The AC is a lie - ICPC Library

Contents

1 Data Structures

1.1 BIT 2D Comprimida

```
// src: tfg50
template<class T = int>
struct Bit2D {
public:
  Bit2D(vector<pair<T, T>> pts) {
    sort(pts.begin(), pts.end());
    for(auto a : pts) {
      if(ord.empty() || a.first != ord.back()) {
        ord.push_back(a.first);
      }
    }
    fw.resize(ord.size() + 1);
    coord.resize(fw.size());
    for(auto &a : pts) {
      swap(a.first, a.second);
    }
    sort(pts.begin(), pts.end());
    for(auto &a : pts) {
      swap(a.first, a.second);
      for(int on = upper_bound(ord.begin(), ord.end(), a.first) - ord.
          begin(); on < fw.size(); on += on & -on) {
        if(coord[on].empty() || coord[on].back() != a.second) {
          coord[on].push_back(a.second);
        }
      }
    }
    for(int i = 0; i < fw.size(); i++) {
      fw[i].assign(coord[i].size() + 1, 0);
    }
  }

  void upd(T x, T y, T v) {
    for(int xx = upper_bound(ord.begin(), ord.end(), x) - ord.begin();
        xx < fw.size(); xx += xx & -xx) {
```

```
      for(int yy = upper_bound(coord[xx].begin(), coord[xx].end(), y)
          - coord[xx].begin(); yy < fw[xx].size(); yy += yy & -yy) {
        fw[xx][yy] += v;
      }
    }
  }

  T qry(T x, T y) {
    T ans = 0;
    for(int xx = upper_bound(ord.begin(), ord.end(), x) - ord.begin();
         xx > 0; xx -= xx & -xx) {
      for(int yy = upper_bound(coord[xx].begin(), coord[xx].end(), y)
          - coord[xx].begin(); yy > 0; yy -= yy & -yy) {
        ans += fw[xx][yy];
      }
    }
    return ans;
  }

  T qry(T x1, T y1, T x2, T y2) {
    return qry(x2, y2) - qry(x2, y1 - 1) - qry(x1 - 1, y2) + qry(x1 -
        1, y1 - 1);
  }

  void upd(T x1, T y1, T x2, T y2, T v) { // !insert these points
    upd(x1, y1, v);
    upd(x1, y2 + 1, -v);
    upd(x2 + 1, y1, -v);
    upd(x2 + 1, y2 + 1, v);
  }
private:
  vector<T> ord;
  vector<vector<T>> fw, coord;
};
```

1.2 Iterative Segment Tree

```
int n, t[2 * ms];

void build() {
  for(int i = n - 1; i > 0; --i) t[i] = t[i<<1] + t[i<<1|1]; // Merge
}

void update(int p, int value) { // set value at position p
  for(t[p += n] = value; p > 1; p >>= 1) t[p>>1] = t[p] + t[p^1]; //
      Merge
}

int query(int l, int r) {
  int res = 0;
  for(l += n, r += n+1; l < r; l >>= 1, r >>= 1) {
    if(l&1) res += t[l++]; // Merge
    if(r&1) res += t[--r]; // Merge
  }
  return res;
}

// If is non-commutative
S query(int l, int r) {
  S resl, resr;
  for (l += n, r += n+1; l < r; l >>= 1, r >>= 1) {
  if (l&1) resl = combine(resl, t[l++]);
  if (r&1) resr = combine(t[--r], resr);
  }
  return combine(resl, resr);
}
```

1.3 Iterative Segment Tree with Lazy Propagation

```
struct LazyContext {
  int v;

  LazyContext(int v = 0) : v(v) { }

  void reset() {
  v = 0;
  }

  void operator += (LazyContext o) {
  v += o.v;
  }
};

struct Node {
  int sz, v;

  Node() { // neutral element
        v = 0; sz = 0;
  }

  Node(int i) { // init
        v = i; sz = 1;
  }

  Node(Node &l, Node &r) { // merge
        sz = l.sz + r.sz;
        v = l.v + r.v;
  }

  void apply(LazyContext lazy) {
  v += lazy.v * sz;
  }
};

Node tree[2*ms];
LazyContext lazy[ms];
bool dirty[ms];
int n, h, a[ms];

void init() {
    h = 0;
    while((1 << h) < n) h++;
    for(int i = 0; i < n; i++) {
        tree[i + n] = Node(a[i]);
    }
    for(int i = n - 1; i > 0; i--) {
        tree[i] = Node(tree[i + i], tree[i + i + 1]);
        lazy[i].reset();
        dirty[i] = 0;
    }
```

```
    }

    void apply(int p, LazyContext &lc) {
        tree[p].apply(lc);
        if(p < n) {
            dirty[p] = true;
            lazy[p] += lc;
        }
    }

    void push(int p) {
        for(int s = h; s > 0; s--) {
            int i = p >> s;
            if(dirty[i]) {
                apply(i + i, lazy[i]);
                apply(i + i + 1, lazy[i]);
                lazy[i].reset();
                dirty[i] = false;
            }
        }
    }

    void build(int p) {
        for(p /= 2; p > 0; p /= 2) {
            tree[p] = Node(tree[p + p], tree[p + p + 1]);
            if(dirty[p]) {
                tree[p].apply(lazy[p]);
            }
        }
    }

    Node query(int l, int r) {
        if(l > r) return Node();
        l += n, r += n+1;
        push(l);
        push(r - 1);
        Node lp, rp;
        for(; l < r; l /= 2, r /= 2) {
            if(l & 1) lp = Node(lp, tree[l++]);
            if(r & 1) rp = Node(tree[--r], rp);
        }
        return Node(lp, rp);
    }

    void update(int l, int r, LazyContext lc) {
        if(l > r) return;
        l += n, r += n+1;
        push(l);
        push(r - 1);
        int l0 = l, r0 = r;
        for(; l < r; l /= 2, r /= 2) {
            if(l & 1) apply(l++, lc);
            if(r & 1) apply(--r, lc);
        }
        build(l0);
        build(r0 - 1);
    }
```

1.4 Segment Tree with Lazy Propagation

```
int arr[ms],  seg[4 * ms], lazy[4 * ms], n;

void build(int idx = 0, int l = 0, int r = n-1) {
  int mid = (l+r)/2;
  lazy[idx] = 0;
  if(l == r) {
    seg[idx] = arr[l];
    return;
  }
  build(2*idx+1, l, mid); build(2*idx+2, mid+1, r);
  seg[idx] = seg[2*idx+1] + seg[2*idx+2]; // Merge
}

void apply(int idx, int l, int r) {
  int mid = (l+r)/2;
  if(lazy[idx] && !canBreak) { // if not beats canBreak = false
    if(l < r) {
      lazy[2*idx+1] += lazy[idx];  // Merge de lazy
      lazy[2*idx+2] += lazy[idx];  // Merge de lazy
    }
    if(canApply) { // if not beats canApply = true
      seg[idx] += lazy[idx] * (r - l + 1); // Aplicar lazy no seg
    } else {
      apply(2*idx+1, l, mid); apply(2*idx+2, mid+1, r);
      seg[idx] = seg[2*idx+1] + seg[2*idx+2]; // Merge
    }
  }
  lazy[idx] = 0; // Limpar a lazy
}

int query(int L, int R, int idx = 0, int l = 0, int r = n-1) {
  int mid = (l+r)/2;
  apply(idx, l, r);
  if(l > R || r < L) return 0; // Valor que nao atrapalhe
  if(L <= l && r <= R) return seg[idx];
  return query(L, R, 2*idx+1, l, mid) + query(L, R, 2*idx+2, mid+1, r)
      ; // Merge
}

void update(int L, int R, int V, int idx = 0, int l = 0, int r = n-1)
    {
  int mid = (l+r)/2;
  apply(idx, l, r);
  if(l > R || r < L) return;
  if(L <= l && r <= R) {
    lazy[idx] = V;
    apply(idx, l, r);
    return;
  }
  update(L, R, V, 2*idx+1, l, mid); update(L, R, V, 2*idx+2, mid+1, r)
      ;
  seg[idx] = seg[2*idx+1] + seg[2*idx+2]; // Merge
}
```

1.5 Persistent Segment Tree

```
struct PSEGTREE{
  private:
    int z, t, sz, *tree, *L, *R, head[112345];
```

```
    void _build(int l, int r, int on, vector<int> &v){
      if(l == r){
        tree[on] = v[l];
        return;
      }
      L[on] = ++z;
      int mid = (l+r)>>1;
      _build(l, mid, L[on], v);
      R[on] = ++z;
      _build(mid+1, r, R[on], v);
      tree[on] = tree[L[on]] + tree[R[on]];
    }

    int _upd(int ql, int qr, int val, int l, int r, int on){
      if(l > qr || r < ql) return on;
      int curr = ++z;
      if(l >= ql && r <= qr){
        tree[curr] = tree[on] + val;
        return curr;
      }
      int mid = (l+r)>>1;
      L[curr] = _upd(ql, qr, val, l, mid, L[on]);
      R[curr] = _upd(ql, qr, val, mid+1, r, R[on]);
      tree[curr] = tree[L[curr]] + tree[R[curr]];
      return curr;
    }

    int _query(int ql, int qr, int l, int r, int on){
      if(l > qr || r < ql) return 0;
      if(l >= ql && r <= qr){
        return tree[on];
      }
      int mid = (l+r)>>1;
      return _query(ql, qr, l, mid, L[on]) + _query(ql, qr, mid+1, r,
          R[on]);
    }

  public:
    PSEGTREE(vector<int> &v){
      tree = new int[1123456];
      L = new int[1123456];
      R = new int[1123456];
      build(v);
    }
    void build(vector<int> &v){
      t = 0, z = 0;
      sz = v.size();
      head[0] = 0;
      _build(0, sz-1, 0, v);
    }

    void upd(int pos, int val, int idx){
      head[++t] = _upd(pos, pos, val, 0, sz-1, head[idx]);
    }

    int query(int l, int r, int idx){
      return _query(l, r, 0, sz-1, head[idx]);
    }
};
```

1.6 Treap

```
mt19937 rng ((int) chrono::steady_clock::now().time_since_epoch().
    count());

typedef int Value;
typedef struct item * pitem;

struct item {
  item () {}
  item (Value v) { // add key if not implicit
    value = v;
    prio = uniform_int_distribution<int>() (rng);
    cnt = 1;
    rev = 0;
    l = r = 0;
  }
  pitem l, r;
  Value value;
  int prio, cnt;
  bool rev;
};

int cnt (pitem it) {
  return it ? it->cnt : 0;
}

void fix (pitem it) {
  if (it)
    it->cnt = cnt(it->l) + cnt(it->r) + 1;
}

void pushLazy (pitem it) {
  if (it && it->rev) {
    it->rev = false;
    swap(it->l, it->r);
    if (it->l)  it->l->rev ^= true;
    if (it->r)  it->r->rev ^= true;
  }
}

void merge (pitem & t, pitem l, pitem r) {
  pushLazy (l); pushLazy (r);
  if (!l || !r) t = l ? l : r;
  else if (l->prio > r->prio)
    merge (l->r, l->r, r),  t = l;
  else
    merge (r->l, l, r->l),  t = r;
  fix (t);
}

void split (pitem t, pitem & l, pitem & r, int key) {
  if (!t) return void( l = r = 0 );
  pushLazy (t);
  int cur_key = cnt(t->l); // t->key if not implicit
  if (key <= cur_key)
    split (t->l, l, t->l, key),  r = t;
  else
    split (t->r, t->r, r, key - (1 + cnt(t->l))),  l = t;
  fix (t);
```

```
}

void reverse (pitem t, int l, int r) {
        pitem t1, t2, t3;
        split (t, t1, t2, l);
        split (t2, t2, t3, r-l+1);
        t2->rev ^= true;
        merge (t, t1, t2);
        merge (t, t, t3);
}

void unite (pitem & t, pitem l, pitem r) {
  if (!l || !r) return void ( t = l ? l : r );
  if (l->prio < r->prio)  swap (l, r);
  pitem lt, rt;
  split (r, lt, rt, l->key);
  unite (l->l, l->l, lt);
  unite (l-> r, l->r, rt);
  t = l;
}
```

1.7 Persistent Treap

```
mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count())
    ;

typedef int Key;
struct Treap {
  Treap(){}
  Treap(char k) {
    key = 1;
    size = 1;
    l = r = NULL;
    val = k;
  }

  Treap *l, *r;
  Key key;
  char val;
  int size;
};

typedef Treap * PTreap;

bool leftSide(PTreap l, PTreap r) {
  return (int) (rng() % (l->size + r->size)) < l->size;
}

void fix(PTreap t) {
  if (t == NULL) {
    return;
  }
  t->size = 1;
  t->key = 1;
  if (t->l) {
    t->size += t->l->size;
    t->key += t->l->size;
  }
  if (t->r) {
    t->size += t->r->size;
  }
}

void split(PTreap t, Key key, PTreap &l, PTreap &r) {
  if (t == NULL) {
    l = r = NULL;
  } else if (t->key <= key) {
    l = new Treap();
    *l = *t;
    split(t->r, key - t->key, l->r, r);
    fix(l);
  } else {
    r = new Treap();
    *r = *t;
    split(t->l, key, l, r->l);
    fix(r);
  }
}

void merge(PTreap &t, PTreap l, PTreap r) {
  if (!l || !r) {
    t = l ? l : r;
    return;
  }
  t = new Treap();
  if (leftSide(l, r)) {
    *t = *l;
    merge(t->r, l->r, r);
  } else {
    *t = *r;
    merge(t->l, l, r->l);
  }
  fix(t);
}

vector<PTreap> ver = {NULL};

PTreap build(int l, int r, string& s) {
  if (l >= r) return NULL;
  int mid = (l + r) >> 1;
  auto ans = new Treap(s[mid]);
  ans->l = build(l, mid, s);
  ans->r = build(mid + 1, r, s);
  fix(ans);
  return ans;
}

int last = 0;

void go(PTreap t, int f) {
  if (!t) return;
  go(t->l, f);
  cout << t->val;
  last += (t->val == 'c') * f;
  go(t->r, f);
}

void insert(PTreap t, int pos, string& s) {
  PTreap l, r;
  split(t, pos + 1, l, r);
  PTreap mid = build(0, s.size(), s);
```

```
  merge(mid, l, mid);
  merge(mid, mid, r);
  ver.push_back(mid);
}

void erase(PTreap t, int L, int R) {
  PTreap l, mid, r;
  split(t, L, l, mid);
  split(mid, R - L + 1, mid, r);
  merge(l, l, r);
  ver.push_back(l);
}
```

1.8 KD-Tree

```
int d;
long long getValue(const PT &a) {return (d & 1) == 0 ? a.x : a.y; }
bool comp(const PT &a, const PT &b) {
  if((d & 1) == 0) { return a.x < b.x; }
  else { return a.y < b.y; }
}
long long sqrDist(PT a, PT b) { return (a - b) * (a - b); }

class KD_Tree {
public:
  struct Node {
    PT point;
    Node *left, *right;
  };

  void init(std::vector<PT> pts) {
    if(pts.size() == 0) {
      return;
    }
    int n = 0;
    tree.resize(2 * pts.size());
    build(pts.begin(), pts.end(), n);
    //assert(n <= (int) tree.size());
  }

  long long nearestNeighbor(PT point) {
    // assert(tree.size() > 0);
    long long ans = (long long) 1e18;
    nearestNeighbor(&tree[0], point, 0, ans);
    return ans;
  }
private:
  std::vector<Node> tree;

  Node* build(std::vector<PT>::iterator l, std::vector<PT>::iterator r
      , int &n, int h = 0) {
    int id = n++;
    if(r - l == 1) {
      tree[id].left = tree[id].right = NULL;
      tree[id].point = *l;
    } else if(r - l > 1) {
      std::vector<PT>::iterator mid = l + ((r - l) / 2);
      d = h;
      std::nth_element(l, mid - 1, r, comp);
      tree[id].point = *(mid - 1);
      // BE CAREFUL!
      // DO EVERYTHING BEFORE BUILDING THE LOWER PART!
      tree[id].left = build(l, mid, n, h^1);
      tree[id].right = build(mid, r, n, h^1);
    }
    return &tree[id];
  }

  void nearestNeighbor(Node* node, PT point, int h, long long &ans) {
    if(!node) {
      return;
    }
    if(point != node->point) {
      // THIS WAS FOR A PROBLEM
      // THAT YOU DON'T CONSIDER THE DISTANCE TO ITSELF!
      ans = std::min(ans, sqrDist(point, node->point));
    }
    d = h;
    long long delta = getValue(point) - getValue(node->point);
    if(delta <= 0) {
      nearestNeighbor(node->left, point, h^1, ans);
      if(ans > delta * delta) {
        nearestNeighbor(node->right, point, h^1, ans);
      }
    } else {
      nearestNeighbor(node->right, point, h^1, ans);
      if(ans > delta * delta) {
        nearestNeighbor(node->left, point, h^1, ans);
      }
    }
  }
};
```

1.9 Sparse Table

```
template<class Info_t>
class SparseTable {
private:
  vector<int> log2;
  vector<vector<Info_t>> table;

  Info_t merge(Info_t &a, Info_t &b) {

  }

public:
  SparseTable(int n, vector<Info_t> v) {
    log2.resize(n + 1);
    log2[1] = 0;
    for (int i = 2; i <= n; i++) {
      log2[i] = log2[i >> 1] + 1;
    }
    table.resize(n + 1);
    for (int i = 0; i < n; i++) {
      table[i].resize(log2[n] + 1);
    }
    for (int i = 0; i < n; i++) {
      table[i][0] = v[i];
    }
    for (int i = 0; i < log2[n]; i++) {
```

```
      for (int j = 0; j < n; j++) {
        if (j + (1 << i) >= n) break;
        table[j][i + 1] = merge(table[j][i], table[j + (1 << i)][i]);
      }
    }
  }

  int get(int l, int r) {
    int k = log2[r - l + 1];
    return merge(table[l][k], table[r - (1 << k) + 1][k]);
  }
};
```

1.10 Max Queue

```
// src: tfg50
template <class T, class C = std::less<T>>
struct MaxQueue {
  MaxQueue() {
    clear();
  }

  void clear() {
    id = 0;
    q.clear();
  }

  void push(T x) {
    std::pair<int, T> nxt(1, x);
    while(q.size() > id && cmp(q.back().second, x)) {
      nxt.first += q.back().first;
      q.pop_back();
    }
    q.push_back(nxt);
  }

  T qry() {
    return q[id].second;
  }

  void pop() {
    q[id].first--;
    if(q[id].first == 0) {
      id++;
    }
  }
private:
  std::vector<std::pair<int, T>> q;
  int id;
  C cmp;
};
```

1.11 Policy Based Structures

```
#include <ext/pb_ds/assoc_container.hpp> // Common file
#include <ext/pb_ds/tree_policy.hpp> // Including
    tree_order_statistics_node_update
using namespace __gnu_pbds;

typedef tree<int, null_type, less<int>, rb_tree_tag,
tree_order_statistics_node_update> ordered_set;

ordered_set X;
X.insert(1);
X.find_by_order(0);
X.order_of_key(-5);
end(X), begin(X);
```

1.12 Color Updates Structure

```
struct range {
  int l, r;
  int v;

  range(int l = 0, int r = 0, int v = 0) : l(l), r(r), v(v) {}

  bool operator < (const range &a) const {
    return l < a.l;
  }
};

set<range> ranges;

vector<range> update(int l, int r, int v) { // [l, r)
  vector<range> ans;
  if(l >= r) return ans;
  auto it = ranges.lower_bound(l);
  if(it != ranges.begin()) {
    it--;
    if(it->r > l) {
      auto cur = *it;
      ranges.erase(it);
      ranges.insert(range(cur.l, l, cur.v));
      ranges.insert(range(l, cur.r, cur.v));
    }
  }
  it = ranges.lower_bound(r);
  if(it != ranges.begin()) {
    it--;
    if(it->r > r) {
      auto cur = *it;
      ranges.erase(it);
      ranges.insert(range(cur.l, r, cur.v));
      ranges.insert(range(r, cur.r, cur.v));
    }
  }
  for(it = ranges.lower_bound(l); it != ranges.end() && it->l < r; it
      ++) {
    ans.push_back(*it);
  }
  ranges.erase(ranges.lower_bound(l), ranges.lower_bound(r));
  ranges.insert(range(l, r, v));
  return ans;
}

int query(int v) { // Substituir -1 por flag para quando nao houver
    resposta
```

```
    auto it = ranges.upper_bound(v);
    if(it == ranges.begin()) {
      return -1;
    }
    it--;
    return it->r >= v ? it->v : -1;
  }
```

2 Dynamic Programming

2.1 Dynamic Hull

```
typedef long double double_t;
typedef long long int ll;

class HullDynamic {
public:
  const double_t inf = 1e9;

  struct Line {
    ll m, b;
    double_t start;
    bool is_query;

    Line() {}

    Line(ll _m, ll _b, double_t _start, bool _is_query) : m(_m), b(_b)
        , start(_start), is_query(_is_query) {}

    ll eval(ll x) {
      return m * x + b;
    }

    double_t intersect(const Line& l) const {
      return (double_t) (l.b - b) / (m - l.m);
    }

    bool operator< (const Line& l) const {
      if (is_query == 0) return m > l.m;
      return (start < l.start);
    }
  };

  typedef set<Line>::iterator iterator_t;

  bool has_prev(iterator_t it) {
    return (it != hull.begin());
  }

  bool has_next(iterator_t it) {
    return (++it != hull.end());
  }

  bool irrelevant(iterator_t it) {
    if (!has_prev(it) || !has_next(it)) return 0;
    iterator_t prev = it, next = it;
    prev--;
    next++;
    return next->intersect(*prev) <= it->intersect(*prev);
  }

  void update_left(iterator_t it) {
    if (it == hull.begin()) return;
    iterator_t pos = it;
    --it;
    vector<Line> rem;
    while(has_prev(it)) {
      iterator_t prev = it;
      --prev;
      if (prev->intersect(*pos) <= prev->intersect(*it)) {
        rem.push_back(*it);
      } else {
        break;
      }
      --it;
    }
    double_t start = pos->intersect(*it);
    Line f = *pos;
    for (Line r : rem) hull.erase(r);
    hull.erase(f);
    f.start = start;
    hull.insert(f);
  }

  void update_right(iterator_t it) {
    if (!has_next(it)) return;
    iterator_t pos = it;
    ++it;
    vector<Line> rem;
    while(has_next(it)) {
      iterator_t next = it;
      ++next;
      if (next->intersect(*pos) <= pos->intersect(*it)) {
        rem.push_back(*it);
      } else {
        break;
      }
      ++it;
    }
    double_t start = pos->intersect(*it);
    Line f = *it;
    for (Line r : rem) hull.erase(r);
    hull.erase(f);
    f.start = start;
    hull.insert(f);
  }

  void insert_line(ll m, ll b) {
    Line f(m, b, -inf, 0);
    iterator_t it = hull.lower_bound(f);
    if (it != hull.end() && it->m == f.m) {
      if (it->b <= f.b) {
        return;
      } else if (it->b > f.b) {
        hull.erase(it);
      }
    }
    hull.insert(f);
    it = hull.lower_bound(f);
```

```
    if (irrelevant(it)) {
      hull.erase(it);
      return;
    }
    update_left(it);
    it = hull.lower_bound(f);
    update_right(it);
  }

  ll get(ll x) {
    Line f(0, 0, x, 1);
    iterator_t it = hull.upper_bound(f);
    assert(it != hull.begin());
    --it;
    return it->m * x + it->b;
  }

private:
  set<Line> hull;
};
```

2.2 Line Container

```
typedef long long int ll;

bool Q;
struct Line {
  mutable ll k, m, p;
  bool operator<(const Line& o) const {
    return Q ? p < o.p : k < o.k;
  }
};

struct LineContainer : multiset<Line> {
  // (for doubles, use inf = 1/.0, div(a,b) = a/b)
  const ll inf = LLONG_MAX;
  ll div(ll a, ll b) { // floored division
    return a / b - ((a ^ b) < 0 && a % b); }
  bool isect(iterator x, iterator y) {
    if (y == end()) { x->p = inf; return false; }
    if (x->k == y->k) x->p = x->m > y->m ? inf : -inf;
    else x->p = div(y->m - x->m, x->k - y->k);
    return x->p >= y->p;
  }
  void add(ll k, ll m) {
    auto z = insert({k, m, 0}), y = z++, x = y;
    while (isect(y, z)) z = erase(z);
    if (x != begin() && isect(--x, y)) isect(x, y = erase(y));
    while ((y = x) != begin() && (--x)->p >= y->p)
      isect(x, erase(y));
  }
  ll query(ll x) {
    assert(!empty());
    Q = 1; auto l = *lower_bound({0,0,x}); Q = 0;
    return l.k * x + l.m;
  }
};
```

2.3 Li Chao Tree

```
// by luucasv
typedef long long T;
const T INF = 1e18, EPS = 1;
const int BUFFER_SIZE = 1e4;

struct Line {
  T m, b;

  Line(T m = 0, T b = INF): m(m), b(b){}
  T apply(T x) { return x * m + b; }
};


struct Node {
  Node *left, *right;
  Line line;
  Node(): left(NULL), right(NULL) {}
};

struct LiChaoTree {
  Node *root, buffer[BUFFER_SIZE];
  T min_value, max_value;
  int buffer_pointer;
  LiChaoTree(T min_value, T max_value): min_value(min_value),
      max_value(max_value + 1) { clear(); }
  void clear() { buffer_pointer = 0; root = newNode(); }
  void insert_line(T m, T b) { update(root, min_value, max_value, Line
      (m, b)); }
  T eval(T x) { return query(root, min_value, max_value, x); }
  void update(Node *cur, T l, T r, Line line) {
    T m = l + (r - l) / 2;
    bool left = line.apply(l) < cur->line.apply(l);
    bool mid = line.apply(m) < cur->line.apply(m);
    bool right = line.apply(r) < cur->line.apply(r);
    if (mid) {
      swap(cur->line, line);
    }
    if (r - l <= EPS) return;
    if (left == right) return;
    if (mid != left) {
      if (cur->left == NULL) cur->left = newNode();
      update(cur->left, l, m, line);
    } else {
      if (cur->right == NULL) cur->right = newNode();
      update(cur->right, m, r, line);
    }
  }
  T query(Node *cur, T l, T r, T x) {
    if (cur == NULL) return INF;
    if (r - l <= EPS) {
      return cur->line.apply(x);
    }
    T m = l + (r - l) / 2;
    T ans;
    if (x < m) {
      ans = query(cur->left, l, m, x);
    } else {
      ans = query(cur->right, m, r, x);
```

```
    }
    return min(ans, cur->line.apply(x));
  }
  Node* newNode() {
      buffer[buffer_pointer] = Node();
      return &buffer[buffer_pointer++];
  }
};
```

2.4 Divide and Conquer Optimization

```
int n, k;
ll dpold[ms], dp[ms], c[ms][ms]; // c(i, j) pode ser funcao

void compute(int l, int r, int optl, int optr) {
    if(l>r) return;
    int mid = (l+r)/2;
    pair<ll, int> best = {inf, -1}; // long long inf
    for(int k = optl; k <= min(mid, optr); k++) {
        best = min(best, {dpold[k-1] + c[k][mid], k});
    }
    dp[mid] = best.first;
    int opt = best.second;
    compute(l, mid-1, optl, opt);
    compute(mid+1, r, opt, optr);
}

ll solve() {
    dp[0] = 0;
    for(int i = 1; i <= n; i++) dp[i] = inf; // initialize row 0 of
        the dp
    for(int i = 1; i <= k; i++) {
        swap(dpold, dp);
        compute(0, n, 0, n); // solve row i of the dp
    }
    return dp[n]; // return dp[k][n]
}
```

2.5 Knuth Optimization

```
int n, m, mid[ms][ms];
ll dp[ms][ms];

void knuth() {
  for(int i = n; i >= 0; i--) { // limites entre 0 e n
    dp[i][i+1] = 0; mid[i][i+1] = i; // caso base
    for(int j = i+2; j <= n; j++) {
      dp[i][j] = inf; // long long inf
      for(int k = mid[i][j-1]; k <= mid[i+1][j]; k++) {
        if(dp[i][j] > dp[i][k] + dp[k][j]) {
          dp[i][j] = dp[i][k] + dp[k][j];
          mid[i][j] = k;
        }
      }
      dp[i][j] += c(i, j); // custo associado ao intervalo
    }
  }
}
```

3 Geometry

3.1 Geometry

```
const double inf = 1e100, eps = 1e-9;
const double PI = acos(-1.0L);

int cmp (double a, double b = 0) {
  if (abs(a-b) < eps) return 0;
  return (a < b) ? -1 : +1;
}

struct PT {
  double x, y;
  PT(double x = 0, double y = 0) : x(x), y(y) {}
  PT operator + (const PT &p) const { return PT(x+p.x, y+p.y); }
  PT operator - (const PT &p) const { return PT(x-p.x, y-p.y); }
  PT operator * (double c) const { return PT(x*c, y*c); }
  PT operator / (double c) const { return PT(x/c, y/c); }

  bool operator <(const PT &p) const {
    if(cmp(x, p.x) != 0) return x < p.x;
    return cmp(y, p.y) < 0;
  }
  bool operator ==(const PT &p) const {
    return !cmp(x, p.x) && !cmp(y, p.y);
  }
  bool operator != (const PT &p) const {
    return !(p == *this);
  }
};
double dot (PT p, PT q) { return p.x * q.x + p.y*q.y; }
double cross (PT p, PT q) { return p.x * q.y - p.y*q.x; }
double dist2 (PT p, PT q = PT(0, 0)) { return dot(p-q, p-q); }
double dist (PT p, PT q) { return hypot(p.x-q.x, p.y-q.y); }
double norm (PT p) { return hypot(p.x, p.y); }
PT normalize (PT p) { return p/hypot(p.x, p.y); }
double angle (PT p, PT q) { return atan2(cross(p, q), dot(p, q)); }
double angle (PT p) { return atan2(p.y, p.x); }
double polarAngle (PT p) {
  double a = atan2(p.y,p.x);
  return a < 0 ? a + 2*PI : a;
}

// - p.y*sen(+90), p.x*sen(+90)
PT rotateCCW90 (PT p) { return PT(-p.y, p.x); }
// - p.y*sen(-90), p.x*sen(-90)
PT rotateCW90 (PT p) { return PT(p.y, -p.x); }

PT rotateCCW (PT p, double t) {
  return PT(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t));
}

// !!! PT (int, int)
typedef pair<PT, int> Line;
PT getDir (PT a, PT b) {
  if (a.x == b.x) return PT(0, 1);
  if (a.y == b.y) return PT(1, 0);
```

```
  int dx = b.x-a.x;
  int dy = b.y-a.y;
  int g = __gcd(abs(dx), abs(dy));
  if (dx < 0) g = -g;
  return PT(dx/g, dy/g);
}

Line getLine (PT a, PT b) {
  PT dir = getDir(a, b);
  return {dir, cross(dir, a)};
}
// Projeta ponto c na linha a - b assumindo a != b
// a.b = |a| cost * |b|
PT projectPointLine (PT a, PT b, PT c) {
  return a + (b-a) * dot(b-a, c-a)/dot(b-a, b-a);
}

PT reflectPointLine (PT a, PT b, PT c) {
  PT p = projectPointLine(a, b, c);
  return p*2 - c;
}

// Projeta ponto c no segmento a - b
PT projectPointSegment (PT a, PT b, PT c) {
  double r = dot(b-a, b-a);
  if (cmp(r) == 0) return a;
  r = dot(b-a, c-a)/r;
  if (cmp(r, 0) < 0) return a;
  if (cmp(r, 1) > 0) return b;
  return a + (b - a) * r;
}

// Calcula distancia entre o ponto c e o segmento a - b
double distancePointSegment (PT a, PT b, PT c) {
  return dist(c, projectPointSegment(a, b, c));
}

// Parallel and opposite directions
// Determina se o ponto c esta em um segmento a - b
bool ptInSegment (PT a, PT b, PT c) {
  if (a == b) return a == c;
  a = a-c, b = b-c;
  return cmp(cross(a, b)) == 0 && cmp(dot(a, b)) <= 0;
}

// Determina se as linhas a - b e c - d sao paralelas ou colineares
bool parallel (PT a, PT b, PT c, PT d) {
  return cmp(cross(b - a, c - d)) == 0;
}

bool collinear (PT a, PT b, PT c, PT d) {
  return parallel(a, b, c, d) && cmp(cross(a - b, a - c)) == 0 && cmp(
      cross(c - d, c - a)) == 0;
}

// Calcula distancia entre o ponto (x, y, z) e o plano ax + by + cz =
    d
double distancePointPlane(double x, double y, double z, double a,
    double b, double c, double d) {
    return abs(a * x + b * y + c * z - d) / sqrt(a * a + b * b + c * c
        );
}

// Determina se o segmento a - b intersecta com o segmento c - d
bool segmentsIntersect (PT a, PT b, PT c, PT d) {
  if (collinear(a, b, c, d)) {
    if (cmp(dist(a, c)) == 0 || cmp(dist(a, d)) == 0 || cmp(dist(b, c)
        ) == 0 || cmp(dist(b, d)) == 0) return true;
    if (cmp(dot(c - a, c - b)) > 0 && cmp(dot(d - a, d - b)) > 0 &&
        cmp(dot(c - b, d - b)) > 0) return false;
    return true;
  }
  if (cmp(cross(d - a, b - a) * cross(c - a, b - a)) > 0) return false
      ;
  if (cmp(cross(a - c, d - c) * cross(b - c, d - c)) > 0) return false
      ;
  return true;
}

// Calcula a intersecao entre as retas a - b e c - d assumindo que uma
     unica intersecao existe
// Para intersecao de segmentos, cheque primeiro se os segmentos se
    intersectam e que nao sao paralelos
// r = a1 + t*d1, (r - a2) x d2 = 0
PT computeLineIntersection (PT a, PT b, PT c, PT d) {
  b = b - a; d = c - d; c = c - a;
  assert(cmp(cross(b, d)) != 0);
  return a + b * cross(c, d) / cross(b, d);
}

// Calcula centro do circulo dado tres pontos
PT computeCircleCenter (PT a, PT b, PT c) {
  b = (a + b) / 2; // bissector
  c = (a + c) / 2; // bissector
  return computeLineIntersection(b, b + rotateCW90(a - b), c, c +
      rotateCW90(a - c));
}

vector<PT> circle2PtsRad (PT p1, PT p2, double r) {
  vector<PT> ret;
  double d2 = dist2(p1, p2);
  double det = r * r / d2 - 0.25;
  if (det < 0.0) return ret;
  double h = sqrt(det);
  for (int i = 0; i < 2; i++) {
    double x = (p1.x + p2.x) * 0.5 + (p1.y - p2.y) * h;
    double y = (p1.y + p2.y) * 0.5 + (p2.x - p1.x) * h;
    ret.push_back(PT(x, y));
    swap(p1, p2);
  }
  return ret;
}

// Calcula intersecao da linha a - b com o circulo centrado em c com
    raio r > 0
bool circleLineIntersection(PT a, PT b, PT c, double r) {
    return cmp(dist(c, projectPointLine(a, b, c)), r) <= 0;
}

vector<PT> circleLine (PT a, PT b, PT c, double r) {
  vector<PT> ret;
  PT p = projectPointLine(a, b, c), p1;
```

```cpp
  double h = norm(c-p);
  if (cmp(h,r) == 0) {
    ret.push_back(p);
  } else if (cmp(h,r) < 0) {
    double k = sqrt(r*r - h*h);
    p1 = p + (b-a)/(norm(b-a))*k;
    ret.push_back(p1);
    p1 = p - (b-a)/(norm(b-a))*k;
    ret.push_back(p1);
  }
  return ret;
}

bool ptInsideTriangle(PT p, PT a, PT b, PT c) {
  if(cross(b-a, c-b) < 0) swap(a, b);
  long long x = cross(b-a, p-b);
  long long y = cross(c-b, p-c);
  long long z = cross(a-c, p-a);
  if(x > 0 && y > 0 && z > 0) return true;
  if(!x) return ptInSegment(a,b,p);
  if(!y) return ptInSegment(b,c,p);
  if(!z) return ptInSegment(c,a,p);
  return false;
}

// Determina se o ponto esta num poligono convexo em O(lgn)
bool pointInConvexPolygon(const vector<PT> &hull, PT point) {
  int n = hull.size();
  if(cmp(cross(point - hull[0], hull[1] - hull[0])) || cmp(cross(point
      - hull[0], hull[n-1] - hull[0]))) return false;
  int l = 1, r = n - 1;
  while(l != r) {
    int mid = (l + r + 1) / 2;
    if(cmp(cross(point - hull[0], hull[mid] - hull[0])) < 0) l = mid;
    else r = mid - 1;
  }
  return cmp(cross(hull[(l+1)%n] - hull[l], point - hull[l])) >= 0;
}

// Determina se o ponto esta num poligono possivelmente nao-convexo
// Retorna 1 para pontos estritamente dentro, 0 para pontos
    estritamente fora do poligno
// e 0 ou 1 para os pontos restantes
// Eh possivel converter num teste exato usando inteiros e tomando
    cuidado com a divisao
// e entao usar testes exatos para checar se esta na borda do poligno
bool pointInPolygon(const vector<PT> &p, PT q) {
  bool c = 0;
  for(int i = 0; i < p.size(); i++){
    int j = (i + 1) % p.size();
    if((p[i].y <= q.y && q.y < p[j].y || p[j].y <= q.y && q.y < p[i].y
        ) &&
      q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) / (p[j].y - p[
          i].y))
      c = !c;
  }
  return c;
}

// Determina se o ponto esta na borda do poligno
bool pointOnPolygon(const vector<PT> &p, PT q) {
  for(int i = 0; i < p.size(); i++)
    if(cmp(dist2(projectPointSegment(p[i], p[(i + 1) % p.size()], q),
        q)) < 0)
      return true;
    return false;
}
// area / semiperimeter
double rIncircle (PT a, PT b, PT c) {
  double ab = norm(a-b), bc = norm(b-c), ca = norm(c-a);
  return abs(cross(b-a, c-a)/(ab+bc+ca));
}

// Calcula intersecao do circulo centrado em a com raio r e o centrado
     em b com raio R
vector<PT> circleCircle (PT a, double r, PT b, double R) {
  vector<PT> ret;
  double d = norm(a-b);
  if (d > r + R || d + min(r, R) < max(r, R)) return ret;
  double x = (d*d - R*R + r*r) / (2*d); // x = r*cos(R opposite angle)
  double y = sqrt(r*r - x*x);
  PT v = (b - a)/d;
  ret.push_back(a + v*x + rotateCCW90(v)*y);
  if (cmp(y) > 0)
    ret.push_back(a + v*x - rotateCCW90(v)*y);
  return ret;
}

double circularSegArea (double r, double R, double d) {
  double ang = 2 * acos((d*d - R*R + r*r) / (2*d*r)); // cos(R
      opposite angle) = x/r
  double tri = sin(ang) * r * r;
  double sector = ang * r * r;
  return (sector - tri) / 2;
}

// Calcula a area ou o centroide de um poligono (possivelmente nao-
    convexo)
// assumindo que as coordenadas estao listada em ordem horaria ou anti
    -horaria
// O centroide eh equivalente a o centro de massa ou centro de
    gravidade
double computeSignedArea (const vector<PT> &p) {
  double area = 0;
  for (int i = 0; i < p.size(); i++) {
    int j = (i+1) % p.size();
    area += p[i].x*p[j].y - p[j].x*p[i].y;
  }
  return area/2.0;
}

double computeArea(const vector<PT> &p) {
  return abs(computeSignedArea(p));
}

PT computeCentroid(const vector<PT> &p) {
  PT c(0,0);
  double scale = 6.0 * ComputeSignedArea(p);
  for(int i = 0; i < p.size(); i++){
    int j = (i + 1) % p.size();
    c = c + (p[i] + p[j]) * (p[i].x * p[j].y - p[j].x * p[i].y);
  }
```

```cpp
  return c / scale;
}

// Testa se o poligno listada em ordem CW ou CCW eh simples (nenhuma
    linha se intersecta)
bool isSimple(const vector<PT> &p) {
  for(int i = 0; i < p.size(); i++) {
    for(int k = i + 1; k < p.size(); k++) {
      int j = (i + 1) % p.size();
      int l = (k + 1) % p.size();
      if (i == l || j == k) continue;
      if (segmentsIntersect(p[i], p[j], p[k], p[l]))
        return false;
    }
  }
  return true;
}

vector< pair<PT, PT> > getTangentSegs (PT c1, double r1, PT c2, double
     r2) {
  if (r1 < r2) swap(c1, c2), swap(r1, r2);
  vector<pair<PT, PT> > ans;
  double d = dist(c1, c2);
  if (cmp(d) <= 0) return ans;
  double dr = abs(r1 - r2), sr = r1 + r2;
  if (cmp(dr, d) >= 0) return ans;
  double u = acos(dr / d);
  PT dc1 = normalize(c2 - c1)*r1;
  PT dc2 = normalize(c2 - c1)*r2;
  ans.push_back(make_pair(c1 + rotateCCW(dc1, +u), c2 + rotateCCW(dc2,
       +u)));
  ans.push_back(make_pair(c1 + rotateCCW(dc1, -u), c2 + rotateCCW(dc2,
       -u)));
  if (cmp(sr, d) >= 0) return ans;
  double v = acos(sr / d);
  dc2 = normalize(c1 - c2)*r2;
  ans.push_back({c1 + rotateCCW(dc1, +v), c2 + rotateCCW(dc2, +v)});
  ans.push_back({c1 + rotateCCW(dc1, -v), c2 + rotateCCW(dc2, -v)});
  return ans;
}
```

3.2 Convex Hull

```cpp
vector<PT> convexHull(vector<PT> p) {
  int n = p.size(), k = 0;
  vector<PT> h(2 * n);
  sort(p.begin(), p.end());
  for(int i = 0; i < n; i++) {
    while(k >= 2 && cmp(cross(h[k - 1] - h[k - 2], p[i] - h[k - 2]))
        <= 0) k--;
    h[k++] = p[i];
  }
  for(int i = n - 2, t = k + 1; i >= 0; i--) {
    while(k >= t && cmp(cross(h[k - 1] - h[k - 2], p[i] - h[k - 2]))
        <= 0) k--;
    h[k++] = p[i];
  }
  h.resize(k); // n+1 points where the first is equal to the last
  return h;
}

void sortByAngle (vector<PT>::iterator first, vector<PT>::iterator
    last, const PT o) {
  first = partition(first, last, [&o] (const PT &a) { return a == o;
      });
  auto pivot = partition(first, last, [&o] (const PT &a) {
    return !(a < o || a == o); // PT(a.y, a.x) < PT(o.y, o.x)
  });
  auto acmp = [&o] (const PT &a, const PT &b) { // C++11 only
    if (cmp(cross(a-o, b-o)) != 0) return cross(a-o, b-o) > 0;
    else return cmp(norm(a-o), norm(b-o)) < 0;
  };
  sort(first, pivot, acmp);
  sort(pivot, last, acmp);
}

vector<PT> graham (vector<PT> v) {
  sort(v.begin(), v.end());
  sortByAngle(v.begin(), v.end(), v[0]);
  vector<PT> u (v.size());
  int top = 0;
  for (int i = 0; i < v.size(); i++) {
    while (top > 1 && cmp(cross(u[top-1] - u[top-2], v[i]-u[top-2]))
        <= 0) top--;
    u[top++] = v[i];
  }
  u.resize(top);
  return u;
}
```

3.3 Cut Polygon

```cpp
struct Segment {
  typedef long double T;
  PT p1, p2;
  T a, b, c;

  Segment() {}

  Segment(PT st, PT en) {
    p1 = st, p2 = en;
    a = -(st.y - en.y);
    b = st.x - en.x;
    c = a * en.x + b * en.y;
  }

  T plug(T x, T y) {
    // plug >= 0 is to the right
    return a * x + b * y - c;
  }

  T plug(PT p) {
    return plug(p.x, p.y);
  }

  bool inLine(PT p) { return cross((p - p1), (p2 - p1)) == 0; }
  bool inSegment(PT p) {
    return inLine(p) && dot((p1 - p2), (p - p2)) >= 0 && dot((p2 - p1)
        , (p - p1)) >= 0;
  }
```

```
  PT lineIntersection(Segment s) {
    long double A = a, B = b, C = c;
    long double D = s.a, E = s.b, F = s.c;
    long double x = (long double) C * E - (long double) B * F;
    long double y = (long double) A * F - (long double) C * D;
    long double tmp = (long double) A * E - (long double) B * D;
    x /= tmp;
    y /= tmp;
    return PT(x, y);
  }

  bool polygonIntersection(const vector<PT> &poly) {
    long double l = -1e18, r = 1e18;
    for(auto p : poly) {
      long double z = plug(p);
      l = max(l, z);
      r = min(r, z);
    }
    return l - r > eps;
  }
};

vector<PT> cutPolygon(vector<PT> poly, Segment seg) {
  int n = (int) poly.size();
  vector<PT> ans;
  for(int i = 0; i < n; i++) {
    double z = seg.plug(poly[i]);
    if(z > -eps) {
      ans.push_back(poly[i]);
    }
    double z2 = seg.plug(poly[(i + 1) % n]);
    if((z > eps && z2 < -eps) || (z < -eps && z2 > eps)) {
      ans.push_back(seg.lineIntersection(Segment(poly[i], poly[(i + 1)
           % n])));
    }
  }
  return ans;
}
```

3.4 Smallest Enclosing Circle

```
typedef pair<PT, double> circle;
bool inCircle (circle c, PT p){
  return cmp(dist(c.first, p), c.second) <= 0;
}

PT circumcenter (PT p, PT q, PT r){
  PT a = p-r, b = q-r;
  PT c = PT(dot(a, p+r)/2, dot(b, q+r)/2);
  return PT(cross(c, PT(a.y,b.y)), cross(PT(a.x,b.x), c)) / cross(a, b
      );
}

mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
circle spanningCircle (vector<PT> &v) {
  int n = v.size();
  shuffle(v.begin(), v.end(), rng);
  circle C(PT(), -1);
  for (int i = 0; i < n; i++) if (!inCircle(C, v[i])) {
    C = circle(v[i], 0);
    for (int j = 0; j < i; j++) if (!inCircle(C, v[j])) {
      C = circle((v[i]+v[j])/2, dist(v[i], v[j])/2);
      for(int k = 0; k < j; k++) if (!inCircle(C, v[k])){
        PT o = circumcenter(v[i], v[j], v[k]);
        C = circle(o,dist(o,v[k]));
      }
    }
  }
  return C;
}
```

3.5 Minkowski

```
bool comp(PT a, PT b){
  int hp1 = (a.x < 0 || (a.x==0 && a.y<0));
  int hp2 = (b.x < 0 || (b.x==0 && b.y<0));
  if(hp1 != hp2) return hp1 < hp2;
  long long R = cross(a, b);
  if(R) return R > 0;
  return dot(a, a) < dot(b, b);
}

vector<PT> minkowskiSum(const vector<PT> &a, const vector<PT> &b){
  if(a.empty() || b.empty()) return vector<PT>(0);
  vector<PT> ret;
  int n1 = a.size(), n2 = b.size();
  if(min(n1, n2) < 2){
    for(int i = 0; i < n1; i++) {
      for(int j = 0; j < n2; j++) {
        ret.push_back(a[i]+b[j]);
      }
    }
    return ret;
  }
  auto insert = [&](PT p) {
    while(ret.size() >= 2 && cmp(cross(p-ret.back(), p-ret[(int)ret.
        size()-2])) == 0) {
      // removing colinear points
      // needs the scalar product stuff it the result is a line
      ret.pop_back();
    }
    ret.push_back(p);
  };
  PT v1, v2, p = a[0]+b[0];
  ret.push_back(p);
  for (int i = 0, j = 0; i + j + 1 < n1+n2; ){
    v1 = a[(i+1)%n1]-a[i];
    v2 = b[(j+1)%n2]-b[j];
    if(j == n2 || (i < n1 && comp(v1, v2))) p = p + v1, i++;
    else p = p + v2, j++;
    insert(p);
  }
  return ret;
}
```

3.6 Half Plane Intersection

```
struct L {
    PT a, b;
    L(){}
    L(PT a, PT b) : a(a), b(b) {}
};

double angle (L la) { return atan2(-(la.a.y - la.b.y), la.b.x - la.a.x
    ); }

bool comp (L la, L lb) {
    if (cmp(angle(la), angle(lb)) == 0) return cross((lb.b - lb.a), (
        la.b - lb.a)) > eps;
    return cmp(angle(la), angle(lb)) < 0;
}

PT computeLineIntersection (L la, L lb) {
    return computeLineIntersection(la.a, la.b, lb.a, lb.b);
}

bool check (L la, L lb, L lc) {
    PT p = computeLineIntersection(lb, lc);
    double det = cross((la.b - la.a),  (p - la.a));
    return cmp(det) < 0;
}

vector<PT> hpi (vector<L> line) { // salvar (i, j) CCW, (j, i) CW
    sort(line.begin(), line.end(), comp);
    vector<L> pl(1, line[0]);
    for (int i = 0; i < (int)line.size(); ++i) if (cmp(angle(line[i]),
         angle(pl.back())) != 0) pl.push_back(line[i]);
    deque<int> dq;
    dq.push_back(0);
    dq.push_back(1);
    for (int i = 2; i < (int)pl.size(); ++i) {
        while ((int)dq.size() > 1 && check(pl[i], pl[dq.back()], pl[dq
            [dq.size() - 2]])) dq.pop_back();
        while ((int)dq.size() > 1 && check(pl[i], pl[dq[0]], pl[dq
            [1]])) dq.pop_front();
        dq.push_back(i);
    }
    while ((int)dq.size() > 1 && check(pl[dq[0]], pl[dq.back()], pl[dq
        [dq.size() - 2]])) dq.pop_back();
    while ((int)dq.size() > 1 && check(pl[dq.back()], pl[dq[0]], pl[dq
        [1]])) dq.pop_front();
    vector<PT> res;
    for (int i = 0; i < (int)dq.size(); ++i){
        res.emplace_back(computeLineIntersection(pl[dq[i]], pl[dq[(i +
             1) % dq.size()]]));
    }
    return res;
}
```

3.7 Closest Pair

```
double closestPair(vector<PT> p) {
  int n = p.size(), k = 0;
  sort(p.begin(), p.end());
  double d = inf;
  set<PT> ptsInv;
  for(int i = 0; i < n; i++) {
    while(k < i && p[k].x < p[i].x - d) {
      ptsInv.erase(swapCoord(p[k++]));
    }
    for(auto it = ptsInv.lower_bound(PT(p[i].y - d, p[i].x - d));
      it != ptsInv.end() && it->x <= p[i].y + d; it++) {
      d = min(d, dist(p[i] - swapCoord(*it), PT(0, 0)));
    }
    ptsInv.insert(swapCoord(p[i]));
  }
  return d;
}
```

3.8 Delaunay Triangulation

```
bool ge(const ll& a, const ll& b) { return a >= b; }
bool le(const ll& a, const ll& b) { return a <= b; }
bool eq(const ll& a, const ll& b) { return a == b; }
bool gt(const ll& a, const ll& b) { return a > b; }
bool lt(const ll& a, const ll& b) { return a < b; }
int sgn(const ll& a) { return a >= 0 ? a ? 1 : 0 : -1; }

struct pt {
    ll x, y;
    pt() { }
    pt(ll _x, ll _y) : x(_x), y(_y) { }
    pt operator-(const pt& p) const {
        return pt(x - p.x, y - p.y);
    }
    ll cross(const pt& p) const {
        return x * p.y - y * p.x;
    }
    ll cross(const pt& a, const pt& b) const {
        return (a - *this).cross(b - *this);
    }
    ll dot(const pt& p) const {
        return x * p.x + y * p.y;
    }
    ll dot(const pt& a, const pt& b) const {
        return (a - *this).dot(b - *this);
    }
    ll sqrLength() const {
        return this->dot(*this);
    }
    bool operator==(const pt& p) const {
        return eq(x, p.x) && eq(y, p.y);
    }
};

const pt inf_pt = pt(1e18, 1e18);

struct QuadEdge {
    pt origin;
    QuadEdge* rot = nullptr;
    QuadEdge* onext = nullptr;
    bool used = false;
    QuadEdge* rev() const {
        return rot->rot;
    }
    QuadEdge* lnext() const {
        return rot->rev()->onext->rot;
```

```
    }
    QuadEdge* oprev() const {
        return rot->onext->rot;
    }
    pt dest() const {
        return rev()->origin;
    }
};

QuadEdge* make_edge(pt from, pt to) {
    QuadEdge* e1 = new QuadEdge;
    QuadEdge* e2 = new QuadEdge;
    QuadEdge* e3 = new QuadEdge;
    QuadEdge* e4 = new QuadEdge;
    e1->origin = from;
    e2->origin = to;
    e3->origin = e4->origin = inf_pt;
    e1->rot = e3;
    e2->rot = e4;
    e3->rot = e2;
    e4->rot = e1;
    e1->onext = e1;
    e2->onext = e2;
    e3->onext = e4;
    e4->onext = e3;
    return e1;
}

void splice(QuadEdge* a, QuadEdge* b) {
    swap(a->onext->rot->onext, b->onext->rot->onext);
    swap(a->onext, b->onext);
}

void delete_edge(QuadEdge* e) {
    splice(e, e->oprev());
    splice(e->rev(), e->rev()->oprev());
    delete e->rot;
    delete e->rev()->rot;
    delete e;
    delete e->rev();
}

QuadEdge* connect(QuadEdge* a, QuadEdge* b) {
    QuadEdge* e = make_edge(a->dest(), b->origin);
    splice(e, a->lnext());
    splice(e->rev(), b);
    return e;
}

bool left_of(pt p, QuadEdge* e) {
    return gt(p.cross(e->origin, e->dest()), 0);
}

bool right_of(pt p, QuadEdge* e) {
    return lt(p.cross(e->origin, e->dest()), 0);
}

template <class T>
T det3(T a1, T a2, T a3, T b1, T b2, T b3, T c1, T c2, T c3) {
    return a1 * (b2 * c3 - c2 * b3) - a2 * (b1 * c3 - c1 * b3) +
           a3 * (b1 * c2 - c1 * b2);
```

```
}

bool in_circle(pt a, pt b, pt c, pt d) {
// If there is __int128, calculate directly.
// Otherwise, calculate angles.
#if defined(__LP64__) || defined(_WIN64)
    __int128 det = -det3<__int128>(b.x, b.y, b.sqrLength(), c.x, c.y,
                                   c.sqrLength(), d.x, d.y, d.
                                       sqrLength());
    det += det3<__int128>(a.x, a.y, a.sqrLength(), c.x, c.y, c.
        sqrLength(), d.x,
                          d.y, d.sqrLength());
    det -= det3<__int128>(a.x, a.y, a.sqrLength(), b.x, b.y, b.
        sqrLength(), d.x,
                          d.y, d.sqrLength());
    det += det3<__int128>(a.x, a.y, a.sqrLength(), b.x, b.y, b.
        sqrLength(), c.x,
                          c.y, c.sqrLength());
    return det > 0;
#else
    auto ang = [](pt l, pt mid, pt r) {
        ll x = mid.dot(l, r);
        ll y = mid.cross(l, r);
        long double res = atan2((long double)x, (long double)y);
        return res;
    };
    long double kek = ang(a, b, c) + ang(c, d, a) - ang(b, c, d) - ang
        (d, a, b);
    if (kek > 1e-8)
        return true;
    else
        return false;
#endif
}

pair<QuadEdge*, QuadEdge*> build_tr(int l, int r, vector<pt>& p) {
    if (r - l + 1 == 2) {
        QuadEdge* res = make_edge(p[l], p[r]);
        return make_pair(res, res->rev());
    }
    if (r - l + 1 == 3) {
        QuadEdge *a = make_edge(p[l], p[l + 1]), *b = make_edge(p[l +
            1], p[r]);
        splice(a->rev(), b);
        int sg = sgn(p[l].cross(p[l + 1], p[r]));
        if (sg == 0)
            return make_pair(a, b->rev());
        QuadEdge* c = connect(b, a);
        if (sg == 1)
            return make_pair(a, b->rev());
        else
            return make_pair(c->rev(), c);
    }
    int mid = (l + r) / 2;
    QuadEdge *ldo, *ldi, *rdo, *rdi;
    tie(ldo, ldi) = build_tr(l, mid, p);
    tie(rdi, rdo) = build_tr(mid + 1, r, p);
    while (true) {
        if (left_of(rdi->origin, ldi)) {
            ldi = ldi->lnext();
            continue;
```

```
        }
        if (right_of(ldi->origin, rdi)) {
            rdi = rdi->rev()->onext;
            continue;
        }
        break;
    }
    QuadEdge* basel = connect(rdi->rev(), ldi);
    auto valid = [&basel](QuadEdge* e) { return right_of(e->dest(),
        basel); };
    if (ldi->origin == ldo->origin)
        ldo = basel->rev();
    if (rdi->origin == rdo->origin)
        rdo = basel;
    while (true) {
        QuadEdge* lcand = basel->rev()->onext;
        if (valid(lcand)) {
            while (in_circle(basel->dest(), basel->origin, lcand->dest
                 (),
                              lcand->onext->dest())) {
                QuadEdge* t = lcand->onext;
                delete_edge(lcand);
                lcand = t;
            }
        }
        QuadEdge* rcand = basel->oprev();
        if (valid(rcand)) {
            while (in_circle(basel->dest(), basel->origin, rcand->dest
                 (),
                              rcand->oprev()->dest())) {
                QuadEdge* t = rcand->oprev();
                delete_edge(rcand);
                rcand = t;
            }
        }
        if (!valid(lcand) && !valid(rcand))
            break;
        if (!valid(lcand) ||
            (valid(rcand) && in_circle(lcand->dest(), lcand->origin,
                                       rcand->origin, rcand->dest())))
            basel = connect(rcand, basel->rev());
        else
            basel = connect(basel->rev(), lcand->rev());
    }
    return make_pair(ldo, rdo);
}

vector<tuple<pt, pt, pt>> delaunay(vector<pt> p) {
    sort(p.begin(), p.end(), [](const pt& a, const pt& b) {
        return lt(a.x, b.x) || (eq(a.x, b.x) && lt(a.y, b.y));
    });
    auto res = build_tr(0, (int)p.size() - 1, p);
    QuadEdge* e = res.first;
    vector<QuadEdge*> edges = {e};
    while (lt(e->onext->dest().cross(e->dest(), e->origin), 0))
        e = e->onext;
    auto add = [&p, &e, &edges]() {
        QuadEdge* curr = e;
        do {
            curr->used = true;
            p.push_back(curr->origin);
            edges.push_back(curr->rev());
            curr = curr->lnext();
        } while (curr != e);
    };
    add();
    p.clear();
    int kek = 0;
    while (kek < (int)edges.size()) {
        if (!(e = edges[kek++])->used)
            add();
    }
    vector<tuple<pt, pt, pt>> ans;
    for (int i = 0; i < (int)p.size(); i += 3) {
        ans.push_back(make_tuple(p[i], p[i + 1], p[i + 2]));
    }
    return ans;
}
```

3.9 Java Geometry Library

```
import java.util.*;
import java.io.*;
import java.awt.geom.*;
import java.lang.*;
//Lazy Geometry
class AWT{
  static Area makeArea(double[] pts){
    Path2D.Double p = new Path2D.Double();
    p.moveTo(pts[0], pts[1]);
    for(int i = 2; i < pts.length; i+=2){
      p.lineTo(pts[i], pts[i+1]);
    }
    p.closePath();
    return new Area(p);
  }
  static double computePolygonArea(ArrayList<Point2D.Double> points) {
    Point2D.Double[] pts = points.toArray(new Point2D.Double[points.
        size()]);
    double area = 0;
    for (int i = 0; i < pts.length; i++){
      int j = (i+1) % pts.length;
      area += pts[i].x * pts[j].y - pts[j].x * pts[i].y;
    }
    return Math.abs(area)/2;
  }
  static double computeArea(Area area) {
    double totArea = 0;
    PathIterator iter = area.getPathIterator(null);
    ArrayList<Point2D.Double> points = new ArrayList<Point2D.Double>()
        ;
    while (!iter.isDone()) {
      double[] buffer = new double[6];
      switch (iter.currentSegment(buffer)) {
        case PathIterator.SEG_MOVETO:
        case PathIterator.SEG_LINETO:
          points.add(new Point2D.Double(buffer[0], buffer[1]));
          break;
        case PathIterator.SEG_CLOSE:
          totArea += computePolygonArea(points);
          points.clear();
```

```
          break;
        }
        iter.next();
      }
      return totArea;
    }
  }
```

4 Graph Algorithms

4.1 Simple Disjoint Set

```
struct dsu {
  vector<int> hist, par, sz;
  vector<ii> changes;
  int n;
  dsu (int n) : n(n) {
    hist.assign(n, 1e9);
    par.resize(n);
    iota(par.begin(), par.end(), 0);
    sz.assign(n, 1);
  }

  int root (int x, int t) {
    if(hist[x] > t) return x;
    return root(par[x], t);
  }

  void join (int a, int b, int t) {
    a = root(a, t);
    b = root(b, t);
    if (a == b) { changes.emplace_back(-1, -1); return; }
    if (sz[a] > sz[b]) swap(a, b);
    par[a] = b;
    sz[b] += sz[a];
    hist[a] = t;
    changes.emplace_back(a, b);
    n--;
  }

  bool same (int a, int b, int t) {
    return root(a, t) == root(b, t);
  }

  void undo () {
    int a, b;
    tie(a, b) = changes.back();
    changes.pop_back();
    if (a == -1) return;
    sz[b] -= sz[a];
    par[a] = a;
    hist[a] = 1e9;
    n++;
  }

  int when (int a, int b) {
    while (1) {
      if (hist[a] > hist[b]) swap(a, b);
      if (par[a] == b) return hist[a];
      if (hist[a] == 1e9) return 1e9;
      a = par[a];
    }
  }
};
```

4.2 Boruvka

```
struct edge {
  int u, v;
  int w;
  int id;
  edge () {};
  edge (int u, int v, int w = 0, int id = 0) : u(u), v(v), w(w), id(id
      ) {};
  bool operator < (edge &other) const { return w < other.w; };
};

vector<edge> boruvka (vector<edge> &edges, int n) {
  vector<edge> mst;
  vector<edge> best(n);
  initDSU(n);
  bool f = 1;
  while (f) {
    f = 0;
    for (int i = 0; i < n; i++) best[i] = edge(i, i, inf);
    for (auto e : edges) {
      int pu = root(e.u), pv = root(e.v);
      if (pu == pv) continue;
      if (e < best[pu]) best[pu] = e;
      if (e < best[pv]) best[pv] = e;
    }
    for (int i = 0; i < n; i++) {
      edge e = best[root(i)];
      if (e.w != inf) {
        join(e.u, e.v);
        mst.push_back(e);
        f = 1;
      }
    }
  }
  return mst;
}
```

4.3 Dinic Max Flow

```
const int ms = 1e3; // Quantidade maxima de vertices
const int me = 1e5; // Quantidade maxima de arestas

int adj[ms], to[me], ant[me], wt[me], z, n;
int copy_adj[ms], fila[ms], level[ms];

void clear() { // Lembrar de chamar no main
  memset(adj, -1, sizeof adj);
  z = 0;
}
```

```
void add(int u, int v, int k) {
  to[z] = v;
  ant[z] = adj[u];
  wt[z] = k;
  adj[u] = z++;
  swap(u, v);
  to[z] = v;
  ant[z] = adj[u];
  wt[z] = 0; // Lembrar de colocar = 0
  adj[u] = z++;
}

int bfs(int source, int sink) {
  memset(level, -1, sizeof level);
  level[source] = 0;
  int front = 0, size = 0, v;
  fila[size++] = source;
  while(front < size) {
    v = fila[front++];
    for(int i = adj[v]; i != -1; i = ant[i]) {
      if(wt[i] && level[to[i]] == -1) {
        level[to[i]] = level[v] + 1;
        fila[size++] = to[i];
      }
    }
  }
  return level[sink] != -1;
}

int dfs(int v, int sink, int flow) {
  if(v == sink) return flow;
  int f;
  for(int &i = copy_adj[v]; i != -1; i = ant[i]) {
    if(wt[i] && level[to[i]] == level[v] + 1 &&
      (f = dfs(to[i], sink, min(flow, wt[i])))) {
      wt[i] -= f;
      wt[i ^ 1] += f;
      return f;
    }
  }
  return 0;
}

int maxflow(int source, int sink) {
  int ret = 0, flow;
  while(bfs(source, sink)) {
    memcpy(copy_adj, adj, sizeof adj);
    while((flow = dfs(source, sink, 1 << 30))) {
      ret += flow;
    }
  }
  return ret;
}
```

4.4 Minimum Vertex Cover

```
// + Dinic
vector<int> coverU, U, coverV, V; // ITA - Parti o U LEFT,
    parti o V RIGHT, 0 indexed
bool Zu[mx], Zv[mx];
int pairU[mx], pairV[mx];
void getreach(int u) {
  if (u == -1 || Zu[u]) return;
  Zu[u] = true;
  for (int i = adj[u]; ~i; i = ant[i]) {
    int v = to[i];
    if (v == SOURCE || v == pairU[u]) continue;
    Zv[v] = true;
    getreach(pairV[v]);
  }
}

void minimumcover () {
  memset(pairU, -1, sizeof pairU);
  memset(pairV, -1, sizeof pairV);
  for (auto i : U) {
    for (int j = adj[i]; ~j; j = ant[j]) {
      if (!(j&1) && !wt[j]) {
        pairU[i] = to[j], pairV[to[j]] = i;
      }
    }
  }
  memset(Zu, 0, sizeof Zu);
  memset(Zv, 0, sizeof Zv);
  for (auto u : U) {
    if (pairU[u] == -1) getreach(u);
  }
  coverU.clear(), coverV.clear();
  for (auto u : U) {
    if (!Zu[u]) coverU.push_back(u);
  }
  for (auto v : V) {
    if (Zv[v]) coverV.push_back(v);
  }
}
```

4.5 Min Cost Max Flow

```
template <class flow_t, class cost_t>
class MinCostMaxFlow {
private:
  typedef pair<cost_t, int> ii;

  struct Edge {
    int to;
    flow_t cap;
    cost_t cost;
    Edge(int to, flow_t cap, cost_t cost) : to(to), cap(cap), cost(
        cost) {}
  };

  int n;
  vector<vector<int>> adj;
  vector<Edge> edges;
  vector<cost_t> dis;
  vector<int> prev, id_prev;
        vector<int> q;
        vector<bool> inq;

  pair<flow_t, cost_t> spfa(int src, int sink) {
```

```
      fill(dis.begin(), dis.end(), int(1e9)); //cost_t inf
      fill(prev.begin(), prev.end(), -1);
      fill(inq.begin(), inq.end(), false);
      q.clear();
      q.push_back(src);
      inq[src] = true;
      dis[src] = 0;
      for(int on = 0; on < (int) q.size(); on++) {
          int cur = q[on];
          inq[cur] = false;
          for(auto id : adj[cur]) {
                  if (edges[id].cap == 0) continue;
                  int to = edges[id].to;
                  if (dis[to] > dis[cur] + edges[id].cost) {
                          prev[to] = cur;
                          id_prev[to] = id;
                          dis[to] = dis[cur] + edges[id].cost;
                          if (!inq[to]) {
                                  q.push_back(to);
                                  inq[to] = true;
                          }
                  }
          }
      }
      flow_t mn = flow_t(1e9);
      for(int cur = sink; prev[cur] != -1; cur = prev[cur]) {
        int id = id_prev[cur];
        mn = min(mn, edges[id].cap);
      }
      if (mn == flow_t(1e9) || mn == 0) return make_pair(0, 0);
      pair<flow_t, cost_t> ans(mn, 0);
      for(int cur = sink; prev[cur] != -1; cur = prev[cur]) {
        int id = id_prev[cur];
        edges[id].cap -= mn;
        edges[id ^ 1].cap += mn;
        ans.second += mn * edges[id].cost;
      }
      return ans;
    }
  public:
    MinCostMaxFlow(int a = 0) {
      n = a;
      adj.resize(n + 2);
      edges.clear();
      dis.resize(n + 2);
      prev.resize(n + 2);
      id_prev.resize(n + 2);
      inq.resize(n + 2);
    }
    void init(int a) {
      n = a;
      adj.resize(n + 2);
      edges.clear();
      dis.resize(n + 2);
      prev.resize(n + 2);
      id_prev.resize(n + 2);
      inq.resize(n + 2);
    }
    void add(int from, int to, flow_t cap, cost_t cost) {
      adj[from].push_back(int(edges.size()));
                  edges.push_back(Edge(to, cap, cost));
                  adj[to].push_back(int(edges.size()));
                  edges.push_back(Edge(from, 0, -cost));
    }
    pair<flow_t, cost_t> maxflow(int src, int sink) {
      pair<flow_t, cost_t> ans(0, 0), got;
      while((got = spfa(src, sink)).first > 0) {
        ans.first += got.first;
        ans.second += got.second;
      }
      return ans;
    }
  };
```

4.6 Euler Path and Circuit

```
int pathV[me], szV, del[me], pathE, szE;
int adj[ms], to[me], ant[me], wt[me], z, n;

// Funcao de add e clear no dinic

void eulerPath(int u) {
  for(int i = adj[u]; ~i; i = ant[u]) if(!del[i]) {
    del[i] = del[i^1] = 1;
    eulerPath(to[i]);
    pathE[szE++] = i;
  }
  pathV[szV++] = u;
}
```

4.7 Articulation Points/Bridges/Biconnected Components

```
int adj[ms], to[me], ant[me], z;
int num[ms], low[ms], timer;
int art[ms], bridge[me], rch;
int bc[ms], nbc;
stack<int> st;
bool f[me];

void clear() { // Lembrar de chamar no main
  memset(adj, -1, sizeof adj);
  z = 0;
}


void add(int u, int v) {
  to[z] = v;
  ant[z] = adj[u];
  adj[u] = z++;
}

void generateBc (int v) {
  while (!st.empty()) {
    int u = st.top();
    st.pop();
    bc[u] = nbc;
    if (v == u) break;
  }
  ++nbc;
```

```
}

void dfs (int v, int p) {
  st.push(v);
  low[v] = num[v] = ++timer;
  for (int i = adj[v]; i != -1; i = ant[i]) {
    if (f[i] || f[i^1]) continue;
    f[i] = 1;
    int u = to[i];
    if (num[u] == -1) {
      dfs(u, v);
      if (low[u] > num[v]) bridge[i] = bridge[i^1] = 1;
      art[v] |= p != -1 && low[u] >= num[v];
      if (p == -1 && rch > 1) art[v] = 1;
      else rch ++;
      low[v] = min(low[v], low[u]);
    } else {
      low[v] = min(low[v], num[u]);
    }
  }
  if (low[v] == num[v]) generateBc(v);
}

void biCon (int n) {
  nbc = 0, timer = 0;
  memset(num, -1, sizeof num);
  memset(bc, -1, sizeof bc);
  memset(bridge, 0, sizeof bridge);
  memset(art, 0, sizeof art);
  memset(f, 0, sizeof f);
  for (int i = 0; i < n; i++) {
    if (num[i] == -1) {
      rch = 0;
      dfs(i, 0);
    }
  }
}
```

4.8 SCC - Strongly Connected Components / 2SAT

```
vector<int> g[ms];
int idx[ms], low[ms], z, comp[ms], ncomp;
stack<int> st;

int dfs(int u) {
  if(~idx[u]) return idx[u] ? idx[u] : z;
  low[u] = idx[u] = z++;
  st.push(u);
  for(int v : g[u]) {
    low[u] = min(low[u], dfs(v));
  }
  if(low[u] == idx[u]) {
    while(st.top() != u) {
      int v = st.top();
      idx[v] = 0;
      low[v] = low[u];
      comp[v] = ncomp;
      st.pop();
    }
    idx[st.top()] = 0;
    st.pop();
    comp[u] = ncomp++;
  }
  return low[u];
}

bool solveSat() {
  memset(idx, -1, sizeof idx);
  z = 1; ncomp = 0;
  for(int i = 0; i < n; i++) dfs(i);
  for(int i = 0; i < n; i++) if(comp[i] == comp[i^1]) return false;
  return true;
}

// Operacoes comuns de 2-sat
// ~v = "nao v"
#define trad(v) (v<0?((~v)*2)^1:v*2)
void addImp(int a, int b) { g[trad(a)].push(trad(b)); }
void addOr(int a, int b) { addImp(~a, b); addImp(~b, a); }
void addEqual(int a, int b) { addOr(a, ~b); addOr(~a, b); }
void addDiff(int a, int b) { addEqual(a, ~b); }
// valoracao: value[v] = comp[trad(v)] < comp[trad(~v)]
```

4.9 LCA - Lowest Common Ancestor

```
int par[ms][mlg+1], lvl[ms];
vector<int> g[ms];

void dfs(int v, int p, int l = 0) { // chamar como dfs(root, root)
  lvl[v] = l;
  par[v][0] = p;
  for(int k = 1; k <= mlg; k++) {
    par[v][k] = par[par[v][k-1]][k-1];
  }
  for(int u : g[v]) {
    if(u != p) dfs(u, v, l + 1);
  }
}

int lca(int a, int b) {
  if(lvl[b] > lvl[a]) swap(a, b);
  for(int i = mlg; i >= 0; i--) {
    if(lvl[a] - (1 << i) >= lvl[b]) a = par[a][i];
  }
  if(a == b) return a;
  for(int i = mlg; i >= 0; i--) {
    if(par[a][i] != par[b][i]) a = par[a][i], b = par[b][i];
  }
  return par[a][0];
}
```

4.10 Heavy Light Decomposition

```
// src: tfg
class HLD {
public:
  void init(int n) {
    // this doesn't delete edges!
```

```
    sz.resize(n);
    in.resize(n);
    out.resize(n);
    rin.resize(n);
    p.resize(n);
    edges.resize(n);
    nxt.resize(n);
    h.resize(n);
  }

  void addEdge(int u, int v) {
    edges[u].push_back(v);
    edges[v].push_back(u);
  }

  void setRoot(int n) {
    t = 0;
    p[n] = n;
    h[n] = 0;
    prep(n, n);
    nxt[n] = n;
    hld(n);
  }

  int getLCA(int u, int v) {
    while(!inSubtree(nxt[u], v)) {
      u = p[nxt[u]];
    }
    while(!inSubtree(nxt[v], u)) {
      v = p[nxt[v]];
    }
    return in[u] < in[v] ? u : v;
  }

  bool inSubtree(int u, int v) {
    // is v in the subtree of u?
    return in[u] <= in[v] && in[v] < out[u];
  }

  vector<pair<int, int>> getPathtoAncestor(int u, int anc) {
    // returns ranges [l, r) that the path has
    vector<pair<int, int>> ans;
    //assert(inSubtree(anc, u));
    while(nxt[u] != nxt[anc]) {
      ans.emplace_back(in[nxt[u]], in[u] + 1);
      u = p[nxt[u]];
    }
    // this includes the ancestor!
    ans.emplace_back(in[anc], in[u] + 1);
    return ans;
  }
private:
  vector<int> in, out, p, rin, sz, nxt, h;
  vector<vector<int>> edges;
  int t;

  void prep(int on, int par) {
    sz[on] = 1;
    p[on] = par;
    for(int i = 0; i < (int) edges[on].size(); i++) {
      int &u = edges[on][i];
      if(u == par) {
        swap(u, edges[on].back());
        edges[on].pop_back();
        i--;
      } else {
        h[u] = 1 + h[on];
        prep(u, on);
        sz[on] += sz[u];
        if(sz[u] > sz[edges[on][0]]) {
          swap(edges[on][0], u);
        }
      }
    }
  }

  void hld(int on) {
    in[on] = t++;
    rin[in[on]] = on;
    for(auto u : edges[on]) {
      nxt[u] = (u == edges[on][0] ? nxt[on] : u);
      hld(u);
    }
    out[on] = t;
  }
};
```

4.11 Centroid Decomposition

```
//Centroid decomposition1

void dfsSize(int v, int pa) {
  sz[v] = 1;
  for(int u : adj[v]) {
    if (u == pa || rem[u]) continue;
    dfsSize(u, v);
    sz[v] += sz[u];
  }
}

int getCentroid(int v, int pa, int tam) {
  for(int u : adj[v]) {
    if (u == pa || rem[u]) continue;
    if (2 * sz[u] > tam) return getCentroid(u, v, tam);
  }
  return v;
}

void decompose(int v, int pa = -1) {
  //cout << v << ' ' << pa << '\n';
  dfsSize(v, pa);
  int c = getCentroid(v, pa, sz[v]);
  //cout << c << '\n';
  par[c] = pa;
  rem[c] = 1;
  for(int u : adj[c]) {
    if (!rem[u] && u != pa) decompose(u, c);
  }
  adj[c].clear();
}
```

```
//Centroid decomposition2

void dfsSize(int v, int par) {
  sz[v] = 1;
  for(int u : adj[v]) {
    if (u == par || removed[u]) continue;
    dfsSize(u, v);
    sz[v] += sz[u];
  }
}

int getCentroid(int v, int par, int tam) {
  for(int u : adj[v]) {
    if (u == par || removed[u]) continue;
    if (2 * sz[u] > tam) return getCentroid(u, v, tam);
  }
  return v;
}

void setDis(int v, int par, int nv, int d) {
  dis[v][nv] = d;
  for(int u : adj[v]) {
    if (u == par || removed[u]) continue;
    setDis(u, v, nv, d + 1);
  }
}

void decompose(int v, int par, int nv) {
  dfsSize(v, par);
  int c = getCentroid(v, par, sz[v]);
  ct[c] = par;
  removed[c] = 1;
  setDis(c, par, nv, 0);
  for(int u : adj[c]) {
    if (!removed[u]) {
      decompose(u, c, nv + 1);
    }
  }
}
```

4.12 Sack

```
void solve(int a, int p, bool f){
  int big = -1;
  for(auto &b : adj[a]){
    if(b != p && (big == -1 || en[b]-st[b] > en[big]-st[big])){
      big = b;
    }
  }
  for(auto &b : adj[a]){
    if(b == p || b == big) continue;
    solve(b, a, 0);
  }
  if(~big) solve(big, a, 1);
  add(cnt[v[a]], -1);
  cnt[v[a]]++;
  add(cnt[v[a]], +1);
  for(auto &b : adj[a]){
    if(b == p || b == big) continue;
    for(int i = st[b]; i < en[b]; i++){
      add(cnt[ett[i]], -1);
      cnt[ett[i]]++;
      add(cnt[ett[i]], +1);
    }
  }
  for(auto &q : Q[a]){
    ans[q.first] = query(mx-1)-query(q.second-1);
  }
  if(!f){
    for(int i = st[a]; i < en[a]; i++){
      add(cnt[ett[i]], -1);
      cnt[ett[i]]--;
      add(cnt[ett[i]], +1);
    }
  }
}
```

4.13 Hungarian Algorithm - Maximum Cost Matching

```
const int inf = 0x3f3f3f3f;

int n, w[ms][ms], maxm;
int lx[ms], ly[ms], xy[ms], yx[ms];
int slack[ms], slackx[ms], prev[ms];
bool S[ms], T[ms];

void init_labels() {
  memset(lx, 0, sizeof lx); memset(ly, 0, sizeof ly);
  for(int x = 0; x < n; x++) for(int y = 0; y < n; y++) {
    lx[x] = max(lx[x], cos[x][y]);
  }
}

void updateLabels() {
  int delta = inf;
  for(int y = 0; y < n; y++) if(!T[y]) delta = min(delta, slack[y]);
  for(int x = 0; x < n; x++) if(S[x]) lx[x] -= delta;
  for(int y = 0; y < n; y++) if(T[y]) ly[y] += delta;
  for(int y = 0; y < n; y++) if(!T[y]) slack[y] -= delta;
}

void addTree(int x, int prevx) {
  S[x] = 1; prev[x] = prevx;
  for(int y = 0; y < n; y++) if(lx[x] + ly[y] - w[x][y] < slack[y]) {
    slack[y] = lx[x] + ly[y] - cost[x][y];
    slackx[y] = x;
  }
}

void augment() {
  if(maxm == n) return;
  int x, y, root;
  int q[ms], wr = 0, rd = 0;
  memset(S, 0, sizeof S); memset(T, 0, sizeof T);
  memset(prev, -1, sizeof prev);
  for(int x = 0; x < n; x++) if(xy[x] == -1) {
    q[wr++] = root = x;
    prev[x] = -2;
    S[x] = 1;
    break;
```

```
    }
    for(int y = 0; y < n; y++) {
      slack[y] = lx[root] + ly[y] - w[root][y];
      slackx[y] = root;
    }
    while(true) {
      while(rd < wr) {
        x = q[rd++];
        for(y = 0; y < n; y++) if(w[x][y] == lx[x] + ly[y] && !T[y]) {
          if(yx[y] == -1) break;
          T[y] = 1;
          q[wr++] = yx[y];
          addTree(yx[y], x);
        }
        if(y < n) break;
      }
      if(y < n) break;
      updateLabels();
      wr = rd = 0;
      for(y = 0; y < n; y++) if(!T[y] && !slack[y]) {
        if(yx[y] == -1) {
          x = slackx[y];
          break;
        } else {
          T[y] = true;
          if(!S[yx[y]]) {
            q[wr++] = yx[y];
            addTree(yx[y], slackx[y]);
          }
        }
      }
      if(y < n) break;
    }
    if(y < n) {
      maxm++;
      for(int cx = x, cy = y,ty; cx != -2; cx = prev[cx], cy = ty) {
        ty = xy[cx];
        yx[cy] = cx;
        xy[cx] = cy;
      }
      augment();
    }
  }

  int hungarian() {
    int ans = 0; maxm = 0;
    memset(xy, -1, sizeof xy); memset(yx, -1, sizeof yx);
    initLabels(); augment();
    for(int x = 0; x < n; x++) ans += w[x][xy[x]];
    return ans;
  }
```

5 Math

5.1 Chinese Remainder Theorem

```
//by leon
#include<bits/stdc++.h>
using namespace std;
const long long N = 20;

long long GCD(long long a, long long b) {
  return (b == 0) ? a : GCD(b, a % b);
}
inline long long get_LCM(long long a, long long b) {
  return a / GCD(a, b) * b;
}
inline long long normalize(long long x, long long mod) {
  x %= mod;
  if (x < 0) x += mod;
  return x;
}

struct GCD_type {
  long long x, y, d;
};
GCD_type ex_GCD(long long a, long long b){
  if (b == 0) return {1, 0, a};
  GCD_type pom = ex_GCD(b, a % b);
  return {pom.y, pom.x - a / b * pom.y, pom.d};
}

long long testCases;
long long t;
long long a[N], n[N], ans, LCM;

int main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  t = 2;
  long long T;
  cin >> T;
  while(T--) {
    for(long long i = 1; i <= t; i++) {
      cin >> a[i] >> n[i];
      normalize(a[i], n[i]);
    }
    ans = a[1];
    LCM = n[1];
    bool impossible = false;
    for(long long i = 2; i <= t; i++) {
      auto pom = ex_GCD(LCM, n[i]);
      long long x1 = pom.x;
      long long d = pom.d;
      if((a[i] - ans) % d != 0) {
        impossible = true;
      }
      ans = normalize(ans + x1 * (a[i] - ans) / d % (n[i] / d) * LCM,
          LCM * n[i] / d);
      LCM = get_LCM(LCM, n[i]);
    }
    if (impossible) cout << "no solution\n";
    else cout << ans << " " << LCM << endl;
  }
  return 0;
}
```

5.2 Diophantine Equations

```
int gcd_ext(int a, int b, int& x, int &y) {
  if (b == 0) {
    x = 1, y = 0;
    return a;
  }
  int nx, ny;
  int gc = gcd_ext(b, a % b, nx, ny);
  x = ny;
  y = nx - (a / b) * ny;
  return gc;
}

vector<int> diophantine(int D, vector<int> l) {
  int n = l.size();
  vector<int> gc(n), ans(n);
  gc[n - 1] = l[n - 1];
  for (int i = n - 2; i >= 0; i--) {
    int x, y;
    gc[i] = gcd_ext(l[i], gc[i + 1], x, y);
  }
  if (D % gc[0] != 0) {
    return vector<int>();
  }
  for (int i = 0; i < n; i++) {
    if (i == n - 1) {
      ans[i] = D / l[i];
      D -= l[i] * ans[i];
      continue;
    }
    int x, y;
    gcd_ext(l[i] / gc[i], gc[i + 1] / gc[i], x, y);
    ans[i] = (long long int) D / gc[i] * x % (gc[i + 1] / gc[i]);
    if (D < 0 && ans[i] > 0) {
      ans[i] -= (gc[i + 1] / gc[i]);
    }
    if (D > 0 && ans[i] < 0) {
      ans[i] += (gc[i + 1] / gc[i]);
    }
    D -= l[i] * ans[i];
  }
  return ans;
}
```

5.3 Discrete Logarithm

```
ll discreteLog(ll a, ll b, ll m) {
  // a^ans == b mod m
  // ou -1 se nao existir
  ll cur = a, on = 1;
  for(int i = 0; i < 100; i++) {
    cur = cur * a % m;
  }
  while(on * on <= m) {
    cur = cur * a % m;
    on++;
  }
  map<ll, ll> position;
  for(ll i = 0, x = 1; i * i <= m; i++) {
    position[x] = i * on;
    x = x * cur % m;
  }
  for(ll i = 0; i <= on + 20; i++) {
    if(position.count(b)) {
      return position[b] - i;
    }
    b = b * a % m;
  }
  return -1;
}
```

5.4 Discrete Root

```
//x^k = a % mod
ll discreteRoot(ll k, ll a, ll mod) {
  ll g = primitiveRoot(mod);
  ll y = discreteLog(fexp(g, k, mod), a, mod);
  if (y == -1) {
    return y;
  }
  return fexp(g, y, mod);
}
```

5.5 Primitive Root

```
int primitiveRoot(int p) {
  vector<int> fact;
  int phi = p - 1, n = phi;
  for (int i = 2; i * i <= n; i++) {
    if (n % i == 0) {
      fact.push_back(i);
      while (n % i == 0) {
        n /= i;
      }
    }
  }
  if (n > 1) {
    fact.push_back(n);
  }
  for (int res = 2; res <= p; res++) {
    bool ok = true;
    for (auto it : fact) {
      ok &= fexp(res, phi / it, p) != 1;
      if (!ok) {
        break;
      }
    }
    if (ok) {
      return res;
    }
  }
  return -1;
}
```

5.6 Extended Euclides

```
// euclides estendido: acha u e v da equacao:
// u * x + v * y = gcd(x, y);
// u eh inverso modular de x no modulo y
// v eh inverso modular de y no modulo x

pair<ll, ll> euclides(ll a, ll b) {
  ll u = 0, oldu = 1, v = 1, oldv = 0;
  while(b) {
    ll q = a / b;
    oldv = oldv - v * q;
    oldu = oldu - u * q;
    a = a - b * q;
    swap(a, b);
    swap(u, oldu);
    swap(v, oldv);
  }
  return make_pair(oldu, oldv);
}
```

5.7 Matrix Fast Exponentiation

```
const ll mod = 1e9+7;
const int m = 2; // size of matrix

struct Matrix {
  ll mat[m][m];
  Matrix operator * (const Matrix &p) {
    Matrix ans;
    for(int i = 0; i < m; i++)
      for(int j = 0; j < m; j++)
        for(int k = ans.mat[i][j] = 0; k < m; k++)
          ans.mat[i][j] = (ans.mat[i][j] + mat[i][k] * p.mat[k][j]) %
              mod;
    return ans;
  }
};

Matrix fExp(Matrix a, ll b) {
  Matrix ans;
  for(int i = 0; i < m; i++) for(int j = 0; j < m; j++)
    ans.mat[i][j] = i == j;
  while(b) {
    if(b & 1) ans = ans * a;
    a = a * a;
    b >>= 1;
  }
  return ans;
}
```

5.8 FFT - Fast Fourier Transform

```
typedef double ld;

const ld PI = acos(-1);

struct Complex {
	ld real, imag;
	Complex conj() { return Complex(real, -imag); }
	Complex(ld a = 0, ld b = 0) : real(a), imag(b) {}
	Complex operator + (const Complex &o) const { return Complex(
	    real + o.real, imag + o.imag); }
	Complex operator - (const Complex &o) const { return Complex(
	    real - o.real, imag - o.imag); }
	Complex operator * (const Complex &o) const { return Complex(
	    real * o.real - imag * o.imag, real * o.imag + imag * o.
	    real); }
	Complex operator / (ld o) const { return Complex(real / o,
	    imag / o); }
	void operator *= (Complex o) { *this = *this * o; }
	void operator /= (ld o) { real /= o, imag /= o; }
};

typedef std::vector<Complex> CVector;

const int ms = 1 << 22;

int bits[ms];
Complex root[ms];

void initFFT() {
	root[1] = Complex(1);
	for(int len = 2; len < ms; len += len) {
		Complex z(cos(PI / len), sin(PI / len));
		for(int i = len / 2; i < len; i++) {
			root[2 * i] = root[i];
			root[2 * i + 1] = root[i] * z;
		}
	}
}

void pre(int n) {
	int LOG = 0;
	while(1 << (LOG + 1) < n) {
		LOG++;
	}
	for(int i = 1; i < n; i++) {
		bits[i] = (bits[i >> 1] >> 1) | ((i & 1) << LOG);
	}
}


CVector fft(CVector a, bool inv = false) {
	int n = a.size();
	pre(n);
	if(inv) {
		std::reverse(a.begin() + 1, a.end());
	}
	for(int i = 0; i < n; i++) {
		int to = bits[i];
		if(to > i) {
			std::swap(a[to], a[i]);
		}
	}
	for(int len = 1; len < n; len *= 2) {
		for(int i = 0; i < n; i += 2 * len) {
```

```
			for(int j = 0; j < len; j++) {
				Complex u = a[i + j], v = a[i + j +
				    len] * root[len + j];
				a[i + j] = u + v;
				a[i + j + len] = u - v;
			}
		}
	}
	if(inv) {
		for(int i = 0; i < n; i++)
			a[i] /= n;
	}
	return a;
}

void fft2in1(CVector &a, CVector &b) {
	int n = (int) a.size();
	for(int i = 0; i < n; i++) {
		a[i] = Complex(a[i].real, b[i].real);
	}
	auto c = fft(a);
	for(int i = 0; i < n; i++) {
		a[i] = (c[i] + c[(n-i) % n].conj()) * Complex(0.5, 0);
		b[i] = (c[i] - c[(n-i) % n].conj()) * Complex(0, -0.5)
		    ;
	}
}

void ifft2in1(CVector &a, CVector &b) {
	int n = (int) a.size();
	for(int i = 0; i < n; i++) {
		a[i] = a[i] + b[i] * Complex(0, 1);
	}
	a = fft(a, true);
	for(int i = 0; i < n; i++) {
		b[i] = Complex(a[i].imag, 0);
		a[i] = Complex(a[i].real, 0);
	}
}


std::vector<long long> mod_mul(const std::vector<long long> &a, const
    std::vector<long long> &b, long long cut = 1 << 15) {
	// TODO cut memory here by /2
	int n = (int) a.size();
	CVector C[4];
	for(int i = 0; i < 4; i++) {
		C[i].resize(n);
	}
	for(int i = 0; i < n; i++) {
		C[0][i] = a[i] % cut;
		C[1][i] = a[i] / cut;
		C[2][i] = b[i] % cut;
		C[3][i] = b[i] / cut;
	}
	fft2in1(C[0], C[1]);
	fft2in1(C[2], C[3]);
	for(int i = 0; i < n; i++) {
		// 00, 01, 10, 11
		Complex cur[4];
		for(int j = 0; j < 4; j++) cur[j] = C[j/2+2][i] * C[j
		    % 2][i];
		for(int j = 0; j < 4; j++) C[j][i] = cur[j];
	}
	ifft2in1(C[0], C[1]);
	ifft2in1(C[2], C[3]);
	std::vector<long long> ans(n, 0);
	for(int i = 0; i < n; i++) {
		// if there are negative values, care with rounding
		ans[i] += (long long) (C[0][i].real + 0.5);
		ans[i] += (long long) (C[1][i].real + C[2][i].real +
		    0.5) * cut;
		ans[i] += (long long) (C[3][i].real + 0.5) * cut * cut
		    ;
	}
	return ans;
}

std::vector<int> mul(const std::vector<int> &a, const std::vector<int>
    &b) {
	int n = 1;
	while (n - 1 < (int) a.size() + (int) b.size() - 2) n += n;
	CVector poly(n);
	for(int i = 0; i < n; i++) {
		if(i < (int) a.size()) {
			poly[i].real = a[i];
		}
		if(i < (int) b.size()) {
			poly[i].imag = b[i];
		}
	}
	poly = fft(poly);
	for(int i = 0; i < n; i++) {
		poly[i] *= poly[i];
	}
	poly = fft(poly, true);
	std::vector<int> c(n, 0);
	for(int i = 0; i < n; i++) {
		c[i] = (int) (poly[i].imag / 2 + 0.5);
	}
	while (c.size() > 0 && c.back() == 0) c.pop_back();
	return c;
}
```

5.9 NTT - Number Theoretic Transform

```
long long int mod = (119ll << 23) + 1, c_root = 3;

namespace NTT {
  typedef long long int ll;

  ll fexp(ll base, ll e) {
    ll ans = 1;
    while(e > 0) {
      if (e & 1) ans = ans * base % mod;
      base = base * base % mod;
      e >>= 1;
    }
    return ans;
  }
```

```
  ll inv_mod(ll base) {
    return fexp(base, mod - 2);
  }

  void ntt(vector<ll>& a, bool inv) {
    int n = (int) a.size();
    if (n == 1) return;

    for(int i = 0, j = 0; i < n; i++) {
      if (i > j) {
        swap(a[i], a[j]);
      }
      for(int l = n / 2; (j ^= l) < l; l >>= 1);
    }

    for(int sz = 1; sz < n; sz <<= 1) {
      ll delta = fexp(c_root, (mod - 1) / (2 * sz)); //delta = w_2sz
      if (inv) {
        delta = inv_mod(delta);
      }
      for(int i = 0; i < n; i += 2 * sz) {
        ll w = 1;
        for(int j = 0; j < sz; j++) {
          ll u = a[i + j], v = w * a[i + j + sz] % mod;
          a[i + j] = (u + v + mod) % mod;
          a[i + j] = (a[i + j] + mod) % mod;
          a[i + j + sz] = (u - v + mod) % mod;
          a[i + j + sz] = (a[i + j + sz] + mod) % mod;
          w = w * delta % mod;
        }
      }
    }
    if (inv) {
      ll inv_n = inv_mod(n);
      for(int i = 0; i < n; i++) {
        a[i] = a[i] * inv_n % mod;
      }
    }
    for(int i = 0; i < n; i++) {
      a[i] %= mod;
      a[i] = (a[i] + mod) % mod;
    }
  }

  void multiply(vector<ll> &a, vector<ll> &b, vector<ll> &ans) {
    int lim = (int) max(a.size(), b.size());
    int n = 1;
    while(n < lim) n <<= 1;
    n <<= 1;
    a.resize(n);
    b.resize(n);
    ans.resize(n);
    ntt(a, false);
    ntt(b, false);
    for(int i = 0; i < n; i++) {
      ans[i] = a[i] * b[i] % mod;
    }
    ntt(ans, true);
  }
};
```

5.10 Miller and Rho

```
typedef long long int ll;

bool overflow(ll a, ll b) {
  return b && (a >= (1ll << 62) / b);
}

ll add(ll a, ll b, ll md) {
  return (a + b) % md;
}

ll mul(ll a, ll b, ll md) {
  if (!overflow(a, b)) return (a * b) % md;
  ll ans = 0;
  while(b) {
    if (b & 1) ans = add(ans, a, md);
    a = add(a, a, md);
    b >>= 1;
  }
  return ans;
}

ll fexp(ll a, ll e, ll md) {
  ll ans = 1;
  while(e) {
    if (e & 1) ans = mul(ans, a, md);
    a = mul(a, a, md);
    e >>= 1;
  }
  return ans;
}

ll my_rand() {
  ll ans = rand();
  ans = (ans << 31) | rand();
  return ans;
}

ll gcd(ll a, ll b) {
  while(b) {
    ll t = a % b;
    a = b;
    b = t;
  }
  return a;
}

bool miller(ll p, int iteracao) {
  if(p < 2) return 0;
  if(p % 2 == 0) return (p == 2);
  ll s = p - 1;
  while(s % 2 == 0) s >>= 1;
  for(int i = 0; i < iteracao; i++) {
    ll a = rand() % (p - 1) + 1, temp = s;
    ll mod = fexp(a, temp, p);
    while(temp != p - 1 && mod != 1 && mod != p - 1) {
      mod = mul(mod, mod, p);
      temp <<= 1;
    }
```

```
    if(mod != p - 1 && temp % 2 == 0) return 0;
  }
  return 1;
}

ll rho(ll n) {
  if (n == 1 || miller(n, 10)) return n;
  if (n % 2 == 0) return 2;
  while(1) {
    ll x = my_rand() % (n - 2) + 2, y = x;
    ll c = 0, cur = 1;
    while(c == 0) {
      c = my_rand() % (n - 2) + 1;
    }
    while(cur == 1) {
      x = add(mul(x, x, n), c, n);
      y = add(mul(y, y, n), c, n);
      y = add(mul(y, y, n), c, n);
      cur = gcd((x >= y ? x - y : y - x), n);
    }
    if (cur != n) return cur;
  }
}
```

5.11 Determinant using Mod

```
// by zchao1995
// Determinante com coordenadas inteiras usando Mod

ll mat[ms][ms];

ll det (int n) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      mat[i][j] %= mod;
    }
  }
  ll res = 1;
  for (int i = 0; i < n; i++) {
    if (!mat[i][i])     {
      bool flag = false;
      for (int j = i + 1; j < n; j++) {
        if (mat[j][i]) {
          flag = true;
          for (int k = i; k < n; k++) {
            swap (mat[i][k], mat[j][k]);
          }
          res = -res;
          break;
        }
      }
      if (!flag) {
        return 0;
      }
    }
    for (int j = i + 1; j < n; j++) {
      while (mat[j][i]) {
        ll t = mat[i][i] / mat[j][i];
        for (int k = i; k < n; k++) {
          mat[i][k] = (mat[i][k] - t * mat[j][k]) % mod;
          swap (mat[i][k], mat[j][k]);
        }
        res = -res;
      }
    }
    res = (res * mat[i][i]) % mod;
  }
  return (res + mod) % mod;
}
```

5.12 Lagrange Interpolation

```
class LagrangePoly {
public:
        LagrangePoly(std::vector<long long> _a) {
                //f(i) = _a[i]
                //interpola o vetor em um polinomio de grau y.size() -
                     1
                y = _a;
                den.resize(y.size());
                int n = (int) y.size();
                for(int i = 0; i < n; i++) {
                        y[i] = (y[i] % MOD + MOD) % MOD;
                        den[i] = ifat[n - i - 1] * ifat[i] % MOD;
                        if((n - i - 1) % 2 == 1) {
                                den[i] = (MOD - den[i]) % MOD;
                        }
                }
        }

        long long getVal(long long x) {
                int n = (int) y.size();
                x %= MOD;
                if(x < n) {
                        //return y[(int) x];
                }
                std::vector<long long> l, r;
                l.resize(n);
                l[0] = 1;
                for(int i = 1; i < n; i++) {
                        l[i] = l[i - 1] * (x - (i - 1) + MOD) % MOD;
                }
                r.resize(n);
                r[n - 1] = 1;
                for(int i = n - 2; i >= 0; i--) {
                        r[i] = r[i + 1] * (x - (i + 1) + MOD) % MOD;
                }
                long long ans = 0;
                for(int i = 0; i < n; i++) {
                        long long coef = l[i] * r[i] % MOD;
                        ans = (ans + coef * y[i] % MOD * den[i]) % MOD
                             ;
                }
                return ans;
        }

private:
        std::vector<long long> y, den;
};
```

```cpp
int main(){
        fat[0] = ifat[0] = 1;
        for(int i = 1; i < ms; i++) {
                fat[i] = fat[i - 1] * i % MOD;
                ifat[i] = fexp(fat[i], MOD - 2);
        }
        // Codeforces 622F
        int x, k;
        std::cin >> x >> k;
        std::vector<long long> a;
        a.push_back(0);
        for(long long i = 1; i <= k + 1; i++) {
                a.push_back((a.back() + fexp(i, k)) % MOD);
        }
        LagrangePoly f(a);
        std::cout << f.getVal(x) << '\n';
}
```

6 Miscellaneous

6.1 LIS - Longest Increasing Subsequence

```cpp
int arr[ms], lisArr[ms], n;
// int bef[ms], pos[ms];

int lis() {
  int len = 1;
  lisArr[0] = arr[0];
  // bef[0] = -1;
  for(int i = 1; i < n; i++) {
    // upper_bound se non-decreasing
    int x = lower_bound(lisArr, lisArr + len, arr[i]) - lisArr;
    len = max(len, x + 1);
    lisArr[x] = arr[i];
    // pos[x] = i;
    // bef[i] = x ? pos[x-1] : -1;
  }
  return len;
}

vi getLis() {
  int len = lis();
  vi ans;
  for(int i = pos[lisArr[len - 1]]; i >= 0; i = bef[i]) {
    ans.push_back(arr[i]);
  }
  reverse(ans.begin(), ans.end());
  return ans;
}
```

6.2 Ternary Search

```cpp
// R
for(int i = 0; i < LOG; i++){
  long double m1 = (A * 2 + B) / 3.0;
  long double m2 = (A + 2 * B) / 3.0;
  if(f(m1) > f(m2))
    A = m1;
  else
    B = m2;
}
ans = f(A);

// Z
while(B - A > 4){
  int m1 = (A + B) / 2;
  int m2 = (A + B) / 2 + 1;
  if(f(m1) > f(m2))
    A = m1;
  else
    B = m2;
}
ans = inf;
for(int i = A; i <= B; i++) ans = min(ans , f(i));
```

6.3 Count Sort

```cpp
int H[(1<<15)+1], to[mx], b[mx];
void sort(int m, int a[]) {
  memset(H, 0, sizeof H);
  for (int i = 1; i <= m; i++) {
    H[a[i] % (1<<15)]++;
  }
  for (int i = 1; i < 1<<15; i++) {
    H[i] += H[i-1];
  }
  for (int i = m; i; i--) {
    to[i] = H[a[i] % (1 << 15)]--;
  }
  for (int i = 1; i <= m; i++) {
    b[to[i]] = a[i];
  }
  memset(H, 0, sizeof H);
  for (int i = 1; i <= m; i++) {
    H[b[i]>>15]++;
  }
  for (int i = 1; i < 1<<15; i++) {
    H[i] += H[i-1];
  }
  for (int i = m; i ; i--) {
    to[i] = H[b[i]>>15]--;
  }
  for (int i = 1; i <= m; i++) {
    a[to[i]] = b[i];
  }
}
```

6.4 Random Number Generator

```cpp
// mt19937_64 se LL
mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
// Random_Shuffle
shuffle(v.begin(), v.end(), rng);
// Random number in interval
```

```
int randomInt = uniform_int_distribution(0, i)(rng);
double randomDouble = uniform_real_distribution(0, 1)(rng);
// bernoulli_distribution, binomial_distribution,
    geometric_distribution
// normal_distribution, poisson_distribution, exponential_distribution
```

6.5 Rectangle Hash

```
namespace {
  struct safe_hash {
    static uint64_t splitmix64(uint64_t x) {
      // http://xorshift.di.unimi.it/splitmix64.c
      x += 0x9e3779b97f4a7c15;
      x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
      x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
      return x ^ (x >> 31);
    }

    size_t operator()(uint64_t x) const {
      static const uint64_t FIXED_RANDOM = std::chrono::steady_clock::
          now().time_since_epoch().count();
      return splitmix64(x + FIXED_RANDOM);
    }
  };
}

struct rect {
  int x1, y1, x2, y2; // x1 < x2, y1 < y2
  rect () {};
  rect (int x1, int y1, int x2, int y2) : x1(x1), x2(x2), y1(y1), y2(
      y2) {};

  rect inter (rect other) {
    int x3 = max(x1, other.x1);
    int y3 = max(y1, other.y1);
    int x4 = min(x2, other.x2);
    int y4 = min(y2, other.y2);
    return rect(x3, y3, x4, y4);
  }

  uint64_t get_hash() {
    safe_hash sh;
    uint64_t ret = sh(x1);
    ret ^= sh(ret ^ y1);
    ret ^= sh(ret ^ x2);
    ret ^= sh(ret ^ y2);
    return ret;
  }
};
```

6.6 Unordered Map Tricks

```
// pair<int, int> hash function
struct HASH{
  size_t operator()(const pair<int,int>&x)const{
    return (size_t) x.first * 37U + (size_t) x.second;
  }
};

unordered_map<int,int>mp;
mp.reserve(1024);
mp.max_load_factor(0.25);
```

6.7 Submask Enumeration

```
for (int s=m; ; s=(s-1)&m) {
  ... you can use s ...
  if (s==0)  break;
}
```

6.8 Sum over Subsets DP

```
// F[i] = Sum of all A[j] where j is a submask of i
for(int i = 0; i<(1<<N); ++i)
        F[i] = A[i];
for(int i = 0;i < N; ++i) for(int mask = 0; mask < (1<<N); ++mask){
        if(mask & (1<<i))
                F[mask] += F[mask^(1<<i)];
}
```

6.9 Java Fast I/O

```
import java.io.OutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.PrintWriter;
import java.util.Arrays;
import java.util.Random;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.StringTokenizer;
import java.io.BufferedReader;
import java.io.InputStream;
import java.util.*;
import java.io.*;
// src petr
public class Main {
  public static void main(String[] args) {

    InputStream inputStream = System.in;
    OutputStream outputStream = System.out;
    InputReader in = new InputReader(inputStream);
    PrintWriter out = new PrintWriter(outputStream);
    TaskA solver = new TaskA();
    solver.solve(1, in, out);
    out.close();
  }

  static class TaskA {
    public void solve(int testNumber, InputReader in, PrintWriter out)
        {

    }
  }
```

```
  static class InputReader {
    public BufferedReader reader;
    public StringTokenizer tokenizer;
    public InputReader(InputStream stream) {
      reader = new BufferedReader(new InputStreamReader(stream),
          32768);
      tokenizer = null;
    }
    public String next() {
      while (tokenizer == null || !tokenizer.hasMoreTokens()) {
        try {
          tokenizer = new StringTokenizer(reader.readLine());
        } catch (IOException e) {
          throw new RuntimeException(e);
        }
      }
      return tokenizer.nextToken();
    }
    public int nextInt() {
      return Integer.parseInt(next());
    }
  }
}
```

7 String Algorithms

7.1 KMP

```
string p, t;
int b[ms], n, m;

void kmpPreprocess() {
  int i = 0, j = -1;
  b[0] = -1;
  while(i < m) {
    while(j >= 0 && p[i] != p[j]) j = b[j];
    b[++i] = ++j;
  }
}

void kmpSearch() {
  int i = 0, j = 0, ans = 0;
  while(i < n) {
    while(j >= 0 && t[i] != p[j]) j = b[j];
    i++; j++;
    if(j == m) {
      //ocorrencia aqui comecando em i - j
      ans++;
      j = b[j];
    }
  }
  return ans;
}
```

7.2 KMP Automaton

```
const int limit =

vector<vector<int>> build_automaton(string s) {
    s += '#'; //tem que ser diferente de todos os caracteres
    int n = (int) s.size();
    vector<vector<int>> ans(n, vector<int>(limit));
    vector<int> fail(n);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < limit; j++) {
            if (i == 0) {
                if (s[i] == j + 'a') {
                    ans[i][j] = i + 1;
                } else {
                    ans[i][j] = 0;
                }
            } else {
                if (s[i] == j + 'a') {
                    ans[i][j] = i + 1;
                } else {
                    ans[i][j] = ans[fail[i - 1]][j];
                }
            }
        }
        if (i == 0) {
            continue;
        }
        int j = fail[i - 1];
        while (j > 0 && s[i] != s[j]) {
            j = fail[j - 1];
        }
        fail[i] = j + (s[i] == s[j]);
    }
    return ans;
}
```

7.3 Trie

```
int trie[ms][sigma], terminal[ms], z;

void init() {
  memset(trie[0], -1, sizeof trie[0]);
  z = 1;
}

int get_id(char c) {
  return c - 'a';
}

void insert(string &p) {
  int cur = 0;
  for(int i = 0; i < p.size(); i++) {
    int id = get_id(p[i]);
    if(trie[cur][id] == -1) {
      memset(trie[z], -1, sizeof trie[z]);
      trie[cur][id] = z++;
    }
    cur = trie[cur][id];
  }
  terminal[cur]++;
}
```

```
int count(string &p) {
  int cur = 0;
  for(int i = 0; i < p.size(); i++) {
    int id = get_id(p[i]);
    if(trie[cur][id] == -1) {
      return false;
    }
    cur = trie[cur][id];
  }
  return terminal[cur];
}
```

7.4 Aho-Corasick

```
// Construa a Trie do seu dicionario com o codigo acima

int fail[ms];
queue<int> q;

void buildFailure() {
  q.push(0);
  while(!q.empty()) {
    int node = q.front();
    q.pop();
    for(int pos = 0; pos < sigma; pos++) {
      int &v = trie[node][pos];
      int f = node == 0 ? 0 : trie[fail[node]][pos];
      if(v == -1) {
        v = f;
      } else {
        fail[v] = f;
        q.push(v);
        // juntar as informacoes da borda para o V ja q um match em V
            implica um match na borda
        terminal[v] += terminal[f];
      }
    }
  }
}

int search(string &txt) {
  int node = 0;
  int ans = 0;
  for(int i = 0; i < txt.length(); i++) {
    int pos = get_id(txt[i]);
    node = trie[node][pos];
    // processar informacoes no no atual
     ans += terminal[node];
  }
  return ans;
}
```

7.5 Algoritmo de Z

```
string s;
int fz[ms], n;

void zfunc() {
  fz[0] = n;
  for(int i = 1, l = 0, r = 0; i < n; i++) {
    fz[i] = max(0, min(r-i, fz[i-l]));
    while(s[fz[i]] == s[i+fz[i]]) ++fz[i];
    if(i + fz[i] > r) {
      l = i;
      r = i + fz[i];
    }
  }
}
```

7.6 Suffix Array

```
namespace SA {
  typedef pair<int, int> ii;

  vector<int> buildSA(string s) {
    int n = (int) s.size();
    vector<int> ids(n), pos(n);
    vector<ii> pairs(n);
    for(int i = 0; i < n; i++) {
      ids[i] = i;
      pairs[i] = ii(s[i], -1);
    }
    sort(ids.begin(), ids.end(), [&](int a, int b) -> bool {
      return pairs[a] < pairs[b];
    });
    int on = 0;
    for(int i = 0; i < n; i++) {
      if (i && pairs[ids[i - 1]] != pairs[ids[i]]) on++;
      pos[ids[i]] = on;
    }
    for(int offset = 1; offset < n; offset <<= 1) {
      //ja tao ordenados pelos primeiros offset caracteres
      for(int i = 0; i < n; i++) {
        pairs[i].first = pos[i];
        if (i + offset < n) {
          pairs[i].second = pos[i + offset];
        } else {
          pairs[i].second = -1;
        }
      }
      sort(ids.begin(), ids.end(), [&](int a, int b) -> bool {
        return pairs[a] < pairs[b];
      });
      int on = 0;
      for(int i = 0; i < n; i++) {
        if (i && pairs[ids[i - 1]] != pairs[ids[i]]) on++;
        pos[ids[i]] = on;
      }
    }
    return ids;
  }

  vector<int> buildLCP(string s, vector<int> sa) {
    int n = (int) s.size();
    vector<int> pos(n), lcp(n, 0);
    for(int i = 0; i < n; i++) {
      pos[sa[i]] = i;
```

```
    }
    int k = 0;
    for(int i = 0; i < n; i++) {
      if (pos[i] + 1 == n) {
        k = 0;
        continue;
      }
      int j = sa[pos[i] + 1];
      while(i + k < n && j + k < n && s[i + k] == s[j + k]) k++;
      lcp[pos[i]] = k;
      k = max(k - 1, 0);
    }
    return lcp;
  }
};

//nlogn

vector<int> suffix_array(const string& in) {
    int n = (int)in.size(), c = 0;
    vector<int> temp(n), pos2bckt(n), bckt(n), bpos(n), out(n);
    for (int i = 0; i < n; i++) out[i] = i;
    sort(out.begin(), out.end(), [&](int a, int b) { return in[a] < in
        [b]; });
    for (int i = 0; i < n; i++) {
        bckt[i] = c;
        if (i + 1 == n || in[out[i]] != in[out[i + 1]]) c++;
    }
    /*Start*/
    for (int h = 1; h < n && c < n; h <<= 1) {// executes log n times
        for (int i = 0; i < n; i++) pos2bckt[out[i]] = bckt[i];
        for (int i = n - 1; i >= 0; i--) bpos[bckt[i]] = i;
        for (int i = 0; i < n; i++)
            if (out[i] >= n - h) temp[bpos[bckt[i]]++] = out[i];
        for (int i = 0; i < n; i++)
            if (out[i] >= h) temp[bpos[pos2bckt[out[i] - h]]++] = out[
                i] - h;
        c = 0;
        for (int i = 0; i + 1 < n; i++) {
            int a = (bckt[i] != bckt[i + 1]) || (temp[i] >= n - h)
                || (pos2bckt[temp[i + 1] + h] != pos2bckt[temp[i] + h
                    ]);
            bckt[i] = c;
            c += a;
        }
        bckt[n - 1] = c++;
        temp.swap(out);
    }
    return out;
}
```

8 Teoremas e formulas uteis

8.1 Grafos

```
Formula de Euler: V - E + F = 2 (para grafo planar)
Handshaking: Numero par de vertices tem grau impar
Kirchhoff's Theorem: Monta matriz onde Mi,i = Grau[i] e Mi,j = -1 se
    houver aresta i-j ou 0 caso contrario, remove uma linha e uma
    coluna qualquer e o numero de spanning trees nesse grafo eh o det
    da matriz

Grafo contem caminho hamiltoniano se:
Dirac's theorem: Se o grau de cada vertice for pelo menos n/2
Ore's theorem: Se a soma dos graus que cada par nao-adjacente de
    vertices for pelo menos n

Trees:
Tem Catalan(N) Binary trees de N vertices
Tem Catalan(N-1) Arvores enraizadas com N vertices
Caley Formula: n^(n-2) arvores em N vertices com label
Prufer code: Cada etapa voce remove a folha com menor label e o label
    do vizinho eh adicionado ao codigo ate ter 2 vertices

Flow:
Max Edge-disjoint paths: Max flow com arestas com peso 1
Max Node-disjoint paths: Faz a mesma coisa mas separa cada vertice em
    um com as arestas de chegadas e um com as arestas de saida e uma
    aresta de peso 1 conectando o vertice com aresta de chegada com
    ele mesmo com arestas de saida
Konig's Theorem: minimum node cover = maximum matching se o grafo for
    bipartido, complemento eh o maximum independent set
Min Node disjoint path cover: formar grafo bipartido de vertices
    duplicados, onde aresta sai do vertice tipo A e chega em tipo B,
    entao o path cover eh N - matching
Min General path cover: Mesma coisa mas colocando arestas de A pra B
    sempre que houver caminho de A pra B
Dilworth's Theorem: Min General Path cover = Max Antichain (set de
    vertices tal que nao existe caminho no grafo entre vertices desse
    set)
Hall's marriage: um grafo tem um matching completo do lado X se para
    cada subconjunto W de X,
    |W| <= |vizinhosW| onde |W| eh quantos vertices tem em W
```

8.2 Math

```
Goldbach's: todo numero par n > 2 pode ser representado com n = a + b
    onde a e b sao primos
Twin prime: existem infinitos pares p, p + 2 onde ambos sao primos
Legendre's: sempre tem um primo entre n^2 e (n+1)^2
Lagrange's: todo numero inteiro pode ser inscrito como a soma de 4
    quadrados
Zeckendorf's: todo numero pode ser representado pela soma de dois
    numeros de fibonnacis diferentes e nao consecutivos
Euclid's: toda tripla de pitagoras primitiva pode ser gerada com
    (n^2 - m^2, 2nm, n^2+m^2) onde n, m sao coprimos e um deles eh par
Wilson's: n eh primo quando (n-1)! mod n = n - 1
Mcnugget: Para dois coprimos x, y o maior inteiro que nao pode ser
    escrito como ax + by eh (x-1)(y-1)/2

Fermat: Se p eh primo entao a^(p-1) % p = 1
Se x e m tambem forem coprimos entao x^k % m = x^(k mod(m-1)) % m
Euler's theorem: x^(phi(m)) mod m = 1 onde phi(m) eh o totiente de
    euler

Chinese remainder theorem:
Para equacoes no formato x = a1 mod m1, ... , x = an mod mn onde todos
     os pares m1, ... , mn sao coprimos
Deixe Xk = m1*m2*..*mn/mk e Xk^-1 mod mk = inverso de Xk mod mk, entao
```

```
x = somatorio com k de 1 ate n de ak*Xk*(Xk,mk^-1 mod mk)
Para achar outra solucao so somar m1*m2*..*mn a solucao existente

Catalan number: exemplo expressoes de parenteses bem formadas
C0 = 1, Cn = somatorio de i=0 -> n-1 de Ci*C(n-1+1)
outra forma: Cn = (2n escolhe n)/(n+1)
Bertrand's ballot theorem: p votos tipo A e q votos tipo B com p>q,
    prob de em todo ponto ter mais As do que Bs antes dele = (p-q)/(p+
    q)
Se puder empates entao prob = (p+1-q)/(p+1), para achar quantidade de
    possibilidades nos dois casos basta multiplicar por (p + q escolhe
     q)

Propriedades de Coeficientes Binomiais:
Somatorio de k = 0 -> m de (-1)^k * (n escolhe k) = (-1)^m* (n -1
    escolhe m)
(N escolhe K) = (N escolhe N-K)
(N escolhe K) = N/K * (n-1 escolhe k-1)
Somatorio de k = 0 -> n de (n escolhe k) = 2^n
Somatorio de m = 0 -> n de (m escolhe k) = (n+1 escolhe k + 1)
Somatorio de k = 0 -> m de (n+k escolhe k) = (n+m+1 escolhe m)
Somatorio de k = 0 -> n de (n escolhe k)^2 = (2n escolhe n)
Somatorio de k = 0 ou 1 -> n de k*(n escolhe k) = n * 2^(n-1)
Somatorio de k = 0 -> n de (n-k escolhe k) = Fib(n+1)

Hockey-stick: Somatorio de i = r -> n de (i escolhe r) = (n + 1
    escolhe r + 1)
Vandermonde: (m+n escolhe r) = somatorio de k = 0 -> r de (m escolhe k
    ) * (n escolhe r - k)

Burnside lemma: colares diferentes nao contando rotacoes quando m =
    cores e n = comprimento
(m^n + somatorio i =1 - > n-1 de m^gcd(i, n))/n

Distribuicao uniforme a,a+1, ..., b Expected[X] = (a+b)/2
Distribuicao binomial com n tentativas de probabilidade p, X =
    sucessos:
    P(X = x) = p^x * (1-p)^(n-x) * (n escolhe x) e E[X] = p*n
Distribuicao geometrica onde continuamos ate ter sucesso, X =
    tentativas:
    P(X = x) = (1-p)^(x-1) * p e E[X] = 1/p
```

```
Linearity of expectation: Tendo duas variaveis X e Y e constantes a e
    b, o valor esperado de aX + bY = a*E[X] + b*E[X]
```

8.3 Geometry

```
Formula de Euler: V - E + F = 2
Pick Theorem: Para achar pontos em coords inteiras num poligono Area =
     i + b/2 - 1 onde i eh o o numero de pontos dentro do poligono e b
     de pontos no perimetro do poligono
Two ears theorem: Todo poligono simples com mais de 3 vertices tem
    pelo menos 2 orelhas, vertices que podem ser removidos sem criar
    um crossing, remover orelhas repetidamente triangula o poligono
Incentro triangulo:  (a(Xa, Ya) + b(Xb, Yb) + c(Xc, Yc))/(a+b+c) onde
    a = lado oposto ao vertice a, incentro eh onde cruzam as
    bissetrizes, eh o centro da circunferencia inscrita e eh
    equidistante aos lados

Delaunay Triangulation: Triangulacao onde nenhum ponto esta dentro de
    nenhum circulo circunscrito nos triangulos
Eh uma triangulacao que maximiza o menor angulo e a MST euclidiana de
    um conjunto de pontos eh um subconjunto da triangulacao

Brahmagupta s formula: Area cyclic quadrilateral
s = (a+b+c+d)/2
area = sqrt((s-a)*(s-b)*(s-c)*(s-d))
d = 0 => area = sqrt((s-a)*(s-b)*(s-c)*s)
```

8.4 Mersenne's Primes

```
Primos de Mersenne 2^n - 1
Lista de Ns que resultam nos primeiros 41 primos de Mersenne:
2; 3; 5; 7; 13; 17; 19; 31; 61; 89; 107; 127; 521; 607; 1.279; 2.203;
    2.281; 3.217; 4.253; 4.423; 9.689; 9.941; 11.213; 19.937; 21.701;
    23.209; 44.497; 86.243; 110.503; 132.049; 216.091; 756.839;
    859.433; 1.257.787; 1.398.269; 2.976.221; 3.021.377; 6.972.593;
    13.466.917; 20.996.011; 24.036.583;
```