Team Reference Document

Contents

1 Varia

1.1 Numbers

- Carmichael numbers: $b^n \equiv b \pmod{n}$: 561, 1105, 1729, 2465, 2821, 6601, 8911, 10585
- Catalan numbers: $C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!} = \prod_{k=2}^{n} \frac{n+k}{k}$: 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012
- Partition Function: $P(n) \sim \frac{1}{4n\sqrt{3}} e^{\pi\sqrt{\frac{2}{3}n}}$, $P(n,k) = P(n-1,k-1) + P(n-k,k)$: 1, 1, 2, 3, 5, 7, 11, 15, 22, 30, 42, 56, 77, 101, 135, 176, 231, 297, 385, 490, 627, 792, 1002, 1255
- Primes: 1000000007, 1000000009, 1000000021, 1000000033, 1000000087, 1000000093

1.2 Template

```cpp
// -*- compile-command: "g++ -Wall -Wextra x.cpp -o x && ./x <test.in" -*-
#include <bits/stdc++.h>

using namespace std;
#define PB push_back
#define MP make_pair
#define sz(v) ((v).size())
#define forn(i,n) for(int i=0;i<(n);i++)
#define forv(i,v) forn(i,sz(v))
typedef long long in;
typedef unsigned long long int llu;

typedef int real_int;
#define int in

real_int main(){
  std::ios::sync_with_stdio(false); // remove this if mixing with printf/scanf
  std::cin.tie(0);
  std::cout << std::setprecision(9); // Set floating point precision
// == DON'T NEED THESE LINES ON ACM ICPC WORLD FINALS ====
#error CHANGE NAME!
#ifndef STDINSTDOUT
  freopen("bla.in","r",stdin);
  freopen("bla.out","w",stdout);
#endif
// == END OF DON'T NEED THESE LINES ON ACM ICPC WORLD FINALS ====
  return 0;
}
```

1.3 Compile script

```
#!/bin/bash
shopt -s nullglob

filename="A.cpp"
progname="${filename%%.*}.prog"

echo "Compiling ${filename}"
if g++ ${filename} -o ${progname} -Wall -Wextra -D_GLIBCXX_DEBUG
then
  for f in *.in; do
    ansfile="${f%%.*}.ans"
    outfile="${f%%.*}.out"
    echo "input ${f}"
    if "./${progname}"<"${f}">"${outfile}"; then
      diff -sy "${ansfile}" "${outfile}"
    fi
  done
fi
```

2 Max Flow

2.1 Sparse max-flow

```
// Adjacency list implementation of a basic flow algorithm with some
↪  heuristic
// (not Dinic).  This is fast enough in practice.
//
// Running time:
//     O(|V|^2 |E|)
//
// INPUT:
//     - graph, constructed using AddEdge()
//     - source
//     - sink
//
// OUTPUT:
//     - maximum flow value
//     - To obtain the actual flow values, look at all edges with
//       capacity > 0 (zero capacity edges are residual edges).

const int INF = 2000000000;

struct Edge {
  int from, to, cap, flow, index;  // index points to the reverse Edge as
↪  G[to][index]
  Edge(int from, int to, int cap, int flow, int index) :
    from(from), to(to), cap(cap), flow(flow), index(index) {}
};

struct MaxFlow {
  int N;
  vector<vector<Edge> > G;
  vector<Edge *> dad;
  vector<int> Q;

  MaxFlow(int N) : N(N), G(N), dad(N), Q(N) {}

  void AddEdge(int from, int to, int cap) {
    G[from].push_back(Edge(from, to, cap, 0, G[to].size()));
    if (from == to) G[from].back().index++; // handle loops: reverse
↪  edge is next
    G[to].push_back(Edge(to, from, 0, 0, G[from].size() - 1));
  }

  long long BlockingFlow(int s, int t) {
    fill(dad.begin(), dad.end(), (Edge *) NULL); // clear BFS tree
    dad[s] = &G[0][0] - 1;

    int head = 0, tail = 0;
    Q[tail++] = s;
    while (head < tail) { // non-empty BFS queue
      int x = Q[head++];
      for (int i = 0; i < G[x].size(); i++) {
        Edge &e = G[x][i];
        if (!dad[e.to] && e.cap - e.flow > 0) { // edge to undiscovered
↪  vtx with free capacity
          dad[e.to] = &G[x][i];
          Q[tail++] = e.to;
        }
      }
    }
    if (!dad[t]) return 0; // stop if t is unreachable
  // collect flows in the BFS tree
    long long totflow = 0;
    for (int i = 0; i < G[t].size(); i++) { // go through all edge to the
↪  sink
      Edge *start = &G[G[t][i].to][G[t][i].index];
      int amt = INF; // trace back the path from this t-edge back to s
↪  and get min. cap. along the way
      for (Edge *e = start; amt && e != dad[s]; e = dad[e->from]) {
        if (!e) { amt = 0; break; }
        amt = min(amt, e->cap - e->flow);
      }
      if (amt == 0) continue; // update the residual graph
      for (Edge *e = start; amt && e != dad[s]; e = dad[e->from]) {
        e->flow += amt;
        G[e->to][e->index].flow -= amt; // reverse edge gets negative
↪  flow
      }
      totflow += amt;
    }
    return totflow;
  }

  long long GetMaxFlow(int s, int t) {
    long long totflow = 0;
    while (long long flow = BlockingFlow(s, t))
      totflow += flow;
    return totflow;
  }
}; // reset the flow values of all edges to zero before calling again
```

2.2 Dinic's max-flow

```
// Adjacency list implementation of Dinic's blocking flow algorithm.
//
// Running time:
//     O(|V|^2 |E|)                          -- arbitrary networks
//     O(min{|V|^(2/3), |E|^(1/2)} |E|)  -- networks with only unit
↪  capacities
//     O(|V|^(1/2) |E|)                      -- bipartite graph/unit networks
//
// See Hopcroft-Karp for a slightly faster implementation of bipartite
↪  matching
//
// Usage:
//    dinic mf(n);
//    mf.add_edge(from, to, capacity);          // directed edge
//    mf.add_undirected(from, to, capacity);    // undirected edge
//    auto maxflow = mf.max_flow(source, sink); // max flow from source
↪  to sink
//    int flow_at_v = mf.g[v][0].flow();        // flow through first
↪  edge of v
//                                              // if cap==0, then reverse
↪  edge

struct dinic {
  struct edge {              // note: cap is not needed, a small speed up
↪  can be
    int to, rcap, cap, rev; //        achieved by simply removing it.
    int flow() const { return cap - rcap; }
  };
  vector<vector<edge> > g;
  vector<int> dist, state;
  vector<edge*> path;

  dinic(int n) : g(n), dist(n), state(n), path(n) {}

  void add_edge(int a, int b, int cap, int rev_cap=0) {
    if (a == b) return;
    g[a].push_back({b, cap, cap, (int)g[b].size()});
    g[b].push_back({a, rev_cap, rev_cap, (int)g[a].size()-1});
  }
  // small speed up for undirected graphs
  void add_undirected(int a, int b, int cap) { add_edge(a, b, cap, cap);
↪  }

  long long blocking_flow(int s, int t) {
    // bfs: construct level graph of all edges with residual capacity > 0
    fill(dist.begin(), dist.end(), -1);
    dist[s] = 0;
    auto head=state.begin(), tail=state.begin(); // queue
    for (*tail++ = s; head != tail; ++head)
      for (auto& e : g[*head])
        if (e.rcap > 0 && dist[e.to] == -1) {
          dist[e.to] = dist[*head] + 1;
          *tail++ = e.to;
        }

    // dfs: repeatedly look for s--t paths in the level graph with
↪  capacity > 0
    long long totflow = 0;
    fill(state.begin(), state.end(), 0); // at i, all edges
↪  0..state[i]-1 were visited
    auto top=path.begin(); // one past the end of end current path
    edge dummy{s, numeric_limits<int>::max(), -1};
    *top++ = &dummy;
    while (top != path.begin()) {
      int n = (*prev(top))->to;
      if (n == t) { // found s--t path, reduce capacity
        auto cmp = [](edge *a, edge *b) { return a->rcap < b->rcap; };
        auto next_top = min_element(path.begin(), top, cmp);
```

```
        int flow = (*next_top)->rcap;
        while (--top != path.begin()) {
          edge &e = **top, &f = g[e.to][e.rev];
          e.rcap -= flow;
          f.rcap += flow;
        }
        totflow += flow;
        top = next_top;
        continue;
      }
      for (int &i = state[n], i_max=g[n].size(), need=dist[n]+1;; ++i) {
        if (i == i_max) { // no more paths to t, set n unreachable, pop
↪   stack
          dist[n] = -1;
          --top;
          break;
        }
        if (dist[g[n][i].to] == need && g[n][i].rcap > 0) {
          *top++ = &g[n][i]; // found unvisited edge, push stack
          break;
        }
      }
    }
    return totflow;
  }

  long long max_flow(int s, int t) {
    long long flow = 0;
    while (auto bf = blocking_flow(s, t))
      flow += bf;
    return flow;
  }
};
```

2.3 Min-cost max-flow

See “Fastest min cost flow and circulation” for a faster code.

```
//Min cost max flow by successive shortest paths using SPFA with reduced
↪   edge weights.
//Runs in O(|flow| * E * V) worst case, O(|flow|*E) in practice.
#include<bits/stdc++.h>
using namespace std;

typedef int flow_t;
typedef long long cost_t;
struct mcFlow{
  struct Edge{
    cost_t c;
    flow_t f;
    int to, rev;
    Edge(int _to, cost_t _c, flow_t _f, int _rev):c(_c), f(_f), to(_to),
↪   rev(_rev){}
  };
  const cost_t INFCOST = numeric_limits<cost_t>::max()/2;
  const cost_t INFFLOW = numeric_limits<flow_t>::max()/2;
  int N, S, T;
  vector<vector<Edge> > G;
  mcFlow(int _N, int _S, int _T):N(_N), S(_S), T(_T), G(_N){}
  void add_edge(int a, int b, cost_t cost, flow_t cap){
    if(a==b){return;}
    assert(a>=0&&a<N&&b>=0&&b<N);
    G[a].emplace_back(b, cost, cap, G[b].size());
    G[b].emplace_back(a, -cost, 0, G[a].size()-1);
  }

  pair<flow_t, cost_t> minCostFlow(){
    vector<cost_t> phi(N, 0);
    vector<cost_t> dist(N);
    vector<int> state(N);
    vector<Edge*> from(N, 0);
    queue<int> q;
    cost_t retCost=0; flow_t retFlow=0;
    do{
      fill(dist.begin(), dist.end(), INFCOST);
      fill(state.begin(), state.end(), 0);
      fill(from.begin(), from.end(), (Edge*)0);
      dist[S]=0; state[S]=1;
      q.push(S);
      while(!q.empty()){
        int cur;
        cur = q.front();q.pop();
        state[cur]=2;
        for(Edge &e:G[cur]){
          if(e.f==0)continue;
          cost_t newDist = dist[cur] + phi[cur] - phi[e.to] + e.c;
          if(newDist < dist[e.to]){
            dist[e.to]=newDist; from[e.to]=&e;
            if(state[e.to] != 1) q.push(e.to);
            state[e.to]=1;
          }
        }
      }
      if(from[T]==0) break;
      flow_t augment=INFFLOW;
      for(Edge*e = from[T];e;e=from[G[e->to][e->rev].to]){
        augment=min(augment, e->f);
      }
      for(Edge*e = from[T];e;e=from[G[e->to][e->rev].to]){
        retCost+=e->c*augment;
        e->f-=augment;
        G[e->to][e->rev].f+=augment;
      }
      retFlow+=augment;
      for(int i=0;i<N;++i){
        phi[i]+=dist[i];
      }
    }while(from[T]);
    return make_pair(retFlow, retCost);
  }
  flow_t getFlow(Edge const&e){
    return G[e.to][e.rev].f;
  }
};
```

2.4 Fastest min cost flow and circulation

```
// Fastest flow and min cost flow/circulation code
// Push-Relabel cost-scaling algorithm
// Runs in O( V^3 * log(V * max_edge_cost))
// In practice runs in O(V * E) with constant < 1
// Operates on integers, costs are multiplied by 2N!!
// Works with negative costs
// To get min_cost_max_flow, don't change anything
// To get max flow, just call max_flow
// To get min cost circulation, remove the call to max_flow
// To get min cost (not max) flow, use circulation
// and add a t->s edge with capacity inf and cost 0
template<typename flow_t = int, typename cost_t = int>
struct mcSFlow{
  struct Edge{
    cost_t c;
    flow_t f;
    int to, rev;
    Edge(int _to, cost_t _c, flow_t _f, int _rev):c(_c), f(_f), to(_to),
↪   rev(_rev){}
  };
  static constexpr cost_t INFCOST = numeric_limits<cost_t>::max()/2;
  cost_t eps;
  int N, S, T;
  vector<vector<Edge> > G;
  vector<unsigned int> isq, cur;
  vector<flow_t> ex;
  vector<cost_t> h;
  mcSFlow(int _N, int _S, int _T):eps(0), N(_N), S(_S), T(_T), G(_N){}
  void add_edge(int a, int b, cost_t cost, flow_t cap){
    assert(cap>=0);
    assert(a>=0&&a<N&&b>=0&&b<N);
    if(a==b){assert(cost>=0); return;}
    cost*=N;
    eps = max(eps, abs(cost));
    G[a].emplace_back(b, cost, cap, G[b].size());
    G[b].emplace_back(a, -cost, 0, G[a].size()-1);
  }
  void add_flow(Edge& e, flow_t f) {
    Edge &back = G[e.to][e.rev];
    if (!ex[e.to] && f)
      hs[h[e.to]].push_back(e.to);
    e.f -= f; ex[e.to] += f;
    back.f += f; ex[back.to] -= f;
  }
  vector<vector<int> > hs;
  vector<int> co;
  // fast max flow
  flow_t max_flow() {
    ex.assign(N, 0);
    h.assign(N, 0); hs.resize(2*N);
    co.assign(2*N, 0); cur.assign(N, 0);
    h[S] = N;
    ex[T] = 1;
    co[0] = N-1;
    for(auto &e:G[S]) add_flow(e, e.f);
    if(hs[0].size())
    for (int hi = 0;hi>=0;) {
      int u = hs[hi].back();
      hs[hi].pop_back();
      while (ex[u] > 0) { // discharge u
        if (cur[u] == G[u].size()) {
          h[u] = 1e9;
          for(unsigned int i=0;i<G[u].size();++i){
            auto &e = G[u][i];
            if (e.f && h[u] > h[e.to]+1){
              h[u] = h[e.to]+1, cur[u] = i;
            }
```

```
        }
        if (++co[h[u]], !--co[hi] && hi < N)
          for(int i=0;i<N;++i)
            if (hi < h[i] && h[i] < N){
              --co[h[i]];
              h[i] = N + 1;
            }
        hi = h[u];
      } else if (G[u][cur[u]].f && h[u] == h[G[u][cur[u]].to]+1)
        add_flow(G[u][cur[u]], min(ex[u], G[u][cur[u]].f));
      else ++cur[u];
    }
    while (hi>=0 && hs[hi].empty()) --hi;
  }
  return -ex[S];
}
// begin min cost flow
void push(Edge &e, flow_t amt){
  if(e.f < amt) amt=e.f;
  e.f-=amt; ex[e.to]+=amt;
  G[e.to][e.rev].f+=amt; ex[G[e.to][e.rev].to]-=amt;
}
void relabel(int vertex){
  cost_t newHeight = -INFCOST;
  for(unsigned int i=0;i<G[vertex].size();++i){
    Edge const&e = G[vertex][i];
    if(e.f && newHeight < h[e.to]-e.c){
      newHeight = h[e.to] - e.c;
      cur[vertex] = i;
    }
  }
  h[vertex] = newHeight - eps;
}
static constexpr int scale=2;
pair<flow_t, cost_t> minCostMaxFlow(){
  cost_t retCost = 0;
  for(int i=0;i<N;++i)
    for(Edge &e:G[i])
      retCost += e.c*(e.f);
  // remove this for circulation
  flow_t retFlow = max_flow();
  h.assign(N, 0); ex.assign(N, 0);
  isq.assign(N, 0); cur.assign(N,0);
  stack<int> q; //queue is usually slower
  for(;eps;eps>>=scale){
    fill(cur.begin(), cur.end(), 0);
    for(int i=0;i<N;++i)
      for(auto &e:G[i])
        if(h[i] + e.c - h[e.to] < 0 && e.f) push(e, e.f);
    for(int i=0;i<N;++i){
      if(ex[i]>0){
        q.push(i);
        isq[i]=1;
      }
    }
    while(!q.empty()){
      int u=q.top();q.pop();
      isq[u]=0;
      while(ex[u]>0){
        if(cur[u] == G[u].size())
          relabel(u);
        for(unsigned int &i=cur[u], max_i = G[u].size();i<max_i;++i){
          Edge &e=G[u][i];
          if(h[u] + e.c - h[e.to] < 0){
            push(e, ex[u]);
            if(ex[e.to]>0 && isq[e.to]==0){
              q.push(e.to);
              isq[e.to]=1;
            }
            if(ex[u]==0) break;
    }  }  }  }
    if(eps>1 && eps>>scale==0){
      eps = 1<<scale;
    }
  }
  for(int i=0;i<N;++i){
    for(Edge &e:G[i]){
      retCost -= e.c*(e.f);
    }
  }
  return make_pair(retFlow, retCost/2/N);
}
flow_t getFlow(Edge const &e){
  return G[e.to][e.rev].f;
}
};
```

2.5 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
//
// Daniel: significantly slower than the Dinic MaxFlow for the tiles
↪   task (a sparse graph)
typedef vector<int> VI;
typedef vector<VI> VVI;

// recursive search for alternating path: Hungarian method
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 found unmatched column and found improvement path
      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++) { // try each row as a start
    VI seen(w[0].size());
    if (FindMatch(i, w, mr, mc, seen)) ct++;
  }
  return ct;
}
```

2.6 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)
typedef vector<int> VI;
typedef vector<VI> VVI;

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);
}
```

2.7 Konig's Theorem (Text)

In any bipartite graph $G = (L \cup R, E), E \subseteq \{\{u, v\} \mid u \in L \wedge v \in R\}$, the number of edges in a maximum matching equals the number of vertices in a minimum vertex cover. To construct such a cover, let U be the set of unmatched vertices in L (possibly empty), and let Z be the set of vertices that are either in U or are connected to U by alternating paths. Then $K := (L \backslash Z) \cup (R \cap Z)$ is a minimum vertex cover.

2.8 General Unweighted Maximum Matching

```
// Unweighted general matching.
// Vertices are numbered from 1 to V.
// G is an adjlist.
// G[x][0] contains the number of neighbours of x.
// The neigbours are then stored in G[x][1] .. G[x][G[x][0]].
// Mate[x] will contain the matching node for x.
// V and E are the number of edges and vertices.
// Slow Version (2x on random graphs) of Gabow's implementation
// of Edmonds' algorithm (O(V^3)).
const int MAXV = 250;
int G[MAXV][MAXV];
int VLabel[MAXV];
int  Queue[MAXV];
int   Mate[MAXV];
int   Save[MAXV];
int   Used[MAXV];
int    Up, Down;
int           V;

void ReMatch(int x, int y)
{
  int m = Mate[x]; Mate[x] = y;
  if (Mate[m] == x)
    {
      if (VLabel[x] <= V)
        {
          Mate[m] = VLabel[x];
          ReMatch(VLabel[x], m);
        }
      else
        {
          int a = 1 + (VLabel[x] - V - 1) / V;
          int b = 1 + (VLabel[x] - V - 1) % V;
          ReMatch(a, b); ReMatch(b, a);
        }
    }
}

void Traverse(int x)
{
  for (int i = 1; i <= V; i++) Save[i] = Mate[i];
  ReMatch(x, x);
  for (int i = 1; i <= V; i++)
    {
      if (Mate[i] != Save[i]) Used[i]++;
      Mate[i] = Save[i];
    }
}

void ReLabel(int x, int y)
{
  for (int i = 1; i <= V; i++) Used[i] = 0;
  Traverse(x); Traverse(y);
  for (int i = 1; i <= V; i++)
    {
      if (Used[i] == 1 && VLabel[i] < 0)
        {
          VLabel[i] = V + x + (y - 1) * V;
          Queue[Up++] = i;
        }
    }
}

void Solve()
{
  for (int i = 1; i <= V; i++)
    if (Mate[i] == 0)
      {
        for (int j = 1; j <= V; j++) VLabel[j] = -1;
        VLabel[i] = 0; Down = 1; Up = 1; Queue[Up++] = i;
        while (Down != Up)
          {
            int x = Queue[Down++];
            for (int p = 1; p <= G[x][0]; p++)
              {
                int y = G[x][p];
                if (Mate[y] == 0 && i != y)
                  {
                    Mate[y] = x; ReMatch(x, y);
                    Down = Up; break;
                  }
                if (VLabel[y] >= 0)
                  {
                    ReLabel(x, y);
                    continue;
                  }
                if (VLabel[Mate[y]] < 0)
                  {
                    VLabel[Mate[y]] = x;
                    Queue[Up++] = Mate[y];
                  }
              }
          }
      }
}

int Size()
{
  int Count = 0;
  for (int i = 1; i <= V; i++)
    if (Mate[i] > i) Count++;
  return Count;
}

int main(int argc, char * argv[])
{
  scanf("%i ", &V);
  while(!feof(stdin)) {
    int a,b;
    scanf("%i %i ", &a, &b);
    G[a][++G[a][0]]=b;
    G[b][++G[b][0]]=a;
  }
  Solve();
  printf("%i\n", 2*Size());
  for (int i = 1; i <= V; i++)
    if (Mate[i] > i) printf("%i %i\n", i, Mate[i]);
}
```

3 Math Algorithms

3.1 Number theoretic algorithms (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.
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) {
  int tmp;
  while(b){a%=b; tmp=a; a=b; b=tmp;}
  return a;
}

// computes lcm(a,b)
int lcm(int a, int b) {
  return a/gcd(a,b)*b;
}

// returns d = gcd(a,b); finds x,y such that d = ax + by
int extended_euclid(int a, int b, int &x, int &y) {
  int xx = y = 0;
  int yy = x = 1;
  while (b) {
    int q = a/b;
    int t = b; b = a%b; a = t;
    t = xx; xx = x-q*xx; x = t;
    t = yy; yy = y-q*yy; y = t;
  }
  return a;
}

// finds all solutions to ax = b (mod n)
VI modular_linear_equation_solver(int a, int b, int n) {
  int x, y;
  VI solutions;
  int d = extended_euclid(a, n, x, y);
  if (!(b%d)) {
    x = mod (x*(b/d), n);
    for (int i = 0; i < d; i++)
      solutions.push_back(mod(x + i*(n/d), n));
  }
```

```
  return solutions;
}

// computes b such that ab = 1 (mod n), returns -1 on failure
int mod_inverse(int a, int n) {
  int x, y;
  int d = extended_euclid(a, n, x, y);
  if (d > 1) return -1;
  return mod(x,n);
}

// Chinese remainder theorem (special case): find z such that
// z % x = a, z % y = b.  Here, z is unique modulo M = lcm(x,y).
// Return (z,M).  On failure, M = -1.
PII chinese_remainder_theorem(int x, int a, int y, int b) {
  int s, t;
  int d = extended_euclid(x, y, s, t);
  if (a%d != b%d) return make_pair(0, -1);
  return make_pair(mod(s*b*x+t*a*y,x*y)/d, x*y/d);
}

// Chinese remainder theorem: find z such that
// z % x[i] = a[i] for all i.  Note that the solution is
// unique modulo M = lcm_i (x[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 &x, const VI &a) {
  PII ret = make_pair(a[0], x[0]);
  for (int i = 1; i < x.size(); i++) {
    ret = chinese_remainder_theorem(ret.first, ret.second, x[i], a[i]);
    if (ret.second == -1) break;
  }
  return ret;
}

// computes x and y such that ax + by = c; on failure, x = y =-1
void linear_diophantine(int a, int b, int c, int &x, int &y) {
  int d = gcd(a,b);
  if (c%d) {
    x = y = -1;
  } else {
    x = c/d * mod_inverse(a/d, b/d);
    y = (c-a*x)/b;
  }
}
```

3.2 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[][]
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;
  }
}
```

3.3 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 nxn matrix
//
// OUTPUT:   rref[][] = an nxm matrix (stored in a[][])
//           returns rank of a[][]
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; 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;
  const int 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] + n);

  int rank = rref (a);

  // expected: 4
  cout << "Rank: " << rank << endl;

  // expected: 1 0 0 1
  //           0 1 0 3
  //           0 0 1 -3
  //           0 0 0 2.78206e-15
  //           0 0 0 3.22398e-15
  cout << "rref: " << endl;
  for (int i = 0; i < 5; i++){
    for (int j = 0; j < 4; j++)
      cout << a[i][j] << ' ';
    cout << endl;
  }

}
```

3.4 Fast Fourier transform

3.4.1 Iterative, with doubles

```
// faster FFT implementation with double, numbers < 2e9 are fine.
// polynomial mod can compute linear recurrences in O(n log n log k).
using ll = long long;
namespace fft{
int log2i(unsigned long long a){
  return __builtin_clzll(1) - __builtin_clzll(a);
}
const double PI = 3.141592653589793238462６;
vector<complex<double> > roots;
void gen_roots(int N){
  if((int)roots.size()!=N){
    roots.clear();
    roots.resize(N);
    for(int i=0;i<N;++i){
      if((i&-i) == i){
        roots[i] = polar(1.0, 2.0*PI*i/N);
      } else {
        roots[i] = roots[i&-i] * roots[i-(i&-i)];
  } } }
}
void fft(complex<double> const*a, complex<double> *to, int n, bool isInv
↪  = false){
  to[0] = a[0];
  for (int i=1, j=0; i<n; ++i) {
    int m = n >> 1;
    for (; j>=m; m >>= 1)
      j -= m;
    j += m;
    to[i] = a[j];
  }
  gen_roots(n);
  for(int iter=1, sh=log2i(n)-1;iter<n;iter*=2, --sh){
    for(int x=0;x<n;x+=2*iter){
      for(int y=0;y<iter;++y){
        complex<double> ome = roots[y<<sh];
        if(isInv) ome = conj(ome);
        complex<double> v = to[x+y], w=to[x+y+iter];
        to[x+y] = v+ome*w;
        to[x+y+iter] = v-ome*w;
  } } }
}
template<ll mod, typename int_t>
vector<int_t> poly_mul(vector<int_t> const&a, vector<int_t> const&b){
  int logn = log2i(a.size()+b.size()-1)+1;
  int n = 1<<logn;
  vector<complex<double> > x(n), y(n), xx(n), yy(n
  for(int i=0;i<(int)a.size();++i) x[i] =
↪  complex<double>(a[i]&((1<<15)-1), a[i]>>15);
  for(int i=0;i<(int)b.size();++i) y[i] =
↪  complex<double>(b[i]&((1<<15)-1), b[i]>>15);

  fft(x.data(), xx.data(), n, false);
  fft(y.data(), yy.data(), n, false);
  for(int i=0;i<n;++i){
    int j = (n-i)&(n-1); //reverse index
    complex<double> rx = (xx[i] + conj(xx[j]))*0.5;
    complex<double> ix = (xx[i] - conj(xx[j]))*complex<double>(0, -0.5);
    complex<double> ry = (yy[i] + conj(yy[j]))*0.5;
    complex<double> iy = (yy[i] - conj(yy[j]))*complex<double>(0, -0.5);
    x[i] = (rx*ry + ix*iy*complex<double>(0, 1.0))/(double)n;
    y[i] = (rx*iy + ix*ry)/(double)n;
  }
  fft(x.data(), xx.data(), n, true);
  fft(y.data(), yy.data(), n, true);
  vector<int_t> ret(a.size()+b.size()-1);
  for(int i=0;i<(int)ret.size();++i){
    ll l = llround(xx[i].real()), m = llround(yy[i].real()),
↪   r=llround(xx[i].imag());
    ret[i] = (l + (m%mod<<15) + (r%mod<<30))%mod;
  }
  return ret;
}
}
template<ll mod>
struct NT{
  static int add(int const&a, int const&b){
    ll ret = a+b; if(ret>=mod) ret-=mod;
    return ret;
  }
  static int& xadd(int& a, int const&b){
    a+=b; if(a>=mod) a-=mod;
    return a;
  }
  static int sub(int const&a, int const&b){
    return add(a, mod-b);
  }
  static int& xsub(int& a, int const&b){
    return xadd(a, mod-b);
  }
  static int mul(int const&a, int const&b){
    return a*(ll)b%mod;
  }
  static int& xmul(int const&a, int const&b){
    return a=mul(a, b);
  }
  static int inv_rec(int const&a, int const&m){
    if(a==1) return 1;
    return m+(1-inv_rec(m%a, a)*(ll)m)/a;
  }
  static int inv(int const&a){
    return inv_rec(a, mod);
  }
};
template<ll mod>
struct poly : vector<int>{
  poly(size_t a):vector<int>(a){}
  poly(size_t a, int b):vector<int>(a, b){}
  poly(vector<int> const&a):vector<int>(a){}
  poly& normalize(){
    while(size()>1 && back()==0) pop_back();
    return *this;
  }
  poly substr(int l, int r)const{
    if(r>(int)size()) r=size();
    if(l>(int)size()) l=size();
    return poly(vector<int>(begin()+l, begin()+r));
  }
  poly reversed()const{
    return poly(vector<int>(rbegin(), rend()));
  }
  poly operator+(poly const&o)const{
    poly ret(max(size(), o.size()));
    copy(begin(), end(), ret.begin());
    for(int i=0;i<(int)o.size();++i)
      NT<mod>::xadd(ret[i], o[i]);
    return ret.normalize();;
  }
  poly operator-(poly const&o)const{
    poly ret(max(size(), o.size()));
    copy(begin(), end(), ret.begin());
    for(int i=0;i<(int)o.size();++i)
      NT<mod>::xsub(ret[i], o[i]);
    return ret.normalize();
  }
  // multiplication in n log n
  poly operator*(poly const&o)const{
    poly ret(fft::poly_mul<mod, int>(*this, o));
    return ret.normalize();
  }
  // inverse mod x^n
  poly inv(int n)const{
    assert(size() && operator[](0));
    if((int)size()>n) return poly(vector<int>(begin(),
↪   begin()+n)).inv(n);
    poly ret(1, NT<mod>::inv(operator[](0)));
    ret.reserve(2*n);
    for(int i=1;i<n;i*=2){
```

```
      poly l = substr(0, i) * ret; // l[0:i] will be 0
      poly r = substr(i, 2*i) * ret; // r[i:2*i] will be irrelevant
      poly up = (l.substr(i, 2*i) + r.substr(0, i)) * ret;
      ret.resize(2*i);
      for(int j=0;j<i;++j){
        ret[i+j] = NT<mod>::sub(0, up[j]);
      }
    }
    ret.resize(n);
    return ret.normalize();
  }
  pair<poly, poly> div(poly const&o, poly const& oinvrev)const{
    if(o.size()>size()) return {poly(1, 0), *this};
    int rsize = size()-o.size()+1;
    poly q = (reversed()*oinvrev.substr(0, rsize));
    q.resize(rsize);
    reverse(q.begin(), q.end());
    poly r = *this - q*(o);
    return make_pair(q, r.normalize());
  }
  // division and mod in O(n log n)
  pair<poly, poly> div(poly const&o)const{
    return div(o, o.reversed().inv(size()+2));
  }
  // operations on power series
  poly derivative()const{
    poly ret(size()-1);
    for(unsigned int i=1;i<size();++i){
      ret[i-1] = NT<mod>::mul(operator[](i), i);
    }
    return ret;
  }
  poly integrated(int const&constant_term = 0)const{
    poly ret(size()+1);
    ret[0] = constant_term;
    for(unsigned int i=0;i<size();++i){
      ret[i+1] = NT<mod>::mul(operator[](i), NT<mod>::inv(i+1));
    }
    return ret;
  }
  poly logarithm(unsigned int const&n)const{
    assert(n>0);
    assert(operator[](0) == 1);
    return (derivative()*(inv(n+5))).substr(0, n-1).integrated(0);
  }
  poly exponential(unsigned int const&n)const{
    assert(operator[](0) == 0);
    poly ret(1, 1), ret_inv(1, 1);
    ret.reserve(2*n); ret_inv.reserve(2*n);
    for(unsigned int i=1;i<n;i*=2){
      ret_inv = ret_inv + ret_inv - (ret_inv*ret_inv*ret).substr(0, i);
      poly q = substr(0, i).derivative();
      poly r = (ret * q);
      r = r.substr(0, i) + r.substr(i, 2*i);
      poly s = ((ret.derivative()-r).shift(1));
      s = s.substr(0, i) + s.substr(i, 2*i);
      poly t = (ret_inv * s).substr(0, i);
      poly u = (substr(0, 2*i) - t.shift(i-1).integrated(0)).substr(i,
↪   2*i);
      poly up = (ret*u).substr(0, i);
      ret.resize(2*i);
      for(unsigned int j=0;j<min((unsigned int)up.size(), i);++j){
        ret[i+j] = up[j];
      }
    }
    ret.resize(n);
    return ret.normalize();
  }
};
```

3.4.2 Non-recursive, in a finite field

```
unsigned long long log2i(unsigned long long n) {
  return CHAR_BIT*sizeof(n) - __builtin_clzll(n);
}
constexpr uint32_t mul(uint32_t a, uint32_t b, uint32_t mod) {
  return uint64_t(a)*b%mod;
}
template <typename F> constexpr uint32_t mul(uint32_t a, uint32_t b) {
  return mul(a, b, F::mod);
}
template <typename F> uint32_t add(uint32_t a, uint32_t b) {
  auto res = a+b; return res < F::mod ? res : res-F::mod;
}
template <typename F>
uint32_t sub(uint32_t a, uint32_t b) { return add<F>(a, F::mod-b); }
constexpr uint64_t cpowm(uint64_t base, uint64_t exp, uint64_t mod) {
  return !exp ? 1 : exp%2
    ? mul(base, cpowm(mul(base, base, mod), exp/2, mod), mod)
    : cpowm(mul(base, base, mod), exp/2, mod);
}
uint32_t powm(uint32_t base, uint32_t exp, uint32_t mod) {
  uint32_t res=1;
  for (; exp; exp>>=1, base = mul(base, base, mod))
    if (exp&1)
      res = mul(res, base, mod);
  return res;
}
template <typename F>
uint32_t powm(uint32_t base, uint32_t exp) { return powm(base, exp,
↪   F::mod); }
template <typename F> uint32_t inv(uint32_t a) {
  return powm(a, F::mod-2, F::mod);
}
constexpr uint64_t cmodinv(uint64_t a, uint64_t mod) {
  return cpowm(a, mod-2, mod);
}
struct fft_field_1 {
  static const uint32_t nth_root = uint32_t(1)<<26;
  static const uint32_t mod = 7*nth_root + 1;
  static const uint32_t root_of_unity = 30;
  static const uint32_t modinv_of_unity = cmodinv(root_of_unity, mod);
};
struct fft_field_2 {
  static const uint32_t nth_root = uint32_t(1)<<25;
  static const uint32_t mod = 5*nth_root + 1;
  static const uint32_t root_of_unity = 17;
  static const uint32_t modinv_of_unity = cmodinv(root_of_unity, mod);
};
template <typename Field>
void fft(uint32_t *f, size_t n, uint32_t const *w) {
  for (size_t i=1, j=0; i<n; ++i) {
    uint32_t m = n >> 1;
    for (; j>=m; m >>= 1)
      j -= m;
    j += m;
    if (i < j)
      swap(f[i], f[j]);
  }
  for (size_t step = 1; step < n; step <<= 1, ++w) {
    for (size_t i = 0; i < n; i += 2*step) {
      uint32_t w_j = 1;
      auto *a = f+i, *b = f+i+step;
      for (size_t j = step;; w_j = mul<Field>(w_j, *w)) {
        auto u = *a, v = mul<Field>(*b, w_j);
        *a++ = add<Field>(u, v);
        *b++ = sub<Field>(u, v);
        if (!--j)
          break;
      }
    }
  }
}
template <typename Field, bool inverse>
vector<uint32_t> generate_roots(size_t n) {
  vector<uint32_t> roots(log2i(n)-1);
  const auto w = inverse ? Field::modinv_of_unity : Field::root_of_unity;
  roots.back() = powm<Field>(w, Field::nth_root/n);
  for (auto it=roots.rbegin(); it+1!=roots.rend(); ++it)
    *next(it) = mul<Field>(*it, *it);
  return roots;
}
const auto n = v.size(), n_inv = inv<Field>(n);
const auto forw = generate_roots<Field, false>(n),
           bakw = generate_roots<Field, true>(n);
fft<Field>(v.data(), n, forw.data()); // fft
fft<Field>(v.data(), n, bakw.data()); // inverse fft
```

3.5 Simplex algorithm

```
// Two-phase simplex algorithm for solving linear programs of the form
//
//     maximize     c^T x
//     subject to   Ax <= b
//                  x >= 0
//
// INPUT: A -- an m x n matrix
//        b -- an m-dimensional vector
//        c -- an n-dimensional vector
//        x -- a vector where the optimal solution will be stored
//
// OUTPUT: value of the optimal solution (infinity if unbounded
//         above, nan if infeasible)
//
// To use this code, create an LPSolver object with A, b, and c as
// arguments.  Then, call Solve(x).
typedef long double DOUBLE;
typedef vector<DOUBLE> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;

const DOUBLE EPS = 1e-9;

struct LPSolver {
```

```
  int m, n;
  VI B, N;
  VVD D;

  LPSolver(const VVD &A, const VD &b, const VD &c) :
    m(b.size()), n(c.size()), B(m), N(n+1), D(m+2, VD(n+2)) {
    for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) D[i][j] =
 ↪  A[i][j];
    for (int i = 0; i < m; i++) { B[i] = n+i; D[i][n] = -1; D[i][n+1] =
 ↪  b[i]; }
    for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c[j]; }
    N[n] = -1; D[m+1][n] = 1;
  }

  void Pivot(int r, int s) {
    for (int i = 0; i < m+2; i++) if (i != r)
      for (int j = 0; j < n+2; j++) if (j != s)
        D[i][j] -= D[r][j] * D[i][s] / D[r][s];
    for (int j = 0; j < n+2; j++) if (j != s) D[r][j] /= D[r][s];
    for (int i = 0; i < m+2; i++) if (i != r) D[i][s] /= -D[r][s];
    D[r][s] = 1.0 / D[r][s];
    swap(B[r], N[s]);
  }

  bool Simplex(int phase) {
    int x = phase == 1 ? m+1 : m;
    while (true) {
      int s = -1;
      for (int j = 0; j <= n; j++) {
        if (phase == 2 && N[j] == -1) continue;
        if (s == -1 || D[x][j] < D[x][s] || (D[x][j] == D[x][s] && N[j] <
 ↪  N[s])) s = j;
      }
      if (D[x][s] >= -EPS) return true;
      int r = -1;
      for (int i = 0; i < m; i++) {
        if (D[i][s] <= EPS) continue;
        if (r == -1 || D[i][n+1] / D[i][s] < D[r][n+1] / D[r][s] ||
            (D[i][n+1] / D[i][s] == D[r][n+1] / D[r][s] && B[i] < B[r]))
 ↪  r = i;
      }
      if (r == -1) return false;
      Pivot(r, s);
    }
  }

  DOUBLE Solve(VD &x) {
    int r = 0;
    for (int i = 1; i < m; i++) if (D[i][n+1] < D[r][n+1]) r = i;
    if (D[r][n+1] <= -EPS) {
      Pivot(r, n);
      if (!Simplex(1) || D[m+1][n+1] < -EPS) return
 ↪  -numeric_limits<DOUBLE>::infinity();
      for (int i = 0; i < m; i++) if (B[i] == -1) {
        int s = -1;
        for (int j = 0; j <= n; j++)
          if (s == -1 || D[i][j] < D[i][s] || (D[i][j] == D[i][s] && N[j]
 ↪  < N[s])) s = j;
        Pivot(i, s);
      }
    }
    if (!Simplex(2)) return numeric_limits<DOUBLE>::infinity();
    x = VD(n);
    for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i][n+1];
    return D[m][n+1];
  }
};

int main() {

  const int m = 4;
  const int n = 3;
  DOUBLE _A[m][n] = {
    { 6, -1, 0 },
    { -1, -5, 0 },
    { 1, 5, 1 },
    { -1, -5, -1 }
  };
  DOUBLE _b[m] = { 10, -4, 5, -5 };
  DOUBLE _c[n] = { 1, -1, 0 };

  VVD A(m);
  VD b(_b, _b + m);
  VD c(_c, _c + n);
  for (int i = 0; i < m; i++) A[i] = VD(_A[i], _A[i] + n);

  LPSolver solver(A, b, c);
  VD x;
  DOUBLE value = solver.Solve(x);

  cerr << "VALUE: "<< value << endl;
  cerr << "SOLUTION:";
  for (size_t i = 0; i < x.size(); i++) cerr << " " << x[i];
  cerr << endl;
  return 0;
}
```

3.6 Fast factorization

```
typedef long long unsigned int llui;
typedef long long int lli;
typedef long double float64;

llui mul_mod(llui a, llui b, llui m){
   llui y = (llui)((float64)a*(float64)b/m+(float64)1/2);
   y = y * m;
   llui x = a * b;
   llui r = x - y;
   if ( (lli)r < 0 ){
      r = r + m; y = y - 1;
   }
   return r;
}

llui C,a,b;
llui gcd(){
   llui c;
   if(a>b){
      c = a; a = b; b = c;
   }
   while(1){
      if(a == 1LL) return 1LL;
      if(a == 0 || a == b) return b;
      c = a; a = b%a;
      b = c;
   }
}

llui f(llui a, llui b){
   llui tmp;
   tmp = mul_mod(a,a,b);
   tmp+=C; tmp%=b;
   return tmp;
}

llui pollard(llui n){
   if(!(n&1)) return 2;
   C=0;
   llui iteracoes = 0;
   while(iteracoes <= 1000){
      llui x,y,d;
      x = y = 2; d = 1;
      while(d == 1){
         x = f(x,n);
         y = f(f(y,n),n);
         llui m = (x>y)?(x-y):(y-x);
         a = m; b = n; d = gcd();
      }
      if(d != n)
         return d;
      iteracoes++; C = rand();
   }
   return 1;
}

llui pot(llui a, llui b, llui c){
   if(b == 0) return 1;
   if(b == 1) return a%c;
   llui resp = pot(a,b>>1,c);
   resp = mul_mod(resp,resp,c);
   if(b&1)
      resp = mul_mod(resp,a,c);
   return resp;
}

bool isPrime(llui n){
   llui d = n-1;
   llui s = 0;
   if(n <=3 || n == 5) return true;
   if(!(n&1)) return false;
   while(!(d&1)){ s++; d>>=1; }
   for(llui i = 0;i<32;i++){
      llui a = rand();
      a <<=32;
      a+=rand();
      a%=(n-3); a+=2;
      llui x = pot(a,d,n);
      if(x == 1 || x == n-1) continue;
      for(llui j = 1;j<= s-1;j++){
         x = mul_mod(x,x,n);
         if(x == 1) return false;
         if(x == n-1)break;
      }
      if(x != n-1) return false;
   }
```

```
      return true;
}
map<llui,int> factors;
void fact(llui n){
   if(!isPrime(n)){
      llui fac = pollard(n);
      fact(n/fac); fact(fac);
   }else{
      map<llui,int>::iterator it;
      it = factors.find(n);
      if(it != factors.end()){
         (*it).second++;
      }else{
         factors[n] = 1;
      }
   }
}
```

3.7 Euler's Totient

```
// This code took less than 0.5s to calculate with MAX = 10^7
#define MAX 10000000

int phi[MAX];
bool pr[MAX];

void totient(){
  for(int i = 0; i < MAX; i++){
    phi[i] = i;
    pr[i] = true;
  }
  for(int i = 2; i < MAX; i++)
    if(pr[i]){
      for(int j = i; j < MAX; j+=i){
        pr[j] = false;
        phi[j] = phi[j] - (phi[j] / i);
      }
      pr[i] = true;
    }
}
```

4 Game theory

Grundy numbers For a two-player, normal-play (last to move wins) game on a graph (V, E) : $G(x) = mex(G(y) : (x, y) \in E)$, where $mex(S) = min\{n \geq 0 : n \notin S\}$. x is losing iff $G(x) = 0$.

Sums of games

- Player chooses a game and makes a move in it. Grundy number of a position is xor of Grundy numbers of positions in summed games.
- Player chooses a non-empty subset of games (possibly, all) and makes moves in all of them. A position is losing iff each game is in a losing position.
- Player chooses a proper subset of games (not empty and not all), and makes moves in all chosen ones. A position is losing iff grundy numbers of all games are equal.
- Player must move in all games, and loses if cant move in some game. A position is losing if any of the games is in a losing position.

Nim A position with pile sizes $a_1, a_2, \cdots, a_n \geq$, not all equal to 1, is losing iff a_1 xor ... xor $a_n = 0$ (like in normal nim). A position with n piles of size 1 is losing iff n is odd.

5 Graphs

5.1 Strongly connected components (C)

```
struct scc {
  vector<vector<int> > g, g_rev;
  vector<bool> visited;
  vector<int> group_num;
  int group_cnt;
  stack<int> s;

  scc(size_t n)
    : g(n), g_rev(n), visited(n), group_num(n),
      group_cnt(0) {}
  void add_edge(int a, int b) {
    g[a].push_back(b);
    g_rev[b].push_back(a);
  }
  void fill_forward(int v) {
    visited[v] = true;
    for (int w : g[v])
      if (!visited[w])
        fill_forward(w);
    s.push(v);
  }
  void fill_backward(int v) {
    visited[v] = false;
    group_num[v] = group_cnt;
    for (int w : g_rev[v])
      if (visited[w])
        fill_backward(w);
  }
  int calculate() {
    for (size_t i=0; i<g.size(); ++i)
      if (!visited[i])
        fill_forward(i);
    for (; !s.empty(); s.pop())
      if (visited[s.top()])
        fill_backward(s.top()), ++group_cnt;
    return group_cnt;
  }
};
```

5.2 Bridges

```
// Finds bridges and cut vertices
// Receives:
// N: number of vertices
// l: adjacency list
// Gives:
// vis, seen, par (used to find cut vertices)
// ap - 1 if it is a cut vertex, 0 otherwise
// brid - vector of pairs containing the bridges
typedef pair<int, int> PII;
const int MAX = 100000;

int N;
vector <int> l[MAX];
vector <PII> brid;
int vis[MAX], seen[MAX], par[MAX], ap[MAX];
int cnt, root;

void dfs(int x){
  if(vis[x] != -1)
    return;
  vis[x] = seen[x] = cnt++;

  int adj = 0;
  for(int i = 0; i < (int)l[x].size(); i++){
    int v = l[x][i];
    if(par[x] == v)
      continue;
    if(vis[v] == -1){
      adj++;
      par[v] = x;
      dfs(v);
      seen[x] = min(seen[x], seen[v]);
      if(seen[v] >= vis[x] && x != root)
        ap[x] = 1;
      if(seen[v] == vis[v])
        brid.push_back(make_pair(v, x));
    }
    else{
      seen[x] = min(seen[x], vis[v]);
      seen[v] = min(seen[x], seen[v]);
    }
  }
  if(x == root) ap[x] = (adj>1);
}

void bridges(){
  brid.clear();
  for(int i = 0; i < N; i++){
    vis[i] = seen[i] = par[i] = -1;
    ap[i] = 0;
  }
  cnt = 0;
  for(int i = 0; i < N; i++)
    if(vis[i] == -1){
      root = i;
      dfs(i);
    }
}
```

5.3 Dominator tree

```
//  Dominator tree in O(M log(N)) time
//  Algorithm by T.Lengauer and R.E.Tarjan
struct Dominator{
```

```
  struct min_DSU{
    vector<int> par, val;
    vector<int> const&semi;
    min_DSU(int N, vector<int> const&semi):par(N, -1),val(N), semi(semi){
      iota(val.begin(), val.end(), 0);
    }
    void comp(int x){
      if(par[par[x]]!=-1){
        comp(par[x]);
        if(semi[val[par[x]]]<semi[val[x]])
          val[x] = val[par[x]];
        par[x]=par[par[x]];
      }
    }
    int f(int x){
      if(par[x]==-1) return x;
      comp(x);
      return val[x];
    }
    void link(int x, int p){
      par[x] = p;
    }
  };
  int N;
  vector<vector<int> > G, rG;
  vector<int> idom, order;
  Dominator(int _N):N(_N), G(N), rG(N){}
  void add_edge(int a, int b){
    G[a].emplace_back(b);
    rG[b].emplace_back(a);
  }
  vector<int> calc_dominators(int S){
    idom.assign(N, -1);
    vector<int> par(N, -1), semi(N, -1);
    vector<vector<int> > bu(N);
    stack<int> s;
    s.emplace(S);
    while(!s.empty()){
      int a=s.top();s.pop();
      if(semi[a]==-1){
        semi[a] = order.size();
        order.emplace_back(a);
        for(int i=0;i<(int)G[a].size();++i){
          if(semi[G[a][i]]==-1){
            par[G[a][i]]=a;
            s.push(G[a][i]);
          }
        }
      }
    }
    min_DSU uni(N, semi);
    for(int i=(int)order.size()-1;i>0;--i){
      int w=order[i];
      for(int f:rG[w]){
        int oval = semi[uni.f(f)];
        if(oval>=0 && semi[w]>oval) semi[w] = oval;
      }
      bu[order[semi[w]]].push_back(w);
      uni.link(w, par[w]);
      for(int v:bu[par[w]]){
        int u=uni.f(v);
        idom[v] = semi[u] < semi[v] ? u : par[w];
      }
      bu[par[w]].clear();
    }
    for(int i=1;i<(int)order.size();++i){
      int w=order[i];
      if(idom[w] != order[semi[w]])
        idom[w] = idom[idom[w]];
    }
    idom[S]=-1;
    return idom;
  }
};
```

5.4 Eulerian path

```
//Steps:
//Pick a starting node and recurse on that node. At each step:
//  If the node has no neighbors, then append the node to the circuit and
↪   return
//  If the node has a neighbor, then make a list of the neighbors and
↪   process them (which includes deleting them from the list of nodes on
↪   which to work) until the node has no more neighbors
//  To process a node, delete the edge between the current node and its
↪   neighbor, recurse on the neighbor, and postpend the current node to
↪   the circuit.
#define MAX 15000
typedef pair<int, int> PII;

int N;
vector<int> circuit;
vector<PII> edges;
vector<int> valid;
vector<int> l[MAX];
int degree[MAX];

void find_path(int x){
  for(int i = 0; i < (int)l[x].size(); i++){
    int e = l[x][i];
    if(!valid[e]) continue;
    int v = edges[e].first;
    if(v == x) v = edges[e].second;
    valid[e] = 0;
    find_path(v);
  }
  circuit.push_back(x);
}

void find_euler_path(){
  circuit.clear();
  //supposes graph is connected and has correct degree
  for(int i = 0; i < N; i++)
    if(degree[i]%2){
      find_path(i);
      return;
    }
  find_path(0);
}

int main(){
  int M;
  while(scanf(" %d %d", &N, &M) && N > 0){
    edges.clear(); valid.clear();
    for(int i = 0; i < N; i++){
      l[i].clear();
      degree[i] = 0;
    }
    for(int i = 0; i < M; i++){
      int x, y;
      scanf("%d%d", &x, &y);
      edges.push_back(make_pair(x, y));
      valid.push_back(1);
      l[x].push_back(i);
      l[y].push_back(i);
      degree[x]++; degree[y]++;
    }
    find_euler_path();
    for(int i = 0; i < (int)circuit.size(); i++)
      printf("%d ", circuit[i]);
    printf("\n");
  }
  return 0;
}
```

6 Data Structures

6.1 Suffix arrays

```
// 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.

// high level idea: first sort all suffixes according the their first
↪  letter, then first 2 letters, then 4,8,16,...
// always combine two order indices of the previous level to get a pair
↪  of order indices for the new level
// total of O(log L) repetitions needed each in O(L log L)

struct SuffixArray {
  const int L;
  string s;
  vector<vector<int> > P; // if only the sorted order is needed and the
↪  memory limit is exceeded, then optimize this to only use the last
↪  two rows of this matrix and save a O(log L) memory factor
  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;
  }
};

int main() {
  // bobocel is the 0'th suffix
  //  obocel is the 5'th suffix
  //   bocel is the 1'st 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;
}
```

6.2 Convex hull (DP optimization)

```
const long long xLeftLeftest = -(1LL<<61);
const long long qQuery = -(1LL<<60);
struct Line{
  long long m, q;
  mutable long double xLeft;
  Line(long long _m, long long _q):m(_m), q(_q), xLeft(xLeftLeftest){}
  bool operator<(const Line& other)const{
    if(q==qQuery){ return m < other.xLeft; }
    if(other.q==qQuery){ return xLeft < m; }
    return m < other.m;
  }
   void recalcXLeft(const Line & pre)const{
    xLeft = -((long double)pre.q-q ) / (pre.m-m);
  }
};
//Max hull for DP optimisation
struct Hull{
  multiset<Line> slopes;
  bool bad(multiset<Line>::iterator it){
    auto suc = next(it);
    if(it==slopes.begin()){
      if(suc==slopes.end()) return false;
      return it->m==suc->m && it->q <=suc->q;
    }
    auto pre = prev(it);
    if(suc==slopes.end()){
      return it->m==pre->m && it->q <= pre->q;
    }
    return ((long double)it->q - suc->q) / (suc->m - it->m) <= ((long
↪   double)pre->q - it->q) / (it->m - pre->m);//check x intersection
  }
  void insert(Line const & l){
    multiset<Line>::iterator e = slopes.insert(l);
    if(bad(e)){
      slopes.erase(e);
      return;
    }
    while(next(e)!=slopes.end() && bad(next(e))) slopes.erase(next(e));
    if(next(e)!=slopes.end()) next(e)->recalcXLeft(*e);
    while(e!=slopes.begin() && bad(prev(e))) slopes.erase(prev(e));
    if(e!=slopes.begin()) e->recalcXLeft(*prev(e));
    else e->xLeft = xLeftLeftest;
  }

  long long query(long long x){
    auto e = slopes.upper_bound(Line(x, qQuery));
    --e;
    return e->m * x + e->q;
  }
};
// for monotonically increasing slopes and increasing query x-values
struct monoHull{
  deque<Line> data;
  bool badBack(Line const & newLine){
    if(data.size()<2) return false;
    Line secondLast = data[(int)data.size()-2];
    Line last = data.back();
    return ((long double)last.q - newLine.q) / (newLine.m - last.m) <=
↪   ((long double)secondLast.q - last.q) / (last.m -
↪   secondLast.m);//check x intersection
  }
  void insert(Line const & l){
    assert(data.empty() || l.m > data.back().m);
    while(badBack(l)) data.pop_back();
    l.recalcXLeft(data.back());
    data.push_back(l);
  }
  long long query(long long x){
    while(data.size()>1 && x>data[1].xLeft) data.pop_front();
    return data.front().m*x+data.front().q;
  }
};
```

7 Strings

7.1 ETH-Collection

```
#define REP(i, a) for (int i = 0, _n = (a); i < _n; ++i)
#define FOR(i, a, b) for (int i = (a), _n = (b); i <= _n; ++i)

// IN: T -- pointer to the text, P -- pointer to int array with result, n
↪   -- number of characters
// OUT:  P[i] -- longest j s.t. T[0]..T[j-1] == T[i]..T[i+j-1]
// time: O(n)
template <class Iter1, class Iter2>
void prefpref(Iter1 T, Iter2 P, int n) {
  P[0] = n;
  int i = 1, t = 0;
  while (i < n) {
    while (i+t < n && T[i+t] == T[t]) ++t;
    P[i] = t;
    int k = 1;
    while (k < t && P[k] != t-k) {
      P[i+k] = min(P[k], t-k);
      ++k;
    }
    i += k, t = max(0, t-k);
  }
}
// Analogously for suffixes (T[n-j]..T[n-1] == T[i-j+1]..T[i])
template <class Iter1, class Iter2>
void sufsuf(Iter1 T, Iter2 P, int n) {
  prefpref(reverse_iterator<Iter1>(T+n), reverse_iterator<Iter2>(P+n),
↪   n);
}

// Duval's O(n) algortihm for lexicographically maximum suffix
// IN: T -- text , n -- length of T
// OUT: index where the max suffix starts
int duval(const char* T, int n) {
  // j - current index, k - max suffix candidate, o - suffix period, r -
↪   remainder (r = (j-k)%o)
  int j = 1, k = 0, o = 1, r = 0;
  while (j < n) {
    if (T[j] > T[k+r]) { k = j-r; j = k+1; o = 1; r = 0; }
    else if (T[j] < T[k+r]) { o = j-k+1; r = 0; ++j; }
    else { ++j; r = (r == o-1 ? 0 : r+1); }
  }
  return k;
}

// KMP in O(n)
// IN: T -- text, n > 0 -- length of T
// OUT: res[i] - longest strict prefix-suffix of T[0..i]
void kmp(const char* T, int n, int* res) {
  int k = res[0] = 0;
  FOR(i, 1, n-1) {
    while (k && T[i]!=T[k]) k = res[k-1];
    if (T[i]==T[k]) ++k;
    res[i] = k;
  }
}
// pattern_matching with KMP in O(n+m)
// IN: pat - pattern of length npat, text - text of length ntext
//    call kmp(pat, npat, kmp) before
// OUT: index of the first occurrence of pat in text or -1 if there is
↪   none
// Can output all the occurences after a small modification.
int kmp_match(const char* pat, int npat, const char* text, int ntext,
↪   const int* kmp) {
  int m = 0;
  REP(i, ntext) {
    while (m && text[i]!=pat[m]) m = kmp[m-1];
    if (text[i] == pat[m]) ++m;
```

```
    if (m == npat) return i-npat+1;
  }
  return -1;
}

// IN: T -- text of length n>0
// OUT: res[i] -- palindromic radius for odd palindromes centred in i
// i.e. largest j s.t. T[i-j+1]..T[i+j-1] is a palindrome
void manacher_np(const char* T, int n, int* res) {
  int t = 1, i = 0;
  while (i < n) {
    while (i-t>=0 && i+t<n && T[i-t]==T[i+t]) ++t;
    res[i] = t;
    int k = 1;
    while (k<t && res[i-k] != res[i]-k) {
      res[i+k] = min(res[i-k], res[i]-k);
      ++k;
    }
    i += k, t = max(1, t-k);
  }
}

// IN: T -- text of lenght n>0
// OUT: res[i] -- palindromic radius for even palindromes ``centred'' in
↪  i
// i.e. largest j s.t. T[i-j]..T[i+j-1] is a palindrome
void manacher_p(const char* T, int n, int *res) {
  int t = 0, i = 0;
  while (i < n) {
    while (i-t-1>=0 && i+t<n && T[i-t-1]==T[i+t]) ++t;
    res[i] = t;
    int k = 1;
    while (k<t && res[i-k] != res[i]-k) {
      res[i+k] = min(res[i-k], res[i]-k);
      ++k;
    }
    i +=k, t = max(0, t-k);
  }
}
```

7.2 Palindrome Tree

```
const int NMAX = 200000;

struct node {
  int next[26];
  int len;
  int sufflink;
  int num[2];
};

char s[NMAX + 1];

node tree[2 * NMAX + 5];
int num, lastNode;

void add(int pos, int m) {
  int cur = lastNode;
  int letter = s[pos] - 'a';

  for (;;) {
    int len = tree[cur].len;
    if (pos - 1 - len >= 0 && s[pos - 1 - len] == s[pos]) {
      break;
    }
    cur = tree[cur].sufflink;
  }

  if (tree[cur].next[letter]) {
    lastNode = tree[cur].next[letter];
    tree[lastNode].num[m]++;
    return ;
  }
  lastNode = ++num;
  tree[cur].next[letter] = lastNode;
  tree[lastNode].len = tree[cur].len + 2;
  tree[lastNode].num[m] = 1;

  if (tree[lastNode].len == 1) {
    tree[lastNode].sufflink = 2;

    return ;
  }

  for (;;) {
    cur = tree[cur].sufflink;
    int len = tree[cur].len;
    if (pos - 1 - len >= 0 && s[pos - 1 - len] == s[pos]) {
      tree[lastNode].sufflink = tree[cur].next[letter];
      break;
    }
  }
}

void init() {
  for (int i = 1; i <= num; i++) {
    memset(&tree[i], 0, sizeof(tree[i]));
  }

  num = 2; lastNode = 2;
  tree[1].len = -1; tree[1].sufflink = 1;
  tree[2].len = 0; tree[2].sufflink = 1;
}

void compute(int m) {
  int n = strlen(s);
  for (int i = 0; i < n; i++) {
    add(i, m);
  }
}

int main() {
  int nr_tests;

  scanf("%d\n", &nr_tests);

  for (int nrt = 1; nrt <= nr_tests; nrt++) {
    scanf("%s\n", s);
    init();
    compute(0);

    scanf("%s\n", s);
    compute(1);

    long long rsp = 0;
    for (int cnt = num; tree[cnt].len >= 1; cnt--) {
      rsp += (long long)tree[cnt].num[0] * tree[cnt].num[1];

      tree[tree[cnt].sufflink].num[0] += tree[cnt].num[0];
      tree[tree[cnt].sufflink].num[1] += tree[cnt].num[1];
    }

    printf("Case #%d: %lld\n", nrt, rsp);
  }

  return 0;
}
```

7.3 Suffix Tree

```
// Suffix tree in O(|s| log |Sigma|)
struct Suffix_tree{
  static const int inf;
  struct Node{
    map<int, Node*> childs;
    Node* link;
    int fpos, len;
    int l, r;
    Node(int _fpos, int _len, Node*_link=0):childs(),link(_link),
↪   fpos(_fpos), len(_len){}
  } *root, *cur;
  int gpos(Node* u){
    return u?u->fpos:0;
  }
  int glen(Node* u){
    return u?u->len:inf;
  }
  int cur_pos;
  string cur_s;

  void walk(){
    while(cur_pos > glen(cur->childs[cur_s[cur_s.size()-cur_pos]])){
      cur = cur->childs[cur_s[cur_s.size()-cur_pos]];
      cur_pos-=glen(cur);
    }
  }

  void add_char(char const&c){
    cur_s.push_back(c);
    Node* last = root;
    ++cur_pos;
    while(cur_pos){
      walk();
      Node*&v = cur->childs[cur_s[cur_s.size()-cur_pos]];
      char t = cur_s[gpos(v)+cur_pos-1];
      if(v==0){
        v = new Node(cur_s.size()-cur_pos, inf);
        last->link = cur;
        last=root;
      } else if(t==c){
        last->link = cur;
        return;
      } else {
        Node* u = new Node(gpos(v), cur_pos-1);
```

```
        u->childs[t] = v;
        u->childs[c] = new Node(cur_s.size()-1, inf);
        v->fpos+=cur_pos-1;
        v->len-=cur_pos-1;
        v=u;
        last->link = u;
        last=u;
      }
      //shouldn't happen?
      if(cur==0) cur=root;
      if(cur==root) --cur_pos;
      else cur = cur->link;
    }
  }
  Suffix_tree():root(new Node(0, 1)), cur(root), cur_pos(0){}
  Suffix_tree(string const&s):Suffix_tree(){
    for(char const&e:s) add_char(e);
  }
  void print(Node*cur, int d){
    if(cur!=root){
      cerr << string(d, ' ') << cur_s.substr(cur->fpos, cur->len) << "
↪   " << cur->l << " " << cur->r << (cur->link ? "" : "!") << "\n";
    } else cerr <<"|\n";
    for(auto const&e:cur->childs){
      print(e.second, cur==root?1:d+cur->len);
    }
  }
  void print(){
    print(root, 0);
    cerr << "last: " << cur->l << "\n";
  }
};
```

7.4 Aho Corasick

```
// aho-corasick string matching
// can be used for DP
constexpr int sigma = 26;
struct AC{
    struct Node{
        array<int,sigma> ch;
        int lll;
        array<int,sigma> link;
        int l, r;
    };
    vector<Node> nodes;
    int root = 1, pre_root = 0;
    int get_node(){
        int ret = nodes.size();
        nodes.emplace_back();
        for(auto &e:nodes.back().ch) e = -1;
        for(auto &e:nodes.back().link) e = -1;
        return ret;
    }
    AC():nodes(2){
        for(auto &e:nodes[root].link) e = pre_root;
        for(auto &e:nodes[root].ch) e = -1;
        for(auto &e:nodes[pre_root].ch) e = root;
        for(auto &e:nodes[pre_root].link) e = -1;
        nodes[root].lll = pre_root;
        nodes[pre_root].lll = -1;
    }
    int add_string(string const&s){
        int cur = root;
        for(auto &e:s){
            int x = nodes[cur].ch[e-'a'];
            if(x == -1){
                x = get_node();
                nodes[cur].ch[e-'a'] = x;
            }
            cur = x;
        }
        return cur;
    }
    void compile(){
        vector<int> ord;
        ord.push_back(root);
        for(int i=0;i<(int)ord.size();++i){
            int a = ord[i];
            Node& cur = nodes[a];
            for(int i=0;i<sigma;++i){
                int e = cur.ch[i];
                if(e!=-1){
                    Node& v = nodes[e];
                    v.lll = cur.link[i];
                    v.lll = nodes[v.lll].ch[i];
                    v.link = nodes[v.lll].link;
                    for(int j=0;j<sigma;++j){
                        if(nodes[v.lll].ch[j]!=-1){
                            v.link[j] = v.lll;
                        }
                    }
                    ord.push_back(e);
                }
            }
        }
        // pre-order on links
        vector<vector<int> > ch(nodes.size());
        for(int i=1;i<nodes.size();++i){
            ch[nodes[i].lll].push_back(i);
        }
        int tim = 0;
        function<void(int)> rec = [&](int u){
            nodes[u].l = ++tim;
            for(auto &e:ch[u]) if(e!=-1){
                rec(e);
            }
            nodes[u].r = tim;
        };
        rec(0);
    }
};
```

8 Miscellaneous

8.1 BigIntegers (c++)

```
struct Bignum {
  static const int BASE = int(1e9), DIGIT = 9;
  vector<int> val;
  bool neg;
  Bignum(int _a):val(1, abs(_a)), neg(_a<0) {}
  Bignum(const Bignum &other):val(other.val), neg(other.neg) {}
  Bignum(string s) {
    if(s.front()=='-') {
      neg=1; s.erase(s.begin());
    } else neg=0;
    int i;
    for(i = (int)s.size()-DIGIT; i>0; i-=DIGIT)
      val.push_back(stoi(s.substr(i, DIGIT)));
    val.push_back(stoi(s.substr(0, i+DIGIT)));
  }
  bool is_zero()const {
    return val.size()==1 && val[0]==0;
  }
  Bignum& reduce() {
    while(val.size()>1&&val.back()==0) val.pop_back();
    if(is_zero()) neg = false;
    return *this;
  }
  void extend(size_t siz) {
    if(val.size()<=siz) val.resize(siz+1);
    else val.push_back(0);
  }
  void swap(Bignum &other) {
    val.swap(other.val);
    std::swap(neg, other.neg);
  }
  Bignum& internal_add(Bignum const& other) {
    extend(other.val.size());
    int carry = 0;
    for(size_t i=0; i<other.val.size(); ++i) {
      val[i]+=other.val[i]+carry; carry=0;
      while(val[i]>=BASE) {
        val[i]-=BASE; ++carry;
      }
    }
    for(size_t i=other.val.size(); carry; ++i) {
      val[i]+=carry; carry=0;
      while(val[i]>=BASE) {
        val[i]-=BASE; ++carry;
      }
    }
    return reduce();
  }
  Bignum& internal_sub(Bignum const&other) {
    assert(val.size()>=other.val.size());
    int carry=0;
    for(size_t i=0; i<other.val.size(); ++i) {
      val[i]-=other.val[i]+carry; carry=0;
      while(val[i]<0) {
        val[i]+=BASE; ++carry;
      }
    }
    for(size_t i=other.val.size(); carry; ++i) {
      val[i]-=carry; carry=0;
      while(val[i]<0) {
        val[i]+=BASE; ++carry;
      }
    }
    return reduce();
  }
  bool absComp(Bignum const&other)const {
```

```
  if(val.size()!=other.val.size()) return val.size()<other.val.size();
  for(int i=(int)min(val.size(), other.val.size())-1; i>=0; --i) {
    if(val[i]<other.val[i]) return true;
    if(val[i]>other.val[i]) return false;
  }
  return false;
}
bool operator<(Bignum const&other)const {
  if(neg!=other.neg) return neg;
  return neg!=absComp(other);
}

Bignum operator+(const Bignum &other)const {
  return Bignum(*this)+=other;
}
Bignum operator+(const long long &other)const {
  return Bignum(*this)+=Bignum(other);
}
Bignum operator-(const Bignum &other)const {
  return Bignum(*this)-=other;
}
Bignum operator-(const long long &other)const {
  return Bignum(*this)-=Bignum(other);
}
Bignum& operator+=(const Bignum &other) {
  if(neg == other.neg) return internal_add(other);
  if(absComp(other)) swap(Bignum(other)+=*this);
  else internal_sub(other);
  return *this;
}
Bignum& operator-=(const Bignum &other) {
  if(neg != other.neg) return internal_add(other);
  if(absComp(other)) {
    swap(Bignum(other)-=*this);
    neg = !neg;
  } else internal_sub(other);
  return *this;
}
Bignum& operator/=(int a) {
  if(a<0) { neg=!neg; a=-a; }
  long long carry=0;
  for(int i=(int)val.size()-1; i>=0; --i) {
    carry = carry*BASE + val[i];
    val[i] = carry/a;
    carry%=a;
  }
  return reduce();
}
Bignum& operator*=(int a) {
  if(a==1) return *this;
  if(a<0) {
    neg=!neg; a=-a;
  }
  long long carry = 0;
  for(size_t i=0; i<val.size(); ++i) {
    carry+= a * (long long)val[i];
    val[i] = carry%BASE;
    carry = carry/BASE;
  }
  while(carry) {
    val.push_back(carry%BASE);
    carry/=BASE;
  }
  return reduce();
}
Bignum operator*(const Bignum&o)const {
  Bignum ret(0);
  for(int i=(int)o.val.size()-1; i>=0; --i) {
    ret*=BASE;
    ret+=(Bignum(*this)*=o.val[i]);
  }
  ret.neg^=o.neg;
  return ret.reduce();
}
Bignum operator*=(const Bignum&o) {
  ((*this)*o).swap(*this);
  return *this;
}
pair<Bignum, Bignum> remDiv(Bignum const&o)const {
  assert(!o.is_zero());
  Bignum rem(*this), ans(0);
  Bignum sub(o), add(1);
  rem.neg=sub.neg = neg;
  ans.neg=add.neg = neg^o.neg;
  while(sub.absComp(rem)) {
    sub*=2; add*=2;
  }
  while(!add.is_zero()) {
    if(!rem.absComp(sub)) {
      rem-=sub; ans+=add;
    }
    sub/=2; add/=2;
  }
  ans.reduce(); rem.reduce();
  return make_pair(ans, rem);
}
Bignum operator/(Bignum const&o)const {
  return remDiv(o).first;
}
Bignum operator%(Bignum const&o)const {
  return remDiv(o).second;
}
friend ostream& operator<<(ostream &o, Bignum const&b) {
  if(b.neg) o << '-';
  o << setfill('0') << setw(1) <<  b.val.back();
  for(int i=(int)b.val.size()-2; i>=0; --i)
    o << setw(DIGIT) << b.val[i];
  return o << setw(0);
}
long long tolong() {
  long long sign = 1-2*neg;
  if(val.size()==1)return sign*val[0];
  return sign*(val[1]*(long long)BASE+val[0]);
}
};
```

8.2 2-Sat

```
struct two_sat {
  int N; // number of variables
  vector<int> val; // assignment of x is at val[2x] and -x at val[2x+1]
  vector<char> valid; // changes made at time i are kept iff valid[i]
  vector<vector<int> > G; // graph of implications G[x][i] = y means (x
↪   -> y)

  two_sat(int N) : N(N) { // create a formula over N variables (numbered
↪   1 to N)
    val.resize(2*N);
    G.resize(2*N);
  }

  int to_ind(int x) { // converts a signed variable index to its position
↪   in val[] and G[]
    return 2*(abs(x)-1) + (x<0);
  }

  // Add the implication: a -> b
  void add_implication(int a, int b) {
    G[to_ind(a)].push_back(to_ind(b));
  }

  // Add the or-clause: (a or b)
  void add_or(int a, int b) {
    add_implication(-a,b);
    add_implication(-b,a);
  }

  // Add condition: x is true
  void add_true(int x) {
    add_or(x,x);
  }

  int time(){
      return valid.size()-1;
  }

  bool dfs(int x) {
    if(valid[abs(val[x])]) return val[x]>0;
    val[x] = time();
    val[x^1] = -time();
    for(int e:G[x])
      if(!dfs(e))
        return false;
    return true;
  }

  bool solve() {
    fill(val.begin(), val.end(), 0);
    valid.assign(1, 0);
    for(int i=0; i<val.size(); i+=2) {
      if(!valid[abs(val[i])]) {
        valid.push_back(1);
        if(!dfs(i)) {
          valid.back()=0;
          valid.push_back(1);
          if(!dfs(i+1)) return false;
        }
      }
    }
    return true;
  }
};
```

8.3 Shunting Yard (Pseudocode)

```
// Add '(' to start of expression, and ')' to end.
O = empty vector of tokens (values or operators)
S = empty stack of tokens (brackets or operators)
for each token:
  if token == value:
    O.push(token)
  else if token == '(':
    S.push(token)
  else if token == ')':
    while S.top() != '(':
      O.push(S.top())
      S.pop()
    S.pop()
  else:
    // Note: If token is a right-associative operator (^), this should be
↪   <=.
  // priority('(') < priority('+') < priority('*').
    while priority(S.top()) < priority(token):
      O.push(S.top())
      S.pop()
    S.push(token)
// Finally, evaluate O as a postfix expression.
```

9 Geometry

9.1 Exact Geometry

```
using point = complex<long long>;

long long dot(point a, point b) { return (conj(a)*b).real(); }
long long cross(point a, point b) { return (conj(a)*b).imag(); }
int signum(long long x) { return x ? (x>0 ? 1 : -1) : 0; }
// 1 if a->b is positive direction, -1 if negative, else 0
int ccw(point a, point b) { return signum(cross(a, b)); }
int ccw(point a, point b, point c) { return ccw(b-a, c-a); }

bool lexicographic_less(point a, point b)
{ return make_pair(a.real(), a.imag()) < make_pair(b.real(), b.imag()); }

// return the convex hull without collinear points
vector<point> convexhull(vector<point> pts) {
  vector<point> hull;
  sort(pts.begin(), pts.end(), lexicographic_less);
  pts.erase(unique(pts.begin(), pts.end()), pts.end());
  for (int _ : {0, 1}) { (void)_;
    size_t ms = hull.size()+1;
    for (auto const& p : pts) {
      while (hull.size() > ms && ccw(hull.rbegin()[1], hull.rbegin()[0],
↪   p) < 0)
        hull.pop_back();
      hull.push_back(p);
    }
    hull.pop_back();
    reverse(pts.begin(), pts.end());
  }
  return hull;
}

// calculates 2*<directed polygon area> (watch out for overflow)
long long polygon_area(vector<point> const& pts) {
 long long r = 0;
 for (size_t i=0; i<pts.size(); ++i)
   r += cross(pts[i], pts[(i+1)%pts.size()]);
 return r;
 // equivalent calculations which are more efficient:
 // r += (pts[(i+1)%pts.size()].real()-pts[i].real())
 //      *(pts[i].imag()+pts[(i+1)%pts.size()].imag());
 // r += pts[i].real()*(pts[(i+1)%pts.size()].imag()
 //      -pts[(i+pts.size()-1)%pts.size()].imag());
}

// rotate a point counterclockwise (ccw) or clockwise (cw)
// relative to the origin (0,0)
point rotate_ccw90(point p) { return p*point(0, 1); }
point rotate_cw90(point p) { return p*point(0, -1); }

// do the boxes (a1, a2) and (b1, b2) defined by the lower left and the
// upper right point intersect?
bool boxes_intersect(point a1, point a2, point b1, point b2) {
  return !(a1.real() > b2.real() || a2.real() < b1.real() ||
           a1.imag() > b2.imag() || a2.imag() < b1.imag());
}

bool segments_cross(point a1, point a2, point b1, point b2) {
  return // handle collinear case
      boxes_intersect({min(a1.real(), a2.real()), min(a1.imag(),
↪   a2.imag())},
                      {max(a1.real(), a2.real()), max(a1.imag(),
↪   a2.imag())},
                      {min(b1.real(), b2.real()), min(b1.imag(),
↪   b2.imag())},
                      {max(b1.real(), b2.real()), max(b1.imag(),
↪   b2.imag())}) &&
      // handle general case
      ccw(a1, a2, b1) * ccw(a1, a2, b2) <= 0 &&
      ccw(b1, b2, a1) * ccw(b1, b2, a2) <= 0;
}

// Pick's theorem: For a polygon with H holes and all vertices in
↪   lattice points
// area = no. of lattice points inside + no. of lattice points on the
↪   border/2 + H - 1

// number. of lattice points on the border
long long points_border(vector<point> const& pts) {
  long long res = 0;
  for (size_t i=0, n=pts.size(); i<n; ++i)
    res += abs(__gcd((pts[(i+1)%n] - pts[i]).real(),
                     (pts[(i+1)%n] - pts[i]).imag()));
  return res;
}

// number of lattice points inside the polygon
long long points_interior(vector<point> const& pts) {
  return (abs(polygon_area(pts)) - points_border(pts))/2 + 1;
}

// minkowski sum of convex polygons
vector<point> minkowski_sum(vector<point> const&a, vector<point>
↪   const&b){
    vector<point> x = convexhull(a), y=convexhull(b), ret;
    assert(x.size()>2&&y.size()>2);
    copy_n(vector<point>(x).begin(), 2, back_inserter(x));
    copy_n(vector<point>(y).begin(), 2, back_inserter(y));
    for(size_t i=0, j=0;i+1<x.size() && j+1 <
↪   y.size();++(ccw(x[i+1]-x[i], y[j+1]-y[j])<0?j:i))
↪   ret.push_back(x[i]+y[j]);
    return convexhull(ret);
}
```

9.2 Radial sweepline

```
// helper functions for radial sorting similar to atan2
using Point = pair<int, int>;
long long cross(Point const&a, Point const&b){
    return a.first*(long long)b.second - a.second*(long long) b.first;
}
int signum(long long x){
    return (x>0)-(x<0);
}
// check if point is in 'upper half'
bool upside(Point const&a){
    if(a.second==0) return a.first>0;
    return a.second>0;
}
// -1: <, 0: ==, 1: >
int point_comp(Point const&a, Point const&b){
    if(upside(a)!=upside(b)) return upside(a)?-1:1;
    return signum(cross(b, a));
}
bool pair_comp(pair<Point, int> const&a, pair<Point, int> const&b){
    int k = point_comp(a.first, b.first);
    if(k) return k<0;
    return a.second<b.second;
}
```

9.3 Floating Point Geometry

```
// Geometry on floating-points.

// Scalar type.
typedef double K;

// Epsilon size.
const K EPS = 1e-9;

bool is_zero(K x) { return -EPS <= x && x <= EPS; }
bool lt_zero(K x) { return x < -EPS; }
bool gt_zero(K x) { return x > EPS; }
// point/vector
struct point {
  K x, y;
  point() {}
  point(K _x, K _y): x(_x), y(_y) { }
  // SQUARE of the Euclidean norm.
  K norm() const { return x*x + y*y; }
};
point operator -(const point& p) {
  return point(-p.x, -p.y);
}
```

```
point operator +(const point& p1, const point& p2) {
  return point(p1.x+p2.x, p1.y+p2.y);
}
point operator -(const point& p1, const point& p2) {
  return point(p1.x-p2.x, p1.y-p2.y);
}
point operator *(const point& p, K t) {
  return point(p.x*t, p.y*t);
}
point operator /(const point& p, K t) {
  return point(p.x/t, p.y/t);
}
// line/segment - two points, p1 != p2
// note the default constructor doesn't construct a valid line
struct line {
  point p1, p2;
  line() { }
  line(const point& _p1, const point& _p2): p1(_p1), p2(_p2) { }
  // constructs a line out of the equation: ax + by + c = 0
  // !is_zero(a) || !is_zero(b)
  line(K a, K b, K c) {
    // if for a vertical line
    if (is_zero(b)) {
      K x = -c/a;
      p1 = point(x, 0);
      p2 = point(x, 1);
    } else {
      p1 = point(0, -c/b);
      p2 = point(1, -(c+a)/b);
    }
  }
};
// Euclidean distance between p1 and p2.
K dist(const point& p1, const point& p2) {
  return (p1-p2).norm();
}
// Scalar product
// |p1| * |p2| * cos(angle(p1, p2))
K scalar(const point& p1, const point& p2) {
  return p1.x*p2.x + p1.y*p2.y;
}
K scalar(const point& p0, const point& p1, const point& p2) {
  return scalar(p1-p0, p2-p0);
}
// Cross product
// |p1| * |p2| * sin(angle(p1, p2))
K cross(const point& p1, const point& p2) {
  return p1.x*p2.y - p1.y*p2.x;
}
// 2*directed area of (p0, p1, p2) triangle (positive <==> angle(p0, p1,
↪  p2) > 0)
K cross(const point& p0, const point& p1, const point& p2) {
  return cross(p1-p0, p2-p0);
}
enum { RIGHT = -1, STRAIGHT, LEFT };
// Direction of turn on path p0->p1->p2
// signum(cross(p0, p1, p2));
int ccw(const point& p0, const point& p1, const point& p2) {
  K cr = cross(p0, p1, p2);
  return lt_zero(cr) ? RIGHT : (gt_zero(cr) ? LEFT : STRAIGHT);
}
// angle between p1 i p2
// note atan2(y,x) returns the angle of (x,y) point on the complex plane.
// NOTE: returned angle in the interval [-M_PI, M_PI]
K angle(const point& p1, const point& p2) {
  return atan2(cross(p1, p2), scalar(p1, p2));
}
K angle(const point& p0, const point& p1, const point& p2) {
  return angle(p1-p0, p2-p0);
}
// if vectors are parallel
bool are_parallel(const point& p1, const point& p2) {
  return is_zero(cross(p1, p2));
}
// if lines are parallel
bool are_parallel(const line& U, const line& V) {
  return are_parallel(U.p2-U.p1, V.p2-V.p1);
}
// if vectors are perpendicular
bool are_perpendicular(const point& p1, const point& p2) {
  return is_zero(scalar(p1, p2));
}
// if lines are perpendicular
bool are_perpendicular(const line& U, const line& V) {
  return are_perpendicular(U.p2-U.p1, V.p2-V.p1);
}
// converts a line to equation ax + by + c == 0
void cvt_line_to_equation(const line& U, K& a, K& b, K& c) {
  a = U.p1.y-U.p2.y, b = U.p2.x-U.p1.x, c = cross(U.p1, U.p2);
}
// is point on the line
bool point_on_line(const point& p, const line& U) {
  return (is_zero(p.x-U.p1.x) && is_zero(p.y-U.p1.y)) ||
    are_parallel(p-U.p1, U.p2-U.p1);
}
// DIRECTED length of the projection of p on U (positive <==>
↪  abs(angle(U.p1, U.p1+p)) < M_PI/2)
K projection_length(const point& p, const line& U) {
  return scalar(U.p1, U.p2, U.p1+p) / sqrt(p.norm());
}
// projection of p on U
point projection_on_line(const point& p, const line& U) {
  K t = scalar(U.p1, U.p2, p) / dist(U.p1, U.p2);
  return U.p1 + (U.p2-U.p1)*t;
}
// does projection of p on line (s.p1, s.p2) lie on s
bool projection_on_segment(const point& p, const line& s) {
  return !lt_zero(scalar(s.p1, s.p2, p)) && !lt_zero(scalar(s.p2, s.p1,
↪  p));
}
// distance of point to line
K distance_point_line(const point& p, const line& U) {
  return cross(U.p1, U.p2, p) / sqrt(dist(U.p1, U.p2));
}
// distance of point to segment
K distance_point_segment(const point& p, const line& U) {
  if (projection_on_segment(p, U))
    return distance_point_line(p, U);
  return min(sqrt(dist(U.p1, p)), sqrt(dist(U.p2, p)));
}
// point symmetrical to p wrt centre s
point point_symmetry(const point& p, const point& s) {
  return s*2.0-p;
}
// point symmetrical to p wrt line U
point line_symmetry(const point& p, const line& U) {
  point s = projection_on_line(p, U);
  return point_symmetry(p, s);
}
// rotation of p around s with angle ang
point rotation(const point& p, const point& s, K ang) {
  K ksin = sin(ang), kcos = cos(ang);
  point v = p-s;
  return s + point(v.x*kcos - v.y*ksin,v.x*ksin + v.y*kcos);
}
// more precise and faster 90 degree rotation
point rotation_90(const point& p, const point& s) {
  return s + point(-(p.y-s.y), p.x-s.x);
}
// bisection of the segment
line segment_bisection(const line& U) {
  point m = (U.p1+U.p2)/2.0;
  return line(m, rotation_90(U.p2, m));
}
// intersection point of two UNPARALLEL lines
point line_line_intersection(const line& U,const line& V) {
  return V.p1 + (V.p2-V.p1)*(cross(U.p1, V.p1, U.p2) / cross(U.p2-U.p1,
↪  V.p2-V.p1));
}

struct circle {
  point s;
  // SQUARE of the radius
  K r;
  circle() { }
  circle(point& _s, K _r): s(_s), r(_r) {}
};

// returns number of intersection points (<=2), stored in res
int line_circle_intersection(const line& U, const circle& O, point* res)
↪  {
  point P = projection_on_line(O.s, U),
    PU = is_zero(dist(P, U.p1)) ? U.p2 : U.p1;
  K sd = dist(P, O.s);

  if (gt_zero(sd-O.r)) return 0;
  else if (!lt_zero(sd-O.r)) { res[0] = res[1] = P; return 1; }

  K pit_dist = O.r - sd, t = sqrt(pit_dist/dist(PU, P));
  point dP = (PU-P)*t;
  res[0] = P+dP, res[1] = P-dP;
  return 2;
}
// do s1 and s2 have nonempty intersection
bool segments_cross(const line& U, const line& V) {
  K x1 = min(U.p1.x, U.p2.x), x2 = max(U.p1.x, U.p2.x),
    x3 = min(V.p1.x, V.p2.x), x4 = max(V.p1.x, V.p2.x),
    y1 = min(U.p1.y, U.p2.y), y2 = max(U.p1.y, U.p2.y),
    y3 = min(V.p1.y, V.p2.y), y4 = max(V.p1.y, V.p2.y);

  // Check if the rectangles intersect.
  if (lt_zero(x4-x1) || gt_zero(x3-x2) || lt_zero(y4-y1) ||
↪  gt_zero(y3-y2))
    return false;

  // If collinear, the intersection would be detected earlier.
```

```
  int d1 = ccw(U.p1, U.p2, V.p1), d2 = ccw(U.p1, U.p2, V.p2);
  if ((d1 == LEFT && d2 == LEFT) || (d1 == RIGHT && d2 == RIGHT)) return
↪   false;
  d1 = ccw(V.p1, V.p2, U.p1), d2 = ccw(V.p1, V.p2, U.p2);
  if ((d1 == LEFT && d2 == LEFT) || (d1 == RIGHT && d2 == RIGHT)) return
↪   false;

  return true;
}
```

9.4 Miscellaneous Geometry

```
// C++ routines for computational geometry (excerpts)
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);}
};
const double EPS=1e-12;
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; }
// 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 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 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;
}
```

9.5 More Miscellaneous Geometry (Java)

```
// In this example, we read an input file containing three lines, each
// containing an even number of doubles, separated by commas.  The first two
// lines represent the coordinates of two polygons, given in counterclockwise
// (or clockwise) order, which we will call "A" and "B".  The last line
// contains a list of points, p[1], p[2], ...
//
// Our goal is to determine:
//   (1) whether B - A is a single closed shape (as opposed to multiple shapes)
//   (2) the area of B - A
//   (3) whether each p[i] is in the interior of B - A
//
// INPUT:
//   0 0 10 0 0 10
//   0 0 10 10 10 0
//   8 6
//   5 1
//
// OUTPUT:
//   The area is singular.
//   The area is 25.0
//   Point belongs to the area.
//   Point does not belong to the area.
import java.util.*;
import java.awt.geom.*;
import java.io.*;
public class JavaGeometry {
    // make an array of doubles from a string
    static double[] readPoints(String s) {
        String[] arr = s.trim().split("\\s++");
        double[] ret = new double[arr.length];
        for (int i = 0; i < arr.length; i++) ret[i] = Double.parseDouble(arr[i]);
        return ret;
    }
    // make an Area object from the coordinates of a polygon
    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);
    }
    // compute area of polygon
    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;
    }
    // compute the area of an Area object containing several disjoint polygons
    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;
    }
    // notice that the main() throws an Exception -- necessary to
    // avoid wrapping the Scanner object for file reading in a
    // try { ... } catch block.
    public static void main(String args[]) throws Exception {
        Scanner scanner = new Scanner(new File("input.txt"));
        // also,
        //   Scanner scanner = new Scanner (System.in);

        double[] pointsA = readPoints(scanner.nextLine());
        double[] pointsB = readPoints(scanner.nextLine());
        Area areaA = makeArea(pointsA);
        Area areaB = makeArea(pointsB);
        areaB.subtract(areaA);
        // also,
        //   areaB.exclusiveOr (areaA);
        //   areaB.add (areaA);
        //   areaB.intersect (areaA);

        // (1) determine whether B - A is a single closed shape (as
        //     opposed to multiple shapes)
        boolean isSingle = areaB.isSingular();
        // also,
        //   areaB.isEmpty();

        if (isSingle)
            System.out.println("The area is singular.");
        else
            System.out.println("The area is not singular.");

        // (2) compute the area of B - A
        System.out.println("The area is " + computeArea(areaB) + ".");

        // (3) determine whether each p[i] is in the interior of B - A
        while (scanner.hasNextDouble()) {
            double x = scanner.nextDouble();
            assert(scanner.hasNextDouble());
            double y = scanner.nextDouble();

            if (areaB.contains(x,y)) {
              System.out.println ("Point belongs to the area.");
            } else {
              System.out.println ("Point does not belong to the area.");
            }
        }

        // Finally, some useful things we didn't use in this example:
        //
        //   Ellipse2D.Double ellipse = new Ellipse2D.Double (double x, double y,
        //                                                    double w, double h);
        //
        //     creates an ellipse inscribed in box with bottom-left corner (x,y)
        //     and upper-right corner (x+y,w+h)
        //
        //   Rectangle2D.Double rect = new Rectangle2D.Double (double x, double y,
        //                                                     double w, double h);
        //
        //     creates a box with bottom-left corner (x,y) and upper-right
        //     corner (x+y,w+h)
        //
        // Each of these can be embedded in an Area object (e.g., new Area (rect)).
    }
}
```

9.6 3D Geometry

```
#define LINE 0
#define SEGMENT 1
#define RAY 2

struct point{
    double x, y, z;
    point(){};
    point(double _x, double _y, double _z){ x=_x; y=_y; z=_z; }
    point operator+ (point p) { return point(x+p.x, y+p.y, z+p.z); }
    point operator- (point p) { return point(x-p.x, y-p.y, z-p.z); }
    point operator* (double c) { return point(x*c, y*c, z*c); }
};

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

double distSq(point a, point b){
    return dot(a-b, a-b);
}

// distance from point p to plane aX + bY + cZ + d = 0
double ptPlaneDist(point p, double a, double b, double c, double d){
    return fabs(a*p.x + b*p.y + c*p.z + d) / sqrt(a*a + b*b + c*c);
}

// distance between parallel planes aX + bY + cZ + d1 = 0 and
// aX + bY + cZ + d2 = 0
double planePlaneDist(double a, double b, double c, double d1, double d2){
    return fabs(d1 - d2) / sqrt(a*a + b*b + c*c);
```

```
}

// square distance between point and line, ray or segment
double ptLineDistSq(point s1, point s2, point p, int type){
    double pd2 = distSq(s1, s2);
    point r;
    if(pd2 == 0)
  r = s1;
    else{
  double u = dot(p-s1, s2-s1) / pd2;
  r = s1 + (s2 - s1)*u;
  if(type != LINE && u < 0.0)
      r = s1;
  if(type == SEGMENT && u > 1.0)
      r = s2;
    }
    return distSq(r, p);
}

double signedTetrahedronArea(point A, point B, point C, point D) {
  double A11 = A.x - B.x;
  double A12 = A.x - C.x;
  double A13 = A.x - D.x;
  double A21 = A.y - B.y;
  double A22 = A.y - C.y;
  double A23 = A.y - D.y;
  double A31 = A.z - B.z;
  double A32 = A.z - C.z;
  double A33 = A.z - D.z;
  double det =
    A11*A22*A33 + A12*A23*A31 +
    A13*A21*A32 - A11*A23*A32 -
    A12*A21*A33 - A13*A22*A31;
  return det / 6;
}

// Parameter is a vector of vectors of points - each interior vector
// represents the 3 points that make up 1 face, in any order.
// Note: The polyhedron must be convex, with all faces given as
↪  triangles.
double polyhedronArea(vector<vector<point> > poly) {
  int i,j;
  point cent(0,0,0);
  for (i=0; i<poly.size(); i++)
    for (j=0; j<3; j++)
      cent=cent+poly[i][j];
  cent=cent*(1.0/(poly.size()*3));
  double v=0;
  for (i=0; i<poly.size(); i++)
    v+=fabs(signedTetrahedronArea(cent,poly[i][0],
↪  poly[i][1],poly[i][2]));
  return v;
}
```

9.7 3D Convex hull

```
using Point = array<long long, 3>;
int signum(long long x) { return x ? (x>0 ? 1 : -1) : 0; }
long long sq(long long x){ return x*x; }
Point operator-(Point const&a, Point const&b){
    return {a[0]-b[0], a[1]-b[1], a[2]-b[2]};
}
Point operator*(Point const&a, Point const&b){
    return {a[1]*b[2]-a[2]*b[1], a[2]*b[0]-a[0]*b[2],
↪  a[0]*b[1]-a[1]*b[0]};
}
long long dot(Point const&a, Point const&b){
    return a[0]*b[0]+a[1]*b[1]+a[2]*b[2];
}
// side of d relative to plane through a,b,c
int side(Point const&a, Point const&b, Point const&c, Point const&d){
    return signum(dot(a-d, (b-d)* (c-d)));
}
double dist(Point const&a, Point const&b, Point const&c, Point const&d){
    Point n = (b-a)*(c-a);
    return dot(n, d-a)/sqrt(dot(n, n));
}
// use long double or __int128 for coordinates >3e4
int slope(Point const&a, Point const&b, Point const&c){
    return signum(sq(b[0]-a[0])*(sq(c[1]-a[1])+sq(c[2]-a[2])) -
↪  sq(c[0]-a[0])*(sq(b[1]-a[1])+sq(b[2]-a[2])));
}
// 3d convex hull, doesn't handle degenerate cases.
vector<array<int, 3> > convexhull3d(vector<Point> const&pts){
    int N = pts.size();
    vector<vector<char> > vis(N, vector<char>(N, 0));
    vector<array<int, 3> > ret;
    int i = min_element(pts.begin(), pts.end())-pts.begin(), j=(i==0);
    for(int k=0;k<N;++k) if(k!=i && slope(pts[i], pts[j], pts[k])>0) j=k;
    stack<pair<int, int> > s;
    s.emplace(i, j);
    while(!s.empty()){
        tie(i, j) = s.top();
        s.pop();
        if(vis[i][j]) continue;
        vis[i][j]=1;
        int k = 0;
        while(k==i||k==j) ++k;
        for(int l=0;l<N;++l){
            if(l==i || l==j || l==k) continue;
            if(side(pts[i], pts[j], pts[k], pts[l])>0) k=l;
        }
        ret.push_back({i, j, k});
        vis[k][i]=vis[j][k]=1;
        s.emplace(k, j);
        s.emplace(i, k);
    }
    return ret;
}
```

9.8 Fast Delaunay triangulation

```
// Delaunay triangulation in Theta(N^2)
// May handle degenerate cases
// Coordinates shouldn't go over 10^6
#include <bits/stdc++.h>
using namespace std;
const long long BOUND = (long long)1e9;
using Point =  complex<long long>;
struct Edge
{
  Point p1, p2;
  Edge(const Point &_p1, const Point &_p2) : p1(_p1), p2(_p2) {};
  Edge(const Edge &e) : p1(e.p1), p2(e.p2) {};

  friend ostream &operator << (ostream &str, Edge const &e) {
    return str << "Edge " << e.p1 << ", " << e.p2;
  }
  bool operator == (const Edge & e2) const{
    return         (p1 == e2.p1 && p2 == e2.p2) || (p1 == e2.p2 && p2 ==
↪  e2.p1);
  }
};
bool is_BOUND(long long val){
    return val<-BOUND/2 || val > BOUND/2;
}
bool is_on_boundry(Point const &p){
    return is_BOUND(p.real())||is_BOUND(p.imag());
  }
bool is_left_of(Point const &a, Point const&b, Point const&c){
    Point tmp = ((b-c)*conj(a-c));
    if(tmp.imag()<0) return true;
    return tmp.imag()==0&&((c-a)*conj(b-a)).real()>0 &&
↪  ((c-a)*conj(c-a)).real()<((b-a)*conj(b-a)).real();
}

struct Triangle{
  Point p1, p2, p3;
  float xn, yn, zn;
  Triangle(const Point &_p1, const Point &_p2, const Point
↪  &_p3):       p1(_p1), p2(_p2), p3(_p3) {
    if(is_left_of(p1, p2, p3)) swap(p2, p3);
    float z1 = (p1*conj(p1)).real(), z2 = (p2*conj(p2)).real(), z3 =
↪  (p3*conj(p3)).real();
    xn = (p2.imag()-p1.imag())*(z3-z1) - (p3.imag()-p1.imag())*(z2-z1);
    yn = (p3.real()-p1.real())*(z2-z1) - (p2.real()-p1.real())*(z3-z1);
    zn = (p2.real()-p1.real())*(p3.imag()-p1.imag()) -
↪  (p3.real()-p1.real())*(p2.imag()-p1.imag());
  }
  bool containsVertex(const Point &v) {
    return p1 == v || p2 == v || p3 == v;
  }
  bool circumCircleContains(const Point &v)const {
    if(is_on_boundry(p1)) return is_left_of(p3, p2, v);
    if(is_on_boundry(p2)) return is_left_of(p1, p3, v);
    if(is_on_boundry(p3)) return is_left_of(p2, p1, v);
    if(is_on_boundry(v)) return false;
    long long z1 = (p1*conj(p1)).real(), z4 = (v*conj(v)).real();
    long long res = (v.real()-p1.real())*xn + (v.imag()-p1.imag())*yn +
↪  (z4-z1)*zn;
    return res <0;
  }
  friend ostream &operator << (ostream &str, const Triangle & t) {
    return str << "Triangle:" << endl << "\t" << t.p1 << endl << "\t" <<
↪  t.p2 << endl << "\t" << t.p3 << endl;
  }
  bool operator == (const Triangle &t2) {
    return        (p1 == t2.p1 || p1 == t2.p2 || p1 == t2.p3) &&
        (p2 == t2.p1 || p2 == t2.p2 || p2 == t2.p3) &&
        (p3 == t2.p1 || p3 == t2.p2 || p3 == t2.p3);
  }
};
struct edgeHasher{
  size_t operator()(const Edge&e)const{
```

```
    long long hash = e.p1.real() + e.p2.real();
    hash ^= (hash<<6) + (hash>>2)+e.p1.imag()+e.p2.imag();
    return hash ^ (hash>>32);
  }
};

struct Delaunay
{
  vector<Triangle> _triangles;

  const vector<Triangle>& triangulate(vector<Point> const&vertices) {
    Point p1( -BOUND, -BOUND), p2(0.0, BOUND), p3(BOUND, -BOUND);
    _triangles.push_back(Triangle(p1, p2, p3));

    for(auto const&p:vertices) {
      vector<Edge> polygon;
      auto it2 = _triangles.begin();
      for(auto it = _triangles.begin();it!=_triangles.end();++it) {
        if(it->circumCircleContains(p)) {
          polygon.emplace_back(it->p1, it->p2);
          polygon.emplace_back(it->p2, it->p3);
          polygon.emplace_back(it->p3, it->p1);
        } else {
          *(it2++)=*it;
        }
      }
      _triangles.erase(it2, _triangles.end());

      unordered_map<Edge, int, edgeHasher> badEdges;
      for(auto const &e:polygon){
        ++badEdges[e];
      }
      polygon.erase(remove_if(begin(polygon), end(polygon),
↪     [&badEdges](Edge const&e){return badEdges[e]>1;}), end(polygon));
      for(auto e = begin(polygon); e != end(polygon); e++)
        _triangles.push_back(Triangle(e->p1, e->p2, p));
    }

    _triangles.erase(remove_if(begin(_triangles), end(_triangles), [p1,
↪     p2, p3](Triangle &t){
      return t.containsVertex(p1) || t.containsVertex(p2) ||
↪     t.containsVertex(p3);
    }), end(_triangles));

    return _triangles;
  }
};
```

9.9 Plane from Points, and 3D Cross Product (Java)

```
static long[] x, y, z;
static long Cx=0, Cy=0, Cz=0;

public static void main(String[] args) {
  //Given some points, do they all lie in the same plane?
  if(n <= 3) {
    System.out.println("YES");
    return;
  }

  //Find a third point not collinear to the first two
  int third = -1;

  for(int i=2; i < n; i++) {
    cp(0, 1, i);
    if(Cx == 0 && Cy == 0 && Cz == 0) continue;

    third = i;
    break;
  }

  if(third == -1) {
    System.out.println("YES");
    return;
  }
  int t = third;

  // Plane equation, Ax + By + Cz + D = 0, given three 3D points.
  long A = y[0]*(z[1]-z[t]) + y[1]*(z[t]-z[0]) + y[t]*(z[0]-z[1]);
  long B = z[0]*(x[1]-x[t]) + z[1]*(x[t]-x[0]) + z[t]*(x[0]-x[1]);
  long C = x[0]*(y[1]-y[t]) + x[1]*(y[t]-y[0]) + x[t]*(y[0]-y[1]);

  long D = x[0]*(y[1]*z[t]-y[t]*z[1]) + x[1]*(y[t]*z[0]-y[0]*z[t])
      + x[t]*(y[0]*z[1]-y[1]*z[0]);
  D = -D;

  // All aboard the plane!
  for(int i=third+1; i < n; i++) {
    long X = A*x[i] + B*y[i] + C*z[i] + D;
    if(X != 0) {
      System.out.println("NO");
      return;
    }
  }
  System.out.println("YES");
}

public static void cp(int i, int j, int k) {
  long Ax = x[j] - x[i],  Ay = y[j] - y[i],  Az = z[j] - z[i];
  long Bx = x[k] - x[i],  By = y[k] - y[i],  Bz = z[k] - z[i];
  Cx = Ay*Bz - By*Az;
  Cy = Az*Bx - Bz*Ax;
  Cz = Ax*By - Bx*Ay;
}
```

9.10 Line-Sphere Intersection (Java)

```
//p is a vector showing some point on the line
//l is a vector showing the direction of the line
//s is a vector showing the center of the sphere
//r is the radius of the sphere

static boolean pos, neg; //Solutions

public static boolean hits(double[] p, double[] l, double[] s, double r)
↪   {
  //Normalize l into a unit vector
  double norml = Math.sqrt(l[0]*l[0] + l[1]*l[1] + l[2]*l[2]);
  l[0] /= norml;
  l[1] /= norml;
  l[2] /= norml;

  //We're going to treat the line as though it went through (0, 0, 0)
  //So we modify the sphere center accordingly
  double[] c = { s[0] - p[0], s[1] - p[1], s[2] - p[2] };

  //The part under the radical
  double inside = dot(l, c)*dot(l, c) - dot(c, c) + r*r;

  //inside < 0 means no solution (no intersection)
  if(inside < 0) return false;

  //Get solutions
  //(p + pos*l) and (p + neg*l) are the two intersection points
  pos = dot(l, c) + Math.sqrt(inside);
  neg = dot(l, c) - Math.sqrt(inside);
  return true;
}

public static double dot(double[] a, double[] b) { return a[0]*b[0] +
↪   a[1]*b[1] + a[2]*b[2]; }
```

9.11 3D Segment Distance (Java)

```
// Input:  two 3D line segments S1 and S2
// Return: the shortest distance between S1 and S2
float dist3D_Segment_to_Segment(Segment S1, Segment S2) {
  Vector u = S1.P1 - S1.P0;
  Vector v = S2.P1 - S2.P0;
  Vector w = S1.P0 - S2.P0;
  float  a = dot(u,u);   // always >= 0
  float  b = dot(u,v);
  float  c = dot(v,v);   // always >= 0
  float  d = dot(u,w);
  float  e = dot(v,w);
  float  D = a*c - b*b;  // always >= 0
  float  sc, sN, sD = D; // sc = sN / sD, default sD = D >= 0
  float  tc, tN, tD = D; // tc = tN / tD, default tD = D >= 0

  // compute the line parameters of the two closest points
  if (D < SMALL_NUM) { // the lines are almost parallel
    sN = 0.0;    // force using point P0 on segment S1
    sD = 1.0;    // to prevent possible division by 0.0 later
    tN = e;
    tD = c;
  } else {        // get the closest points on the infinite lines
    sN = (b*e - c*d);
    tN = (a*e - b*d);
    if (sN < 0.0) {     // sc < 0 => the s=0 edge is visible
      sN = 0.0;
      tN = e;
      tD = c;
    } else if (sN > sD) {  // sc > 1 => the s=1 edge is visible
      sN = sD;
      tN = e + b;
      tD = c;
    }
  }

  if (tN < 0.0) { // tc < 0 => the t=0 edge is visible
    tN = 0.0;
    // recompute sc for this edge
```

```
    if (-d < 0.0) {
      sN = 0.0;
    } else if (-d > a) {
      sN = sD;
    } else {
      sN = -d;
      sD = a;
    }
  }
  else if (tN > tD) { // tc > 1 => the t=1 edge is visible
    tN = tD;
    // recompute sc for this edge
    if ((-d + b) < 0.0)
      sN = 0;
    else if ((-d + b) > a)
      sN = sD;
    else {
      sN = (-d + b);
      sD = a;
    }
  }
  // finally do the division to get sc and tc
  sc = (abs(sN) < SMALL_NUM ? 0.0 : sN / sD);
  tc = (abs(tN) < SMALL_NUM ? 0.0 : tN / tD);

  // get the difference of the two closest points
  Vector dP = w + (sc * u) - (tc * v);  // = S1(sc) - S2(tc)
  return norm(dP);   // return the closest distance
}
```

9.12 2D Centroid (Text)

$C_x = \frac{1}{6A}\sum_{i=0}^{n-1}(x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$,
$C_y = \frac{1}{6A}\sum_{i=0}^{n-1}(y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$,
$A = \frac{1}{2}\sum_{i=0}^{n-1}(x_i y_{i+1} - x_{i+1} y_i)$ Note that A is signed area.

10 Number Theory

10.1 Polynomial Coefficients (Text)

Given a polynomial $(x_1 + x_2 + ... + x_k)^n$, the coefficient on the term $x_1^{c_1} * x_2^{c_2} * ... x_k^{c_k}$ is $n!/(c_1! * c_2! * ... * c_k!)$ The simple form for binomial coefficients is: $n!/(c_1! * c_2!)$ ...which is of course $\binom{n}{c_1}$

10.2 Mobius Function (Text)

$\mu(n) = 1$ if n is a square-free positive integer with an even number of prime factors.
$\mu(n) =$ -1 if n is a square-free positive integer with an odd number of prime factors.
$\mu(n) = 0$ if n is not square-free.
The Mobius function is multiplicative ($\mu(ab) = \mu(a) * \mu(b)$ whenever a and b are coprime).
For all d dividing n, the sum of $\mu(d)$ is 0 if $n > 1$, and 1 if $n = 1$.

Mobius Inversion

If $g(n) = \sum_{d|n} f(d)$ for all $n \geq 1$, then $f(n) = \sum_{d|n} \mu(d) g(n/d)$ for all $n \geq 1$.

```
// compute sum_i={1...n} f(i) where f is multiplicative,
// assuming sum_i={1...n} g(i) and sum_i={1...n} f*g
// can be computed quickly (f*g is dirichlet convolution)
// takes O(n^{2/3}) time in total
// good candidates for f and f*g are:
// f(x) = x^k, f(x) = (x==1), f(x)=1
// FOR BIG NUMERS ADD MOD
struct Dirichlet_Sum{
    using func  = ll(*)(ll);
    ll n, th; //threshold, around n^{2/3}
    func sum_f; // prefix for f (1<=x<=th)
    func sum_g; // prefix for g (0<=x<=n)
    func sum_fg; // prefix for f*g (0<=x<=n)
    unordered_map<ll, ll> cache;

    Dirichlet_Sum(func s_f, func s_g, func s_fg):
        sum_f(s_f), sum_g(s_g), sum_fg(s_fg){}

    ll calc_rec(ll x){
        if(x<=th) return sum_f(x);
        auto it = cache.find(x);
        if(it!=cache.end()) return it->second;
        ll ret = sum_fg(x);
        for(ll i=2, nex;i<=x;i=nex+1){
            nex = x/(x/i);
            ret-= (sum_g(nex) - sum_g(i-1))*calc_rec(x/i);
        }
        ret/=sum_g(1);
        return cache[x] = ret;
    }
    ll get_sum(ll _n, ll _th){
        if(_n<=0) return 0;
        n = _n; th=_th;
        return calc_rec(n);
    }
};
// computes prefix sums of multiplicative function f
// takes O(n log n) time
struct Linear_Sieve{
    using func  = ll(*)(ll, ll, int);
    int n;
    func f; // f(p^k, p, k)
    vector<ll> sum;
    Linear_Sieve(func _f):f(_f){}
    void compute(int _n){
        n=_n; sum.assign(n, 0);
        vector<char> isp(n, 1);
        sum[1] = f(1, 1, 0);
        for(int i=2;i<n;++i){
            if(isp[i]){
                ll pk = i;
                for(int k=1;pk<n;++k, pk*=i){
                    sum[pk] = f(pk, i, k);
                    for(int j=2;j<=(n-1)/pk;++j){
                        isp[j*pk] = 0;
                        sum[j*pk] = sum[j]*sum[pk];
                    }
                }
            }
        }
        partial_sum(sum.begin(), sum.end(), sum.begin());
    }
};
// example for euler totient
ll sg(ll x){return x;}
ll sfg(ll x){return x*(x+1)/2;}
Linear_Sieve ls([](ll pk, ll p, int){return pk==1?1:pk-pk/p;});
ll sf(ll x){return ls.sum[x];}
```

10.3 Burnside's Lemma (Text)

$|X/G| = \frac{1}{G}\sum_{g \in G}|X^g|$

G is a finite group that acts on a set X. For each $g \in G$ let X^g denote the set of elements in X that are fixed by g.

A standard problem that is best solved using Burnside's lemma is: consider a circular stripe of n cells. How many ways are there to color these cells with two colors, black and white, up to a rotation? Here, X is a set of all colored stripes (it has 2^n elements), and G is the group of its rotations (it has n elements: rotation by 0 cells, by 1 cell, by 2 cells, etc, by $n-1$ cells), and an orbit is exactly the set of all stripes that can be obtained from each other using rotations, so the number of orbits will be the number of distinct stripes up to a rotation. Now let's apply the lemma, and find the number of stripes that are fixed by the rotation by K cells. If a stripe becomes itself after rotating by K cells, then its 1st cell must have the same color as its (1+K modulo n)-th cell, which is in turn the same as its (1+2K modulo n)-th cell, etc, until we get back to the 1st cell when m*K modulo n=0. One may notice that this will happen when $m = n/\gcd(K, n)$, and thus we get $n/\gcd(K, n)$ cells that must all be of the same color. Analogously, the same amount of cells must be of the same color starting with cell 2, (2+K modulo n), etc. Thus, all cells are separated into $\gcd(K, n)$ groups, with each group being of one color, and that yields us $2^{\gcd(K,n)}$ choices. An by Burnside's lemma, the answer to the original problem is $sum(2^{\gcd(K,n)})/n$, where the sum is taken over K from 0 to $n-1$.

That was rather complicated; here's a somewhat simpler example: Consider a square of $2n$ times $2n$ cells. How many ways are there to color it into X colors, up to rotations and/or reflections? Here, the group has only 8 elements (rotations by 0, 90, 180 and 270 degrees, reflections over two diagonals, over a vertical line and over a horizontal line). Every coloring stays itself after rotating by 0 degrees, so that rotation has X^{4n^2} fixed points. Rotation by 180 degrees and reflections over a horizonal/vertical line split all cells in pairs that must be of the same color for a coloring to be unaffected by such rotation/reflection, thus there exist X^{2n^2} such colorings for each of them. Rotations by 90 and 270 degrees split cells in groups of four, thus yielding X^{n^2} fixed colorings. Reflections over diagonals split cells into $2n$ groups of 1 (the diagonal itself) and $2n^2 - n$ groups of 2 (all remaining cells), thus yielding $X^{2n^2-n+2n} = X^{2n^2+n}$ unaffected colorings. So, the answer is $(X^{4n^2} + 3X^{2n^2} + 2X^{n^2} + 2X^{2n^2+n})/8$.

11 Data Structures

11.1 Treap/Cartesian Tree

```
// Treap: Data structure like std::set that keeps all elements in
// sorted order.
//
// It supports the following operations:
// - insert/lower_bound/find: O(log n)
// - erase: O(log n)
// - kth: O(log n) -- access to the k-th smallest element
//
// Treaps are a binary search tree where the keys (x) satisfy a binary
// search tree property and the priorities (y) the heap property.

struct node {
  explicit node(int x) : y(rand()), x(x), cnt(1), left(0), right(0) {}
  int y, x, cnt;
  node *left, *right;
  node *update_cnt() {
    cnt = 1 + (left?left->cnt:0) + (right?right->cnt:0); return this;
  }
  node *rotate_right() {
    node *root=left; left=left->right; root->right=this;
    update_cnt(); return root->update_cnt();
  }
  node *rotate_left() {
    node *root=right; right=right->left; root->left=this;
    update_cnt(); return root->update_cnt();
  }
  node *insert(node * n) {
    if (n->x < x) {
      left = left ? left->insert(n) : n;
      return n->y > y ? rotate_right() : update_cnt();
    }
    right=right?right->insert(n):n;
    return n->y > y ? rotate_left() : update_cnt();
  }
  node *erase(int v) {
    if (v != x) {
      if (v < x && left)  return (left = left->erase(v)), update_cnt();
      if (v > x && right) return (right=right->erase(v)), update_cnt();
      return this;
    }
    if (!left)  { node *r=right; delete this; return r; }
    if (!right) { node *l=left;  delete this; return l; }
    if (left->y >= right->y) {
      node *root = rotate_right();
      root->right = root->right->erase(v);
      return root->update_cnt();
    } else {
      node *root = rotate_left();
      root->left = root->left->erase(v);
      return root->update_cnt();
    }
  }
  bool find(int v) {
    return(v<x)&&left&&left->find(v)
↪   ||x==v||(v>x)&&right&&right->find(v);
  }
  int lower_bound(int v) {
    if (v <= x) return left ? left->lower_bound(v) : 0;
    return 1 + (left?left->cnt:0) + (right?right->lower_bound(v):0);
  }
  int kth(int at) {
    int left_sz = left ? left->cnt : 0;
    if (at == left_sz) return x;
    if (at <  left_sz) return left->kth(at);
    return right ? right->kth(at - left_sz - 1) : -1;
  }
};
struct treap {
  treap() : root(0) {}
  node *root;
  int kth(int at) { return root?root->kth(at) : -1; }
  void erase(int v) { root=root?root->erase(v):0; }
  void insert(int v) { root=(!root?new node(v):root->insert(new
↪   node(v))); }
  int lower_bound(int v) { return root?root->lower_bound(v):0; }
  bool find(int v) { return root && root->find(v); }
};
```

11.2 Implicit Treap/Cartesian Tree

```
// By sorting the elements by their position, treaps can be used to
// implement arrays that support the following operations:
// - insert at index: O(log n)
// - erase at index: O(log n)
//
// Those operations are built upon the more general operations:
// - concat of two treaps: O(log n)
// - split of two treaps by index: O(log n)
//
// This treap is implemented with split/merge instead of rotations.

template <typename T, typename... Args>
unique_ptr<T> make_unique(Args&&... args) {
  return unique_ptr<T>(new T(std::forward<Args>(args)...));
}

template <typename T>
struct treap {
  struct node {
    int y;
    T x;
    size_t cnt;
    unique_ptr<node> l, r;

    node(T x) : y(rand()), x(x), cnt(1) {}

    size_t left_cnt() const { return l ? l->cnt : 0; }
    size_t right_cnt() const { return r ? r->cnt : 0; }
    void update_cnt() { cnt = 1 + left_cnt() + right_cnt(); }
  };

  unique_ptr<node> merge(unique_ptr<node> l, unique_ptr<node> r) {
    if (!l) return r;
    if (!r) return l;
    if (l->y < r->y) {
      l->r = merge(move(l->r), move(r));
      l->update_cnt();
      return l;
    } else {
      r->l = merge(move(l), move(r->l));
      r->update_cnt();
      return r;
    }
  }

  pair<unique_ptr<node>, unique_ptr<node> >
  split(unique_ptr<node> v, int index) {
    if (!v) { return {nullptr, nullptr}; }
    int lcnt = v->left_cnt();
    unique_ptr<node> l, r;
    if (index == lcnt) {
      l = move(v->l);
      v->update_cnt();
      r = move(v);
    } else if (index > lcnt) {
      tie(l, r) = split(move(v->r), index - lcnt - 1);
      v->r = move(l);
      v->update_cnt();
      l = move(v);
    } else {
      tie(l, r) = split(move(v->l), index);
      v->l = move(r);
      v->update_cnt();
      r = move(v);
    }
    return {move(l), move(r)};
  }

  unique_ptr<node> root;

  void insert(int index, T value) {
    unique_ptr<node> l, r;
    tie(l, r) = split(move(root), index);
    root = merge(merge(move(l), make_unique<node>(value)), move(r));
  }
  T erase(int index) {
    unique_ptr<node> l, m, r;
    tie(l, m) = split(move(root), index);
    tie(m, r) = split(move(m), 1);
    root = merge(move(l), move(r));
    return m->x;
  }

  size_t size() const { return root ? root->cnt : 0; }
};
```

12 Formulas

12.1 Binomial coefficients

$$\sum_{k=0}^{n} \binom{k}{c} = \binom{n+1}{c+1}$$
$$\sum_{k=0}^{n} \binom{r+k}{k} = \binom{r+n+1}{n}$$
$$\sum_{k=0}^{m} \binom{n-k}{m-k} = \binom{n+1}{m}$$
$$\sum_{k=0}^{n} \binom{n-k}{k} = f_{n+1}$$
$$\sum_{j=0}^{n} \binom{n}{j}\binom{m}{k-j} = \binom{n+m}{k}$$

where $f_1 = f_2 = 1, f_{n+2} = f_{n+1} + f_n$ for all $n \geq 1$.

12.2 Sums of powers

$$\sum_{k=0}^{n} k^1 = \frac{n(n+1)}{2}$$
$$\sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$
$$\sum_{k=0}^{n} k^3 = \frac{n^2(n+1)^2}{4}$$

12.3 Sums with floor function

```
// sum K=1...N floor(kp/q)
long long floor_sum(long long N, long long p, long long q){
    long long t = __gcd(p, q);
    p/=t, q/=t;
    long long s=0, z=1;
    while(q>0 && N>0){
        t = p/q;
        s+= N*(N+1)/2*z*t;
        p-= q*t;
        t = N/q;
        s+= z*p*t*(N+1) - z*t*(p*q*t+p+q-1)/2;
        N-= q*t;
        t = N*p/q;
        s+= z*t*N;
        N = t;
        swap(p, q);
        z*= -1;
    }
    return s;
}
// number of integer points the triangle
// ax + by <=c && x, y > 0; where a, b, c > 0
int64_t count_triangle(int64_t a, int64_t b, int64_t c){
    if(b>a) swap(a, b);
    int64_t m = c/a;
    if(a==b) return m*(m-1)/2;
    int64_t k= (a-1)/b, h = (c-a*m)/b;
    return m*(m-1)/2*k + m*h + count_triangle(b, a-b*k, c-b*(k*m+h));
}
```