13. Arithmetic. Before we lunge into the world of geometric algorithms, let's build up some confidence by polishing off some subroutines that will be needed to ensure correct results. We assume that **long** integers are less than 2^{31}.

First is a routine to calculate $s = \lfloor 2^{10}\sqrt{x} + \frac{1}{2} \rfloor$, the nearest integer to 2^{10} times the square root of a given nonnegative integer x. If $x > 0$, this is the unique integer such that $2^{20}x - s \le s^2 < 2^{20}x + s$.

The following routine appears to work by magic, but the mystery goes away when one considers the invariant relations

$$m = \lfloor 2^{2k-21} \rfloor, \qquad 0 < y = \lfloor 2^{20-2k}x \rfloor - s^2 + s \le q = 2s.$$

(Exception: We might actually have $y = 0$ for a short time when $q = 2$.)

```
⟨ Subroutines for arithmetic 13 ⟩ ≡
  long int_sqrt(x)
      long x;
  { register long y, m, q = 2;
    long k;
    if (x ≤ 0) return 0;
    for (k = 25, m = #20000000; x < m; k−−, m ≫= 2) ;      /* find the range */
    if (x ≥ m + m) y = 1;
    else y = 0;
    do ⟨ Decrease k by 1, maintaining the invariant relations between x, y, m, and q 14 ⟩ while (k);
    return q ≫ 1;
  }
```

See also sections 15 and 24.

This code is used in section 4.

14. ⟨ Decrease k by 1, maintaining the invariant relations between x, y, m, and q 14 ⟩ ≡

```
  {
    if (x & m) y += y + 1;
    else y += y;
    m ≫= 1;
    if (x & m) y += y − q + 1;
    else y += y − q;
    q += q;
    if (y > q) y −= q, q += 2;
    else if (y ≤ 0) q −= 2, y += q;
    m ≫= 1;
    k−−;
  }
```

This code is used in section 13.

15. We are going to need multiple-precision arithmetic in order to calculate certain geometric predicates properly, but it turns out that we do not need to implement general-purpose subroutines for bignums. It suffices to have a single special-purpose routine called $sign\_test(x1, x2, x3, y1, y2, y3)$, which computes a single-precision integer having the same sign as the dot product

$$x1 * y1 + x2 * y2 + x3 * y3$$

when we have $-2^{29} < x1, x2, x3 < 2^{29}$ and $0 \le y1, y2, y3 < 2^{29}$.

```
⟨ Subroutines for arithmetic 13 ⟩ +≡
  long sign_test(x1, x2, x3, y1, y2, y3)
       long x1, x2, x3, y1, y2, y3;
  { long s1, s2, s3;    /* signs of individual terms */
    long a, b, c;    /* components of a redundant representation of the dot product */
    register long t;    /* temporary register for swapping */
    ⟨ Determine the signs of the terms 16 ⟩;
    ⟨ If the answer is obvious, return it without further ado; otherwise, arrange things so that x3 * y3 has
         the opposite sign to x1 * y1 + x2 * y2 17 ⟩;
    ⟨ Compute a redundant representation of x1 * y1 + x2 * y2 + x3 * y3 18 ⟩;
    ⟨ Return the sign of the redundant representation 19 ⟩;
  }
```

16. ⟨ Determine the signs of the terms 16 ⟩ ≡

```
  if (x1 ≡ 0 ∨ y1 ≡ 0) s1 = 0;
  else {
    if (x1 > 0) s1 = 1;
    else x1 = −x1, s1 = −1;
  }
  if (x2 ≡ 0 ∨ y2 ≡ 0) s2 = 0;
  else {
    if (x2 > 0) s2 = 1;
    else x2 = −x2, s2 = −1;
  }
  if (x3 ≡ 0 ∨ y3 ≡ 0) s3 = 0;
  else {
    if (x3 > 0) s3 = 1;
    else x3 = −x3, s3 = −1;
  }
```

This code is used in section 15.

17. The answer is obvious unless one of the terms is positive and one of the terms is negative.

⟨ If the answer is obvious, return it without further ado; otherwise, arrange things so that $x3 * y3$ has the opposite sign to $x1 * y1 + x2 * y2$ 17 ⟩ ≡

```
if ((s1 ≥ 0 ∧ s2 ≥ 0 ∧ s3 ≥ 0) ∨ (s1 ≤ 0 ∧ s2 ≤ 0 ∧ s3 ≤ 0)) return (s1 + s2 + s3);
if (s3 ≡ 0 ∨ s3 ≡ s1) {
  t = s3; s3 = s2; s2 = t;
  t = x3; x3 = x2; x2 = t;
  t = y3; y3 = y2; y2 = t;
} else if (s3 ≡ s2) {
  t = s3; s3 = s1; s1 = t;
  t = x3; x3 = x1; x1 = t;
  t = y3; y3 = y1; y1 = t;
}
```

This code is used in section 15.

18. We make use of a redundant representation $2^{28}a + 2^{14}b + c$, which can be computed by brute force. (Everything is understood to be multiplied by $-s3$.)

⟨ Compute a redundant representation of $x1 * y1 + x2 * y2 + x3 * y3$ 18 ⟩ ≡

```
{ register long lx, rx, ly, ry;
  lx = x1/#4000; rx = x1 % #4000;    /* split off the least significant 14 bits */
  ly = y1/#4000; ry = y1 % #4000;
  a = lx * ly; b = lx * ry + ly * rx; c = rx * ry;
  lx = x2/#4000; rx = x2 % #4000;
  ly = y2/#4000; ry = y2 % #4000;
  a += lx * ly; b += lx * ry + ly * rx; c += rx * ry;
  lx = x3/#4000; rx = x3 % #4000;
  ly = y3/#4000; ry = y3 % #4000;
  a −= lx * ly; b −= lx * ry + ly * rx; c −= rx * ry;
}
```

This code is used in section 15.

19. Here we use the fact that $|c| < 2^{29}$.

⟨ Return the sign of the redundant representation 19 ⟩ ≡

```
  if (a ≡ 0) goto ez;
  if (a < 0) a = −a, b = −b, c = −c, s3 = −s3;
  while (c < 0) {
    a−−; c += #10000000;
    if (a ≡ 0) goto ez;
  }
  if (b ≥ 0) return −s3;    /* the answer is clear when a > 0 ∧ b ≥ 0 ∧ c ≥ 0 */
  b = −b;
  a −= b/#4000;
  if (a > 0) return −s3;
  if (a ≤ −2) return s3;
  return −s3 * ((a * #4000 − b % #4000) * #4000 + c);
ez: if (b ≥ #8000) return −s3;
  if (b ≤ −#8000) return s3;
  return −s3 * (b * #4000 + c);
```

This code is used in section 15.

20. Determinants. The *delaunay* routine bases all of its decisions on two geometric predicates, which depend on whether certain determinants are positive or negative.

The first predicate, $ccw(u,v,w)$, is true if and only if the three points (u,v,w) have a counterclockwise orientation. This means that if we draw the unique circle through those points, and if we travel along that circle in the counterclockwise direction starting at u, we will encounter v before w.

It turns out that $ccw(u,v,w)$ holds if and only if the determinant

$$\begin{vmatrix} x_u & y_u & 1 \\ x_v & y_v & 1 \\ x_w & y_w & 1 \end{vmatrix} = \begin{vmatrix} x_u - x_w & y_u - y_w \\ x_v - x_w & y_v - y_w \end{vmatrix}$$

is positive. The evaluation must be exact; if the answer is zero, a special tie-breaking rule must be used because the three points were collinear. The tie-breaking rule is tricky (and necessarily so, according to the theory in *Axioms and Hulls*).

Integer evaluation of that determinant will not cause **long** integer overflow, because we have assumed that all x and y coordinates lie between 0 and $2^{14}-1$, inclusive. In fact, we could go up to $2^{15}-1$ without risking overflow; but the limitation to 14 bits will be helpful when we consider a more complicated determinant below.

```
⟨ Other subroutines 12 ⟩ +≡
  long ccw(u, v, w)
      Vertex *u, *v, *w;
  { register long wx = w→x_coord, wy = w→y_coord;      /* x_w, y_w */
    register long det = (u→x_coord − wx) * (v→y_coord − wy) − (u→y_coord − wy) * (v→x_coord − wx);
    Vertex *t;
    if (det ≡ 0) {
      det = 1;
      if (u→z_coord > v→z_coord) {
        t = u; u = v; v = t; det = −det;
      }
      if (v→z_coord > w→z_coord) {
        t = v; v = w; w = t; det = −det;
      }
      if (u→z_coord > v→z_coord) {
        t = u; u = v; v = t; det = −det;
      }
      if (u→x_coord > v→x_coord ∨ (u→x_coord ≡ v→x_coord ∧
            (u→y_coord > v→y_coord ∨ (u→y_coord ≡ v→y_coord ∧
            (w→x_coord > u→x_coord ∨ (w→x_coord ≡ u→x_coord ∧ w→y_coord ≥ u→y_coord)))))) det = −det;
    }
    return (det > 0);
  }
```

21. The other geometric predicate, *incircle*(t,u,v,w), is true if and only if point t lies outside the circle passing through u, v, and w, assuming that *ccw*(u,v,w) holds. This predicate makes us work harder, because it is equivalent to the sign of a 4×4 determinant that requires twice as much precision:

$$\begin{vmatrix} x_t & y_t & x_t^2+y_t^2 & 1 \\ x_u & y_u & x_u^2+y_u^2 & 1 \\ x_v & y_v & x_v^2+y_v^2 & 1 \\ x_w & y_w & x_w^2+y_w^2 & 1 \end{vmatrix} = \begin{vmatrix} x_t-x_w & y_t-y_w & (x_t-x_w)^2+(y_t-y_w)^2 \\ x_u-x_w & y_u-y_w & (x_u-x_w)^2+(y_u-y_w)^2 \\ x_v-x_w & y_v-y_w & (x_v-x_w)^2+(y_v-y_w)^2 \end{vmatrix}.$$

The sign can, however, be deduced by the *sign_test* subroutine we had the foresight to provide earlier.

```
⟨ Other subroutines 12 ⟩ +≡
long incircle(t, u, v, w)
    Vertex *t, *u, *v, *w;
{ register long wx = w→x_coord, wy = w→y_coord;    /* x_w, y_w */
  long tx = t→x_coord − wx, ty = t→y_coord − wy;    /* x_t − x_w, y_t − y_w */
  long ux = u→x_coord − wx, uy = u→y_coord − wy;    /* x_u − x_w, y_u − y_w */
  long vx = v→x_coord − wx, vy = v→y_coord − wy;    /* x_v − x_w, y_v − y_w */
  register long det = sign_test(tx * uy − ty * ux, ux * vy − uy * vx, vx * ty − vy * tx,
      vx * vx + vy * vy, tx * tx + ty * ty, ux * ux + uy * uy);
  Vertex *s;
  if (det ≡ 0) {
    ⟨ Sort (t, u, v, w) by ID number 22 ⟩;
    ⟨ Remove incircle degeneracy 23 ⟩;
  }
  return (det > 0);
}
```

22. ⟨ Sort (t,u,v,w) by ID number 22 ⟩ ≡

```
det = 1;
if (t→z_coord > u→z_coord) {
  s = t; t = u; u = s; det = −det;
}
if (v→z_coord > w→z_coord) {
  s = v; v = w; w = s; det = −det;
}
if (t→z_coord > v→z_coord) {
  s = t; t = v; v = s; det = −det;
}
if (u→z_coord > w→z_coord) {
  s = u; u = w; w = s; det = −det;
}
if (u→z_coord > v→z_coord) {
  s = u; u = v; v = s; det = −det;
}
```

This code is used in section 21.

23. By slightly perturbing the points, we can always make them nondegenerate, although the details are complicated. A sequence of 12 steps, involving up to four auxiliary functions

$$ff(t,u,v,w) = \begin{vmatrix} x_t - x_v & (x_t - x_w)^2 + (y_t - y_w)^2 - (x_v - x_w)^2 - (y_v - y_w)^2 \\ x_u - x_v & (x_u - x_w)^2 + (y_u - y_w)^2 - (x_v - x_w)^2 - (y_v - y_w)^2 \end{vmatrix},$$

$$gg(t,u,v,w) = \begin{vmatrix} y_t - y_v & (x_t - x_w)^2 + (y_t - y_w)^2 - (x_v - x_w)^2 - (y_v - y_w)^2 \\ y_u - y_v & (x_u - x_w)^2 + (y_u - y_w)^2 - (x_v - x_w)^2 - (y_v - y_w)^2 \end{vmatrix},$$

$$hh(t,u,v,w) = (x_u - x_t)(y_v - y_w),$$

$$jj(t,u,v,w) = (x_u - x_v)^2 + (y_u - y_w)^2 - (x_t - x_v)^2 - (y_t - y_w)^2,$$

does the trick, as explained in *Axioms and Hulls*.

$\langle$ Remove incircle degeneracy 23 $\rangle \equiv$

```
{ long dd;
  if ((dd = ff(t,u,v,w)) < 0 ∨ (dd ≡ 0 ∧
        ((dd = gg(t,u,v,w)) < 0 ∨ (dd ≡ 0 ∧
        ((dd = ff(u,t,w,v)) < 0 ∨ (dd ≡ 0 ∧
        ((dd = gg(u,t,w,v)) < 0 ∨ (dd ≡ 0 ∧
        ((dd = ff(v,w,t,u)) < 0 ∨ (dd ≡ 0 ∧
        ((dd = gg(v,w,t,u)) < 0 ∨ (dd ≡ 0 ∧
        ((dd = hh(t,u,v,w)) < 0 ∨ (dd ≡ 0 ∧
        ((dd = jj(t,u,v,w)) < 0 ∨ (dd ≡ 0 ∧
        ((dd = hh(v,t,u,w)) < 0 ∨ (dd ≡ 0 ∧
        ((dd = jj(v,t,u,w)) < 0 ∨ (dd ≡ 0 ∧ jj(t,w,u,v) < 0)))))))))))))))))))) det = −det;
}
```

This code is used in section 21.

24. ⟨ Subroutines for arithmetic 13 ⟩ +≡

```
long ff(t, u, v, w)
    Vertex *t, *u, *v, *w;
{ register long wx = w->x_coord, wy = w->y_coord;   /* x_w, y_w */
  long tx = t->x_coord - wx, ty = t->y_coord - wy;   /* x_t - x_w, y_t - y_w */
  long ux = u->x_coord - wx, uy = u->y_coord - wy;   /* x_u - x_w, y_u - y_w */
  long vx = v->x_coord - wx, vy = v->y_coord - wy;   /* x_v - x_w, y_v - y_w */
  return sign_test(ux - tx, vx - ux, tx - vx, vx * vx + vy * vy, tx * tx + ty * ty, ux * ux + uy * uy);
}
long gg(t, u, v, w)
    Vertex *t, *u, *v, *w;
{ register long wx = w->x_coord, wy = w->y_coord;   /* x_w, y_w */
  long tx = t->x_coord - wx, ty = t->y_coord - wy;   /* x_t - x_w, y_t - y_w */
  long ux = u->x_coord - wx, uy = u->y_coord - wy;   /* x_u - x_w, y_u - y_w */
  long vx = v->x_coord - wx, vy = v->y_coord - wy;   /* x_v - x_w, y_v - y_w */
  return sign_test(uy - ty, vy - uy, ty - vy, vx * vx + vy * vy, tx * tx + ty * ty, ux * ux + uy * uy);
}
long hh(t, u, v, w)
    Vertex *t, *u, *v, *w;
{
  return (u->x_coord - t->x_coord) * (v->y_coord - w->y_coord);
}
long jj(t, u, v, w)
    Vertex *t, *u, *v, *w;
{ register long vx = v->x_coord, wy = w->y_coord;
  return (u->x_coord - vx) * (u->x_coord - vx) + (u->y_coord - wy) * (u->y_coord - wy)
      - (t->x_coord - vx) * (t->x_coord - vx) - (t->y_coord - wy) * (t->y_coord - wy);
}
```