */) {
  if (!Touch(n)) return {0, 0};
  if (/* Condition for checking if n belongs to the left tree. */) {
    auto s = Split(n->r, /* Some args. */);
    return {Join(n->l, n, s.first), s.second};
  } else /* n belongs to the right tree. */ {
    auto s = Split(n->l, /* Some args. */);
    return {s.first, Join(s.second, n, n->r)};
  }
}
```

marek/FibonacciCycle.cpp

```
// Zwraca rozmiar cyklu ciągu Fibonacciego modulo @mod. Znaleziony cykl
// niekoniecznie jest najmniejszym cyklem, wiadomo jednak, że cykl(m) <= 6m.
// Dla m, które nie są postaci 2 * 5^r, zachodzi ograniczenie: cykl(m) <= 4m.
// Jeśli nie zrobi się Nww, to wiadomo, że cykl(m) <= m * 2**(1 + #dz.pierw.m).
int FibonacciCycle(int mod) {
  auto PrimeCycle = [](int p) -> int {
    switch (p % 10) {
      case 2: return 3;   // Tylko 2 spełnia ten case.
      case 5: return 20;  // Tylko 5 spełnia ten case.
      case 1: case 9: return p - 1;
      case 3: case 7: return 2 * (p + 1);
      default: __builtin_unreachable();
    }
  };
  int cycle = 1;
  for (const pair<int, int>& pk : Factor(p)) {
    const int p = pk.first;
    const int k = pk.second;
    cycle = Nww(cycle, PrimeCycle(p) * Power(p, k - 1));
  }
  return cycle;
}
```

marek/Link_Cut.cpp

```
struct Splay {
  Splay *l = nullptr, *r = nullptr, *p = nullptr;
  bool flip = false;
  int roz = 1;    // SUBTREE  Rozmiar poddrzewa.
  int axroz = 1;  // SUBTREE  Pomocniczny rozmiar poddrzewa.
  void update() {
    assert(!flip and (!l or !l->flip) and (!r or !r->flip));
    axroz = roz;                // SUBTREE
    if (l) axroz += l->axroz;   // SUBTREE
    if (r) axroz += r->axroz;   // SUBTREE
  }
  void touch() {
    if (flip) {
      swap(l, r);
      if (l) l->flip = !l->flip;
      if (r) r->flip = !r->flip;
      flip = false;
    }
  }
  bool sroot() { return !p or (p->l != this and p->r != this); }
  void connect(Splay* c, bool left) { (left ? l : r) = c; if (c) c->p = this; }
  void rotate() {
    Splay* f = p;
    Splay* t = f->p;
    const bool isr = f->sroot();
    const bool left = (this == f->l);
    f->connect(left ? r : l, left);
    connect(f, !left);
    if (isr) p = t;
    else t->connect(this, f == t->l);
    f->update();
  }
  void push() {
    sroot() ? touch() : p->push();
    if (l) l->touch(); if (r) r->touch();
  }
  void splay() {
    push();
    while (!sroot()) {
      Splay* x = p->p;
      if (!p->sroot()) (((p->l == this) == (x->l == p)) ? p : this)->rotate();
      rotate();
    }
    update();
  }
  // Przenosi wierzchołek do korzenia. Prawe dziecko v będzie równe nullptr.
  // Aby zrobić coś na ścieżce od korzenia do v:
  //   >> v->expose();
  //   >> /* v reprezentuje ścieżkę, można z nim robić co się chce.
  //   >>  * Jeśli się odwiedzi wierzchołki poniżej v, to na najniższym
  //   >>  * trzeba wywołać ->splay(); */
  // Aby znaleźć LCA u i v (muszą być w tym samym poddrzewie):
  //   >> u->expose(); lca = v->expose();
  Splay* expose() {
    Splay *q = this, *x = nullptr;
    while (q) {
      q->splay();
      if (q->r) q->roz += q->r->axroz;  // SUBTREE
      if (x) q->roz -= x->axroz;        // SUBTREE
      q->r = x;
      q->update();
```

```
      x = q;
      q = q->p;
    }
    splay();
    return x;
  }
  // Zwraca roota drzewowego (nie splejowego!).
  Splay* root() {
    expose();
    Splay* s = this;
    while (s->touch(), s->l) s = s->l;
    s->splay();
    return s;
  }
  // Zakłada, że (*this) nie jest korzeniem drzewa.
  // Usuwa krawędź znajdującą się nad danym wierzchołkiem.
  void cut() {
    expose(); assert(l /* Nie jest rootem. */);
    Splay* s = l;
    while (s->touch(), s->r) s = s->r;
    s->splay(); s->r->p = nullptr; s->r = nullptr;
  }
  void link(Splay* to) {
    expose(); assert(!l /* Jest rootem. */);
    p = to;
    p->expose();        // SUBTREE
    p->roz += axroz;    // SUBTREE
    p->axroz += axroz;  // SUBTREE
  }
  // Sprawia, że wierzchołek jest rootem w logicznym i w splayowym drzewie.
  void make_root() { expose(); flip = !flip; touch(); }
};
```

marek/Lyndon.cpp

```
// 1) Przyjmuje słowo s (wypełnione na pozycjach 0, 1, ..., n-1).
//    Dzieli słowo s na pewną liczbę słów Lyndona p_1, ... p_k tak, że:
//    p_1 >= p_2 >= ... >= p_k (leksykograficznie)
//    Podział jest zapisywany w tablicy b - na i-tej pozycji jest true,
//    jeśli nastąpiło cięcie przed i-tą literką.
// 2) Znajduje minimalne leksykograficznie przesunięcie cykliczne słowa.
// 3) Znajduje minimalny leksykograficznie sufiks słowa.
void lyndon(char * s,   // Słowo zaczynające się na pozycji 0:
                        //   2) s powinno być sklejone: xx.
            int n,      // Długość słowa s (licząc ew. podwojenie).
            int& suf,   // 3) pozycja minimalnego leksykograficznie sufiksu.
            int& cyk,   // 2) pozycja minimalnego leksykograficznie przes. cykl.
            bool* b) {  // Tablica cięcia b.
  for (int i = 0; i < n; i++) b[i] = false;  // wykomentuj, jeśli nie 1)
  int p = 0, k = 0, m = 1;
  while (p < n) {
    if (m == n or s[m] < s[k]) {
      if (p < n / 2) cyk = p;  // wykomentuj, jeśli nie 2)
      while (p <= k) {
        p += m - k;
        if (p < n) {
          suf = p;  // wykomentuj, jeśli nie 3)
          b[p] = true;  // wykomentuj, jeśli nie 1)
        }
      }
      m = (k = p) + 1;
    } else if (s[m++] != s[k++]) k = p;
  }
}
```

marek/Manacher.cpp

```
// @s[0..n-1]  - napis długości @n.
// @r[0..2n-2] - tablica promieni palindromów.
// s: a   b   a   a   b   a   a   c   a   a   b   b   b   b   a   a   c   a   c
// r: 0 0 1 0 0 3 0 0 2 0 0 1 0 0 3 0 0 1 0 0 0 1 1 6 1 1 0 0 0 1 0 0 1 0 1 0 0
void Manacher(const char* s, int n, int* r) {
  for (int i = 0, m = 0, k = 0, p = 0; i < 2 * n - 1; m = i++ - 1) {
    while (p < k and i / 2 + r[m] != k)
      r[i++] = min(r[m--], (k + 1 - p++) / 2);
    while (k + 1 < n and p > 0 and s[k + 1] == s[p - 1])
      k++, p--;
    r[i] = (k + 1 - p++) / 2;
  }
}
```

marek/Mobius.tex

Niech $M(n) = \sum_{i=1}^{n} \mu(i)$. Można policzyć $M(n)$ w $O\left(n^{2/3} \cdot log(\text{smth})\right)$. Dla $u = n^{1/3}$, wystarczy spreprocesować M do $n^{2/3}$ i obliczyć $M(n)$ wzorem:

$$M(n) = M(u) - \sum_{m=1}^{u} \mu(m) \sum_{i=\left\lfloor \frac{u}{m} \right\rfloor + 1}^{\left\lfloor \frac{n}{m} \right\rfloor} M\left(\left\lfloor \frac{n}{mi} \right\rfloor\right).$$

marek/Modulo_2_to_61_minus_1.cpp

```
// Zwraca (a * b) % (2**p - 1).
ull Mnoz(ull a, ull b) {
  constexpr int p = 61;
  constexpr ull mod = (1llu << p) - 1;
  const auto A = (__uint128_t) a * b;
  ull result = (ull) (A & mod) + (ull) (A >> p);
  if (mod <= result) {
    result -= mod;
  }
  return result;
}
```

marek/PierwiastekModulo.cpp

```
using T = int;          // Typ, w którym mieści się modulo.
using T2 = long long;   // Typ, w którym mieści się kwadrat modulo.

// Dla pierwszego p > 2, potrafi szukać pierwiastków modulo p.
class Pierwiastek {  // Jeśli kod ma być używany dla stałego p, to warto p
 public:             // przekazać jako argument templata, a nie konstruktora.
  Pierwiastek(T p) : s(0), p(p), m(p - 1) {
    assert(p > 2);
    while (m % 2 == 0) { m /= 2; s++; }
    c = 2;
    while (Pot(c, p / 2) == 1) {
      c = rand() % (p - 1) + 1;  // Uwaga, rand() musi zwracać wartości rzędu p.
    }
  }
```

```
  T Pot(T a, T pot) const;  // Zwraca a**pot % p.

  T Licz(T a) const {  // Znajduje pierwiastek z a modulo p.
    if (a == 0) return 0;               // Sprawdza, czy a jest resztą
    if (Pot(a, p / 2) != 1) return -1;  // kwadratową. Jeśli nie, zwraca -1.
    T z = Pot(c, m);
    T v = Pot(a, m / 2);
    T u = (T2) v * v % p;
    v = (T2) v * a % p;
    u = (T2) u * a % p;
    for (int i = s - 1; i >= 1; i--) {
      if (Pot(u, T(1) << (i - 1)) != 1) {
        u = (T2) u * z % p;
        u = (T2) u * z % p;
        v = (T2) v * z % p;
      }
      z = (T2) z * z % p;
    }
    return v;  // Pierwiastkami liczby a są: {v, (p - v) % p}.
  }

 private:
  int s;
  T p, m, c;
};
```

marek/Pref.cpp

```
void Pref(const char* s, int n, int* p) {
  p[0] = n;
  int i = 1, m = 0;
  while (i < n) {
    while (m + i < n and s[m + i] == s[m]) m++;
    p[i++] = m;
    m = max(m - 1, 0);
    for (int k = 1; p[k] < m; m--) p[i++] = p[k++];
  }
}
```

marek/Simplex.cpp

```
struct Simplex {         // Maximize c*x subject to Ax <= b.
  using T = double;      // Initialize the structure, set A, b, c and then run
  vector<vector<T>> A;   // solve(). Max objective is stored in res. To recover
  vector<T> b, c;        // the best result, use getVars().
  int V, E;
  vector<int> eqIds, varIds, cols;
  T res;
  static constexpr T kEps = 1e-9;

  Simplex(int vars, int eqs) : A(eqs, vector<T>(vars)), b(eqs), c(vars),
      V(vars), E(eqs), eqIds(eqs), varIds(vars), res(0) {
    iota(varIds.begin(), varIds.end(), 0);
    iota(eqIds.begin(), eqIds.end(), vars);
  }

  void pivot(int eq, int var) {
    T coef = 1 / A[eq][var];
    cols.clear();
    for (int i = 0; i < V; i++) {
      if (abs(A[eq][i]) > kEps) { cols.push_back(i); A[eq][i] *= coef; }
    }
    A[eq][var] *= coef; b[eq] *= coef;
    for (int row = 0; row < E; row++) {
      if (row == eq || abs(A[row][var]) < kEps) { continue; }
      T k = -A[row][var];
      A[row][var] = 0;
      for (int i : cols) { A[row][i] += k * A[eq][i]; }
      b[row] += k * b[eq];
    }
    T q = c[var]; c[var] = 0;
    for (int i : cols) { c[i] -= q * A[eq][i]; }
    res += q * b[eq];
    swap(varIds[var], eqIds[eq]);
  }

  bool solve() {
    while (true) {
      int eq = -1, var = -1;
      for (int i = 0; i < E; i++) { if (b[i] < -kEps) { eq = i; break; } }
      if (eq == -1) { break; }
      for (int i = 0; i < V; i++) { if (A[eq][i] < -kEps) { var = i; break; } }
      if (var == -1) { res = -1e9; return false; /* No solution */ }
      pivot(eq, var);
    }
    while (true) {
      int var = -1, eq = -1;
      for (int i = 0; i < V; i++) { if (c[i] > kEps) { var = i; break; } }
      if (var == -1) { break; }
      for (int i = 0; i < E; i++) {
        if (A[i][var] < kEps) { continue; }
        if (eq >= 0 && b[i] / A[i][var] >= b[eq] / A[eq][var]) { continue; }
        eq = i;
      }
      if (eq == -1) { res = 1e9; return false; /* Unbounded */ }
      pivot(eq, var);
    }
    return true;
  }

  vector<T> getVars() { // Optimal assignment of variables.
    vector<T> result(V);
    for (int i = 0; i < E; i++) if (eqIds[i] < V) result[eqIds[i]] = b[i];
    return result;
  }
};
```

marek/SuffixArray.cpp

```
// Buduje tablicę sufiksową w czasie O(n + alpha).
// s[0..n-1],   s[i] in [0..alpha-1],    sa[0..n-1],   lcp[0..n-2].
void Sufar(const int* s, int n, int alpha, int* sa, int* lcp = nullptr) {
  if (n > 0) sa[0] = 0;
  if (n <= 1) return;
  vector<int> roz(alpha + 1), wsk(alpha), typ(n + 1), ids(n, -1), news, pos;
  auto star = [&](int i) -> bool { return typ[i] == 3; };
  auto Indukuj = [&]() -> void {
    copy(roz.begin(), roz.end() - 1, wsk.begin());
    sa[wsk[s[n - 1]]++] = n - 1;
```

```
    for (int i = 0; i < n; i++)
      if (sa[i] > 0 and !typ[sa[i] - 1])
        sa[wsk[s[sa[i] - 1]]++] = sa[i] - 1;
    copy(roz.begin() + 1, roz.end(), wsk.begin());
    for (int i = n - 1; i >= 0; i--)
      if (sa[i] > 0 and typ[sa[i] - 1])
        sa[--wsk[s[sa[i] - 1]]] = sa[i] - 1;
  };
  typ[n] = 3;
  for (int i = n - 1; i >= 0; i--) {
    sa[i] = -1;
    roz[s[i] + 1]++;
    if (i != n - 1 and s[i] < s[i + 1] + !!typ[i + 1]) {
      typ[i] = 1;
    } else if (typ[i + 1]) {
      typ[i + 1] = 3;
    }
  }
  partial_sum(roz.begin(), roz.end(), roz.begin());
  copy(roz.begin() + 1, roz.end(), wsk.begin());
  for (int i = 0; i < n; i++) if (star(i)) sa[--wsk[s[i]]] = i;
  Indukuj();
  int nast_id = 0, b = -1;
  for (int i = 0; i < n; i++) {
    int a = sa[i];
    if (!star(a)) continue;
    if (b >= 0) while (a == sa[i] or !star(a) or !star(b)) {
      if (star(a) != star(b) or s[a++] != s[b++]) {
        nast_id++;
        break;
      }
    }
    ids[b = sa[i]] = nast_id;
  }
  for (int i = 0; i < n; i++) {
    if (ids[i] == -1) continue;
    news.push_back(ids[i]);
    pos.push_back(i);
  }
  vector<int> new_sa(news.size());
  Sufar(news.data(), (int) news.size(), nast_id + 1, new_sa.data());
  fill(sa, sa + n, -1);
  copy(roz.begin() + 1, roz.end(), wsk.begin());
  reverse(new_sa.begin(), new_sa.end());
  for (int j : new_sa) sa[--wsk[s[pos[j]]]] = pos[j];
  Indukuj();
  if (lcp) {
    for (int i = 0; i < n; i++) ids[sa[i]] = i;
    for (int i = 0, k = 0; i < n; i++, k = max(0, k - 1)) {
      if (ids[i] == n - 1) { k = 0; continue; }
      const int j = sa[ids[i] + 1];
      while (i + k < n and j + k < n and s[i + k] == s[j + k]) k++;
      lcp[ids[i]] = k;
    }
  }
}
```

marek/Ukkonen.cpp

```
using Char = char;  // kInfinity musi być ściśle większe niż długość słowa.
constexpr int kInfinity = numeric_limits<int>::max();
struct Ukkonen {
  struct Node {
    map<Char, pair<Node*, pair<int, int>>> transition;
    Node* suflink;
  };
  // Ta metoda jest wywoływana zawsze gdy tworzona jest krawędź {node}[a, +oo).
  void CreateLeafCallback(Node* node, int a) {}
  // Ta metoda jest wywoływana zawsze gdy krawędź {node}[a, b] zamienia się
  // w dwie krawędzie: {node}[a, c-1], {middle}[c, b].
  void SplitEdgeCallback(Node* node, int a, int b, Node* middle, int c) {}
  Node* NewNode() { Node* node = new Node(); return node; /* Leaks. */ }
  Node *root, *pin, *last_explicit_node;
  vector<Char> text;  // Słowo powinno zajmować indeksy [0..n-1].
  int last_length;    // Liczba literek do ostatniego wierzchołka ,,implicit''.
  Ukkonen(const int reserve = 0) : root(nullptr), pin(nullptr) {
    text.reserve(reserve);              // "reserve" warto ustawić na maksymalną
    root = NewNode(); pin = NewNode();  // długość słowa, ale wcale nie trzeba.
    root->suflink = pin;
    last_explicit_node = root;
    last_length = 0;
  }
  void Canonize(Node** s, int* a, int b) {
    if (b < *a) return;
    pair<Node*, pair<int, int>> t = (*s)->transition[text[*a]];
    Node* sp = t.first;
    int ap = t.second.first, bp = t.second.second;
    while (bp - ap <= b - *a) {
      *a = *a + bp - ap + 1; *s = sp;
      if (*a <= b) {
        t = (*s)->transition[text[*a]];
        sp = t.first; ap = t.second.first; bp = t.second.second;
      }
    }
  }
  bool TestAndSplit(Node* s, int a, int b, Char c, Node** ret) {
    if (a <= b) {
      pair<Node*, pair<int, int>>& t = s->transition[text[a]];
      Node* sp = t.first;
      int ap = t.second.first, bp = t.second.second;
      if (c == text[ap + b - a + 1]) {
        *ret = s;
        return true;
      }
      *ret = NewNode();
      t.second.second = ap + b - a;
      t.first = *ret;
      (*ret)->transition[text[ap + b - a + 1]] =
          make_pair(sp, make_pair(ap + b - a + 1, bp));
      SplitEdgeCallback(s, ap, bp, *ret, ap + b - a + 1);
      return false;
    }
    *ret = s;
    return s->transition.find(c) != s->transition.end();
  }
  void Update(Node** s, int* a, int i) {
```

```
    Node *oldr = root, *r;
    bool end = TestAndSplit(*s, *a, i - 1, text[i], &r);
    while (!end) {
      CreateLeafCallback(r, i);
      r->transition[text[i]] = make_pair(nullptr, make_pair(i, kInfinity));
      if (oldr != root) oldr->suflink = r;
      oldr = r;
      *s = (*s)->suflink;
      Canonize(s, a, i - 1);
      end = TestAndSplit(*s, *a, i - 1, text[i], &r);
    }
    if (oldr != root) oldr->suflink = *s;
  }
  void AddLetter(Char z) {  // Dodaje kolejną literę do drzewa.
    const int i = static_cast<int>(text.size());
    text.push_back(z);
    auto it = pin->transition.find(z);
    if (it == pin->transition.end())
      pin->transition[z] = make_pair(root, make_pair(i, i));
    Update(&last_explicit_node, &last_length, i);
    Canonize(&last_explicit_node, &last_length, i);
  }
  void ClearInfinities(Node* node = nullptr) {  // Zamienia wszystkie krawędzie:
    if (node == nullptr) node = root;           // [x, +oo)->[x, text.size()-1].
    for (auto& it : node->transition) {
      if (it.second.second.second == kInfinity)
        it.second.second.second = (int) text.size() - 1;
      else ClearInfinities(it.second.first);
    }
  }
};
int main() {  // Przykład użycia.
  string s = "abcdefgh#";
  Ukkonen<char> u(s.size() /* reserve */);
  for (char c : s) u.AddLetter(c);
  u.ClearInfinities();
}
```

mateusz/CRT.cpp

```
/* Chińskie twierdzenie o resztach O(n*log(zakres)).                    */
/* Zarówno wynik, jak i argumenty są postaci x = first (mod second).    */
/* Jeśli kongruencja jest niespełnialna to zwraca (-1, -1).             */
using pll = pair<ll, ll>;
void eukl(ll &x, ll &y, ll a, ll b) {
  if (!a) { x = 0; y = 1; return; }
  eukl(y, x, b % a, a);
  x -= y * (b / a);
}
ll mno(ll a, ll b, ll mod) {//a moze byc ujemne
  return (__int128(a)*b)%mod;
}
pll crt2(ll p, ll a, ll q, ll b) {
  if (a==-1)
    return {-1, -1};
  ll x, y;
  eukl(x, y, a, b);
  ll nwd=x*a+y*b;
  if ((p%nwd)!=(q%nwd))
    return {-1, -1};
  a/=nwd;
  b/=nwd;
  ll nww=a*b;
  ll ret=mno(x*a, q/nwd, nww)+mno(y*b, p/nwd, nww);
  if ((ret%=nww)<0)
    ret+=nww;
  return {ret*nwd+(p%nwd), nww*nwd};
}
pll crt(vector <pll> wek) {
  pll ret={0, 1};
  for (auto i : wek)
    ret=crt2(ret.first, ret.second, i.first, i.second);
  return ret;
}
```

mateusz/DinicMarka.cpp

```
using T = long long;
bool iszero(T v) { return !v; /* Zmienić dla doubli. */ }
struct Flow {
  struct E { int dest; T orig, *lim, *rev; };
  int zr, uj, n = 0;
  vector<unique_ptr<T>> ts;
  vector<vector<E>> graf;
  vector<int> ptr, odl;
  void vert(int v) {
    n = max(n, v + 1);
    graf.resize(n); ptr.resize(n); odl.resize(n);
  }
  void bfs() {
    fill(odl.begin(), odl.end(), 0);
    vector<int> kol = {zr};
    odl[zr] = 1;
    for (int i = 0; i < (int) kol.size(); i++) {
      for (E& e : graf[kol[i]]) {
        if (!odl[e.dest] and !iszero(*e.lim)) {
          odl[e.dest] = odl[kol[i]] + 1;
          kol.push_back(e.dest);
        }
      }
    }
  }
  T dfs(int v, T lim) {
    if (v == uj) return lim;
    T ret = 0, wez;
    for (int& i = ptr[v]; i < (int) graf[v].size(); i++) {
      E& e = graf[v][i];
      if (odl[e.dest] == odl[v] + 1 and !iszero(*e.lim) and
          !iszero(wez = dfs(e.dest, min(*e.lim, lim)))) {
        ret += wez; *e.lim -= wez; *e.rev += wez; lim -= wez;
        if (iszero(lim)) break;
      }
    }
    return ret;
  }
  void add_edge(int u, int v, T lim, bool bi = false /* bidirectional? */) {
    vert(max(u, v));
    T *a = new T(lim), *b = new T(lim * bi);
```

```
    ts.emplace_back(a); ts.emplace_back(b);
    graf[u].push_back(E{v, lim,      a, b});
    graf[v].push_back(E{u, lim * bi, b, a});
  }
  T dinic(int zr_, int uj_) {
    zr = zr_; uj = uj_;
    vert(max(zr, uj));
    T ret = 0;
    while (true) {
      bfs();
      fill(ptr.begin(), ptr.end(), 0);
      const T sta = dfs(zr, numeric_limits<T>::max());  // Dla doubli można dać
      if (iszero(sta)) break;                           // infinity() zamiast
      ret += sta;                                       // max().
    }
    return ret;
  }
  vector<int> cut() {
    vector<int> ret;
    bfs();
    for (int i = 0; i < n; i++) if (odl[i]) ret.push_back(i);
    return ret;
  }
  map<pair<int, int>, T> get_flowing() {  // Tam gdzie plynie 0 może nie być
    map<pair<int, int>, T> ret;           // krawędzi.
    for (int i = 0; i < n; i++) for (E& e : graf[i])
        if (*e.lim < e.orig) ret[make_pair(i, e.dest)] += e.orig - *e.lim;
    for (auto& i : ret) {
      const pair<int, int> rev{i.first.second, i.first.first};
      const T x = min(i.second, ret[rev]);
      i.second -= x;
      ret[rev] -= x;
    }
    return ret;
  }
};
```

mateusz/DMST.cpp

```
#define int long long//jeśli long longi potrzebne
struct DMST {
  int N;
  vector<int> eFrom, eTo, eCost, ePrev, visited, cycle, parent;
  vector<vector<int>> cycles, adj, curEdge;
  int Root, fstEdge;

  DMST(int V) : N(V), visited(2*V), parent(2*V), cycles(2*V), adj(2*V),
        curEdge(2*V, vector<int>(2*V, -1)) {}

  void addEdge(int u, int v, int c, int prev = -1) {
    if (prev != -1) {
      if (curEdge[u][v] != -1) {
        int id = curEdge[u][v];
        if (eCost[id] > c) { eCost[id] = c; ePrev[id] = prev; }
        return;
      }
    }
    int id = (int)eFrom.size();
    if (u == v) {
      u = v = c = -1;
    } else {
      adj[u].push_back(id);
      curEdge[u][v] = id;
    }
    eFrom.push_back(u);
    eTo.push_back(v);
    eCost.push_back(c);
    ePrev.push_back(prev);
  }

  bool dfsCyc(int v) {
    if (v == Root) { return false; }
    visited[v] = 1;
    cycle.push_back(parent[v]);
    int p = eFrom[parent[v]];
    if (visited[p] == 1) { fstEdge = parent[p]; }
    bool res = visited[p] == 1 || (!visited[p] && dfsCyc(p));
    visited[v] = 2;
    return res;
  }

  vector<int> compute(int root) {
    Root = root;
    vector<bool> current(2 * N), onCycle(2 * N);
    vector<int> best(2 * N);
    fill_n(current.begin(), N, true);
    int curSz = N;

    while (true) {
      fill(best.begin(), best.end(), Infty);
      fill(onCycle.begin(), onCycle.end(), false);

      for (int i = 0; i < 2 * N; i++) {
        if (!current[i]) { continue; }
        for (int e : adj[i]) {
          int v = eTo[e], c = eCost[e];
          if (v != root && current[v] && c < best[v]) {
            best[v] = c; parent[v] = e;
          }
        }
      }
      fill(visited.begin(), visited.end(), 0);
      for (int i = 0; i < 2 * N; i++) {
        if (current[i] && !visited[i]) {
          cycle.clear();
          if (dfsCyc(i)) { break; } else { cycle.clear(); }
        }
      }
      if (cycle.empty()) { break; }
      cycle.erase(cycle.begin(), find(cycle.begin(), cycle.end(), fstEdge));
      cycles[curSz] = cycle;
      for (int v : cycle) { onCycle[eFrom[v]] = true; }

      for (int v = 0; v < 2 * N; v++) {
        if (!current[v]) { continue; }
        vector<int> edges = adj[v];
```

```
      for (int e : edges) {
        int s = eTo[e], c = eCost[e];
        if (!current[s]) { continue; }
        if (!(onCycle[v] ^ onCycle[s])) { continue; }
        if (onCycle[s]) { c -= best[s]; }
        addEdge(onCycle[v] ? curSz : v, onCycle[s] ? curSz : s, c, e);
      }
    }

    for (int v : cycle) { current[eFrom[v]] = false; }
    current[curSz++] = true;
  }

  for (int cyc = curSz - 1; cyc >= N; cyc--) {
    for (int v : cycles[cyc]) { parent[eTo[v]] = v; }
    int e = ePrev[parent[cyc]];
    parent[eTo[e]] = e;
    for (int v = 0; v < 2 * N; v++) {
      if (v != root && eFrom[parent[v]] == cyc) {
        parent[v] = ePrev[parent[v]];
      }
    }
  }
  parent[root] = -1;
  return vector<int>(parent.begin(), parent.begin() + N);
 }

 int getValue(vector<int> sol) {
  int total = 0;
  for (int i = 0; i < N; i++) { if (i != Root) { total += eCost[sol[i]]; } }
  return total;
 }

 const int Infty = 1e9;
};
#undef int
```

mateusz/Dominators.cpp

```
struct Dominators{
 int n_orig, n;
 vector<int> parent, semi, vertex, dom, ancestor, label;
 vector<vector<int>> succ, pred, bucket;

 Dominators(int _n):n_orig(_n), n(2 * (_n + 1)), parent(n), semi(n), vertex(n),
, dom(n), ancestor(n), label(n), succ(n), pred(n), bucket(n) {
  n = n_orig;
 }
 void add_edge(int a,int b){
  a++; b++;
  succ[a].push_back(b);
 }
 void COMPRESS(int v) {
  if (ancestor[ancestor[v]] != 0) {
   COMPRESS(ancestor[v]);
   if (semi[label[ancestor[v]]] < semi[label[v]]) {
    label[v] = label[ancestor[v]];
   }
   ancestor[v]=ancestor[ancestor[v]];
  }
 }
 void LINK(int v, int w) {
  ancestor[w]=v;
 }
 int EVAL(int v) {
  if(ancestor[v] == 0)
   return v;
  else {
   COMPRESS(v);
   return label[v];
  }
 }
 void DFS(int v) {
  semi[v] = ++n;
  vertex[n] = v;
  for(auto ng : succ[v]) {
   if(semi[ng] == 0) {
    parent[ng]=v;
    DFS(ng);
   }
   pred[ng].push_back(v);
  }
 }
 //dominatory z wierzcholka 0
 //zwraca vector dominatorow (-1 dla 0)
 vector<int> doit() {
  iota(label.begin(), label.end(), 0);
  DFS(1);
  for (int i = n; i >= 2; --i) {
   int w = vertex[i];
   for (auto ng : pred[w]) {
    int u = EVAL(ng);
    if (semi[u] < semi[w]) { semi[w] = semi[u]; }
   }
   bucket[vertex[semi[w]]].push_back(w);
   LINK(parent[w],w);
   while (!bucket[parent[w]].empty()) {
    int v = bucket[parent[w]].back();
    bucket[parent[w]].pop_back();
    int u = EVAL(v);
    if (semi[u] < semi[v]) {
     dom[v] = u;
    } else {
     dom[v] = parent[w];
    }
   }
  }
  for (int i = 2; i <= n; ++i) {
   int w = vertex[i];
   if (dom[w] != vertex[semi[w]]) { dom[w] = dom[dom[w]]; }
  }
  dom[1] = 0;
  vector<int> res(n_orig);
  for (int i = 0; i < n_orig; i++)res[i] = dom[i + 1] - 1;
  return res;
 }
```

```
};
```

mateusz/Eertree.cpp

```
/* DRZEWO PALINDROMÓW O(n).                                                */
/* n to aktualna liczba dodanych liter+1, last to najdłuższy palindro-sufiks  */
/* aktualnego słowa, sz to aktualny rozmiar drzewa                         */
/*                                                                         */
/* Poniżej kod liczący podział na parzyste palindromy minimalizujący liczbę   */
/* palindromów dłuższych niż 2.                                            */
const int maxn = 1000*1000+7, sigma = 26;
int len[maxn], link[maxn], to[maxn][sigma];
int slink[maxn], diff[maxn];
pair <int,int> series_ans[maxn];
int ans[maxn], z[maxn];
int sz, last, n;
char s[maxn];

void init() {
    s[n++] = -1;
    link[0] = 1;
    len[1] = -1;
    sz = 2;
}
int get_link(int v) {
    while(s[n - len[v] - 2] != s[n - 1]) v = link[v];
    return v;
}
void add_letter(char c) {
    s[n++] = c -= 'a';
    last = get_link(last);
    if(!to[last][c])
    {
        len[sz] = len[last] + 2;
        link[sz] = to[get_link(link[last])][c];
        diff[sz] = len[sz] - len[link[sz]];
        if(diff[sz] == diff[link[sz]])
            slink[sz] = slink[link[sz]];
        else
            slink[sz] = link[sz];
        to[last][c] = sz++;
    }
    last = to[last][c];
}
int main() {
    init();
    for(int i = 1; i <= nn; i++) {
        add_letter(tek[i]);
        for(int v = last; len[v] > 0; v = slink[v]) {
            series_ans[v] = {ans[i - (len[slink[v]] + diff[v])], i - (len[slink
↳[v]] + diff[v])};
            if(diff[v] == diff[link[v]])
                series_ans[v] = min(series_ans[v], series_ans[link[v]]);
            if (!(i&1)) {
                if (series_ans[v].first+1<ans[i]) {
                    ans[i] = series_ans[v].first + 1;
                    z[i] = series_ans[v].second;
                }
            }
        }
        if (!(i&1) && tek[i]==tek[i-1] && ans[i-2]<ans[i]) {
            ans[i]=min(ans[i], ans[i-2]);
            z[i]=i-2;
        }
    }
}
```

mateusz/Gomory_Hu.cpp

```
//wymaga naszego dinica
#define int long long//jeśli long longi potrzebne
struct GomoryHu {
  vector<vector< pair<int,int> >> graph, tree;
  vector<vector<int>> nodes;
  vector<bool> visited;
  vector<int> groupId, contrId;
  int wnode, n;
  GomoryHu(int N) : graph(N), visited(N), groupId(N), contrId(N), tree(N), n(N)
↳ {}

  void addEdge(int u, int v, int cap) {
    graph[u].emplace_back(v, cap);
    graph[v].emplace_back(u, cap);
  }
  void dfs(int v, int type) {
    visited[v] = true; contrId[v] = type;
    for (auto P : tree[v]) { if (!visited[P.first]) { dfs(P.first, type); } }
  }

  vector <pair<pair<int,int>,int>> run() {
    vector<int> allNodes(n);
    iota(allNodes.begin(), allNodes.end(), 0);
    nodes = vector<vector<int>>{allNodes};
    tree = vector<vector<pair<int,int>>>(n);
    fill(groupId.begin(), groupId.end(), 0);

    for (int step = 1; step < n; step++) {
      Flow flow;
      for (int i = 0; i < (int)nodes.size(); i++) {
        if ((int)nodes[i].size() > 1) { wnode = i; break; }
      }
      fill(visited.begin(), visited.end(), false);

      visited[wnode] = true;
      for (auto P : tree[wnode]) { dfs(P.first, nodes[P.first][0]); }
      for (int v = 0; v < n; v++) {
        int a = groupId[v] == wnode ? v : contrId[groupId[v]];
        for (auto& P : graph[v]) {
          int b = groupId[P.first] == wnode ? P.first : contrId[groupId[P.first
↳]];
          if (a != b) { flow.add_edge(a, b, P.second); }
        }
      }

      int a = nodes[wnode][0], b = nodes[wnode][1], f = flow.dinic(a, b);
      auto pom = flow.cut();
      vector <bool> cut(n, false);
      for (int i : pom)
```

```
      cut[i]=1;

    for (int v = 0; v < step; v++) {
      if (v == wnode) { continue; }
      for (auto& P : tree[v]) {
        if (P.first == wnode && !cut[contrId[v]]) { P.first = step; }
      }
    }
    vector<pair<int,int>> PA, PB;
    for (auto& P : tree[wnode]) { (cut[contrId[P.first]] ? PA : PB).push_back
(P); }
    tree[wnode] = PA; tree[step] = PB;
    tree[wnode].emplace_back(step, f);
    tree[step].emplace_back(wnode, f);
    vector<int> A, B;
    for (int v : nodes[wnode]) {
      (cut[v] ? A : B).push_back(v);
      if (!cut[v]) { groupId[v] = step; }
    }
    nodes[wnode] = A;
    nodes.push_back(B);
  }

  vector <pair<pair<int,int>,int>> res;
  for (int i = 0; i < n; i++)
    for (auto P : tree[i])
      if (nodes[i][0]<nodes[P.first][0])
        res.push_back({{nodes[i][0], nodes[P.first][0]}, P.second});
  return res;
 }
};
#undef int
```

mateusz/Graf_Podslow.cpp

```
struct suffix_automaton {
  vector<map<char,int>> edges;
  vector<int> link;
  vector<int> length;
  int last;// wierzcholek z calym stringiem, byc moze to nie najwiekszy numer
  suffix_automaton(string s) {
    // add the initial node
    edges.push_back(map<char,int>());
    link.push_back(-1);
    length.push_back(0);
    last = 0;
    for (int i=0; i<s.size(); i++) {
      // construct r
      edges.push_back(map<char,int>());
      length.push_back(i+1);
      link.push_back(0);
      int r = edges.size() - 1;
      // add edges to r and find p with link to q
      int p = last;
      while (p >= 0 && edges[p].find(s[i]) == edges[p].end()) {
        edges[p][s[i]] = r;
        p = link[p];
      }
      if (p != -1) {
        int q = edges[p][s[i]];
        if (length[p] + 1 == length[q]) {
          // we do not have to split q, just set the correct suffix link
          link[r] = q;
        }
        else {
          // we have to split, add q'
          edges.push_back(edges[q]); // copy edges of q
          length.push_back(length[p] + 1);
          link.push_back(link[q]); // copy parent of q
          int qq = edges.size()-1;
          // add qq as the new parent of q and r
          link[q] = qq;
          link[r] = qq;
          // move short classes pointing to q to point to q'
          while (p >= 0 && edges[p][s[i]] == q) {
            edges[p][s[i]] = qq;
            p = link[p];
          }
        }
      }
      last = r;
    }
  }
};
```

mateusz/HLD.cpp

```
// Przedziały odpowiadające ścieżce z v do lca mają first>=second, zaś te dla
// ścieżki z lca do u mają first<=second, przedziały są po kolei, lca występuje
// tam dwa razy, najpierw jako second, a zaraz potem jako first.
const int nax = 100 * 1007;
vector<int> drz[nax];
int prel, roz[nax], jump[nax], pre[nax], post[nax], fad[nax];
void dfs_roz(int v) {
  roz[v] = 1;
  for (int& i : drz[v]) {
    fad[i] = v;
    dfs_roz(i);
    roz[v] += roz[i];
    if (roz[i] > roz[drz[v][0]]) swap(i, drz[v][0]);
  }
}
void dfs_pre(int v) {
  if (!jump[v]) jump[v] = v;
  pre[v] = ++prel;
  if (!drz[v].empty()) jump[drz[v][0]] = jump[v];
  for (int i : drz[v]) dfs_pre(i);
  post[v] = prel;
}
int lca(int v, int u) {
  while (jump[v] != jump[u]) {
    if (pre[v] < pre[u]) swap(v, u);
    v = fad[jump[v]];
  }
  return (pre[v] < pre[u] ? v : u);
}
vector<pair<int, int>> path_up(int v, int u) {
  vector<pair<int, int>> ret;
```

```
  while (jump[v] != jump[u]) {
    ret.emplace_back(pre[jump[v]], pre[v]);
    v = fad[jump[v]];
  }
  ret.emplace_back(pre[u], pre[v]);
  return ret;
}
vector<pair<int, int>> get_path(int v, int u) {
  int w = lca(v, u);
  auto ret = path_up(v, w);
  auto pom = path_up(u, w);
  for (auto& i : ret) swap(i.first, i.second);
  while (!pom.empty()) {
    ret.push_back(pom.back());
    pom.pop_back();
  }
  return ret;
}
```

mateusz/Hungarian.cpp

```
/* HUNGARIAN O(n^3) - Maksymalne najdroższe skojarzenie w pełnym grafie
 dwudzielnym o równolicznych zbiorach wierzchołków. Dostaje macierz z wagami.
 Zwraca wektor 'one', gdzie wierzcholek 'i' jest sparowany z 'one[i]' po prawej.
 Jak chcemy najtańsze, to bierzemy wszystko z minusem. Jak nie ma krawędzi,
 to dajemy -INF przy założeniu: n * |waga| < INF. Indeksujemy od 0. */
#define REP(i, n) for(int i = 0; i < int(n); ++i)
vector<int> hungarian2(const vector<vector<int>> & w) {
  const int n = w.size();
  vector<int> one(n, -1), two(n, -1), L(n), R(n), par(n);
  REP(i, n) L[i] = *max_element(w[i].begin(), w[i].end());
  REP(rep, n) {
    vector<bool> left(n), right(n);
    vector<int> slack(n, INT_MAX), q;
    int x = -1;
    REP(i, n) if(one[i] == -1) q.push_back(i);
    while(x == -1) {
      REP(z, q.size()) {
        int a = q[z];
        left[a] = true;
        REP(b, n) {
          int tmp = L[a] + R[b] - w[a][b];
          if(!right[b] && tmp < slack[b]) {
            par[b] = a;
            slack[b] = tmp;
            if(tmp == 0) {
              right[b] = true;
              if(two[b] != -1) q.push_back(two[b]);
              else { x = b; goto koniec; }
            }
          }
        }
      }
      int val = INT_MAX;
      REP(i, n) if(!right[i]) val = min(val, slack[i]);
      REP(i, n) {
        if(left[i]) L[i] -= val;
        if(right[i]) R[i] += val;
        else if((slack[i] -= val) == 0) {
          right[i] = true;
          if(two[i] != -1) q.push_back(two[i]);
          else x = i;
        }
      }
    }
    koniec:
    while(x != -1) {
      int tmp = one[par[x]];
      one[par[x]] = x;
      two[x] = par[x];
      x = tmp;
    }
  }
  return one;
}
```

mateusz/Pi.cpp

```
struct Primes {
  vector <ll> w, dp;
  int gdz(ll v) {
    if (v<=w.back()/v)
      return v-1;
    return w.size()-w.back()/v;
  }
  ll pi(ll n) {
    for (ll i=1; i*i<=n; i++) {
      w.push_back(i);
      if ((n/i)!=i)
        w.push_back(n/i);
    }
    sort(w.begin(), w.end());
    for (ll i : w)
      dp.push_back(i-1);
    for (ll i=1; (i+1)*(i+1)<=n; i++) {
      if (dp[i]==dp[i-1])
        continue;
      for (int j=(int)w.size()-1; w[j]>=(i+1)*(i+1); j--)
        dp[j]-=dp[gdz(w[j]/(i+1))]-dp[i-1];
    }
    return dp.back();
  }
  ll ask(ll v) {//v==n/u for some u
    return dp[gdz(v)];
  }
};
```

mateusz/SSS.cpp

```
const int nax=100*1007;
vector <int> graf[nax], farg[nax];
int ost[nax], bylo[nax], post[nax], spo[nax], counter, coudfs;
vector <vector<pair<int,int>>> mer;
void dfs1(int v) {
  if (bylo[v]) return;
  bylo[v]=1;
  for (int i : graf[v]) dfs1(i);
  coudfs--;
  post[coudfs]=v;
}
```

```
void dfs2(int v, int s) {
  if (spo[v]>=0) return;
  spo[v]=s;
  for (int i : farg[v]) dfs2(i, s);
}
void rek(int l, int r, vector <pair<pair<int,int>,int>> &kra) {
  if (l>r) return;
  counter++;
  vector <int> ver;
  for (auto i : kra) {
    if (ost[i.first.first]<counter) {
      ver.push_back(i.first.first);
      ost[i.first.first]=counter;
    }
    if (ost[i.first.second]<counter) {
      ver.push_back(i.first.second);
      ost[i.first.second]=counter;
    }
  }
  for (int i : ver) {
    bylo[i]=0;
    spo[i]=-1;
    graf[i].clear();
    farg[i].clear();
  }
  int s=(l+r)>>1;
  for (auto i : kra) {
    if (i.second<=s) {
      graf[i.first.first].push_back(i.first.second);
      farg[i.first.second].push_back(i.first.first);
    }
  }
  coudfs=ver.size();
  for (int i : ver) dfs1(i);
  for (int i=0; i<(int)ver.size(); i++)
    dfs2(post[i], post[i]);
  for (int i : ver)
    if (i!=spo[i])
      mer[s].push_back({i, spo[i]});
  vector <pair<pair<int,int>,int>> lew, pra;
  for (auto i : kra) {
    if (spo[i.first.first]==spo[i.first.second])
      lew.push_back(i);
    else
      pra.push_back({{spo[i.first.first],spo[i.first.second]}, i.second});
  }
  rek(l, s-1, lew);
  rek(s+1, r, pra);
}
void sss(vector <pair<int,int>> kra)
{
  mer.clear();
  mer.resize(kra.size());
  vector <pair<pair<int,int>,int>>daj;
  for (int i=0; i<(int)kra.size(); i++) {
    daj.push_back({kra[i], i});
    ost[kra[i].first]=-1;
    ost[kra[i].second]=-1;
  }
  counter=0;
  rek(0, (int)kra.size()-1, daj);
}
```

mateusz/wzory.tex

Długość wykresu funkcji $f : [a, b] \to \mathbb{R}$: $\int_a^b \sqrt{f'(x)^2 + 1}\, dx$.

Pole figury obrotowej $f : [a, b] \to \mathbb{R}$: $2\pi \int_a^b |f(x)| \sqrt{f'(x)^2 + 1}\, dx$.

Objętość figury obrotowej $f : [a, b] \to \mathbb{R}$: $\pi \int_a^b f(x)^2\, dx$.

$\int \sqrt{x^2+1}\, dx = \frac{1}{2}\left(x\sqrt{x^2+1} + \operatorname{arcsinh} x\right) + c$

$\int \sqrt{x^2+1}\,\mathrm{dx} = \frac{1}{2}\left(x\sqrt{x^2+1} + \operatorname{arcsinh} x\right) + c$

$\int \sqrt{x^2+1}\,\mathrm{dx} = \frac{1}{2}\left(x\sqrt{x^2+1} + \operatorname{arcsinh} x\right) + c$ $(\operatorname{arcsinh} = \texttt{asinh})$

$\int \sqrt{1-x^2}\,\mathrm{dx} = \frac{1}{2}\left(x\sqrt{1-x^2} + \arcsin x\right) + c$

$\int \frac{1}{ax^2+bx+c}\,\mathrm{dx} = \frac{2}{\sqrt{4ac-b^2}} \arctan \frac{2ax+b}{\sqrt{4ac-b^2}}$ $(\Delta < 0)$

$\int \frac{x}{ax^2+bx+c}\,\mathrm{dx} = \frac{1}{2a} \ln|ax^2+bx+c| - \frac{b}{2a} \int \frac{\mathrm{dx}}{ax^2+bx+c}$

$\int \tan x\,\mathrm{dx} = -\ln|\cos x| + c$

$(\arcsin x)' = \frac{1}{\sqrt{1-x^2}}$, $(\arccos x)' = -\frac{1}{\sqrt{1-x^2}}$

$\frac{1}{\pi} = 0.31831$, $\pi^2 = 9.86960$, $\frac{1}{\pi^2} = 0.10132$, $\frac{1}{e} = 0.36788$, $\gamma = 0.577215664901532$

$H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + O(n^{-4})$

$\ln n! = n \ln n - n + \frac{1}{2}\ln(2\pi n) + \frac{1}{12n} - \frac{1}{360n^3} + \frac{1}{1260n^5} - O(n^{-7})$