IME ACM-ICPC Team Notebook

Contents

1 Template + vimrc

1.1 Template

```cpp
#include <bits/stdc++.h>
using namespace std;

#define st first
#define nd second
#define mp make_pair
#define pb push_back
#define cl(x, v) memset((x), (v), sizeof(x))
#define gcd(x,y) __gcd((x),(y))

#ifndef ONLINE_JUDGE
  #define db(x) cerr << #x << " == " << x << endl
  #define dbs(x) cerr << x << endl
  #define _ << ", " <<
#else
  #define db(x) ((void)0)
  #define dbs(x) ((void)0)
#endif

typedef long long ll;
typedef long double ld;

typedef pair<int, int> pii;
typedef pair<int, pii> piii;
typedef pair<ll, ll> pll;
typedef vector<int> vi;

const ld EPS = 1e-9, PI = acos(-1.);
```

```
const int INF = 0x3f3f3f3f, MOD = 1e9+7;
const int N = 1e5+5;

int main() {
  //freopen("in", "r", stdin);
  //freopen("out", "w", stdout);
  return 0;
}
```

1.2 vimrc

```
syntax on
set et ts=2 sw=0 sts=-1 ai nu hls cindent
nnoremap ; :
vnoremap ; :
noremap <c-j> 15gj
noremap <c-k> 15gk
nnoremap <s-k> i<CR><ESC>
inoremap ,. <esc>
vnoremap ,. <esc>
nnoremap ,. <esc>
```

2 Dynamic Programming

2.1 Convex Hull Trick (emaxx)

```
struct Point{
  ll x, y;
  Point(ll x = 0, ll y = 0):x(x), y(y) {}
  Point operator-(Point p){ return Point(x - p.x, y - p.y); }
  Point operator+(Point p){ return Point(x + p.x, y + p.y); }
  Point ccw(){ return Point(-y, x); }
  ll operator%(Point p){ return x*p.y - y*p.x; }
  ll operator*(Point p){ return x*p.x + y*p.y; }
  bool operator<(Point p) const { return x == p.x ? y < p.y : x < p.x; }
};

pair<vector<Point>, vector<Point>> ch(Point *v){
  vector<Point> hull, vecs;
  for(int i = 0; i < n; i++){
    if(hull.size() and hull.back().x == v[i].x) continue;

    while(vecs.size() and vecs.back()*(v[i] - hull.back()) <= 0)
      vecs.pop_back(), hull.pop_back();

    if(hull.size())
      vecs.pb((v[i] - hull.back()).ccw());

    hull.pb(v[i]);
  }
  return {hull, vecs};
}

ll get(ll x) {
    Point query = {x, 1};
    auto it = lower_bound(vecs.begin(), vecs.end(), query, [](Point a, Point b) {
        return a%b > 0;
    });
    return query*hull[it - vecs.begin()];
}
```

2.2 Convex Hull Trick

```
// Convex Hull Trick

// ATTENTION: This is the maximum convex hull. If you need the minimum
// CHT use {-b, -m} and modify the query function.

// In case of floating point parameters swap long long with long double
typedef long long type;
struct line { type b, m; };

line v[N]; // lines from input
int n; // number of lines
// Sort slopes in ascending order (in main):
sort(v, v+n, [](line s, line t){
     return (s.m == t.m) ? (s.b < t.b) : (s.m < t.m); });

// nh: number of lines on convex hull
// pos: position for linear time search
// hull: lines in the convex hull
int nh, pos;
line hull[N];

bool check(line s, line t, line u) {
  // verify if it can overflow. If it can just divide using long double
  return (s.b - t.b)*(u.m - s.m) < (s.b - u.b)*(t.m - s.m);
}

// Add new line to convex hull, if possible
// Must receive lines in the correct order, otherwise it won't work
void update(line s) {
  // 1. if first lines have the same b, get the one with bigger m
  // 2. if line is parallel to the one at the top, ignore
  // 3. pop lines that are worse
  // 3.1 if you can do a linear time search, use
  // 4. add new line

  if (nh == 1 and hull[nh-1].b == s.b) nh--;
  if (nh > 0  and hull[nh-1].m >= s.m) return;
  while (nh >= 2 and !check(hull[nh-2], hull[nh-1], s)) nh--;
  pos = min(pos, nh);
  hull[nh++] = s;
}

type eval(int id, type x) { return hull[id].b + hull[id].m * x; }

// Linear search query - O(n) for all queries
// Only possible if the queries always move to the right
type query(type x) {
  while (pos+1 < nh and eval(pos, x) < eval(pos+1, x)) pos++;
  return eval(pos, x);
  // return -eval(pos, x);    ATTENTION: Uncomment for minimum CHT
}

// Ternary search query - O(logn) for each query
/*
type query(type x) {
  int lo = 0, hi = nh-1;
  while (lo < hi) {
    int mid = (lo+hi)/2;
    if (eval(mid, x) > eval(mid+1, x)) hi = mid;
    else lo = mid+1;
  }
  return eval(lo, x);
  // return -eval(lo, x);     ATTENTION: Uncomment for minimum CHT
}

// better use geometry line_intersect (this assumes s and t are not parallel)
ld intersect_x(line s, line t) { return (t.b - s.b)/(ld)(s.m - t.m); }
ld intersect_y(line s, line t) { return s.b + s.m * intersect_x(s, t); }
*/
```

2.3 Divide and Conquer Optimization

```
// Divide and Conquer DP Optimization - O(k*n^2) => O(k*n*logn)
//
// dp[i][j] = min k<i { dp[k][j-1] + C[k][i] }
//
// Condition: A[i][j] <= A[i+1][j]
// A[i][j] is the smallest k that gives an optimal answer to dp[i][j]
//
// reference (pt-br): https://algorithmmarch.wordpress.com/2016/08/12/a-otimizacao-de-pds-e-o-garcom-da-
     maratona/

int n, maxj;
int dp[N][J], a[N][J];

// declare the cost function
int cost(int i, int j) {
  // ...
}

void calc(int l, int r, int j, int kmin, int kmax) {
  int m = (l+r)/2;
  dp[m][j] = LINF;
```

```
  for (int k = kmin; k <= kmax; ++k) {
    ll v = dp[k][j-1] + cost(k, m);

    // store the minimum answer for d[m][j]
    // in case of maximum, use v > dp[m][j]
    if (v < dp[m][j]) a[m][j] = k, dp[m][j] = v;
  }

  if (l < r) {
    calc(l,   m, j, kmin,    a[m][k]);
    calc(m+1, r, j, a[m][k], kmax   );
  }
}

// run for every j
for (int j = 2; j <= maxj; ++j)
  calc(1, n, j, 1, n);
```

2.4 Knuth Optimization

```
// Knuth DP Optimization - O(n^3) -> O(n^2)
//
// 1) dp[i][j] = min i<k<j { dp[i][k] + dp[k][j] } + C[i][j]
// 2) dp[i][j] = min k<i { dp[k][j-1] + C[k][i] }
//
// Condition: A[i][j-1] <= A[i][j] <= A[i+1][j]
// A[i][j] is the smallest k that gives an optimal answer to dp[i][j]
//
// reference (pt-br): https://algorithmmarch.wordpress.com/2016/08/12/a-otimizacao-de-pds-e-o-garcom-da-
    maratona/
//

// 1) dp[i][j] = min i<k<j { dp[i][k] + dp[k][j] } + C[i][j]
int n;
int dp[N][N], a[N][N];

// declare the cost function
int cost(int i, int j) {
  // ...
}

void knuth() {
  // calculate base cases
  memset(dp, 63, sizeof(dp));
  for (int i = 1; i <= n; i++) dp[i][i] = 0;

  // set initial a[i][j]
  for (int i = 1; i <= n; i++) a[i][i] = i;

  for (int j = 2; j <= n; ++j)
    for (int i = j; i >= 1; --i)
      for (int k = a[i][j-1]; k <= a[i+1][j]; ++k) {
        ll v = dp[i][k] + dp[k][j] + cost(i, j);

        // store the minimum answer for d[i][k]
        // in case of maximum, use v > dp[i][k]
        if (v < dp[i][j])
          a[i][j] = k, dp[i][j] = v;
      }
}


// 2) dp[i][j] = min k<i { dp[k][j-1] + C[k][i] }
int n, maxj;
int dp[N][J], a[N][J];

// declare the cost function
int cost(int i, int j) {
  // ...
}

void knuth() {
  // calculate base cases
  memset(dp, 63, sizeof(dp));
  for (int i = 1; i <= n; i++) dp[i][1] = // ...

  // set initial a[i][j]
  for (int i = 1; i <= n; i++) a[i][1] = 1, a[n+1][i] = n;

  for (int j = 2; j <= maxj; j++)
    for (int i = n; i >= 1; i--)
      for (int k = a[i][j-1]; k <= a[i+1][j]; k++) {
        ll v = dp[k][j-1] + cost(k, i);

        // store the minimum answer for d[i][k]
        // in case of maximum, use v > dp[i][k]
        if (v < dp[i][j])
          a[i][j] = k, dp[i][j] = v;
      }
}
```

2.5 Longest Increasing Subsequence

```
// Longest Increasing Subsequence - O(nlogn)
//
// dp(i) = max j<i { dp(j) | a[j] < a[i] } + 1
//

// int dp[N], v[N], n, lis;

memset(dp, 63, sizeof dp);
for (int i = 0; i < n; ++i) {
  // increasing: lower_bound
  // non-decreasing: upper_bound
  int j = lower_bound(dp, dp + lis, v[i]) - dp;
  dp[j] = min(dp[j], v[i]);
  lis = max(lis, j + 1);
}
```

2.6 SOS DP

```
// O(N * 2^N)
// A[i] = initial values
// Calculate F[i] = Sum of A[j] for j subset of i
for(int i = 0; i < (1 << N); i++)
        F[i] = A[i];
for(int i = 0; i < N; i++)
        for(int j = 0; j < (1 << N); j++)
                if(j & (1 << i))
                        F[j] += F[j ^ (1 << i)];
```

2.7 Steiner tree

```
// Steiner-Tree O(2^t*n^2 + n*3^t + APSP)

// N - number of nodes
// T - number of terminals
// dist[N][N] - Adjacency matrix
// steiner_tree() = min cost to connect first t nodes, 1-indexed
// dp[i][bit_mask] = min cost to connect nodes active in bitmask rooting in i
// min{dp[i][bit_mask]}, i <= n if root doesn't matter

int n, t, dp[N][(1 << T)], dist[N][N];

int steiner_tree() {
  for (int k = 1; k <= n; ++k)
    for (int i = 1; i <= n; ++i)
      for (int j = 1; j <= n; ++j)
        dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);

  for(int i = 1; i <= n; i++)
    for(int j = 0; j < (1 << t); j++)
      dp[i][j] = INF;
  for(int i = 1; i <= t; i++) dp[i][1 << (i-1)] = 0;

  for(int msk = 0; msk < (1 << t); msk++) {
    for(int i = 1; i <= n; i++) {
      for(int ss = msk; ss > 0; ss = (ss - 1) & msk)
        dp[i][msk] = min(dp[i][msk], dp[i][ss] + dp[i][msk - ss]);

      if(dp[i][msk] != INF)
        for(int j = 1; j <= n; j++)
          dp[j][msk] = min(dp[j][msk], dp[i][msk] + dist[i][j]);
    }
  }

  int mn = INF;
  for(int i = 1; i <= n; i++) mn = min(mn, dp[i][(1 << t) - 1]);
  return mn;
}
```

3 Graphs

3.1 2-SAT Kosaraju

```
/*********************************************************************************
* 2-SAT (TELL WHETHER A SERIES OF STATEMENTS CAN OR CANNOT BE FEASIBLE AT THE    *
* SAME TIME)                                                                     *
* Time complexity: O(V+E)                                                        *
* Usage: n          -> number of variables, 1-indexed                           *
*        p = v(i)   -> picks the "true" state for variable i                    *
*        p = nv(i)  -> picks the "false" state for variable i, i.e. ~i          *
*        add(p, q)  -> add clause p => q (which also means ~q => ~p)            *
*        run2sat()  -> true if possible, false if impossible                    *
*        val[i]     -> tells if i has to be true or false for that solution     *
*********************************************************************************/

int n, vis[2*N], ord[2*N], ordn, cnt, cmp[2*N], val[N];
vector<int> adj[2*N], adjt[2*N];

// for a variable u with idx i
// u is 2*i and !u is 2*i+1
// (a v b) == !a -> b ^ !b -> a

int v(int x) { return 2*x; }
int nv(int x) { return 2*x+1; }

// add clause (a v b)
void add(int a, int b){
  adj[a^1].push_back(b);
  adj[b^1].push_back(a);
  adjt[b].push_back(a^1);
  adjt[a].push_back(b^1);
}

void dfs(int x){
  vis[x] = 1;
  for(auto v : adj[x]) if(!vis[v]) dfs(v);
  ord[ordn++] = x;
}

void dfst(int x){
  cmp[x] = cnt, vis[x] = 0;
  for(auto v : adjt[x]) if(vis[v]) dfst(v);
}

bool run2sat(){
  for(int i = 1; i <= n; i++) {
    if(!vis[v(i)]) dfs(v(i));
    if(!vis[nv(i)]) dfs(nv(i));
  }
  for(int i = ordn-1; i >= 0; i--)
    if(vis[ord[i]]) cnt++, dfst(ord[i]);
  for(int i = 1; i <= n; i ++){
    if(cmp[v(i)] == cmp[nv(i)]) return false;
    val[i] = cmp[v(i)] > cmp[nv(i)];
  }
  return true;
}

int main () {
    for (int i = 1; i <= n; i++) {
        if (val[i]); // i-th variable is true
        else         // i-th variable is false
}
```

3.2 2-SAT Tarjan

```
// 2-SAT - O(V+E)
// For each variable x, we create two nodes in the graph: u and !u
// If the variable has index i, the index of u and !u are: 2*i and 2*i+1
// Adds a statment u => v
void add(int u, int v){
  adj[u].pb(v);
  adj[v^1].pb(u^1);
}

//0-indexed variables; starts from var_0 and goes to var_n-1
for(int i = 0; i < n; i++){
  tarjan(2*i), tarjan(2*i + 1);
  //cmp is a tarjan variable that says the component from a certain node
  if(cmp[2*i] == cmp[2*i + 1]) //Invalid
  if(cmp[2*i] < cmp[2*i + 1]) //Var_i is true
  else //Var_i is false

  //its just a possible solution!
}
```

3.3 Shortest Path (Bellman-Ford)

```
/*********************************************************************************
* BELLMAN-FORD ALGORITHM (SHORTEST PATH TO A VERTEX - WITH NEGATIVE COST)        *
* Time complexity: O(VE)                                                         *
* Usage: dist[node]                                                              *
* Notation: m:           number of edges                                         *
*           n:           number of vertices                                      *
*           (a, b, w):   edge between a and b with weight w                      *
*           s:           starting node                                           *
*********************************************************************************/
const int N = 1e4+10; // Maximum number of nodes
vector<int> adj[N], adjw[N];
int dist[N], v, w;

memset(dist, 63, sizeof(dist));
dist[0] = 0;
for (int i = 0; i < n-1; ++i)
  for (int u = 0; u < n; ++u)
    for (int j = 0; j < adj[u].size(); ++j)
      v = adj[u][j], w = adjw[u][j],
      dist[v] = min(dist[v], dist[u]+w);
```

3.4 Block Cut

```
// Tarjan for Block Cut Tree (Node Biconnected Componentes) - O(n + m)
#define pb push_back
#include <bits/stdc++.h>
using namespace std;

const int N = 1e5+5;

// Regular Tarjan stuff
int n, num[N], low[N], cnt, ch[N], art[N];
vector<int> adj[N], st;

int lb[N]; // Last block that node is contained
int bn; // Number of blocks
vector<int> blc[N]; // List of nodes from block

void dfs(int u, int p) {
  num[u] = low[u] = ++cnt;
  ch[u] = adj[u].size();
  st.pb(u);

  if (adj[u].size() == 1) blc[++bn].pb(u);

  for(int v : adj[u]) {
    if (!num[v]) {
      dfs(v, u), low[u] = min(low[u], low[v]);
      if (low[v] == num[u]) {
        if (p != -1 or ch[u] > 1) art[u] = 1;
        blc[++bn].pb(u);
        while(blc[bn].back() != v)
          blc[bn].pb(st.back()), st.pop_back();
      }
    }
    else if (v != p) low[u] = min(low[u], num[v]), ch[v]--;
  }

  if (low[u] == num[u]) st.pop_back();
}

// Nodes from 1 .. n are blocks
// Nodes from n+1 .. 2*n are articulations
vector<int> bct[2*N]; // Adj list for Block Cut Tree

void build_tree() {
  for(int u=1; u<=n; ++u) for(int v : adj[u]) if (num[u] > num[v]) {
    if (lb[u] == lb[v] or blc[lb[u]][0] == v) /* edge u-v belongs to block lb[u] */;
    else { /* edge u-v belongs to block cut tree */;
```

```
      int x = (art[u] ? u + n : lb[u]), y = (art[v] ? v + n : lb[v]);
      bct[x].pb(y), bct[y].pb(x);
    }
  }
}

void tarjan() {
  for(int u=1; u<=n; ++u) if (!num[u]) dfs(u, -1);
  for(int b=1; b<=bn; ++b) for(int u : blc[b]) lb[u] = b;
  build_tree();
}
```

3.5 Articulation points and bridges

```
// Articulation points and Bridges O(V+E)
int par[N], art[N], low[N], num[N], ch[N], cnt;

void articulation(int u) {
  low[u] = num[u] = ++cnt;
  for (int v : adj[u]) {
    if (!num[v]) {
      par[v] = u; ch[u]++;
      articulation(v);
      if (low[v] >= num[u]) art[u] = 1;
      if (low[v] >  num[u]) { /* u-v bridge */ }
      low[u] = min(low[u], low[v]);
    }
    else if (v != par[u]) low[u] = min(low[u], num[v]);
  }
}

for (int i = 0; i < n; ++i) if (!num[i])
  articulation(i), art[i] = ch[i]>1;
```

3.6 Max Flow

```
// Dinic - O(V^2 * E)
// Bipartite graph or unit flow - O(sqrt(V) * E)
// Small flow - O(F * (V + E))
// USE INF = 1e9!

/*******************************************************************************
* DINIC (FIND MAX FLOW / BIPARTITE MATCHING)                                    *
* Time complexity: O(EV^2)                                                      *
* Usage: dinic()                                                                *
*        add_edge(from, to, capacity)                                           *
* Testcase:                                                                     *
* add_edge(src, 1, 1);   add_edge(1, snk, 1);   add_edge(2, 3, INF);            *
* add_edge(src, 2, 1);   add_edge(2, snk, 1);   add_edge(3, 4, INF);            *
* add_edge(src, 2, 1);   add_edge(3, snk, 1);                                   *
* add_edge(src, 2, 1);   add_edge(4, snk, 1);   => dinic() = 4                  *
*******************************************************************************/

#include <bits/stdc++.h>
using namespace std;

const int N = 1e5+1, INF = 1e9;
struct edge {int v, c, f;};

int n, src, snk, h[N], ptr[N];
vector<edge> edgs;
vector<int> g[N];

void add_edge (int u, int v, int c) {
  int k = edgs.size();
  edgs.push_back({v, c, 0});
  edgs.push_back({u, 0, 0});
  g[u].push_back(k);
  g[v].push_back(k+1);
}

void clear() {
    memset(h, 0, sizeof h);
    memset(ptr, 0, sizeof ptr);
    edgs.clear();
    for (int i = 0; i < N; i++) g[i].clear();
    src = 0;
    snk = N-1;
}

bool bfs() {
  memset(h, 0, sizeof h);
  queue<int> q;
  h[src] = 1;
  q.push(src);
  while(!q.empty()) {
    int u = q.front(); q.pop();
    for(int i : g[u]) {
      int v = edgs[i].v;
      if (!h[v] and edgs[i].f < edgs[i].c)
        q.push(v), h[v] = h[u] + 1;
    }
  }
  return h[snk];
}

int dfs (int u, int flow) {
  if (!flow or u == snk) return flow;
  for (int &i = ptr[u]; i < g[u].size(); ++i) {
    edge &dir = edgs[g[u][i]], &rev = edgs[g[u][i]^1];
    int v = dir.v;
    if (h[v] != h[u] + 1)  continue;
    int inc = min(flow, dir.c - dir.f);
    inc = dfs(v, inc);
    if (inc) {
      dir.f += inc, rev.f -= inc;
      return inc;
    }
  }
  return 0;
}

int dinic() {
  int flow = 0;
  while (bfs()) {
    memset(ptr, 0, sizeof ptr);
    while (int inc = dfs(src, INF)) flow += inc;
  }
  return flow;
}

int main () {
    clear();
    return 0;
}
```

3.7 Erdos Gallai

```
// Erdos-Gallai - O(nlogn)
// check if it's possible to create a simple graph (undirected edges) from
// a sequence of vertice's degrees
bool gallai(vector<int> v) {
  vector<ll> sum;
  sum.resize(v.size());

  sort(v.begin(), v.end(), greater<int>());
  sum[0] = v[0];
  for (int i = 1; i < v.size(); i++) sum[i] = sum[i-1] + v[i];
  if (sum.back() % 2) return 0;

  for (int k = 1; k < v.size(); k++) {
    int p = lower_bound(v.begin(), v.end(), k, greater<int>()) - v.begin();
    if (p < k) p = k;
    if (sum[k-1] > 1ll*k*(p-1) + sum.back() - sum[p-1]) return 0;
  }
  return 1;
}
```

3.8 Eulerian Path

```
vector<int> ans, adj[N];
int in[N];

void dfs(int v){
  while(adj[v].size()){
    int x = adj[v].back();
    adj[v].pop_back();
    dfs(x);
  }
  ans.pb(v);
```

```
}

// Verify if there is an eulerian path or circuit
vector<int> v;
for(int i = 0; i < n; i++) if(adj[i].size() != in[i]){
  if(abs((int)adj[i].size() - in[i]) != 1) //-> There is no valid eulerian circuit/path
  v.pb(i);
}

if(v.size()){
  if(v.size() != 2) //-> There is no valid eulerian path
  if(in[v[0]] > adj[v[0]].size()) swap(v[0], v[1]);
  if(in[v[0]] > adj[v[0]].size()) //-> There is no valid eulerian path
  adj[v[1]].pb(v[0]); // Turn the eulerian path into a eulerian circuit
}

dfs(0);
for(int i = 0; i < cnt; i++)
  if(adj[i].size()) //-> There is no valid eulerian circuit/path in this case because the graph is not
       conected

ans.pop_back(); // Since it's a curcuit, the first and the last are repeated
reverse(ans.begin(), ans.end());

int bg = 0; // Is used to mark where the eulerian path begins
if(v.size()){
  for(int i = 0; i < ans.size(); i++)
    if(ans[i] == v[1] and ans[(i + 1)%ans.size()] == v[0]){
      bg = i + 1;
      break;
    }
}
```

3.9 Fast Kuhn

```
const int N = 1e5+5;

int x, marcB[N], matchB[N], matchA[N], ans, n, m, p;
vector<int> adj[N];

bool dfs(int v){
    for(int i = 0; i < adj[v].size(); i++){
        int viz = adj[v][i];
        if(marcB[viz] == 1 ) continue;
        marcB[viz] = 1;

        if((matchB[viz] == -1) || dfs(matchB[viz])){
            matchB[viz] = v;
            matchA[v] = viz;
            return true;
        }

    }
    return false;
}

int main(){
    //...
    for(int i = 0; i<=n; i++) matchA[i] = -1;
    for(int j = 0; j<=m; j++) matchB[j] = -1;

    bool aux = true;
    while(aux){
        for(int j=1; j<=m; j++) marcB[j] = 0;
        aux = false;
        for(int i=1; i<=n; i++){
            if(matchA[i] != -1) continue;
            if(dfs(i)){
                ans++;
                aux = true;
            }
        }
    }
    //...
}
```

3.10 Floyd Warshall

```
/*******************************************************************************
* FLOYD-WARSHALL ALGORITHM (SHORTEST PATH TO ANY VERTEX)                       *
* Time complexity: O(V^3)                                                      *
* Usage: dist[from][to]                                                        *
* Notation: m:            number of edges                                      *
*           n:            number of vertices                                   *
*           (a, b, w):    edge between a and b with weight w                   *
*******************************************************************************/

int adj[N][N]; // no-edge = INF

for (int k = 0; k < n; ++k)
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      adj[i][j] = min(adj[i][j], adj[i][k]+adj[k][j]);
```

3.11 Hungarian

```
// Hungarian - O(m*n^2)
// Assignment Problem

int n, m;
int pu[N], pv[N], cost[N][M];
int pairV[N], way[M], minv[M], used[M];

void hungarian() {
  for(int i = 1, j0 = 0; i <= n; i++) {
    pairV[0] = i;
    memset(minv, 63, sizeof minv);
    memset(used, 0, sizeof used);
    do {
      used[j0] = 1;
      int i0 = pairV[j0], delta = INF, j1;
      for(int j = 1; j <= m; j++) {
        if(used[j]) continue;
        int cur = cost[i0][j] - pu[i0] - pv[j];
        if(cur < minv[j]) minv[j] = cur, way[j] = j0;
        if(minv[j] < delta) delta = minv[j], j1 = j;
      }

      for(int j = 0; j <= m; j++) {
        if(used[j]) pu[pairV[j]] += delta, pv[j] -= delta;
        else minv[j] -= delta;
      }
      j0 = j1;
    } while(pairV[j0]);

    do {
      int j1 = way[j0];
      pairV[j0] = pairV[j1];
      j0 = j1;
    } while(j0);
  }
}

// in main
// for(int j = 1; j <= m; j++)
//  if(pairV[j]) ans += cost[pairV[j]][j];
//
```

3.12 Hungarian Navarro

```
// Hungarian - O(n^2 * m)
template<bool is_max = false, class T = int, bool is_zero_indexed = false>
struct Hungarian {
  bool swap_coord = false;
  int lines, cols;
  T ans;

  vector<int> pairV, way;
  vector<bool> used;
  vector<T> pu, pv, minv;
  vector<vector<T>> cost;

  Hungarian(int _n, int _m) {
    if (_n > _m) {
      swap(_n, _m);
      swap_coord = true;
    }

    lines = _n + 1, cols = _m + 1;
```

```
    clear();
    cost.resize(lines);
    for (auto& line : cost) line.assign(cols, 0);
  }

  void clear() {
    pairV.assign(cols, 0);
    way.assign(cols, 0);
    pv.assign(cols, 0);
    pu.assign(lines, 0);
  }

  void update(int i, int j, T val) {
    if (is_zero_indexed) i++, j++;
    if (is_max) val = -val;
    if (swap_coord) swap(i, j);

    assert(i < lines);
    assert(j < cols);

    cost[i][j] = val;
  }

  T run() {
    T _INF = numeric_limits<T>::max();
    for (int i = 1, j0 = 0; i < lines; i++) {
      pairV[0] = i;
      minv.assign(cols, _INF);
      used.assign(cols, 0);
      do {
        used[j0] = 1;
        int i0 = pairV[j0], j1;
        T delta = _INF;
        for (int j = 1; j < cols; j++) {
          if (used[j]) continue;
          T cur = cost[i0][j] - pu[i0] - pv[j];
          if (cur < minv[j]) minv[j] = cur, way[j] = j0;
          if (minv[j] < delta) delta = minv[j], j1 = j;
        }

        for (int j = 0; j < cols; j++) {
          if (used[j]) pu[pairV[j]] += delta, pv[j] -= delta;
          else minv[j] -= delta;
        }
        j0 = j1;
      } while (pairV[j0]);

      do {
        int j1 = way[j0];
        pairV[j0] = pairV[j1];
        j0 = j1;
      } while (j0);
    }

    ans = 0;
    for (int j = 1; j < cols; j++) if (pairV[j]) ans += cost[pairV[j]][j];

    if (is_max) ans = -ans;
    if (is_zero_indexed) {
      for (int j = 0; j + 1 < cols; j++) pairV[j] = pairV[j + 1], pairV[j]--;
      pairV[cols - 1] = -1;
    }
    if (swap_coord) {
      vector<int> pairV_sub(lines, 0);
      for (int j = 0; j < cols; j++) if (pairV[j] >= 0) pairV_sub[pairV[j]] = j;
      swap(pairV, pairV_sub);
    }

    return ans;
  }
};

template <bool is_max = false, bool is_zero_indexed = false>
struct HungarianMult : public Hungarian<is_max, long double, is_zero_indexed> {
  using super = Hungarian<is_max, long double, is_zero_indexed>;

  HungarianMult(int _n, int _m) : super(_n, _m) {}

  void update(int i, int j, long double x) {
    super::update(i, j, log2(x));
  }
};
```

3.13 Toposort

```
/*******************************************************************************
 * KAHN'S ALGORITHM (TOPOLOGICAL SORTING)                                      *
                                                                    *
 * Time complexity: O(V+E)                                                     *
 * Notation: adj[i]:  adjacency matrix for node i                              *
 *           n:       number of vertices                                       *
 *           e:       number of edges                                          *
 *           a, b:    edge between a and b                                     *
 *           inc:     number of incoming arcs/edges                            *
 *           q:       queue with the independent vertices                      *
 *           tsort:   final topo sort, i.e. possible order to traverse graph   *
 *******************************************************************************/


vector <int> adj[N];
int inc[N]; // number of incoming arcs/edges

// undirected graph: inc[v] <= 1
// directed graph:   inc[v] == 0

queue<int> q;
for (int i = 1; i <= n; ++i) if (inc[i] <= 1) q.push(i);

while (!q.empty()) {
  int u = q.front(); q.pop();
  for (int v : adj[u])
    if (inc[v] > 1 and --inc[v] <= 1)
      q.push(v);
}
```

3.14 Strongly Connected Components

```
// Kosaraju - SCC O(V+E)
// For undirected graph uncomment lines below

vi adj[N], adjt[N];
int n, ordn, cnt, vis[N], ord[N], cmp[N];
//int par[N];

void dfs(int u) {
  vis[u] = 1;
  for (auto v : adj[u]) if (!vis[v]) dfs(v);
  // for (auto v : adj[u]) if(!vis[v]) par[v] = u, dfs(v);
  ord[ordn++] = u;
}

void dfst(int u) {
  cmp[u] = cnt, vis[u] = 0;
  for (auto v : adjt[u]) if (vis[v]) dfst(v);
  // for (auto v : adj[u]) if(vis[v] and u != par[v]) dfst(v);
}

// in main
for (int i = 1; i <= n; ++i) if (!vis[i]) dfs(i);
for (int i = ordn-1; i >= 0; --i) if (vis[ord[i]]) cnt++, dfst(ord[i]);
```

3.15 MST (Kruskal)

```
/*******************************************************************************
 * KRUSKAL'S ALGORITHM (MINIMAL SPANNING TREE - INCREASING EDGE SIZE)          *
 * Time complexity: O(ElogE)                                                   *
 * Usage: cost, sz[find(node)]                                                 *
 * Notation: cost: sum of all edges which belong to such MST                   *
 *           sz:   vector of subsets sizes, i.e. size of the subset a node is in *
 *******************************************************************************/

// + Union-find

int cost = 0;
vector <pair<int, pair<int, int>>> edges; //mp(dist, mp(node1, node2))

int main () {
    // ...
    sort(edges.begin(), edges.end());
    for (auto e : edges)
        if (find(e.nd.st) != find(e.nd.nd))
            unite(e.nd.st, e.nd.nd), cost += e.st;

    return 0;
}
```

3.16 Max Bipartite Cardinality Matching (Kuhn)

```
/*********************************************************************************
* KUHN'S ALGORITHM (FIND GREATEST NUMBER OF MATCHINGS - BIPARTITE GRAPH)          *
* Time complexity: O(VE)                                                          *
* Notation: ans:      number of matchings                                         *
*           b[j]:     matching edge b[j] <-> j                                    *
*           adj[i]:   adjacency list for node i                                   *
*           vis:      visited nodes                                               *
*           x:        counter to help reuse vis list                              *
*********************************************************************************/

// TIP: If too slow, shuffle nodes and try again.
int x, vis[N], b[N], ans;

bool match(int u) {
  if (vis[u] == x) return 0;
  vis[u] = x;
  for (int v : adj[u])
    if (!b[v] or match(b[v])) return b[v]=u;
  return 0;
}

for (int i = 1; i <= n; ++i) ++x, ans += match(i);

// Maximum Independent Set on bipartite graph
MIS + MCBM = V

// Minimum Vertex Cover on bipartite graph
MVC = MCBM
```

3.17 Lowest Common Ancestor

```
// Lowest Common Ancestor <O(nlogn), O(logn)>
const int N = 1e6, M = 25;
int anc[M][N], h[N], rt;

// TODO: Calculate h[u] and set anc[0][u] = parent of node u for each u

// build (sparse table)
anc[0][rt] = rt; // set parent of the root to itself
for (int i = 1; i < M; ++i)
  for (int j = 1; j <= n; ++j)
    anc[i][j] = anc[i-1][anc[i-1][j]];

// query
int lca(int u, int v) {
  if (h[u] < h[v]) swap(u, v);
  for (int i = M-1; i >= 0; --i) if (h[u]-(1<<i) >= h[v])
    u = anc[i][u];

  if (u == v) return u;

  for (int i = M-1; i >= 0; --i) if (anc[i][u] != anc[i][v])
    u = anc[i][u], v = anc[i][v];
  return anc[0][u];
}
```

3.18 Max Weight on Path

```
// Using LCA to find max edge weight between (u, v)

const int N = 1e5+5;  // Max number of vertices
const int K = 20;     // Each 1e3 requires ~ 10 K
const int M = K+5;
int n;                // Number of vertices
vector <pii> adj[N];
int vis[N], h[N], anc[N][M], mx[N][M];

void dfs (int u) {
    vis[u] = 1;
    for (auto p : adj[u]) {
        int v = p.st;
        int w = p.nd;
        if (!vis[v]) {
            h[v] = h[u]+1;
            anc[v][0] = u;
            mx[v][0] = w;
            dfs(v);
        }
    }
}

void build () {
    // cl(mn, 63) -- Don't forget to initialize with INF if min edge!
    anc[1][0] = 1;
    dfs(1);
    for (int j = 1; j <= K; j++) for (int i = 1; i <= n; i++) {
        anc[i][j] = anc[anc[i][j-1]][j-1];
        mx[i][j] = max(mx[i][j-1], mx[anc[i][j-1]][j-1]);
    }
}

int mxedge (int u, int v) {
    int ans = 0;

    if (h[u] < h[v]) swap(u, v);
    for (int j = K; j >= 0; j--) if (h[anc[u][j]] >= h[v]) {
        ans = max(ans, mx[u][j]);
        u = anc[u][j];
    }
    if (u == v) return ans;
    for (int j = K; j >= 0; j--) if (anc[u][j] != anc[v][j]) {
        ans = max(ans, mx[u][j]);
        ans = max(ans, mx[v][j]);
        u = anc[u][j];
        v = anc[v][j];
    }
    return max({ans, mx[u][0], mx[v][0]});
}
```

3.19 Min Cost Max Flow

```
// USE INF = 1e9!

/*********************************************************************************
* MIN COST MAX FLOW (MINIMUM COST TO ACHIEVE MAXIMUM FLOW)                        *
* Description: Given a graph which represents a flow network where every edge has *
* a capacity and a cost per unit, find the minimum cost to establish the maximum  *
* possible flow from s to t.                                                      *
* Note: When adding edge (a, b), it is a directed edge!                           *
* Usage: min_cost_max_flow()                                                      *
*          add_edge(from, to, cost, capacity)                                     *
* Notation: flw: max flow                                                         *
*             cst: min cost to achieve flw                                        *
* Testcase:                                                                       *
* add_edge(src, 1, 0, 1);   add_edge(1, snk, 0, 1);   add_edge(2, 3, 1, INF);     *
* add_edge(src, 2, 0, 1);   add_edge(2, snk, 0, 1);   add_edge(3, 4, 1, INF);     *
* add_edge(src, 2, 0, 1);   add_edge(3, snk, 0, 1);                               *
* add_edge(src, 2, 0, 1);   add_edge(4, snk, 0, 1);   => flw = 4, cst = 3         *
*********************************************************************************/

// w: weight or cost, c : capacity
struct edge {int v, f, w, c; };

int n, flw_lmt=INF, src, snk, flw, cst, p[N], d[N], et[N];
vector<edge> e;
vector<int> g[N];

void add_edge(int u, int v, int w, int c) {
  int k = e.size();
  g[u].push_back(k);
  g[v].push_back(k+1);
  e.push_back({ v, 0,  w, c });
  e.push_back({ u, 0, -w, 0 });
}

void clear() {
  flw_lmt = INF;
  for(int i=0; i<=n; ++i) g[i].clear();
  e.clear();
}

void min_cost_max_flow() {
  flw = 0, cst = 0;
  while (flw < flw_lmt) {
    memset(et, 0, (n+1) * sizeof(int));
    memset(d, 63, (n+1) * sizeof(int));
    deque<int> q;
    q.push_back(src), d[src] = 0;
```

```
    while (!q.empty()) {
      int u = q.front(); q.pop_front();
      et[u] = 2;

      for(int i : g[u]) {
        edge &dir = e[i];
        int v = dir.v;
        if (dir.f < dir.c and d[u] + dir.w < d[v]) {
          d[v] = d[u] + dir.w;
          if (et[v] == 0) q.push_back(v);
          else if (et[v] == 2) q.push_front(v);
          et[v] = 1;
          p[v] = i;
        }
      }
    }

    if (d[snk] > INF) break;

    int inc = flw_lmt - flw;
    for (int u=snk; u != src; u = e[p[u]^1].v) {
      edge &dir = e[p[u]];
      inc = min(inc, dir.c - dir.f);
    }

    for (int u=snk; u != src; u = e[p[u]^1].v) {
      edge &dir = e[p[u]], &rev = e[p[u]^1];
      dir.f += inc;
      rev.f -= inc;
      cst += inc * dir.w;
    }

    if (!inc) break;
    flw += inc;
  }
}
```

3.20 MST (Prim)

```
// Prim - MST O(ElogE)
vi adj[N], adjw[N];
int vis[N];

priority_queue<pii> pq;
pq.push(mp(0, 0));

while (!pq.empty()) {
  int u = pq.top().nd;
  pq.pop();
  if (vis[u]) continue;
  vis[u]=1;
  for (int i = 0; i < adj[u].size(); ++i) {
    int v = adj[u][i];
    int w = adjw[u][i];
    if (!vis[v]) pq.push(mp(-w, v));
  }
}
```

3.21 Shortest Path (SPFA)

```
// Shortest Path Faster Algoritm O(VE)
int dist[N], inq[N];

cl(dist,63);
queue<int> q;
q.push(0); dist[0] = 0; inq[0] = 1;

while (!q.empty()) {
  int u = q.front(); q.pop(); inq[u]=0;
  for (int i = 0; i < adj[u].size(); ++i) {
    int v = adj[u][i], w = adjw[u][i];
    if (dist[v] > dist[u] + w) {
      dist[v] = dist[u] + w;
      if (!inq[v]) q.push(v), inq[v] = 1;
    }
  }
}
```

3.22 Stoer Wagner (Stanford)

```
// a is a N*N matrix storing the graph we use; a[i][j]=a[j][i]
memset(use,0,sizeof(use));
ans=maxlongint;
for (int i=1;i<N;i++)
{
    memcpy(visit,use,505*sizeof(int));
    memset(reach,0,sizeof(reach));
    memset(last,0,sizeof(last));
    t=0;
    for (int j=1;j<=N;j++)
        if (use[j]==0) {t=j;break;}
    for (int j=1;j<=N;j++)
        if (use[j]==0) reach[j]=a[t][j],last[j]=t;
    visit[t]=1;
    for (int j=1;j<=N-i;j++)
    {
        maxc=maxk=0;
        for (int k=1;k<=N;k++)
            if ((visit[k]==0)&&(reach[k]>maxc)) maxc=reach[k],maxk=k;
        c2=maxk,visit[maxk]=1;
        for (int k=1;k<=N;k++)
            if (visit[k]==0) reach[k]+=a[maxk][k],last[k]=maxk;
    }
    c1=last[c2];
    sum=0;
    for (int j=1;j<=N;j++)
        if (use[j]==0) sum+=a[j][c2];
    ans=min(ans,sum);
    use[c2]=1;
    for (int j=1;j<=N;j++)
        if ((c1!=j)&&(use[j]==0)) {a[j][c1]+=a[j][c2];a[c1][j]=a[j][c1];}
}
```

3.23 Tarjan

```
// Tarjan for SCC and Edge Biconnected Componentes - O(n + m)
vector<int> adj[N];
stack<int> st;
bool inSt[N];

int id[N], cmp[N];
int cnt, cmpCnt;

void clear(){
  memset(id, 0, sizeof id);
  cnt = cmpCnt = 0;
}

int tarjan(int n){
  int low;
  id[n] = low = ++cnt;
  st.push(n), inSt[n] = true;

  for(auto x : adj[n]){
    if(id[x] and inSt[x]) low = min(low, id[x]);
    else if(!id[x]) {
      int lowx = tarjan(x);
      if(inSt[x])
        low = min(low, lowx);
    }
  }

  if(low == id[n]){
    while(st.size()){
      int x = st.top();
      inSt[x] = false;
      cmp[x] = cmpCnt;

      st.pop();
      if(x == n) break;
    }
    cmpCnt++;
  }
  return low;
}
```

3.24 Zero One BFS

```cpp
// 0-1 BFS - O(V+E)

const int N = 1e5 + 5;

int dist[N];
vector<pii> adj[N];
deque<pii> dq;

void zero_one_bfs (int x){
    cl(dist, 63);
    dist[x] = 0;
    dq.push_back({x, 0});
    while(!dq.empty()){
        int u = dq.front().st;
        int ud = dq.front().nd;
        dq.pop_front();
        if(dist[u] < ud) continue;
        for(auto x : adj[u]){
            int v = x.st;
            int w = x.nd;
            if(dist[u] + w < dist[v]){
                dist[v] = dist[u] + w;
                if(w) dq.push_back({v, dist[v]});
                else dq.push_front({v, dist[v]});
            }
        }
    }
}
```

4 Mathematics

4.1 Basics

```cpp
// Greatest Common Divisor & Lowest Common Multiple
ll gcd(ll a, ll b) { return b ? gcd(b, a%b) : a; }
ll lcm(ll a, ll b) { return a/gcd(a, b)*b; }

// Multiply caring overflow
ll mulmod(ll a, ll b, ll m = MOD) {
  ll r=0;
  for (a %= m; b; b>>=1, a=(a*2)%m) if (b&1) r=(r+a)%m;
  return r;
}

// Another option for mulmod is using long double
ull mulmod(ull a, ull b, ull m = MOD) {
  ull q = (ld) a * (ld) b / (ld) m;
  ull r = a * b - q * m;
  return (r + m) % m;
}

// Fast exponential
ll fexp(ll a, ll b, ll m = MOD) {
  ll r=1;
  for (a %= m; b; b>>=1, a=(a*a)%m) if (b&1) r=(r*a)%m;
  return r;
}
```

4.2 Advanced

```cpp
/* Multiplicative Inverse over MOD for all 1..N - 1 < MOD in O(N)
 Only works for prime MOD. If all 1..MOD - 1 needed, use N = MOD */
ll inv[N];
inv[1] = 1;
for(int i = 2; i < N; ++i)
        inv[i] = MOD - (MOD / i) * inv[MOD % i] % MOD;

/* Catalan
 f(n) = sum(f(i) * f(n - i - 1)), i in [0, n - 1] = (2n)! / ((n+1)! * n!) = ...
 If you have any function f(n) (there are many) that follows this sequence (0-indexed):
 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440
 than it's the Catalan function */
ll cat[N];
cat[0] = 1;
for(int i = 1; i + 1 < N; i++) // needs inv[i + 1] till inv[N - 1]
        cat[i] = 2ll * (2ll * i - 1) * inv[i + 1] % MOD * cat[i - 1] % MOD;

/* Floor(n / i), i = [1, n], has <= 2 * sqrt(n) diff values.
 Proof: i = [1, sqrt(n)] has sqrt(n) diff values.
 For i = [sqrt(n), n] we have that 1 <= n / i <= sqrt(n)
 and thus has <= sqrt(n) diff values.
*/
/* l = first number that has floor(N / l) = x
 r = last number that has floor(N / r) = x
 N / r >= floor(N / l)
 r <= N / floor(N / l)*/
for(int l = 1, r; l <= n; l = r + 1){
        r = n / (n / l);
        // floor(n / i) has the same value for l <= i <= r
}

/* Recurrence using matriz
 h[i + 2] = a1 * h[i + 1] + a0 * h[i]
 [h[i] h[i-1]] = [h[1] h[0]] * [a1 1] ^ (i - 1)
                               [a0 0]          */

/* Fibonacci in O(log(N)) with memoization
 f(0) = f(1) = 1
 f(2*k) = f(k)^2 + f(k - 1)^2
 f(2*k + 1) = f(k)*[f(k) + 2*f(k - 1)] */

/* Wilson's Theorem Extension
 B = b1 * b2 * ... * bm (mod n) = +-1, all bi <= n such that gcd(bi, n) = 1
 if(n <= 4 or n = (odd prime)^k or n = 2 * (odd prime)^k) B = -1; for any k
 else B = 1; */

/* Stirling numbers of the second kind
 S(n, k) = Number of ways to split n numbers into k non-empty sets
 S(n, 1) = S(n, n) = 1
 S(n, k) = k * S(n - 1, k) + S(n - 1, k - 1)
 Sr(n, k) = S(n, k) with at least r numbers in each set
 Sr(n, k) = k * Sr(n - 1, k) + (n - 1) * Sr(n - r, k - 1)
                               (r - 1)
 S(n - d + 1, k - d + 1) = S(n, k) where if indexes i, j belong to the same set, then |i - j| >= d */

/* Burnside's Lemma
 |Classes| = 1 / |G| * sum(K ^ C(g)) for each g in G
 G = Different permutations possible
 C(g) = Number of cycles on the permutation g
 K = Number of states for each element

 Different ways to paint a necklace with N beads and K colors:
 G = {(1, 2, ... N), (2, 3, ... N, 1), ... (N, 1, ... N - 1)}
 gi = (i, i + 1, ... i + N), (taking mod N to get it right) i = 1 ... N
 i -> 2i -> 3i ..., Cycles in gi all have size n / gcd(i, n), so C(gi) = gcd(i, n)
 Ans = 1 / N * sum(K ^ gcd(i, n)), i = 1 ... N
 (For the brave, you can get to Ans = 1 / N * sum(euler_phi(N / d) * K ^ d), d | N) */

/* Mobius Inversion
 Sum of gcd(i, j), 1 <= i, j <= N?
 sum(k->N) k * sum(i->N) sum(j->N) [gcd(i, j) == k], i = a * k, j = b * k
 = sum(k->N) k * sum(a->N/k) sum(b->N/k) [gcd(a, b) == 1]
 = sum(k->N) k * sum(a->N/k) sum(b->N/k) sum(d->N/k) [d | a] * [d | b] * mi(d)
 = sum(k->N) k * sum(d->N/k) mi(d) * floor(N / kd)^2, l = kd, l <= N, k | l, d = l / k
 = sum(l->N) floor(N / l)^2 * sum(k|l) k * mi(l / k)
 If f(n) = sum(x|n)(g(x) * h(x)) with g(x) and h(x) multiplicative, than f(n) is multiplicative
 Hence, g(l) = sum(k|l) k * mi(l / k) is multiplicative
 = sum(l->N) floor(N / l)^2 * g(l) */
```

4.3 Discrete Log (Baby-step Giant-step)

```cpp
// O(sqrt(m))
// Solve c * a^x = b mod(m) for integer x >= 0.
// Return the smallest x possible, or -1 if there is no solution
// If all solutions needed, solve c * a^x = b mod(m) and (a*b) * a^y = b mod(m)
// x + k * (y + 1) for k >= 0 are all solutions
// Works for any integer values of c, a, b and positive m
ll discrete_log(ll c, ll a, ll b, ll m){
        c = ((c % m) + m) % m, a = ((a % m) + m) % m, b = ((b % m) + m) % m;
        if(c == b)
                return 0;

        ll g = __gcd(a, m);
        if(b % g) return -1;

        if(g > 1){
```

```
                ll r = discrete_log(c * a / g, a, b / g, m / g);
                return r + (r >= 0);
        }

        unordered_map<ll, ll> babystep;
        ll n = 1, an = a % m;

        // set n to the ceil of sqrt(m):
        while(n * n < m) n++, an = (an * a) % m;

        // babysteps:
        ll bstep = b;
        for(ll i = 0; i <= n; i++){
                babystep[bstep] = i;
                bstep = (bstep * a) % m;
        }

        // giantsteps:
        ll gstep = c * an % m;
        for(ll i = 1; i <= n; i++){
                if(babystep.find(gstep) != babystep.end())
                        return n * i - babystep[gstep];
                gstep = (gstep * an) % m;
        }
        return -1;
}
```

4.4 Euler Phi

```
// Euler phi (totient)
int ind = 0, pf = primes[0], ans = n;
while (1ll*pf*pf <= n) {
  if (n%pf==0) ans -= ans/pf;
  while (n%pf==0) n /= pf;
  pf = primes[++ind];
}
if (n != 1) ans -= ans/n;

// IME2014
int phi[N];
void totient() {
  for (int i = 1; i < N; ++i)  phi[i]=i;
  for (int i = 2; i < N; i+=2) phi[i]>>=1;
  for (int j = 3; j < N; j+=2) if (phi[j]==j) {
    phi[j]--;
    for (int i = 2*j; i < N; i+=j) phi[i]=phi[i]/j*(j-1);
  }
}
```

4.5 Extended Euclidean and Chinese Remainder

```
// Extended Euclid:
void euclid(ll a, ll b, ll &x, ll &y) {
  if (b) euclid(b, a%b, y, x), y -= x*(a/b);
  else x = 1, y = 0;
}

// find (x, y) such that a*x + b*y = c or return false if it's not possible
// [x + k*b/gcd(a, b), y - k*a/gcd(a, b)] are also solutions
bool diof(ll a, ll b, ll c, ll &x, ll &y){
  euclid(abs(a), abs(b), x, y);
  ll g = abs(__gcd(a, b));
  if(c % g) return false;
  x *= c / g;
  y *= c / g;
  if(a < 0) x = -x;
  if(b < 0) y = -y;
  return true;
}

// auxiliar to find_all_solutions
void shift_solution (ll &x, ll &y, ll a, ll b, ll cnt) {
  x += cnt * b;
  y -= cnt * a;
}

// Find the amount of solutions of
// ax + by = c
// in given intervals for x and y
ll find_all_solutions (ll a, ll b, ll c, ll minx, ll maxx, ll miny, ll maxy) {
  ll x, y, g = __gcd(a, b);
  if(!diof(a, b, c, x, y)) return 0;
  a /= g; b /= g;

  int sign_a = a>0 ? +1 : -1;
  int sign_b = b>0 ? +1 : -1;

  shift_solution (x, y, a, b, (minx - x) / b);
  if (x < minx)
    shift_solution (x, y, a, b, sign_b);
  if (x > maxx)
    return 0;
  int lx1 = x;

  shift_solution (x, y, a, b, (maxx - x) / b);
  if (x > maxx)
    shift_solution (x, y, a, b, -sign_b);
  int rx1 = x;

  shift_solution (x, y, a, b, - (miny - y) / a);
  if (y < miny)
    shift_solution (x, y, a, b, -sign_a);
  if (y > maxy)
    return 0;
  int lx2 = x;

  shift_solution (x, y, a, b, - (maxy - y) / a);
  if (y > maxy)
    shift_solution (x, y, a, b, sign_a);
  int rx2 = x;

  if (lx2 > rx2)
    swap (lx2, rx2);
  int lx = max (lx1, lx2);
  int rx = min (rx1, rx2);

  if (lx > rx) return 0;
  return (rx - lx) / abs(b) + 1;
}

bool crt_auxiliar(ll a, ll b, ll m1, ll m2, ll &ans){
  ll x, y;
  if(!diof(m1, m2, b - a, x, y)) return false;
  ll lcm = m1 / __gcd(m1, m2) * m2;
  ans = ((a + x % (lcm / m1) * m1) % lcm + lcm) % lcm;
  return true;
}

// find ans such that ans = a[i] mod b[i] for all 0 <= i < n or return false if not possible
// ans + k * lcm(b[i]) are also solutions
bool crt(int n, ll a[], ll b[], ll &ans){
  if(!b[0]) return false;
  ans = a[0] % b[0];
  ll l = b[0];
  for(int i = 1; i < n; i++){
   if(!b[i]) return false;
    if(!crt_auxiliar(ans, a[i] % b[i], l, b[i], ans)) return false;
    l *= (b[i] / __gcd(b[i], l));
  }
  return true;
}
```

4.6 Fast Fourier Transform(Tourist)

```
//
//  FFT made by tourist. It if faster and more supportive, although it requires more lines of code.
//  Also, it allows operations with MOD, which is usually an issue in FFT problems.
//
namespace fft {
  typedef double dbl;

  struct num {
    dbl x, y;
    num() { x = y = 0; }
    num(dbl x, dbl y) : x(x), y(y) {}
  };

  inline num operator+ (num a, num b) { return num(a.x + b.x, a.y + b.y); }
  inline num operator- (num a, num b) { return num(a.x - b.x, a.y - b.y); }
  inline num operator* (num a, num b) { return num(a.x * b.x - a.y * b.y, a.x * b.y + a.y * b.x); }
  inline num conj(num a) { return num(a.x, -a.y); }

  int base = 1;
  vector<num> roots = {{0, 0}, {1, 0}};
```

```
  vector<int> rev = {0, 1};

  const dbl PI = acosl(-1.0);

  void ensure_base(int nbase) {
    if(nbase <= base) return;

    rev.resize(1 << nbase);
    for(int i=0; i < (1 << nbase); i++) {
      rev[i] = (rev[i >> 1] >> 1) + ((i & 1) << (nbase - 1));
    }
    roots.resize(1 << nbase);

    while(base < nbase) {
      dbl angle = 2*PI / (1 << (base + 1));
      for(int i = 1 << (base - 1); i < (1 << base); i++) {
        roots[i << 1] = roots[i];
        dbl angle_i = angle * (2 * i + 1 - (1 << base));
        roots[(i << 1) + 1] = num(cos(angle_i), sin(angle_i));
      }
      base++;
    }
  }

  void fft(vector<num> &a, int n = -1) {
    if(n == -1) {
      n = a.size();
    }
    assert((n & (n-1)) == 0);
    int zeros = __builtin_ctz(n);
    ensure_base(zeros);
    int shift = base - zeros;
    for(int i = 0; i < n; i++) {
      if(i < (rev[i] >> shift)) {
        swap(a[i], a[rev[i] >> shift]);
      }
    }
    for(int k = 1; k < n; k <<= 1) {
      for(int i = 0; i < n; i += 2 * k) {
        for(int j = 0; j < k; j++) {
          num z = a[i+j+k] * roots[j+k];
          a[i+j+k] = a[i+j] - z;
          a[i+j] = a[i+j] + z;
        }
      }
    }
  }

  vector<num> fa, fb;
  vector<int> multiply(vector<int> &a, vector<int> &b) {
    int need = a.size() + b.size() - 1;
    int nbase = 0;
    while((1 << nbase) < need) nbase++;
    ensure_base(nbase);
    int sz = 1 << nbase;
    if(sz > (int) fa.size()) {
      fa.resize(sz);
    }
    for(int i = 0; i < sz; i++) {
      int x = (i < (int) a.size() ? a[i] : 0);
      int y = (i < (int) b.size() ? b[i] : 0);
      fa[i] = num(x, y);
    }
    fft(fa, sz);
    num r(0, -0.25 / sz);
    for(int i = 0; i <= (sz >> 1); i++) {
      int j = (sz - i) & (sz - 1);
      num z = (fa[j] * fa[j] - conj(fa[i] * fa[i])) * r;
      if(i != j) {
        fa[j] = (fa[i] * fa[i] - conj(fa[j] * fa[j])) * r;
      }
      fa[i] = z;
    }
    fft(fa, sz);
    vector<int> res(need);
    for(int i = 0; i < need; i++) {
      res[i] = fa[i].x + 0.5;
    }
    return res;
  }

  vector<int> multiply_mod(vector<int> &a, vector<int> &b, int m, int eq = 0) {
    int need = a.size() + b.size() - 1;
    int nbase = 0;
    while ((1 << nbase) < need) nbase++;
    ensure_base(nbase);
    int sz = 1 << nbase;
    if (sz > (int) fa.size()) {
      fa.resize(sz);
    }
    for (int i = 0; i < (int) a.size(); i++) {
      int x = (a[i] % m + m) % m;
      fa[i] = num(x & ((1 << 15) - 1), x >> 15);
    }
    fill(fa.begin() + a.size(), fa.begin() + sz, num {0, 0});
    fft(fa, sz);
    if (sz > (int) fb.size()) {
      fb.resize(sz);
    }
    if (eq) {
      copy(fa.begin(), fa.begin() + sz, fb.begin());
    } else {
      for (int i = 0; i < (int) b.size(); i++) {
        int x = (b[i] % m + m) % m;
        fb[i] = num(x & ((1 << 15) - 1), x >> 15);
      }
      fill(fb.begin() + b.size(), fb.begin() + sz, num {0, 0});
      fft(fb, sz);
    }
    dbl ratio = 0.25 / sz;
    num r2(0, -1);
    num r3(ratio, 0);
    num r4(0, -ratio);
    num r5(0, 1);
    for (int i = 0; i <= (sz >> 1); i++) {
      int j = (sz - i) & (sz - 1);
      num a1 = (fa[i] + conj(fa[j]));
      num a2 = (fa[i] - conj(fa[j])) * r2;
      num b1 = (fb[i] + conj(fb[j])) * r3;
      num b2 = (fb[i] - conj(fb[j])) * r4;
      if (i != j) {
        num c1 = (fa[j] + conj(fa[i]));
        num c2 = (fa[j] - conj(fa[i])) * r2;
        num d1 = (fb[j] + conj(fb[i])) * r3;
        num d2 = (fb[j] - conj(fb[i])) * r4;
        fa[i] = c1 * d1 + c2 * d2 * r5;
        fb[i] = c1 * d2 + c2 * d1;
      }
      fa[j] = a1 * b1 + a2 * b2 * r5;
      fb[j] = a1 * b2 + a2 * b1;
    }
    fft(fa, sz);
    fft(fb, sz);
    vector<int> res(need);
    for (int i = 0; i < need; i++) {
      long long aa = fa[i].x + 0.5;
      long long bb = fb[i].x + 0.5;
      long long cc = fa[i].y + 0.5;
      res[i] = (aa + ((bb % m) << 15) + ((cc % m) << 30)) % m;
    }
    return res;
  }

  vector<int> square_mod(vector<int> &a, int m) {
    return multiply_mod(a, a, m, 1);
  }
}
```

4.7 Fast Fourier Transform

```
// Fast Fourier Transform - O(nlogn)

/*
// Use struct instead. Performance will be way better!
typedef complex<ld> T;
T a[N], b[N];
*/

struct T {
  ld x, y;
  T() : x(0), y(0) {}
  T(ld a, ld b=0) : x(a), y(b) {}

  T operator/=(ld k) { x/=k; y/=k; return (*this); }
  T operator*(T a) const { return T(x*a.x - y*a.y, x*a.y + y*a.x); }
  T operator+(T a) const { return T(x+a.x, y+a.y); }
  T operator-(T a) const { return T(x-a.x, y-a.y); }
} a[N], b[N];

// a: vector containing polynomial
// n: power of two greater or equal product size
/*
// Use iterative version!
void fft_recursive(T* a, int n, int s) {
```

```
  if (n == 1) return;
  T tmp[n];
  for (int i = 0; i < n/2; ++i)
    tmp[i] = a[2*i], tmp[i+n/2] = a[2*i+1];

  fft_recursive(&tmp[0], n/2, s);
  fft_recursive(&tmp[n/2], n/2, s);

  T wn = T(cos(s*2*PI/n), sin(s*2*PI/n)), w(1,0);
  for (int i = 0; i < n/2; i++, w=w*wn)
    a[i] = tmp[i] + w*tmp[i+n/2],
    a[i+n/2] = tmp[i] - w*tmp[i+n/2];
}
*/

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

  for(int i = 1; (1<<i) <= n; i++){
    int M = 1 << i;
    int K = M >> 1;
    T wn = T(cos(s*2*PI/M), sin(s*2*PI/M));
    for(int j = 0; j < n; j += M) {
      T w = T(1, 0);
      for(int l = j; l < K + j; ++l){
        T t = w*a[l + K];
        a[l + K] = a[l]-t;
        a[l] = a[l] + t;
        w = wn*w;
      }
    }
  }
}

// assert n is a power of two greater of equal product size
// n = na + nb; while (n&(n-1)) n++;
void multiply(T* a, T* b, int n) {
  fft(a,n,1);
  fft(b,n,1);
  for (int i = 0; i < n; i++) a[i] = a[i]*b[i];
  fft(a,n,-1);
  for (int i = 0; i < n; i++) a[i] /= n;
}

// Convert to integers after multiplying:
// (int)(a[i].x + 0.5);
```

4.8 Fast Walsh-Hadamard Transform

```
// Fast Walsh-Hadamard Transform - O(nlogn)
//
// Multiply two polynomials, but instead of x^a * x^b = x^(a+b)
// we have x^a * x^b = x^(a XOR b).
//
// WARNING: assert n is a power of two!
void fwht(ll* a, int n, bool inv) {
  for(int l=1; 2*l <= n; l<<=1) {
    for(int i=0; i < n; i+=2*l) {
      for(int j=0; j<l; j++) {
        ll u = a[i+j], v = a[i+l+j];

        a[i+j] = (u+v) % MOD;
        a[i+l+j] = (u-v+MOD) % MOD;
        // % is kinda slow, you can use add() macro instead
        // #define add(x,y) (x+y >= MOD ? x+y-MOD : x+y)
      }
    }
  }

  if(inv) {
    for(int i=0; i<n; i++) {
      a[i] = a[i] / n;
    }
  }
}


/* FWHT AND
  Matrix : Inverse
  0 1      -1 1
  1 1       1 0
*/
void fwht_and(vi &a, bool inv) {
  vi ret = a;
  ll u, v;
  int tam = a.size() / 2;
  for(int len = 1; 2 * len <= tam; len <<= 1) {
    for(int i = 0; i < tam; i += 2 * len) {
      for(int j = 0; j < len; j++) {
        u = ret[i + j];
        v = ret[i + len + j];
        if(!inv) {
          ret[i + j] = v;
          ret[i + len + j] = u + v;
        }
        else {
          ret[i + j] = -u + v;
          ret[i + len + j] = u;
        }
      }
    }
  }
  a = ret;
}


/* FWHT OR
  Matrix : Inverse
  1 1      0  1
  1 0      1 -1
*/
void fft_or(vi &a, bool inv) {
  vi ret = a;
  ll u, v;
  int tam = a.size() / 2;
  for(int len = 1; 2 * len <= tam; len <<= 1) {
    for(int i = 0; i < tam; i += 2 * len) {
      for(int j = 0; j < len; j++) {
        u = ret[i + j];
        v = ret[i + len + j];
        if(!inv) {
          ret[i + j] = u + v;
          ret[i + len + j] = u;
        }
        else {
          ret[i + j] = v;
          ret[i + len + j] = u - v;
        }
      }
    }
  }
  a = ret;
}
```

4.9 Gaussian Elimination (extended inverse)

```
// Gauss-Jordan Elimination with Scaled Partial Pivoting
// Extended to Calculate Inverses - O(n^3)
// To get more precision choose m[j][i] as pivot the element such that m[j][i] / mx[j] is maximized.
// mx[j] is the element with biggest absolute value of row j.

ld C[N][M]; // N = 1000, M = 2*N+1;
int row, col;

bool elim() {
  for(int i=0; i<row; ++i) {
    int p = i; // Choose the biggest pivot
    for(int j=i; j<row; ++j) if (abs(C[j][i]) > abs(C[p][i])) p = j;
    for(int j=i; j<col; ++j) swap(C[i][j], C[p][j]);

    if (!C[i][i]) return 0;

    ld c = 1/C[i][i]; // Normalize pivot line
    for(int j=0; j<col; ++j) C[i][j] *= c;

    for(int k=i+1; k<col; ++k) {
      ld c = -C[k][i]; // Remove pivot variable from other lines
      for(int j=0; j<col; ++j) C[k][j] += c*C[i][j];
    }
  }

  // Make triangular system a diagonal one
  for(int i=row-1; i>=0; --i) for(int j=i-1; j>=0; --j) {
    ld c = -C[j][i];
    for(int k=i; k<col; ++k) C[j][k] += c*C[i][k];
  }
```

```
  return 1;
}

// Finds inv, the inverse of matrix m of size n x n.
// Returns true if procedure was successful.
bool inverse(int n, ld m[N][N], ld inv[N][N]) {
  for(int i=0; i<n; ++i) for(int j=0; j<n; ++j)
    C[i][j] = m[i][j], C[i][j+n] = (i == j);

  row = n, col = 2*n;
  bool ok = elim();

  for(int i=0; i<n; ++i) for(int j=0; j<n; ++j) inv[i][j] = C[i][j+n];
  return ok;
}

// Solves linear system m*x = y, of size n x n
bool linear_system(int n, ld m[N][N], ld *x, ld *y) {
  for(int i = 0; i < n; ++i) for(int j = 0; j < n; ++j) C[i][j] = m[i][j];
  for(int j = 0; j < n; ++j) C[j][n] = x[j];

  row = n, col = n+1;
  bool ok = elim();

  for(int j=0; j<n; ++j) y[j] = C[j][n];
  return ok;
}
```

4.10 Gaussian Elimination (modulo prime)

```
//ll A[N][M+1], X[M]

for(int j=0; j<m; j++) { //collumn to eliminate
  int l = j;
  for(int i=j+1; i<n; i++) //find nonzero pivot
    if(A[i][j]%p)
      l=i;
  for(int k = 0; k < m+1; k++) { //Swap lines
    swap(A[l][k],A[j][k]);
  }
  for(int i = j+1; i < n; i++) { //eliminate column
    ll t=mulmod(A[i][j],inv(A[j][j],p),p);
    for(int k = j; k < m+1; k++)
      A[i][k]=(A[i][k]-mulmod(t,A[j][k],p)+p)%p;
  }
}

for(int i = m-1; i >= 0; i--) { //solve triangular system
  for(int j = m-1; j > i; j--)
    A[i][m] = (A[i][m] - mulmod(A[i][j],X[j],p)+p)%p;
    X[i] = mulmod(A[i][m],inv(A[i][i],p),p);
}
```

4.11 Gaussian Elimination (xor)

```
// Gauss Elimination for xor boolean operations
// Return false if not possible to solve
// Use boolean matrixes 0-indexed
// n equations, m variables, O(n * m * m)
// eq[i][j] = coefficient of j-th element in i-th equation
// r[i] = result of i-th equation
// Return ans[j] = xj that gives the lexicographically greatest solution (if possible)
// (Can be changed to lexicographically least, follow the comments in the code)
// WARNING!! The arrays get changed during de algorithm

bool eq[N][M], r[N], ans[M];

bool gauss_xor(int n, int m){
        for(int i = 0; i < m; i++)
                ans[i] = true;
        int lid[N] = {0}; // id + 1 of last element present in i-th line of final matrix
        int l = 0;
        for(int i = m - 1; i >= 0; i--){
                for(int j = l; j < n; j++)
                        if(eq[j][i]){ // pivot
                                swap(eq[l], eq[j]);
                                swap(r[l], r[j]);
                        }
                if(l == n || !eq[l][i])
                        continue;
                lid[l] = i + 1;
                for(int j = l + 1; j < n; j++){ // eliminate column
                        if(!eq[j][i])
                                continue;
                        for(int k = 0; k <= i; k++)
                                eq[j][k] ^= eq[l][k];
                        r[j] ^= r[l];
                }
                l++;
        }
        for(int i = n - 1; i >= 0; i--){ // solve triangular matrix
                for(int j = 0; j < lid[i + 1]; j++)
                        r[i] ^= (eq[i][j] && ans[j]);
                // for lexicographically least just delete the for bellow
                for(int j = lid[i + 1]; j + 1 < lid[i]; j++){
                        ans[j] = true;
                        r[i] ^= eq[i][j];
                }
                if(lid[i])
                        ans[lid[i] - 1] = r[i];
                else if(r[i])
                        return false;
        }
        return true;
}
```

4.12 Gaussian Elimination (double)

```
//Gaussian Elimination
//double A[N][M+1], X[M]

// if n < m, there's no solution
// column m holds the right side of the equation
// X holds the solutions

for(int j=0; j<m; j++) { //collumn to eliminate
  int l = j;
  for(int i=j+1; i<n; i++) //find largest pivot
    if(abs(A[i][j])>abs(A[l][j]))
      l=i;
  if(abs(A[i][j]) < EPS) continue;
  for(int k = 0; k < m+1; k++) { //Swap lines
    swap(A[l][k],A[j][k]);
  }
  for(int i = j+1; i < n; i++) { //eliminate column
    double t=A[i][j]/A[j][j];
    for(int k = j; k < m+1; k++)
      A[i][k]-=t*A[j][k];
  }
}

for(int i = m-1; i >= 0; i--) { //solve triangular system
  for(int j = m-1; j > i; j--)
    A[i][m] -= A[i][j]*X[j];
  X[i]=A[i][m]/A[i][i];
}
```

4.13 Golden Section Search (Ternary Search)

```
double gss(double l, double r) {
    double m1 = r-(r-l)/gr, m2 = l+(r-l)/gr;
    double f1 = f(m1), f2 = f(m2);
    while(fabs(l-r)>EPS) {
        if(f1>f2) l=m1, f1=f2, m1=m2, m2=l+(r-l)/gr, f2=f(m2);
        else r=m2, f2=f1, m2=m1, m1=r-(r-l)/gr, f1=f(m1);
    }
    return l;
}
```

4.14 Josephus

```
// UFMG
/* Josephus Problem - It returns the position to be, in order to not die. O(n)*/
/* With k=2, for instance, the game begins with 2 being killed and then n+2, n+4, ... */
ll josephus(ll n, ll k) {
  if(n==1) return 1;
```

```
  else return (josephus(n-1, k)+k-1)%n+1;
}

/* Another Way to compute the last position to be killed - O(d * log n) */
ll josephus(ll n, ll d) {
  ll K = 1;
  while (K <= (d - 1)*n) K = (d * K + d - 2) / (d - 1);
  return d * n + 1 - K;
}
```

4.15 Mobius Inversion

```
// multiplicative function calculator
// euler_phi and mobius are multiplicative
// if another f[N] needed just remove comments
// O(N)

bool p[N];
vector<ll> primes;
ll g[N];
// ll f[N];

void mfc(){
    // if g(1) != 1 than it's not multiplicative
    g[1] = 1;
    // f[1] = 1;
    primes.clear();
    primes.reserve(N / 10);
    for(ll i = 2; i < N; i++){
        if(!p[i]){
            primes.push_back(i);
            for(ll j = i; j < N; j *= i){
                g[j] = // g(p^k) you found
                // f[j] = f(p^k) you found
                p[j] = (j != i);
            }
        }
        for(ll j : primes){
            if(i * j >= N || i % j == 0)
                break;
            for(ll k = j; i * k < N; k *= j){
                g[i * k] = g[i] * g[k];
                // f[i * k] = f[i] * f[k];
                p[i * k] = true;
            }
        }
    }
}
```

4.16 Mobius Function

```
// 1 if n == 1
// 0 if exists x | n%(x^2) == 0
// else (-1)^k, k = #(p) | p is prime and n%p == 0

//Calculate Mobius for all integers using sieve
//O(n*log(log(n)))
void mobius() {
  for(int i = 1; i < N; i++) mob[i] = 1;

  for(ll i = 2; i < N; i++) if(!sieve[i]){
    for(ll j = i; j < N; j += i) sieve[j] = i, mob[j] *= -1;
    for(ll j = i*i; j < N; j += i*i) mob[j] = 0;
  }
}

/*
//Calculate Mobius for 1 integer
//O(sqrt(n))
int mobius(int n){
  if(n == 1) return 1;
  int p = 0;
  for(int i = 2; i*i <= n; i++)
    if(n%i == 0){
      n /= i;
      p++;
      if(n%i == 0) return 0;
    }
  if(n > 1) p++;
  return p&1 ? -1 : 1;
}
*/
```

4.17 Number Theoretic Transform

```
// Number Theoretic Transform - O(nlogn)

// if long long is not necessary, use int instead to improve performance
const int mod = 20*(1<<23)+1;
const int root = 3;

ll w[N];

// a: vector containing polynomial
// n: power of two greater or equal product size
void ntt(ll* a, int n, bool inv) {
  for (int i=0, j=0; i<n; i++) {
    if (i>j) swap(a[i], a[j]);
    for (int l=n/2; (j^=l) < l; l>>=1);
  }

  // TODO: Rewrite this loop using FFT version
  ll k, t, nrev;
  w[0] = 1;
  k = exp(root, (mod-1) / n, mod);
  for (int i=1;i<=n;i++) w[i] = w[i-1] * k % mod;
  for(int i=2; i<=n; i<<=1) for(int j=0; j<n; j+=i) for(int l=0; l<(i/2); l++) {
    int x = j+l, y = j+l+(i/2), z = (n/i)*l;
    t = a[y] * w[inv ? (n-z) : z] % mod;
    a[y] = (a[x] - t + mod) % mod;
    a[x] = (a[j+l] + t) % mod;
  }

  nrev = exp(n, mod-2, mod);
  if (inv) for(int i=0; i<n; ++i) a[i] = a[i] * nrev % mod;
}

// assert n is a power of two greater of equal product size
// n = na + nb; while (n&(n-1)) n++;
void multiply(ll* a, ll* b, int n) {
  ntt(a, n, 0);
  ntt(b, n, 0);
  for (int i = 0; i < n; i++) a[i] = a[i]*b[i] % mod;
  ntt(a, n, 1);
}
```

4.18 Pollard-Rho

```
// factor(N, v) to get N factorized in vector v
// O(N ^ (1 / 4)) on average
// Miller-Rabin - Primarily Test O(|base|*(logn)^2)
ll addmod(ll a, ll b, ll m){
        if(a >= m - b) return a + b - m;
        return a + b;
}

ll mulmod(ll a, ll b, ll m){
        ll ans = 0;
        while(b){
                if(b & 1) ans = addmod(ans, a, m);
                a = addmod(a, a, m);
                b >>= 1;
        }
        return ans;
}

ll fexp(ll a, ll b, ll n){
        ll r = 1;
        while(b){
                if(b & 1) r = mulmod(r, a, n);
                a = mulmod(a, a, n);
                b >>= 1;
        }
        return r;
}

bool miller(ll a, ll n){
        if (a >= n) return true;
        ll s = 0, d = n - 1;
        while(d % 2 == 0) d >>= 1, s++;
```

```
        ll x = fexp(a, d, n);
        if (x == 1 || x == n - 1) return true;
        for (int r = 0; r < s; r++, x = mulmod(x,x,n)){
                if (x == 1) return false;
                if (x == n - 1) return true;
        }
        return false;
}

bool isprime(ll n){
        if(n == 1) return false;
        int base[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
        for (int i = 0; i < 12; ++i) if (!miller(base[i], n)) return false;
        return true;
}

ll pollard(ll n){
        ll x, y, d, c = 1;
        if (n % 2 == 0) return 2;
        while(true){
                y = x = 2;
                while(true){
                        x = addmod(mulmod(x,x,n), c, n);
                        y = addmod(mulmod(y,y,n), c, n);
                        y = addmod(mulmod(y,y,n), c, n);
                        if (x == y) break;
                        d = __gcd(abs(x-y), n);
                        if (d > 1) return d;
                }
                c++;
        }
}

vector<ll> factor(ll n){
        if (n == 1 || isprime(n)) return {n};
        ll f = pollard(n);
        vector<ll> l = factor(f), r = factor(n / f);
        l.insert(l.end(), r.begin(), r.end());
        sort(l.begin(), l.end());
        return l;
}

//n < 2,047 base = {2};
//n < 9,080,191 base = {31, 73};
//n < 2,152,302,898,747 base = {2, 3, 5, 7, 11};
//n < 318,665,857,834,031,151,167,461 base = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
//n < 3,317,044,064,679,887,385,961,981 base = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41};
```

4.19 Pollard-Rho Optimization

```
// We recomend you to use pollard-rho.cpp! I've never needed this code, but here it is.
// This uses Brent's algorithm for cycle detection
//
std::mt19937 rng((int) std::chrono::steady_clock::now().time_since_epoch().count());

ull func(ull x, ull n, ull c) { return (mulmod(x, x, n) + c) % n; // f(x) = (x^2 + c) % n; }

ull pollard(ull n) {
  // Finds a positive divisor of n
  ull x, y, d, c;
  ull pot, lam;
  if(n % 2 == 0) return 2;
  if(isprime(n)) return n;

  while(1) {
    y = x = 2; d = 1;
    pot = lam = 1;
    while(1) {
      c = rng() % n;
      if(c != 0 and (c+2)%n != 0) break;
    }
    while(1) {
      if(pot == lam) {
        x = y;
        pot <<= 1;
        lam = 0;
      }
      y = func(y, n, c);
      lam++;
      d = gcd(x >= y ? x-y : y-x, n);
      if (d > 1) {
        if(d == n) break;
        else return d;
      }
      }
    }
}

void fator(ull n, vector<ull> &v) {
  // prime factorization of n, put into a vector v.
  //
  // for each prime factor of n, it is repeated the amount of times
  // that it divides n
  //
  // ex : n == 120, v = {2, 2, 2, 3, 5};
  //
  //
  if(isprime(n)) { v.pb(n); return; }
  vector<ull> w, t; w.pb(n); t.pb(1);

  while(!w.empty()) {
    ull bck = w.back();
    ull div = pollard(bck);

    if(div == w.back()) {
      int amt = 0;
      for(int i=0; i < (int) w.size(); i++) {
        int cur = 0;
        while(w[i] % div == 0) {
          w[i] /= div;
          cur++;
        }
        amt += cur * t[i];
        if(w[i] == 1) {
          swap(w[i], w.back());
          swap(t[i], t.back());
          w.pop_back();
          t.pop_back();
        }
      }
      while(amt--) v.pb(div);
    }
    else {
      int amt = 0;
      while(w.back() % div == 0) {
        w.back() /= div;
        amt++;
      }
      amt *= t.back();
      if(w.back() == 1) {
        w.pop_back();
        t.pop_back();
      }

      w.pb(div);
      t.pb(amt);
    }
  }

  // the divisors will not be sorted, so you need to sort it afterwards
  sort(v.begin(), v.end());
}
```

4.20 Prime Factors

```
// Prime factors (up to 9*10^13. For greater see Pollard Rho)
vi factors;
int ind=0, pf = primes[0];
while (pf*pf <= n) {
  while (n%pf == 0) n /= pf, factors.pb(pf);
  pf = primes[++ind];
}
if (n != 1) factors.pb(n);
```

4.21 Primitive Root

```
// Finds a primitive root modulo p
// To make it works for any value of p, we must add calculation of phi(p)
// n is 1, 2, 4 or p^k or 2*p^k (p odd in both cases)
ll root(ll p) {
  ll n = p-1;
  vector<ll> fact;

  for (int i=2; i*i<=n; ++i) if (n % i == 0) {
```

```
    fact.push_back (i);
    while (n % i == 0) n /= i;
  }

  if (n > 1) fact.push_back (n);

  for (int res=2; res<=p; ++res) {
    bool ok = true;
    for (size_t i=0; i<fact.size() && ok; ++i)
      ok &= exp(res, (p-1) / fact[i], p) != 1;
    if (ok)  return res;
  }

  return -1;
}
```

4.22 Sieve of Eratosthenes

```
// Sieve of Erasthotenes
int p[N]; vi primes;

for (ll i = 2; i < N; ++i) if (!p[i]) {
  for (ll j = i*i; j < N; j+=i) p[j]=1;
  primes.pb(i);
}
```

4.23 Simpson Rule

```
// Simpson Integration Rule
// define the function f
double f(double x) {
  // ...
}

double simpson(double a, double b, int n = 1e6) {
    double h = (b - a) / n;
    double s = f(a) + f(b);
    for (int i = 1; i < n; i += 2) s += 4 * f(a + h*i);
    for (int i = 2; i < n; i += 2) s += 2 * f(a + h*i);
    return s*h/3;
}
```

4.24 Discrete Log

```
// O(sqrt(m))
// Solve c * a^x = b mod(m) for integer x >= 0.
// Return the smallest x possible, or -1 if there is no solution
// If all solutions needed, solve c * a^x = b mod(m) and (a*b) * a^y = b mod(m)
// x + k * (y + 1) for k >= 0 are all solutions
// Works for any integer values of c, a, b and positive m
ll discrete_log(ll c, ll a, ll b, ll m){
        c = ((c % m) + m) % m, a = ((a % m) + m) % m, b = ((b % m) + m) % m;
        if(c == b)
                return 0;

        ll g = __gcd(a, m);
        if(b % g) return -1;

        if(g > 1){
                ll r = discrete_log(c * a / g, a, b / g, m / g);
                return r + (r >= 0);
        }

        unordered_map<ll, ll> babystep;
        ll n = 1, an = a % m;

        // set n to the ceil of sqrt(m):
        while(n * n < m) n++, an = (an * a) % m;

        // babysteps:
        ll bstep = b;
        for(ll i = 0; i <= n; i++){
                babystep[bstep] = i;
                bstep = (bstep * a) % m;
        }

        // giantsteps:
        ll gstep = c * an % m;
        for(ll i = 1; i <= n; i++){
                if(babystep.find(gstep) != babystep.end())
                        return n * i - babystep[gstep];
                gstep = (gstep * an) % m;
        }
        return -1;
}
```

4.25 Simplex (Stanford)

```
// 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).

#include <iostream>
#include <iomanip>
#include <vector>
#include <cmath>
#include <limits>

using namespace std;
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()), N(n + 1), B(m), 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; // VALUE: 1.29032
  cerr << "SOLUTION:"; // SOLUTION: 1.74194 0.451613 1
  for (size_t i = 0; i < x.size(); i++) cerr << " " << x[i];
  cerr << endl;
  return 0;
}
```

5 Geometry

5.1 Miscelanneous

```
/*
1)  Square (n = 4) is the only regular polygon with integer coordinates

2)  Pick's theorem:  A = i + b/2 - 1
    A: area of the polygon
    i: number of interior points
    b: number of points on the border

3) Conic Rotations
    Given elipse: Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0
    Convert it to: Ax^2 + Bxy + Cy^2 + Dx + Ey = 1 (this formula suits better for elipse, before doing this
          verify F = 0)
    Final conversion: A(x + D/2A)^2 + C(y + E/2C)^2 = 1 + D^2/4A + E^2/4C
        B != 0 (Rotate):
            theta = atan2(b, c-a)/2.0;
            A' = (a + c + b/sin(2.0*theta))/2.0; // A
            C' = (a + c - b/sin(2.0*theta))/2.0; // C
            D' = d*sin(theta) + e*cos(theta); // D
            E' = d*cos(theta) - e*sin(theta); // E
        Remember to rotate again after!
*/


// 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;
}
```

5.2 Basics (Point)

```
#include <bits/stdc++.h>

using namespace std;

#define st first
#define nd second
#define pb push_back
#define cl(x,v) memset((x), (v), sizeof(x))
#define db(x) cerr << #x << " == " << x << endl
#define dbs(x) cerr << x << endl
#define _ << ", " <<

typedef long long ll;
typedef long double ld;
typedef pair<int,int> pii;
typedef pair<int, pii> piii;
typedef pair<ll,ll> pll;
typedef pair<ll, pll> plll;
typedef vector<int> vi;
typedef vector <vi> vii;

const ld EPS = 1e-9, PI = acos(-1.);
const ll LINF = 0x3f3f3f3f3f3f3f3f;
const int INF = 0x3f3f3f3f, MOD = 1e9+7;
const int N = 1e5+5;

typedef long double type;
//for big coordinates change to long long

bool ge(type x, type y) { return x + EPS > y; }
bool le(type x, type y) { return x - EPS < y; }
bool eq(type x, type y) { return ge(x, y) and le(x, y); }
int sign(type x) { return ge(x, 0) - le(x, 0); }

struct point {
    type x, y;

    point() : x(0), y(0) {}
    point(type x, type y) : x(x), y(y) {}

    point operator -() { return point(-x, -y); }
    point operator +(point p) { return point(x + p.x, y + p.y); }
    point operator -(point p) { return point(x - p.x, y - p.y); }

    point operator *(type k) { return point(k*x, k*y); }
    point operator /(type k) { return point(x/k, y/k); }

    //inner product
    type operator *(point p) { return x*p.x + y*p.y; }
    //cross product
    type operator %(point p) { return x*p.y - y*p.x; }

    bool operator ==(const point &p) const{ return x == p.x and y == p.y; }
    bool operator !=(const point &p) const{ return x != p.x  or y != p.y; }
    bool operator <(const point &p) const { return (x < p.x) or (x == p.x and y < p.y); }

    // 0 => same direction
    // 1 => p is on the left
    //-1 => p is on the right
    int dir(point o, point p) {
        type x = (*this - o) % (p - o);
        return ge(x,0) - le(x,0);
    }

    bool on_seg(point p, point q) {
        if (this->dir(p, q)) return 0;
        return ge(x, min(p.x, q.x)) and le(x, max(p.x, q.x)) and ge(y, min(p.y, q.y)) and le(y, max(p.y, q.y)
            );
```

```cpp
    }

    ld abs() { return sqrt(x*x + y*y); }
    type abs2() { return x*x + y*y; }
    ld dist(point q) { return (*this - q).abs(); }
    type dist2(point q) { return (*this - q).abs2(); }

    ld arg() { return atan2l(y, x); }

    // Project point on vector y
    point project(point y) { return y * ((*this * y) / (y * y)); }

    // Project point on line generated by points x and y
    point project(point x, point y) { return x + (*this - x).project(y-x); }

    ld dist_line(point x, point y) { return dist(project(x, y)); }

    ld dist_seg(point x, point y) {
        return project(x, y).on_seg(x, y) ? dist_line(x, y) :  min(dist(x), dist(y));
    }

    point rotate(ld sin, ld cos) { return point(cos*x - sin*y, sin*x + cos*y); }
    point rotate(ld a) { return rotate(sin(a), cos(a)); }

    // rotate around the argument of vector p
    point rotate(point p) { return rotate(p.y / p.abs(), p.x / p.abs()); }

};

int direction(point o, point p, point q) { return p.dir(o, q); }

point rotate_ccw90(point p)   { return point(-p.y,p.x); }
point rotate_cw90(point p)    { return point(p.y,-p.x); }

//for reading purposes avoid using * and % operators, use the functions below:
type dot(point p, point q)     { return p.x*q.x + p.y*q.y; }
type cross(point p, point q)   { return p.x*q.y - p.y*q.x; }

//double area
type area_2(point a, point b, point c) { return cross(a,b) + cross(b,c) + cross(c,a); }

int angle_less(const point& a1, const point& b1, const point& a2, const point& b2) {
    //angle between (a1 and b1) vs angle between (a2 and b2)
    //1  : bigger
    //-1 : smaller
    //0  : equal
    point p1(dot(   a1, b1), abs(cross(   a1, b1)));
    point p2(dot(   a2, b2), abs(cross(   a2, b2)));
    if(cross(p1, p2) < 0) return 1;
    if(cross(p1, p2) > 0) return -1;
    return 0;
}

ostream &operator<<(ostream &os, const point &p) {
    os << "(" << p.x << "," << p.y << ")";
    return os;
}
```

5.3 Circle

```cpp
#include "basics.cpp"
#include "lines.cpp"

struct circle {
    point c;
    ld r;
    circle() { c = point(); r = 0; }
    circle(point _c, ld _r) : c(_c), r(_r) {}
    ld area() { return acos(-1.0)*r*r; }
    ld chord(ld rad) { return  2*r*sin(rad/2.0); }
    ld sector(ld rad) { return 0.5*rad*area()/acos(-1.0); }
    bool intersects(circle other) {
        return le(c.dist(other.c), r + other.r);
    }
    bool contains(point p) { return le(c.dist(p), r); }
    pair<point, point> getTangentPoint(point p) {
        ld d1 = c.dist(p), theta = asin(r/d1);
        point p1 = (c - p).rotate(-theta);
        point p2 = (c - p).rotate(theta);
        p1 = p1*(sqrt(d1*d1 - r*r)/d1) + p;
        p2 = p2*(sqrt(d1*d1 - r*r)/d1) + p;
        return make_pair(p1,p2);
    }
};
```

```cpp
circle circumcircle(point a, point b, point c) {
    circle ans;
    point u = point((b - a).y, -(b - a).x);
    point v = point((c - a).y, -(c - a).x);
    point n = (c - b)*0.5;
    ld t = cross(u,n)/cross(v,u);
    ans.c = ((a + c)*0.5) + (v*t);
    ans.r = ans.c.dist(a);
    return ans;
}

point compute_circle_center(point a, point b, point c) {
    //circumcenter
    b = (a + b)/2;
    c = (a + c)/2;
    return compute_line_intersection(b, b + rotate_cw90(a - b), c, c + rotate_cw90(a - c));
}

int inside_circle(point p, circle c) {
    if (fabs(p.dist(c.c) - c.r)<EPS) return 1;
    else if (p.dist(c.c) < c.r) return 0;
    else return 2;
} //0 = inside/1 = border/2 = outside

circle incircle( point p1, point p2, point p3 ) {
    ld m1 = p2.dist(p3);
    ld m2 = p1.dist(p3);
    ld m3 = p1.dist(p2);
    point c = (p1*m1 + p2*m2 + p3*m3)*(1/(m1 + m2 + m3));
    ld s = 0.5*(m1 + m2 + m3);
    ld r = sqrt(s*(s - m1)*(s - m2)*(s - m3))/s);
    return circle(c, r);
}

circle minimum_circle(vector<point> p) {
    random_shuffle(p.begin(), p.end());
    circle C = circle(p[0], 0.0);
    for(int i = 0; i < (int)p.size(); i++) {
        if (C.contains(p[i])) continue;
        C = circle(p[i], 0.0);
        for(int j = 0; j < i; j++) {
            if (C.contains(p[j])) continue;
            C = circle((p[j] + p[i])*0.5, 0.5*p[j].dist(p[i]));
            for(int k = 0; k < j; k++) {
                if (C.contains(p[k])) continue;
                C = circumcircle(p[j], p[i], p[k]);
            }
        }
    }
    return C;
}

// compute intersection of line through points a and b with
// circle centered at c with radius r > 0
vector<point> circle_line_intersection(point a, point b, point c, ld r) {
    vector<point> ret;
    b = b - a;
    a = a - c;
    ld A = dot(b, b);
    ld B = dot(a, b);
    ld C = dot(a, a) - r*r;
    ld D = B*B - A*C;
    if (D < -EPS) return ret;
    ret.push_back(c + a + b*(sqrt(D + EPS) - B)/A);
    if (D > EPS)
        ret.push_back(c + a + b*(-B - sqrt(D))/A);
    return ret;
}

vector<point> circle_circle_intersection(point a, point b, ld r, ld R) {
    vector<point> ret;
    ld d = sqrt(a.dist2(b));
    if (d > r + R || d + min(r, R) < max(r, R)) return ret;
    ld x = (d*d - R*R + r*r)/(2*d);
    ld y = sqrt(r*r - x*x);
    point v = (b - a)/d;
    ret.push_back(a + v*x + rotate_ccw90(v)*y);
    if (y > 0)
        ret.push_back(a + v*x - rotate_ccw90(v)*y);
    return ret;
}

//GREAT CIRCLE

double gcTheta(double pLat, double pLong, double qLat, double qLong) {
        pLat *= acos(-1.0) / 180.0; pLong *= acos(-1.0) / 180.0; // convert degree to radian
        qLat *= acos(-1.0) / 180.0; qLong *= acos(-1.0) / 180.0;
        return acos(cos(pLat)*cos(pLong)*cos(qLat)*cos(qLong) +
                    cos(pLat)*sin(pLong)*cos(qLat)*sin(qLong) +
```

```
                sin(pLat)*sin(qLat));
}

double gcDistance(double pLat, double pLong, double qLat, double qLong, double radius) {
        return radius*gcTheta(pLat, pLong, qLat, qLong);
}


/*
 * Codeforces 101707B
 */
/*
point A, B;
circle C;

double getd2(point a, point b) {
        double h = dist(a, b);
        double r = C.r;
        double alpha = asin(h/(2*r));
        while (alpha < 0) alpha += 2*acos(-1.0);
        return dist(a, A) + dist(b, B) + r*2*min(alpha, 2*acos(-1.0) - alpha);
}

int main() {
        scanf("%lf %lf", &A.x, &A.y);
        scanf("%lf %lf", &B.x, &B.y);
        scanf("%lf %lf %lf", &C.c.x, &C.c.y, &C.r);
        double ans;
        if (distToLineSegment(C.c, A, B) >= C.r) {
                ans = dist(A, B);
        }
        else {
                pair<point, point> tan1 = C.getTangentPoint(A);
                pair<point, point> tan2 = C.getTangentPoint(B);
                ans = 1e+30;
                ans = min(ans, getd2(tan1.first, tan2.first));
                ans = min(ans, getd2(tan1.first, tan2.second));
                ans = min(ans, getd2(tan1.second, tan2.first));
                ans = min(ans, getd2(tan1.second, tan2.second));
        }
        printf("%.18f\n", ans);
        return 0;
}*/
```

5.4 Half Plane Intersection

```
// Intersection of halfplanes - O(nlogn)
// Points are given in counterclockwise order
//
// by Agnez

typedef vector<point> polygon;

int cmp(ld x, ld y = 0, ld tol = EPS) {
    return (x <= y + tol) ? (x + tol < y) ? -1 : 0 : 1; }

bool comp(point a, point b){
    if((cmp(a.x) > 0 || (cmp(a.x) == 0 && cmp(a.y) > 0) ) && (cmp(b.x) < 0 || (cmp(b.x) == 0 && cmp(b.y) < 0))
        )) return 1;
    if((cmp(b.x) > 0 || (cmp(b.x) == 0 && cmp(b.y) > 0) ) && (cmp(a.x) < 0 || (cmp(a.x) == 0 && cmp(a.y) < 0))
        )) return 0;
    ll R = a%b;
    if(R) return R > 0;
    return false;
}

namespace halfplane{
  struct L{
    point p,v;
    L(){}
    L(point P, point V):p(P),v(V){}
    bool operator<(const L &b)const{ return comp(v, b.v); }
  };
  vector<L> line;
  void addL(point a, point b){line.pb(L(a,b-a));}
  bool left(point &p, L &l){ return cmp(l.v % (p-l.p))>0; }
  bool left_equal(point &p, L &l){ return cmp(l.v % (p-l.p))>=0; }
  void init(){ line.clear(); }

  point pos(L &a, L &b){
    point x=a.p-b.p;
    ld t = (b.v % x)/(a.v % b.v);
    return a.p+a.v*t;
  }

  polygon intersect(){
    sort(line.begin(), line.end());
    deque<L> q; //linhas da intersecao
    deque<point> p; //pontos de intersecao entre elas
    q.push_back(line[0]);
    for(int i=1; i < (int) line.size(); i++){
      while(q.size()>1 && !left(p.back(), line[i]))
        q.pop_back(), p.pop_back();
      while(q.size()>1 && !left(p.front(), line[i]))
        q.pop_front(), p.pop_front();
      if(!cmp(q.back().v % line[i].v) && !left(q.back().p,line[i]))
        q.back() = line[i];
      else if(cmp(q.back().v % line[i].v))
        q.push_back(line[i]), p.push_back(point());
      if(q.size()>1)
        p.back()=pos(q.back(),q[q.size()-2]);
    }
    while(q.size()>1 && !left(p.back(),q.front()))
      q.pop_back(), p.pop_back();
    if(q.size() <= 2) return polygon(); //Nao forma poligono (pode nao ter intersecao)
    if(!cmp(q.back().v % q.front().v)) return polygon(); //Lados paralelos -> area infinita
    point ult = pos(q.back(),q.front());

    bool ok = 1;
    for(int i=0; i < (int) line.size(); i++)
      if(!left_equal(ult,line[i])){ ok=0; break; }

    if(ok) p.push_back(ult); //Se formar um poligono fechado
    polygon ret;
    for(int i=0; i < (int) p.size(); i++)
      ret.pb(p[i]);
    return ret;
  }
};

//
// Detect whether there is a non-empty intersection in a set of halfplanes
// Complexity O(n)
//
// By Agnez
//
pair<char, point> half_inter(vector<pair<point,point> > &vet){
        random_shuffle(all(vet));
        point p;
        rep(i,0,sz(vet)) if(ccw(vet[i].x,vet[i].y,p) != 1){
                point dir = (vet[i].y-vet[i].x)/abs(vet[i].y-vet[i].x);
                point l = vet[i].x - dir*1e15;
                point r = vet[i].x + dir*1e15;
                if(r<l) swap(l,r);
                rep(j,0,i){
                        if(ccw(point(),vet[i].x-vet[i].y,vet[j].x-vet[j].y)==0){
                                if(ccw(vet[j].x, vet[j].y, p) == 1)
                                        continue;
                                return mp(0,point());
                        }
                        if(ccw(vet[j].x, vet[j].y, l) != 1)
                                l = max(l, line_intersect(vet[i].x,vet[i].y,vet[j].x,vet[j].y));
                        if(ccw(vet[j].x, vet[j].y, r) != 1)
                                r = min(r, line_intersect(vet[i].x,vet[i].y,vet[j].x,vet[j].y));
                        if(!(l<r)) return mp(0,point());
                }
                p=r;
        }
        return mp(1, p);
}
```

5.5 Lines

```
#include "basics.cpp"
//functions tested at: https://codeforces.com/group/3qadGzUdR4/contest/101706/problem/B

//WARNING: all distance functions are not realizing sqrt operation
//Suggestion: for line intersections check LineLineIntersection and then use ComputeLineIntersection

point project_point_line(point c, point a, point b) {
    ld r = dot(b - a,b - a);
    if (fabs(r) < EPS) return a;
    return a + (b - a)*dot(c - a, b - a)/dot(b - a, b - a);
}

point project_point_ray(point c, point a, point b) {
    ld r = dot(b - a, b - a);
    if (fabs(r) < EPS) return a;
    r = dot(c - a, b - a) / r;
    if (le(r, 0)) return a;
```

```
    return a + (b - a)*r;
}

point project_point_segment(point c, point a, point b) {
    ld r = dot(b - a, b - a);
    if (fabs(r) < EPS) return a;
    r = dot(c - a, b - a)/r;
    if (le(r, 0)) return a;
    if (ge(r, 1)) return b;
    return a + (b - a)*r;
}

ld distance_point_line(point c, point a, point b) {
    return c.dist2(project_point_line(c, a, b));
}

ld distance_point_ray(point c, point a, point b) {
    return c.dist2(project_point_ray(c, a, b));
}

ld distance_point_segment(point c, point a, point b) {
    return c.dist2(project_point_segment(c, a, b));
}

//not tested
ld distance_point_plane(ld x, ld y, ld z,
                        ld a, ld b, ld c, ld d)
{
    return fabs(a*x + b*y + c*z - d)/sqrt(a*a + b*b + c*c);
}

bool lines_parallel(point a, point b, point c, point d) {
    return fabs(cross(b - a, d - c)) < EPS;
}

bool lines_collinear(point a, point b, point c, point d) {
  return lines_parallel(a, b, c, d)
      && fabs(cross(a-b, a-c)) < EPS
      && fabs(cross(c-d, c-a)) < EPS;
}

point lines_intersect(point p, point q, point a, point b) {
    point r = q - p, s = b - a, c(p%q, a%b);
    if (eq(r%s,0)) return point(LINF, LINF);
    return point(point(r.x, s.x) % c, point(r.y, s.y) % c) / (r%s);
}

//be careful: test LineLineIntersection before using this function
point compute_line_intersection(point a, point b, point c, point 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);
}

bool line_line_intersect(point a, point b, point c, point d) {
    if(!lines_parallel(a, b, c, d)) return true;
    if(lines_collinear(a, b, c, d)) return true;
    return false;
}


//rays in direction a -> b, c -> d
bool ray_ray_intersect(point a, point b, point c, point d){
    if (a.dist2(c) < EPS || a.dist2(d) < EPS ||
        b.dist2(c) < EPS || b.dist2(d) < EPS) return true;
    if (lines_collinear(a, b, c, d)) {
        if(ge(dot(b - a, d - c), 0)) return true;
        if(ge(dot(a - c, d - c), 0)) return true;
        return false;
    }
    if(!line_line_intersect(a, b, c, d)) return false;
    point inters = lines_intersect(a, b, c, d);
    if(ge(dot(inters - c, d - c), 0) && ge(dot(inters - a, b - a), 0)) return true;
    return false;
}

bool segment_segment_intersect(point a, point b, point c, point d) {
    if (a.dist2(c) < EPS || a.dist2(d) < EPS ||
        b.dist2(c) < EPS || b.dist2(d) < EPS) return true;
    int d1, d2, d3, d4;
    d1 = direction(a, b, c);
    d2 = direction(a, b, d);
    d3 = direction(c, d, a);
    d4 = direction(c, d, b);
    if (d1*d2 < 0 and d3*d4 < 0) return 1;
    return a.on_seg(c, d) or b.on_seg(c, d) or
           c.on_seg(a, b) or d.on_seg(a, b);
}

bool segment_line_intersect(point a, point b, point c, point d){
    if(!line_line_intersect(a, b, c, d)) return false;
    point inters = lines_intersect(a, b, c, d);
    if(inters.on_seg(a, b)) return true;
    return false;
}

//ray in direction c -> d
bool segment_ray_intersect(point a, point b, point c, point d){
    if (a.dist2(c) < EPS || a.dist2(d) < EPS ||
        b.dist2(c) < EPS || b.dist2(d) < EPS) return true;
    if (lines_collinear(a, b, c, d)) {
        if(c.on_seg(a, b)) return true;
        if(ge(dot(d - c, a - c), 0)) return true;
        return false;
    }
    if(!line_line_intersect(a, b, c, d)) return false;
    point inters = lines_intersect(a, b, c, d);
    if(!inters.on_seg(a, b)) return false;
    if(ge(dot(inters - c, d - c), 0)) return true;
    return false;
}

//ray in direction a -> b
bool ray_line_intersect(point a, point b, point c, point d){
    if (a.dist2(c) < EPS || a.dist2(d) < EPS ||
        b.dist2(c) < EPS || b.dist2(d) < EPS) return true;
    if (!line_line_intersect(a, b, c, d)) return false;
    point inters = lines_intersect(a, b, c, d);
    if(!line_line_intersect(a, b, c, d)) return false;
    if(ge(dot(inters - a, b - a), 0)) return true;
    return false;
}

ld distance_segment_line(point a, point b, point c, point d){
    if(segment_line_intersect(a, b, c, d)) return 0;
    return min(distance_point_line(a, c, d), distance_point_line(b, c, d));
}

ld distance_segment_ray(point a, point b, point c, point d){
    if(segment_ray_intersect(a, b, c, d)) return 0;
    ld min1 = distance_point_segment(c, a, b);
    ld min2 = min(distance_point_ray(a, c, d), distance_point_ray(b, c, d));
    return min(min1, min2);
}

ld distance_segment_segment(point a, point b, point c, point d){
    if(segment_segment_intersect(a, b, c, d)) return 0;
    ld min1 = min(distance_point_segment(c, a, b), distance_point_segment(d, a, b));
    ld min2 = min(distance_point_segment(a, c, d), distance_point_segment(b, c, d));
    return min(min1, min2);
}

ld DistanceRayLine(point a, point b, point c, point d){
    if(ray_line_intersect(a, b, c, d)) return 0;
    ld min1 = distance_point_line(a, c, d);
    return min1;
}

ld DistanceRayRay(point a, point b, point c, point d){
    if(ray_ray_intersect(a, b, c, d)) return 0;
    ld min1 = min(distance_point_ray(c, a, b), distance_point_ray(a, c, d));
    return min1;
}

ld DistanceLineLine(point a, point b, point c, point d){
    if(line_line_intersect(a, b, c, d)) return 0;
    return distance_point_line(a, c, d);
}
```

5.6 Minkowski Sum

```
// Given two polygons, returns the minkowski sum of them.
//
// By Agnez
bool comp(point a, point b){
        if((a.x > 0 || (a.x==0 && a.y>0) ) && (b.x < 0 || (b.x==0 && b.y<0))) return 1;
        if((b.x > 0 || (b.x==0 && b.y>0) ) && (a.x < 0 || (a.x==0 && a.y<0))) return 0;
        ll R = a%b;
        if(R) return R > 0;
        return a*a < b*b;
}

polygon poly_sum(polygon a, polygon b){
        //Lembre de nao ter pontos repetidos
```

```
	//	passar poligonos ordenados
	//	se nao tiver pontos colineares, pode usar:
	//pivot = *min_element(all(a));
	//sort(all(a),radialcomp);
	//a.resize(unique(all(a))-a.begin());
	//pivot = *min_element(all(b));
	//sort(all(b),radialcomp);
	//b.resize(unique(all(b))-b.begin());
	if(!sz(a) || !sz(b)) return polygon(0);
	if(min(sz(a),sz(b)) < 2){
		polygon ret(0);
		rep(i,0,sz(a)) rep(j,0,sz(b)) ret.pb(a[i]+b[j]);
		return ret;
	}
	polygon ret;
	ret.pb(a[0]+b[0]);
	int pa = 0, pb = 0;
	while(pa < sz(a) || pb < sz(b)){
		point p = ret.back();
		if(pb == sz(b) || (pa < sz(a) && comp((a[(pa+1)%sz(a)]-a[pa]),(b[(pb+1)%sz(b)]-b[pb]))))
			p = p + (a[(pa+1)%sz(a)]-a[pa]), pa++;
		else p = p + (b[(pb+1)%sz(b)]-b[pb]), pb++;
		//descomentar para tirar pontos colineares (o poligono nao pode ser degenerado)
//		while(sz(ret) > 1 && !ccw(ret[sz(ret)-2], ret[sz(ret)-1], p))
//			ret.pop_back();
		ret.pb(p);
	}
	assert(ret.back() == ret[0]);
	ret.pop_back();
	return ret;
}

//ITA MINKOWSKI
#include <cmath>
#define EPS 1e-9

/*
 * Point 2D
 */

struct point {
	double x, y;
	point() { x = y = 0.0; }
	point(double _x, double _y) : x(_x), y(_y) {}
	bool operator < (point other) const {
		if (fabs(x - other.x) > EPS) return x < other.x;
		else return y < other.y;
	}
	point operator +(point other) const {
		return point(x + other.x, y + other.y);
	}
	point operator -(point other) const {
		return point(x - other.x, y - other.y);
	}
	point operator *(double k) const {
		return point(x*k, y*k);
	}
};

double dist(point p1, point p2) {
	return hypot(p1.x - p2.x, p1.y - p2.y);
}

double inner(point p1, point p2) {
	return p1.x*p2.x + p1.y*p2.y;
}

double cross(point p1, point p2) {
	return p1.x*p2.y - p1.y*p2.x;
}

bool collinear(point p, point q, point r) {
	return fabs(cross(p-q, r-p)) < EPS;
}

/*
 * Polygon 2D
 */

#include <vector>
#include <algorithm>
using namespace std;

typedef vector<point> polygon;

double signedArea(polygon & P) {
	double result = 0.0;
	int n = P.size();
	for (int i = 0; i < n; i++) {
		result += cross(P[i], P[(i+1)%n]);
	}
	return result / 2.0;
}

int leftmostIndex(vector<point> & P) {
	int ans = 0;
	for(int i=1; i<(int)P.size(); i++) {
		if (P[i] < P[ans]) ans = i;
	}
	return ans;
}

polygon make_polygon(vector<point> P) {
	if (signedArea(P) < 0.0) reverse(P.begin(), P.end());
	int li = leftmostIndex(P);
	reverse(P.begin(), P.begin()+li);
	reverse(P.begin()+li, P.end());
	reverse(P.begin(), P.end());
	return P;
}

/*
 * Minkowski sum
 */

polygon minkowski(polygon & A, polygon & B) {
	polygon P; point v1, v2;
	int n1 = A.size(), n2 = B.size();
	P.push_back(A[0]+B[0]);
	for(int i = 0, j = 0; i < n1 || j < n2;) {
		v1 = A[(i+1)%n1]-A[i%n1];
		v2 = B[(j+1)%n2]-B[j%n2];
		if (j == n2 || cross(v1, v2) > EPS) {
			P.push_back(P.back() + v1); i++;
		}
		else if (i == n1 || cross(v1, v2) < -EPS) {
			P.push_back(P.back() + v2); j++;
		}
		else {
			P.push_back(P.back() + (v1+v2));
			i++; j++;
		}
	}
	P.pop_back();
	return P;
}

/*
 * Triangle 2D
 */

struct triangle {
	point a, b, c;
	triangle() { a = b = c = point(); }
	triangle(point _a, point _b, point _c) : a(_a), b(_b), c(_c) {}
	int isInside(point p) {
		double u = cross(b-a,p-a)*cross(b-a,c-a);
		double v = cross(c-b,p-b)*cross(c-b,a-b);
		double w = cross(a-c,p-c)*cross(a-c,b-c);
		if (u > 0.0 && v > 0.0 && w > 0.0) return 0;
		if (u < 0.0 || v < 0.0 || w < 0.0) return 2;
		else return 1;
	} //0 = inside/ 1 = border/ 2 = outside
};

int isInsideTriangle(point a, point b, point c, point p) {
	return triangle(a,b,c).isInside(p);
} //0 = inside/ 1 = border/ 2 = outside

/*
 * Convex query
 */

bool query(polygon &P, point q) {
	int i = 1, j = P.size()-1, m;
	if (cross(P[i]-P[0], P[j]-P[0]) < -EPS)
		swap(i, j);
	while(abs(j-i) > 1) {
		int m = (i+j)/2;
		if (cross(P[m]-P[0], q-P[0]) < 0) j = m;
		else i = m;
	}
	return isInsideTriangle(P[0], P[i], P[j], q) != 2;
}

/*
 * Codeforces 87E
 */
```

```
#include <cstdio>

void printpolygon(polygon & P) {
	printf("printing polygon:\n");
	for(int i=0; i<(int)P.size(); i++) {
		printf("%.2f %.2f\n", P[i].x, P[i].y);
	}
}

polygon city[3], P;

int main() {
	double x, y;
	for(int i = 0, n; i < 3; i++) {
		scanf("%d", &n);
		P.clear();
		while(n --> 0) {
			scanf("%lf %lf", &x, &y);
			P.push_back(point(x, y));
		}
		city[i] = make_polygon(P);
	}
	P = minkowski(city[0], city[1]);
	P = minkowski(P, city[2]);

	int m;
	scanf("%d", &m);
	while(m --> 0) {
		scanf("%lf %lf", &x, &y);
		if (query(P, point(x, y)*3.0)) printf("YES\n");
		else printf("NO\n");
	}
	return 0;
}
```

5.7 Nearest Neighbour

```
// Closest Neighbor - O(n * log(n))
const ll N = 1e6+3, INF = 1e18;
ll n, cn[N], x[N], y[N]; // number of points, closes neighbor, x coordinates, y coordinates

ll sqr(ll i) { return i*i; }
ll dist(int i, int j) { return sqr(x[i]-x[j]) + sqr(y[i]-y[j]); }
ll dist(int i) { return i == cn[i] ? INF : dist(i, cn[i]); }

bool cpx(int i, int j) { return x[i] < x[j] or (x[i] == x[j] and y[i] < y[j]); }
bool cpy(int i, int j) { return y[i] < y[j] or (y[i] == y[j] and x[i] < x[j]); }

ll calc(int i, ll x0) {
  ll dlt = dist(i) - sqr(x[i]-x0);
  return dlt >= 0 ? ceil(sqrt(dlt)) : -1;
}

void updt(int i, int j, ll x0, ll &dlt) {
  if (dist(i) > dist(i, j)) cn[i] = j, dlt = calc(i, x0);
}

void cmp(vi &u, vi &v, ll x0) {
  for(int a=0, b=0; a<u.size(); ++a) {
    ll i = u[a], dlt = calc(i, x0);
    while(b < v.size() and y[i] > y[v[b]]) b++;
    for(int j = b-1; j >= 0     and y[i] - dlt <= y[v[j]]; j--) updt(i, v[j], x0, dlt);
    for(int j = b; j < v.size() and y[i] + dlt >= y[v[j]]; j++) updt(i, v[j], x0, dlt);
  }
}

void slv(vi &ix, vi &iy) {
  int n = ix.size();
  if (n == 1) { cn[ix[0]] = ix[0]; return; }

  int m = ix[n/2];

  vi ix1, ix2, iy1, iy2;
  for(int i=0; i<n; ++i) {
    if (cpx(ix[i], m)) ix1.push_back(ix[i]);
    else ix2.push_back(ix[i]);

    if (cpx(iy[i], m)) iy1.push_back(iy[i]);
    else iy2.push_back(iy[i]);
  }

  slv(ix1, iy1);
  slv(ix2, iy2);
  cmp(iy1, iy2, x[m]);
  cmp(iy2, iy1, x[m]);
}

void slv(int n) {
  vi ix, iy;
  ix.resize(n);
  iy.resize(n);
  for(int i=0; i<n; ++i) ix[i] = iy[i] = i;
  sort(ix.begin(), ix.end(), cpx);
  sort(iy.begin(), iy.end(), cpy);
  slv(ix, iy);
}
```

5.8 Closest Pair of Points

```
#include "basics.cpp"
#include "lines.cpp"

//Graham scan NOT TESTED ENOUGH, not safe, prefer monotone chain!
point origin;

int above(point p){
    if(p.y == origin.y) return p.x > origin.x;
    return p.y > origin.y;
}

bool cmp(point p, point q){
    int tmp = above(q) - above(p);
    if(tmp) return tmp > 0;
    return p.dir(origin,q) > 0;
    //Be Careful: p.dir(origin,q) == 0
}

// Graham Scan O(nlog(n))
vector<point> graham_hull(vector<point> pts) {
    vector<point> ch(pts.size());
    point mn = pts[0];

    for(point p : pts) if (p.y < mn.y or (p.y == mn.y and p.x < mn.x)) mn = p;

    origin = mn;
    sort(pts.begin(), pts.end(), cmp);

    int n = 0;

    // IF: Convex hull without collinear points
    // for(point p : pts) {
    //     while (n > 1 and ch[n-1].dir(ch[n-2], p) < 1) n--;
    //     ch[n++] = p;
    // }

    //ELSE IF: Convex hull with collinear points
    for(point p : pts) {
    while (n > 1 and ch[n-1].dir(ch[n-2], p) < 0) n--;
    ch[n++] = p;
    }

    /*this part not safe
    for(int i=pts.size()-1; i >=1; --i)
    if (n > 1 and pts[i] != ch[n-1] and !pts[i].dir(pts[0], ch[n-1]))
        ch[n++] = pts[i];*/
    // END IF

    ch.resize(n);
    return ch;
}

//Monotone chain O(nlog(n))
#define REMOVE_REDUNDANT
#ifdef REMOVE_REDUNDANT
bool between(const point &a, const point &b, const point &c) {
    return (fabs(area_2(a,b,c)) < EPS && (a.x-b.x)*(c.x-b.x) <= 0 && (a.y-b.y)*(c.y-b.y) <= 0);
}
#endif

void monotone_hull(vector<point> &pts) {
    sort(pts.begin(), pts.end());
    pts.erase(unique(pts.begin(), pts.end()), pts.end());
    vector<point> up, dn;
    for (int i = 0; i < pts.size(); i++) {
        while (up.size() > 1 && area_2(up[up.size()-2], up.back(), pts[i]) >= 0) up.pop_back();
        while (dn.size() > 1 && area_2(dn[dn.size()-2], dn.back(), pts[i]) <= 0) dn.pop_back();
        up.push_back(pts[i]);
        dn.push_back(pts[i]);
```

```
    }
    pts = dn;
    for (int i = (int) up.size() - 2; i >= 1; i--) pts.push_back(up[i]);

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

//avoid using long double for comparisons, change type and remove division by 2
ld compute_signed_area(const vector<point> &p) {
    ld 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;
}

ld compute_area(const vector<point> &p) {
    return fabs(compute_signed_area(p));
}

ld compute_perimeter(vector<point> &p) {
    ld per = 0;
    for(int i = 0; i < p.size(); i++) {
        int j = (i+1) % p.size();
        per += p[i].dist(p[j]);
    }
    return per;
}

//not tested
// TODO: test this code. This code has not been tested, please do it before proper use.
// http://codeforces.com/problemset/problem/975/E is a good problem for testing.
point compute_centroid(vector<point> &p) {
    point c(0,0);
    ld scale = 6.0 * compute_signed_area(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;

}

// TODO: test this code. This code has not been tested, please do it before proper use.
// http://codeforces.com/problemset/problem/975/E is a good problem for testing.
point centroid(vector<point> &v) {
  int n = v.size();
  type da = 0;
  point m, c;

  for(point p : v) m = m + p;
  m = m / n;

  for(int i=0; i<n; ++i) {
    point p = v[i] - m, q = v[(i+1)%n] - m;
    type x = p % q;
    c = c + (p + q) * x;
    da += x;
  }

  return c / (3 * da);
}

//O(n^2)
bool is_simple(const vector<point> &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 (segment_segment_intersect(p[i], p[j], p[k], p[l]))
                return false;
        }
    }
    return true;
}

bool point_in_triangle(point a, point b, point c, point cur){
    ll s1 = abs(cross(b - a, c - a));
    ll s2 = abs(cross(a - cur, b - cur)) + abs(cross(b - cur, c - cur)) + abs(cross(c - cur, a - cur));
    return s1 == s2;
}

void sort_lex_hull(vector<point> &hull){
    int n = hull.size();

    //Sort hull by x
    int pos = 0;
    for(int i = 1; i < n; i++) if(hull[i] <  hull[pos]) pos = i;
    rotate(hull.begin(), hull.begin() + pos, hull.end());
}

//determine if point is inside or on the boundary of a polygon (O(logn))
bool point_in_convex_polygon(vector<point> &hull, point cur){
    int n = hull.size();
    //Corner cases: point outside most left and most right wedges
    if(cur.dir(hull[0], hull[1]) != 0 && cur.dir(hull[0], hull[1]) != hull[n - 1].dir(hull[0], hull[1]))
        return false;
    if(cur.dir(hull[0], hull[n - 1]) != 0 && cur.dir(hull[0], hull[n - 1]) != hull[1].dir(hull[0], hull[n -
        1]))
        return false;

    //Binary search to find which wedges it is between
    int l = 1, r = n - 1;
    while(r - l > 1){
        int mid = (l + r)/2;
        if(cur.dir(hull[0], hull[mid]) <= 0)l = mid;
        else r = mid;
    }
    return point_in_triangle(hull[l], hull[l + 1], hull[0], cur);
}

// determine if point is on the boundary of a polygon (O(N))
bool point_on_polygon(vector<point> &p, point q) {
for (int i = 0; i < p.size(); i++)
    if (q.dist2(project_point_segment(p[i], p[(i+1)%p.size()], q)) < EPS) return true;
    return false;
}

//Shamos - Hoey for test polygon simple in O(nlog(n))
inline bool adj(int a, int b, int n) {return (b == (a + 1)%n or a == (b + 1)%n);}

struct edge{
    point ini, fim;
    edge(point ini = point(0,0), point fim = point(0,0)) : ini(ini), fim(fim) {}
};

//< here means the edge on the top will be at the begin
bool operator < (const edge& a, const edge& b) {
    if (a.ini == b.ini) return direction(a.ini, a.fim, b.fim) < 0;
    if (a.ini.x < b.ini.x) return direction(a.ini, a.fim, b.ini) < 0;
    return direction(a.ini, b.fim, b.ini) < 0;
}

bool is_simple_polygon(const vector<point> &pts){
    vector <pair<point, pii>> eve;
    vector <pair<edge, int>> edgs;
    set <pair<edge, int>> sweep;
    int n = (int)pts.size();
    for(int i = 0; i < n; i++){
        point l = min(pts[i], pts[(i + 1)%n]);
        point r = max(pts[i], pts[(i + 1)%n]);
        eve.pb({l, {0, i}});
        eve.pb({r, {1, i}});
        edgs.pb(make_pair(edge(l, r), i));
    }
    sort(eve.begin(), eve.end());
    for(auto e : eve){
        if(!e.nd.st){
            auto cur = sweep.lower_bound(edgs[e.nd.nd]);
            pair<edge, int> above, below;
            if(cur != sweep.end()){
                below = *cur;
                if(!adj(below.nd, e.nd.nd, n) and segment_segment_intersect(pts[below.nd], pts[(below.nd + 1)
                    %n], pts[e.nd.nd], pts[(e.nd.nd + 1)%n]))
                    return false;
            }
            if(cur != sweep.begin()){
                above = *(--cur);
                if(!adj(above.nd, e.nd.nd, n) and segment_segment_intersect(pts[above.nd], pts[(above.nd + 1)
                    %n], pts[e.nd.nd], pts[(e.nd.nd + 1)%n]))
                    return false;
            }
```

```
            sweep.insert(edgs[e.nd.nd]);
        }
        else{
            auto below = sweep.upper_bound(edgs[e.nd.nd]);
            auto cur = below, above = --cur;
            if(below != sweep.end() and above != sweep.begin()){
                --above;
                if(!adj(below->nd, above->nd, n) and segment_segment_intersect(pts[below->nd], pts[(below->nd
                    + 1)%n], pts[above->nd], pts[(above->nd + 1)%n]))
                    return false;
            }
            sweep.erase(cur);
        }
    }
    return true;
}

//code copied from https://github.com/tfg50/Competitive-Programming/blob/master/Biblioteca/Math/2D%20Geometry
    /ConvexHull.cpp
int maximize_scalar_product(vector<point> &hull, point vec) {
        // this code assumes that there are no 3 colinear points
        int ans = 0;
        int n = hull.size();
        if(n < 20) {
                for(int i = 0; i < n; i++) {
                        if(hull[i] * vec > hull[ans] * vec) {
                                ans = i;
                        }
                }
        } else {
                if(hull[1] * vec > hull[ans] * vec) {
                        ans = 1;
                }
                for(int rep = 0; rep < 2; rep++) {
                        int l = 2, r = n - 1;
                        while(l != r) {
                                int mid = (l + r + 1) / 2;
                                bool flag = hull[mid] * vec >= hull[mid-1] * vec;
                                if(rep == 0) { flag = flag && hull[mid] * vec >= hull[0] * vec; }
                                else { flag = flag || hull[mid-1] * vec < hull[0] * vec; }
                                if(flag) {
                                        l = mid;
                                } else {
                                        r = mid - 1;
                                }
                        }
                        if(hull[ans] * vec < hull[l] * vec) {
                                ans = l;
                        }
                }
        }
        return ans;
}

//find tangents related to a point outside the polygon, essentially the same for maximizing scalar product
int tangent(vector<point> &hull, point vec, int dir_flag) {
        // this code assumes that there are no 3 colinear points
    // dir_flag = -1 for right tangent
    // dir_flag =  1 for left taangent
        int ans = 0;
        int n = hull.size();
        if(n < 20) {
                for(int i = 0; i < n; i++) {
                        if(hull[ans].dir(vec, hull[i]) == dir_flag) {
                                ans = i;
                        }
                }
        } else {
                if(hull[ans].dir(vec, hull[1]) == dir_flag) {
                        ans = 1;
                }
                for(int rep = 0; rep < 2; rep++) {
                        int l = 2, r = n - 1;
                        while(l != r) {
                                int mid = (l + r + 1) / 2;
                                bool flag = hull[mid - 1].dir(vec, hull[mid]) == dir_flag;
                                if(rep == 0) { flag = flag && (hull[0].dir(vec, hull[mid]) == dir_flag); }
                                else { flag = flag || (hull[0].dir(vec, hull[mid - 1]) != dir_flag); }
                                if(flag) {
                                        l = mid;
                                } else {
                                        r = mid - 1;
                                }
                        }
                        if(hull[ans].dir(vec, hull[l]) == dir_flag) {
                                ans = l;
                        }
                }
        }
        return ans;
}
```

5.9 Radial Sort

```
#include "basics.cpp"
point origin;

int above(point p){
    if(p.y == origin.y) return p.x > origin.x;
    return p.y > origin.y;
}

bool cmp(point p, point q){
    int tmp = above(q) - above(p);
    if(tmp) return tmp > 0;
    return p.dir(origin,q) > 0;
    //Be Careful: p.dir(origin,q) == 0
}
```

5.10 Stanford Delaunay

```
// Slow but simple Delaunay triangulation. Does not handle
// degenerate cases (from O'Rourke, Computational Geometry in C)
//
// Running time: O(n^4)
//
// INPUT:    x[] = x-coordinates
//           y[] = y-coordinates
//
// OUTPUT:   triples = a vector containing m triples of indices
//                     corresponding to triangle vertices

#include<vector>
using namespace std;

typedef double T;

struct triple {
    int i, j, k;
    triple() {}
    triple(int i, int j, int k) : i(i), j(j), k(k) {}
};

vector<triple> delaunayTriangulation(vector<T>& x, vector<T>& y) {
        int n = x.size();
        vector<T> z(n);
        vector<triple> ret;

        for (int i = 0; i < n; i++)
            z[i] = x[i] * x[i] + y[i] * y[i];

        for (int i = 0; i < n-2; i++) {
            for (int j = i+1; j < n; j++) {
                for (int k = i+1; k < n; k++) {
                    if (j == k) continue;
                    double xn = (y[j]-y[i])*(z[k]-z[i]) - (y[k]-y[i])*(z[j]-z[i]);
                    double yn = (x[k]-x[i])*(z[j]-z[i]) - (x[j]-x[i])*(z[k]-z[i]);
                    double zn = (x[j]-x[i])*(y[k]-y[i]) - (x[k]-x[i])*(y[j]-y[i]);
                    bool flag = zn < 0;
                    for (int m = 0; flag && m < n; m++)
                        flag = flag && ((x[m]-x[i])*xn +
                                        (y[m]-y[i])*yn +
                                        (z[m]-z[i])*zn <= 0);
                    if (flag) ret.push_back(triple(i, j, k));
                }
            }
        }
        return ret;
}

int main()
{
    T xs[]={0, 0, 1, 0.9};
    T ys[]={0, 1, 0, 0.9};
    vector<T> x(&xs[0], &xs[4]), y(&ys[0], &ys[4]);
    vector<triple> tri = delaunayTriangulation(x, y);

    //expected: 0 1 3
    //          0 3 2
```

```
    int i;
    for(i = 0; i < tri.size(); i++)
        printf("%d %d %d\n", tri[i].i, tri[i].j, tri[i].k);
    return 0;
}
```

5.11 Ternary Search

```
//Ternary Search - O(log(n))
//Max version, for minimum version just change signals

ll ternary_search(ll l, ll r){
    while(r - l > 3) {
        ll m1 = (l+r)/2;
        ll m2 = (l+r)/2 + 1;
        ll f1 = f(m1), f2 = f(m2);
        //if(f1 > f2) l = m1;
        if (f1 < f2) l = m1;
        else r = m2;
    }
    ll ans = 0;
    for(int i = l; i <= r; i++){
        ll tmp = f(i);
        //ans = min(ans, tmp);
        ans = max(ans, tmp);
    }
    return ans;
}

//Faster version - 300 iteratons up to 1e-6 precision
double ternary_search(double l, double r, int No = 300){
    // for(int i = 0; i < No; i++){
    while(r - l > EPS){
        double m1 = l + (r - l) / 3;
        double m2 = r - (r - l) / 3;
        // if (f(m1) > f(m2))
        if (f(m1) < f(m2))
            l = m1;
        else
            r = m2;
    }
    return f(l);
}
```

6 Miscellaneous

6.1 Bitset

```
//Goes through the subsets of a set x :
int b = 0;
do {
// process subset b
} while (b=(b-x)&x);
```

6.2 builtin

```
__builtin_ctz(x) // trailing zeroes
__builtin_clz(x) // leading zeroes
__builtin_popcount(x) // # bits set
__builtin_ffs(x) // index(LSB) + 1 [0 if x==0]

// Add ll to the end for long long [__builtin_clzll(x)]
```

6.3 Date

```
struct Date {
  int d, m, y;
  static int mnt[], mntsum[];

  Date() : d(1), m(1), y(1) {}
  Date(int d, int m, int y) : d(d), m(m), y(y) {}
  Date(int days) : d(1), m(1), y(1) { advance(days); }

  bool bissexto() { return (y%4 == 0 and y%100) or (y%400 == 0); }

  int mdays() { return mnt[m] + (m == 2)*bissexto(); }
  int ydays() { return 365+bissexto(); }

  int msum()  { return mntsum[m-1] + (m > 2)*bissexto(); }
  int ysum()  { return 365*(y-1) + (y-1)/4 - (y-1)/100 + (y-1)/400; }

  int count() { return (d-1) + msum() + ysum(); }

  int day() {
    int x = y - (m<3);
    return (x + x/4 - x/100 + x/400 + mntsum[m-1] + d + 6)%7;
  }

  void advance(int days) {
    days += count();
    d = m = 1, y = 1 + days/366;
    days -= count();
    while(days >= ydays()) days -= ydays(), y++;
    while(days >= mdays()) days -= mdays(), m++;
    d += days;
  }
};

int Date::mnt[13] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
int Date::mntsum[13] = {};
for(int i=1; i<13; ++i) Date::mntsum[i] = Date::mntsum[i-1] + Date::mnt[i];
```

6.4 Parentesis to Poslish (ITA)

```
#include <cstdio>
#include <map>
#include <stack>
using namespace std;

/*
 * Parenthetic to polish expression conversion
 */

inline bool isOp(char c) {
        return c=='+' || c=='-' || c=='*' || c=='/' || c=='^';
}

inline bool isCarac(char c) {
        return (c>='a' && c<='z') || (c>='A' && c<='Z') || (c>='0' && c<='9');
}

int paren2polish(char* paren, char* polish) {
        map<char, int> prec;
        prec['('] = 0;
        prec['+'] = prec['-'] = 1;
        prec['*'] = prec['/'] = 2;
        prec['^'] = 3;
        int len = 0;
        stack<char> op;
        for (int i = 0; paren[i]; i++) {
                if (isOp(paren[i])) {
                        while (!op.empty() && prec[op.top()] >= prec[paren[i]]) {
                                polish[len++] = op.top(); op.pop();
                        }
                        op.push(paren[i]);
                }
                else if (paren[i]=='(') op.push('(');
                else if (paren[i]==')') {
                        for (; op.top()!='('; op.pop())
                                polish[len++] = op.top();
                        op.pop();
                }
                else if (isCarac(paren[i]))
                        polish[len++] = paren[i];
        }
        for(; !op.empty(); op.pop())
                polish[len++] = op.top();
        polish[len] = 0;
        return len;
}

/*
```

```
 * TEST MATRIX
 */

int main() {
        int N, len;
        char polish[400], paren[400];
        scanf("%d", &N);
        for (int j=0; j<N; j++) {
                scanf(" %s", paren);
                paren2polish(paren, polish);
                printf("%s\n", polish);
        }
        return 0;
}
```

6.5 Merge Sort (Inversion Count)

```
// Merge-sort with inversion count - O(nlog n)

int n, inv;
vector<int> v, ans;

void mergesort(int l, int r, vector<int> &v){
    if(l == r) return;
    int mid = (l+r)/2;
    mergesort(l, mid, v), mergesort(mid+1, r, v);
    int i = l, j = mid + 1, k = l;
    while(i <= mid or j <= r){
        if(i <= mid and (j > r or v[i] <= v[j])) ans[k++] = v[i++];
        else ans[k++] = v[j++], inv += j-k;
    }
    for(int i = l; i <= r; i++) v[i] = ans[i];
}

//in main
ans.resize(v.size());
```

6.6 Modular Int (Struct)

```
// Struct to do basic modular arithmetic

template <int MOD>
struct Modular {
  int v;

  static int minv(int a, int m) {
    a %= m;
    assert(a);
    return a == 1 ? 1 : int(m - ll(minv(m, a)) * ll(m) / a);
  }

  Modular(ll _v = 0) : v(int(_v % MOD)) {
    if (v < 0) v += MOD;
  }

  bool operator==(const Modular& b) const { return v == b.v; }
  bool operator!=(const Modular& b) const { return v != b.v; }

  friend Modular inv(const Modular& b) { return Modular(minv(b.v, MOD)); }

  friend ostream& operator<<(ostream& os, const Modular& b) { return os << b.v; }
  friend istream& operator>>(istream& is, Modular& b) {
    ll _v;
    is >> _v;
    b = Modular(_v);
    return is;
  }

  Modular operator+(const Modular& b) const {
    Modular ans;
    ans.v = v >= MOD - b.v ? v + b.v - MOD : v + b.v;
    return ans;
  }

  Modular operator-(const Modular& b) const {
    Modular ans;
    ans.v = v < b.v ? v - b.v + MOD : v - b.v;
    return ans;
  }

  Modular operator*(const Modular& b) const {
    Modular ans;
    ans.v = int(ll(v) * ll(b.v) % MOD);
    return ans;
  }

  Modular operator/(const Modular& b) const {
    return (*this) * inv(b);
  }

  Modular& operator+=(const Modular& b) { return *this = *this + b; }
  Modular& operator-=(const Modular& b) { return *this = *this - b; }
  Modular& operator*=(const Modular& b) { return *this = *this * b; }
  Modular& operator/=(const Modular& b) { return *this = *this / b; }
};

using Mint = Modular<MOD>;
```

6.7 Parallel Binary Search

```
// Parallel Binary Search - O(nlog n * cost to update data structure + qlog n * cost for binary search
     condition)

struct Query { int i, ans; /*+ query related info*/ };
vector<Query> req;

void pbs(vector<Query>& qs, int l /* = min value*/, int r /* = max value*/) {
  if (qs.empty()) return;

  if (l == r) {
    for (auto& q : qs) req[q.i].ans = l;
    return;
  }

  int mid = (l + r) / 2;
  // mid = (l + r + 1) / 2 if different from simple upper/lower bound

  for (int i = l; i <= mid; i++) {
    // add value to data structure
  }

  vector<Query> vl, vr;
  for (auto& q : qs) {
    if (/* cond */) vl.push_back(q);
    else vr.push_back(q);
  }

  pbs(vr, mid + 1, r);

  for (int i = l; i <= mid; i++) {
    // remove value from data structure
  }

  pbs(vl, l, mid);
}
```

6.8 prime numbers

2	3	5	7	11	13	17	19	23	29
31	37	41	43	47	53	59	61	67	71
73	79	83	89	97	101	103	107	109	113
127	131	137	139	149	151	157	163	167	173
179	181	191	193	197	199	211	223	227	229
233	239	241	251	257	263	269	271	277	281
283	293	307	311	313	317	331	337	347	349
353	359	367	373	379	383	389	397	401	409
419	421	431	433	439	443	449	457	461	463
467	479	487	491	499	503	509	521	523	541
547	557	563	569	571	577	587	593	599	601
607	613	617	619	631	641	643	647	653	659
661	673	677	683	691	701	709	719	727	733
739	743	751	757	761	769	773	787	797	809
811	821	823	827	829	839	853	857	859	863
877	881	883	887	907	911	919	929	937	941
947	953	967	971	977	983	991	997	1009	1013
1019	1021	1031	1033	1039	1049	1051	1061	1063	1069
1087	1091	1093	1097	1103	1109	1117	1123	1129	1151
1153	1163	1171	1181	1187	1193	1201	1213	1217	1223
1229	1231	1237	1249	1259	1277	1279	1283	1289	1291
1297	1301	1303	1307	1319	1321	1327	1361	1367	1373

```
1381 1399 1409 1423 1427 1429 1433 1439 1447 1451
1453 1459 1471 1481 1483 1487 1489 1493 1499 1511
1523 1531 1543 1549 1553 1559 1567 1571 1579 1583
1597 1601 1607 1609 1613 1619 1621 1627 1637 1657
1663 1667 1669 1693 1697 1699 1709 1721 1723 1733
1741 1747 1753 1759 1777 1783 1787 1789 1801 1811
1823 1831 1847 1861 1867 1871 1873 1877 1879 1889
1901 1907 1913 1931 1933 1949 1951 1973 1979 1987

      970'997       971'483    921'281'269   999'279'733
1'000'000'009 1'000'000'021 1'000'000'409 1'005'012'527
```

6.9 Python

```python
# reopen
import sys
sys.stdout = open('out','w')
sys.stdin  = open('in' ,'r')

//Dummy example
R = lambda: map(int, input().split())
n, k = R(),
v, t = [], [0]*n
for p, c, i in sorted(zip(R(), R(), range(n))):
    t[i] = sum(v)+c
    v += [c]
    v = sorted(v)[::-1]
    if len(v) > k:
        v.pop()
print(' '.join(map(str, t)))
```

6.10 Sqrt Decomposition

```cpp
// Square Root Decomposition (Mo's Algorithm) - O(n^(3/2))
const int N = 1e5+1, SQ = 500;
int n, m, v[N];

void add(int p) { /* add value to aggregated data structure */ }
void rem(int p) { /* remove value from aggregated data structure */ }

struct query { int i, l, r, ans; } qs[N];

bool c1(query a, query b) {
  if(a.l/SQ != b.l/SQ) return a.l < b.l;
  return a.l/SQ&1 ? a.r > b.r : a.r < b.r;
}

bool c2(query a, query b) { return a.i < b.i; }

/* inside main */
int l = 0, r = -1;
sort(qs, qs+m, c1);
for (int i = 0; i < m; ++i) {
  query &q = qs[i];
  while (r < q.r) add(v[++r]);
  while (r > q.r) rem(v[r--]);
  while (l < q.l) rem(v[l++]);
  while (l > q.l) add(v[--l]);

  q.ans = /* calculate answer */;
}

sort(qs, qs+m, c2); // sort to original order
```

6.11 Latitude Longitude (Stanford)

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

#include <iostream>
#include <cmath>

using namespace std;

struct ll
{
  double r, lat, lon;
};

struct rect
{
  double x, y, z;
};

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

  return Q;
}

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

  return P;
}

int main()
{
  rect A;
  ll B;

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

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

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

6.12 Week day

```cpp
int v[] = { 0, 3, 2, 5, 0, 3, 5, 1, 4, 6, 2, 4 };
int day(int d, int m, int y) {
  y -= m<3;
  return (y + y/4 - y/100 + y/400 + v[m-1] + d)%7;
}
```

7 Math Extra

7.1 Combinatorial formulas

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

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

$\sum_{k=0}^{n} k^4 = (6n^5+15n^4+10n^3-n)/30$

$\sum_{k=0}^{n} k^5 = (2n^6+6n^5+5n^4-n^2)/12$

$\sum_{k=0}^{n} x^k = (x^{n+1}-1)/(x-1)$

$\sum_{k=0}^{n} kx^k = (x-(n+1)x^{n+1}+nx^{n+2})/(x-1)^2$

$\binom{n}{k} = \frac{n!}{(n-k)!k!}$

$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$

$\binom{n}{k} = \frac{n}{n-k}\binom{n-1}{k}$

$\binom{n}{k} = \frac{n-k+1}{k}\binom{n}{k-1}$

$\binom{n+1}{k} = \frac{n+1}{n-k+1}\binom{n}{k}$
$\binom{n}{k+1} = \frac{n-k}{k+1}\binom{n}{k}$
$\sum_{k=1}^{n} k\binom{n}{k} = n2^{n-1}$
$\sum_{k=1}^{n} k^2\binom{n}{k} = (n+n^2)2^{n-2}$
$\binom{m+n}{r} = \sum_{k=0}^{r}\binom{m}{k}\binom{n}{r-k}$
$\binom{n}{k} = \prod_{i=1}^{k}\frac{n-k+i}{i}$

7.2 Number theory identities

Lucas' Theorem: For non-negative integers m and n and a prime p,

$$\binom{m}{n} \equiv \prod_{i=0}^{k}\binom{m_i}{n_i} \pmod{p},$$

where

$$m = m_k p^k + m_{k-1}p^{k-1} + \cdots + m_1 p + m_0$$

is the base p representation of m, and similarly for n.

7.3 Stirling Numbers of the second kind

Number of ways to partition a set of n numbers into k non-empty subsets.

$$\left\{ {n \atop k} \right\} = \frac{1}{k!}\sum_{j=0}^{k}(-1)^{(k-j)}\binom{k}{j}j^n$$

Recurrence relation:

$$\left\{ {0 \atop 0} \right\} = 1$$

$$\left\{ {n \atop 0} \right\} = \left\{ {0 \atop n} \right\} = 1$$

$$\left\{ {n+1 \atop k} \right\} = k\left\{ {n \atop k} \right\} + \left\{ {n \atop k-1} \right\}$$

7.4 Burnside's Lemma

Let G be 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, which means $X^g = \{x \in X | g(x) = x\}$. Burnside's lemma assers the following formula for the number of orbits, denoted $|X/G|$:

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

7.5 Numerical integration

RK4: to integrate $\dot{y} = f(t, y)$ with $y_0 = y(t_0)$, compute

$$\begin{aligned}
k_1 &= f(t_n, y_n) \\
k_2 &= f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1) \\
k_3 &= f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2) \\
k_4 &= f(t_n + h, y_n + hk_3) \\
y_{n+1} &= y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)
\end{aligned}$$

	S	R	X	Assunto	Descricao	Diff
A						
B						
C						
D						
E						
F						
G						
H						
I						
J						
K						
L						