



```
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

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

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

**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  $n$ )

→  $\mathcal{O}(n^2)$  z.B. Kreis oder konvexe Kurve (z.B. Parabel)

Best-case:  $h = \text{konstant}$  z.B.  $h=3$

**Bessere Laufzeit**

$\mathcal{O}(n * \log_2(n))$  (→ 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:

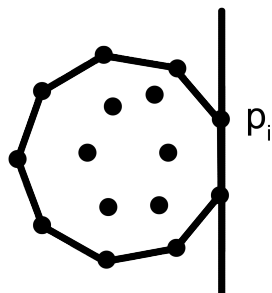
- i) Inkrementell
- ii) Divide & Conquer
- iii) Scan / sweep

□

**Inkrementeller Algorithmus**

1.) Sortiere die Punkteliste nach xy-Ordnung → Folge von Punkten ( $\mathcal{O}(n * \log_2(n))$ )

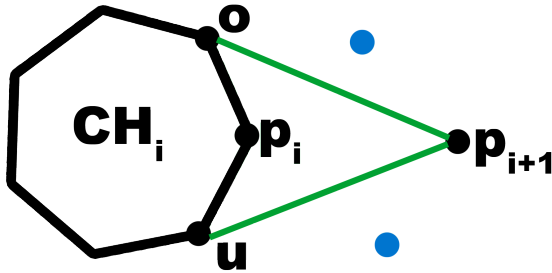
Idee: Durchlaufe die Punkte → aktueller Punkt  $p_i$  ( $i = 1, \dots, n$ )



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

Schritt  $i \rightarrow i + 1$

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



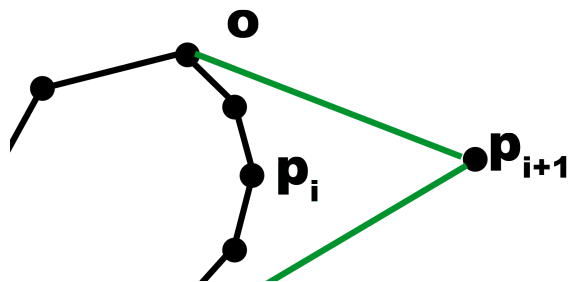
Die blauen Punkte sind mögliche Stellen, an denen  $p_{i+1}$  auch liegen kann.

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 Tangentenberührungspunkte o (oberer Berührungspunkt) und u (unterer Berührungspunkt) 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 Berührungspunkt 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 ( $\text{orientation}(p_1, p_2, p_3) \neq 0$ )

korrekt orientiert:  $\text{orientation}(p_1, p_2, p_3) > 0$

- iii) Schleife für  $p_1, \dots, p_n$

```
for(i=4; i ≤ n; i++){
    // oberer Berührungspunkt
    p ← pi-1;
    while(orientation(pi, p, CH.cyclic_succ(p)) ≤ 0){
        // Für Triangulierung
        "Gibt Δ(pi, p, CH.cyclic_succ(p)) aus";
        p ← CH.cyclic_succ(p);
    }
    o ← p;
    // unterer Berührungspunkt
    p ← pi-1;
    while(orientation(pi, p, CH.cyclic_pred(p)) ≥ 0){
        // Für Triangulierung
        "Gibt Δ(pi, 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 pi nach u ein
    CH.insert_after(pi, 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**

$p_1, \dots, p_n$  sortiert.

```
CH( $p_1, \dots, p_n$ ) {  
  if ( $m \leq 3$ ) {  
    Konstruiere Dreieck  
  }  
  else {  
     $m \leftarrow \lfloor n/2 \rfloor$ ;  
     $A \leftarrow \text{CH}(p_1, \dots, p_m)$  ;  
     $B \leftarrow \text{CH}(p_{m+1}, \dots, p_n)$   
  }  
}
```