Geometria computazionale

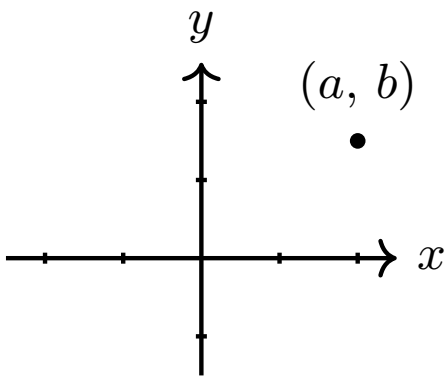
Alessandro Bortolin

Volterra, 19 maggio 2023

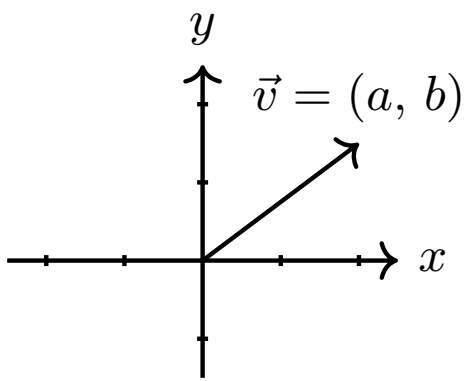
Punti

Punti

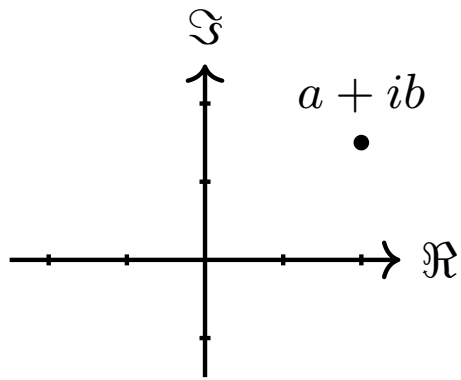
Cos'è un punto?



Una coppia di coordinate?



Un vettore?



Un numero complesso?

Rappresentazione dei punti

Come rappresentare un punto?

Rappresentazione mediante struct

```
1   struct pt {
2       ll x, y;
3       pt operator+(pt p) { ... }
4       pt operator-(pt p) { ... }
5       pt operator*(ll d) { ... }
6       pt operator/(ll d) { ... }
7       bool operator==(pt p) { ... }
8       bool operator!=(pt p) { ... }
9   };
```

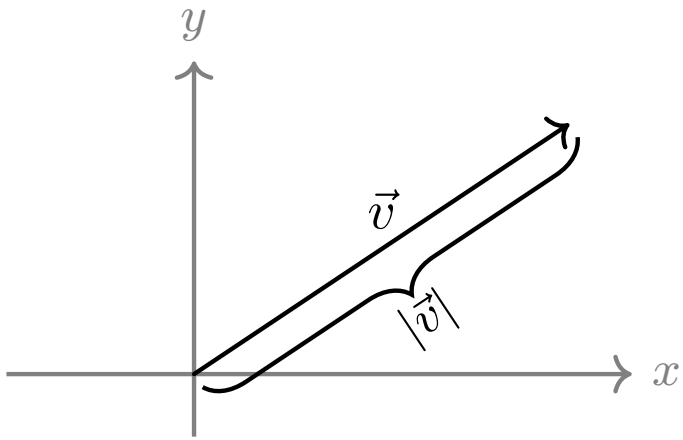
Rappresentazione mediante numeri complessi per persone prigre

```
1   #include <complex>
2   typedef std::complex<ll> pt;
3   #define x real()
4   #define y imag()
```

Norma di un vettore

Implementazione

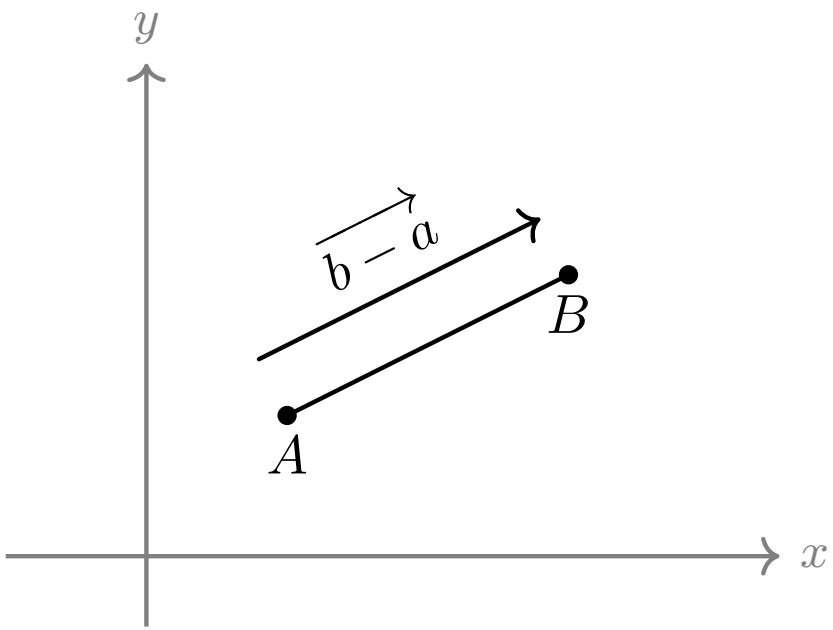
```
1   ll norm(pt a) {
2       return a.x * a.x + a.y * a.y;
3   }
4
5   double abs(pt a) {
6       return std::sqrt(norm(a));
7   }
8   // oppure
9   double abs(pt a) {
10      // Previene l'overflow
11      return std::hypot(a.x, a.y);
12  }
13
14  pt unit(pt a) {
15      if (a == pt{0, 0}) return a;
16      return a / abs(a); // Richiede i float
17  }
```



Distanza tra punti

Implementazione

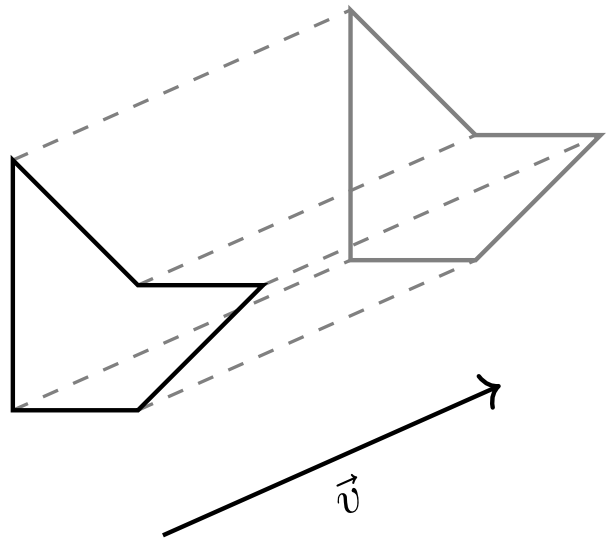
```
1  double dist(pt a, pt b) {
2      return abs(b - a);
3  }
```



Traslazione

Implementazione

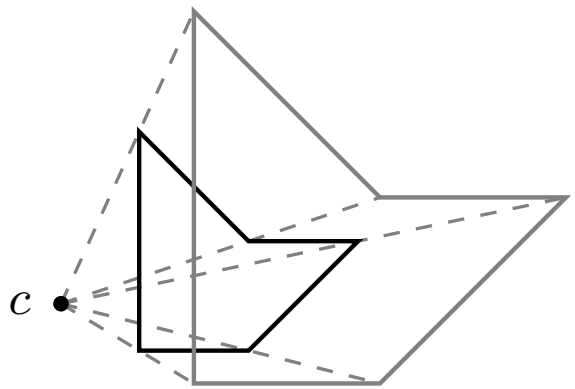
```
1  pt translate(pt p, pt v) {
2      return p + v;
3  }
4  pt translate(pt p, pt dir, double dist) {
5      return p + unit(dir) * dist;
6  }
```

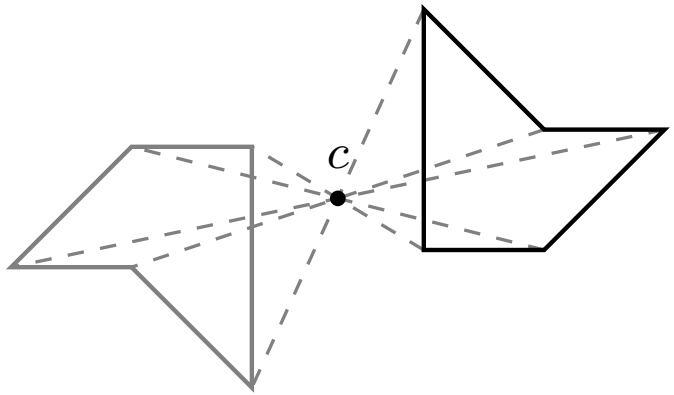


Ridimensionare rispetto a un punto (omotetia)

Implementazione

```
1  pt scale(pt p, ll factor, pt c) {
2      return c + (p - c) * factor;
3  }
4  pt reflect(pt p, pt c) {
5      return scale(p, -1, c);
6  }
```

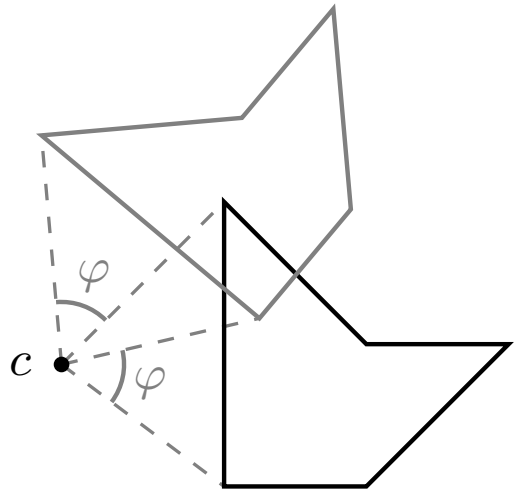




Rotazione

Implementazione

```
1   pt rotate(pt p, double a) {
2       // Richiede i float
3       return {
4           p.x * std::cos(a) - p.y * std::sin(a),
5           p.x * std::sin(a) + p.y * std::cos(a)
6       };
7   }
8   pt rotate(pt p, double a, pt c) {
9       return c + rotate(p - c, a);
10  }
11  pt perp(pt p) {
12      return {-p.y, p.x};
13  }
```



Trasformazioni lineari

Traslazione

$$\begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + dx \\ y + dy \\ 1 \end{bmatrix}$$

Ridimensionare

$$\begin{bmatrix} c & 0 & 0 \\ 0 & c & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} cx \\ cy \\ 1 \end{bmatrix}$$

Rotazione

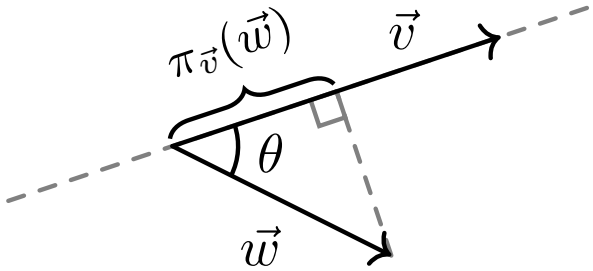
$$\begin{bmatrix} \cos(a) & \sin(a) & 0 \\ -\sin(a) & \cos(a) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x\cos(a) - y\sin(a) \\ x\sin(a) + y\cos(a) \\ 1 \end{bmatrix}$$

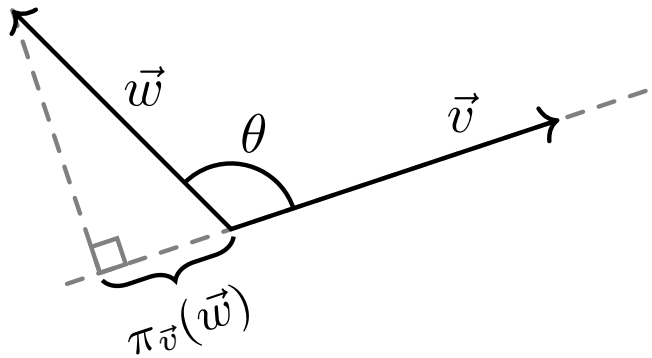
Angoli

Dot product (prodotto scalare)

Implementazione

```
1  ll dot(pt v, pt w) {
2      return v.x * w.x + v.y * w.y;
3  }
```



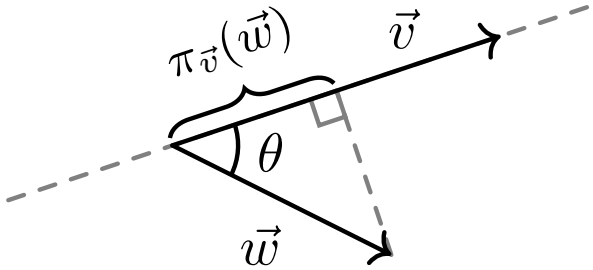


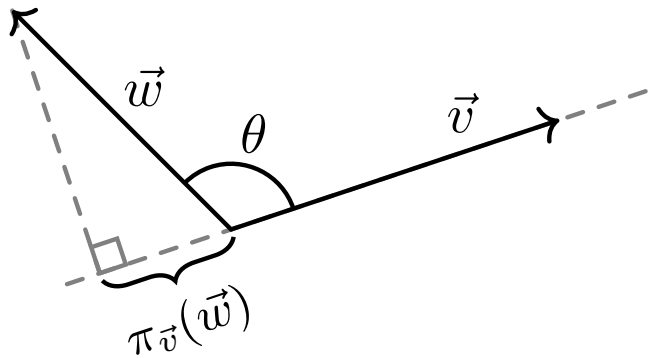
$$\vec{v} \cdot \vec{w} = |\vec{v}| \cdot |\vec{w}| \cos\theta = |\vec{v}| \pi_{\vec{v}}(\vec{w})$$

Dot product (prodotto scalare)

Proprietà

- $\vec{v} \cdot \vec{w} = 0$ se $\vec{v}$ e $\vec{w}$ sono perpendicolari: $\theta = \frac{\pi}{2}$.
- $\vec{v} \cdot \vec{w} > 0$ se l'angolo tra $\vec{v}$ e $\vec{w}$ è acuto: $\theta < \frac{\pi}{2}$.
- $\vec{v} \cdot \vec{w} < 0$ se l'angolo tra $\vec{v}$ e $\vec{w}$ è ottuso: $\theta > \frac{\pi}{2}$.
- $\vec{v} \cdot \vec{w} = \vec{w} \cdot \vec{v}$.



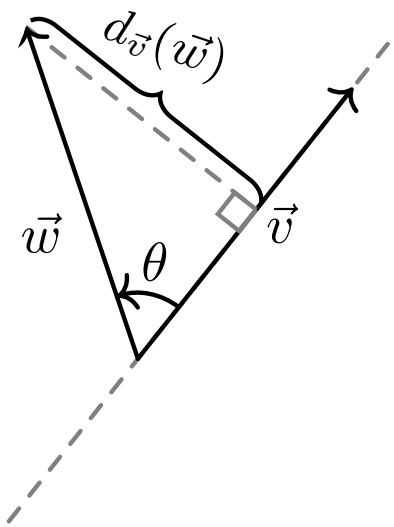


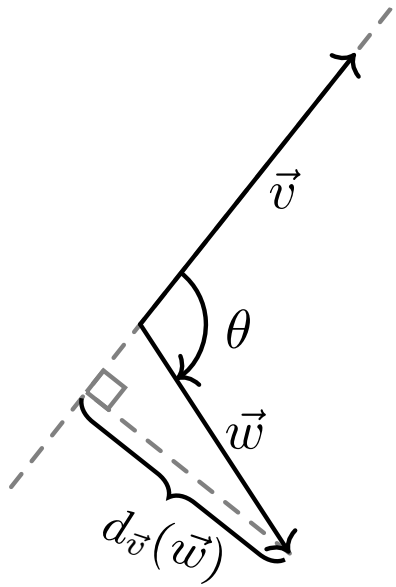
$$\vec{v} \cdot \vec{w} = |\vec{v}| \cdot |\vec{w}| \cos\theta = |\vec{v}| \pi_{\vec{v}}(\vec{w})$$

Cross product (prodotto vettoriale)

Implementazione

```
1  ll cross(pt v, pt w) {
2      return v.x * w.y - v.y * w.x;
3  }
```



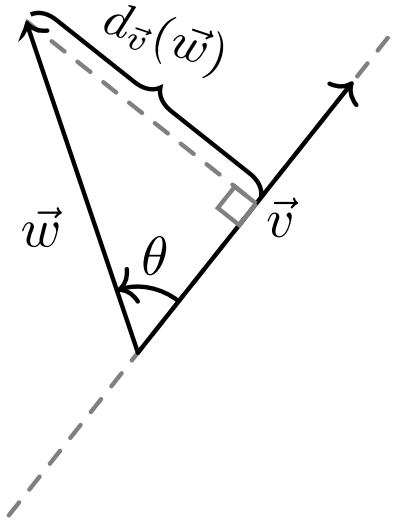


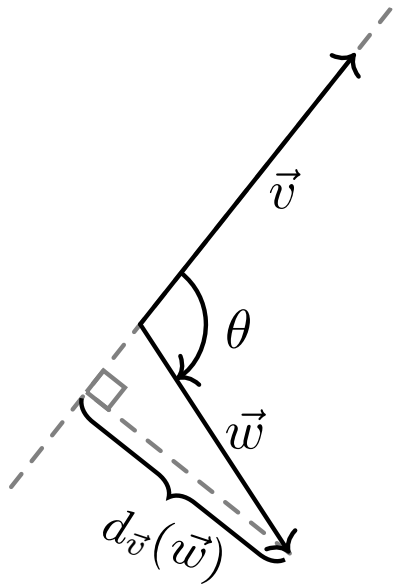
$$\vec{v} \times \vec{w} = |\vec{v}| \cdot |\vec{w}| \sin\theta = |\vec{v}| d_{\vec{v}}(\vec{w})$$

Cross product (prodotto vettoriale)

Proprietà

- $\vec{v} \times \vec{w} = 0$ se $\vec{v}$ e $\vec{w}$ paralleli.
- $\vec{v} \times \vec{w} > 0$ se $\vec{w}$ è a "sinistra" di $\vec{v}$.
- $\vec{v} \times \vec{w} < 0$ se $\vec{w}$ è a "destra" di $\vec{v}$.
- $\vec{v} \times \vec{w} = -\vec{w} \times \vec{v}$, l'ordine è importante!

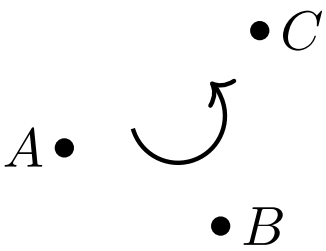




Orientare i punti

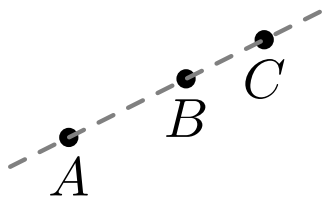
Implementazione

```
1  ll orient(pt a, pt b, pt c) {
2      return cross(b - a, c - a);
3  }
```



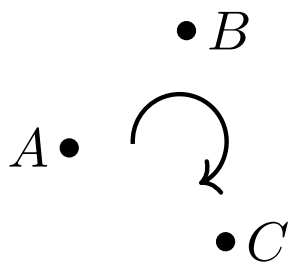
Senso antiorario.

`orient(A, B, C) > 0`



Punti allineati.

`orient(A, B, C) = 0`



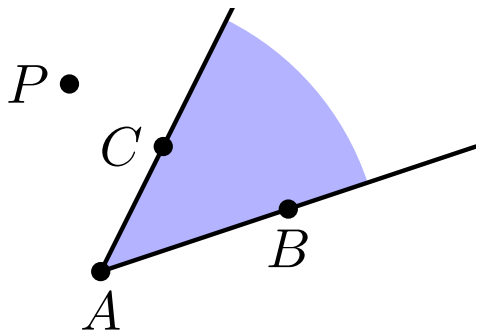
Senso orario.

`orient(A, B, C) < 0`

Punti interni a un angolo

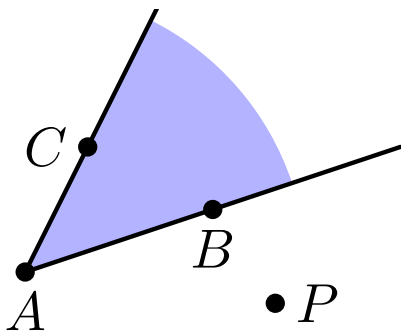
Implementazione

```
1  bool inAngle(pt a, pt b, pt c, pt p) {
2      return orient(a, b, p) >= 0 && orient(a, c, p) <= 0;
3  }
```



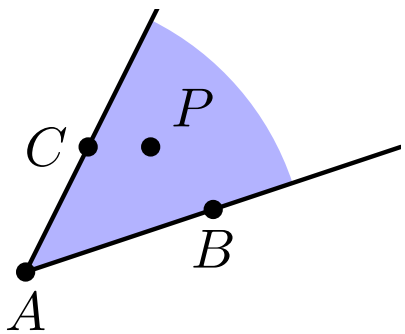
NO

orient(A, B, P) ≥ 0
orient(A, C, P) > 0



NO

orient(A, B, P) < 0
orient(A, C, P) ≤ 0



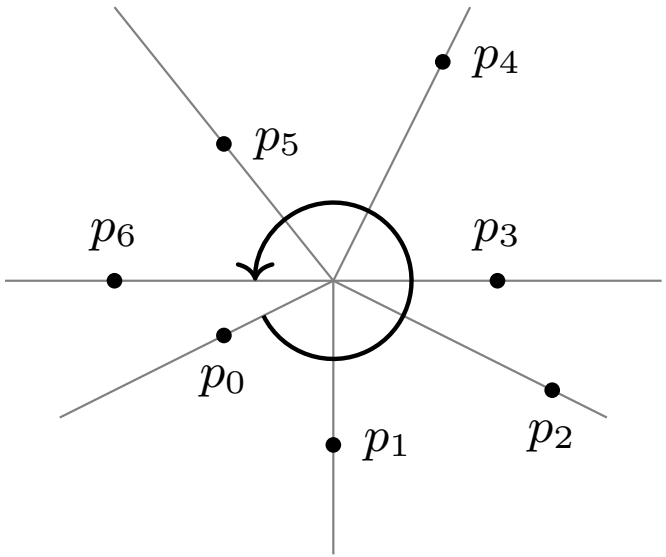
SÍ

orient(A, B, P) ≥ 0
orient(A, C, P) ≤ 0

Polar sort

Una versione semplice

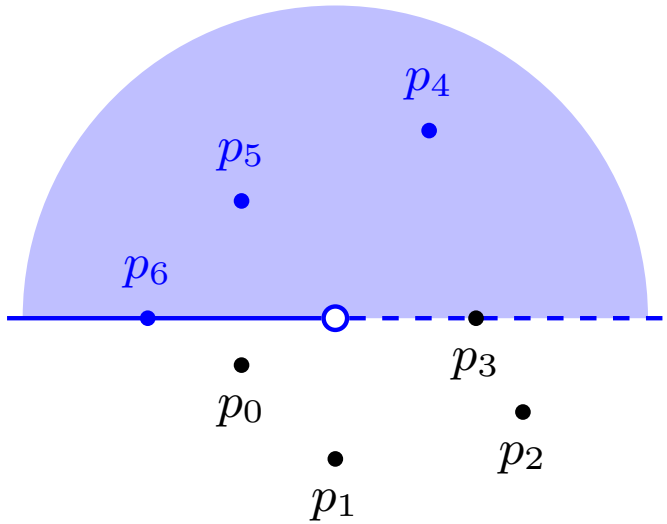
```
1  void polarSort(vector<pt>& P) {
2      sort(P.begin(), P.end(), [](pt v, pt w) {
3          return atan2l(v.y, v.x) < atan2l(w.y, w.x);
4      });
5  }
```



Polar sort

Una versione migliore

```
1  bool half(pt p) {
2      return p.y > 0 || (p.y == 0 && p.x < 0);
3  }
4  void polarSort(vector<pt>& P) {
5      sort(P.begin(), P.end(), [](pt v, pt w) {
6          return pair(half(v), 0ll) < pair(half(w), cross(v,w));
7      });
8  }
```

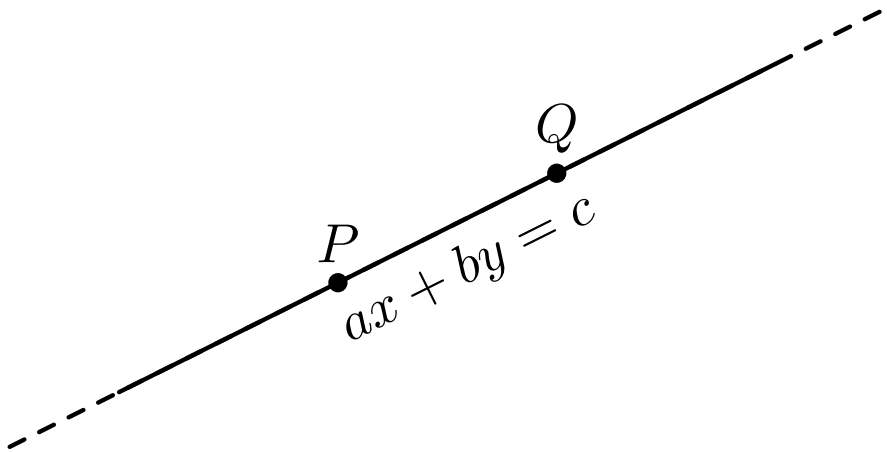


Rette e segmenti

Rappresentazione delle rette

Come rappresentare una retta?

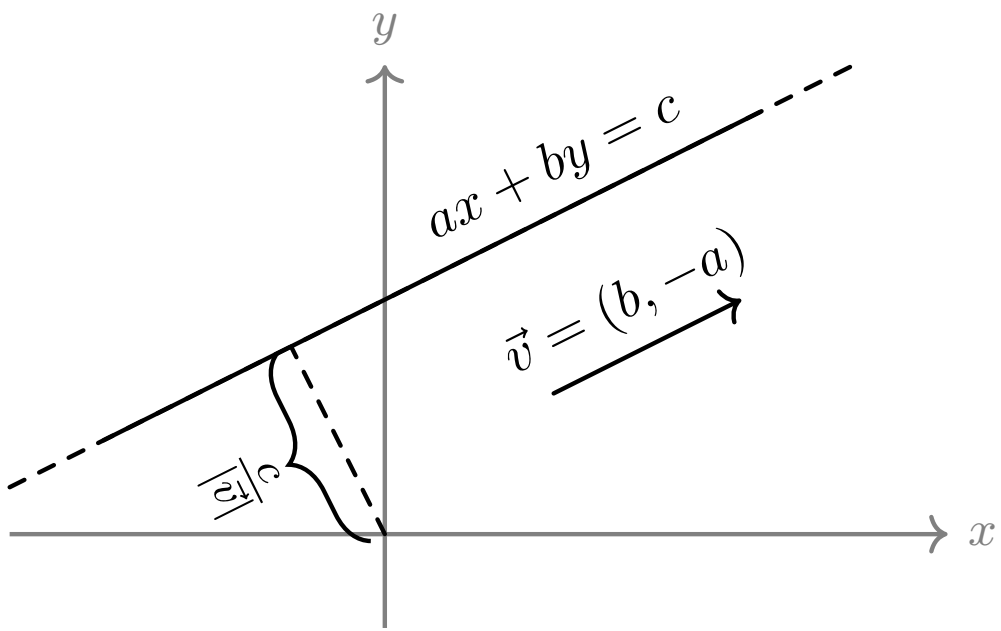
- Forma implicita: $ax + by = c$. — Non particolarmente comodo.
- Forma esplicita: $y = mx + q$. — Comodo per il convex hull trick.
- Dati due punti: (P, Q). — Comodo per rappresentare i segmenti.
- Data la direzione e l'offset: $(\vec{v}, c)$. — Comodo per effettuare operazioni.



Rappresentazione delle rette

Implementazione

```
1  struct line {
2      pt v; ll c;
3      line(pt v, ll c) : v(v), c(c) {}                  // (v, c)
4      line(ll a, ll b, ll c) : v({b, -a}), c(c) {}      // ax + by = c
5  };
```

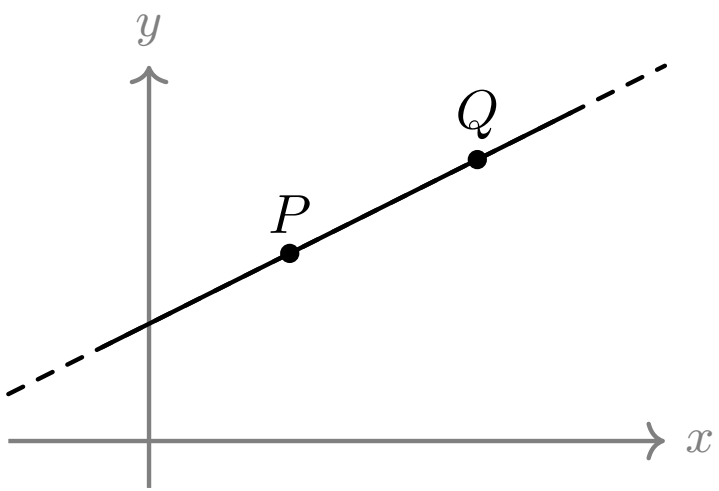


Retta per due punti

Definizione

$$\vec{v} = \vec{q} - \vec{p}$$

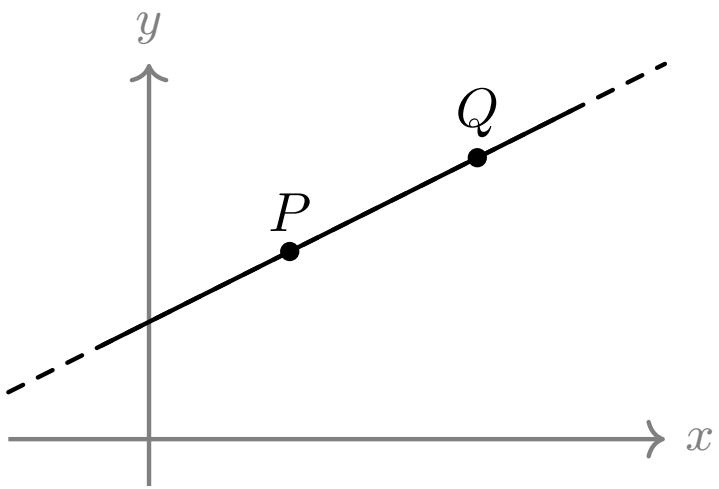
$$\overrightarrow{(b, -a)} \times \overrightarrow{(x, y)} = ax + by = c \quad \Longrightarrow \quad c = \vec{v} \times \vec{p}$$



Retta per due punti

Implementazione

```
1  struct line {
2      ...
3      line(pt p, pt q) : v(q - p), c(cross(v, p)) {}
4  };
```



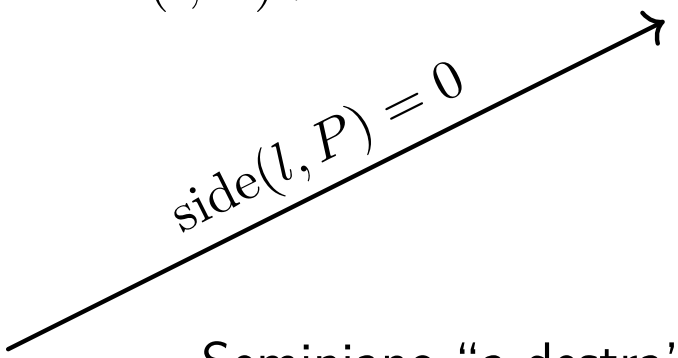
Punto-retta

Implementazione

```
1  ll side(line l, pt p) {
2      return cross(l.v, p) - l.c;
3  }
```

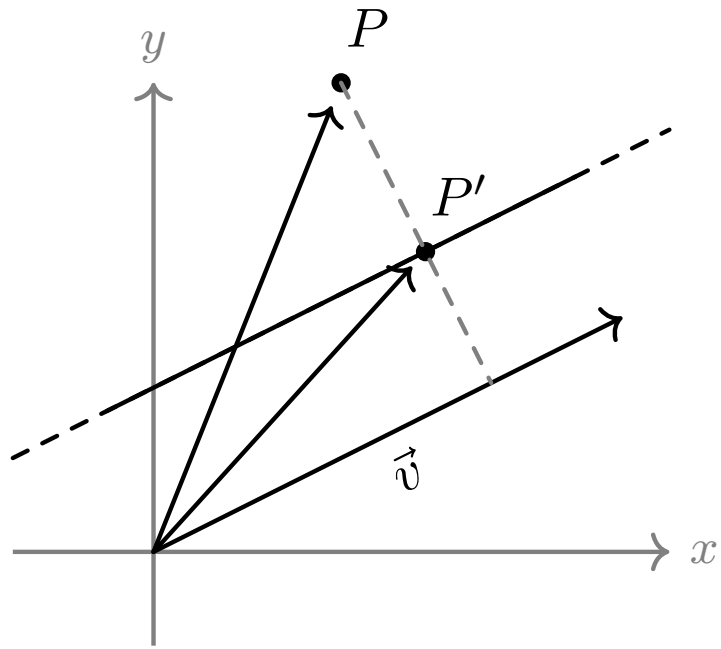
Semipiano "a sinistra"

$$\text{side}(l, P) > 0$$



Semipiano "a destra"

$$\text{side}(l, P) < 0$$

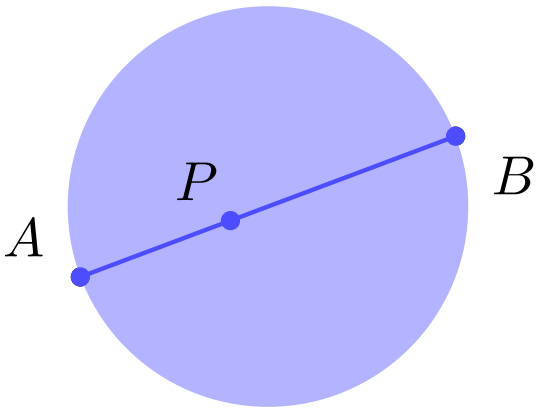


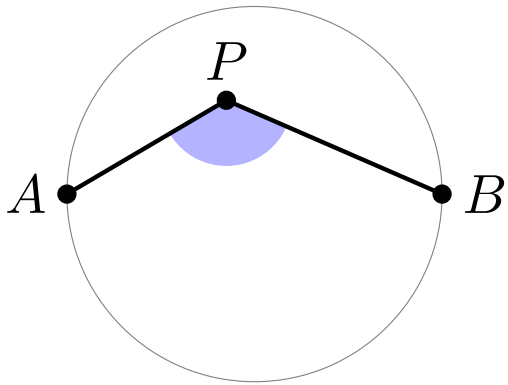
$$\vec{v} \times \vec{p} \overset{?}{\lessgtr} \vec{v} \times \vec{p'}$$

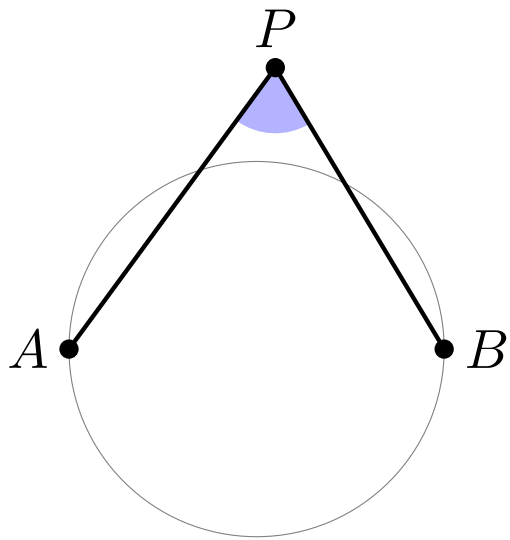
Punto-segmento

Implementazione

```
1  bool onSegment(pt a, pt b, pt p) {
2      return orient(a, b, p) == 0 && dot(a - p, b - p) <= 0;
3  }
```



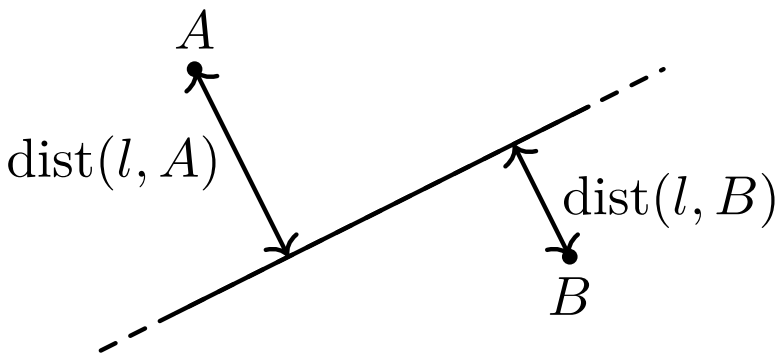


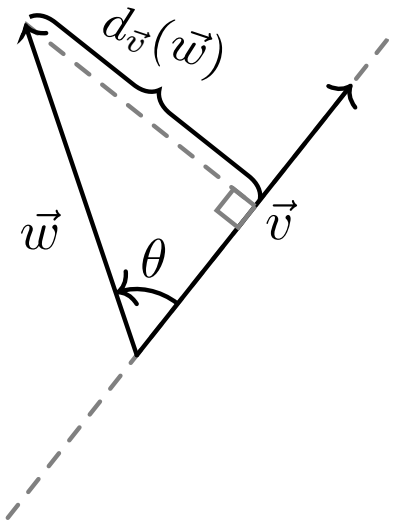


Distanza punto-retta

Implementazione

```
1  double dist(line l, pt p) {
2      return std::abs(side(l, p)) / abs(l.v);
3  }
```



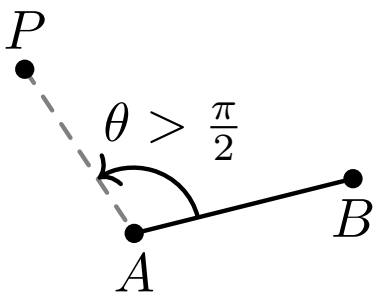


$$\vec{v} \times \vec{w} = |\vec{v}| d_{\vec{v}}(\vec{w})$$

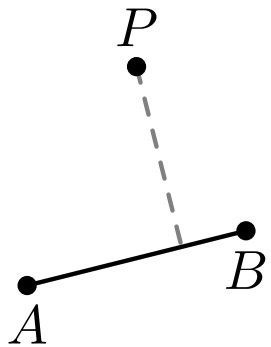
Distanza punto-segmento

Implementazione

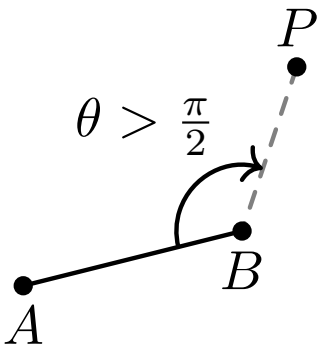
```
1  double dist(pt a, pt b, pt p) {
2      if (dot(p - a, b - a) > 0 && dot(p - b, a - b) > 0) {
3          return dist({a, b}, p);
4      } else {
5          return std::min(dist(p, a), dist(p, b));
6      }
7  }
```



Più vicino a A.



Più vicino alla proiezione.

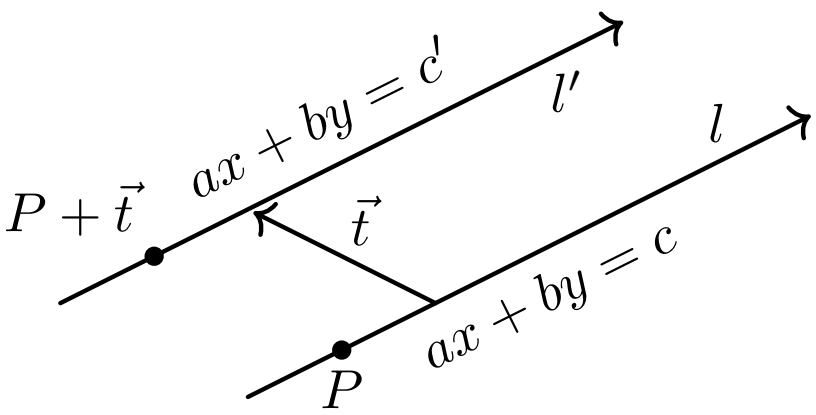


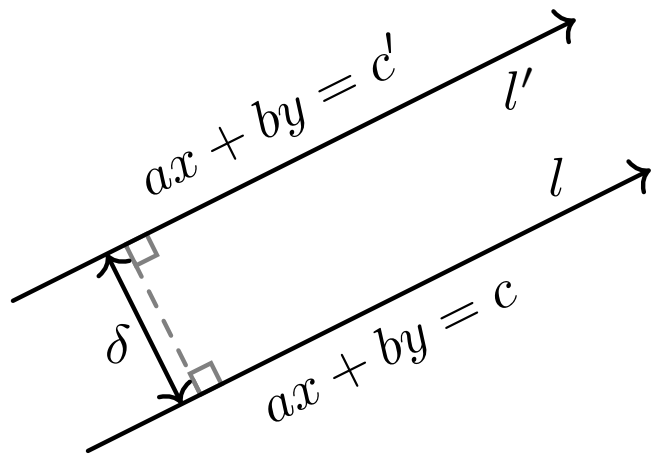
Più vicino a B.

Traslazione

Implementazione

```
1  line translate(line l, pt t) {
2      return {l.v, l.c + cross(l.v, t)};
3  }
4  line translate(line l, double dist) {
5      return {l.v, l.c + dist * abs(l.v)}; // Richiede i float
6  }
```

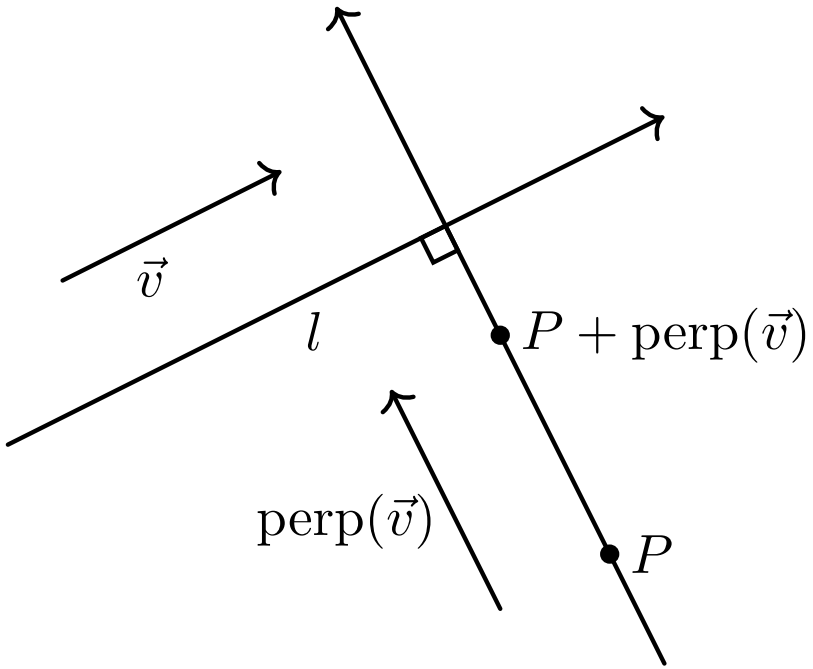




Retta parallela e perpendicolare

Implementazione

```
1  line paral(line l, pt p) {
2      return {p, p + l.v};
3  }
4  line perp(line l, pt p) {
5      return {p, p + perp(l.v)};
6  }
```



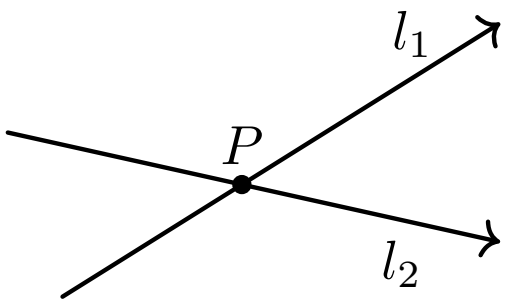
Altre operazioni

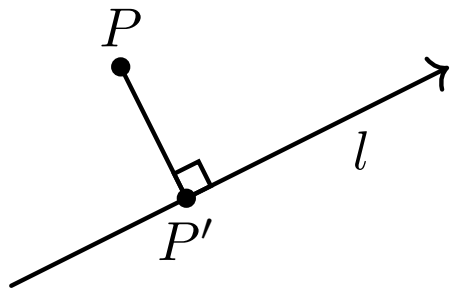
Intersezione tra due rette

```
1  // Richiede i float
2  std::optional<pt> intersect(line l1, line l2) {
3      double d = cross(l1.v, l2.v);
4      if (std::abs(d) < EPS) return {};
5      return (l2.v * l1.c - l1.v * l2.c) / d;
6  }
```

Proiezione di un punto su una retta

```
1  // Richiede i float
2  pt project(line l, pt p) {
3      return p - perp(l.v) * side(l, p) / norm(l.v);
4  }
```



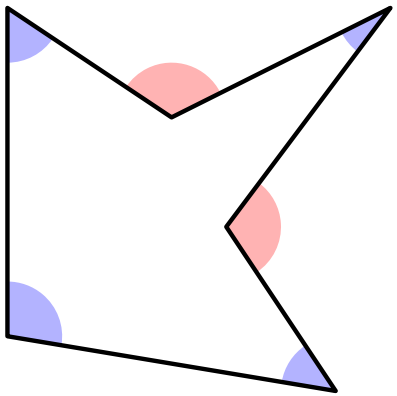


Poligoni

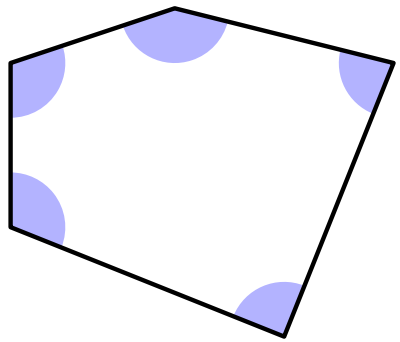
Convessità

Implementazione

```
1  bool isConvex(int N, vector<pt> P) {
2      bool hasPos = false, hasNeg = false;
3      for (int i = 0; i < N; i++) {
4          ll o = orient(P[i], P[(i + 1) % N], p[(i + 2) % N]);
5          hasPos += (o > 0);
6          hasNeg += (o < 0);
7      }
8      return !(hasPos && hasNeg);
9  }
```



Non convesso

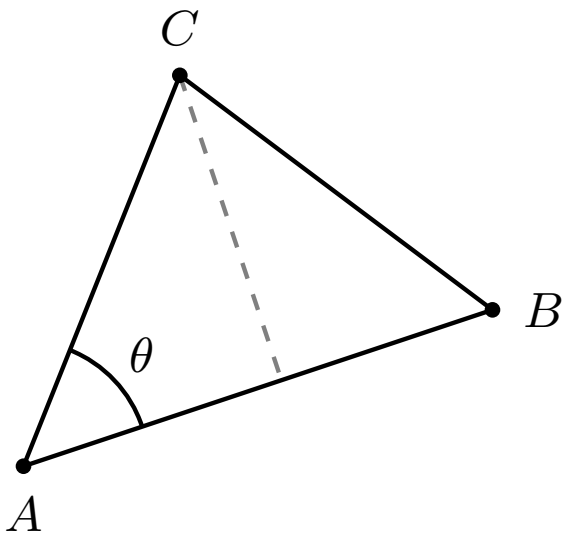


Convesso

Area di un triangolo

Implementazione

```
1  double areaTriangle(pt a, pt b, pt c) {
2      return std::abs(cross(b - a, c - a)) / 2.0;
3  }
```

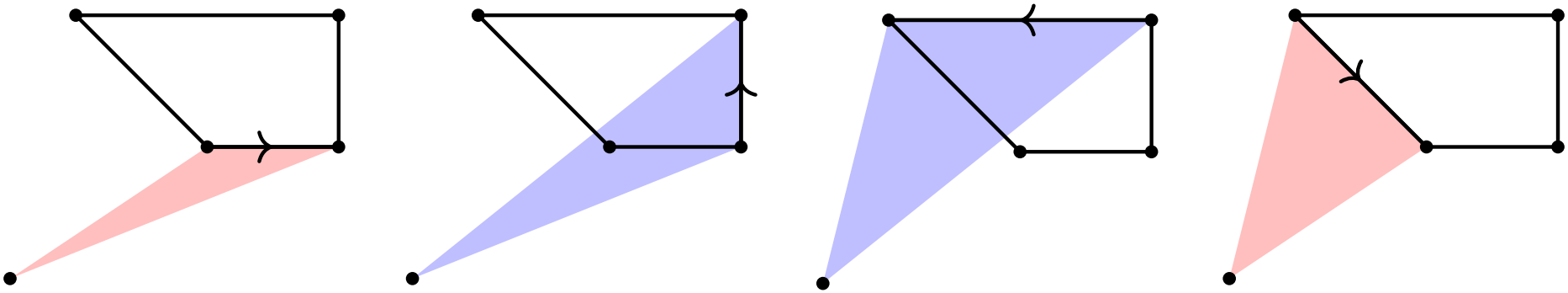


$$|\overset{\triangle}{ABC}| = \frac{1}{2}|AB||AC|\sin\theta = \frac{1}{2}\left|\overrightarrow{AB} \times \overrightarrow{AC}\right|$$

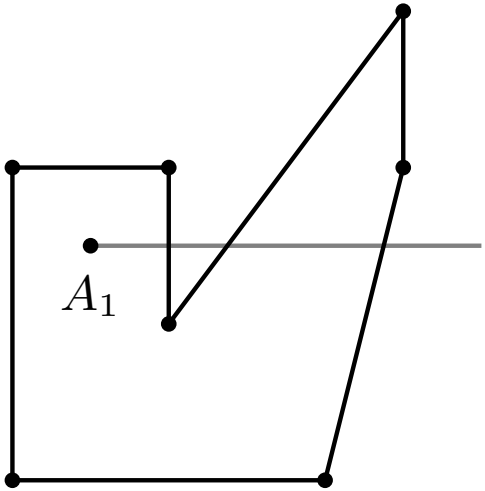
Area di un poligono

Implementazione

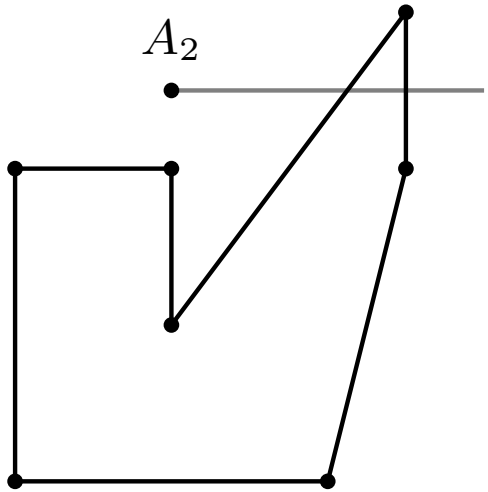
```
1  double area(vector<pt> P) {
2      ll res = 0;
3      for (size_t i = 0; i < P.size(); i++) {
4          res += cross(P[i], P[(i + 1) % P.size()]);
5      }
6      return std::abs(res) / 2.0;
7  }
```



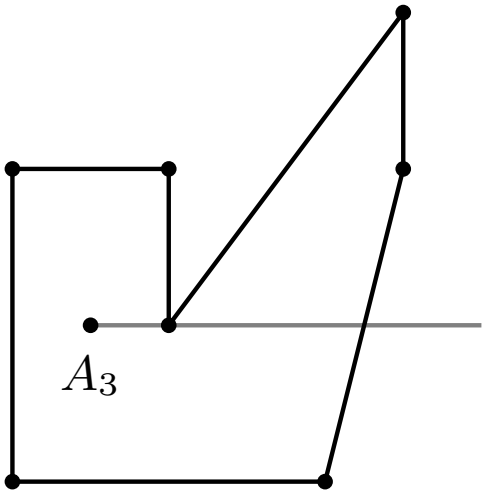
Punto interno a un poligono



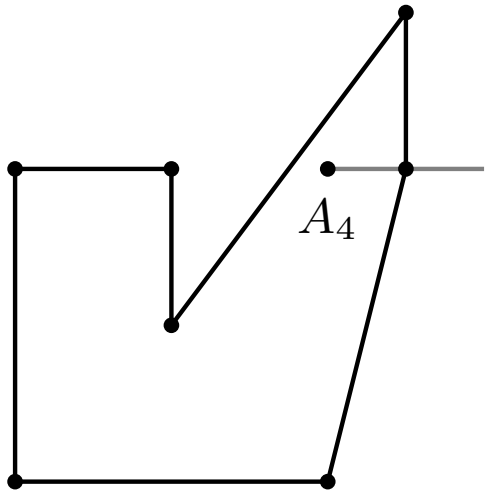
3 intersezioni $\Longrightarrow$ punto interno



2 intersezioni $\Longrightarrow$ punto esterno

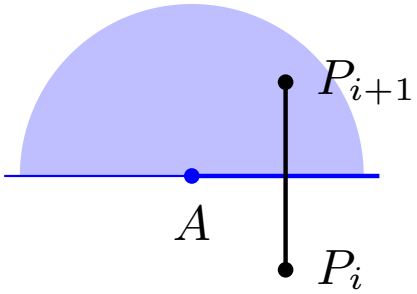


2 intersezioni?

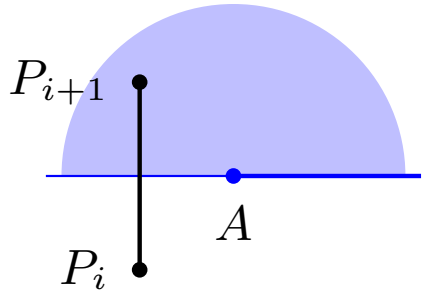


1 intersezione?

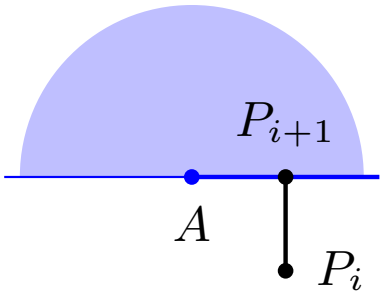
Punto interno a un poligono



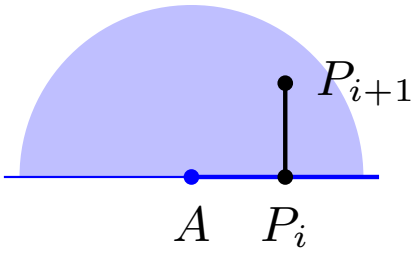
- Intersezione: ✓
- Semipiani diversi: ✓
- Segmento valido: ✓



- Intersezione: ×
- Semipiani diversi: ✓
- Segmento non valido: ×



- Intersezione: ✓
- Semipiani diversi: ✓
- Segmento valido: ✓



- Intersezione: ✓
- Semipiani diversi: ×
- Segmento non valido: ×

Punto interno a un poligono

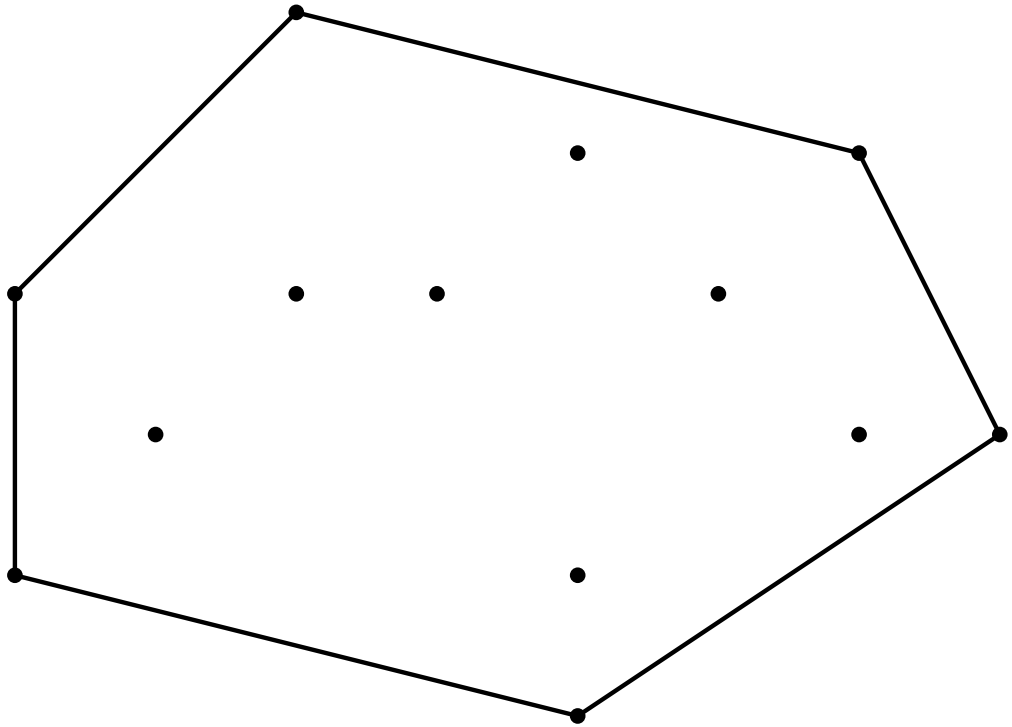
Implementazione

```
1  bool above(pt a, pt p) {
2      return p.y >= a.y;
3  }
4  bool crossesRay(pt a, pt p, pt q) {
5      return (above(a, q) - above(a, p)) * orient(a, p, q) > 0;
6  }
7  bool inPolygon(vector<pt> P, pt a, bool strict = false) {
8      int cnt = 0;
9      for (size_t i = 0; i < P.size(); i++) {
10         if (onSegment(P[i], P[(i + 1) % P.size()], a))
11             return !strict;
12         cnt += crossesRay(a, P[i], P[(i + 1) % P.size()]);
13     }
14     return cnt % 2;
15 }
```

Convex hull

Definizione

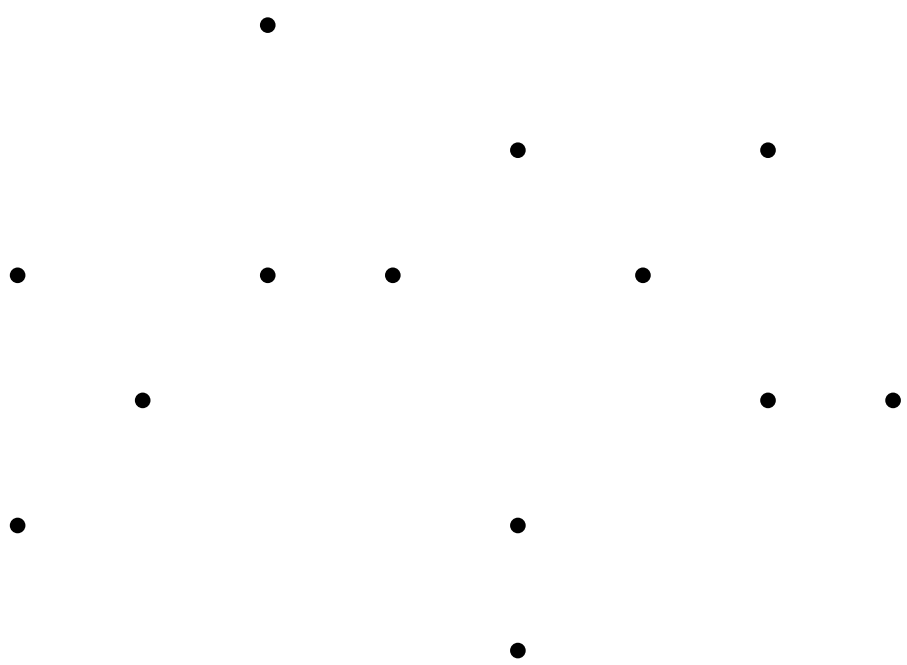
Il convex hull è il più piccolo poligono **convesso** che contiene tutti i punti di un dato insieme.



Convex hull

Andrew's algorithm

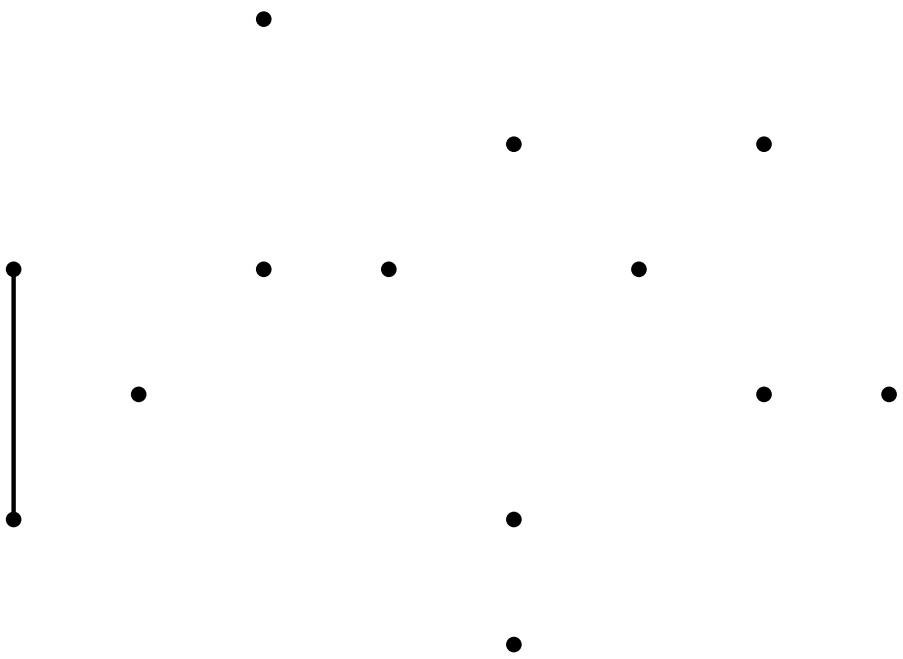
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

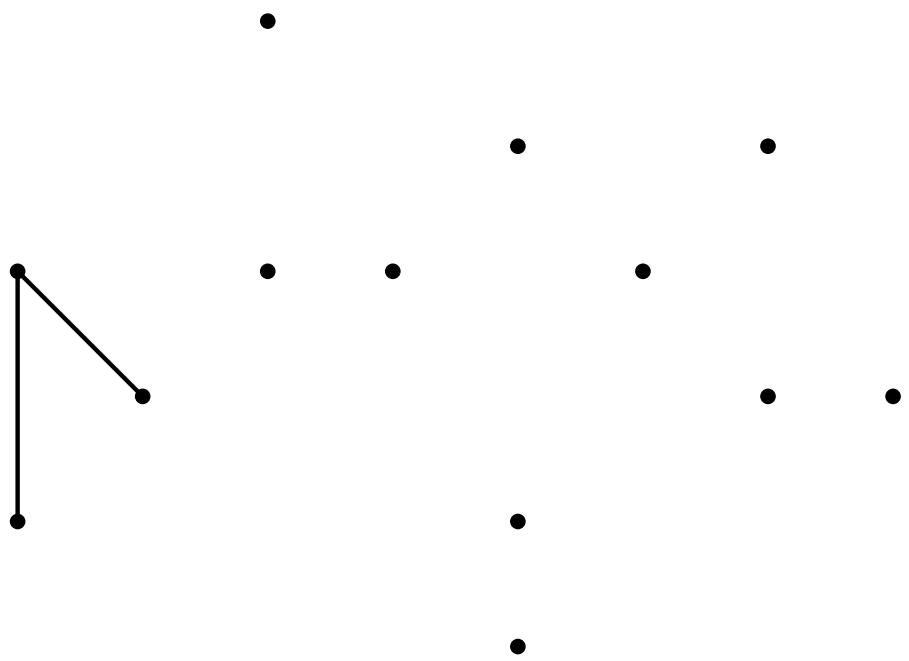
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

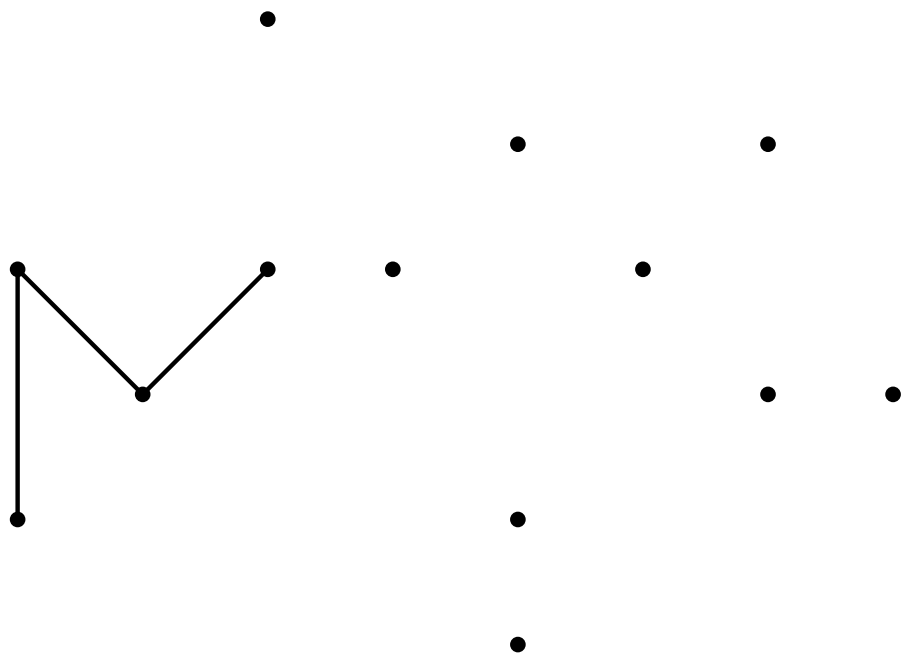
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

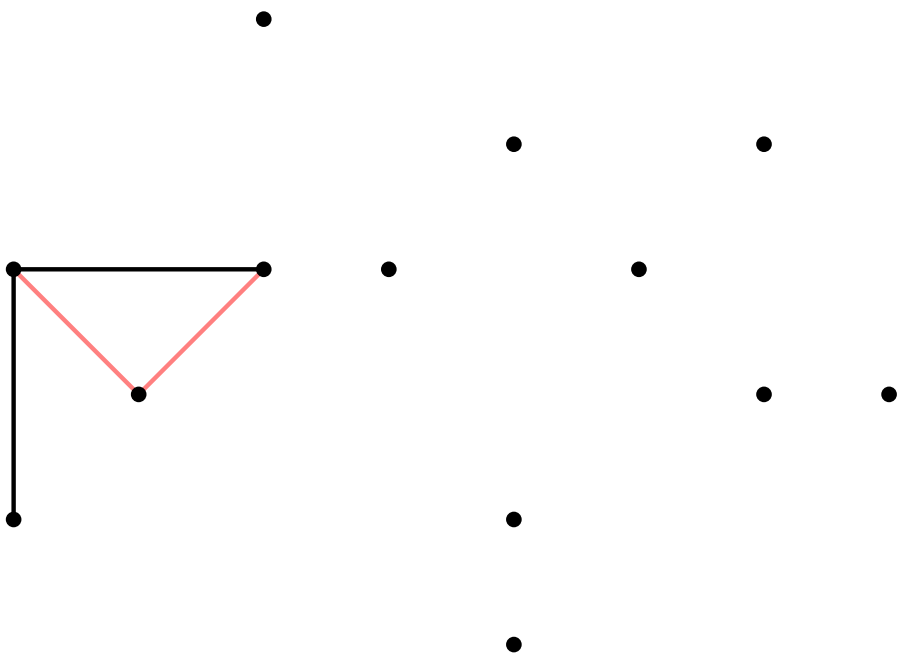
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

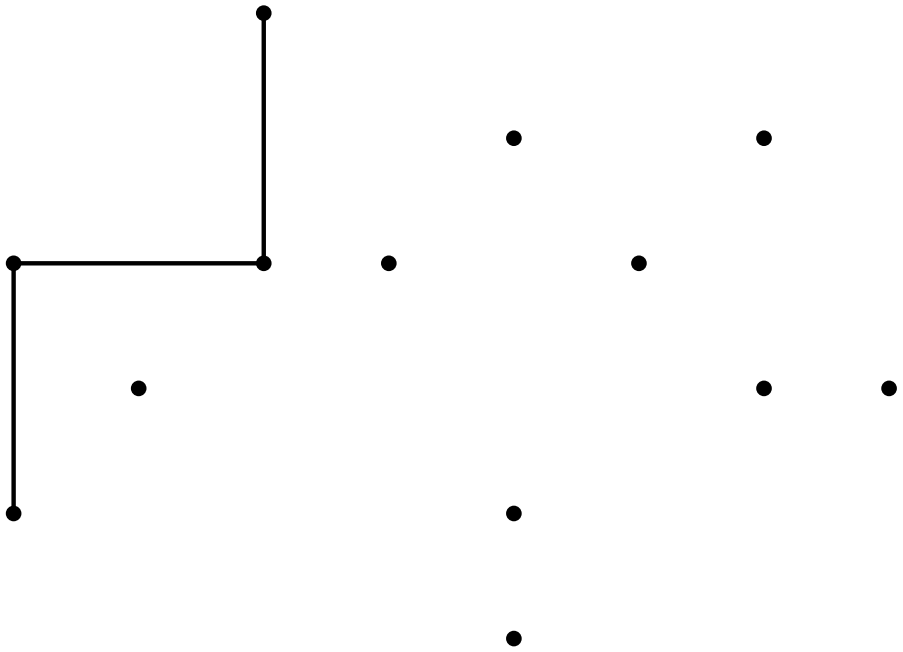
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

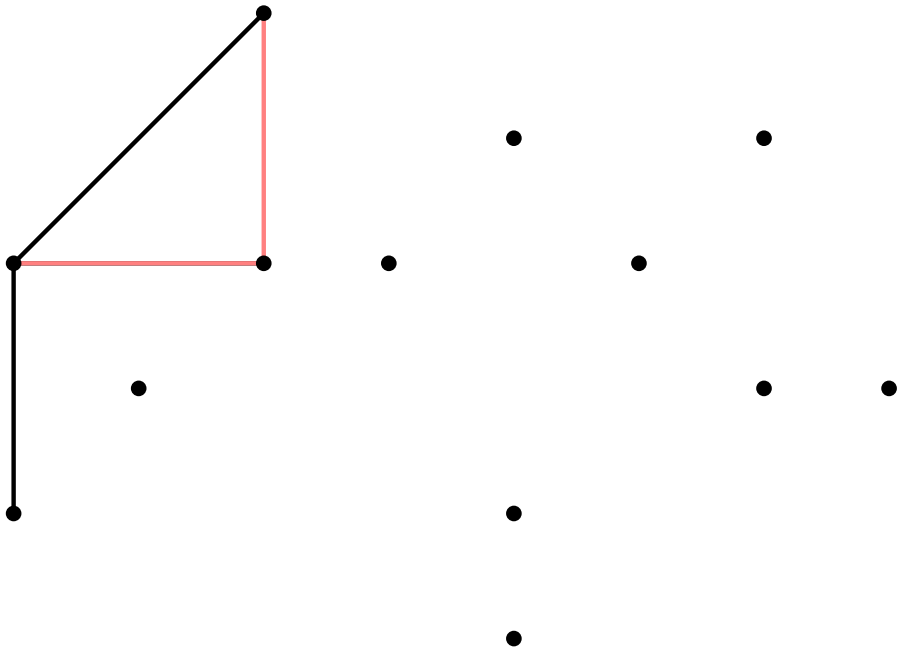
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

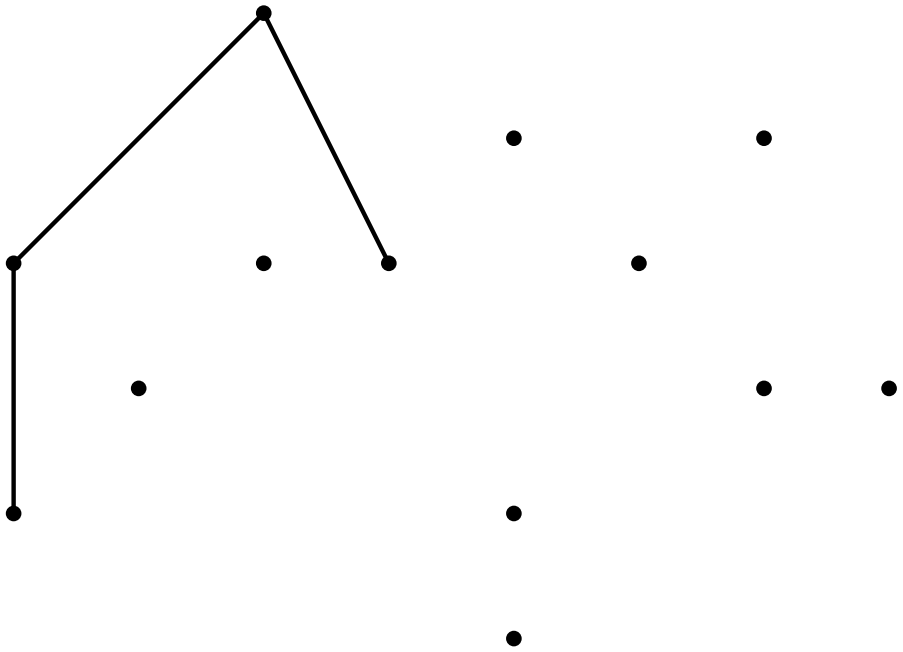
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

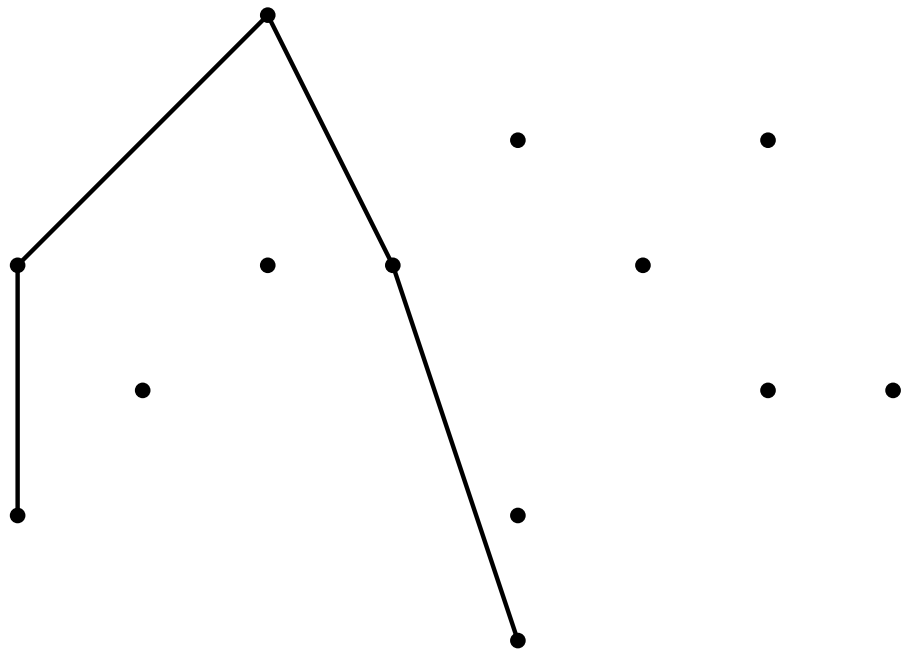
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

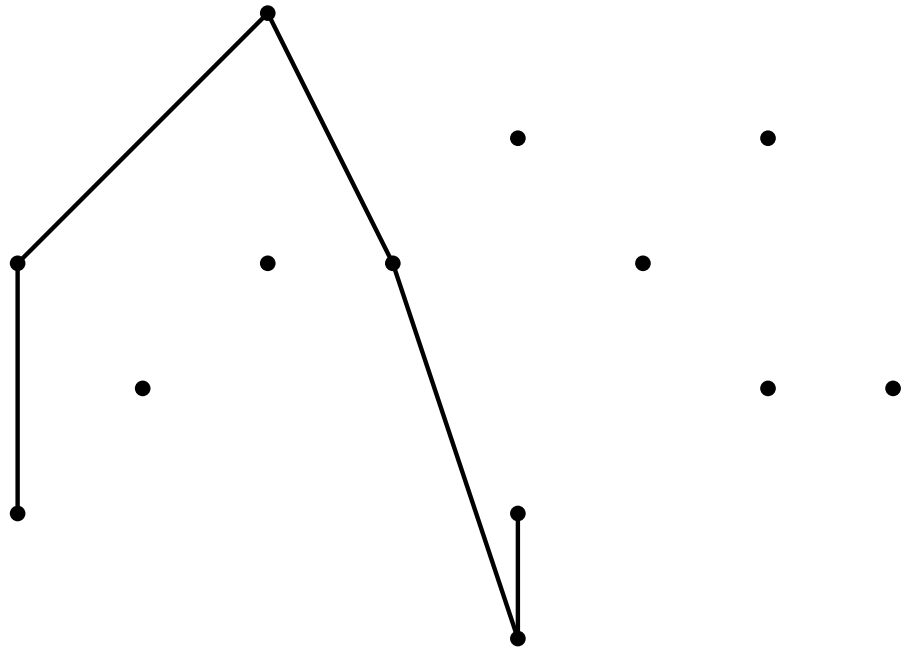
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

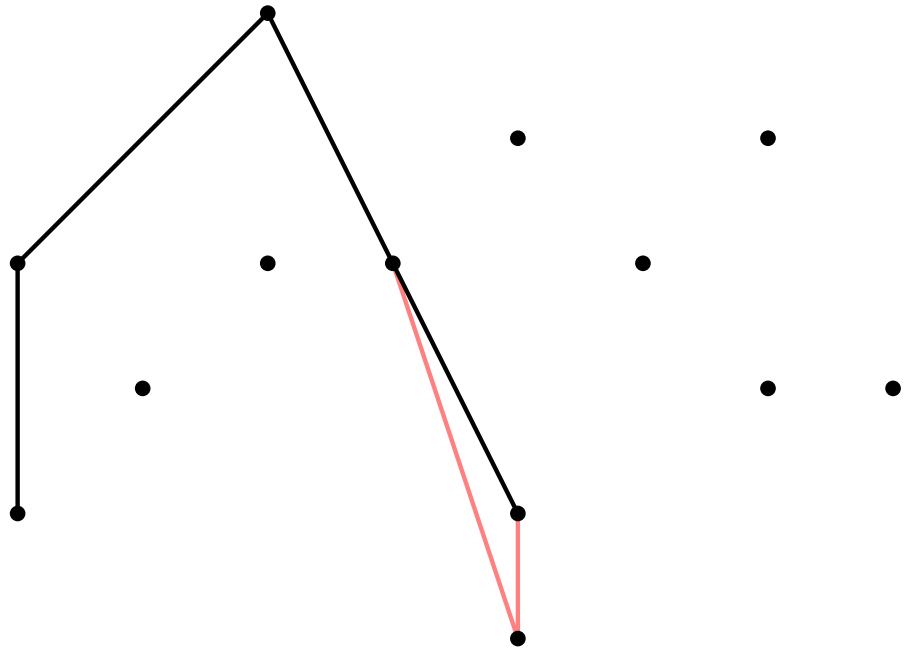
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

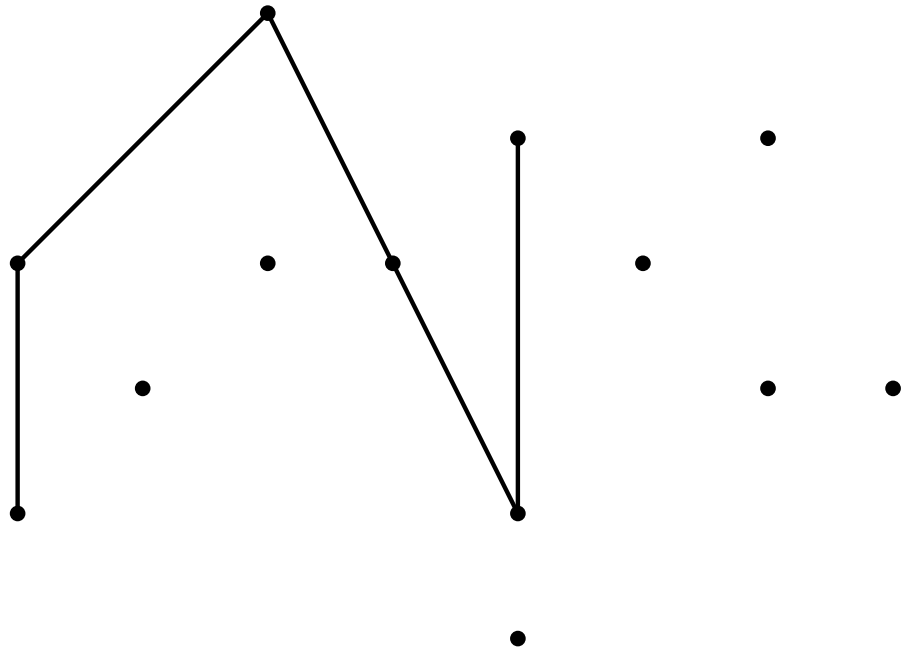
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

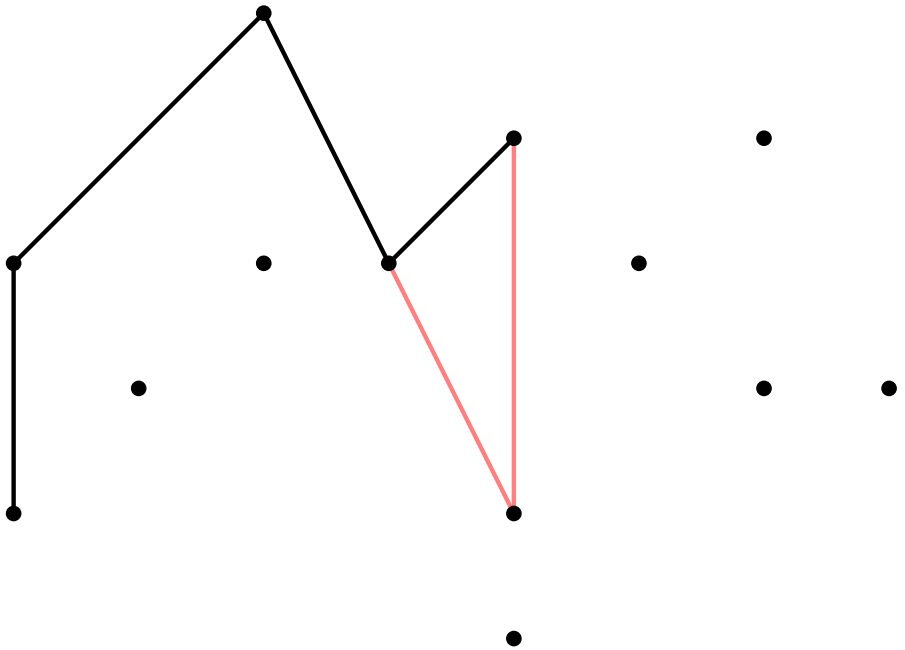
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

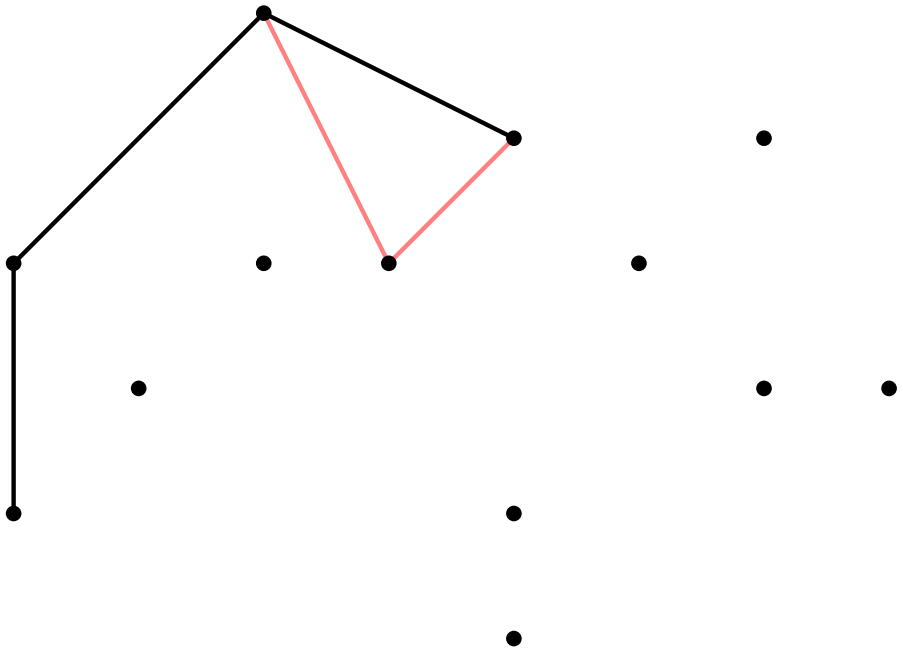
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

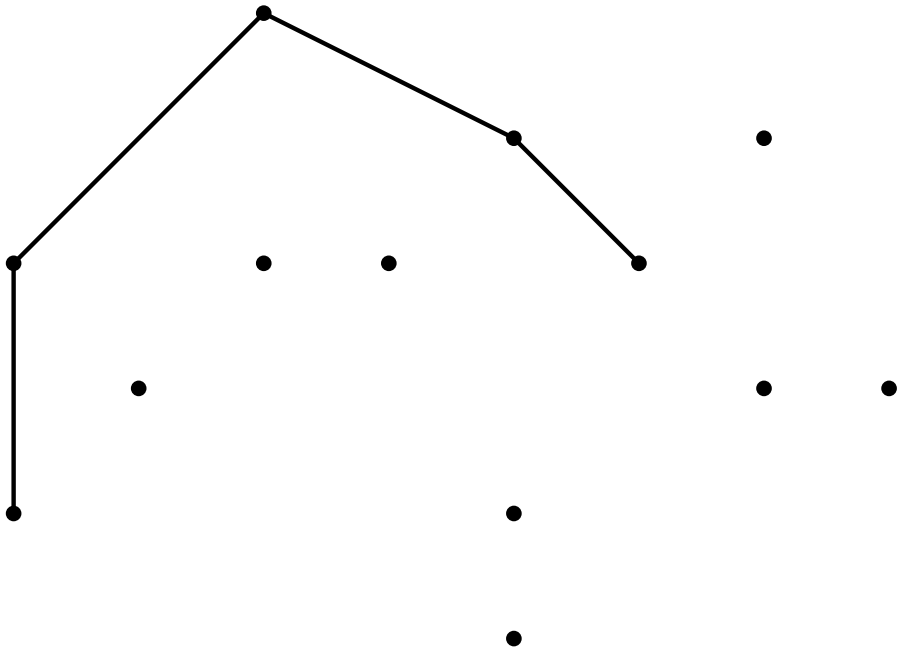
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

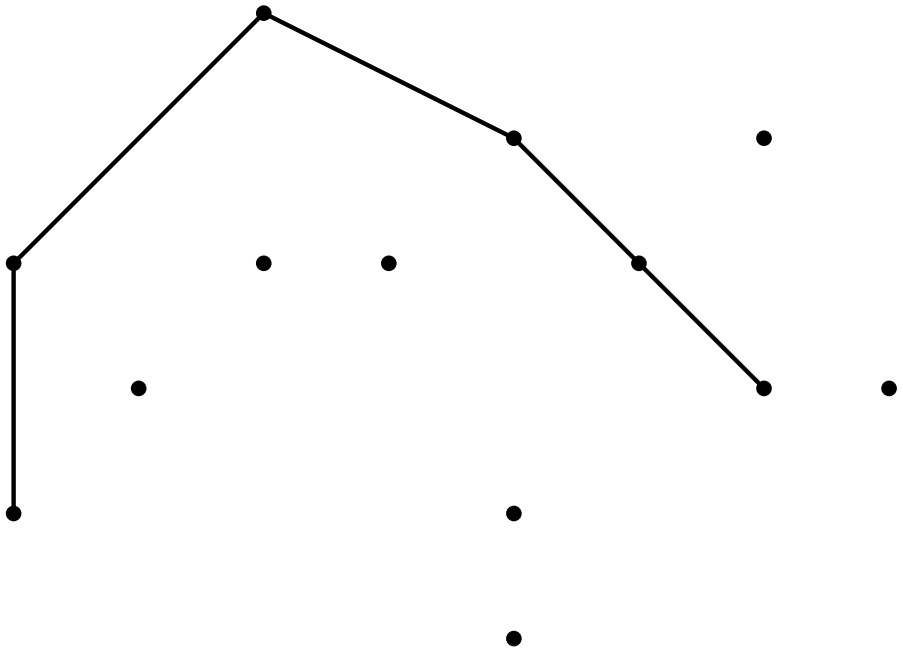
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

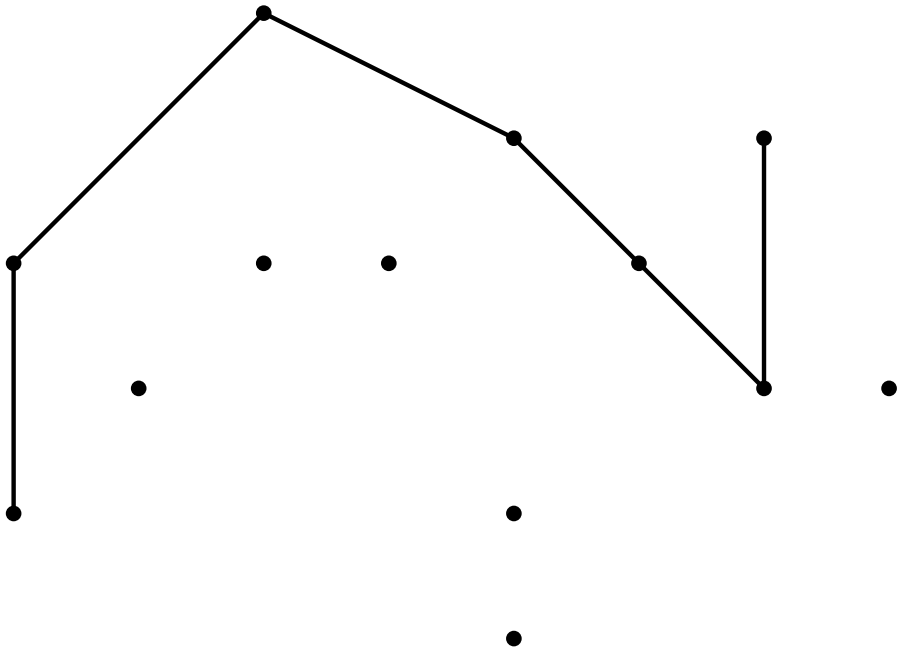
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

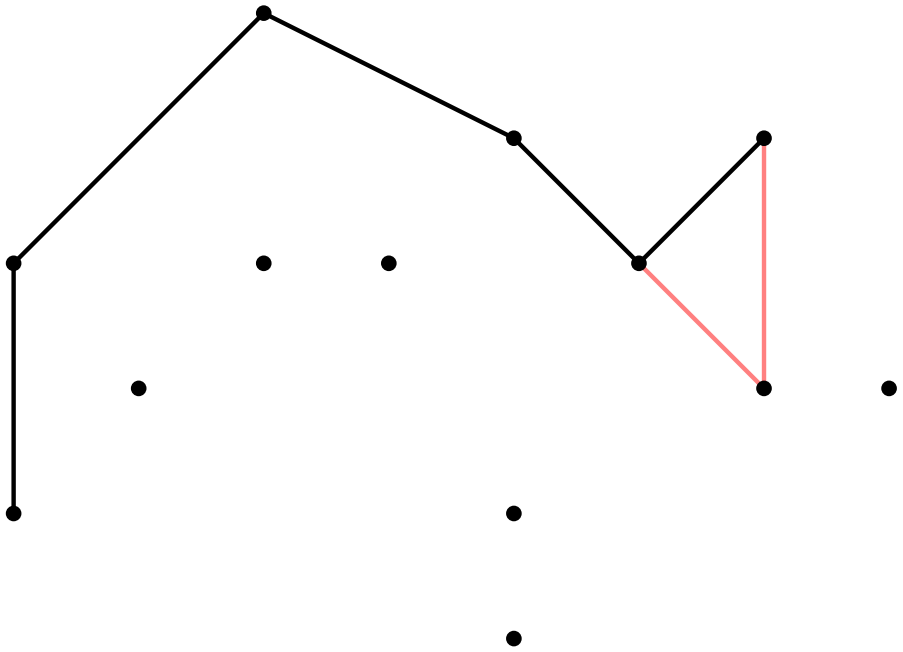
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

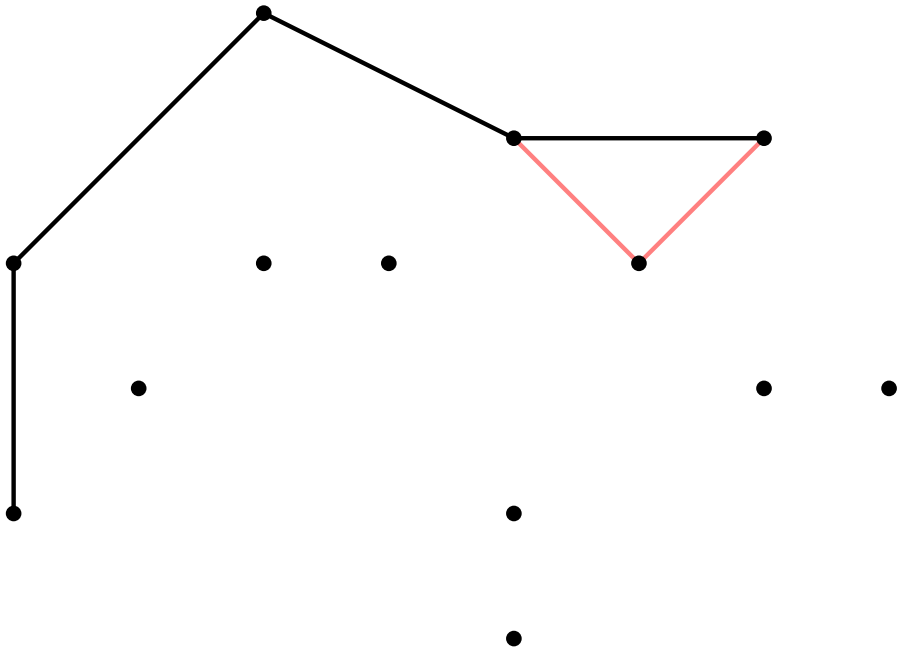
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

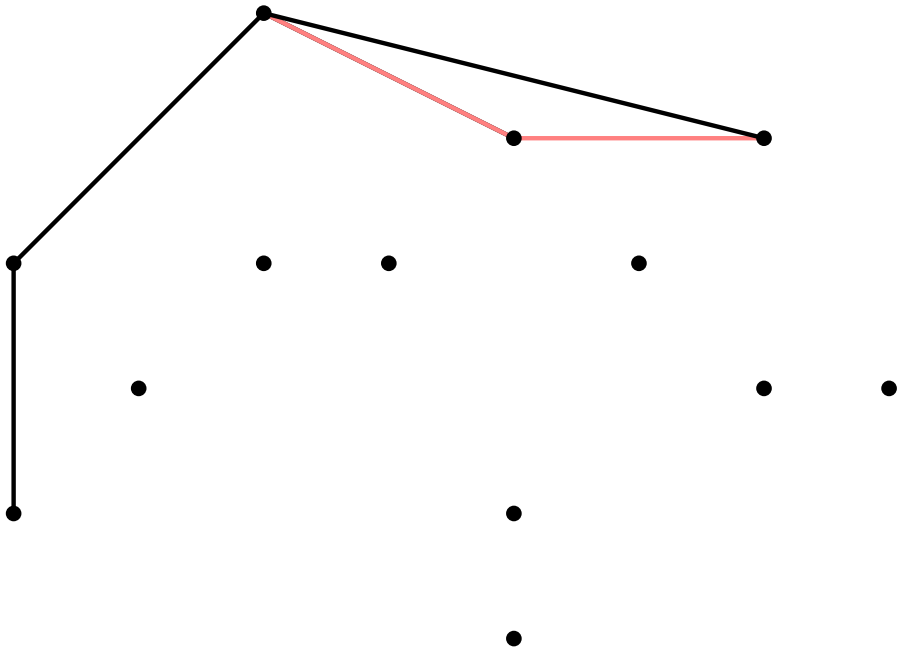
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

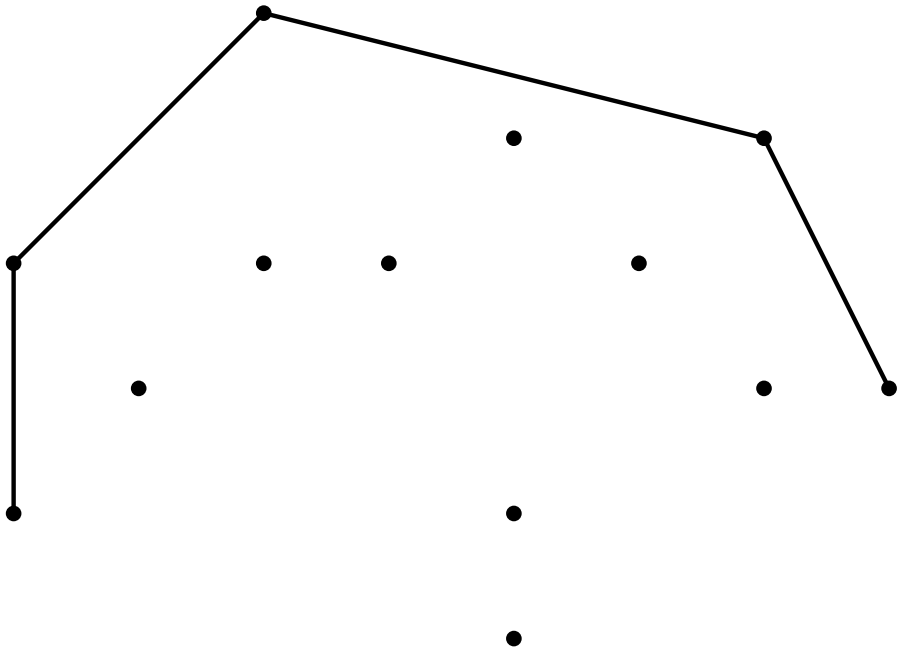
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

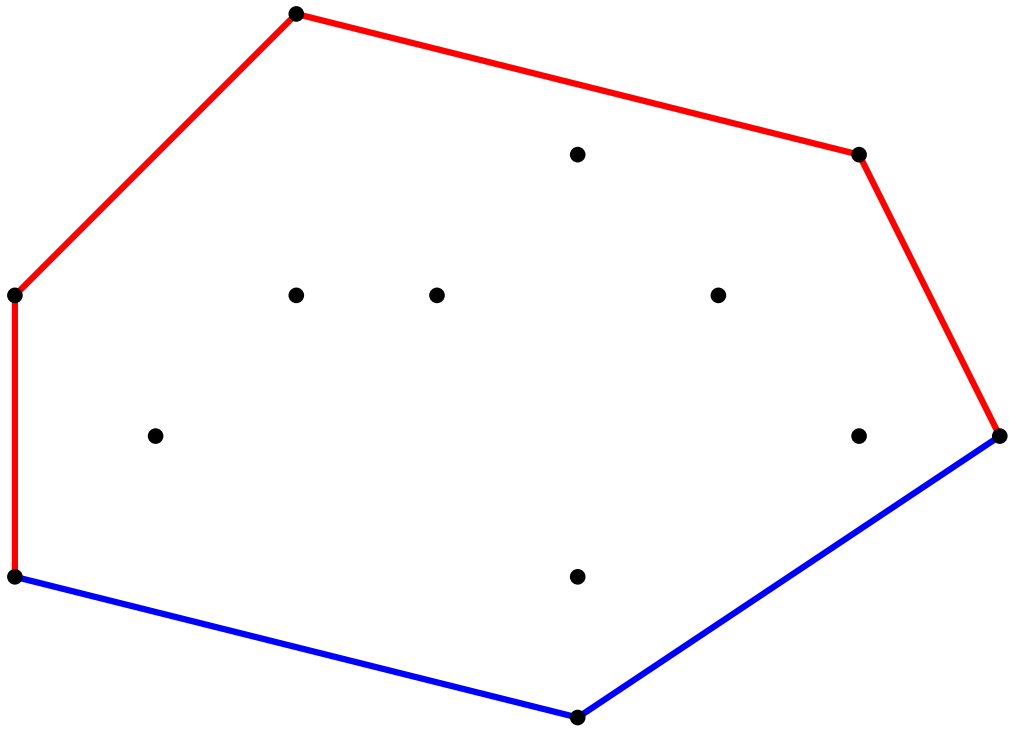
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

- infine ripetiamo il procedimento in ordine decrescente di coordinata x per determinare la metà inferiore del convex hull.

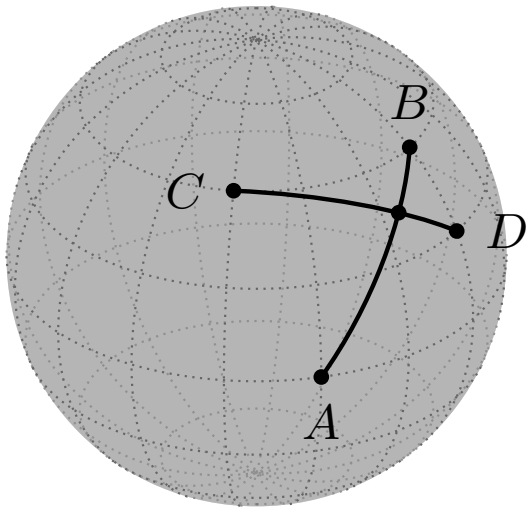


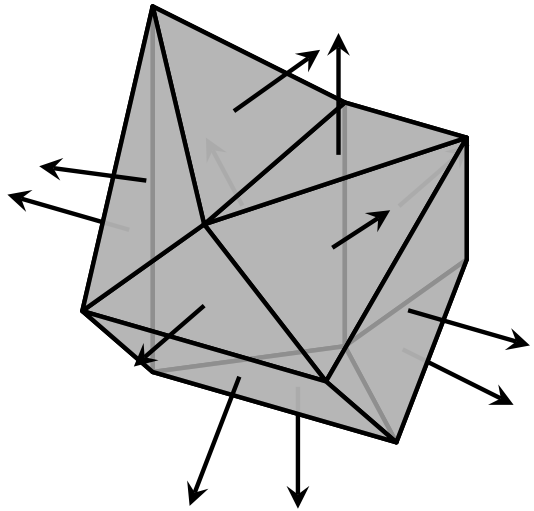
Convex hull

Implementazione

```
1   vector<pt> convexHull(vector<pt> P, bool all = true) {
2       sort(P.begin(), P.end(), [](pt a, pt b) {
3           return tie(a.x, a.y) < tie(b.x, b.y);
4       });
5
6       vector<pt> ch;
7       for (int i = 0; i < 2; i++) {
8           for (pt p : P) {
9               while (ch.size() >= 2 &&
10                      orient(ch[ch.size()-2], ch.back(), p) >= (all ? 0 : 1)) {
11                  ch.pop_back();
12              }
13              ch.push_back(p);
14          }
15          ch.pop_back();
16          reverse(P.begin(), P.end());
17      }
18
19      return ch;
20  }
```

Geometria 3D





lolnope

Variabili floating point

Evitare i float

Usare i float solo se necessario

- Usare i float solo quando è strettamente necessario.
- Cercare sempre di usare una rappresentazione alternativa in cui i valori sono interi ($2x$, x^2, ...).

Quale tipo usare?

Tipo	Precisione	Errore relativo	Valore massimo	Velocità
`float`	24 bit	$6.0 \cdot 10^{-8}$	$3.4 \cdot 10^{38}$	veloce
`double`	53 bit	$1.1 \cdot 10^{-16}$	$1.7 \cdot 10^{308}$	decente
`long double`	64 bit	$5.4 \cdot 10^{-20}$	$1.1 \cdot 10^{4932}$	lento
`__float128`	113 bit	$9.6 \cdot 10^{-35}$	$1.2 \cdot 10^{4932}$	molto lento

Consigli

Come usare correttamente i float

- Compensare l'errore durante i confronti.
- Minimizzare le operazioni che amplificano l'errore ($1/x$, $\sqrt{x}$, ...).
- Controllare il dominio delle funzioni ($\sqrt{x}$, $\arccos$, $\log$, ...).
- Cercare di eseguire operazioni con numeri dello stesso magnitudo.
- Usare `-ffast-math` (abilitato da `-Ofast`) può rendere il codice più veloce, ma lo rende non conforme allo standard IEEE.

Esercizi

Problemi

- https://cses.fi/problemset/task/2195
- https://codeforces.com/problemset/problem/1477/C
- https://codeforces.com/problemset/problem/1158/D
- https://codeforces.com/gym/101635

Bibliografia


Antti Laaksonen.
Guide to Competitive Programming.
Springer, 2017.


Victor Lecomte.
Handbook of geometry for competitive programmers.
`https://victorlecomte.com/cp-geo.pdf`, 2018.