**0D Objects: Points**

#define EPS 1e-9

**struct** point\_i { **int** x, y; // whenever possible, work with point\_i

point\_i(**int** \_x, **int** \_y) { x = \_x, y = \_y; }}; // constructor (optional)

**struct** point { **double** x, y;

point(**double** \_x, **double** \_y) { x = \_x, y = \_y; } // constructor

**bool** **operator** < (point other) { // override ‘less than’ operator

**if** (fabs(x - other.x) < EPS) // useful for sorting

**return** x < other.x; // first criteria , by x-axis

**return** y < other.y; }}; // second criteria, by y-axis

**// in int main(), assuming we already have a populated vector<point> P;**

**sort**(P.**begin**(), P.**end**()); // comparison operator is defined above

**bool** areSame(point\_i p1, point\_i p2) { // integer version

**return** p1.x == p2.x && p1.y == p2.y; } // precise comparison

**bool** areSame(point p1, point p2) { // floating point version

**return** fabs(p1.x - p2.x) < EPS && fabs(p1.y - p2.y) < EPS; }

**double** dist(point p1, point p2) { // Euclidean distance

**return** hypot(p1.x - p2.x, p1.y - p2.y); } // return double

**// rotate p by theta degrees CCW w.r.t origin (0,0)**

point **rotate**(point p, **double** theta) {

// rotation matrix R(theta) = [cos(theta) -sin(theta)]

// [sin(theta) cos(theta)]

// usage: [x’] = R(theta) \* [x]

// [y’] [y]

**double** rad = DEG\_to\_RAD(theta); // multiply theta with PI/180.0

**return** point(p.x\*cos(rad) - p.y\*sin(rad), p.x\*sin(rad) + p.y\*cos(rad));}

**1D Objects: Lines**

**// ax + by + c = 0**

**struct** line { **double** a, b, c; }; // a way to represent a line

**// the answer is stored in the third parameter (pass by reference)**

**void** pointsToLine(point p1, point p2, line \*l){

**if** (p1.x == p2.x) { // vertical line is handled nicely here

l->a = 1.0; l->b = 0.0; l->c = -p1.x;} // default values

**else** {

l->a = -(double)(p1.y - p2.y) / (p1.x - p2.x);

l->b = 1.0;

l->c = -(double)(l->a \* p1.x) - (l->b \* p1.y);}}

**bool** areParallel(line l1, line l2) { // check coefficient a + b

**return** (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS); }

**bool** areSame(line l1, line l2) { // also check coefficient c

**return** areParallel(l1 ,l2) && (fabs(l1.c - l2.c) < EPS); }

**// returns true (+ intersection point) if two lines are intersect**

**bool** areIntersect(line l1, line l2, point \*p) {

**if** (areSame(l1, l2)) **return** **false**; // all points intersect

**if** (areParallel(l1, l2)) **return** **false**; // no intersection

// solve system of 2 linear algebraic equations with 2 unknowns

p->x = (l2.b \* l1.c - l1.b \* l2.c)/(l2.a \* l1.b - l1.a \* l2.b);

**if** (fabs(l1.b) > EPS) // special case: test for vertical line

p->y = -(l1.a \* p->x + l1.c)/l1.b; // avoid division by zero

**else** p->y = -(l2.a \* p->x + l2.c)/l2.b;

**return** **true**; }

**// vector**

**struct** vec { **double** x, y; // we use ‘vec’ to differentiate with STL vector

vec(**double** \_x, **double** \_y) { x = \_x, y = \_y; } };

vec toVector(point p1, point p2) { // convert 2 points to vector

**return** vec(p2.x - p1.x, p2.y - p1.y); }

vec scaleVector(vec v, **double** s) { // nonnegative s = [<1 ... 1 ... >1]

**return** vec(v.x \* s, v.y \* s); } // shorter v same v longer v

point translate(point p, vec v) { // translate p according to v

**return** point(p.x + v.x , p.y + v.y); }

**// returns the distance from p to the line defined by two points A and B**

**// (A and B must be different) the closest point is stored in the 4th parameter**

**double** distToLine(point p, point A, point B, point \*c) {

// formula: cp = A + (p-A).(B-A) / |B-A| \* (B-A)

**double** scale = (double) ((p.x - A.x)\*(B.x - A.x) + (p.y - A.y)\*(B.y - A.y))/ ((B.x - A.x)\*(B.x - A.x) + (B.y - A.y)\*(B.y - A.y));

c->x = A.x + scale\*(B.x - A.x);

c->y = A.y + scale\*(B.y - A.y);

**return** dist(p, \*c); } // Euclidean distance between p and \*c

**// returns the distance from p to the line segment ab (still OK if A == B)**

**// the closest point is stored in the 4th parameter (by reference)**

**double** distToLineSegment(point p, point A, point B, point\* c) {

**if** ((B.x - A.x)\*(p.x - A.x) + (B.y - A.y)\*(p.y - A.y) < EPS) {

c->x = A.x; c->y = A.y; // closer to A

**return** dist(p, A); } // Euclidean distance between p and A

**if** ((A.x - B.x)\*(p.x - B.x) + (A.y - B.y)\*(p.y-B.y) < EPS) {

c->x = B.x; c->y = B.y; // closer to B

**return** dist(p, B); } // Euclidean distance between p and B

**return** distToLine(p, A, B, c); } // call distToLine as above

**double** cross(point p, point q, point r) { // cross product

**return** (r.x - q.x)\*(p.y - q.y) - (r.y - q.y)\*(p.x - q.x); }

**// returns true if point r is on the same line as the line pq**

**bool** collinear(point p, point q, point r) {

**return** fabs(cross(p, q, r)) < EPS; } // notice the comparison with EPS

**// returns true if point r is on the left side of line pq**

**bool** ccw(point p, point q, point r) {

**return** cross(p, q, r) > 0; } // can be modified to accept collinear points

**2D Objects: Circles**

**//(x − a)\*(x − a) + (y − b)\*(y − b) = r\*r**

**int** inCircle(point\_i p, point\_i c, **int** r) { // all integer version

**int** dx = p.x - c.x, dy = p.y - c.y;

**int** Euc = dx \* dx + dy \* dy, rSq = r \* r; // all integer

**return** Euc < rSq ? 0 : Euc == rSq ? 1 : 2; } // inside/border/outside

**// Given 2 points circle (p1 and p2) and radius r of the corresponding circle, we**

**// can determine the location of the centers (c1 and c2)**

**bool** circle2PtsRad(point p1, point p2, **double** r, point \*c) { // answer at \*c

**double** d2 = (p1.x - p2.x)\*(p1.x - p2.x) + (p1.y - p2.y)\*(p1.y - p2.y);

**double** det = r \* r / d2 - 0.25;

**if** (det < 0.0) **return** **false**;

**double** h = **sqrt**(det);

c->x = (p1.x + p2.x)\*0.5 + (p1.y - p2.y)\*h;

c->y = (p1.y + p2.y)\*0.5 + (p2.x - p1.x)\*h;

**return** true; } // to get the other center, reverse p1 and p2

**3D Objects: Spheres**

**// Great-Circle Distance between any two points p and q in the spheres**

**double** gcDistance(dd pLat, dd pLong, dd qLat, dd qLong, dd radius) {

pLat \*= PI/180; pLong \*= PI/180; // conversion from degree to radian

qLat \*= PI/180; qLong \*= PI/180;

**return** radius\*acos(cos(pLat)\*cos(pLong)\*cos(qLat)\*cos(qLong) +

cos(pLat)\*sin(pLong)\*cos(qLat)\*sin(qLong) + sin(pLat)\*sin(qLat));}

**// polygon representation**

**struct** point { **double** x, y; // reproduced here

point (**double** \_x, **double** \_y) { x = \_x, y = \_y; } };

// 6 points, entered in counter clockwise order, 0-based indexing

**vector**<point> P; //7 P5--------------P4

P.**push\_back**(point(1, 1)); //6 | \

P.**push\_back**(point(3, 3)); //5 | \

P.**push\_back**(point(9, 1)); //4 | P3

P.**push\_back**(point(12, 4)); //3 | P1\_\_\_ /

P.**push\_back**(point(9, 7)); //2 | / \ \_\_\_ /

P.**push\_back**(point(1, 7)); //1 P0 P2

P.**push\_back**(P[0]); // important: loop back //0 1 2 3 4 5 6 7 8 9 101112

**// perimeter of a polygon**

**double** dist(point p1, point p2) { // get Euclidean distance of two points

**return** hypot(p1.x - p2.x, p1.y - p2.y); } // as shown earlier

// returns the perimeter, which is the sum of Euclidian distances

// of consecutive line segments (polygon edges)

**double** perimeter(vector<point> P) {

**double** result = 0.0;

**for** (**int** i = 0; i < (**int**)P.**size**() - 1; i++) // assume that the first vertex

result += dist(P[i], P[(i + 1)]); **return** result; } // is equal to the last vertex

**// area of a polygon**

|x0 y0 |

|x1 y1 |

|x2 y2 |

1 |x3 y3 |

A = - \* |. . | = 1/2 \* (x0y1 + x1y2 + x2y3 + ... + x(n-1)y0

2 |. . | -x1y0 - x2y1 - x3y2 - ... - x0y(n-1))

|. . |

|x(n-1) y(n-1)|

|x0 y0 | <-- cycle back to the first vertex

**// returns the area, which is half the determinant**

**double** area(**vector**<point> P) {

**double** result = 0.0, x1, y1, x2, y2;

**for** (**int** i = 0; i < (**int**)P.**size**() - 1; i++) {

x1 = P[i].x; x2 = P[(i + 1)].x; // assume that the first vertex

y1 = P[i].y; y2 = P[(i + 1)].y; // is equal to the last vertex

result += (x1 \* y2 - x2 \* y1); }

**return** fabs(result) / 2.0; }

**// checking if a polygon is convex**

**// returns true if all three consecutive vertices of P form the same turns**

**bool** isConvex(**vector**<point> P) {

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

**if** (sz < 3) // boundary case, we treat a point or a line as not convex

**return** **false**;

**bool** isLeft = ccw(P[0], P[1], P[2]); // remember one turn result

**for** (**int** i = 1; i < (**int**)P.**size**(); i++) // then compare with the others

**if** (ccw(P[i], P[(i + 1) % sz], P[(i + 2) % sz]) != isLeft)

**return** **false**; // if different sign, then this polygon is concave

**return** **true**; } // this polygon is convex

**// checking if a point is inside a polygon**

**double** angle(point a, point b, point c) {

**double** ux = b.x - a.x, uy = b.y - a.y;

**double** vx = c.x - a.x, vy = c.y - a.y;

**return** acos((ux\*vx + uy\*vy)/sqrt((ux\*ux + uy\*uy) \* (vx\*vx + vy\*vy))); }

**// returns true if point p is in either convex/concave polygon P**

**bool** inPolygon(point p, **vector**<point> P) {

**if** ((**int**)P.**size**() == 0) **return** **false**;

**double** sum = 0;

**for** (**int** i = 0; i < (**int**)P.**size**() - 1; i++) { // assume that the first vertex

**if** (cross(p, P[i], P[i + 1]) < 0) // is equal to the last vertex

sum -= angle(p, P[i], P[i + 1]); // right turn/cw

**else** sum += angle(p, P[i], P[i + 1]); } // left turn/ccw

**return** (fabs(sum - 2\*PI) < EPS || fabs(sum + 2\*PI) < EPS); }

**// cutting polygon with a straight line**

**// line segment p-q intersect with line A-B.**

point lineIntersectSeg(point p, point q, point A, point B) {

**double** a = B.y - A.y;

**double** b = A.x - B.x;

**double** c = B.x \* A.y - A.x \* B.y;

**double** u = fabs(a \* p.x + b \* p.y + c);

**double** v = fabs(a \* q.x + b \* q.y + c);

**return** point((p.x \* v + q.x \* u)/(u + v), (p.y \* v + q.y \* u)/(u + v)); }

**// cuts polygon Q along the line formed by point a -> point b**

**// (note: the last point must be the same as the first point)**

**vector**<point> cutPolygon(point a, point b, **vector**<point> Q) {

**vector**<point> P;

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

**double** left1 = cross(a, b, Q[i]), left2 = 0.0;

**if** (i != (**int**)Q.**size**() - 1) left2 = cross(a, b, Q[i + 1]);

**if** (left1 > -EPS) P.**push\_back**(Q[i]);

**if** (left1 \* left2 < -EPS)

P.**push\_back**(lineIntersectSeg(Q[i], Q[i + 1], a, b));}

**if** (P.**empty**()) **return** P;

**if** (fabs(P.back().x - P.**front**().x) > EPS || fabs(P.**back**().y - P.**front**().y) > EPS)

P.**push\_back**(P.**front**());

**return** P; }

**// generating list of prime numbers**

#include <bitset> // compact STL for Sieve, more efficient than vector<bool>!

ll \_sieve\_size; // ll is defined as: typedef long long ll;

**bitset**<10000010> bs; // 10^7 should be enough for most cases

**vector**<**int**> primes;

**void** sieve(ll upperbound) { // create list of primes in [0..upperbound]

\_sieve\_size = upperbound + 1; // add 1 to include upperbound

bs.**set**(); // set all bits to 1

bs[0] = bs[1] = 0; // except index 0 and 1

**for** (ll i = 2; i <= \_sieve\_size; i++) if (bs[i]) {

// cross out multiples of i starting from i \* i!

**for** (ll j = i \* i; j <= \_sieve\_size; j += i) bs[j] = 0;

primes.**push\_back**((**int**)i); }} // also add this vector containing list of primes

// call this method in main method

**bool** isPrime(ll N) { // a good enough deterministic prime tester

**if** (N <= \_sieve\_size) **return** bs[N]; // O(1) for small primes

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

**if** (N % primes[i] == 0) **return** **false**;

**return** **true**; } // it takes longer time if N is a large prime!

// inside **int main()**

sieve(10000000); // can go up to 10^7 (need few seconds)

printf("%d\n", isPrime(2147483647)); // is a prime

**// finding prime factors with optimized trial divisions**

vi primeFactors(ll N) { // remember: vi is vector<int>, ll is long long

vi factors;

ll PF\_idx = 0, PF = primes[PF\_idx]; // using PF = 2, then 3,5,7,... is also ok

**while** (N != 1 && (PF \* PF <= N)) { // stop at sqrt(N), but N can get smaller

**while** (N % PF == 0) { N /= PF; factors.**push\_back**(PF); } // remove this PF

PF = primes[++PF\_idx]; } // only consider primes!

**if** (N != 1) factors.**push\_back**(N); // special case if N is actually a prime

**return** factors; } // if N does not fit in 32-bit integer and is a prime number

// then 'factors' will have to be changed to vector<ll>

// inside intmain(), assuming sieve(1000000) has been called before

vi res = primeFactors(2147483647); // slowest, 2147483647 is a prime

res = primeFactors(136117223861LL); // slow, 2 large pfactors 104729\*1299709

res = primeFactors(142391208960LL); // faster, 2^10\*3^4\*5\*7^4\*11\*13

**// functions involving prime factors**

**// numPF(N):Count the number of prime factors of N**

ll numPF(ll N) {

ll PF\_idx = 0, PF = primes[PF\_idx], ans = 0;

**while** (N != 1 && (PF \* PF <= N)) {

**while** (N % PF == 0) { N /= PF; ans++; }

PF = primes[++PF\_idx];

}

**if** (N != 1) ans++;

**return** ans;

}

**// numDiffPF(N): count the number of different prime factors of N**

ll numDiffPF(ll N) {   
 ll PF\_idx = 0, PF = primes[PF\_idx], ans = 0;  
 **while** (PF \* PF <= N) {  
 **if** (N % PF == 0) ans ++; // count this pf only once  
 **while** (N % PF == 0) N /= PF;  
 PF = primes[++PF\_idx];  
 }  
 **if** (N != 1) ans++;  
 **return** ans;  
}

**// sumPF(N): sum the prime factors of N**

ll sumPF(ll N) {  
 ll PF\_idx = 0, PF = primes[PF\_idx], ans = 0;  
 **while** (PF \* PF <= N) {  
 **while** (N % PF == 0) { N /= PF; ans += PF; }  
 PF = primes[++PF\_idx];  
 }  
 **if** (N != 1) ans += N;  
 **return** ans;  
}

**// numDiv(N): count the number of divisors of N**

ll numDiv(ll N) {

ll PF\_idx = 0, PF = primes[PF\_idx], ans = 1; // start from ans = 1

**while** (N != 1 && (PF \* PF <= N)) {

ll power = 0; // count the power

**while** (N % PF == 0) { N /= PF; power++; }

ans \*= (power + 1); // according to the formula

PF = primes[++PF\_idx];

}

**if** (N != 1) ans \*= 2; // (last factor has pow = 1, we add 1 to it)

**return** ans;

}

**// sumDiv(N): sum the divisors of N**

ll sumDiv(ll N) {

ll PF\_idx = 0, PF = primes[PF\_idx], ans = 1; // start from ans = 1

**while** (N != 1 && (PF \* PF <= N)) {

ll power = 0;

**while** (N % PF == 0) { N /= PF; power++; }

ans \*= ((ll)pow((**double**)PF, power + 1.0) - 1)/(PF - 1); // formula

PF = primes[++PF\_idx];

}

**if** (N != 1) ans \*= ((ll)pow((**double**)N, 2.0) - 1) / (N - 1); // last one

**return** ans;

}

**// EulerPhi(N): count the number of positive integers < N that are**

**// relatively prime to N.**

ll EulerPhi(ll N) {

ll PF\_idx = 0, PF = primes[PF\_idx], ans = N; // start from ans = N

**while** (N != 1 && (PF \* PF <= N)) {

if (N % PF == 0) ans -= ans / PF; // only count unique factor

**while** (N % PF == 0) N /= PF;

PF = primes[++PF\_idx];

}

**if** (N != 1) ans -= ans / N; // last factor

**return** ans;

}

**// square matrix exponentiation**

#define MAX\_N 105 // increase/decrease this value as needed

**struct** Matrix {

**int** mat[MAX\_N][MAX\_N]; // we will return a 2D array

};

Matrix matMul(Matrix a, Matrix b) { // O(n^3)

Matrix ans; **int** i, j, k;

**for** (i = 0; i < MAX\_N; i++)

**for** (j = 0; j < MAX\_N; j++)

**for** (ans.mat[i][j] = k = 0; k < MAX\_N; k++) // if necessary, use

ans.mat[i][j] += a.mat[i][k] \* b.mat[k][j]; // modulo arithmetic

**return** ans;

}

Matrix matPow(Matrix base, **int** p) { // O(n^3 log p)

Matrix ans; **int** i, j;

**for** (i = 0; i < MAX\_N; i++)

**for** (j = 0; j < MAX\_N; j++)

ans.mat[i][j] = (i == j); // prepare identity matrix

**while** (p) { // iterative version of Divide & Conquer exponentiation

**if** (p & 1) ans = matMul(ans, base); // if p is odd (last bit is on)

base = matMul(base, base); // square the base

p >>= 1; // divide p by 2

}

**return** ans;

}