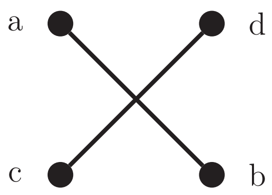
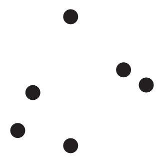
Test mit orientation



Schnitt $\Leftrightarrow$
orientation(a,b,c) $\neq$ orientation(a,b,d)
$\wedge$ orientation(c,d,a) $\neq$ orientation(c,d,b)

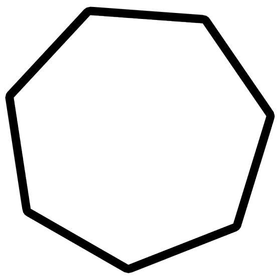
6.2.5 Problem: Konvexe Hülle

Eingabe: Liste von Punkten `list<point> L;`

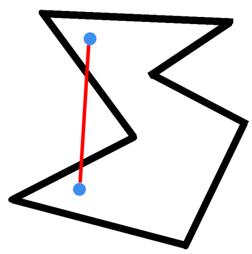


Ausgabe: kleinstes Polynom P, das alle Punkte aus L enthält.

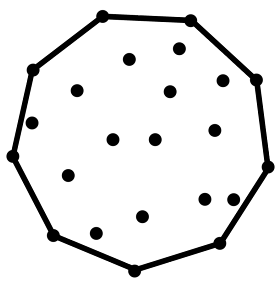
Konvex



Nicht konvex



Menge S kovex $\Leftrightarrow \forall\ a, b \in S : \overline{ab} \subseteq S$



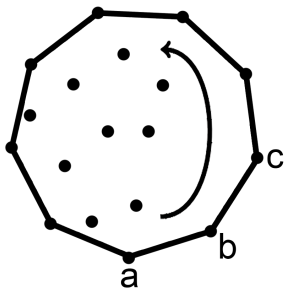
Man kann es sich wie ein Gummiband vorstellen, das außen um eine Menge an Nägeln gelegt wird.

Menge S kovex $\Leftrightarrow (a, b \in S : \overline{ab} \subseteq S)$
$\rightarrow$ Polygon $p = CH(L)$

Die Ecken vom Polygon p sind Punkte aus L.

Ausgabe: Ecken von P gegen den Uhrzeigersinn sortiert → Rand von P

Funktion: `list<point> convexHull(list<point>& L)`



konvex ⇔ $a.orientation(b, c) > 0$
(a,b,c machen einen links-Knick)

Einfacher Algorithmus: Gift-Wrapping

Im $\mathbb{R}^2$ einwickeln mit einer Schnur.

$L = \{q_1, .., q_n\}$ mit Ecken $p_1, \ldots$

1.) Startpunkt $p_1 \leftarrow min_{xy}(L)$ (Punkt unten links).

2.) Wie findet man p_2?
Betrachte horizontalen Strahl (l) nach rechts, der in p_1 startet.
Drehe l so lange gegen den Uhrzeigersinn bis er auf einen Punkt von L trifft.
$p_2 \leftarrow$ der Punkt q auf l mit max $dist(p_1, q)$ *

3.) Wiederhole Schritt 2 mit Ecke p_2 bis wir wieder bei p_1 angelangt sind.

* Lineare Suche in L nach der nächsten Ecke

```
list<point> ConvexHull(list<point>& L){
   list<point> CH; // Resultat
   // Startpunkt (Ecke) -> lineare Suche
    point q0 = L.first();
    point p;
    forall(p,L){
      if(p.cmp_xy(q0) == -1){
         q0 = p;
      }
   }
   CH.append(q0);
   L.del(q0);
   while(true){
      point q = L.first()
      forall(p,L){ // lineare Suche nach max. Winkel
         if(CH.last().orientation(q,p) == -1
            || ((CH.last().orientation(q,p) == 0)
            && CH.last().cmp_dist(q,p) == -1)){
            q = p;
         }
      }
      L.del(q);
      if(q == q0) break;
      CH.append(q);
   }
   return CH;
}
```

orientation(p,q,r) und

$$cmp\_xy(a,b) = \begin{cases} +1 & a >_{xy} b \\ 0 & a =_{xy} b \\ -1 & a <_{xy} b \end{cases} \quad \text{und}$$

$$cmp\_dist(a,b,c) = \begin{cases} +1 & dist(a,c) > dist(a,b) \\ 0 & dist(a,c) = dist(a,b) \\ -1 & dist(a,c) < dist(a,b) \end{cases} \quad \text{stehen zur Verfügung.}$$

Laufzeit (Output sensitiv)
Schritt 2 kostet $\mathcal{O}(n)$.
$\mathcal{O}(h * n)$ wobei $h = \#Ecken$ von $CH(S)$
Für jede Ecke 1x lineare Suche

Worst-case: h=n (oder Bruchteil von n)
$\rightarrow \mathcal{O}(n^2)$ z.B Kreis oder konvexe Kurve (z.B. Parabel)

Best-case: h = konstant z.B. h=3

Bessere Laufzeit
$\mathcal{O}(n * log_2(n))$ ($\rightarrow$ Sortieren)

Diese Laufzeit ist optimal, da das Finden einer konvexen Hülle (CH) mindestens so schwer ist, wie das Sortieren von n verschiedenen Zahlen.

Beweis Reduktion von Sortieren auf CH
Verschiedene Algorithmen mit Laufzeit $\mathcal{O}(n * log_2(n))$

Strategie:

i) Inkrementell

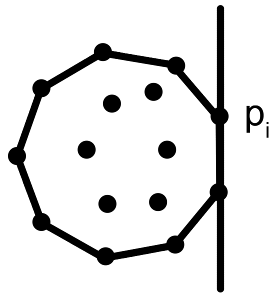
ii) Divide & Conquer

iii) Scan / sweap

□

Inkrementeller Algorithmus

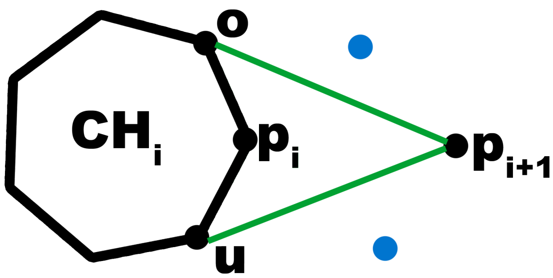
1.) Sortiere die Punkteliste nach xy-Ordnung $\rightarrow$ Folge von Punkten ($\mathcal{O}(n * log_2(n))$)
Idee: Durchlaufe die Punkte $\rightarrow$ aktueller Punkt p_i $(i = 1, \ldots, n)$



$CH_i = CH(p_1, \ldots, p_i)$ ist berechnet.

Schritt $i \to i+1$

Da p_i extrem ist in $\{p_1, \ldots, p_i\}$ ist p_i (rechteste) Ecke von CH_i.



Die blauen Punkte sind mögliche Stellen, an denen p_{i+1} auch liegen kann.

intuitiv:

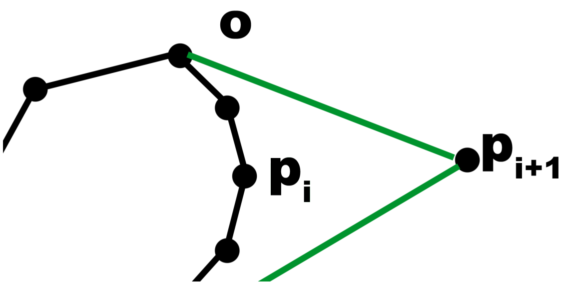
i) Entferne alles (Ecken), das von p_{i+1} sichtbar (beleuchtet) ist, aus CH_i.

ii) Füge p_{i+1} stattdessen ein

formaler:

i) Finde die Tangentenberührpunkte o (oberer Berührpunkt) und u (unterer Berührpunkt) und entferne alle Ecken zwischen o und u.

ii) Füge p_{i+1} unmittelbar nach u ein (vor o)



Die Ecke, an der es das erste Mal einen Links-Knick von p_{i+1} aus, wird als oberer Berührpunkt genommen.

Implementierung

i) Datenstruktur für CH: zyklische Liste L von Punkten.
list<point> L, CH;
L.cyclic_succ(p) zyklischer Nachfolger
L.cyclic_pred(p) zyklischer Vorgänger
Punkte in L sind gegen den Uhrzeigersinn sortiert.

ii) Initialisierung: Dreieck aus p_1, p_2, p_3
Annahme: Nicht colinear (orientation$(p_1, p_2, p_3) \neq 0$
korrekt orientiert: oruientation$(p_1, p_2, p_3) > 0$

iii) Schleife für $p_1, \ldots, p_n$

```
for(i=4; i≤ n; i++){
    // oberer Berührpunkt
     p ← p_{i−1};
    while(orientation(p_i,p,CH.cyclic_succ(p)) ≤ 0){
        // Für Triangulierung
        "Gibt Δ(p_i,p,CH.cyclic_succ(p)) aus";
        p ← CH.cyclic_succ(p);
    }
    o ← p;
    // unterer Berührpunkt
    p ← p_{i−1};
    while(orientation(p_i,p,CH.cyclic_pred(p)) ≥ 0){
        // Für Triangulierung
        "Gibt Δ(p_i,p,CH.cyclic_pred(p)) aus";
        p ← CH.cyclic_pred(p);
    }
    u ← p;
    // Entferne alle Punkte dazwischen
    while(CH.cyclic_succ(u) ≠ o){
        CH.delete(CH.cyclic_succ(u));
    }
    // Füge p_i nach u ein
    CH.insert_after(p_i,u);
}
```

Analyse

1) Sortieren $\mathcal{O}(n * log(n))$
2) for-Schleife
   Beobachtung: Jeder Punkt wird genau einmal in die Liste CH eingefügt und höchstens einmal entfernt
   $\Rightarrow$ for-Schleife hat Laufzeit $\mathcal{O}(n)$

Laufzeit $\mathcal{O}(n * log(n))$
$\mathcal{O}(n)$, falls die Punkte sortiert sind

Divide & Conquer

$p_1, \ldots, p_n$ sortiert.

```
CH(p_1,...,p_n){
    if(m≤ 3){
        Konstruiere Dreieck
    }
    else{
        m ← ⌊ n/2 ⌋;
        A ← CH(p_1,...,p_m);
        B ← CH(p_{m+1},...,p_n)
    }
}
```