Geometry

Atli FF

6. nóvember 2023

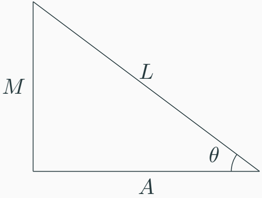
School of Computer Science
Reykjavík University

Today's material

- Trigonometry
- Geometry
- Computational geometry

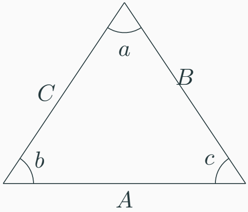
Trigonometry

- Before we even dive into the geometry and how to do it on a computer, let's jog your memories.
- You should all hopefully be familiar with the trigonometric functions.
- We consider a triangle to be right-angled if it has a corner that's 90°.
- For such triangles we have:
  - $\frac{A}{L} = \cos\theta$.
  - $\frac{M}{L} = \sin\theta$.
  - $\frac{M}{A} = \frac{M}{L}\frac{L}{A} = \frac{\sin\theta}{\cos\theta} = \tan\theta$.
- We also have the pythagorean theorem $L^2 = A^2 + M^2$.



More trig

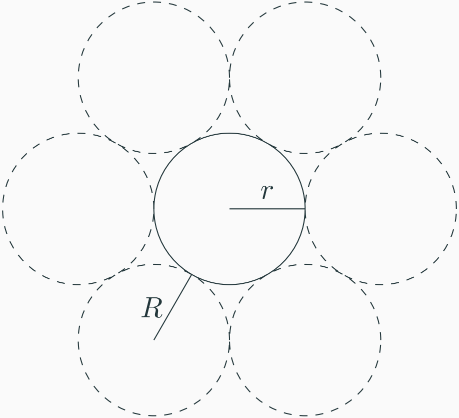
- More generally we have:
  - $\dfrac{\sin a}{A} = \dfrac{\sin b}{B} = \dfrac{\sin c}{C}$ (sine law).
  - $A^2 = B^2 + C^2 - 2BC\cos a$ (cosine law)
- **Exercise:** Prove the pythagorean theorem using the cosine law.



Example: NN and the Optical Illusion

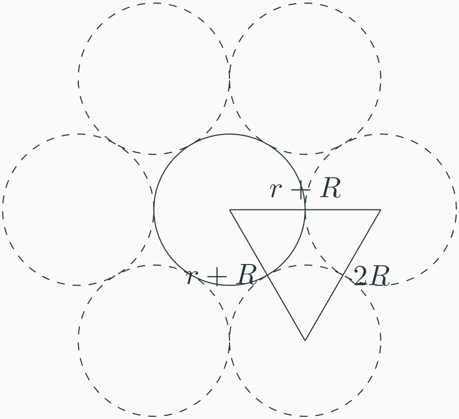
- You are given an integer n and a real number r.
- You then draw a circle of radius r.
- You then want to draw n circles of the same size tangent to the outside of this circle and such that they are tangent to their neighbours.
- What radius will the outer circles have?

$N = 6$ image



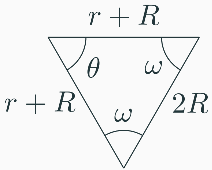
Towards a solution

We see that the distance from the center of the circle in the middle to the center of an outer circle is $r + R$. We thus get an isosceles triangle.



Closer and closer

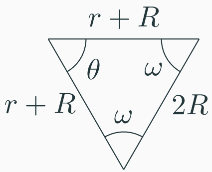
- Now we have $\theta = \dfrac{360^\circ}{n}$ and $\omega = \dfrac{180^\circ - \theta}{2}$.

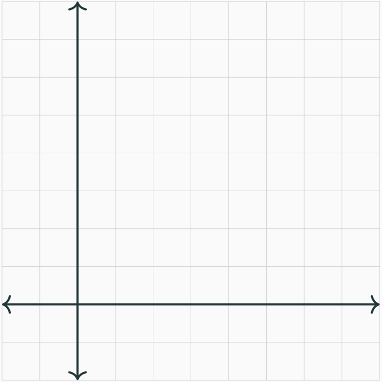


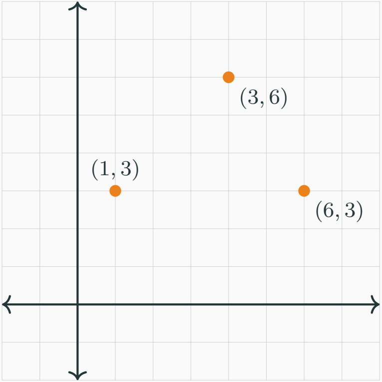
Solution

- Finally the law of sines gives us

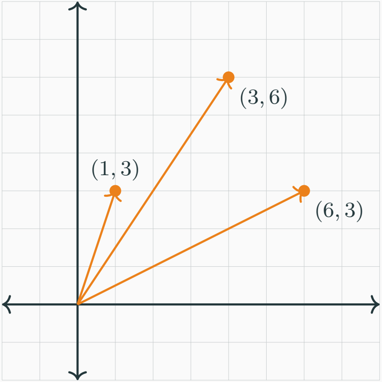
$$\begin{aligned}\frac{2R}{\sin\theta} = \frac{r+R}{\sin\omega} &\Rightarrow 2R\sin\omega = r\sin\theta + R\sin\theta\\ &\Rightarrow 2R\sin\omega - R\sin\theta = r\sin\theta\\ &\Rightarrow R = \frac{r\sin\theta}{2\sin\omega - \sin\theta}.\end{aligned}$$



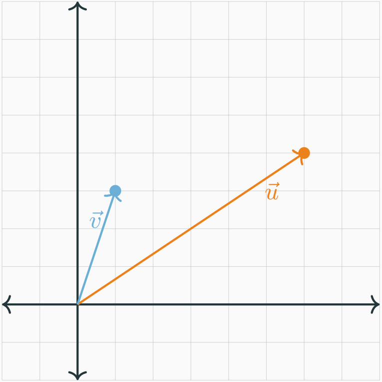




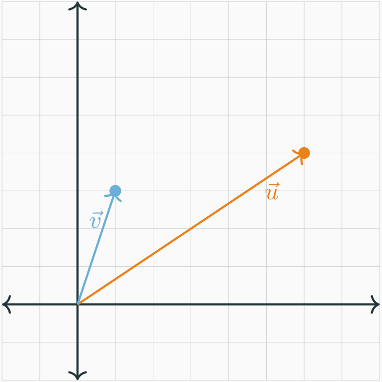
- Points are represented by a pair of numbers, (x, y).



- Points are represented by a pair of numbers, (x, y).
- Vectors are represented in the same way.
- Thinking of points as vectors allows us to do many things.

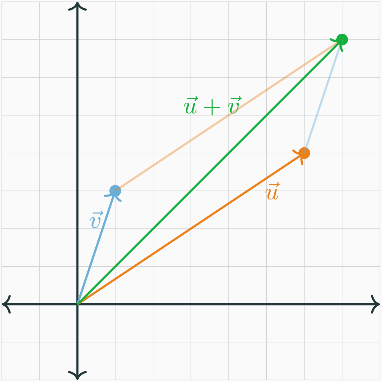


$\vec{v}$
$\vec{u}$



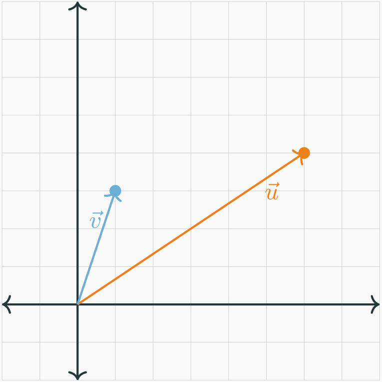
- Simplest operation, addition is defined as

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_0 + x_1 \\ y_0 + y_1 \end{pmatrix}$$



- Simplest operation, addition is defined as

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_0 + x_1 \\ y_0 + y_1 \end{pmatrix}$$

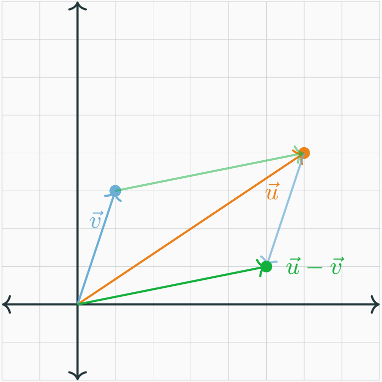


- Simplest operation, addition is defined as

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_0 + x_1 \\ y_0 + y_1 \end{pmatrix}$$

- Subtraction is defined in the same manner

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} - \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_0 - x_1 \\ y_0 - y_1 \end{pmatrix}$$



- Simplest operation, addition is defined as

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_0 + x_1 \\ y_0 + y_1 \end{pmatrix}$$

- Subtraction is defined in the same manner

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} - \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_0 - x_1 \\ y_0 - y_1 \end{pmatrix}$$

```cpp
struct point {
    double x, y;
    point(double _x, double _y) {
        x = _x, y = _y;
    }

    point operator+(const point &oth){
        return point(x + oth.x, y + oth.y);
    }

    point operator-(const point &oth){
        return point(x - oth.x, y - oth.y);
    }
};
```

…or we could use the `complex<double>` class.

```
using points = complex<double>;
```

. . . or we could use the `complex<double>` class.

```
using points = complex<double>;
```

The `complex` class in `C++` and `Java` has methods defined for

- Addition
- Subtraction
- Multiplication by a scalar
- Length
- Trigonometric functions
- And much more!

Complex numbers

- We define $\mathbb{C} := \mathbb{R} \times \mathbb{R}$.
- Then we define addition on $\mathbb{C}$ such that for $(a, b), (c, d) \in \mathbb{C}$ we get
$$(a, b) + (c, d) = (a + c, b + d).$$
- We also define multiplication on $\mathbb{C}$ such that for $(a, b), (c, d) \in \mathbb{C}$ we get
$$(a, b) \cdot (c, d) = (ac - bd, ad + bc).$$
- We usually denote $(0, 1) \in \mathbb{C}$ by i and $(x, y) \in \mathbb{C}$ by $x + yi$.
- Note that $(x, y) = (x, 0) + i \cdot (y, 0)$ here.
- We call these numbers in $\mathbb{C}$ *complex numbers*.

Complex numbers ctd.

- If $z = x + yi \in \mathbb{C}$ then
  - We call x the *real part* of z and y the *imaginary part* of z.
  - We define the *magnitude* of z by $|z| = \sqrt{x^2 + y^2}$.
  - We call $x - yi$ the *conjugate* of z, denoted by $\overline{z}$.
  - We call the angle (x, y) makes with the positive x-axis the *argument* of z and denote it by $\mathrm{Arg}(z)$.

Operations

- Let $w, z \in \mathbb{C}$.
- Then $w + z$ will be z translated by w, as if we were adding vectors.
- If $|w| = 1$ then $z \cdot w$ will be z rotated around 0 by $\operatorname{Arg}(w)$ radians.
- If $|z| = r$ and $\operatorname{Arg}(z) = \theta$ we can write $z = re^{i\theta}$.
- If $z = r_1 e^{i\theta_1}$ and $w = r_2 e^{i\theta_2}$ then $z \cdot w = r_1 r_2 e^{i(\theta_1 + \theta_2)}$.

Using complex in C++

- Usually we do `using point = complex<double>`
- Then we can initialize a point with `point z(x, y)`
  - `real(z)` returns the x-coordinate
  - `imag(z)` returns the y-coordinate
  - `abs(z)` returns the magnitude $|z|$
  - `abs(z - w)` returns the distance from z to w
  - `arg(z)` returns the argument of z
  - `conj(z)` returns the conjugate $\overline{z}$

Example

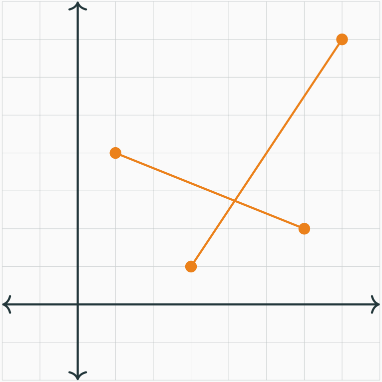
- Let us consider a problem.
- You start at $(0, 0)$ and get a sequence of commands.
- All the commands consist of a single letter and a number. The commands are:
    - ...`f x` you move forward x meters..
    - ...`b x` you move backwards x meters.
    - ...`r x` you rotate x radians to the right.
    - ...`l x` you rotate x radians to the left.
- How far from $(0, 0)$ do you end up after following the commands?

Solution

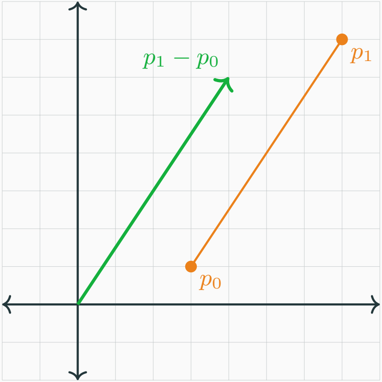
- If we are stood at $p \in \mathbb{C}$ and want to take a step of r meters in the direction θ we simply add $re^{i\theta}$ to p.
- What direction we are facing at the start makes no difference since it gives the same distance at the end.

Code

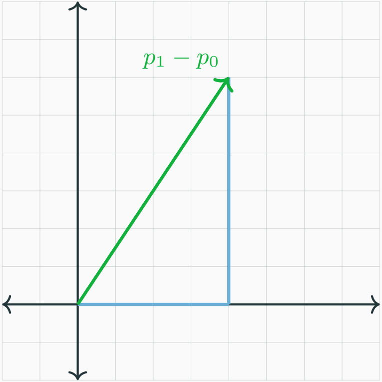
```cpp
#include <bits/stdc++.h>
using namespace std;
using point = complex<double>;
int main() {
    int n; cin >> n;
    double x, r = 0.0;
    point p(0, 0);
    while (n--) {
        char c; cin >> c >> x;
        if (c == 'f')      p += x*exp(1i*r);
        else if (c == 'b') p -= x*exp(1i*r);
        else if (c == 'l') r += x;
        else if (c == 'r') r -= x;
        else assert(0);
        }
    cout << setprecision(15) << abs(p) << endl;
}
```



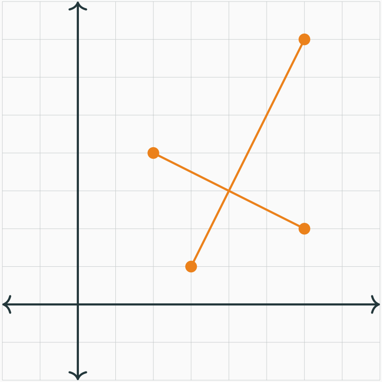
- Line segments are represented by a pair of points, $((x_0, y_0), (x_1, y_1))$.



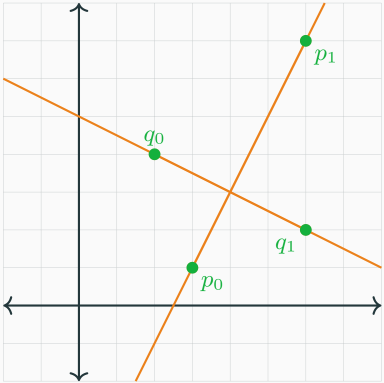
- Line segments are represented by a pair of points, $((x_0, y_0), (x_1, y_1))$.



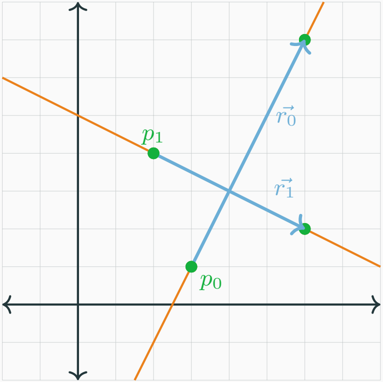
- Line segments are represented by a pair of points, $((x_0, y_0), (x_1, y_1))$.



- Line representation same as line segments.

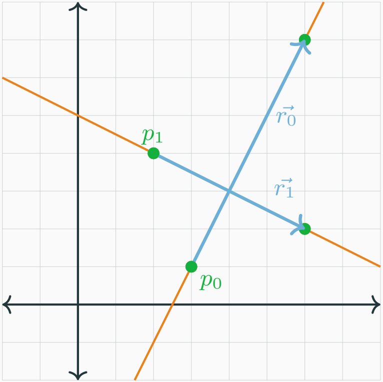


- Line representation same as line segments.
- Treat them as lines passing through the two points.



- Line representation same as line segments.
- Treat them as lines passing through the two points.
- Or as a point and a direction vector.

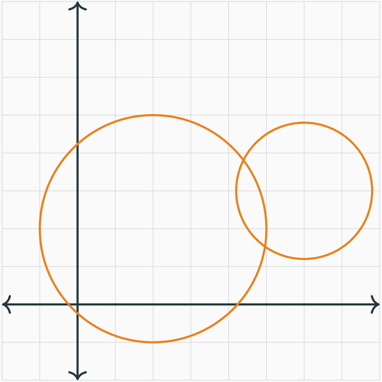
$$p + t \cdot \vec{r}$$



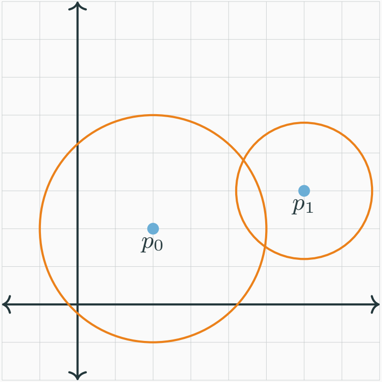
- Line representation same as line segments.
- Treat them as lines passing through the two points.
- Or as a point and a direction vector.

$$p + t \cdot \vec{r}$$

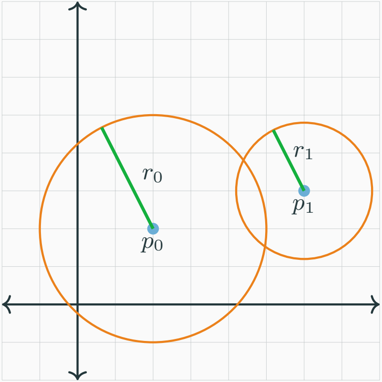
- Either way `pair<point,point>`



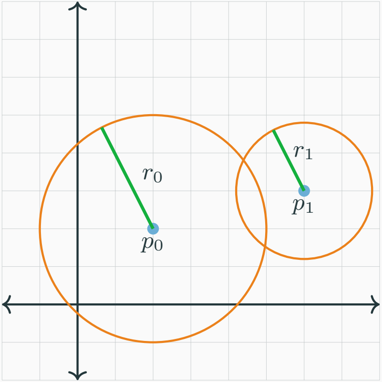
- Circles are very easy to represent.



- Circles are very easy to represent.
- Center point $p = (x, y)$.



- Circles are very easy to represent.
- Center point $p = (x, y)$.
- And the radius r.



- Circles are very easy to represent.
- Center point $p = (x, y)$.
- And the radius r.

`pair<point,double>`

Given two vectors

$$\vec{u} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \qquad \vec{v} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}$$

the dot product of $\vec{u}$ and $\vec{v}$ is defined as

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = x_0 \cdot x_1 + y_0 \cdot y_1$$

Given two vectors

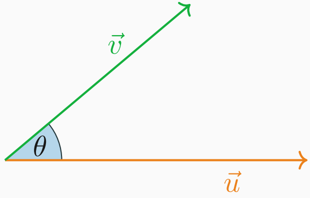
$$\vec{u} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \qquad \vec{v} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}$$

the dot product of $\vec{u}$ and $\vec{v}$ is defined as

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = x_0 \cdot x_1 + y_0 \cdot y_1$$

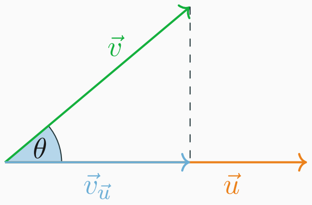
Which in geometric terms is

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



- Allows us to calculate the angle between $\vec{u}$ and $\vec{v}$.

$$\theta = \arccos\left(\frac{\vec{u}\cdot\vec{v}}{|\vec{u}||\vec{v}|}\right)$$

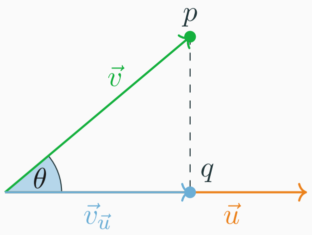


- Allows us to calculate the angle between $\vec{u}$ and $\vec{v}$.

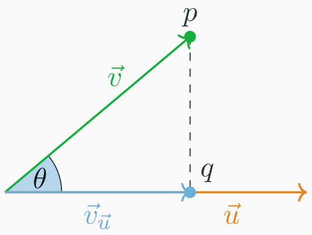
$$\theta = \arccos\left(\frac{\vec{u} \cdot \vec{v}}{|\vec{u}||\vec{v}|}\right)$$

- And the projection of $\vec{v}$ onto $\vec{u}$.

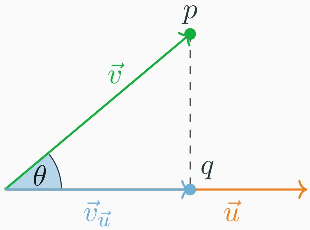
$$\vec{v}_{\vec{u}} = \left(\frac{\vec{u} \cdot \vec{v}}{|u|^2}\right)\vec{u}$$



- The closest point on $\vec{u}$ to p is q.



- The closest point on $\vec{u}$ to p is q.
- The distance from p to $\vec{u}$ is the distance from p to q.



- The closest point on $\vec{u}$ to p is q.
- The distance from p to $\vec{u}$ is the distance from p to q.
- Unless q is outside $\vec{u}$, then the closest point is either of the endpoints.

Rest of the code will use the `complex` class.

```cpp
#define P(p) const point &p
#define L(p0, p1) P(p0), P(p1)
double dot(P(a), P(b)) {
    return real(a) * real(b) + imag(a) * imag(b);
}
double angle(P(a), P(b), P(c)) {
    return acos(dot(b - a, c - b) / abs(b - a) / abs(c - b));
}
point closest_point(L(a, b), P(c), bool segment = false) {
    if (segment) {
        if (dot(b - a, c - b) > 0) return b;
        if (dot(a - b, c - a) > 0) return a;
    }
    double t = dot(c - a, b - a) / norm(b - a);
    return a + t * (b - a);
}
```

Given two vectors

$$\vec{u} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \qquad \vec{v} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}$$

the cross product of $\vec{u}$ and $\vec{v}$ is defined as

$$\left| \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \times \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \right| = x_0 \cdot y_1 - y_0 \cdot x_1$$

Given two vectors

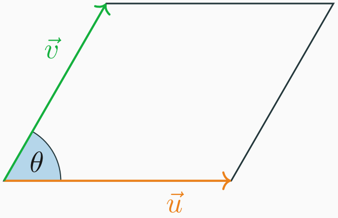
$$\vec{u} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \qquad \vec{v} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}$$

the cross product of $\vec{u}$ and $\vec{v}$ is defined as

$$\left| \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \times \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \right| = x_0 \cdot y_1 - y_0 \cdot x_1$$

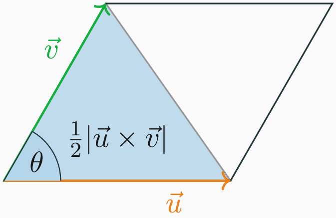
Which in geometric terms is

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



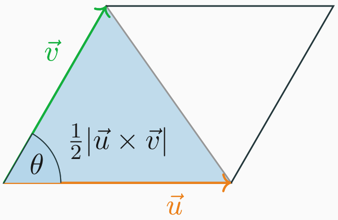
- Allows us to calculate the area of the triangle formed by $\vec{u}$ and $\vec{v}$.

$$\frac{|\vec{u} \times \vec{v}|}{2}$$



- Allows us to calculate the area of the triangle formed by $\vec{u}$ and $\vec{v}$.

$$\frac{|\vec{u} \times \vec{v}|}{2}$$

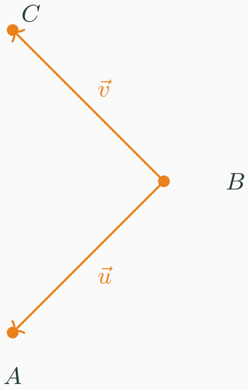


- Allows us to calculate the area of the triangle formed by $\vec{u}$ and $\vec{v}$.

$$\frac{|\vec{u} \times \vec{v}|}{2}$$

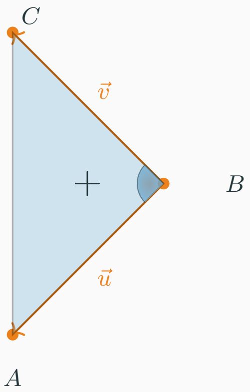
- And can tell us if the angle between $\vec{u}$ and $\vec{v}$ is positive or negative.

$$\begin{aligned} |\vec{u} \times \vec{v}| < 0 \quad &\text{iff} \quad \theta < \pi \\ |\vec{u} \times \vec{v}| = 0 \quad &\text{iff} \quad \theta = \pi \\ |\vec{u} \times \vec{v}| > 0 \quad &\text{iff} \quad \theta > \pi \end{aligned}$$



- Given three points A, B and C, we want to know if they form a counter-clockwise angle in that order.

$$A \to B \to C$$



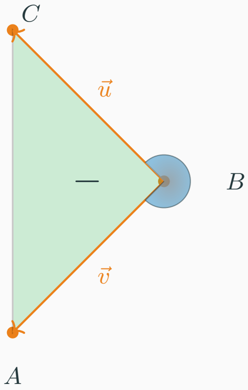
- Given three points A, B and C, we want to know if they form a counter-clockwise angle in that order.

$$A \to B \to C$$

- We can examine the cross product of and the area of the triangle formed by

$$\vec{u} = B - C \quad \vec{v} = B - A$$

$$\vec{u} \times \vec{v} > 0$$



- The points in the reverse order do not form a counter clockwise angle.

$$C \to B \to A$$

- In the reverse order the vectors swap places

$$\vec{u} = B - A \quad \vec{v} = B - C$$
$$\vec{u} \times \vec{v} < 0$$

- The points in the reverse order do not form a counter clockwise angle.

$$C \to B \to A$$

- In the reverse order the vectors swap places

$$\vec{u} = B - A \quad \vec{v} = B - C$$
$$\vec{u} \times \vec{v} < 0$$

- If the points A, B and C are on the same line, then the area will be 0.

```cpp
double cross(P(a), P(b)) {
    return real(a)*imag(b) - imag(a)*real(b);
}
double ccw(P(a), P(b), P(c)) {
    return cross(b - a, c - b);
}
bool collinear(P(a), P(b), P(c)) {
    return abs(ccw(a, b, c)) < EPS;
}
```

Very common task is to find the intersection of two lines or line segments.

Very common task is to find the intersection of two lines or line segments.

- Given a pair of points (x_0, y_0), (x_1, y_1), representing a line we want to start by obtaining the form $Ax + By = C$.

Very common task is to find the intersection of two lines or line segments.

- Given a pair of points (x_0, y_0), (x_1, y_1), representing a line we want to start by obtaining the form $Ax + By = C$.
- We can do so by setting

$$A = y_1 - y_0$$

$$B = x_0 - x_1$$

$$C = A \cdot x_0 + B \cdot y_1$$

Very common task is to find the intersection of two lines or line segments.

- Given a pair of points (x_0, y_0), (x_1, y_1), representing a line we want to start by obtaining the form $Ax + By = C$.
- We can do so by setting

$$
\begin{aligned}
A &= y_1 - y_0 \\
B &= x_0 - x_1 \\
C &= A \cdot x_0 + B \cdot y_1
\end{aligned}
$$

- If we have two lines given by such equations, we simply need to solve for the two unknowns, x and y.

For two lines

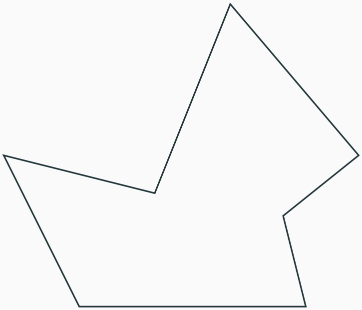
$$A_0x + B_0y = C_0$$
$$A_1x + B_1y = C_1$$

The intersection point is

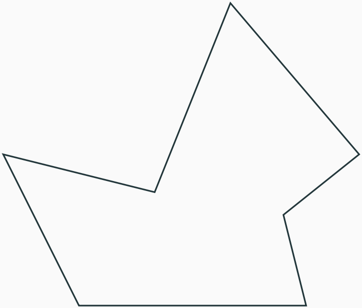
$$x = \frac{(B_1 \cdot C_0 - B_0 \cdot C_1)}{D}$$
$$y = \frac{(A_0 \cdot C_1 - A_1 \cdot C_0)}{D}$$

Where

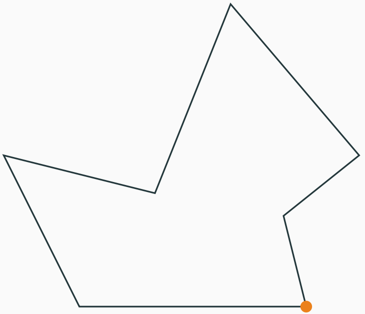
$$D = A_0 \cdot B_1 - A_1 \cdot B_0$$



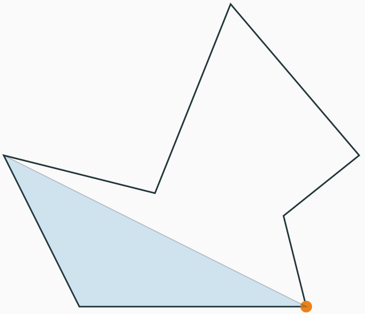
- Polygons are represented by a list of points in the order representing the edges.



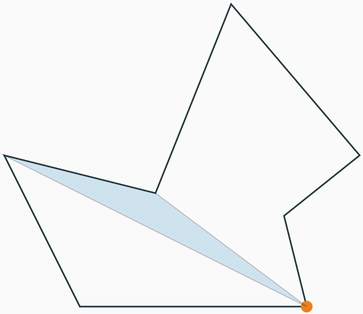
- Polygons are represented by a list of points in the order representing the edges.
- To calculate the area



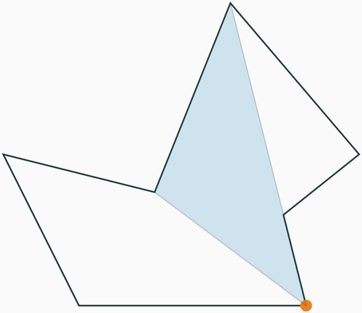
- Polygons are represented by a list of points in the order representing the edges.
- To calculate the area
    - We pick one starting point.



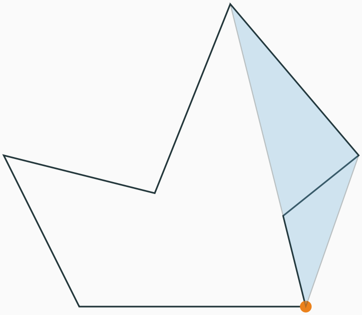
- Polygons are represented by a list of points in the order representing the edges.
- To calculate the area
  - We pick one starting point.
  - Go through all the other adjacent pair of points and sum the area of the triangulation.



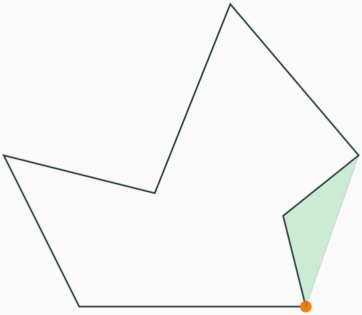
- Polygons are represented by a list of points in the order representing the edges.
- To calculate the area
  - We pick one starting point.
  - Go through all the other adjacent pair of points and sum the area of the triangulation.



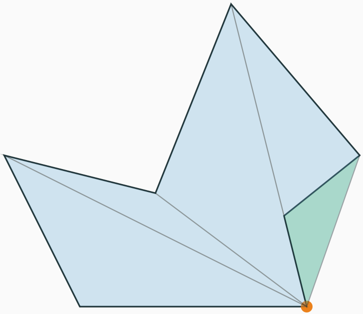
- Polygons are represented by a list of points in the order representing the edges.
- To calculate the area
  - We pick one starting point.
  - Go through all the other adjacent pair of points and sum the area of the triangulation.



- Polygons are represented by a list of points in the order representing the edges.
- To calculate the area
  - We pick one starting point.
  - Go through all the other adjacent pair of points and sum the area of the triangulation.



- Polygons are represented by a list of points in the order representing the edges.
- To calculate the area
  - We pick one starting point.
  - Go through all the other adjacent pair of points and sum the area of the triangulation.
  - Even if we sum up area outside the polygon, due to the cross product, it is subtracted later.

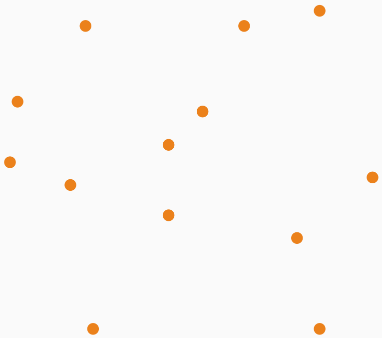


- Polygons are represented by a list of points in the order representing the edges.
- To calculate the area
    - We pick one starting point.
    - Go through all the other adjacent pair of points and sum the area of the triangulation.
    - Even if we sum up area outside the polygon, due to the cross product, it is subtracted later.

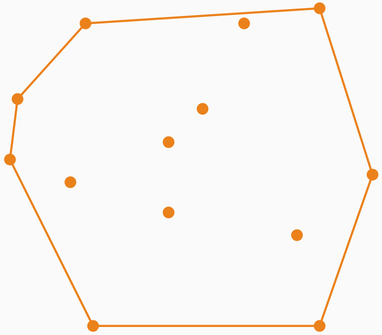
- Given a set of points, we want to find the convex hull of the points.

- Given a set of points, we want to find the convex hull of the points.
- The convex hull of points can be visualized as the shape formed by a rubber band around the set of points.

- Given a set of points, we want to find the convex hull of the points.
- The convex hull of points can be visualized as the shape formed by a rubber band around the set of points.



- Given a set of points, we want to find the convex hull of the points.
- The convex hull of points can be visualized as the shape formed by a rubber band around the set of points.



Graham scan:

Graham scan:

- Pick the point p_0 with the lowest y coordinate.

Graham scan:

- Pick the point p_0 with the lowest y coordinate.
- Sort all the points by polar angle with p_0.

Graham scan:

- Pick the point p_0 with the lowest y coordinate.
- Sort all the points by polar angle with p_0.
- Iterate through all the points

Graham scan:

- Pick the point p_0 with the lowest y coordinate.
- Sort all the points by polar angle with p_0.
- Iterate through all the points
- If the current point forms a clockwise angle with the last two points, remove last point from the convex set.

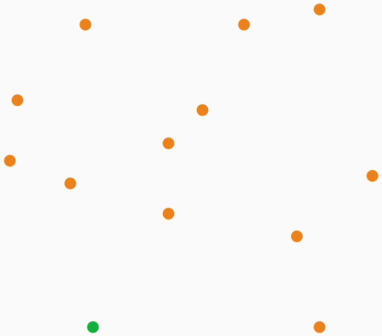
Graham scan:

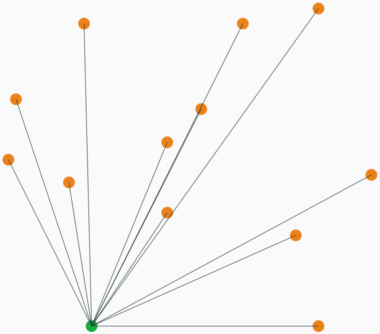
- Pick the point p_0 with the lowest y coordinate.
- Sort all the points by polar angle with p_0.
- Iterate through all the points
- If the current point forms a clockwise angle with the last two points, remove last point from the convex set.
- Otherwise, add the current point to the convex set.

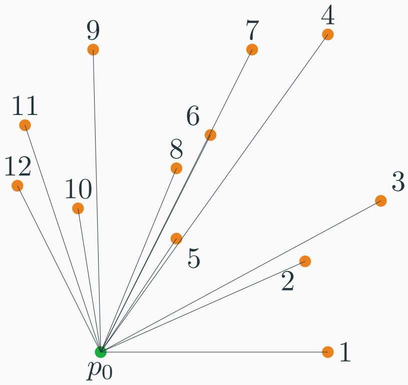
Graham scan:

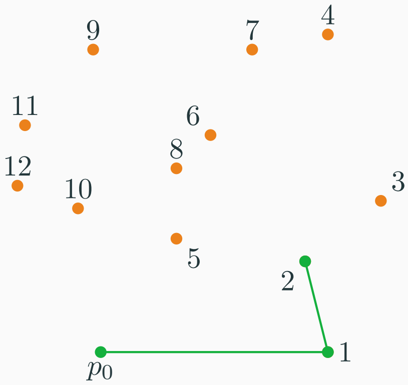
- Pick the point p_0 with the lowest y coordinate.
- Sort all the points by polar angle with p_0.
- Iterate through all the points
- If the current point forms a clockwise angle with the last two points, remove last point from the convex set.
- Otherwise, add the current point to the convex set.

Time complexity $O(N \log N)$.

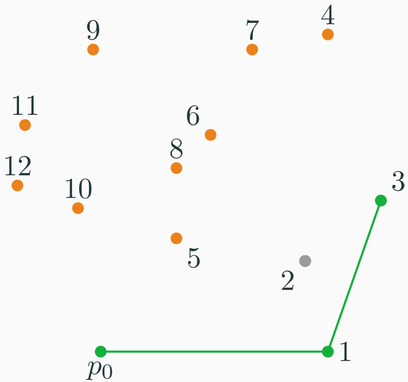


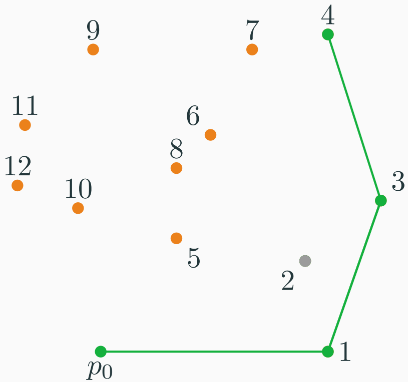



9
7
4
11
6
8
12
10
3
5
2
1
p_0

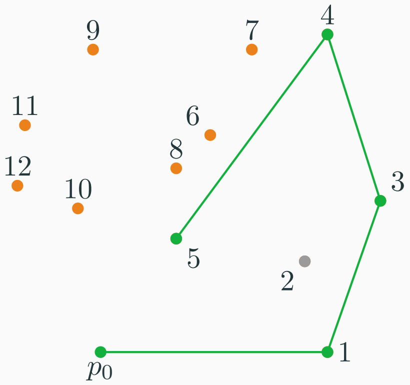


9
7
4
11
6
8
12
10
3
5
2
1
p_0

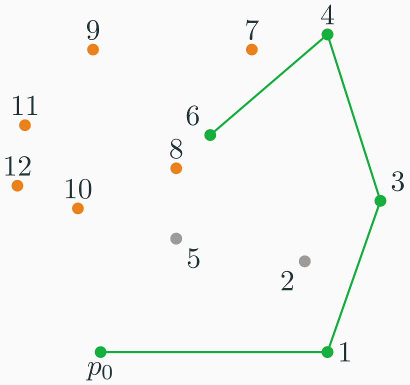

9
7
4
11
6
8
12
3
10
5
2
1
p_0



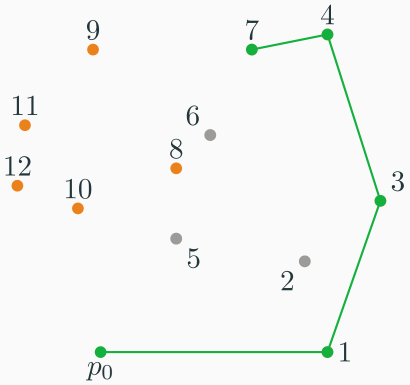
4
9
7
11
6
8
3
12
10
5
2
1
p_0



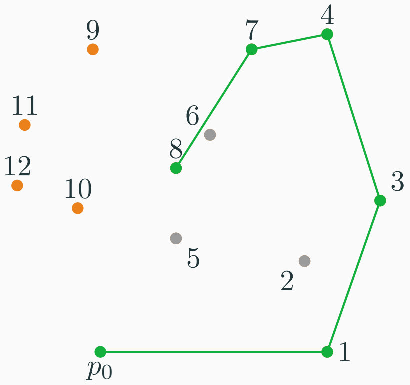
4
9
7
11
6
8
12
3
10
5
2
1
p_0

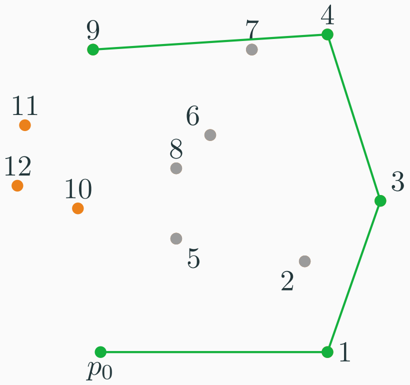


9
7
4
11
6
8
3
12
10
5
2
1
p_0

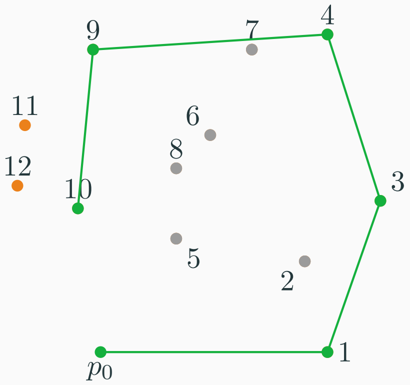


9
7
4
11
6
8
12
3
10
5
2
1
p_0

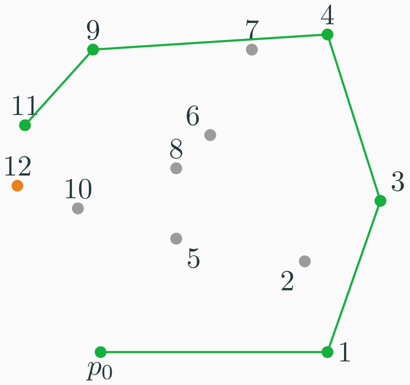

9
11
12
10
7
4
6
8
3
5
2
1
p_0



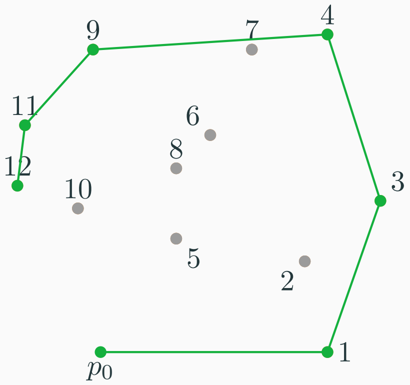
9
7
4
11
6
8
12
10
3
5
2
1
p_0



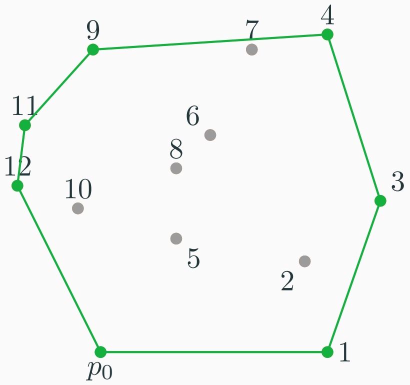
9
7
4
11
6
8
12
3
10
5
2
1
p_0



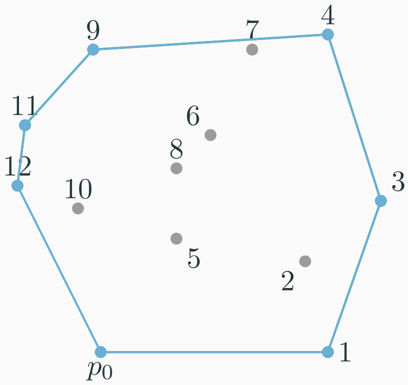
9
7
4
11
6
8
12
10
3
5
2
1
p_0



9
7
4
11
6
8
12
10
3
5
2
1
p_0



9
7
4
11
6
8
12
10
3
5
2
1
p_0


9
7
4
11
6
8
12
10
3
5
2
1
p_0

```
point hull[MAXN];
bool cmp(const point &a, const point &b) {
  return abs(real(a) - real(b)) > EPS ?
    real(a) < real(b) : imag(a) < imag(b); }
int convex_hull(vector<point> p) {
    int n = size(p), l = 0;
    sort(p.begin(), p.end(), cmp);
    for (int i = 0; i < n; i++) {
        if (i > 0 && p[i] == p[i - 1])
            continue;
        while (l >= 2 && ccw(hull[l - 2], hull[l - 1], p[i]) >= 0)
            l--;
        hull[l++] = p[i]; }
    int r = l;
    for (int i = n - 2; i >= 0; i--) {
        if (p[i] == p[i + 1])
            continue;
        while (r - l >= 1 && ccw(hull[r - 2], hull[r - 1], p[i]) >= 0)
            r--;
        hull[r++] = p[i]; }
    return l == 1 ? 1 : r - 1; }
```

Many other algorithms exist

Many other algorithms exist

- Gift wrapping aka Jarvis march.

Many other algorithms exist

- Gift wrapping aka Jarvis march.
- Quick hull, similar idea to quicksort.

Many other algorithms exist

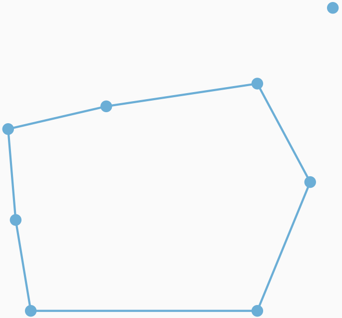
- Gift wrapping aka Jarvis march.
- Quick hull, similar idea to quicksort.
- Divide and conquer.

Many other algorithms exist

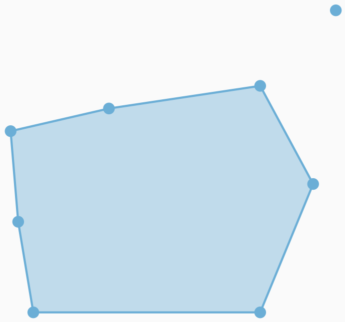
- Gift wrapping aka Jarvis march.
- Quick hull, similar idea to quicksort.
- Divide and conquer.

Some can be extended to three dimensions, or higher.

Simple algorithm to check if a point is in a convex polygon.

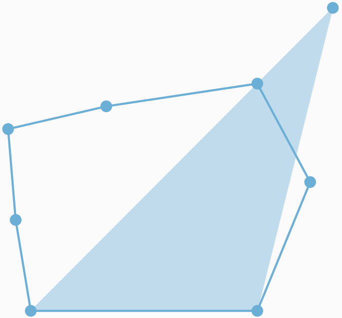


Simple algorithm to check if a point is in a convex polygon.



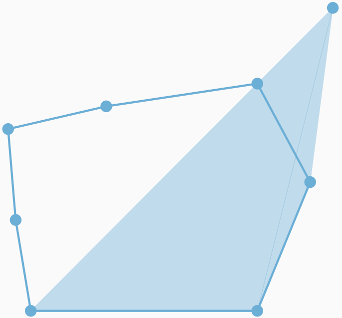
- We start by calculating the area of the polygon.

Simple algorithm to check if a point is in a convex polygon.



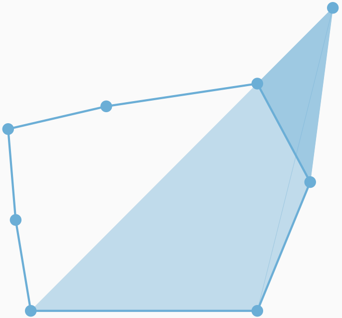
- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.

Simple algorithm to check if a point is in a convex polygon.



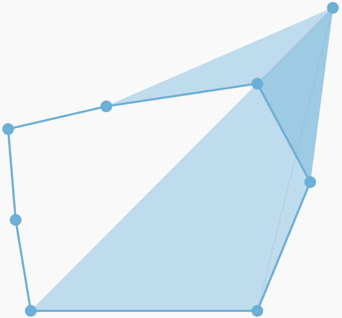
- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.

Simple algorithm to check if a point is in a convex polygon.



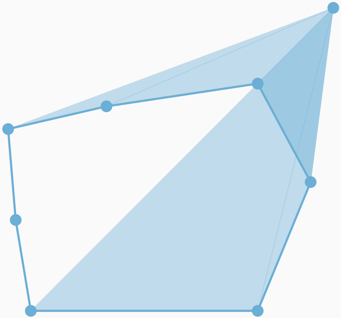
- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.

Simple algorithm to check if a point is in a convex polygon.



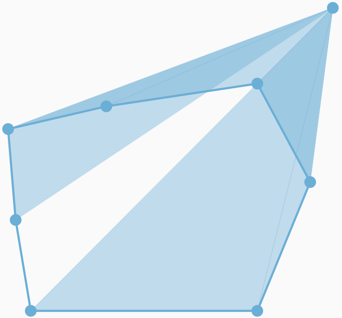
- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.

Simple algorithm to check if a point is in a convex polygon.



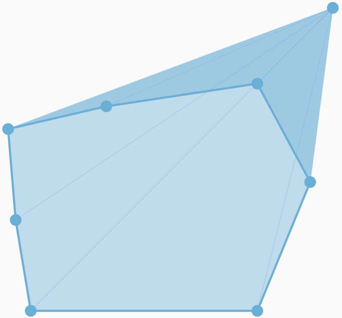
- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.

Simple algorithm to check if a point is in a convex polygon.



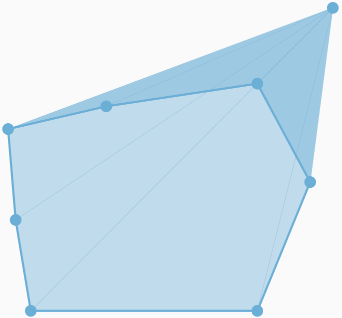
- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.

Simple algorithm to check if a point is in a convex polygon.



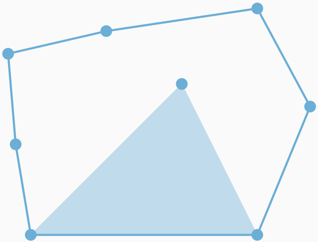
- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.

Simple algorithm to check if a point is in a convex polygon.



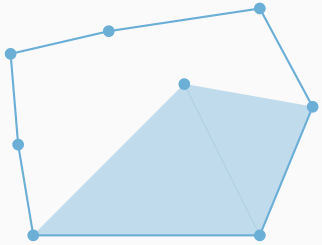
- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.
- The total area of the triangles is equal to the area of the polygon iff the point is inside the polygon.

Simple algorithm to check if a point is in a convex polygon.



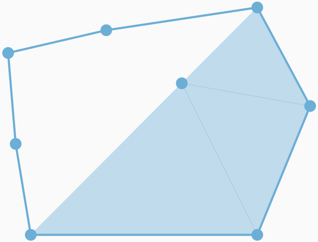
- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.
- The total area of the triangles is equal to the area of the polygon iff the point is inside the polygon.

Simple algorithm to check if a point is in a convex polygon.



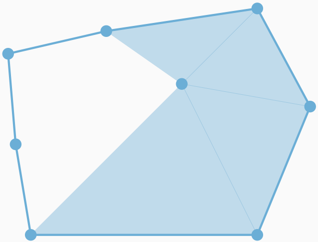
- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.
- The total area of the triangles is equal to the area of the polygon iff the point is inside the polygon.

Simple algorithm to check if a point is in a convex polygon.



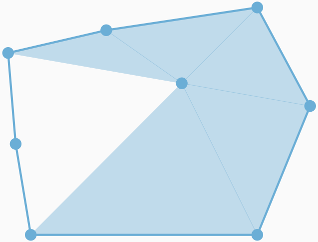
- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.
- The total area of the triangles is equal to the area of the polygon iff the point is inside the polygon.

Simple algorithm to check if a point is in a convex polygon.



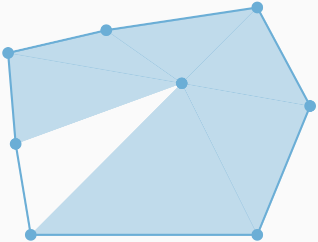
- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.
- The total area of the triangles is equal to the area of the polygon iff the point is inside the polygon.

Simple algorithm to check if a point is in a convex polygon.



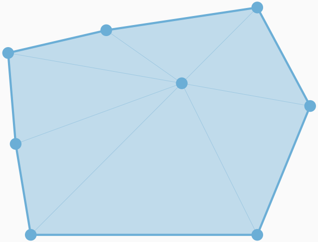
- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.
- The total area of the triangles is equal to the area of the polygon iff the point is inside the polygon.

Simple algorithm to check if a point is in a convex polygon.



- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.
- The total area of the triangles is equal to the area of the polygon iff the point is inside the polygon.

Simple algorithm to check if a point is in a convex polygon.



- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.
- The total area of the triangles is equal to the area of the polygon iff the point is inside the polygon.

How about non convex polygon?

How about non convex polygon?

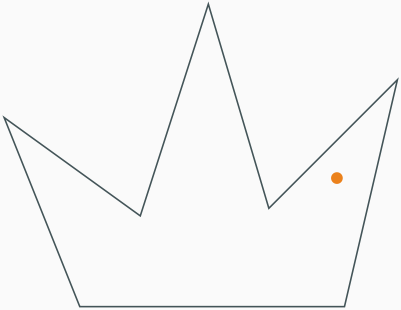
- The *even-odd rule* algorithm.

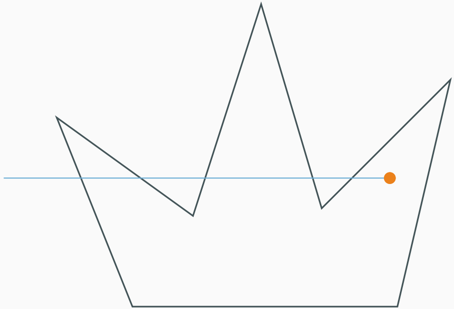
How about non convex polygon?

- The *even-odd rule* algorithm.
- We examine a ray passing through the polygon to the point.

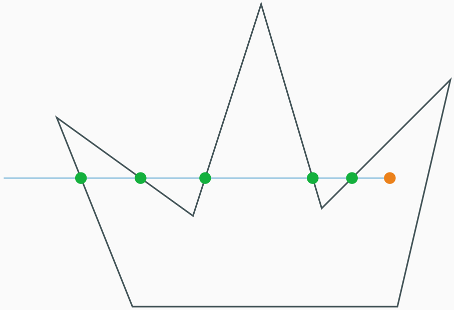
How about non convex polygon?

- The *even-odd rule* algorithm.
- We examine a ray passing through the polygon to the point.
- If the ray crosses the boundary of the polygon, then it alternately goes from outside to inside, and outside to inside.

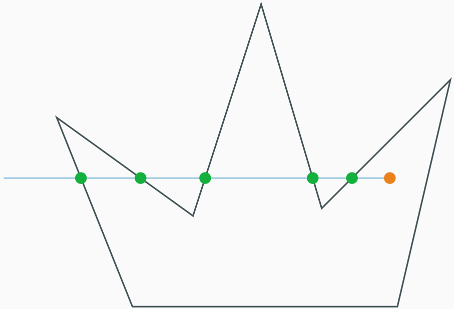




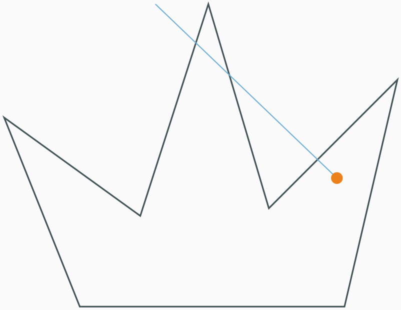
- Ray from the outside of the polygon to the point.



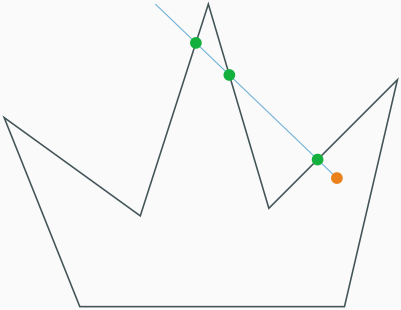
- Ray from the outside of the polygon to the point.
- Count the number of intersection points.



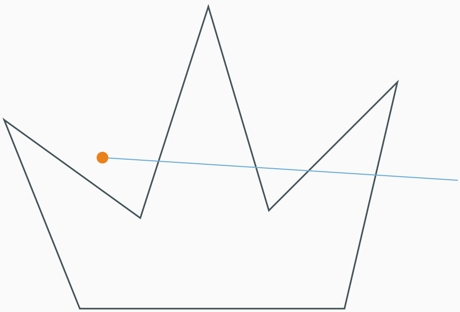
- Ray from the outside of the polygon to the point.
- Count the number of intersection points.
- If odd, then the point is inside the polygon.
- If even, then the point is outside the polygon.



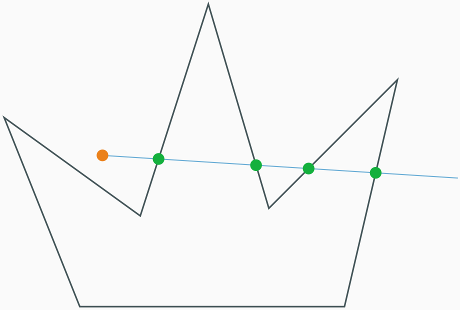
- Ray from the outside of the polygon to the point.
- Count the number of intersection points.
- If odd, then the point is inside the polygon.
- If even, then the point is outside the polygon.
- Does not matter which ray we pick.



- Ray from the outside of the polygon to the point.
- Count the number of intersection points.
- If odd, then the point is inside the polygon.
- If even, then the point is outside the polygon.
- Does not matter which ray we pick.



- Ray from the outside of the polygon to the point.
- Count the number of intersection points.
- If odd, then the point is inside the polygon.
- If even, then the point is outside the polygon.
- Does not matter which ray we pick.



- Ray from the outside of the polygon to the point.
- Count the number of intersection points.
- If odd, then the point is inside the polygon.
- If even, then the point is outside the polygon.
- Does not matter which ray we pick.

An algorithm

- Computational geometry has a lot of impressive and technical algorithms.
- The most famous one is probably Delaunay triangulation.
- But that one is a bit too hard for this course, so we will instead look at the classical closest point algorithm.
- We are given n points in the plan, find the pair of points that are closest to one another.
- We can clearly solve this in $\mathcal{O}(n^2)$ time, but can we do better?

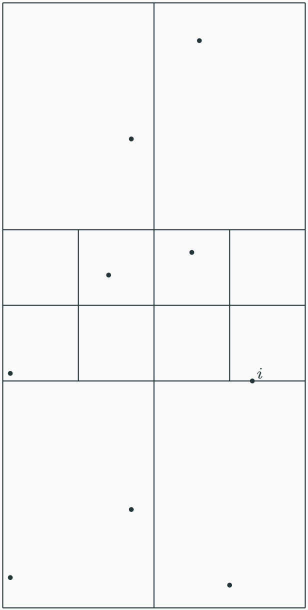
Divide and conquer

- We sort the points by x-coordinate and split the list in half.
- Let x_0 be such that it's between the coordinates of the left and right halves.
- Start by solving each half recursively.
- We now have to find if there's some pair with one point in each half that does better.
- We can't simply try all pairs, that's too slow. Suppose the smallest distance we found recursively was d.
- Then we can ignore all points with x-coordinte outside $[x_0 - d, x_0 + d]$.
- Sort the points inside of this interval by their y-coordinate.
- The big trick is now that we only need to consider a few neighbours for each point.

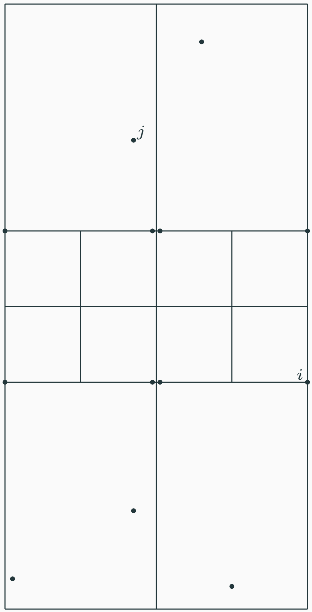
Neighbours

- Divide the area above x_i into 8 squares, each with side length $d/2$.
- If he distance between all points in each half is at least d, then we can have at most each point per square.
- All points outside these squares are at a distance of at least d from x_i, so we can ignore them.
- Thus we only need to look at the distance from x_i to x_j when $j - i \leq 7$.

Diagram



Diagram



Complexity

- Each recursive call is $\mathcal{O}(n \log(n))$.
- Thus by the master theorem the total complexity is $\mathcal{O}(n \log^2(n))$.
- If we sort the y values as we go using mergesort, we can actually do each call in $\mathcal{O}(n)$.
- This way the complexity is actually $\mathcal{O}(n \log(n))$.