Ohio State University ICPC Team Notebook

Contents

1 Essentials

1.1 C++ header

```
# include <bits/stdc++.h>
using namespace std;
# define rep(i, a, b) for(int i = a; i < (b); ++i)
# define trav(a, x) for(auto& a : x)
typedef long long ll;
typedef pair<int, int> pii;
typedef vector<int> vi;
```

1.2 C++ flags

```
# Add this to the CMakeLists in CLion to crash with bad memory accesses and give
     better warnings.
# Don't include this comment, comments don't work in CMakeLists.
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Wextra -Wno-sign-compare -D
    _GLIBCXX_DEBUG -D _GLIBCXX_DEBUG_PEDANTIC ")
```

1.3 C++ input/output

```
#include <iostream>
#include <iomanip>
#include <bitset>
using namespace std;

int main() {

    // Output a specific number of digits past the decimal point,
    // in this case 5
    cout.setf(ios::fixed);
    cout << setprecision(5);
    cout << 100.0 / 7.0 << " " << 10.0 << endl; // 14.28571 10.00000
    cout.unsetf(ios::fixed);

    // Output a '+' before positive values
    cout.setf(ios::showpos);
    cout << 100 << " " << -100 << endl; // +100 -100
    cout.unsetf(ios::showpos);

    // Output numerical values in hexadecimal. Also works for oct
    cout << hex << 500 << dec << endl; // 1f4 (1*256 + 15*16 + 4*1)
    // Output numerical values in binary
    std::bitset<10> bs(500);
    cout << bs << endl; // 0111110100

    // Read until end of file.
    string line;
    getline(cin, line);
    while (!line.empty()) { // Input in CP problems always ends with an empty
         line.
        int intV; string stringV;
        stringstream line_stream(line);
        line_stream >> stringV >> intV; // Just read like usual from the stream
        getline(cin, line);
    }
}
```

2 Data structures

2.1 Unordered Set/Map

```
// An example of policy hashtable with a custom object in cpp. It is
// it is better than the built in unordered_map in that
// it is ~5 times faster. (https://codeforces.com/blog/entry/60737)
// No real downsides (normal map is just as annoying with custom objects),
// but be careful with the hash function, the number of buckets is a power of 2.
#include <bits/stdc++.h>
using namespace std;

struct Coordinate {
    int x;
    int y;
    bool operator==(const Coordinate &other) const {
        return x == other.x && y == other.y;
    }
};

ostream &operator<<(ostream &stream, const Coordinate &l) {
    return stream << "{" << l.x << " " << l.y << "}";
}
#include <ext/pb_ds/assoc_container.hpp>

struct chash {
    static auto const c = uint64_t(7e18) + 13; // Big prime
    uint64_t operator()(const Coordinate &l) const {
        return __builtin_bswap64((l.x + l.y) * c);
    }
};

template<class k, class v>
using hash_map = __gnu_pbds::gp_hash_table<k, v, chash>;
template<class k>
using hash_set = __gnu_pbds::gp_hash_table<k, __gnu_pbds::null_type, chash>;
template<typename k, typename v>
bool contains(hash_map<k, v> map, k val) {
    return map.find(val) != map.end();
}

int main() {
    // After importing, writing the template code, overloading ==
    // and << (print) operator like above, you can use the map
    hash_map<Coordinate, int> my_map;
    my_map[{1, 2}] = 17;
    cout << my_map[{1, 2}] << endl; // Prints 17
    assert(contains(my_map, {1, 2}));
    assert(!contains(my_map, {3, 4}));
    cout << my_map[{3, 4}] << endl; // Prints 0
    assert(my_map.size() == 2); // We just set {3, 4} to 0 by accessing it.
    for (auto pair : my_map) {
        cout << pair.first << "=" << pair.second << " "; // {3 4}=0 {1 2}=17
    }

    hash_set<Coordinate> my_set;
    assert(my_set.empty());
    my_set.insert({1, 2});
    assert(contains(my_set, {1, 2}));
    my_set.insert({4, 5});
    // hash_set does the correct thing, and when you iterate over it you get
       keys,
    // not key-value pairs with a null value.
    for (auto it = my_set.begin(); it != my_set.end(); it++) {
        cout << *it << " ";  // print {4, 5} {1, 2}.
    }
    // Standard C Library Equivalent Declarations:
    // unordered_map<Coordinate, int, chash> my_map;
    // unordered_set<Coordinate, chash> my_set;
}
```

2.2 Ordered Set/Map

```
// An example of using an ordered map with a custom object.
// Also include code for the gnu policy tree, which gives
// a easy (~2x slower) segment tree by implementing
// find_by_order and order_of_key
#include <bits/stdc++.h>

using namespace std;

struct Coordinate {
    int x;
    int y;
    // Overloaded for ordered map. If !(c1<c2), !(c2<c1), then
    // c1 will be considered equal to c2.
    bool operator<(const Coordinate &o) const {
        return x == o.x ? y < o.y : x < o.x;
    }
};

ostream &operator<<(ostream &stream, const Coordinate &l) {
    return stream << "{" << l.x << " " << l.y << "}";
}

#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace __gnu_pbds;
template<class k, class v>
using ordered_map = tree<k, v, less<k>,
        rb_tree_tag, // Red black tree. Can use splay_tree_tag for a splay tree,
        // but split operation for splay is linear time so it may be terrible.
        tree_order_statistics_node_update // To get find_by_order and
             order_of_key methods
>;
template<class k> // Same as ordered map almost
using ordered_set = tree<k, null_type, less<k>,
        rb_tree_tag, tree_order_statistics_node_update>;

int main() {
    map<Coordinate, int> c_map; // Standard C Library Ordered Map
    set<Coordinate> c_set; // Standard C Library Ordered Set
    ordered_map<Coordinate, int> gnu_map; // Gnu map declaration
    ordered_set<Coordinate> gnu_set;// Gnu set declaration
    for (int i = 0; i < 10; i++) {
        gnu_set.insert({0, i * 10});
    }
    cout << *gnu_set.find({0, 30}) << endl; // {0, 30}
    cout << *gnu_set.lower_bound({0, 53}) << endl; // {0, 60}
    cout << *gnu_set.upper_bound({0, 53}) << endl; // {0, 60}
    cout << *gnu_set.lower_bound({0, 50}) << endl; // {0, 50}
    cout << *gnu_set.upper_bound({0, 50}) << endl; // {0, 60}
    // Example of the operations only supported by gnu_set
    cout << *gnu_set.find_by_order(2) << endl; // {0 20}
    cout << *gnu_set.find_by_order(4) << endl; // {0 40}
    assert(end(gnu_set) == gnu_set.find_by_order(10));
    cout << gnu_set.order_of_key({0, -99}) << endl; // 0
    cout << gnu_set.order_of_key({0, 0}) << endl; // 0
    cout << gnu_set.order_of_key({0, 11}) << endl; // 2
    cout << gnu_set.order_of_key({0, 999}) << endl; // 10
}
```

2.3 Suffix array

```cpp
// Suffix array construction in O(L log^2 L) time.  Routine for
// computing the length of the longest common prefix of any two
// suffixes in O(log L) time.
//
// INPUT:   string s
//
// OUTPUT:  array suffix[] such that suffix[i] = index (from 0 to L-1)
//          of substring s[i...L-1] in the list of sorted suffixes.
//          That is, if we take the inverse of the permutation suffix[],
//          we get the actual suffix array.

#include <vector>
#include <iostream>
#include <string>

using namespace std;

struct SuffixArray {
    const int L;
    string s;
    vector<vector<int> > P;
    vector<pair<pair<int, int>, int> > M;

    SuffixArray(const string &s) : L(s.length()), s(s), P(1, vector<int>(L, 0)),
          M(L) {
        for (int i = 0; i < L; i++) P[0][i] = int(s[i]);
        for (int skip = 1, level = 1; skip < L; skip *= 2, level++) {
            P.push_back(vector<int>(L, 0));
            for (int i = 0; i < L; i++)
                M[i] = make_pair(make_pair(P[level - 1][i], i + skip < L ? P[
                     level - 1][i + skip] : -1000), i);
            sort(M.begin(), M.end());
            for (int i = 0; i < L; i++)
                P[level][M[i].second] = (i > 0 && M[i].first == M[i - 1].first)
                     ? P[level][M[i - 1].second] : i;
        }
    }

    vector<int> GetSuffixArray() { return P.back(); }

    // returns the length of the longest common prefix of s[i...L-1] and s[j...L
         -1]
    int LongestCommonPrefix(int i, int j) {
        int len = 0;
        if (i == j) return L - i;
        for (int k = P.size() - 1; k >= 0 && i < L && j < L; k--) {
            if (P[k][i] == P[k][j]) {
                i += 1 << k;
                j += 1 << k;
                len += 1 << k;
            }
        }
        return len;
    }
};

// BEGIN CUT
// The following code solves UVA problem 11512: GATTACA.
#define TESTING
#ifdef TESTING
int main() {
    int T;
    cin >> T;
    for (int caseno = 0; caseno < T; caseno++) {
        string s;
        cin >> s;
        SuffixArray array(s);
        vector<int> v = array.GetSuffixArray();
        int bestlen = -1, bestpos = -1, bestcount = 0;
        for (int i = 0; i < s.length(); i++) {
            int len = 0, count = 0;
            for (int j = i + 1; j < s.length(); j++) {
                int l = array.LongestCommonPrefix(i, j);
                if (l >= len) {
                    if (l > len) count = 2; else count++;
                    len = l;
                }
            }
            if (len > bestlen || len == bestlen && s.substr(bestpos, bestlen) >
                 s.substr(i, len)) {
                bestlen = len;
                bestcount = count;
                bestpos = i;
            }
        }
        if (bestlen == 0) {
            cout << "No repetitions found!" << endl;
        } else {
            cout << s.substr(bestpos, bestlen) << " " << bestcount << endl;
        }
    }
}

#else
// END CUT
int main() {

  // bobocel is the 0'th suffix
  //  obocel is the 5'th suffix
  //   bocel is the 1'st suffix
  //    ocel is the 6'th suffix
  //     cel is the 2'nd suffix
  //      el is the 3'rd suffix
  //       l is the 4'th suffix
  SuffixArray suffix("bobocel");
  vector<int> v = suffix.GetSuffixArray();

  // Expected output: 0 5 1 6 2 3 4
  //                  2
  for (int i = 0; i < v.size(); i++) cout << v[i] << " ";
  cout << endl;
  cout << suffix.LongestCommonPrefix(0, 2) << endl;
}
// BEGIN CUT
#endif
// END CUT
```

2.4 Union-find set

```cpp
/**
 * Description: Disjoint-set data structure.
 * Time: $O(alpha(N))$
 */
struct UF {
    // E is parent set number if positive, and the size if negative.
    // If negative, it's the root of a set.
    vi e;
    UF(int n) : e(n, -1) {}
    bool sameSet(int a, int b) { return find(a) == find(b); }
    int size(int x) { return -e[find(x)]; }
    int find(int x) { return e[x] < 0 ? x : e[x] = find(e[x]); }
    int find2(int x) { // Dennis's faster union find
```

```
        while (e[x] >= 0) {e[x] = e[e[x]]; x = e[x]}
        return x;
    }
    bool join(int a, int b) {
        a = find(a), b = find(b);
        if (a == b) return false;
        if (e[a] > e[b]) swap(a, b);
        e[a] += e[b]; e[b] = a;
        return true;
    }
};
```

2.5 KD-tree

```
// -----------------------------------------------------------------
// A straightforward, but probably sub-optimal KD-tree implmentation
// that's probably good enough for most things (current it's a
// 2D-tree)
//
//  - constructs from n points in O(n lg^2 n) time
//  - handles nearest-neighbor query in O(lg n) if points are well
//    distributed
//  - worst case for nearest-neighbor may be linear in pathological
//    case
//
// Sonny Chan, Stanford University, April 2009
// -----------------------------------------------------------------

#include <iostream>
#include <vector>
#include <limits>
#include <cstdlib>

using namespace std;

// number type for coordinates, and its maximum value
typedef long long ntype;
const ntype sentry = numeric_limits<ntype>::max();

// point structure for 2D-tree, can be extended to 3D
struct point {
    ntype x, y;
    point(ntype xx = 0, ntype yy = 0) : x(xx), y(yy) {}
};

bool operator==(const point &a, const point &b) {
    return a.x == b.x && a.y == b.y;
}
// sorts points on x-coordinate
bool on_x(const point &a, const point &b) {
    return a.x < b.x;
}
// sorts points on y-coordinate
bool on_y(const point &a, const point &b) {
    return a.y < b.y;
}
// squared distance between points
ntype pdist2(const point &a, const point &b) {
    ntype dx = a.x - b.x, dy = a.y - b.y;
    return dx * dx + dy * dy;
}

// bounding box for a set of points
struct bbox {
    ntype x0, x1, y0, y1;
    bbox() : x0(sentry), x1(-sentry), y0(sentry), y1(-sentry) {}
    // computes bounding box from a bunch of points
    void compute(const vector<point> &v) {
        for (int i = 0; i < v.size(); ++i) {
            x0 = min(x0, v[i].x);
            x1 = max(x1, v[i].x);
            y0 = min(y0, v[i].y);
            y1 = max(y1, v[i].y);
        }
    }
    // squared distance between a point and this bbox, 0 if inside
    ntype distance(const point &p) {
        if (p.x < x0) {
            if (p.y < y0) return pdist2(point(x0, y0), p);
            else if (p.y > y1) return pdist2(point(x0, y1), p);
            else return pdist2(point(x0, p.y), p);
        } else if (p.x > x1) {
            if (p.y < y0) return pdist2(point(x1, y0), p);
            else if (p.y > y1) return pdist2(point(x1, y1), p);
            else return pdist2(point(x1, p.y), p);
        } else {
            if (p.y < y0) return pdist2(point(p.x, y0), p);
            else if (p.y > y1) return pdist2(point(p.x, y1), p);
            else return 0;
        }
    }
};

// stores a single node of the kd-tree, either internal or leaf
struct kdnode {
    bool leaf;      // true if this is a leaf node (has one point)
    point pt;       // the single point of this is a leaf
    bbox bound;     // bounding box for set of points in children

    kdnode *first, *second; // two children of this kd-node

    kdnode() : leaf(false), first(0), second(0) {}
    ~kdnode() {
        if (first) delete first;
        if (second) delete second;
    }
    // intersect a point with this node (returns squared distance)
    ntype intersect(const point &p) {
        return bound.distance(p);
    }
    // recursively builds a kd-tree from a given cloud of points
    void construct(vector<point> &vp) {
        // compute bounding box for points at this node
        bound.compute(vp);

        // if we're down to one point, then we're a leaf node
        if (vp.size() == 1) {
            leaf = true;
            pt = vp[0];
        } else {
            // split on x if the bbox is wider than high (not best heuristic...)
            if (bound.x1 - bound.x0 >= bound.y1 - bound.y0)
                sort(vp.begin(), vp.end(), on_x);
                // otherwise split on y-coordinate
            else
                sort(vp.begin(), vp.end(), on_y);

            // divide by taking half the array for each child
            // (not best performance if many duplicates in the middle)
            int half = vp.size() / 2;
            vector<point> vl(vp.begin(), vp.begin() + half);
            vector<point> vr(vp.begin() + half, vp.end());
            first = new kdnode();
            first->construct(vl);
            second = new kdnode();
            second->construct(vr);
        }
    }
};

// simple kd-tree class to hold the tree and handle queries
struct kdtree {
    kdnode *root;
    // constructs a kd-tree from a points (copied here, as it sorts them)
    kdtree(const vector<point> &vp) {
        vector<point> v(vp.begin(), vp.end());
        root = new kdnode();
        root->construct(v);
    }
    ~kdtree() { delete root; }
    // recursive search method returns squared distance to nearest point
    ntype search(kdnode *node, const point &p) {
        if (node->leaf) {
            // commented special case tells a point not to find itself
//            if (p == node->pt) return sentry;
//            else
            return pdist2(p, node->pt);
        }

        ntype bfirst = node->first->intersect(p);
        ntype bsecond = node->second->intersect(p);

        // choose the side with the closest bounding box to search first
        // (note that the other side is also searched if needed)
        if (bfirst < bsecond) {
            ntype best = search(node->first, p);
            if (bsecond < best)
                best = min(best, search(node->second, p));
            return best;
        } else {
            ntype best = search(node->second, p);
            if (bfirst < best)
                best = min(best, search(node->first, p));
            return best;
        }
    }
    // squared distance to the nearest
    ntype nearest(const point &p) {
        return search(root, p);
    }
};

// --------------------------------------------------------------------------
// some basic test code here

int main() {
    // generate some random points for a kd-tree
    vector<point> vp;
    for (int i = 0; i < 100000; ++i) {
        vp.push_back(point(rand() % 100000, rand() % 100000));
    }
    kdtree tree(vp);

    // query some points
    for (int i = 0; i < 10; ++i) {
        point q(rand() % 100000, rand() % 100000);
        cout << "Closest squared distance to (" << q.x << ", " << q.y << ")"
             << " is " << tree.nearest(q) << endl;
    }

    return 0;
}

// --------------------------------------------------------------------------
```

2.6 Segment tree

```
#include <bits/stdc++.h>
using namespace std;
typedef vector<int> vi;
struct Tree {
    typedef int T;
    static constexpr T unit = INT_MIN;
    T f(T a, T b) { return max(a, b); } // (any associative fn)
    vector<T> s; int n;
    Tree(int n = 0, T def = unit) : s(2*n, def), n(n) {}
    void update(int pos, T val) {
        for (s[pos += n] = val; pos /= 2;)
            s[pos] = f(s[pos * 2], s[pos * 2 + 1]);
    }
    T query(int b, int e) { // query [b, e)
        T ra = unit, rb = unit;
        for (b += n, e += n; b < e; b /= 2, e /= 2) {
            if (b % 2) ra = f(ra, s[b++]);
            if (e % 2) rb = f(s[--e], rb);
        }
        return f(ra, rb);
    }
};
```

2.7 Lazy segment tree

```
#include <bits/stdc++.h>
using namespace std;
typedef vector<int> vi;
const int inf = 1e9;
// A lazy segment tree supporting range add, range set, and range get max
struct Node {
    Node *l = 0, *r = 0;
    int lo, hi, mset = inf, madd = 0, val = -inf;
    Node(int lo,int hi):lo(lo),hi(hi){} // Large interval of -inf
    // Initialize based on the values in the vector v.
    // main will call this with Node(v, 0, v.size())
    Node(vi& v, int lo, int hi) : lo(lo), hi(hi) {
        if (lo + 1 < hi) {
            int mid = lo + (hi - lo)/2;
            l = new Node(v, lo, mid); r = new Node(v, mid, hi);
            val = max(l->val, r->val);
        }
        else val = v[lo];
    }
    // query [L, R)
    int query(int L, int R) {
        if (R <= lo || hi <= L) return -inf;
        if (L <= lo && hi <= R) return val;
        push();
        return max(l->query(L, R), r->query(L, R));
    }
    // set all elements in [L, R) to x
    void set(int L, int R, int x) {
        if (R <= lo || hi <= L) return;
        if (L <= lo && hi <= R) {
            // Update the range [lo, hi) to x
            mset = val = x, madd = 0;
        }
        else {
            push(), l->set(L, R, x), r->set(L, R, x);
            val = max(l->val, r->val);
        }
    }
    // add x to all elements in [L, R)
    void add(int L, int R, int x) {
        if (R <= lo || hi <= L) return;
        if (L <= lo && hi <= R) {
            // Add x to all elements in the range [lo, hi)
            if (mset != inf) mset += x;
            else madd += x;
            val += x;
        }
        else {
            push(), l->add(L, R, x), r->add(L, R, x);
            val = max(l->val, r->val);
        }
    }
    // Push the lazily stored values.
    void push() {
        if (!l) {
            int mid = lo + (hi - lo)/2;
            l = new Node(lo, mid); r = new Node(mid, hi);
        }
        if (mset != inf)
            l->set(lo,hi,mset), r->set(lo,hi,mset), mset = inf;
        else if (madd)
            l->add(lo,hi,madd), r->add(lo,hi,madd), madd = 0;
    }
};
```

2.8 Lowest common ancestor

```
/**
 * Description: Calculate power of two jumps in a tree,
 * to support fast upward jumps and LCAs.
 * Assumes the root node points to itself.
 * Time: construction $O(N \log N)$, queries $O(\log N)$
 * Status: Tested at Petrozavodsk, also stress-tested via LCA.cpp
 */
 // Takes an array of parent pointers and returns
 // the LCA table.
vector<vi> treeJump(vi& P){
    int on = 1, d = 1;
    while(on < sz(P)) on *= 2, d++;
    vector<vi> jmp(d, P);
    rep(i,1,d) rep(j,0,sz(P))
    jmp[i][j] = jmp[i-1][jmp[i-1][j]];
    return jmp;
}
// helper function to jump a certain number of parents
// up from a node in O(log(steps)) time.
int jmp(vector<vi>& tbl, int nod, int steps){
    rep(i,0,sz(tbl))
    if(steps&(1<<i)) nod = tbl[i][nod];
    return nod;
}
// Find least common ancestor of two nodes.
int lca(vector<vi>& tbl, vi& depth, int a, int b) {
    if (depth[a] < depth[b]) swap(a, b);
    a = jmp(tbl, a, depth[a] - depth[b]);
    if (a == b) return a;
    for (int i = sz(tbl); i--;) {
        int c = tbl[i][a], d = tbl[i][b];
        if (c != d) a = c, b = d;
    }
    return tbl[0][a];
}
```

2.9 Li Chao Tree

```
// Li-Chao Tree. Store a family of functions with domain a subset of R
// where no two functions intersect
// more than once. Support querying for the max of all functions in the tree
// in O(log(n)) with O(log(n)) insertion
struct LiChao {
    typedef int ftype;
    typedef pair<int, int> params;
    int maxn;

    vector<params> best_params;

    LiChao(int maxN) {
        maxn = maxN;
        best_params = vector<params>(maxn * 4);
    }

    // The function you add to the tree. It is a family of functions
    // parameterized by a
    // Any two functions f(a, -), f(b, -) must intersect at most once,
    // else the tree will not work.
    ftype f(params a, ftype x) {
        return a.first * x + a.second;
    }

    // Add the function parameterized by nw to the tree
    void add_fn(params nw, int v, int l, int r) {
        int m = (l + r) / 2;
        bool lef = f(nw, l) < f(best_params[v], l);
        bool mid = f(nw, m) < f(best_params[v], m);
        if (mid) {
            swap(best_params[v], nw);
        }
        if (r - l == 1) {
            return;
        } else if (lef != mid) {
            add_fn(nw, 2 * v, l, m);
        } else {
            add_fn(nw, 2 * v + 1, m, r);
        }
    }

    void add_fn(params nw) {
        return add_fn(nw, 1, 0, maxn);
    }

    // Compute the maximum valued function over all x
    int get(int x, int v, int l, int r) {
        int m = (l + r) / 2;
        if (r - l == 1) {
            return f(best_params[v], x);
        } else if (x < m) {
            return min(f(best_params[v], x), get(x, 2 * v, l, m));
        } else {
            return min(f(best_params[v], x), get(x, 2 * v + 1, m, r));
        }
    }

    int get(int x) {
        return get(x, 1, 0, maxn);
    }
};
```

3 Combinatorial optimization

3.1 Sparse max-flow

```
// Adjacency list implementation of Dinic's blocking flow algorithm.
// This is very fast in practice, and only loses to push-relabel flow.
//
// Running time:
//     O(|V|^2 |E|)
//
// INPUT:
//     - graph, constructed using AddEdge()
//     - source and sink
//
// OUTPUT:
//     - maximum flow value
//     - To obtain actual flow values, look at edges with capacity > 0
//       (zero capacity edges are residual edges).

#include<cstdio>
#include<vector>
#include<queue>
using namespace std;
typedef long long LL;

struct Edge {
    int u, v;
    LL cap, flow;
    Edge() {}
    Edge(int u, int v, LL cap) : u(u), v(v), cap(cap), flow(0) {}
};

struct Dinic {
    int N;
    vector<Edge> E;
    vector<vector<int>> g;
    vector<int> d, pt;
    Dinic(int N) : N(N), E(0), g(N), d(N), pt(N) {}
    void AddEdge(int u, int v, LL cap) {
        if (u != v) {
            E.emplace_back(u, v, cap);
            g[u].emplace_back(E.size() - 1);
            E.emplace_back(v, u, 0);
            g[v].emplace_back(E.size() - 1);
        }
    }
    bool BFS(int S, int T) {
        queue<int> q({S});
        fill(d.begin(), d.end(), N + 1);
        d[S] = 0;
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            if (u == T) break;
            for (int k: g[u]) {
                Edge &e = E[k];
                if (e.flow < e.cap && d[e.v] > d[e.u] + 1) {
                    d[e.v] = d[e.u] + 1;
                    q.emplace(e.v);
                }
            }
        }
        return d[T] != N + 1;
    }
    LL DFS(int u, int T, LL flow = -1) {
        if (u == T || flow == 0) return flow;
        for (int &i = pt[u]; i < g[u].size(); ++i) {
            Edge &e = E[g[u][i]];
            Edge &oe = E[g[u][i] ^ 1];
            if (d[e.v] == d[e.u] + 1) {
                LL amt = e.cap - e.flow;
                if (flow != -1 && amt > flow) amt = flow;
                if (LL pushed = DFS(e.v, T, amt)) {
                    e.flow += pushed;
                    oe.flow -= pushed;
                    return pushed;
                }
            }
        }
        return 0;
    }
    LL MaxFlow(int S, int T) {
        LL total = 0;
        while (BFS(S, T)) {
            fill(pt.begin(), pt.end(), 0);
            while (LL flow = DFS(S, T))
                total += flow;
        }
        return total;
    }
};

// BEGIN CUT
// The following code solves SPOJ problem #4110: Fast Maximum Flow (FASTFLOW)

int main() {
    int N, E;
    scanf("%d%d", &N, &E);
    Dinic dinic(N);
    for (int i = 0; i < E; i++) {
        int u, v;
        LL cap;
        scanf("%d%d%lld", &u, &v, &cap);
        dinic.AddEdge(u - 1, v - 1, cap);
        dinic.AddEdge(v - 1, u - 1, cap);
    }
    printf("%lld\n", dinic.MaxFlow(0, N - 1));
    return 0;
}

// END CUT
```

3.2 Min-cost max-flow

```
// Implementation of min cost max flow algorithm using adjacency
// matrix (Edmonds and Karp 1972).  This implementation keeps track of
// forward and reverse edges separately (so you can set cap[i][j] !=
// cap[j][i]).  For a regular max flow, set all edge costs to 0.
//
// Running time, O(|V|^2) cost per augmentation
//     max flow:           O(|V|^3) augmentations
//     min cost max flow:  O(|V|^4 * MAX_EDGE_COST) augmentations
//
// INPUT:
//     - graph, constructed using AddEdge()
//     - source
//     - sink
//
// OUTPUT:
//     - (maximum flow value, minimum cost value)
//     - To obtain the actual flow, look at positive values only.
```

```
#include <cmath>
#include <vector>
#include <iostream>

using namespace std;

typedef vector<int> VI;
typedef vector<VI> VVI;
typedef long long L;
typedef vector<L> VL;
typedef vector<VL> VVL;
typedef pair<int, int> PII;
typedef vector<PII> VPII;
const L INF = numeric_limits<L>::max() / 4;

struct MinCostMaxFlow {
    int N;
    VVL cap, flow, cost;
    VI found;
    VL dist, pi, width;
    VPII dad;
    MinCostMaxFlow(int N) :
            N(N), cap(N, VL(N)), flow(N, VL(N)), cost(N, VL(N)),
            found(N), dist(N), pi(N), width(N), dad(N) {}
    void AddEdge(int from, int to, L cap, L cost) {
        this->cap[from][to] = cap;
        this->cost[from][to] = cost;
    }
    void Relax(int s, int k, L cap, L cost, int dir) {
        L val = dist[s] + pi[s] - pi[k] + cost;
        if (cap && val < dist[k]) {
            dist[k] = val;
            dad[k] = make_pair(s, dir);
            width[k] = min(cap, width[s]);
        }
    }
    L Dijkstra(int s, int t) {
        fill(found.begin(), found.end(), false);
        fill(dist.begin(), dist.end(), INF);
        fill(width.begin(), width.end(), 0);
        dist[s] = 0;
        width[s] = INF;

        while (s != -1) {
            int best = -1;
            found[s] = true;
            for (int k = 0; k < N; k++) {
                if (found[k]) continue;
                Relax(s, k, cap[s][k] - flow[s][k], cost[s][k], 1);
                Relax(s, k, flow[k][s], -cost[k][s], -1);
                if (best == -1 || dist[k] < dist[best]) best = k;
            }
            s = best;
        }

        for (int k = 0; k < N; k++)
            pi[k] = min(pi[k] + dist[k], INF);
        return width[t];
    }
    pair<L, L> GetMaxFlow(int s, int t) {
        L totflow = 0, totcost = 0;
        while (L amt = Dijkstra(s, t)) {
            totflow += amt;
            for (int x = t; x != s; x = dad[x].first) {
                if (dad[x].second == 1) {
                    flow[dad[x].first][x] += amt;
                    totcost += amt * cost[dad[x].first][x];
                } else {
                    flow[x][dad[x].first] -= amt;
                    totcost -= amt * cost[x][dad[x].first];
                }
            }
        }
        return make_pair(totflow, totcost);
    }
};

// BEGIN CUT
// The following code solves UVA problem #10594: Data Flow

int main() {
    int N, M;

    while (scanf("%d%d", &N, &M) == 2) {
        VVL v(M, VL(3));
        for (int i = 0; i < M; i++)
            scanf("%Ld%Ld%Ld", &v[i][0], &v[i][1], &v[i][2]);
        L D, K;
        scanf("%Ld%Ld", &D, &K);

        MinCostMaxFlow mcmf(N + 1);
        for (int i = 0; i < M; i++) {
            mcmf.AddEdge(int(v[i][0]), int(v[i][1]), K, v[i][2]);
            mcmf.AddEdge(int(v[i][1]), int(v[i][0]), K, v[i][2]);
        }
        mcmf.AddEdge(0, 1, D, 0);

        pair<L, L> res = mcmf.GetMaxFlow(0, N);

        if (res.first == D) {
            printf("%Ld\n", res.second);
        } else {
            printf("Impossible.\n");
        }
    }

    return 0;
}

// END CUT
```

3.3 Min-cost matching

```
///////////////////////////////////////////////////////////////////////
// Min cost bipartite matching via shortest augmenting paths
//
// This is an O(n^3) implementation of a shortest augmenting path
// algorithm for finding min cost perfect matchings in dense
// graphs.  In practice, it solves 1000x1000 problems in around 1
// second.
//
//   cost[i][j] = cost for pairing left node i with right node j
//   Lmate[i] = index of right node that left node i pairs with
//   Rmate[j] = index of left node that right node j pairs with
//
// The values in cost[i][j] may be positive or negative.  To perform
// maximization, simply negate the cost[][] matrix.
///////////////////////////////////////////////////////////////////////

#include <algorithm>
#include <cstdio>
#include <cmath>
#include <vector>

using namespace std;
```

```
typedef vector<double> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;
double MinCostMatching(const VVD &cost, VI &Lmate, VI &Rmate) {
    int n = int(cost.size());

    // construct dual feasible solution
    VD u(n);
    VD v(n);
    for (int i = 0; i < n; i++) {
        u[i] = cost[i][0];
        for (int j = 1; j < n; j++) u[i] = min(u[i], cost[i][j]);
    }
    for (int j = 0; j < n; j++) {
        v[j] = cost[0][j] - u[0];
        for (int i = 1; i < n; i++) v[j] = min(v[j], cost[i][j] - u[i]);
    }

    // construct primal solution satisfying complementary slackness
    Lmate = VI(n, -1);
    Rmate = VI(n, -1);
    int mated = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (Rmate[j] != -1) continue;
            if (fabs(cost[i][j] - u[i] - v[j]) < 1e-10) {
                Lmate[i] = j;
                Rmate[j] = i;
                mated++;
                break;
            }
        }
    }

    VD dist(n);
    VI dad(n);
    VI seen(n);

    // repeat until primal solution is feasible
    while (mated < n) {

        // find an unmatched left node
        int s = 0;
        while (Lmate[s] != -1) s++;

        // initialize Dijkstra
        fill(dad.begin(), dad.end(), -1);
        fill(seen.begin(), seen.end(), 0);
        for (int k = 0; k < n; k++)
            dist[k] = cost[s][k] - u[s] - v[k];

        int j = 0;
        while (true) {

            // find closest
            j = -1;
            for (int k = 0; k < n; k++) {
                if (seen[k]) continue;
                if (j == -1 || dist[k] < dist[j]) j = k;
            }
            seen[j] = 1;

            // termination condition
            if (Rmate[j] == -1) break;

            // relax neighbors
            const int i = Rmate[j];
            for (int k = 0; k < n; k++) {
                if (seen[k]) continue;
                const double new_dist = dist[j] + cost[i][k] - u[i] - v[k];
                if (dist[k] > new_dist) {
                    dist[k] = new_dist;
                    dad[k] = j;
                }
            }
        }

        // update dual variables
        for (int k = 0; k < n; k++) {
            if (k == j || !seen[k]) continue;
            const int i = Rmate[k];
            v[k] += dist[k] - dist[j];
            u[i] -= dist[k] - dist[j];
        }
        u[s] += dist[j];

        // augment along path
        while (dad[j] >= 0) {
            const int d = dad[j];
            Rmate[j] = Rmate[d];
            Lmate[Rmate[j]] = j;
            j = d;
        }
        Rmate[j] = s;
        Lmate[s] = j;

        mated++;
    }

    double value = 0;
    for (int i = 0; i < n; i++)
        value += cost[i][Lmate[i]];

    return value;
}
```

3.4 Max bipartite matching

```
// This code performs maximum bipartite matching.
//
// Running time: O(|E| |V|) -- often much faster in practice
//
//   INPUT: w[i][j] = edge between row node i and column node j
//   OUTPUT: mr[i] = assignment for row node i, -1 if unassigned
//           mc[j] = assignment for column node j, -1 if unassigned
//           function returns number of matches made

#include <vector>

using namespace std;

typedef vector<int> VI;
typedef vector<VI> VVI;
bool FindMatch(int i, const VVI &w, VI &mr, VI &mc, VI &seen) {
    for (int j = 0; j < w[i].size(); j++) {
        if (w[i][j] && !seen[j]) {
            seen[j] = true;
            if (mc[j] < 0 || FindMatch(mc[j], w, mr, mc, seen)) {
                mr[i] = j;
                mc[j] = i;
                return true;
            }
        }
    }
    return false;
}
```

```
int BipartiteMatching(const VVI &w, VI &mr, VI &mc) {
    mr = VI(w.size(), -1);
    mc = VI(w[0].size(), -1);

    int ct = 0;
    for (int i = 0; i < w.size(); i++) {
        VI seen(w[0].size());
        if (FindMatch(i, w, mr, mc, seen)) ct++;
    }
    return ct;
}
```

3.5 Global min-cut

```
// Adjacency matrix implementation of Stoer-Wagner min cut algorithm.
//
// Running time:
//     O(|V|^3)
//
// INPUT:
//     - graph, constructed using AddEdge()
//
// OUTPUT:
//     - (min cut value, nodes in half of min cut)

#include <cmath>
#include <vector>
#include <iostream>

using namespace std;

typedef vector<int> VI;
typedef vector<VI> VVI;
const int INF = 1000000000;
pair<int, VI> GetMinCut(VVI &weights) {
    int N = weights.size();
    VI used(N), cut, best_cut;
    int best_weight = -1;

    for (int phase = N - 1; phase >= 0; phase--) {
        VI w = weights[0];
        VI added = used;
        int prev, last = 0;
        for (int i = 0; i < phase; i++) {
            prev = last;
            last = -1;
            for (int j = 1; j < N; j++)
                if (!added[j] && (last == -1 || w[j] > w[last])) last = j;
            if (i == phase - 1) {
                for (int j = 0; j < N; j++) weights[prev][j] += weights[last][j
                    ];
                for (int j = 0; j < N; j++) weights[j][prev] = weights[prev][j];
                used[last] = true;
                cut.push_back(last);
                if (best_weight == -1 || w[last] < best_weight) {
                    best_cut = cut;
                    best_weight = w[last];
                }
            } else {
                for (int j = 0; j < N; j++)
                    w[j] += weights[last][j];
                added[last] = true;
            }
        }
    }
    return make_pair(best_weight, best_cut);
}

// BEGIN CUT
// The following code solves UVA problem #10989: Bomb, Divide and Conquer
int main() {
    int N;
    cin >> N;
    for (int i = 0; i < N; i++) {
        int n, m;
        cin >> n >> m;
        VVI weights(n, VI(n));
        for (int j = 0; j < m; j++) {
            int a, b, c;
            cin >> a >> b >> c;
            weights[a - 1][b - 1] = weights[b - 1][a - 1] = c;
        }
        pair<int, VI> res = GetMinCut(weights);
        cout << "Case #" << i + 1 << ": " << res.first << endl;
    }
}
// END CUT
```

4 Geometry

4.1 Convex hull

```
// Compute the 2D convex hull of a set of points using the monotone chain
// algorithm.  Eliminate redundant points from the hull if REMOVE_REDUNDANT is
// #defined.
//
// Running time: O(n log n)
//
//   INPUT:   a vector of input points, unordered.
//   OUTPUT:  a vector of points in the convex hull, counterclockwise, starting
//            with bottommost/leftmost point

#include <cstdio>
#include <cassert>
#include <vector>
#include <algorithm>
#include <cmath>
// BEGIN CUT
#include <map>
// END CUT

using namespace std;

#define REMOVE_REDUNDANT

typedef double T;
const T EPS = 1e-7;

struct PT {
    T x, y;
    PT() {}
    PT(T x, T y) : x(x), y(y) {}
    bool operator<(const PT &rhs) const { return make_pair(y, x) < make_pair(rhs
        .y, rhs.x); }
    bool operator==(const PT &rhs) const { return make_pair(y, x) == make_pair(
        rhs.y, rhs.x); }
};

T cross(PT p, PT q) { return p.x * q.y - p.y * q.x; }
T area2(PT a, PT b, PT c) { return cross(a, b) + cross(b, c) + cross(c, a); }

#ifdef REMOVE_REDUNDANT
bool between(const PT &a, const PT &b, const PT &c) {
    return (fabs(area2(a, b, c)) < EPS && (a.x - b.x) * (c.x - b.x) <= 0 && (a.y
          - b.y) * (c.y - b.y) <= 0);
}
#endif

void ConvexHull(vector<PT> &pts) {
    sort(pts.begin(), pts.end());
    pts.erase(unique(pts.begin(), pts.end()), pts.end());
    vector<PT> up, dn;
    for (int i = 0; i < pts.size(); i++) {
        while (up.size() > 1 && area2(up[up.size() - 2], up.back(), pts[i]) >=
             0) up.pop_back();
        while (dn.size() > 1 && area2(dn[dn.size() - 2], dn.back(), pts[i]) <=
             0) dn.pop_back();
        up.push_back(pts[i]);
        dn.push_back(pts[i]);
    }
    pts = dn;
    for (int i = (int) up.size() - 2; i >= 1; i--) pts.push_back(up[i]);

#ifdef REMOVE_REDUNDANT
    if (pts.size() <= 2) return;
    dn.clear();
    dn.push_back(pts[0]);
    dn.push_back(pts[1]);
    for (int i = 2; i < pts.size(); i++) {
        if (between(dn[dn.size() - 2], dn[dn.size() - 1], pts[i])) dn.pop_back()
             ;
        dn.push_back(pts[i]);
    }
    if (dn.size() >= 3 && between(dn.back(), dn[0], dn[1])) {
        dn[0] = dn.back();
        dn.pop_back();
    }
    pts = dn;
#endif
}

// BEGIN CUT
// The following code solves SPOJ problem #26: Build the Fence (BSHEEP)

int main() {
    int t;
    scanf("%d", &t);
    for (int caseno = 0; caseno < t; caseno++) {
        int n;
        scanf("%d", &n);
        vector<PT> v(n);
        for (int i = 0; i < n; i++) scanf("%lf%lf", &v[i].x, &v[i].y);
        vector<PT> h(v);
        map<PT, int> index;
        for (int i = n - 1; i >= 0; i--) index[v[i]] = i + 1;
        ConvexHull(h);

        double len = 0;
        for (int i = 0; i < h.size(); i++) {
            double dx = h[i].x - h[(i + 1) % h.size()].x;
            double dy = h[i].y - h[(i + 1) % h.size()].y;
            len += sqrt(dx * dx + dy * dy);
        }

        if (caseno > 0) printf("\n");
        printf("%.2f\n", len);
        for (int i = 0; i < h.size(); i++) {
            if (i > 0) printf(" ");
            printf("%d", index[h[i]]);
        }
        printf("\n");
    }
}

// END CUT
```

4.2 Miscellaneous geometry

```
// C++ routines for computational geometry.

#include <iostream>
#include <vector>
#include <cmath>
#include <cassert>

using namespace std;

double INF = 1e100;
double EPS = 1e-12;

struct PT {
    double x, y;
    PT() {}
    PT(double x, double y) : x(x), y(y) {}
    PT(const PT &p) : x(p.x), y(p.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); }
};

double dot(PT p, PT q) { return p.x * q.x + p.y * q.y; }
double dist2(PT p, PT q) { return dot(p - q, p - q); }
double cross(PT p, PT q) { return p.x * q.y - p.y * q.x; }
ostream &operator<<(ostream &os, const PT &p) {
    return os << "(" << p.x << "," << p.y << ")";
}
// rotate a point CCW or CW around the origin
PT RotateCCW90(PT p) { return PT(-p.y, p.x); }
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));
}
// project point c onto line through a and b
// assuming a != b
PT ProjectPointLine(PT a, PT b, PT c) {
    return a + (b - a) * dot(c - a, b - a) / dot(b - a, b - a);
}
// project point c onto line segment through a and b
PT ProjectPointSegment(PT a, PT b, PT c) {
    double r = dot(b - a, b - a);
    if (fabs(r) < EPS) return a;
    r = dot(c - a, b - a) / r;
    if (r < 0) return a;
    if (r > 1) return b;
    return a + (b - a) * r;
}
// compute distance from c to segment between a and b
double DistancePointSegment(PT a, PT b, PT c) {
    return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
}
// compute distance between point (x,y,z) and plane ax+by+cz=d
double DistancePointPlane(double x, double y, double z,
                          double a, double b, double c, double d) {
    return fabs(a * x + b * y + c * z - d) / sqrt(a * a + b * b + c * c);
}
// determine if lines from a to b and c to d are parallel or collinear
bool LinesParallel(PT a, PT b, PT c, PT d) {
    return fabs(cross(b - a, c - d)) < EPS;
```

```
}
bool LinesCollinear(PT a, PT b, PT c, PT d) {
    return LinesParallel(a, b, c, d)
           && fabs(cross(a - b, a - c)) < EPS
           && fabs(cross(c - d, c - a)) < EPS;
}
// determine if line segment from a to b intersects with
// line segment from c to d
bool SegmentsIntersect(PT a, PT b, PT c, PT d) {
    if (LinesCollinear(a, b, c, d)) {
        if (dist2(a, c) < EPS || dist2(a, d) < EPS ||
            dist2(b, c) < EPS || dist2(b, d) < EPS)
            return true;
        if (dot(c - a, c - b) > 0 && dot(d - a, d - b) > 0 && dot(c - b, d - b)
             > 0)
            return false;
        return true;
    }
    if (cross(d - a, b - a) * cross(c - a, b - a) > 0) return false;
    if (cross(a - c, d - c) * cross(b - c, d - c) > 0) return false;
    return true;
}
// compute intersection of line passing through a and b
// with line passing through c and d, assuming that unique
// intersection exists; for segment intersection, check if
// segments intersect first
PT ComputeLineIntersection(PT a, PT b, PT c, PT d) {
    b = b - a;
    d = c - d;
    c = c - a;
    assert(dot(b, b) > EPS && dot(d, d) > EPS);
    return a + b * cross(c, d) / cross(b, d);
}
// compute center of circle given three points
PT ComputeCircleCenter(PT a, PT b, PT c) {
    b = (a + b) / 2;
    c = (a + c) / 2;
    return ComputeLineIntersection(b, b + RotateCW90(a - b), c, c + RotateCW90(a
          - c));
}
// determine if point is in a possibly non-convex polygon (by William
// Randolph Franklin); returns 1 for strictly interior points, 0 for
// strictly exterior points, and 0 or 1 for the remaining points.
// Note that it is possible to convert this into an *exact* test using
// integer arithmetic by taking care of the division appropriately
// (making sure to deal with signs properly) and then by writing exact
// tests for checking point on polygon boundary
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;
}
// determine if point is on the boundary of a polygon
bool PointOnPolygon(const vector<PT> &p, PT q) {
    for (int i = 0; i < p.size(); i++)
        if (dist2(ProjectPointSegment(p[i], p[(i + 1) % p.size()], q), q) < EPS)
            return true;
    return false;
}
// compute intersection of line through points a and b with
// circle centered at c with radius r > 0
vector<PT> CircleLineIntersection(PT a, PT b, PT c, double r) {
    vector<PT> ret;
    b = b - a;
    a = a - c;
    double A = dot(b, b);
    double B = dot(a, b);
    double C = dot(a, a) - r * r;
    double D = B * B - A * C;
    if (D < -EPS) return ret;
    ret.push_back(c + a + b * (-B + sqrt(D + EPS)) / A);
    if (D > EPS)
        ret.push_back(c + a + b * (-B - sqrt(D)) / A);
    return ret;
}
// compute intersection of circle centered at a with radius r
// with circle centered at b with radius R
vector<PT> CircleCircleIntersection(PT a, PT b, double r, double R) {
    vector<PT> ret;
    double d = sqrt(dist2(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);
    double y = sqrt(r * r - x * x);
    PT v = (b - a) / d;
    ret.push_back(a + v * x + RotateCCW90(v) * y);
    if (y > 0)
        ret.push_back(a + v * x - RotateCCW90(v) * y);
    return ret;
}
// This code computes the area or centroid of a (possibly nonconvex)
// polygon, assuming that the coordinates are listed in a clockwise or
// counterclockwise fashion.  Note that the centroid is often known as
// the "center of gravity" or "center of mass".
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 fabs(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);
    }
    return c / scale;
}
// tests whether or not a given polygon (in CW or CCW order) is simple
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;
}
int main() {

    // expected: (-5,2)
    cerr << RotateCCW90(PT(2, 5)) << endl;
```

```
  // expected: (5,-2)
  cerr << RotateCW90(PT(2, 5)) << endl;

  // expected: (-5,2)
  cerr << RotateCCW(PT(2, 5), M_PI / 2) << endl;

  // expected: (5,2)
  cerr << ProjectPointLine(PT(-5, -2), PT(10, 4), PT(3, 7)) << endl;

  // expected: (5,2) (7.5,3) (2.5,1)
  cerr << ProjectPointSegment(PT(-5, -2), PT(10, 4), PT(3, 7)) << " "
       << ProjectPointSegment(PT(7.5, 3), PT(10, 4), PT(3, 7)) << " "
       << ProjectPointSegment(PT(-5, -2), PT(2.5, 1), PT(3, 7)) << endl;

  // expected: 6.78903
  cerr << DistancePointPlane(4, -4, 3, 2, -2, 5, -8) << endl;

  // expected: 1 0 1
  cerr << LinesParallel(PT(1, 1), PT(3, 5), PT(2, 1), PT(4, 5)) << " "
       << LinesParallel(PT(1, 1), PT(3, 5), PT(2, 0), PT(4, 5)) << " "
       << LinesParallel(PT(1, 1), PT(3, 5), PT(5, 9), PT(7, 13)) << endl;

  // expected: 0 0 1
  cerr << LinesCollinear(PT(1, 1), PT(3, 5), PT(2, 1), PT(4, 5)) << " "
       << LinesCollinear(PT(1, 1), PT(3, 5), PT(2, 0), PT(4, 5)) << " "
       << LinesCollinear(PT(1, 1), PT(3, 5), PT(5, 9), PT(7, 13)) << endl;

  // expected: 1 1 1 0
  cerr << SegmentsIntersect(PT(0, 0), PT(2, 4), PT(3, 1), PT(-1, 3)) << " "
       << SegmentsIntersect(PT(0, 0), PT(2, 4), PT(4, 3), PT(0, 5)) << " "
       << SegmentsIntersect(PT(0, 0), PT(2, 4), PT(2, -1), PT(-2, 1)) << " "
       << SegmentsIntersect(PT(0, 0), PT(2, 4), PT(5, 5), PT(1, 7)) << endl;

  // expected: (1,2)
  cerr << ComputeLineIntersection(PT(0, 0), PT(2, 4), PT(3, 1), PT(-1, 3)) <<
       endl;

  // expected: (1,1)
  cerr << ComputeCircleCenter(PT(-3, 4), PT(6, 1), PT(4, 5)) << endl;

  vector<PT> v;
  v.push_back(PT(0, 0));
  v.push_back(PT(5, 0));
  v.push_back(PT(5, 5));
  v.push_back(PT(0, 5));

  // expected: 1 1 1 0 0
  cerr << PointInPolygon(v, PT(2, 2)) << " "
       << PointInPolygon(v, PT(2, 0)) << " "
       << PointInPolygon(v, PT(0, 2)) << " "
       << PointInPolygon(v, PT(5, 2)) << " "
       << PointInPolygon(v, PT(2, 5)) << endl;

  // expected: 0 1 1 1 1
  cerr << PointOnPolygon(v, PT(2, 2)) << " "
       << PointOnPolygon(v, PT(2, 0)) << " "
       << PointOnPolygon(v, PT(0, 2)) << " "
       << PointOnPolygon(v, PT(5, 2)) << " "
       << PointOnPolygon(v, PT(2, 5)) << endl;

  // expected: (1,6)
  //           (5,4) (4,5)
  //           blank line
  //           (4,5) (5,4)
  //           blank line
  //           (4,5) (5,4)
  vector<PT> u = CircleLineIntersection(PT(0, 6), PT(2, 6), PT(1, 1), 5);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " ";
  cerr << endl;
  u = CircleLineIntersection(PT(0, 9), PT(9, 0), PT(1, 1), 5);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " ";
  cerr << endl;
  u = CircleCircleIntersection(PT(1, 1), PT(10, 10), 5, 5);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " ";
  cerr << endl;
  u = CircleCircleIntersection(PT(1, 1), PT(8, 8), 5, 5);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " ";
  cerr << endl;
  u = CircleCircleIntersection(PT(1, 1), PT(4.5, 4.5), 10, sqrt(2.0) / 2.0);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " ";
  cerr << endl;
  u = CircleCircleIntersection(PT(1, 1), PT(4.5, 4.5), 5, sqrt(2.0) / 2.0);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " ";
  cerr << endl;

  // area should be 5.0
  // centroid should be (1.1666666, 1.166666)
  PT pa[] = {PT(0, 0), PT(5, 0), PT(1, 1), PT(0, 5)};
  vector<PT> p(pa, pa + 4);
  PT c = ComputeCentroid(p);
  cerr << "Area: " << ComputeArea(p) << endl;
  cerr << "Centroid: " << c << endl;

  return 0;
}
```

4.3 3D geometry

```
public class Geom3D {
  // distance from point (x, y, z) to plane aX + bY + cZ + d = 0
  public static double ptPlaneDist(double x, double y, double z,
      double a, double b, double c, double d) {
    return Math.abs(a*x + b*y + c*z + d) / Math.sqrt(a*a + b*b + c*c);
  }

  // distance between parallel planes aX + bY + cZ + d1 = 0 and
  // aX + bY + cZ + d2 = 0
  public static double planePlaneDist(double a, double b, double c,
      double d1, double d2) {
    return Math.abs(d1 - d2) / Math.sqrt(a*a + b*b + c*c);
  }

  // distance from point (px, py, pz) to line (x1, y1, z1)-(x2, y2, z2)
  // (or ray, or segment; in the case of the ray, the endpoint is the
  // first point)
  public static final int LINE = 0;
  public static final int SEGMENT = 1;
  public static final int RAY = 2;
  public static double ptLineDistSq(double x1, double y1, double z1,
      double x2, double y2, double z2, double px, double py, double pz,
      int type) {
    double pd2 = (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) + (z1-z2)*(z1-z2);

    double x, y, z;
    if (pd2 == 0) {
      x = x1;
      y = y1;
      z = z1;
    } else {
      double u = ((px-x1)*(x2-x1) + (py-y1)*(y2-y1) + (pz-z1)*(z2-z1)) / pd2;
      x = x1 + u * (x2 - x1);
      y = y1 + u * (y2 - y1);
      z = z1 + u * (z2 - z1);
      if (type != LINE && u < 0) {
        x = x1;
        y = y1;
```

```
          z = z1;
        }
        if (type == SEGMENT && u > 1.0) {
          x = x2;
          y = y2;
          z = z2;
        }
      }

      return (x-px)*(x-px) + (y-py)*(y-py) + (z-pz)*(z-pz);
    }

    public static double ptLineDist(double x1, double y1, double z1,
        double x2, double y2, double z2, double px, double py, double pz,
        int type) {
      return Math.sqrt(ptLineDistSq(x1, y1, z1, x2, y2, z2, px, py, pz, type));
    }
  }
```

5 Numerical algorithms

5.1 Number theory (modular, Chinese remainder, linear Diophantine)

```
// This is a collection of useful code for solving problems that
// involve modular linear equations.  Note that all of the
// algorithms described here work on nonnegative integers.

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

typedef vector<int> VI;
typedef pair<int, int> PII;
// return a % b (positive value)
int mod(int a, int b) {
    return ((a % b) + b) % b;
}
// computes gcd(a,b)
int gcd(int a, int b) {
    while (b) {
        int t = a % b;
        a = b;
        b = t;
    }
    return a;
}
// computes lcm(a,b)
int lcm(int a, int b) {
    return a / gcd(a, b) * b;
}
// (a^b) mod m via successive squaring
int powermod(int a, int b, int m) {
    int ret = 1;
    while (b) {
        if (b & 1) ret = mod(ret * a, m);
        a = mod(a * a, m);
        b >>= 1;
    }
    return ret;
}
// Finds two integers $x$ and $y$, such that $ax+by=\gcd(a,b)$. If
// If $a$ and $b$ are coprime, then $x$ is the inverse of $a \pmod{b}$.
// Returns gcd(a, b)
ll extended_euclid(ll a, ll b, ll &x, ll &y) {
    if (!b) return x = 1, y = 0, a;
    ll d = euclid(b, a % b, y, x);
    return y -= a/b * x, d;
}
// finds all solutions to ax = b (mod n)
VI modular_linear_equation_solver(int a, int b, int n) {
    int x, y;
    VI ret;
    int g = extended_euclid(a, n, x, y);
    if (!(b % g)) {
        x = mod(x * (b / g), n);
        for (int i = 0; i < g; i++)
            ret.push_back(mod(x + i * (n / g), n));
    }
    return ret;
}
// computes b such that ab = 1 (mod n), returns -1 on failure
int mod_inverse(int a, int n) {
    int x, y;
    int g = extended_euclid(a, n, x, y);
    if (g > 1) return -1;
    return mod(x, n);
}
// compute mod inverse of all numbers up to n
vector<ll> precompute_inv_mod(int n, ll mod) {
    vector<ll> inv(n + 1);
    inv[1] = 1;
    for (int i = 2; i <= n; ++i) {
        inv[i] = mod - (mod / i) * inv[mod % i] % mod;
    }
    return inv;
}
// Chinese remainder theorem (special case): find z such that
// z % m1 = r1, z % m2 = r2.  Here, z is unique modulo M = lcm(m1, m2).
// Return (z, M).  On failure, M = -1.
PII chinese_remainder_theorem(int m1, int r1, int m2, int r2) {
    int s, t;
    int g = extended_euclid(m1, m2, s, t);
    if (r1 % g != r2 % g) return make_pair(0, -1);
    return make_pair(mod(s * r2 * m1 + t * r1 * m2, m1 * m2) / g, m1 * m2 / g);
}
// Chinese remainder theorem: find z such that
// z % m[i] = r[i] for all i.  Note that the solution is
// unique modulo M = lcm_i (m[i]).  Return (z, M). On
// failure, M = -1. Note that we do not require the a[i]'s
// to be relatively prime.
PII chinese_remainder_theorem(const VI &m, const VI &r) {
    PII ret = make_pair(r[0], m[0]);
    for (int i = 1; i < m.size(); i++) {
        ret = chinese_remainder_theorem(ret.second, ret.first, m[i], r[i]);
        if (ret.second == -1) break;
    }
    return ret;
}
// computes x and y such that ax + by = c
// returns whether the solution exists
bool linear_diophantine(int a, int b, int c, int &x, int &y) {
    if (!a && !b) {
        if (c) return false;
        x = 0; y = 0;
        return true;
    }
    if (!a) {
        if (c % b) return false;
        x = 0; y = c / b;
        return true;
    }
    if (!b) {
        if (c % a) return false;
        x = c / a; y = 0;
        return true;
    }
    int g = gcd(a, b);
    if (c % g) return false;
    x = c / g * mod_inverse(a / g, b / g);
    y = (c - a * x) / b;
    return true;
}
int main() {
    int x, y;
    int g = extended_euclid(14, 30, x, y);
    cout << g << " " << x << " " << y << endl; //2 -2 1
    VI sols = modular_linear_equation_solver(14, 30, 100);
    for (int i = 0; i < sols.size(); i++) cout << sols[i] << " "; // 95 451
    cout << endl;
    cout << mod_inverse(8, 9) << endl; // 8
    PII ret = chinese_remainder_theorem(VI({3, 5, 7}), VI({2, 3, 2}));
    cout << ret.first << " " << ret.second << endl; // 23 105
    ret = chinese_remainder_theorem(VI({4, 6}), VI({3, 5}));
    cout << ret.first << " " << ret.second << endl; // 11 12
    if (!linear_diophantine(7, 2, 5, x, y)) cout << "ERROR" << endl;
    cout << x << " " << y << endl; // 5 -15
    return 0;
}
```

5.2 Modular Arithmetic

```
template <int MOD=998'244'353>
struct Modular {
    int value;
    static const int MOD_value = MOD;

    Modular(long long v = 0) { value = v % MOD; if (value < 0) value += MOD;}
    Modular(long long a, long long b) : value(0){ *this += a; *this /= b;}

    Modular& operator+=(Modular const& b) {value += b.value; if (value >= MOD)
         value -= MOD; return *this;}
    Modular& operator-=(Modular const& b) {value -= b.value; if (value < 0)
         value += MOD;return *this;}
    Modular& operator*=(Modular const& b) {value = (long long)value * b.value %
         MOD;return *this;}

    friend Modular mexp(Modular a, long long e) {
        Modular res = 1; while (e) { if (e&1) res *= a; a *= a; e >>= 1; }
        return res;
    }
    friend Modular inverse(Modular a) { return mexp(a, MOD - 2); }

    Modular& operator/=(Modular const& b) { return *this *= inverse(b); }
    friend Modular operator+(Modular a, Modular const b) { return a += b; }
    friend Modular operator-(Modular a, Modular const b) { return a -= b; }
    friend Modular operator-(Modular const a) { return 0 - a; }
    friend Modular operator*(Modular a, Modular const b) { return a *= b; }
    friend Modular operator/(Modular a, Modular const b) { return a /= b; }
    friend std::ostream& operator<<(std::ostream& os, Modular const& a) {return
         os << a.value;}
    friend bool operator==(Modular const& a, Modular const& b) {return a.value
         == b.value;}
    friend bool operator!=(Modular const& a, Modular const& b) {return a.value
         != b.value;}
};
// Chained Multiplication or Successive Simple Multiplication
Modular<998244353> a=1, m=123456789;
a *= m * m * m; // a = 519994069
// Inverse
a=inverse(m) // a=25170271
// fractions
Modular<> frac=(1,2); // frac=1*2^(-1) % 998244353 = 499122177
// Modular exponentiation
Modular<> power(2);
power=mexp(power,500); // power = 616118644
```

5.3 Systems of linear equations, matrix inverse, determinant

```
// Gauss-Jordan elimination with full pivoting.
//
// Uses:
//   (1) solving systems of linear equations (AX=B)
//   (2) inverting matrices (AX=I)
//   (3) computing determinants of square matrices
//
// Running time: O(n^3)
//
// INPUT:    a[][] = an nxn matrix
//           b[][] = an nxm matrix
//
// OUTPUT:   X      = an nxm matrix (stored in b[][])
//           A^{-1} = an nxn matrix (stored in a[][])
//           returns determinant of a[][]

#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

const double EPS = 1e-10;
typedef vector<int> VI;
typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;
T GaussJordan(VVT &a, VVT &b) {
    const int n = a.size();
    const int m = b[0].size();
    VI irow(n), icol(n), ipiv(n);
    T det = 1;

    for (int i = 0; i < n; i++) {
        int pj = -1, pk = -1;
        for (int j = 0; j < n; j++)
            if (!ipiv[j])
                for (int k = 0; k < n; k++)
                    if (!ipiv[k])
                        if (pj == -1 || fabs(a[j][k]) > fabs(a[pj][pk])) {
                            pj = j;
                            pk = k;
                        }
        if (fabs(a[pj][pk]) < EPS) {
            cerr << "Matrix is singular." << endl;
            exit(0);
        }
        ipiv[pk]++;
        swap(a[pj], a[pk]);
        swap(b[pj], b[pk]);
        if (pj != pk) det *= -1;
        irow[i] = pj;
        icol[i] = pk;
        T c = 1.0 / a[pk][pk];
        det *= a[pk][pk];
        a[pk][pk] = 1.0;
        for (int p = 0; p < n; p++) a[pk][p] *= c;
        for (int p = 0; p < m; p++) b[pk][p] *= c;
        for (int p = 0; p < n; p++)
            if (p != pk) {
                c = a[p][pk];
                a[p][pk] = 0;
                for (int q = 0; q < n; q++) a[p][q] -= a[pk][q] * c;
                for (int q = 0; q < m; q++) b[p][q] -= b[pk][q] * c;
            }
    }

    for (int p = n - 1; p >= 0; p--)
        if (irow[p] != icol[p]) {
            for (int k = 0; k < n; k++) swap(a[k][irow[p]], a[k][icol[p]]);
        }

    return det;
}
int main() {
    const int n = 4;
    const int m = 2;
    double A[n][n] = {{1, 2, 3, 4},
                      {1, 0, 1, 0},
                      {5, 3, 2, 4},
                      {6, 1, 4, 6}};
    double B[n][m] = {{1, 2},
                      {4, 3},
                      {5, 6},
                      {8, 7}};
    VVT a(n), b(n);
    for (int i = 0; i < n; i++) {
        a[i] = VT(A[i], A[i] + n);
        b[i] = VT(B[i], B[i] + m);
    }

    double det = GaussJordan(a, b);

    // expected: 60
    cout << "Determinant: " << det << endl;

    // expected: -0.233333 0.166667 0.133333 0.0666667
    //           0.166667 0.166667 0.333333 -0.333333
    //           0.233333 0.833333 -0.133333 -0.0666667
    //           0.05 -0.75 -0.1 0.2
    cout << "Inverse: " << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
            cout << a[i][j] << ' ';
        cout << endl;
    }

    // expected: 1.63333 1.3
    //           -0.166667 0.5
    //           2.36667 1.7
    //           -1.85 -1.35
    cout << "Solution: " << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++)
            cout << b[i][j] << ' ';
        cout << endl;
    }
}
```

5.4 Reduced row echelon form, matrix rank

```
// Reduced row echelon form via Gauss-Jordan elimination
// with partial pivoting.  This can be used for computing
// the rank of a matrix.
//
// Running time: O(n^3)
//
// INPUT:    a[][] = an nxm matrix
//
// OUTPUT:   rref[][] = an nxm matrix (stored in a[][])
//           returns rank of a[][]

#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

const double EPSILON = 1e-10;

typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

int rref(VVT &a) {
    int n = a.size();
    int m = a[0].size();
    int r = 0;
    for (int c = 0; c < m && r < n; c++) {
        int j = r;
        for (int i = r + 1; i < n; i++)
            if (fabs(a[i][c]) > fabs(a[j][c])) j = i;
        if (fabs(a[j][c]) < EPSILON) continue;
        swap(a[j], a[r]);

        T s = 1.0 / a[r][c];
        for (int j = 0; j < m; j++) a[r][j] *= s;
        for (int i = 0; i < n; i++)
            if (i != r) {
                T t = a[i][c];
                for (int j = 0; j < m; j++) a[i][j] -= t * a[r][j];
            }
        r++;
    }
    return r;
}

int main() {
    const int n = 5, m = 4;
    double A[n][m] = {
            {16, 2,  3,  13},
            {5,  11, 10, 8},
            {9,  7,  6,  12},
            {4,  14, 15, 1},
            {13, 21, 21, 13}};
    VVT a(n);
    for (int i = 0; i < n; i++)
        a[i] = VT(A[i], A[i] + m);

    int rank = rref(a);

    cout << "Rank: " << rank << endl; // 3

    // expected: 1 0 0 1
    //           0 1 0 3
    //           0 0 1 -3
    //          0 0 0 3.10862e-15
    //          0 0 0 2.22045e-15
    cout << "rref: " << endl;
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 4; j++)
            cout << a[i][j] << ' ';
        cout << endl;
    }
}
```

5.5 Fast Fourier transform

```
#include <cstdio>
#include <cmath>

struct cpx {
    cpx() {}
    cpx(double aa) : a(aa), b(0) {}
    cpx(double aa, double bb) : a(aa), b(bb) {}
    double a, b;
    double modsq(void) const {
        return a * a + b * b;
    }
    cpx bar(void) const {
        return cpx(a, -b);
    }
};

cpx operator+(cpx a, cpx b) {
    return cpx(a.a + b.a, a.b + b.b);
}
cpx operator*(cpx a, cpx b) {
    return cpx(a.a * b.a - a.b * b.b, a.a * b.b + a.b * b.a);
}
cpx operator/(cpx a, cpx b) {
    cpx r = a * b.bar();
    return cpx(r.a / b.modsq(), r.b / b.modsq());
}
cpx EXP(double theta) {
    return cpx(cos(theta), sin(theta));
}
const double two_pi = 4 * acos(0);
// in:     input array
// out:    output array
// step:   {SET TO 1} (used internally)
// size:   length of the input/output {MUST BE A POWER OF 2}
// dir:    either plus or minus one (direction of the FFT, 1 is first)
// RESULT: out[k] = \sum_{j=0}^{size - 1} in[j] * exp(dir * 2pi * i * j * k /
    size)
void FFT(cpx *in, cpx *out, int step, int size, int dir) {
    if (size < 1) return;
    if (size == 1) {
        out[0] = in[0];
        return;
    }
    FFT(in, out, step * 2, size / 2, dir);
    FFT(in + step, out + size / 2, step * 2, size / 2, dir);
    for (int i = 0; i < size / 2; i++) {
        cpx even = out[i];
        cpx odd = out[i + size / 2];
        out[i] = even + EXP(dir * two_pi * i / size) * odd;
        out[i + size / 2] = even + EXP(dir * two_pi * (i + size / 2) / size) *
            odd;
    }
}

// Usage:
// f[0...N-1] and g[0..N-1] are numbers
// Want to compute the convolution h, defined by
// h[n] = sum of f[k]g[n-k] (k = 0, ..., N-1).
// Here, the index is cyclic; f[-1] = f[N-1], f[-2] = f[N-2], etc.
// Let F[0...N-1] be FFT(f), and similarly, define G and H.
// The convolution theorem says H[n] = F[n]G[n] (element-wise product).
// To compute h[] in O(N log N) time, do the following:
//   1. Compute F and G (pass dir = 1 as the argument).
//   2. Get H by element-wise multiplying F and G.
//   3. Get h by taking the inverse FFT (use dir = -1 as the argument)
//      and *dividing by N*. DO NOT FORGET THIS SCALING FACTOR.

int main(void) {
    printf("If rows come in identical pairs, then everything works.\n");

    cpx a[8] = {0, 1, cpx(1, 3), cpx(0, 5), 1, 0, 2, 0};
    cpx b[8] = {1, cpx(0, -2), cpx(0, 1), 3, -1, -3, 1, -2};
    cpx A[8];
    cpx B[8];
    FFT(a, A, 1, 8, 1);
    FFT(b, B, 1, 8, 1);

    for (int i = 0; i < 8; i++) {
        printf("%7.2lf%7.2lf", A[i].a, A[i].b);
    }
    printf("\n");
    for (int i = 0; i < 8; i++) {
        cpx Ai(0, 0);
        for (int j = 0; j < 8; j++) {
            Ai = Ai + a[j] * EXP(j * i * two_pi / 8);
        }
        printf("%7.2lf%7.2lf", Ai.a, Ai.b);
    }
    printf("\n");

    cpx AB[8];
    for (int i = 0; i < 8; i++)
        AB[i] = A[i] * B[i];
    cpx aconvb[8];
    FFT(AB, aconvb, 1, 8, -1);
    for (int i = 0; i < 8; i++)
        aconvb[i] = aconvb[i] / 8;
    for (int i = 0; i < 8; i++) {
        printf("%7.2lf%7.2lf", aconvb[i].a, aconvb[i].b);
    }
    printf("\n");
    for (int i = 0; i < 8; i++) {
        cpx aconvbi(0, 0);
        for (int j = 0; j < 8; j++) {
            aconvbi = aconvbi + a[j] * b[(8 + i - j) % 8];
        }
        printf("%7.2lf%7.2lf", aconvbi.a, aconvbi.b);
    }
    printf("\n");

    return 0;
}
```

5.6 Euler's Toitent Function

```
/**
 * Author: Hakan Terelius
 * Date: 2009-09-25
 * License: CC0
 * Description: Precompute the number of positive integers coprime to N up to a
     given limit.
 * - The sum phi(d) for all divisors d of n is equal to n.
 * - The sum of all positive numbers less than n that are coprime to n is n phi(
     n) / 2 (n > 1)
 * - For any a, n coprime, a^(phi(n)) = 1 mod n
 * - Specifically, for any prime p, any number a, a^{p-1} = 1 mod p
 * Status: Tested
 */
#pragma once

const int LIM = 5000000;
int phi[LIM];

void calculatePhi() {
    rep(i,0,LIM) phi[i] = i&1 ? i : i/2;
    for (int i = 3; i < LIM; i += 2) if(phi[i] == i)
            for (int j = i; j < LIM; j += i) phi[j] -= phi[j] / i;
}
```

5.7 Partitions

```
# include <bits/stdc++.h>
using namespace std;
typedef vector<int> vi;
// Ways to write n as a sum of positive numbers.
// parition(4)=5 because 4 = 3 + 1 = 2 + 2 = 2 + 1 + 1 = 1 + 1 + 1 + 1
int partition(int n) {
    if(n==0) return 1;
    assert(n > 0);
    vi dp = vi(n + 1);
    dp[0] = 1;
    for (int i = 1; i <= n; i++) {
        for (int j = 1, r = 1; i - (3 * j * j - j) / 2 >= 0; j++, r *= -1) {
            dp[i] += dp[i - (3 * j * j - j) / 2] * r;
            if (i - (3 * j * j + j) / 2 >= 0) {
                dp[i] += dp[i - (3 * j * j + j) / 2] * r;
            }
        }
    }
    return dp[n];
}
int main() {
    // 0 1, 1 1, 2 2, 3 3, 4 5, 5 7, 6 11, 7 15, 8 22, 9 30, 10 42
    // 11 56, 12 77, 13 101, 14 135, 15 176, 16 231, 17 297
    for (int i = 0; i <= 17; ++i) {
        cout << i << " " << partition(i) << ", ";
    }
    return 0;
}
```

6 Graph algorithms

6.1 Bellman-Ford shortest paths with negative edge weights (C++)

```
// This function runs the Bellman-Ford algorithm for single source
// shortest paths with negative edge weights.  The function returns
// false if a negative weight cycle is detected.  Otherwise, the
// function returns true and dist[i] is the length of the shortest
// path from start to i.
//
// Running time: O(|V|^3)
//
//   INPUT:   start, w[i][j] = cost of edge from i to j
//   OUTPUT:  dist[i] = min weight path from start to i
//            prev[i] = previous node on the best path from the
//                      start node

#include <iostream>
#include <queue>
#include <cmath>
#include <vector>

using namespace std;

typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;
typedef vector<int> VI;
typedef vector<VI> VVI;
bool BellmanFord(const VVT &w, VT &dist, VI &prev, int start) {
    int n = w.size();
    prev = VI(n, -1);
    dist = VT(n, 1000000000);
    dist[start] = 0;

    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (dist[j] > dist[i] + w[i][j]) {
                    if (k == n - 1) return false;
                    dist[j] = dist[i] + w[i][j];
                    prev[j] = i;
                }
            }
        }
    }

    return true;
}
```

6.2 Topological sort (C++)

```
// This function uses performs a non-recursive topological sort.
//
// Running time: O(|V|^2).  If you use adjacency lists (vector<map<int> >),
//               the running time is reduced to O(|E|).
//
//   INPUT:   w[i][j] = 1 if i should come before j, 0 otherwise
//   OUTPUT:  a permutation of 0,...,n-1 (stored in a vector)
//            which represents an ordering of the nodes which
//            is consistent with w
//
// If no ordering is possible, false is returned.

#include <iostream>
#include <queue>
#include <vector>

using namespace std;

typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

typedef vector<int> VI;
typedef vector<VI> VVI;

bool TopologicalSort(const VVI &w, VI &order) {
    int n = w.size();
    VI parents(n);
    queue<int> q;
    order.clear();

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
            if (w[j][i]) parents[i]++;
        if (parents[i] == 0) q.push(i);
    }

    while (q.size() > 0) {
        int i = q.front();
        q.pop();
        order.push_back(i);
        for (int j = 0; j < n; j++)
            if (w[i][j]) {
                parents[j]--;
                if (parents[j] == 0) q.push(j);
            }
    }

    return (order.size() == n);
}
```

6.3 Fast Dijkstra's algorithm

```
// Implementation of Dijkstra's algorithm using adjacency lists
// and priority queue for efficiency.
//
// Running time: O(|E| log |V|)

#include <queue>
#include <cstdio>

using namespace std;
const int INF = 2000000000;
typedef pair<int, int> PII;
int main() {
    int N, s, t;
    scanf("%d%d%d", &N, &s, &t);
    vector<vector<PII> > edges(N);
    for (int i = 0; i < N; i++) {
        int M;
        scanf("%d", &M);
        for (int j = 0; j < M; j++) {
            int vertex, dist;
            scanf("%d%d", &vertex, &dist);
            edges[i].push_back(make_pair(dist, vertex)); // note order of
                 arguments here
        }
    }

    // use priority queue in which top element has the "smallest" priority
    priority_queue<PII, vector<PII>, greater<PII> > Q;
    vector<int> dist(N, INF), dad(N, -1);
    Q.push(make_pair(0, s));
    dist[s] = 0;
    while (!Q.empty()) {
        PII p = Q.top();
        Q.pop();
        int here = p.second;
        if (here == t) break;
        if (dist[here] != p.first) continue;

        for (vector<PII>::iterator it = edges[here].begin(); it != edges[here].
             end(); it++) {
            if (dist[here] + it->first < dist[it->second]) {
                dist[it->second] = dist[here] + it->first;
                dad[it->second] = here;
                Q.push(make_pair(dist[it->second], it->second));
            }
        }
    }

    printf("%d\n", dist[t]);
    if (dist[t] < INF)
        for (int i = t; i != -1; i = dad[i])
            printf("%d%c", i, (i == s ? '\n' : ' '));
    return 0;
}

/*
Sample input:
5 0 4
2 1 2 3 1
2 2 4 4 5
3 1 4 3 3 4 1
2 0 1 2 3
2 1 5 2 1

Expected:
5
4 2 3 0
*/
```

6.4 Strongly connected components

```
vi val, comp, z, cont;
int Time, ncomps;
// A function that will be called with the indicies of all elements
// in each component as the parameter once per component after running scc.
void f(vi node_inds) {};
int dfs(int j, vector<vi>& g) {
    int low = val[j] = ++Time, x; z.push_back(j);
    for (auto e : g[j]) if (comp[e] < 0)
            low = min(low, val[e] ?: dfs(e,g));

    if (low == val[j]) {
        do {
            x = z.back(); z.pop_back();
            comp[x] = ncomps;
            cont.push_back(x);
        } while (x != j);
        f(cont); cont.clear();
        ncomps++;
    }
    return val[j] = low;
}
void scc(vector<vi>& g) {
    int n = g.size();
    val.assign(n, 0); comp.assign(n, -1);
    Time = ncomps = 0;
    rep(i,0,n) if (comp[i] < 0) dfs(i, g);
}
```

6.5 Eulerian path

```
struct Edge;
typedef list<Edge>::iterator iter;

struct Edge {
    int next_vertex;
    iter reverse_edge;
    Edge(int next_vertex)
            : next_vertex(next_vertex) {}
};

const int max_vertices =;
int num_vertices;
list <Edge> adj[max_vertices];        // adjacency list

vector<int> path;
void find_path(int v) {
    while (adj[v].size() > 0) {
        int vn = adj[v].front().next_vertex;
        adj[vn].erase(adj[v].front().reverse_edge);
        adj[v].pop_front();
        find_path(vn);
    }
    path.push_back(v);
}
void add_edge(int a, int b) {
    adj[a].push_front(Edge(b));
    iter ita = adj[a].begin();
    adj[b].push_front(Edge(a));
    iter itb = adj[b].begin();
    ita->reverse_edge = itb;
    itb->reverse_edge = ita;
}
```

6.6 Minimum spanning trees

```
// This function runs Prim's algorithm for constructing minimum
// weight spanning trees.
//
// Running time: O(|V|^2)
//
//   INPUT:   w[i][j] = cost of edge from i to j
//
//            NOTE: Make sure that w[i][j] is nonnegative and
//            symmetric.  Missing edges should be given -1
//            weight.
//
//   OUTPUT:  edges = list of pair<int,int> in minimum spanning tree
//            return total weight of tree

#include <iostream>
#include <queue>
#include <cmath>
#include <vector>

using namespace std;

typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

typedef vector<int> VI;
typedef vector<VI> VVI;
typedef pair<int, int> PII;
typedef vector<PII> VPII;

T Prim(const VVT &w, VPII &edges) {
    int n = w.size();
    VI found(n);
    VI prev(n, -1);
    VT dist(n, 1000000000);
    int here = 0;
    dist[here] = 0;

    while (here != -1) {
        found[here] = true;
        int best = -1;
        for (int k = 0; k < n; k++)
            if (!found[k]) {
                if (w[here][k] != -1 && dist[k] > w[here][k]) {
                    dist[k] = w[here][k];
                    prev[k] = here;
                }
                if (best == -1 || dist[k] < dist[best]) best = k;
            }
        here = best;
    }

    T tot_weight = 0;
    for (int i = 0; i < n; i++)
        if (prev[i] != -1) {
            edges.push_back(make_pair(prev[i], i));
            tot_weight += w[prev[i]][i];
        }
    return tot_weight;
}

int main() {
    int ww[5][5] = {
            {0,   400, 400, 300, 600},
            {400, 0,   3,   -1,  7},
            {400, 3,   0,   2,   0},
            {300, -1,  2,   0,   5},
            {600, 7,   0,   5,   0}
    };
    VVT w(5, VT(5));
    for (int i = 0; i < 5; i++)
        for (int j = 0; j < 5; j++)
            w[i][j] = ww[i][j];


    VPII edges;
    cout << Prim(w, edges) << endl; // 305
    for (int i = 0; i < edges.size(); i++)
        cout << edges[i].first << " " << edges[i].second << endl;
    //          2 1
    //          3 2
    //          0 3
    //          2 4
}
```

7 Strings

7.1 AhoCorasick

```
#define foreach(x, v) for (typeof (v).begin() x=(v).begin(); x !=(v).end(); ++x)
#define For(i, a, b) for (int i=(a); i<(b); ++i)
#define D(x) cout << #x " is " << x << endl
const int MAXS = 6 * 50 + 10; // Max number of states in the matching machine.
// Should be equal to the sum of the length of all keywords.
const int MAXC = 26; // Number of characters in the alphabet.
int out[MAXS]; // Output for each state, as a bitwise mask.
int f[MAXS]; // Failure function
int g[MAXS][MAXC]; // Goto function, or -1 if fail.
int buildMatchingMachine(const vector <string> &words, char lowestChar = 'a',
                          char highestChar = 'z') {
    memset(out, 0, sizeof out);
    memset(f, -1, sizeof f);
    memset(g, -1, sizeof g);
    int states = 1; // Initially, we just have the 0 state
    for (int i = 0; i < words.size(); ++i) {
        const string &keyword = words[i];
        int currentState = 0;
        for (int j = 0; j < keyword.size(); ++j) {
            int c = keyword[j] - lowestChar;
            if (g[currentState][c] == -1) { // Allocate a new node
                g[currentState][c] = states++;
            }
            currentState = g[currentState][c];
        }
        out[currentState] |= (1 << i); // There's a match of keywords[i] at node
              currentState.
    }
    // State 0 should have an outgoing edge for all characters.
    for (int c = 0; c < MAXC; ++c) {
        if (g[0][c] == -1) {
            g[0][c] = 0;
        }
    }

    // Now, let's build the failure function
    queue<int> q;
    for (int c = 0; c <= highestChar - lowestChar; ++c) { // Iterate over every
         possible input
        // All nodes s of depth 1 have f[s] = 0
        if (g[0][c] != -1 and g[0][c] != 0) {
            f[g[0][c]] = 0;
            q.push(g[0][c]);
        }
    }
    while (q.size()) {
        int state = q.front();
        q.pop();
        for (int c = 0; c <= highestChar - lowestChar; ++c) {
            if (g[state][c] != -1) {
                int failure = f[state];
                while (g[failure][c] == -1) {
                    failure = f[failure];
                }
                failure = g[failure][c];
                f[g[state][c]] = failure;
                out[g[state][c]] |= out[failure]; // Merge out values
                q.push(g[state][c]);
            }
        }
    }

    return states;
}
int findNextState(int currentState, char nextInput, char lowestChar = 'a') {
    int answer = currentState;
    int c = nextInput - lowestChar;
    while (g[answer][c] == -1)
        answer = f[answer];
    return g[answer][c];
}
int main() {
    vector <string> keywords;
    keywords.push_back("he"); keywords.push_back("she");
    keywords.push_back("hers"); keywords.push_back("his");
    string text = "ahishers";
    buildMatchingMachine(keywords, 'a', 'z');
    int currentState = 0;
    for (int i = 0; i < text.size(); ++i) {
        currentState = findNextState(currentState, text[i], 'a');
        if (out[currentState] == 0)
            continue; // Nothing new, let's move on to the next character.
        for (int j = 0; j < keywords.size(); ++j) {
            if (out[currentState] & (1 << j)) { // Matched keywords[j]
                cout << "Keyword " << keywords[j] << " appears from "
                     << i - keywords[j].size() + 1 << " to " << i << endl;
            }
        }
    }
    return 0;
}
```

7.2 Longest increasing subsequence

```
// Given a list of numbers of length n, this routine extracts a
// longest increasing subsequence.
//
// Running time: O(n log n)
//
//   INPUT: a vector of integers
//   OUTPUT: a vector containing the longest increasing subsequence

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

typedef vector<int> VI;
typedef pair<int, int> PII;
typedef vector<PII> VPII;

#define STRICTLY_INCREASNG

VI LongestIncreasingSubsequence(VI v) {
    VPII best;
    VI dad(v.size(), -1);

    for (int i = 0; i < v.size(); i++) {
#ifdef STRICTLY_INCREASNG
        PII item = make_pair(v[i], 0);
        VPII::iterator it = lower_bound(best.begin(), best.end(), item);
        item.second = i;
#else
        PII item = make_pair(v[i], i);
        VPII::iterator it = upper_bound(best.begin(), best.end(), item);
#endif
        if (it == best.end()) {
            dad[i] = (best.size() == 0 ? -1 : best.back().second);
            best.push_back(item);
        } else {
            dad[i] = it == best.begin() ? -1 : prev(it)->second;
            *it = item;
        }
    }

    VI ret;
    for (int i = best.back().second; i >= 0; i = dad[i])
        ret.push_back(v[i]);
    reverse(ret.begin(), ret.end());
    return ret;
}
```

7.3 Longest common subsequence

```
/*
Calculates the length of the longest common subsequence of two vectors.
Backtracks to find a single subsequence or all subsequences. Runs in
O(m*n) time except for finding all longest common subsequences, which
may be slow depending on how many there are.
*/

#include <iostream>
#include <vector>
#include <set>
#include <algorithm>

using namespace std;

typedef int T;
typedef vector<T> VT;
typedef vector<VT> VVT;
typedef vector<int> VI;
typedef vector<VI> VVI;
void backtrack(VVI &dp, VT &res, VT &A, VT &B, int i, int j) {
    if (!i || !j) return;
    if (A[i - 1] == B[j - 1]) {
        res.push_back(A[i - 1]);
        backtrack(dp, res, A, B, i - 1, j - 1);
    }
    else {
        if (dp[i][j - 1] >= dp[i - 1][j]) backtrack(dp, res, A, B, i, j - 1);
        else backtrack(dp, res, A, B, i - 1, j);
    }
}
void backtrackall(VVI &dp, set<VT> &res, VT &A, VT &B, int i, int j) {
    if (!i || !j) {
        res.insert(VI());
        return;
    }
    if (A[i - 1] == B[j - 1]) {
        set<VT> tempres;
        backtrackall(dp, tempres, A, B, i - 1, j - 1);
        for (set<VT>::iterator it = tempres.begin(); it != tempres.end(); it++)
            {
            VT temp = *it;
            temp.push_back(A[i - 1]);
            res.insert(temp);
        }
    } else {
        if (dp[i][j - 1] >= dp[i - 1][j]) backtrackall(dp, res, A, B, i, j - 1);
        if (dp[i][j - 1] <= dp[i - 1][j]) backtrackall(dp, res, A, B, i - 1, j);
    }
}
VT LCS(VT &A, VT &B) {
    VVI dp;
    int n = A.size(), m = B.size();
    dp.resize(n + 1);
    for (int i = 0; i <= n; i++) dp[i].resize(m + 1, 0);

    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++) {
            if (A[i - 1] == B[j - 1]) dp[i][j] = dp[i - 1][j - 1] + 1;
            else dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
        }

    VT res;
    backtrack(dp, res, A, B, n, m);
    reverse(res.begin(), res.end());
    return res;
}
set<VT> LCSall(VT &A, VT &B) {
    VVI dp;
    int n = A.size(), m = B.size();
    dp.resize(n + 1);
    for (int i = 0; i <= n; i++) dp[i].resize(m + 1, 0);
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++) {
            if (A[i - 1] == B[j - 1]) dp[i][j] = dp[i - 1][j - 1] + 1;
            else dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
        }
    set<VT> res;
    backtrackall(dp, res, A, B, n, m);
    return res;
}
int main() {
    int a[] = {0, 5, 5, 2, 1, 4, 2, 3}, b[] = {5, 2, 4, 3, 2, 1, 2, 1, 3};
    VI A = VI(a, a + 8), B = VI(b, b + 9);
    VI C = LCS(A, B);

    for (int i = 0; i < C.size(); i++) cout << C[i] << " ";
    cout << endl << endl;

    set<VI> D = LCSall(A, B);
    for (set<VI>::iterator it = D.begin(); it != D.end(); it++) {
        for (int i = 0; i < (*it).size(); i++) cout << (*it)[i] << " ";
        cout << endl;
    }
}
```

7.4 Knuth-Morris-Pratt

```
/*
Finds all occurrences of the pattern string p within the
text string t. Running time is O(n + m), where n and m
are the lengths of p and t, respecitvely.
*/

#include <iostream>
#include <string>
#include <vector>

using namespace std;

typedef vector<int> VI;
void buildPi(string &p, VI &pi) {
    pi = VI(p.length());
    int k = -2;
    for (int i = 0; i < p.length(); i++) {
        while (k >= -1 && p[k + 1] != p[i])
            k = (k == -1) ? -2 : pi[k];
        pi[i] = ++k;
    }
}
int KMP(string &t, string &p) {
    VI pi;
    buildPi(p, pi);
    int k = -1;
    for (int i = 0; i < t.length(); i++) {
        while (k >= -1 && p[k + 1] != t[i])
            k = (k == -1) ? -2 : pi[k];
        k++;
        if (k == p.length() - 1) {
            // p matches t[i-m+1, ..., i]
            cout << "matched at index " << i - k << ": ";
            cout << t.substr(i - k, p.length()) << endl;
            k = (k == -1) ? -2 : pi[k];
        }
    }
    return 0;
}
```

```
int main() {
    string a = "AABAACAADAABAABA", b = "AABA";
    KMP(a, b); // expected matches at: 0, 9, 12
    return 0;
}
```

7.5 Longest Common Prefix

```
/**
 * Author: chilli
 * License: CC0
 * Description: z[x] computes the length of the longest common prefix of s[i:]
      and s, except z[0] = 0. (abacaba -> 0010301)
 * Time: O(n)
 * Status: stress-tested
 */
#pragma once

vi Z(string S) {
    vi z(sz(S));
    int l = -1, r = -1;
    rep(i,1,sz(S)) {
        z[i] = i >= r ? 0 : min(r - i, z[i - l]);
        while (i + z[i] < sz(S) && S[i + z[i]] == S[z[i]])
            z[i]++;
        if (i + z[i] > r)
            l = i, r = i + z[i];
    }
    return z;
}
```

7.6 Palindromes

```
/**
 * Author: User adamant on CodeForces
 * Source: http://codeforces.com/blog/entry/12143
 * Description: For each position in a string, computes p[0][i] = half length of
 *  longest even palindrome around pos i, p[1][i] = longest odd (half rounded
      down).
 * Time: O(N)
 * Status: Stress-tested
 */
array<vi, 2> manacher(const string& s) {
    int n = sz(s);
    array<vi,2> p = {vi(n+1), vi(n)};
    rep(z,0,2) for (int i=0,l=0,r=0; i < n; i++) {
            int t = r-i+!z;
            if (i<r) p[z][i] = min(t, p[z][l+t]);
            int L = i-p[z][i], R = i+p[z][i]-!z;
            while (L>=1 && R+1<n && s[L-1] == s[R+1])
                p[z][i]++, L--, R++;
            if (R>r) l=L, r=R;
        }
    return p;
}
```

8 Miscellaneous

8.1 Prime numbers

```
# include <bits/stdc++.h>
using namespace std;
#define EPS 1e-7
typedef long long LL;

bool IsPrime(LL x) {
    if (x <= 1) return false;
    if (x <= 3) return true;
    if (!(x % 2) || !(x % 3)) return false;
    LL s = (LL) (sqrt((double) (x)) + EPS);
    for (LL i = 5; i <= s; i += 6) {
        if (!(x % i) || !(x % (i + 2))) return false;
    }
    return true;
}

// Factor every number up until n in O(n) time.
// minFact[i] = the minimum factor of i higher than 1. minFact[0] = minFact[1] =
      0
// primes[i] = the ith prime.
vector<int> factorAll(int n) {
    vector<int> primes(0);
    vector<int> minFact(n + 1);
    for (int i = 2; i <= n; i++) {
        if (minFact[i] == 0) {
            primes.push_back(i);
            minFact[i] = i;
        }
        for (int j = 0; j < primes.size() && primes[j] <= minFact[i] && i *
             primes[j] <= n; ++j) {
            minFact[i * primes[j]] = primes[j];
        }
    }
    return primes;
}
// Primes close to 1e9: 999'999'937, 1'000'000'007, 1'000'000'009
```

8.2 Binary Search

```
// This code is guaranteed to work in the min number of ops
// for any MAX that fits in an ll.
ll MAX = 1LL << 62;
// Binary search integers in the range [0, MAX] (or higher)
// for the last element satisfying condition.
ll lo = 0
for (ll j = 1LL << (ll) (log2(MAX)); j != 0; j >>= 1) {
    if (condition(lo + j)) {
        lo += j;
    }
}
// Binary search integers in the range (1, MAX] (or higher)
// for the first element satisfying condition.
ll hi = 1LL << (ll) (log2(MAX) + 1);
for (ll j = 1LL << (ll) (log2(MAX)); j != 0; j >>= 1) {
    if (condition(hi - j)) {
        hi -= j;
    }
}
// Search list[lo:hi], targets occur at list[bisect_left:bisect_right]
while (lo < hi) { // bisect_right
    mid = lo + (hi - lo) / 2;
    if (target < list[mid]) { hi = mid; }
    else { lo = mid + 1; }
} return lo;
while (lo < hi) { // bisect_left
    mid = lo + (hi - lo) / 2;
    if (list[mid] < target) { lo = mid + 1; }
    else { hi = mid; }
} return lo;
```

8.3 Latitude/longitude

```
/*
Converts from rectangular coordinates to latitude/longitude and vice
versa. Uses degrees (not radians).
*/

#include <iostream>
#include <cmath>

using namespace std;

struct ll {
    double r, lat, lon;
};
struct rect {
    double x, y, z;
};
ll convert(rect &P) {
    ll Q;
    Q.r = sqrt(P.x * P.x + P.y * P.y + P.z * P.z);
    Q.lat = 180 / M_PI * asin(P.z / Q.r);
    Q.lon = 180 / M_PI * acos(P.x / sqrt(P.x * P.x + P.y * P.y));

    return Q;
}
rect convert(ll &Q) {
    rect P;
    P.x = Q.r * cos(Q.lon * M_PI / 180) * cos(Q.lat * M_PI / 180);
    P.y = Q.r * sin(Q.lon * M_PI / 180) * cos(Q.lat * M_PI / 180);
    P.z = Q.r * sin(Q.lat * M_PI / 180);

    return P;
}
int main() {
    rect A;
    ll B;

    A.x = -1.0;
    A.y = 2.0;
    A.z = -3.0;

    B = convert(A);
    cout << B.r << " " << B.lat << " " << B.lon << endl;

    A = convert(B);
    cout << A.x << " " << A.y << " " << A.z << endl;
}
```

8.4 Hilbert curve for Mo's Algorithm

```
struct Query {
    int l, r, idx;
    int64_t ord;
    inline void calcOrder() {
        ord = hilbertOrder(l, r, 21, 0);
    }
};

inline bool operator<(const Query &a, const Query &b) {
    return a.ord < b.ord;
}
// contant time optimization to Mo's algorithm (~3x faster lol)
// https://codeforces.com/blog/entry/61203
inline int64_t hilbertOrder(int x, int y, int pow, int rotate) {
    if (pow == 0) {
        return 0;
    }
    int hpow = 1 << (pow - 1);
    int seg = (x < hpow) ? (
            (y < hpow) ? 0 : 3
    ) : (
                        (y < hpow) ? 1 : 2
            );
    seg = (seg + rotate) & 3;
    const int rotateDelta[4] = {3, 0, 0, 1};
    int nx = x & (x ^ hpow), ny = y & (y ^ hpow);
    int nrot = (rotate + rotateDelta[seg]) & 3;
    int64_t subSquareSize = int64_t(1) << (2 * pow - 2);
    int64_t ans = seg * subSquareSize;
    int64_t add = hilbertOrder(nx, ny, pow - 1, nrot);
    ans += (seg == 1 || seg == 2) ? add : (subSquareSize - add - 1);
    return ans;
}
```