Chapter 7

(Computational) Geometry

Let no man ignorant of geometry enter here.
— **Plato's Academy in Athens**

7.1 Overview and Motivation

(Computational[1]) Geometry is yet another topic that frequently appears in programming contests. Almost all ICPC problem sets have *at least one* geometry problem. If you are lucky, it will ask you for some geometry solution that you have learned before. Usually you draw the geometrical object(s) and then derive the solution from some basic geometric formulas. However, many geometry problems are the *computational* ones that require some complex algorithm(s).

In IOI, the existence of geometry-specific problems depends on the tasks chosen by the Scientific Committee that year. In recent years (2009-2012), IOI tasks do not feature *pure* geometry-specific problems. However, in the earlier years [67], every IOI contain one or two geometry related problems.

We have observed that geometry-related problems are usually not attempted during the early part of the contest time for *strategic reason* because the solutions for geometry-related problems have *lower* probability of getting Accepted (AC) during contest time compared to the solutions for other problem types in the problem set, e.g. Complete Search or Dynamic Programming problems. The typical issues with geometry problems are as follow:

- Many geometry problems have one and usually several tricky 'corner test cases', e.g. What if the lines are vertical (infinite gradient)?, What if the points are collinear?, What if the polygon is concave?, What if the convex hull of a set of points is the set of points itself?, etc. Therefore, it is usually a very good idea to test your team's geometry solution with lots of corner test cases before you submit it for judging.
- There is a possibility of having floating point precision errors that cause even a 'correct' algorithm to get a Wrong Answer (WA) response.
- The solutions for geometry problems usually involve *tedious* coding.

These reasons cause many contestants to view that spending precious minutes attempting *other* problem types in the problem set more worthwhile than attempting a geometry problem that has lower probability of acceptance.

[1]We differentiate between *pure* geometry problems and the *computational* geometry ones. Pure geometry problems can normally be solved by hand (pen and paper method). Computational geometry problems typically require running an algorithm using computer to obtain the solution.

However, another not-so-good reason for the lack of attempts for geometry problems is because the contestants are not well prepared.

- The contestants forget some important basic formulas or are unable to derive the required (more complex) formulas from the basic ones.
- The contestants do not prepare well-written library functions before contest and their attempts to code such functions during stressful contest environment end up with one, but usually several[2], bug(s). In ICPC, the top teams usually fill a sizeable part of their hard copy material (which they can bring into the contest room) with lots of geometry formulas and library functions.

The main aim of this chapter is therefore to increase the number of attempts (and also AC solutions) for geometry-related problems in programming contests. Study this chapter for some ideas on tackling (computational) geometry problems in ICPCs and IOIs. There are only two sections in this chapter.

In Section 7.2, we present many (it is impossible to enumerate all) English geometric terminologies[3] and various basic formulas for 0D, 1D, 2D, and 3D **geometry objects** commonly found in programming contests. This section can be used as a quick reference when contestants are given geometry problems and are not sure of certain terminologies or forget some basic formulas.

In Section 7.3, we discuss several algorithms on 2D **polygons**. There are several nice pre-written library routines which can differentiate good from average teams (contestants) like the algorithms for deciding if a polygon is convex or concave, deciding if a point is inside or outside a polygon, cutting a polygon with a straight line, finding the convex hull of a set of points, etc.

The implementations of the formulas and computational geometry algorithms shown in this chapter use the following techniques to increase the probability of acceptance:

1. We highlight the special cases that can potentially arise and/or choose the implementation that reduces the number of such special cases.
2. We try to avoid floating point operations (i.e. division, square root, and any other operations that can produce numerical errors) and work with precise integers whenever possible (i.e. integer additions, subtractions, multiplications).
3. If we really need to work with floating point, we do floating point equality test this way: `fabs(a - b) < EPS` where `EPS` is a small number[4] like `1e-9` instead of testing if `a == b`. When we need to check if a floating point number $x \geq 0.0$, we use `x > -EPS` (similarly to test if $x \leq 0.0$, we use `x < EPS`).

[2] As a reference, the library code on points, lines, circles, triangles, and polygons shown in this chapter require several iterations of bug fixes to ensure that as many (usually subtle) bugs and special cases are handled properly.

[3] ACM ICPC and IOI contestants come from various nationalities and backgrounds. Therefore, we would like to get everyone familiarized with the English geometric terminologies.

[4] Unless otherwise stated, this `1e-9` is the default value of EPS(ilon) that we use in this chapter.

7.2 Basic Geometry Objects with Libraries

7.2.1 0D Objects: Points

1. **Point** is the basic building block of higher dimensional geometry objects. In 2D Euclidean[5] space, points are usually represented with a struct in C/C++ (or Class in Java) with two[6] members: The **x** and **y** coordinates w.r.t origin, i.e. coordinate (0, 0). If the problem description uses integer coordinates, use `int`s; otherwise, use `double`s. In order to be generic, we use the floating-point version of `struct point` in this book. A default and user-defined constructors can be used to (slightly) simplify coding later.

```
// struct point_i { int x, y; };    // basic raw form, minimalist mode
struct point_i { int x, y;     // whenever possible, work with point_i
  point_i() { x = y = 0; }                          // default constructor
  point_i(int _x, int _y) : x(_x), y(_y) {} };          // user-defined

struct point { double x, y;   // only used if more precision is needed
  point() { x = y = 0.0; }                          // default constructor
  point(double _x, double _y) : x(_x), y(_y) {} };     // user-defined
```

2. Sometimes we need to sort the points. We can easily do that by overloading the less than operator inside `struct point` and use sorting library.

```
struct point { double x, y;
  point() { x = y = 0.0; }
  point(double _x, double _y) : x(_x), y(_y) {}
  bool operator < (point other) const { // override less than operator
    if (fabs(x - other.x) > EPS)                   // useful for sorting
      return x < other.x;          // first criteria , by x-coordinate
    return y < other.y; } };       // second criteria, by y-coordinate

// in int main(), assuming we already have a populated vector<point> P
  sort(P.begin(), P.end());    // comparison operator is defined above
```

3. Sometimes we need to test if two points are equal. We can easily do that by overloading the equal operator inside `struct point`.

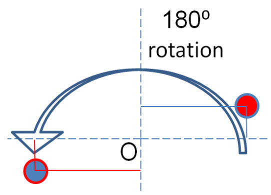
```
struct point { double x, y;
  point() { x = y = 0.0; }
  point(double _x, double _y) : x(_x), y(_y) {}
  // use EPS (1e-9) when testing equality of two floating points
  bool operator == (point other) const {
   return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS)); } };

// in int main()
  point P1(0, 0), P2(0, 0), P3(0, 1);
  printf("%d\n", P1 == P2);                                      // true
  printf("%d\n", P1 == P3);                                     // false
```

[5]For simplicity, the 2D and 3D Euclidean spaces are the 2D and 3D world that we encounter in real life.
[6]Add one more member, `z`, if you are working in 3D Euclidean space.

4. We can measure Euclidean distance[7] between two points by using the function below.

```
double dist(point p1, point p2) {                    // Euclidean distance
                      // hypot(dx, dy) returns sqrt(dx * dx + dy * dy)
  return hypot(p1.x - p2.x, p1.y - p2.y); }               // return double
```

5. We can rotate a point by angle[8] θ counter clockwise around origin (0, 0) by using a rotation matrix:



$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix}$$

Figure 7.1: Rotating point (10, 3) by 180 degrees counter clockwise around origin (0, 0)

```
// rotate p by theta degrees CCW w.r.t origin (0, 0)
point rotate(point p, double theta) {
  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)); }
```

Exercise 7.2.1.1: Compute the Euclidean distance between point (2, 2) and (6, 5)!

Exercise 7.2.1.2: Rotate a point (10, 3) by 90 degrees counter clockwise around origin. What is the new coordinate of the rotated point? (easy to compute by hand).

Exercise 7.2.1.3: Rotate the same point (10, 3) by 77 degrees counter clockwise around origin. What is the new coordinate of the rotated point? (this time you need to use calculator and the rotation matrix).

7.2.2 1D Objects: Lines

1. **Line** in 2D Euclidean space is the set of points whose coordinates satisfy a given linear equation $ax + by + c = 0$. Subsequent functions in this subsection assume that this linear equation has $b = 1$ for non vertical lines and $b = 0$ for vertical lines unless otherwise stated. Lines are usually represented with a struct in C/C++ (or Class in Java) with three members: The coefficients a, b, and c of that line equation.

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

2. We can compute the required line equation if we are given *at least* two points that pass through that line via the following function.

[7]The Euclidean distance between two points is simply the distance that can be measured with ruler. Algorithmically, it can be found with Pythagorean formula that we will see again in the subsection about triangle below. Here, we simply use a library function.

[8]Humans usually work with degrees, but many mathematical functions in most programming languages (e.g. C/C++/Java) work with radians. To convert an angle from degrees to radians, multiply the angle by $\frac{\pi}{180.0}$. To convert an angle from radians to degrees, multiply the angle with $\frac{180.0}{\pi}$.

```
// the answer is stored in the third parameter (pass by reference)
void pointsToLine(point p1, point p2, line &l) {
  if (fabs(p1.x - p2.x) < EPS) {              // vertical line is fine
    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;              // IMPORTANT: we fix the value of b to 1.0
    l.c = -(double)(l.a * p1.x) - p1.y;
} }
```

3. We can test whether two lines are *parallel* by checking if their coefficients a and b are the same. We can further test whether two lines are *the same* by checking if they are parallel and their coefficients c are the same (i.e. all three coefficients a, b, c are the same). Recall that in our implementation, we have fixed the value of coefficient b to 0.0 for all vertical lines and to 1.0 for all *non* vertical lines.

```
bool areParallel(line l1, line l2) {       // check coefficients 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); }
```

4. If two lines[9] are not parallel (and therefore also not the same), they will intersect at a point. That intersection point (x, y) can be found by solving the system of two linear algebraic equations[10] with two unknowns: $a_1x + b_1y + c_1 = 0$ and $a_2x + b_2y + c_2 = 0$.

```
// returns true (+ intersection point) if two lines are intersect
bool areIntersect(line l1, line l2, point &p) {
  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);
  // special case: test for vertical line to avoid division by zero
  if (fabs(l1.b) > EPS) p.y = -(l1.a * p.x + l1.c);
  else                  p.y = -(l2.a * p.x + l2.c);
  return true; }
```

5. **Line Segment** is a line with two end points with *finite length*.

6. **Vector**[11] is a line segment (thus it has two end points and length/magnitude) with a *direction*. Usually[12], vectors are represented with a struct in C/C++ (or Class in Java) with two members: The **x** and **y** magnitude of the vector. The magnitude of the vector can be scaled if needed.

7. We can translate (move) a point w.r.t a vector as a vector describes the displacement magnitude in x and y-axis.

[9] To avoid confusion, please differentiate between line intersection versus line *segment* intersection.
[10] See Section 9.9 for the general solution for a system of linear equations.
[11] Do not confuse this with C++ STL `vector` or Java `Vector`.
[12] Another potential design strategy is to merge `struct point` with `struct vec` as they are similar.

```
struct vec { double x, y;  // name: ‘vec’ is different from STL vector
  vec(double _x, double _y) : x(_x), y(_y) {} };

vec toVec(point a, point b) {       // convert 2 points to vector a->b
  return vec(b.x - a.x, b.y - a.y); }

vec scale(vec v, double s) {        // nonnegative s = [<1 .. 1 .. >1]
  return vec(v.x * s, v.y * s); }               // shorter.same.longer

point translate(point p, vec v) {        // translate p according to v
  return point(p.x + v.x , p.y + v.y); }
```

8. Given a point p and a line l (described by two points a and b), we can compute the minimum distance from p to l by first computing the location of point c in l that is closest to point p (see Figure 7.2—left) and then obtain the Euclidean distance between p and c. We can view point c as point a translated by a scaled magnitude u of vector ab, or $c = a + u \times ab$. To get u, we do scalar projection of vector ap onto vector ab by using dot product (see the dotted vector $ac = u \times ab$ in Figure 7.2—left). The short implementation of this solution is shown below.

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

double norm_sq(vec v) { return v.x * v.x + v.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 (byref)
double distToLine(point p, point a, point b, point &c) {
  // formula: c = a + u * ab
  vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  c = translate(a, scale(ab, u));                    // translate a to c
  return dist(p, c); }           // Euclidean distance between p and c
```

Note that this is not the only way to get the required answer.
Solve **Exercise 7.2.2.10** for the alternative way.

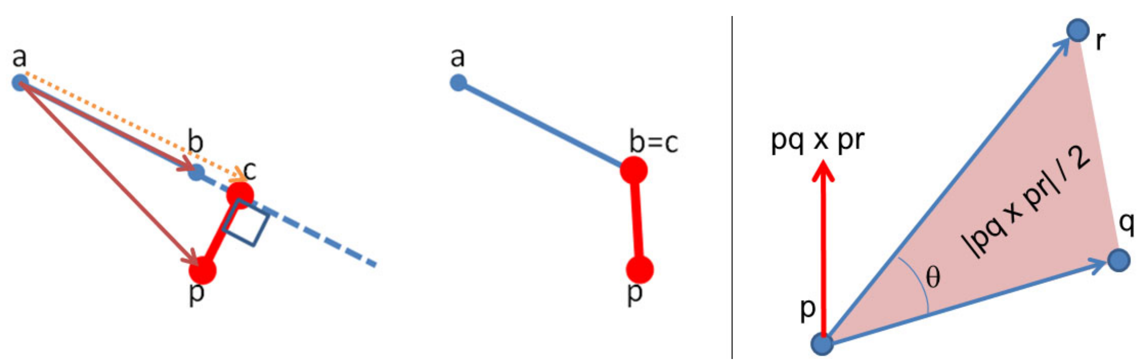


Figure 7.2: Distance to Line (left) and to Line Segment (middle); Cross Product (right)

9. If we are given a line *segment* instead (defined by two *end* points a and b), then the minimum distance from point p to line segment ab must also consider two special cases, the end points a and b of that line segment (see Figure 7.2—middle). The implementation is very similar to `distToLine` function above.

```
// returns the distance from p to the line segment ab defined by
// two points a and b (still OK if a == b)
// the closest point is stored in the 4th parameter (byref)
double distToLineSegment(point p, point a, point b, point &c) {
  vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  if (u < 0.0) { c = point(a.x, a.y);                   // closer to a
    return dist(p, a); }         // Euclidean distance between p and a
  if (u > 1.0) { c = point(b.x, b.y);                   // closer to b
    return dist(p, b); }         // Euclidean distance between p and b
  return distToLine(p, a, b, c); }          // run distToLine as above
```

10. We can compute the angle aob given three points: a, o, and b, using dot product[13]. Since $oa \cdot ob = |oa| \times |ob| \times \cos(\theta)$, we have $theta = \arccos(oa \cdot ob/(|oa| \times |ob|))$.

```
double angle(point a, point o, point b) {  // returns angle aob in rad
  vec oa = toVector(o, a), ob = toVector(o, b);
  return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob))); }
```

11. Given a line defined by two points p and q, we can determine whether a point r is on the left/right side of the line, or whether the three points p, q, and r are collinear. This can be determined with *cross product*. Let pq and pr be the two vectors obtained from these three points. The cross product $pq \times pr$ result in another vector that is perpendicular to both pq and pr. The magnitude of this vector is equal to the area of the *parallelogram* that the vectors span[14]. If the magnitude is positive/zero/negative, then we know that $p \to q \to r$ is a left turn/collinear/right turn, respectively (see Figure 7.2—right). The left turn test is more famously known as the **CCW (Counter Clockwise) Test**.

```
double cross(vec a, vec b) { return a.x * b.y - a.y * b.x; }

// note: to accept collinear points, we have to change the ‘> 0’
// returns true if point r is on the left side of line pq
bool ccw(point p, point q, point r) {
  return cross(toVec(p, q), toVec(p, r)) > 0; }

// 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(toVec(p, q), toVec(p, r))) < EPS; }
```

Source code: `ch7_01_points_lines.cpp/java`

[13] `acos` is the C/C++ function name for mathematical function `arccos`.
[14] The area of triangle pqr is therefore *half* of the area of this parallelogram.

Exercise 7.2.2.1: A line can also be described with this mathematical equation: $y = mx + c$ where m is the 'gradient'/'slope' of the line and c is the 'y-intercept' constant.
Which form is better ($ax + by + c = 0$ or the slope-intercept form $y = mx + c$)? Why?

Exercise 7.2.2.2: Compute line equation that pass through two points (2, 2) and (4, 3)!

Exercise 7.2.2.3: Compute line equation that pass through two points (2, 2) and (2, 4)!

Exercise 7.2.2.4: Suppose we insist to use the other line equation: $y = mx + c$. Show how to compute the required line equation given two points that pass through that line! Try on two points (2, 2) and (2, 4) as in **Exercise 7.2.2.3**. Do you encounter any problem?

Exercise 7.2.2.5: We can also compute the line equation if we are given *one* point and the gradient/slope of that line. Show how to compute line equation given a point and gradient!

Exercise 7.2.2.6: Translate a point c (3, 2) according to a vector ab defined by two points: a (2, 2) and b (4, 3). What is the new coordinate of the point?

Exercise 7.2.2.7: Same as **Exercise 7.2.2.6** above, but now the magnitude of vector ab is reduced by *half*. What is the new coordinate of the point?

Exercise 7.2.2.8: Same as **Exercise 7.2.2.6** above, then rotate the resulting point by 90 degrees counter clockwise around origin. What is the new coordinate of the point?

Exercise 7.2.2.9: Rotate a point c (3, 2) by 90 degrees counter clockwise around origin, then translate the resulting point according to a vector ab. Vector ab is the same as in **Exercise 7.2.2.6** above. What is the new coordinate of the point? Is the result similar with the previous **Exercise 7.2.2.8** above? What can we learn from this phenomenon?

Exercise 7.2.2.10: Rotate a point c (3, 2) by 90 degrees counter clockwise but around point p (2, 1) (note that point p is *not* the origin). Hint: You need to translate the point.

Exercise 7.2.2.11: We can compute the location of point c in line l that is closest to point p by finding the other line l' that is perpendicular with line l and pass through point p. The closest point c is the intersection point between line l and l'. Now, how to obtain a line perpendicular to l? Are there special cases that we have to be careful with?

Exercise 7.2.2.12: Given a point p and a line l (described by two points a and b), show how to compute the location of a reflection point r of point p when mirrored against line l.

Exercise 7.2.2.13: Given three points: a (2, 2), o (2, 4), and b (4, 3), compute the angle aob in degrees!

Exercise 7.2.2.14: Determine if point r (35, 30) is on the left side of, collinear with, or is on the right side of a line that passes through two points p (3, 7) and q (11, 13).

7.2.3 2D Objects: Circles

1. **Circle** centered at coordinate (a, b) in a 2D Euclidean space with **radius** r is the set of all points (x, y) such that $(x - a)^2 + (y - b)^2 = r^2$.

2. To check if a point is inside, outside, or exactly on the border of a circle, we can use the following function. Modify this function a bit for the floating point version.

```
int insideCircle(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
```

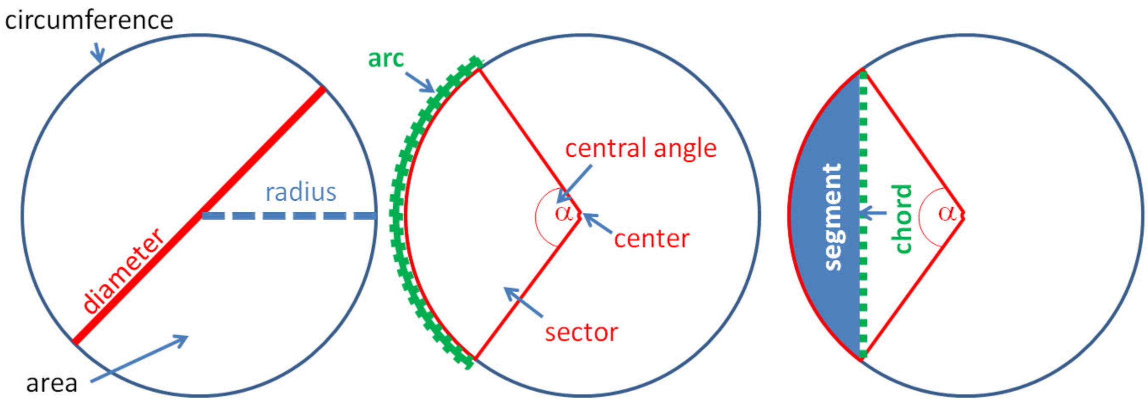


Figure 7.3: Circles

3. The constant **Pi** (π) is the ratio of *any* circle's circumference to its diameter. To avoid precision error, the safest value for programming contest if this constant π is not defined in the problem description is `pi = acos(-1.0)` or `pi = 2 * acos(0.0)`.

4. A circle with radius r has **diameter** $d = 2 \times r$ and **circumference** (or **perimeter**) $c = 2 \times \pi \times r$.

5. A circle with radius r has **area** $A = \pi \times r^2$

6. **Arc** of a circle is defined as a connected section of the circumference c of the circle. Given the central angle α (angle with vertex at the circle's center, see Figure 7.3—middle) in degrees, we can compute the length of the corresponding arc as $\frac{\alpha}{360.0} \times c$.

7. **Chord** of a circle is defined as a line segment whose endpoints lie on the circle[15]. A circle with radius r and a central angle α in degrees (see Figure 7.3—right) has the corresponding chord with length $sqrt(2 \times r^2 \times (1 - cos(\alpha)))$. This can be derived from the **Law of Cosines**—see the explanation of this law in the discussion about Triangles later. Another way to compute the length of chord given r and α is to use Trigonometry: $2 \times r \times sin(\alpha/2)$. Trigonometry is also discussed below.

8. **Sector** of a circle is defined as a region of the circle enclosed by two radius and an arc lying between the two radius. A circle with area A and a central angle α (in degrees)—see Figure 7.3, middle—has the corresponding sector area $\frac{\alpha}{360.0} \times A$.

9. **Segment** of a circle is defined as a region of the circle enclosed by a chord and an arc lying between the chord's endpoints (see Figure 7.3—right). The area of a segment can be found by subtracting the area of the corresponding sector from the area of an isosceles triangle with sides: r, r, and chord-length.

[15]Diameter is the longest chord in a circle.

10. Given 2 points on the circle ($p1$ and $p2$) and radius r of the corresponding circle, we can determine the location of the centers ($c1$ and $c2$) of the two possible circles (see Figure 7.4). The code is shown in **Exercise 7.2.3.1** below.

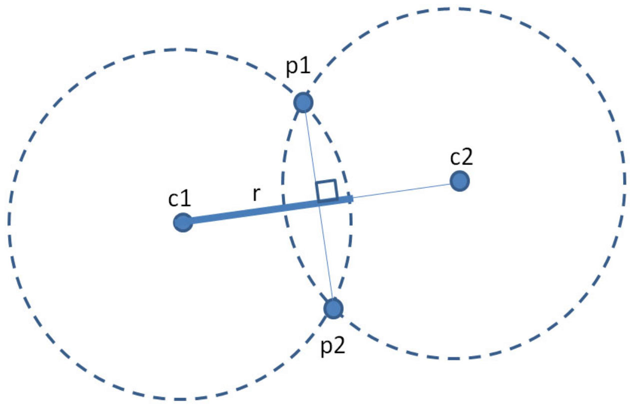


Figure 7.4: Circle Through 2 Points and Radius

Source code: `ch7_02_circles.cpp/java`

Exercise 7.2.3.1: Explain what is computed by the code below!

```
bool circle2PtsRad(point p1, point p2, double r, point &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
```

7.2.4 2D Objects: Triangles

1. **Triangle** (three angles) is a polygon with three vertices and three edges.
   There are several types of triangles:
   a. **Equilateral**: Three equal-length edges and all inside (interior) angles are 60 degrees;
   b. **Isosceles**: Two edges have the same length and two interior angles are the same.
   c. **Scalene**: All edges have different length;
   d. **Right**: *One* of its interior angle is 90 degrees (or a **right angle**).

2. A triangle with base b and height h has **area** $A = 0.5 \times b \times h$.

3. A triangle with three sides: a, b, c has **perimeter** $p = a + b + c$ and **semi-perimeter** $s = 0.5 \times p$.

4. A triangle with 3 sides: a, b, c and semi-perimeter s has area $A = sqrt(s \times (s-a) \times (s-b) \times (s-c))$. This formula is called the **Heron's Formula**.

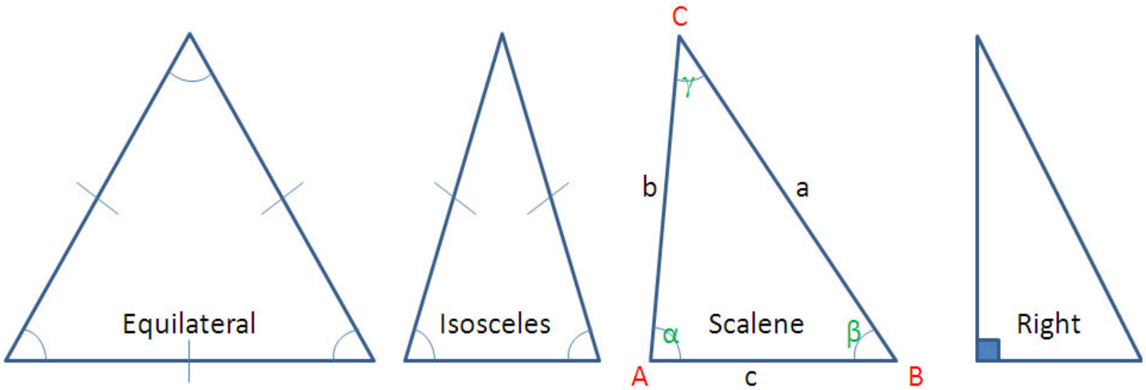


Figure 7.5: Triangles

5. A triangle with area A and semi-perimeter s has an **inscribed circle (incircle)** with radius $r = A/s$.

```
double rInCircle(double ab, double bc, double ca) {
  return area(ab, bc, ca) / (0.5 * perimeter(ab, bc, ca)); }

double rInCircle(point a, point b, point c) {
  return rInCircle(dist(a, b), dist(b, c), dist(c, a)); }
```

6. The center of incircle is the meeting point between the triangle's *angle bisectors* (see Figure 7.6—left). We can get the center if we have two angle bisectors and find their intersection point. The implementation is shown below:

```
// assumption: the required points/lines functions have been written
// returns 1 if there is an inCircle center, returns 0 otherwise
// if this function returns 1, ctr will be the inCircle center
// and r is the same as rInCircle
int inCircle(point p1, point p2, point p3, point &ctr, double &r) {
  r = rInCircle(p1, p2, p3);
  if (fabs(r) < EPS) return 0;                   // no inCircle center

  line l1, l2;                    // compute these two angle bisectors
  double ratio = dist(p1, p2) / dist(p1, p3);
  point p = translate(p2, scale(toVec(p2, p3), ratio / (1 + ratio)));
  pointsToLine(p1, p, l1);

  ratio = dist(p2, p1) / dist(p2, p3);
  p = translate(p1, scale(toVec(p1, p3), ratio / (1 + ratio)));
  pointsToLine(p2, p, l2);

  areIntersect(l1, l2, ctr);           // get their intersection point
  return 1; }
```

7. A triangle with 3 sides: a, b, c and area A has an **circumscribed circle (circumcircle)** with radius $R = a \times b \times c/(4 \times A)$.

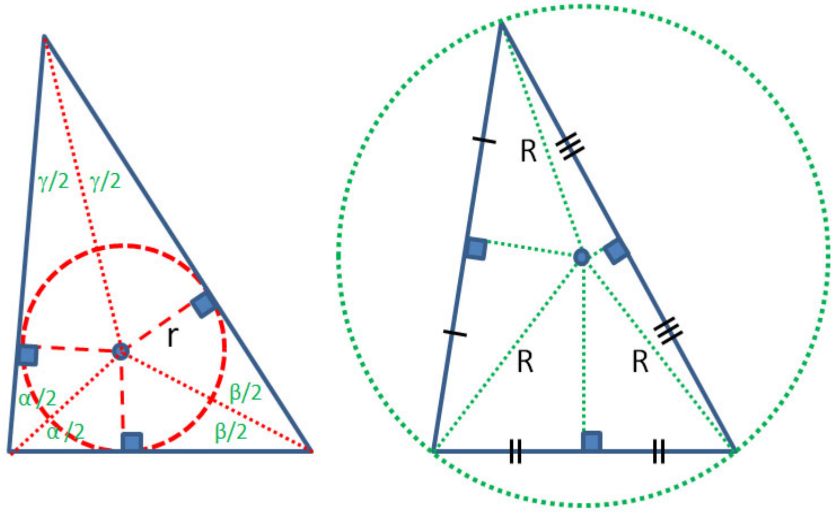


Figure 7.6: Incircle and Circumcircle of a Triangle

```
double rCircumCircle(double ab, double bc, double ca) {
  return ab * bc * ca / (4.0 * area(ab, bc, ca)); }

double rCircumCircle(point a, point b, point c) {
  return rCircumCircle(dist(a, b), dist(b, c), dist(c, a)); }
```

8. The center of circumcircle is the meeting point between the triangle's *perpendicular bisectors* (see Figure 7.6—right).

9. To check if three line segments of length a, b and c can form a triangle, we can simply check these *triangle inequalities*: $(a + b > c)$ && $(a + c > b)$ && $(b + c > a)$.
   If the result is false, then the three line segments cannot form a triangle.
   If the three lengths are sorted, with a being the smallest and c the largest, then we can simplify the check to just $(a + b > c)$.

10. When we study triangle, we should not forget **Trigonometry**—a study about the relationships between triangle sides and the angles between sides.

    In Trigonometry, the **Law of Cosines** (a.k.a. the **Cosine Formula** or the **Cosine Rule**) is a statement about a general triangle that relates the lengths of its sides to the cosine of one of its angles. See the scalene (middle) triangle in Figure 7.5. With the notations described there, we have: $c^2 = a^2 + b^2 - 2 \times a \times b \times cos(\gamma)$, or $\gamma = acos(\frac{a^2+b^2-c^2}{2 \times a \times b})$. The formula for the other two angles α and β are similarly defined.

11. In Trigonometry, the **Law of Sines** (a.k.a. the **Sine Formula** or the **Sine Rule**) is an equation relating the lengths of the sides of an arbitrary triangle to the sines of its angle. See the scalene (middle) triangle in Figure 7.5. With the notations described there and R is the radius of its circumcircle, we have: $\frac{a}{\sin(\alpha)} = \frac{b}{\sin(\beta)} = \frac{c}{\sin(\gamma)} = 2R$.

12. The **Pythagorean Theorem** specializes the Law of Cosines. This theorem only applies to right triangles. If the angle γ is a right angle (of measure 90° or $\pi/2$ radians), then $cos(\gamma) = 0$, and thus the Law of Cosines reduces to: $c^2 = a^2 + b^2$. Pythagorean theorem is used in finding the Euclidean distance between two points shown earlier.

13. The **Pythagorean Triple** is a triple with three positive integers a, b, and c—commonly written as (a, b, c)—such that $a^2 + b^2 = c^2$. A well-known example is $(3, 4, 5)$. If (a, b, c) is a Pythagorean triple, then so is (ka, kb, kc) for any positive integer k. A Pythagorean Triple describes the integer lengths of the three sides of a Right Triangle.

Source code: `ch7_03_triangles.cpp/java`

Exercise 7.2.4.1: Let a, b, and c of a triangle be 2^{18}, 2^{18}, and 2^{18}. Can we compute the area of this triangle with Heron's formula as shown in point 4 above without experiencing overflow (assuming that we use 64-bit integers)? What should we do to avoid this issue?

Exercise 7.2.4.2*: Implement the code to find the center of the circumCircle of three points a, b, and c. The function structure is similar as function `inCircle` shown in this section.

Exercise 7.2.4.3*: Implement another code to check if a point d is inside the circumCircle of three points a, b, and c.

7.2.5 2D Objects: Quadrilaterals

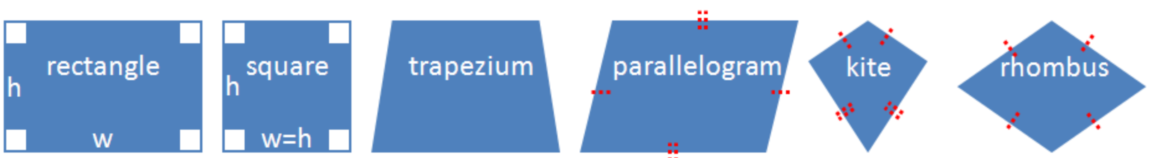


Figure 7.7: Quadrilaterals

1. **Quadrilateral** or **Quadrangle** is a polygon with four edges (and four vertices). The term 'polygon' itself is described in more details below (Section 7.3). Figure 7.7 shows a few examples of Quadrilateral objects.

2. **Rectangle** is a polygon with four edges, four vertices, and four right angles.

3. A rectangle with width w and height h has **area** $A = w \times h$ and **perimeter** $p = 2 \times (w + h)$.

4. **Square** is a special case of a rectangle where $w = h$.

5. **Trapezium** is a polygon with four edges, four vertices, and one pair of parallel edges. If the two non-parallel sides have the same length, we have an **Isosceles Trapezium**.

6. A trapezium with a pair of parallel edges of lengths $w1$ and $w2$; and a height h between both parallel edges has area $A = 0.5 \times (w1 + w2) \times h$.

7. **Parallelogram** is a polygon with four edges and four vertices. Moreover, the opposite sides must be parallel.

8. **Kite** is a quadrilateral which has two pairs of sides of the same length which are adjacent to each other. The area of a kite is $diagonal_1 \times diagonal_2/2$.

9. **Rhombus** is a special parallelogram where every side has equal length. It is also a special case of kite where every side has equal length.

Remarks about 3D Objects

Programming contest problems involving 3D objects are rare. But when such a problem does appear in a problem set, it can be one of the hardest. In the list of programming exercises below, we include an initial list of problems involving 3D objects.

Programming Exercises related to Basic Geometry:

- Points and Lines:
  1. UVa 00152 - Tree's a Crowd (sort the 3D points first)
  2. UVa 00191 - Intersection (line segment intersection)
  3. UVa 00378 - Intersecting Lines (use `areParallel`, `areSame`, `areIntersect`)
  4. UVa 00587 - There's treasure everywhere (Euclidean distance `dist`)
  5. UVa 00833 - Water Falls (recursive check, use the `ccw` tests)
  6. UVa 00837 - Light and Transparencies (line segments, sort x-coords first)
  7. **UVa 00920 - Sunny Mountains *** (Euclidean distance `dist`)
  8. UVa 01249 - Euclid (LA 4601, Southeast USA Regional 2009, vector)
  9. UVa 10242 - Fourth Point (`toVector`; `translate` points w.r.t that vector)
  10. *UVa 10250 - The Other Two Trees* (vector, rotation)
  11. **UVa 10263 - Railway *** (use `distToLineSegment`)
  12. UVa 10357 - Playball (Euclidean distance `dist`, simple Physics simulation)
  13. UVa 10466 - How Far? (Euclidean distance `dist`)
  14. UVa 10585 - Center of symmetry (sort the points)
  15. *UVa 10832 - Yoyodyne Propulsion ...* (3D Euclidean distance; simulation)
  16. UVa 10865 - Brownie Points (points and quadrants, simple)
  17. UVa 10902 - Pick-up sticks (line segment intersection)
  18. **UVa 10927 - Bright Lights *** (sort points by gradient, Euclidean distance)
  19. UVa 11068 - An Easy Task (simple 2 linear equations with 2 unknowns)
  20. UVa 11343 - Isolated Segments (line segment intersection)
  21. UVa 11505 - Logo (Euclidean distance `dist`)
  22. *UVa 11519 - Logo 2* (vectors and angles)
  23. *UVa 11894 - Genius MJ* (about rotating and translating points)
- Circles (only)
  1. *UVa 01388 - Graveyard* (divide the circle into n sectors first and then into $(n + m)$ sectors)
  2. **UVa 10005 - Packing polygons *** (complete search; use `circle2PtsRad` discussed in Chapter 7)
  3. UVa 10136 - Chocolate Chip Cookies (similar to UVa 10005)
  4. UVa 10180 - Rope Crisis in Ropeland (closest point from AB to origin; arc)
  5. UVa 10209 - Is This Integration? (square, arcs, similar to UVa 10589)
  6. UVa 10221 - Satellites (finding arc and chord length of a circle)
  7. *UVa 10283 - The Kissing Circles* (derive the formula)
  8. UVa 10432 - Polygon Inside A Circle (area of n-sided reg-polygon in circle)
  9. UVa 10451 - Ancient ... (inner/outer circle of n-sided reg polygon)
  10. UVa 10573 - Geometry Paradox (there is no 'impossible' case)
  11. **UVa 10589 - Area *** (check if point is inside intersection of 4 circles)

12. **UVa 10678 - The Grazing Cows *** (area of an *ellipse*, generalization of the formula for area of a circle)
13. *UVa 12578 - 10:6:2* (area of rectangle and circle)

- Triangles (plus Circles)
  1. UVa 00121 - Pipe Fitters (use Pythagorean theorem; grid)
  2. UVa 00143 - Orchard Trees (count integer points in triangle; precision issue)
  3. UVa 00190 - Circle Through Three ... (triangle's circumcircle)
  4. UVa 00375 - Inscribed Circles and ... (triangle's incircle**s**!)
  5. UVa 00438 - The Circumference of ... (triangle's circumcircle)
  6. UVa 10195 - The Knights Of The ... (triangle's incircle, Heron's formula)
  7. UVa 10210 - Romeo & Juliet (basic trigonometry)
  8. UVa 10286 - The Trouble with a ... (Law of Sines)
  9. UVa 10347 - Medians (given 3 medians of a triangle, find its area)
  10. UVa 10387 - Billiard (expanding surface, *trigonometry*)
  11. UVa 10522 - Height to Area (derive the formula, uses Heron's formula)
  12. **UVa 10577 - Bounding box *** (get center+radius of outer circle from 3 points, get all vertices, get the min-x/max-x/min-y/max-y of the polygon)
  13. *UVa 10792 - The Laurel-Hardy Story* (derive the trigonometry formulas)
  14. UVa 10991 - Region (Heron's formula, Law of Cosines, area of sector)
  15. **UVa 11152 - Colourful ... *** (triangle's (in/circum)circle; Heron's formula)
  16. *UVa 11164 - Kingdom Division* (use Triangle properties)
  17. *UVa 11281 - Triangular Pegs in ...* (the min bounding circle of a non obtuse triangle is its circumcircle; if the triangle is obtuse, the the radii of the min bounding circle is the largest side of the triangle)
  18. *UVa 11326 - Laser Pointer* (trigonometry, tangent, reflection trick)
  19. *UVa 11437 - Triangle Fun* (hint: $\frac{1}{7}$)
  20. UVa 11479 - Is this the easiest problem? (property check)
  21. UVa 11579 - Triangle Trouble (sort; greedily check if three successive sides satisfy triangle inequality and if it is the largest triangle found so far)
  22. UVa 11854 - Egypt (Pythagorean theorem/triple)
  23. **UVa 11909 - Soya Milk *** (Law of Sines (or tangent); two possible cases!)
  24. UVa 11936 - The Lazy Lumberjacks (see if 3 sides form a valid triangle)

- Quadrilaterals
  1. UVa 00155 - All Squares (recursive counting)
  2. **UVa 00460 - Overlapping Rectangles *** (rectangle-rectangle intersection)
  3. UVa 00476 - Points in Figures: ... (similar to UVa 477 and 478)
  4. UVa 00477 - Points in Figures: ... (similar to UVa 476 and 478)
  5. **UVa 11207 - The Easiest Way *** (cutting rectangle into 4-equal-sized squares)
  6. UVa 11345 - Rectangles (rectangle-rectangle intersection)
  7. UVa 11455 - Behold My Quadrangle (property check)
  8. UVa 11639 - Guard the Land (rectangle-rectangle intersection, use flag array)
  9. *UVa 11800 - Determine the Shape* (use `next_permutation` to help you try all possible $4! = 24$ permutations of 4 points; check if they can satisfy square, rectangle, rhombus, parallelogram, trapezium, in that order)
  10. **UVa 11834 - Elevator *** (packing two circles in a rectangle)
  11. *UVa 12256 - Making Quadrilaterals* (LA 5001, KualaLumpur 10, start with three sides of 1, 1, 1, then the fourth side onwards must be the sum of the previous three to make a line; repeat until we reach the n-th side)

- 3D Objects
  1. **UVa 00737 - Gleaming the Cubes *** (cube and cube intersection)
  2. **UVa 00815 - Flooded *** (volume, greedy, sort by height, simulation)
  3. **UVa 10297 - Beavergnaw *** (cones, cylinders, volumes)

Profile of Algorithm Inventor

Pythagoras of Samos (≈ 500 BC) was a Greek mathematician and philosopher born on the island of Samos. He is best known for the Pythagorean theorem involving right triangle.

Euclid of Alexandria (≈ 300 BC) was a Greek mathematician, the 'Father of Geometry'. He was from the city of Alexandria. His most influential work in mathematics (especially geometry) is the 'Elements'. In the 'Elements', Euclid deduced the principles of what is now called Euclidean geometry from a small set of axioms.

Heron of Alexandria (≈ 10-70 AD) was an ancient Greek mathematician from the city of Alexandria, Roman Egypt—the same city as Euclid. His name is closely associated with his formula for finding the area of a triangle from its side lengths.

Ronald Lewis Graham (born 1935) is an American mathematician. In 1972, he invented the Graham's scan algorithm for finding convex hull of a finite set of points in the plane. There are now many other algorithm variants and improvements for finding convex hull.

7.3 Algorithm on Polygon with Libraries

Polygon is a plane figure that is bounded by a closed path (path that starts and ends at the same vertex) composed of a finite sequence of straight line segments. These segments are called edges or sides. The point where two edges meet is the polygon's vertex or corner. Polygon is a source of many (computational) geometry problems as it allows the problem author to present more realistic objects than the ones discussed in Section 7.2.

7.3.1 Polygon Representation

The standard way to represent a polygon is to simply enumerate the vertices of the polygon in either clockwise or counter clockwise order, with the first vertex being equal to the last vertex (some of the functions mentioned later in this section require this arrangement, see **Exercise 7.3.4.1***). In this book, our default vertex ordering is counter clockwise. The resulting polygon after executing the code below is shown in Figure 7.8—right.

```
// 6 points, entered in counter clockwise order, 0-based indexing
vector<point> P;
P.push_back(point(1, 1));  // P0
P.push_back(point(3, 3));  // P1
P.push_back(point(9, 1));  // P2
P.push_back(point(12, 4)); // P3
P.push_back(point(9, 7));  // P4
P.push_back(point(1, 7));  // P5
P.push_back(P[0]); // important: loop back
```

7.3.2 Perimeter of a Polygon

The perimeter of a polygon (either convex or concave) with n vertices given in some order (either clockwise or counter-clockwise) can be computed via this simple function below.

```
// returns the perimeter, which is the sum of Euclidian distances
// of consecutive line segments (polygon edges)
double perimeter(const vector<point> &P) {
  double result = 0.0;
  for (int i = 0; i < (int)P.size()-1; i++)  // remember that P[0] = P[n-1]
    result += dist(P[i], P[i+1]);
  return result; }
```

7.3.3 Area of a Polygon

The signed area A of (either convex or concave) polygon with n vertices given in some order (either clockwise or counter-clockwise) can be found by computing the determinant of the matrix as shown below. This formula can be easily written into the library code.

$$A = \frac{1}{2} \times \begin{bmatrix} x_0 & y_0 \\ x_1 & y_1 \\ x_2 & y_2 \\ \ldots & \ldots \\ x_{n-1} & y_{n-1} \end{bmatrix} = \frac{1}{2} \times (x_0 \times y_1 + x_1 \times y_2 + \ldots + x_{n-1} \times y_0 - x_1 \times y_0 - x_2 \times y_1 - \ldots - x_0 \times y_{n-1})$$

```
// returns the area, which is half the determinant
double area(const 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;
    y1 = P[i].y; y2 = P[i+1].y;
    result += (x1 * y2 - x2 * y1);
  }
  return fabs(result) / 2.0; }
```

7.3.4 Checking if a Polygon is Convex

A polygon is said to be **Convex** if any line segment drawn inside the polygon does not intersect any edge of the polygon. Otherwise, the polygon is called **Concave**.

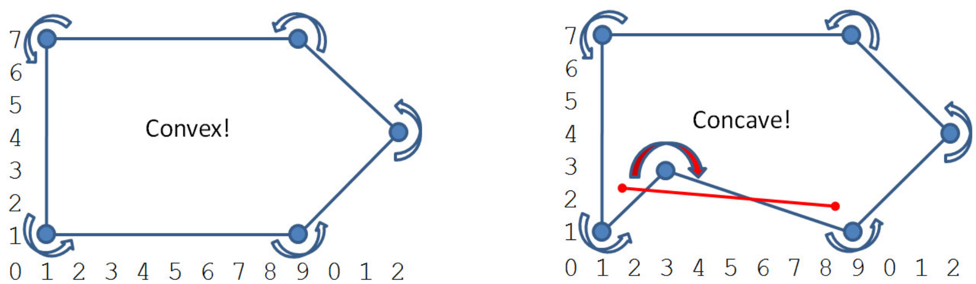


Figure 7.8: Left: Convex Polygon, Right: Concave Polygon

However, to test if a polygon is convex, there is an easier computational approach than "trying to check if all line segments can be drawn inside the polygon". We can simply check whether all three consecutive vertices of the polygon form the same turns (all left turns/ccw if the vertices are listed in counter clockwise order or all right turn/cw if the vertices are listed in clockwise order). If we can find at least one triple where this is false, then the polygon is concave (see Figure 7.8).

```
bool isConvex(const vector<point> &P) {        // returns true if all three
  int sz = (int)P.size();  // consecutive vertices of P form the same turns
  if (sz <= 3) return false;   // a point/sz=2 or a line/sz=3 is not convex
  bool isLeft = ccw(P[0], P[1], P[2]);               // remember one result
  for (int i = 1; i < sz-1; i++)            // then compare with the others
    if (ccw(P[i], P[i+1], P[(i+2) == sz ? 1 : i+2]) != isLeft)
      return false;            // different sign -> this polygon is concave
  return true; }                                  // this polygon is convex
```

Exercise 7.3.4.1*: Which part of the code above that you should modify to accept collinear points? Example: Polygon {(0,0), (2,0), (4,0), (2,2), (0,0)} should be treated as convex.

Exercise 7.3.4.2*: If the first vertex is not repeated as the last vertex, will the function `perimeter`, `area`, and `isConvex` presented as above work correctly?

7.3.5 Checking if a Point is Inside a Polygon

Another common test performed on a polygon P is to check if a point pt is inside or outside polygon P. The following function that implements 'winding number algorithm' allows such check for *either* convex or concave polygon. It works by computing the sum of angles between three points: $\{P[i], pt, P[i+1]\}$ where ($P[i]$-$P[i+1]$) are consecutive sides of polygon P, taking care of left turns (add the angle) and right turns (subtract the angle) respectively. If the final sum is 2π (360 degrees), then pt is inside polygon P (see Figure 7.9).

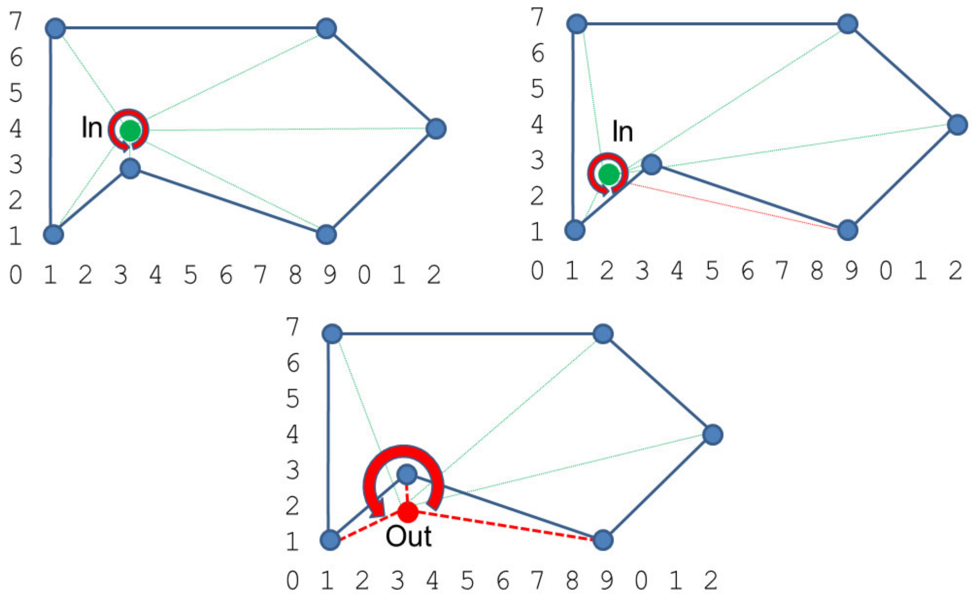


Figure 7.9: Top Left: inside, Top Right: also inside, Bottom: outside

```
// returns true if point p is in either convex/concave polygon P
bool inPolygon(point pt, const vector<point> &P) {
  if ((int)P.size() == 0) return false;
  double sum = 0;    // assume the first vertex is equal to the last vertex
  for (int i = 0; i < (int)P.size()-1; i++) {
    if (ccw(pt, P[i], P[i+1]))
         sum += angle(P[i], pt, P[i+1]);                   // left turn/ccw
    else sum -= angle(P[i], pt, P[i+1]); }                 // right turn/cw
  return fabs(fabs(sum) - 2*PI) < EPS; }
```

Exercise 7.9.1*: What happen to the `inPolygon` routine if point `pt` is on one of the edge of polygon `P`, e.g. `pt = P[0]` or `pt` is the mid-point between `P[0]` and `P[1]`, etc? What should be done to address that situation?

Exercise 7.9.2*: Discuss the pros and the cons of the following alternative methods for testing if a point is inside a polygon:

1. Triangulate a convex polygon into triangles and check if the sum of triangle areas equal to the area of the convex polygon.
2. Ray casting algorithm: We draw a ray from the point to any fixed direction so that the ray intersects the edge(s) of the polygon. If there are odd/even number of intersections, the point is inside/outside, respectively.

7.3.6 Cutting Polygon with a Straight Line

Another interesting thing that we can do with a *convex* polygon (see **Exercise 7.3.6.2*** for concave polygon) is to cut it into two convex sub-polygons with a straight line defined with two points a and b. See some programming exercises listed below that use this function.

The basic idea of the following `cutPolygon` routine is to iterate through the vertices of the original polygon Q one by one. If line ab and polygon vertex v form a left turn (which implies that v is on the left side of the line ab), we put v inside the new polygon P. Once we find a polygon edge that intersects with the line ab, we use that intersection point as part of the new polygon P (see Figure 7.10—left, point 'C'). We then skip the next few vertices of Q that are located on the right side of line ab. Sooner or later, we will revisit another polygon edge that intersect with line ab again (see Figure 7.10—left, point 'D' which happens to be one of the original vertex of polygon Q). We continue appending vertices of Q into P again because we are now on the left side of line ab again. We stop when we have returned to the starting vertex and returns the resulting polygon P (see Figure 7.10—right).

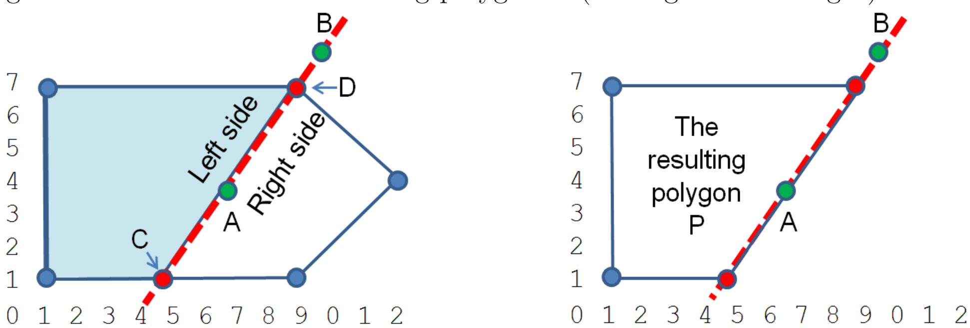


Figure 7.10: Left: Before Cut, Right: After Cut

```
// 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, const vector<point> &Q) {
  vector<point> P;
  for (int i = 0; i < (int)Q.size(); i++) {
    double left1 = cross(toVec(a, b), toVec(a, Q[i])), left2 = 0;
    if (i != (int)Q.size()-1) left2 = cross(toVec(a, b), toVec(a, Q[i+1]));
    if (left1 > -EPS) P.push_back(Q[i]);       // Q[i] is on the left of ab
    if (left1 * left2 < -EPS)        // edge (Q[i], Q[i+1]) crosses line ab
      P.push_back(lineIntersectSeg(Q[i], Q[i+1], a, b));
  }
  if (!P.empty() && !(P.back() == P.front()))
    P.push_back(P.front());        // make P's first point = P's last point
  return P; }
```

To further help readers to understand these algorithms on polygon, we have build a visualization tool for the third edition of this book. The reader can draw their own polygon and asks the tool to visually explain the algorithm on polygon discussed in this section.

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/polygon.html`

Exercise 7.3.6.1: This `cutPolygon` function only returns the left side of the polygon Q after cutting it with line ab. What should we do if we want the right side instead?

Exercise 7.3.6.2*: What happen if we run `cutPolygon` function on a *concave* polygon?

7.3.7 Finding the Convex Hull of a Set of Points

The **Convex Hull** of a set of points P is the smallest convex polygon $CH(P)$ for which each point in P is either on the boundary of $CH(P)$ or in its interior. Imagine that the points are nails on a flat 2D plane and we have a long enough rubber band that can enclose all the nails. If this rubber band is released, it will try to enclose as small an area as possible. That area is the area of the convex hull of these set of points/nails (see Figure 7.11). Finding convex hull of a set of points has natural applications in *packing* problems.

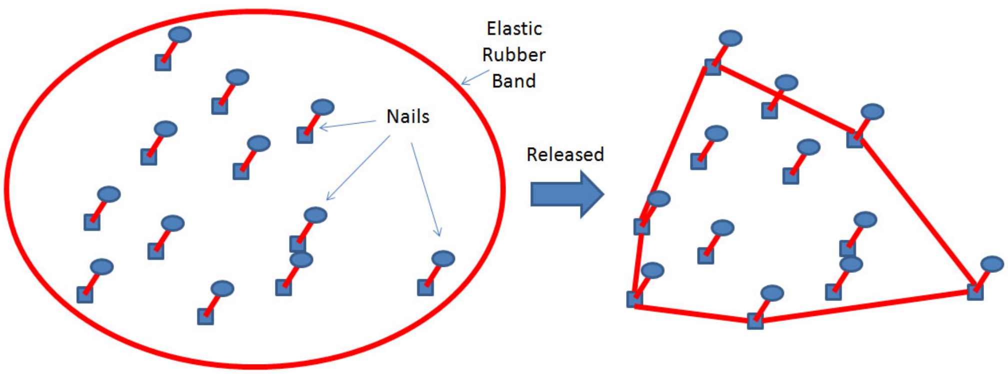


Figure 7.11: Rubber Band Analogy for Finding Convex Hull

As every vertex in $CH(P)$ is a vertex in the set of points P, the algorithm for finding convex hull is essentially an algorithm to decide which points in P should be chosen as part of the convex hull. There are several convex hull finding algorithms available. In this section, we choose the $O(n \log n)$ Ronald *Graham's Scan* algorithm.

Graham's scan algorithm first sorts all the n points of P where the first point does not have to be replicated as the last point (see Figure 7.12.A) based on their angles w.r.t a point called pivot. In our example, we pick the bottommost and rightmost point in P as pivot. After sorting based on angles w.r.t this pivot, we can see that edge 0-1, 0-2, 0-3, ..., 0-10, and 0-11 are in counter clockwise order (see point 1 to 11 w.r.t point 0 in Figure 7.12.B)!

```
point pivot(0, 0);
bool angleCmp(point a, point b) {                  // angle-sorting function
  if (collinear(pivot, a, b))                                // special case
    return dist(pivot, a) < dist(pivot, b);    // check which one is closer
  double d1x = a.x - pivot.x, d1y = a.y - pivot.y;
  double d2x = b.x - pivot.x, d2y = b.y - pivot.y;
  return (atan2(d1y, d1x) - atan2(d2y, d2x)) < 0; }   // compare two angles
```

```
vector<point> CH(vector<point> P) {   // the content of P may be reshuffled
  int i, j, n = (int)P.size();
  if (n <= 3) {
    if (!(P[0] == P[n-1])) P.push_back(P[0]); // safeguard from corner case
    return P; }                          // special case, the CH is P itself

  // first, find P0 = point with lowest Y and if tie: rightmost X
  int P0 = 0;
  for (i = 1; i < n; i++)
    if (P[i].y < P[P0].y || (P[i].y == P[P0].y && P[i].x > P[P0].x))
      P0 = i;

  point temp = P[0]; P[0] = P[P0]; P[P0] = temp;    // swap P[P0] with P[0]

  // second, sort points by angle w.r.t. pivot P0
  pivot = P[0];                    // use this global variable as reference
  sort(++P.begin(), P.end(), angleCmp);              // we do not sort P[0]
// to be continued
```

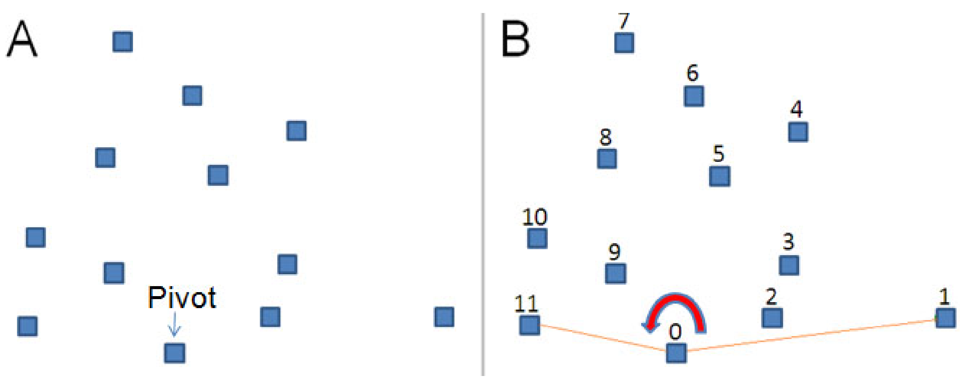


Figure 7.12: Sorting Set of 12 Points by Their Angles w.r.t a Pivot (Point 0)

Then, this algorithm maintains a stack S of candidate points. Each point of P is pushed *once* on to S and points that are not going to be part of $CH(P)$ will be eventually popped from S. Graham's Scan maintains this invariant: The top three items in stack S must always make a left turn (which is a basic property of a convex polygon).

Initially we insert these three points, point N-1, 0, and 1. In our example, the stack initially contains (bottom) 11-0-1 (top). This always form a left turn.

Now, examine Figure 7.13.C. Here, we try to insert point 2 and 0-1-2 is a left turn, so we accept point 2. Stack S is now (bottom) 11-0-1-2 (top).

Next, examine Figure 7.13.D. Here, we try to insert point 3 and 1-2-3 is a *right* turn. This means, if we accept the point before point 3, which is point 2, we will not have a convex polygon. So we have to pop point 2. Stack S is now (bottom) 11-0-1 (top) again. Then we re-try inserting point 3. Now 0-1-3, the *current* top three items in stack S form a left turn, so we accept point 3. Stack S is now (bottom) 11-0-1-3 (top).

We repeat this process until all vertices have been processed (see Figure 7.13.E-F-G-...-H). When Graham's Scan terminates, whatever that is left in S are the points of $CH(P)$ (see Figure 7.13.H, the stack contains (bottom) 11-0-1-4-7-10-11 (top)). Graham Scan's eliminates all the right turns! As three consecutive vertices in S always make left turns, we have a convex polygon.

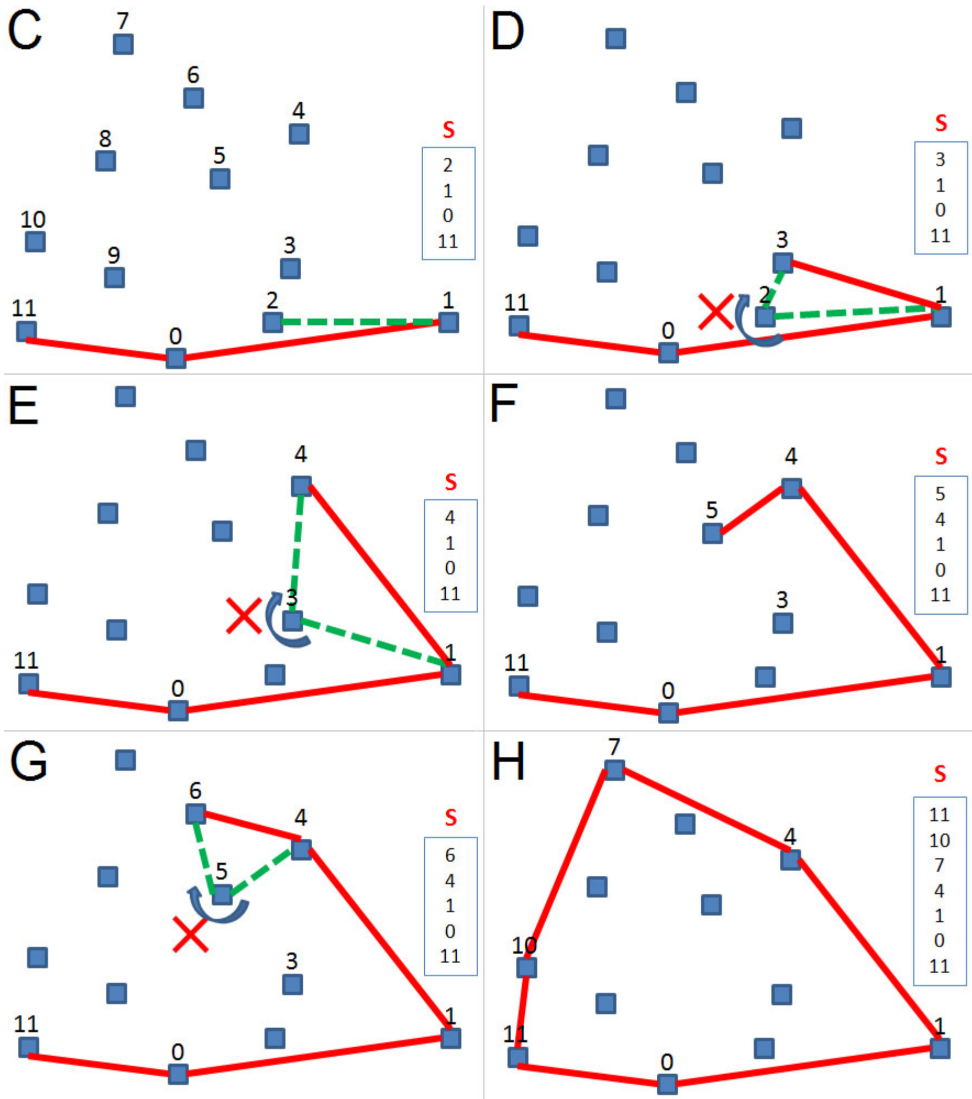


Figure 7.13: The Main Part of Graham's Scan algorithm

The implementation of Graham's Scan is shown below. We simply use a `vector<point> S` that behaves like a stack instead of using `stack<point> S`. The first part of Graham's Scan (finding the pivot) is just $O(n)$. The third part (the ccw tests) is also $O(n)$. This can be analyzed from the fact that each of the n vertices can only be pushed onto the stack once and popped from the stack once. The second part (sorts points by angle w.r.t pivot `P[0]`) is the *bulkiest* part that requires $O(n \log n)$. Overall, Graham's scan runs in $O(n \log n)$.

```
// continuation from the earlier part
  // third, the ccw tests
  vector<point> S;
  S.push_back(P[n-1]); S.push_back(P[0]); S.push_back(P[1]);   // initial S
  i = 2;                                              // then, we check the rest
  while (i < n) {            // note: N must be >= 3 for this method to work
    j = (int)S.size()-1;
    if (ccw(S[j-1], S[j], P[i])) S.push_back(P[i++]);  // left turn, accept
    else S.pop_back(); }   // or pop the top of S until we have a left turn
  return S; }                                           // return the result
```

We end this section and this chapter by pointing readers to another visualization tool, this time the visualization of several convex hull algorithms, including Graham's Scan, Andrew's Monotone Chain algorithm (see **Exercise 7.3.7.4***), and Jarvis's March algorithm. We also encourage readers to explore our source code to solve various programming exercises listed in this section.

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/convexhull.html`

Source code: `ch7_04_polygon.cpp/java`

Exercise 7.3.7.1: Suppose we have 5 points, $P = \{(0,0),(1,0),(2,0),(2,2),(0,2))\}$. The convex hull of these 5 points are actually these 5 points themselves (plus one, as we loop back to vertex (0, 0)). However, our Graham's scan implementation removes point (1, 0) as (0, 0)-(1, 0)-(2, 0) are collinear. Which part of the Graham's scan implementation that we have to modify to accept collinear points?

Exercise 7.3.7.2: In function `angleCmp`, there is a call to function: `atan2`. This function is used to compare the two angles but what is actually returned by `atan2`? Investigate!

Exercise 7.3.7.3*: Test the Graham's Scan code above: `CH(P)` on these corner cases. What is the convex hull of:

1. A single point, e.g. $P_1 = \{(0,0)\}$?
2. Two points (a line), e.g. $P_2 = \{(0,0),(1,0)\}$?
3. Three points (a triangle), e.g. $P_3 = \{(0,0),(1,0),(1,1)\}$?
4. Three points (a collinear line), e.g. $P_4 = \{(0,0),(1,0),(2,0)\}$?
5. Four points (a collinear line), e.g. $P_5 = \{(0,0),(1,0),(2,0),(3,0)\}$?

Exercise 7.3.7.4*: The Graham's Scan implementation above can be inefficient for large n as `atan2` is recalculated every time an angle comparison is made (and it is quite problematic when the angle is close to 90 degrees). Actually, the same basic idea of Graham's Scan also works if the input is sorted based on x-coordinate (and in case of a tie, by y-coordinate) instead of angle. The hull is now computed in 2 steps producing the *upper* and *lower* parts of the hull. This modification was devised by A. M. Andrew and known as Andrew's Monotone Chain Algorithm. It has the same basic properties as Graham's Scan but avoids costly comparisons between angles [9]. Investigate this algorithm and implement it!

Below, we provide a list of programming exercises related to polygon. Without pre-written library code discussed in this section, many of these problems look 'hard'. With the library code, they become manageable as the problem can now be decomposed into a few library routines. Spend some time to attempt them, especially the **must try *** ones.

Programming Exercises related to Polygon:

1. UVa 00109 - Scud Busters (find `CH`, test if point `inPolygon`, `area` of polygon)
2. *UVa 00137 - Polygons* (convex polygon intersection, line segment intersection, `inPolygon`, `CH`, `area`, inclusion-exclusion principle)
3. UVa 00218 - Moth Eradication (find `CH`, `perimeter` of polygon)
4. UVa 00361 - Cops and Robbers (check if a point is inside `CH` of Cop/Robber; if a point *pt* is inside a convex hull, then there is definitely a triangle formed using three vertices of the convex hull that contains *pt*)
5. UVa 00478 - Points in Figures: ... (`inPolygon`/inTriangle; if the given polygon P is *convex*, there is another way to check if a point *pt* is inside or outside P other than the way mentioned in this section; we can triangulate P into triangles with *pt* as one of the vertex, then sum the areas of the triangles; if it is the same as the area of polygon P, then *pt* is inside P; if it is larger, then *pt* is outside P)
6. *UVa 00596 - The Incredible Hull* (`CH`, output formatting is a bit tedious)
7. UVa 00634 - Polygon (`inPolygon`; the polygon can be convex or concave)
8. UVa 00681 - Convex Hull Finding (pure `CH` problem)
9. UVa 00858 - Berry Picking (ver line-polygon intersect; sort; alternating segments)
10. ***UVa 01111 - Trash Removal **** (LA 5138, World Finals Orlando11, `CH`, distance of each CH side—which is parallel to the side—to each vertex of the CH)
11. UVa 01206 - Boundary Points (LA 3169, Manila06, convex hull `CH`)
12. *UVa 10002 - Center of Mass?* (centroid, center of `CH`, `area` of polygon)
13. UVa 10060 - A Hole to Catch a Man (`area` of polygon)
14. UVa 10065 - Useless Tile Packers (find `CH`, `area` of polygon)
15. UVa 10112 - Myacm Triangles (test if point `inPolygon`/inTriangle, see UVa 478)
16. UVa 10406 - Cutting tabletops (vector, `rotate`, `translate`, then `cutPolygon`)
17. **UVa 10652 - Board Wrapping *** (`rotate`, `translate`, `CH`, `area`)
18. UVa 11096 - Nails (very classic `CH` problem, start from here)
19. **UVa 11265 - The Sultan's Problem *** (`cutPolygon`, `inPolygon`, `area`)
20. UVa 11447 - Reservoir Logs (`area` of polygon)
21. UVa 11473 - Campus Roads (`perimeter` of polygon)
22. UVa 11626 - Convex Hull (find `CH`, be careful with collinear points)

7.4 Solution to Non-Starred Exercises

Exercise 7.2.1.1: 5.0.

Exercise 7.2.1.2: (-3.0, 10.0).

Exercise 7.2.1.3: (-0.674, 10.419).

Exercise 7.2.2.1: The line equation $y = mx + c$ cannot handle all cases: Vertical lines has 'infinite' gradient/slope in this equation and 'near vertical' lines are also problematic. If we use this line equation, we have to treat vertical lines separately in our code which decreases the probability of acceptance. Fortunately, this can be avoided by using the better line equation $ax + by + c = 0$.

Exercise 7.2.2.2: -0.5 * x + 1.0 * y - 1.0 = 0.0

Exercise 7.2.2.3: 1.0 * x + 0.0 * y - 2.0 = 0.0. If you use the $y = mx + c$ line equation, you will have $x = 2.0$ instead, but you cannot represent a vertical line using this form $y =?$.

Exercise 7.2.2.4: Given 2 points $(x1, y1)$ and $(x2, y2)$, the slope can be calculated with $m = (y2-y1)/(x2-x1)$. Subsequently the y-intercept c can be computed from the equation by substitution of the values of a point (either one) and the line gradient m. The code will looks like this. See that we have to deal with vertical line separately and awkwardly.

```
struct line2 { double m, c; };      // another way to represent a line

int pointsToLine2(point p1, point p2, line2 &l) {
 if (p1.x == p2.x) {                          // special case: vertical line
   l.m = INF;                       // l contains m = INF and c = x_value
   l.c = p1.x;                     // to denote vertical line x = x_value
   return 0;   // we need this return variable to differentiate result
 }
 else {
   l.m = (double)(p1.y - p2.y) / (p1.x - p2.x);
   l.c = p1.y - l.m * p1.x;
   return 1;     // l contains m and c of the line equation y = mx + c
} }
```

Exercise 7.2.2.5:

```
// convert point and gradient/slope to line
void pointSlopeToLine(point p, double m, line &l) {
  l.a = -m;                                                // always -m
  l.b = 1;                                                  // always 1
  l.c = -((l.a * p.x) + (l.b * p.y)); }                 // compute this
```

Exercise 7.2.2.6: (5.0, 3.0).

Exercise 7.2.2.7: (4.0, 2.5).

Exercise 7.2.2.8: (-3.0, 5.0).

Exercise 7.2.2.9: (0.0, 4.0). The result is different from **Exercise 7.2.2.8**. 'Translate then Rotate' is different from 'Rotate then Translate'. Be careful in sequencing them.

Exercise 7.2.2.10: (1.0, 2.0). If the rotation center is not origin, we need to translate the input point c (3, 2) by a vector described by $-p$, i.e. (-2, -1) to point c' (1, 1). Then, we perform the 90 degrees counter clockwise rotation around origin to get c'' (-1, 1). Finally, we translate c'' to the final answer by a vector described by p to point (1, 2).

Exercise 7.2.2.11: The solution is shown below:

```
void closestPoint(line l, point p, point &ans) {
  line perpendicular;         // perpendicular to l and pass through p
  if (fabs(l.b) < EPS) {              // special case 1: vertical line
    ans.x = -(l.c);   ans.y = p.y;      return; }

  if (fabs(l.a) < EPS) {            // special case 2: horizontal line
    ans.x = p.x;      ans.y = -(l.c);   return; }

  pointSlopeToLine(p, 1 / l.a, perpendicular);          // normal line
  // intersect line l with this perpendicular line
  // the intersection point is the closest point
  areIntersect(l, perpendicular, ans); }
```

Exercise 7.2.2.12: The solution is shown below. Other solution exists:

```
// returns the reflection of point on a line
void reflectionPoint(line l, point p, point &ans) {
  point b;
  closestPoint(l, p, b);                       // similar to distToLine
  vec v = toVector(p, b);                             // create a vector
  ans = translate(translate(p, v), v); }         // translate p twice
```

Exercise 7.2.2.13: 63.43 degrees.

Exercise 7.2.2.14: Point p (3,7) → point q (11,13) → point r (35,30) form a right turn. Therefore, point p is on the right side of a line that passes through point p and point r. Note: If point r is at (35, 31), then p, q, r are collinear.

Exercise 7.2.3.1: See Figure 7.14 below.

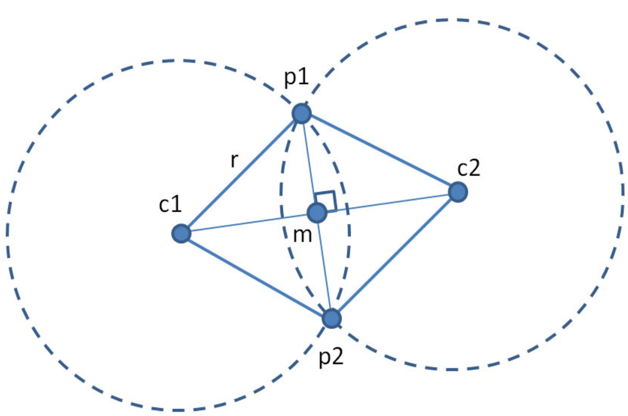


Figure 7.14: Explanation for Circle Through 2 Points and Radius

Let $c1$ and $c2$ be the centers of the 2 possible circles that go through 2 given points $p1$ and $p2$ and have radius r. The quadrilateral $p1 - c2 - p2 - c1$ is a rhombus, since its four sides are equal. Let m be the intersection of the 2 diagonals of the rhombus $p1 - c2 - p2 - c1$. According to the property of a rhombus, m bisects the 2 diagonals, and the 2 diagonals are perpendicular to each other. We realize that $c1$ and $c2$ can be calculated by scaling the vectors $mp1$ and $mp2$ by an appropriate ratio ($mc1/mp1$) to get the same magnitude as $mc1$, then rotating the points $p1$ and $p2$ around m by 90 degrees. In the implementation given in **Exercise 7.2.3.1**, the variable h is *half* the ratio $mc1/mp1$ (one can work out on paper why h can be calculated as such). In the 2 lines calculating the coordinates of one of the centers, the first operands of the additions are the coordinates of m, while the second operands of the additions are the result of scaling and rotating the vector $mp2$ around m.

Exercise 7.2.4.1: We can use double data type that has larger range. However, to further reduce the chance of overflow, we can rewrite the Heron's formula into $A = sqrt(s) \times sqrt(s-a) \times sqrt(s-b) \times sqrt(s-c))$. However, the result will be slightly less precise as we call $sqrt$ 4 times instead of once.

Exercise 7.3.6.1: Swap point a and b when calling `cutPolygon(a, b, Q)`.

Exercise 7.3.7.1: Edit the `ccw` function to accept collinear points.

Exercise 7.3.7.2: The function `atan2` computes the inverse tangent of $\frac{y}{x}$ using the signs of arguments to correctly determine quadrant.

7.5 Chapter Notes

Some material in this chapter are derived from the material courtesy of **Dr Cheng Holun, Alan** from School of Computing, National University of Singapore. Some library functions are customized from **Igor Naverniouk**'s library: `http://shygypsy.com/tools/`.

Compared to the first edition of this book, this chapter has, just like Chapter 5 and 6, grown to about twice its original size. However, the material mentioned here are still far from complete, especially for ICPC contestants. If you are preparing for ICPC, it is a good idea to dedicate one person in your team to study this topic in depth. This person should master basic geometry formulas and advanced computational geometry techniques, perhaps by reading relevant chapters in the following books: [50, 9, 7]. But not just the theory, he must also train himself to code *robust* geometry solutions that are able to handle degenerate (special) cases and precision errors.

The other computational geometry techniques that have not been discussed yet in this chapter are the **plane sweep** technique, intersection of **other geometric objects** including line segment-line segment intersection, various Divide and Conquer solutions for several classical geometry problems: **The Closest Pair Problem**, **The Furthest Pair Problem**, **Rotating Calipers** algorithm, etc. Some of these problems are discussed in Chapter 9.

Statistics	First Edition	Second Edition	Third Edition
Number of Pages	13	22 (+69%)	29 (+32%)
Written Exercises	-	20	22+9*=31 (+55%)
Programming Exercises	96	103 (+7%)	96 (-7%)

The breakdown of the number[16] of programming exercises from each section is shown below:

Section	Title	Appearance	% in Chapter	% in Book
7.2	**Basic Geometry Objects ...**	74	77%	4%
7.3	**Algorithm on Polygon ...**	22	23%	1%

[16]The total decreases a bit although we have added several new problems because some of the problems are moved to Chapter 8

L-R: Mr Raymond, Felix, Andrian, Andoko @ ACM ICPC World Finals, Tokyo 2007