Template

```
#include <bits/stdc++.h>
using namespace std;
#define REP(i,a,b) for (int i = (a); i <= (b); ++i)
#define REPD(i,a,b) for (int i = (a); i >= (b); --i)
#define FORI(i,n) REP(i,1,n)
#define FOR(i,n) REP(i,0,int(n)-1)
#define mp make_pair
#define pb push_back
#define pii pair<int,int>
#define vi vector<int>
#define ll long long
#define SZ(x) int((x).size())
#define DBG(v) cerr << #v << " = " << (v) << endl;
#define FOREACH(i,t) for (auto i = t.begin(); i != t.end(); ++i)
#define fi first
#define se second

int main () {
}
```

Template replicator, Makefile, practise sessions

```
for i in {a..k}; do cp temp.cpp $i.cpp; touch $i.in; done

CXXFLAGS=-O2 -std=gnu++0x -W -Wall -Wshadow -Wconversion -Wfloat-equal
                                                    -U_FORTIFY_SOURCE
```

On practice session: (1) check if __int128 works: (or maybe __int128_t)
`__int128 x = 123456789123456789LL; cout << (((x*x) % 355) == 16);`
(2) stack limits?
(3) check if **`cerr << "lol";`** gives WA (should not)

Hashmap (and a few large primes)

```
struct hasz { size_t operator() (const pair<ll,ll>& a) const {
   return a.fi % 1394063599LL + a.se % 2394067599LL;
} };
unordered_multiset<pair<ll,ll>,hasz> st;
unordered_map<pair<ll,ll>,int,hasz> hm;

// some other primes:
// 1000035677, 1000035689, 1000035697, 1000035733, 1000035737
```

Tips for debugging

1. Is there no overflow in int? Better only use ints for indices.
2. You know, pii also contains an int. Same for FORI, FOR, REP.
3. Is INF big enough?
4. Are arrays large enough? If n <= 100000, m <= 300000 – just use 300000 everywhere
5. (-5) % 3 = -2, not 1. Always do **if (res < 0) res += mod;** before printfing.
6. (-1) / 2 = 0, not -1.
7. int(-0.01) = 0, not -1. Better use floor.
8. If you're using a set – should it be a multiset?
9. In a multiset, s.erase(x) erases all x's, s.erase(s.find(x)) – just one.
10. If multiple testcases in one file, remember to clear global vectors/sets/arrays.
11. If there is an n x m grid – easy to confuse n with m.
12. Consider boundary cases of inputs.
13. Consider boundary cases of outputs.
14. Binary searching on doubles? Just do 100 iterations.
15. Before submitting, try to imagine you got WA. Can you *immediately* see why?
16. Do a maxtest even if you don't know the answer – maybe the program will RE, output something absurd, or break an assertion.
17. Examine *break*/*continue*/*return* statements in your code. Can they be problematic (if multiple testcases – do you always read all the input)?

Graphs

BFS

```
// returns a vector of distances to source (INF = unreachable)
vi bfs (int n, int source, vi *adj) {
   vi dist(n+1,INF);
   deque<int> q;
   q.pb(source);
   dist[source] = 0;
   while (!q.empty()) {
      int v = q.front(); q.pop_front();
      FOREACH(it,adj[v]) if (dist[*it] == INF) {
         dist[*it] = dist[v] + 1;
         q.pb(*it);
      }
   }
   return dist;
}
```

DFS

```
// input: graph on vertices 1..n. computes enter & exit times and parent in DFS tree
// src = -1 => explore all connected components
#define N 1000003
int enterT[N], exitT[N], parent[N], t;
void dfs(int v, vi *adj) {
   enterT[v] = ++t;
   FOREACH(it,adj[v]) if (parent[*it] == -1) { parent[*it] = v; dfs(*it,adj); }
   exitT[v] = ++t;
}
void runDfs (int n, vi *adj, int src = -1) {
   t = -1; FORI(i,n) enterT[i] = exitT[i] = parent[i] = -1;
   if (src == -1) { FORI(i,n) if (parent[i] == -1) dfs(i,adj); } else dfs(src,adj);
}
```

2COL (bipartiteness)

```
// twocol(n,adj) takes a graph on vertices 1..n and tries to 2-color it
// if it fails, it returns empty vector
vi twocol (int n, vi *adj) {
   vi ans(n+1,2);
   FORI(i,n) {
      if (ans[i] != 2) continue;
      deque<pii> q;
      q.pb(mp(i,0));
      while (!q.empty()) {
         int v = q.front().fi, col = q.front().se;
         q.pop_front();
         if (ans[v] == 2) {
            ans[v] = col;
            FOREACH(it,adj[v]) q.pb(mp(*it,1-col));
         } else if (ans[v] != col) return vi();
      }
   }
   return ans;
}
```

Dijkstra

```
// takes directed weighted graph on vertices 1..n. output:
// dist[i] – distance from src
// parent[i] – parent in the tree of shortest distances
#define MAXN 100006
int dist[MAXN], parent[MAXN];
struct comp { inline bool operator () (int v, int w) const {
   if (dist[v] != dist[w]) return dist[v] < dist[w]; return v < w;
} };
void dijkstra (int n, vector<pii> *adj, int src) {
   FORI(i,n) { dist[i] = INF; parent[i] = -1; }
   dist[src] = 0; set<int,comp> q; q.insert(src);
   while (!q.empty()) {
      int v = *(q.begin()); q.erase(q.begin());
      FOREACH(it,adj[v]) {
         int d = dist[v] + it->se, w = it->fi;
         if (d < dist[w]) { q.erase(w); dist[w] = d; q.insert(w); parent[w] = v; }
}  }  }
```

Bellman-Ford

```
// takes directed weighted graph on vertices 1..n. output:
// if there is a negative-weight cycle: vi()
// otherwise: vector of distances from src
vi bellmanFord (int n, vector<pii> *adj, int src) {
   vi dist(n+1,INF); dist[src] = 0;
   FOR(it,n) {
      bool changed = 0;
      FORI(v,n) FOREACH(x,adj[v]) {
         if (dist[v] + x->se < dist[x->fi]) {
            dist[x->fi] = dist[v] + x->se;
            changed = 1;
         }
      }
      if (!changed) return dist;
   }
```

```
   return vi();
}
```

Prim (MST)

```
// input: (n,adj) – connected graph on vertices 1..n
// output: cost of MST. Also leaves the MST (set of edges <from,to>) in mst vector
vector<pii> mst;
ll prim2 (int n, vector<pii> *adj) {
   ll cost = 0; mst.clear(); vector<bool> done(n+1,0);
   priority_queue<pair<int,pii> > q;
   q.push(mp(0,mp(-1,1))); // minus cost, from, to
   while (!q.empty()) {
      int v = q.top().se.se, d = -q.top().fi, from = q.top().se.fi;
      q.pop(); if (done[v]) continue; done[v] = 1; cost += d;
      if (from != -1) mst.pb(mp(from,v));
      FOREACH(it,adj[v]) if (!done[it->fi]) q.push(mp(-it->se,mp(v,it->fi)));
   }
   return cost;
}
```

Topological sorting

```
// input: directed graph on vertices 1..n. output:
// if not acyclic: vi()
// otherwise: vector of vertices in the topological order
vi toposort(int n, vi *adj) {
   vi ins(n+1,0), res; FORI(i,n) FOREACH(it,adj[i]) ++ins[*it];
   deque<int> q; FORI(i,n) if (ins[i] == 0) q.pb(i);
   while (!q.empty()) {
      int v = q.front(); q.pop_front();
      FOREACH(it,adj[v]) if ((--ins[*it]) == 0) q.pb(*it);
      res.pb(v);
   }
   return SZ(res)==n ? res : vi();
}
```

Decomposition into connected components

```
// graph with vertices 1..n (if 0..n-1 -> change FORI to FOR)
vector<vi> components (int n, vi *adj) {
   vector<bool> vis(n+1,0); vector<vi> res;
   FORI(i,n) {
      if (vis[i]) continue; vi comp; deque<int> q; q.pb(i);
      while (!q.empty()) {
         int v = q.front(); q.pop_front(); if (vis[v]) continue;
         vis[v] = 1; comp.pb(v); FOREACH(w,adj[v]) q.pb(*w);
      }
      res.pb(comp);
   }
   return res;
}
```

Longest path in DAG

```
#define N 200007
int n;     // input
vi adj[N]; // input
int dp[N]; // temp; -1 = unseen, -2 = processing, nonnegative = computed value
void dfs (int v) {
       if (dp[v] == -2) throw 0; if (dp[v] != -1) return;
       dp[v] = -2; int best = 0;
       FOREACH(x,adj[v]) { dfs(*x);  best = max(best, 1 + dp[*x]); }
       dp[v] = best;
}
int longestPathInDAG () {
       try {
              FORI(i,n) dp[i] = -1; FORI(i,n) dfs(i); int best = 0;
              FORI(i,n) best = max(best, dp[i]); return best; // number of edges
       } catch (int x) { return -1; /* cycle */ }
}
```

LCA Lowest Common Ancestor (n log n)

```
// before calling set: N, LOGN, n, parent (vertices numbered 1..n)
// parent - array: parent[i] is the parent of i (parent[root] is irrelevant)
// works for a tree. if you have a forest, make it into a tree by adding a superroot
#define N 100007
#define LOGN 17
int n, parent[N], // input
pos[N], anc[LOGN+1][N]; // temp

int getPos (int u) {
   if (pos[u] == -1) pos[u] = getPos(parent[u])+1;
   return pos[u];
}
void init (int root) {
   FORI(i,n) anc[0][i] = parent[i];
   FORI(i,n) pos[i] = -1;
   pos[root] = 0;
   FORI(i,n) if (pos[i] == -1) getPos(i);
   FORI(i,LOGN) FORI(j,n) anc[i][j] = anc[i-1][anc[i-1][j]];
}
int lca (int u, int v) {
   if (pos[u] < pos[v]) swap(u,v);
   int diff = pos[u] - pos[v];
   REPD(k,LOGN,0) if ((1<<k) <= diff) {
      diff -= 1<<k;
      u = anc[k][u];
   }
   // now they are equidistant from root
   if (u==v) return u;
   REPD(k,LOGN,0) if (anc[k][u] != anc[k][v]) {
      u = anc[k][u];
      v = anc[k][v];
   }
   return parent[u];
}
```

Strongly connected components (Tarjan)

```
// PRE: set n – number of vertices (1..n), adj, N
// POST: SCCs have numbers 1..n2, t[i] – number of component of vertex i

#define N 100007
int n, ind[N], low[N], onstack[N], last, n2, t[N];
vi adj[N], st;

void go (int v) {
```

```
   ind[v] = low[v] = ++last;
   st.pb(v);
   onstack[v] = 1;
   FOREACH(w,adj[v]) {
      if (!ind[*w]) {
         go(*w);
         low[v] = min(low[v], low[*w]);
      } else if (onstack[*w]) {
         low[v] = min(low[v], low[*w]);
      }
   }
   if (low[v] == ind[v]) { // v is a root node
      ++n2;
      for (int w = -1; w != v; ) {
         w = st.back();
         st.pop_back();
         onstack[w] = 0;
         t[w] = n2;
      }
   }
}

void scc() {
   last = n2 = 0;
   FORI(i,n) ind[i] = onstack[i] = 0;
   FORI(i,n) if (!ind[i]) go(i);
   FORI(i,n) t[i] = n2 + 1 - t[i]; // ordinary topological ordering, not reverse
}
// GRAPH COMPRESSION
// POST: n2 and adj2 encode a new graph on vertex set 1..n2 arising from compressing
//       the SCCs; the new graph is acyclic and has vertices in the topological
//       order (edges go e.g. 1->2)
vi adj2[N];
void compress() {   FORI(i,n2) adj2[i].clear();
   FORI(i,n) FOREACH(v,adj[i]) if (t[i] != t[*v]) adj2[t[i]].pb(t[*v]);     }
// ERASING MULTIPLE EDGES (also sorts all lists adj2[i])
vi temp[N];
void uniquify() {
   FORI(i,n2) temp[i].clear();   FORI(i,n2) FOREACH(v,adj2[i]) temp[*v].pb(i);
   FORI(i,n2) adj2[i].clear();   FORI(i,n2) FOREACH(v,temp[i]) if (adj2[*v].empty()
|| adj2[*v].back() != i) adj2[*v].pb(i);         }
```

2SAT

```
// PRE: * type in SCCs (scc() is enough)
//      * N (before scc()) = TWO TIMES the max number of variables
//      * k: number of variables (1..k)
//      * clauses: vector of pairs (i,j) denoting a clause (xi v xj)
//      *          negation is a minus, e.g. (-3,5) denotes (~x3 v x5)
// RESULT: is it satisfiable? if yes, then val[i] = valuation of xi
int k, val[N]; vector<pii> clauses; vi adjtr[N];
void twosat_dfs(int v, bool wart, vi *adja) {
   if (val[v] != -1) return;
   val[v] = wart;
   FOREACH(it,adja[v]) twosat_dfs(*it,wart,adja);
}
bool twosat() {
   n = 2*k;
   FORI(i,n) adj[i].clear(), adjtr[i].clear();
   FOR(i,SZ(clauses)) {
      int a = clauses[i].fi, b = clauses[i].se;
      if (a < 0) a = n+1+a; // negation
      if (b < 0) b = n+1+b;
      adj[n+1-a].pb(b);
      adj[n+1-b].pb(a);
      adjtr[b].pb(n+1-a);
      adjtr[a].pb(n+1-b);
   }
   scc(); // scc plus topological sorting
   FORI(i,k) if (t[i] == t[n+1-i]) return 0; // check if formula is satisfiable
   FORI(i,n) val[i] = -1;
   FORI(i,k) if (val[i] == -1) {
      int v = (t[i] > t[n+1-i]) ? i : n+1-i;
      // no path from v to ~v
      twosat_dfs(v,1,adj);
      twosat_dfs(n+1-v,0,adjtr);
   }
   return 1;
}
```

Bridges, Articulation points, biconnected Components, Low function

```
// lines denoted [C], [B], [A], [L] necessary only if we're looking for:
// numbering of biconnected [C]omponents, [B]ridges, [A]rticulation points, [L]ow
function
// every Bridge forms its own biconnected Component

#include <list> // can't change list<Edge> to vector<Edge>, don't try
#define N 1000007

struct Edge { // PRE: has to be bcc = -1 and bridge = 0 (constructors sets this)
    Edge* rev;
    int dest;
    int bcc;        // OUT: number of biconnected Component
    bool bridge;    // OUT: is it a bridge                      // B
    Edge(int v) : dest(v), bcc(-1)
                , bridge(0)                                     // B
    {};
};

int n;                  // IN: number of vertices
list<Edge> adj[N];      // IN: adjacency lists
int visit[N];
bool artp[N];  // OUT: is it an articulation point              // A
int bcc_num;   // OUT: number of biconnected Components         // C
int low[N];    // OUT: Low function                             // L

void add_edge (int a, int b) {
   adj[a].pb(Edge(b));   adj[b].pb(Edge(a));
   adj[a].back().rev = &adj[b].back();
   adj[b].back().rev = &adj[a].back();
}
stack<Edge*> st;                                                // C
int dfsTime;
int bccDFS (int v, bool root = 0) {
    int lo = visit[v] = ++dfsTime;
    FOREACH(it,adj[v]) {
        if (it->bcc != -1) continue;
```

```
        st.push(&*it);                                          // C
        it->rev->bcc = -2;
        if (!visit[it->dest]) {
            int ulo = bccDFS(it->dest);
            lo = min(ulo, lo);
            it->bridge = it->rev->bridge = (ulo > visit[v]);    // B
            if (ulo >= visit[v]) {                              // AC
                artp[v] = !root; root = 0;                      // A
                Edge* edge;                                     // C
                do {                                            // C
                    edge = st.top(); st.pop();                  // C
                    edge->bcc = edge->rev->bcc = bcc_num;       // C
                } while (edge != &*it);                         // C
                ++bcc_num;                                      // C
            }                                                   // AC
        } else lo = min(lo, visit[it->dest]);
    }
    low[v] = lo;                                                // L
    return lo;
}
void computeBCC(){
    fill(artp, artp+n+1, 0);                                    // A
    fill(visit, visit+n+1, 0);
    dfsTime = 1;
    bcc_num = 0;                                                // C
    FORI(i,n) if(!visit[i]) bccDFS(i, 1); // maybe change FORI to FOR
}
```

Euler path and cycle (undirected graph)

```
// usage: clear(n), add_edge(a,b), euler_undirected(n) (G on vertices ⊆ 0..n)
// OUT: no path -> vi(), path -> a vector of vertices, cycle -> v[0]==v.back()
#define N 300005
vector<pii> e;
vi adj[N];
void clear (int n) { e.clear(); REP(i,0,n) adj[i].clear(); }
void add_edge (int a, int b) { adj[a].pb(SZ(e)); adj[b].pb(SZ(e)); e.pb(pii(a,b)); }
vi euler_undirected (int n) {
   int start = -1, odddeg = 0;
   REP(i,0,n) {
      if (SZ(adj[i]) & 1) odddeg++, start = i;
      else if (!adj[i].empty() && start == -1) start = i;
   }
   if (odddeg > 2) return vi();
   vector<bool> used(SZ(e),0); // usage of edges
   vi stack, res, which(n+1,0); // which edge will be processed now
   stack.pb(start);
   while (!stack.empty()) {
      int v = stack.back();
      while (which[v] < SZ(adj[v]) && used[adj[v][which[v]]]) ++which[v]; // rewind
the used edges
      if (which[v] == SZ(adj[v])) { // back out
         stack.pop_back();
         res.pb(v);
      } else {
         int ed = adj[v][which[v]];
         int w = e[ed].fi == v ? e[ed].se : e[ed].fi;
         used[ed] = 1;
         stack.pb(w);
      }
   }
   if (count(ALL(used),0)) return vi(); // graph is not connected
   return res;
}
```

Bipartite matching O(mn)

```
#define MAXV 100005
int n,m; vi adj[MAXV]; int pair[MAXV]; bool vis[MAXV];

bool augment (int v) {          // 1 <= v <= n (lub v = -1)
   if (v == -1) return 1;
   if (vis[v]++) return 0;
   FOREACH(x,adj[v]) {
      if (augment(pair[*x])) { pair[*x] = v; pair[v] = *x; return 1; }
   }
   return 0;
}

// vertices on the left should have numbers 1..n, on the right - n+1..n+m
// PRE: set n,m,adj (adj needs to be filled out only for vertices 1..n), and
// the constant MAXV = maximum n+m
int matching () {
   FORI(i,n+m) pair[i] = -1;
   int w = 0;
   FORI(i,n) {
      FORI(j,n) vis[j] = 0;
      w += augment(i);
   }
   return w;
}
```

Bipartite matching Hopcroft-Karp O(m√n)

```
#define MAXV 100005
int n,m; vi adj[MAXV]; int pair[MAXV], dist[MAXV];

bool bfs() {
   deque<int> q;
   FORI(v,n) { if (!pair[v]) { dist[v] = 0; q.pb(v); } else dist[v] = INF; }
   dist[0] = INF;
   while (!q.empty()) {
      int v = q.front(); q.pop_front();
      FOREACH(u,adj[v]) if (dist[pair[*u]] == INF) {
         dist[pair[*u]] = dist[v]+1;
         q.pb(pair[*u]);
      }
   }
   return dist[0] != INF;
}
bool dfs (int v) {
   if (!v) return 1;
   FOREACH(u,adj[v]) if (dist[pair[*u]] == dist[v]+1) {
      if (dfs(pair[*u])) { pair[*u] = v; pair[v] = *u; return 1; }
   }
   dist[v] = INF;
   return 0;
}
```

```
// vertices on the left should have numbers 1..n, on the right - n+1..n+m
// PRE: set n,m,adj (adj needs to be filled out only for vertices 1..n) (adj[0] must
be empty), and the constant MAXV = maximum n+m
int matching () {
   FOR(i,n+m+1) pair[i] = 0;
   int w = 0;
   while (bfs()) FORI(v,n) if (!pair[v]) if (dfs(v)) ++w;
   return w;
}
```

Weighted bipartite matching: Hungarian in $O(n^3)$

```
// finds the MAXIMUM WEIGHT among all PERFECT matchings in an n x n bipartite graph
// before usage set n and w[0..n-1][0..n-1]
typedef int T; // type of weights
#define N 128
T w[N][N]; int n;   // input
T lx[N], ly[N], sl[N];
int skojx[N], skojy[N], par[N], q[N];
bool markx[N], marky[N];

vi hungarian () {
   FOR(i,n) {
      skojx[i] = skojy[i] = -1;
      ly[i] = 0;
      lx[i] = *max_element(w[i],w[i]+n);
   }
   FOR(syf,n) {
      int v = -1, qb = 0, qe = 0;
      FOR(i,n) {
         marky[i] = markx[i] = 0;
         sl[i] = -1;
         if (skojx[i] == -1) q[qe++] = i;
      }
      while (v == -1) {
         while (qb < qe) {
            int i = q[qb++];
            markx[i] = 1;
            FOR(j,n) if (!marky[j] && (sl[j] == -1 || sl[j] > lx[i]+ly[j]-w[i][j])){
               if ((sl[j] = lx[par[j] = i] + ly[j] - w[i][j]) == 0) {
                  marky[j] = 1;
                  if (skojy[j] != -1) q[qe++] = skojy[j];
                  else { v = j; goto end; }
               }
            }
         }
         T x = -1;
         FOR(i,n) if (!marky[i] && (x == -1 || sl[i] < x)) x = sl[i];
         FOR(i,n) {
            if (markx[i]) lx[i] -= x;
            if (marky[i]) ly[i] += x;
            else if ((sl[i] -= x) == 0) {
               marky[i] = 1;
               if (skojy[i] != -1) q[qe++] = skojy[i];
               else v = i;
            }
         }
      }
      end:
      while (v != -1) {
         int y = skojx[par[v]];
         skojx[par[v]] = v;
         skojy[v] = par[v];
         v = y;
      }
   }
   return vi(skojx,skojx+n);
}
```

Matching in general graphs: Edmonds in $O(n^3)$

```
#define N 307
int n;                  // IN: number of vertices
bool edge[N][N];        // IN: adjacency matrix (can be changed to vectors if needed)
int mate[N];            // OUT: matched vertex (-1 = none)
int label[N], base[N], prev1[N], prev2[N];
bool mark[N];

bool prepare (int v) {
    while(1) {
        mark[v] = !mark[v];
        if (mate[v] == -1) return mark[v];
        v = base[prev2[mate[v]]];
    }
}
int shrink (int v, int b1, int b2, queue<int> &Q) {
    while (mark[v]) {
        prev1[v] = b1; prev2[v] = b2;
        mark[mate[v]] = 1;
        Q.push(mate[v]);
        v = base[prev2[mate[v]]];
    }
    return v;
}
bool make_blos (int i, int j, int bi, int bj, queue<int> &Q) {
    if (label[i]!=1 || i==j) return 0;
    if (prepare(i), prepare(j)) return 1;
    int b = (shrink(i, bi, bj, Q), shrink(j, bj, bi, Q));
    FOR(v,n) if (mark[base[v]]) base[v] = b;
    return 0;
}
void rematch(int i, int j) {
    int nxt = mate[i];
    mate[i] = j;
    if (nxt==-1) return;
    mate[nxt] = -1;
    rematch(prev2[nxt], prev1[nxt]);
    rematch(prev1[nxt], prev2[nxt]);
}
bool augment() {
    queue<int> Q;
    FOR(i,n) {
        label[i] = mate[i]==-1;
        if (mate[i]==-1) Q.push(i);
        mark[i] = 0;
        base[i] = i;
    }
```

```
    while (!Q.empty()) {
        int cur = Q.front(); Q.pop();
        FOR(i,n) if (edge[cur][i] && i!=mate[cur]) {
            if (!label[i]) {
                label[i] = -1;
                label[mate[i]] = 1;
                Q.push(mate[i]);
                prev1[i] = i; prev2[i] = cur;
            } else if (make_blos(base[i], base[cur], i, cur, Q)) {
                rematch(i, cur); rematch(cur, i);
                return 1;
            }
        }
    }
    return 0;
}
int compute_gcm() { // returns the cardinality of maximum matching
    fill_n(mate, n, -1);
    int res = 0;
    while (augment()) ++res;
    return res;
}
```

Max Flow (Ford-Fulkerson) short code

```
const int N = 1000; // INPUT
int n, cap[N][N];   // INPUT (vertices 0..n-1)
int flow[N][N];     // OUTPUT
int edmonds (int s, int t) {
   int b, e, q[n], p[n], res = 0;
   vi adj[n];
   FOR(i,n) FOR(j,n) { flow[i][j] = 0; if (cap[i][j]) adj[i].pb(j), adj[j].pb(i); }
   while (1) {
      FOR(i,n) p[i] = -1;
      for (q[b=e=0] = s; b <= e; ++b) FOREACH(v,adj[q[b]])
            if (flow[q[b]][*v] < cap[q[b]][*v] && p[*v] < 0) p[q[++e] = *v] = q[b];
      if (p[t] < 0) return res;
      int d = cap[p[t]][t] - flow[p[t]][t];
      for (int i = t; i != s; i = p[i]) d = min(d, cap[p[i]][i] - flow[p[i]][i]);
      for (int i = t; i != s; i = p[i]) flow[p[i]][i] += d, flow[i][p[i]] -= d;
      res += d;
   }
}
```

Max Flow Dinic (optimized)

```
// usage: init(n, s, t), addEdge(u, v, capa u->v, capa v->u), dinicFlow()

const int N = 100000 + 7, M = 100000 + 7;
typedef int T; // if changing this to long long, also change INF to 10^18

int nodes, src, dest, nedge;
int point[M], nxt[M], flow[M], capa[M];
int head[N], dist[N], Q[N], work[N];

void init (int _nodes, int _src, int _dest) {
   nodes = _nodes + 2; // just to be safe
   src = _src;
   dest = _dest;
   FOR(i,nodes) head[i] = -1;
   nedge = 0;
}
void addEdge(int u, int v, int c1, int c2 = 0) {
   point[nedge]=v, capa[nedge]=c1, flow[nedge]=0, nxt[nedge]=head[u],
head[u]=(nedge++);
   point[nedge]=u, capa[nedge]=c2, flow[nedge]=0, nxt[nedge]=head[v],
head[v]=(nedge++);
}

bool dinicBfs() {
   FOR(i,nodes) dist[i] = -1;
   dist[src] = 0;
   int szQ = 1;
   Q[0] = src;
   FOR(cl,szQ) {
      for (int k = Q[cl], i = head[k]; i >= 0; i = nxt[i]) {
         if (flow[i] < capa[i] && dist[point[i]] < 0) {
            dist[point[i]] = dist[k] + 1;
            Q[szQ++] = point[i];
         }
      }
   }
   return dist[dest] >= 0;
}

T dinicDfs (int x, T exp) {
   if (x == dest) return exp;
   T res = 0;
   for (int& i = work[x]; i >= 0; i = nxt[i]) {
      int v = point[i]; T tmp;
      if (flow[i] < capa[i] && dist[x]+1 == dist[v]
          && (tmp = dinicDfs(v, min(exp, (T)capa[i] - flow[i]))) > 0) {
         flow[i] += tmp;
         flow[i^1] -= tmp;
         res += tmp;
         exp -= tmp;
         if (0 == exp) break;
      }
   }
   return res;
}

T dinicFlow () {
   T res = 0;
   while (dinicBfs()) {
      FOR(i,nodes) work[i] = head[i];
      res += dinicDfs(src,INF);
   }
   return res;
}
```

Min Cost Max Flow

```
namespace MCF {
   #define MAXN 100010
   #define MAXM 100010
   #define wint int
   #define cint int
```

```
   const wint wEPS = 0;
   const wint wINF = 1001001001;
   const cint cEPS = 0;
   const cint cINF = 1001001001;
   int n, m, ptr[MAXN], nxt[MAXM], zu[MAXM];
   wint capa[MAXM], tof;
   cint cost[MAXM], toc, d[MAXN], pot[MAXN];
   int vis[MAXN], pree[MAXN];
   void init(int _n) {
      n = _n+2; // to be safe
      m = 0;
      memset(ptr, ~0, n << 2);
   }
   void ae(int u, int v, wint w, cint c) {
      nxt[m] = ptr[u]; ptr[u] = m; zu[m] = v; capa[m] = w; cost[m] = +c; ++m;
      nxt[m] = ptr[v]; ptr[v] = m; zu[m] = u; capa[m] = 0; cost[m] = -c; ++m;
   }
   bool spRep(int src, int ink, wint flo = wINF) { // result is: (tof,toc)
      wint f;
      cint c, cc;
      int i, v;
      memset(pot, 0, n * sizeof(cint));
   //*
      for (bool cont = 1; cont; ) {
         cont = 0;
         FOR(u,n) for (i = ptr[u]; ~i; i = nxt[i]) if (capa[i] > wEPS) {
            if (pot[zu[i]] > pot[u] + cost[i] + cEPS) {
               pot[zu[i]] = pot[u] + cost[i]; cont = 1;
            }
         }
      }
   //*/
      for (toc = 0, tof = 0; tof + wEPS < flo; ) {
         typedef pair<cint,int> node;
         priority_queue< node,vector<node>,greater<node> > q;
         FOR(u,n) { d[u] = cINF; vis[u] = 0; }
         for (q.push(mp(d[src] = 0, src)); !q.empty(); ) {
            c = q.top().first; int u = q.top().second; q.pop();
            if (vis[u]++) continue;
            for (i = ptr[u]; ~i; i = nxt[i]) if (capa[i] > wEPS) {
               cc = c + cost[i] + pot[u] - pot[v = zu[i]];
               if (d[v] > cc) { q.push(mp(d[v] = cc, v)); pree[v] = i; }
            }
         }
         if (!vis[ink]) return 0;
         f = flo - tof;
         for (v = ink; v != src; v = zu[i ^ 1]) { i = pree[v]; REMIN(f, capa[i]); }
         for (v = ink; v != src; v = zu[i ^ 1]) { i = pree[v]; capa[i] -= f; capa[i
^ 1] += f; }
         tof += f;
         toc += f * (d[ink] - pot[src] + pot[ink]);
         FOR(u,n) pot[u] += d[u];
      }
      return 1;
   }
}
```

Numbers

Euclidean algorithm, modular inverse, Chinese remaindering

```
// for gcd, just use __gcd
ll lcm (ll a, ll b) { return a / __gcd(a,b) * b; }

// fast exponentiation modulo MOD
ll powe (ll x, ll p) {
        if (p == 0) return 1;
        ll w = powe((x*x)%MOD, p/2);
        if (p & 1) w = (w*x)%MOD;
        return w;
}

// modular inverse (MOD must be prime)
ll modinv (ll i) { return powe(i,MOD-2); }

// computing inverses of 1..n in linear time and short code (MOD must be prime)
ll inv[n+1];
inv[1] = 1;
REP(i,2,n) inv[i] = MOD - MOD / i * inv[MOD % i] % MOD;

// a*x - b*y = return value = GCD(a,b). x,y >= 0
ll egcd(ll a, ll b, ll &x, ll &y) { // watch out: MINUS B*Y
        if (b == 0) {
                x = 1;
                y = 0;
                return a;
        }
        ll d = egcd(b, a%b, y, x);
        y = a - x*(a/b) - y;
        x = b - x;
        return d;
}

// modular inverse (works for any modulus p)
ll inverse(ll a, ll p) {
   ll x,y;
   egcd(a%p + p, p, x, y);
   return x%p;
}

// solves the system of congruences:
// x = a (mod m)
// x = b (mod n)
// returns res such that x = res (mod lcm(m,n))
// no such res => throws 0 (so you must wrap it in try { ... } catch (int _) { ... }
ll CRT(ll a, ll m, ll b, ll n) {
        b = (b+n-(a%n))%n;
        ll d = __gcd(m,n);
        if (b%d != 0) throw 0;
        ll old_m = m;
        m/=d; b/=d; n/=d;
        return ((b*inverse(m,n))%n)*old_m + a;
}
```

Prime numbers: Eratosthenes sieve and O(√n) factorization

```
// very fast: for m = 10^7 works in 0.1s
vector<bool> sieve (int m) { // m = max_inclusive
   vector<bool> pr(m+1,1); pr[0] = pr[1] = 0;
   for (int x = 2; x*x <= m; ++x) if (pr[x])
      for (int y = x*x; y <= m; y += x) pr[y] = 0;
   return pr;
}
vi listOfPrimes (int m) { // m = max_inclusive
   vector<bool> pr = sieve(m); vi ans; FORI(i,m) if (pr[i]) ans.pb(i); return ans;
}

typedef vector<pair<ll,int> > Fact;
Fact factorize (ll x) { // factorization: {<prime, multiplicity>}
   Fact res;
   for (int d = 2; (ll)d*d <= x; ++d) {
      if (x % d == 0) {
         int k = 0; while (x % d == 0) { x /= d; ++k; } res.pb(mp(d,k));
      }
   }
   if (x != 1) res.pb(mp(x,1)); return res;
}
```

Miller-Rabin primality test and Pollard-rho factorization

```
#define ull unsigned ll
// for n < 2^32: inline ull mul(ull a, ull b, ull mod) { return (a*b) % mod; }
const int _k = 16; const ull _mask = (1<<_k)-1;
ull mul (ull a, ull b, ull mod) { // assumption: b, mod < 2^(64-_k)
    ull result = 0;
    while (a) {
        ull temp = (b * (a & _mask)) % mod;
        result = (result + temp) % mod;
        a >>= _k;
        b = (b << _k) % mod;
    }
    return result;
}

ull pow (ull a, ull w, ull mod) {
    ull res = 1;
    while (w){
        if (w&1) res = mul(res, a, mod);
        a = mul(a, a, mod);
        w /= 2;
    }
    return res;
}
bool primetest (ull n, int a) {
    if (a > n-1) return 1;
    ull d = n-1;
    int s = 0;
    while (!(d&1)) { d /= 2; s++; }
    ull x = pow(a, d, n);
    if (x == 1 || x == n-1) return 1;
    FOR(i,s-1){
        x = mul(x, x, n);
        if (x == 1) return 0;
        if (x == n-1) return 1;
    }
    return 0;
}

bool isPrime(ull n) {
    if (n < 4) return n > 1;
    bool pr = n%2;
    // some if-branches below may be often skipped (e.g. if we only have n < 2^32)
    if (n < (1LL << 32)) {
       for (int a : {2,7,61}) pr = pr && primetest(n,a);
    } else if (n < (1LL << 48)) {
       for (int a : {2,3,5,7,11,13,17}) pr = pr && primetest(n,a);
    } else {
       for (int a : {2,325,9375,28178,450775,9780504,1795265022}) pr = pr &&
primetest(n,a);
    }
    return pr;
}
// if needed, the test can be sped up by first checking divisibility by first
// several primes

// Pollard-rho factorization
// works in time O(n^1/4), leaves result in the map fact; before usage, fact.clear()!
map<ull,int> fact; // factorization: {<prime, multiplicity>}
ull find_factor(ull z) {
    if (!(z&1)) return 2;
    ull c = rand() % z, x = 2, y = 2, d = 1;
    while (d == 1) {
        ull tp = (mul(y,y,z) + c) % z;
        y = (mul(tp,tp,z) + c) % z;
        x = (mul(x,x,z) + c) % z;
        d = __gcd(x>y ? x-y : y-x, z);
    }
    return d;
}
void rhofact(ull z) {
    if (z == 1) return;
    if (isPrime(z)) { fact[z]++; return; }
    while (1) {
        ull f = find_factor(z);
        if (f != z) { rhofact(f); rhofact(z/f); break; }
    }
}
```

Newton symbol DP

```
#define NEED 66  // max n st [for every k, (n choose k) fits into long long] = 66
ll nt[NEED+1][NEED+1];
void initnewton () {
   FOR(n,NEED+1) nt[n][0] = 1; FORI(k,NEED) nt[0][k] = 0;
   FORI(n,NEED) FORI(k,NEED) nt[n][k] = nt[n-1][k-1] + nt[n-1][k];
}
```

Matrices

```
typedef vector<int> row;   // 10 times slower on long longs
int R, mod;                // set this before usage. R = size
struct matrix : vector<row> { matrix() : vector<row>(R,row(R,0)) {} };
```

```
matrix id() { matrix w; FOR(i,R) w[i][i] = 1; return w; }

matrix mult(const matrix &m1, const matrix &m2) {
   matrix w;
   FOR(i,R) FOR(j,R) {
      ll x = 0;
      FOR(k,R) x += (ll)m1[i][k] * m2[k][j] % mod;
      // if mod <= 10^9+9 and R < 9, you can remove "% mod" in the line above
      w[i][j] = x % mod;
   }
   return w;
}

matrix power(const matrix &m, ll k) { // did you read k from input as a longlong?
   if (!k) return id();
   matrix x = power(m,k/2);
   x = mult(x,x);
   if (k&1) return mult(m,x);
   return x;
}

// example
int fib (ll k){ matrix m; m[0][0] = m[0][1] = m[1][0] = 1; return power(m,k)[1][0];}
```

Bignums

```
const int DIGITS = 6, BASE = 1000000; // remark: number_of_blocks*BASE^2 has to fit
                                      // into longlong
#define SIZ int(size())
struct bignum : public vi {
   bignum() { pb(0); }
   bignum(ll num) {
      if(num < BASE) pb(num);
      else while(num > 0) { pb(num % BASE); num /= BASE; }
   }
   bignum(const string &s) {
      for (int i = 0; i < SZ(s); i += DIGITS) {
         int kon = SZ(s)-i, pocz = max(0,kon-DIGITS);
         pb(atoi(s.substr(pocz,kon-pocz).c_str()));
      }
      cut0();
   }
   string str() const {
     stringstream ss;
     ss << back() << setfill('0');
     for(int i = SIZ-2; i>=0; --i) ss << setw(DIGITS) << at(i);
     return ss.str();
   }
   bignum& operator+= (const bignum& y) {
      int maxsz = max(SIZ,SZ(y)); bool carry = 0;
      FOR(i,maxsz) {
         int sum = carry + (i < SIZ ? at(i) : 0) + (i < SZ(y) ? y[i] : 0);
         if (sum >= BASE) { carry = 1; sum -= BASE; } else { carry = 0; }
         if (i < SIZ) at(i) = sum; else pb(sum);
      }
      if (carry) pb(1); return *this;
   }
   void cut0() {
      while (size()>1 && back()==0) pop_back();
   }
   bignum operator* (const bignum& y) const {
      vector<ll> w(size()+SZ(y)+2, 0);
      FOR(i,SIZ) FOR(j,SZ(y)) w[i+j] += (ll)at(i) * y[j];
      ll carry = 0;
      FOR(i,SZ(w)) { w[i] += carry; carry = w[i] / BASE; w[i] %= BASE; }
      bignum res; res.resize(SZ(w)); FOR(i,SZ(w)) res[i]=w[i]; res.cut0();
      return res;
   }
   bignum& operator -= (const bignum &y) { // only works for x>=y!
      FOR(i,SIZ) {
         if (i < SZ(y)) at(i) -= y[i];
         if (at(i) < 0) { at(i) += BASE; at(i+1) -= 1; }
      }
      cut0(); return *this;
   }
   int compare (const bignum &y) const {
      if (SIZ < SZ(y)) return -1; if (SIZ > SZ(y)) return 1;
      for (int i = size()-1; i >= 0; --i) if (at(i) != y[i]) {
         if (at(i) < y[i]) return -1; else return 1;
      }
      return 0;
   }
   int operator % (int m) const {
      ll w = back()%m;
      for (int i = size()-2; i >= 0; --i) w = (w*BASE + at(i))%m;
      return w;
   }
   bignum& operator /= (int y) { // only works for y < BASE!
      ll dz = 0;
      for (int i = size()-1; i >= 0; --i) {
         ll x = dz*BASE + at(i); at(i) = x/y; dz = x%y;
      }
      cut0(); return *this;
   }
   bool operator < (const bignum &y) const { return compare(y)<0; }
   bool operator > (const bignum &y) const { return compare(y)>0; }
   bool operator <= (const bignum &y) const { return compare(y)<=0; }
   bool operator >= (const bignum &y) const { return compare(y)>=0; }
   bool operator == (const bignum &y) const { return compare(y)==0; }
   bool operator != (const bignum &y) const { return compare(y)!=0; }
   bignum operator + (const bignum& y) const { bignum res = *this; res += y; return
res; }
   bignum operator - (const bignum &y) const { bignum res = *this; res -= y; return
res; } // only works for x>=y!
   bignum& operator *= (const bignum &y) { return *this = *this * y; }
   bignum operator / (int y) const { bignum res = *this; res /= y; return res; } //
only works for y < BASE!
};
ostream& operator << (ostream &s, const bignum &y) { return s << y.str(); }
```

Gauss in R (on doubles)

```
const long double EPS = 1e-9;
inline bool IsZero(double x){ return x>=-EPS && x<=EPS; }
typedef vector<double> VD;
// Solves ax=b
int GaussDouble(vector<VD> a, VD b, VD &x) {
```

```
    int m = a.size(), n = a[0].size(), k, r;
    vi q; // which column in taken care of on the k-th step
    for (k = 0; k < min(m, n); k++)  {
        int i, j;
        for (j = k; j < n; j++) for (i = k; i < m; i++)
            if (!IsZero(a[i][j])) goto found;
        break;
        found:
        if (j != k) FOR(t, m) swap(a[t][j], a[t][k]);
        q.pb(j);
        if (i != k) { swap(a[i], a[k]); swap(b[i], b[k]); }
        REP(j,k+1,m-1) if (!IsZero(a[j][k])) {
            double l = (a[j][k] / a[k][k]);
            REP(i,k,n-1) a[j][i] = a[j][i] - (l * a[k][i]);
            b[j] = b[j] - (l * b[k]);
        }
    }
    r = k; // r == rank of matrix a
    x.clear(); x.resize(n, 0);
    REP(k,r,m-1) if (!IsZero(b[k])) return -1; // no solution
    REPD(k,r-1,0) {
        double s = b[k];
        REP(j,k+1,r-1) s = s - (a[k][j] * x[j]);
        x[k] = s / a[k][k];
    }
    REPD(k,r-1,0) swap(x[k], x[q[k]]); // determinant = product of a[i][i] over i
    return n - r; // the dimension of the space of solutions
}
```

Gauss in Z_2

```
template <size_t s> int GaussZ2 (vector<bitset<s> > equ, vi vals, vi &res) {
   int n = SZ(equ), a = 0, b = 0, c;
   res.resize(s, 0);
   for ( ; a < n && b < s; ++b) {
      for (c = a; c < n && !equ[c][b]; ++c);
      if (c < n) {
         if (a != c) { equ[a] ^= equ[c]; vals[a] ^= vals[c]; }
         FOR(y,n) if (a != y && equ[y][b]) { equ[y] ^= equ[a]; vals[y] ^= vals[a]; }
         res[b] = ++a;
      }
   }
   FOR(x,b) if (res[x]) res[x] = vals[res[x] - 1];
   FOR(x,n) {
      c = 0;
      FOR(z,s) if (equ[x][z]) c ^= res[z];
      if (c != vals[x]) return -1;
   }
   return s-a;
}
```

Gauss in Z_p

```
// remark: for p=2 there's a simpler&faster one above (but this will work too)
ll mpow(ll a, ll b, ll P) {
    a %= P; ll w = 1;
    while (b) {
        if (b % 2) w = (w * a) % P;
        a = (a * a) % P;
        b /= 2;
    }
    return w;
}
inline ll rev (int x, int P) { return mpow(x,P-2,P); }
// for large P, add casts to ll where needed!
int GaussZp(vector<vi> A,vi b,vi &x,int P) {
    int m=SZ(A),n=SZ(A[0]),k,r;
    vi q;
    for(k=0;k<min(m,n);k++) {
        int i,j;
        for(j=k;j<n;j++) for(i=k;i<m;i++) if (A[i][j]) goto found;
        break;
        found:
        if (j!=k) FOR(t,m) swap(A[t][j],A[t][k]);
        q.pb(j);
        if (i!=k){swap(A[i],A[k]);swap(b[i],b[k]);}
        REP(j,k+1,m-1) if (A[j][k]) {
            int l=(A[j][k]*rev(A[k][k],P))%P;
            REP(i,k,n-1) A[j][i]=(P+A[j][i]-(l*A[k][i])%P)%P;
            b[j]=(P+b[j]-(l*b[k])%P)%P;
        }
    }
    r=k;x.clear();x.resize(n,0);
    REP(k,r,m-1) if (b[k]) return -1;
    REPD(k,r-1,0) {
        int s=b[k];
        REP(j,k+1,r-1) s=(P+s-(A[k][j]*x[j])%P)%P;
        x[k]=(s*rev(A[k][k],P))%P;
    }
    REPD(k,r-1,0) swap(x[k],x[q[k]]);
    return n-r;
}
```

FFT

```
// Computes in-place the following transform:
// y[i] = A(w^(dir*i)), where w = exp(2pii/N)
// A(x) = a[0] + a[1] x + ... + a[N-1] x^{N-1},
// dir is FFT's direction (+1=forward, -1=inverse).
// Notes: * N must be a power of 2,
//        * scaling by 1/N after inverse FFT is caller's resposibility.

typedef long double T; // long doubles work a little slower (~20%), better use them
T PI = 2 * acos(0.0L);
T rnd (T x) { return x < 0.0 ? ceil(x - 0.5) : floor(x + 0.5); }


// OPTION 1 - less code, but almost twice slower
typedef complex<T> Complex;

// OPTION 2 - own complex class: more code, 2x faster
struct Complex {
   T re, im;
   Complex () {}
   Complex (T r) : re(r), im(0) {}
   Complex (T r, T i) : re(r), im(i) {}
   Complex operator * (const Complex &c) const {
      return Complex(re * c.re - im * c.im, im * c.re + re * c.im);
```

```
   }
   Complex operator + (const Complex &c) const {
      return Complex(re + c.re, im + c.im);
   }
   Complex operator - (const Complex &c) const {
      return Complex(re - c.re, im - c.im);
   }
};
T real (Complex x) { return x.re; }

void FFT(Complex *a, int N, int dir) {
   int lgN;
   for (lgN = 1; (1 << lgN) < N; lgN++);
   //assert((1 << lgN) == N);

   FOR(i,N) {
      int j = 0;
      FOR(k,lgN) {
         j |= ((i>>k)&1) << (lgN-1-k);
      }
      if (i < j) swap(a[i], a[j]);
   }
   FORI(s,lgN) {
      int h = 1 << (s - 1);
      Complex t, w, w_m(cos(dir*PI/h), sin(dir*PI/h));
      for (int k = 0; k < N; k += h+h) {
         w = 1;
         FOR(j,h) {
            t = w * a[k+j+h];
            a[k+j+h] = a[k+j] - t;
            a[k+j] = a[k+j] + t;
            w = w * w_m;
         }
      }
   }
}

// multiplication of polynomials in O(n log n)
vector<ll> convolve (vector<ll> &a1, vector<ll> &a2) {
   int N = 8;
   while (N < SZ(a1)+SZ(a2)+2) N *= 2;
   Complex *t1 = new Complex[N], *t2 = new Complex[N];
   FOR(i,N) t1[i] = t2[i] = 0;
   FOR(i,SZ(a1)) t1[i] = a1[i];
   FOR(i,SZ(a2)) t2[i] = a2[i];
   FFT(t1,N,1);
   FFT(t2,N,1);
   FOR(i,N) t1[i] = t1[i] * t2[i];
   delete [] t2;
   FFT(t1,N,-1);
   vector<ll> res(SZ(a1)+SZ(a2)-1);
   FOR(i,SZ(res)) res[i] = (ll)rnd(real(t1[i])/N);
   delete [] t1;
   return res;
}

// fast multiplication of bignums: almost the same as convolve
// recommended parameters for good stability and speed: DIGITS = 5 and long doubles
// (then, on numbers of length ~10^4, works 3x faster than normal multiplication,
//  if you use OPTION 2); you can squeeze out more but it's risky
```

Simplex (linear programming)

```
// max cx with Ax <= b, x >= 0, returns x (polytope unbounded or empty => x empty)
const double EPS = 10e-9; typedef long double T; typedef vector<T> VT;
namespace Simplex {
  vector<VT> A;  VT b,c,res;  vi kt,N;  int m;
  inline void pivot(int k,int l,int e){
      int x=kt[l]; T p=A[l][e];
      FOR(i,k) A[l][i]/=p; b[l]/=p; N[e]=0;
      FOR(i,m) if (i!=l) b[i]-=A[i][e]*b[l],A[i][x]=A[i][e]*-A[l][x];
      FOR(j,k) if (N[j]){
        c[j]-=c[e]*A[l][j];
        FOR(i,m) if (i!=l) A[i][j]-=A[i][e]*A[l][j];
      }
      kt[l]=e; N[x]=1; c[x]=c[e]*-A[l][x];
  }
  VT doit(int k){
    VT res; T best;
    while (1){
      int e=-1,l=-1; FOR(i,k) if (N[i] && c[i]>EPS) {e=i; break;}
      if (e==-1) break;
      FOR(i,m) if (A[i][e]>EPS && (l==-1 || best>b[i]/A[i][e]))
        best=b[ l=i ]/A[i][e];
      if (l==-1)
               return VT();
      pivot(k,l,e);
    }
    res.resize(k,0); FOR(i,m) res[kt[i]]=b[i];
    return res;
  }
  VT simplex (const vector<VT> &AA, const VT &bb, const VT &cc){
    int n=AA[0].size(),k;
    m=AA.size(); k=n+m+1; kt.resize(m); b=bb; c=cc; c.resize(n+m);
    A=AA; FOR(i,m){ A[i].resize(k); A[i][n+i]=1; A[i][k-1]=-1; kt[i]=n+i;}
    N=vi(k,1); FOR(i,m) N[kt[i]]=0;
    int pos=min_element(ALL(b))-b.begin();
    if (b[pos]<-EPS){
      c=VT(k,0); c[k-1]=-1; pivot(k,pos,k-1); res=doit(k);
      if (res[k-1]>EPS) return VT();
      FOR(i,m) if (kt[i]==k-1)
          FOR(j,k-1) if (N[j] && (A[i][j]<-EPS || EPS<A[i][j])){
            pivot(k,i,j); break;
          }
      c=cc; c.resize(k,0); FOR(i,m) FOR(j,k) if (N[j]) c[j]-=c[kt[i]]*A[i][j];
    }
    res=doit(k-1); if (!res.empty()) res.resize(n);
    return res;
  }
};
```

Strings

Hashing strings

```
const int P1 = 999999929, P2 = 998999999, P3 = 989999993; // primes
// first method:
const Int Q(500029), Q1(747412597, 452664601, 463322824); // Q^-1 mod P1, P2, P3
h[0] = 0; FORI(i,n) h[i] = h[i-1] + Qpow[i-1] * t[i-1]; // initialization
```

```
Int hash (int from, int to) // hash of subword t[a..b] (a,b 0-based)
{ return (h[to+1] - h[from]) * Q1pow[from]; }

computing modular inverse in Python (locally): e.g. pow(29,p-2,p)

// second method:
h[0] = 0; FORI(i,n) h[i] = h[i-1] * Q + t[i-1];
ll hash (int a, int b) // hash of subword t[a..b] (a,b 0-based)
{ return h[b+1] - Qpow[b-a+1]*h[a]; }
```

Manacher (radii of palindromes)

```
// example: for "aaaa" even returns: 0 1 2 1, odd: 0 1 1 0
vi manacher(string s, bool even) {
   int n = SZ(s);
   vi res(n);
   int l = 0, r = -1;
   FOR(i,n) {
        int k;
        if (i>r) k = 1;
        else k = min(res[l+r-i+even], r-i+even) + 1;
        while (i+k-even < n && i-k >= 0 && s[i+k-even] == s[i-k]) ++k;
        res[i] = --k;
        if (i+k-even > r){
          l = i-k;
          r = i+k-even;
       }
   }
   return res;
}
```

KMP

```
// returns 0-based indices of matches
vi kmp(string t, string p) { // text, pattern
   vi kn(SZ(p)+1,-1); // Knuth prefix function
   FORI(i,SZ(p)) {
      int j = kn[i-1];
      while (j != -1 && p[j] != p[i-1]) j=kn[j];
      kn[i] = j+1;
   }
   // kn is computed

   vi ans;
   int ppos=0,tpos=0;
   while (tpos<SZ(t)) {
      while (ppos!=-1 && (ppos == SZ(p) || p[ppos]!=t[tpos])) ppos=kn[ppos];
      ppos++; tpos++;
      if (ppos==SZ(p)) ans.pb(tpos-SZ(p));
   }
   return ans;
}
// caching the transition function (e.g. to make KMP on tree):
REP(c,'a','z') trans[0][c-'a'] = (c == p[0]);
REP(ppos,1,szp) REP(c,'a','z') {
   if (ppos == szp || p[ppos] != c) trans[ppos][c-'a'] = trans[kn[ppos]][c-'a'];
   else trans[ppos][c-'a'] = ppos + 1;
}
```

Z-function (PREF function)

```
// Return values: z[i] = maximum k such that s[i..i+k-1] = s[0..k-1].
// Application: pattern-matching – to find S in T, compute z-fun for S$T
vi zfun (const string &s) {
       int N = SZ(s), a=0, b=0;
       vi z(N,N);
       REP(i,1,N-1) {
              int k = i < b ? min(b-i, z[i-a]) : 0;
              while (i+k < N && s[i+k] == s[k]) ++k;
              z[i] = k;
              if (i+k > b) a=i, b=i+k;
       }
       return z;
}
```

Lexicographically minimal cyclic shift (Duval)

```
// uses Duval algorithm; O(n) time, O(n) memory (can be made O(1))
string lexicographically_minimal_cyclic_shift (string s) {
       s += s;
       int len = SZ(s), i=0, res=0;
       while (i < len/2) {
              res = i;
              int j=i+1, k=i;
              while (j < len && s[k] <= s[j]) {
                     if (s[k] < s[j]) k = i; else ++k;
                     ++j;
              }
              while (i <= k) i += j-k;
       }
       return s.substr(res, len/2);
}
```

Aho-Corasick

```
// usage: first add all words (addWord(s)), then calcLink(), then make searches
// using searchAll or searchFirst
struct mkmp {
   struct node {
      map<char,node*> son, cache;
      node *lnk, *wo; // pi function; link to max prefsufa which is a pattern
      int el; // number of pattern (-1 if not a pattern)
      node() : el(-1) {}
   };

   vi len;
   node root;

   node* mv (node *w, char l) {
      node*& r = w->cache[l];
      return r ? r : r = (w->son.count(l) ? w->son[l] : w == &root ? w : mv(w->lnk,
l)); // watch out if there are stack limit – deep recursion!
   }

   // adds a pattern and assigns a number to it; returns this number
   int addWord (const char *s) {
      int l = strlen(s);
      node *p = &root;
```

```
      for (; *s; ++s) {
         auto e = p->son.find(*s);
         p = (e == p->son.end()) ? p->son[*s] = new node : e->se;
      }
      if (p->el == -1) { p->el = SZ(len); len.pb(l); }
      return p->el;
   }

   void calcLink () {
      vector<node*> l;
      node *w;
      root.lnk = root.wo = 0;
      FOREACH(it,root.son) { l.pb(it->se); it->se->lnk = &root; }
      FOR(x,SZ(l)) {
         l[x]->wo = (l[x]->lnk->el != -1) ? l[x]->lnk : l[x]->lnk->wo;
         FOREACH(it,l[x]->son) {
            l.pb(it->se); w = l[x]->lnk; w = mv(w,it->fi); it->se->lnk = w;
         }
      }
   }

   // finds all occurences of every pattern, result: {<position, number of pattern>}
   vector<pii> searchAll(const char *s) {
      vector<pii> ans;
      node *p = &root, *r;
      for (int x = 0; s[x]; ++x) {
         p = mv(p, s[x]);
         for (r = p; r; r = r->wo) if (r->el != -1) ans.pb(mp(x - len[r->el] + 1, r-
>el));
      }
      return ans;
   }

   // finds at most one (the first) occurence of every pattern,
   // result: {<position, number of pattern>} (watch out: it breaks ->el pointers)
   vector<pii> searchFirst(const char *s) {
      vector<pii> ans;
      node *p = &root, *r, *t;
      for (int x = 0; s[x]; ++x) {
         r = p = mv(p, s[x]);
         while (r) {
            if (r->el != -1) ans.pb(mp(x - len[r->el] + 1, r->el));
            r->el = -1; t = r; r = r->wo; t->wo = 0;
         }
      }
      return ans;
   }

   // finds at every position the longest pattern occuring there;
   // result: {<position, number of pattern>}
   vector<pii> searchLongest(const char *s) {
      vector<pii> ans;
      node *p = &root, *r;
      for (int x = 0; s[x]; ++x) {
         r = p = mv(p, s[x]);
         if (r->el == -1) {
            r = r->wo;
            if (!r || r->el == -1) continue;
         }
         ans.pb(mp(x - len[r->el] + 1, r->el));
      }
      return ans;
   }
};
```

Baker (searching for a 2D pattern, linear-time)

```
// first type Aho-Corasick (only searchAll) and KMP (change string to vi)
char P[1005][1005], T[1005][1005];
int n1,n2,m1,m2;

// finds P[m1 x m2] in T[n1 x n2], returns a list of occurences: {<row,column>}
vector<pii> baker () {
   mkmp ac;
   vi rp;
   FOR(i,m1) rp.pb(1 + ac.addWord(P[i]));
   ac.calcLink();

   vector<vi> Q(n2,vi(n1,-1));
   FOR(i,n1) {
      vector<pii> srch = ac.searchAll(T[i]);
      FOREACH(x,srch) Q[x->fi][i] = 1 + x->se;
   }
   vector<pii> ans;
   FOR(j,n2) {
      vi srch = kmp(Q[j], rp);
      FOREACH(x,srch) ans.pb(mp(*x,j));
   }
   return ans;
}
```

Suffix array O(n log^2 n)

```
#define N 1000007
int sa[N], lcp[N], rnk[N], n, g[N], b[N], h;
inline bool comp (int i, int j) {
   if (i == j || g[i] != g[j]) return g[i] < g[j];
   return g[i+h] < g[j+h];
}
void init(const string &t) {
   n = SZ(t);
   REP(i,0,n) sa[i] = i, g[i] = t[i];
   g[n] = -1;
   h = b[0] = b[n] = 0;
   sort(sa, sa+n+1, comp);
   for (h = 1; b[n] != n; h *= 2) {
      sort(sa, sa+n+1, comp);
      FOR(i,n) b[i+1] = b[i] + comp(sa[i], sa[i+1]);
      REP(i,0,n) g[sa[i]] = b[i];
   }
   FOR(i,n) sa[i] = sa[i+1]; // 1-based -> 0-based
   FOR(i,n) rnk[sa[i]] = i;
   // Kasai algorithm for LCP:
   for (int i = 0, k = 0; i < n; i++) {
      if (k > 0) k--;
      if (rnk[i] == n-1) { lcp[n-1] = -1; k = 0; continue; }
      int j = sa[rnk[i]+1];
      while (t[i+k] == t[j+k]) k++;
```

```
      lcp[rnk[i]] = k;
   }
}
// runtimes for a random string:
// over {a,...,z}: 200.000 -> 0.2s, 500.000 -> 0.5s, 1.000.000 -> 1.3s
// over {a,b}:     200.000 -> 0.9s, 500.000 -> 1.8s, 1.000.000 -> 7.0s
```

Data structures

RMQ <O(n log n), O(1)>

```
// code is very fast (0.3 sec for 2 mln) but eats n log n memory
const int LOGN = 18, N = 1<<LOGN;
int rm[LOGN][N]; // rm[k][i] = min(tab[i], ..., tab[i + 2^k - 1])
void init (int *tab, int n) { // tab[0..n-1]
   FOR(i,n) rm[0][i] = tab[i];
   REP(k,1,LOGN-1) REP(i,0,n-(1<<k)) rm[k][i] = min(rm[k-1][i],
                                                    rm[k-1][i+(1<<(k-1))]);
}
int query (int a, int b) { // a <= b or get a segfault!
   int k = 31 - __builtin_clz(b-a+1);
   return min(rm[k][a], rm[k][b-(1<<k)+1]);
}
// if this is for the suffix array:
// initialization: init(s); init(lcp, SZ(s));
int lcpBetween (int a, int b) { // assume: a != b
   a = rnk[a], b = rnk[b];   if (a > b) swap(a,b);   return query(a, b-1);   }
```

Tree of minima

```
struct MinTree {
   int* el, s;
   MinTree (int h) { // domain of elements is [0..2^h-1]
      el = new int[2*(s = 1<<h)];
      FOR(x,2*s) el[x] = INF; // maybe you want 0 here?
   }
   ~MinTree() { delete [] el; }
   void Set (int p, int v) { // watch out: will overwrite a smaller value
      for (p += s, el[p] = v, p /= 2; p > 0; p /= 2) el[p] = min(el[2*p],el[2*p+1]);
   }
   int Find (int p, int k) { // min on segment [p,k]
      int m = INF; p += s; k += s;
      while (p < k) {
         if (p&1)    m = min(m, el[p++]);
         if (!(k&1)) m = min(m, el[k--]);
         p /= 2; k /= 2;
      }
      if (p == k) m = min(m, el[p]);
      return m;
   }
};
```

CoverTree

```
// remark: the tree keeps segments, not points (an atom could be [0,1], which is of
length 1, not 2). If you're inserting [a,b], maybe you should be inserting [a,b+1]
instead?
struct CoverTree {
#define nr (wp + wk + 1) >> 1
   int *el, *ma, s, p, k, il;
   CoverTree (int h) { // ends of segments are from [0..2^h-1]
      el = new int[s = 1 << (h + 1)]; ma = new int[s];
      FOR(x,s) el[x] = ma[x] = 0;
   }
~CoverTree() { delete [] el; delete [] ma; }
   void Mark (int wp, int wk, int g) {
      if (k <= wp || p >= wk) return;
      if (p <= wp && k >= wk) el[g] += il; else {
         Mark(wp, nr, 2 * g); Mark(nr, wk, 2 * g + 1);
      }
      ma[g] = el[g] > 0 ? wk - wp : (wk - 1 == wp ? 0 : ma[2 * g] + ma[2 * g + 1]);
   }
   // add i1 occurences of segment [p1..k1] to the tree. i1 can be negative, then
   // the occurences are deleted. Do not delete segments that you have not inserted
   void Add (int p1, int k1, int i1) { p = p1; k = k1; il = i1; Mark(0, s / 2, 1); }
   int F (int wp, int wk, int g) {
      if (wp >= k || wk <= p) return 0;
      if (el[g] > 0) return min(k, wk) - max(p, wp);
      if (p <= wp && wk <= k) return ma[g];
      return wp == wk - 1 ? 0 : F(wp, nr, 2 * g) + F(nr, wk, 2 * g + 1);
   }
   // compute the length of the part of [p1,k1] which is covered by segments
   int Find (int p1, int k1) { p = p1; k = k1; return F(0, s / 2, 1); }
};
```

Tree for incrementing a segment and taking minimum on a segment

```
// tree on interval 0..n-1
// before usage set MAXX, n and call init()
// inc increments values  on interval [aw,bw] by x
// getmin returns minimum on interval [aw,bw]
// MAXX should be 2 * (next power of two after n) + 10
#define MAXX 524298
ll incr[MAXX], minn[MAXX]; int n;
void init () { FOR(i,MAXX) incr[i] = minn[i] = 0; }
int aw,bw,x; // temp

void inc2 (int v, int a, int b) {
   if (aw > b || bw < a) return;
   if (a >= aw && b <= bw) {
      incr[v] += x;
      minn[v] += x;
   } else {
      int mid = (a+b)/2;
      inc2(2*v, a, mid);
      inc2(2*v+1, mid+1, b);
      minn[v] = incr[v] + min(minn[2*v], minn[2*v + 1]);
   }
}
void inc (int _aw, int _bw, int _x) {
   aw = _aw; bw = _bw; x = _x;
   inc2(1,0,n-1);
}
ll getmin (int v, int a, int b) {
   if (aw > b || bw < a) return (ll)INF * INF;
   if (a >= aw && b <= bw) {
      return minn[v];
   } else {
      int mid = (a+b)/2;
```

```
      return incr[v] + min(getmin(2*v, a, mid), getmin(2*v+1, mid+1, b));
   }
}
ll getmin (int _aw, int _bw) {
   aw = _aw; bw = _bw;
   return getmin(1,0,n-1);
}
```

Fenwick tree + range updates

```
#define N 200007
void add (ll *a, int n, ll x) { // val[n] += x
   for (; n < N; n |= n+1) a[n] += x;
}
ll sum (ll *a, int n) { // val[0] + val[1] + ... + val[n]
   ll s = 0;
   while (n >= 0) {
      s += a[n];
      n = (n&(n+1))-1;
   }
   return s;
}
ll sumbetween(ll *a, int le, int ri) { // val[le] +...+ val[ri]. le=0 is ok
   return sum(a,ri) – sum(a,le-1);
}


// RANGE QUERIES + RANGE UPDATES
ll tadd[N], tmul[N], tsq[N];
void addOnSegment (int le, int ri, ll x) { // add to [le..ri]: [x,x,...,x]
   add(tmul, le, x);  add(tadd, le, -x * (le-1));
   add(tmul, ri, -x); add(tadd, ri,  x * ri);
}
void addSaw (int le, int ri, ll x) { // add to [le,ri]: [x,2x,...,(ri-le+1)x]
   add(tsq, le, x);    add(tadd, le,  -x * le * (le-1) / 2);
   add(tsq, ri+1, -x); add(tadd, ri+1, x * ri * (ri+1) / 2);
   addOnSegment(le, ri, -x * (le-1));
}
void addBackSaw (int le, int ri, ll x) { // add to [le,ri]: [(ri-le+1)x,...,2x,x]
   addSaw(le,ri,-x); addOnSegment(le,ri,(ri-le+2)*x);
}
ll query (int n) { // get sum[0..n]
   return sum(tadd,n) + sum(tmul,n) * n + sum(tsq,n) * n*(n+1)/2;
}
// RANGE QUERIES + RANGE UPDATES end

// returns min p st sum[0,p] > s (or N if no such p exists)
int lowerBound (ll *a, ll s) {
   int pos = -1;
   for (int bs = 1 << (31 - __builtin_clz(N)); bs != 0; bs /= 2) {
      int nxt = pos + bs;
      if (nxt < N && s >= a[nxt]) { s -= a[nxt]; pos = nxt; }
   }
   return pos + 1;
}
```

Convex hull of lines (online, O(log n) per operation)

```
const ll INF = 1LL<<61; bool cmpA;
struct Line { // ax+b. xl = the x-coord of intersection with previous line in set
   ll a, b;
   mutable long double xl;
   bool operator < (const Line &l) const { if (cmpA) return a < l.a; else return xl
< l.xl; } };
struct DynamicHull : multiset<Line> {
   iterator prev (iterator y) { return --y; } // skip if using c++0x/c++11
   iterator next (iterator y) { return ++y; } // skip if using c++0x/c++11
   bool bad (iterator y) {
      iterator z = next(y), x;
      if (y == begin()) {
         if (z == end()) return 0;
         return y->a == z->a && y->b <= z->b;
      }
      x = prev(y);
      if (z == end()) return y->a == x->a && y->b <= x->b;
      return 1.0L * (x->b - y->b) * (z->a - y->a) >= 1.0L * (y->b - z->b) * (y->a -
x->a);
   }
   void add (ll a, ll b) { // add a line ax+b to the convex hull
      cmpA = 1; iterator y = insert((Line){a,b,-INF});
      if (bad(y)) { erase(y); return; }
      while (next(y) != end() && bad(next(y))) erase(next(y));
      while (y != begin() && bad(prev(y))) erase(prev(y));
      if (next(y) != end()) next(y)->xl = 1.0L * (y->b - next(y)->b) / (next(y)->a -
y->a);
      if (y != begin()) y->xl = 1.0L * (y->b - prev(y)->b) / (prev(y)->a - y->a);
   }
   ll eval (ll x) { // find max ax+b, over all lines added, for given x
      if (empty()) return -INF;
      cmpA = 0; iterator it = prev(lower_bound((Line){0,0,1.0L*x}));
      return it->a * x + it->b;
   }
};
```

Augmented Treap

```
// tree cannot store duplicates. If trying to insert a duplicate, insert uses +. For
example, when + is the regular sum of ints, then: tr.insert(6,2); tr.insert(6,3);
will produce a tree containing <6,5>
// you can change this behavior to replacement by uncommenting the penultimate line
of insert_aux. Then in our example we would get <6,3>.

template <typename T, typename Aug> struct AugTreap {
   struct Node {
      T v; // value
      int pr; // priority
      Node *le, *ri;
      Aug al, ag; // aug for the node and for the whole subtree
   };
   Node *root;
   AugTreap() : root(0) {}
   inline static void recalc(Node* &n) { // compute new ag for n
      Aug left = n->le ? n->le->ag + n->al : n->al;
      n->ag = n->ri ? left + n->ri->ag : left;
   }
```

```
   inline void rotle(Node* &n) { // bring left child of n up
      Node *syn = n->le;
      n->le = syn->ri;
      syn->ri = n;
      n = syn;
      recalc(n->ri);
      recalc(n);
   }
   inline void rotri(Node* &n) { // bring right child of n up
      Node *syn = n->ri;
      n->ri = syn->le;
      syn->le = n;
      n = syn;
      recalc(n->le);
      recalc(n);
   }
   void insert_aux(Node* &n, const T &x, const Aug &aug) {
      if (!n) {
         n = new Node;
         n->v = x;
         n->pr = rand();
         n->le = n->ri = 0;
         n->al = n->ag = aug;
      } else if (x < n->v) {
         insert_aux(n->le, x, aug);
         if (n->pr > n->le->pr) rotle(n);
         else recalc(n);
      } else if (x > n->v) {
         insert_aux(n->ri, x, aug);
         if (n->pr > n->ri->pr) rotri(n);
         else recalc(n);
      } else { // x is in the tree - we update
         n->al = n->al + aug;
         // n->al = aug; // OVERWRITE
         recalc(n);
      }
   }
   void insert (const T &x, const Aug &aug) { insert_aux(root, x, aug); }

   Aug sumgeq_aux (const T &x, Node *n) {
      if (!n) return Aug();

      if (x > n->v) return sumgeq_aux(x, n->ri);
      if (x < n->v) {
         Aug left = sumgeq_aux(x, n->le) + n->al;
         return n->ri ? left + n->ri->ag : left;
      }
      // x == n->v
      return n->ri ? n->al + n->ri->ag : n->al;
   }
   // sum of Augs over all keys >= x
   Aug sumgeq (const T &x) { return sumgeq_aux(x,root); }

// OPTIONAL: SUMS IN A SEMIGROUP
// (if Aug is a group, then instead of sumbetween(L,R)
//  you can do sumgeq(L)-sumgeq(R+1))

   Aug sumleq_aux (const T &x, Node *n) {
      if (!n) return Aug();
      if (x < n->v) return sumleq_aux(x, n->le);
      if (x > n->v) {
         Aug right = n->al + sumleq_aux(x, n->ri);
         return n->le ? n->le->ag + right : right;
      }
      // x == n->v
      return n->le ? n->le->ag + n->al : n->al;
   }
   // sum of Augs over all keys <= x
   Aug sumleq (const T &x) { return sumleq_aux(x,root); }
   Aug sumbetween_aux (const T &x, const T &y, Node *n) {
      if (!n) return Aug();
      if (y < n->v) return sumbetween_aux(x,y,n->le);
      if (x > n->v) return sumbetween_aux(x,y,n->ri);
      // x <= n->v <= y
      return sumgeq_aux(x,n->le) + n->al + sumleq_aux(y,n->ri);
   }
   // sum of Augs over all keys from [x,y]
   Aug sumbetween (const T &x, const T &y) { return sumbetween_aux(x,y,root); }

// OPTIONAL: REMOVAL (instead of removing you can usually set Aug of element to 0)

   bool remove (const T &x) { return remove_aux(root,x); }
   bool remove_aux (Node* &n, const T &x) {
      if (!n) return 0;
      if (x < n->v) {
         bool res = remove_aux(n->le, x); if (res) recalc(n); return res; }
      if (x > n->v) {
         bool res = remove_aux(n->ri, x); if (res) recalc(n); return res; }
      // got him
      remove_node(n); return 1;
   }
   void remove_node (Node* &n) {
      if (!n->le) { Node *syn = n->ri; delete n; n = syn; }
      else if (!n->ri) { Node *syn = n->le; delete n; n = syn; }
      else if (n->le->pr < n->ri->pr) { rotle(n);remove_node(n->ri); recalc(n); }
      else { rotri(n); remove_node(n->le); recalc(n); }
   }
};

// Aug should be a semigroup (associative) with zero
// zero is the value of zero-parameter constructor (i.e. Aug())
// the action is the operator +
// (remark: int() = 0, so you can just do AugTreap<T,int>)
// another example of an Aug class, for taking minimums:
struct IntMin {
   int x;
   IntMin(int _x = INF) : x(_x) {}
   inline IntMin operator + (const IntMin &s) const { return IntMin(min(x,s.x)); }
};
```

Union-Find

```
struct UnionFind {
   vi tab, cnt;
   UnionFind(int n) : // creates UF on set 0, 1, ..., n-1
      tab(n), cnt(n,1) {
      FOR(i,n) tab[i] = i;
   }
```

```
   int find(int a) {
      if (tab[a] == a) return a;
      return tab[a] = find(tab[a]);
   }
   bool unia(int a, int b) { // true if there really was a union
      a = find(a), b = find(b);
      if (a==b) return 0;
      if (cnt[a] > cnt[b]) swap(a,b);
      cnt[b] += cnt[a];
      tab[a] = b;
      return 1;
   }
};
```

Geometry

```
#define real double // or: long double
const real eps=1e-9;
inline bool iszero(real x){return x<=eps && x>=-eps;}
struct pt {
    real x,y;
    pt(real xx=0,real yy=0):x(xx),y(yy){}
    bool operator==(pt &a){return iszero(a.x-x) && iszero(a.y-y);}
};
bool operator<(const pt &a, const pt &b) {
    if (a.x!=b.x) return a.x<b.x;
    return a.y<b.y;
}
ostream& operator<<(ostream &s,pt p) {return s<<"("<<p.x<<","<<p.y<<")";}

pt operator+(pt a,pt b){return pt(a.x+b.x,a.y+b.y);}
pt operator-(pt a,pt b){return pt(a.x-b.x,a.y-b.y);}
pt operator*(pt a,real r){return pt(a.x*r,a.y*r);}
real vec(pt a,pt b){return a.x*b.y-a.y*b.x;}
real det(pt a,pt b,pt c){return vec(b-a,c-a);}
// det(a,b,c) >= 0 iff A,B,C are in counterclockwise order

pt operator*(pt a,pt b){return pt(a.x*b.x-a.y*b.y,b.x*a.y+b.y*a.x);}
real sqabs(pt a){return a.x*a.x+a.y*a.y;}
pt operator/(pt a,pt b) {return (a*pt(b.x,-b.y))/sqabs(b);}

real abs(pt a){return sqrt(a.x*a.x+a.y*a.y);}
real dist(pt a,pt b){return sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));}
real sqdist(pt a,pt b){return (a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y);}

real arg(pt a){return atan2(a.y,a.x);}//z przedzialu [-pi,pi]
real scal(pt a,pt b){return a.x*b.x+a.y*b.y;}

// distance of point A to line BC
real dist(pt a,pt b,pt c){return abs(det(b,c,a))/dist(b,c);}
// length (signed) of projection of point A onto line 0,B
real lenproj(pt a,pt b){return scal(a,b)/abs(b);}

pt proj(pt a,pt b,pt c) { // projection of point A onto line BC
    pt d=c-b;
    return b+d*(scal(a-b,d)/sqabs(d));
}

bool insegment(pt a,pt b,pt c) { // is point A on segment BC
    if (iszero(det(a,b,c)))
        if (min(b.x,c.x)-eps<=a.x && a.x-eps<=max(b.x,c.x))
            if (min(b.y,c.y)-eps<=a.y && a.y-eps<=max(b.y,c.y)) return 1;
    return 0;
}

bool intersection(pt a,pt b,pt c,pt d) { // is intersection of AB and CD nonempty
    real d1=vec(b-a,c-a),d2=vec(b-a,d-a);
    if ((d1>eps && d2>eps) || (d1<-eps && d2<-eps)) return 0;
    if (iszero(d1) && iszero(d2)) {
        if (iszero(a.x-b.x) && iszero(c.x-d.x))
            a=a*pt(0,1);b=b*pt(0,1);c=c*pt(0,1);d=d*pt(0,1);
        if (a.x>b.x) swap(a,b);
        if (c.x>d.x) swap(c,d);
        if (a.x<c.x+eps && c.x<b.x+eps) return 1;
        if (a.x<d.x+eps && d.x<b.x+eps) return 1;
        if (c.x<a.x+eps && a.x<d.x+eps) return 1;
        return 0;
    }
    d1=vec(d-c,a-c),d2=vec(d-c,b-c);
    if ((d1>eps && d2>eps) || (d1<-eps && d2<-eps)) return 0;
    return 1;
}

// do AB and CD intersect in exactly one point, which is not A,B,C or D?
bool intersection_proper(pt a,pt b,pt c,pt d) {
    real d1=vec(b-a,c-a),d2=vec(b-a,d-a);
    if (!(d1>eps && d2<-eps) && !(d1<-eps && d2>eps)) return 0;
    d1=vec(d-c,a-c),d2=vec(d-c,b-c);
    if (!(d1>eps && d2<-eps) && !(d1<-eps && d2>eps)) return 0;
    return 1;
}

// 0 - none, 1 - intersection, 2 - overlap
int line_cross(pt a, pt b,pt c,pt d,pt& res) {
    real pczw=vec(b-a,c-d);
    if (iszero(pczw)) {
        if (iszero(det(a,b,c))) return 2;
        else return 0;
    }
    real ptr=vec(b-a,c-a);
    res=c+(d-c)*(ptr/pczw);
    return 1;
}

// intersection of two circles
vector<pt> circle_cross(pt c1,real c1r,pt c2,real c2r)  {
    vector<pt> res;
    real d=sqabs(c2-c1), r1=c1r*c1r/d, r2=c2r*c2r/d;
    pt u=c1*((r2-r1+1)*0.5)+c2*((r1-r2+1)*0.5);
    if (r1>r2) swap(r1,r2);
    real a=(r1-r2+1)*0.5; a*=a;
    if (a>=r1+eps) return res;
    if (a>r1-eps) {res.pb(u);return res;}
    pt v(c2-c1);
    v=pt(-v.y,v.x);
    real h=sqrt(r1-a);
    res.pb(u+v*h);
    res.pb(u-v*h);
    return res;
```

```
}

// intersection of circle with line AB
vector<pt> circle_line_cross(pt c,real cr,pt a,pt b) {
    vector<pt> r;
    pt d=proj(c,a,b);
    real X=dist(c,d);
    if (iszero(X-cr)){r.pb(d);return r;} // one point
    if (X>cr) return r; // line is too far
    real Y=sqrt(cr*cr-X*X);
    pt K=b-a;
    K=K*(Y/abs(K));
    r.pb(d+K);r.pb(d-K);
    return r;
}

bool circle_3points(pt a,pt b,pt c,pt &sr,real &r) {
    pt sym1[2],sym2[2];
    sym1[0]=(a+b)*0.5;sym1[1]=sym1[0]+(b-a)*pt(0,10.0);
    sym2[0]=(b+c)*0.5;sym2[1]=sym2[0]+(c-b)*pt(0,10.0);
    pt center;
    if (line_cross(sym1[0],sym1[1],sym2[0],sym2[1],sr)!=1) return 0;
    r=dist(sr,a);
    return 1;
}

// is A inside POL? if it's on the boundary, you might get either answer
bool inpoly(pt a, vector<pt> &pol) {
    pt b(3e8+500.0,4e6+77777.0);
    int pr=0;
    FOR(i,SZ(pol)) pr+=intersection_proper(a,b,pol[i],pol[(i+1)%SZ(pol)]);
    return pr%2;
}

bool onborder(pt a, vector<pt> &pol) { // is A on boundary of POL
    FOR(i,SZ(pol)) if (insegment(a,pol[i],pol[(i+1)%SZ(pol)])) return 1;
    return 0;
}

// true iff inside or on border. use for real = ll!
bool PointInConvexPol(vector<pt>& l, pt p) {
    int a = 1, b = SZ(l)-1, c;
    if (det(l[0], l[a], l[b]) > 0) swap(a,b);
    if (det(l[0], l[a], p) > 0 || det(l[0], l[b], p) < 0) return 0;
    while(abs(a-b) > 1) {
        c = (a+b)/2;
        if (det(l[0], l[c], p) > 0) b = c; else a = c;
    }
    return det(l[a], l[b], p) <= 0;
}
// true iff strictly inside. use for real = ll!
bool PointInsideConvexPol(vector<pt>& l, pt p) {
    int a = 1, b = SZ(l)-1, c;
    if (det(l[0], l[a], l[b]) > 0) swap(a,b);
    if (det(l[0], l[a], p) >= 0 || det(l[0], l[b], p) <= 0) return 0;
    while(abs(a-b) > 1) {
        c = (a+b)/2;
        if (det(l[0], l[c], p) > 0) b = c; else a = c;
    }
    return det(l[a], l[b], p) < 0;
}

// area of PO. for integral arguments, result is half-integral
real pole(vector<pt> &po) {
    real pole=0.0;
    int dl=SZ(po);
    FOR(i,dl) pole+=po[i].x*po[(i+1)%dl].y-po[(i+1)%dl].x*po[i].y;
    return fabs(pole)/2.0;
}

// intersection of convex polygon P with the half-plane {x:det(a,b,x)<=0}
vector<pt> poly_halfplane(vector<pt> p,pt a,pt b) { // complexity O(|p|)
    int n=SZ(p);
    if (!n) return p;
    p.pb(p[0]);
    vector<pt> res;
    vector<bool> side(n+1);
    pt cross;
    FOR(i,n+1) side[i]=(det(a,b,p[i])>=-eps);
    FOR(i,n) {
        if (side[i]==1) {
            res.pb(p[i]);
            if (side[i+1]==0 && line_cross(p[i],p[i+1],a,b,cross)==1
                             && !(cross==p[i])) res.pb(cross);
        }
        if (side[i]==0 && side[i+1]==1 && line_cross(p[i],p[i+1],a,b,cross)==1
                       && !(cross==p[i+1])) res.pb(cross);
    }
    return res;
}

bool right(pt X,pt Y,pt Z){return det(X,Y,Z)<=eps;}
// version without having to type det:
// bool right(pt X,pt Y,pt Z){return ((X.x-Z.x)*(Y.y-Z.y)-(X.y-Z.y)*(Y.x-
Z.x))<=eps;}
// for integers, change this to det(X,Y,Z)<=0
// right(X,Y,Z) iff Z is on the right of line XY

// returns the 2D convex hull in counterclockwise direction.
// consecutive vertices are not collinear.
// also works with pii instead of pt, after just changing x->fi, y->se
//      and pt->pii in function right. Then you don't have to define pt, == or <
vector<pt> convexhull(vector<pt> ab) {
    sort(ab.begin(),ab.end());
    ab.erase(unique(ab.begin(),ab.end()),ab.end());
    int l=SZ(ab),i,j,k;
    vector<pt> res(l+1);
    if (l<3) return ab;
    j=0;
    for(i=0;i<l;i++) {
        while (j-2>=0 && right(res[j-2],res[j-1],ab[i])) j--;
        res[j++]=ab[i];
    }
    k=j;
    for(i=l-2;i>=0;i--) {
        while (j-1>=k && right(res[j-2],res[j-1],ab[i])) j--;
        res[j++]=ab[i];
    }
```

```
    if (res[j-1]==res[0]) j--;
    return vector<pt>(res.begin(),res.begin()+j);
}
```

Closest pair of points

```
struct Point {
   int x,y;
   Point(int _x = 0, int _y = 0) : x(_x),y(_y) {}
   bool operator == (Point &a) { return a.x == x && a.y == y; }
};
bool OrdXY(Point *a, Point *b) { return a->x == b->x ? a->y < b->y : a->x < b->x; }
bool OrdYX(Point *a, Point *b) { return a->y == b->y ? a->x < b->x : a->y < b->y; }
inline double sqr (int a) { return (double)a*a; }
struct NearestPoints {
   vector<Point*> l;
   Point *p1, *p2;
   double dist;
   void Filter(vector<Point*> &l, double p) {
      int s = 0;
      FOR(x,SZ(l)) if (sqr(l[x]->x - p) <= dist) l[s++] = l[x];
      l.resize(s);
   }
   void Calc (int p, int k, vector<Point*> &ys) {
      if (k - p > 1) {
         vector<Point*> lp, rp;
         int c = (k+p-1)/2;
         FOREACH(it,ys) { if (OrdXY(l[c], *it)) rp.pb(*it); else lp.pb(*it); }
         Calc(p, c+1, lp); Calc(c+1, k, rp);
         Filter(lp, l[c]->x); Filter(rp, l[c]->x);
         int p = 0; double k; int rps = SZ(rp)-1;
         FOREACH(it,lp) {
            while (p < rps && rp[p+1]->y < (*it)->y) p++;
            REP(x,max(0,p-2),min(rps,p+1)) {
              if (dist > (k = sqr((*it)->x - rp[x]->x) + sqr((*it)->y - rp[x]->y))){
                  dist = k; p1 = *it; p2 = rp[x];
               }
            }
         }
      }
   }
   NearestPoints(vector<Point> &p) {
      FOREACH(it,p) l.pb(&(*it));
      sort(ALL(l), OrdXY);
      FORI(x,SZ(l)-1) {
         if (l[x-1]->x == l[x]->x && l[x-1]->y == l[x]->y) {
            dist = 0; p1 = l[x-1]; p2 = l[x]; return;
         }
      }
      dist = double(INF)*double(INF);
      vector<Point*> v = l;
      sort(v.begin(), v.end(), OrdYX);
      Calc(0, SZ(l), v);
      dist = sqrt(dist);
   }
};

int main () { // example of usage
   int n; Point p; vector<Point> l;
   cin >> n; FOR(x,n) { cin >> p.x >> p.y; l.pb(p); }
   NearestPoints str(l);
   DBG(str.dist)   DBG(mp(str.p1->x, str.p1->y))   DBG(mp(str.p2->x, str.p2->y))
}
```

Geometry in 3D

```
struct pt{
  ll x,y,z;
  pt(ll X=0,ll Y=0,ll Z=0) : x(X), y(Y), z(Z) {}
};

bool operator <(pt a,pt b){
  if (a.x-b.x) return a.x<b.x;
  if (a.y-b.y) return a.y<b.y;
  return a.z<b.z;
}
ll det(pt a,pt b,pt c) { return a.x*b.y*c.z+a.y*b.z*c.x+a.z*b.x*c.y-
a.z*b.y*c.x-a.x*b.z*c.y-a.y*b.x*c.z; }
pt operator ^(pt a,pt b){ return pt(a.y*b.z-a.z*b.y, a.z*b.x-a.x*b.z,
a.x*b.y-a.y*b.x); } // vector product
pt operator -(pt a,pt b){ return pt(a.x-b.x,a.y-b.y,a.z-b.z); }
ll operator *(pt a,pt b){ return a.x*b.x+a.y*b.y+a.z*b.z; }
bool operator ==(pt a,pt b){ return a.x==b.x && a.y==b.y && a.z==b.z; }
double norm(pt a){ return sqrt(a.x*a.x+a.y*a.y+a.z*a.z); }

// distance of point Z to plane A,B,C
double plane_dist(pt a,pt b,pt c,pt z) {
    double basearea=norm((b-a)^(c-a));
    double vol=det(a-z,b-z,c-z);
    return abs(vol/basearea);
}
```

Misc

Bit tricks (more in the math appendix)

Counting set bits: `__builtin_popcount(x)` or `__builtin_popcountll(x)`
Most significant set bit in a word: `ll f = 1LL << (63 - __builtin_clzll(x));`

```
// iterating over nonempty subsets of mask:
for (int x = mask; x > 0; x = (x-1) & mask) { ... }
// hack: fast minimum/maximum (because of no ifs):
inline int fastMax(int x, int y) { return (((y-x)>>(32-1))&(x^y))^y; }
inline int fastMin(int x, int y) { return (((y-x)>>(32-1))&(x^y))^x; }
```

LIS (longest increasing subsequence)

```
int a[1000007]; // a[j] = min x st there is an increasing subsequence of length j+1
                // ending with x
int lis (const vi &v) {
   int q = 0;
   FOREACH(y,v) {
      int k = lower_bound(a,a+q,*y) - a;
      a[k] = *y;
      if (k == q) q++;
   }
   return q;
}
```

Sqrt-decomposition (Mo's algorithm)

```
inline void add (int i) {}    inline void erase (int i) {}    ll getres () {}
// Usage: set Q, sq=sqrt(n), q, le[0..q-1], ri[0..q-1] and
// provide add, erase, getres. Then go() fills in res[0..q-1].
#define Q 200006
int sq, q, le[Q], ri[Q]; // q = number of queries [0 .. q-1]
ll res[Q]; // output
bool comp (int i, int j) { return mp(le[i] / sq, ri[i]) < mp(le[j] / sq, ri[j]); }
void go () {
   vi tests;
   FOR(i,q) tests.pb(i);
   sort(ALL(tests),comp);
   int cl = 1, cr = 0; // [cl,cr]
   FOREACH(it,tests) {
      int i = *it, l = le[i], r = ri[i];
      while (cl > l) add(--cl);
      while (cr < r) add(++cr);
      while (cl < l) erase(cl++);
      while (cr > r) erase(cr--);
      res[i] = getres();
   }
}
```

Measuring time

```
#include <ctime> (probably unnecessary)
clock_t startTime = clock();
while (clock() - startTime < 1.8 * CLOCKS_PER_SEC) { iterate; }
```

FastIO (short)

```
const int max_size=100000;
char buf[max_size];
int buf_size=0,pos=0;
#define getbuf() buf_size=fread(buf,1,max_size,stdin)
#define get(tok) {if (pos>=buf_size) { if (feof(stdin)) tok=0;    else
{ getbuf();pos=0; tok=buf[pos++];}}else tok=buf[pos++];}
#define in(v) {int tok;get(tok);while (tok==' ' ||
tok=='\n')get(tok);v=tok-'0';get(tok);while(tok>='0' && tok<='9')
{v*=10;v+=(tok-'0');get(tok);}}
```