COMPUTATIONAL GEOMETRY

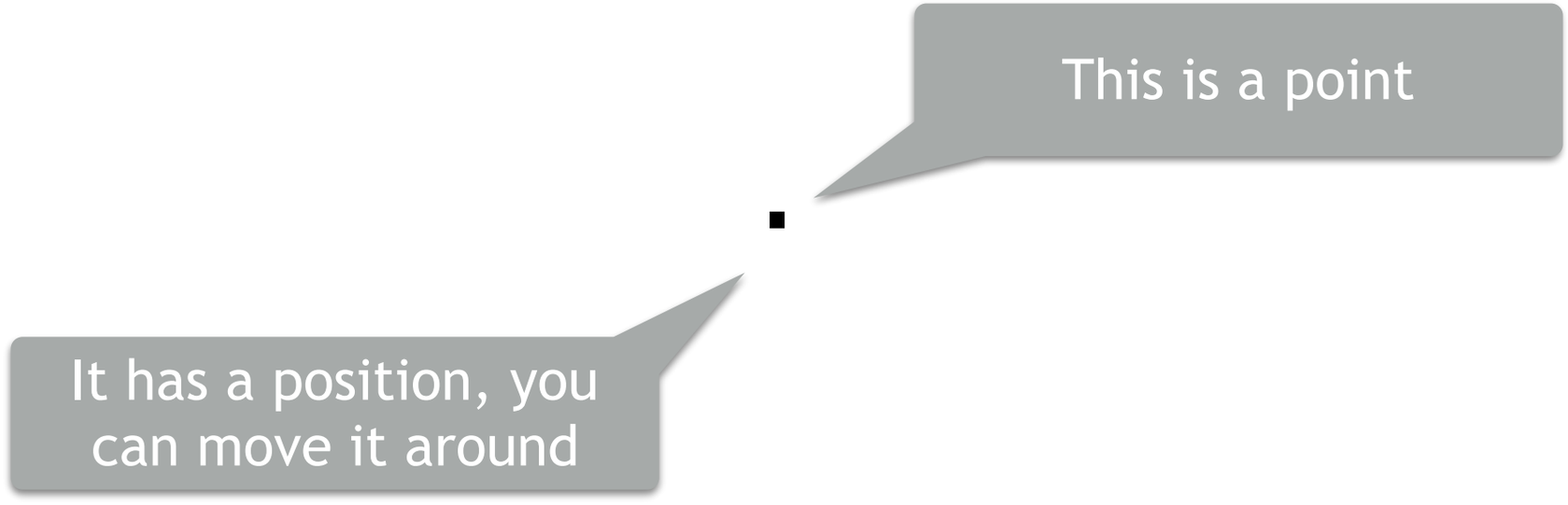
ISIS 2801

Constants

```
final double DEG_TO_RAD = π / 180.0

final double RAD_TO_DEG = 180.0 / π

final double EPS = 1e-9
```

POINTS

Points



Points

```
class Point:
  int x = 0
  int y = 0
  def sum(p):
    self.x = self.x + p.x
    self.y = self.y + p.y
  def prod(p):
    self.x = self.x*p.x - self.y*p.y
    self.y = self.x*p.x + self.y*p.y
  def conj:
    Point(self.x, -self.y)
```

use doubles for precision

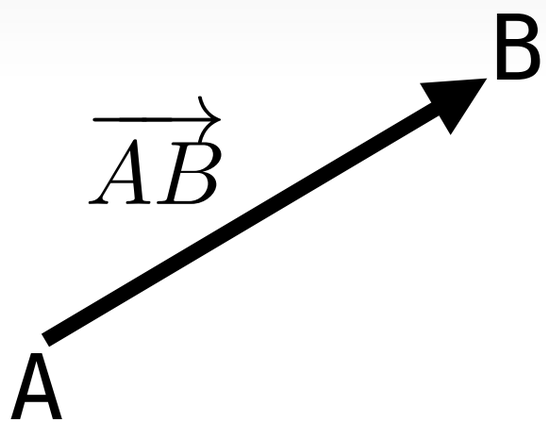
Points

use doubles for precision

```
Point p = new Point(1, 2);
Point2D p2d = new Point2D.Double(1.0, 2.0);
```

LINES AND VECTORS

Vectors



A **vector** consists of a starting (0,0) and finishing points with a magnitude, and a direction

$$|\overrightarrow{A}| = \sqrt{\sum A_i^2}$$

Given two points A,B a vector between them is defined as ⟨Bx − Ax, By − Ay⟩

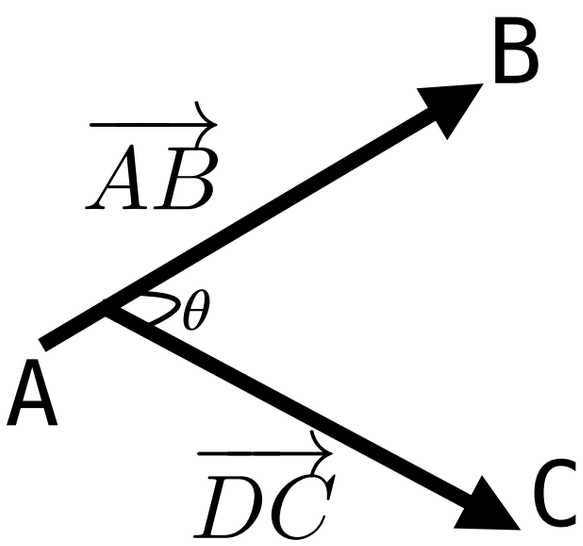
Vectors

use doubles for precision

```
struct vector2D {
    int x, y;
    double magnitude;
    vector2D(int _x1, int _x2, int _y1, int _y2) :
     x(_y1 - _x1), y(_y2 - _x2){
       magnitude = sqrt(x*x + y*y);
     }
};
```

Dot product

Orthogonality test. Two vectors are orthogonal if their **dot product** is 0

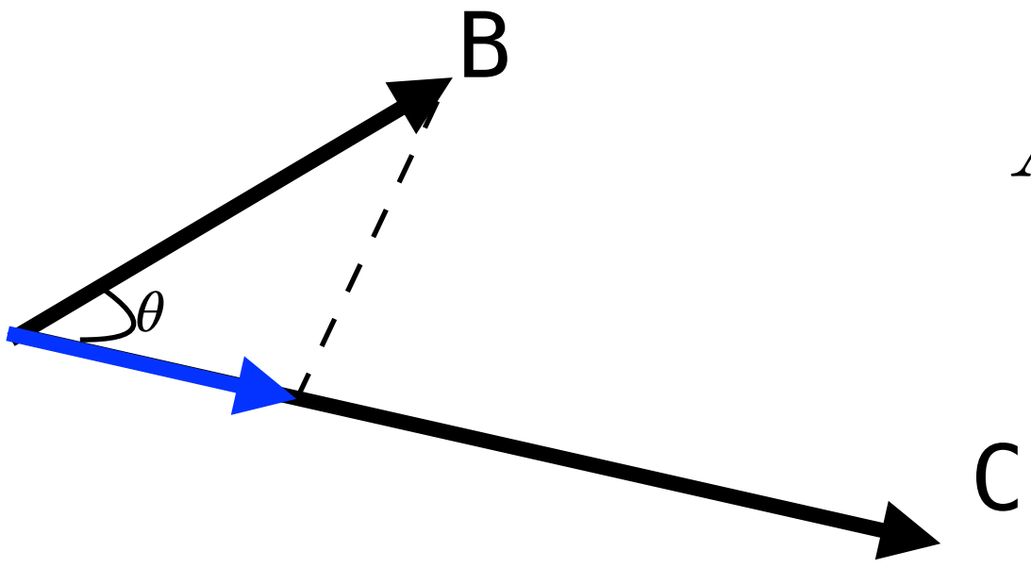


$$A \cdot B = |A| * |B| * \cos(\theta)$$

```
int dotProduct(vector2D v1, vector2D v2) {
  return v1.x*v2.x + v1.y*v2.y;
}

double dotProduct(vector2D v1, vector2D v2, double theta) {
  return v1.magnitude*v2.magnitude*cos(DEG_to_RAD(theta));
}
```

Dot product

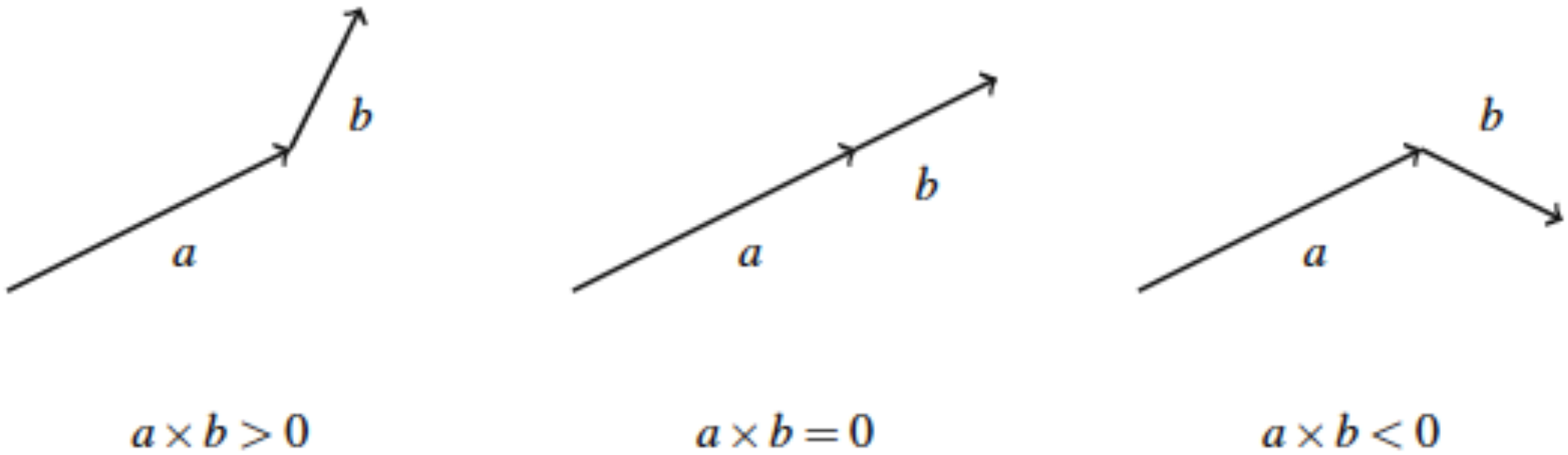
The **dot product** can also be used to find the projection of a vector onto another one



$$A_B = A \cdot \hat{B} = |A| * |\hat{B}| * \cos(\theta)$$
$$= |A| * \cos(\theta)$$

Cross product

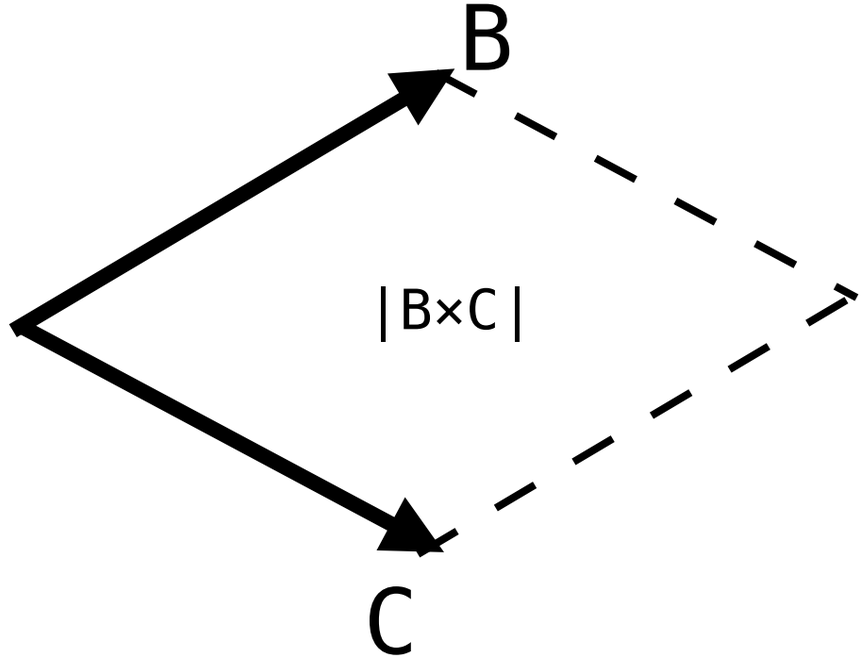
Colineatlity test. Two vectors are colineal if their **cross product** is 0. The cross product represents a vector orthogonal to both vectors in $\mathbb{R}^3$, $\langle 0, 0, A \times B \rangle$




```
double crossProduct(vector2D v1, vector2D v2) {
    return v1.x*v2.x - v1.y*v2.y;
}
```

Cross product

The **cross product** also represents the area of the parallelogram between the two vectors



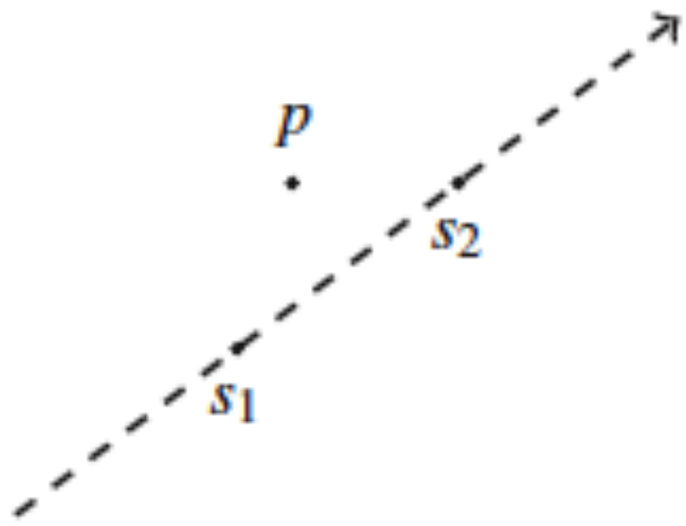
Cross product

The **cross product** can be used to test where a point is relative to a line.

vector between p and s2

$$cp = (p - s_1) \times (p - s_2)$$

```
if cp > 0 then left
if cp < 0 then right
if cp = 0 then colineal
```



Lines

$$Ax + By + C = 0$$

```
void line(point p1, point p2, line &l) {
  if(fabs(p1.x - p2.x) < EPS) {
      l.a = 1.0;
      l.b = 0.0;
      l.c = -p1.x;
  } else {
      l.a = -(p1.y - p2.y)/(p1.x - p2.x);
      l.b = 1.0;
      l.c = -(l.a*p1.x) - p1.y;
  }
}
```

vertical lines

Lines

l1 : Ax + By + C = 0 **l2 : A’x + B’y + C’ = 0**

```
dotProduct(l1, l2) = 0
```

perpendicular

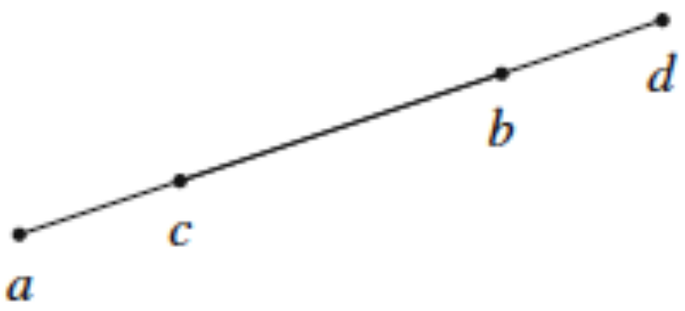
```
crossProduct(l1, l2) = 0
```

parallel

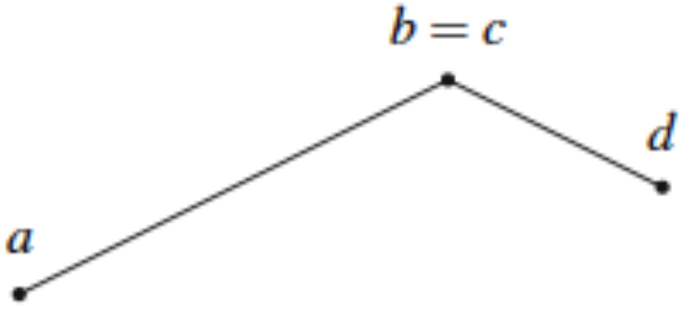
```
A == A’ && B == B’
```

Line intersection

$$\vec{ab} \cap \vec{cd}?$$



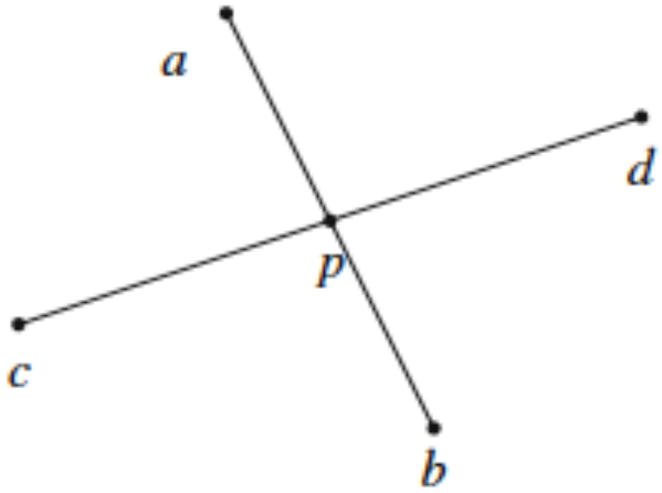

```
(c - a) x (c - b) = 0 ?
(d - a) x (d - b) = 0 ?
```



a = c ?
a = d ?
b = c ?
b = d ?

Line intersection

$$\vec{ab} \cap \vec{cd}?$$



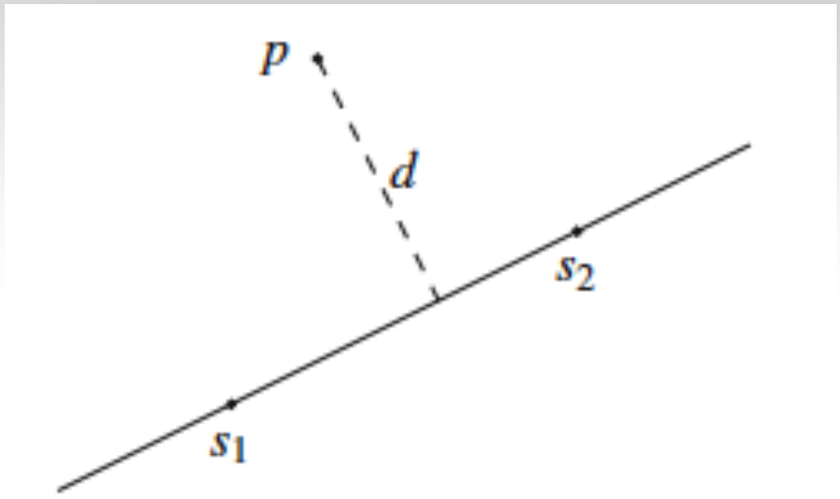

```
(c - a) x (c - b) > 0 ?
(d - a) x (d - b) < 0 ?

(a - c) x (a - d) > 0 ?
(b - c) x (b - d) < 0 ?
```

Line intersection

l1 : Ax + By + C = 0 **l2 : A'x + B'y + C' = 0**

```
bool intersection(struct line l1, struct line l2, point &p) {
    if(parallelP(l1, l2))
        return false;
    p.x = (l2.b*l1.c - l1.b*l2.c) / (l2.a*l1.a - l1.a*l2.b);
    if(fabs(l1.b) > EPS)
        p.y = -(l1.a*p.x + l1.c);
    else
        p.y = -(l2.a*p.x + l2.c);
    return true;
}
```

Point line distance



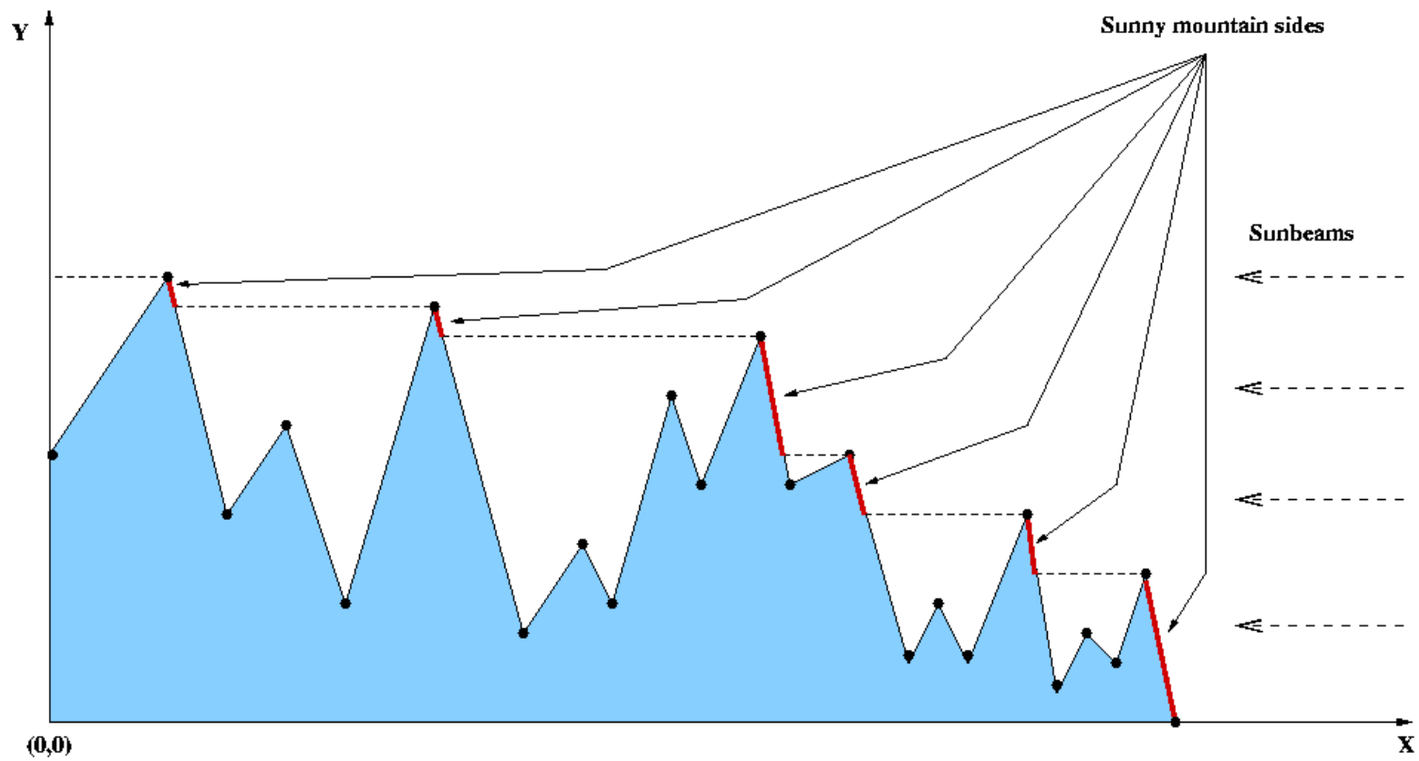
$$d = \frac{(s_1 - p) \times (s_2 - p)}{|s_2 - s_1|}$$

```
double distanceToLine(point p, point s1, point s2, point &c) {
    vector2D s1p = toVector2D(s1, p);
    vector2D s2p = toVector2D(s1, p);
    vector2D s1s2 = toVector2D(s1, s2);
    double u = crossProduct(s1p, s2p) / s1s2.getMagnitude();
    c = translate(s1, scale(s1s2, u));
    return u;
}
```

UVa 920

During their honeymoon, Mrs and Mr Smith went to the Himalayas. How they were surprised when they observed that, during the sunset, all the snow touched by the sunbeams turned red.
Such a magnificent landscape leaves everyone plenty of emotion, but Mr Smith's number obsession overcame all this. He rapidly began evaluating distances, which made Mrs Smith quite upset.
Your work is to help him calculate the size, in meters, of the mountainsides that became red as the sun sets. Mr Smith's honeymoon depends on you! Please be quick and efficient.

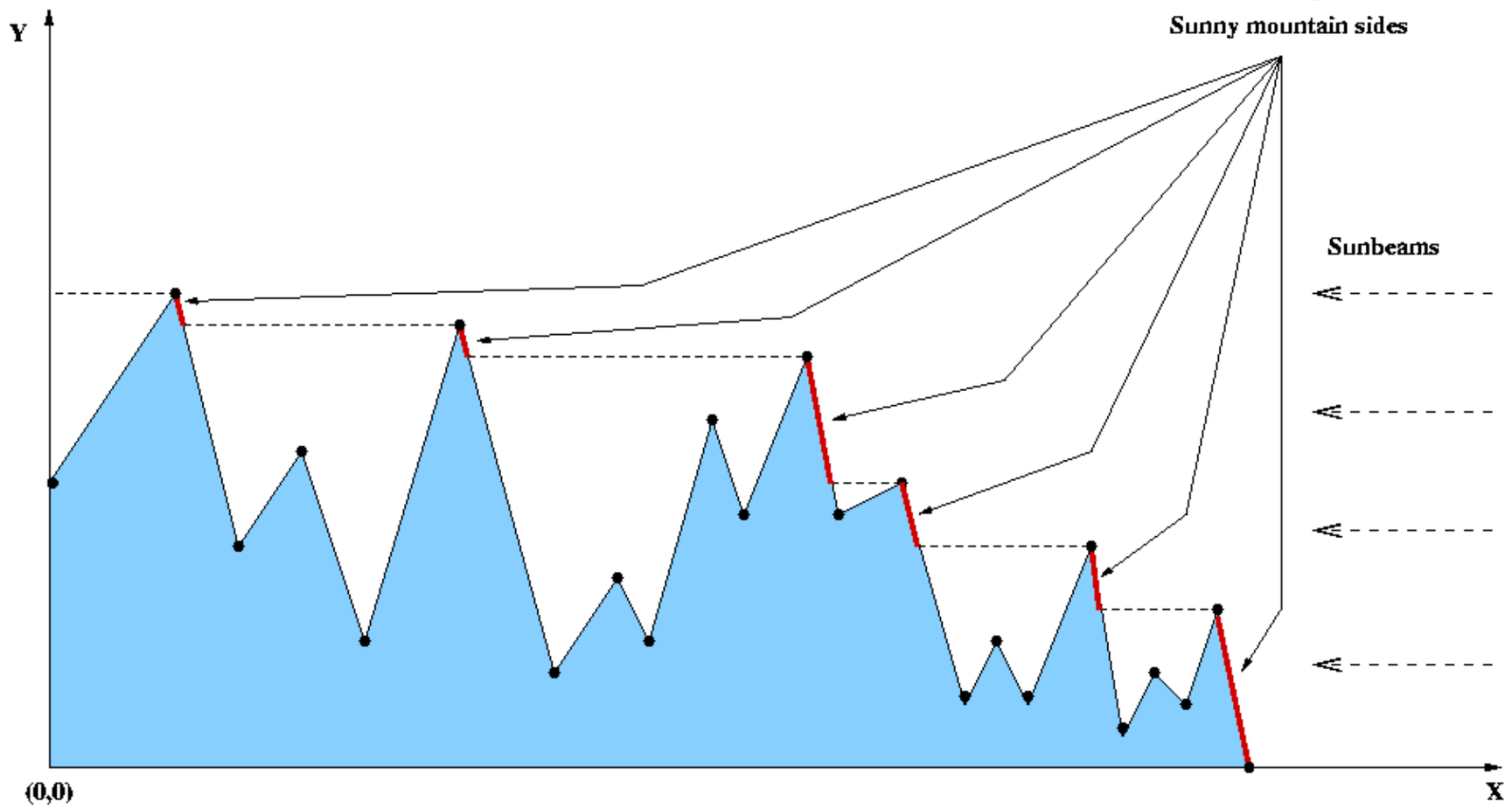
For the sake of simplicity, consider that, during the sunset, the sunbeams are horizontal and assume that the landscape is described by the set of coordinates of the mountain peaks and cols. This can be depicted by the following figure. A landscape, in this context, is then a sequence of peaks and cols (i.e., only a col follows a peak and conversely).



UVa 920

Note that, in this picture, the sunny mountainsides are emphasized by bold lines and the coordinates of the landscape are emphasized by bold points. Thus, the goal of this problem is to calculate the total length in meters of the bold lines.

For this task consider that: (1) for all coordinates (x,y), $0 \leq x \leq 30000$ and $0 \leq y \leq 8848$; (2) the unit is the meter; (3) all the X-coordinates are pair-wise distinct; (4) the leftmost point has 0 as X-coordinate and the rightmost point has 0 as Y-coordinate; (5) The total number of coordinates given is $n \leq 100$.



UVa 920

Input
The first line of input contains C (0 < C < 100), the number of test cases that follows.
Each test case starts with a line containing the number N of coordinate pairs. The remaining N lines for each test case contain the coordinates defining the landscape. Each of these lines contains two integers, x and y, separated by a single space. The first integer, x, is the X-coordinate, and the second, y, is the Y-coordinate of the considered point.

Output
The output is formed by a sequence of lines, one for each test case. Each line contains a single real number with exactly two decimal digits. This number represents the length in meters of the sunny mountainsides for the corresponding test case.

UVa 920

Sample Input

2
11
1100 1200
0 500
1400 100
600 600
2800 0
400 1100
1700 600
1500 800
2100 300
1800 700
2400 500
2
0 1000
1000 0

Sample Output

1446.34
1414.21

UVa 920

Strategy

- Get all the tops and bottoms of the mountains
- **Order them** (with respect to the x coordinate in descending order)
- **Add** the difference between the in distance between the low point and the top point of the mountain, from the point of the previous mountain

UVa 920

```
vector<point> points;
for (int i = 1; i <= n; ++i) {
  point p;
  cin >> p.x >> p.y;
  points.push_back(p);
}
```

order the points

```
sort(points.begin(), points.end(), comp);

int currentMaxY = 0;
double length = 0;
for (int i = 1; i < points.size(); ++i) {
  if (points[i].y > currentMaxY) {
    length += dist(points[i], points[i - 1])
                * (points[i].y - currentMaxY) /
                  (points[i].y - points[i - 1].y);
    currentMaxY = points[i].y;
  }
}
```

add the length

SHAPES

Circles

A **circle** centered at `(x,y)` with radius `r` is a **set** of points `(a,b)` such that $(a - x)^2 + (b - y)^2 = r^2$

```
int insideCircle(point c, int radius, point p) {
    int dx = p.x - c.x;
    int dy = p.y - c.y;
    int dist = dx*dx + dy*dy;
    int rad = radius*radius;
    if(dist == rad) {
        return 0;
    } else if(dist < rad)
        return 1;
    else
        return 2;
}
```

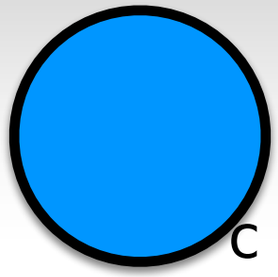
on

in

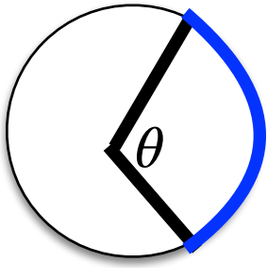
out

Circles

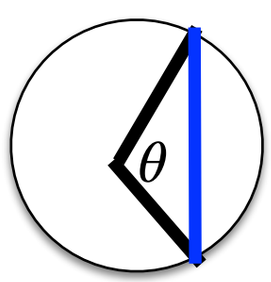
$A = \pi r^2$



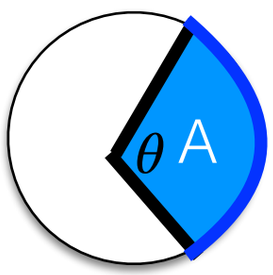
$c = 2\pi r$



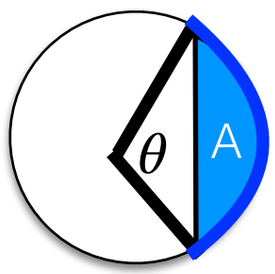
$l_a = c\theta/360$



$chord = 2rsin(\theta/2)$

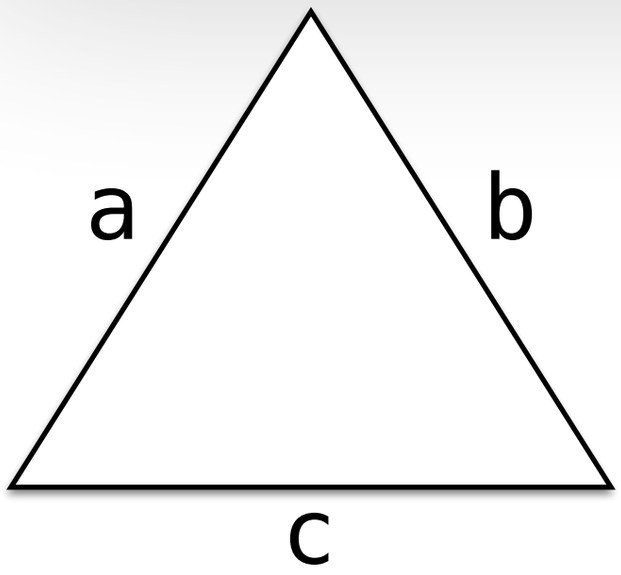


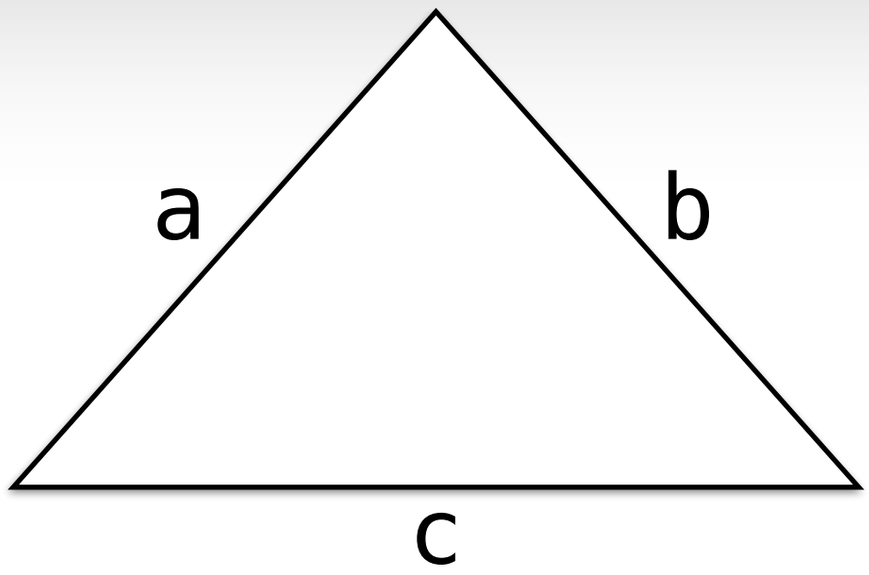
$A_s = A\theta/360$

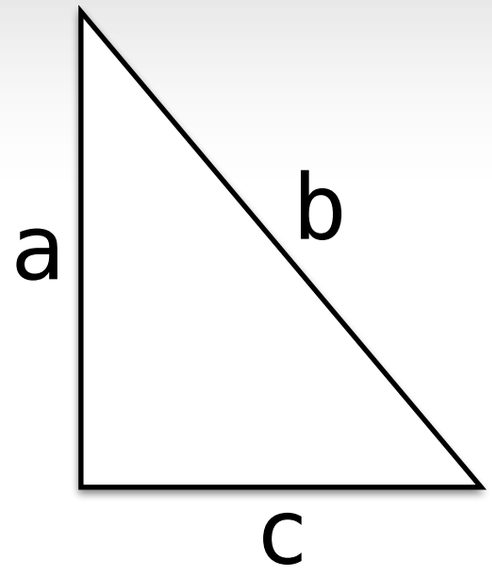


$A_{seg} = A_s - A_{\Delta r}$

Triangles

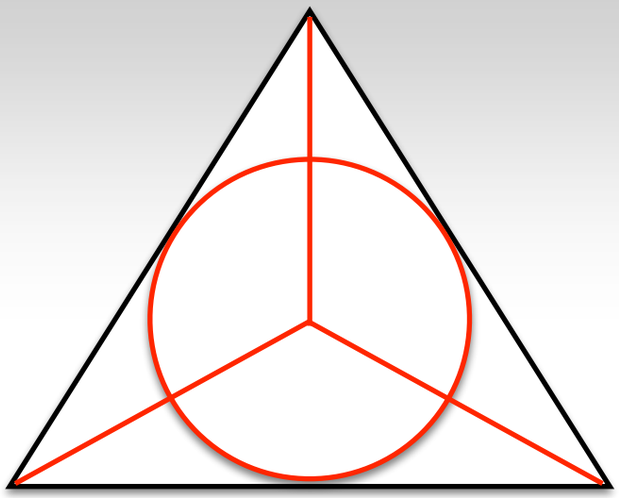






Perimeter: P = a + b + c
Semi-perimeter: s = 1/2(P)
Area: A = πr^2
Better area: A = √s(s−a)(s−b)(s−c)
Centroid: (a + b + c) /3

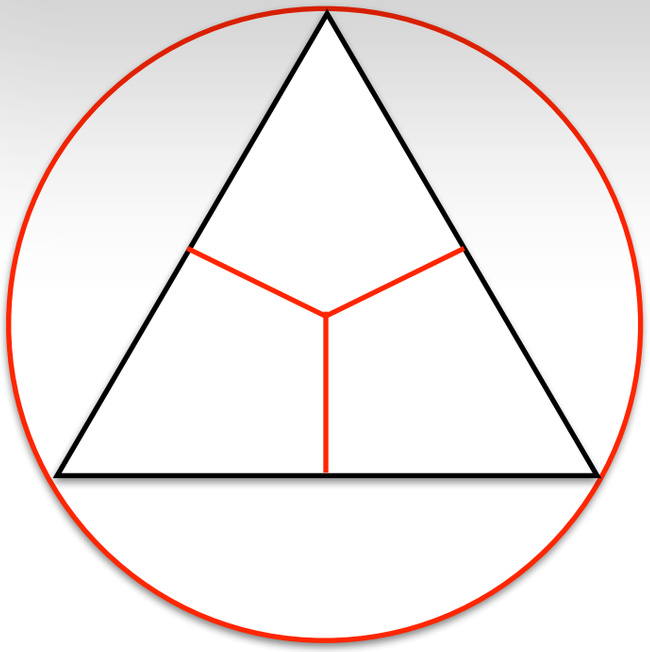
Triangles



Radius: $r = A_\Delta / s$

```
int inscribedCirc(point p1, point p2, point p3, point &c, double &r) {
  r = 2*area(p1,p2,p3)/perimeter(p1,p2,p3);
  if(fabs(r) < EPS)
      return 0;
  line l1, l2;
  double ratio = dis(p1, p2)/dist(p1, p3);
  point p = translate(p2, scale(toVec(p2,p3), ratio / (1+ratio)));
  line(p1, p, l1);

  ratio = dist(p2,p1) / dist(p2, p3);
  p = translate(p1, sclae(toVec(p1,p3), ratio / (1+ratio)));
  line(p2, p, l2);
  intersection(l1, l2, c);
  return 1;
}
```

Triangles



Radius: r = abc/4A

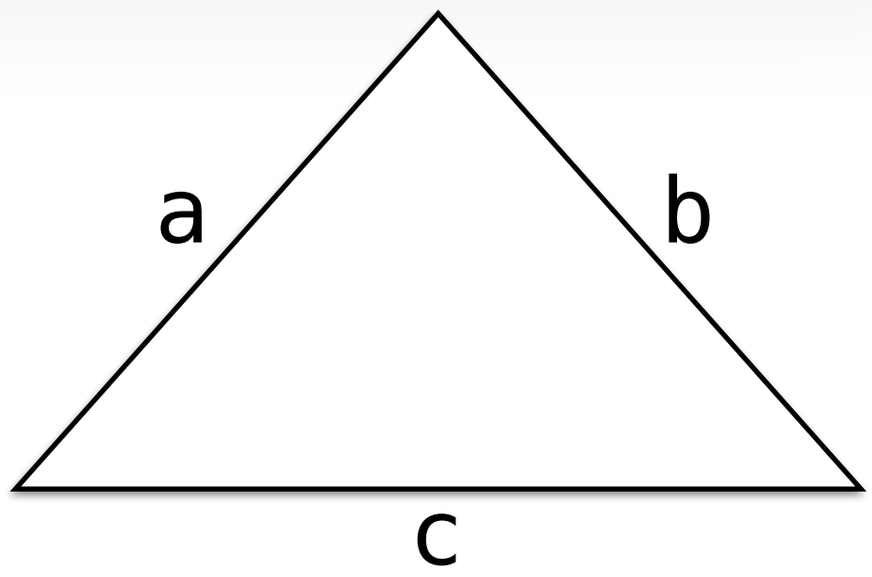
Diameter: 2r = a/sin(α)

Triangles

To check if three line segments form a triangle, you need to check that:

- a+b > c
- a+c > b
- b+c > a

only check this if the lengths are sorted



https://codeforces.com/problemset/problem/6/A

Johnny has a younger sister Anne, who is very clever and smart. As she came home from the kindergarten, she told his brother about the task that her kindergartener asked her to solve. The task was just to construct a triangle out of four sticks of different colours. Naturally, one of the sticks is extra. It is not allowed to break the sticks or use their partial length. Anne has perfectly solved this task, now she is asking Johnny to do the same.

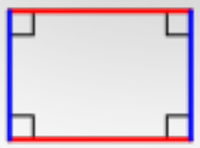
The boy answered that he would cope with it without any difficulty. However, after a while he found out that different tricky things can occur. It can happen that it is impossible to construct a triangle of a positive area, but it is possible to construct a degenerate triangle. It can be so, that it is impossible to construct a degenerate triangle even. As Johnny is very lazy, he does not want to consider such a big amount of cases, he asks you to help him.

https://codeforces.com/problemset/problem/6/A

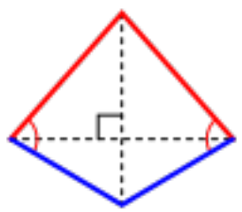
Input
The first line of the input contains four space-separated positive integer numbers not exceeding 100 — lengthes of the sticks.

Output
Output TRIANGLE if it is possible to construct a non-degenerate triangle. Output SEGMENT if the first case cannot take place and it is possible to construct a degenerate triangle. Output IMPOSSIBLE if it is impossible to construct any triangle. Remember that you are to use three sticks. It is not allowed to break the sticks or use their partial length.

Polygons

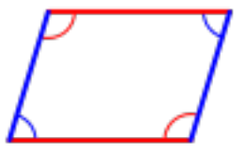


Rectangle: four right angles

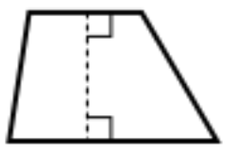


Kite: Two pair of edges of the same length, adjacent to each other

A = $d_1d_2/2$



Parallelogram: Oposite sides are parallel



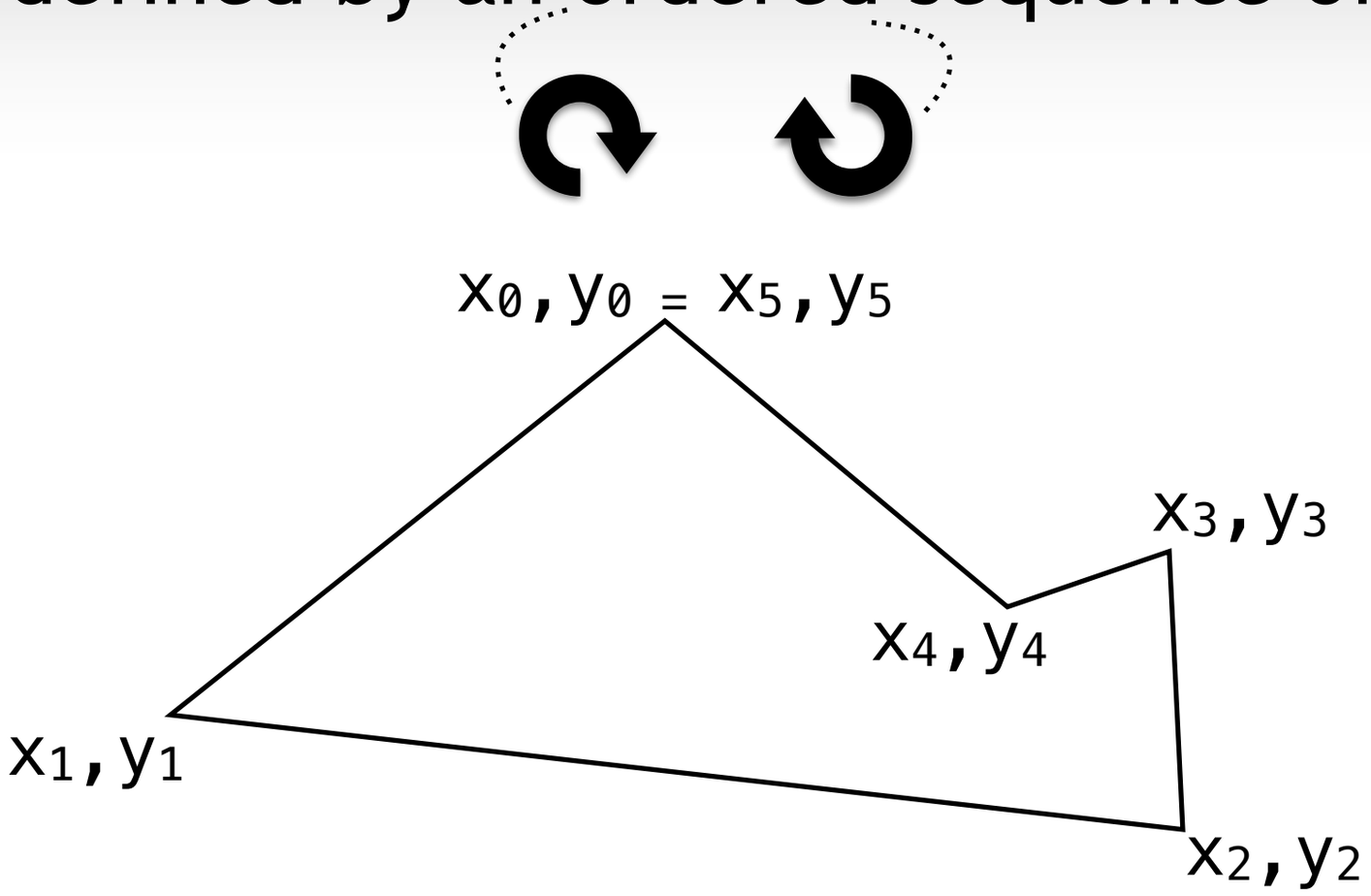
Trapezium: A pair of parallel edges

A = (e1 + e2)h/2

Quadrilateral: four edges, and four vertices

Polygons

A **polygon** is defined by an ordered sequence of points

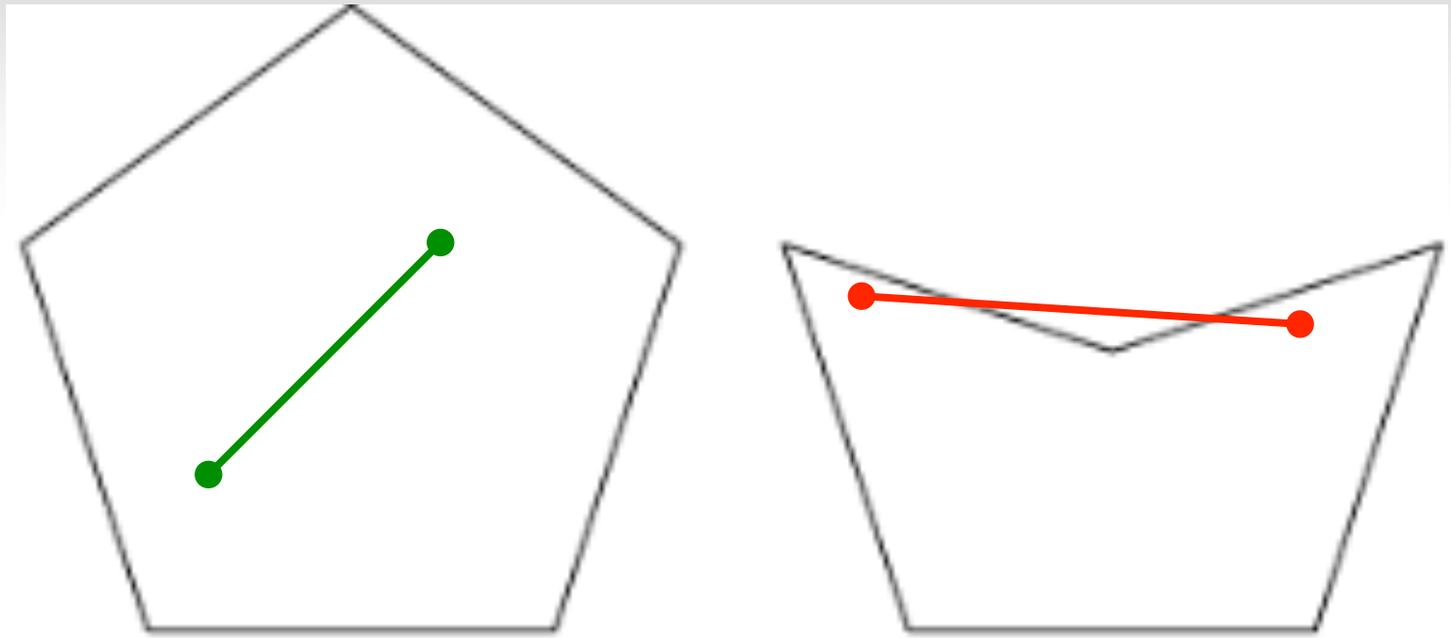


$P =$

x_0, y_0	x_1, y_1	x_2, y_2	x_3, y_3	x_4, y_4

circular arrays
(aka python)

Polygons



convex polygon

concave polygon

lines between **any two** points **are completely contained** in the polygon

lines between two point **may “exit”** the polygon

Polygons

Convexity?

```
bool convexP(const vector<point> &P) {
    int size = (int)P.size();
    if(size <= 3)
        return false;
    bool isLeft = ccw(P[0], P[1], P[2]);
    for(int i=1; i< size -1; i++) {
        if(ccw(P[i], P[i+1], P[(i+2) == size ? 1 : i+2]) != isLeft)
            return false;
    }
    return true;
}
```

remember cross-product

Polygons

Convexity?

```
bool convexP(const vector<point> &P) {
    int size = (int)P.size();
    if(size <= 3)
        return false;
    for(int i=1; i< size -1; i++) {
        if(arcos(|P[i]P[i+1]|*|P[i+1]P[i+2]|/
                dotProduct(P[i]P[i+1],P[i+1]P[i+2]) > 180.0)
            return false;
    }
    return true;
}
```

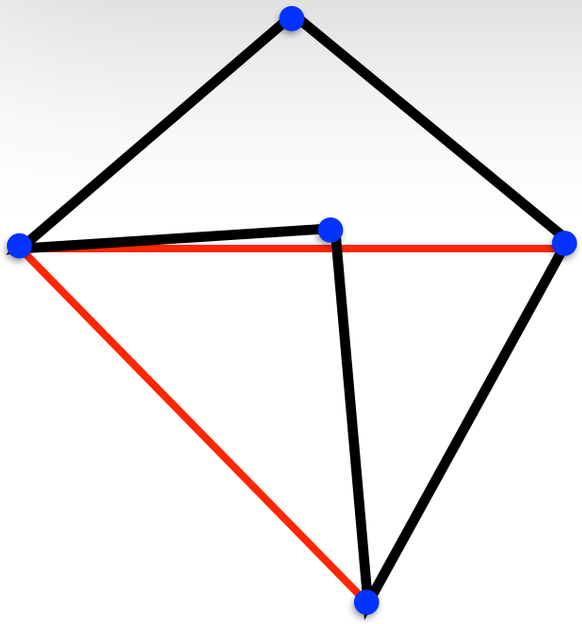
Polygons

Convexity?

```
bool convexP(const vector<point> &P) {
    int size = (int)P.size();
    if(size <= 3)
        return false;
    for(int i=1; i< size -1; i++) {
        if(arcos(|P[i]P[i+1]|*|P[i+1]P[i+2]|/
                 dotProduct(P[i]P[i+1],P[i+1]P[i+2]) > 180.0)
            return false;
    }
    return true;
}
```

costly operation!

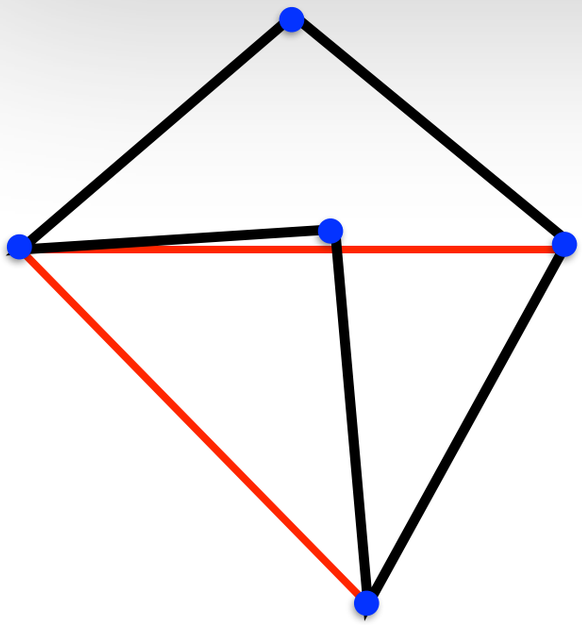
Polygons



$$A =$$

- Tessel the polygon from the origin and add up the areas
- Go through the points adding up the cross product (this removes areas not in the polygon)
- Take the absolute value at the end

Polygons

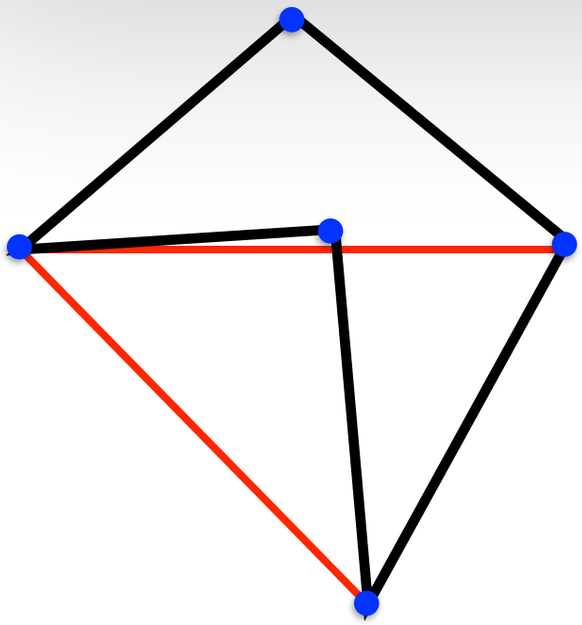


this isn't easy **at all**

A =

- Tessel the polygon from the origin and add up the areas
- Go through the points adding up the cross product (this removes areas not in the polygon)
- Take the absolute value at the end

Polygons

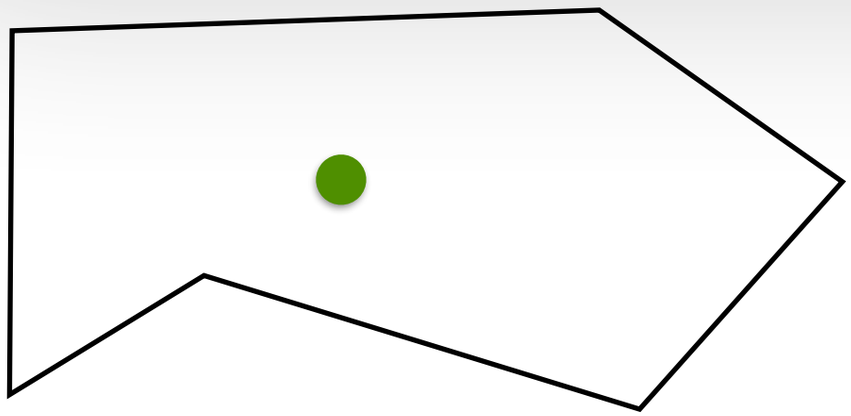


$$\mathbf{A} = 1/2 \det(P)$$
$$= 1/2(x_0y_1 + \ldots + x_{n-1}y_0 - x_1y_0 - \ldots - x_0y_{n-1})$$

```
double area(const vector<point> &P) {
    double res = 0.0, x1, y1, x2, y2;
    for(int i=0; i<P.size()-1; i++) {
        x1 = P[i].x;
        y1 = P[i].y;
        x2 = P[i+1].x;
        y2 = P[i+1].y;
        res += x1*y2 - x2*y1;
    }
    return fabs(res)/2.0;
}
```

Polygons

Point in P?

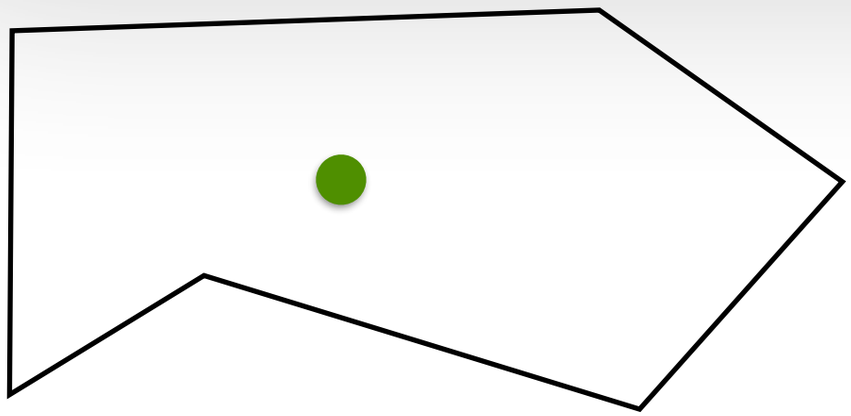


```
Polygon p = new Polygon();
p.contains(new Point(6, 3));
```

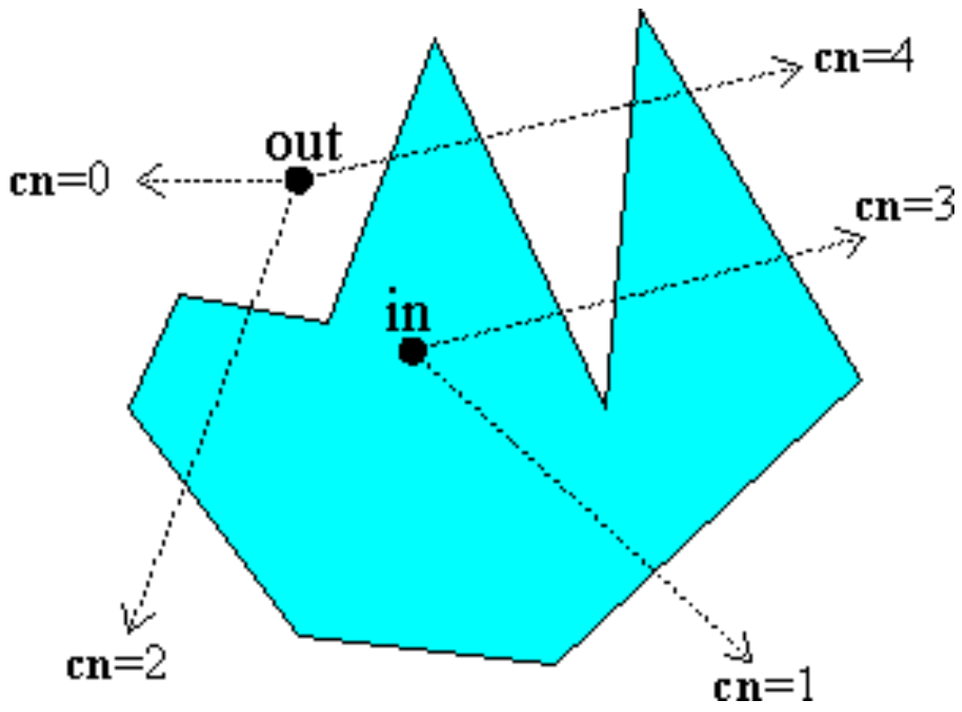
```
bool inPolygon(point pt, const vector<point> &P) {
  if((int)P.size() == 0) return false;
  double sum = 0;
  for(int i = 0; i < (int)P.size()-1; i++) {
    if(ccw(pt, P[i], P[i+1]))
      sum += angle(P[i], pt, P[i+1]);   // ccw
    else
      sum -= angle(P[i], pt, P[i+1]); } // cw
  return fabs(fabs(sum) - 2*PI) < EPS;
}
```

Polygons

Point in P?



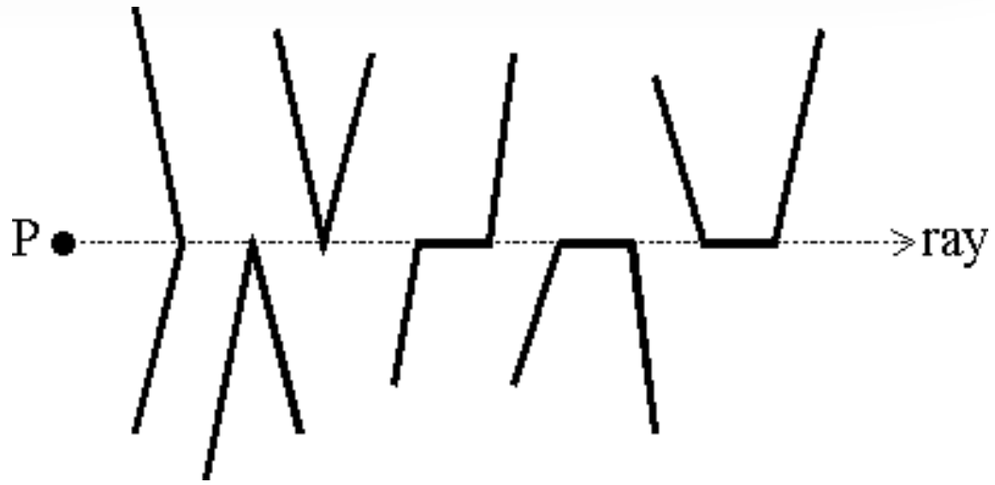
Jordan's theorem: every Jordan curve divides the plane into two regions, interior, exterior



Polygons

Point in P?

- add epsilon to the rays (accounting for the error)
- If there are intersections on an edge:
  - exclude horizontal rays
  - upward edges include the starting point and exclude the end point
  - downward edges include the end point and exclude the starting point



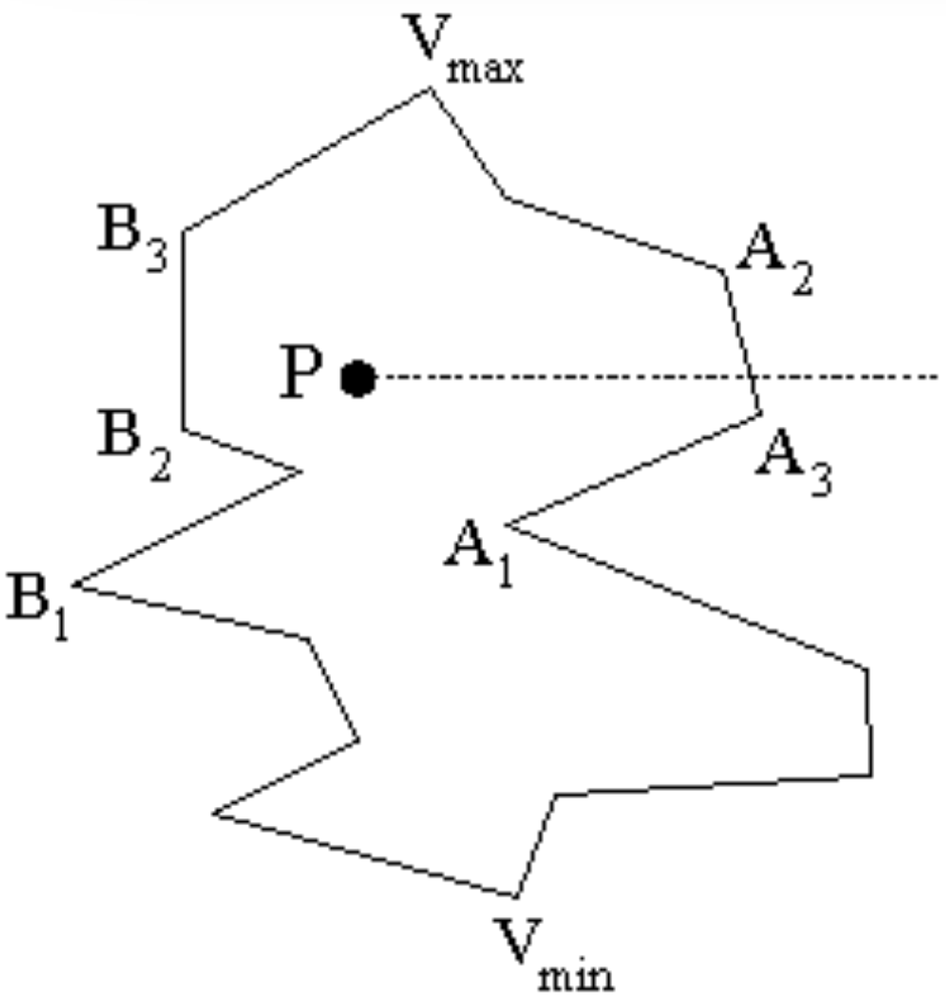
Polygons

Point in P?

```
int inPolygon(point p, point* v, int n) {
  int cn = 0;
  for(int i=0; i<n; i++) {    // edge from V[i]  to V[i+1]
    if (((v[i].y <= p.y) && (v[i+1].y > p.y)) || //upward crossing
     ((v[i].y > p.y) && (v[i+1].y <=  p.y))) {   //downward crossing
      // compute  the actual edge-ray intersect x-coordinate
      float vt = (float)(p.y  - v[i].y) / (v[i+1].y - v[i].y);
      if(p.x <  v[i].x + vt * (v[i+1].x - v[i].x)) // P.x < intersect
        ++cn;   // a valid crossing of y=P.y right of P.x
      }
  }
  return (cn&1);    // 0 if even (out), and 1 if  odd (in)

}
```

Polygons

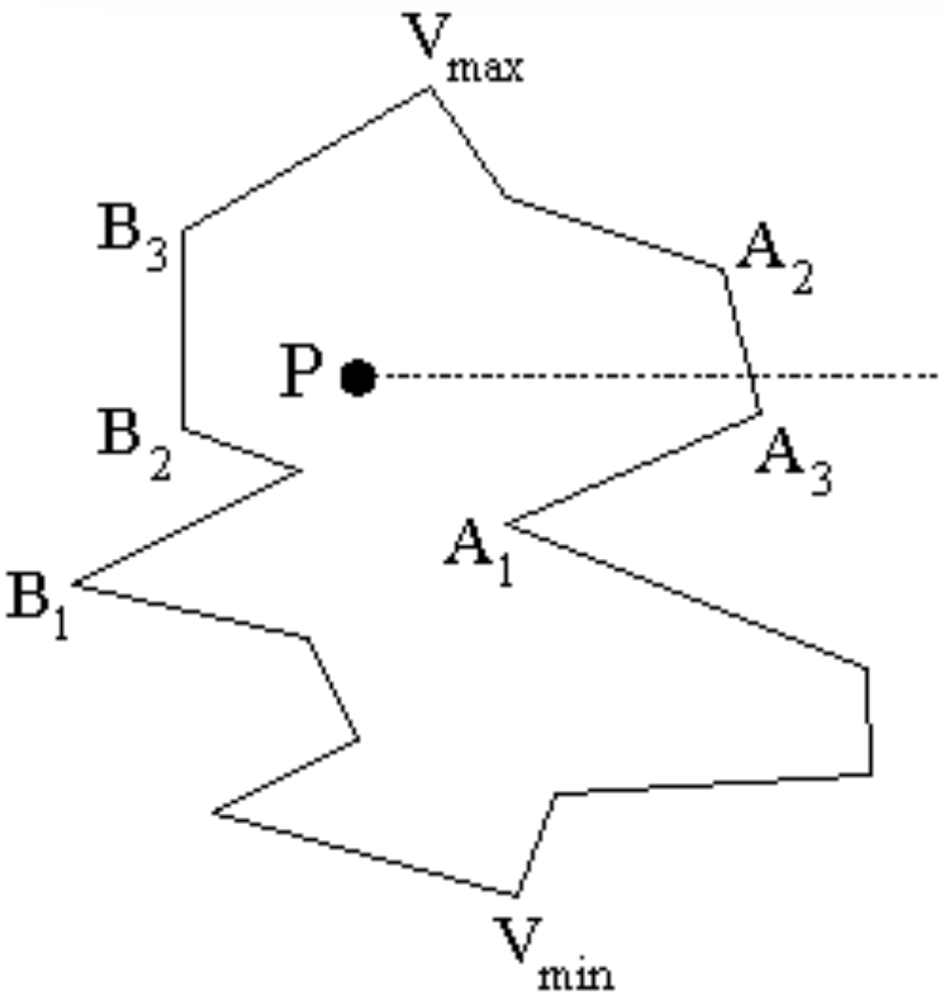
Point in P?



if the polygon is convex, use **cross product** to check if the point lies in the same side for all arcs.

Polygons

Point in P?



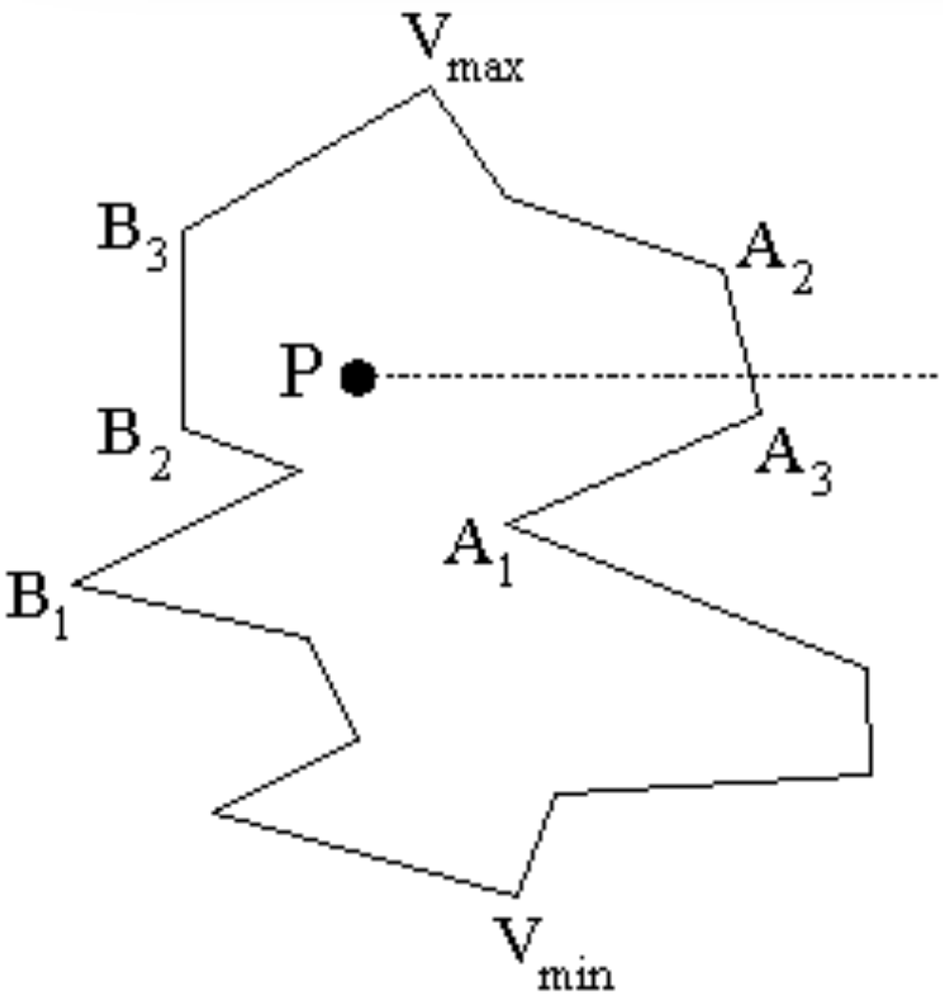
if the polygon is convex, use **cross product** to check if the point lies in the same side for all arcs.

O(😕)

Order the points from the **extremes** use binary search to check whether they intersect the ray from P.

Polygons

Point in P?



if the polygon is convex, use **cross product** to check if the point lies in the same side for all arcs.

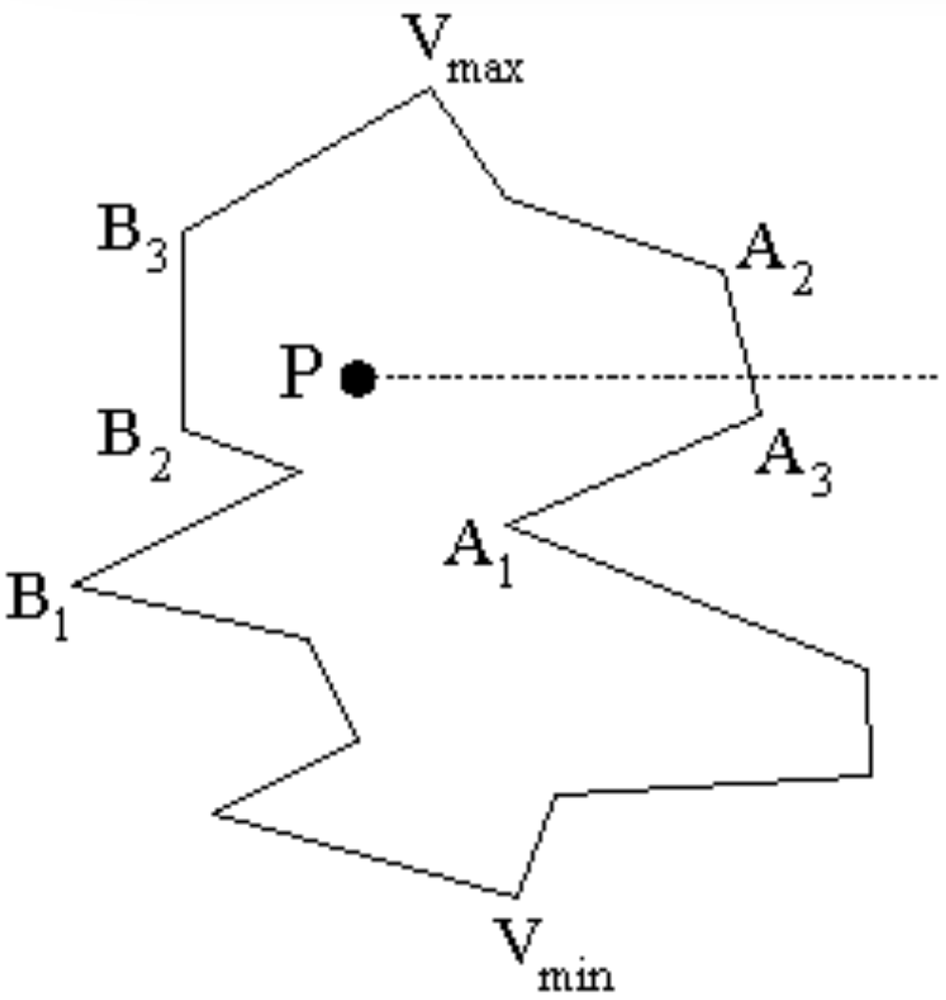
O(😕)

Order the points from the **extremes** use binary search to check whether they intersect the ray from P.

O(😃)

Polygons

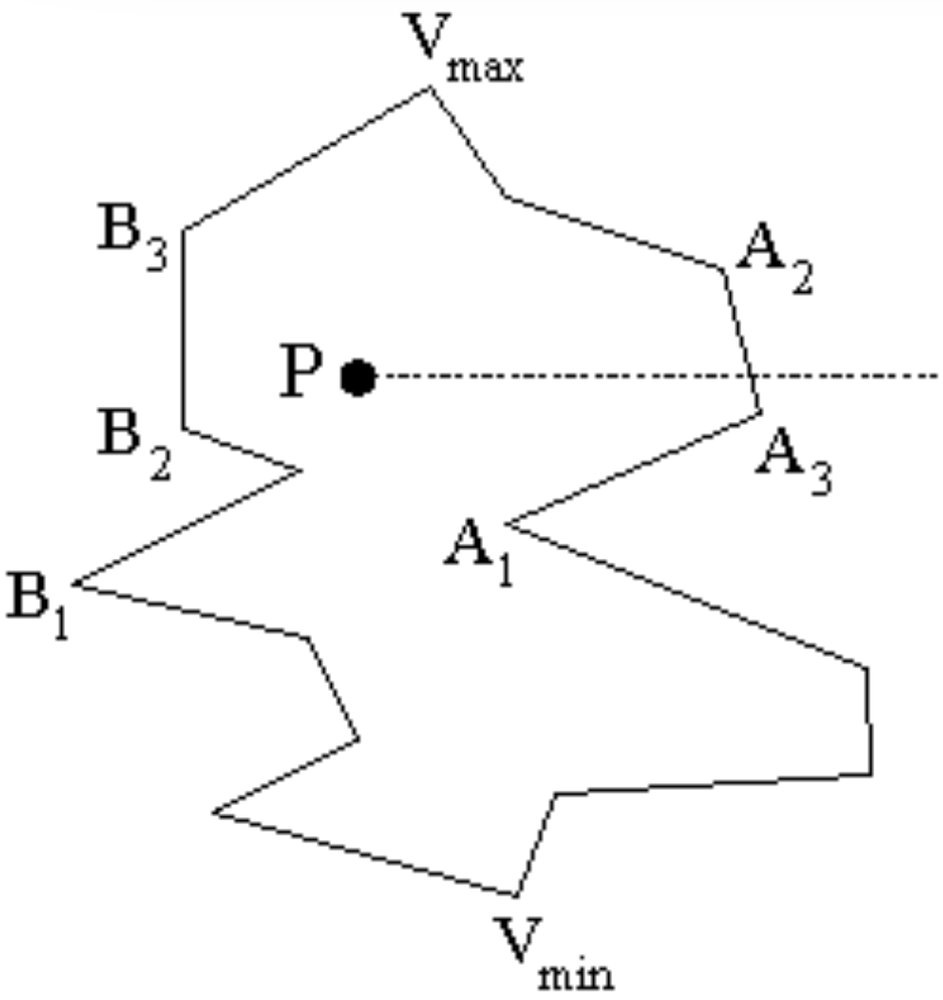
Point in P?



if the polygon is convex, use **cross product** to check if the point lies in the same side for all arcs.

Polygons

Point in P?



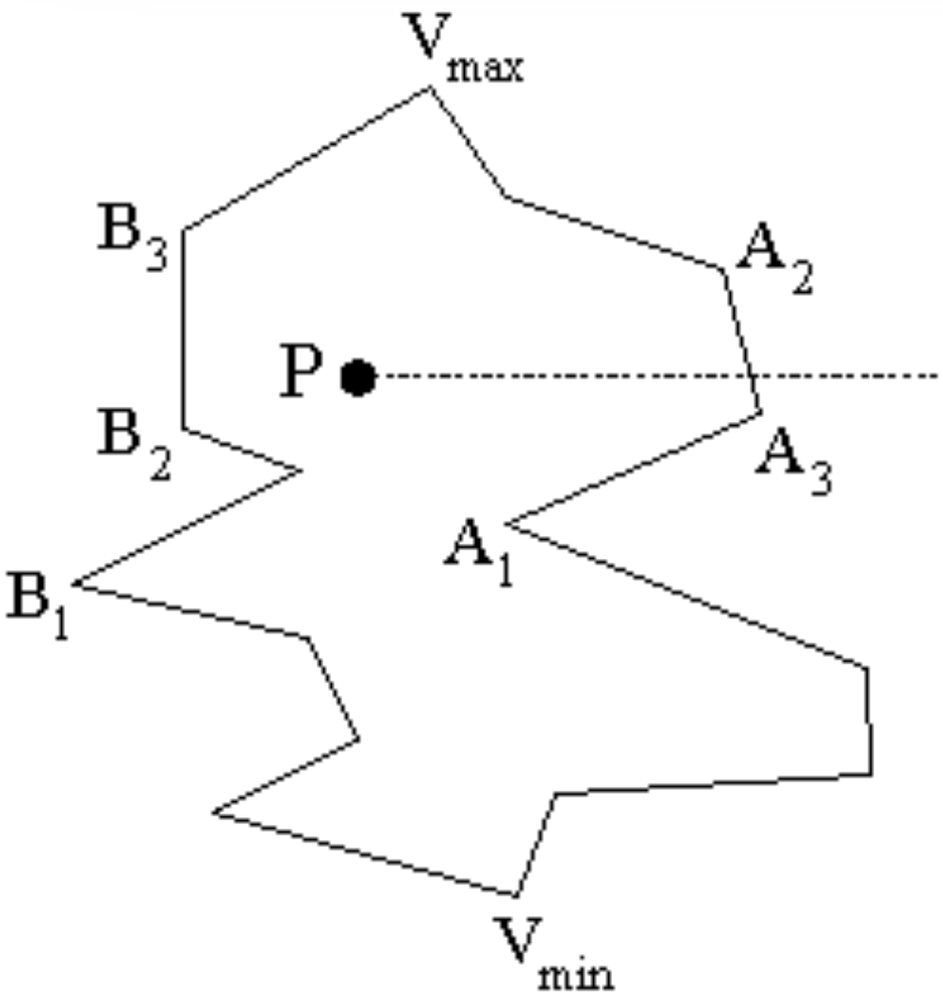
if the polygon is convex, use **cross product** to check if the point lies in the same side for all arcs.

O(😕)

Order the points from the **extremes** use binary search to check whether they intersect the ray from P.

Polygons

Point in P?



if the polygon is convex, use **cross product** to check if the point lies in the same side for all arcs.

O(😕)

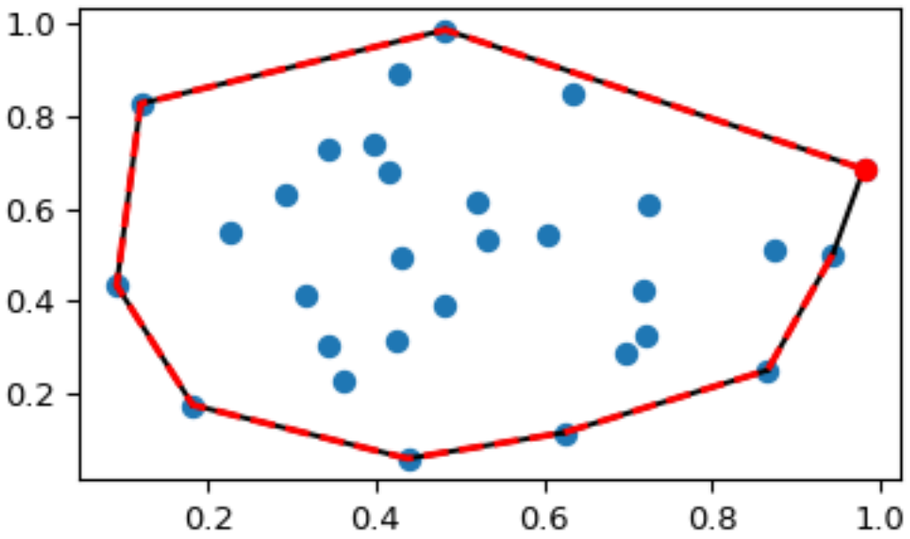
Order the points from the **extremes** use binary search to check whether they intersect the ray from P.

O(😃)

ALGORITHMS

Convex hull

The **convex hull** is the smallest polygon that contains a set of points

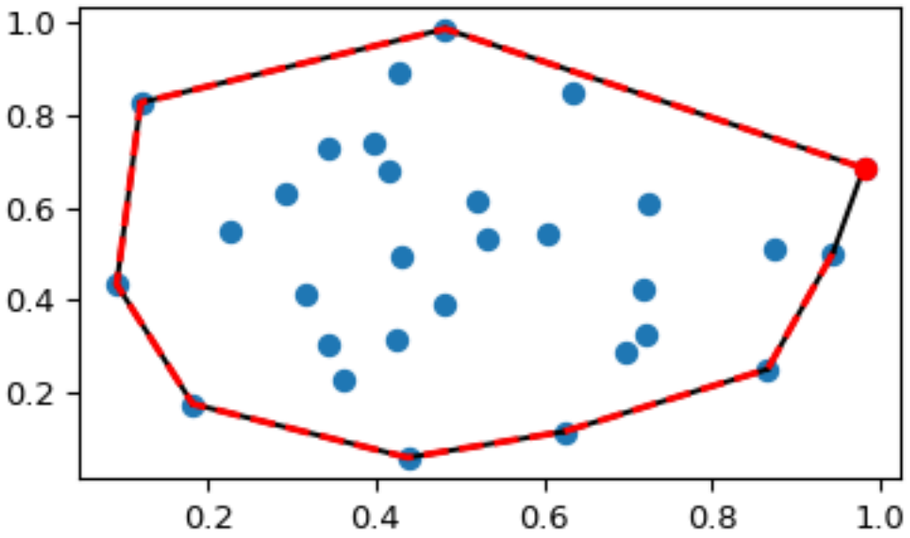


Idea:

- For every two points draw the line between them. Check if **all other points** are on the same side
- An edge is in the convex hull iff all the points are in the same side

Convex hull

The **convex hull** is the smallest polygon that contains a set of points

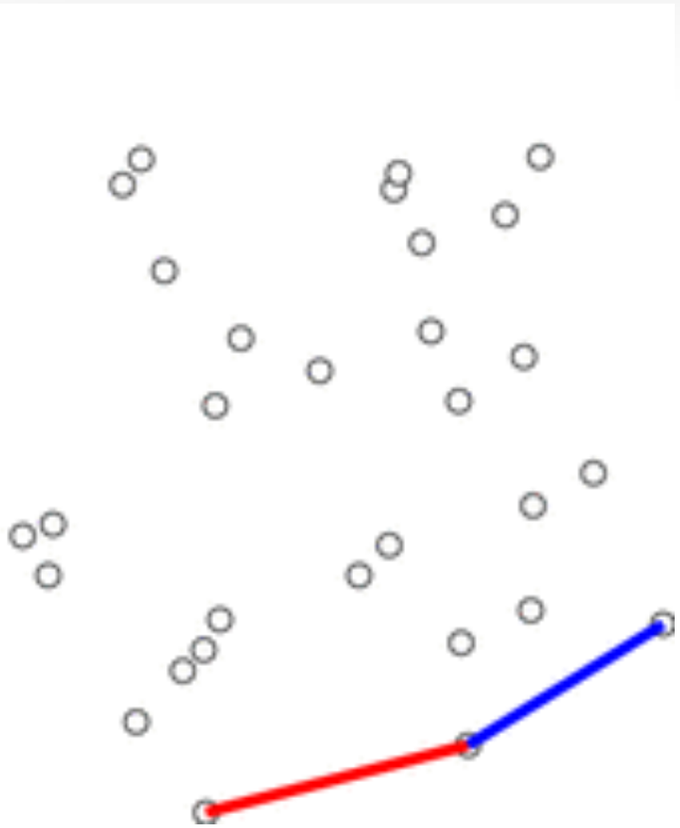


Idea:

- For every two points draw the line between them. Check if **all other points** are on the same side
- An edge is in the convex hull iff all the points are in the same side

O(😭)

Convex hull

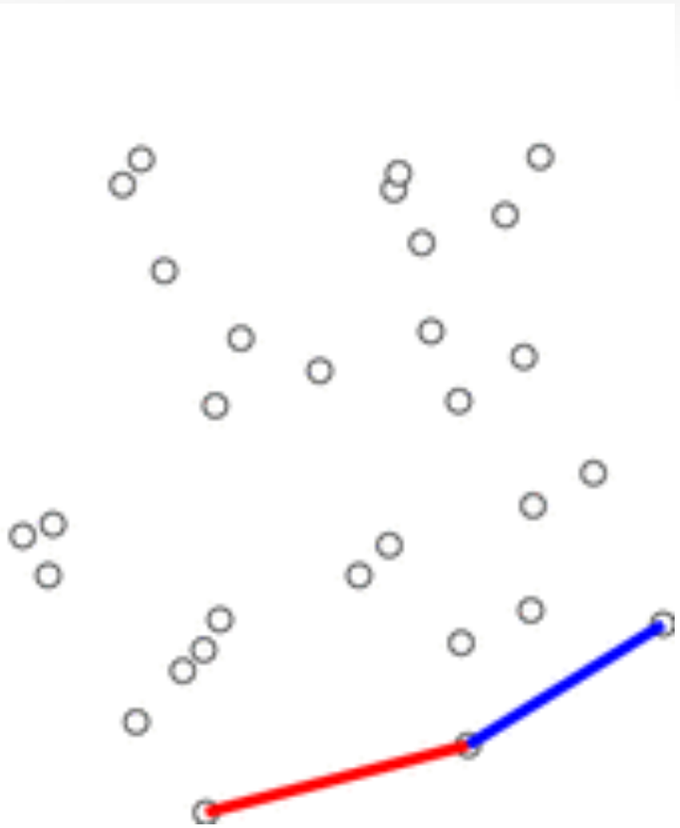


Take two points on the convex hull (normally the ends)

Split the points to the *top* and *bottom* set of points with respect to the line

Order the points (on the x coordinate). The points *above* the line in ascending order and the points *below* the line in descending order.

Convex hull

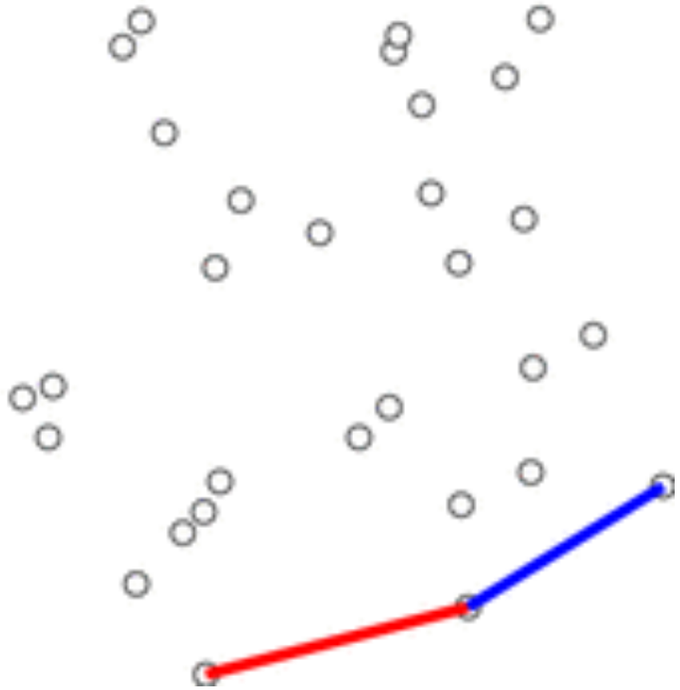


Take two points on the convex hull (normally the ends)

Split the points to the *top* and *bottom* set of points with respect to the line

Order the points (on the x coordinate). The points *above* the line in ascending order and the points *below* the line in descending order.

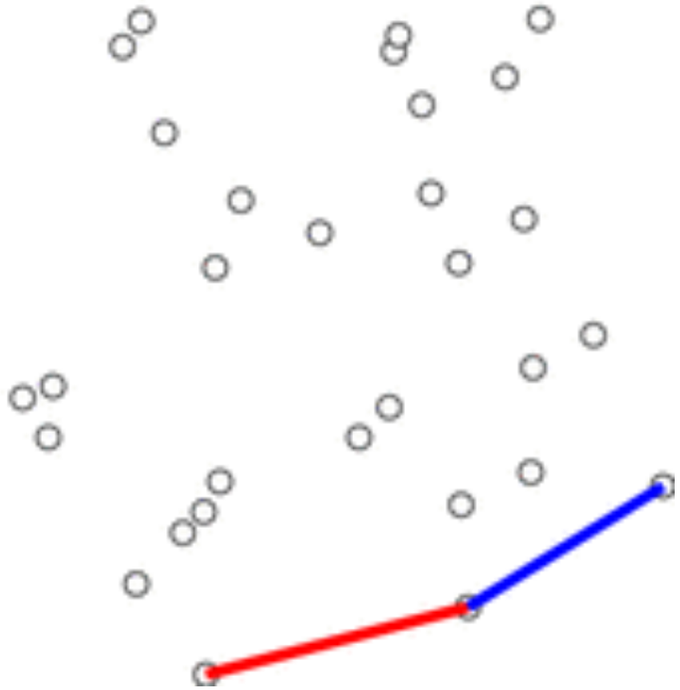
Convex hull



Keep a stack of the point in the convex hull

If the next point to add has an angle greater than 180, remove the top of the stack and try again

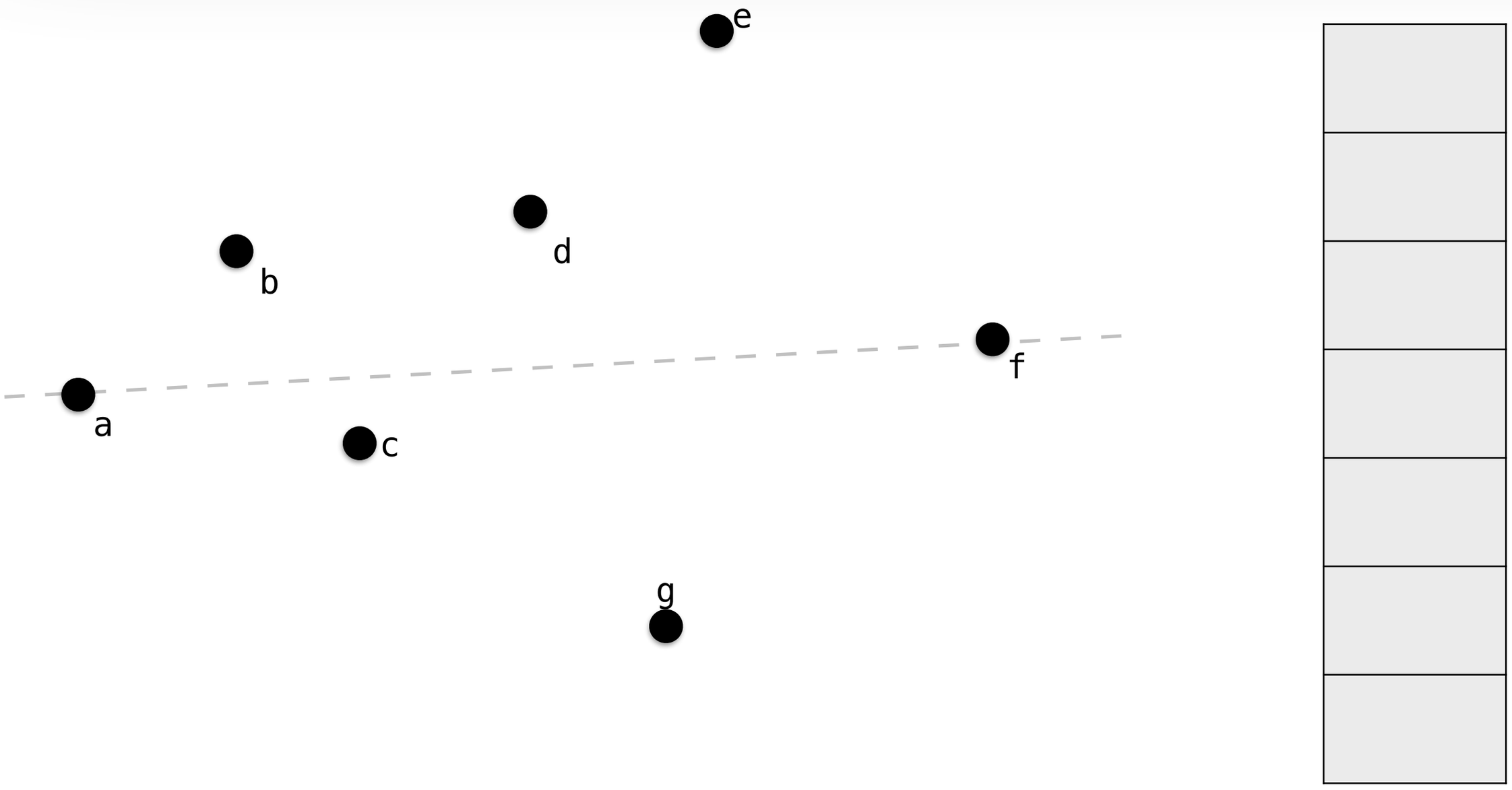
Convex hull



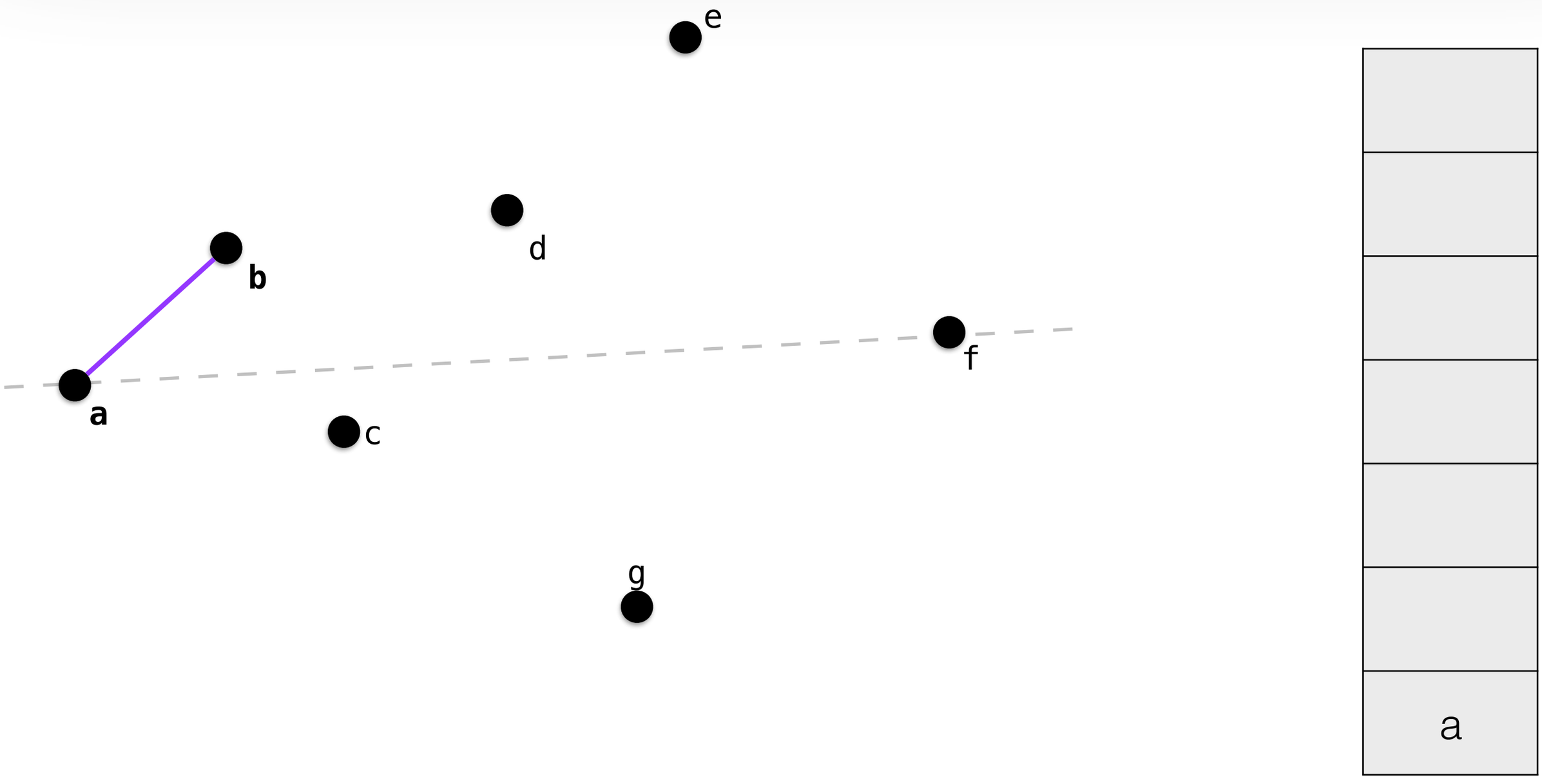
Keep a stack of the point in the convex hull

If the next point to add has an angle greater than 180, remove the top of the stack and try again

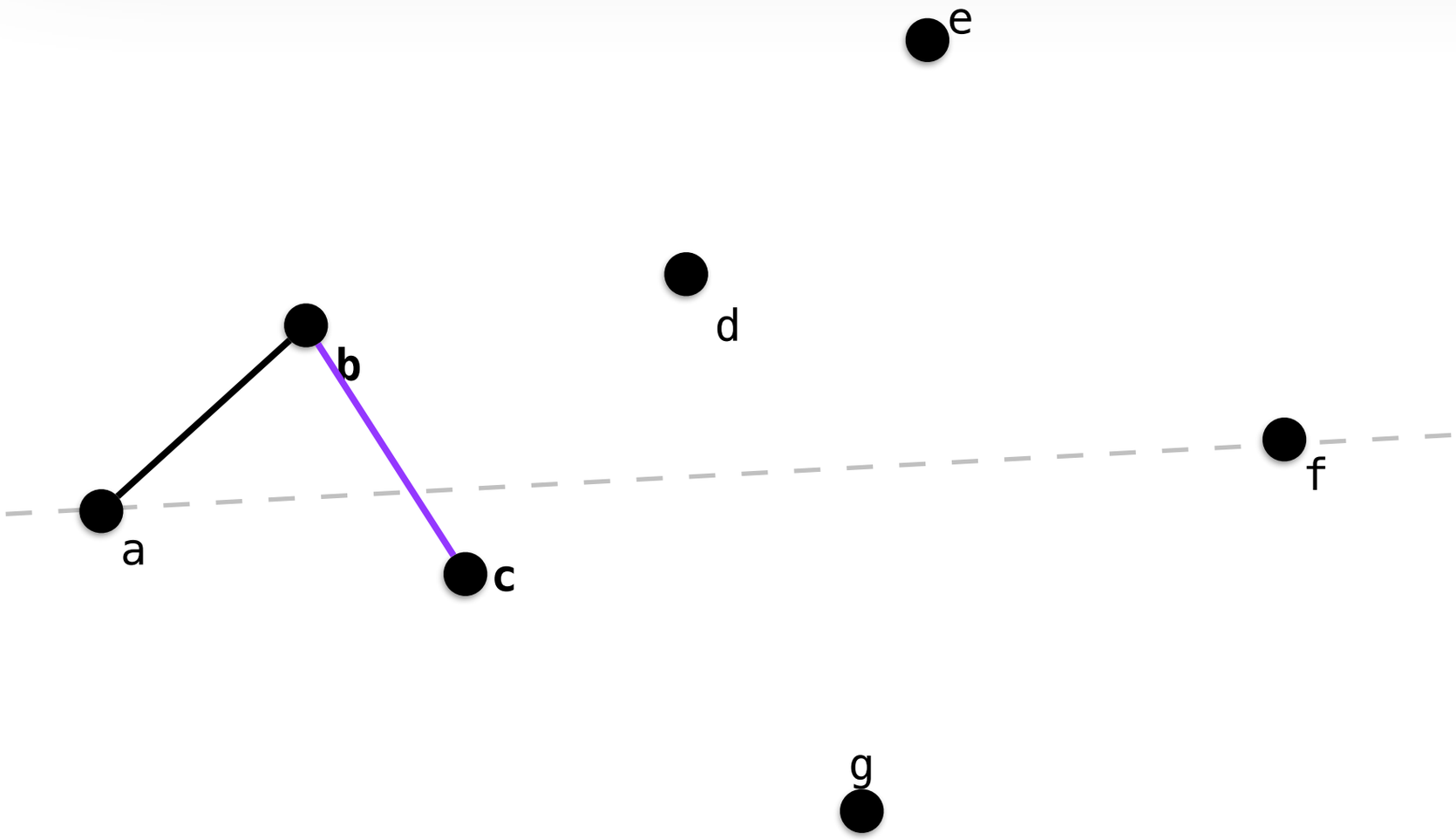
Convex hull

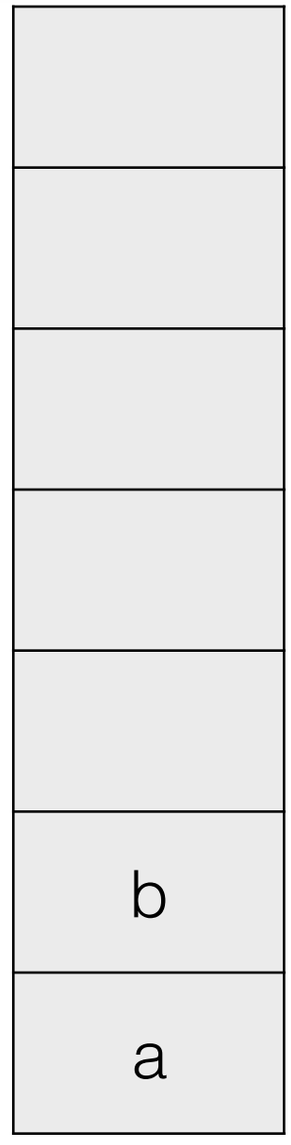


Convex hull

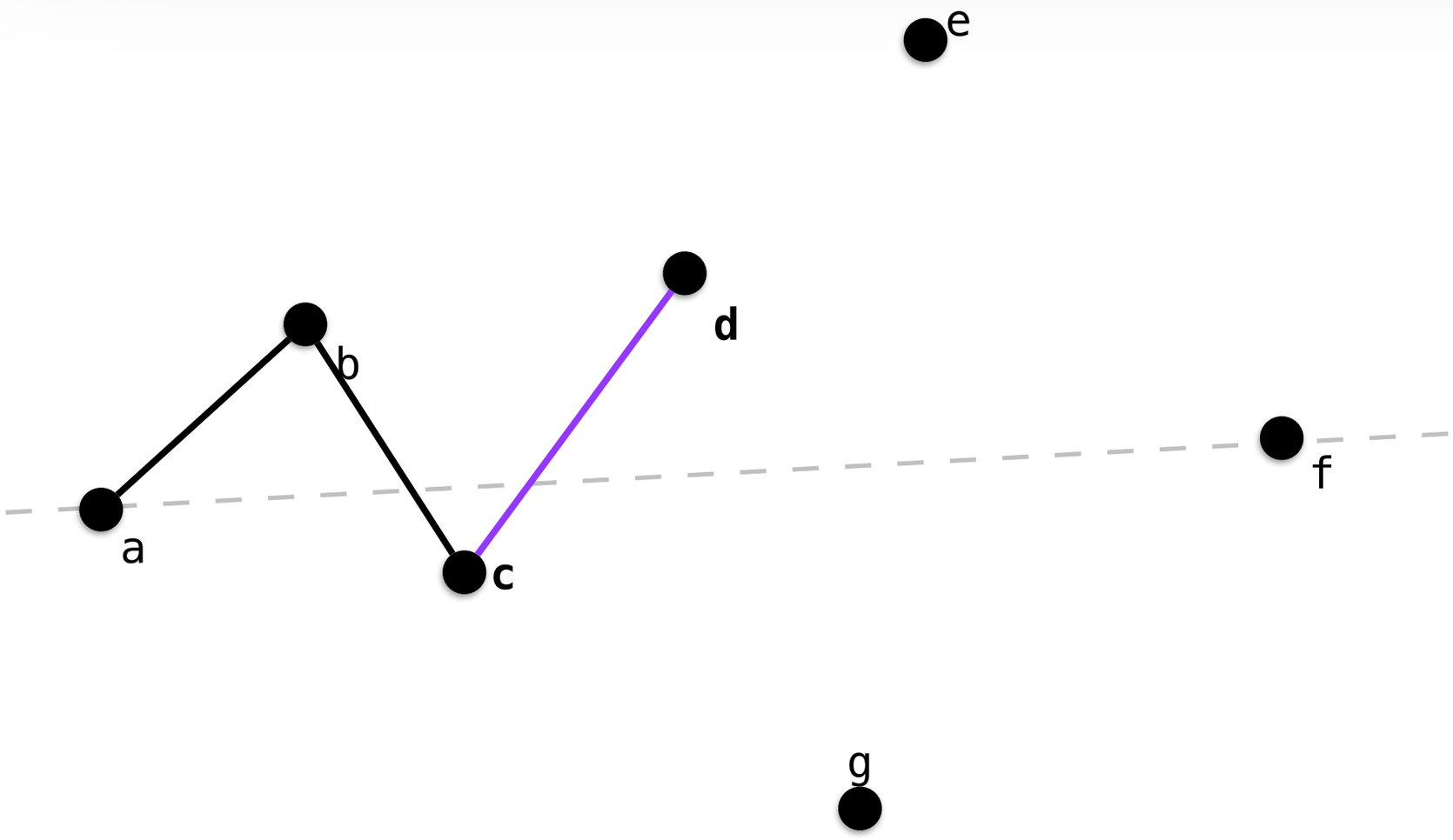


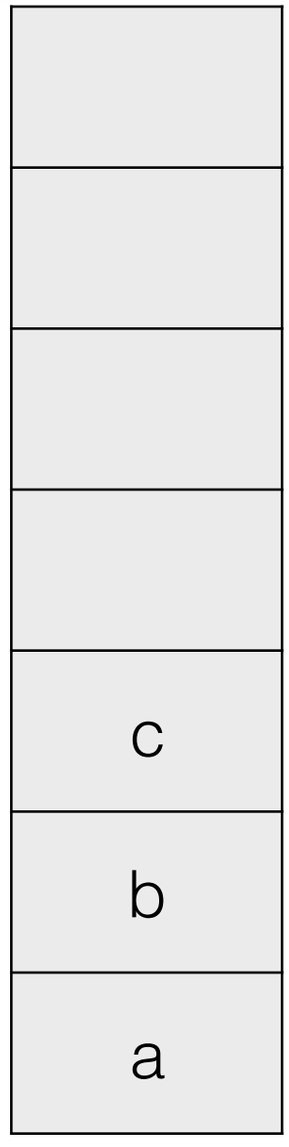
Convex hull



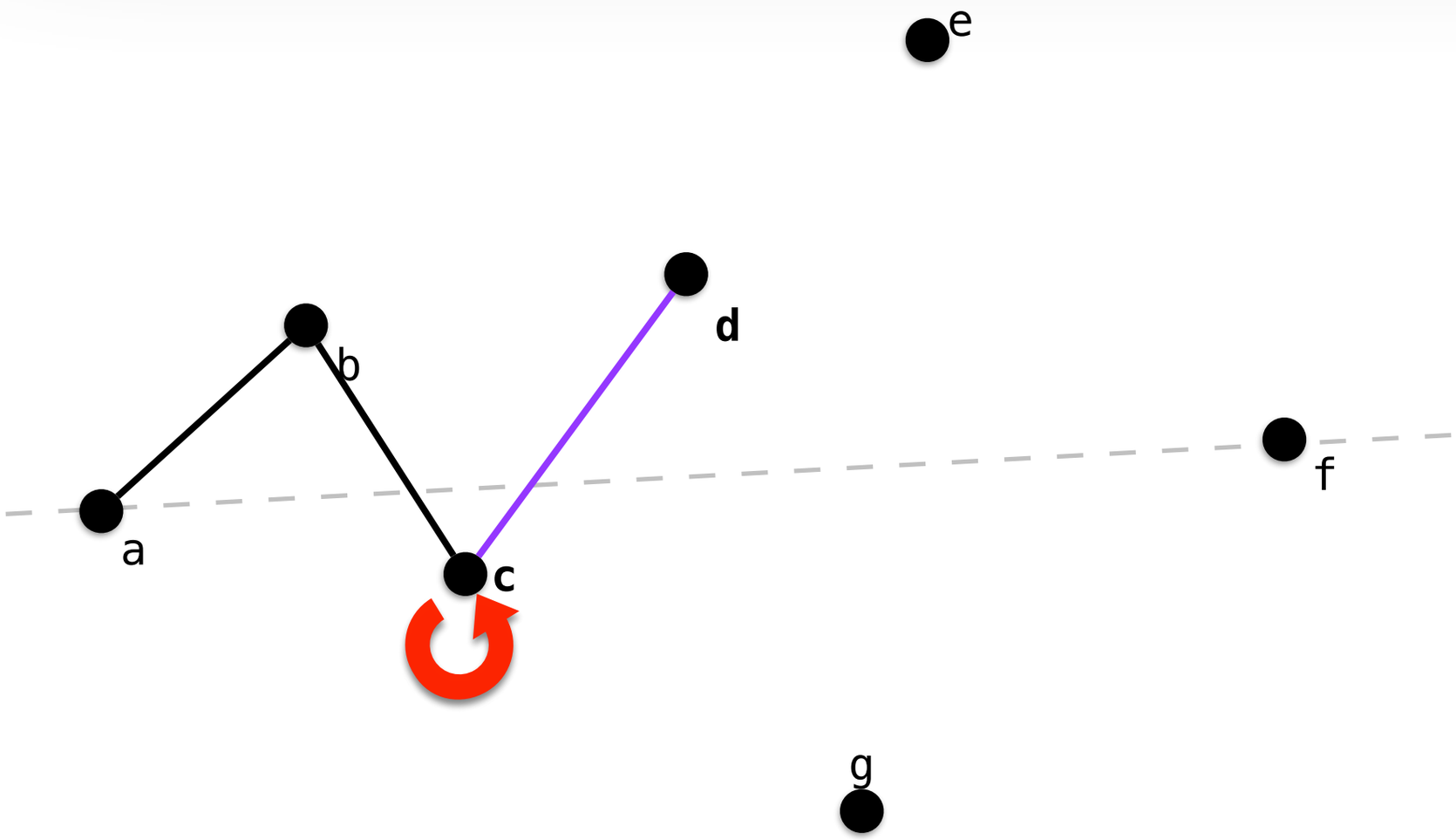


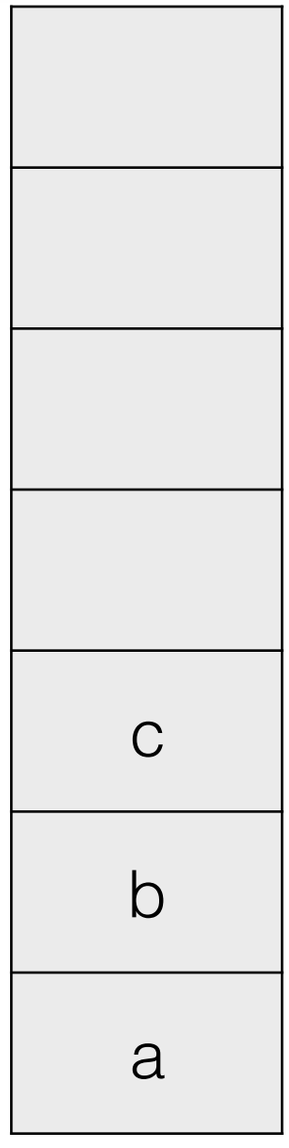
Convex hull



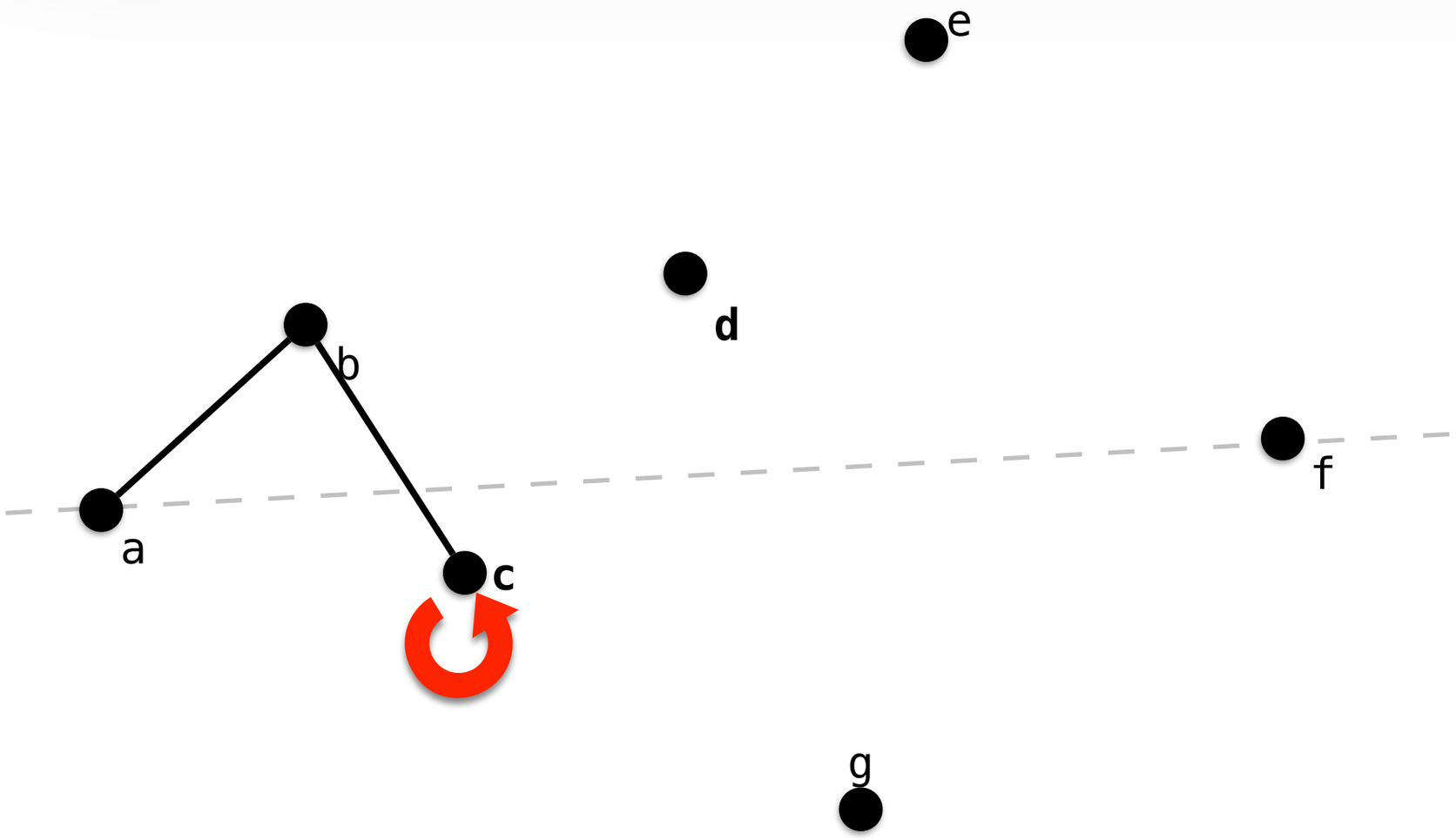


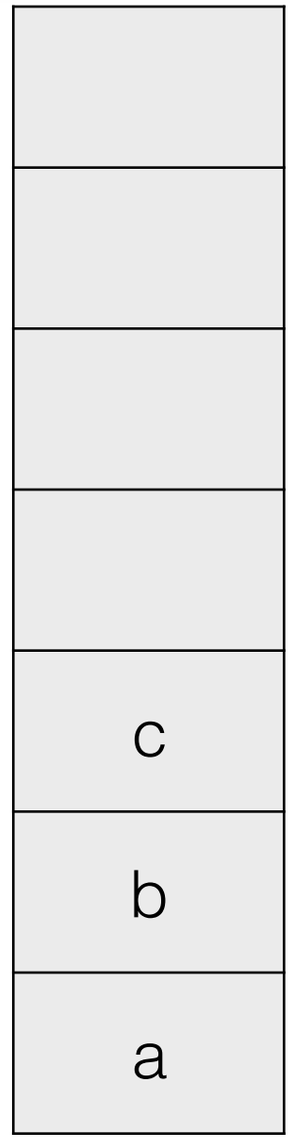
Convex hull



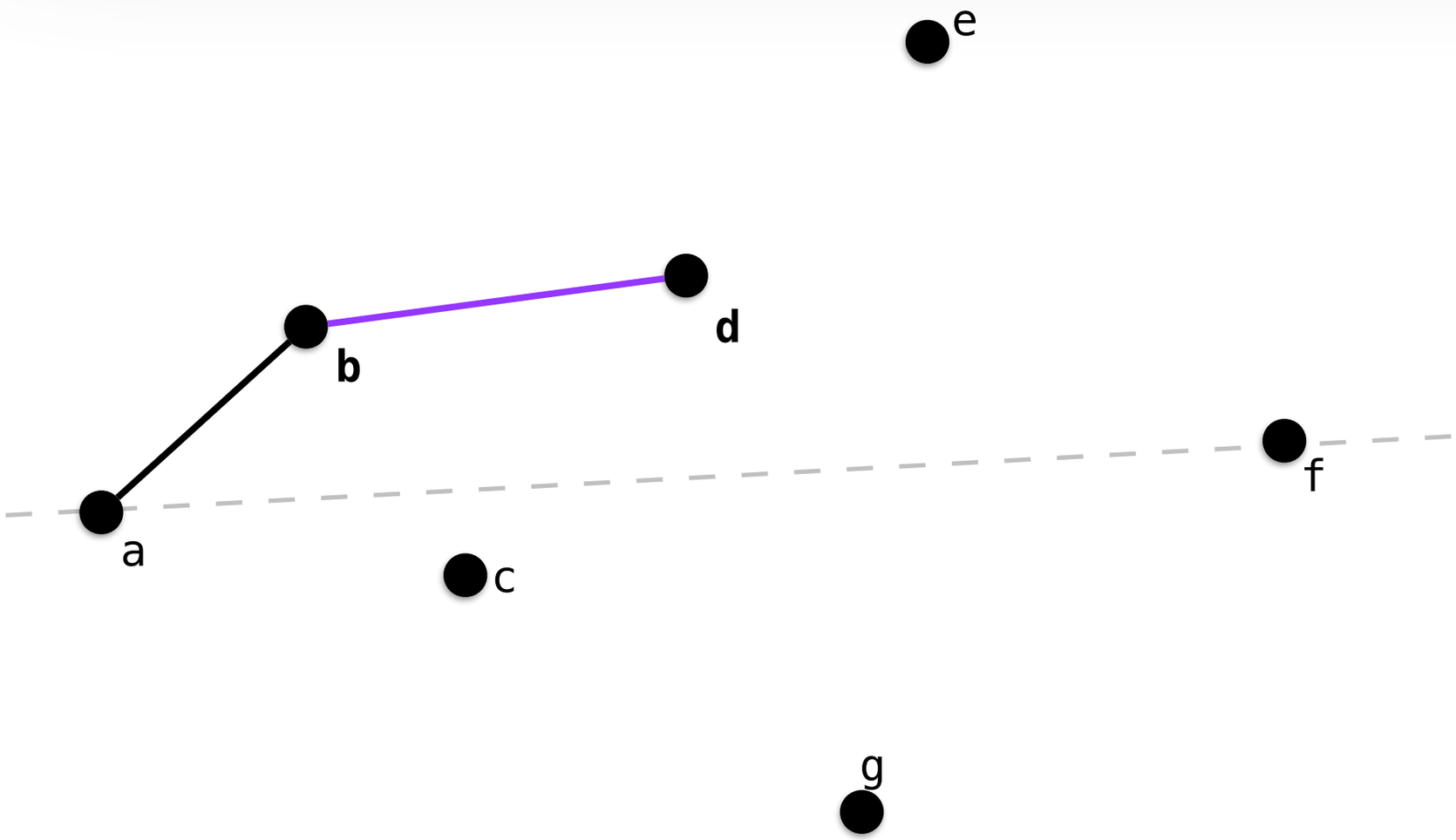


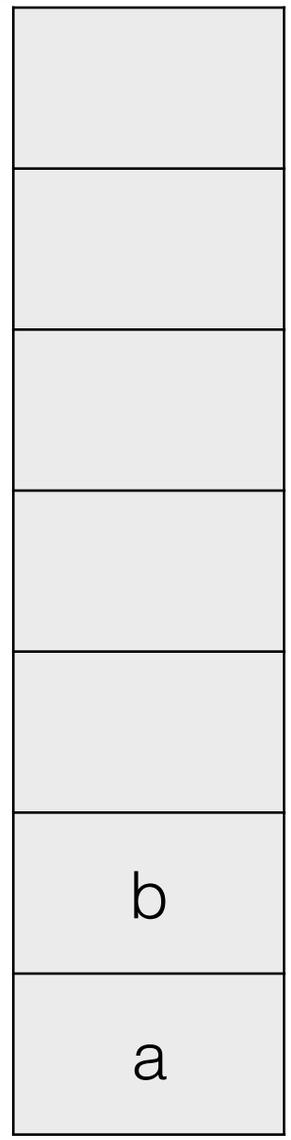
Convex hull



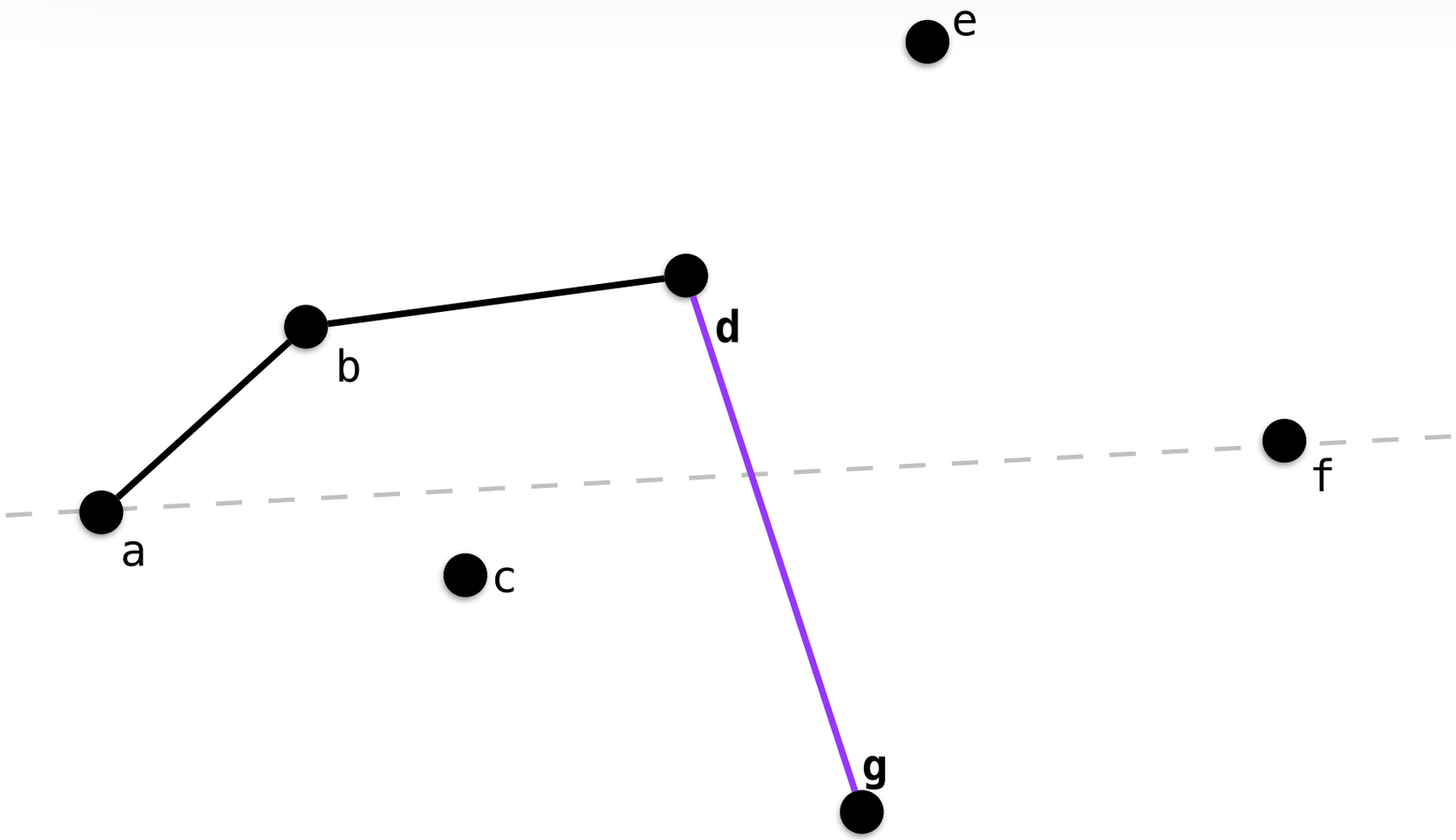


Convex hull

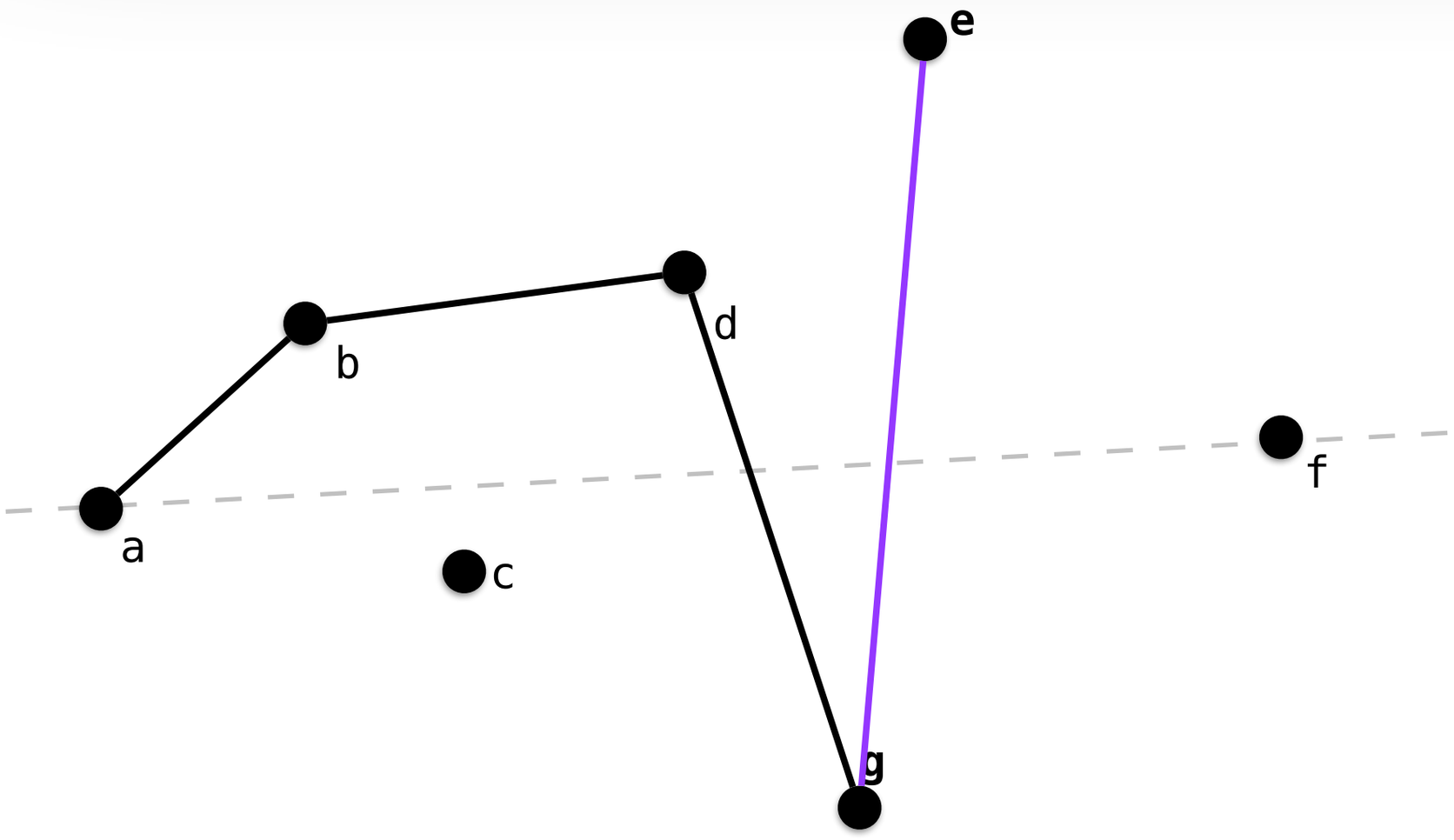


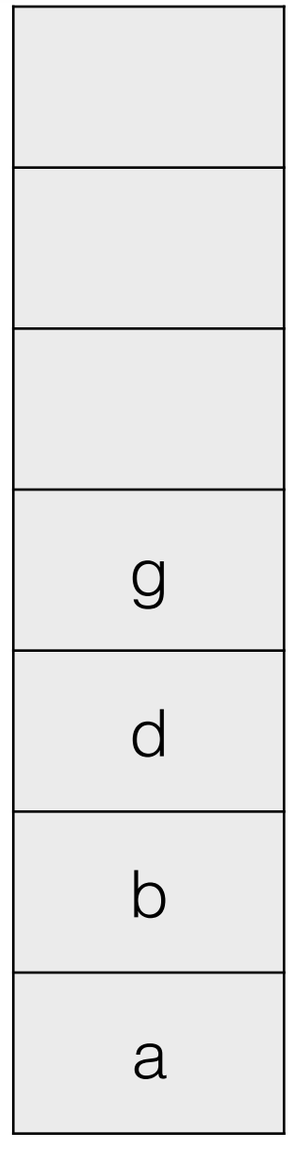


Convex hull

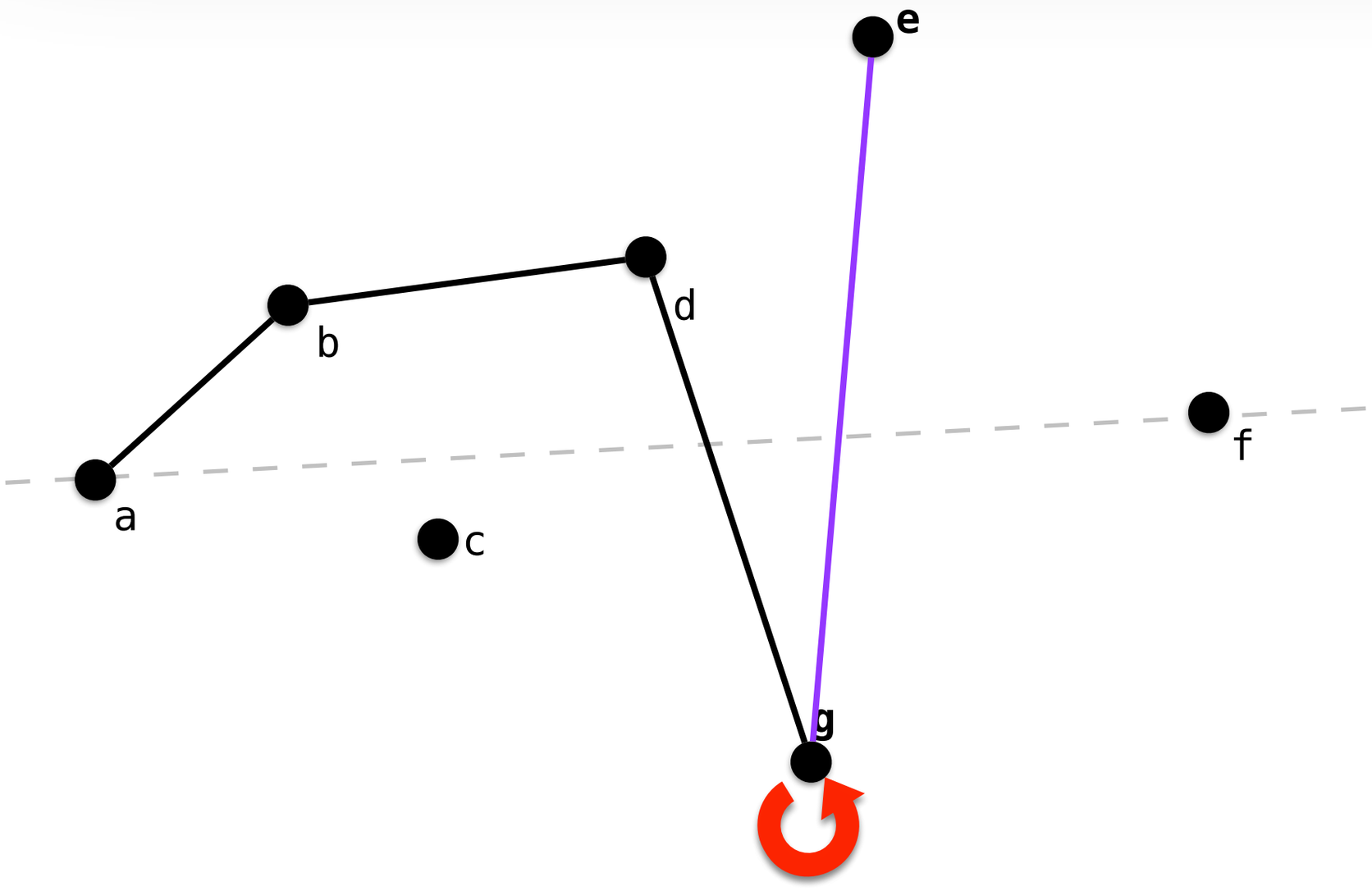


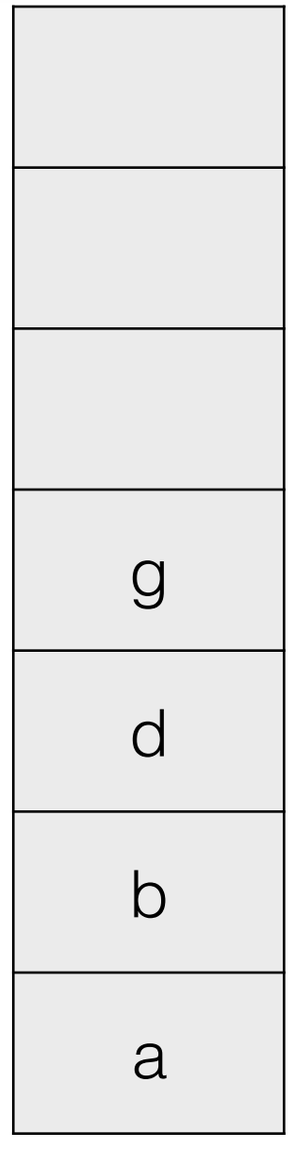
Convex hull



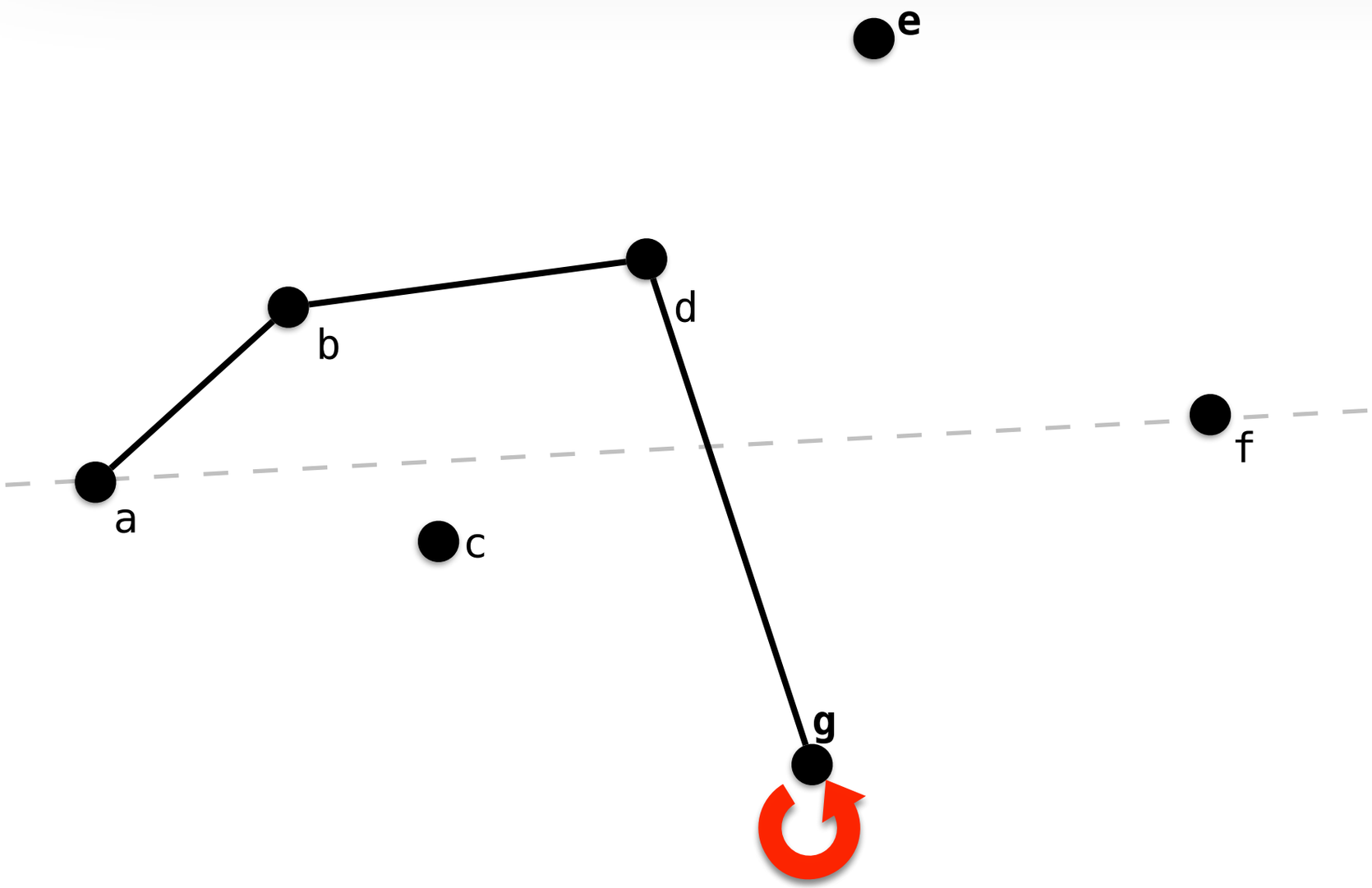


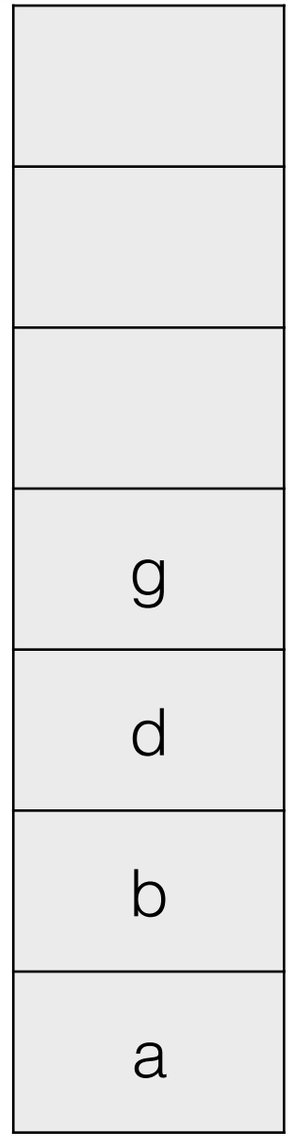
Convex hull



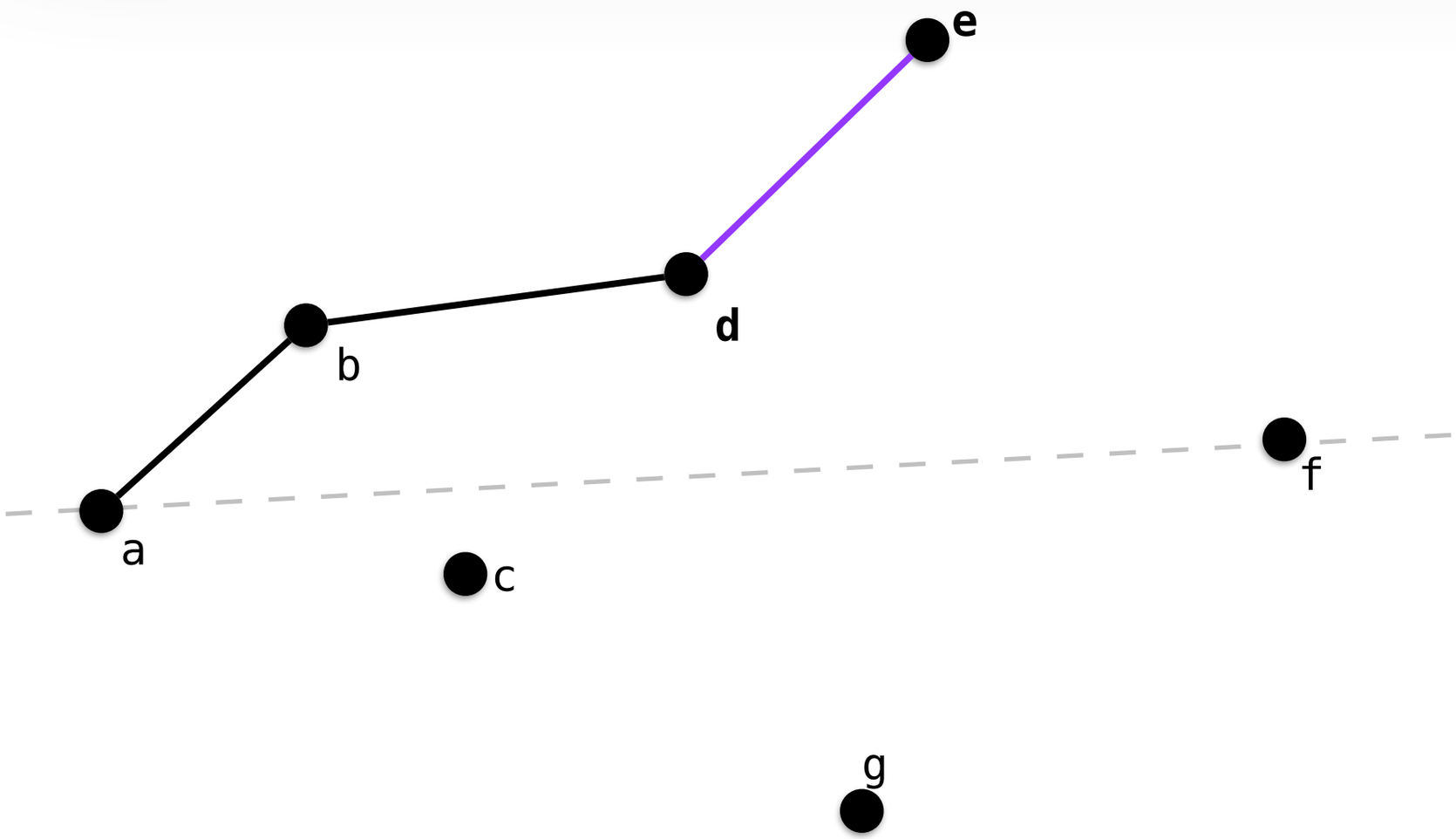


Convex hull

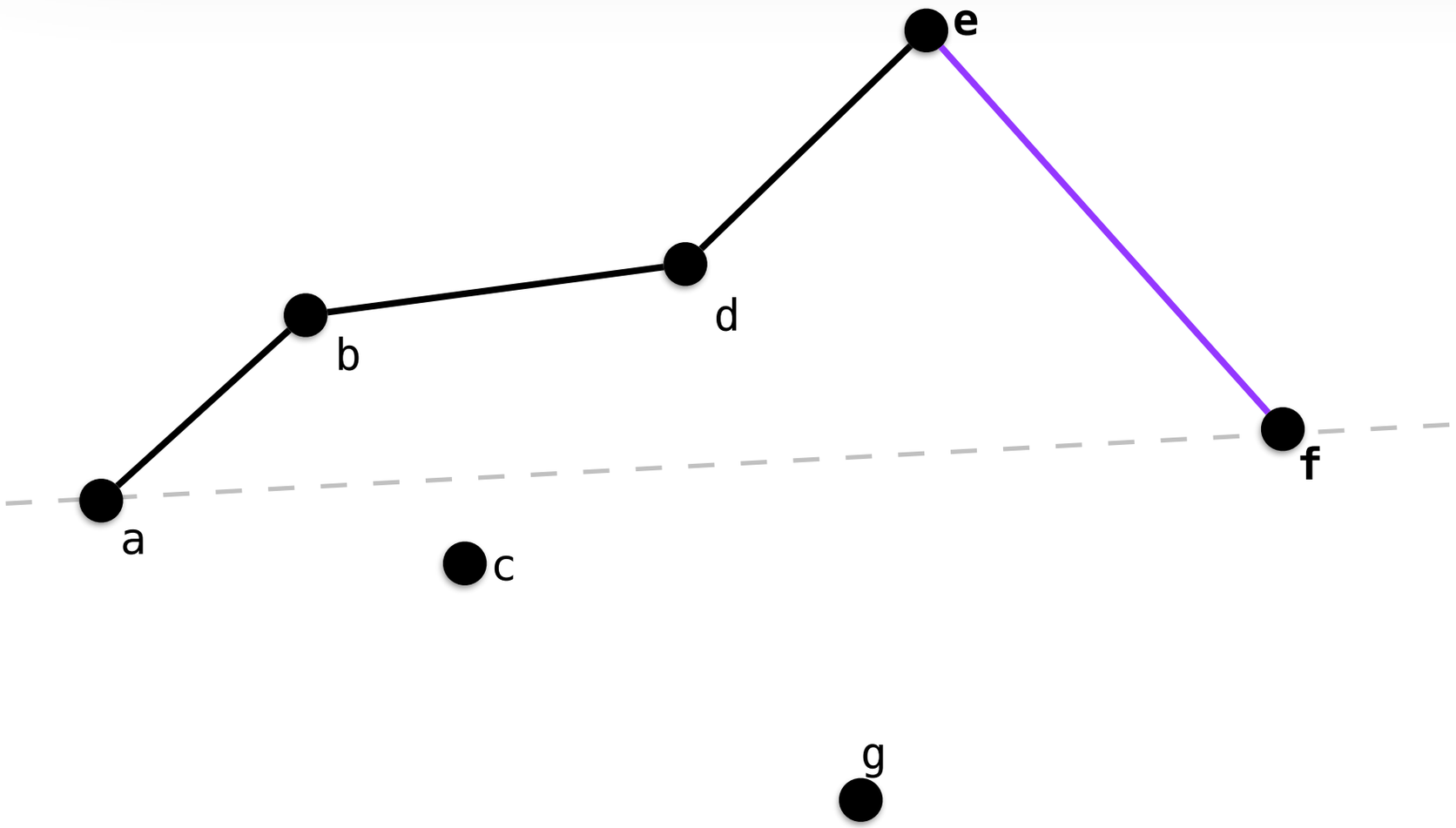


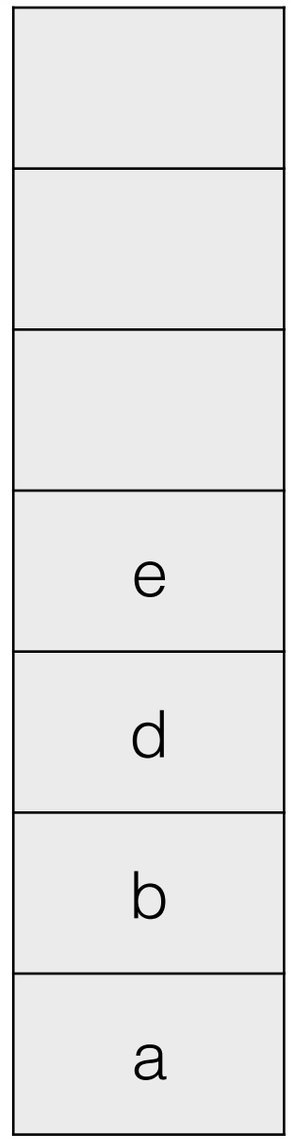


Convex hull

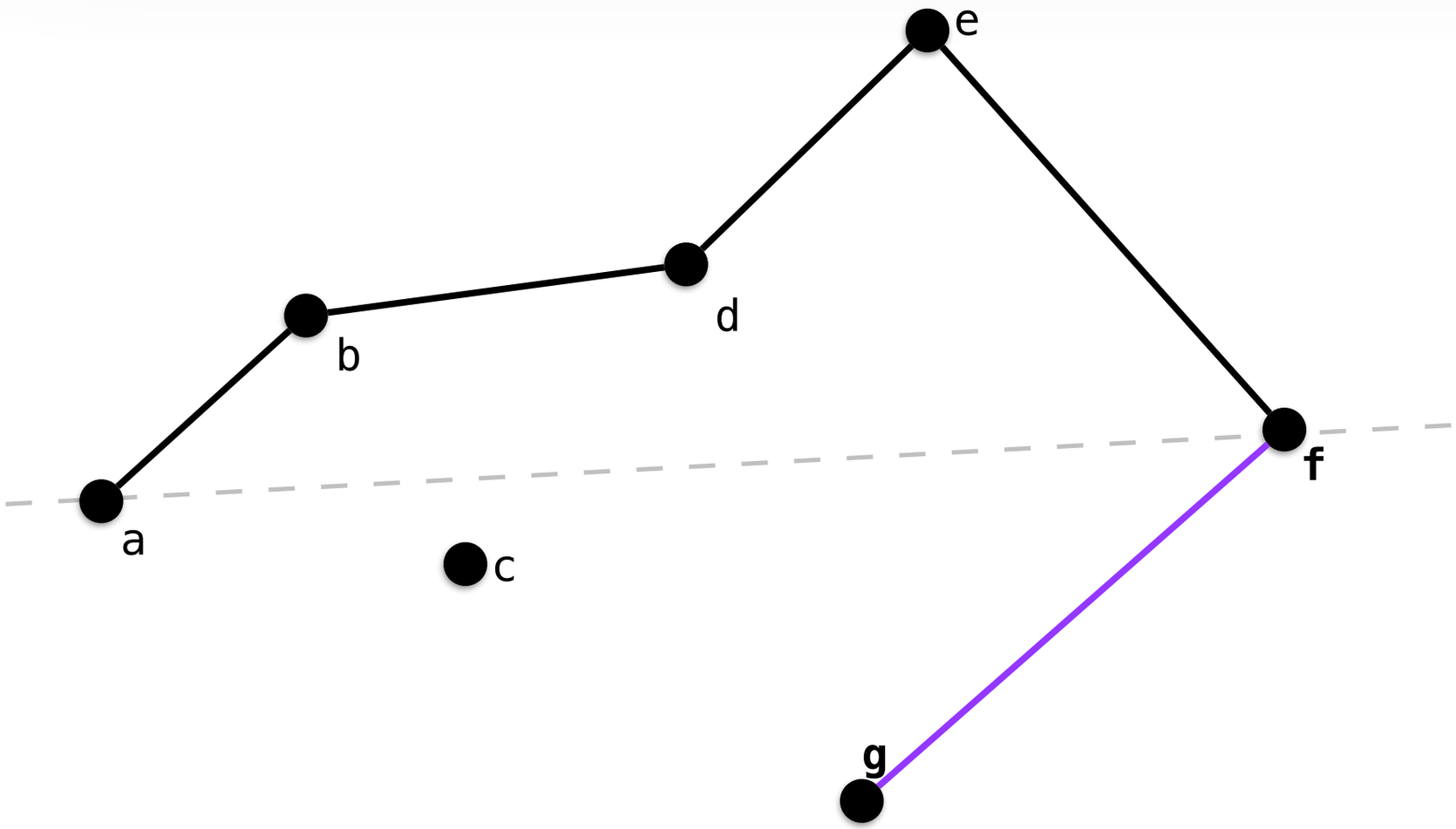


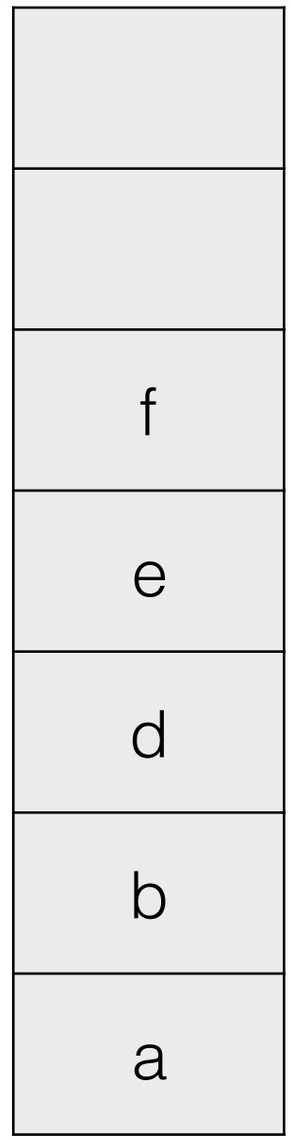
Convex hull



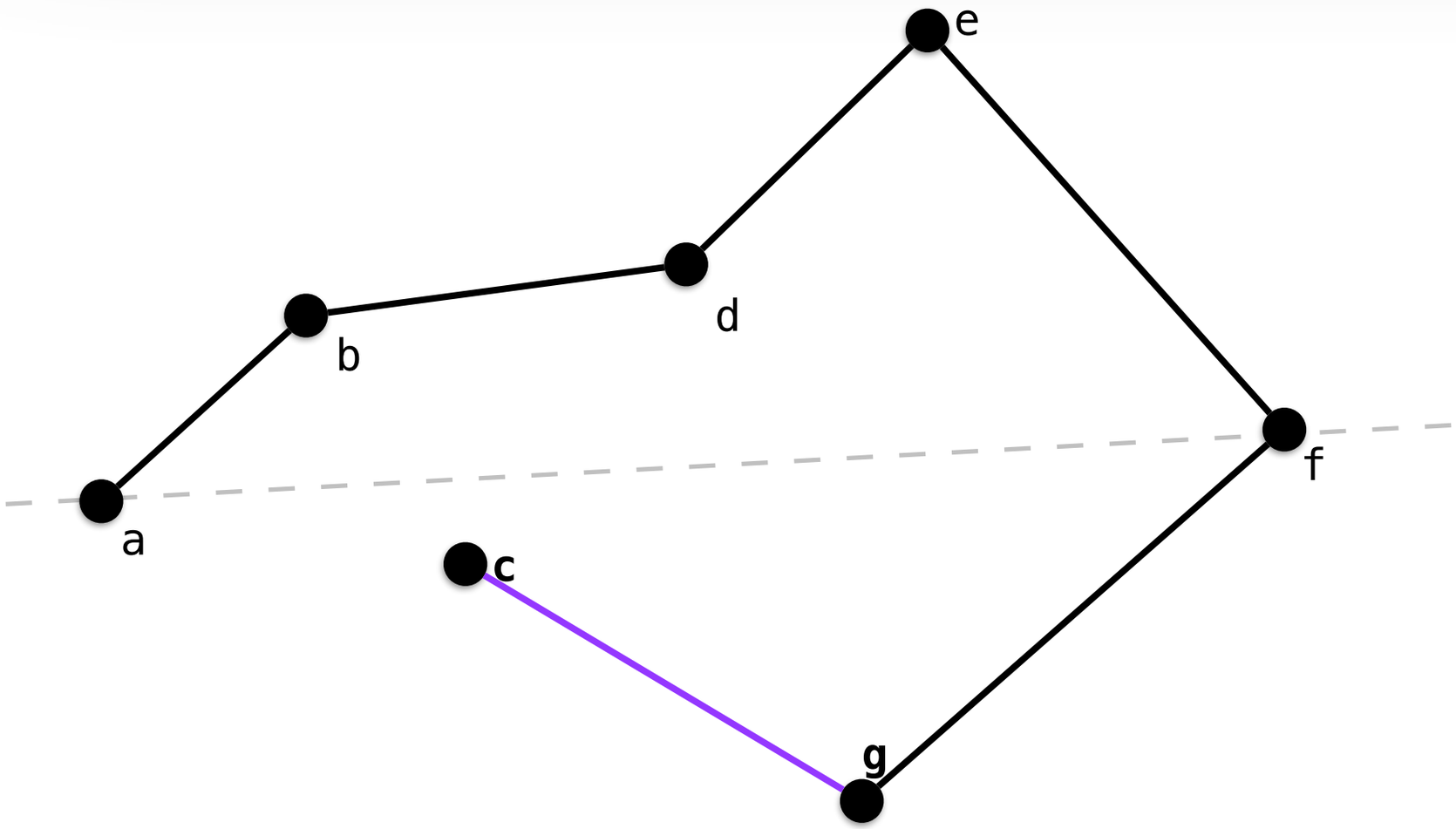


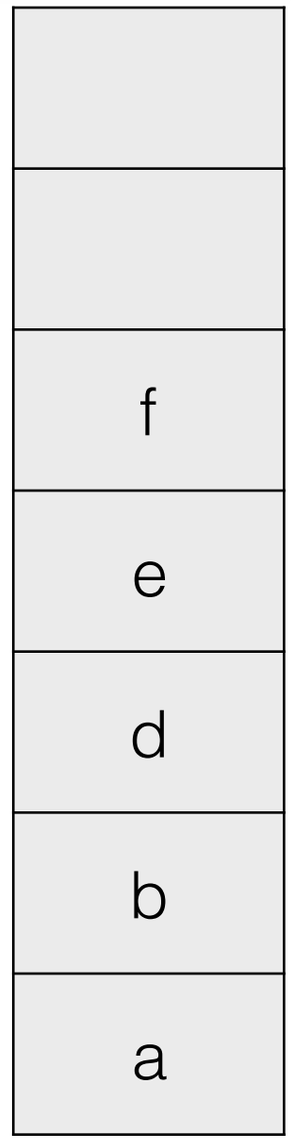
Convex hull



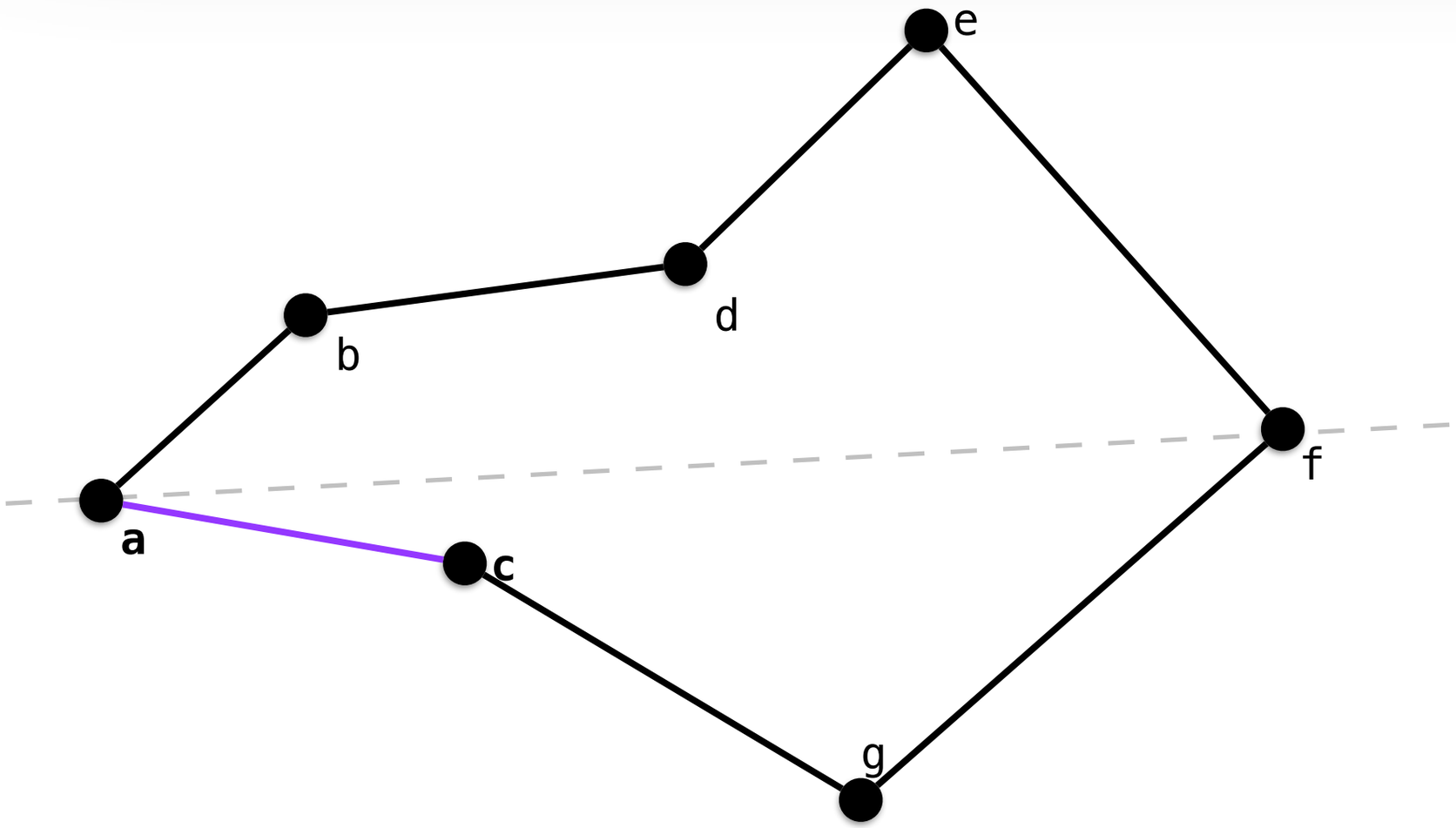


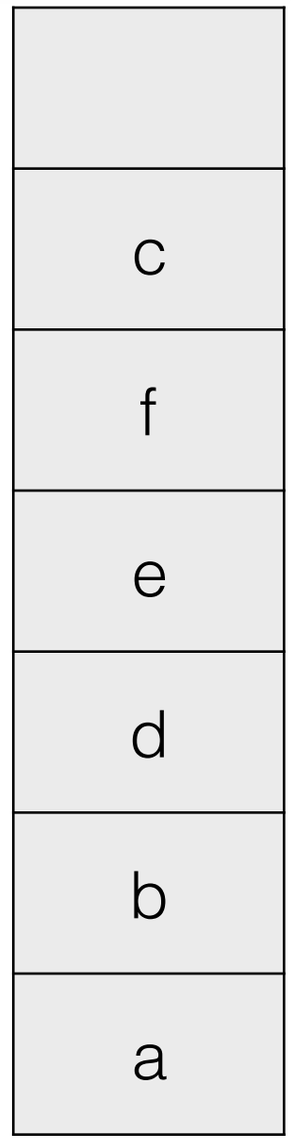
Convex hull



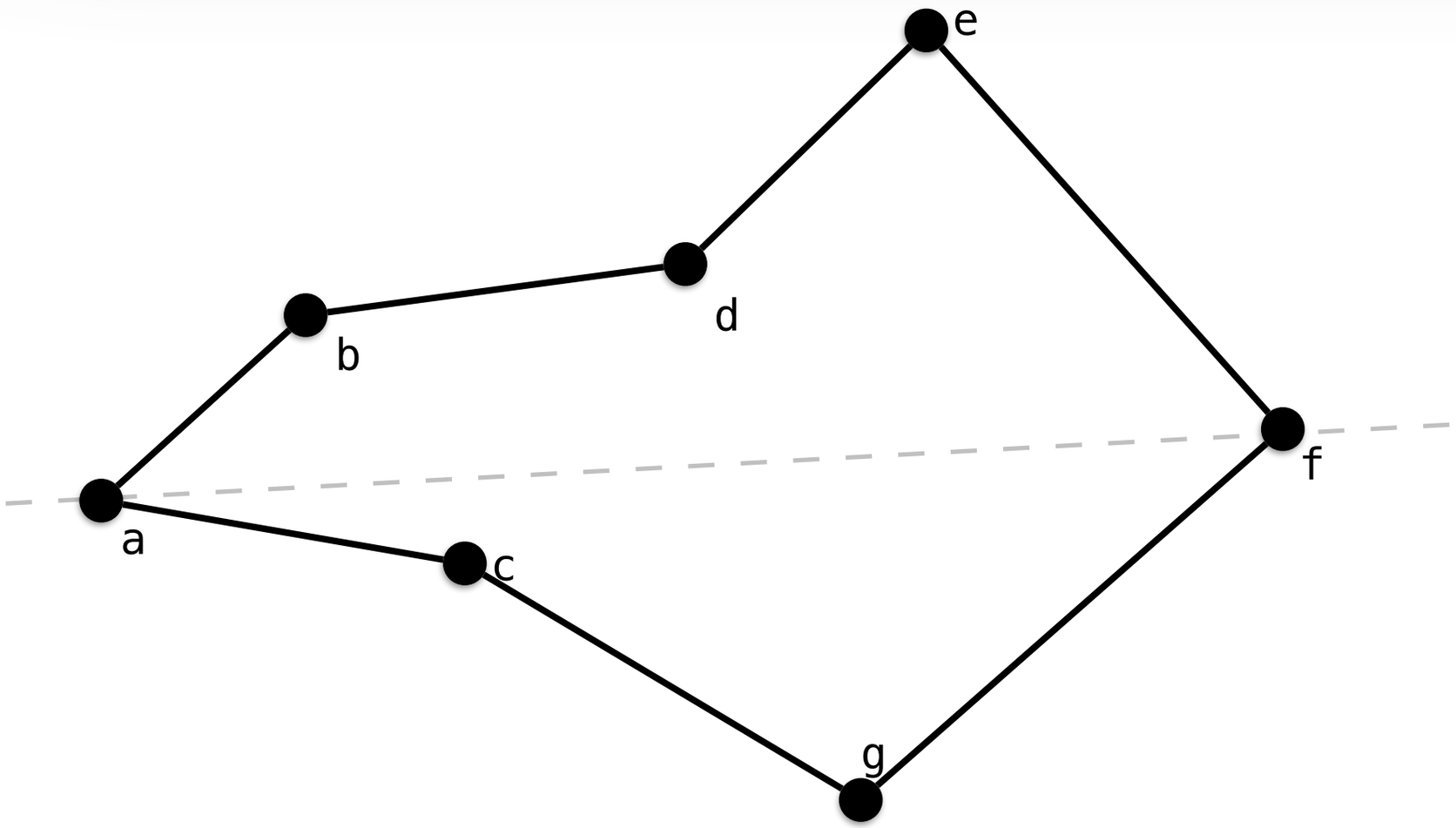


Convex hull



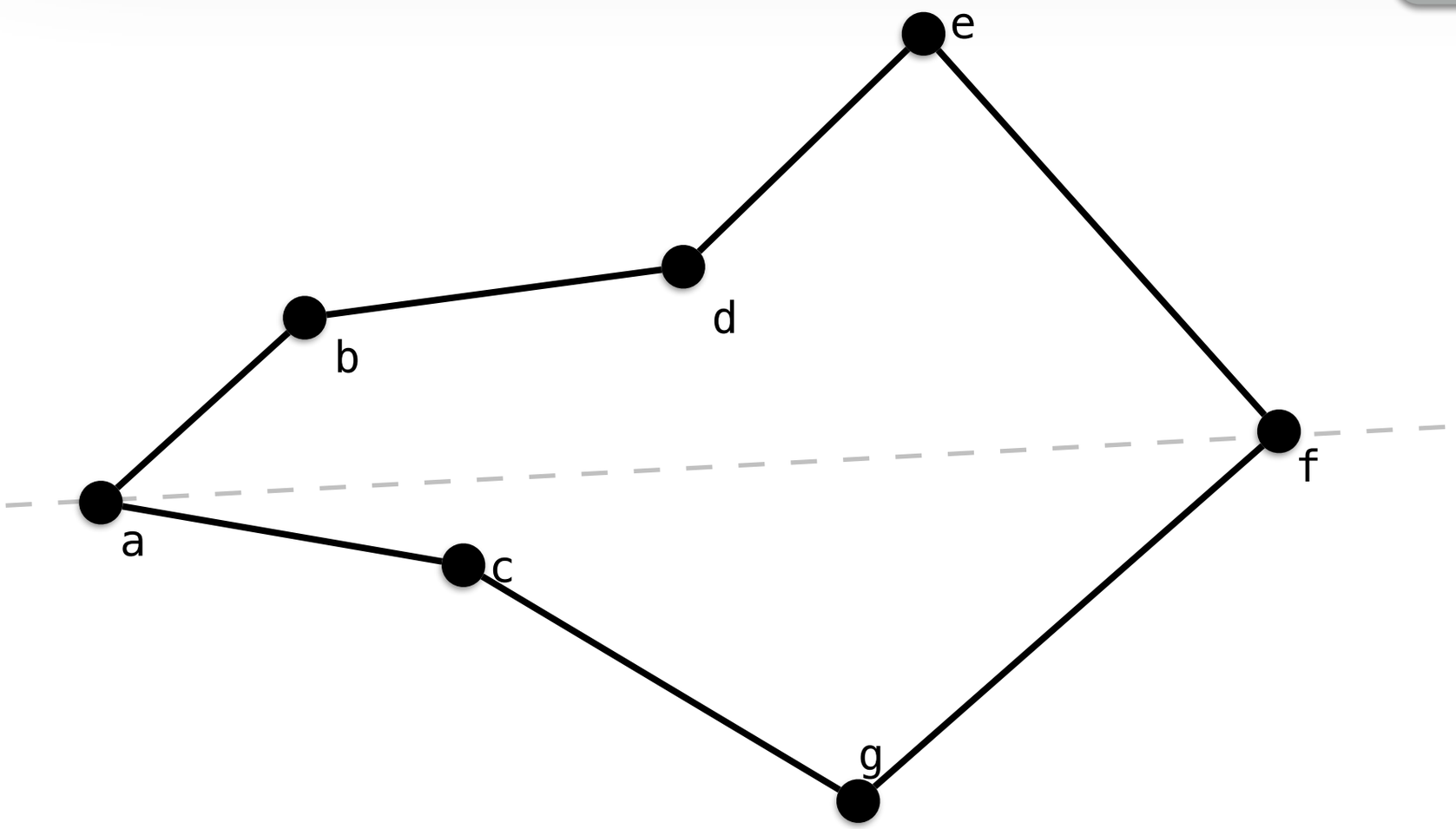


Convex hull



a
c
f
e
d
b
a

Convex hull



convex hull

- a
- c
- f
- e
- d
- b
- a

Convex hull

```
public ConvexHull(Point2D[] points) {
  Arrays.sort(points, new PointComparator());
  //Minimum y point, or if tie, leftmost point.
  hull.push(points[0]);

  Arrays.sort(points, new PointComparator(points[0]));
  hull.push(points[1]);
  hull.push(points[2]);

  for(int i = 3; i < points.length; ++i) {
    Point2D top = hull.peek();
    Point2D nextToTop = hull.get(hull.size() - 2);
    while(polarAngle(points[i], nextToTop, top) >= 0 &&
          hull.size() >= 3) {
      hull.pop();
      top = hull.peek();
      nextToTop = hull.get(hull.size() - 2);
    }
    hull.push(points[i]);
  }
}
```

Minimal enveloping rectangle

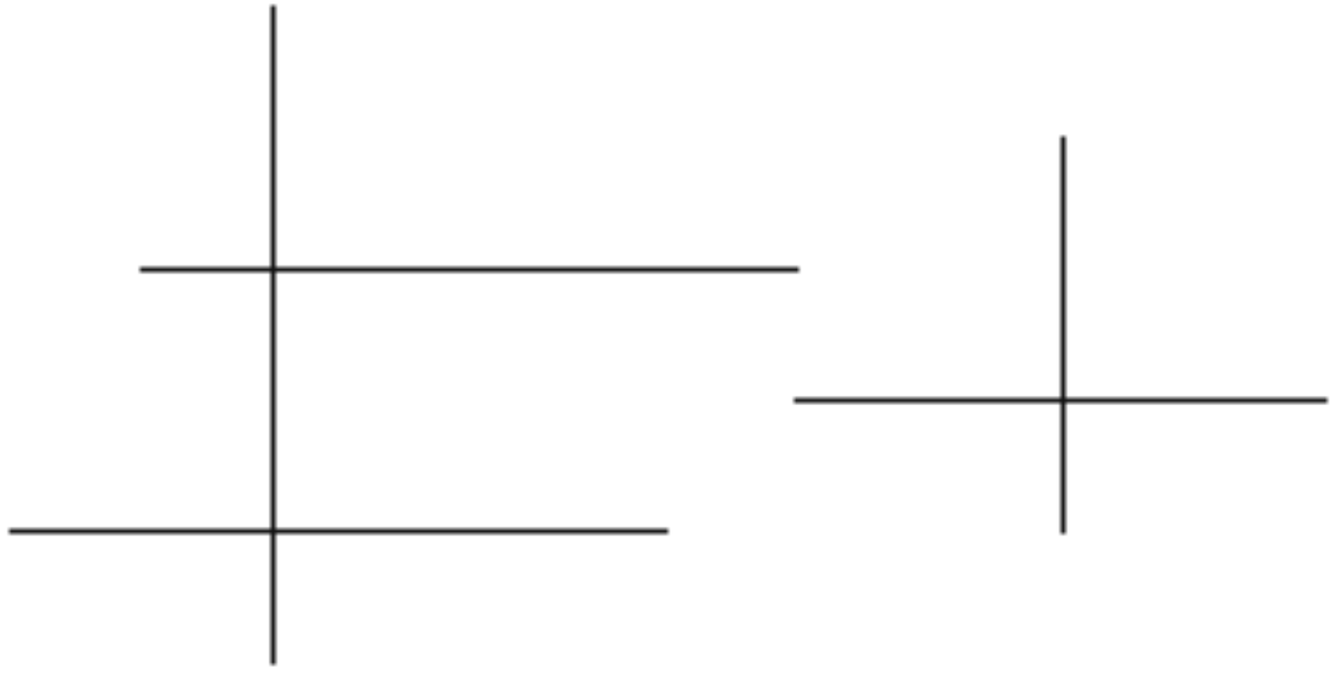
- Get the convex hull
- The solution **shares** an edge with the convex hull
- Take the end vertices and project them to the line in the rectangle
- Rotate the shape to calculate new points using a sliding window

Sweep line

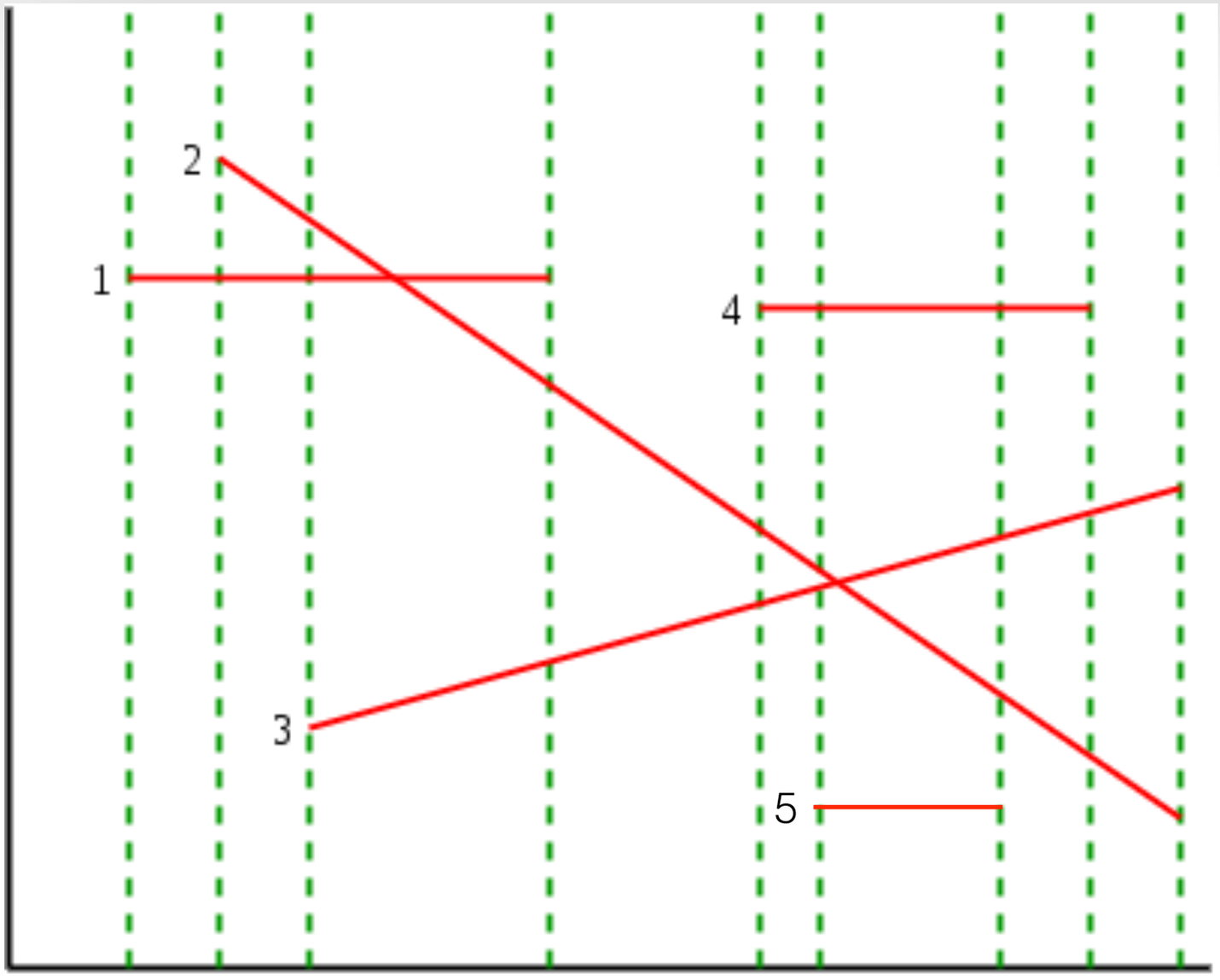
Sweep line

Intersection points

Given n line segments (v or h). Count the total number of intersections



Intersection points



1. start of a line
2. intersections
3. end of a line

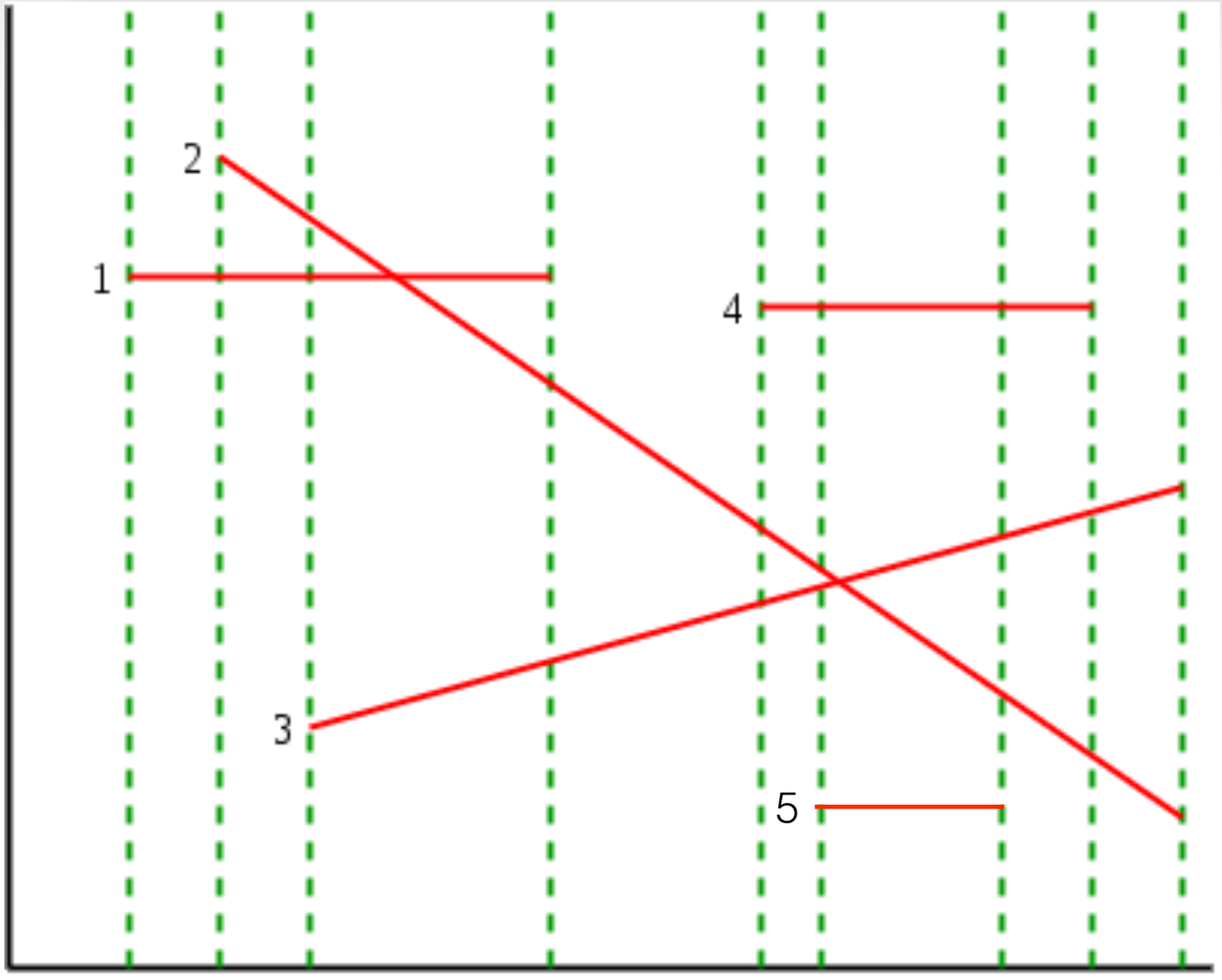
events:
s1,s2,s3,e1,s4,s5,e5,34,
e2,e3

sweep:

intersections:

self balancing BST
(in the y coordinate)

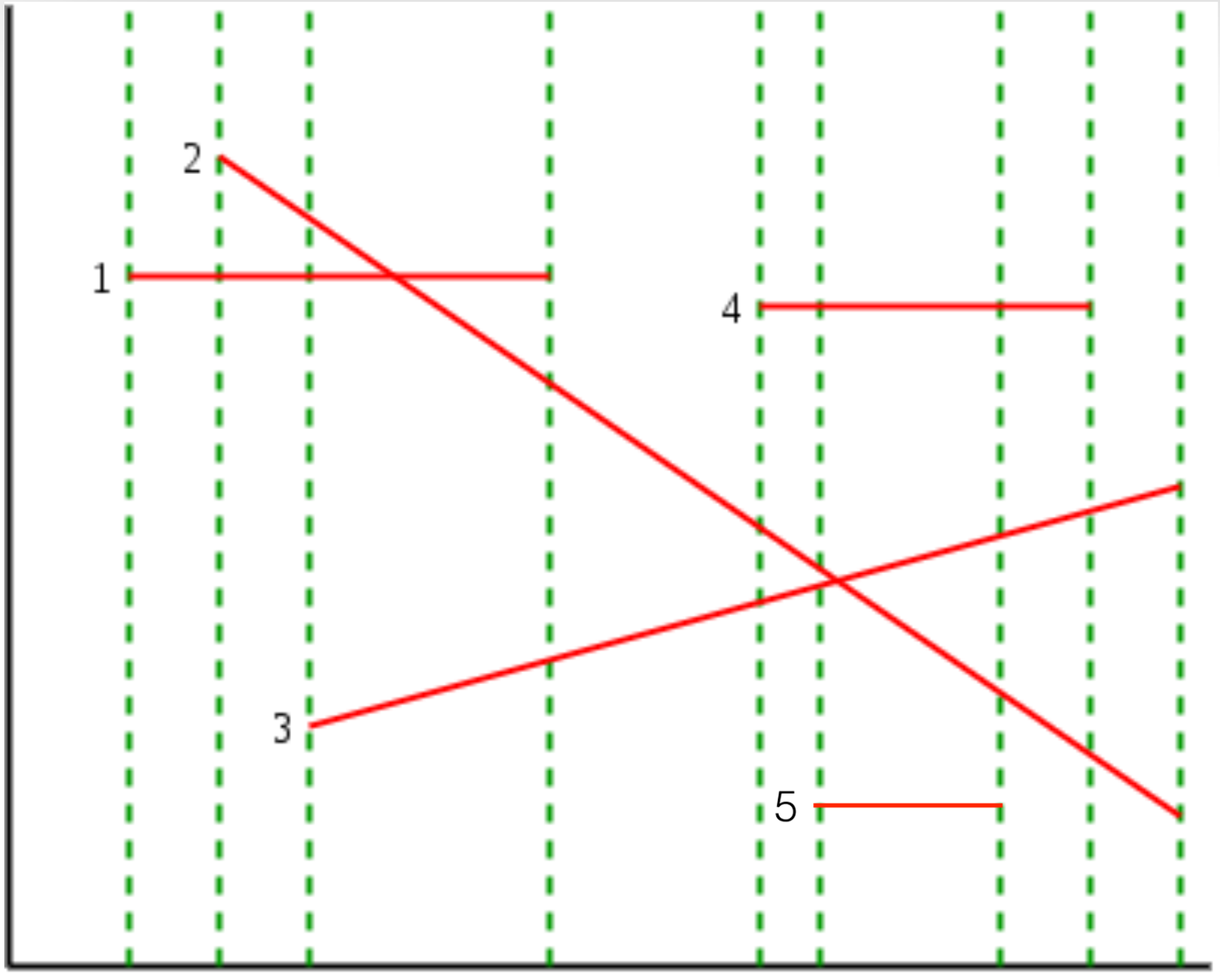
Intersection points



events:
s1,s2,s3,e1,s4,s5,e5,34,
e2,e3

sweep:
1

Intersection points



events:
s1,**s2**,s3,e1,s4,s5,e5,34,
e2,e3

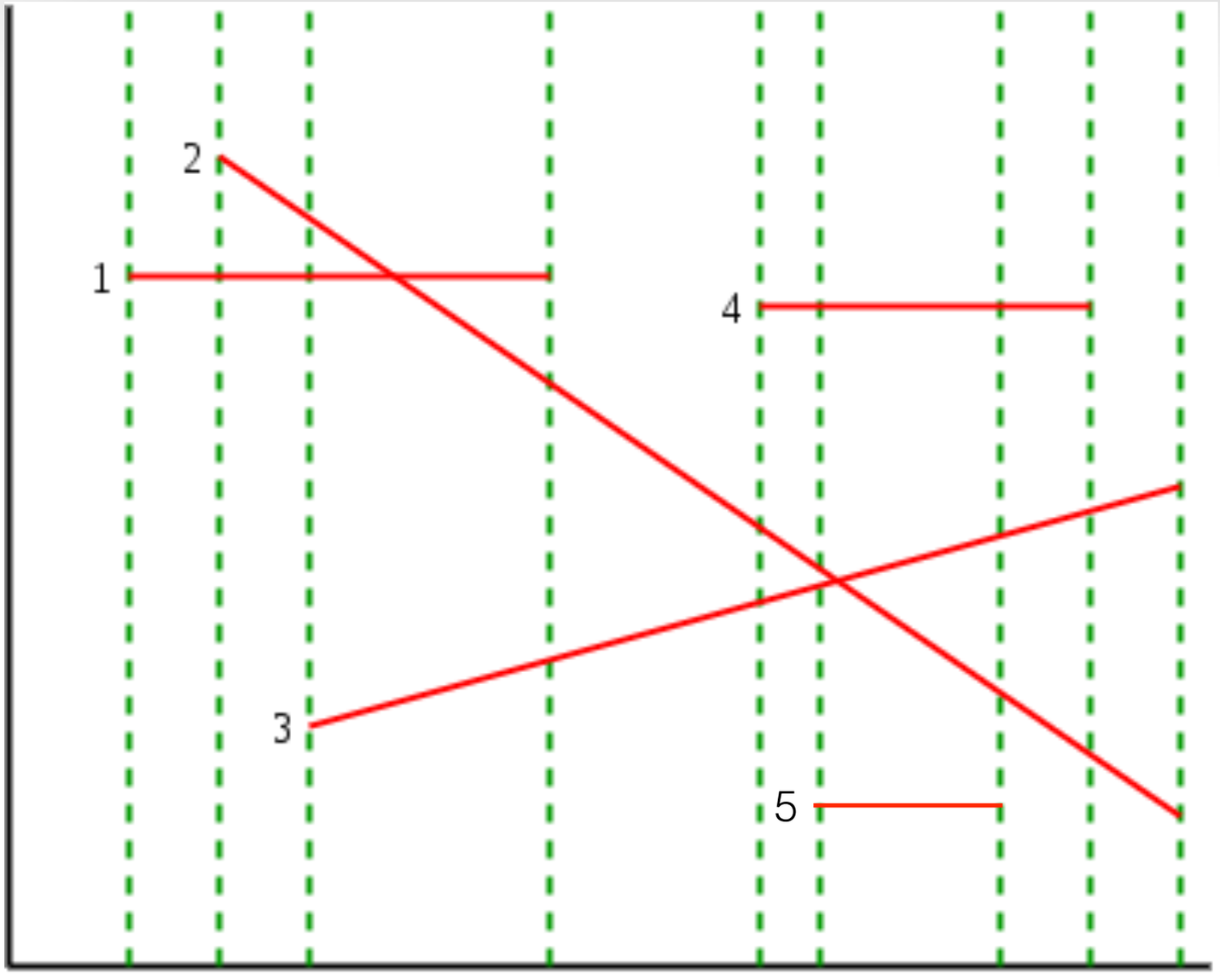
sweep:

1,2

check intersection

intersections: 1

Intersection points



events:
s1,**s2**,**s3**,e1,s4,s5,e5,34,
e2,e3

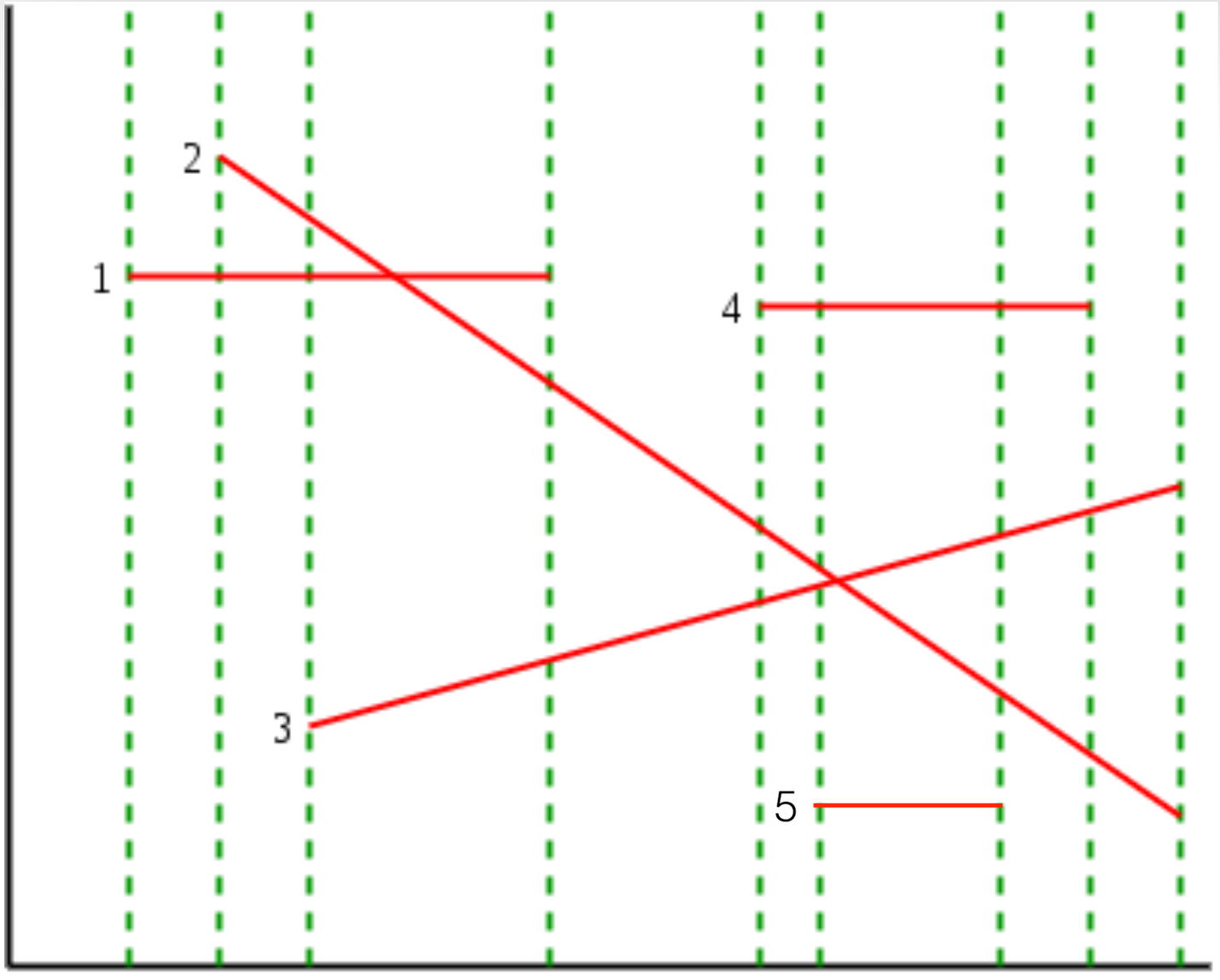
sweep:

2,1,3

check intersection

intersections: 1

Intersection points



events:
s1,**s2**,**s3**,**e1**,s4,s5,e5,34,
e2,e3

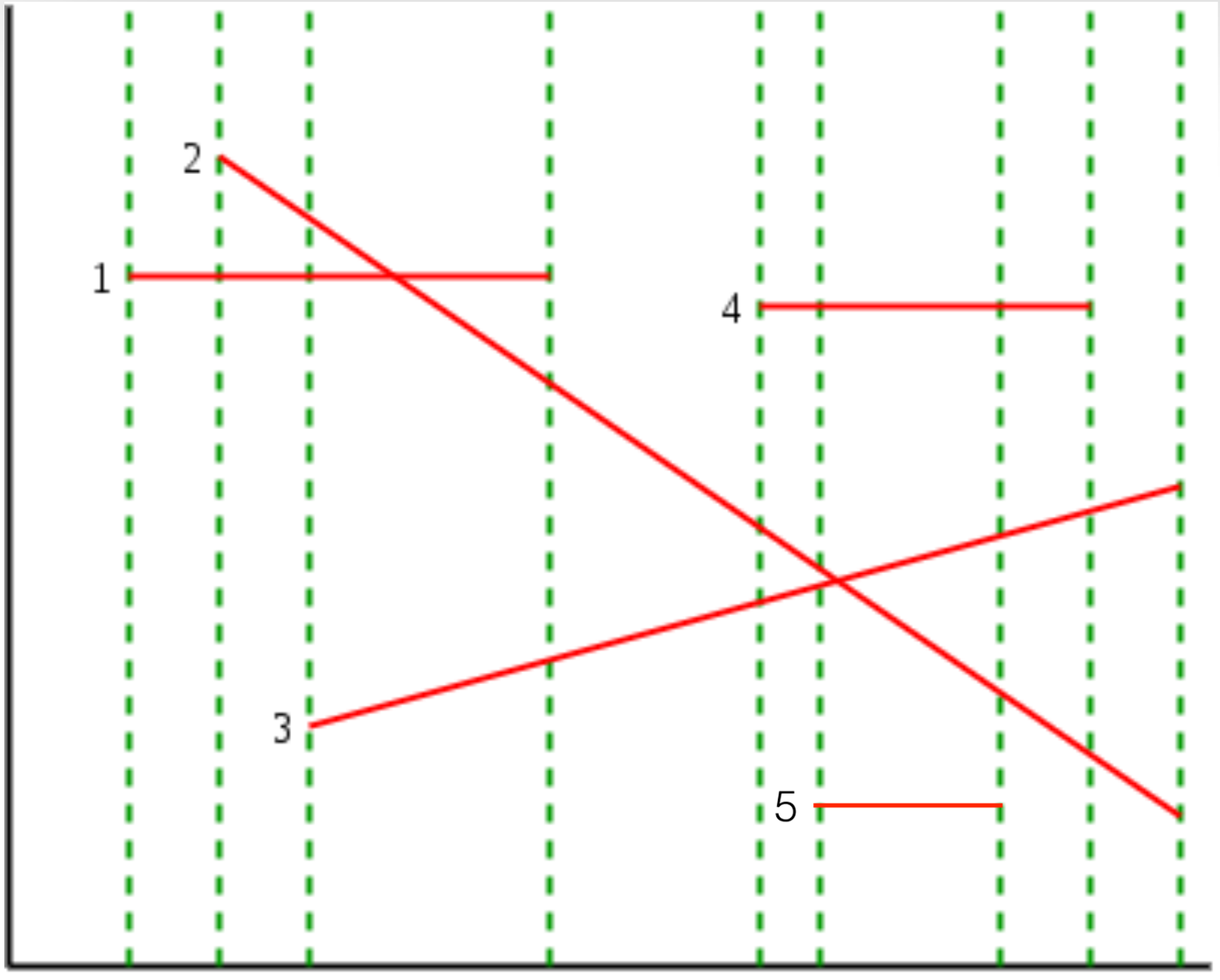
sweep:

2,3

check intersection

intersections: 2

Intersection points



events:
s1,**s2**,**s3**,**e1**,s4,s5,e5,34,
e2,e3

sweep:

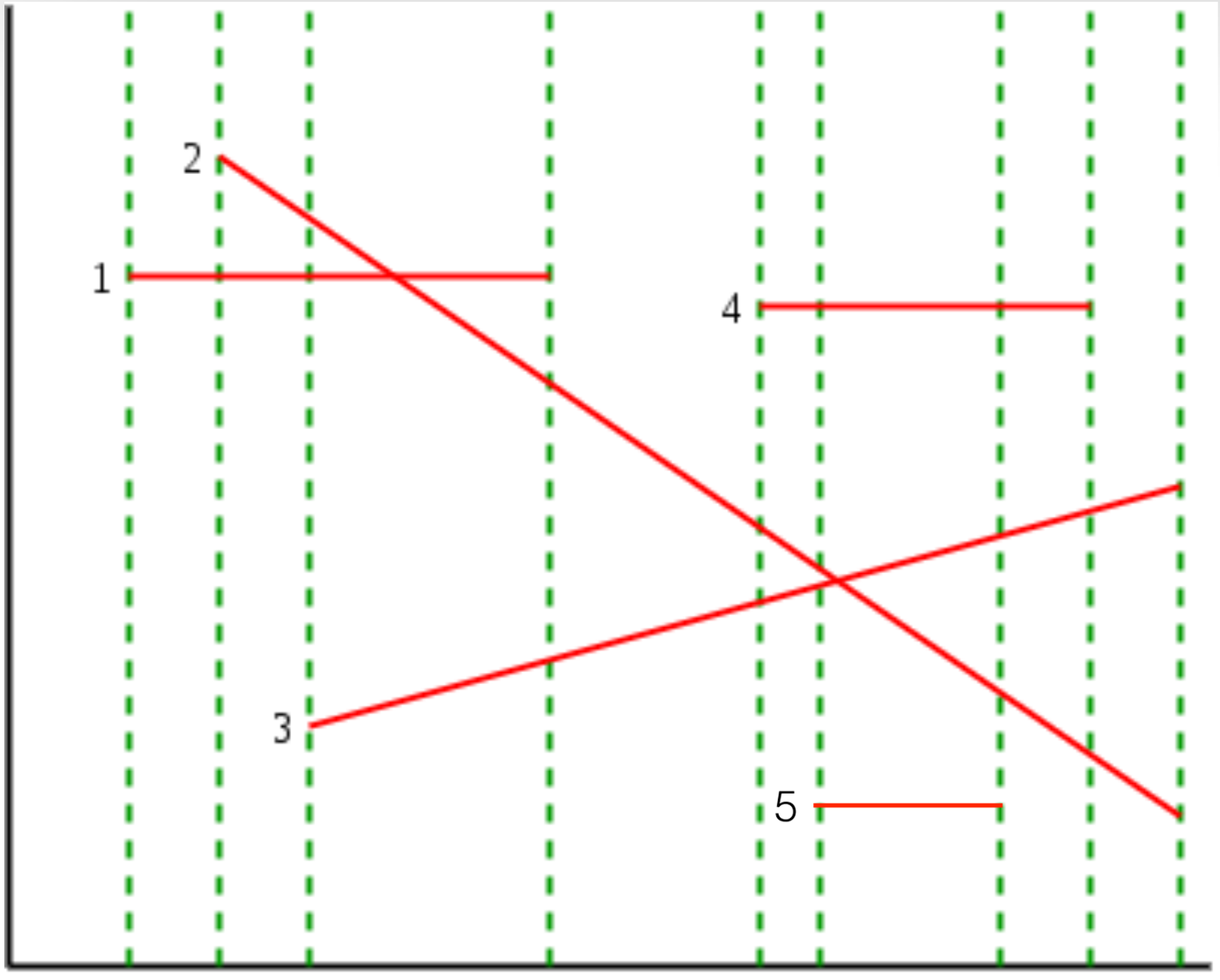
2,3

check intersection

intersections: 2

The algorithm can stop here!

Intersection points

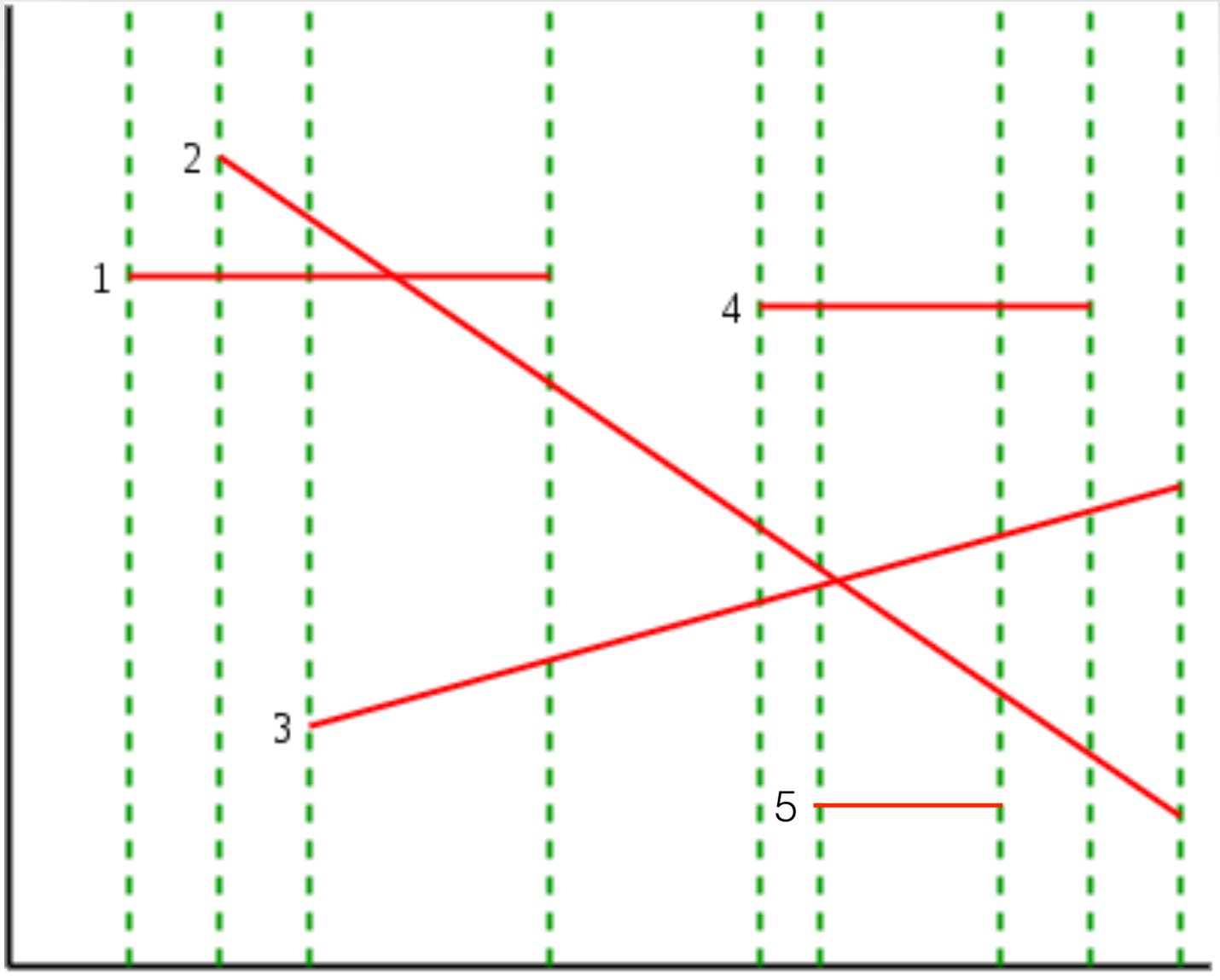


events:
s1,**s2**,**s3**,**e1**,**s4**,s5,e5,34,
e2,e3

sweep:
2,4,3

intersections: 2

Intersection points

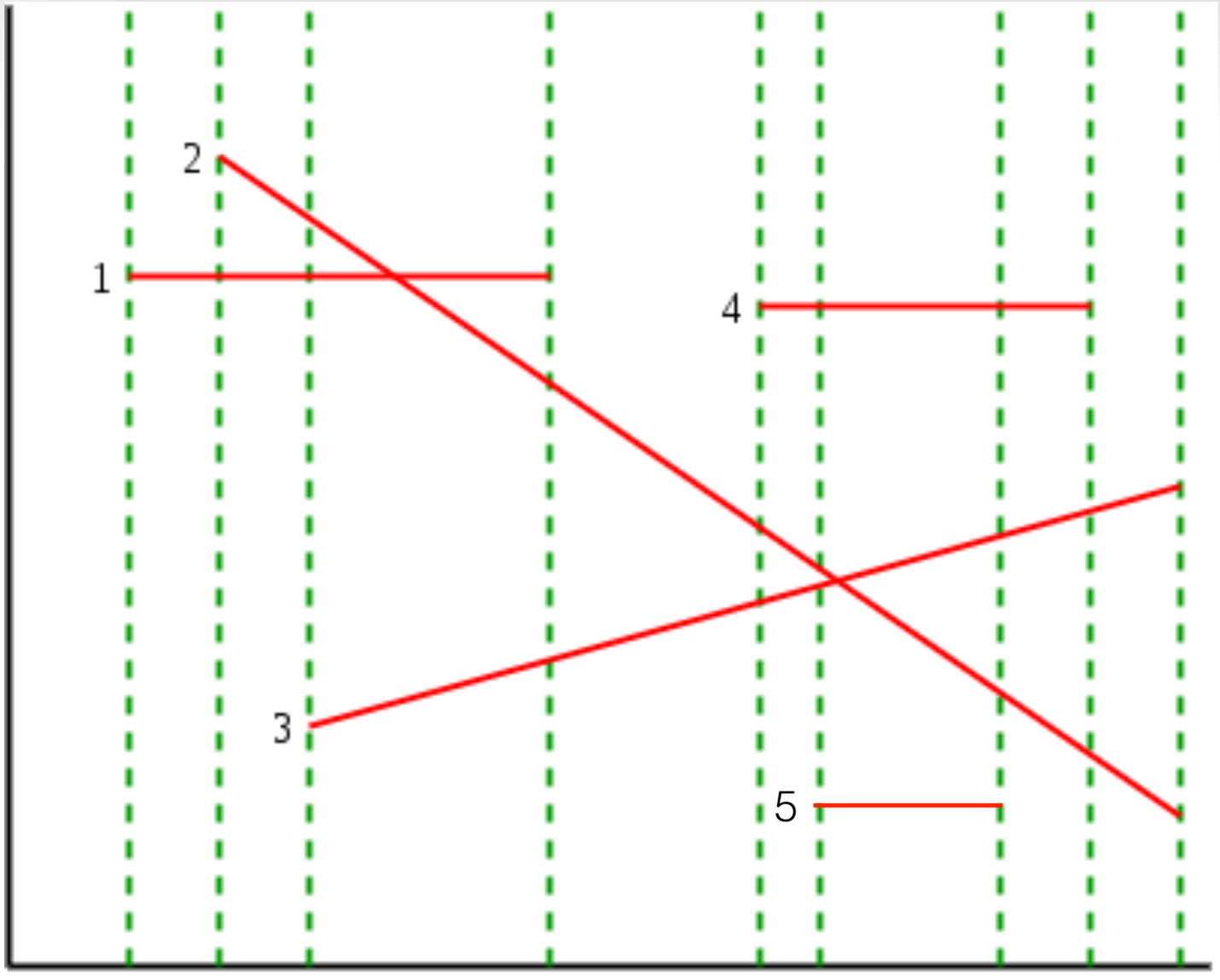


events:
s1,**s2**,**s3**,**e1**,**s4**,**s5**,e5,34,
e2,e3

sweep:
2,4,3,5

intersections: 2

Intersection points

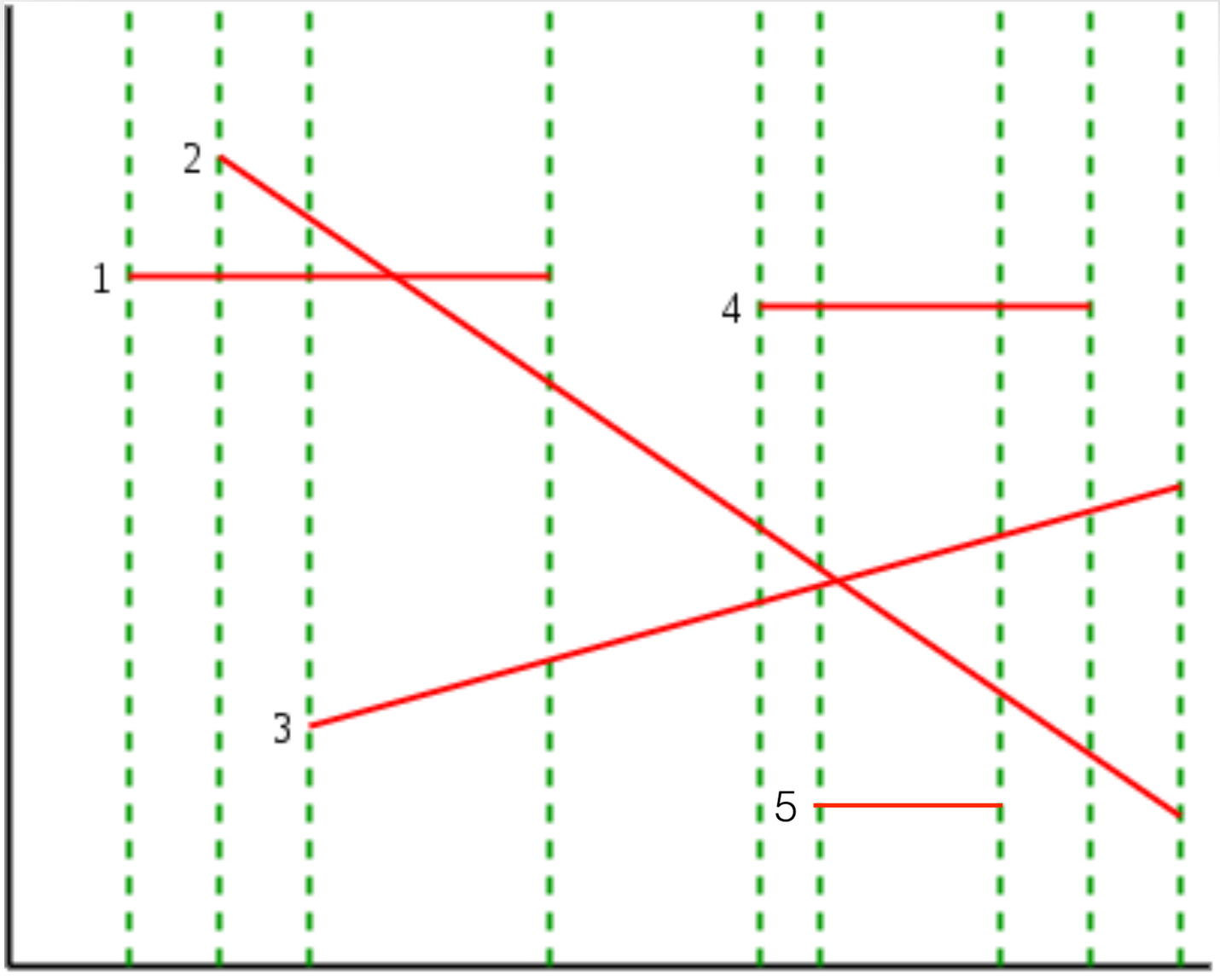


events:
s1,**s2**,**s3**,**e1**,**s4**,**s5**,**e5**,e4,
e2,e3

sweep:
2,4,3

intersections: 2

Intersection points

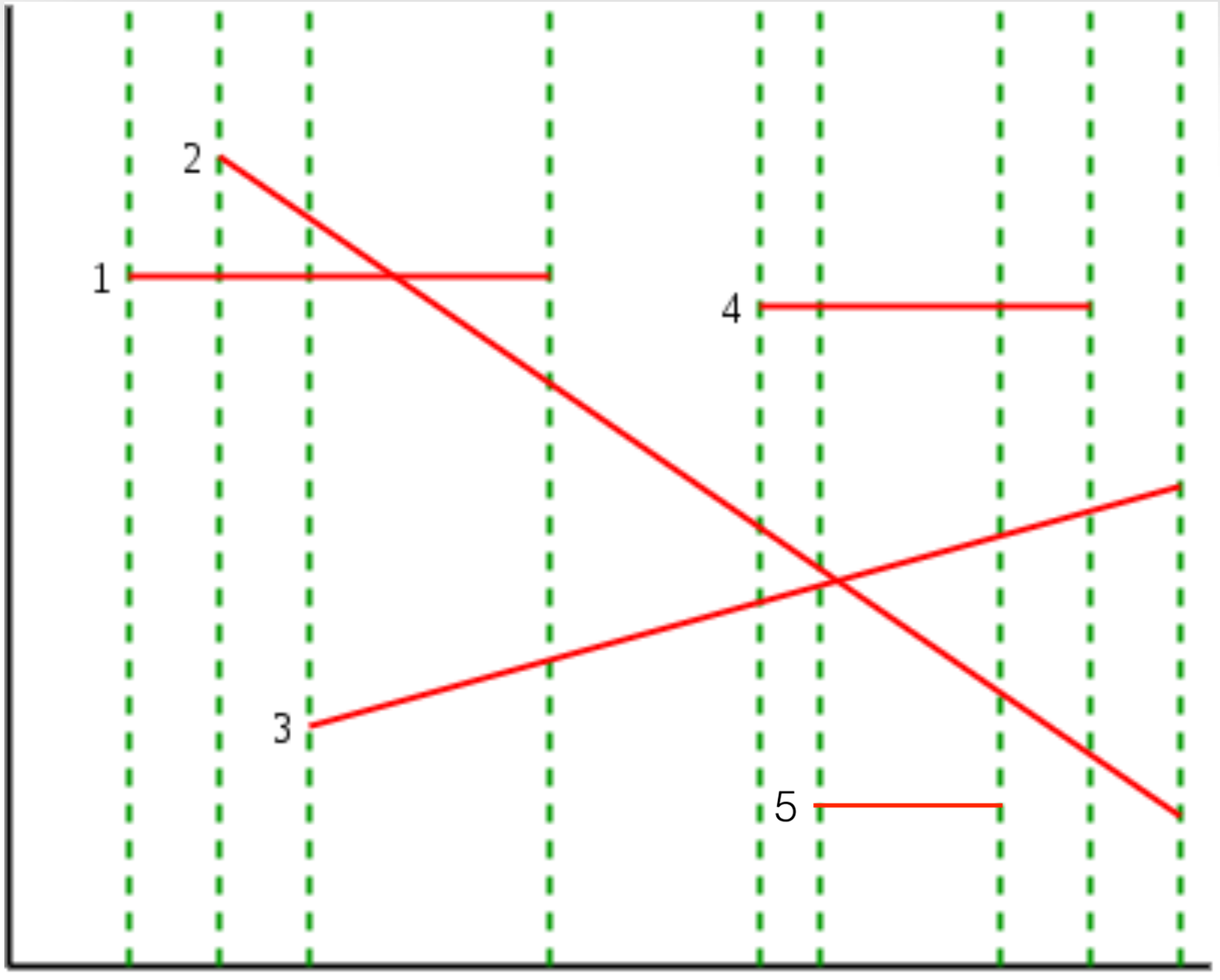


events:
s1,**s2**,**s3**,**e1**,**s4**,**s5**,**e5**,**e4**,
e2,e3

sweep:
2,3

intersections: 2

Intersection points

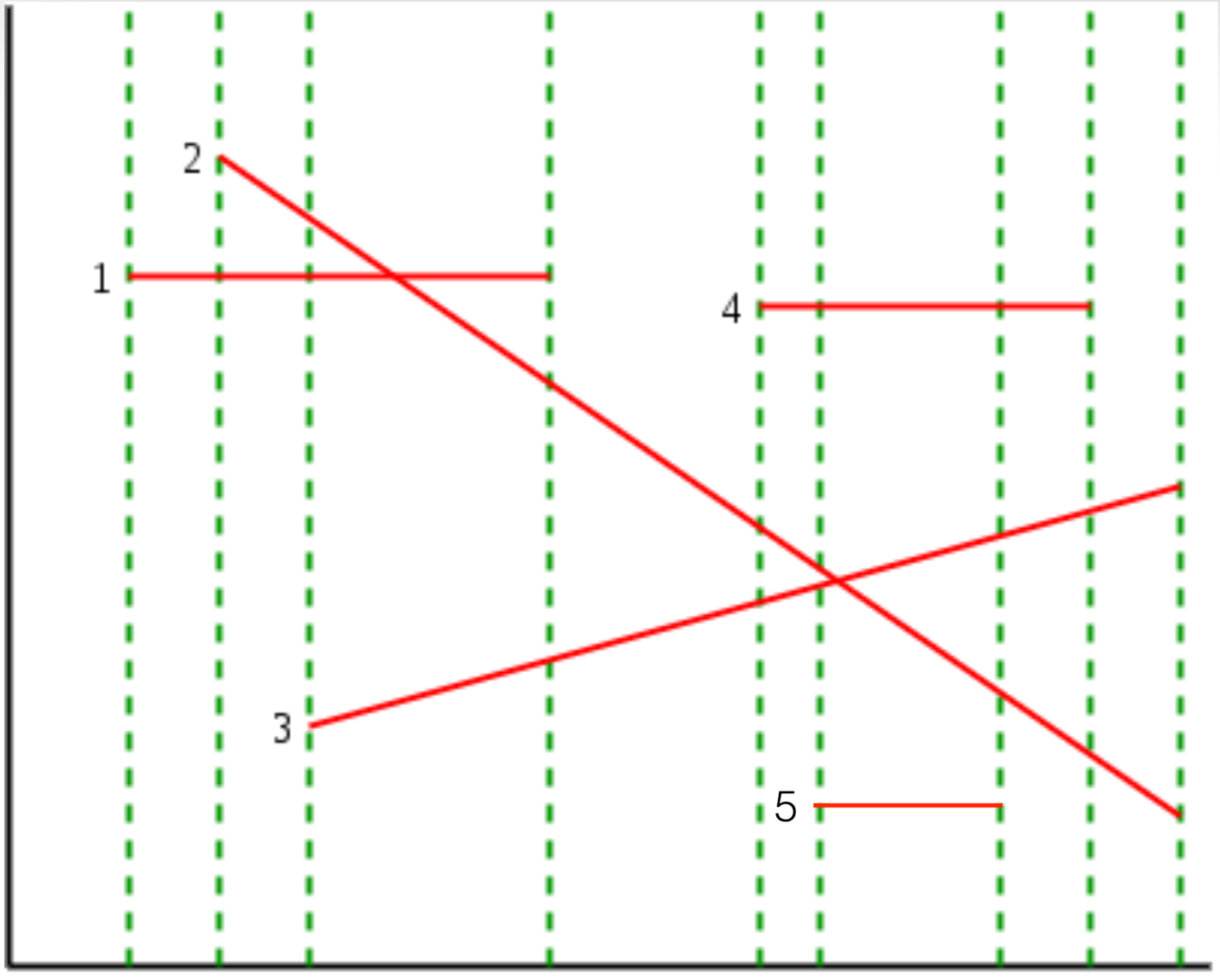


events:
s1,**s2**,**s3**,**e1**,**s4**,**s5**,**e5**,**e4**,
e2,e3

sweep:
3

intersections: 2

Intersection points



events:
s1,**s2**,**s3**,**e1**,**s4**,**s5**,**e5**,**e4**,
e2,**e3**

sweep:

intersections: 2

Intersection points

```
sorted = [] //sort_points
t = BST //y coordinate ordered
for i in range(0 2n-1):
  if sorted[i].left:
    t.insert(sorted[i].line())
    if(intersect(sorted[i].line(), t.pred(sorted[i])
      intersect++
  if(intersect(sorted[i].line(), t.succ(sorted[i])
    intersect++
else:
  if(intersect(t.pred(sorted[i]),
  t.succ(sorted[i])
    intersect++
  t.delete(sorted[i].line())
```

Nearest points

- Order the points left to right
- keep the minim **distance d**
- At each point, check the distance to the points in the range `[x–d, x], [y – d, y + d]`