INF280: Competitive programming

Geometry

Louis Jachiet

Useful notions for competitive programming

Essentially high-school level geometry:

- plane geometry
- vectors
- scalar product
- cross product
- angles (tan/sin/cos)
- projection of a point
- signed area, signed angle

Vectors

Everything is a vector

- a point P can be thought as the vector $\vec{OP}$
- no useful distinction between points and vectors

Vectors

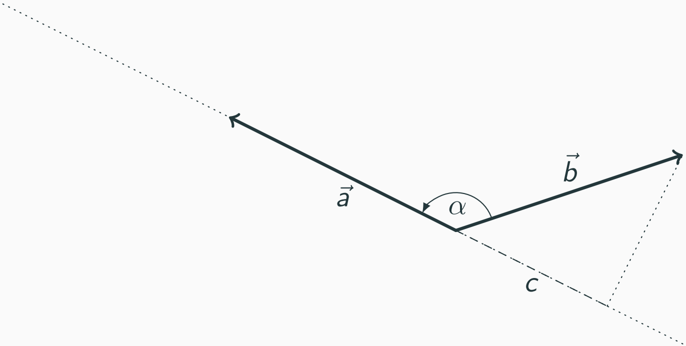
Everything is a vector

- a point P can be thought as the vector $\vec{OP}$
- no useful distinction between points and vectors

For 2D, everything is a complex number!

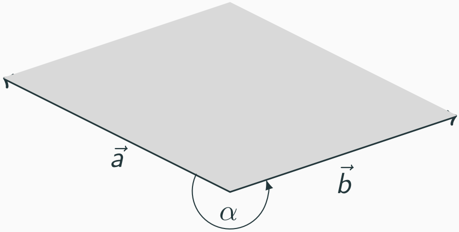
- all operations are easily translated
- no need to reimplement everything!

Dot product

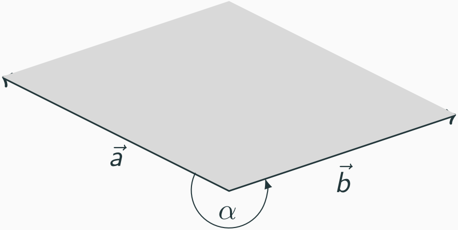


$$\vec{a} \cdot \vec{b} = a \times b \times \cos(\alpha) = a \times c$$

Cross product

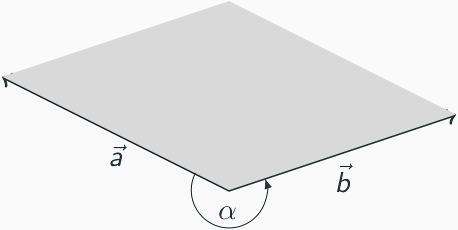


Cross product



$$\vec{a} \times \vec{b} = a \times b \times \sin(\alpha) = \text{signed area}$$

Cross product

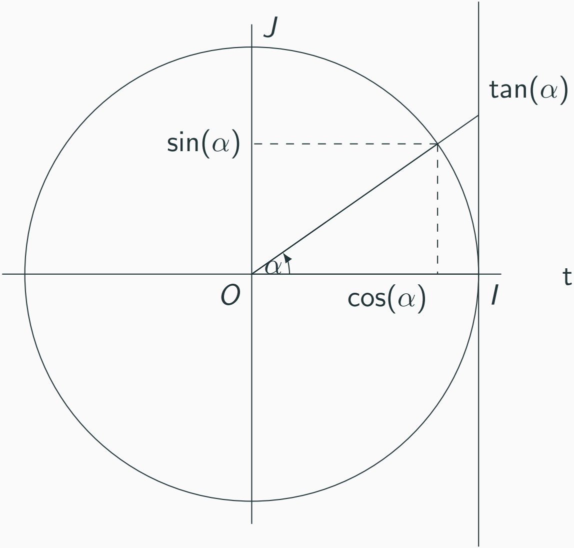


$$\vec{a} \times \vec{b} = a \times b \times \sin(\alpha) = \text{signed area}$$

Using the sign of $\vec{a} \times \vec{b}$:

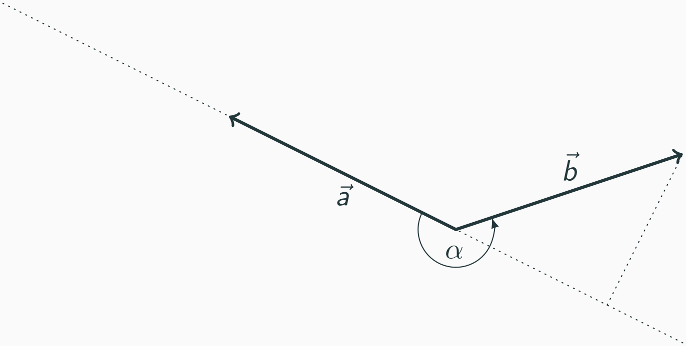
- $\vec{a} \times \vec{b} > 0$ when $\vec{b}$ is counter-clockwise to $\vec{a}$
- $\vec{a} \times \vec{b} < 0$ when $\vec{b}$ is clockwise to $\vec{a}$
- $\vec{a} \times \vec{b} = 0$ when $\vec{b}$ is co-linear to $\vec{a}$

Trigonometry



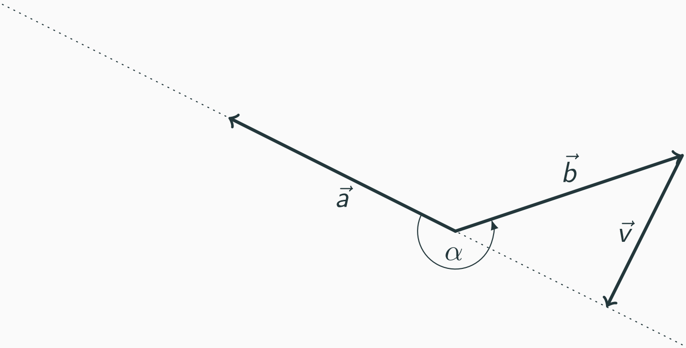
$$\tan = \frac{\sin}{\cos}$$

Projection on a line



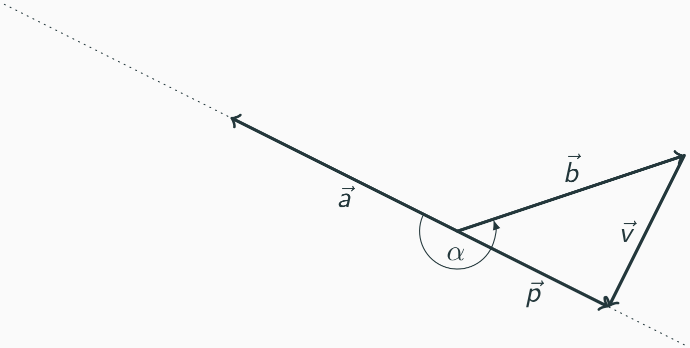
$$proj_{\vec{a}}(\vec{b}) = \vec{p} = \vec{a} \times \frac{\vec{a} \cdot \vec{b}}{\vec{a} \cdot \vec{a}} = \frac{\vec{a}}{|\vec{a}|} |\vec{b}| \cos \alpha$$

Projection on a line



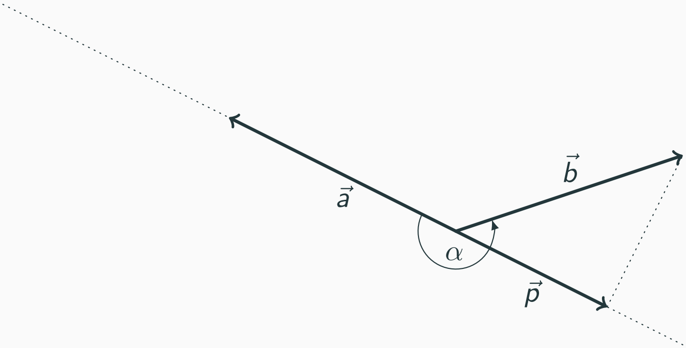
$$proj_{\vec{a}}(\vec{b}) = \vec{p} = \vec{a} \times \frac{\vec{a} \cdot \vec{b}}{\vec{a} \cdot \vec{a}} = \frac{\vec{a}}{|\vec{a}|} |\vec{b}| \cos \alpha$$

Projection on a line



$$proj_{\vec{a}}(\vec{b}) = \vec{p} = \vec{a} \times \frac{\vec{a} \cdot \vec{b}}{\vec{a} \cdot \vec{a}} = \frac{\vec{a}}{|\vec{a}|}|\vec{b}| \cos \alpha$$

Projection on a line



$$proj_{\vec{a}}(\vec{b}) = \vec{p} = \vec{a} \times \frac{\vec{a} \cdot \vec{b}}{\vec{a} \cdot \vec{a}} = \frac{\vec{a}}{|\vec{a}|} |\vec{b}| \cos \alpha$$

Geometry for competitive programming

Useful functions

All the classical ones:

- `sin`, `cos`, `tan`
- `fmod`
- `fabs`
- `ceil`, `floor`
- `sqrt`, `pow`, `log`
- `atan`, `atan2` ($\texttt{atan2}(x, y) = \tan^{-1}(y/x)$)

Vectors

Using `complex<T>`

- in theory only defined for T a float type (float, double, etc.)
- but works with integral types for basic stuff

Useful operations

- addition, subtraction and multiplication by a scalar are defined
- scalar product $\vec{a} \cdot \vec{b} = \text{real}(\bar{a} \times b)$
- cross product $\vec{a} \times \vec{b} = \text{imag}(\bar{a} \times b)$

Complex in practice

```
using namespace std ;

int main() {
  complex<double> z(1,1), i(0,1);
  arg(z); // pi/4
  real(i); // 0
  imag(i); // 1
  norm(z); // 1^2+1^2 = 2, squared norm
  abs(z);  // (1^2+1^2)^{1/2} = sqrt(2)
  conj(z); // conjugate, i.e. z = complex<double>(1,-1)
  z*i; // product of two complex numbers
  z+i; // sum
}
```

Classical algorithms for computational geometry

Classical algorithms for computational geometry

Areas

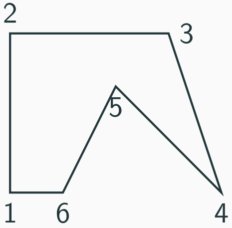
Area of a triangle

```
double signed_triangle_area(pt a, pt b, pt c) {
  return imag((conj(b-a))*(c-a))/2 ; // cross product/2
}
double triangle_area(pt a, pt b, pt c) {
  return fabs(signed_triangle_area(a,b,c));
}
```

Area of a polygon

First idea

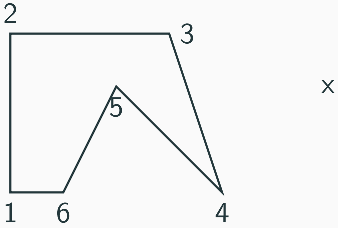
Take any point, sum the triangle area from that point.



Area of a polygon

First idea

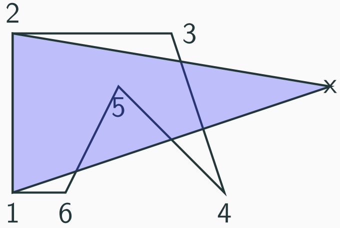
Take any point, sum the triangle area from that point.



Area of a polygon

First idea

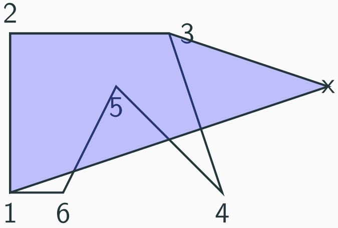
Take any point, sum the triangle area from that point.



Area of a polygon

First idea

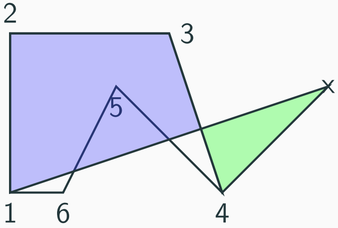
Take any point, sum the triangle area from that point.



Area of a polygon

First idea

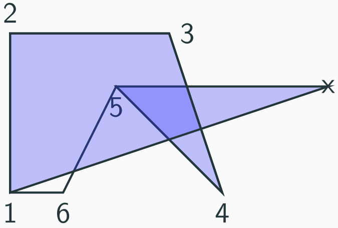
Take any point, sum the triangle area from that point.



Area of a polygon

First idea

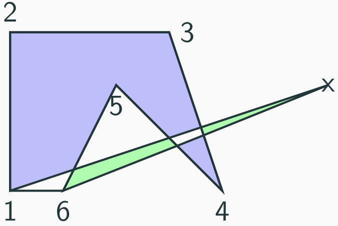
Take any point, sum the triangle area from that point.



Area of a polygon

First idea

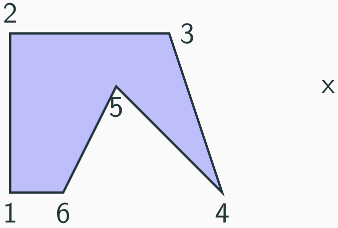
Take any point, sum the triangle area from that point.



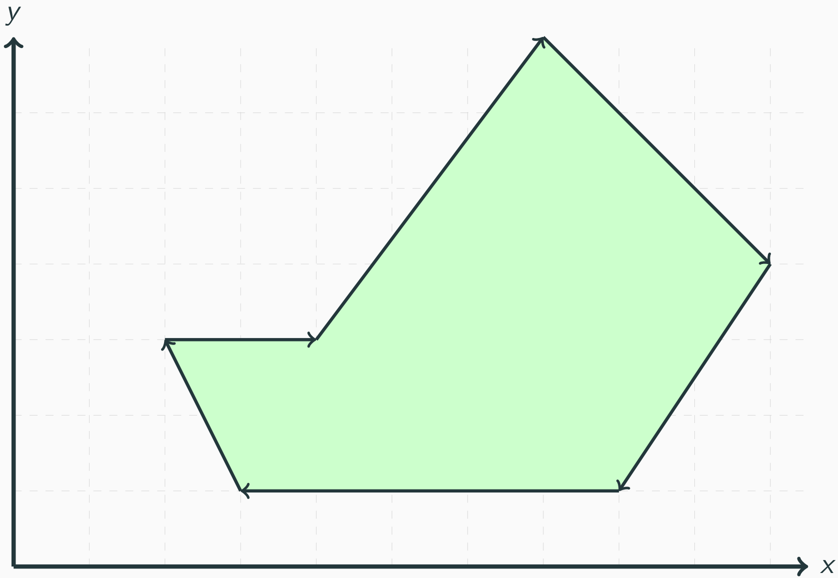
Area of a polygon

First idea

Take any point, sum the triangle area from that point.

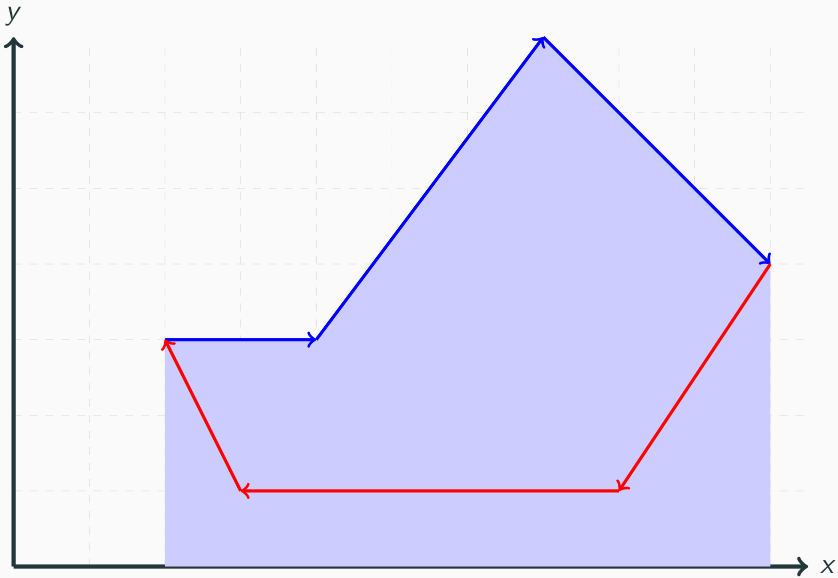


Area of polygon



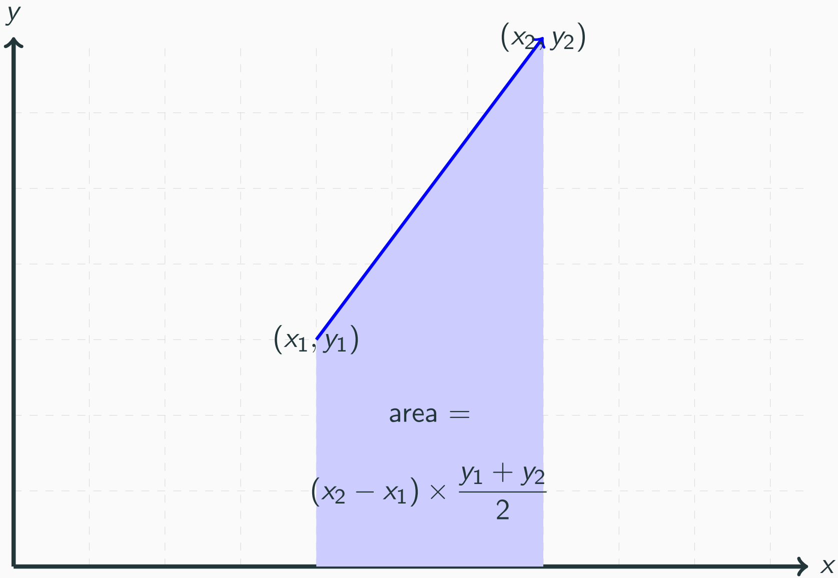
Area of polygon

Area of polygon



Area of polygon

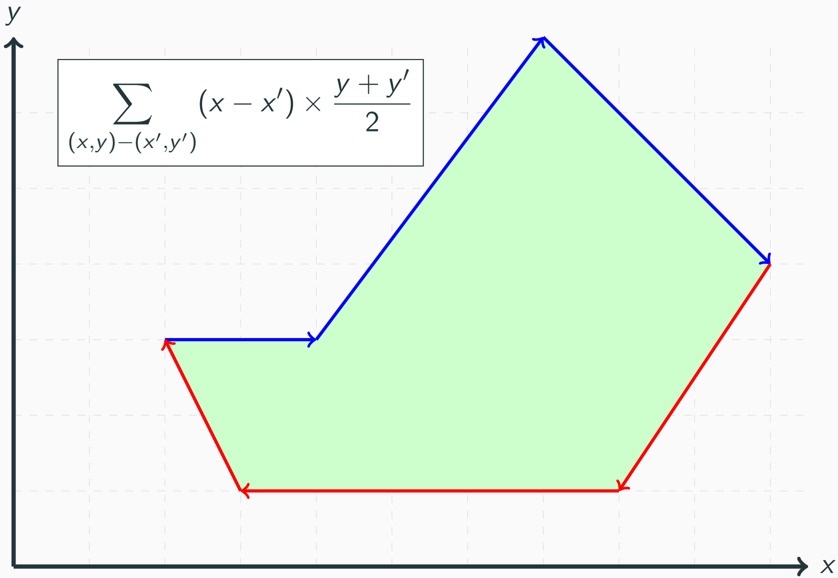
Area of polygon



area =

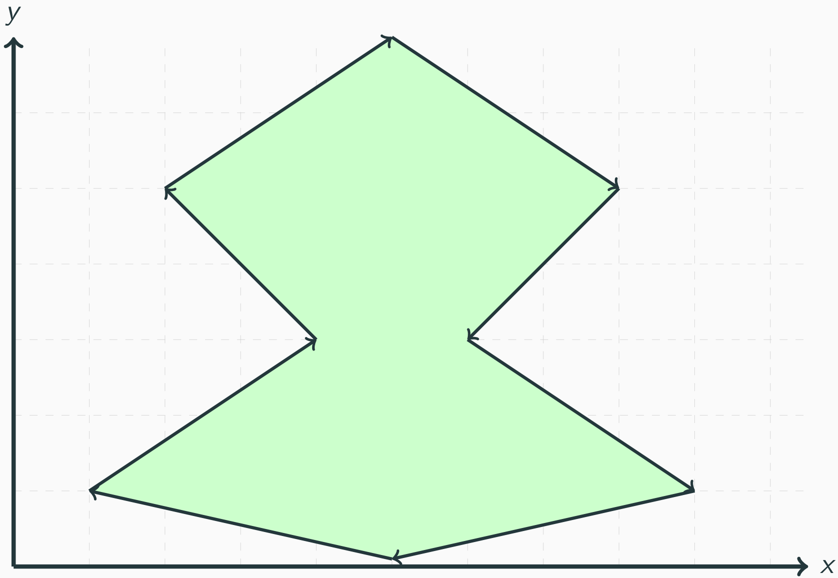
$$(x_2 - x_1) \times \frac{y_1 + y_2}{2}$$

Area of polygon

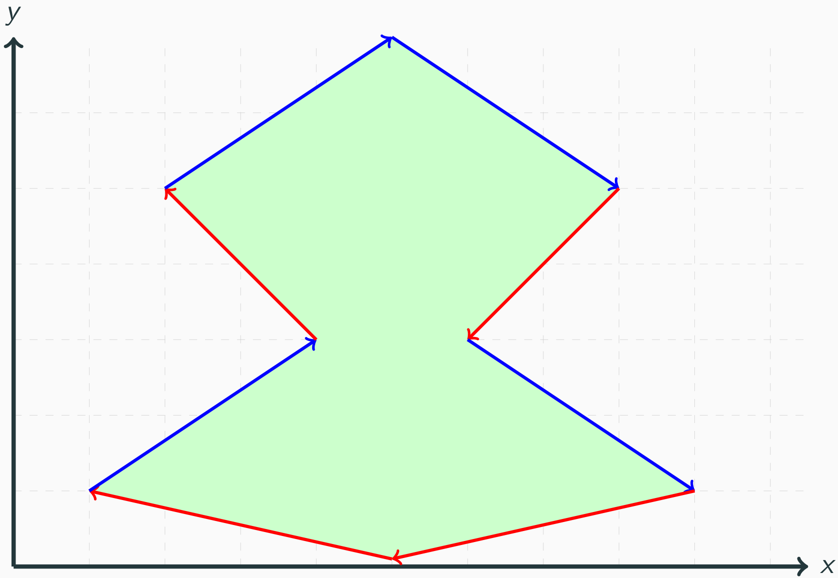


$$\sum_{(x,y)-(x',y')} (x - x') \times \frac{y + y'}{2}$$

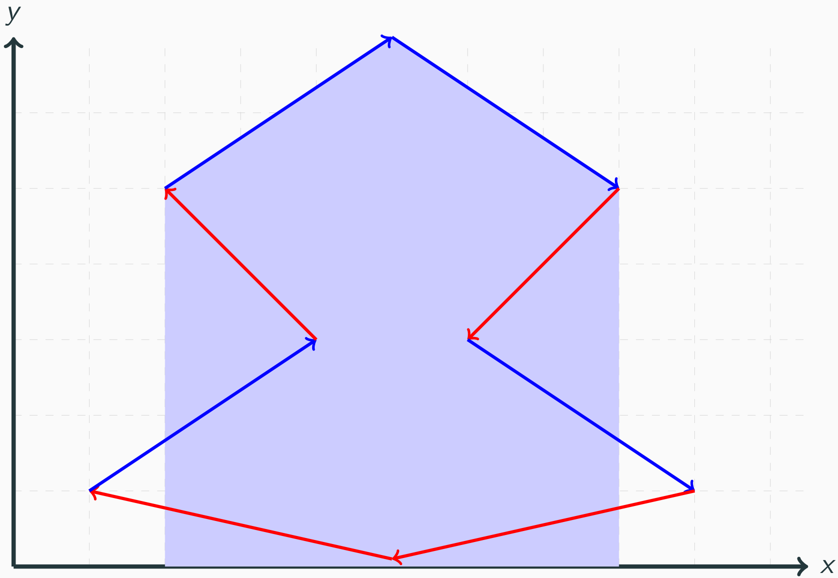
Area of polygons



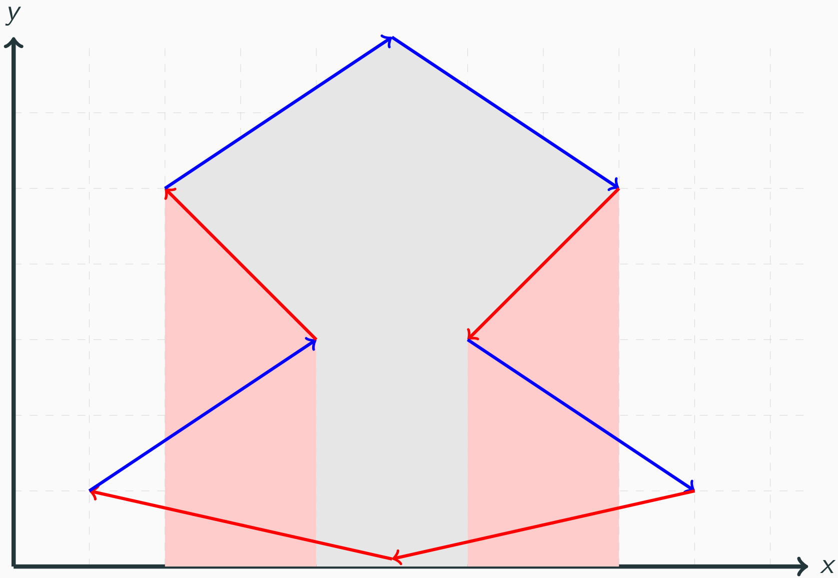
Area of polygons



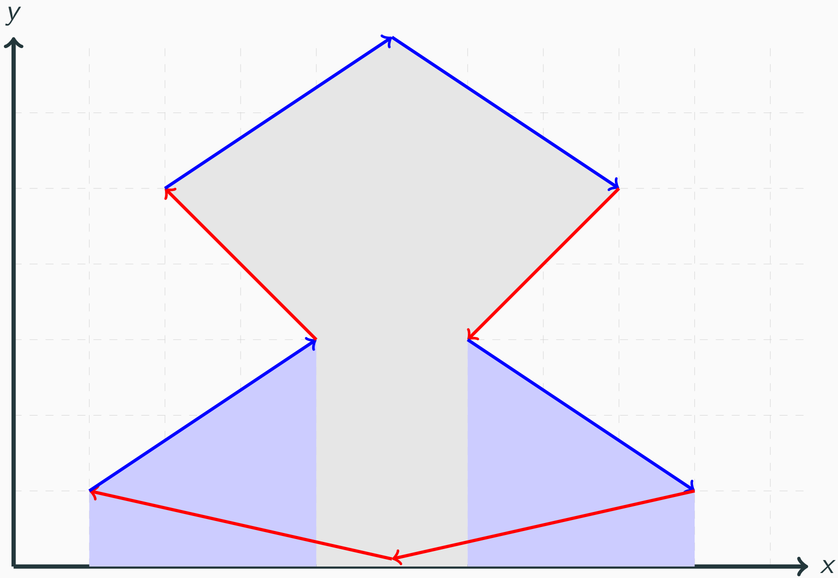
Area of polygons



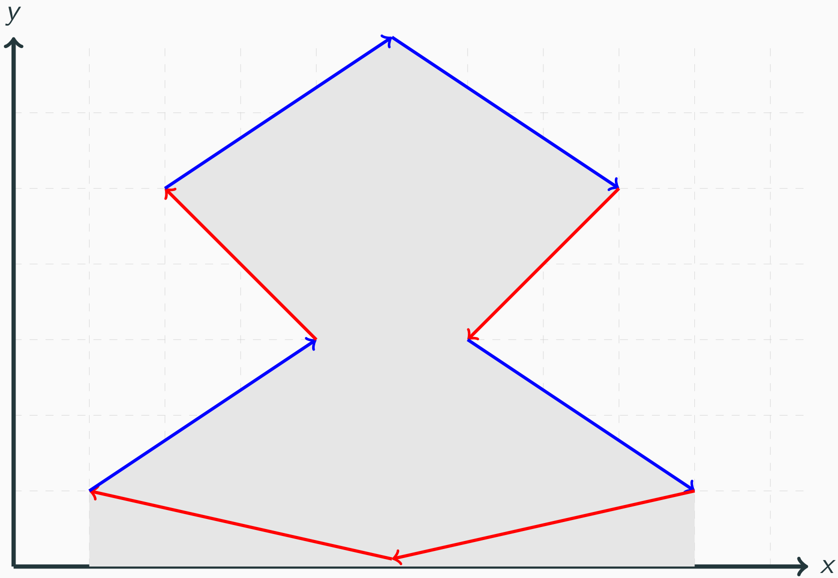
Area of polygons



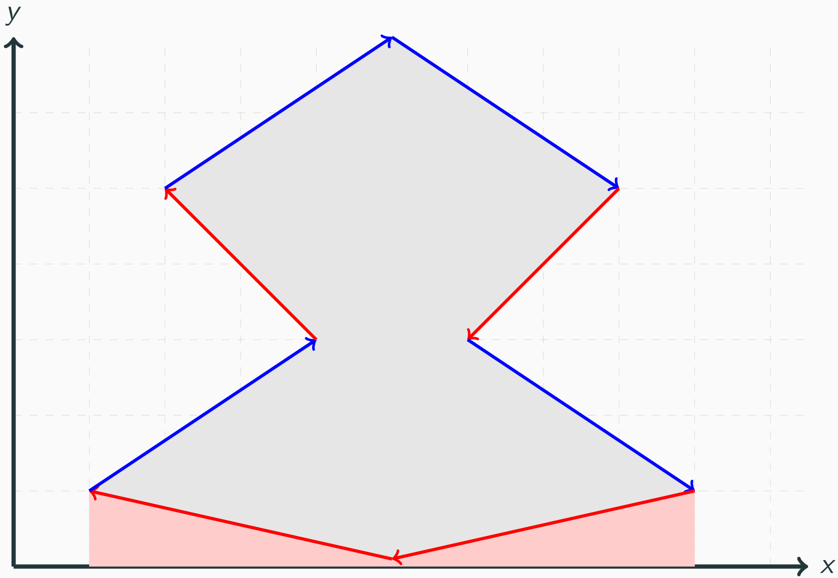
Area of polygons



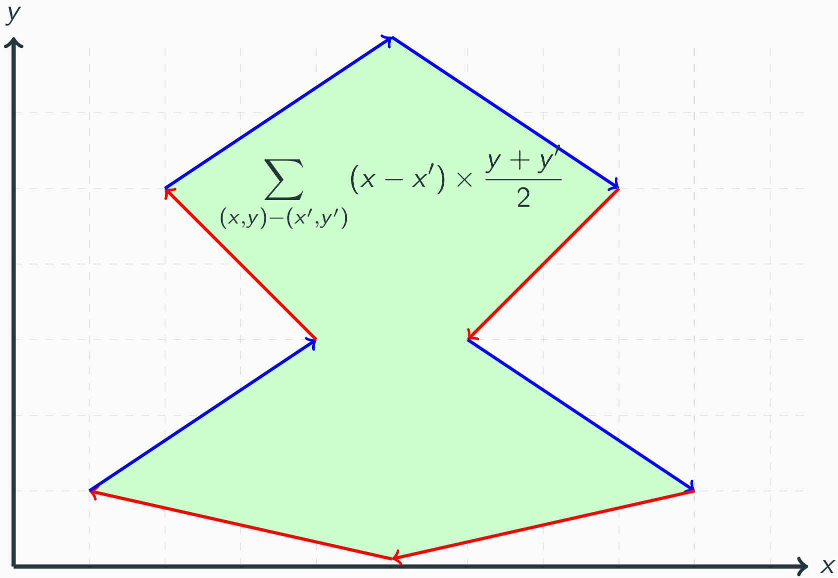
Area of polygons



Area of polygons

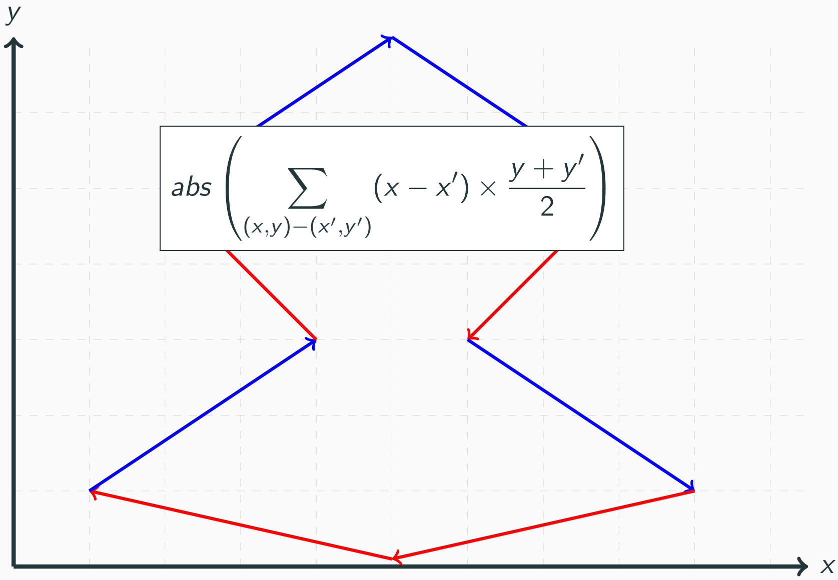


Area of polygons



$$\sum_{(x,y)-(x',y')} (x - x') \times \frac{y + y'}{2}$$

Area of polygons



$$abs\left(\sum_{(x,y)-(x',y')} (x-x') \times \frac{y+y'}{2}\right)$$

Classical algorithms for computational geometry

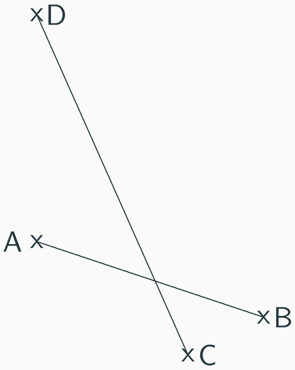
Intersections

Check if two lines intersect

Idea

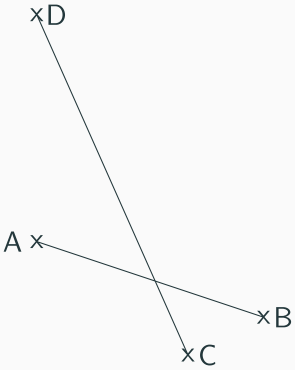
Take two vectors, check whether they are parallel (with cross product), if yes, check whether they are the same line.

Check if two segments intersect



Any idea for an algorithm?

Check if two segments intersect



Any idea for an algorithm?

We need to check that D and C are on both sides of (AB) and that A and B are on both sides of (DC).

Check if two lines intersect

```
int sign_cross(pt A, pt B) {
  const double c = (conj(A) * B).imag() ;
  if(c < 0) return -1;
  if(c > 0) return 1 ;
  return 0;
}

bool checkIntersection(pt A, pt B, pt C, pt D) {
  return sign_cross(C-A,B-A) != sign_cross(D-A,B-A) &&
         sign_cross(A-C,D-C) != sign_cross(B-C,D-C) ;
}
```

Intersection of other objects

Several ideas to try:

- Can you summarize them as lines (e.g. polygon intersection)?
- Can you use equations (e.g. circles)?
- Do you need the intersections points or just whether there is an intersection?
- Gradient descent for very complex shapes?
- Use bounding boxes to speed up the computation?
- "Rasterize" the problem?

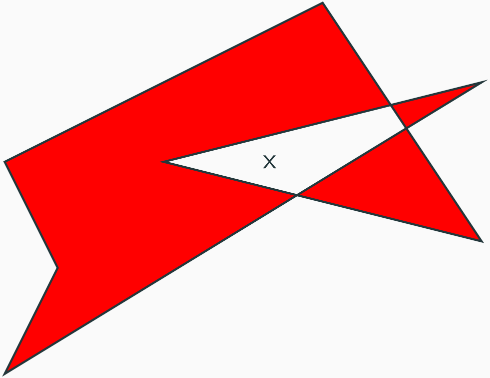
Classical algorithms for computational geometry

Points in polygons

Test if point p is in polygon P 1/2

A simple idea… but hard to put into place

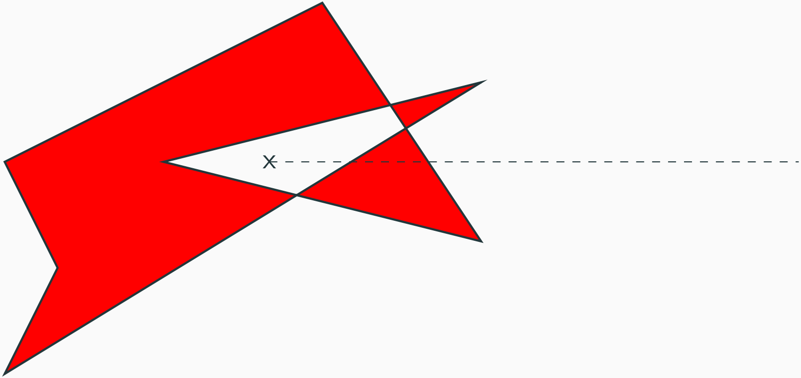
Imagine p shoots a laser in any direction, count the parity of the number of intersections



Test if point p is in polygon P 1/2

A simple idea… but hard to put into place

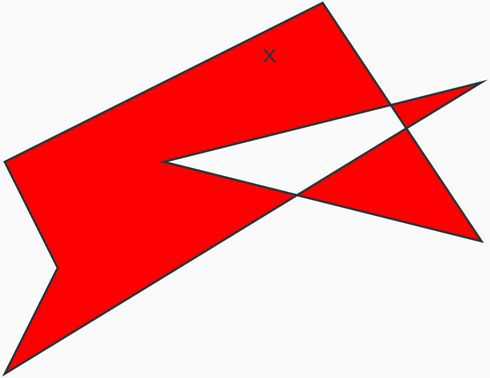
Imagine p shoots a laser in any direction, count the parity of the number of intersections



Test if point p is in polygon P 1/2

A simple idea... but hard to put into place

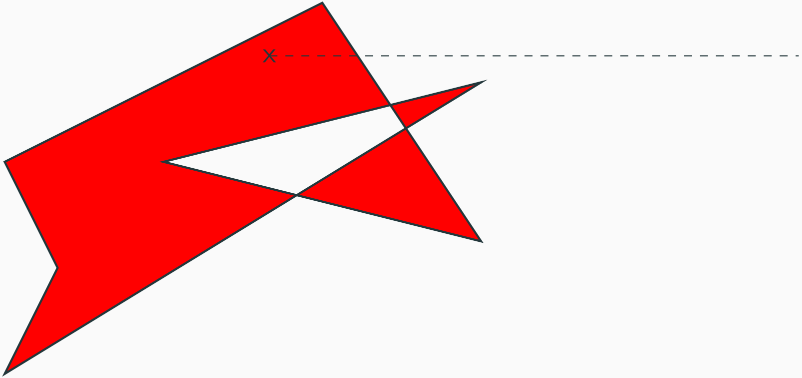
Imagine p shoots a laser in any direction, count the parity of the number of intersections



Test if point p is in polygon P 1/2

A simple idea... but hard to put into place

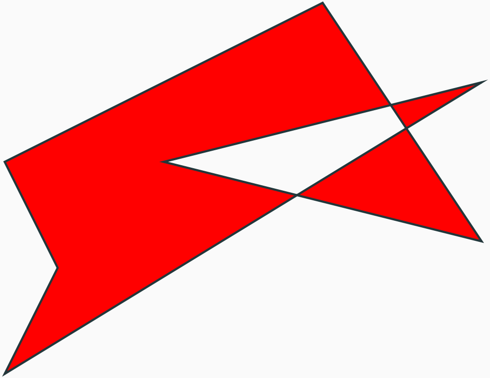
Imagine p shoots a laser in any direction, count the parity of the number of intersections



Test if point p is in polygon P 1/2

A simple idea… but hard to put into place

Imagine p shoots a laser in any direction, count the parity of the number of intersections



What happens when the ray aligns perfectly with a border?

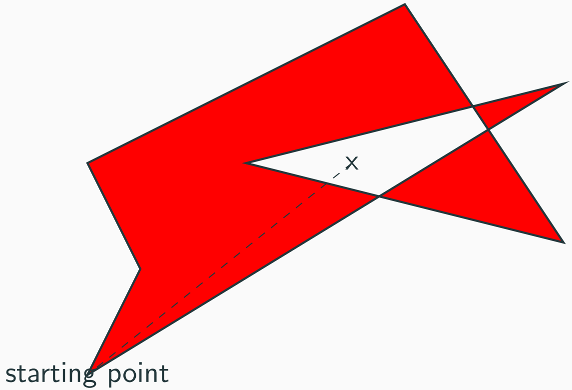
Ray casting algorithm

```
typedef complex<double> vec;
vector<vec>  polygon ;
const double PI = acos(-1);
bool testIn(vec p) {
  vec ray(10000000,1) ; // make sure it is big enough
  vec last = polygon.back();
  int nb_cuts = 0 ;
  for(auto cur : polygon) {
    if(intersect(p,ray,last,cur))
      nb_cuts ++ ;
    last = cur;
  }
  return (nb_cuts%2) == 0;
}
```

Test if point p is in polygon P 2/2

Another idea...

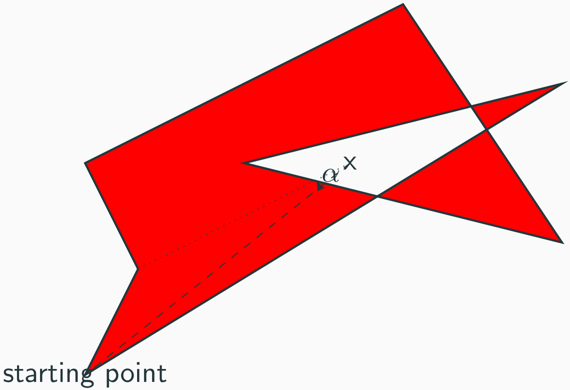
Follow the boundary of the polygon with your eyes, how many turns did you make?



Test if point p is in polygon P 2/2

Another idea...

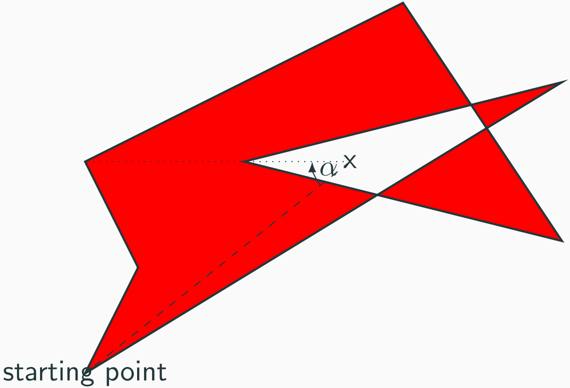
Follow the boundary of the polygon with your eyes, how many turns did you make?



Test if point p is in polygon P 2/2

Another idea...

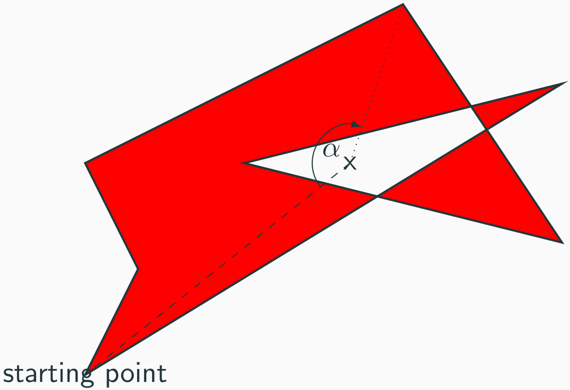
Follow the boundary of the polygon with your eyes, how many turns did you make?



Test if point p is in polygon P 2/2

Another idea…

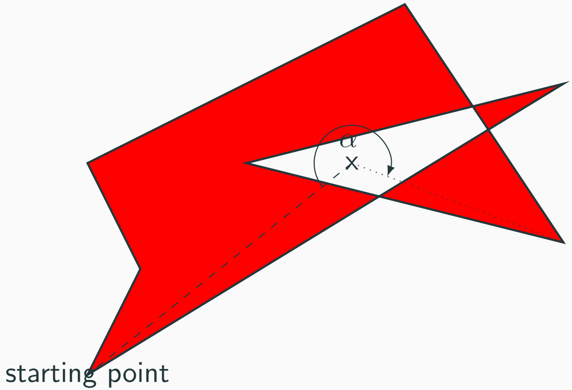
Follow the boundary of the polygon with your eyes, how many turns did you make?



Test if point p is in polygon P 2/2

Another idea...

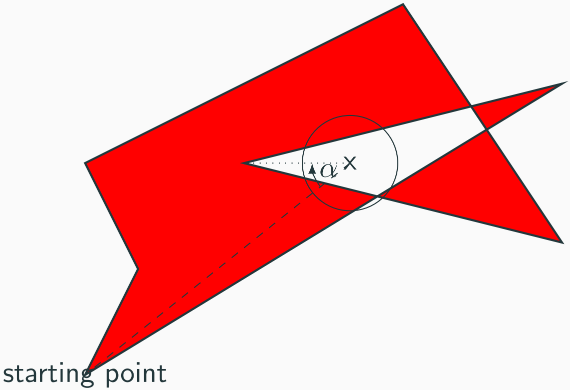
Follow the boundary of the polygon with your eyes, how many turns did you make?



Test if point p is in polygon P 2/2

Another idea...

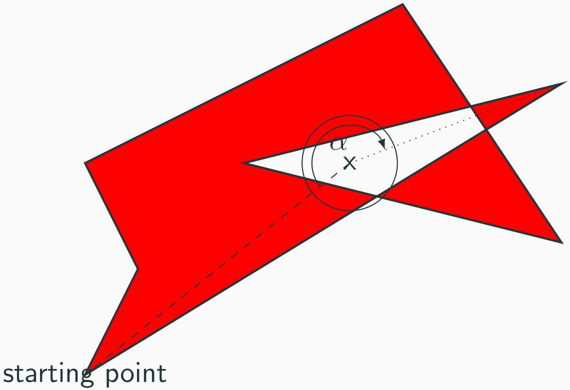
Follow the boundary of the polygon with your eyes, how many turns did you make?



Test if point p is in polygon P 2/2

Another idea...

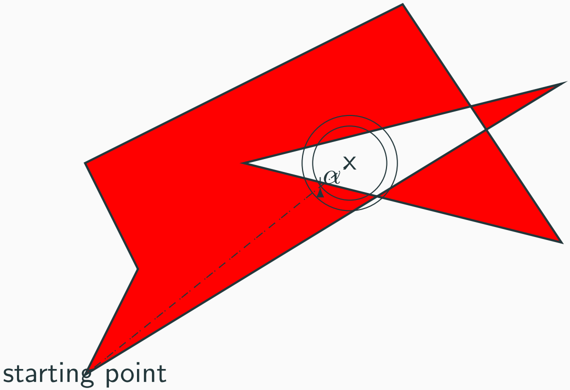
Follow the boundary of the polygon with your eyes, how many turns did you make?



Test if point p is in polygon P 2/2

Another idea...

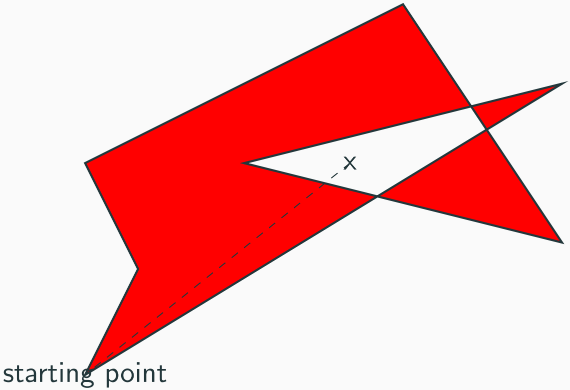
Follow the boundary of the polygon with your eyes, how many turns did you make?



Test if point p is in polygon P 2/2

Another idea...

Follow the boundary of the polygon with your eyes, how many turns did you make?

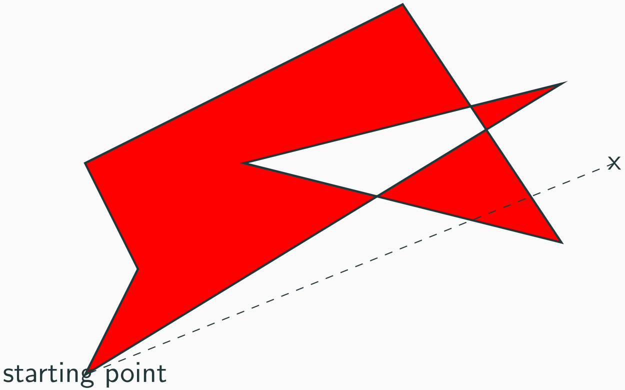


Two turns ⇒ outside!

Test if point p is in polygon P 2/2

Another idea...

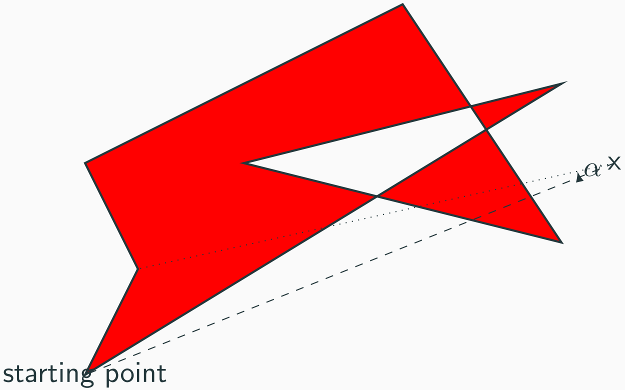
Follow the boundary of the polygon with your eyes, how many turns did you make?



Test if point p is in polygon P 2/2

Another idea...

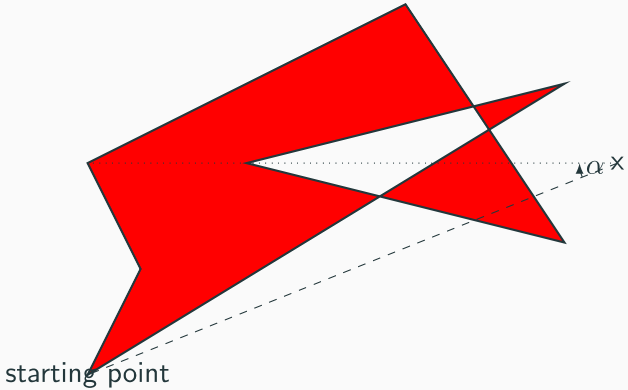
Follow the boundary of the polygon with your eyes, how many turns did you make?



Test if point p is in polygon P 2/2

Another idea...

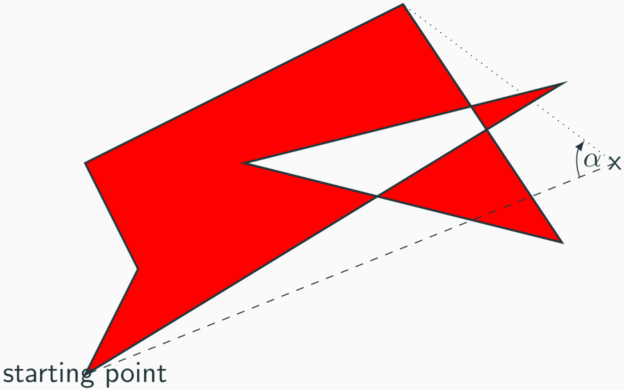
Follow the boundary of the polygon with your eyes, how many turns did you make?



Test if point p is in polygon P 2/2

Another idea...

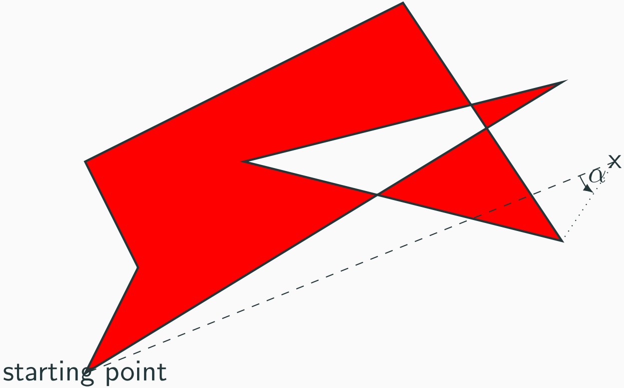
Follow the boundary of the polygon with your eyes, how many turns did you make?



Test if point p is in polygon P 2/2

Another idea...

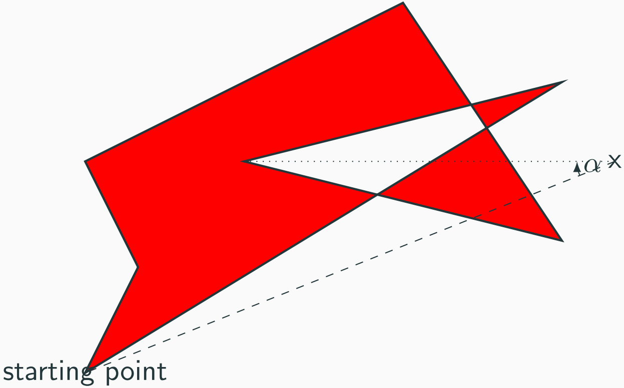
Follow the boundary of the polygon with your eyes, how many turns did you make?



Test if point p is in polygon P 2/2

Another idea...

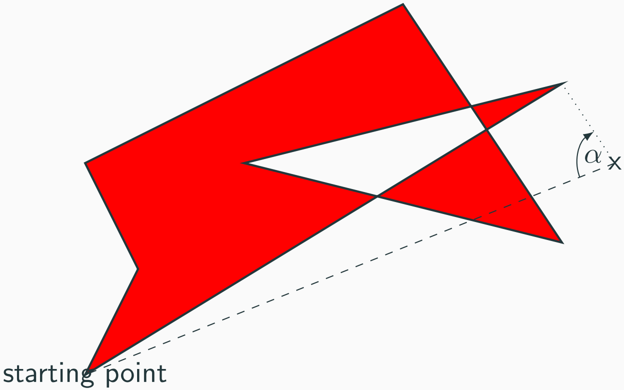
Follow the boundary of the polygon with your eyes, how many turns did you make?



Test if point p is in polygon P 2/2

Another idea...

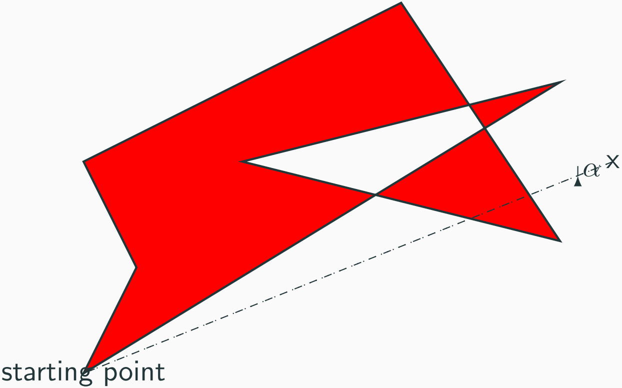
Follow the boundary of the polygon with your eyes, how many turns did you make?



Test if point p is in polygon P 2/2

Another idea...

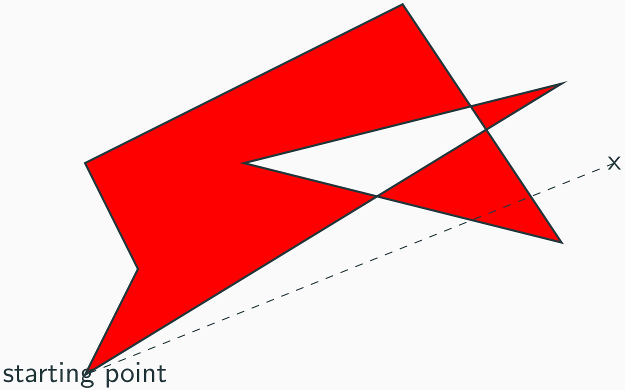
Follow the boundary of the polygon with your eyes, how many turns did you make?



Test if point p is in polygon P 2/2

Another idea...

Follow the boundary of the polygon with your eyes, how many turns did you make?

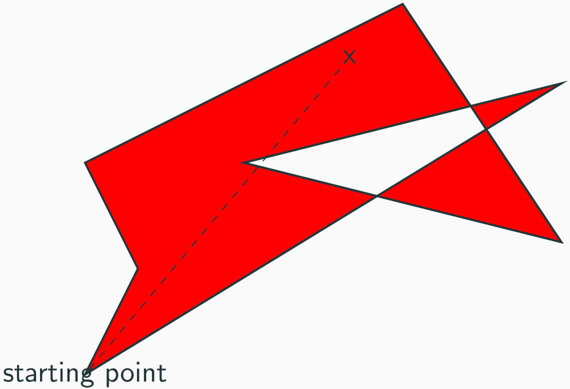


Zero turns ⇒ outside!

Test if point p is in polygon P 2/2

A more subtle idea... but easy to code

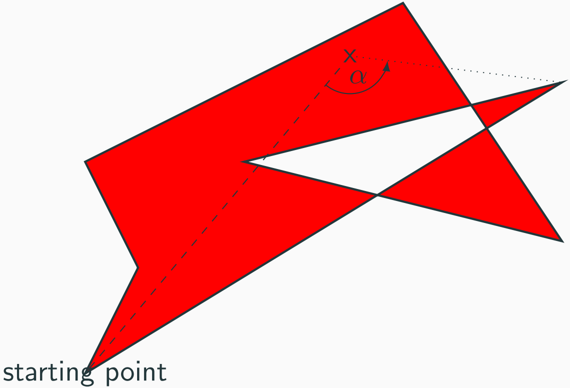
Follow the boundary of the polygon with your eyes, how many turns did you make?



Test if point p is in polygon P 2/2

A more subtle idea... but easy to code

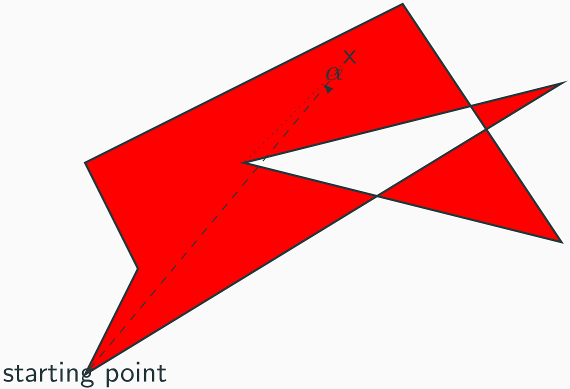
Follow the boundary of the polygon with your eyes, how many turns did you make?



Test if point p is in polygon P 2/2

A more subtle idea... but easy to code

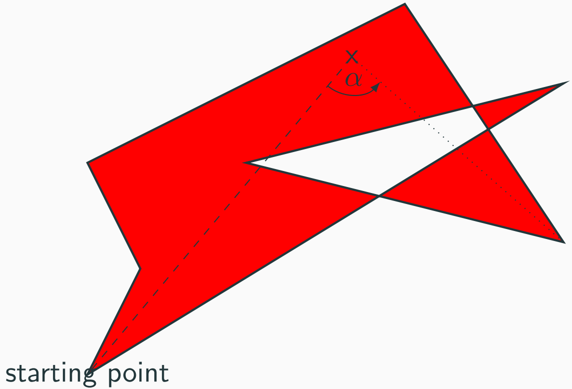
Follow the boundary of the polygon with your eyes, how many turns did you make?



Test if point p is in polygon P 2/2

A more subtle idea... but easy to code

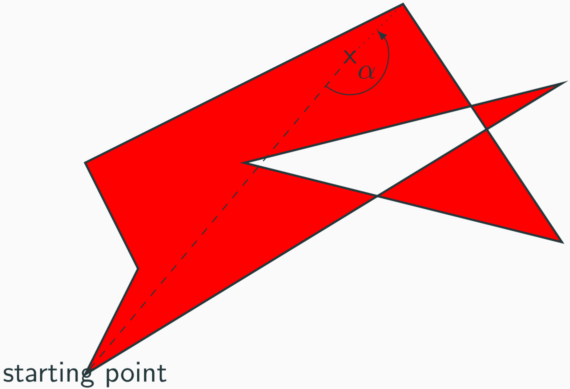
Follow the boundary of the polygon with your eyes, how many turns did you make?



Test if point p is in polygon P 2/2

A more subtle idea... but easy to code

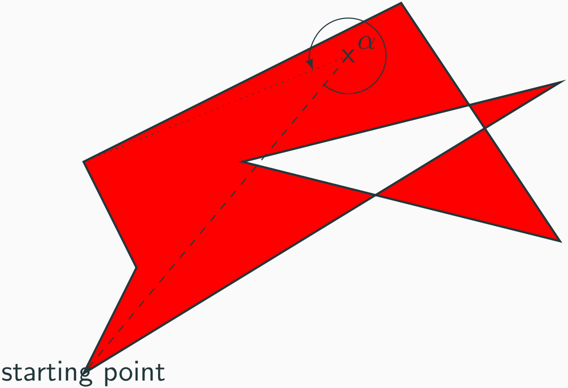
Follow the boundary of the polygon with your eyes, how many turns did you make?



Test if point p is in polygon P 2/2

A more subtle idea... but easy to code

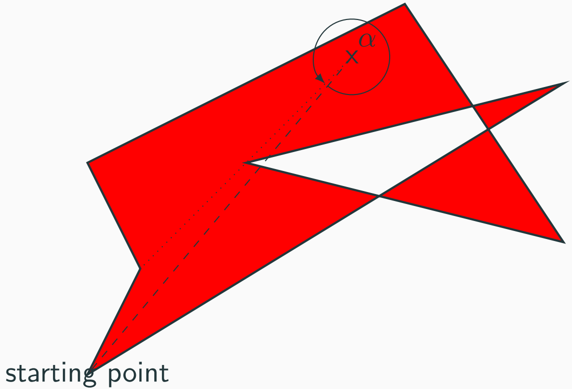
Follow the boundary of the polygon with your eyes, how many turns did you make?



Test if point p is in polygon P 2/2

A more subtle idea... but easy to code

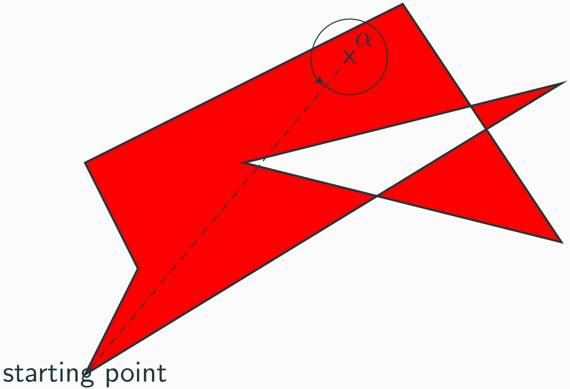
Follow the boundary of the polygon with your eyes, how many turns did you make?



Test if point p is in polygon P 2/2

A more subtle idea... but easy to code

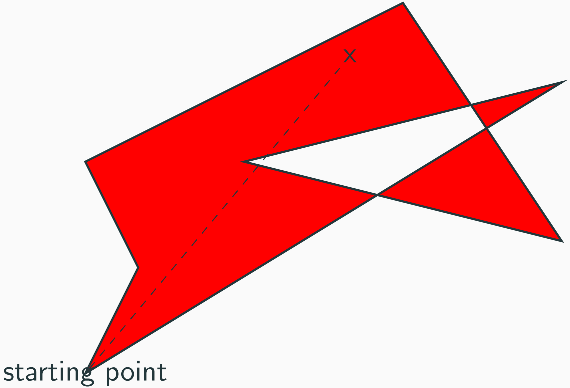
Follow the boundary of the polygon with your eyes, how many turns did you make?



Test if point p is in polygon P 2/2

A more subtle idea... but easy to code

Follow the boundary of the polygon with your eyes, how many turns did you make?

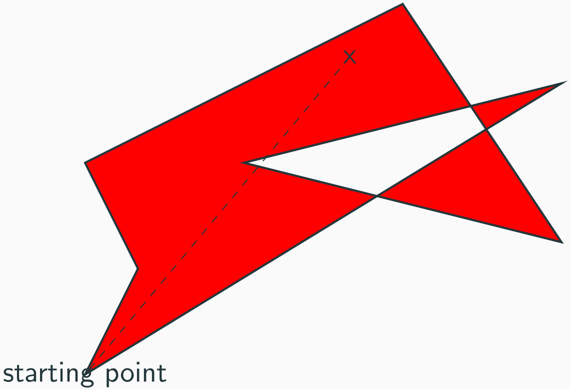


One turn ⇒ inside!

Test if point p is in polygon P 2/2

A more subtle idea... but easy to code

Follow the boundary of the polygon with your eyes, how many turns did you make?



For even-odd polygons, even number of turns = outside.

Test if point p is in polygon P

```cpp
typedef complex<double>  vec;
vector<vec>  polygon ;
const double PI = acos(-1);
double angle(vec a, vec b) {
  const double angle = fmod(a.arg()-b.arg(),2*PI);
  if(angle <= PI) return angle ;
  return angle-2*PI;
}
bool windingNumber(vec p) {
  double totalArg = 0;
  vec last = polygon.back();
  for(auto cur : polygon) {
    totalArg += angle(cur-p, last-p);
    last=cur;
  }
  return fabs(fmod(totalArg/(2*PI),2))<0.1 ;
}
```

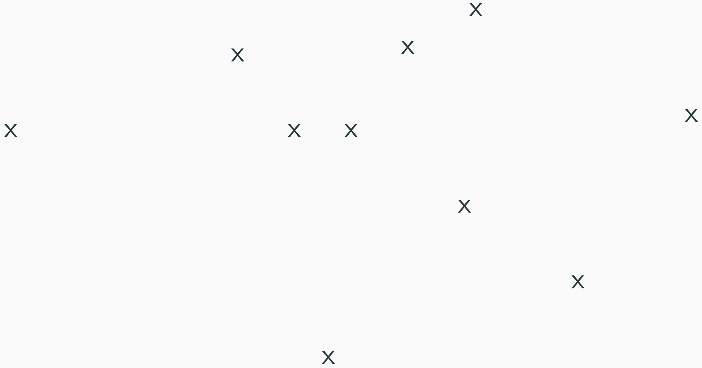
Classical algorithms for computational geometry

Convex Hulls

Convex hulls

Definition

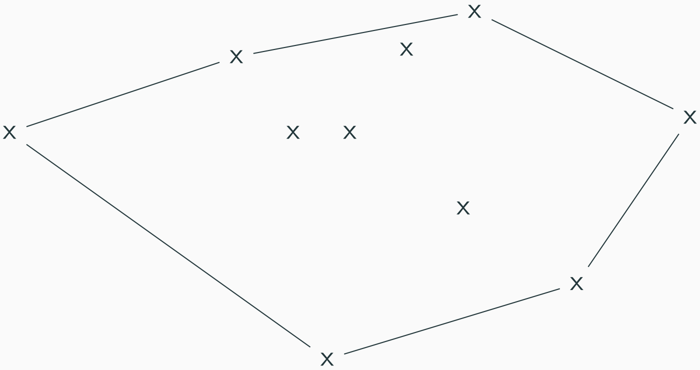
Smallest convex polygon containing a set of points on a grid.



Convex hulls

Definition

Smallest convex polygon containing a set of points on a grid.



Monotone chain / Andrew algorithm in $O(n \ln(n))$

Idea to compute the top of the CH

- sort the point lexicographically
- remove points that have the same x
- for each point p by increasing x
  - add p to the hull
  - if the last three points in the hull turn right
    - remove the penultimate point
- if needed, restart the CH for the bottom part

Convex hulls

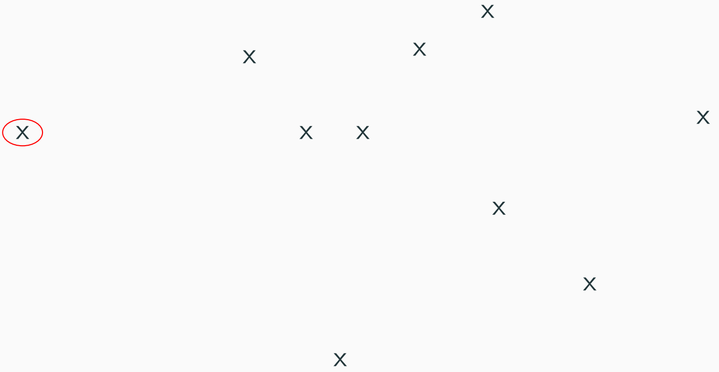
Definition

Smallest convex polygon containing a set of points on a grid.

Convex hulls

Definition

Smallest convex polygon containing a set of points on a grid.



Convex hulls

Definition

Smallest convex polygon containing a set of points on a grid.

Convex hulls

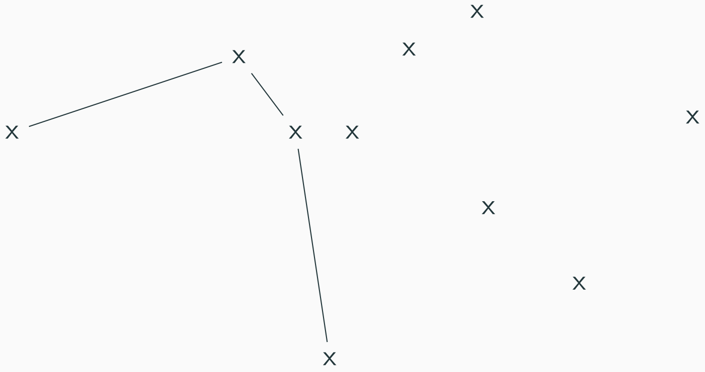
Definition

Smallest convex polygon containing a set of points on a grid.

Convex hulls

Definition

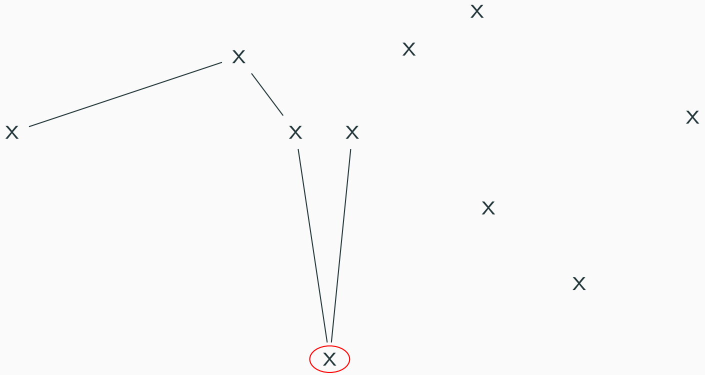
Smallest convex polygon containing a set of points on a grid.



Convex hulls

Definition

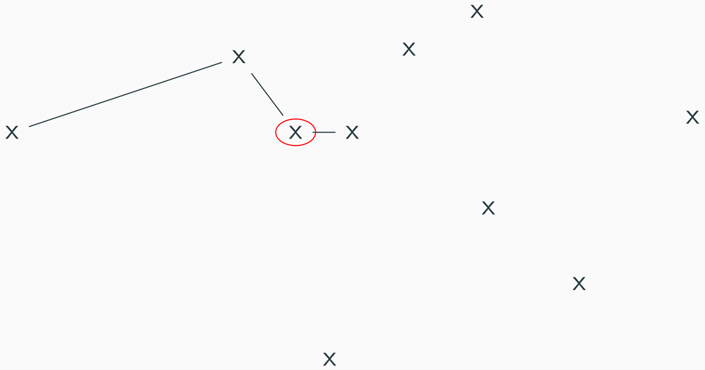
Smallest convex polygon containing a set of points on a grid.



Convex hulls

Definition

Smallest convex polygon containing a set of points on a grid.



Convex hulls

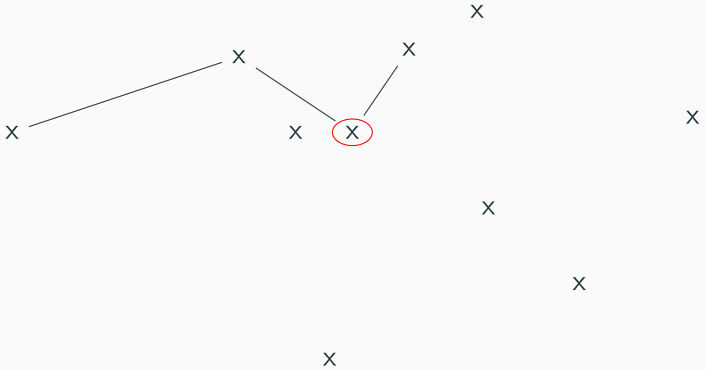
Definition

Smallest convex polygon containing a set of points on a grid.

Convex hulls

Definition

Smallest convex polygon containing a set of points on a grid.



Convex hulls

Definition

Smallest convex polygon containing a set of points on a grid.

Convex hulls

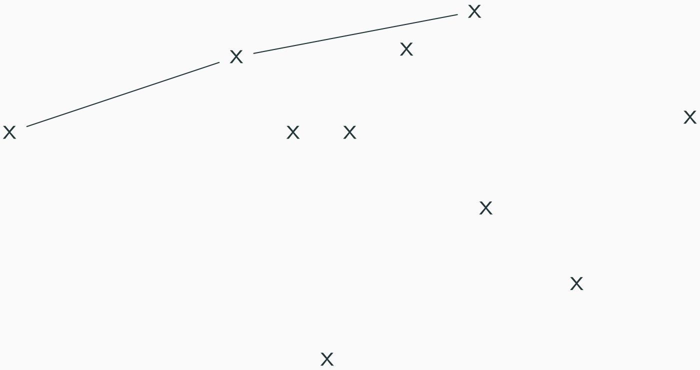
Definition

Smallest convex polygon containing a set of points on a grid.

Convex hulls

Definition

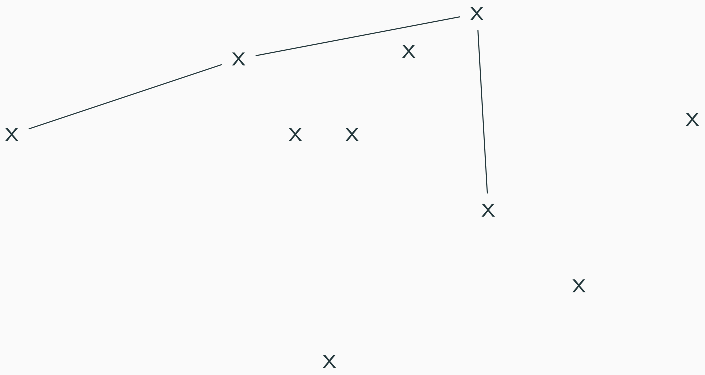
Smallest convex polygon containing a set of points on a grid.



Convex hulls

Definition

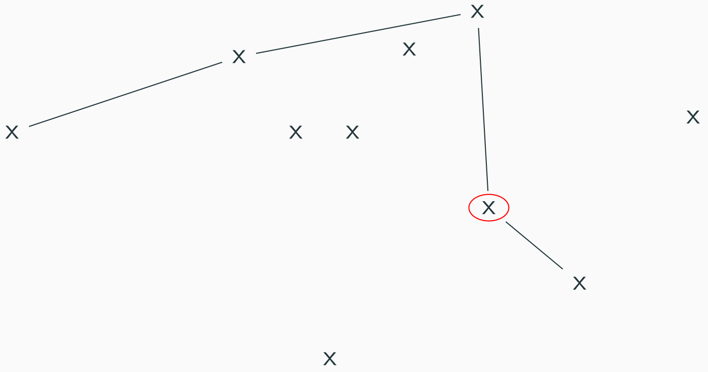
Smallest convex polygon containing a set of points on a grid.



Convex hulls

Definition

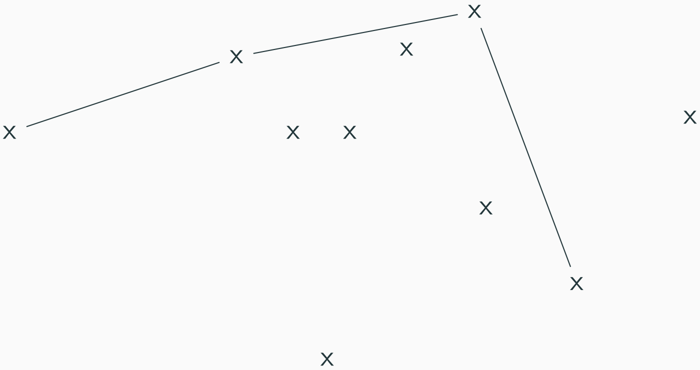
Smallest convex polygon containing a set of points on a grid.



Convex hulls

Definition

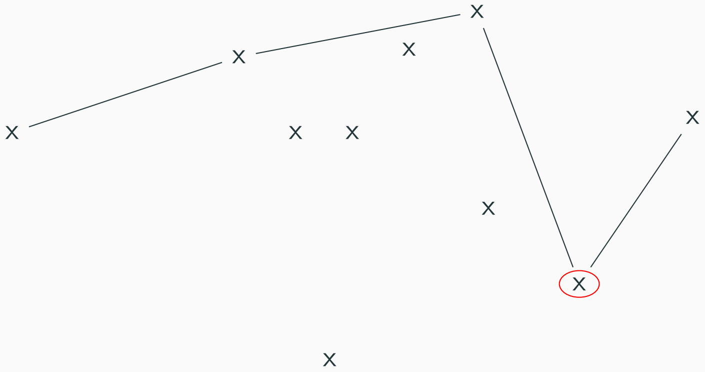
Smallest convex polygon containing a set of points on a grid.



Convex hulls

Definition

Smallest convex polygon containing a set of points on a grid.



Convex hulls

Definition

Smallest convex polygon containing a set of points on a grid.

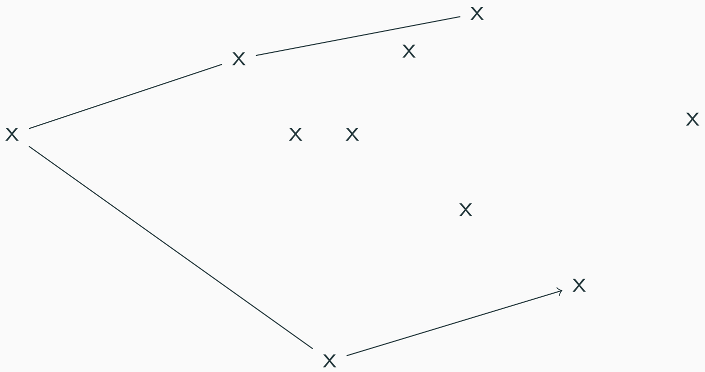
Andrew algorithm

```
vector<pii> convexHull(vector<pii> in) {
  map<int,int> coord ;
  for( pii p : in)
    coord[p.first] = max(coord[p.first],p.second) ;
  vector<pii> res ;
  for(auto & p : coord) {
    res.push_back(p);
    while(res.size() >= 3 &&
       turnRight(res[res.size()-3],
                 res[res.size()-2],res[res.size()-1]))
        res.erase(res.end()-2);
  }
  return res;
}
```

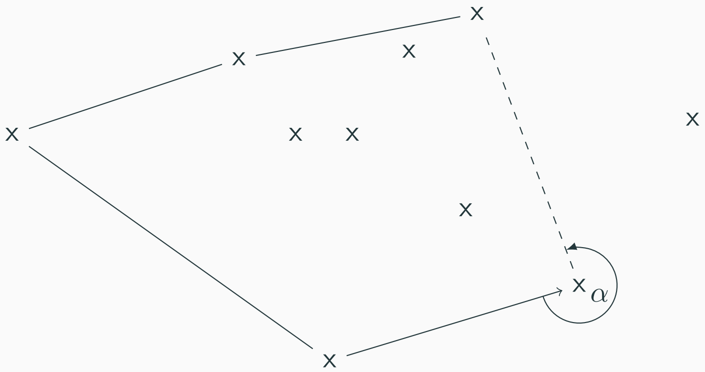
Gift wrapping algorithm a.k.a. Jarvis march

- start from the leftmost point with a vertical downward segment
- while we have not returned to the starting point
    - find next point in hull by minimizing angle with line
    - add point to hull
    - set line as the last two points in hull

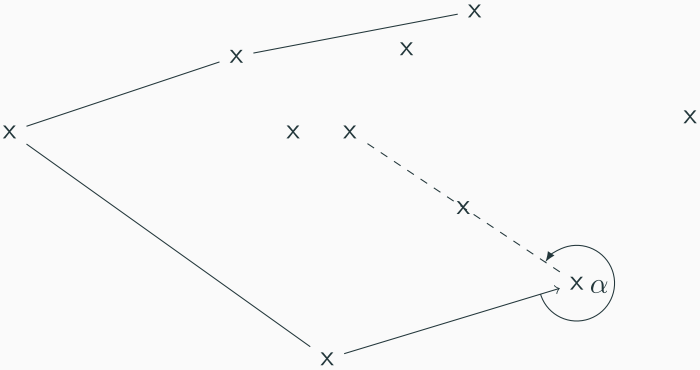
Jarvis march



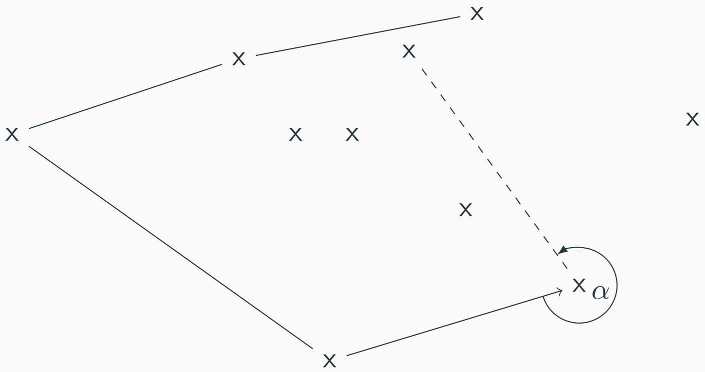
Jarvis march



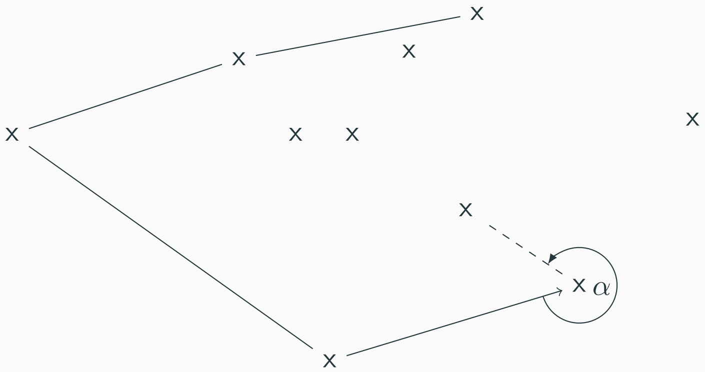
Jarvis march



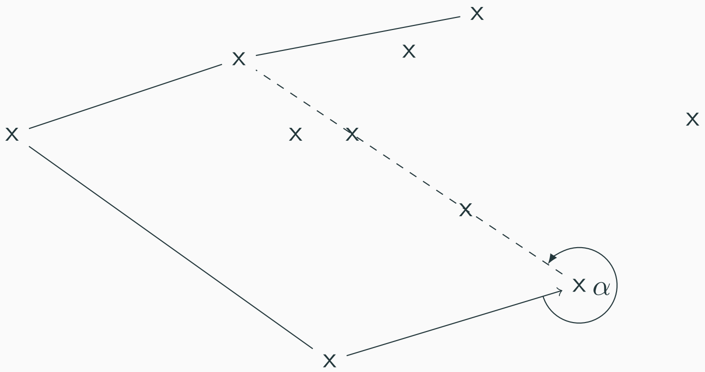
Jarvis march



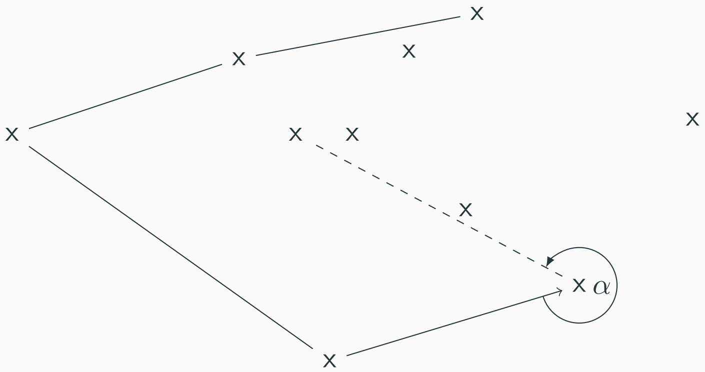
Jarvis march



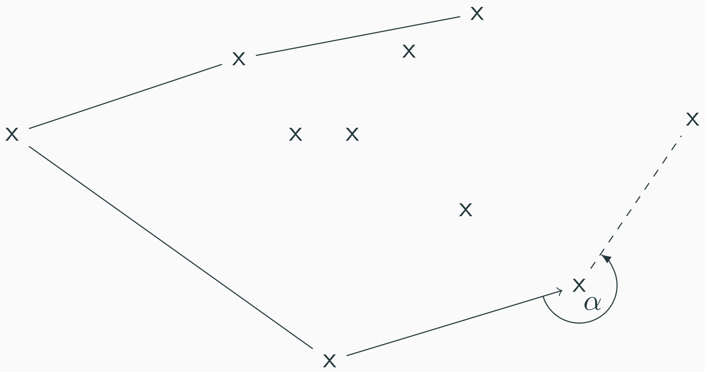
Jarvis march



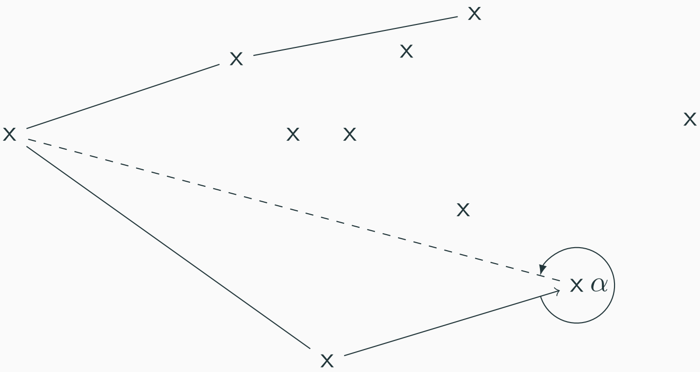
Jarvis march



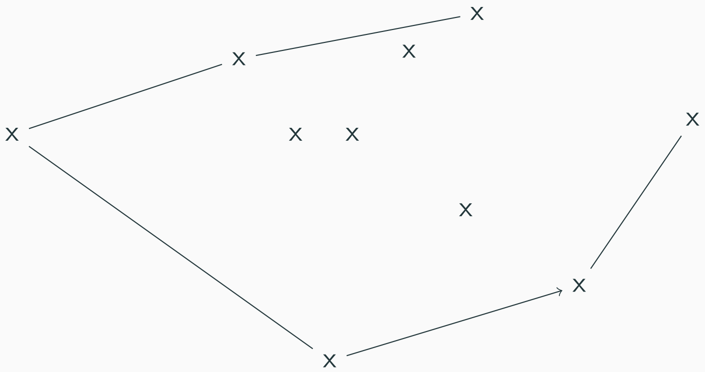
Jarvis march



Jarvis march



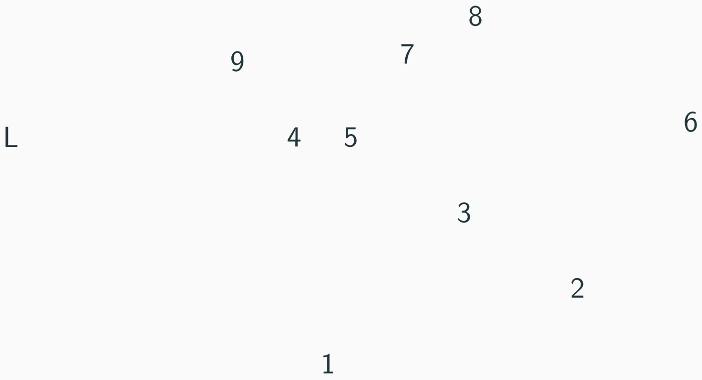
Jarvis march



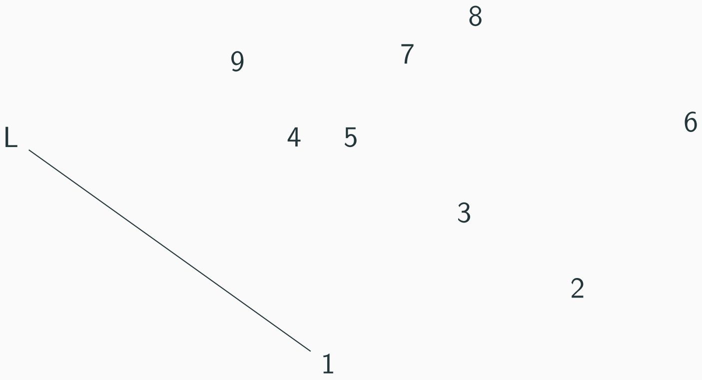
Graham scan

- start from the leftmost point L
- for all points P in increasing order of angle $\vec{LP}$
  - add P to convex hull
  - while the last three points turn right
    - remove penultimate point of the hull

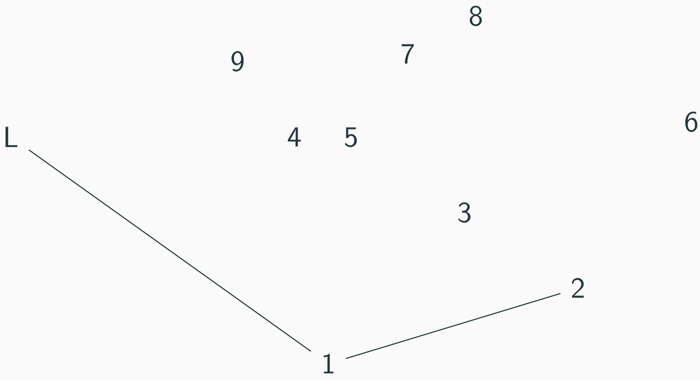
Graham scan



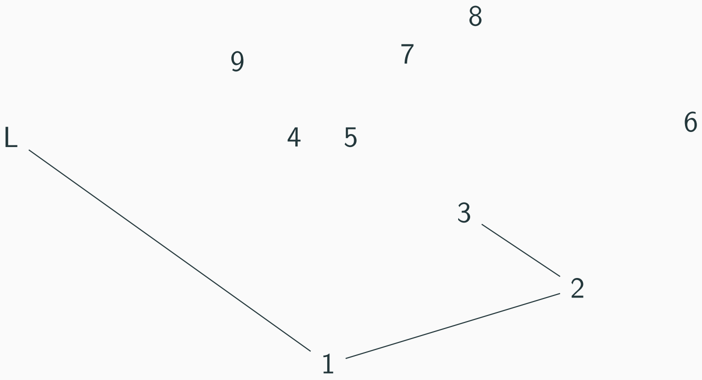
Graham scan



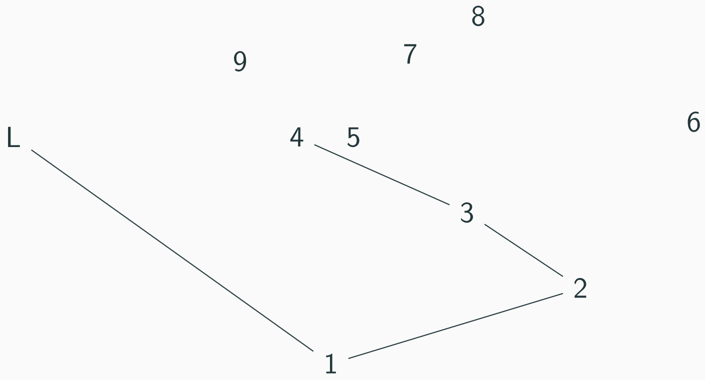
Graham scan



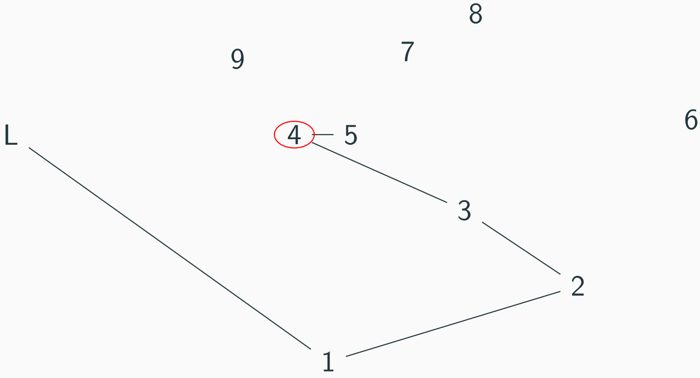
Graham scan



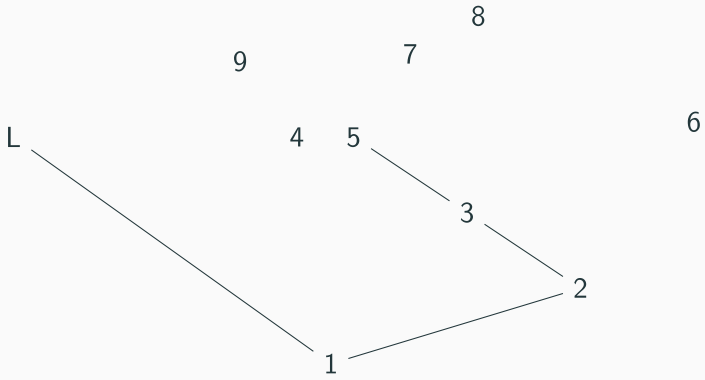
Graham scan



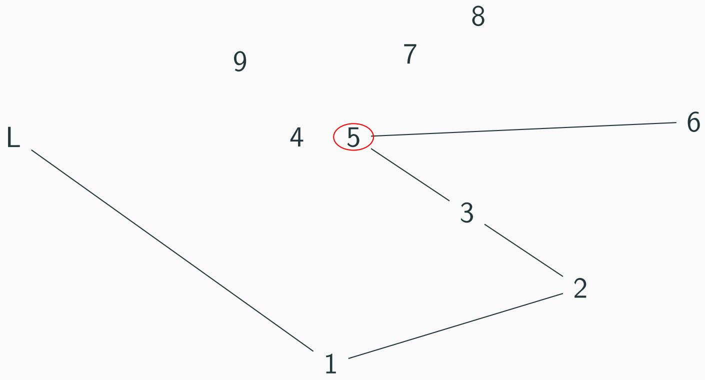
Graham scan



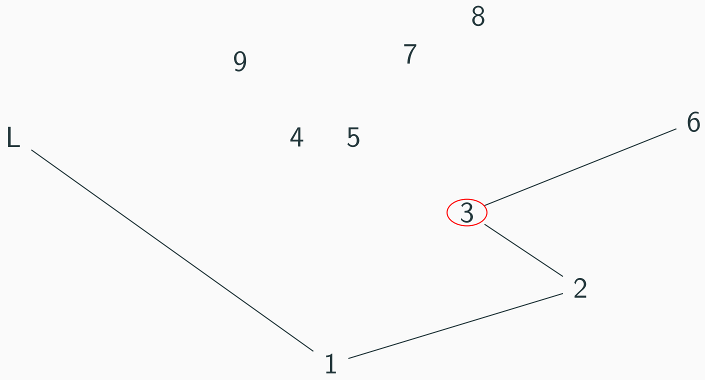
Graham scan



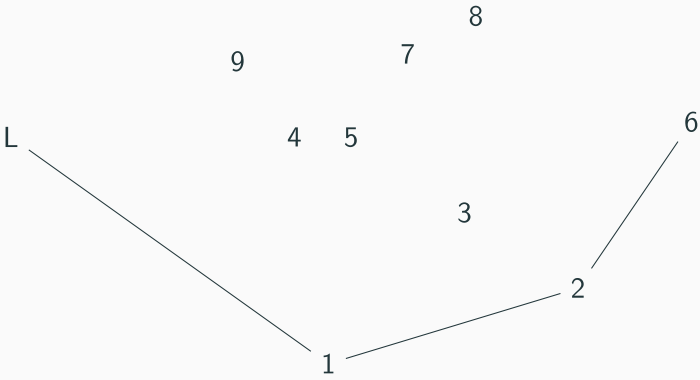
Graham scan



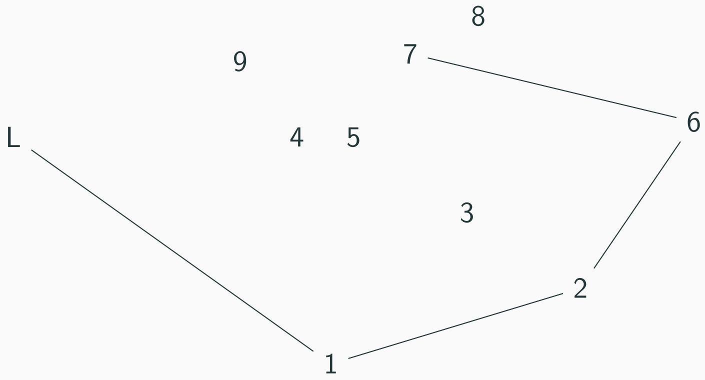
Graham scan



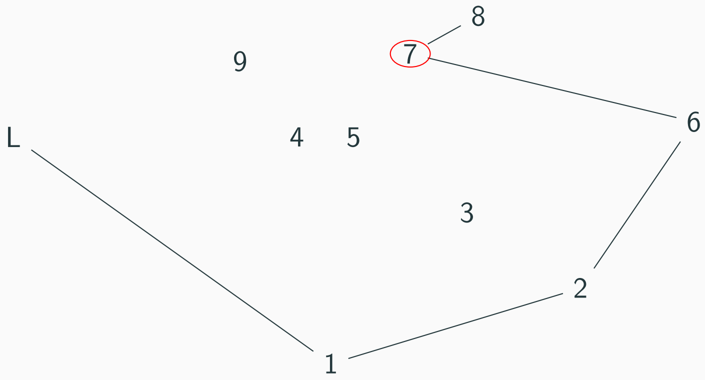
Graham scan



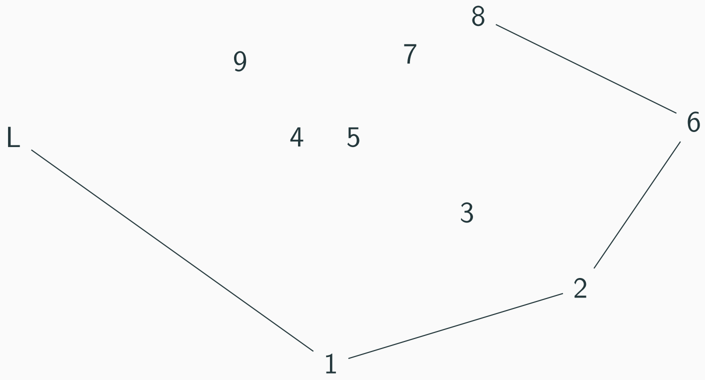
Graham scan



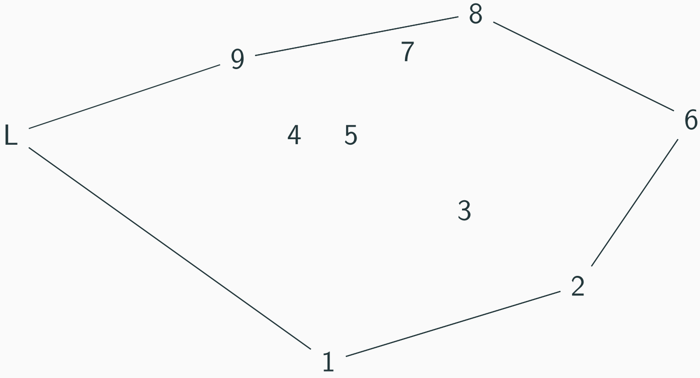
Graham scan



Graham scan



Graham scan



Convex Hull algorithms

Andrew algorithm $O(n \ln(n))$

Simple algorithm for the top of the convex hull

Jarvis march $O(nh)$

Simple algorithm and efficient when h is small

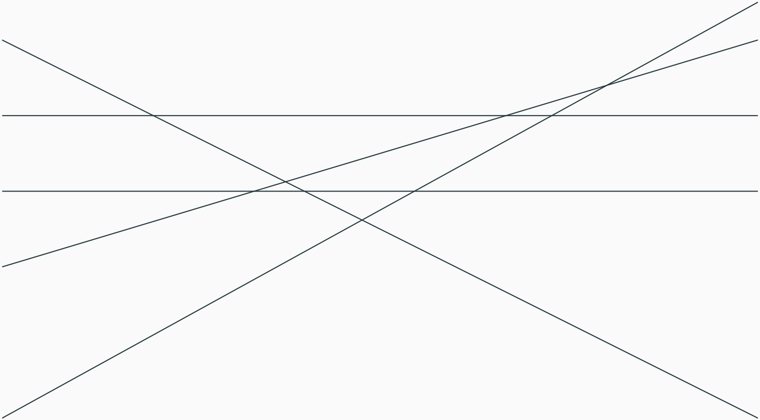
Graham scan $O(n \ln(n))$

Works on all cases

Hidden convex hulls

Problem idea

You are given n linear functions $f_i(x) = a_i x + b_i$, compute the function $f(x) = max_i f_i(x)$.



Classical algorithms for computational geometry

Sweep line

Sweep line method

General idea:

- sort all points lexicographically
- maintain a sliding window

Example problem:

Given n rectangles, check whether at least two intersect

Solution:

Maintain an ordered binary search tree for intervals $[y_1, y_2]$. Each rectangle (x_1, y_1, x_2, y_2) will be considered twice, at x_1 (opening) and at x_2 (closing).

Handle opening and closing events by increasing x: for openings we add (y_1, y_2) to the BST and check the y-intervals before and after, for closings we remove the interval.

Classical algorithms for computational geometry

Using different norms

Sweep line method

L_2 norm:

Usual norms, above algorithms apply

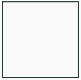
L_∞ norm:

Circles are squares, this often boils down to sliding window algorithm

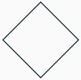
L_1 norm:

One can apply the transformation $(x, y) \to (x + y, x - y)$ to recover squares.

L_2 circle



L_∞ circle



L_1 circle