void **Zfunction**(char\* str, int\* z) {

int L = 0, R = 0, n = strlen(str);

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

for (int i = 1; i < n; i++) {

if (i <= R) z[i] = min(z[i-L], R - i + 1);

while (i + z[i] < n && str[i+z[i]] == str[z[i]])

z[ i ]++;

if (i + z[i] - 1 > R) L = i, R = i + z[i] - 1;

}

}

**int** rad[ 2 \* N ];

**void** **Manacher**(){

**int** size = **strlen**( s );

**int** i , j , k;

**for** ( i = j = 0; i < 2 \* size - 1; i += k ) {

**while** ( i - j >= 0 && i + j + 1 < 2 \* size &&

s[ ( i - j ) / 2 ] == s[ ( i + j + 1 ) / 2 ] )

j++;

rad[i] = j;

**for**(k=1; k <= rad[i] && rad[ i-k ] != rad[i] - k; k++)

rad[ i + k ] = min( rad[ i - k ], rad[i] - k );

j = max( j - k, 0 );

}

}

**int** n, \_sa[LEN], \_b[LEN], top[LEN], \_tmp[LEN], LCP[LEN],

\*SA = \_sa, \*B = \_b, \*tmp = \_tmp;

**void** **build\_sa** (){

**int** na = (n < 256 ? 256 : n);

**for** (**int** i = 0; i < n ; ++i)

top[B[i] = s[i]]++;

**for** (**int** i = 1; i < na; ++i)

top[i] += top[i - 1];

**for** (**int** i = 0; i < n ; ++i)

SA[--top[B[i]]] = i;

**for** (**int** ok = 1, j = 0; ok < n && j < n - 1; ok <<= 1){

**for** (**int** i = 0; i < n; ++i){

j = SA[i] - ok;

**if** (j < 0)

j += n;

tmp[top[B[j]]++] = j;

}

SA[tmp[top[0] = 0]] = j = 0;

**for** (**int** i = 1; i < n; ++i) {

**if** (B[tmp[i]] != B[tmp[i - 1]] || B[tmp[i] + ok] != B[tmp[i - 1] + ok])

top[++j] = i;

SA[tmp[i]] = j;

}

swap(B, SA), swap(SA, tmp);

}

}

**void** **build\_lcp** (){

**for**(**int** i = 0, k = 0; i < n; ++i){

**if**(B[i] == n - 1)

**continue**;

**for**(**int** j = SA[B[i] + 1]; s[i + k] == s[j + k]; ++k);

LCP[B[i]] = k;

**if** (k) --k;

}

}

**const** **int** LEN = 1e5 + 5, MAXS = 2 \* LEN, K = 26;

**int** go[MAXS][K], slink[MAXS], length[MAXS], final[MAXS], p, q, size = 2, last = 1, n;

**char** s[LEN], t[LEN];

**void** **extend** (**int** c){

**for** (p = last; !go[p][c]; p = slink[p])

go[p][c] = size;

q = go[p][c];

length[size] = length[last] + 1;

last = size++;

**if** (!p)

slink[last] = 1;

**else** **if** (length[p] + 1 == length[q])

slink[last] = q;

**else**{

**for**(length[size] = length[p] + 1; go[p][c] == q; p = slink[p])

go[p][c] = size;

**memcpy**(go[size], go[q], **sizeof** go[size]);

slink[size] = slink[q];

slink[last] = slink[q] = size++;

}

}

**void** **build** (){

fill(go[0], go[0] + K, 1);

length[0] = -1;

**for** (**int** i = 0; s[i]; ++i)

extend(s[i] - 'A');

**for** (; last > 0; last = slink[last])

final[last] = 1;

}

**void** **lcs** (){

**int** pos, best = 0;

**for** (**int** i = 0, len = 0, st = 1; t[i]; ++i){

**char** c = t[i] - 'A';

**if** (!go[st][c]){

**for** (; !go[st][c]; st = slink[st]);

len = length[st];

}

st = go[st][c];

**if** (++len > best)

best = len, pos = i - len + 1;

}

t[pos + best] = 0;

cout << t + pos << endl;

}

**const** **int** MAXN = 2e5, MAXS = 2 \* MAXN, K = 26;

**int** go[MAXS][K], slink[MAXS], length[MAXS], num[MAXS];

**int** sz;

**int** suff;// max suffix palindrome

ll pass[MAXN];

**char** s[MAXN];

**bool** **add**(**int** pos){

**int** st = suff, curlen = 0;

**int** c = s[pos]-'a';

**while**(**true**){

curlen = length[st];

**if**(pos-1-curlen>=0 && s[pos-1-curlen]==s[pos])**break**;

st = slink[st];

}

**if**(go[st][c]){

suff = go[st][c];

pass[suff]++;

**return** **false**;

}

sz++;

suff = sz;

length[suff] = length[st] + 2;

go[st][c] = suff;

**if**(length[suff] == 1){

slink[suff] = 2;

num[suff] = 1;

pass[suff]++;

**return** **true**;

}

**while**(**true**) {

st = slink[st];

curlen = length[st];

**if** (pos-1-curlen >= 0 && s[pos-1-curlen]==s[pos]){

slink[suff] = go[st][c];

**break**;

}

}

pass[suff]++;

num[suff] = num[slink[suff]] + 1;

**return** **true**;

}

// node 1 - root with len -1, node 2 - root with len 0

**void** **init**(){

sz = 2, suff = 2;

length[1] = -1; slink[1] = 1;

length[2] = 0; slink[2] = 1;

}

ll hash[MAXN], pot[MAXN];

ll **get**(**int** i , **int** j ){

**if**(i == 0)

**return** hash[j];

**return** (hash[j] - pot[j - i + 1] \* hash[i - 1] % MOD + MOD) % MOD;

}

**int** **main**(){

init();

cin >> s;

**int** n = **strlen**(s);

**for** (**int** i = 0; i < n; i++)

add(i);

**for**( **int** i = sz; i > 2; i-- ){

pass[slink[i]] += pass[i];

}

}

,

Pick’s theorem: I = A – B/2 –h + 1 (real division, h is number oh holes)

, ,

The distance between the incenter and circumcenter is

**const** **double** PI = **acos**(-1);

**const** **double** EPS = 1e-8;

**#define** sqr(x) ((x)\*(x))

**#define** xx real()

**#define** yy imag()

**typedef** complex<**double**> P;

**struct** L : **public** vector<P> {

**L**(**const** P &a, **const** P &b) {

push\_back(a); push\_back(b);

}

**L**() {}

};

**typedef** vector<P> G;

**#define** curr(P, i) P[i]

**#define** next(P, i) P[(i+1)%P.size()]

**double** **cross**(**const** P& a, **const** P& b) {

**return** imag(conj(a)\*b);

}

**double** **dot**(**const** P& a, **const** P& b) {

**return** real(conj(a)\*b);

}

**int** **ccw**(P a, P b, P c) {

b -= a; c -= a;

**if** (cross(b, c) > 0) **return** +1; // counter clockwise

**if** (cross(b, c) < 0) **return** -1; // clockwise

**if** (dot(b, c) < 0) **return** +2; // c--a--b on line

**if** (norm(b) < norm(c)) **return** -2; // a--b--c on line

**return** 0; // a--c--b (leftover)

}

**namespace** std {

**bool** **operator <** (**const** P& a, **const** P& b) {

**if** (abs(a-b)<EPS) **return** 0;

**return** real(a) != real(b) ?

real(a) < real(b) : imag(a) < imag(b);

}

}

G **convex\_hull**(G ps){

**int** n = ps.size(), k = 0;

sort(ps.begin(), ps.end());

G ch(2 \* n);

// lower-hull

**for** (**int** i = 0 ; i < n; ch[k++] = ps[i++])

**while** (k >= 2 && ccw(ch[k - 2], ch[k - 1], ps[i]) <= 0)

--k;

// upper-hull

**for** (**int** i = n-2 , t = k + 1 ; i>=0 ; ch[k++] = ps[i--]) **while** (k >= t && ccw(ch[k-2], ch[k-1], ps[i]) <= 0)

--k;

ch.resize(k - 1);

**return** ch;

}

**bool** intersectLL(**const** L &l, **const** L &m) {

**return** abs(cross(l[1]-l[0], m[1]-m[0])) > EPS ||

abs(cross(l[1]-l[0], m[0]-l[0])) < EPS;

}

**bool** intersectLS(**const** L &l, **const** L &s) {

**return** cross(l[1]-l[0], s[0]-l[0])\*

cross(l[1]-l[0], s[1]-l[0]) < EPS;

}

**bool** intersectLP(**const** L &l, **const** P &p) {

**return** abs(cross(l[1]-p, l[0]-p)) < EPS;

}

**bool** intersectSS(**const** L &s, **const** L &t) {

**return** ccw(s[0],s[1],t[0])\*ccw(s[0],s[1],t[1]) <= 0 &&

ccw(t[0],t[1],s[0])\*ccw(t[0],t[1],s[1]) <= 0;

}

**bool** intersectSS2(**const** L &s, **const** L &t) {

REP(i, 2) {

**if** (ccw(s[0], s[1], t[i]) == 0) {

**int** c = ccw(s[0],s[1],t[!i]);

**if** (s[0] == t[i]) {

**if** (c!=-2&&c) **return** 0;

} **else** **if** (s[1] == t[i]) {

**if** (c!=2&&c) **return** 0;

} **else** **if** (abs(c)==1) **return** 0;

}

}

**return** ccw(s[0],s[1],t[0])\*ccw(s[0],s[1],t[1]) <= 0 &&

ccw(t[0],t[1],s[0])\*ccw(t[0],t[1],s[1]) <= 0;

}

**bool** intersectSP(**const** L &s, **const** P &p) {

**return** abs(s[0]-p)+abs(s[1]-p)-abs(s[1]-s[0]) < EPS;

}

P projection(**const** L &l, **const** P &p) {

**double** t = dot(p-l[0], l[0]-l[1]) / norm(l[0]-l[1]);

**return** l[0] + t\*(l[0]-l[1]);

}

P reflection(**const** L &l, **const** P &p) {

**return** p + P(2,0) \* (projection(l, p) - p);

}

**double** distanceLP(**const** L &l, **const** P &p) {

**return** abs(p - projection(l, p));

}

**double** distanceLL(**const** L &l, **const** L &m) {

**return** intersectLL(l, m) ? 0 : distanceLP(l, m[0]);

}

**double** istanceLS(**const** L &l, **const** L &s) {

**if** (intersectLS(l, s)) **return** 0;

**return** min(distanceLP(l, s[0]), distanceLP(l, s[1]));

}

**double** distanceSP(**const** L &s, **const** P &p) {

**const** P r = projection(s, p);

**if** (intersectSP(s, r)) **return** abs(r - p);

**return** min(abs(s[0] - p), abs(s[1] - p));

}

**double** distanceSS(**const** L &s, **const** L &t) {

**if** (intersectSS(s, t)) **return** 0;

**return** min(min(distanceSP(s, t[0]), distanceSP(s, t[1])),

min(distanceSP(t, s[0]), distanceSP(t, s[1])));

}

P crosspoint(**const** L &l, **const** L &m) {

**double** A = cross(l[1] - l[0], m[1] - m[0]);

**double** B = cross(l[1] - l[0], l[1] - m[0]);

**if** (abs(A) < EPS && abs(B) < EPS) **return** m[0]; // same line

**if** (abs(A) < EPS) assert(**false**); // no intersection !!!

**return** m[0] + B / A \* (m[1] - m[0]);

}

**double** area(**const** G& g) {

**double** A = 0;

**for** (**int** i = 0; i < g.size(); ++i) {

A += cross(g[i], next(g, i));

}

**return** abs(A/2);

}

**double** perimeter(**const** G& g) {

**double** A = 0;

**for** (**int** i = 0; i < g.size(); ++i) {

A += abs(g[i] - next(g, i));

}

**return** A;

}

// true if any of the rectangles fits into the other

**bool** rect\_in\_rect(**int** a, **int** b, **int** c, **int** d){

// if want to test (a,b) inside (c,d) comment this

**if** (a\*b > c\*d) swap(a,c), swap(b,d);

**if** (a<b) swap(a,b);

**if** (c<d) swap(c,d);

**if** (a<=c && b<=d) **return** **true**;

**double** lo, mi, hi, A, B; // see KTU(2)A

lo = 0; //largest doesn't fit

hi = mi = M\_PI/2; //largest fits

**for**(**int** it=0; it<=200; it++){

mi = (lo+hi)/2;

**if** (a\*cos(mi) + b\*sin(mi) <= c ) hi = mi;

**else** lo = mi;

}

A = a\*cos(mi) + b\*sin(mi);

B = a\*sin(mi) + b\*cos(mi);

**return** (A<=c+EPS && B<=d+EPS);

}

G convex\_cut(**const** G& g, **const** L& l) {

G Q;

REP(i, g.size()) {

P A = curr(g, i), B = next(g, i);

**if** (ccw(l[0], l[1], A) != -1) Q.push\_back(A);

**if** (ccw(l[0], l[1], A)\*ccw(l[0], l[1], B) < 0)

Q.push\_back(crosspoint(L(A, B), l));

}

**return** Q;

}

P centroid(**const** vector<P> &v) {

**double** S = 0;

P res;

REP(i,v.size()) {

**int** j = i+1;

**if** (j == v.size()) j = 0;

**double** tmp = cross(v[i], v[j]);

S += tmp;

res += (v[i] + v[j]) \* tmp;

}

S /= 2;

res /= 6\*S;

**return** res;

}

**double** manDistanceSP(**const** L &l, **const** P &p) {

**double** res = INF;

L xl = L(p, p + P(1,0));

**if** (intersectLS(xl, l)){

P cp = crosspoint(xl, l);

**double** d = abs(p-cp);

res = min(res, d);

}

L yl = L(p, p + P(0,1));

**if** (intersectLS(yl, l)) {

P cp = crosspoint(yl, l);

**double** d = abs(p-cp);

res = min(res, d);

}

res = min(res, abs(l[0].real()-p.real()) + abs(l[0].imag()-p.imag()));

res = min(res, abs(l[1].real()-p.real()) + abs(l[1].imag()-p.imag()));

**return** res;

}

**bool** convex\_contain(**const** G &g, **const** P &p) {

REP(i,g.size())

**if** (ccw(g[i], next(g, i), p) == -1) **return** 0;

**return** 1;

}

**bool** intersectGG(**const** G &g1, **const** G &g2) {

**if** (convex\_contain(g1, g2[0])) **return** 1;

**if** (convex\_contain(g2, g1[0])) **return** 1;

REP(i,g1.size()) REP(j,g2.size()) {

**if** (intersectSS(L(g1[i], next(g1, i)), L(g2[j], next(g2, j)))) **return** 1;

}

**return** 0;

}

**double** distanceGP(**const** G &g, **const** P &p) {

**if** (convex\_contain(g, p)) **return** 0;

**double** res = INF;

REP(i, g.size()) {

res = min(res, distanceSP(L(g[i], next(g, i)), p));

}

**return** res;

}

L bisector(**const** P &a, **const** P &b) {

P A = (a+b)\*P(0.5,0);

**return** L(A, A+(b-a)\*P(0, PI/2));

}

G voronoi\_cell(G g, **const** vector<P> &v, **int** s) {

REP(i, v.size())

**if** (i!=s)

g = convex\_cut(g, bisector(v[s], v[i]));

**return** g;

}

**double** angle(**const** P &a, **const** P &b) {

**double** ret = arg(b)-arg(a);

**return** (ret>=0) ? ret : ret + 2\*PI;

}

**double** angle2(**const** P &a, **const** P &b) {

**return** min(angle(a,b), angle(b,a));

}

**double** rtod(**double** rad) {

**return** rad\*180/PI;

}

**double** dtor(**double** deg) {

**return** deg\*PI/180;

}

P rotate(P p, **double** ang) {

**return** p \* P(cos(ang), sin(ang));

}

L rotate(L l, **double** ang) {

**return** L(rotate(l[0], ang),rotate(l[1], ang));

}

P rotate(P p, P about, **double** ang){

**return** (p - about) \* exp(P(0, ang)) + about;

}

P rotleft(P p){

**return** P(-p.imag(), p.real());

}

P rotright(P p){

**return** P(p.imag(), -p.real());

}

P trunc(P p, **double** l){

**return** p\*(l/abs(p));

}

**double** angle(P p){

**return** atan2(p.real(), p.imag());

}

P angled(P p, **double** ang){

P ret = p+P(0, -1);

**return** rotate(ret, p, -ang);

}

**struct** circle{

P p; **double** r;

circle(**const** P &p, **double** r) : p(p), r(r) { }

circle(){}

circle(P a, P b, P c){

p = crosspoint(bisector(a, b), bisector(b, c));

r = abs(a-p);

}

**bool** **operator**< (circle o)**const**{

**if** (abs(r-o.r)>EPS) **return** r<o.r;

**if** (abs(p.xx-o.p.xx)>EPS) **return** p.xx<o.p.xx;

**if** (abs(p.yy-o.p.yy)>EPS) **return** p.yy<o.p.yy;

**return** **false**;

}

**int** get\_circles(P a, P b, **double** r, circle&c1, circle&c2){

circle x(a, r), y(b, r);

**int** ret = x.crosspoint\_circle(y, c1.p, c2.p);

**if** (!ret) **return** 0;

c1.r = c2.r = 3;

**return** ret;

}

**int** crosspoint\_circle(circle c, P&p1, P&p2){

**double** d = abs(p - c.p);

**if** (d > r+c.r)

**return** 0;

**if** (d < abs(r-c.r))

**return** 0;

**double** l = (d+(sqr(r)-sqr(c.r))/d)/2;

**double** h = sqrt(sqr(r)-sqr(l));

p1 = p + trunc(c.p-p, l) + trunc(rotleft(c.p-p), h);

p2 = p + trunc(c.p-p, l) + trunc(rotright(c.p-p), h);

**return** 2;

}

**int** crosspoint\_line(L l, P&p1, P&p2) {

P a = projection(l, p);

**double** d = abs(a-p);

**if** (d > r)

**return** 0;

d = sqrt(r\*r - d\*d);

p1= a- trunc(l[1] - l[0], d);

p2= a+ trunc(l[1] - l[0], d);

**return** 2; // pending... 1

}

};

circle minCircle(vector<P> & p\_) {

vector<P> p = convex\_hull(p\_);

**int** n = SZ(p);

cerr << n << endl;

**if** (n == 1)

**return** circle(p[0], 0);

**for** (**int** i = 0; i < n; i++)

swap(p[rand()%n], p[rand()%n]);

**double** r = norm(p[0]-p[1]) / 4.0;

P q = (p[0] + p[1]) \* 0.5;

**for** (**int** i = 0; i < n; i++) {

**if** (norm(p[i]-q) < r + EPS) **continue**;

r = norm(p[i]-p[0]) / 4.0;

q = (p[i] + p[0]) \* 0.5;

**for** (**int** j = 1; j < i; j++) {

**if** (norm(p[j]-q) < r + EPS) **continue**;

r = norm(p[j]-p[i]) / 4.0;

q = (p[j] + p[i]) \* 0.5;

**for** (**int** k = 0; k < j; k++) {

**if** (norm(p[k]-q) < r+EPS) **continue**;

q = crosspoint(bisector(p[i], p[j]), bisector(p[j], p[k]));

r = norm(p[i]-q) / 4.0;

}

}

}

**return** circle(q, sqrt(r));

}

**int** **dblcmp**(**double** d)

{

**if** (**fabs**(d)<EPS)**return** 0;

**return** d>EPS?1:-1;

}

**struct** point3

{

**double** x,y,z;

**point3**(){}

**point3**(**double** \_x,**double** \_y,**double** \_z):

x(\_x),y(\_y),z(\_z){};

**bool** **operator==**(point3 a){

**return** dblcmp(a.x-x)==0&&dblcmp(a.y-y)==0&&dblcmp(a.z-z)==0;

}

**bool** **operator<**(point3 a)**const**{

**return** dblcmp(a.x-x)==0?dblcmp(y-a.y)==0?dblcmp(z-a.z)<0:y<a.y:x<a.x;

}

**double** **len**(){ **return** **sqrt**(len2()); }

**double** **len2**(){ **return** x\*x+y\*y+z\*z; }

**double** **distance**(point3 p){

**return** **sqrt**((p.x-x)\*(p.x-x)+(p.y-y)\*(p.y-y)+(p.z-z)\*(p.z-z));

}

point3 **add**(point3 p){ **return** point3(x+p.x,y+p.y,z+p.z); }

point3 **sub**(point3 p){ **return** point3(x-p.x,y-p.y,z-p.z); }

point3 **mul**(**double** d){ **return** point3(x\*d,y\*d,z\*d); }

point3 **div**(**double** d){ **return** point3(x/d,y/d,z/d); }

**double** **dot**(point3 p){

**return** x\*p.x+y\*p.y+z\*p.z;

}

point3 **det**(point3 p){

**return** point3(y\*p.z-p.y\*z,p.x\*z-x\*p.z,x\*p.y-p.x\*y);

}

**double** **rad**(point3 a,point3 b){

point3 p=(\***this**);

**return** **acos**(a.sub(p).dot(b.sub(p))/(a.distance(p)\*b.distance(p)));

}

point3 **trunc**(**double** r){

r/=len();

**return** point3(x\*r,y\*r,z\*r);

}

point3 **rotate**(point3 o,**double** r) // is below

{

}

};

**struct** line3

{

point3 a,b;

**line3**(){}

**line3**(point3 \_a,point3 \_b){

a=\_a;

b=\_b;

}

**bool** **operator==**(line3 v){

**return** (a==v.a)&&(b==v.b);

}

**double** **length**(){

**return** a.distance(b);

}

**bool** **pointonseg**(point3 p){

**return** dblcmp(p.sub(a).det(p.sub(b)).len())==0&&dblcmp(a.sub(p).dot(b.sub(p)))<=0;

}

**double** **dispointtoline**(point3 p)

{

**return** b.sub(a).det(p.sub(a)).len()/a.distance(b);

}

**double** **dispointtoseg**(point3 p)

{

**if** (dblcmp(p.sub(b).dot(a.sub(b)))<0||dblcmp(p.sub(a).dot(b.sub(a)))<0)

{

**return** min(p.distance(a),p.distance(b));

}

**return** dispointtoline(p);

}

point3 **lineprog**(point3 p)

{

**return** a.add(b.sub(a).trunc(b.sub(a).dot(p.sub(a))/b.distance(a)));

}

point3 **rotate**(point3 p,**double** ang)

{

**if** (dblcmp((p.sub(a).det(p.sub(b)).len()))==0)**return** p;

point3 f1=b.sub(a).det(p.sub(a));

point3 f2=b.sub(a).det(f1);

**double** len=**fabs**(a.sub(p).det(b.sub(p)).len()/a.distance(b));

f1=f1.trunc(len);f2=f2.trunc(len);

point3 h=p.add(f2);

point3 pp=h.add(f1);

**return** h.add((p.sub(h)).mul(**cos**(ang\*1.0))).add((pp.sub(h)).mul(**sin**(ang\*1.0)));

}

};

**struct** plane

{

point3 a,b,c,o;

**plane**(){}

**plane**(point3 \_a,point3 \_b,point3 \_c)

{

a=\_a;

b=\_b;

c=\_c;

o=pvec();

}

**plane**(**double** \_a,**double** \_b,**double** \_c,**double** \_d)

{

//ax+by+cz+d=0

o=point3(\_a,\_b,\_c);

**if** (dblcmp(\_a)!=0)

{

a=point3((-\_d-\_c-\_b)/\_a,1,1);

}

**else** **if** (dblcmp(\_b)!=0)

{

a=point3(1,(-\_d-\_c-\_a)/\_b,1);

}

**else** **if** (dblcmp(\_c)!=0)

{

a=point3(1,1,(-\_d-\_a-\_b)/\_c);

}

}

point3 **pvec**()

{

**return** b.sub(a).det(c.sub(a));

}

**bool** **pointonplane**(point3 p)

{

**return** dblcmp(p.sub(a).dot(o))==0;

}

**int** **pointontriangle**(point3 p)

{

**if** (!pointonplane(p))**return** 0;

**double** s=a.sub(b).det(c.sub(b)).len();

**double** s1=p.sub(a).det(p.sub(b)).len();

**double** s2=p.sub(a).det(p.sub(c)).len();

**double** s3=p.sub(b).det(p.sub(c)).len();

**if** (dblcmp(s-s1-s2-s3))**return** 0;

**if** (dblcmp(s1)&&dblcmp(s2)&&dblcmp(s3))**return** 2;

**return** 1;

}

**bool** **relationplane**(plane f)

{

**if** (dblcmp(o.det(f.o).len()))**return** 0;

**if** (pointonplane(f.a))**return** 2;

**return** 1;

}

**double** **angleplane**(plane f)

{

**return** **acos**(o.dot(f.o)/(o.len()\*f.o.len()));

}

**double** **dispoint**(point3 p)

{

**return** **fabs**(p.sub(a).dot(o)/o.len());

}

point3 **pttoplane**(point3 p)

{

line3 u=line3(p,p.add(o));

crossline(u,p);

**return** p;

}

**int** **crossline**(line3 u,point3 &p)

{

**double** x=o.dot(u.b.sub(a));

**double** y=o.dot(u.a.sub(a));

**double** d=x-y;

**if** (dblcmp(**fabs**(d))==0)**return** 0;

p=u.a.mul(x).sub(u.b.mul(y)).div(d);

**return** 1;

}

**int** **crossplane**(plane f,line3 &u)

{

point3 oo=o.det(f.o);

point3 v=o.det(oo);

**double** d=**fabs**(f.o.dot(v));

**if** (dblcmp(d)==0)**return** 0;

point3 q=a.add(v.mul(f.o.dot(f.a.sub(a))/d));

u=line3(q,q.add(oo));

**return** 1;

}

};

**double** trunc\_volume(**double** r, **double** R, **double** h){

**double** a = r\*r\*M\_PI;

**double** A = R\*R\*M\_PI;

**return** ((a+A+sqrt(a\*A))/3) \* h; // by Heron

}

**double** sphere\_volume(**double** r){

**return** r\*r\*r\*M\_PI\*(4.0/3.0);

}

**double** cone\_volume(**double** r, **double** h){

**return** r\*r\*h\*M\_PI\*(1.0/3.0);

}

**double** pyramid\_volume(**double** A, **double** h){

**return** A\*h\*(1.0/3.0);

}

**double** ellipsoid\_volume(**double** a, **double** b, **double** c){

// semiaxis

**return** a\*b\*c\*M\_PI\*(4.0/3.0);

}

// rotar p con eje de rotacion r

XYZ rotate(XYZ p, XYZ r, double theta){

XYZ q(0,0,0);

double costheta,sintheta;

r.normalize();

costheta = cos(theta);

sintheta = sin(theta);

q.x += (costheta + (1-costheta) \* r.x \* r.x) \* p.x;

q.x += ((1-costheta) \* r.x \* r.y - r.z \* sintheta) \* p.y;

q.x += ((1-costheta) \* r.x \* r.z + r.y \* sintheta) \* p.z;

q.y += ((1-costheta) \* r.x \* r.y + r.z \* sintheta) \* p.x;

q.y += (costheta + (1 - costheta) \* r.y \* r.y) \* p.y;

q.y += ((1-costheta) \* r.y \* r.z - r.x \* sintheta) \* p.z;

q.z += ((1-costheta) \* r.x \* r.z - r.y \* sintheta) \* p.x;

q.z += ((1-costheta) \* r.y \* r.z + r.x \* sintheta) \* p.y;

q.z += (costheta + (1 - costheta) \* r.z \* r.z) \* p.z;

return q;

}

**namespace** Antipodals{

**typedef** complex<ll> point;

**typedef** pair<**int**, **int**> pii;

**#define** NEXT(i) (((i) + 1) % n)

**typedef** vector<point> polygon;

**inline** ll **area** (point a, point b, point c){ //actually 2 \* area

**return** abs(cross(b - a, c - a));

}

vector<pii> **get\_antip\_pairs** (polygon &P){

vector<pii> ans;

**int** n = P.size();

**if** (P.size() == 2)

ans.push\_back(make\_pair(0, 1));

**if** (P.size() < 3)

**return** ans;

ans.reserve(3 \* n);

**int** q0 = 0;

**while** (area(P[n - 1], P[0], P[NEXT(q0)]) > area(P[n - 1], P[0], P[q0]))

++q0;

**for** (**int** q = q0, p = 0; q != 0 && p <= q0; ++p) {

ans.push\_back(make\_pair(p, q));

**while** (area(P[p], P[NEXT(p)], P[NEXT(q)]) > area(P[p], P[NEXT(p)], P[q])){

q = NEXT(q);

**if** (p != q0 || q != 0)

ans.push\_back(make\_pair(p, q));

**else**

**return** ans;

}

**if** (area(P[p], P[NEXT(p)], P[NEXT(q)]) == area(P[p], P[NEXT(p)], P[q])){

**if** (p != q0 || q != n - 1)

ans.push\_back(make\_pair(p, NEXT(q)));

**else**

ans.push\_back(make\_pair(NEXT(p), q));

}

}

// sort(ans.begin(), ans.end()); to get points\_segment...

**return** ans;

}

**bool** **CONTAINS**(vector<pii> &a , pii x){

**return** \*(lower\_bound(a.begin() , a.end() , x)) == x;

}

**bool** check(**int** p , **int** q , **int** qq , vector<pii> &antipodal\_pairs){

pii p1 = make\_pair( min(p , q), max(p , q));

pii p2 = make\_pair( min(p , qq), max(p , qq));

**return** CONTAINS(antipodal\_pairs, p1) && CONTAINS(antipodal\_pairs, p2);

}

//INPUT

//n:ammount of vertices in convex hull;

//antipodal\_pairs: list of antipodal pairs

//OUTPUT: //vector<pii> vertex-p , edge: q-next(q)

vector<pii> get\_vertex\_edge\_pairs(**int** n, vector<pii> &antipodal\_pairs){

vector<pii> ans;

**int** sz = antipodal\_pairs.size();

**for**(**int** i = 0; i < sz; i++ ){

pii idx = antipodal\_pairs[i];

**if**(check(idx.first , idx.second, NEXT(idx.second), antipodal\_pairs))

ans.push\_back(idx);

swap(idx.first , idx.second);

**if**(check(idx.first , idx.second, NEXT(idx.second), antipodal\_pairs))

ans.push\_back(idx);

}

**return** ans;

}

**double** linePointDist(**const** point &A,**const** point &B, **const** point &C){

**double** dist = cross(B - A,C - A) / norm(A-B);

**return** abs(dist);

}

**double** width (polygon &p){

p = convex\_hull(p);

**int** n = p.size();

vector<pii> pairs = get\_antip\_pairs(p);

pairs = get\_vertex\_edge\_pairs(p.size(), pairs);

**int** sz = pairs.size();

**double** sol = 1e9;

**for**( **int** i = 0; i < sz; i++ ){

pii pair = pairs[i];

**double** dist = linePointDist(p[pair.second], p[(pair.second+1)%n], p[pair.first]);

sol = min(sol, dist);

}

**return** sol;

}

}

point bary(point A, point B, point C, **double** a, **double** b, **double** c) {

**return** (A\*a + B\*b + C\*c) / (a + b + c);

}

// geometric center of mass

point centroid(point A, point B, point C) {

**return** bary(A, B, C, 1, 1, 1);

}

// intersection of perpendicular bisectors

point circumcenter(point A, point B, point C) {

**double** a = norm(B - C), b = norm(C - A), c = norm(A - B);

**return** bary(A, B, C, a\*(b+c-a), b\*(c+a-b), c\*(a+b-c));

}

// intersection of internal angle bisectors

point incenter(point A, point B, point C) {

**return** bary(A, B, C, abs(B-C), abs(A-C), abs(A-B));

}

// intersection of altitudes

point orthocenter(point A, point B, point C) {

**double** a = norm(B - C), b = norm(C - A), c = norm(A - B);

**return** bary(A, B, C, (a+b-c)\*(c+a-b), (b+c-a)\*(a+b-c), (c+a-b)\*(b+c-a));

}

// intersection of two external angle bisectors

point excenter(point A, point B, point C) {

**double** a = abs(B - C), b = abs(A - C), c = abs(A - B);

**return** bary(A, B, C, -a, b, c);

//// NOTE: there are three excenters

// return bary(A, B, C, a, -b, c);

// return bary(A, B, C, a, b, -c);

}

ld norm(ld a){

**while**(a < -PI) a += 2\*PI;

**while**(a > PI) a -= 2\*PI;

**return** a;

}

**void** add(ld l, ld r){

E[cnt++] = (event){l, **true**};

E[cnt++] = (event){r, **false**};

**if** (l>r)

curr++;

}

**void** dadd(ld alpha, ld theta){

ld l = norm(alpha-theta);

ld r = norm(alpha+theta);

add(l, r);

}

// angle events (segment)

**void** add(point p1, point p2){

ld a1 = atan2(p1.xx, p1.yy);

ld a2 = atan2(p2.xx, p2.yy);

**if** (a1>a2) swap(a1, a2);

**if** (a2-a1 > PI) swap(a1, a2);

dadd(a1, a2);

}

**const** **int** N = 600;

**int** isin[N];

**const** **double** eps = 1e-13;

**struct** angle{

**double** deg;

**int** cid;

}angles[N \* N];

**bool** cmp(**const** angle &a, **const** angle &b){

**if**(fabs(a.deg - b.deg) < eps)

**return** a.cid < b.cid;

**return** a.deg< b.deg;

}

**int** solve( **int** n , **double** r , **double** \*x , **double** \* y){

memset(isin , 0 , **sizeof** isin);

**double** tmpang , theta , dsqr , d;

**int** cnt = 0, i , j , tmp , tmp2, c\_ang = 0;

**for**( i = 0, cnt= (n > 0); i < n; i++){

**for**(j = 0, c\_ang = 0; j < n; j++ ){

**if**(j != i){

dsqr = sqr(x[j] - x[i]) + sqr(y[j] - y[i]);

**if**(sqr(2 \*r) + eps < dsqr)**continue**;

d = sqrt(dsqr);

**if**(x[i] == x[j] && y[i] == y[j])**continue**;

tmpang = atan2((**double**)(y[j] - y[i]), (**double**)(x[j] - x[i])), theta = acos(d / (2 \* r));

angles[c\_ang].deg = tmpang + theta + eps, angles[c\_ang++].cid = j;

angles[c\_ang].deg = tmpang - theta - eps, angles[c\_ang++].cid = j;

}

}

**for**(j = 0; j < c\_ang; angles[j].deg += (angles[j].deg < 0 ? 2 \* PI : 0), j++ );

sort(angles , angles + c\_ang, cmp);

**for**(j = tmp = 0; j < n; j++){

**if**(sqr(x[j] - (x[i] + r)) + sqr(y[j] - y[i]) <= sqr(r) + eps)

isin[j] = 1, tmp++;

**else** isin[j] = 0;

}

**for**(j = 0, cnt = max(cnt , tmp); j < c\_ang; cnt = max(cnt , tmp), j++ ){

tmp2 = angles[j].cid;

tmp += isin[tmp2] ? -1 : 1;

isin[tmp2] ^= 1;

}

}

**return** cnt;

}

**namespace** Voronoi{

**namespace** mnumeric{

**const** **double** eps = 1e-9, oo = 1e9;

**bool** lt(**double** a, **double** b) { **return** a + eps < b; }

**bool** gt(**double** a, **double** b) { **return** lt(b, a); }

**bool** eq(**double** a, **double** b) { **return** !lt(a, b) && !gt(a, b); }

**double** sq(**double** x) { **return** x \* x; }

}

**using** **namespace** mnumeric;

**namespace** mpoint{

**struct** point {

**double** x, y;

point() {}

point(**double** x, **double** y) :

x(x), y(y) {}

};

**bool** **operator**<(point a, point b) {

**if** (eq(a.x, b.x))

**return** lt(a.y, b.y);

**return** lt(a.x, b.x);

}

**bool** **operator**==(point a, point b)

{ **return** eq(a.x, b.x) && eq(a.y, b.y); }

point midpoint(point a, point b)

{ **return** point(0.5 \* (a.x + b.x), 0.5 \* (a.y + b.y)); }

}

**using** **namespace** mpoint;

**namespace** mvoronoi{

**bool** open;

**double** X, Y; // intersection point

**double** w, h; // bounding rectangle

**struct** bisector;

**typedef** deque< bisector > vcell;

**struct** bisector {

**double** A, B, C;

**double** ang; // angle

**double** hyp; // distance between points

**int** id;

bisector(**double** A, **double** B, **double** C, **double** hyp) :

A(A), B(B), C(C), hyp(hyp) {

ang = atan2(A, B);

id = -1;

}

bisector(point a, point b, **int** id) : id(id) {

point m = midpoint(a, b);

A = b.x - a.x;

B = b.y - a.y;

C = - A \* m.x - B \* m.y;

ang = atan2(A, B);

hyp = hypot(A, B);

}

};

**bool** **operator**<(bisector a, bisector b) {

**if** (eq(a.ang, b.ang))

**return** gt(a.hyp, b.hyp);

**return** lt(a.ang, b.ang);

}

**bool** isect(bisector a, bisector b) {

**double** det = a.A \* b.B - a.B \* b.A;

**if** (eq(det, 0.0)) **return** **false**;

X = -(a.C \* b.B - a.B \* b.C) / det;

Y = +(a.C \* b.A - a.A \* b.C) / det;

**return** **true**;

}

**bool** shadowed(bisector a, bisector b, bisector c) {

isect(a, b);

**return** gt(c.A \* X + c.B \* Y + c.C, 0.0);

}

vcell cell(**int** i, vector< point > &p, **bool** bounded) {

vcell bs;

**int** n = p.size();

bounded = **true**; // not bounded still not fully supported

**if** (bounded){

bs.push\_back(bisector(-1, 0, -w, 2\*(p[i].x + w))); //x=0

bs.push\_back(bisector(0, -1, -h, 2\*(p[i].y + h))); //y=0

bs.push\_back(bisector(1, 0, -w, 2\*(w - p[i].x))); // x=w

bs.push\_back(bisector(0, 1, -h, 2\*(h - p[i].y))); // y=h

}

**for** (**int** j = 0; j < n; ++j) {

**if** (i == j) **continue**;

bs.push\_back(bisector(p[i], p[j], j));

}

sort(bs.begin(), bs.end());

deque< bisector > stk;

**for** (**int** j = 0; j < bs.size(); ++j) {

**while** (!stk.empty() && eq(bs[j].ang, stk.back().ang))

stk.pop\_back();

**while** (stk.size() >= 2 && shadowed(stk[stk.size() - 2], stk[stk.size() - 1], bs[j]))

stk.pop\_back();

stk.push\_back(bs[j]);

}

**bool** change = **true**;

**while** (change) {

change = **false**;

**while** (stk.size() >= 3 && shadowed(stk[1], stk[0], stk.back()))

stk.pop\_front(), change = **true**;

**while** (stk.size() >= 3 && shadowed(stk[stk.size() - 1], stk[stk.size() - 2], stk[0]))

stk.pop\_back(), change = **true**;

}

open = **false**;

**for**(**int** I=0; I<SZ(stk); I++)

open |= stk[I].id==-1;

**return** stk;

}

**void** print(vcell v){

**for**(**int** I=0; I<SZ(v); I++)

cout << v[I].id << " ";

cout << endl;

}

}

}

**using** **namespace** Voronoi::mvoronoi;

**int** N;

**double** sol;

vector< point > P;

**void** read(){

cin >> N;

P.resize(N);

**for**(**int** I=0; I<N; I++) cin>>P[I].x>>P[I].y;

}

**void** closest\_pair\_distance{

read();

sol = oo;

**for**(**int** I=0; I<N; I++){

vcell curr = cell(I, P, **true**);

**for**(**int** K=0; K<SZ(curr); K++){

**int** id = curr[K].id;

**if** (id == -1) **continue**;

sol = min(sol, hypot(P[I].x-P[id].x, P[I].y-P[id].y) );

}

}

cout << sol << endl;

}

**void** convex\_hull(){

// bisectors with id == -1 indicate an open direction

// just find first open cell and ‘walk’ through adjacent

// open cells (with a -1 bisector)

}

**namespace** SecondDistance{

**double** curr[3];

vector<point> cand;

**void** solve(vector< point > &p) {

**int** n = p.size();

**for** (**int** i = 0; i < n; ++i) {

vcell bs = cell(i, p, **true**); // no open cells

bs.PB(bs.back());

**for** (**int** j = 0; j < SZ(bs)-1; ++j)

**if** (isect(bs[j], bs[j+1]))

cand.push\_back(point(X, Y)); // voronoi segment-vertices

}

}

**void** maximum\_second\_distance{

cin >> w >> h >> N;

P.resize(N);

**for**(**int** I=0; I<N; I++)

cin >> P[I].x >> P[I].y;

**for**(**int** I=0; I<N; I++){

vcell curr = cell(I, P, **true**);

vector< point > L;

**for**(**int** K=0; K<SZ(curr); K++)

**if** (curr[K].id != -1)

L.PB(P[curr[K].id]);

solve(L);

}

sol = 0;

**for**(**int** I=0; I<SZ(cand); I++){

curr[0]=curr[1]=curr[2]=oo;

**for**(**int** K=0; K<N; K++){

curr[2] = hypot(cand[I].x-P[K].x, cand[I].y-P[K].y);

sort(curr, curr+3);

}

sol = max(sol, curr[1]);

}

cout << sol << endl;

}

}

**typedef** pair<**int**, **int**> pii;  
**int** **cross**(**int** ax, **int** ay, **int** bx, **int** by, **int** cx, **int** cy) {  
    **return** (bx - ax) \* (cy - ay) - (by - ay) \* (cx - ax);  
}  
**int** **cross**(pii a, pii b, pii c) {  
    **return** cross(a.first, a.second, b.first, b.second, c.first, c.second);  
}  
**class** segment {  
    public:  
    pii a, b;  
    **int** id;  
    **segment**(pii a, pii b, **int** id) :  
        **a**(a), **b**(b), **id**(id) {  
    }  
    **bool** **operator**<(**const** segment &o) **const** {  
        **if** (a.first < o.a.first) {  
            **int** s = cross(a, b, o.a);  
            **return** (s > **0** || s == **0** && a.second < o.a.second);  
        } else {  
            **int** s = cross(o.a, o.b, a);  
            **return** (s < **0** || s == **0** && a.second < o.a.second);  
        }  
        **return** a.second < o.a.second;  
    }  
};  
**bool** **intersect**(segment s1, segment s2) {  
    **int** x1 = s1.a.first, y1 = s1.a.second, x2 = s1.b.first, y2 = s1.b.second;  
    **int** x3 = s2.a.first, y3 = s2.a.second, x4 = s2.b.first, y4 = s2.b.second;  
    **if** (**max**(x1, x2) < **min**(x3, x4) || **max**(x3, x4) < **min**(x1, x2) || **max**(y1, y2) < **min**(y3, y4) || **max**(y3, y4) < **min**(y1, y2)) {  
        **return** **false**;  
    }  
    **int** z1 = (x3 - x1) \* (y2 - y1) - (y3 - y1) \* (x2 - x1);  
    **int** z2 = (x4 - x1) \* (y2 - y1) - (y4 - y1) \* (x2 - x1);  
    **if** (z1 < **0** && z2 < **0** || z1 > **0** && z2 > **0**) {  
        **return** **false**;  
    }  
    **int** z3 = (x1 - x3) \* (y4 - y3) - (y1 - y3) \* (x4 - x3);  
    **int** z4 = (x2 - x3) \* (y4 - y3) - (y2 - y3) \* (x4 - x3);  
    **if** (z3 < **0** && z4 < **0** || z3 > **0** && z4 > **0**) {  
        **return** **false**;  
    }  
    **return** **true**;  
}  
**class** event {  
    public:  
    pii p;  
    **int** id;  
    **int** type;  
    **event**(pii p, **int** id, **int** type) :  
        **p**(p), **id**(id), **type**(type) {  
    }  
    **bool** **operator**<(**const** event &o) **const** {  
        **return** p.first < o.p.first || p.first == o.p.first && (type > o.type || type == o.type && p.second < o.p.second);  
    }  
};  
  
pii **findIntersection**(vector<segment> a) {  
    **int** n = a.size();  
    vector<event> e;  
    **for** (**int** i = **0**; i < n; ++i) {  
        **if** (a[i].a > a[i].b)  
            swap(a[i].a, a[i].b);  
        e.push\_back(event(a[i].a, i, **1**));  
        e.push\_back(event(a[i].b, i, -**1**));  
    }  
    sort(e.begin(), e.end());  
  
    set<segment> q;  
  
    **for** (**int** i = **0**; i < n \* **2**; ++i) {  
        **int** id = e[i].id;  
        **if** (e[i].type == **1**) {  
            set<segment>::iterator it = q.lower\_bound(a[id]);  
            **if** (it != q.**end**() && **intersect**(\*it, a[id]))  
                **return** make\_pair(it->id, a[id].id);  
            **if** (it != q.**begin**() && **intersect**(\*--it, a[id]))  
                **return** make\_pair(it->id, a[id].id);  
            q.insert(a[id]);  
        } else {  
            set<segment>::iterator it = q.lower\_bound(a[id]), next = it, prev = it;  
            **if** (it != q.**begin**() && it != --q.**end**()) {  
                ++next, --prev;  
                **if** (**intersect**(\*next, \*prev))  
                    **return** make\_pair(next->id, prev->id);  
            }  
            q.erase(it);  
        }  
    }  
    **return** make\_pair(-**1**, -**1**);  
}

Sum of divisors:

Postage stamps/McNuggets problem. Let a, b, be relatively-prime integers. There are exaclty numbers **not** of form ax+by (), and the largest is .

Pythagoren triples. All relatively prime triples are given by x=2mn, y=m2-n2, z=m2+n2 where m>n, gcd(m,n)=1 and m≠n(mod 2).

Fermat’s two-squares theorem. Odd prime p can be represented as a sum of two squares iff p=1(mod 4). A product of two sums of two squares is a sum of two squares. Thus, n is a sum of twos squares iff every prime of form p=4k+3 occurs an even number of times in n’s factorization.

//Solve ax+by=(a,b)

**int** **gcd**(**int** a,**int** b,**int** &x,**int** &y) {

**if**(b==0) {x=1;y=0;**return** a;}

**int** r=gcd(b,a%b,y,x);

y-=a/b\*x;

**return** r;

}

//Solve a^x=b(mod m)

**int** **dlog**(**int** a,**int** b,**int** m,**int** p[]) {

hash\_map<**int**,**int**> hash;

**int** n=phi(m,p),k=sqrt(n);

**for**(**int** i=0,t=1;i<k;i++)

{

hash[t]=i;

t=(**ll**)t\*(**ll**)a%m;

}

**int** c=pow\_mod(a,n-k,m);

**for**(**int** i=0;i\*k<n;i++)

{

**if**(hash.find(b)!=hash.end()) **return** i\*k+hash[b];

b=(**ll**)b\*(**ll**)c%m;

}

**return** -1;

}

//Solve x=ai(mod mi), for any i and j, (mi,mj)|ai-aj.

//Return x0 in [0,[m1..mn]). All solutions are x=x0+t[m1..mn].

**int** **LinearCon**(**int** a[],**int** m[],**int** n)

{

**int** u=a[0],v=m[0],p,q,r,t;

**for**(**int** i=1;i<n;i++)

{

r=gcd(v,m[i],p,q);

t=v;

v=v/r\*m[i];

u=((a[i]-u)/r\*p\*t+u)%v;

}

**if**(u<0) u+=v;

**return** u;

}

//n shouldn't be prime

ll **PollardRho**(ll n) {

**if**(!(n&1)) **return** 2;

**while**(**true**)

{

ll x=(ll)**rand**()%n,y=x;

ll c=**rand**()%n;

**if**(c==0||c==2) c=1;

**for**(**int** i=1,k=2;;i++)

{

x=mul\_mod(x,x,n);

**if**(x>=c) x-=c; **else** x+=n-c;

**if**(x==n) x=0;

**if**(x==0) x=n-1; **else** x--;

ll d=gcd(x>y?x-y:y-x,n);

**if**(d==n) **break**;

**if**(d!=1) **return** d;

**if**(i==k) {y=x;k<<=1;}

}

}

}

//returns true if a is quadratic residue modulo p (prime)

**bool** **is\_quadratic\_residue**(**int** a, **int** p) {

**if**(a==0) **return** **true**;

**return** powmod(a,(p-1)/2,p)==1;

}

//solves a\*x = b (mod p)

**ll** **solve\_linear**(**ll** a, **ll** b, **int** p) {

**return** (b\*inverse(a,p))%p;

}

//kth discrete roots of a (mod n), (k, phi(n)) = 1

**int** **discrete\_root**(**int** k, **int** a, **int** n) {

**int** phi = phi(n);

**int** s = (**int**)inverse(k,phi);

**return** (**int**)powmod(a,s,n);

}

//x^2=a (mod p)

**ll** **solve\_quadratic**(**ll** a, **int** p) {

**if**(a==0) **return** 0;

**if**(p==2) **return** a;

**if**(powmod(a,(p-1)/2,p)!= 1) **return** -1;

**int** phi = p-1;

**int** n = 0,k = 0;

**while**(phi%2==0)

{

phi/=2; n++;

}

k = phi;

**int** q = 0;

**for**(**int** j = 2; j < p; j++)

**if**(powmod(j,(p-1)/2,p)==p-1)

{

q = j; **break**;

}

**ll** t = powmod(a,(k+1)/2,p);

**ll** r = powmod(a,k,p);

**while**(r!=1)

{

**int** i = 0, v = 1;

**while**(powmod(r,v,p)!=1){v\*=2;i++;}

**ll** e = powmod(2,n-i-1,p);

**ll** u = powmod(q,k\*e,p);

t = (t\*u)%p;

r = (r\*u\*u)%p;

}

**return** (**int**)t;

}

//solves a\*x^2 + b\*x +c=0 (mod p)

set<ll> **solve\_quadratic**(ll a, ll b, ll c, **int** p){

set<ll> ans;

**if**(c==0) ans.insert(0L);

**if**(a==0) ans.insert(solve\_linear((p-b)%p,c,p));

**else** **if**(p==2 && (a+b+c)%2==0) ans.insert(1L);

**else**

{

ll r = ((b\*b)%p -(4\*a\*c)%p+ p)%p;

ll x = solve\_quadratic(r,p);

**if**(x == -1) **return** ans;

ll w = solve\_linear((2\*a)%p,(x-b+p)%p,p);

ans.insert(w);

w = solve\_linear((2\*a)%p,(p-x-b+p)%p,p);

ans.insert(w);

}

**return** ans;

}

//primitive roots

**int** **primitive\_root**(**int** m,**int** p[]) {

//only 2, 4, p^n, 2p^n have primitive root

**if**(m==1) **return** 0;

**if**(m==2) **return** 1;

**if**(m==4) **return** 3;

**int** t=m;

**if**((t&1)==0) t>>=1;

**for**(**int** i=0;p[i]\*p[i]<=t;i++)

{

**if**(t%p[i]) **continue**;

**do** t/=p[i]; **while**(t%p[i]==0);

**if**(t>1||p[i]==2) **return** 0;

}

**int** x=phi(m,p),y=x,f[32],n=0;

**for**(**int** i=0;p[i]\*p[i]<=y;i++)

{

**if**(y%p[i]) **continue**;

**do** y/=p[i]; **while**(y%p[i]==0);

f[n++]=p[i];

}

**if**(y>1) f[n++]=y;

**for**(**int** i=1;i<m;i++)

{

**if**(gcd(i,m)>1) **continue**;

**bool** flag=**true**;

**for**(**int** j=0;j<n;j++)

**if**(pow\_mod(i,x/f[j],m)==1)

{

flag=**false**;

**break**;

}

**if**(flag) **return** i;

}

**return** 0;

}

**bool witness**(ll a, ll s, ll d, ll n) {

int64 x = modPow(a, d, n);

**if** (x == 1 || x == n - 1) **return false**; // composite

REP(i, s - 1) {

x = modMul(x, x, n);

**if** (x == 1) **return true**; // probably prime

**if** (x == n - 1) **return false**; // composite

}

**return true**; // probably prime

}

**bool isPrime**(ll n)

{

**if** (n < 2) **return false**;

**if** (n == 2) **return true**;

**if** (n % 2 == 0) **return false**;

ll d = n - 1, s = 0;

**while** (d % 2 == 0) ++s, d /= 2;

ll test[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 0};

**for** (**int** i = 0; test[i] && test[i] < n; ++i)

**if** (witness(test[i], s, d, n))

**return false**; // composite

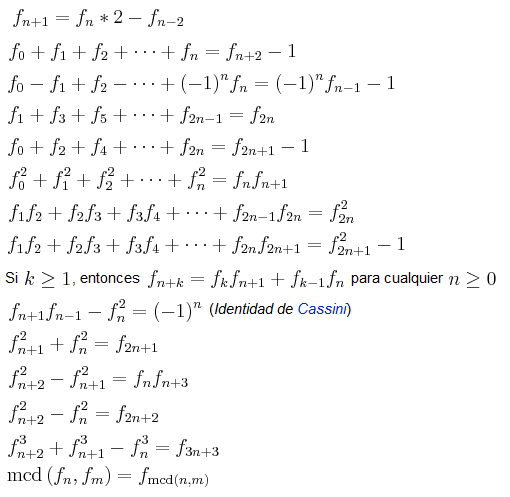
**return true**; // probably prime

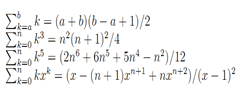
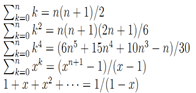
}

Number of multisets of size k from n elements is

Number of n-tuples of ≥ 0 ints with sum = s:

Number of n-tuples of ≥ 0 ints with sum ≤ s:

****

****

Stirling 1st kind: The number of ways to split n elements into k ordered partitions up to a permutation of the partitions among themselves and rotations within the partitions.

Stirling 2nd kind: is the number of ways to partition an n-set into exactly k non-empty disjoint subsets up to a permutation of the sets among themselves.

Bell is the number of equivalence relations on an n-set or, alternatively, the number of partitions of an n-set.



Lucca’s Theorem: (for a prime p)

Derangements: 1, 0, 1, 2, 9, 44, 265, 1854, 133496

In the following, let G be a finite group that acts on a set X. For each g in G let Xg denote the set of elements in X that are fixed by g (also said to be left invariant by g), i.e. Xg = { x ∈ X | g.x = x }. Burnside's lemma asserts the following formula for the number of orbits, denoted |X/G|:

When considering rotations on a necklace, all N rotations are needed.

INCL

Euler’s theorem. For any planar graph

Matrix-tree theorem. Let matrix T=[tij], where tij is the number of multiedges between i and j, for i≠j, and tii=-degi. Number of spanning trees of a graph is equal to the determinant of a matrix obtained by deleting any k-th row and kth column from T.

Number of connected labeled graphs:

**namespace** EulerTour{

**const** **int** MAXN = 100100;

**int** N, a, b; // 1-indexed

unordered\_set<**int**> E[MAXN];

**void** DFS(**int** n, vector<**int**> &sol){

**if** (E[n].empty()) sol.PB(n);

**while**(!E[n].empty()){

**int** x = \*E[n].begin();

E[n].erase(x);

E[x].erase(n);

DFS(x, sol);

sol.PB(n);

}

}

vector<**int**> Path(){

vector<**int**> ret;

a = b = 1;

**for**(**int** I=1; I<=N; I++)

**if** (SZ(E[I]) & 1) a = b, b = I;

DFS(a, ret);

assert(ret.back() == a && ret.front() == b);

**return** ret;

}

}

**namespace** DirectedEulerTour{

**const** **int** MAXN = 100100;

**int** N, a, b; // 1-indexed

vector<**int**> E[MAXN];

**int** indeg[MAXN], outdeg[MAXN];

**void** init(){

**for**(**int** I=1; I<=N; I++) {

outdeg[I] = SZ(E[I]);

**for**(**auto** x: E[I]) indeg[x] ++;

}

}

**bool** hasTour(){

**for**(**int** I=1; I<=N; I++) **if** (indeg[I]!=outdeg[I]) **return** **false**;

**return** **true**;

}

**bool** hasPath(){

**int** source = 0;

**int** sink = 0;

**for**(**int** I=1; I<=N; I++) {

**if** (outdeg[I]>indeg[I]+1 || outdeg[I]+1<indeg[I]) **return** **false**;

source += outdeg[I]>indeg[I]+1;

sink += outdeg[I]+1<indeg[I];

}

**return** source==sink && (source <= 1);

}

**void** DFS(**int** n, vector<**int**> &sol){

**if** (E[n].empty()) sol.PB(n);

**while**(!E[n].empty()){

**int** x = E[n].back();

E[n].pop\_back();

DFS(x, sol);

sol.PB(n);

}

}

vector<**int**> Path(){

vector<**int**> ret;

assert(hasPath());

a = b = 1;

**for**(**int** I=1; I<=N; I++){

**if** (outdeg[I]>indeg[I]+1) a = I;

**if** (outdeg[I]+1<indeg[I]) b = I;

}

DFS(a, ret);

assert(ret.back() == a && ret.front() == b);

**return** ret;

}

}

**int** V, E ,i, j,a, b, size, gtime;

stack <pii> Q;

**int** disc[MaxV], back[MaxV];

**bool** mark[MaxE];

vector<pii> bic[MaxV];

vector<pii> graph[MaxV];

**void** **dfs**(**int** v) {

gtime++;

disc[v] = gtime;

back[v] = gtime;

**for** (**int** k = graph[v].size() - 1; k >= 0; k--) {

**int** next = graph[v][k].first;

**int** edge = graph[v][k].second;

**if** (!mark[edge]) {

Q.push(pii(v, next));

mark[edge] = **true**;

}

**if** (!disc[next]) {

dfs(next);

back[v] = min(back[v], back[next]);

**if** (back[next] >= disc[v]) {

size++;

**for** (;;) {

pii x = Q.top(); Q.pop();

bic[size].push\_back(x);

**if** (x == pii(v, next))

**break**;

}

}

} **else** back[v] = min(back[v], disc[next]);

}

}

**int** **main**() {

cin >> V >> E;

**for** (i = 0; i < E; i++) {

cin >> a >> b;

graph[a].push\_back(pii(b, i));

graph[b].push\_back(pii(a, i));

}

**for** (i = 1; i <= V; i++)

**if** (!disc[i]) dfs(i);

**for** (i = 1; i <= size; i++) {

cout << "Biconnected Component: " << i << endl;

**for** (j = bic[i].size() - 1; j >= 0; j--)

cout << bic[i][j].first << " " << bic[i][j].second << endl;

}

**return** 0;

}

**namespace** SAT{

**const** **int** MAXV = 1 << 17;

// 2-SAT

**int** V, SCC, dtime;

**int** low[2 \* MAXV], dfsnum[2 \* MAXV], scomp[2 \* MAXV];

**bool** deleted[2 \* MAXV], used[2 \* MAXV]; // solution

**bool** mark[2 \* MAXV];

vector< **int** > G[2 \* MAXV], C[2 \* MAXV]; // SCC list

vector< **int** > order;

stack< **int** > S;

**int** VAR( **int** x ) { **return** x; }

**int** NOT( **int** x ) { **return** ( x < V ) ? x + V : x - V; }

// some useful clauses

**void** add\_OR( **int** u, **int** v ) { G[ NOT(u) ].push\_back( v ); G[ NOT(v) ].push\_back( u ); }

**void** add\_AND( **int** u, **int** v ) { add\_OR( u, u ); add\_OR( v, v ); }

**void** add\_XOR( **int** u, **int** v ) { add\_OR( u, v ); add\_OR( NOT(u), NOT(v) ); }

**void** add\_EQ( **int** u, **int** v ) { add\_OR( NOT(u), v ); add\_OR( u, NOT(v) ); }

**void** add\_ON( **int** u ) { add\_OR( u, u ); }

**void** add\_OFF( **int** u ) { add\_OR( NOT(u), NOT(u) ); }

**void** dfs( **int** u ) {

dfsnum[u] = low[u] = ++dtime;

S.push( u );

REP( i, G[u].size() ) {

**int** v = G[u][i];

**if** ( !dfsnum[v] ) {

dfs( v );

low[u] = min( low[u], low[v] );

} **else**

**if** ( !deleted[v] )

low[u] = min( low[u], dfsnum[v] );

}

**if** ( dfsnum[u] == low[u] ) {

**while** ( !deleted[u] ) {

**int** x = S.top();

deleted[x] = **true**;

scomp[x] = SCC;

C[SCC].push\_back( x );

S.pop();

}

order.push\_back( SCC ); // topological order

SCC++;

}

}

**bool** solve(){

REP(i, 2\*V)

**if** (!dfsnum[i])

dfs(i);

REP(i, V) **if** (scomp[i] == scomp[NOT(i)])

**return** **false**;

REP(i, SCC)

**if** ( !mark[ order[i] ] )

REP( j, C[ order[i] ].size() ) {

**int** x = C[ order[i] ][j];

used[x] = **true**;

mark[ scomp[ NOT(x) ] ] = **true**;

}

**return** **true**;

}

}

**const** **int** MAXN = 200100, MAXK = 1000010; // always check !!!!!!

**int** desc[MAXN], n , k;

**bool** mk[MAXN];

vector<pii> G[MAXN];

**int** **FindCentre**( **int** u , **int** nodes , **int** parent = -1 ){

**int** size = G[u].size();

desc[u] = 0;

**bool** flag = **true**;

**for**( **int** i = 0; i < size; i++ ){

**int** v = G[u][i].first;

**if**(!mk[v] && v != parent){

**int** c = FindCentre(v , nodes , u);

**if**(c != -1)**return** c;

flag &= (desc[v] \* 2 <= nodes);

desc[u] += desc[v];

}

}

++desc[u];

flag &= ((nodes - desc[u]) \* 2 <= nodes);

**return** flag ? u : -1;

}

**int** **countNodes**(**int** u, **int** parent = -1){

**int** sum = 1;

**int** sz = G[u].size();

**for**( **int** i = 0; i < sz; i++ ){

**int** v = G[u][i].first;

**if**(!mk[v] && v != parent)

sum += countNodes(v , u);

}

**return** sum;

}

**void** **solve\_using\_u**( **int** u ){

//solve the subproblem using vertex u

}

**void** **DivideAndConquer**( **int** u ){

**int** nodes = countNodes(u);

**int** centre = FindCentre(u , nodes);

assert(centre != -1);

mk[centre] = 1;

solve\_using\_u( centre );

**int** sz = G[centre].size();

**for**(**int** i = 0; i < sz; i++ ){

**int** v = G[centre][i].first;

**if**(!mk[v])

DivideAndConquer(v);

}

}

**int** heavy[MAXN], sz[MAXN], p[MAXN], depth[MAXN]; //heavy

**int** t[4 \* MAXN]; //segment tree

**int** s[MAXN], arr[MAXN];

**void** **dfs**( **int** u, **int** par = -1){

p[u] = par;

sz[u] = 1;

**for**( **int** i = 0; i < G[u].size(); i++ ){

**int** v = G[u][i].first;

**int** w = G[u][i].second;

**if**(v != par){

s[v] = w;

depth[v] = depth[u] + 1;

dfs(v, u);

sz[u] += sz[v];

**if**(!heavy[u] || sz[v] > sz[heavy[u]])

heavy[u] = v;

}

}

}

//You always fill segment tree array on this method

//ONE-BASED!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

**int** head[MAXN], chainpos[MAXN], chain[MAXN], ch;

**void** **hld**(){

dfs(1);

**for**( **int** i = 1, j = 0; i <= n; ++i)

**if**(p[i] == -1 || heavy[p[i]] != i) {

++ch;

**for**( **int** k = i ; k; k = heavy[k] )

arr[j] = s[k], chainpos[k] = j++, head[k] = i, chain[k] = ch;

}

//build segment tree

build(1 , 0 , n - 1);

}

**int** **solve**( **int** a , **int** b ){

**int** res = 0;

**while**( chain[a] != chain[b] ){

**if**(depth[head[a]] < depth[head[b]]) swap(a , b);

**int** start = head[a];

res = res + query(1 , 0 , n - 1, chainpos[start], chainpos[a]);

a = p[start];

}

**if**(depth[a] > depth[b])

swap(a , b);

res = res + query(1 , 0 , n - 1, chainpos[a], chainpos[b]);

res = res - arr[chainpos[a]];

**return** res;

}

**const** **int** MAXV = 1e4+100, MAXE = 1e6, oo = 0x3f3f3f3f;

**int** E, last[MAXV], next[MAXE], adj[MAXE], level[MAXV], now[MAXV], Q[MAXV], source, sink;

ll cap[MAXE], flow[MAXE];

**void** **add\_edge** (**int** u, **int** v, **int** c){

cap[E] = c, flow[E] = 0, adj[E] = v, next[E] = last[u], last[u] = E++;

cap[E] = 0, flow[E] = 0, adj[E] = u, next[E] = last[v], last[v] = E++;

}

**#define** RES(e) (cap[e] - flow[e])

**bool** **bfs** (**int** s, **int** t) {

**memset**(level, -1, **sizeof** level);

level[s] = 0;

Q[0] = s;

**for** (**int** ql = 0, qr = 1; ql < qr && level[t] == - 1; ++ql){

**int** u = Q[ql];

**for** (**int** e = last[u]; e != -1; e = next[e]){

**int** v = adj[e];

**if** (level[v] == -1 && RES(e) > 0){

level[v] = level[u] + 1;

Q[qr++] = v;

}

}

}

**return** level[t] != -1;

}

ll **dfs** (**int** u, ll curr) {

**if** (u == sink)

**return** curr;

**for** (**int** e = now[u]; e != -1; now[u] = e = next[e]){

**int** v = adj[e];

**if** (level[v] == level[u] + 1 && RES(e) > 0){

ll ans = dfs(v, min(curr, RES(e)));

**if** (ans > 0){

flow[e] += ans;

flow[e ^ 1] -= ans;

**return** ans;

}

}

}

**return** 0;

}

ll **max\_flow** (){

ll ans = 0, res;

**while** (bfs(source, sink)){

**memcpy**(now, last, **sizeof** now);

**while** ((res = dfs(source, oo)) > 0)

ans += res;

}

**return** ans;

}

**void** **init**(**int** \_source, **int** \_sink ){

source = \_source;

sink = \_sink;

**memset**(last , -1 , **sizeof** last);

E = 0;

}

**namespace** MCF {

**typedef** **double** value;

**typedef** pair<**int**, value> piv;

**const** **int** N = 200000, MAXE = 1000000;

**const** **int** oo = 1 << 29;

**const** value cost\_oo = 1e40; //change case int cost to 1 << 29

**int** E, nodes;

**int** last[N], head[MAXE], pprev[MAXE];

**int** flow[MAXE], cap[MAXE]; value cost[MAXE];

value prio[N]; **int** curflow[N], edge[N], st[N], q[N];

**void** clear(**int** \_nodes) {

fill(last, last + N, -1);

E = 0;

nodes = \_nodes;

}

**void** add(**int** u, **int** v, **int** \_cap, value \_cost) {

head[E] = v;

cap[E] = \_cap;

cost[E] = \_cost;

flow[E] = 0;

pprev[E] = last[u];

last[u] = E++;

}

**void** add\_edge(**int** u, **int** v, **int** cap, value cost) {

add(u, v, cap, cost);

add(v, u, 0, -cost);

}

piv solve(**int** s, **int** t) {

**int** f = 0; value flowCost = 0;

**while** (**true**) {

curflow[s] = oo;

fill(st, st + nodes, 2);

fill(prio, prio + nodes, cost\_oo);

prio[s] = 0;

**int** qh = 0, qt = 0;

q[qt++] = s;

**if** (qt == nodes)qt = 0;

**while** (qh != qt) {

**int** u = q[qh++];

**if** (qh == nodes)qh = 0;

st[u] = 0;

**for** (**int** e = last[u]; e >= 0; e = pprev[e]) {

**int** v = head[e];

value nprio = prio[u] + cost[e];

**if** (cap[e] > flow[e] && prio[v] > nprio + 1e-9) { //remember +1e-9 here case double costs

prio[v] = nprio;

edge[v] = e;

curflow[v] = min(curflow[u], cap[e] - flow[e]);

**if** (st[v] == 2) {

q[qt++] = v;

**if** (qt == nodes)qt = 0;

} **else** **if** (st[v] == 0) {

**if** (--qh == -1)qh = nodes - 1;

q[qh] = v;

}

st[v] = 1;

}

}

}

**if** (prio[t] == cost\_oo)

**break**;

**int** df = curflow[t];

f += df;

flowCost += df \* prio[t];

**for** (**int** v = t; v != s; v = head[edge[v] ^ 1]) {

flow[edge[v]] += df;

flow[edge[v] ^ 1] -= df;

}

}

**return** {f, flowCost};

}

};

**namespace** MaxMatching

{

**const** **int** MaxV = 1001;

**int** V, E, match[MaxV], head, tail, Q[MaxV];

**int** start, finish, newbase, queue[MaxV];

**int** father[MaxV], base[MaxV];

**bool** graph[MaxV][MaxV], inpath[MaxV];

**bool** inblossom[MaxV], inqueue[MaxV];

**void** initialize(**int** \_\_nodes){

V = \_\_nodes;

memset(graph, **false**, **sizeof**(graph));

}

**void** addEdge(**int** u, **int** v){

graph[u][v] = **true**;

graph[v][u] = **true**;

}

**void** push(**int** u){

Q[tail++] = u;

inqueue[u] = **true**;

}

**int** pop(){

**return** Q[head++];

}

**int** findCommonAncestor(**int** u, **int** v){

memset(inpath, 0, **sizeof**(inpath));

**while** (**true**) {

u = base[u];

inpath[u] = **true**;

**if** (u == start)

**break**;

u = father[ match[u] ];

}

**while** (**true**) {

v = base[v];

**if** (inpath[v])

**break**;

v = father[ match[v] ];

}

**return** v;

}

**void** resetTrace(**int** u){

**while** (base[u] != newbase) {

**int** v = match[u];

inblossom[ base[u] ]= **true**;

inblossom[ base[v] ]= **true**;

u = father[ v ];

**if** (base[u] != newbase)

father[u] = v;

}

}

**void** blossomContract(**int** u, **int** v){

newbase = findCommonAncestor(u, v);

memset(inblossom, **false**, **sizeof**(inblossom));

resetTrace(u);

resetTrace(v);

**if** (base[u] != newbase) father[u]= v;

**if** (base[v] != newbase) father[v]= u;

**for** (**int** i = 1; i <= V; i++)

**if** (inblossom[ base[i] ]) {

base[ i ] = newbase;

**if** (!inqueue[i])

push(i);

}

}

**void** find\_augmenting\_path(){

memset(inqueue, **false**, **sizeof**(inqueue));

memset(father, 0, **sizeof**(father));

**for** (**int** i = 1; i <= V; i++)

base[i] = i;

head = 0;

tail = 0;

push(start);

finish = 0;

**while** (head < tail){

**int** u = pop();

**for** (**int** v = 1; v <= V; v++)

**if** (graph[u][v] && (base[u] != base[v]) && (match[u] != v)) {

**if** ((v == start) || ((match[v] > 0) && (father[match[v]] > 0))){

blossomContract(u, v);

**continue**;

}

**if** (father[v] == 0) {

father[v] = u;

**if** (match[v] > 0)

push(match[v]);

**else** {

finish = v;

**return**;

}

}

}

}

}

**void** augment\_path(){

**int** u = finish;

**while** (u > 0) {

**int** v = father[u];

**int** w = match[v];

match[v] = u;

match[u] = v;

u = w;

}

}

**void** edmonds(){

memset(match, 0, **sizeof**(match));

**for** (**int** i = 1; i <= V; i++)

**if** (!match[ i ]) {

start = i;

find\_augmenting\_path( );

**if** (finish > 0)

augment\_path();

}

}

};

**namespace** MinCut{

**typedef** vector<**int**> vi;

**typedef** vector<vi> vvi;

**const** ll oo = 1ll<<55;

//(min cut value, nodes in half of min cut)

pair<**int**, vi> StoerWagner(vvi &weights) { // undirected graphs.. O(V^3)

**int** N = weights.size();

vi used(N), cut, best\_cut;

**int** best\_weight = -1;

**for** (**int** phase = N-1; phase >= 0; phase--) {

vi w = weights[0];

vi added = used;

**int** prev, last = 0;

**for** (**int** i = 0; i < phase; i++) {

prev = last;

last = -1;

**for** (**int** j = 1; j < N; j++)

**if** (!added[j] && (last == -1 || w[j] > w[last])) last = j;

**if** (i == phase-1) {

**for** (**int** j = 0; j < N; j++) weights[prev][j] += weights[last][j];

**for** (**int** j = 0; j < N; j++) weights[j][prev] = weights[prev][j];

used[last] = **true**;

cut.push\_back(last);

**if** (best\_weight == -1 || w[last] < best\_weight) {

best\_cut = cut;

best\_weight = w[last];

}

} **else** {

**for** (**int** j = 0; j < N; j++)

w[j] += weights[last][j];

added[last] = **true**;

}

}

}

**return** make\_pair(best\_weight, best\_cut);

}

pair<**int**, vi> Bruteforce(vvi &weights) {

**int** N = weights.size();

**int** best\_weight = -1;

vi best\_cut;

REP(msk, (1<<N)-1) **if** (msk>0) {

**int** curr = 0;

REP(I, N) REP(K, N) **if** (((msk&(1<<I))>0) != ((msk&(1<<K))>0))

curr += weights[I][K];

**if** (best\_weight == -1 || curr<best\_weight){

best\_weight = curr;

best\_cut.clear();

REP(I, N) **if** (msk&(1<<I)) best\_cut.PB(I);

}

}

**return** make\_pair(best\_weight/2, best\_cut);

}

}

**const** **int** mod = 1e9+7

**const** **int** MAXN = 1 << 19;

**namespace** FFT {

**#define** FOR(i, a, b) **for** (**int** i = (a); i < (b); ++i)

**#define** REP(i, n) FOR(i, 0, n)

**typedef** ll value;

**typedef** complex<**double**> comp;

**const** **double** pi = 2 \* acos(0);

**int** N;

comp omega[MAXN];

comp a1[MAXN], a2[MAXN];

comp z1[MAXN], z2[MAXN];

value x[MAXN], y[MAXN], z[MAXN];

**void** fft(comp \*a, comp \*z, **int** m = N) {

**if** (m == 1) {

z[0] = a[0];

} **else** {

**int** s = N/m;

m /= 2;

fft(a, z, m);

fft(a+s, z+m, m);

REP(i, m) {

comp c = omega[s\*i] \* z[m+i];

z[m+i] = z[i] - c;

z[i] += c;

}

}

}

**void** mult(value \*a, value \*b, value \*c, **int** len) {

N = 2\*len;

**while** (N & (N-1)) ++N;

assert(N <= MAXN);

REP(i, N) a1[i] = 0;

REP(i, N) a2[i] = 0;

REP(i, len) a1[i] = a[i];

REP(i, len) a2[i] = b[i];

REP(i, N) omega[i] = polar(1.0, 2\*pi/N\*i);

fft(a1, z1, N);

fft(a2, z2, N);

REP(i, N) omega[i] = comp(1, 0) / omega[i];

REP(i, N) a1[i] = z1[i] \* z2[i] / comp(N, 0);

fft(a1, z1, N);

REP(i, 2\*len) c[i] = round(z1[i].real());

}

**void** mult2(vector<**int**>&a, vector<**int**>&b, vector<**int**>&c) {

// K = sqrt(mod)

**const** **int** K = 1 << 15;

**int** len = max(a.size() , b.size());

REP(i , 2 \* len)x[i] = y[i] = z[i] = 0;

c.resize(2 \* len);

REP(i, 2\*len) c[i] = 0;

REP(ca, 2) REP(cb, 2) {

REP(i, len) {

x[i] = ca ? a[i] / K : a[i] % K;

y[i] = cb ? b[i] / K : b[i] % K;

}

mult(x, y, z, len);

**int** k = mul(ca ? K : 1, cb ? K : 1);

REP(i, 2\*len)

add(c[i], mul(k, z[i] % mod));

}

}

}

**namespace** NTT{

**const** **int** MOD = (7\*17 << 23) + 1; // this choice is an important one

**const** **int** ROOT = 3; // where does this comes from ???

**int** modPower(**int** a, **int** n) {

**int** res = 1;

**for**(; n > 0; n >>= 1) {

**if** ((n & 1) == 1) res = 1LL \* res \* a % MOD; // \* / and % have the same precedence and are left associative

a = 1LL \* a \* a % MOD;

}

**return** res;

}

**int** modInverse(**int** num) {

**return** modPower(num, MOD - 2);

}

vector<**int**> fft(vector<**int**> a, **bool** invert) {

**int** n = a.size();

assert((n & (n - 1)) == 0);

**int** lg = 0, root = modPower(ROOT, (MOD - 1) / n);

**if**(invert) root = modInverse(root);

**while**((1 << lg) != n) ++lg;

**for**(**int** i = 0; i < n; ++i) {

**int** j = 0;

**for**(**int** p = 0; p < lg; ++p)

**if**((i & 1 << p) != 0)

j |= 1 << (lg - p - 1);

**if**(i < j) swap(a[i], a[j]);

}

**for**(**int** len = 2; len <= n; len \*= 2) {

**int** wlen = modPower(ROOT, (MOD - 1) / len);

**if**(invert) wlen = modInverse(wlen);

**for**(**int** i = 0; i < n; i += len) {

**int** w = 1;

**for**(**int** j = 0; j < len / 2; ++j) {

**int** u = a[i + j], v = 1LL \* a[i + j + len / 2] \* w % MOD;

a[i + j] = (u + v) % MOD;

a[i + j + len / 2] = (u - v + MOD) % MOD;

w = 1LL \* w \* wlen % MOD;

}

}

}

**if**(invert) {

**int** mul = modInverse(n);

**for**(**int** i = 0; i < n; ++i) a[i] = 1LL \* a[i] \* mul % MOD;

}

**return** a;

}

vector<**int**> polyMultiply(vector<**int**> a, vector<**int**> b) {

**int** n = 2 \* max(a.size(), b.size());

assert((n & (n - 1)) == 0);

a.resize(n); a = fft(a, **false**);

b.resize(n); b = fft(b, **false**);

**for**(**int** i = 0; i < n; ++i) a[i] = 1LL \* a[i] \* b[i] % MOD;

a = fft(a, **true**);

**return** a;

}

vector<**int**> polyInverse(**const** vector<**int**> &a) {

**int** n = a.size();

assert(a[0] != 0);

assert((n & (n - 1)) == 0);

vector<**int**> r (1, modInverse(a[0]));

**for**(**int** len = 2; len <= n; len \*= 2) {

vector<**int**> nr = polyMultiply(polyMultiply(r, r), vector<**int**>(a.begin(), a.begin() + len));

nr.resize(len);

**for**(**int** i = 0; i < len; ++i) nr[i] = (MOD - nr[i]) % MOD;

**for**(**int** i = 0; i < len / 2; ++i) nr[i] = (2LL \* r[i] + nr[i]) % MOD;

r = nr;

}

**return** r;

}

vector<**int**> polySqrt(**const** vector<**int**> &a) {

assert(a[0] == 1);

**int** n = a.size();

assert((n & (n - 1)) == 0);

vector<**int**> r (1, 1);

**for**(**int** len = 2; len <= n; len \*= 2) {

r.resize(len);

vector<**int**> nr = polyMultiply(polyInverse(r), vector<**int**>(a.begin(), a.begin() + len));

nr.resize(len);

**for**(**int** i = 0; i < len / 2; ++i) nr[i] = (nr[i] + r[i]) % MOD;

**for**(**int** i = 0; i < len; ++i) nr[i] = nr[i] \* (MOD + 1LL) / 2 % MOD;

r = nr;

}

**return** r;

}

};

// 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 to store the optimal solution

//

// OUTPUT: value of the optimal solution (inf if unbounded

// above, -inf if infeasible)

**typedef** **double** DOUBLE;

**typedef** vector<DOUBLE> vec;

**typedef** vector<vec> mat;

**typedef** vector<**int**> vi;

**const** DOUBLE eps = 1e-9;

**const** DOUBLE oo = numeric\_limits<DOUBLE>::*infinity*();

**struct** LPSolver {

**int** m, n;

vi B, N;

mat D;

**LPSolver**(**const** mat &A, **const** vec &b, **const** vec &c) :

m(b.size()), n(c.size()), N(n + 1), B(m), D(m + 2, vec(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**(vec &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** -oo;

**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** oo;

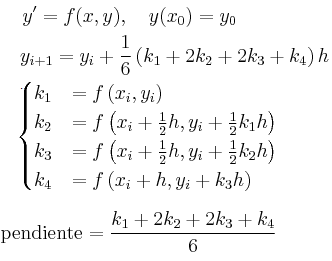
x = vec(n);

**for** (**int** i = 0; i < m; i++) **if** (B[i] < n) x[B[i]] = D[i][n + 1];

**return** D[m][n + 1];

}

};



// Assume F' = f

// input: interval [a,b] and a function f

// ouput: F(b)-F(a)

**inline** **int** **cmp**(**double** x, **double** y=0, **double** tol=EPS){

**return** (x<=y+tol) ? (x+tol<y) ? -1 : 0 : 1;

}

**long** **double** **romberg**(**int** a, **int** b, **double**(\*func)(**double**)) {

**long** **double** approx[2][50];

**long** **double** \*cur=approx[1], \*prev=approx[0];

prev[0] = 1/2.0 \* (b-a) \* (func(a) + func(b));

**for**(**int** it = 1; it < 25; ++it, swap(cur, prev)) {

**if**(it > 1 && cmp(prev[it-1], prev[it-2]) == 0)

**return** prev[it-1];

cur[0] = 1/2.0 \* prev[0];

**long** **double** div = (b-a)/pow(2, it);

**for**(**long** **double** sample = a + div; sample < b; sample += 2 \* div)

cur[0] += div \* func(a + sample);

**for**(**int** j = 1; j <= it; ++j)

cur[j] = cur[j-1] + 1/(pow(4, it) - 1)\*(cur[j-1] + prev[j-1]);

}

**return** prev[24];

}

Integral de f(x) de [a,b] == (b-a) \* (f(a) + 4\* f( (a + b) / 2) + f(b) ) / 6

The volume of the solid formed by rotating the area between the curves of f(x) and g(x) and the lines x=a and x=b about the x-axis is given by

V = pi \* integral[a , b] de |f^2(x) - g^2(x)|

If g(x) = 0 (e.g. revolving an area between curve and x-axis), this reduces to:

V = pi \* integral[a , b] de f^2(x)

The volume of the solid formed by rotating the area between the curves of f(x) and g(x) and the lines x=a and x=b about the y-axis is given by:

V = 2 \* pi \* integral[a , b] de x \* |f(x) - g(x)|

If g(x) = 0 (e.g. revolving an area between curve and x-axis), this reduces to:

V = 2 \* pi \* integral[a , b] de x \* f(x)

**int** gauss (vector < vector<**double**> > a, vector<**double**> & ans) {

**int** n = (**int**) a.size();

**int** m = (**int**) a[0].size() - 1;

vector<**int**> where (m, -1);

**for** (**int** col=0, row=0; col<m && row<n; ++col) {

**int** sel = row;

**for** (**int** i=row; i<n; ++i)

**if** (abs (a[i][col]) > abs (a[sel][col]))

sel = i;

**if** (abs (a[sel][col]) < EPS)

**continue**;

**for** (**int** i=col; i<=m; ++i)

swap (a[sel][i], a[row][i]);

where[col] = row;

// consider dividing all row here ot get identity

**for** (**int** i=0; i<n; ++i)

**if** (i != row) {

**double** c = a[i][col] / a[row][col];

**for** (**int** j=col; j<=m; ++j)

a[i][j] -= a[row][j] \* c;

}

++row;

}

ans.assign (m, 0);

**for** (**int** i=0; i<m; ++i)

**if** (where[i] != -1)

ans[i] = a[where[i]][m] / a[where[i]][i];

**for** (**int** i=0; i<n; ++i) {

**double** sum = 0;

**for** (**int** j=0; j<m; ++j)

sum += ans[j] \* a[i][j];

**if** (abs (sum - a[i][m]) > EPS)

**return** 0;

}

**for** (**int** i=0; i<m; ++i)

**if** (where[i] == -1)

**return** INF;

**return** 1;

}

**int** gauss(vector<bitset<N>> a, **int** n, **int** m, bitset<N> &ans) {

vector<**int**> where (m, -1);

**for** (**int** col=0, row=0; col<m && row<n; ++col) {

**for** (**int** i=row; i<n; ++i)

**if** (a[i][col]) {

swap (a[i], a[row]);

**break**;

}

**if** (! a[row][col])

**continue**;

where[col] = row;

**for** (**int** i=0; i<n; ++i)

**if** (i != row && a[i][col])

a[i] ^= a[row];

++row;

}

// The rest of implementation is the same as above

}

**const** **double** EPS = 1E-9;

**int** n;

vector < vector<**double**> > a (n, vector<**double**> (n));

**double** det = 1;

**for** (**int** i=0; i<n; ++i) {

**int** k = i;

**for** (**int** j=i+1; j<n; ++j)

**if** (abs (a[j][i]) > abs (a[k][i]))

k = j;

**if** (abs (a[k][i]) < EPS) {

det = 0;

**break**;

}

swap (a[i], a[k]);

**if** (i != k)

det = -det;

det \*= a[i][i];

**for** (**int** j=i+1; j<n; ++j)

a[i][j] /= a[i][i];

**for** (**int** j=0; j<n; ++j)

**if** (j != i && abs (a[j][i]) > EPS)

**for** (**int** k=i+1; k<n; ++k)

a[j][k] -= a[i][k] \* a[j][i];

}

**const** **double** EPS = 5e-7;

**const** **double** phi=(3.0-**sqrt**(5.0))/2.0;

**#define** MAX 100100

**int** N;

**double** x[MAX], v[MAX], lo, hi, a, b, ax, bx;

**double** **predicate**(**double** t){

}

**int** **main**(){

lo = 0;

hi = 1e9;

a = lo + (hi-lo)\*phi;

b = hi - (hi-lo)\*phi;

ax = calc(a);

bx = calc(b);

**for**(**int** it=0; it<80; it++){

**if** (ax<bx) hi = b, b = a, a = lo+(hi-lo)\*phi, bx = ax, ax = **predicate** (a);

**else** lo = a, a = b, b = hi-(hi-lo)\*phi, ax = bx, bx = **predicate** (b);

}

**printf**("%.2lf\n", calc(lo) );

}

**namespace** PersistentSegmentTree{

**const** **int** MAXN = 3e5 + 5, MAXK = 1e5 + 5, MAXS = 4 \* MAXK \* 20;

**int** n, k, p, a, b, root[MAXN];

**int** lch[MAXS], rch[MAXS], tree[MAXS], sz;

**int** new\_node (**int** val){

assert(sz < MAXS);

tree[sz] = val;

**return** sz++;

}

**int** new\_node (**int** l, **int** r){

lch[sz] = l;

rch[sz] = r;

**return** new\_node(tree[l] + tree[r]);

}

**int** build (**int** b, **int** e){

**if** (b == e)

**return** new\_node(0);

**int** m = (b + e) / 2;

**return** new\_node(build(b, m), build(m + 1, e));

}

**int** update (**int** v, **int** b, **int** e, **int** p){

**if** (b > p || e < p)

**return** v;

**if** (b == e)

**return** new\_node(tree[v] + 1);

**int** m = (b + e) / 2;

**return** new\_node(update(lch[v], b, m, p), update(rch[v], m + 1, e, p));

}

**int** query (**int** r1, **int** r2, **int** k){

**int** b = 0, e = ::k;

**while** (b < e){

**int** m = (b + e) / 2;

**int** aux = tree[lch[r2]] - tree[lch[r1]];

**if** (aux > k){

r1 = lch[r1];

r2 = lch[r2];

e = m;

}

**else**{

r1 = rch[r1];

r2 = rch[r2];

b = m + 1;

k -= aux;

}

}

**return** b;

}

}

**typedef** **int** node;

**const** **int** MAXN = 2e5 + 5, MAXS = 8e6, NIL = 0, oo = 1e9 + 5;

node lch[MAXS], rch[MAXS], root;

**int** val[MAXS], //val[i] = value of array at corresponding index to node i

cnt[MAXS], //cnt[i] = gets the number of nodes in the subtree rooted at node i

rev[MAXS], //rev[i] = says if the node i is reversed

acc[MAXS], //acc[i] = accumulated value for sum in all the subtree rooted at node i

sum[MAXS], //sum[i] = gets the sum of values of all nodes in the subtree rooted at node i

mim[MAXS], //mim[i] = gets the minimum value of all nodes in the subreee rooted at node i

arr[MAXN], //arr[i] = value of initial array at index i

sz;

**int** **random** () { **return** (**rand**() << 15) + **rand**(); }

**void** **init** (){

sz = 1; //important!!!

}

**int** **get\_cnt** (node t) { **return** t != NIL ? cnt[t] : 0; }

**int** **get\_sum** (node t) { **return** t != NIL ? sum[t] : 0; }

**int** **get\_max** (node t) { **return** t != NIL ? mim[t] : -99999999; }

**void** **push** (node t){

**if** (t != NIL) {

**if** (rev[t]){ //handling reverse query

rev[t] = **false**;

**if** (lch[t] != NIL)

rev[lch[t]] ^= 1;

**if** (rch[t] != NIL)

rev[rch[t]] ^= 1;

swap(lch[t], rch[t]);

}

**if** (acc[t]){//handling interval sum query

val[t] += acc[t];

**if** (lch[t] != NIL)

acc[lch[t]] += acc[t];

**if** (rch[t] != NIL)

acc[rch[t]] += acc[t];

acc[t] = 0;

}

}

}

**void** **debug2** (node t){

**if** (t != NIL) {

push(t);

debug2(lch[t]);

cerr << val[t] << " ";

debug2(rch[t]);

}

}

**void** **debug** (){

cerr << "debugging" << endl;

debug2(root);

cerr << endl;

}

**void** **debug** (node t){

cerr << "debugging" << endl;

debug2(t);

cerr << endl;

}

**void** **update** (node t){

**if** (t != NIL) {

cnt[t] = get\_cnt(lch[t]) + get\_cnt(rch[t]) + 1;

sum[t] = get\_sum(lch[t]) + get\_sum(rch[t]) + val[t];

**int** mc = max(get\_max(lch[t]), get\_max(rch[t]));

mim[t] = max(mc, val[t]);

}

}

node **new\_node** (**int** v, node l, node r){

assert(sz < MAXS);

val[sz] = v;

mim[sz] = v;

lch[sz] = l;

rch[sz] = r;

update(sz);

**return** sz++;

}

//WARNING: Call root = build(0, n - 1)

node **build** (**int** b, **int** e){

**if** (b > e)

**return** NIL;

**int** m = (b + e) >> 1;

**return** new\_node(arr[m], build(b, m - 1), build(m + 1, e));

}

**void** **split** (node t, node &l, node &r, **int** cnt){

**if** (t == NIL)

l = r = NIL;

**else** {

push(t);

**int** tmp = get\_cnt(lch[t]);

**if** (tmp >= cnt) {

split(lch[t], l, lch[t], cnt);

r = t;

}

**else** {

split(rch[t], rch[t], r, cnt - tmp - 1);

l = t;

}

update(t);

}

}

node **merge** (node l, node r){

**if** (l == NIL || r == NIL)

**return** l == NIL ? r : l;

push(l);

push(r);

**if** (random() % (cnt[l] + cnt[r]) < cnt[l]) {

rch[l] = merge(rch[l], r);

update(l);

**return** l;

}

lch[r] = merge(l, lch[r]);

update(r);

**return** r;

}

**void** **insert** (**int** idx, **int** val){

**if** (idx > get\_cnt(root))

**return**;

node n1, n2;

split(root, n1, n2, idx);

root = merge(n1, new\_node(val, NIL, NIL));

root = merge(root, n2);

}

**void** **erase** (**int** idx){

**if** (get\_cnt(root) <= idx)

**return**;

node n1, n2, n3;

split(root, n1, n2, idx);

split(n2, n2, n3, 1);

root = merge(n1, n3);

}

**void** **reverse** (**int** l, **int** r){

node n1, n2, n3;

split(root, n1, n2, l);

split(n2, n2, n3, r - l + 1);

rev[n2] ^= 1;

root = merge(n1, n2);

root = merge(root, n3);

}

//Rotate arr[l...r] k times to the right

**void** **rotate** (**int** l, **int** r, **int** k){

**if** (l > r)

**return**;

node n1, n2, n3, n4;

split(root, n1, n2, l);

split(n2, n2, n3, r - l + 1);

k %= get\_cnt(n2);

split(n2, n2, n4, get\_cnt(n2) - k);

root = merge(n1, n4);

root = merge(root, n2);

root = merge(root, n3);

}

**int** **sum\_query** (**int** l, **int** r){

node n1, n2, n3;

split(root, n1, n2, l);

split(n2, n2, n3, r - l + 1);

**int** ans = get\_sum(n2);

root = merge(n1, n2);

root = merge(root, n3);

**return** ans;

}

**void** **sum\_query** (**int** l, **int** r, **int** v){

node n1, n2, n3;

split(root, n1, n2, l);

split(n2, n2, n3, r - l + 1);

acc[n2] += v;

root = merge(n1, n2);

root = merge(root, n3);

}

**int** **max\_query** (**int** l, **int** r){

node n1, n2, n3;

split(root, n1, n2, l);

split(n2, n2, n3, r - l + 1);

**int** ans = get\_max(n2);

root = merge(n1, n2);

root = merge(root, n3);

**return** ans;

}

//!!!!always do

//namespace treap

//treap::init();

//fill treap::arr[]

//treap::root = treap::build(0, N - 1);

**namespace** Treap{

**int** random () { **return** (rand() << 15) + rand(); }

**const** **int** MAXN = 2e5, MAXS = 8e6;

**int** A[MAXN]; //input

**int** val[MAXS], cnt[MAXS], lch[MAXS], rch[MAXS], num = 1; // 0 used as null node

**void** update (**int** t){

**if** (t){

cnt[t] = cnt[lch[t]] + cnt[rch[t]] + 1;

}

}

**int** new\_node (**int** v, **int** l, **int** r){

assert(num < MAXS);

val[num] = v;

lch[num] = l; rch[num] = r;

update(num);

**return** num++;

}

**int** build (**int** b, **int** e){

**if** (b > e)

**return** 0;

**int** m = (b + e) / 2;

**return** new\_node(A[m], build(b, m - 1), build(m + 1, e));

}

**int** build (**int** n){

sort(A, A+n);

build(0, n-1);

}

// elements on l are < than v..

**void** split (**int** t, **int** v, **int** &l, **int** &r){

**if** (t==0)

l = r = 0;

**else**{

**if** (val[t] >= v){

split(lch[t], v, l, r);

r = new\_node(val[t], r, rch[t]);

}

**else**{

split(rch[t], v, l, r);

l = new\_node(val[t], lch[t], l);

}

}

}

**int** merge (**int** l, **int** r){

**if** (l==0 || r==0)

**return** l ? l : r;

**if** (random() % (cnt[l] + cnt[r]) < cnt[l])

**return** new\_node(val[l], lch[l], merge(rch[l], r));

**return** new\_node(val[r], merge(l, lch[r]), rch[r]);

}

**bool** find (**int** t, **int** v){

**int** l, m, r;

split(t, v, l, r);

split(r, v+1, m, r);

**return** cnt[m]>0;

}

**int** insert (**int** t, **int** v){

**int** l, m, r;

split(t, v, l, r);

m = new\_node(v, 0, 0);

r = merge(m, r);

**return** merge(l, r);

}

**int** erase (**int** t, **int** v){

**int** l, m, r;

split(t, v, l, r);

split(r, v+1, m, r);

**return** merge(l, r);

}

**int** erase\_one (**int** t, **int** v){

**int** l, m, r;

split(t, v, l, r);

split(r, v+1, m, r);

m = merge( lch[m], rch[m] );

r = merge(m, r);

**return** merge(l, r);

}

**int** lowest (**int** t){

assert(t);

**if** (!lch[t]) **return** val[t];

**else** **return** lowest(lch[t]);

}

**int** lower\_bound (**int** t, **int** v){

**int** l, m, r;

split(t, v, l, r);

**if** (cnt[r]==0) **return** -1;

**return** lowest( r );

}

};

**#include** <ext/pb\_ds/assoc\_container.hpp>

**#include** <ext/pb\_ds/tree\_policy.hpp>

**using** **namespace** std;

**using** **namespace** \_\_gnu\_pbds;

**typedef** **long** **long** ll;

**typedef** tree<**int**,null\_mapped\_type,less<**int**>,rb\_tree\_tag,tree\_order\_statistics\_node\_update> mset;

**typedef** tree<**int**,**int**,less<**int**>,rb\_tree\_tag,tree\_order\_statistics\_node\_update> mmap;

mset S;

mmap M;

S.insert(I), M[I]=I+1; \*S.find\_by\_order(I) S.order\_of\_key(I) M.find\_by\_order(I)->second M.order\_of\_key(I)

**const** ll is\_query = -(1LL<<62);

**struct** Line {

ll m, b;

**mutable** function<**const** Line\*()> succ;

**bool** **operator<**(**const** Line& rhs) **const** {

**if** (rhs.b != is\_query) **return** m < rhs.m;

**const** Line\* s = succ();

**if** (!s) **return** 0;

ll x = rhs.m;

**return** b - s->b < (s->m - m) \* x;

}

};

**struct** HullDynamic : **public** multiset<Line> { // will maintain upper hull for maximum

**bool** **bad**(iterator y) {

**auto** z = next(y);

**if** (y == begin()) {

**if** (z == end()) **return** 0;

**return** y->m == z->m && y->b <= z->b;

}

**auto** x = prev(y);

**if** (z == end()) **return** y->m == x->m && y->b <= x->b;

**return** (x->b - y->b)\*(z->m - y->m) >= (y->b - z->b)\*(y->m - x->m);

}

**void** **insert\_line**(ll m, ll b) {

**auto** y = insert({ m, b });

y->succ = [=] { **return** next(y) == end() ? 0 : &\*next(y); };

**if** (bad(y)) { erase(y); **return**; }

**while** (next(y) != end() && bad(next(y))) erase(next(y));

**while** (y != begin() && bad(prev(y))) erase(prev(y));

}

ll **eval**(ll x) {

**auto** l = \*lower\_bound((Line) { x, is\_query });

**return** l.m \* x + l.b;

}

};

**#define** MAX 200100

**#define** MAX4 600100

**#define** MAXLOG 19

**#define** MOD 1000000007

**int** N, Q, u, v, dd, c, root;

vector<**int**> E[MAX], curr;

**bool** used[MAX];

**int** C[MAX], H[MAX], D[MAX];

**int** P[MAX]; // size, size of bigger child, depth in the virtual tree, parent in the virtual tree

vector<vector<pii>> X[MAX], SX[MAX];

vector<pii> Y[MAX], SY[MAX];

**int** pos[MAX], sz[MAX]; //position on parent's list, size of parent's list (for the VT)

//int tmp[MAX], DP[MAX]; // distance to father in VT (not useful at all)

**int** getsz(**int** n, **int** p){

C[n] = 1;

H[n] = 0;

curr.PB(n);

foreach(it, E[n]) **if**(\*it!=p && !used[\*it]){

//tmp[\*it] = tmp[n]+1; // to calc DP

getsz(\*it, n);

C[n] += C[\*it];

H[n] = max(H[n], C[\*it]);

}

}

**int** centroid(**int** n){

curr.clear();

//tmp[n] = 0;

getsz(n, -1);

foreach(it, curr){

H[\*it] = max(H[\*it], C[n]-C[\*it]);

**if** (H[\*it]\*2 <= C[n])

**return** \*it;

}

assert(**false**);

}

**int** init(**int** n, **int** d){

**int** r = centroid(n);

D[r] = d;

//if (d) DP[r] = tmp[r]+1;

used[r] = **true**;

foreach(it, E[r]) **if** (!used[\*it]){

**int** c = init(\*it, d+1);

P[c] = r;

pos[c] = sz[r]++;

}

**return** r;

}

**void** inc(ll &x, ll y){

x = (((x+y)%MOD)+MOD)%MOD;

}

**int** first[MAX], lcurr;

**int** mlog[MAX4], LQ, RMQ[MAXLOG][MAX4];

**void** DFS(**int** n, **int** p, **int** d){

first[n]=lcurr;

RMQ[0][lcurr++]= d;

D[n] = d;

foreach(it, E[n])

**if** (\*it != p){

DFS(\*it, n, d+1);

RMQ[0][lcurr++]=d;

}

}

**int** dst(**int** u, **int** v){

**int** fu=first[u];

**int** fv=first[v];

**if** (fu>fv) swap(fu, fv);

LQ = mlog[fv - fu + 1];

**return** D[u] + D[v] - 2\*min(RMQ[LQ][fu], RMQ[LQ][fv-(1<<LQ)+1]);

}

**void** update(**int** n, **int** p){

**int** d = dd-dst(n, u);

**if** (d<0) **return**; // this is a HUGE bug

//cout << n << " " << u << " " << dd << " " << dst(0, 0) << " " << d << endl;

Y[n].PB(pii(d, c));

**if** (p>=0) X[n][p].PB(pii(d, c));

//d -= DP[n]; huge error!!

**if** (n != root)

update(P[n], pos[n]);

}

vector<pii>::iterator it;

ll query(**int** n, **int** p){

**int** d = dst(n, u);

ll ret = 0;

it = lower\_bound(ALL(SY[n]), pii(d,-1));

**if** (it!=SY[n].end()) inc(ret, it->second);

**if** (p>=0) {

it = lower\_bound(ALL(SX[n][p]), pii(d,-1));

**if** (it!=SX[n][p].end()) inc(ret, -it->second);

}

//d += DP[n]; huge error!!

**if** (n != root)

inc(ret, query(P[n], pos[n]));

**return** ret;

}

**void** prepare(){

REP(I, N){

SX[I].clear();

REP(K, SZ(X[I])) {

sort(ALL(X[I][K]));

SX[I].PB(X[I][K]);

**for**(**int** L=SZ(SX[I][K])-2; L>=0; L--) inc(SX[I][K][L].second, SX[I][K][L+1].second);

}

sort(ALL(Y[I]));

SY[I] = Y[I];

**for**(**int** L=SZ(SY[I])-2; L>=0; L--) inc(SY[I][L].second, SY[I][L+1].second);

//REP(K, SZ(X[I])) X[I][K].clear();

//X[I].clear();

//Y[I].clear();

}

}

**int** main(){

ios\_base::sync\_with\_stdio(0);

cin.tie(0);

cin >> N >> Q;

REP(I, N-1){

cin >> u >> v;

u--, v--;

E[u].PB(v);

E[v].PB(u);

}

root = init(0, -1);

P[root] = -1;

memset(D, 0, **sizeof**(D));

// preparations

DFS(0, -1, 0);

**for**(**int** L=0; L<MAXLOG-1; L++)

**for**(**int** I=0; I<lcurr; I++) {

RMQ[L+1][I] = RMQ[L][I];

**if** (I+(1<<L) < lcurr)

RMQ[L+1][I] = min( RMQ[L][I], RMQ[L][I+ (1<<L)] );

}

**for**(**int** I=2; I<MAX4; I++)

mlog[I]=mlog[I-1] + (1<<(mlog[I-1]+1) <= I);

//REP(I, N) REP(K, N) if (I<=K) cout<<I<<" "<<K<<" -> "<<dst(I, K)<<" "<<dst(K, I)<<endl;

//REP(I, N) cout<< D[I]<<" \n"[I+1==N];

//REP(I, N) cout<< P[I]<<" \n"[I+1==N];

//REP(I, N) cout<<DP[I]<<" \n"[I+1==N];

//cout << endl;

REP(I, N){

X[I].resize(sz[I]);

REP(K, SZ(X[I])) X[I][K].clear();

}

REP(q,Q){

cin >> u >> dd >> c;

u--;

update(u, -1);

//prepare();

//cout << query(3, -1) << endl;

//WAIT;

}

prepare();

REP(I, N) u=I, cout<<query(I, -1)<<" \n"[I==N];

}

**struct** Line {

ll a, b, get(ll x) { **return** a \* x + b; }

};

**struct** Hull {

**int** size;

Line \*hull;

**Hull**(**int** maxSize) {

hull = **new** Line[++maxSize], size = 0;

}

**bool** **is\_bad**(ll curr, ll prev, ll next) {

Line c = hull[curr], p = hull[prev], n = hull[next];

**return** (c.b - n.b) \* (c.a - p.a) <= (p.b - c.b) \* (n.a - c.a);

}

**void** **add\_line**(ll a, ll b) {

hull[size++] = (Line){a, b};

**while** (size > 2 && is\_bad(size - 2, size - 3, size - 1))

hull[size - 2] = hull[size - 1], size--;

}

ll **query**(ll x) {

**int** l = -1, r = size - 1;

**while** (r - l > 1) {

**int** m = (l + r) / 2;

**if** (hull[m].get(x) <= hull[m + 1].get(x))

l = m;

**else**

r = m;

}

**return** hull[r].get(x);

}

};

**void** dfs( **int** u , **bool** match ) {

mk[u] = **true**;

A[u] = **true**;

**for**(**int** e = MF::last[u]; e != -1; e = MF::next[e]) {

**int** v = MF::adj[e];

**if**(v == sink || v == source)**continue**;

**if**(!mk[v] && max(MF::flow[e], MF::flow[e ^ 1]) == match)

dfs(v, !match);

}

}

set<**int**> x, y;

//minimum vertex cover Z = (X/A) U (Y intersection A)

**for**(**int** e = MF::last[source]; e != -1; e = MF::next[e]) {

**int** v = MF::adj[e];

x.insert(v);// build x

**if**(!MF::flow[e])

dfs(v, 0);

}

//build y

**for**(**int** i = 0; i < n; i++ )**if**(color[i])y.insert(i);

// X/A

**for**(**int** i = 0; i < n; i++ )

**if**(A[i] && x.find(i) != x.end())x.erase(i);

//x will be the vertex cover

//adding to x (y interseccion A)

**for**( **auto** v : y )

**if**( A[v] && x.find(v) == x.end() ) x.insert(v);

// Independent set is the complement of minimum vertex cover

**for**( **int** i = 0; i < n; i++ ) {

**if**(x.find(i) == x.end())

cout << i + 1 << " ";

}

cout << endl;

**namespace** Polynomial{

**const** ll MAXN = 4200;

**const** ll MOD = 1000000007; //must be prime..

ll N;

ll P[MAXN]; // output.. 0-indexed

ll x[MAXN], y[MAXN]; // 0 indexed...

ll a[MAXN], g[2][MAXN], f[2][MAXN];

ll inv[MAXN\*2];

ll mod(ll n){ **return** ((n%MOD)+MOD)%MOD; }

**void** inc(ll &a, ll b){ a = mod(a+b); }

ll minv(ll a, ll m) { **return** a < 2 ? a : ((1 - m \* minv(m % a, a)) / a % m + m) % m; } // by bjin

**void** Interpolate(){

memset(a, 0, **sizeof**(a));

memset(g, 0, **sizeof**(g));

memset(P, 0, **sizeof**(P));

**for**(ll I=1; I<MAXN\*2; I++) inv[I]=minv(I, MOD);

REP(I, N) f[0][I] = y[I]; a[0]=f[0][0];

**int** curr = 1;

**for**(**int** len=2; len<=N; len++){

REP(K, N-len+1)

// I know in this case that x[K+len-1] > x[K]

f[curr][K] = mod(mod(f[1-curr][K+1] - f[1-curr][K]) \* inv[x[K+len-1]-x[K]]);

a[len-1] = f[curr][0];

curr = 1-curr;

}

g[0][0] = curr = 1;

inc(P[0], g[0][0]\*a[0]);

**for**(ll I=1; I<=N; I++){

memset(g[curr], 0, **sizeof**(g[curr]));

**for**(ll K=0; K<=N; K++) inc(g[curr][K], g[1-curr][K]\*(-x[I-1]));

**for**(ll K=1; K<=N; K++) inc(g[curr][K], g[1-curr][K-1]);

**for**(ll K=0; K<=I; K++) inc(P[K], g[curr][K]\*a[I]);

curr = 1-curr;

}

}

ll eval(ll x){

ll xp=1;

ll ret = 0;

REP(I, N){

inc(ret, P[I]\*xp);

xp = mod(xp\*x);

}

**return** ret;

}

}

**struct** obj

{

**char** symbol;

obj \* lft, \*rgt;

**bool** isLeaf;

}root;

**char** s[LEN];

**char** \*it = s;

obj \* **expr**();

obj \* **item**(){

**if**(\*it == '(')

{

it++;

obj \* res = expr();

it++;

**return** res;

}

**else**

{

**char** c = \*it;

it++;

obj \* res = **new** obj{ c , NULL , NULL, **true**};

**return** res;

}

}

obj \* **factor**(){

obj \* t = item();

**if**(\*it == '^'){

**char** c = \*it;

it++;

t = **new** obj{c, t , factor(), **false**};

}

**return** t;

}

obj \* **term**(){

obj \* t = factor();

**while**(\*it == '\*' || \*it == '/'){

**char** c = \*it;

it++;

t = **new** obj{ c , t , factor() , **false**};

}

**return** t;

}

obj \* **expr**(){

obj \* t = term();

**while**(\*it == '+' || \*it == '-'){

**char** c = \*it;

it++;

t = **new** obj{ c , t , term(), **false**};

}

**return** t;

}

**void** **print**(obj \* root){

**if**(root->isLeaf)

{

cout << root->symbol << endl;

**return**;

}

cout << root->symbol << endl;

print(root->lft);

print(root->rgt);

}

**static** **char** stdinBuffer[1024];

**static** **char**\* stdinDataEnd = stdinBuffer + **sizeof** (stdinBuffer);

**static** **const** **char**\* stdinPos = stdinDataEnd;

**void** readAhead(size\_t amount){

**int** remaining = stdinDataEnd - stdinPos; // size\_t is unsigned

**if** (remaining < amount) {

memmove(stdinBuffer, stdinPos, remaining);

size\_t sz = fread(stdinBuffer + remaining, 1, **sizeof** (stdinBuffer) - remaining, stdin);

stdinPos = stdinBuffer;

stdinDataEnd = stdinBuffer + remaining + sz;

**if** (stdinDataEnd != stdinBuffer + **sizeof** (stdinBuffer))

\*stdinDataEnd = 0;

}

}

// could be made ll on demand

**int** readInt() {

readAhead(16); // should guarantee more read bytes for ll, and if using untested 'SkipWhiteSpaces'

**int** x = 0; // ll goes here

**bool** neg = **false**;

**if** (\*stdinPos == '-') {

++stdinPos;

neg = **true**;

}

**while** (\*stdinPos >= '0' && \*stdinPos <= '9') {

x \*= 10;

x += \*stdinPos - '0';

++stdinPos;

}

**return** neg ? -x : x;

/\* SkipWhiteSpaces... untested

while (stdinPos < sizeof (stdinBuffer) && (\*stdinPos == ' ' || \*stdinPos == '\r' || \*stdinPos == '\t' || \*stdinPos == '\n') )

stdinPos++;

\*/

}

**struct** Intervals{

**typedef** pair<ll, ll> pii;

pii curr;

set<pii> S;

set<pii>::iterator it;

**void** print(pii p){ cout<<"["<<p.xx<<","<<p.yy<<"]"<<endl; }

**void** printall(){

cout << "-----------------ALL-------------------" << endl;

foreach(it, S) print(\*it);

cout << "--------------END OF ALL---------------" << endl;

}

**void** intersect(pii p){

//cout << "Intersecting with ", print(p);

**while**(!S.empty()){

curr = \*S.begin();

S.erase(S.begin());

curr.xx = max(curr.xx, p.xx);

**if** (curr.xx <= curr.yy){

S.insert(curr);

**break**;

}

}

**while**(!S.empty()){

curr = \*S.rbegin();

S.erase(\*S.rbegin());

curr.yy = min(curr.yy, p.yy);

**if** (curr.xx <= curr.yy){

S.insert(curr);

**break**;

}

}

//printall();

}

**void** substract(pii p){

//cout << "Substracting ", print(p);

**while**((it = S.lower\_bound(pii(p.xx, -1))) != S.end()){

curr = \*it;

S.erase(it);

curr.xx = max(curr.xx, p.yy+1);

**if** (curr.xx <= curr.yy){

it = (S.insert(curr)).first;

**break**;

}

}

//if (S.empty()) return;

**if** (it == S.begin()) **return**; //this should to be enough..

curr = \*(--it);

S.erase(it);

**if** (curr.yy > p.yy){

S.insert(pii(curr.xx, p.xx-1));

S.insert(pii(p.yy+1, curr.yy));

}**else** S.insert(pii(curr.xx, min(curr.yy, p.xx-1)));

//printall();

}

**void** tiny\_intersect(pii p){ // needs testing

substract(pii(-oo, p.xx-1));

substract(pii(p.yy+1, +oo));

}

**void** add(pii p){ // needs testing

//cout << "Adding ", print(p);

**while**((it = S.lower\_bound(pii(p.xx, -1))) != S.end()){

curr = \*it;

S.erase(it);

**if** (curr.xx > p.yy+1){

it = (S.insert(curr)).first;

**break**;

}**else** p.yy = max(p.yy, curr.yy);

}

**if** (it != S.begin()){

curr = \*(--it);

**if** (curr.yy >= p.xx-1) S.erase(\*it), p.xx=min(p.xx, curr.xx), p.yy=max(p.yy, curr.yy);

}

S.insert(p);

//printall();

}

**void** naive\_add(pii p){

S.insert(p);

}

**bool** empty(){ **return** S.empty(); }

**int** size(){ **return** S.size(); }

set<pii>::iterator begin(){ **return** S.begin(); }

set<pii>::iterator end(){ **return** S.end(); }

**void** tiny\_intersect(Intervals o){ foreach(it, o) tiny\_intersect(\*it); } // actually this isn't what i want, i should substract the complement..

**void** tiny\_substract(Intervals o){ foreach(it, o) substract(\*it); }

**void** tiny\_add(Intervals o){ foreach(it, o) add(\*it); }

} IT;

**#define** MAX 100100 // at least 2\*deg

**#define** MOD 1000000007

**int** RQ[MAX] = {1, -1, -1}, RA[MAX] = {0, 1}, DEG = 2;

**int** Q[MAX], A[MAX], tmp[MAX];

**int** B[MAX];

ll sg(**int** k){ **return** (k&1)?-1ll:1ll; }

**void** inc(**int** &x, ll y){

x = (x+y)%MOD;

**if** (x<0) x += MOD;

}

**int** solve(**int** n){

**for**(**int** I=0; I<=DEG; I++) Q[I]=RQ[I], A[I]=RA[I];

**while**(**true**){

**for**(**int** I=DEG ; I<=2\*DEG; I++) A[I]=0;

**for**(**int** I=DEG+1; I<=2\*DEG; I++) Q[I]=0;

**for**(**int** I=0 ; I<=2\*DEG; I++) tmp[I]=0;

**for**(**int** I=DEG; I<=2\*DEG; I++)

**for**(**int** K=1; K<=DEG; K++) inc(A[I], -1ll \* Q[K] \* A[I-K]); // next K values for the recurrence

**if** (n <= 2\*DEG) **return** A[n];

**for**(**int** I=0; I<=2\*DEG; I+=2)

**for**(**int** K=0; K<=I; K++)

inc(tmp[I], sg(K) \* Q[K] \* Q[I-K]); // getting new recurrence using only previous DEG elements with the same parity

**for**(**int** I=1; I<=DEG; I++) Q[I]=tmp[2\*I]; // seeing that the terms of the sequence with the correct parity do not depend on the other half

**for**(**int** I=0; I<DEG; I++) A[I]=A[I\*2 + (n&1) ]; // now we reduce the problem to finding N/2th term of the half sequence recurrence (need new start terms)

//for(int I=0; I<=DEG; I++) cout<<Q[I]<<" \n"[I==DEG];

//for(int I=0; I<DEG; I++) cout<<A[I]<<" \n"[I+1==DEG];

n = n/2;

}

}

**int** main(){

ios\_base::sync\_with\_stdio(0);

cin.tie(0);

**for**(**int** I=0; I<DEG; I++) B[I]=RA[I];

**for**(**int** I=DEG; I<MAX; I++)

**for**(**int** K=1; K<=DEG; K++) inc(B[I], - 1ll\*RQ[K]\*B[I-K]);

**for**(**int** N=0; N<10100; N++) **if** (solve(N) != B[N]){

cout << N << " -> " << solve(N) << " " << B[N] << endl;

WAIT;

}

}

**struct** pairhash {

**public**:

**template** <**typename** T, **typename** U>

std::size\_t **operator**()(**const** std::pair<T, U> &x) **const**

{

**return** std::hash<T>()(x.first)\*31 ^ std::hash<U>()(x.second);

}

};

unordered\_map<pii, **int**, pairhash> dp;

Player chooses a non-empty subset of games (possibly, all) and makes moves in all of them. A position is losing iff each game is in a losing position.

Player chooses a proper subset of games (not empty and not all), and makes moves in all chosen ones. A position is losing iff grundy numbers of all games are equal.

Player must move in all games, and loses if can't move in some game. A position is losing if any of the games is in a losing position.

vector<int> stable\_matching(vector< vector<int> >& orderM,

vector< vector<int> >& orderW) {

const int N = orderM.size();

vector< vector<int> > preferW(N, vector<int>(N+1, N));

vector<int> matchW(N, N), proposedM(N);

for (int w = 0; w < N; ++w)

for (int i = 0; i < N; ++i)

preferW[w][ orderW[w][i] ] = i;

for (int m\_ = 0; m\_ < N; ++m\_) {

for (int m = m\_; m < N; ) {

int w = orderM[m][ proposedM[m]++ ];

if (preferW[w][ m ] < preferW[w][ matchW[w] ])

swap(m, matchW[w]);

}

}

return matchW;

}

pii floyd\_cycle\_finding (**int** (\*f)(**int**), **int** x0){

**int** tortoise = f(x0), hare = f(f(x0

**while** (tortoise != hare){

tortoise = f(tortoise);

hare = f(f(hare));

}

**int** mu = 0;

hare = tortoise;

tortoise = x0;

**while** (tortoise != hare){

tortoise = f(tortoise);

hare = f(hare);

mu += 1;

}

**int** lambda = 1;

hare = f(tortoise);

**while** (tortoise != hare){

hare = f(hare);

lambda += 1;

}

**return** make\_pair(mu, lambda);

}

**double** curtime() { **return** static\_cast<**double**>(clock()) / CLOCKS\_PER\_SEC; }

**int** simulated\_annealing(**int** n, **double** seconds) {

default\_random\_engine rng;

uniform\_real\_distribution<**double**> randfloat(0.0, 1.0);

uniform\_int\_distribution<**int**> randint(0, n - 2);

// random initial solution

vi sol(n);

rep(i,0,n) sol[i] = i + 1;

random\_shuffle(sol.begin(), sol.end());

// initialize score

**int** score = 0;

rep(i,1,n) score += abs(sol[i] - sol[i-1]);

**int** iters = 0;

**double** T0 = 100.0, T1 = 0.001,

progress = 0, temp = T0,

starttime = curtime();

**while** (**true**) {

**if** (!(iters & ((1 << 4) - 1))) {

progress = (curtime() - starttime) / seconds;

temp = T0 \* pow(T1 / T0, progress);

**if** (progress > 1.0) **break**; }

// random mutation

**int** a = randint(rng);

// compute delta for mutation

**int** delta = 0;

**if** (a > 0) delta += abs(sol[a+1] - sol[a-1]) - abs(sol[a] - sol[a-1]);

**if** (a+2 < n) delta += abs(sol[a] - sol[a+2]) - abs(sol[a+1] - sol[a+2]);

// maybe apply mutation

**if** (delta >= 0 || randfloat(rng) < exp(delta / temp)) {

swap(sol[a], sol[a+1]);

score += delta;

// if (score >= target) return;

}

iters++; }

**return** score;

}

**const** **int** maxn = **100000**;  
**int** tx[maxn \* 4];  
**int** ty[maxn \* 4];  
**bool** divX[maxn \* 4];  
  
**bool** **cmpX**(**const** pii &a, **const** pii &b) {  
    **return** a.first < b.first;  
}  
  
**bool** **cmpY**(**const** pii &a, **const** pii &b) {  
    **return** a.second < b.second;  
}  
  
**void** **buildTree**(**int** node, **int** left, **int** right, pii points[]) {  
    **if** (left > right)  
        **return**;  
    **int** mid = (left + right) >> **1**;  
  
    //sort(points + left, points + right + **1**, divX ? cmpX : cmpY);  
    **int** minx = INT\_MAX;  
    **int** maxx = INT\_MIN;  
    **int** miny = INT\_MAX;  
    **int** maxy = INT\_MIN;  
    **for** (**int** i = left; i <= right; i++) {  
        minx = min(minx, points[i].first);  
        maxx = max(maxx, points[i].first);  
        miny = min(miny, points[i].second);  
        maxy = max(maxy, points[i].second);  
    }  
    divX[node] = (maxx - minx) >= (maxy - miny);  
    nth\_element(points + left, points + mid, points + right + **1**, divX[node] ? cmpX : cmpY);  
  
    tx[node] = points[mid].first;  
    ty[node] = points[mid].second;  
    **if** (left == right)  
        **return**;  
    buildTree(node \* **2**, left, mid - **1**, points);  
    buildTree(node \* **2** + **1**, mid + **1**, right, points);  
}  
  
**long** **long** closestDist;  
**int** closestNode;  
  
**void** **findNearestNeighbour**(**int** node, **int** left, **int** right, **int** x, **int** y) {  
    **if** (left > right)  
        **return**;  
    **int** dx = x - tx[node];  
    **int** dy = y - ty[node];  
    **long** **long** d = dx \* (**long** **long**) dx + dy \* (**long** **long**) dy;  
    **if** (closestDist > d) {  
        closestDist = d;  
        closestNode = node;  
    }  
    **if** (left == right)  
        **return**;  
  
    **int** delta = divX[node] ? dx : dy;  
    **long** **long** delta2 = delta \* (**long** **long**) delta;  
    **int** mid = (left + right) >> **1**;  
    **int** n1 = node << **1**;  
    **int** l1 = left;  
    **int** r1 = mid - **1**;  
    **int** n2 = node << **1** | **1**;  
    **int** l2 = mid + **1**;  
    **int** r2 = right;  
    **if** (delta > **0**)  
        swap(l1, l2), swap(r1, r2), swap(n1, n2);  
  
    findNearestNeighbour(n1, l1, r1, x, y);  
    **if** (delta2 < closestDist)  
        findNearestNeighbour(n2, l2, r2, x, y);  
}  
  
**int** **findNearestNeighbour**(**int** n, **int** x, **int** y) {  
    closestDist = LLONG\_MAX;  
    findNearestNeighbour(**1**, **0**, n - **1**, x, y);  
    **return** closestNode;  
}  
  
**int** **main**() {  
    vpii p;  
    p.push\_back(make\_pair(**0**, **2**));  
    p.push\_back(make\_pair(**0**, **3**));  
    p.push\_back(make\_pair(-**1**, **0**));  
  
    p.resize(unique(p.begin(), p.end()) - p.begin());  
  
    **int** n = p.size();  
    buildTree(**1**, **0**, n - **1**, &(vpii(p)[0]));  
    **int** res = findNearestNeighbour(n, **0**, **0**);  
  
    cout << p[res].first << " " << p[res].second << endl;  
  
    **return** **0**;  
}

**import** java.io.\*;

**import** java.util.\*;

**import** java.math.\*;

**public** **class** Main

{

BufferedReader in;

PrintStream out;

StringTokenizer tok;

**public** Main() **throws** NumberFormatException, IOException

{

in = **new** BufferedReader(**new** InputStreamReader(System.*in*));

out = System.*out*;

run();

}

**void** run() **throws** NumberFormatException, IOException

{

}

**public** **static** **void** main(String[] args) **throws** NumberFormatException, IOException

{

**new** Main();

}

String nextToken() **throws** IOException

{

String line = "";

**while**(tok == **null** || !tok.hasMoreTokens()) {

**if**((line = in.readLine()) != **null**)

tok = **new** StringTokenizer(line);

**else**

**return** **null**;

}

**return** tok.nextToken();

}

**int** nextInt() **throws** NumberFormatException, IOException

{

**return** Integer.*parseInt*(nextToken());

}

**long** nextLong() **throws** NumberFormatException, IOException

{

**return** Long.*parseLong*(nextToken());

}

**double** nextDouble() **throws** NumberFormatException, IOException

{

**return** Double.*parseDouble*(nextToken());

}

}

