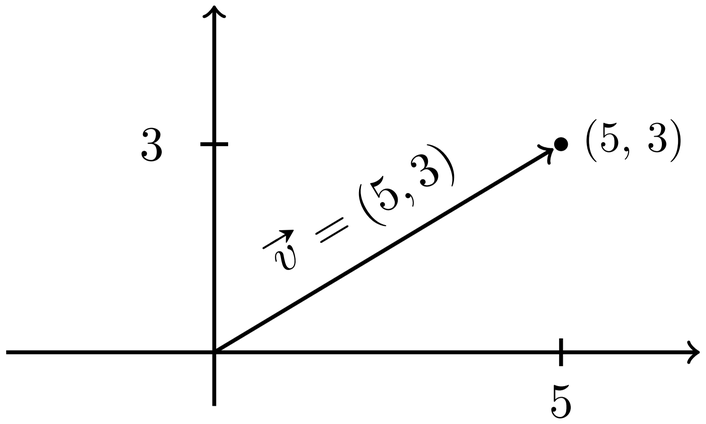
Geometría Computacional

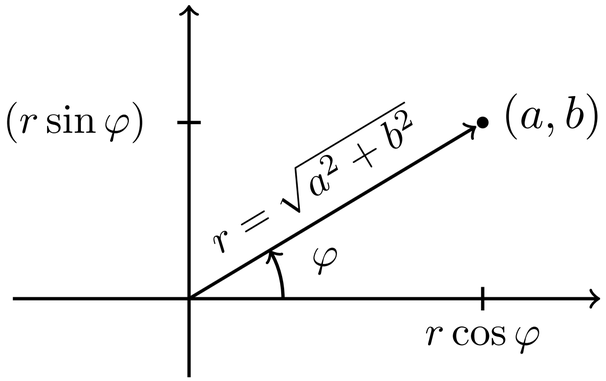
(para concursos de programación)

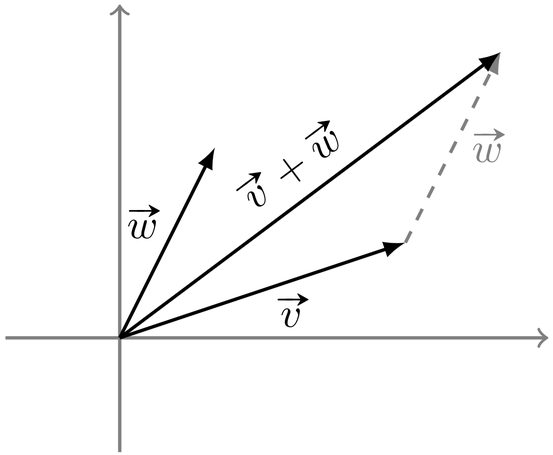
Alberto Verdejo

Prácticamente todo el material se ha obtenido del "Handbook of geometry for competitive programming", de Victor Lecomte, en `http://vlecomte.github.io/cp-geo.pdf`

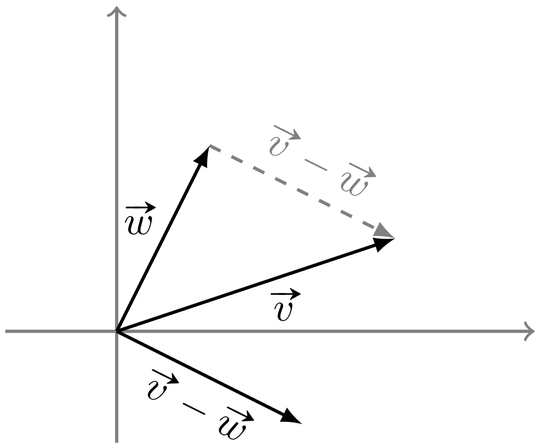
Puntos y vectores



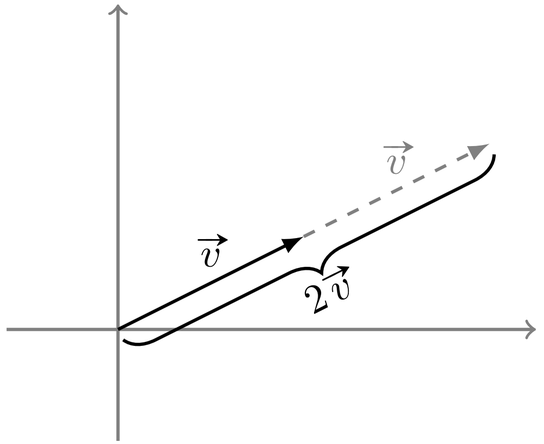




addition



subtraction



multiplication by scalar

Puntos y vectores

```
using T = double;  // using T = int;

struct pt {
   T x, y;
   pt operator+(pt p) const { return {x+p.x, y+p.y}; }
   pt operator-(pt p) const { return {x-p.x, y-p.y}; }
   pt operator*(T d) const { return {x*d, y*d}; }
   pt operator/(T d) const { return {x/d, y/d}; } // only for floating-point

   bool operator==(pt o) const { return x == o.x && y == o.y; }
   bool operator!=(pt o) const { return !(*this == o); }

   bool operator<(pt o) const { // sort points lexicographically
      if (x == o.x) return y < o.y;
      return x < o.x;
   }
//   bool operator<(pt o) const {   // #define EPS 1e-9
//       if (fabs(x - o.x) < EPS) return y < o.y;
//       return x < o.x;
//   }
};
```

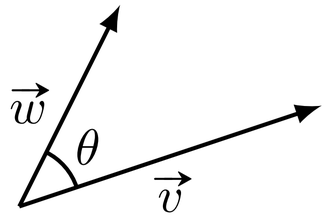
Puntos y vectores

```
#include <cmath>


T sq(pt v) { return v.x*v.x + v.y*v.y; }

double abs(pt v) { return sqrt(sq(v)); }



double dist(pt a, pt b) { // Euclidean distance
    return hypot(a.x - b.x, a.y - b.y);
}

T dist2(pt a, pt b) {
    return (a.x - b.x)*(a.x - b.x) + (a.y - b.y)*(a.y - b.y);
}
```

Producto escalar (dot product)

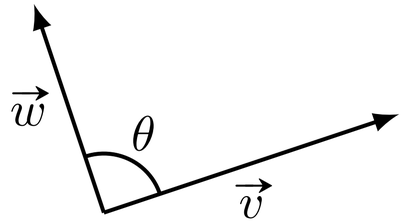
Medida de cómo de iguales son sus direcciones

$$\vec{v} \cdot \vec{w} = \|\vec{v}\|\|\vec{w}\| \cos\theta$$



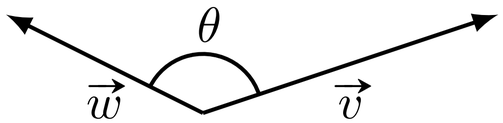
$\theta < \pi/2$

$\vec{v} \cdot \vec{w} = 5$



$\theta = \pi/2$

$\vec{v} \cdot \vec{w} = 0$



$\theta > \pi/2$

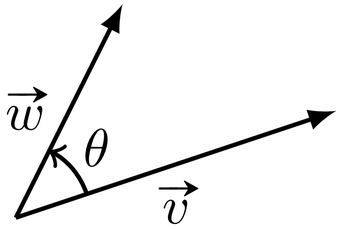
$\vec{v} \cdot \vec{w} = -5$

```
T dot(pt v, pt w) { return v.x*w.x + v.y*w.y; }

bool isPerp(pt v, pt w) { return dot(v,w) == 0; }

double angle(pt v, pt w) {  // [0..PI]
    double cosTheta = dot(v,w) / abs(v) / abs(w);
    return acos(max(-1.0, min(1.0, cosTheta)));
}
```

Producto vectorial (cross product)

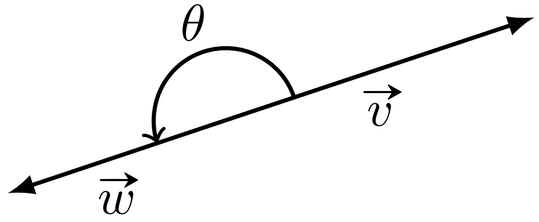
Medida de cómo de perpendiculares son

$$\vec{v} \times \vec{w} = \|\vec{v}\|\|\vec{w}\| \sin\theta$$



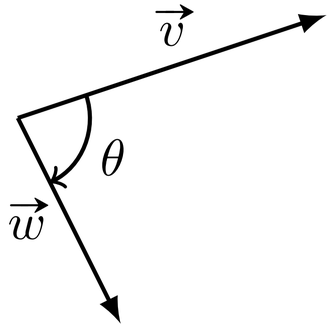
$0 < \theta < \pi$

$\vec{v} \times \vec{w} = 5$



$\theta = \pi$

$\vec{v} \times \vec{w} = 0$



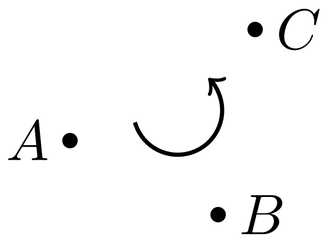
$-\pi < \theta < 0$

$\vec{v} \times \vec{w} = -7$

```
T cross(pt v, pt w) { return v.x*w.y - v.y*w.x; }
```

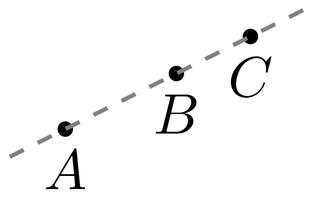
Orientación

$$\text{orient}(A, B, C) = \overrightarrow{AB} \times \overrightarrow{AC}$$



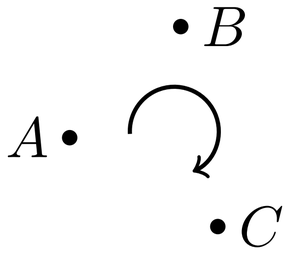
left turn

$\text{orient}(A, B, C) > 0$



collinear

$\text{orient}(A, B, C) = 0$

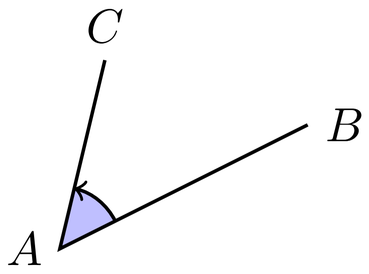


right turn

$\text{orient}(A, B, C) < 0$

```
// positivo/cero/negativo: c a la izquierda/sobre/derecha de a-b
T orient(pt a, pt b, pt c) { return cross(b - a, c - a); }
```

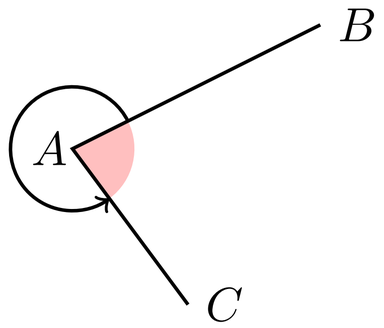
Orientación



$\text{orient}(A, B, C) > 0$

`angle()` $= 50°$

`orientedAngle()` $= 50°$



$\text{orient}(A, B, C) < 0$

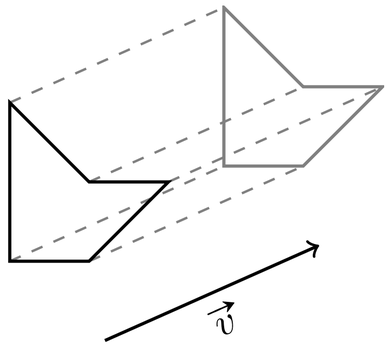
`angle()` $= 80°$

`orientedAngle()` $= 280°$

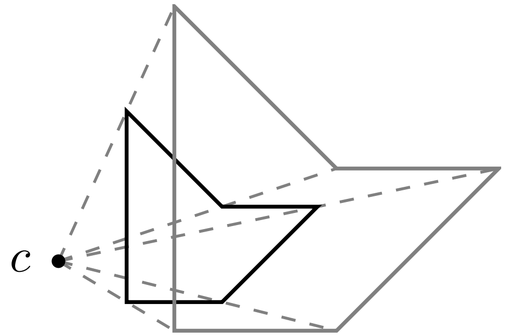
```
const double PI = acos(-1);

double orientedAngle(pt a, pt b, pt c) {
    if (orient(a, b, c) >= 0)
        return angle(b - a, c - a);
    else
        return 2*PI - angle(b - a, c - a);
}
```

Transformaciones

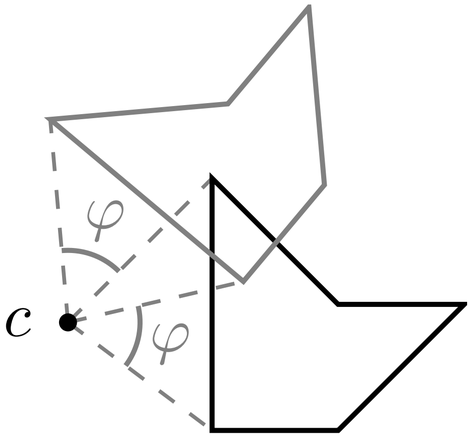
```
pt translate(pt p, pt v) { return p + v; }
```



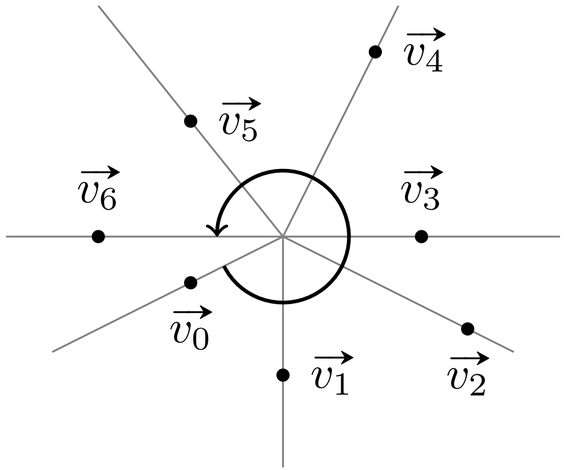

```
// scale p by a certain factor around a center c
pt scale(pt c, double factor, pt p) {
    return c + (p - c)*factor;
}
```

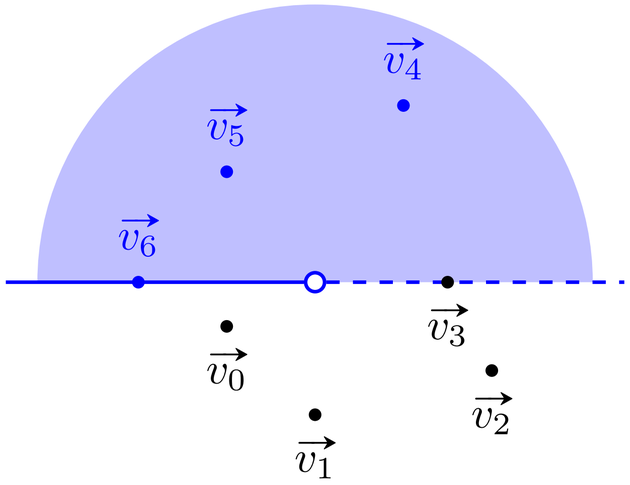



```
// rotate p by a certain angle a counter-clockwise around origin
pt rotate(pt p, double a) {
    return { p.x*cos(a) - p.y*sin(a), p.x*sin(a) + p.y*cos(a) };
}

// rotate 90º counterclockwise
pt perp(pt p) { return {-p.y, p.x}; }
```



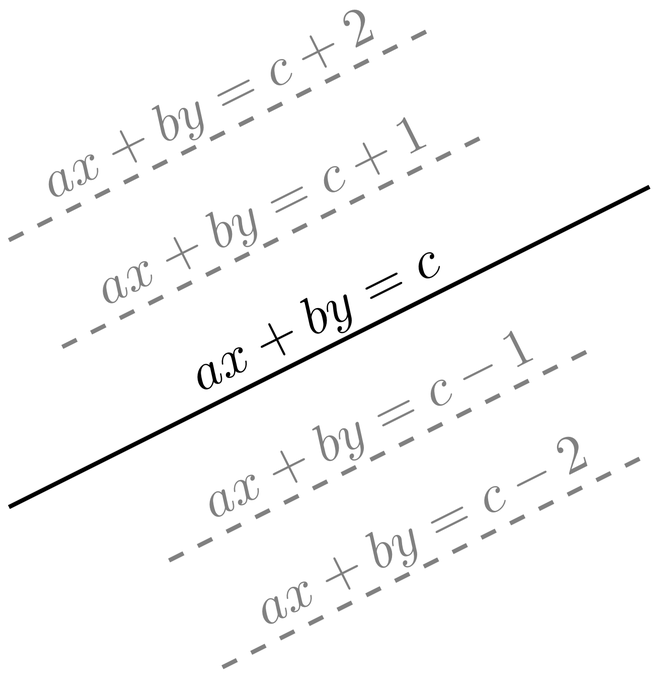
Polar sort

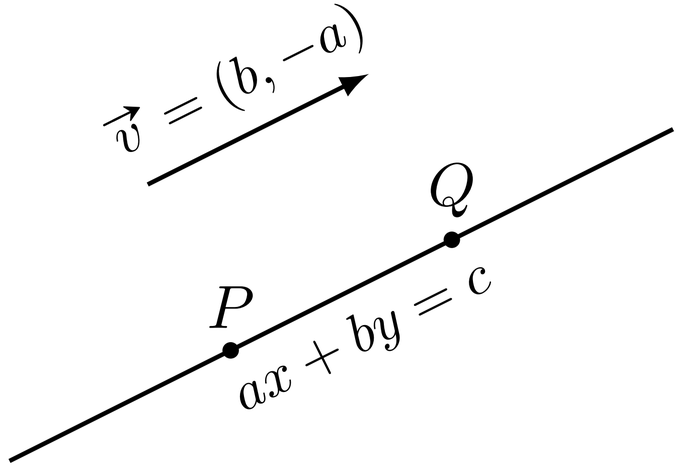





```
bool half(pt p) { // true if in blue half
    assert(p.x != 0 || p.y != 0); // the argument of (0,0) is undefined
    return p.y > 0 || (p.y == 0 && p.x < 0);
}

void polarSort(vector<pt> & v) {
   sort(v.begin(), v.end(), [](pt v, pt w) {
      return make_tuple(half(v), 0, sq(v))
           < make_tuple(half(w), cross(v,w), sq(w));
   });
}
```

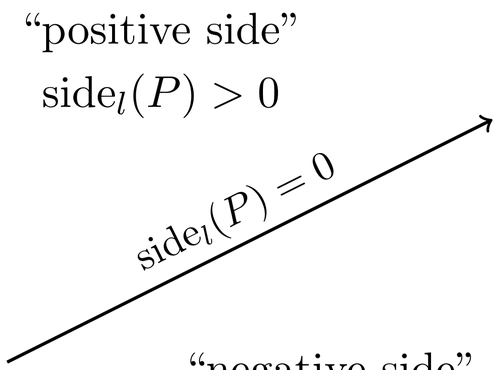
Líneas

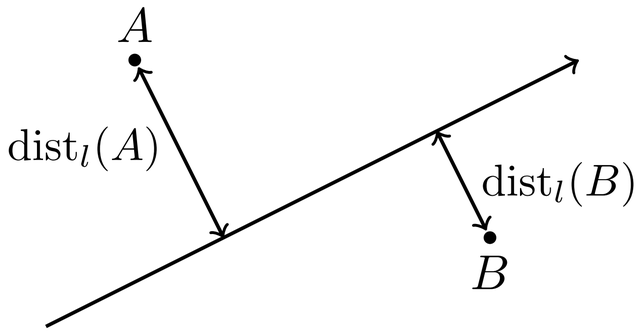


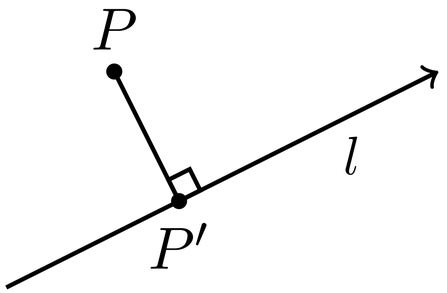


Líneas

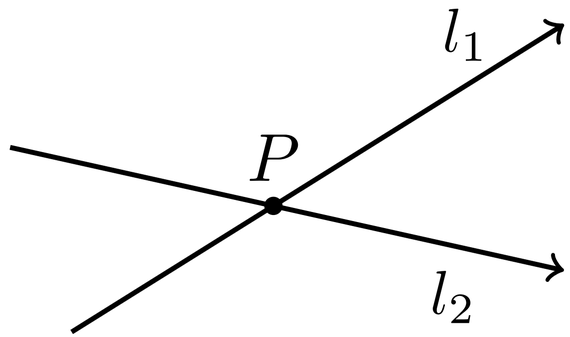
```
struct line {
    pt v; T c;
    // from direction vector v and offset c
    line(pt v, T c) : v(v), c(c) {}
    // from equation ax + by = c
    line(T a, T b, T c) : v({b,-a}), c(c) {}
    // from points p and q
    line(pt p, pt q) : v(q-p), c(cross(v,p)) {}

    T side(pt p) { return cross(v,p) - c; }

    double dist(pt p) { return abs(side(p)) / abs(v); }

    pt proj(pt p) { return p - perp(v)*side(p)/sq(v); }
};
```







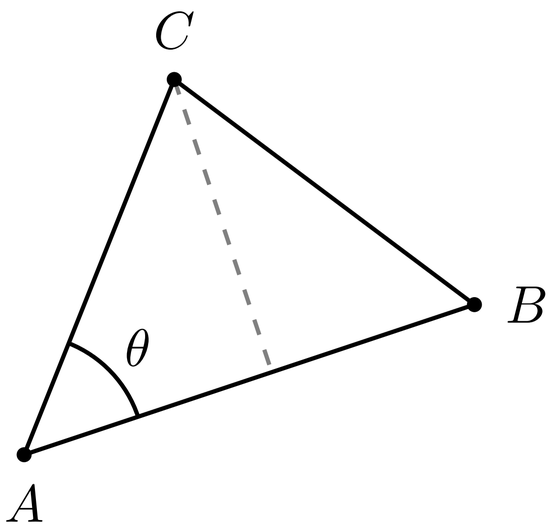
Intersección de líneas



$$P = \frac{c_{l_1}\overrightarrow{v_{l_2}} - c_{l_2}\overrightarrow{v_{l_1}}}{\overrightarrow{v_{l_1}} \times \overrightarrow{v_{l_2}}}$$

```
bool inter(line l1, line l2, pt & P) {
    T d = cross(l1.v, l2.v);
    if (d == 0) return false;
    P = (l2.v*l1.c - l1.v*l2.c) / d; // requires floating-point coordinates
    return true;
}
```

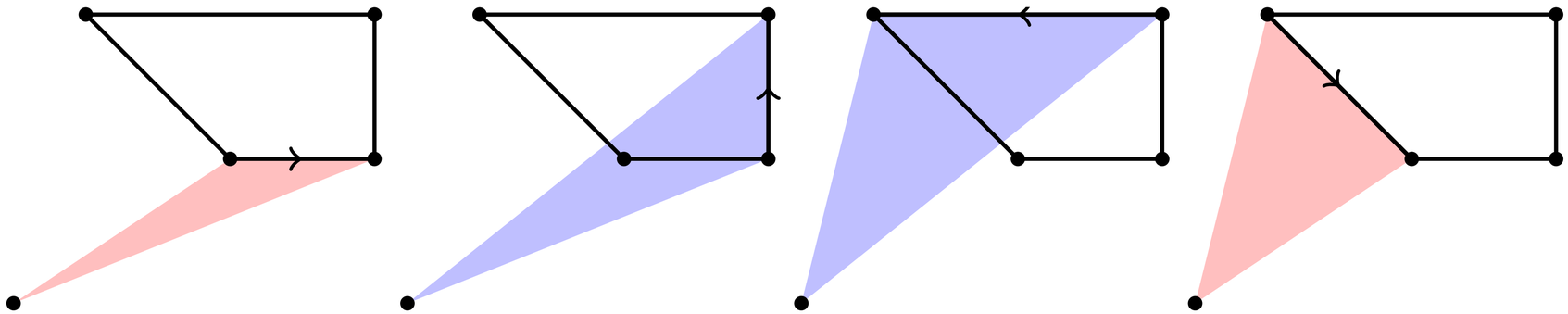
Área de un triángulo



$$\frac{1}{2}|AB||AC|\sin\theta = \frac{1}{2}\left|\overrightarrow{AB} \times \overrightarrow{AC}\right|$$

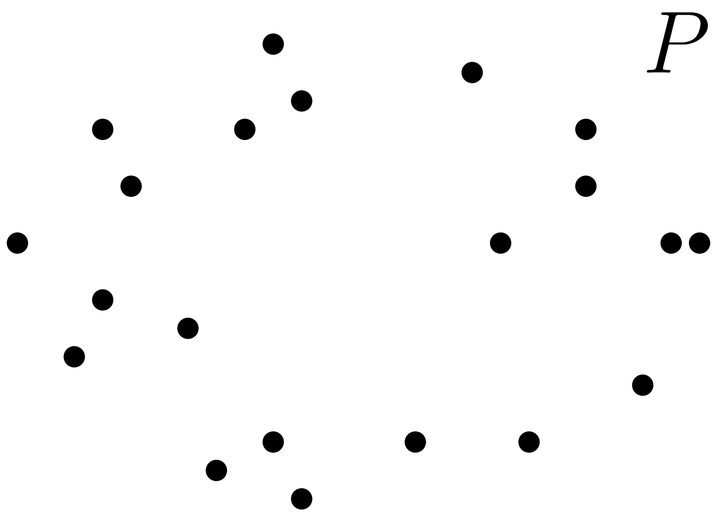
```
double areaTriangle(pt a, pt b, pt c) {
    return abs(cross(b - a, c - a)) / 2.0;
}
```

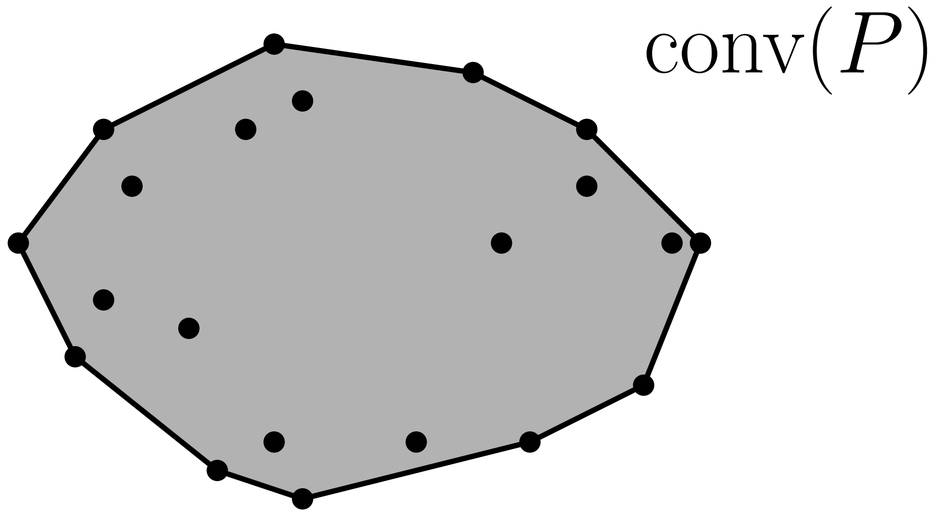
Área de un polígono



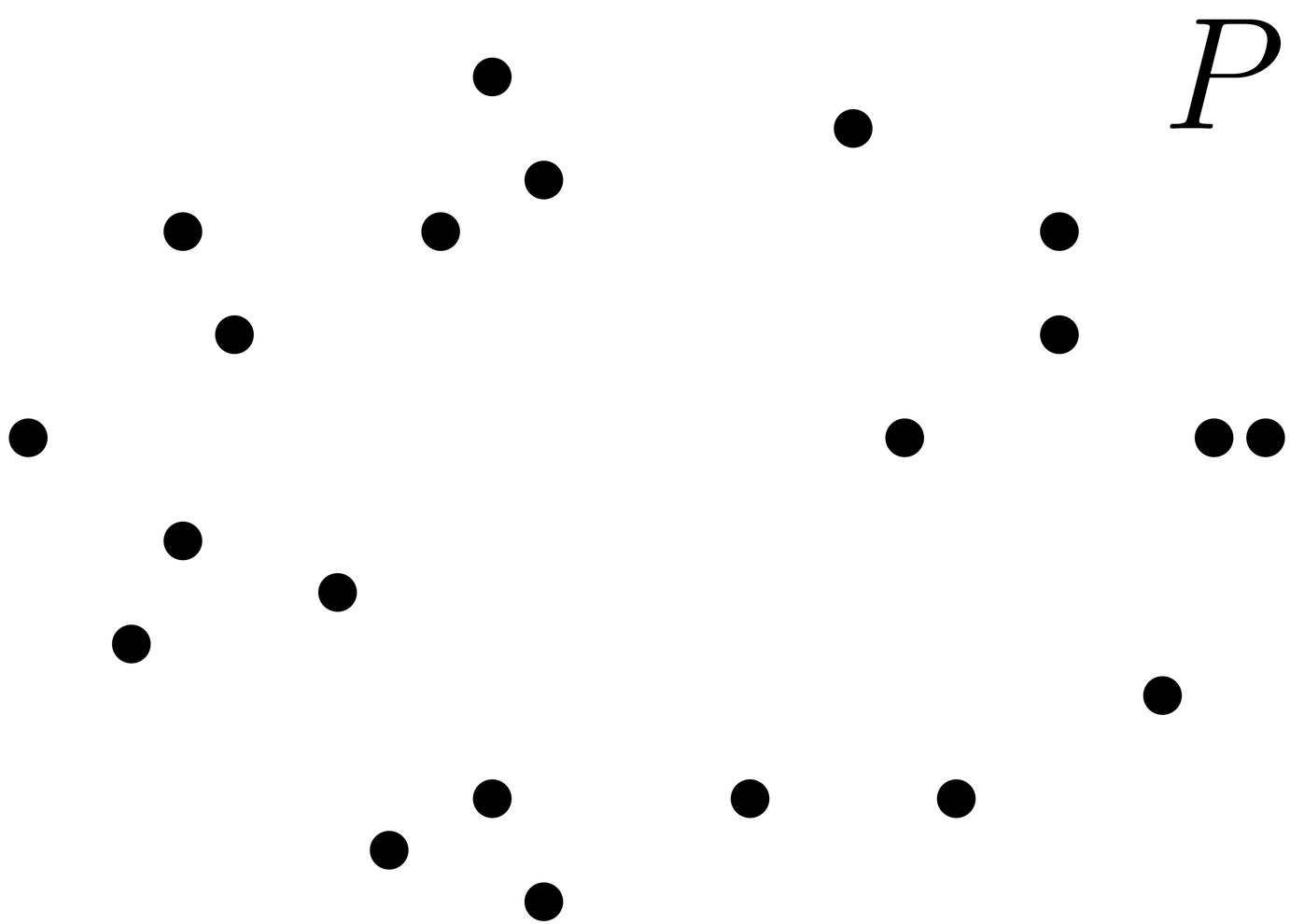
```
double areaPolygon(vector<pt> const& p) {
    double area = 0.0;
    for (int i = 0; i < int(p.size()) - 1; ++i) {
        area += cross(p[i], p[i+1]);
    }
    return abs(area) / 2.0;
}
```

Convex Hull

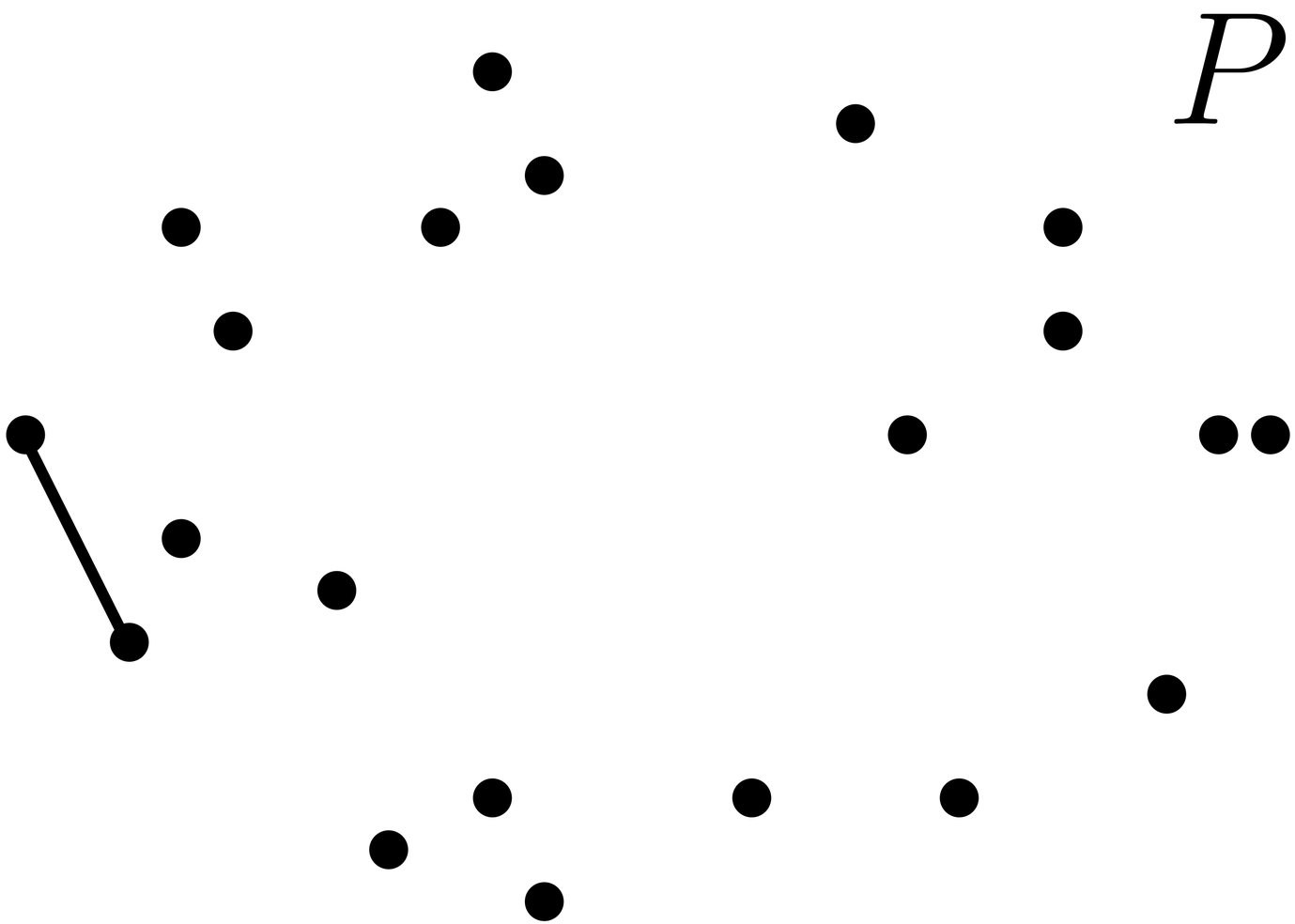




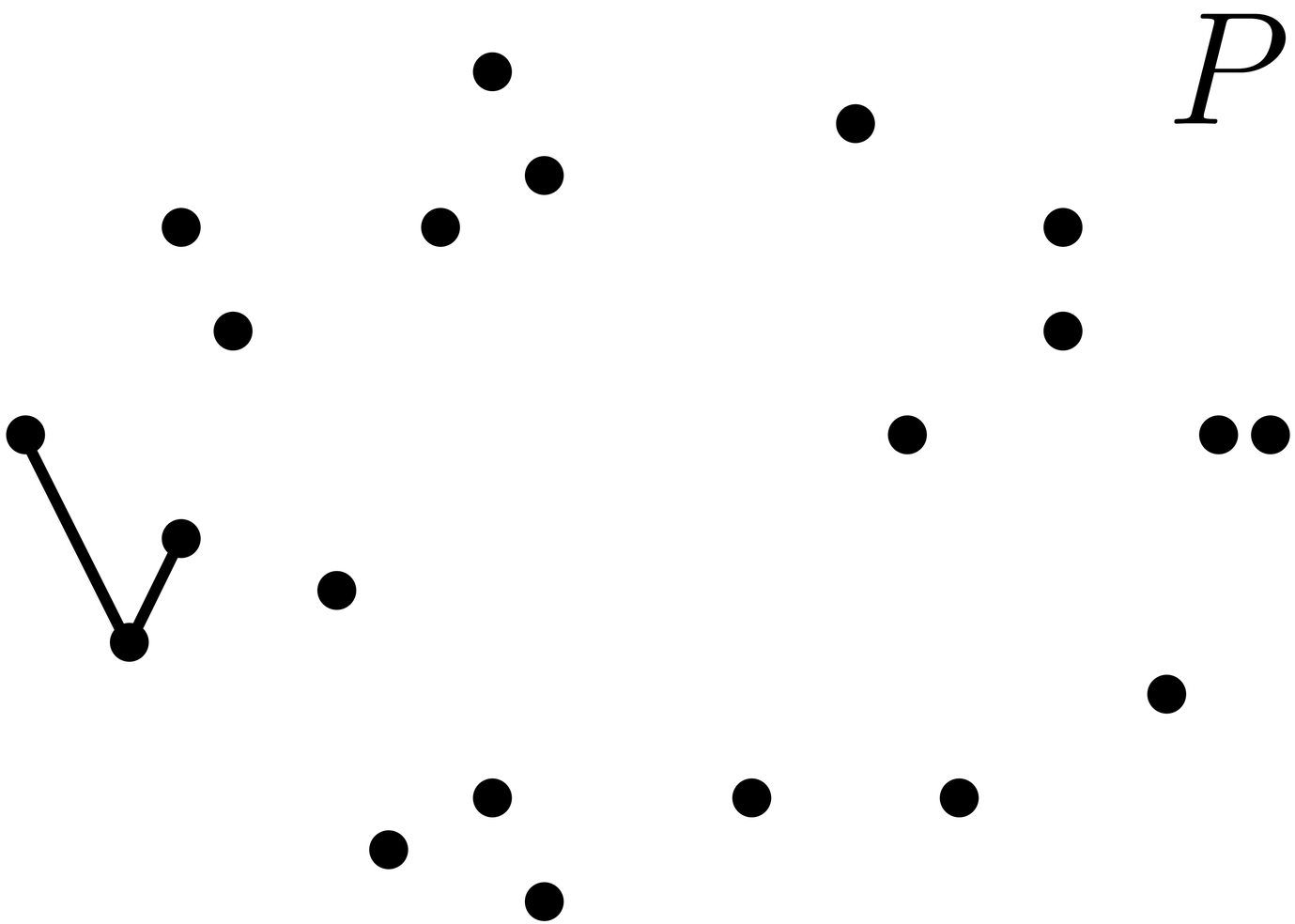
Convex Hull



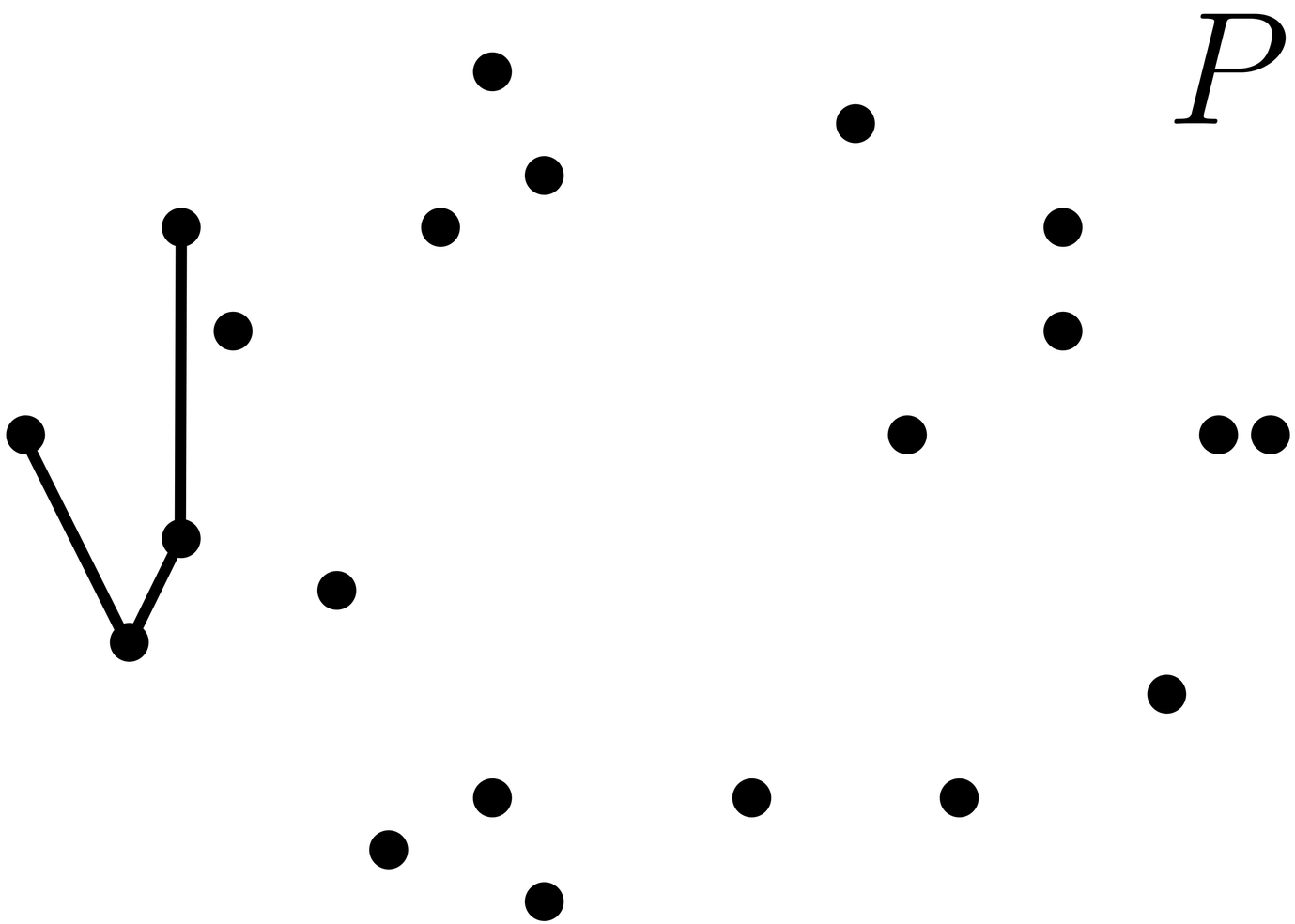
Convex Hull



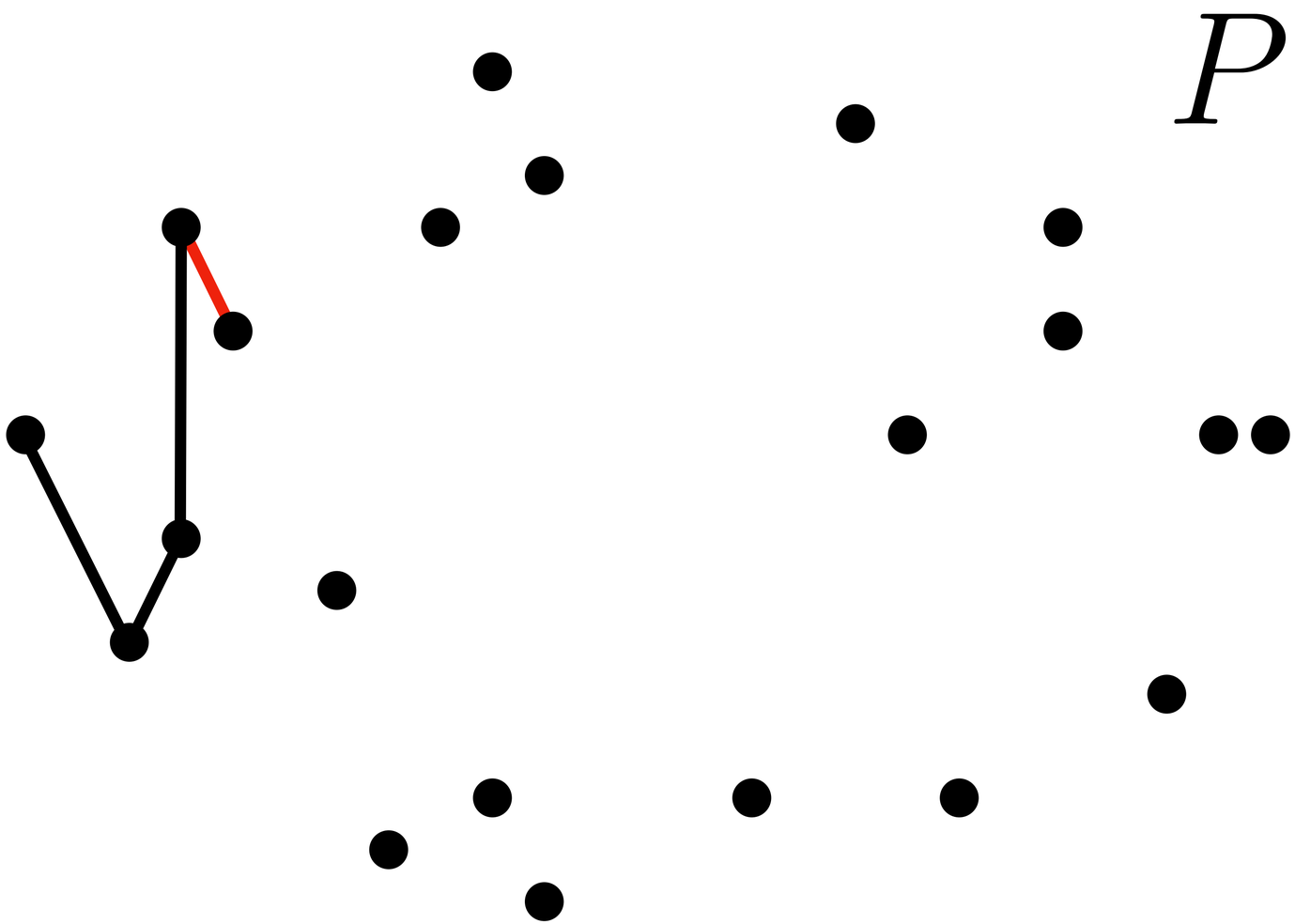
Convex Hull



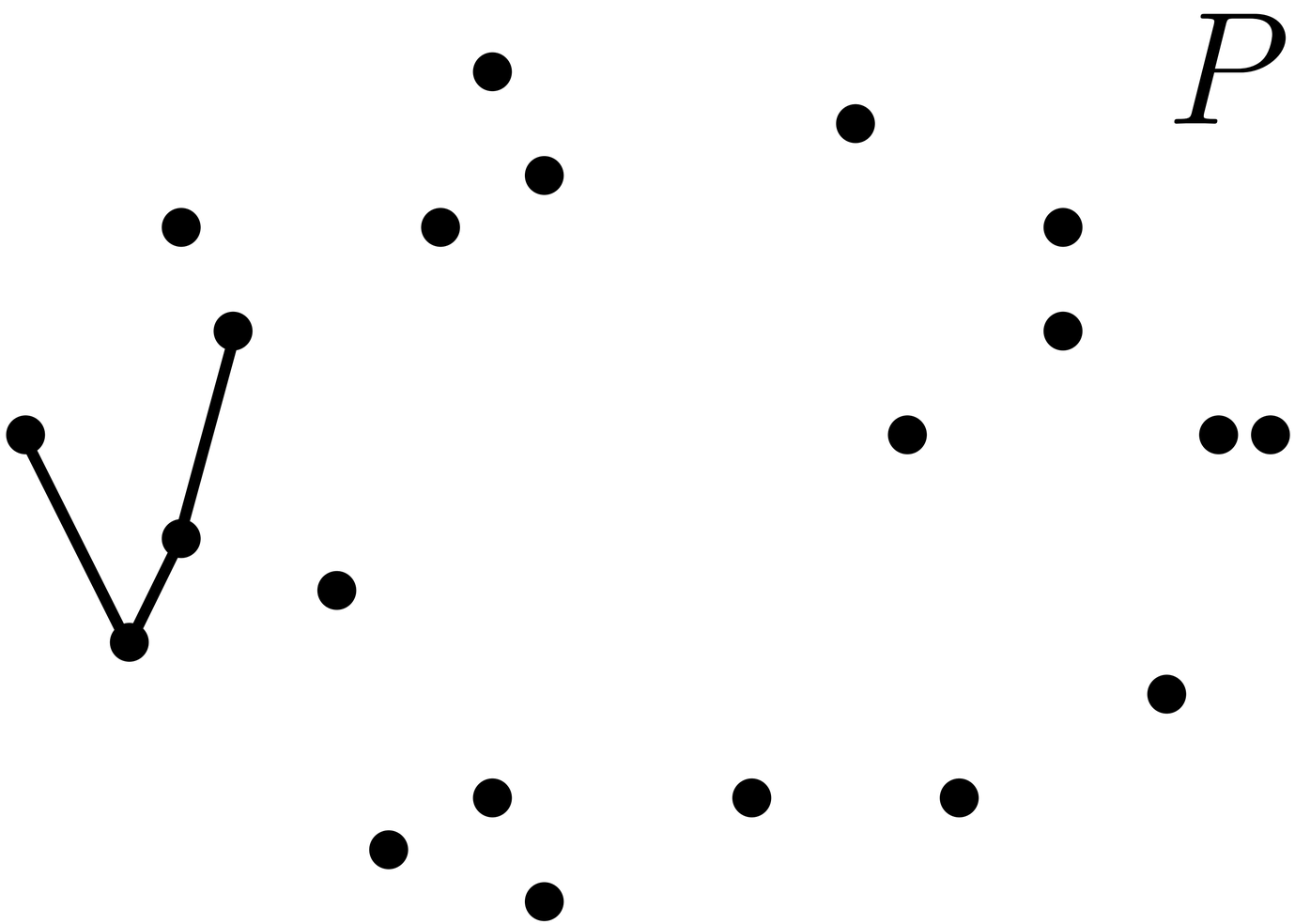
Convex Hull



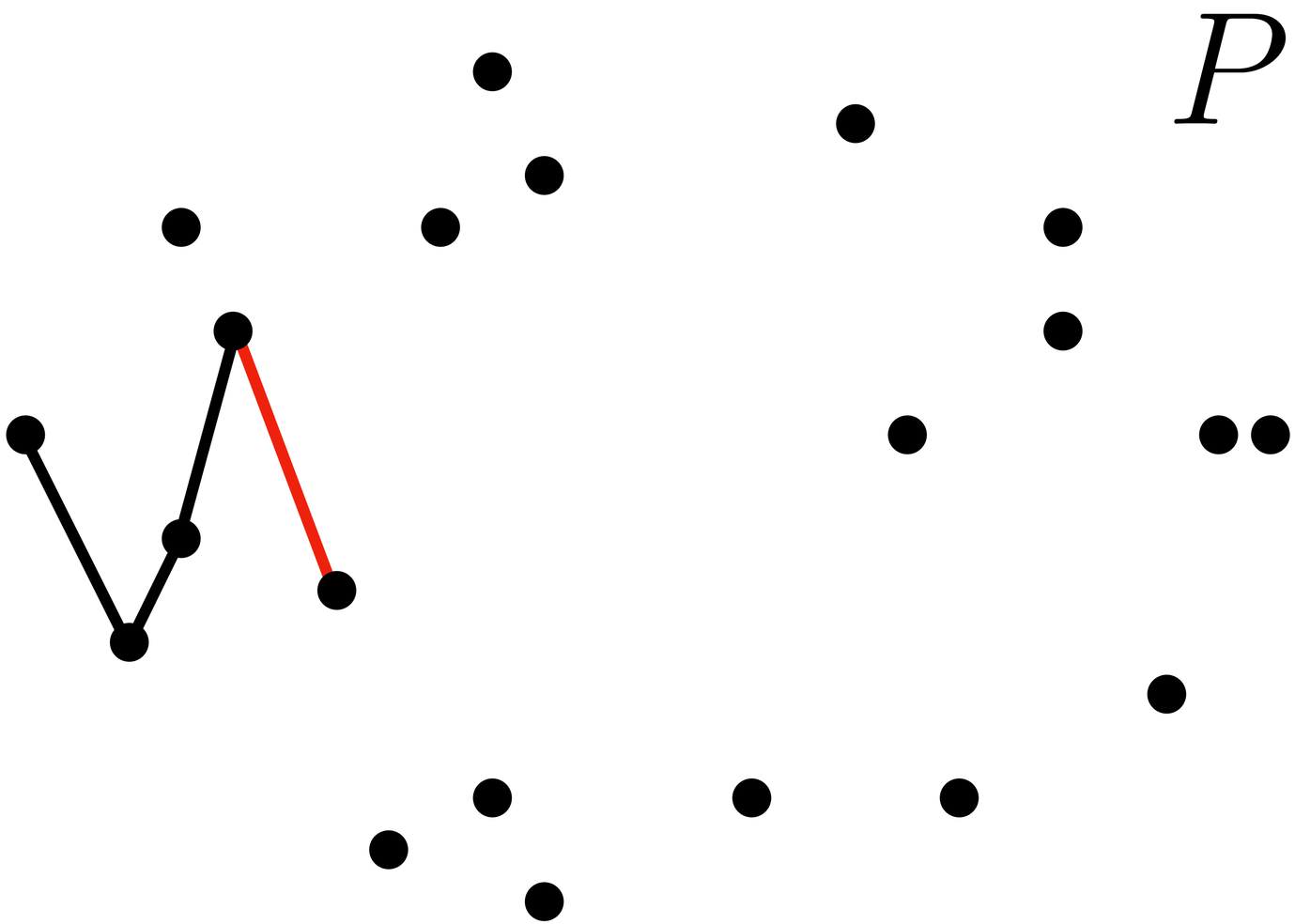
Convex Hull



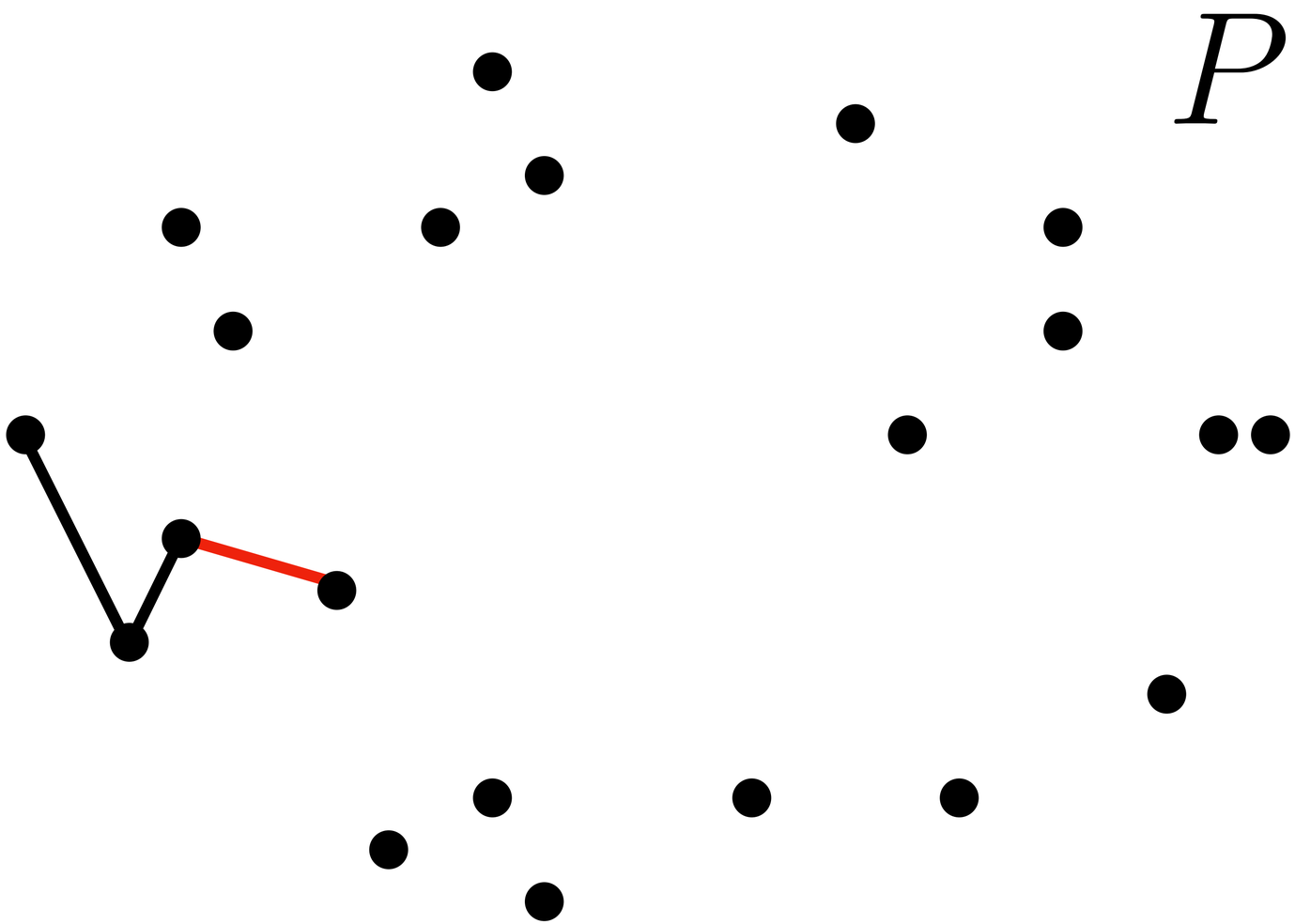
Convex Hull



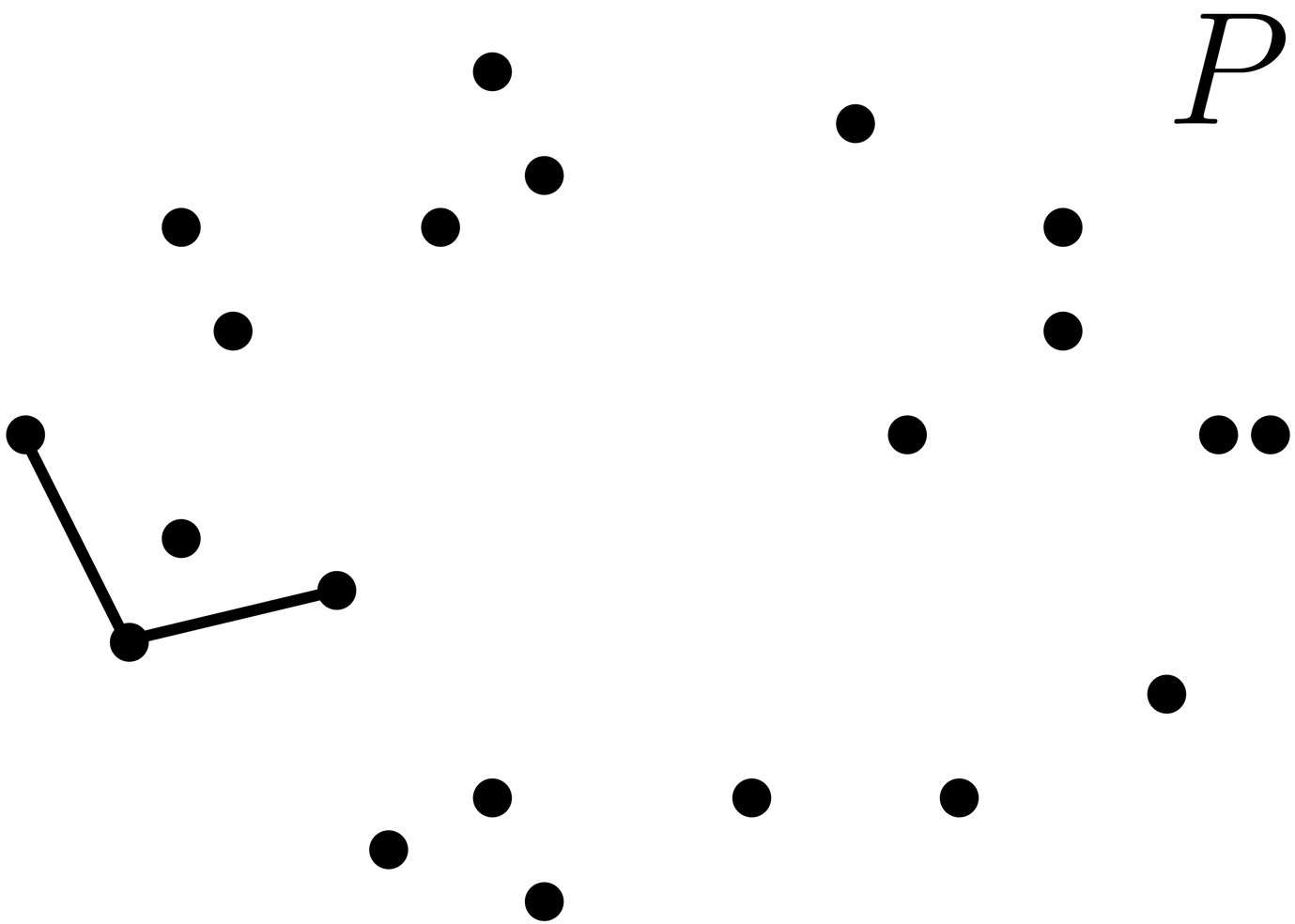
Convex Hull



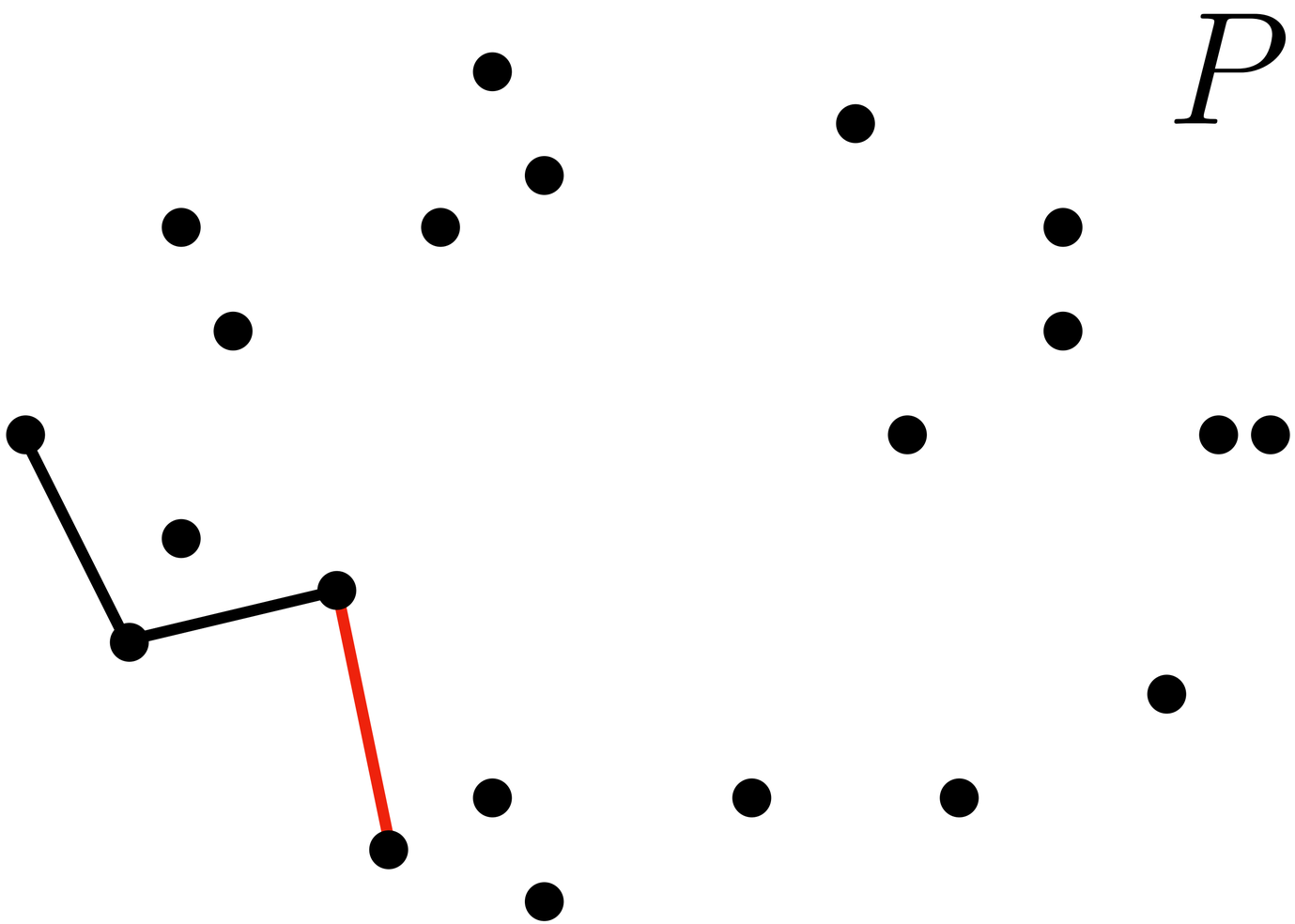
Convex Hull



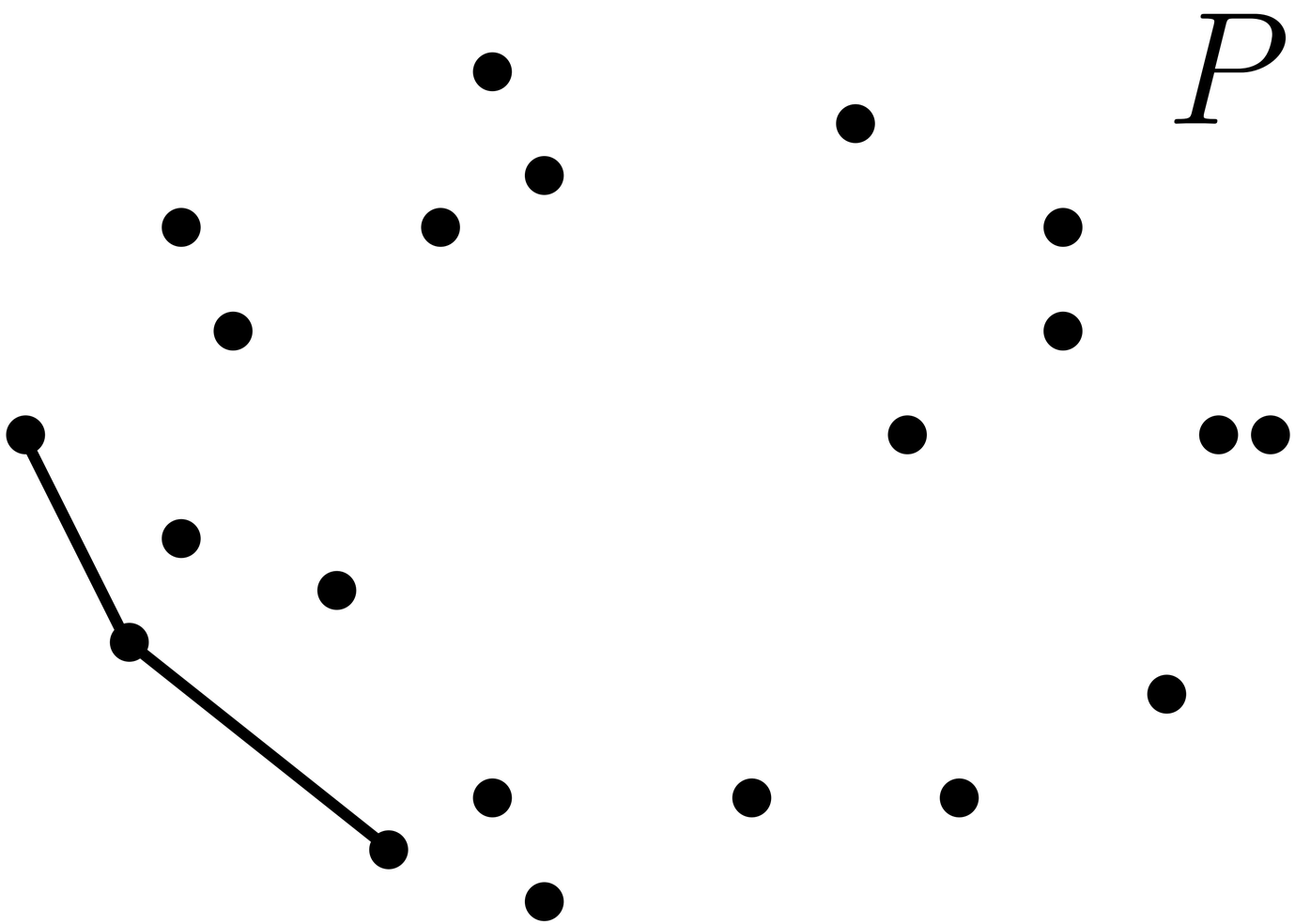
Convex Hull



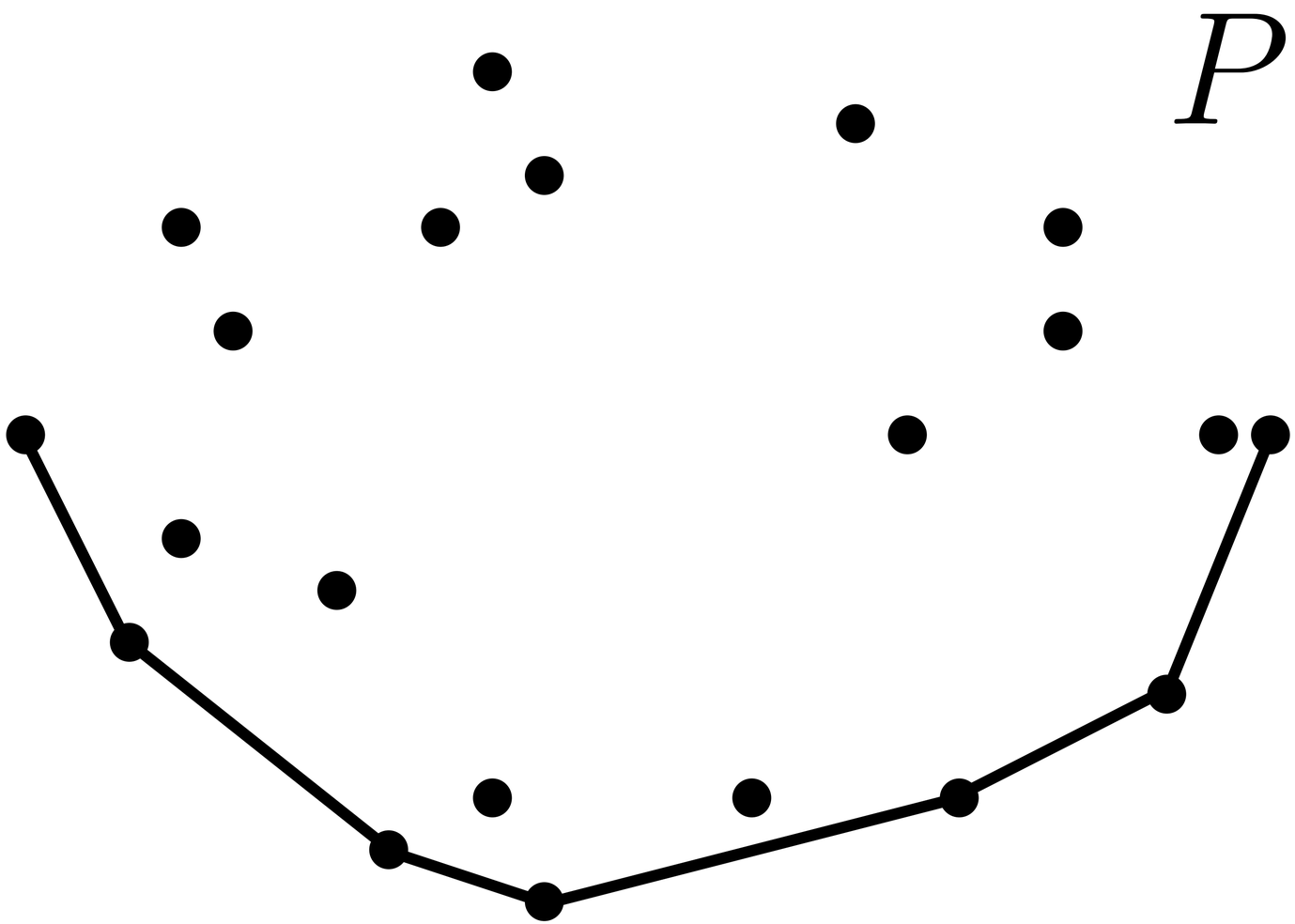
Convex Hull



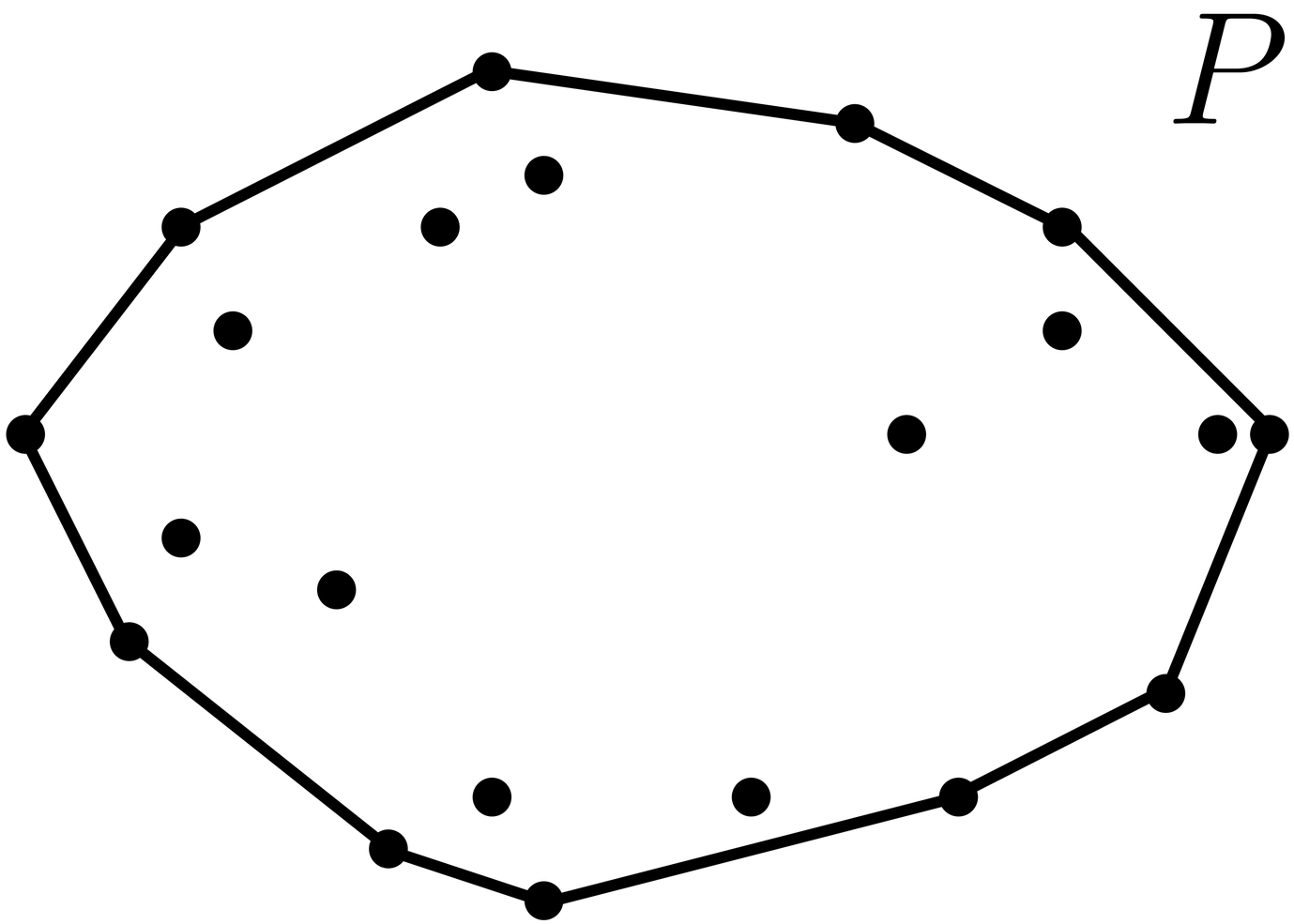
Convex Hull



Convex Hull



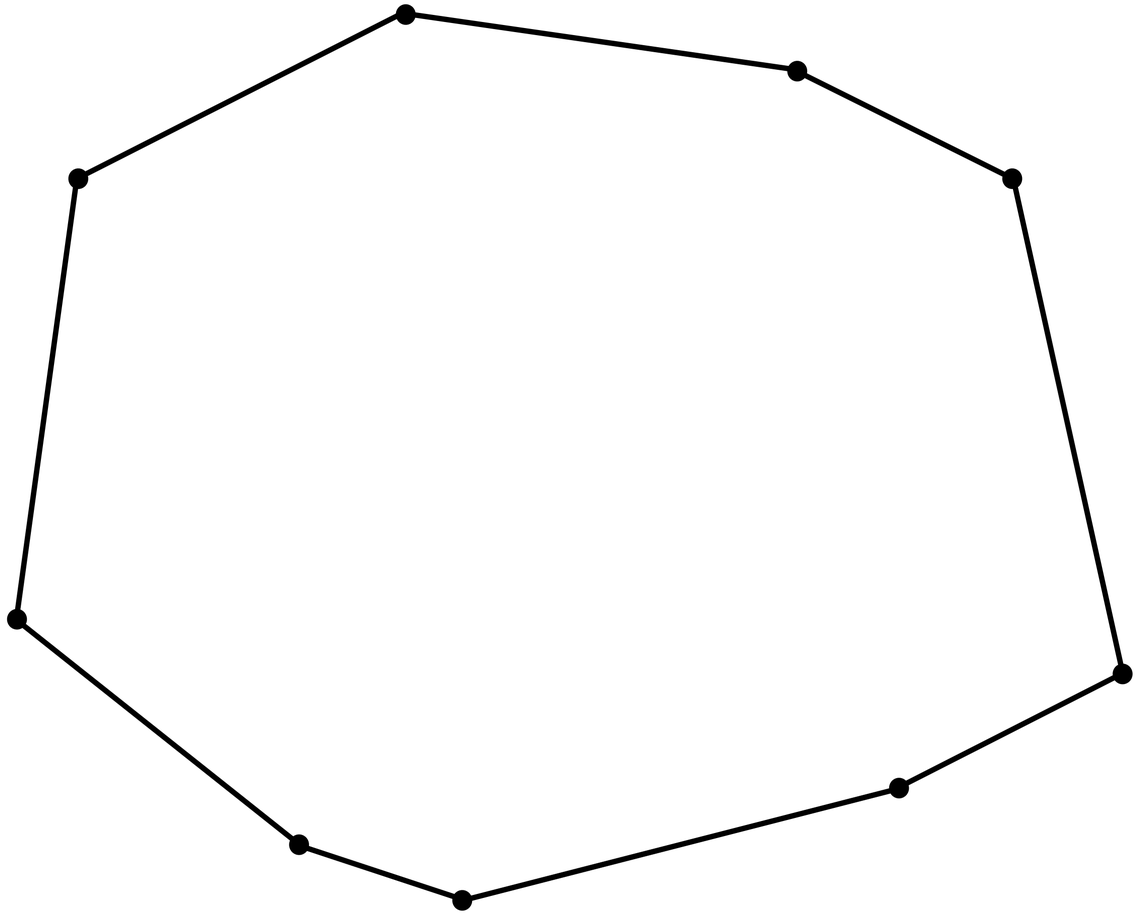
Convex Hull



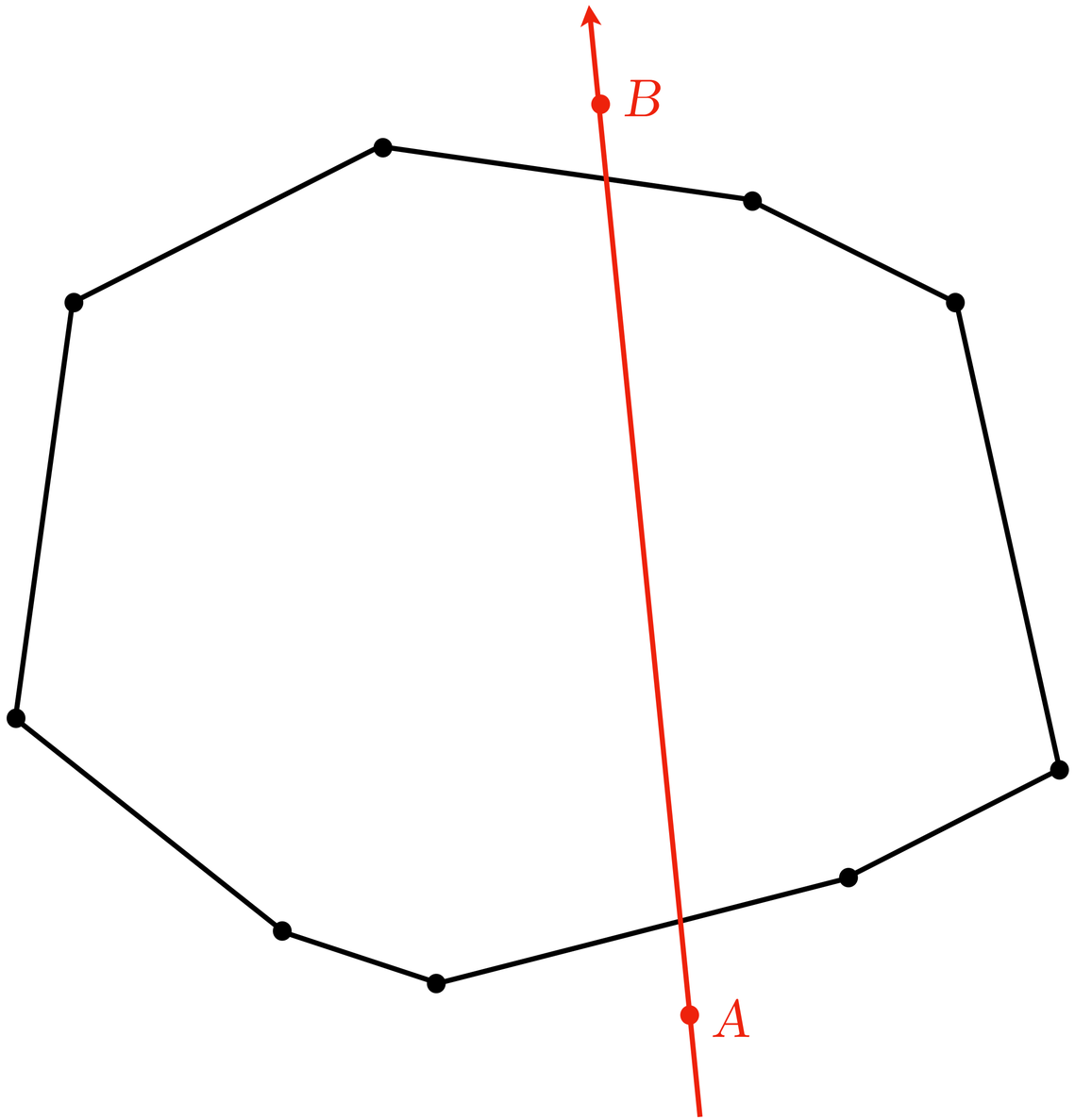
Andrew's monotone chain convex hull algorithm

```
// devuelve true si r está en el lado izquierdo de la línea pq
bool ccw(pt p, pt q, pt r) {
   return orient(p, q, r) > 0;  // >= para puntos colineales
}

// devuelve un polígono con la envolvente convexa de la nube de puntos P
vector<pt> convexHull(vector<pt> & P) {
   int n = int(P.size()), k = 0; vector<pt> H(2*n);
   sort(P.begin(), P.end());
   for (int i = 0; i < n; ++i) { // build lower hull
      while (k >= 2 && !ccw(H[k-2], H[k-1], P[i])) --k;
      H[k++] = P[i];
   }
   for (int i = n-2, t = k+1; i >= 0; --i) { // build upper hull
      while (k >= t && !ccw(H[k-2], H[k-1], P[i])) --k;
      H[k++] = P[i];
   }
   H.resize(k);
   return H;
}
```

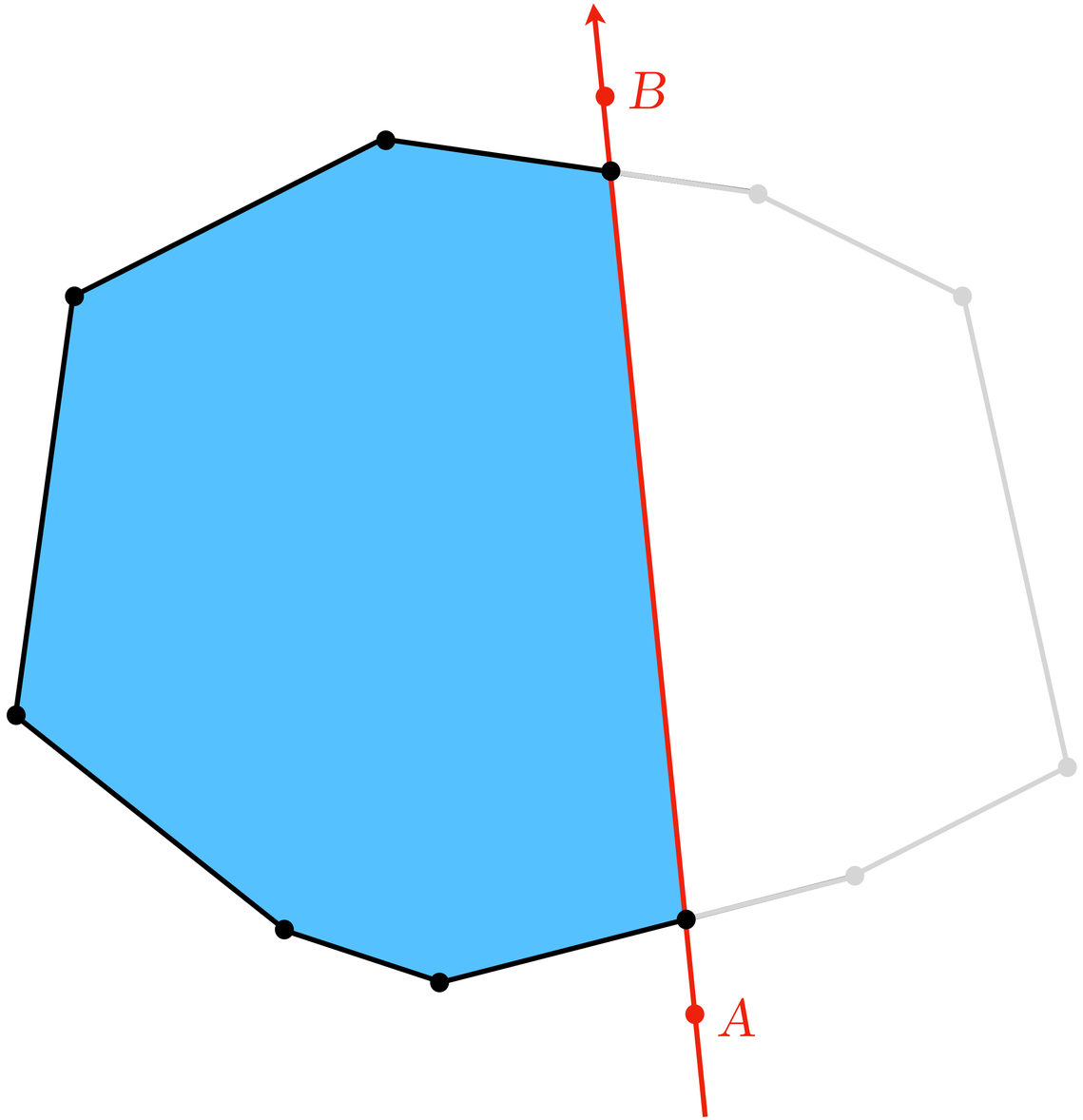
División de un polígono



División de un polígono



División de un polígono



División de un polígono

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