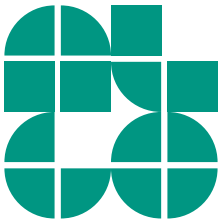
Karlsruhe Institute of Technology

# Übung Algorithmische Geometrie
## Konvexe Hülle & Streckenschnitte

LEHRSTUHL FÜR ALGORITHMIK · INSTITUT FÜR THEORETISCHE INFORMATIK · FAKULTÄT FÜR INFORMATIK

Benjamin Niedermann

30.04.2014



# Ablauf

Karlsruhe Institute of Technology

## Übungsblatt 1

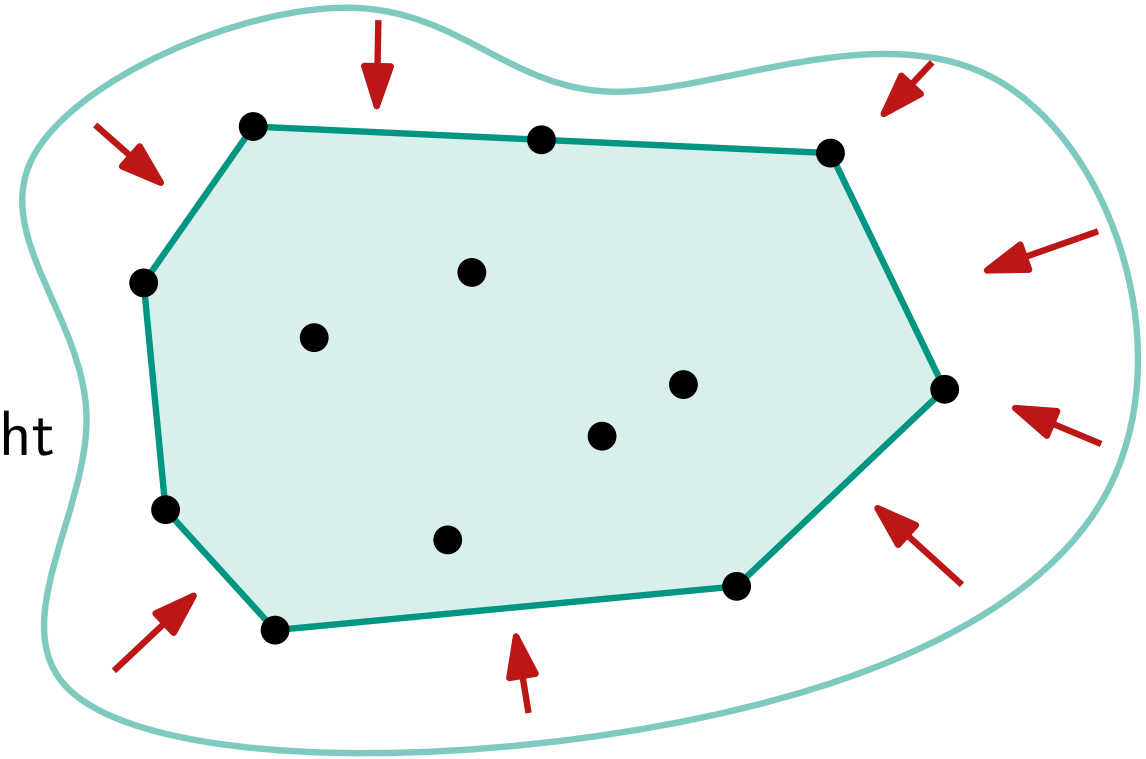
## Übungsblatt 2

# Definition Konvexe Hülle

**Def:** Eine Menge $S \subseteq \mathbb{R}^2$ heißt **konvex**, wenn für je zwei Punkte $p, q \in S$ auch gilt $\overline{pq} \in S$.
Die **konvexe Hülle** $CH(S)$ von $S$ ist die kleinste konvexe Menge, die $S$ enthält.

In der Physik:

- lege großes Gummiband um alle Punkte
- und lass es los!
- hilft algorithmisch leider nicht



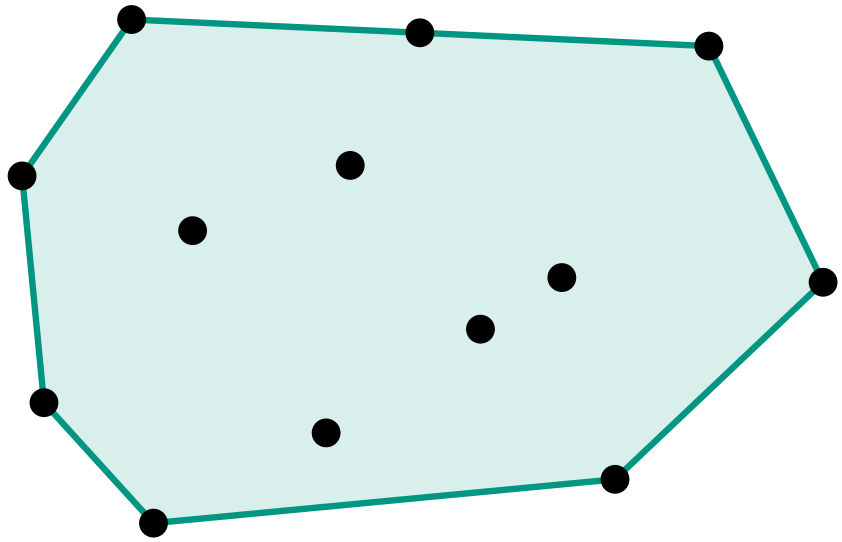
In der Mathematik:

- definiere $CH(S) = \bigcap_{C \supseteq S:\ C \text{ konvex}} C$
- hilft auch nicht :-(

# Algorithmischer Ansatz

**Lemma:**

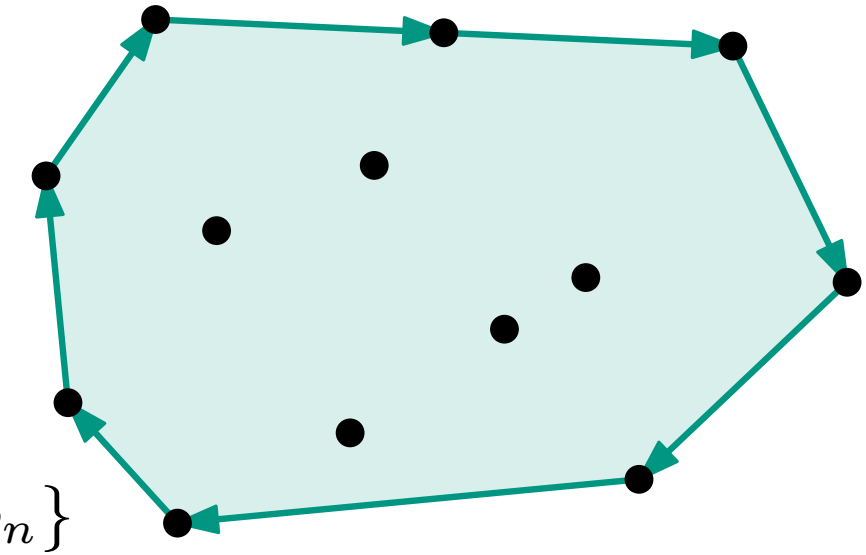
Für eine Punktmenge $P \subseteq \mathbb{R}^2$ ist $CH(P)$ ein konvexes Polygon, das $P$ enthält und dessen Ecken in $P$ liegen.



# Algorithmischer Ansatz

**Lemma:**

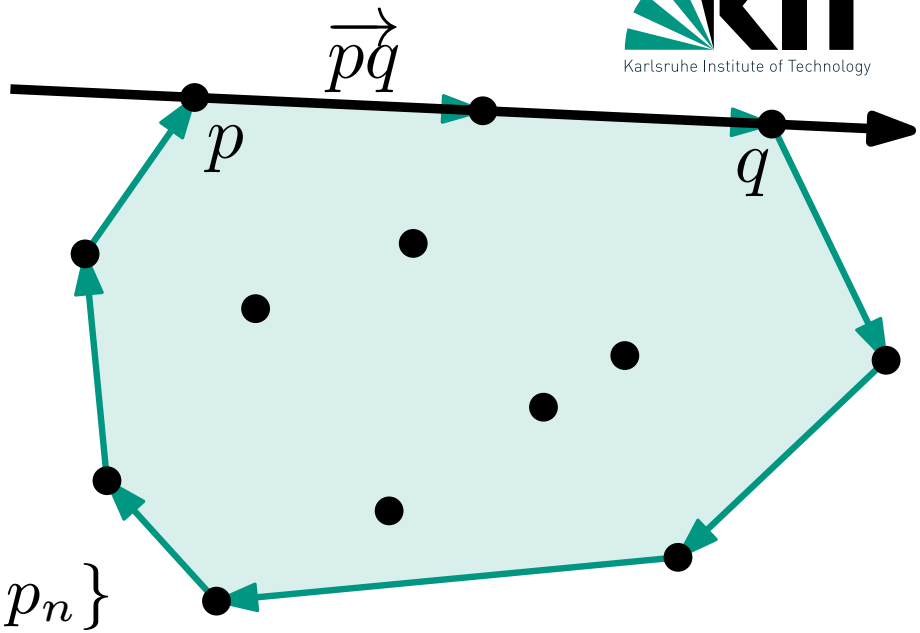
Für eine Punktmenge $P \subseteq \mathbb{R}^2$ ist $CH(P)$ ein konvexes Polygon, das $P$ enthält und dessen Ecken in $P$ liegen.



**Eingabe:** Punktmenge $P = \{p_1, \ldots, p_n\}$

**Ausgabe:** Knotenliste von $CH(P)$ im UZS

# Algorithmischer Ansatz




**Lemma:**

Für eine Punktmenge $P \subseteq \mathbb{R}^2$ ist $CH(P)$ ein konvexes Polygon, das $P$ enthält und dessen Ecken in $P$ liegen.

**Eingabe:** Punktmenge $P = \{p_1, \ldots, p_n\}$

**Ausgabe:** Knotenliste von $CH(P)$ im UZS

**Beobachtung:**

$(p, q)$ ist Kante von $CH(P) \Leftrightarrow$ jeder Punkt $r \in P \setminus \{p, q\}$ liegt

- strikt rechts der orientierten Geraden $\overrightarrow{pq}$ oder
- auf der Strecke $\overline{pq}$

# Laufzeitanalyse

FirstConvexHull($P$)

$E \leftarrow \emptyset$

**foreach** $(p, q) \in P \times P$ with $p \neq q$ **do** $(n^2 - n)\cdot$

  *valid* $\leftarrow$ *true*

  **foreach** $r \in P$ **do**

    **if not** ($r$ strikt rechts von $\overrightarrow{pq}$ **or** $r \in \overline{pq}$) **then** $\Theta(1)$

      *valid* $\leftarrow$ *false*

  $\Theta(n)$

  **if** *valid* **then**

    $E \leftarrow E \cup \{(p, q)\}$

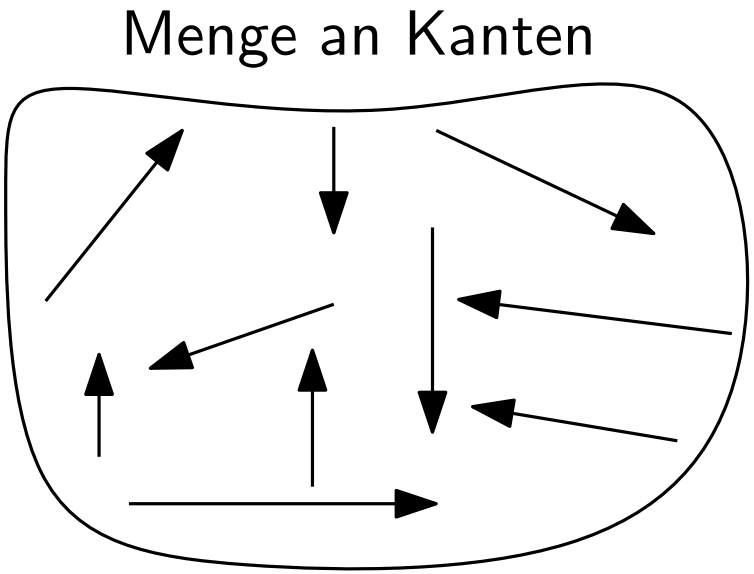
$\Theta(n^3)$

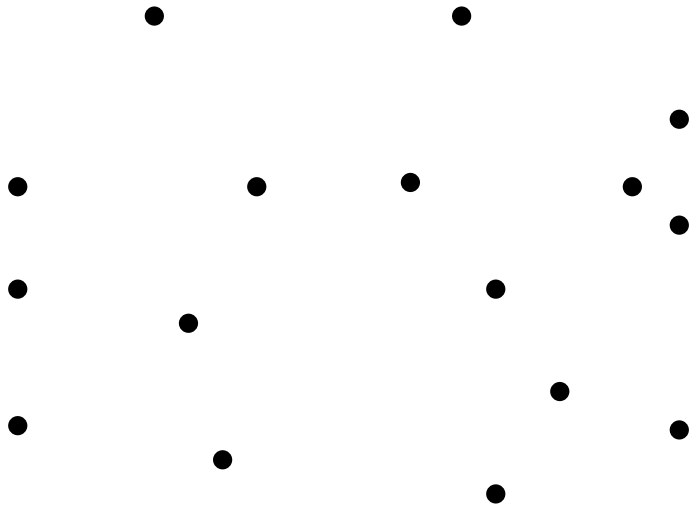
konstruiere sortierte Knotenliste $L$ von $CH(P)$ aus $E$

**return** $L$

**Aufgabe:** Wie implementiert man das in $O(n \log n)$?

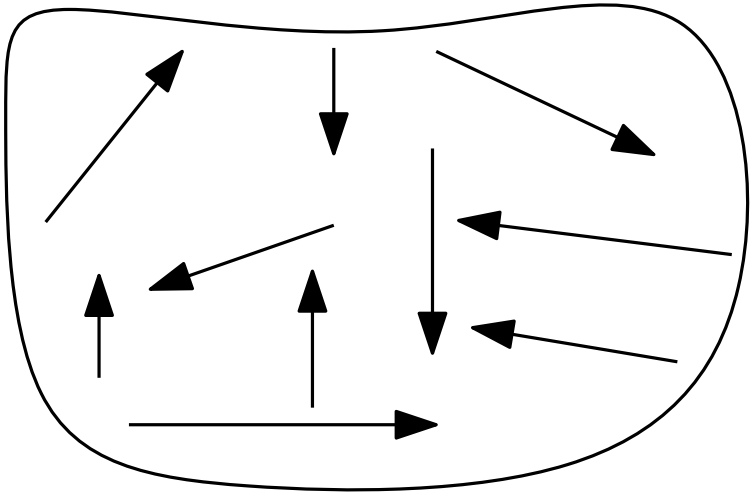
# Lösung



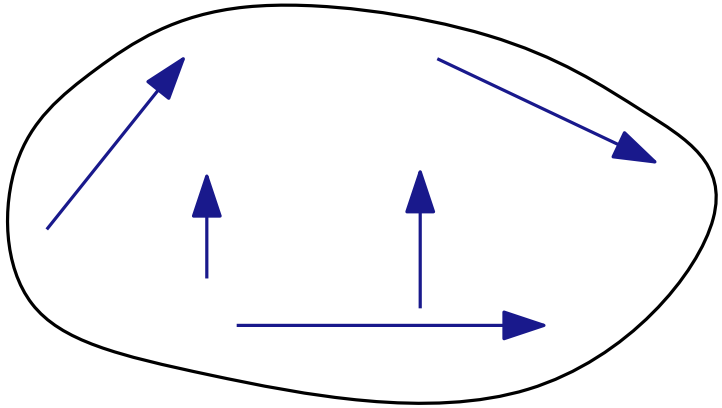
# Lösung

Menge an Kanten



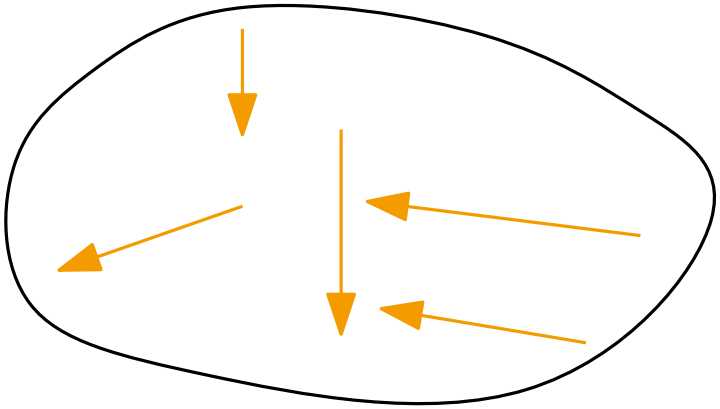
Sortiere von links nach rechts*.

bzgl. Quellknoten

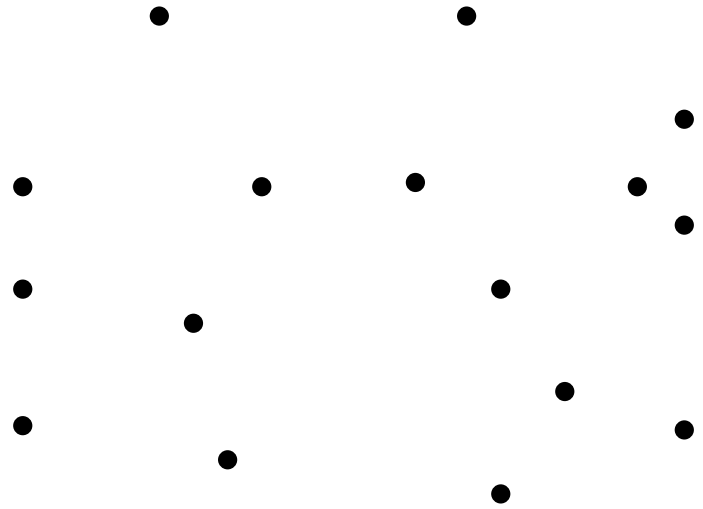


Kanten die nach rechts oder oben zeigen.

Sortiere von bezüglich rechts nach links*. Quellknoten

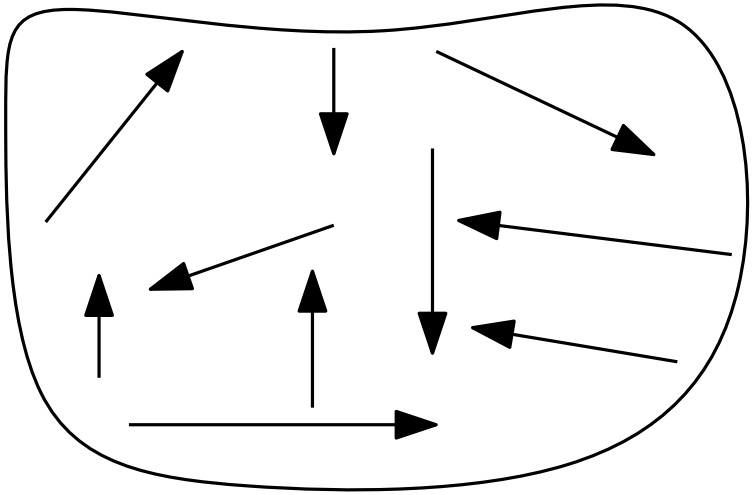


Kanten die nach links oder unten zeigen.



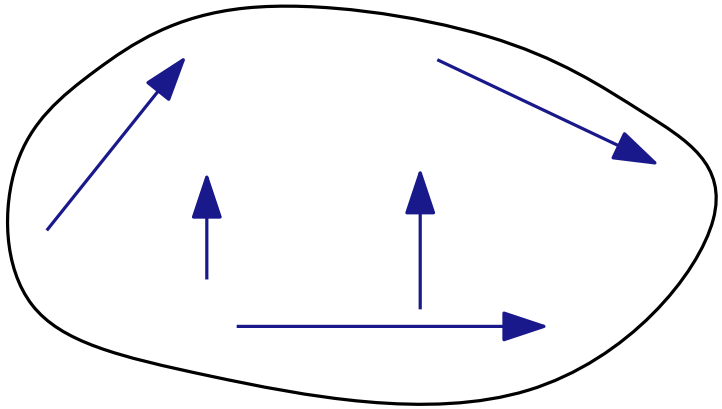
# Lösung

Menge an Kanten



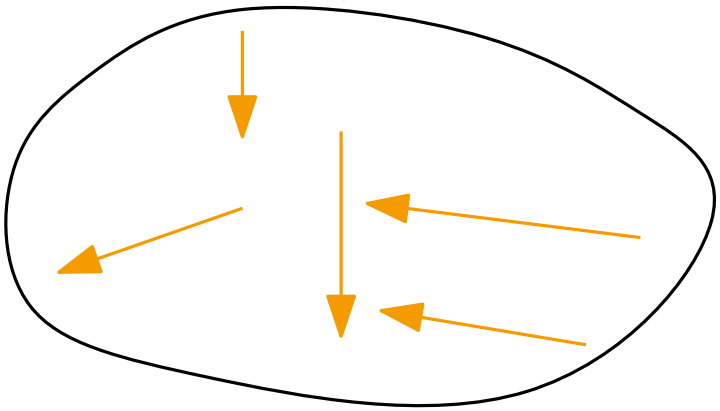
Sortiere von links nach rechts*.

bzgl. Quellknoten



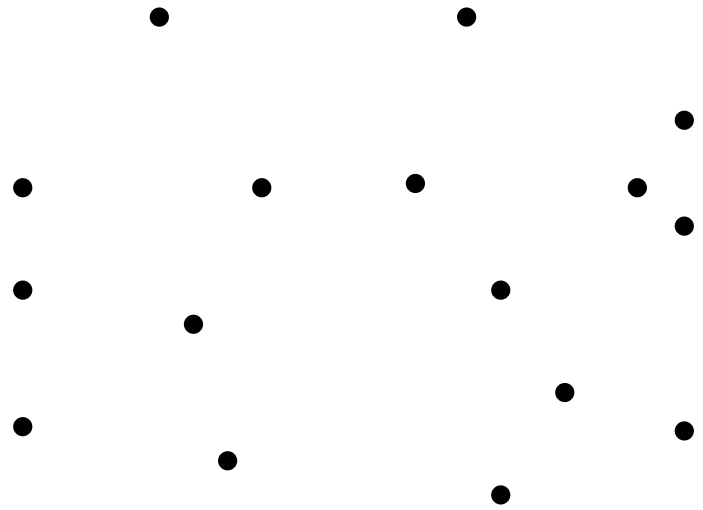
Kanten die nach rechts oder oben zeigen.

Sortiere von bezüglich rechts nach links*. Quellknoten

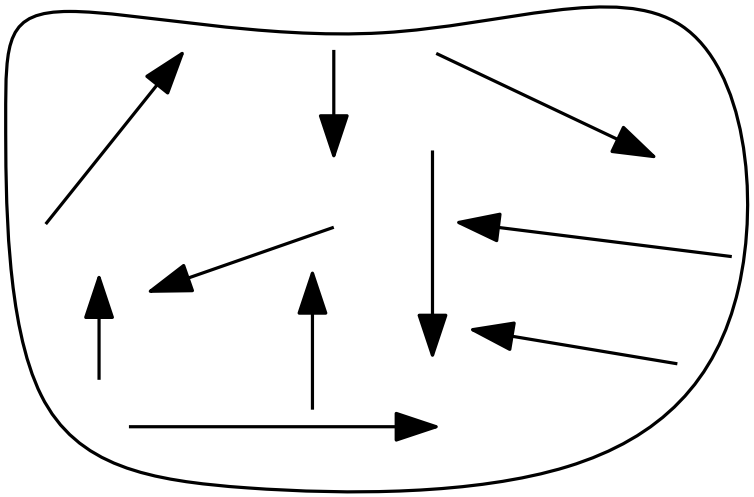


Kanten die nach links oder unten zeigen.

*wenn nicht eindeutig:
von unten nach oben.
von oben nach unten.



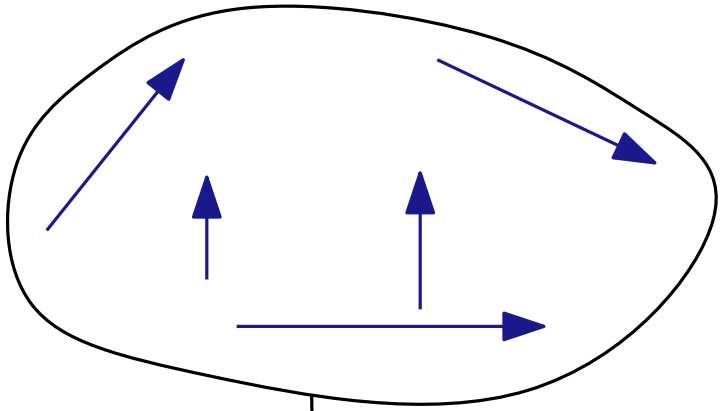
# Lösung



Menge an Kanten

Sortiere von links nach rechts*.

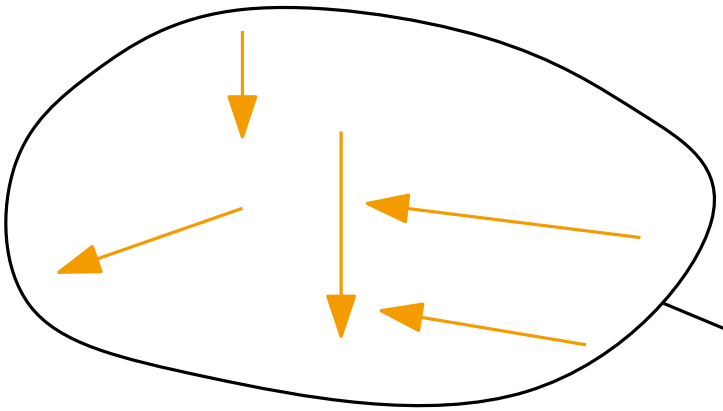
bzgl. Quellknoten



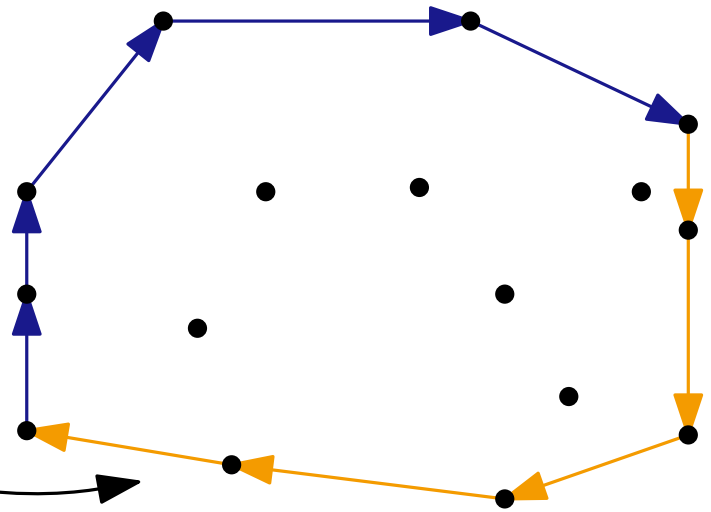
Kanten die nach rechts oder oben zeigen.

Sortiere von bezüglich rechts nach links*. Quellknoten

*wenn nicht eindeutig:
von unten nach oben.
von oben nach unten.



Kanten die nach links oder unten zeigen.



# Alternativer Ansatz: Gift Wrapping

Karlsruhe Institute of Technology

**Idee:** beginnend mit einem Punkt $p_1$ von $CH(P)$ finde nächste Kante von $CH(P)$ im UZS

# Alternativer Ansatz: Gift Wrapping

**Idee:** beginnend mit einem Punkt $p_1$ von $CH(P)$ finde nächste Kante von $CH(P)$ im UZS

GiftWrapping($P$)

$p_1 = (x_1, y_1) \leftarrow$ rechtester Punkt in $P$; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$

**while** true **do**

$p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$

**if** $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$

**return** $(p_1, \ldots, p_{j+1})$

# Alternativer Ansatz: Gift Wrapping

**Idee:** beginnend mit einem Punkt $p_1$ von $CH(P)$ finde nächste Kante von $CH(P)$ im UZS

GiftWrapping($P$)

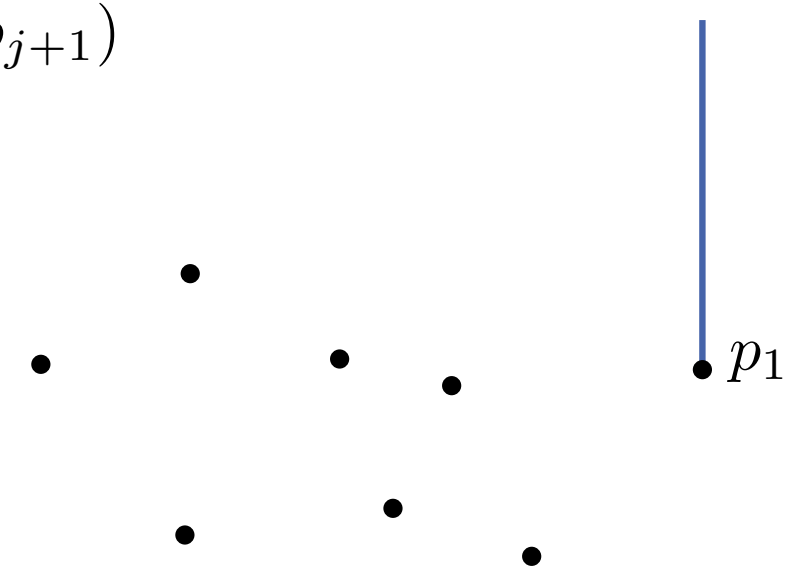
$p_1 = (x_1, y_1) \leftarrow$ rechtester Punkt in $P$; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$

**while** true **do**

$p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$

**if** $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$

**return** $(p_1, \ldots, p_{j+1})$



# Alternativer Ansatz: Gift Wrapping

**Idee:** beginnend mit einem Punkt $p_1$ von $CH(P)$ finde nächste Kante von $CH(P)$ im UZS

GiftWrapping($P$)

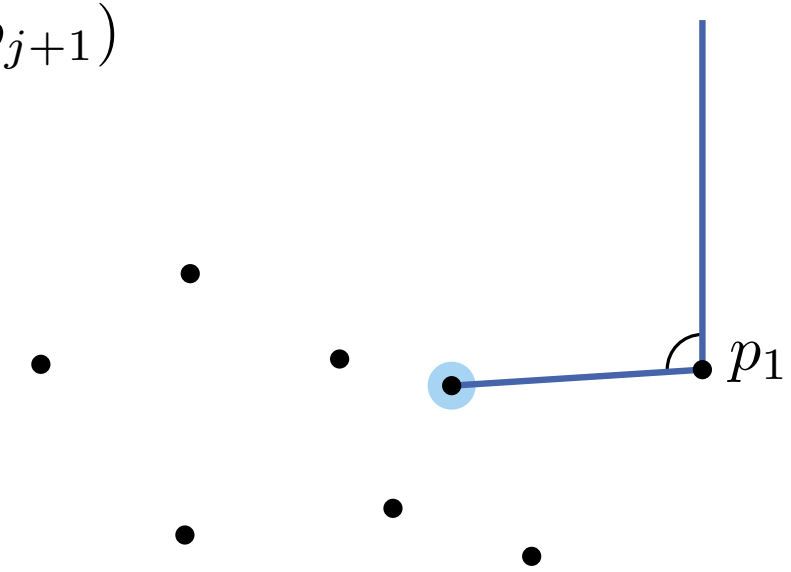
$p_1 = (x_1, y_1) \leftarrow$ rechtester Punkt in $P$; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$

**while** true **do**

$p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$

**if** $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$

**return** $(p_1, \ldots, p_{j+1})$



# Alternativer Ansatz: Gift Wrapping

**Idee:** beginnend mit einem Punkt $p_1$ von $CH(P)$ finde nächste Kante von $CH(P)$ im UZS

GiftWrapping($P$)

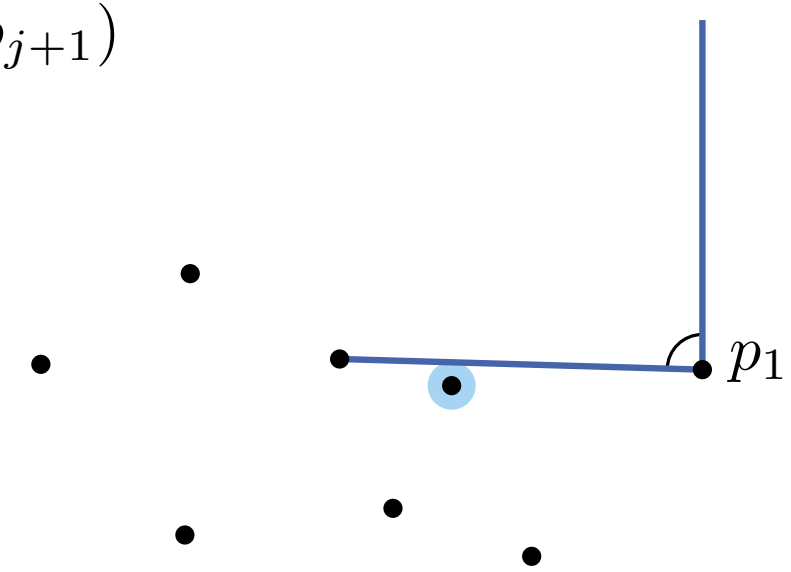
$p_1 = (x_1, y_1) \leftarrow$ rechtester Punkt in $P$; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$

**while** true **do**

$p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$

**if** $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$

**return** $(p_1, \ldots, p_{j+1})$



# Alternativer Ansatz: Gift Wrapping

**Idee:** beginnend mit einem Punkt $p_1$ von $CH(P)$ finde nächste Kante von $CH(P)$ im UZS

GiftWrapping($P$)

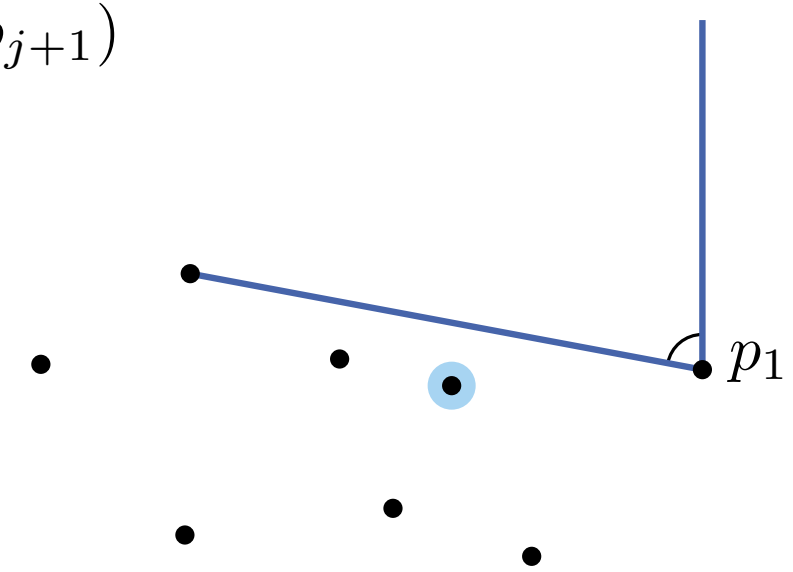
$p_1 = (x_1, y_1) \leftarrow$ rechtester Punkt in $P$; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$

**while** true **do**

$p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$

**if** $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$

**return** $(p_1, \ldots, p_{j+1})$



# Alternativer Ansatz: Gift Wrapping

**Idee:** beginnend mit einem Punkt $p_1$ von $CH(P)$ finde nächste Kante von $CH(P)$ im UZS

GiftWrapping($P$)

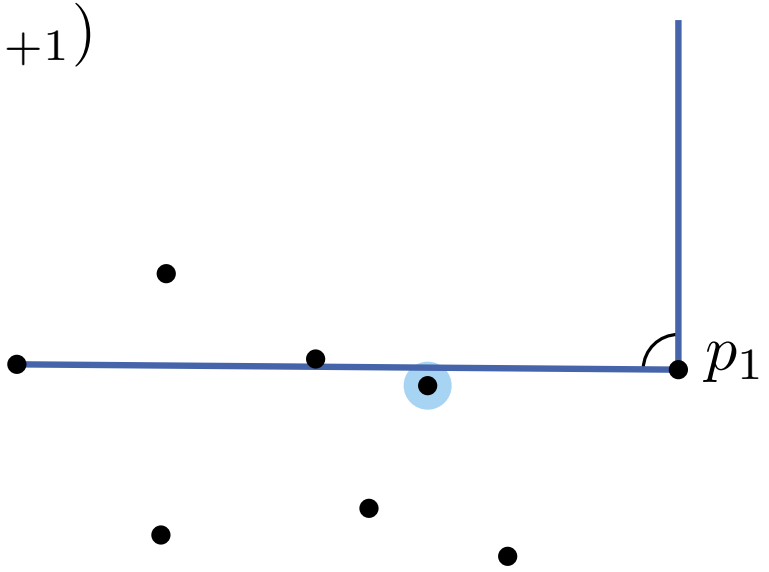
$p_1 = (x_1, y_1) \leftarrow$ rechtester Punkt in $P$; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$

**while** true **do**

$p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$

**if** $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$

**return** $(p_1, \ldots, p_{j+1})$



# Alternativer Ansatz: Gift Wrapping

**Idee:** beginnend mit einem Punkt $p_1$ von $CH(P)$ finde nächste Kante von $CH(P)$ im UZS

GiftWrapping($P$)

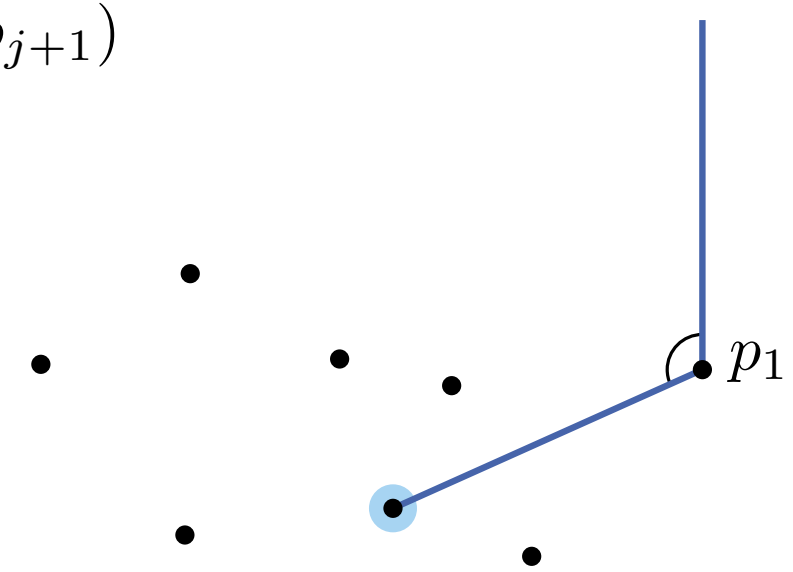
$p_1 = (x_1, y_1) \leftarrow$ rechtester Punkt in $P$; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$

**while** true **do**

$p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$

**if** $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$

**return** $(p_1, \ldots, p_{j+1})$



# Alternativer Ansatz: Gift Wrapping

**Idee:** beginnend mit einem Punkt $p_1$ von $CH(P)$ finde nächste Kante von $CH(P)$ im UZS

GiftWrapping($P$)

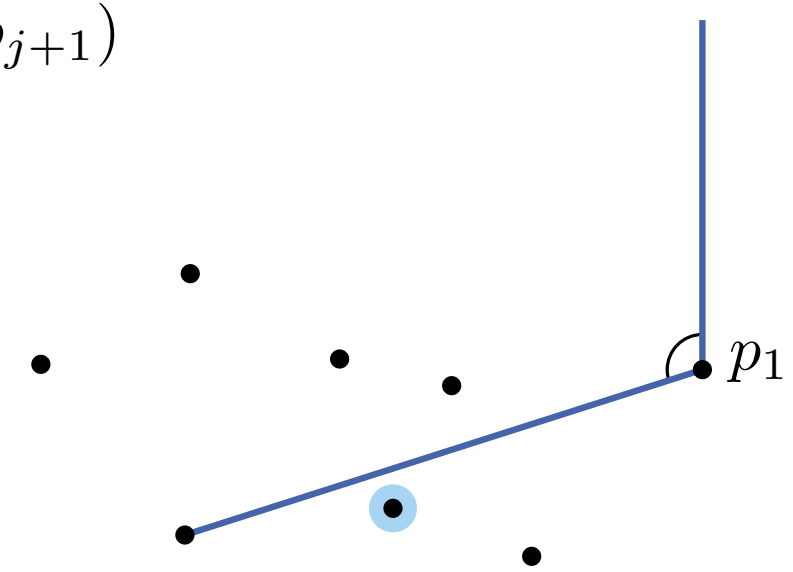
$p_1 = (x_1, y_1) \leftarrow$ rechtester Punkt in $P$; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$

**while** true **do**

$p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$

**if** $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$

**return** $(p_1, \ldots, p_{j+1})$



# Alternativer Ansatz: Gift Wrapping

**Idee:** beginnend mit einem Punkt $p_1$ von $CH(P)$ finde nächste Kante von $CH(P)$ im UZS

GiftWrapping($P$)

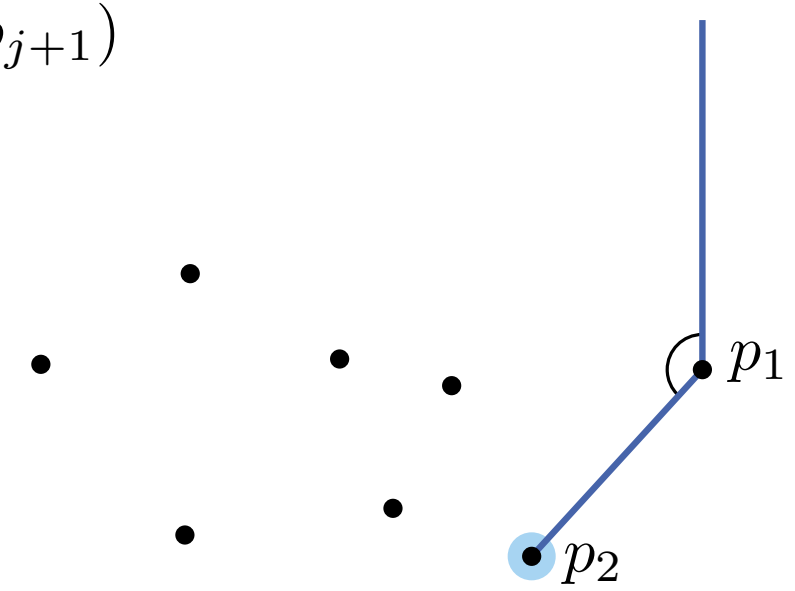
$p_1 = (x_1, y_1) \leftarrow$ rechtester Punkt in $P$; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$

**while** true **do**

$p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$

**if** $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$

**return** $(p_1, \ldots, p_{j+1})$



# Alternativer Ansatz: Gift Wrapping

**Idee:** beginnend mit einem Punkt $p_1$ von $CH(P)$ finde nächste Kante von $CH(P)$ im UZS

GiftWrapping($P$)

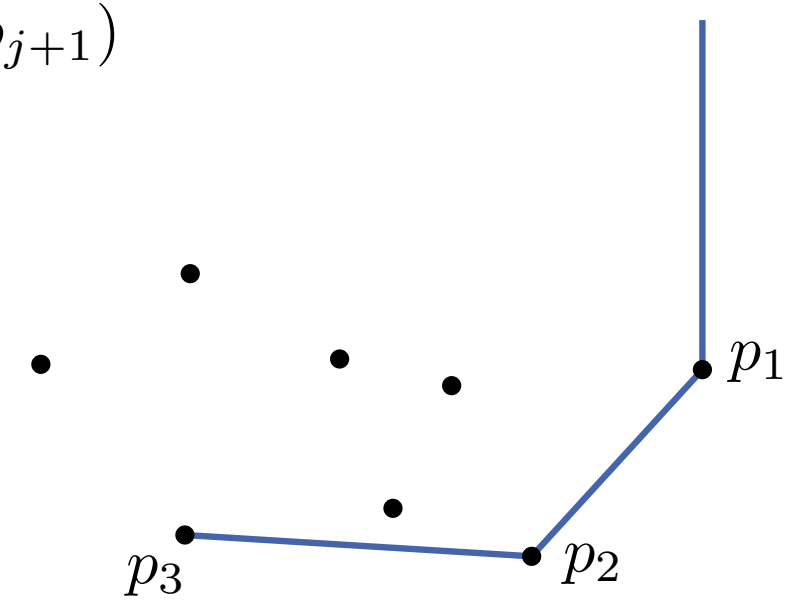
$p_1 = (x_1, y_1) \leftarrow$ rechtester Punkt in $P$; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$

**while** true **do**

$p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$

**if** $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$

**return** $(p_1, \ldots, p_{j+1})$



# Alternativer Ansatz: Gift Wrapping

**Idee:** beginnend mit einem Punkt $p_1$ von $CH(P)$ finde nächste Kante von $CH(P)$ im UZS

GiftWrapping($P$)

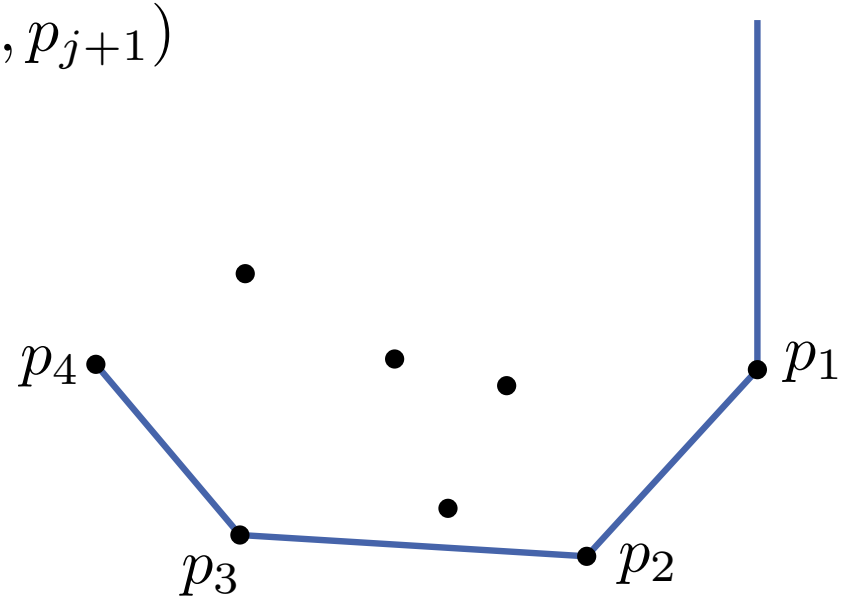
$p_1 = (x_1, y_1) \leftarrow$ rechtester Punkt in $P$; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$

**while** true **do**

$p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$

**if** $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$

**return** $(p_1, \ldots, p_{j+1})$



# Alternativer Ansatz: Gift Wrapping

**Idee:** beginnend mit einem Punkt $p_1$ von $CH(P)$ finde nächste Kante von $CH(P)$ im UZS

GiftWrapping($P$)

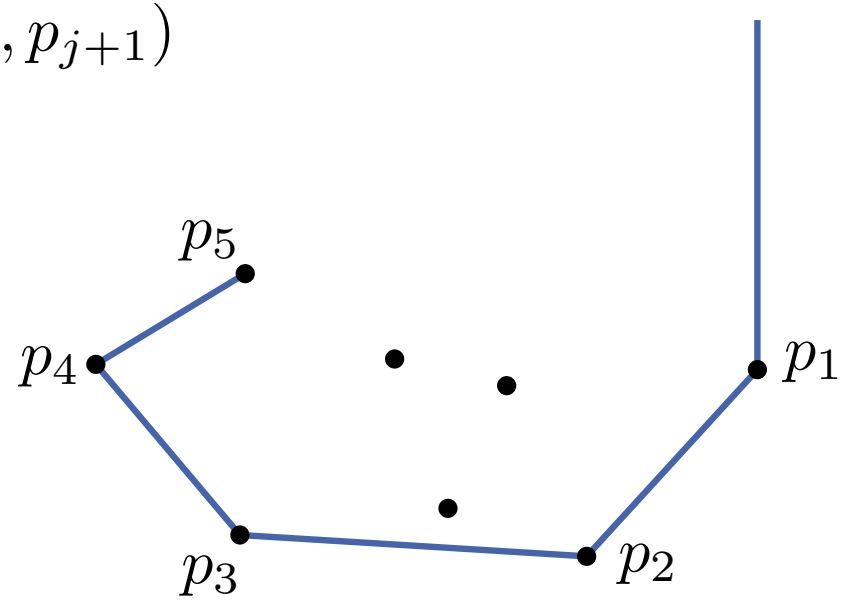
$p_1 = (x_1, y_1) \leftarrow$ rechtester Punkt in $P$; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$

**while** true **do**

$p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$

**if** $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$

**return** $(p_1, \ldots, p_{j+1})$



# Alternativer Ansatz: Gift Wrapping

**Idee:** beginnend mit einem Punkt $p_1$ von $CH(P)$ finde nächste Kante von $CH(P)$ im UZS

GiftWrapping($P$)

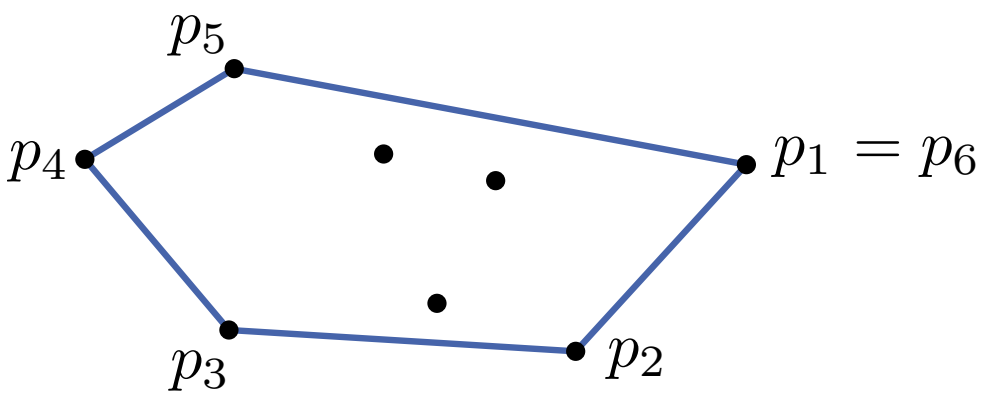
$p_1 = (x_1, y_1) \leftarrow$ rechtester Punkt in $P$; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$

**while** true **do**

$p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$

**if** $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$

**return** $(p_1, \ldots, p_{j+1})$



# Alternativer Ansatz: Gift Wrapping

**Idee:** beginnend mit einem Punkt $p_1$ von $CH(P)$ finde nächste Kante von $CH(P)$ im UZS

GiftWrapping($P$)

$p_1 = (x_1, y_1) \leftarrow$ rechtester Punkt in $P$; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$

**while** true **do**

  $p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$

  **if** $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$

**return** $(p_1, \ldots, p_{j+1})$

**Korrektheit (Beweisidee):**

# Alternativer Ansatz: Gift Wrapping

**Idee:** beginnend mit einem Punkt $p_1$ von $CH(P)$ finde nächste Kante von $CH(P)$ im UZS

GiftWrapping($P$)

$p_1 = (x_1, y_1) \leftarrow$ rechtester Punkt in $P$; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$
**while** true **do**
  $p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$
  **if** $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$
**return** $(p_1, \ldots, p_{j+1})$

## Korrektheit (Beweisidee):

- IA: $p_1$ liegt auf der konvexen Hülle

# Alternativer Ansatz: Gift Wrapping

**Idee:** beginnend mit einem Punkt $p_1$ von $CH(P)$ finde nächste Kante von $CH(P)$ im UZS

GiftWrapping($P$)

$p_1 = (x_1, y_1) \leftarrow$ rechtester Punkt in $P$; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$
**while** true **do**
  $p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$
  **if** $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$
**return** $(p_1, \ldots, p_{j+1})$

## Korrektheit (Beweisidee):

- IA: $p_1$ liegt auf der konvexen Hülle
- IV: Ersten $i$ Punkte sind die ersten $i$ Punkte der CH($P$)

# Alternativer Ansatz: Gift Wrapping

**Idee:** beginnend mit einem Punkt $p_1$ von $CH(P)$ finde nächste Kante von $CH(P)$ im UZS

GiftWrapping($P$)

$p_1 = (x_1, y_1) \leftarrow$ rechtester Punkt in $P$; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$
**while** true **do**
  $p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$
  **if** $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$
**return** $(p_1, \ldots, p_{j+1})$

## Korrektheit (Beweisidee):

- IA: $p_1$ liegt auf der konvexen Hülle
- IV: Ersten $i$ Punkte sind die ersten $i$ Punkte der CH($P$)
- IS: Nach IV liegt $p_{i+1}$ rechts der Geraden $\overrightarrow{p_{i-1}p_i} \Rightarrow$ 'Rechtsknick'
- Nach Wahl des Winkels: alle Punkte liegen rechts der Geraden $\overrightarrow{p_ip_{i+1}}$

# Alternativer Ansatz: Gift Wrapping

**Idee:** beginnend mit einem Punkt $p_1$ von $CH(P)$ finde nächste Kante von $CH(P)$ im UZS

GiftWrapping($P$)

$p_1 = (x_1, y_1) \leftarrow$ rechtester Punkt in $P$; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$

**while** true **do**

  $p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$

  **if** $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$

**return** $(p_1, \ldots, p_{j+1})$

## Degenerierte Fälle:

# Alternativer Ansatz: Gift Wrapping

**Idee:** beginnend mit einem Punkt $p_1$ von $CH(P)$ finde nächste Kante von $CH(P)$ im UZS

GiftWrapping($P$)

$p_1 = (x_1, y_1) \leftarrow$ rechtester Punkt in $P$; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$

**while** true **do**

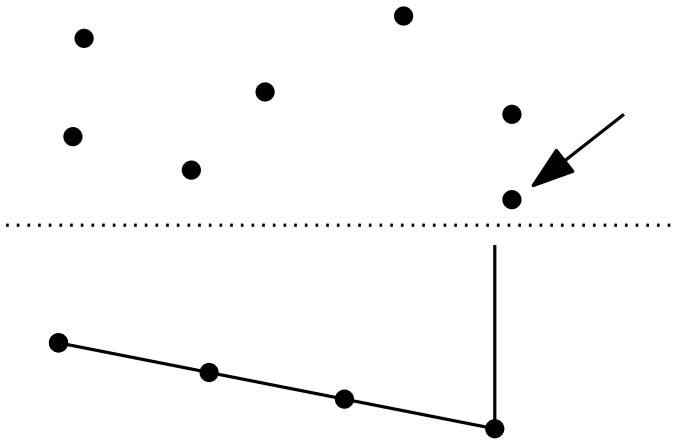
$p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$

**if** $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$

**return** $(p_1, \ldots, p_{j+1})$

## Degenerierte Fälle:

1. Wahl von $p_1$ ist nicht eindeutig.
   Wähle unteresten der rechtesten Punkte.
2. Wahl von $p_{j+1}$ ist nicht eindeutig.
   Wähle den am weitesten entfernten Punkt.



# Tangentenberechnung

**geg:** konvexe Polygon $P$ (im Uhrzeigersinn) und Punkt $p$ außerhalb von $P$
**ges:** *rechte* Tangente an $P$ durch $p$ in $O(\log n)$ Zeit.

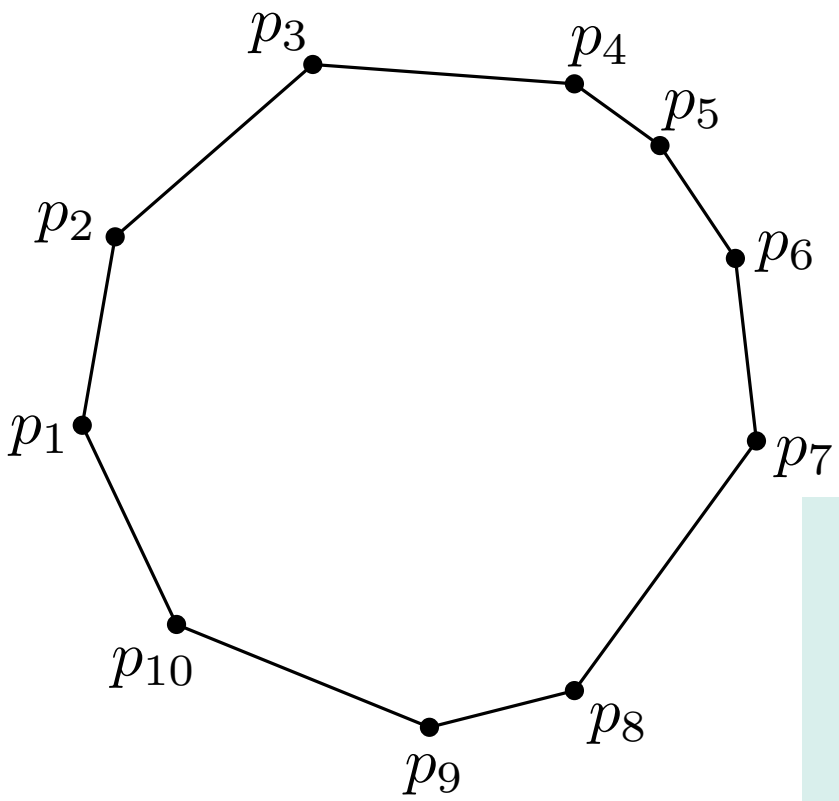
Betrachte hier rechte Tangente, d.h. Polygon ist links von Tangente.

# Tangentenberechnung

**geg:** konvexe Polygon $P$ (im Uhrzeigersinn) und Punkt $p$ außerhalb von $P$
**ges:** *rechte* Tangente an $P$ durch $p$ in $O(\log n)$ Zeit.

**Idee:** Verwende eine binäre Suche.

Betrachte hier rechte Tangente, d.h. Polygon ist links von Tangente.





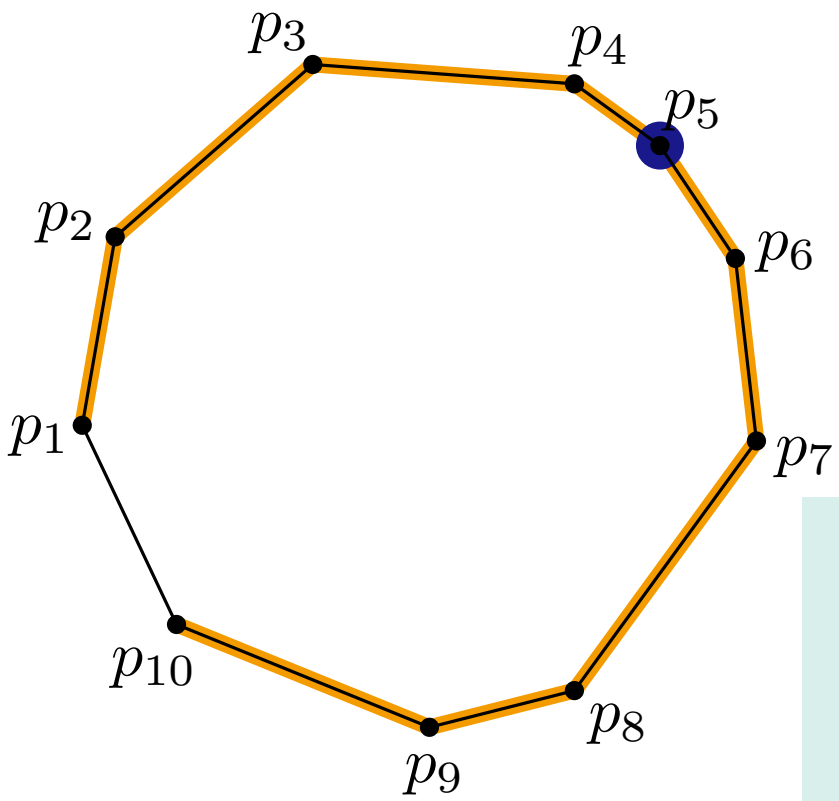
$[a, b] \leftarrow [1, n]$
**solange** Tangente nicht gefunden **tue**
  Bestimme Mitte $c = \lfloor \frac{a+b}{2} \rfloor$
  **wenn** $\overline{pp_c}$ ist Tangente **dann return** $p_c$
  **wenn** $[c, b]$ enthält Index des Berührpunkts **dann**
    $[a, b] \leftarrow [c, b]$
  **sonst**
    $[a, b] \leftarrow [a, c]$

# Tangentenberechnung

**geg:** konvexe Polygon $P$ (im Uhrzeigersinn) und Punkt $p$ außerhalb von $P$
**ges:** *rechte* Tangente an $P$ durch $p$ in $O(\log n)$ Zeit.

**Idee:** Verwende eine binäre Suche.

Betrachte hier rechte Tangente, d.h. Polygon ist links von Tangente.





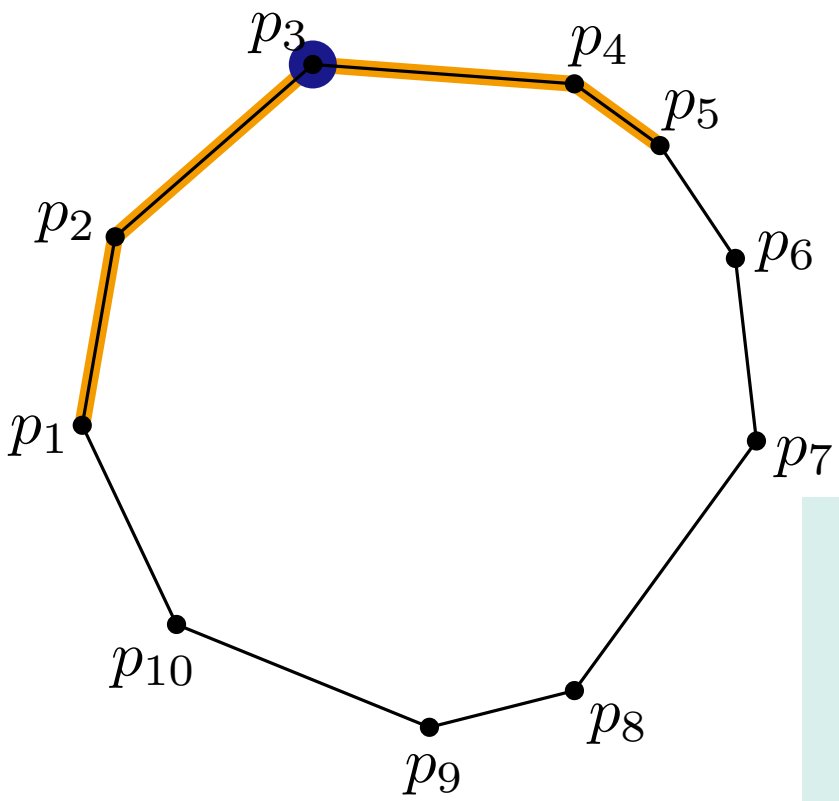
$[a, b] \leftarrow [1, n]$
**solange** Tangente nicht gefunden **tue**
  Bestimme Mitte $c = \lfloor \frac{a+b}{2} \rfloor$
  **wenn** $\overline{pp_c}$ ist Tangente **dann return** $p_c$
  **wenn** $[c, b]$ enthält Index des Berührpunkts **dann**
    $[a, b] \leftarrow [c, b]$
  **sonst**
    $[a, b] \leftarrow [a, c]$

# Tangentenberechnung

**geg:** konvexe Polygon $P$ (im Uhrzeigersinn) und Punkt $p$ außerhalb von $P$
**ges:** *rechte* Tangente an $P$ durch $p$ in $O(\log n)$ Zeit.

**Idee:** Verwende eine binäre Suche.

Betrachte hier rechte Tangente, d.h. Polygon ist links von Tangente.



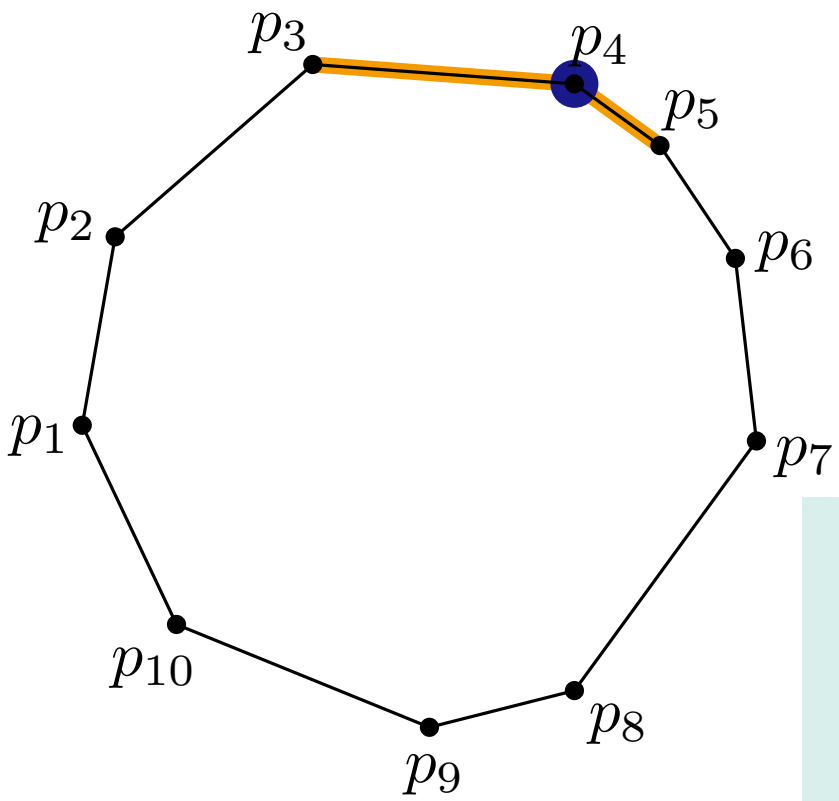


$[a, b] \leftarrow [1, n]$
**solange** Tangente nicht gefunden **tue**
  Bestimme Mitte $c = \lfloor \frac{a+b}{2} \rfloor$
  **wenn** $\overline{pp_c}$ ist Tangente **dann return** $p_c$
  **wenn** $[c, b]$ enthält Index des Berührpunkts **dann**
    $[a, b] \leftarrow [c, b]$
  **sonst**
    $[a, b] \leftarrow [a, c]$

# Tangentenberechnung

**geg:** konvexe Polygon $P$ (im Uhrzeigersinn) und Punkt $p$ außerhalb von $P$
**ges:** *rechte* Tangente an $P$ durch $p$ in $O(\log n)$ Zeit.

**Idee:** Verwende eine binäre Suche.




Betrachte hier rechte Tangente, d.h. Polygon ist links von Tangente.

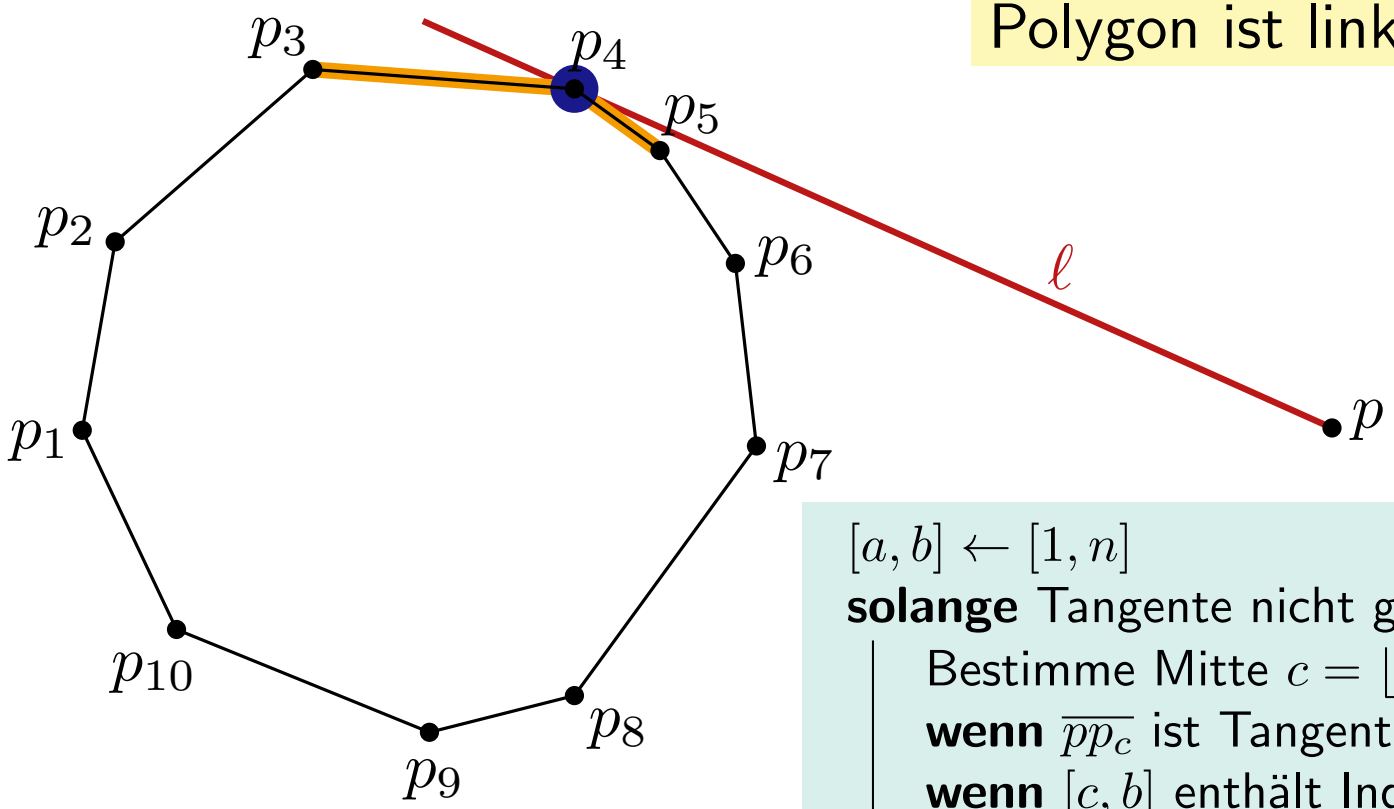

$[a, b] \leftarrow [1, n]$
**solange** Tangente nicht gefunden **tue**
  Bestimme Mitte $c = \lfloor \frac{a+b}{2} \rfloor$
  **wenn** $\overline{pp_c}$ ist Tangente **dann return** $p_c$
  **wenn** $[c, b]$ enthält Index des Berührpunkts **dann**
    $[a, b] \leftarrow [c, b]$
  **sonst**
    $[a, b] \leftarrow [a, c]$

# Tangentenberechnung

**geg:** konvexe Polygon $P$ (im Uhrzeigersinn) und Punkt $p$ außerhalb von $P$
**ges:** *rechte* Tangente an $P$ durch $p$ in $O(\log n)$ Zeit.

**Idee:** Verwende eine binäre Suche.

Betrachte hier rechte Tangente, d.h. Polygon ist links von Tangente.



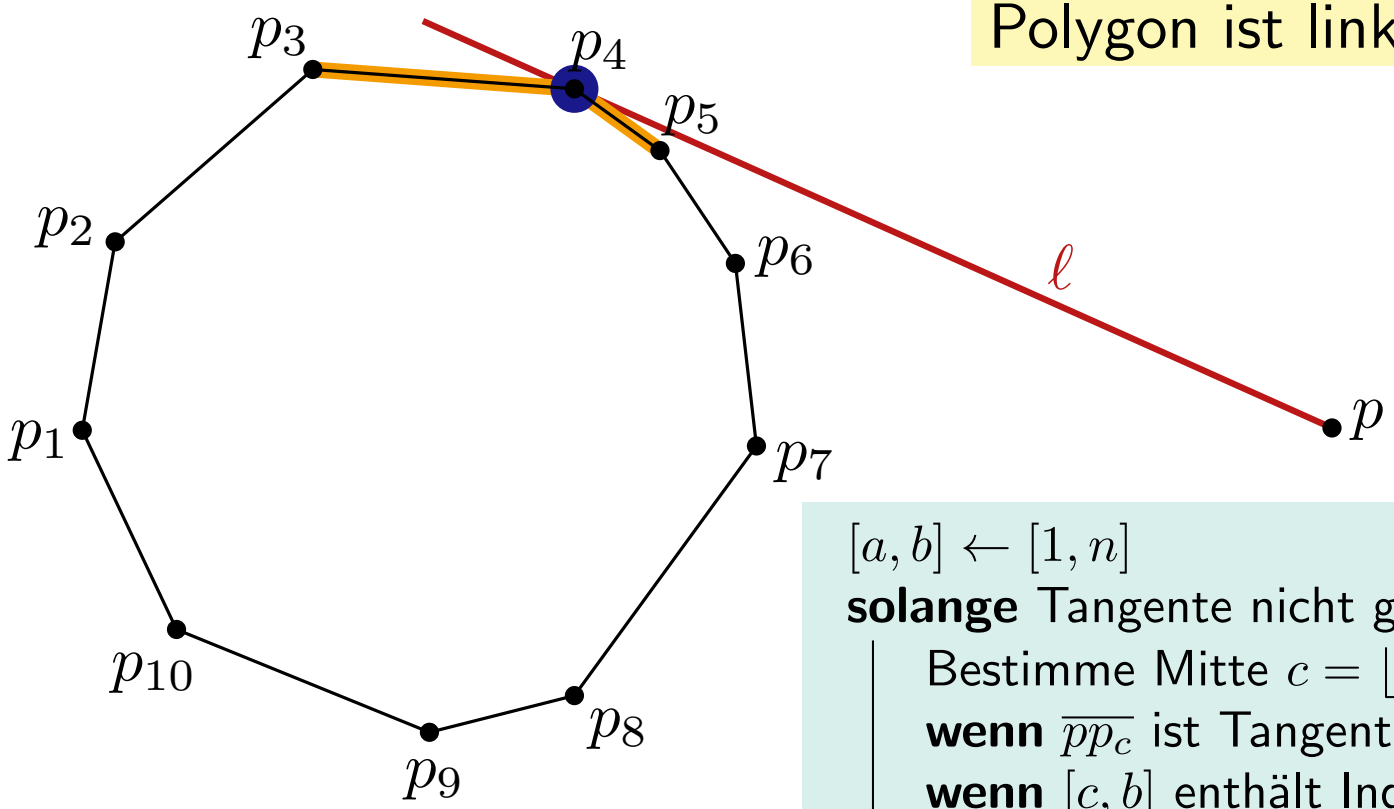

$[a, b] \leftarrow [1, n]$
**solange** Tangente nicht gefunden **tue**
  Bestimme Mitte $c = \lfloor \frac{a+b}{2} \rfloor$
  **wenn** $\overline{pp_c}$ ist Tangente **dann return** $p_c$
  **wenn** $[c, b]$ enthält Index des Berührpunkts **dann**
    $[a, b] \leftarrow [c, b]$
  **sonst**
    $[a, b] \leftarrow [a, c]$

# Tangentenberechnung

**geg:** konvexe Polygon $P$ (im Uhrzeigersinn) und Punkt $p$ außerhalb von $P$
**ges:** *rechte* Tangente an $P$ durch $p$ in $O(\log n)$ Zeit.

**Idee:** Verwende eine binäre Suche.

Betrachte hier rechte Tangente, d.h. Polygon ist links von Tangente.



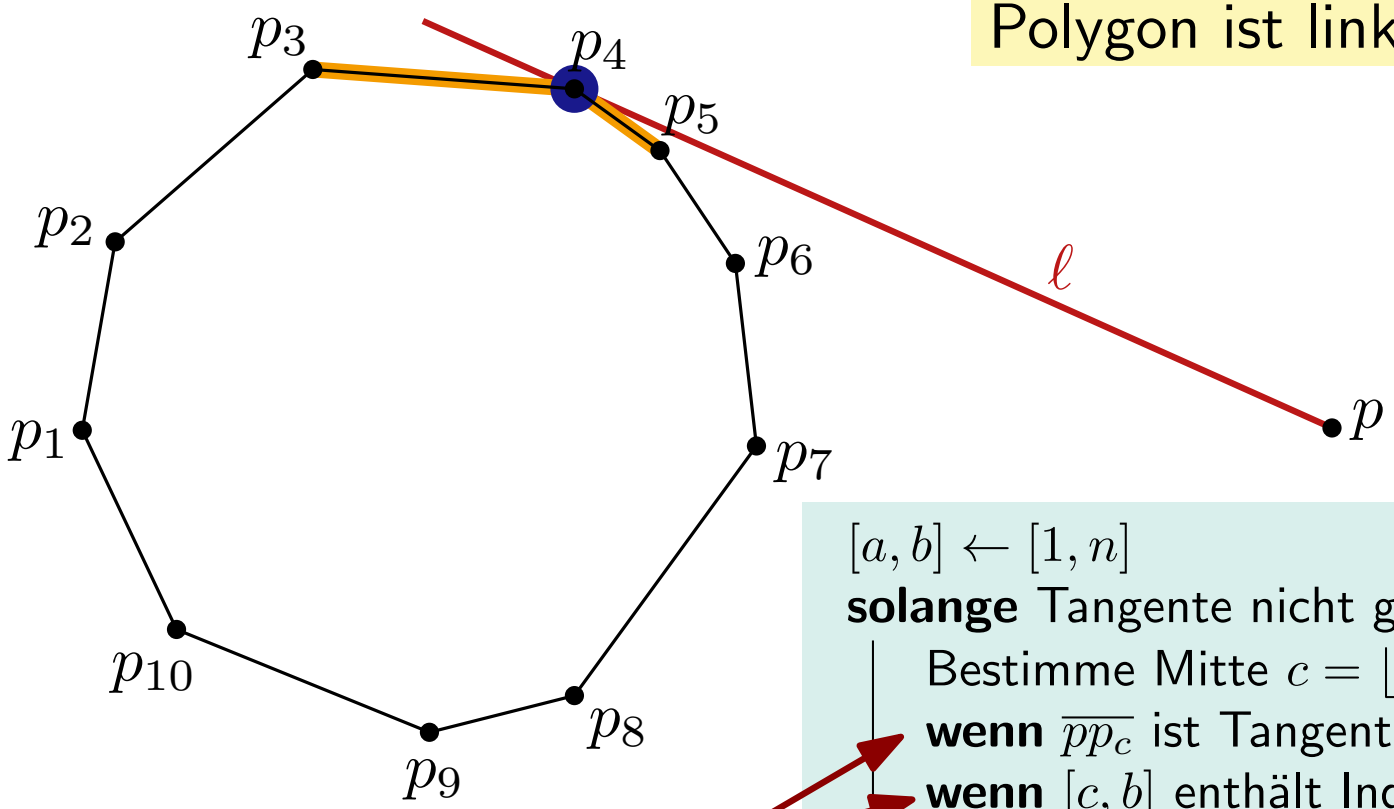

$[a, b] \leftarrow [1, n]$
**solange** Tangente nicht gefunden **tue**
  Bestimme Mitte $c = \lfloor \frac{a+b}{2} \rfloor$
  **wenn** $\overline{pp_c}$ ist Tangente **dann return** $p_c$
  **wenn** $[c, b]$ enthält Index des Berührpunkts **dann**
    $[a, b] \leftarrow [c, b]$
  **sonst**
    $[a, b] \leftarrow [a, c]$

# Tangentenberechnung

**geg:** konvexe Polygon $P$ (im Uhrzeigersinn) und Punkt $p$ außerhalb von $P$
**ges:** *rechte* Tangente an $P$ durch $p$ in $O(\log n)$ Zeit.

**Idee:** Verwende eine binäre Suche.

Betrachte hier rechte Tangente, d.h. Polygon ist links von Tangente.

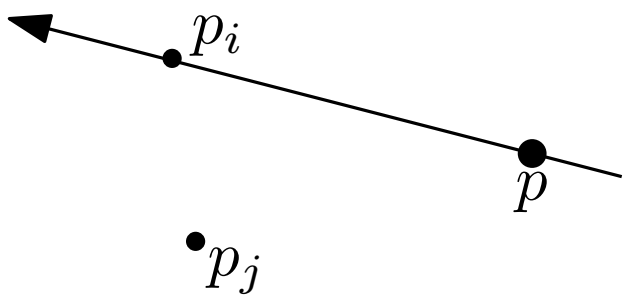



$[a, b] \leftarrow [1, n]$
**solange** Tangente nicht gefunden **tue**
  Bestimme Mitte $c = \lfloor \frac{a+b}{2} \rfloor$
  **wenn** $\overline{pp_c}$ ist Tangente **dann return** $p_c$
  **wenn** $[c, b]$ enthält Index des Berührpunkts **dann**
    $[a, b] \leftarrow [c, b]$
  **sonst**
    $[a, b] \leftarrow [a, c]$

Wie testen?

# Tangentenberechnung

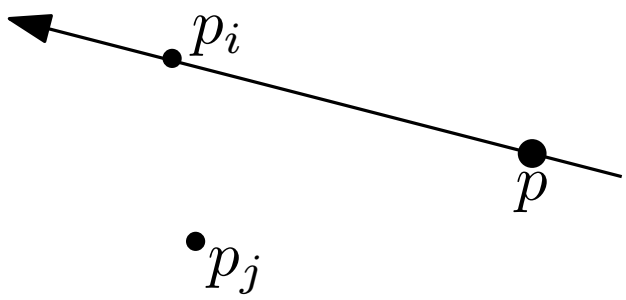
$[a, b] \leftarrow [1, n]$
**solange** Tangente nicht gefunden **tue**
  Bestimme Mitte $c = \lfloor \frac{a+b}{2} \rfloor$
  **wenn** $\overline{pp_c}$ ist Tangente **dann return** $p_c$
  **wenn** $[c, b]$ enthält Index des Berührpunkts **dann**
    $[a, b] \leftarrow [c, b]$
  **sonst**
    $[a, b] \leftarrow [a, c]$




$p_i$ liegt oberhalb von $p_j$, wenn $p_j$ links von $\overrightarrow{pp_i}$ liegt.

# Tangentenberechnung

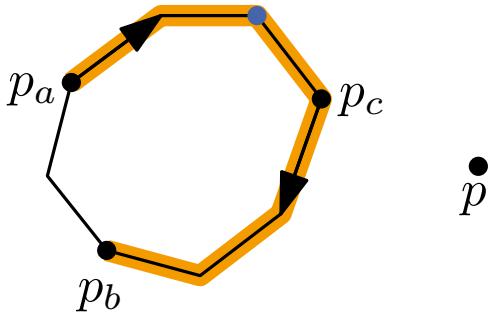
$[a, b] \leftarrow [1, n]$
**solange** Tangente nicht gefunden **tue**
  Bestimme Mitte $c = \lfloor \frac{a+b}{2} \rfloor$
  **wenn** $\overline{pp_c}$ ist Tangente **dann return** $p_c$
  **wenn** $[c, b]$ enthält Index des Berührpunkts **dann**
    $[a, b] \leftarrow [c, b]$
  **sonst**
    $[a, b] \leftarrow [a, c]$




$p_i$ liegt oberhalb von $p_j$, wenn $p_j$ links von $\overrightarrow{pp_i}$ liegt.

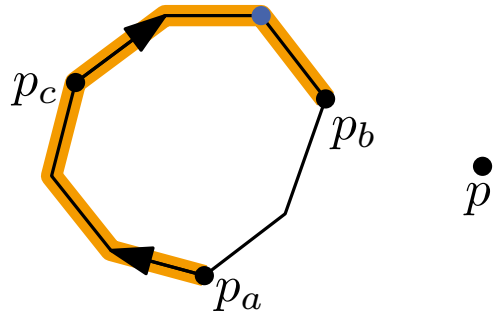
**$p_{a+1}$ oberhalb von $p_a$:**

$p_c$ oberhalb von $p_{c+1}$



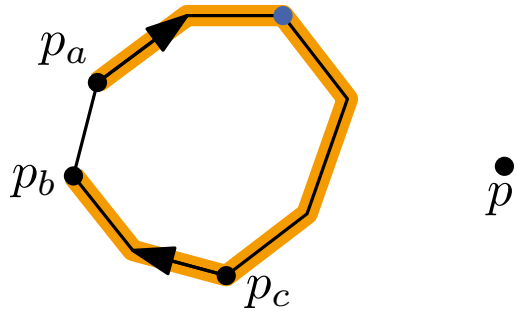

$[a, b] \leftarrow [a, c]$

$p_{c+1}$ oberhalb von $p_c$
$p_c$ oberhalb von $p_a$




$[a, b] \leftarrow [c, b]$

$p_{c+1}$ oberhalb von $p_c$
$p_a$ oberhalb von $p_c$




$[a, b] \leftarrow [a, c]$

$p_a$ oberhalb von $p_{a+1}$ kann analog betrachtet werden.

# Aufgabe 4

Von einem Algorithmus, der die konvexe Hülle einer gegebenen Punktmenge berechnet, fordern wir, dass er die Punkte als (im Uhrzeigersinn) sortierte Liste ausgibt.

# Aufgabe 4

Karlsruhe Institute of Technology

Von einem Algorithmus, der die konvexe Hülle einer gegebenen Punktmenge berechnet, fordern wir, dass er die Punkte als (im Uhrzeigersinn) sortierte Liste ausgibt.

a) Berechnung konvexer Hülle benötigt $\Omega(n \log n)$.

# Aufgabe 4

Karlsruhe Institute of Technology

Von einem Algorithmus, der die konvexe Hülle einer gegebenen Punktmenge berechnet, fordern wir, dass er die Punkte als (im Uhrzeigersinn) sortierte Liste ausgibt.

a) Berechnung konvexer Hülle benötigt $\Omega(n \log n)$.

b) Warum ist Teilaufgabe a) kein Widerspruch zu Gift Wrapping?

# Aufgabe 4

c) **Gegeben:** einfaches [nicht notwendigerweise kovexes] Polygon. Konstruiere daraus in $\mathcal{O}(n)$ seine konvexe Hülle.

# Aufgabe 4

c) **Gegeben:** einfaches [nicht notwendigerweise kovexes] Polygon. Konstruiere daraus in $\mathcal{O}(n)$ seine konvexe Hülle.

**Idee:**

$P_1, P_2 \leftarrow$ trenne Polygon zwischen linkesten und rechtesten Punkt in zwei Pfade auf. // Sei $P_1$ *oberer Pfad* und $P_2$ *unterer* Pfad.
UpperConvexHull($P_1$)
LowerConvexHull($P_2$)

UpperConvexHull($P$)
  $\langle p_1, p_2, \ldots, p_n \rangle \leftarrow P$
  $L \leftarrow \langle p_1, p_2 \rangle$
  **for** $i \leftarrow 3$ **to** $n$ **do**
    $L$.append($p_i$)
    **while** $|L| > 2$ **and** letzte 3 Punkte in $L$ kein Rechtsknick **do**
      entferne vorletzten Punkt aus $L$
  **return** $L$

keine Sortierung notwendig!

# Aufgabe 4

c) **Gegeben:** einfaches [nicht notwendigerweise kovexes] Polygon. Konstruiere daraus in $\mathcal{O}(n)$ seine konvexe Hülle.

**Idee:**

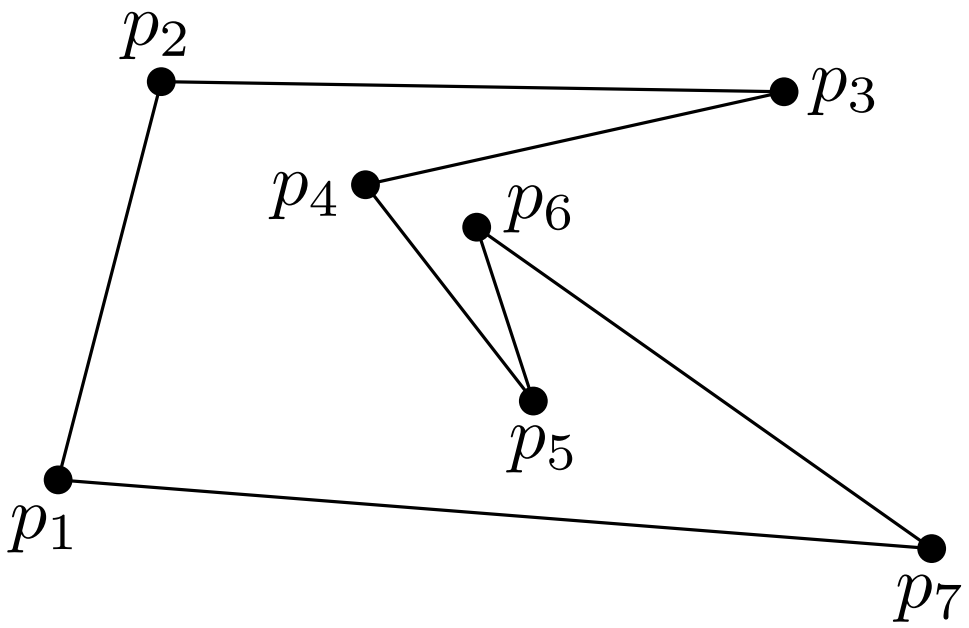
$P_1, P_2 \leftarrow$ trenne Polygon zwischen linkesten und rechtesten Punkt in zwei Pfade auf. // Sei $P_1$ *oberer Pfad* und $P_2$ *unterer* Pfad.
UpperConvexHull($P_1$)
LowerConvexHull($P_2$)

UpperConvexHull($P$)
  $\langle p_1, p_2, \ldots, p_n \rangle \leftarrow P$
  $L \leftarrow \langle p_1, p_2 \rangle$
  **for** $i \leftarrow 3$ **to** $n$ **do**
    $L$.append($p_i$)
    **while** $|L| > 2$ **and** letzte 3 Punkte in $L$ kein Rechtsknick **do**
      entferne vorletzten Punkt aus $L$
  **return** $L$

keine Sortierung notwendig!

*siehe nächste Folie.

# Aufgabe 4




UpperConvexHull($P$)

$\langle p_1, p_2, \ldots, p_n \rangle \leftarrow P$

$L \leftarrow \langle p_1, p_2 \rangle$

**for** $i \leftarrow 3$ **to** $n$ **do**

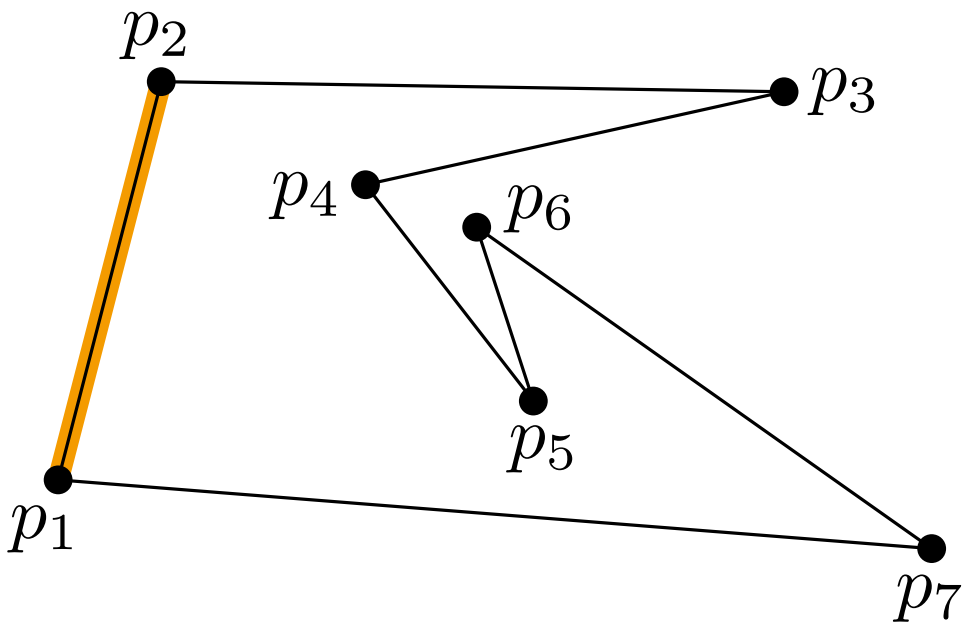
  $L$.append($p_i$)

  **while** $|L| > 2$ **and** letzte 3 Punkte in $L$ kein Rechtsknick **do**

    entferne vorletzten Punkt aus $L$

**return** $L$

# Aufgabe 4




UpperConvexHull($P$)

$\langle p_1, p_2, \ldots, p_n \rangle \leftarrow P$

$L \leftarrow \langle p_1, p_2 \rangle$

**for** $i \leftarrow 3$ **to** $n$ **do**

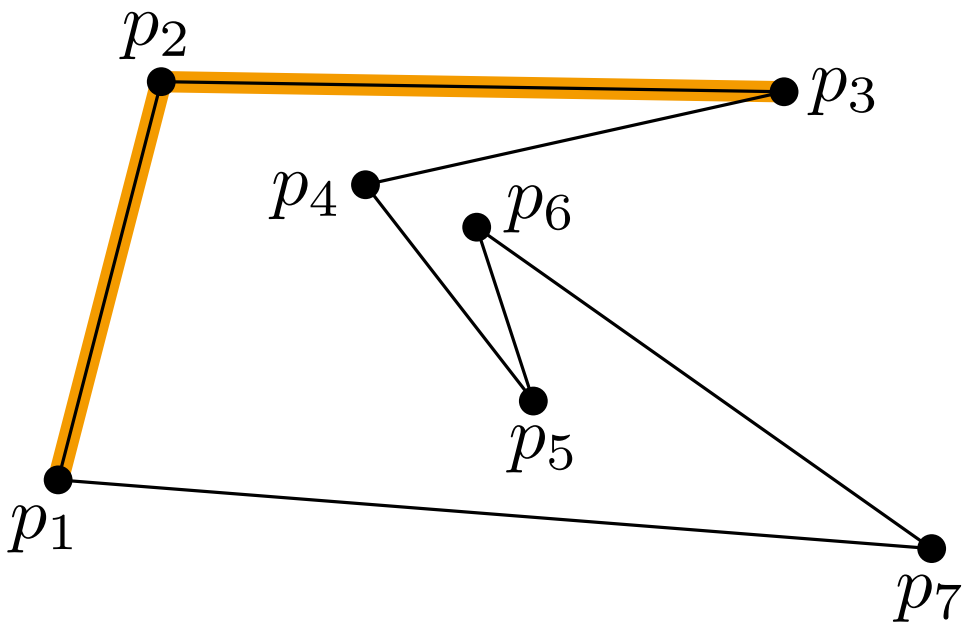
  $L$.append($p_i$)

  **while** $|L| > 2$ **and** letzte 3 Punkte in $L$ kein Rechtsknick **do**

    entferne vorletzten Punkt aus $L$

**return** $L$

# Aufgabe 4




UpperConvexHull($P$)

$\langle p_1, p_2, \ldots, p_n \rangle \leftarrow P$

$L \leftarrow \langle p_1, p_2 \rangle$

**for** $i \leftarrow 3$ **to** $n$ **do**

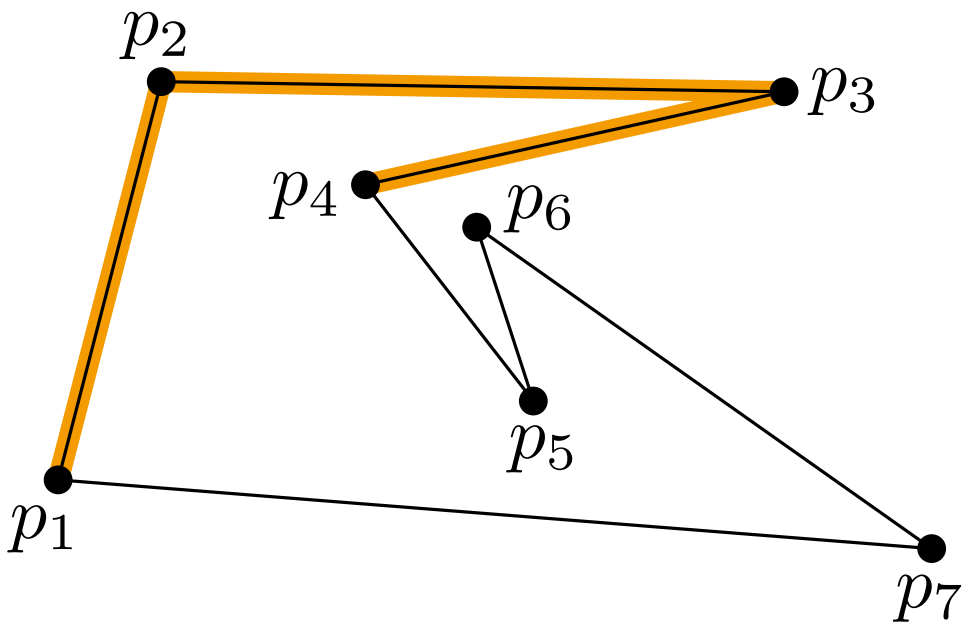
  $L$.append($p_i$)

  **while** $|L| > 2$ **and** letzte 3 Punkte in $L$ kein Rechtsknick **do**

    entferne vorletzten Punkt aus $L$

**return** $L$

# Aufgabe 4




UpperConvexHull($P$)

$\langle p_1, p_2, \ldots, p_n \rangle \leftarrow P$

$L \leftarrow \langle p_1, p_2 \rangle$

**for** $i \leftarrow 3$ **to** $n$ **do**

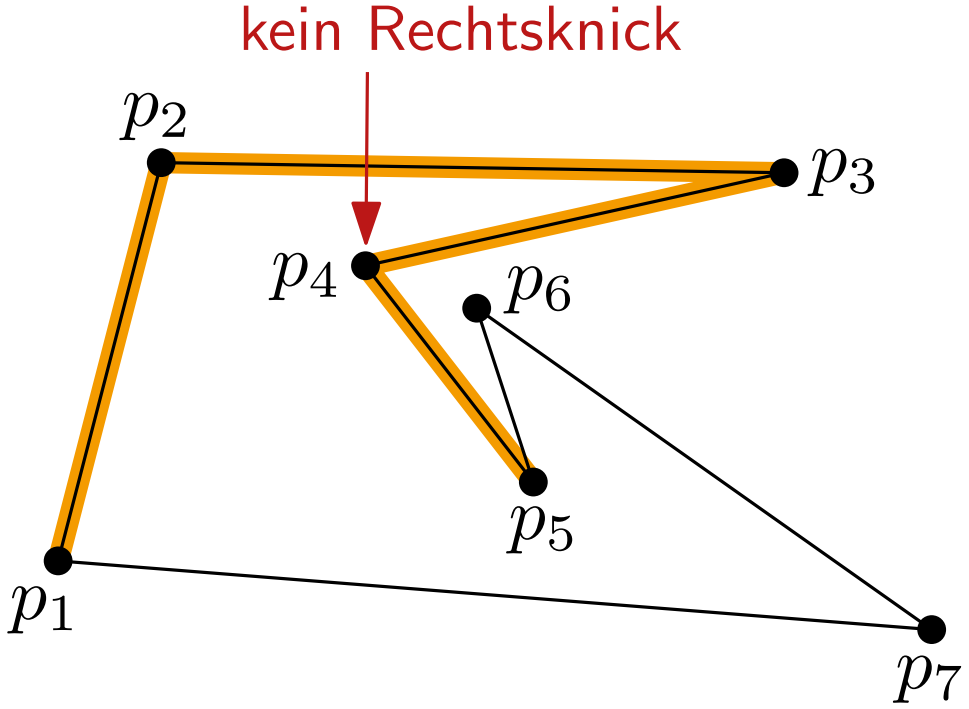
  $L$.append($p_i$)

  **while** $|L| > 2$ **and** letzte 3 Punkte in $L$ kein Rechtsknick **do**

    entferne vorletzten Punkt aus $L$

**return** $L$

# Aufgabe 4




UpperConvexHull($P$)

$\langle p_1, p_2, \ldots, p_n \rangle \leftarrow P$

$L \leftarrow \langle p_1, p_2 \rangle$

**for** $i \leftarrow 3$ **to** $n$ **do**

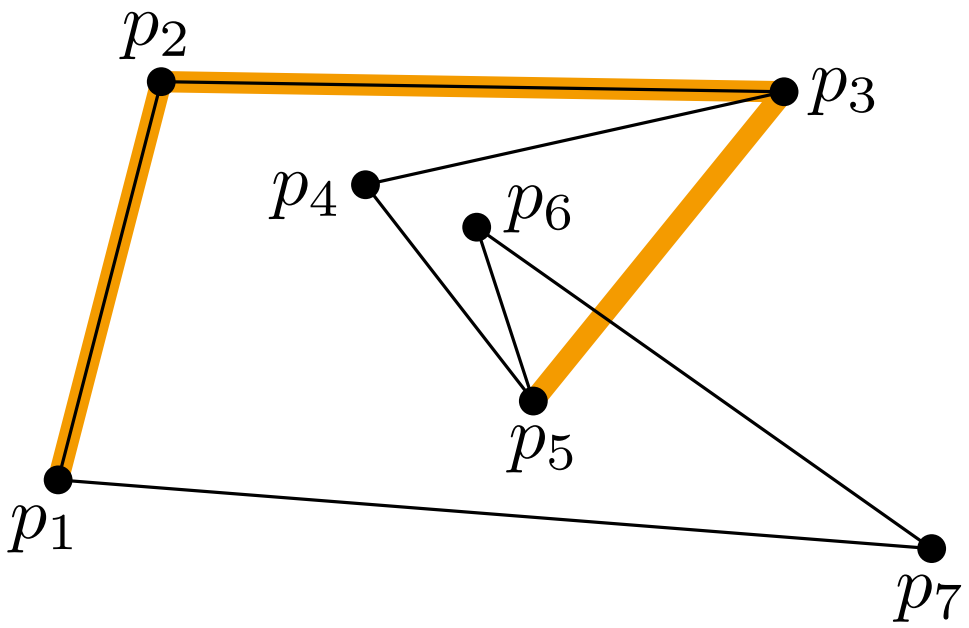
  $L$.append($p_i$)

  **while** $|L| > 2$ **and** letzte 3 Punkte in $L$ kein Rechtsknick **do**

    entferne vorletzten Punkt aus $L$

**return** $L$

# Aufgabe 4




UpperConvexHull($P$)

$\langle p_1, p_2, \ldots, p_n \rangle \leftarrow P$

$L \leftarrow \langle p_1, p_2 \rangle$

**for** $i \leftarrow 3$ **to** $n$ **do**

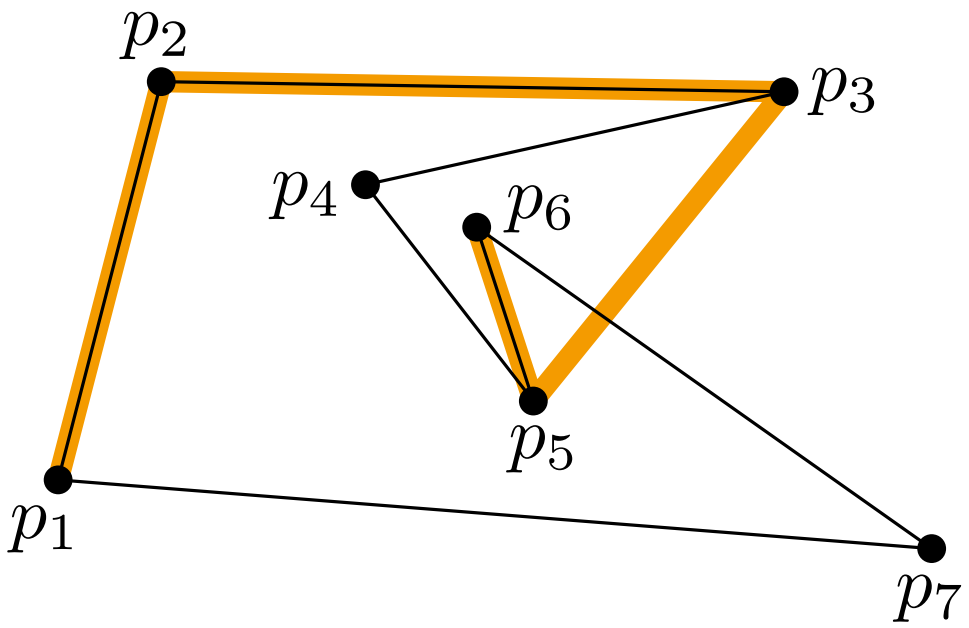
  $L$.append($p_i$)

  **while** $|L| > 2$ **and** letzte 3 Punkte in $L$ kein Rechtsknick **do**

    entferne vorletzten Punkt aus $L$

**return** $L$

# Aufgabe 4




UpperConvexHull($P$)

$\langle p_1, p_2, \ldots, p_n \rangle \leftarrow P$

$L \leftarrow \langle p_1, p_2 \rangle$

**for** $i \leftarrow 3$ **to** $n$ **do**

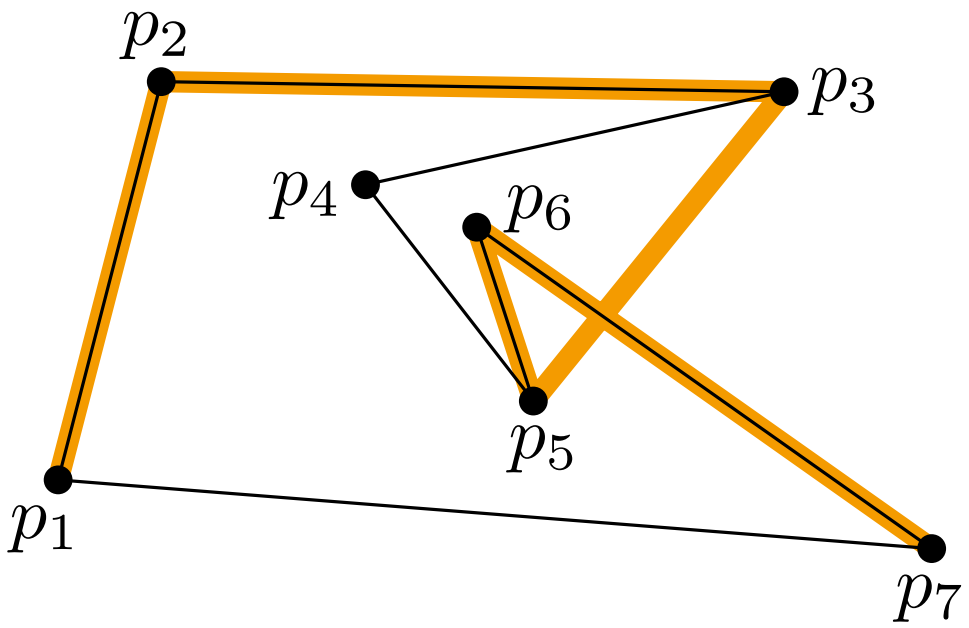
  $L$.append($p_i$)

  **while** $|L| > 2$ **and** letzte 3 Punkte in $L$ kein Rechtsknick **do**

    entferne vorletzten Punkt aus $L$

**return** $L$

# Aufgabe 4




UpperConvexHull($P$)

$\langle p_1, p_2, \ldots, p_n \rangle \leftarrow P$

$L \leftarrow \langle p_1, p_2 \rangle$

**for** $i \leftarrow 3$ **to** $n$ **do**

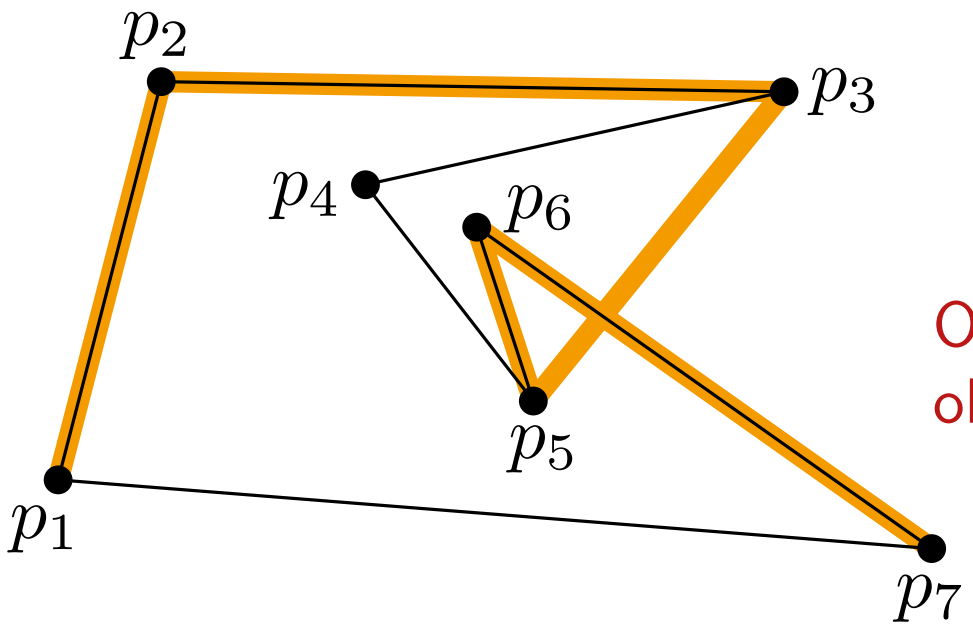
  $L$.append($p_i$)

  **while** $|L| > 2$ **and** letzte 3 Punkte in $L$ kein Rechtsknick **do**

    entferne vorletzten Punkt aus $L$

**return** $L$

# Aufgabe 4




Offensichtlich nicht obere konvexe Hülle!

UpperConvexHull($P$)

$\langle p_1, p_2, \ldots, p_n \rangle \leftarrow P$

$L \leftarrow \langle p_1, p_2 \rangle$

**for** $i \leftarrow 3$ **to** $n$ **do**

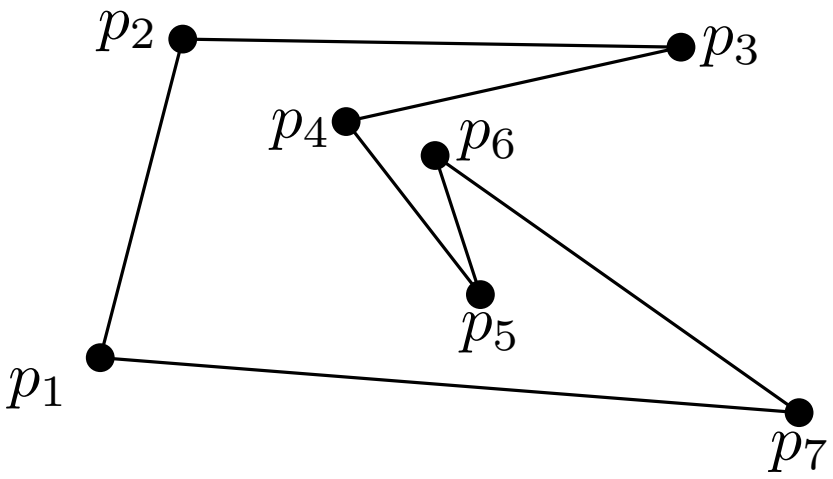
  $L$.append($p_i$)

  **while** $|L| > 2$ **and** letzte 3 Punkte in $L$ kein Rechtsknick **do**

    entferne vorletzten Punkt aus $L$

**return** $L$

# Aufgabe 4 – Melkmans Algorithmus



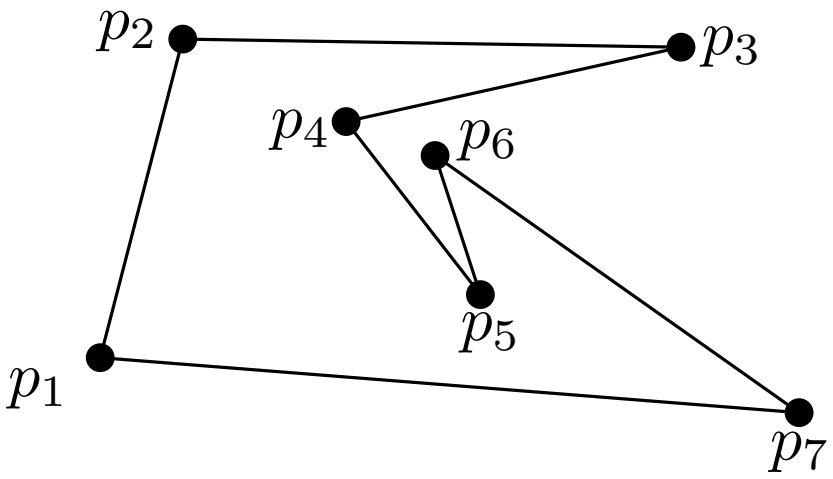

top(Q)=orient. Gerade vom zweiten zum ersten Element aus $Q$.

bottom(Q)=orient. Gerade vom letzten zum vorletzten Element von $Q$.

ConvexHull(einfaches Polygon $P = (p_1, \ldots, p_n)$)

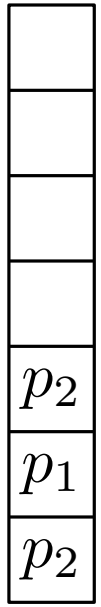
$Q \leftarrow$ double-ended queue $(p_2, p_1, p_2)$
**for** $i = 3, \ldots, n$ **do**
  **while** $p_i$ nicht rechts von top$(Q)$ **do** pop-top$(Q)$
  **while** $p_i$ nicht rechts von bottom$(Q)$ **do** pop-bottom$(Q)$
  **if** popped **then** push-top$(Q, p_i)$; push-bottom$(Q, p_i)$

# Aufgabe 4 – Melkmans Algorithmus




top(Q)=orient. Gerade vom zweiten zum ersten Element aus $Q$.

bottom(Q)=orient. Gerade vom letzten zum vorletzten Element von $Q$.




ConvexHull(einfaches Polygon $P = (p_1, \ldots, p_n)$)

$Q \leftarrow$ double-ended queue $(p_2, p_1, p_2)$

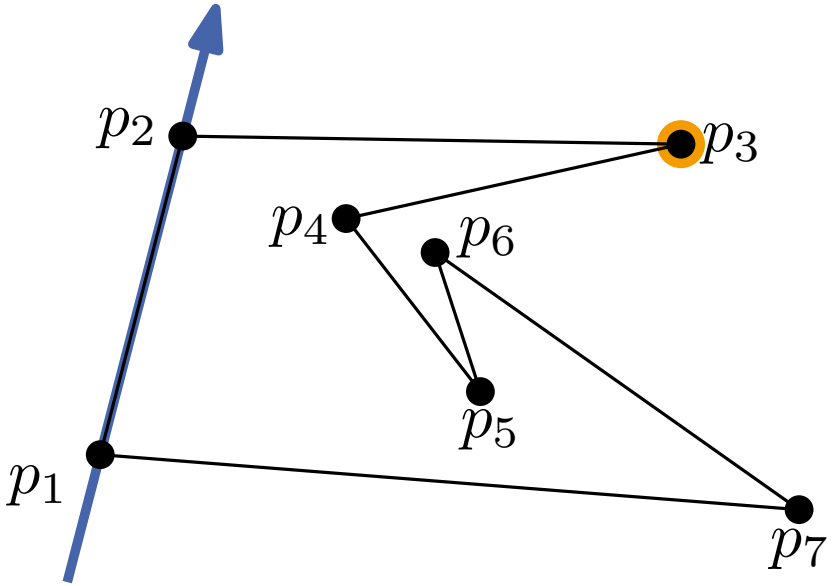
**for** $i = 3, \ldots, n$ **do**

  **while** $p_i$ nicht rechts von top$(Q)$ **do** pop-top$(Q)$

  **while** $p_i$ nicht rechts von bottom$(Q)$ **do** pop-bottom$(Q)$

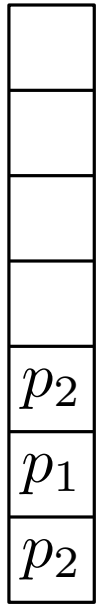
  **if** popped **then** push-top$(Q, p_i)$; push-bottom$(Q, p_i)$

# Aufgabe 4 – Melkmans Algorithmus




top(Q)=orient. Gerade vom zweiten zum ersten Element aus $Q$.

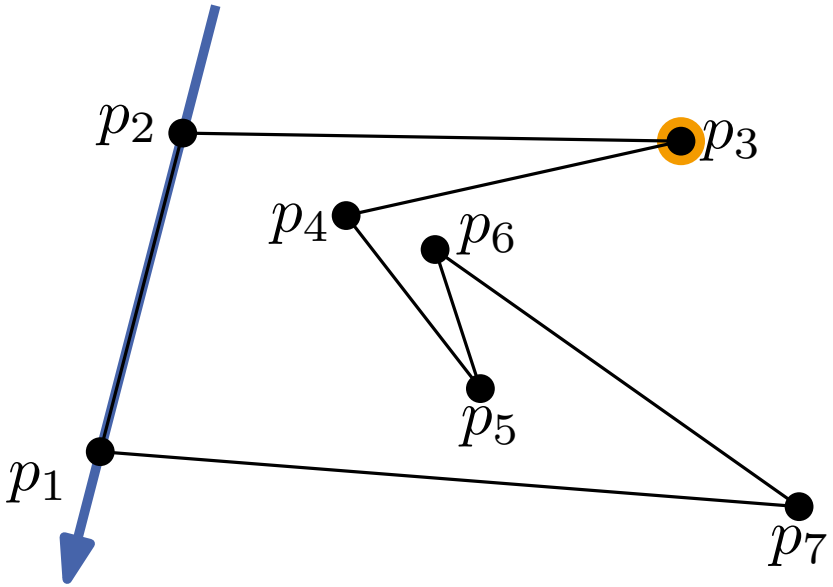
bottom(Q)=orient. Gerade vom letzten zum vorletzten Element von $Q$.




ConvexHull(einfaches Polygon $P = (p_1, \ldots, p_n)$)

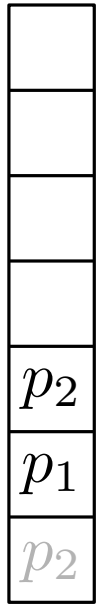
$Q \leftarrow$ double-ended queue $(p_2, p_1, p_2)$
**for** $i = 3, \ldots, n$ **do**
  **while** $p_i$ nicht rechts von top$(Q)$ **do** pop-top$(Q)$
  **while** $p_i$ nicht rechts von bottom$(Q)$ **do** pop-bottom$(Q)$
  **if** popped **then** push-top$(Q, p_i)$; push-bottom$(Q, p_i)$

# Aufgabe 4 – Melkmans Algorithmus




top(Q)=orient. Gerade vom zweiten zum ersten Element aus $Q$.

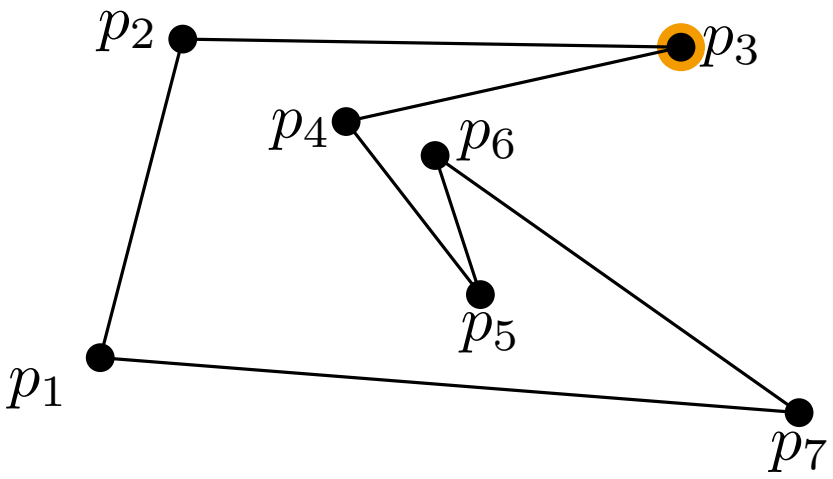
bottom(Q)=orient. Gerade vom letzten zum vorletzten Element von $Q$.




ConvexHull(einfaches Polygon $P = (p_1, \ldots, p_n)$)

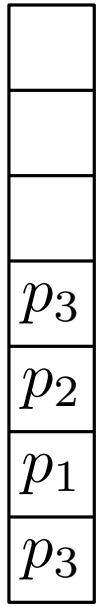
$Q \leftarrow$ double-ended queue $(p_2, p_1, p_2)$
**for** $i = 3, \ldots, n$ **do**
  **while** $p_i$ nicht rechts von top$(Q)$ **do** pop-top$(Q)$
  **while** $p_i$ nicht rechts von bottom$(Q)$ **do** pop-bottom$(Q)$
  **if** popped **then** push-top$(Q, p_i)$; push-bottom$(Q, p_i)$

# Aufgabe 4 – Melkmans Algorithmus




top(Q)=orient. Gerade vom zweiten zum ersten Element aus $Q$.

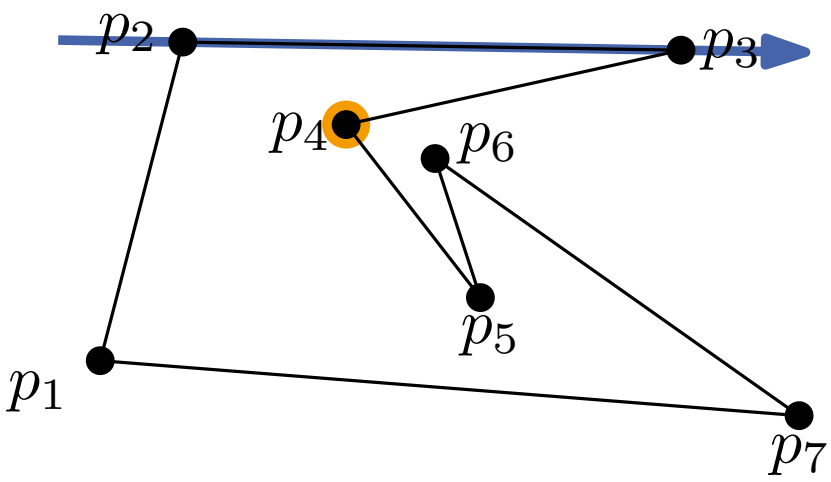
bottom(Q)=orient. Gerade vom letzten zum vorletzten Element von $Q$.




ConvexHull(einfaches Polygon $P = (p_1, \ldots, p_n)$)

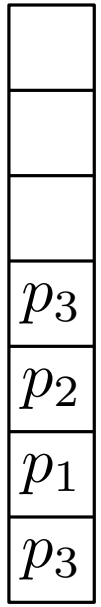
$Q \leftarrow$ double-ended queue $(p_2, p_1, p_2)$
**for** $i = 3, \ldots, n$ **do**
  **while** $p_i$ nicht rechts von top$(Q)$ **do** pop-top$(Q)$
  **while** $p_i$ nicht rechts von bottom$(Q)$ **do** pop-bottom$(Q)$
  **if** popped **then** push-top$(Q, p_i)$; push-bottom$(Q, p_i)$

# Aufgabe 4 – Melkmans Algorithmus




top(Q)=orient. Gerade vom zweiten zum ersten Element aus $Q$.

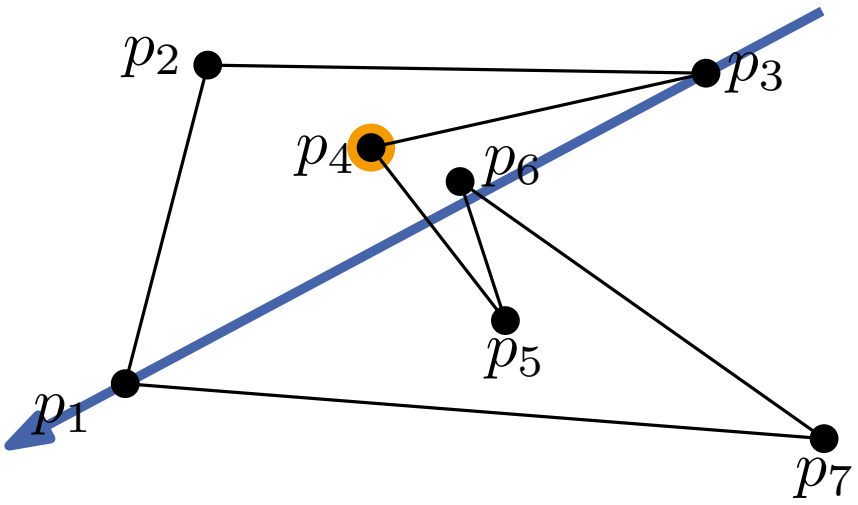
bottom(Q)=orient. Gerade vom letzten zum vorletzten Element von $Q$.




ConvexHull(einfaches Polygon $P = (p_1, \ldots, p_n)$)

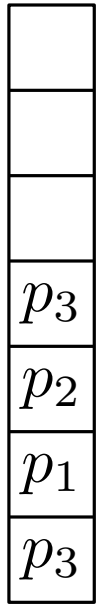
$Q \leftarrow$ double-ended queue $(p_2, p_1, p_2)$
**for** $i = 3, \ldots, n$ **do**
  **while** $p_i$ nicht rechts von top$(Q)$ **do** pop-top$(Q)$
  **while** $p_i$ nicht rechts von bottom$(Q)$ **do** pop-bottom$(Q)$
  **if** popped **then** push-top$(Q, p_i)$; push-bottom$(Q, p_i)$

# Aufgabe 4 – Melkmans Algorithmus




top(Q)=orient. Gerade vom zweiten zum ersten Element aus $Q$.

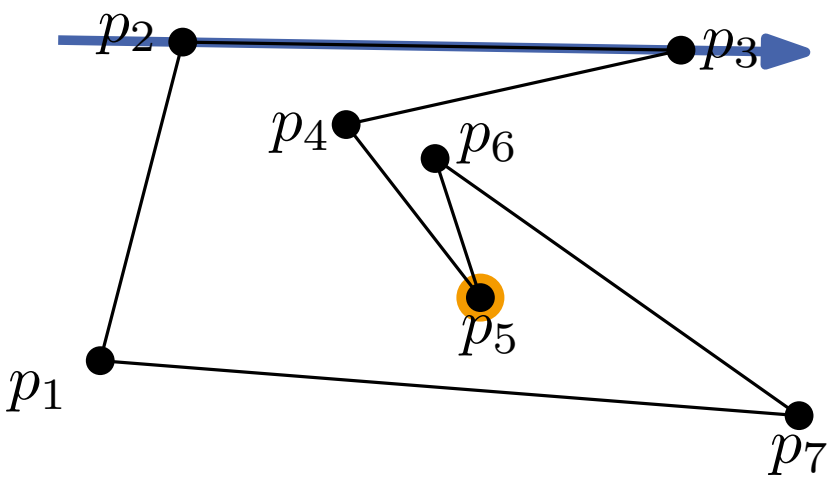
bottom(Q)=orient. Gerade vom letzten zum vorletzten Element von $Q$.




ConvexHull(einfaches Polygon $P = (p_1, \ldots, p_n)$)

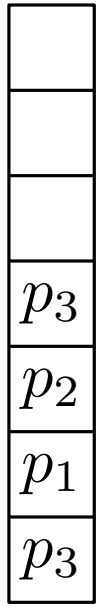
$Q \leftarrow$ double-ended queue $(p_2, p_1, p_2)$
**for** $i = 3, \ldots, n$ **do**
  **while** $p_i$ nicht rechts von top$(Q)$ **do** pop-top$(Q)$
  **while** $p_i$ nicht rechts von bottom$(Q)$ **do** pop-bottom$(Q)$
  **if** popped **then** push-top$(Q, p_i)$; push-bottom$(Q, p_i)$

# Aufgabe 4 – Melkmans Algorithmus




top(Q)=orient. Gerade vom zweiten zum ersten Element aus $Q$.

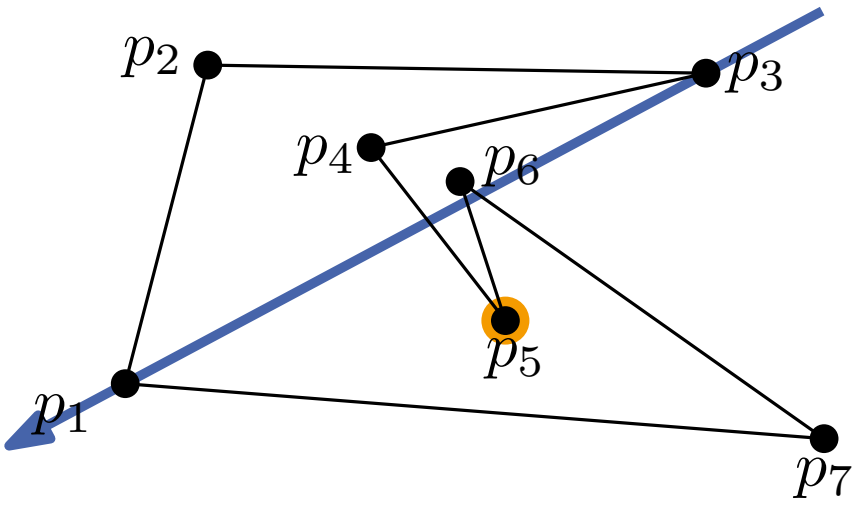
bottom(Q)=orient. Gerade vom letzten zum vorletzten Element von $Q$.




ConvexHull(einfaches Polygon $P = (p_1, \ldots, p_n)$)

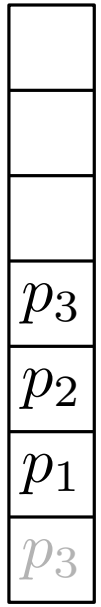
$Q \leftarrow$ double-ended queue $(p_2, p_1, p_2)$
**for** $i = 3, \ldots, n$ **do**
  **while** $p_i$ nicht rechts von top$(Q)$ **do** pop-top$(Q)$
  **while** $p_i$ nicht rechts von bottom$(Q)$ **do** pop-bottom$(Q)$
  **if** popped **then** push-top$(Q, p_i)$; push-bottom$(Q, p_i)$

# Aufgabe 4 – Melkmans Algorithmus




top(Q)=orient. Gerade vom zweiten zum ersten Element aus $Q$.

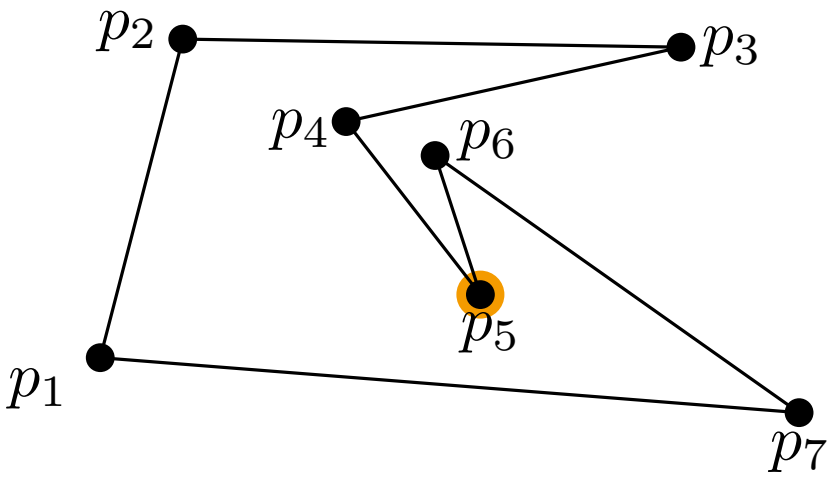
bottom(Q)=orient. Gerade vom letzten zum vorletzten Element von $Q$.




ConvexHull(einfaches Polygon $P = (p_1, \ldots, p_n)$)

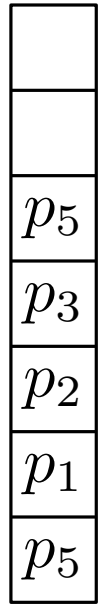
$Q \leftarrow$ double-ended queue $(p_2, p_1, p_2)$
**for** $i = 3, \ldots, n$ **do**
  **while** $p_i$ nicht rechts von top$(Q)$ **do** pop-top$(Q)$
  **while** $p_i$ nicht rechts von bottom$(Q)$ **do** pop-bottom$(Q)$
  **if** popped **then** push-top$(Q, p_i)$; push-bottom$(Q, p_i)$

# Aufgabe 4 – Melkmans Algorithmus




top(Q)=orient. Gerade vom zweiten zum ersten Element aus $Q$.

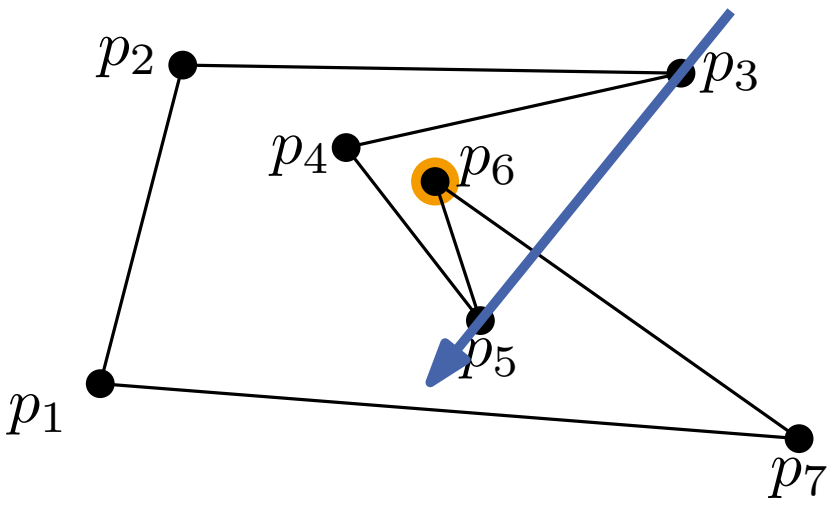
bottom(Q)=orient. Gerade vom letzten zum vorletzten Element von $Q$.




ConvexHull(einfaches Polygon $P = (p_1, \ldots, p_n)$)

$Q \leftarrow$ double-ended queue $(p_2, p_1, p_2)$
**for** $i = 3, \ldots, n$ **do**
  **while** $p_i$ nicht rechts von top$(Q)$ **do** pop-top$(Q)$
  **while** $p_i$ nicht rechts von bottom$(Q)$ **do** pop-bottom$(Q)$
  **if** popped **then** push-top$(Q, p_i)$; push-bottom$(Q, p_i)$

# Aufgabe 4 – Melkmans Algorithmus




top(Q)=orient. Gerade vom zweiten zum ersten Element aus $Q$.

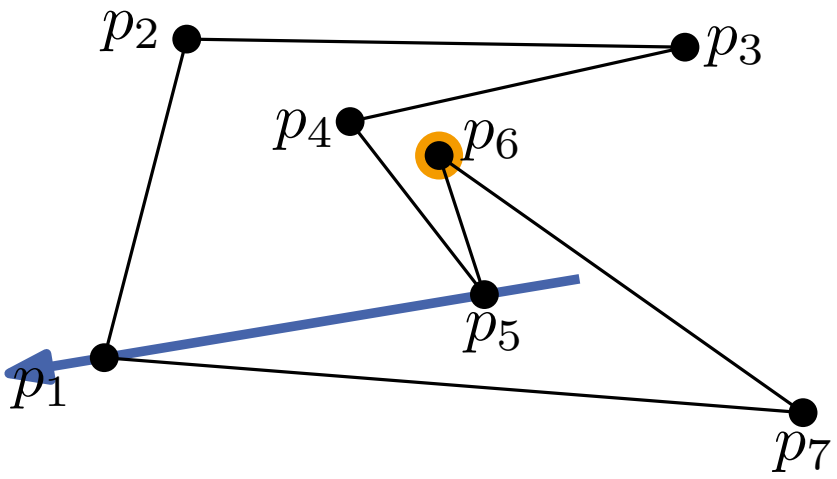
bottom(Q)=orient. Gerade vom letzten zum vorletzten Element von $Q$.

| |
|---|
| |
| $p_5$ |
| $p_3$ |
| $p_2$ |
| $p_1$ |
| $p_5$ |

ConvexHull(einfaches Polygon $P = (p_1, \ldots, p_n)$)

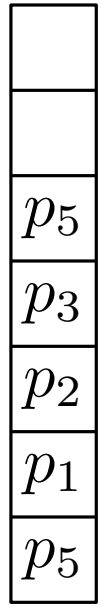
$Q \leftarrow$ double-ended queue $(p_2, p_1, p_2)$
**for** $i = 3, \ldots, n$ **do**
  **while** $p_i$ nicht rechts von top$(Q)$ **do** pop-top$(Q)$
  **while** $p_i$ nicht rechts von bottom$(Q)$ **do** pop-bottom$(Q)$
  **if** popped **then** push-top$(Q, p_i)$; push-bottom$(Q, p_i)$

# Aufgabe 4 – Melkmans Algorithmus




top(Q)=orient. Gerade vom zweiten zum ersten Element aus $Q$.

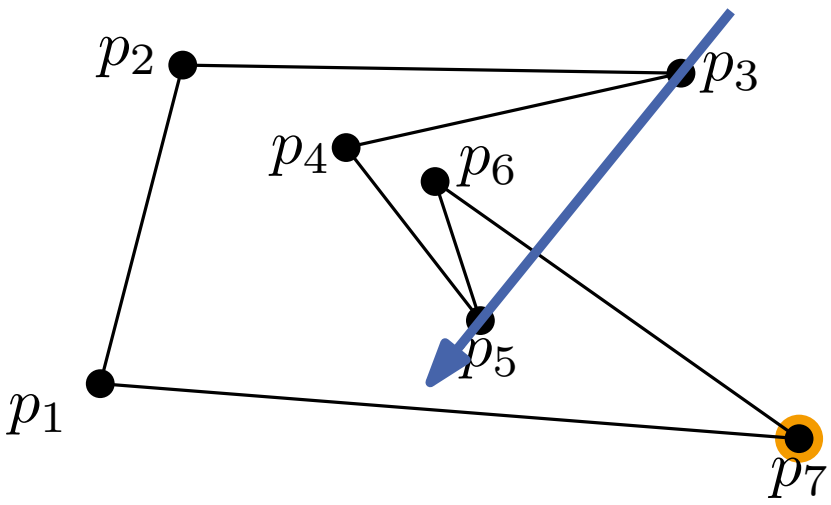
bottom(Q)=orient. Gerade vom letzten zum vorletzten Element von $Q$.




ConvexHull(einfaches Polygon $P = (p_1, \ldots, p_n)$)

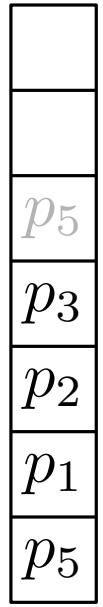
$Q \leftarrow$ double-ended queue $(p_2, p_1, p_2)$
**for** $i = 3, \ldots, n$ **do**
  **while** $p_i$ nicht rechts von top$(Q)$ **do** pop-top$(Q)$
  **while** $p_i$ nicht rechts von bottom$(Q)$ **do** pop-bottom$(Q)$
  **if** popped **then** push-top$(Q, p_i)$; push-bottom$(Q, p_i)$

# Aufgabe 4 – Melkmans Algorithmus




top(Q)=orient. Gerade vom zweiten zum ersten Element aus $Q$.

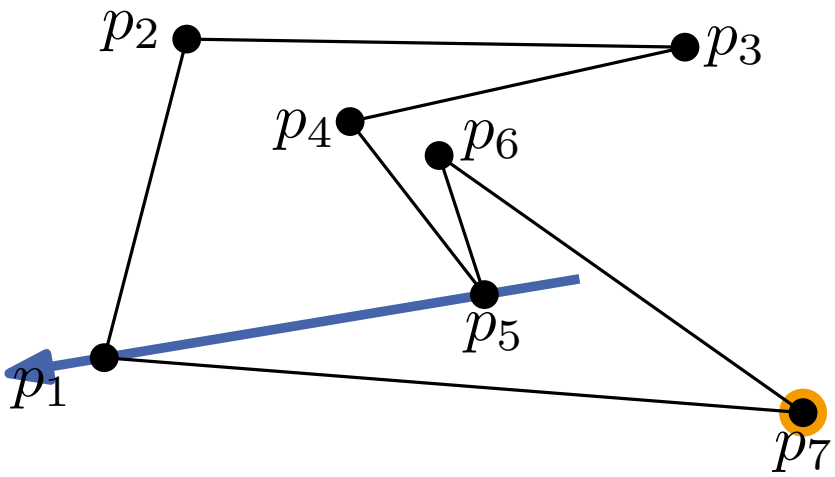
bottom(Q)=orient. Gerade vom letzten zum vorletzten Element von $Q$.




ConvexHull(einfaches Polygon $P = (p_1, \ldots, p_n)$)

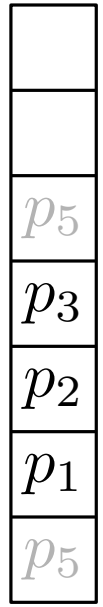
$Q \leftarrow$ double-ended queue $(p_2, p_1, p_2)$
**for** $i = 3, \ldots, n$ **do**
  **while** $p_i$ nicht rechts von top$(Q)$ **do** pop-top$(Q)$
  **while** $p_i$ nicht rechts von bottom$(Q)$ **do** pop-bottom$(Q)$
  **if** popped **then** push-top$(Q, p_i)$; push-bottom$(Q, p_i)$

# Aufgabe 4 – Melkmans Algorithmus




top(Q)=orient. Gerade vom zweiten zum ersten Element aus $Q$.

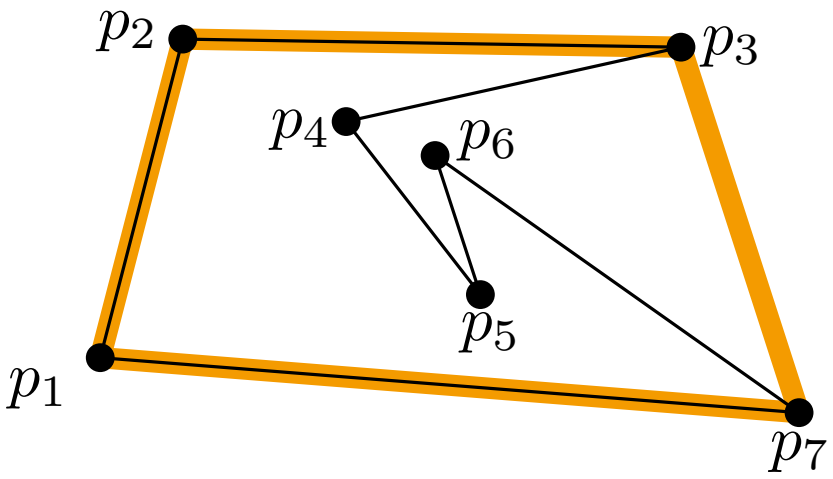
bottom(Q)=orient. Gerade vom letzten zum vorletzten Element von $Q$.




ConvexHull(einfaches Polygon $P = (p_1, \ldots, p_n)$)

$Q \leftarrow$ double-ended queue $(p_2, p_1, p_2)$
**for** $i = 3, \ldots, n$ **do**
  **while** $p_i$ nicht rechts von top$(Q)$ **do** pop-top$(Q)$
  **while** $p_i$ nicht rechts von bottom$(Q)$ **do** pop-bottom$(Q)$
  **if** popped **then** push-top$(Q, p_i)$; push-bottom$(Q, p_i)$

# Aufgabe 4 – Melkmans Algorithmus




top(Q)=orient. Gerade vom zweiten zum ersten Element aus $Q$.

bottom(Q)=orient. Gerade vom letzten zum vorletzten Element von $Q$.

| |
|---|
| |
| $p_7$ |
| $p_3$ |
| $p_2$ |
| $p_1$ |
| $p_7$ |

ConvexHull(einfaches Polygon $P = (p_1, \ldots, p_n)$)

$Q \leftarrow$ double-ended queue $(p_2, p_1, p_2)$
**for** $i = 3, \ldots, n$ **do**
  **while** $p_i$ nicht rechts von top$(Q)$ **do** pop-top$(Q)$
  **while** $p_i$ nicht rechts von bottom$(Q)$ **do** pop-bottom$(Q)$
  **if** popped **then** push-top$(Q, p_i)$; push-bottom$(Q, p_i)$

# Ablauf

Karlsruhe Institute of Technology

## Übungsblatt 1

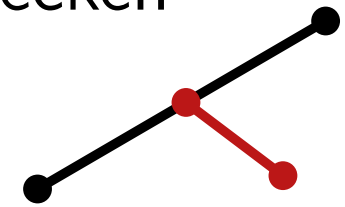
## Übungsblatt 2

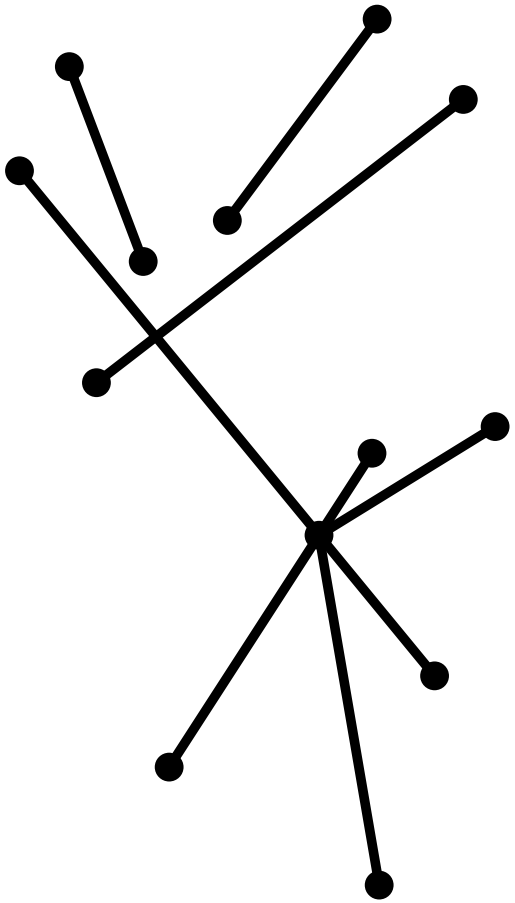
# Problemstellung

**Geg:** Menge $S = \{s_1, \ldots, s_n\}$ von Strecken in der Ebene

**Ges:**
- alle Schnittpunkte von zwei oder mehr Strecken
- für jeden Schnittpunkt die beteiligten Strecken

**Def:** Strecken sind **abgeschlossene** Mengen

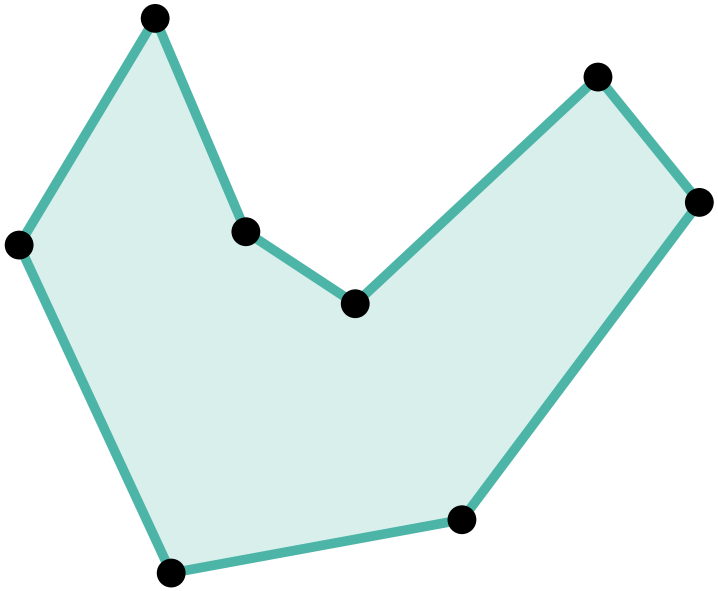


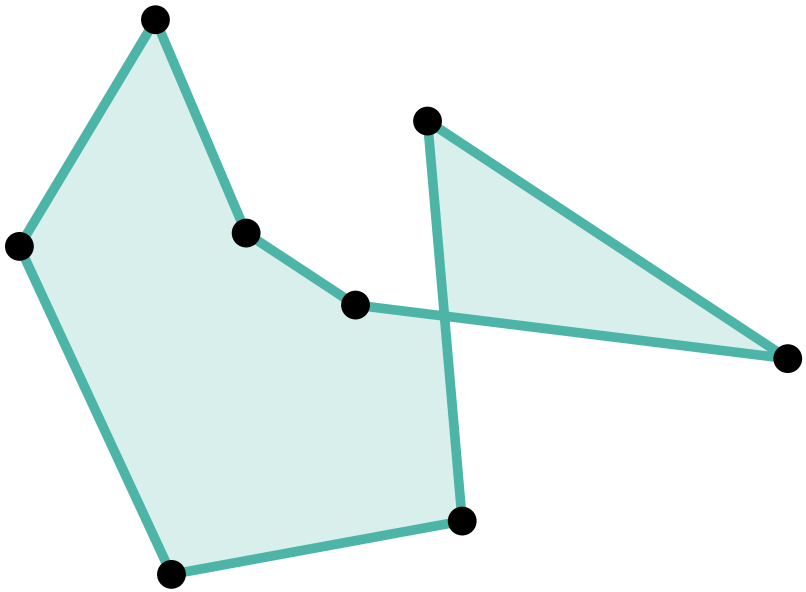


# Übungsblatt 2 - Aufgabe 1

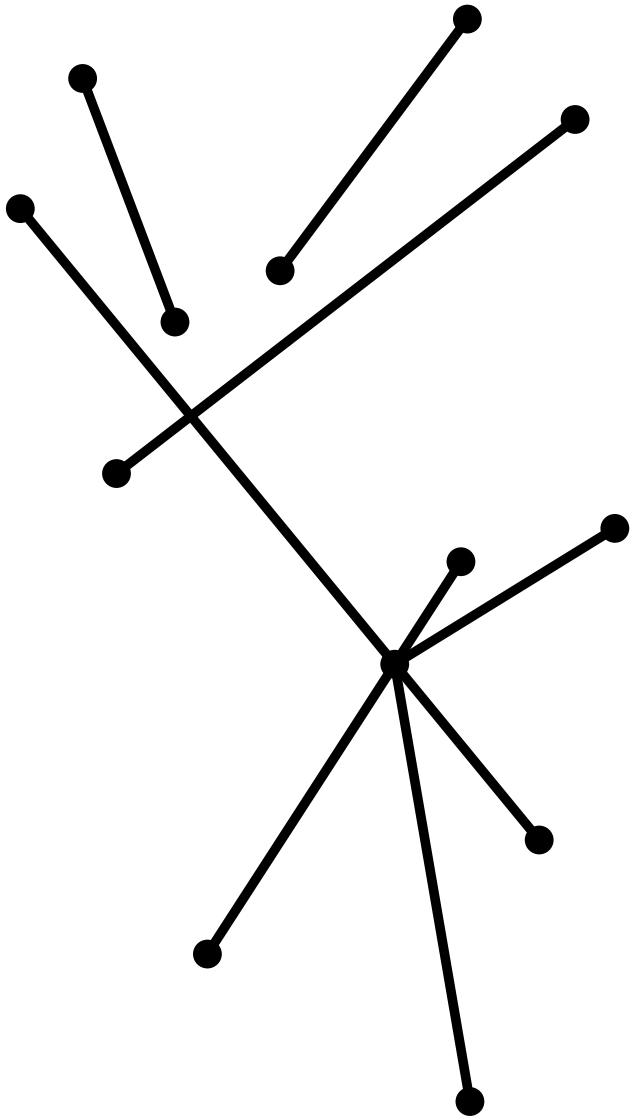
**Gesucht:**

Algorithmus der in $\mathcal{O}(n \log n)$ bestimmt ob ein Polygon einfach (d.h. schnittfrei) ist.

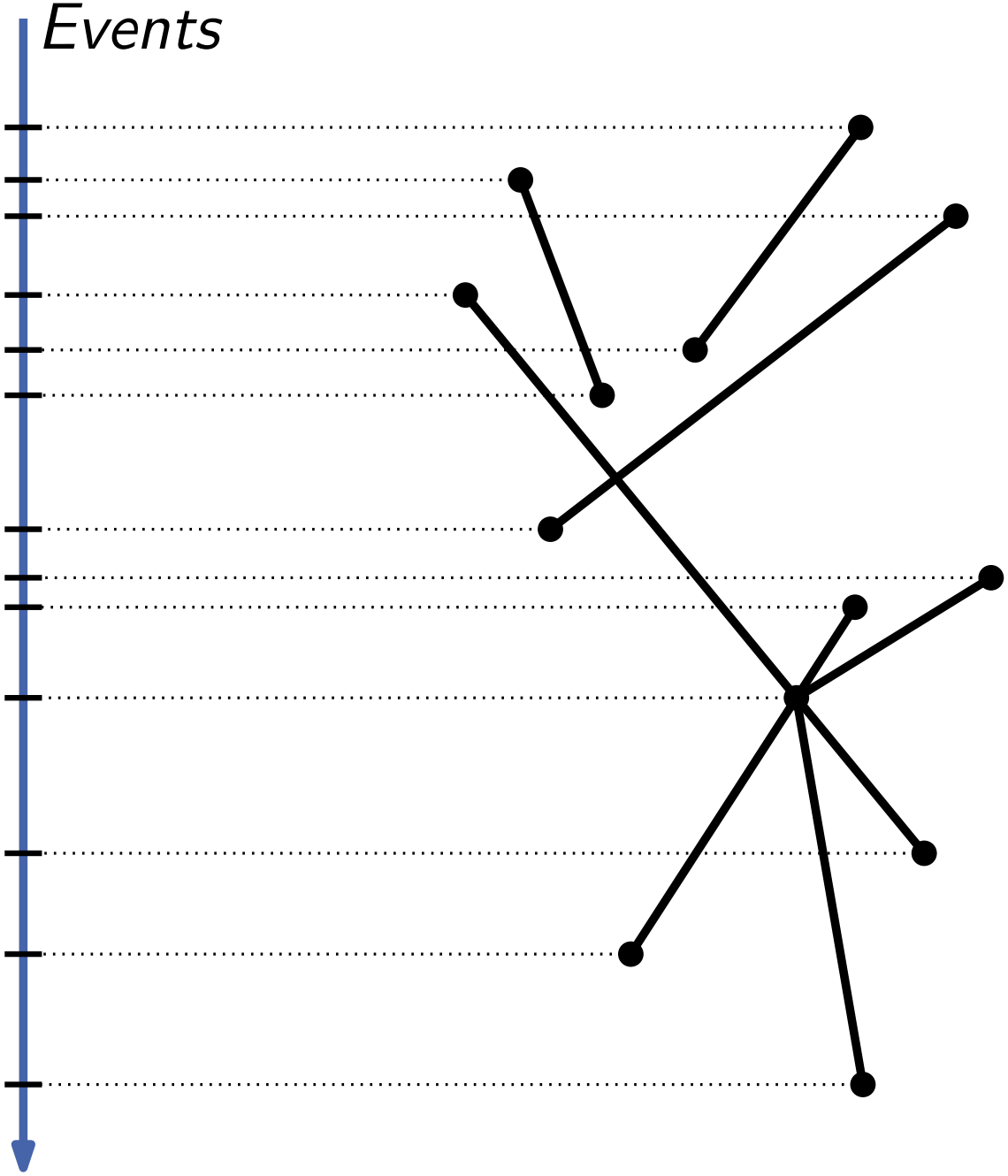




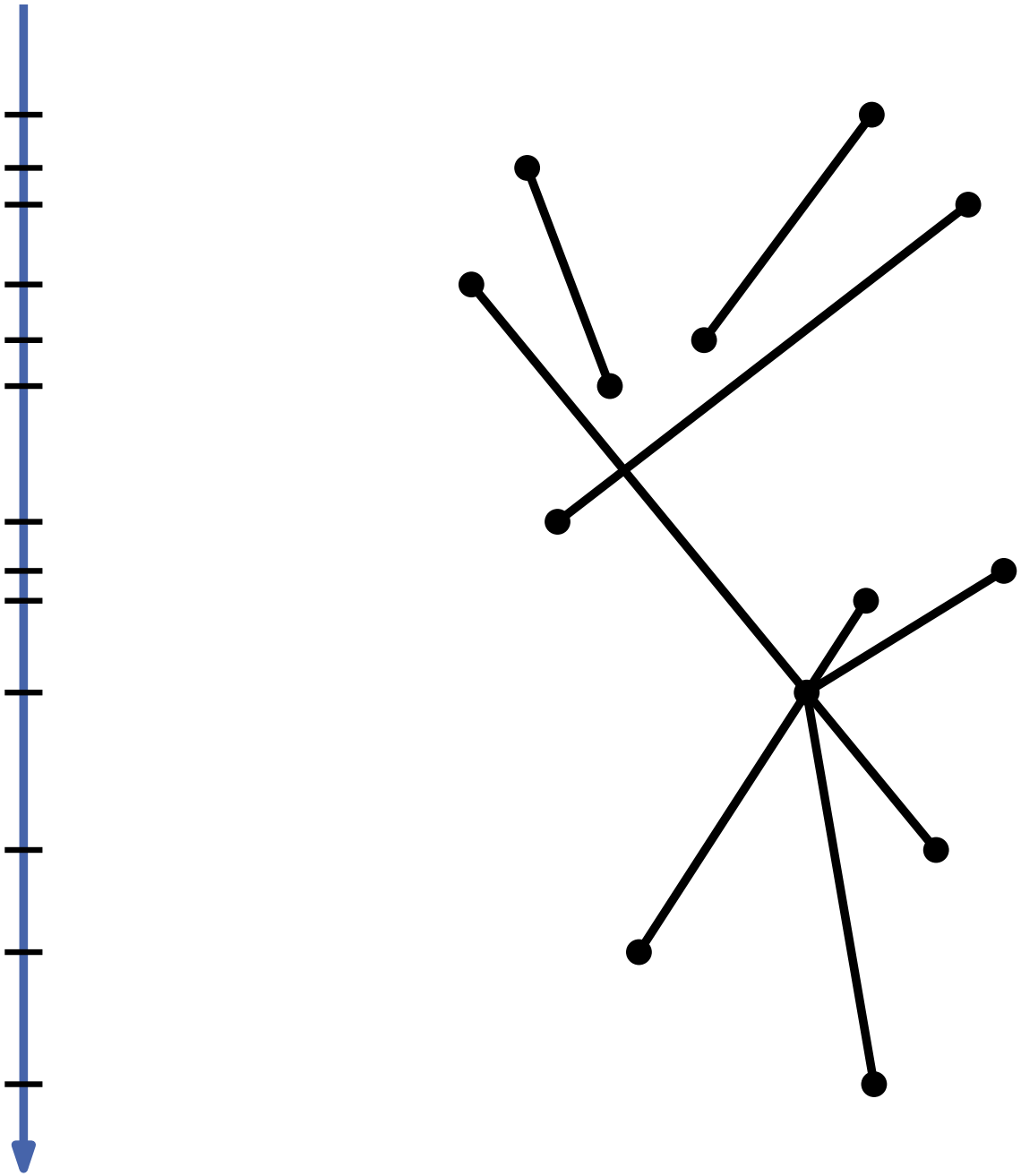
# Das Sweep-Line Verfahren: Beispiel



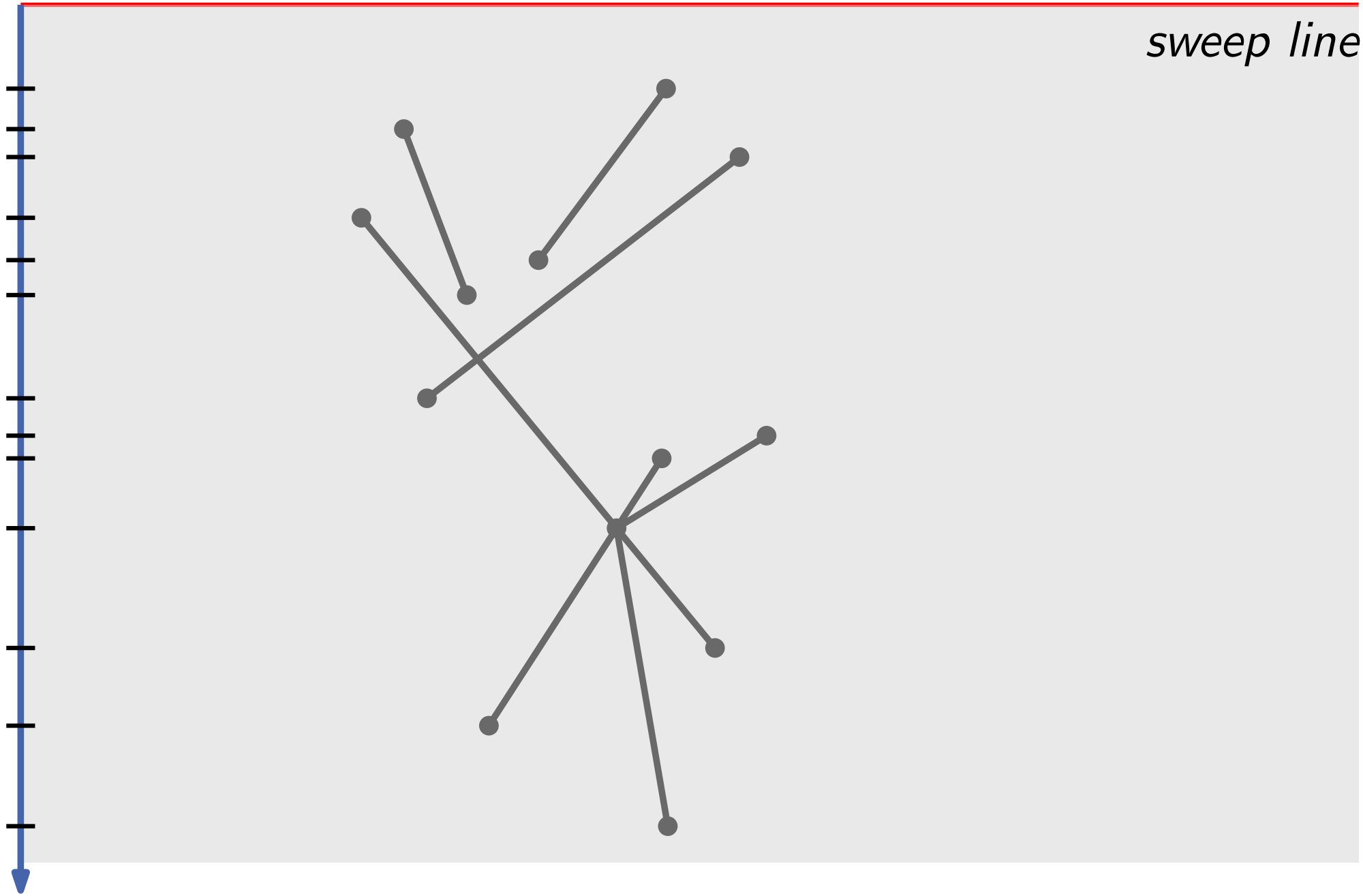
# Das Sweep-Line Verfahren: Beispiel



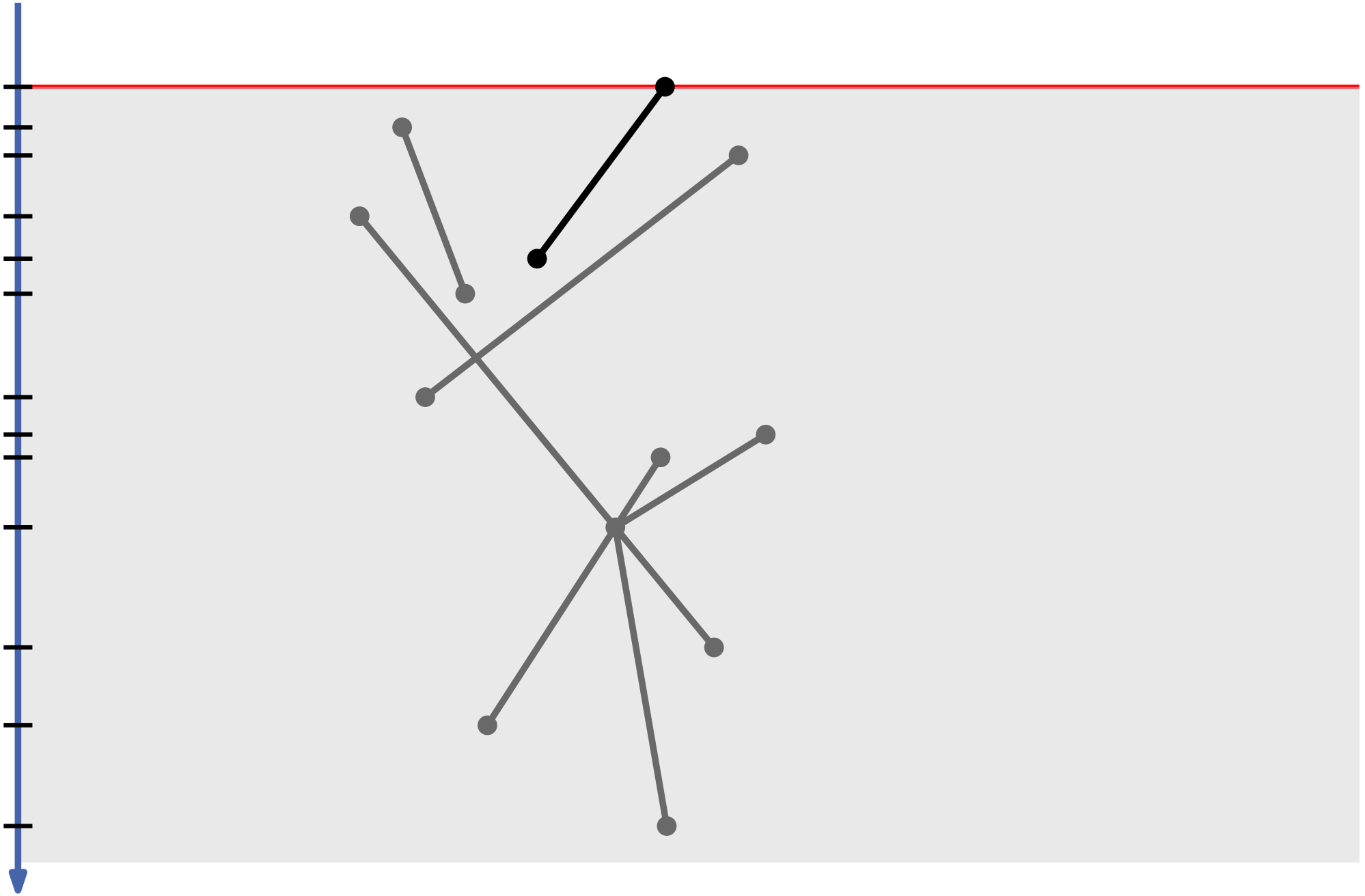
# Das Sweep-Line Verfahren: Beispiel



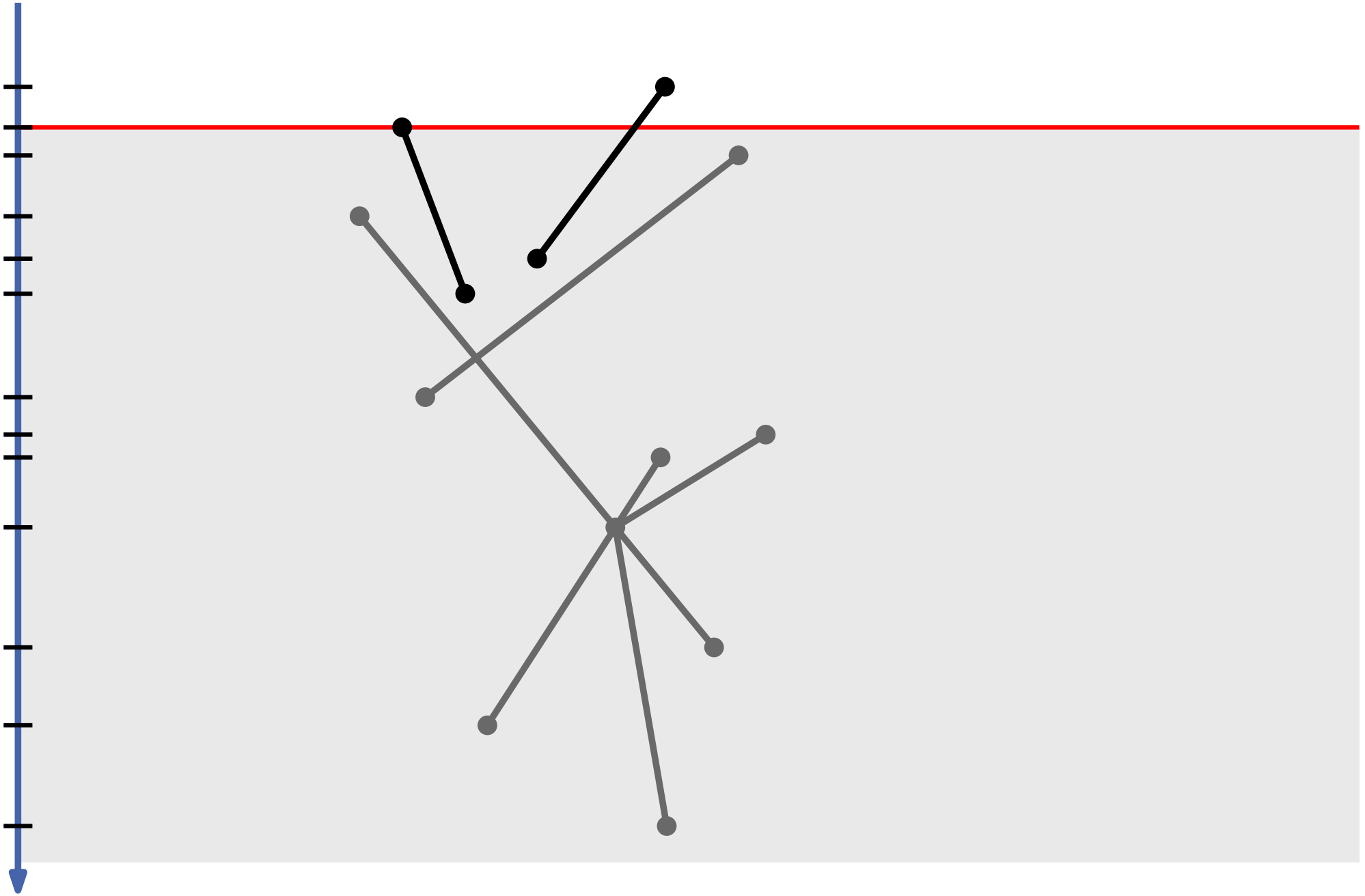
# Das Sweep-Line Verfahren: Beispiel



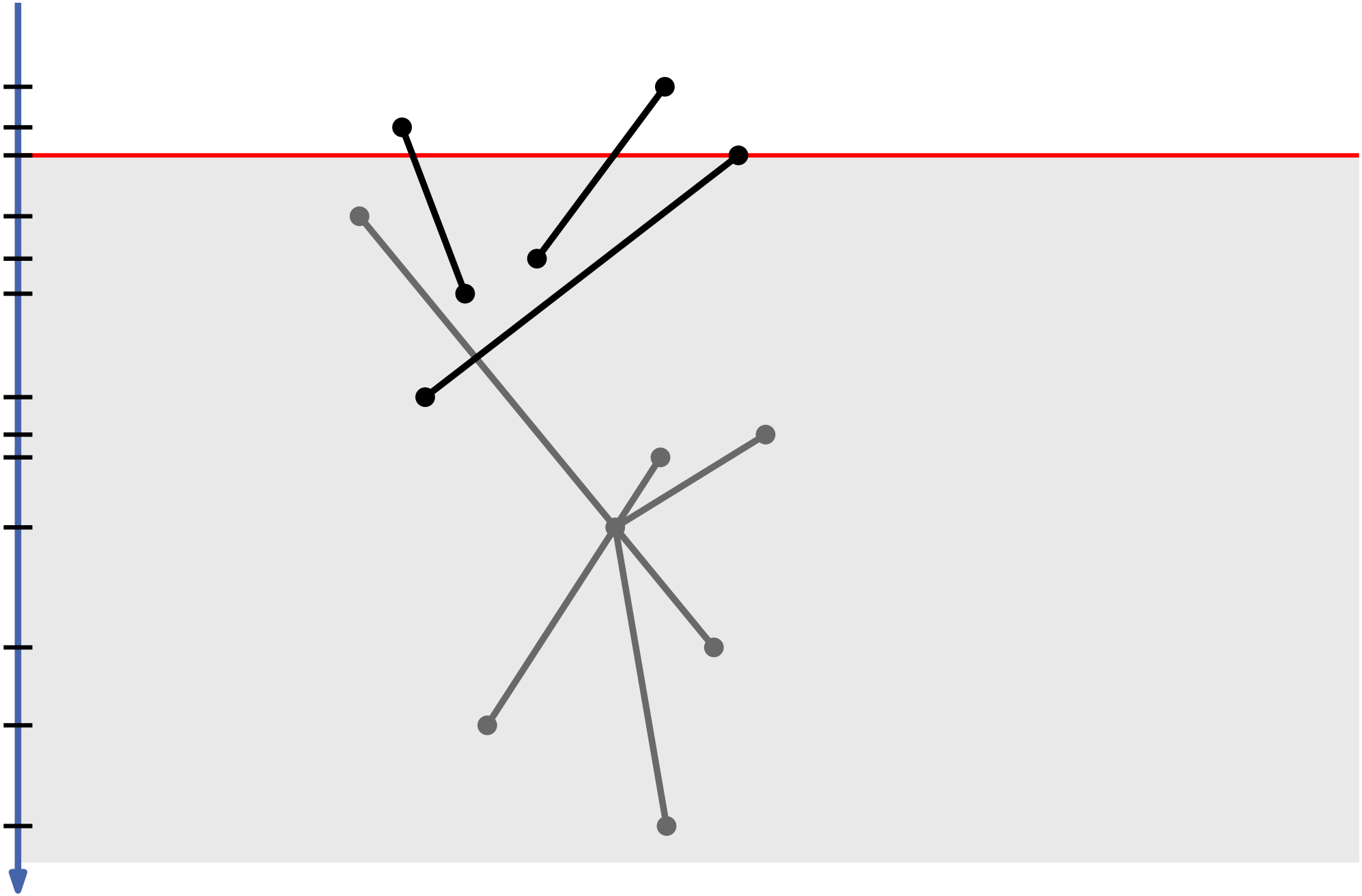
# Das Sweep-Line Verfahren: Beispiel



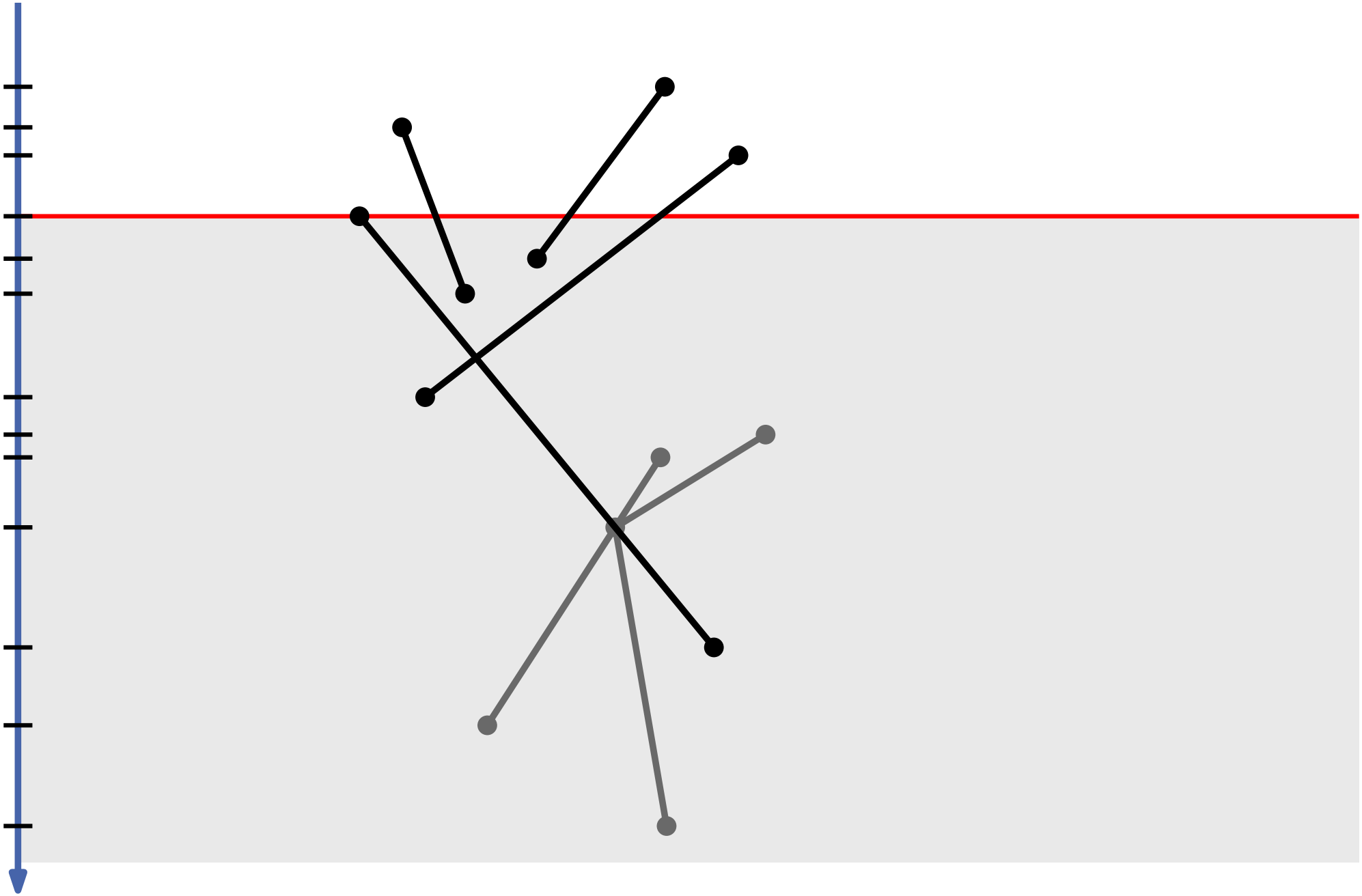
# Das Sweep-Line Verfahren: Beispiel



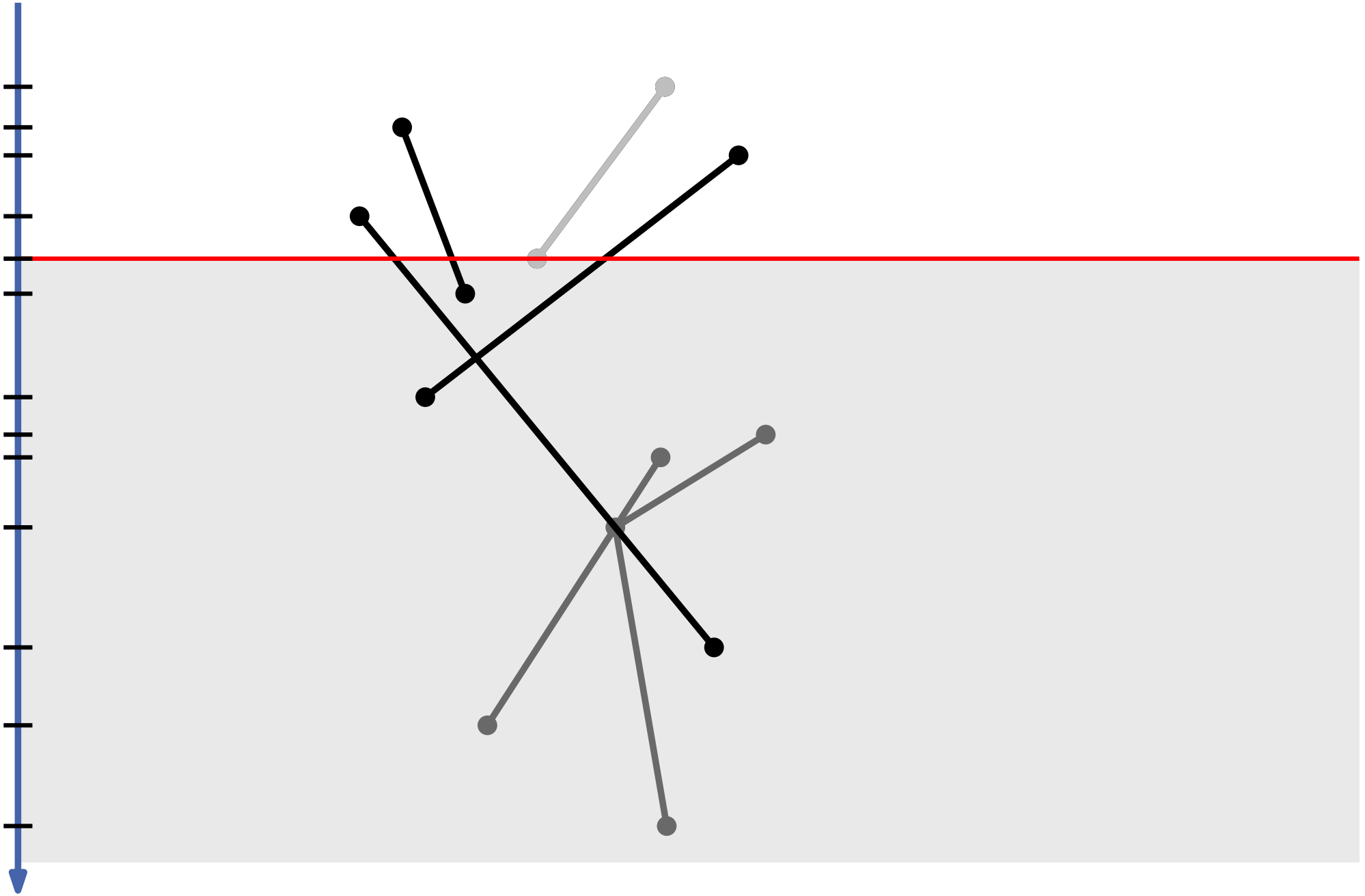
# Das Sweep-Line Verfahren: Beispiel



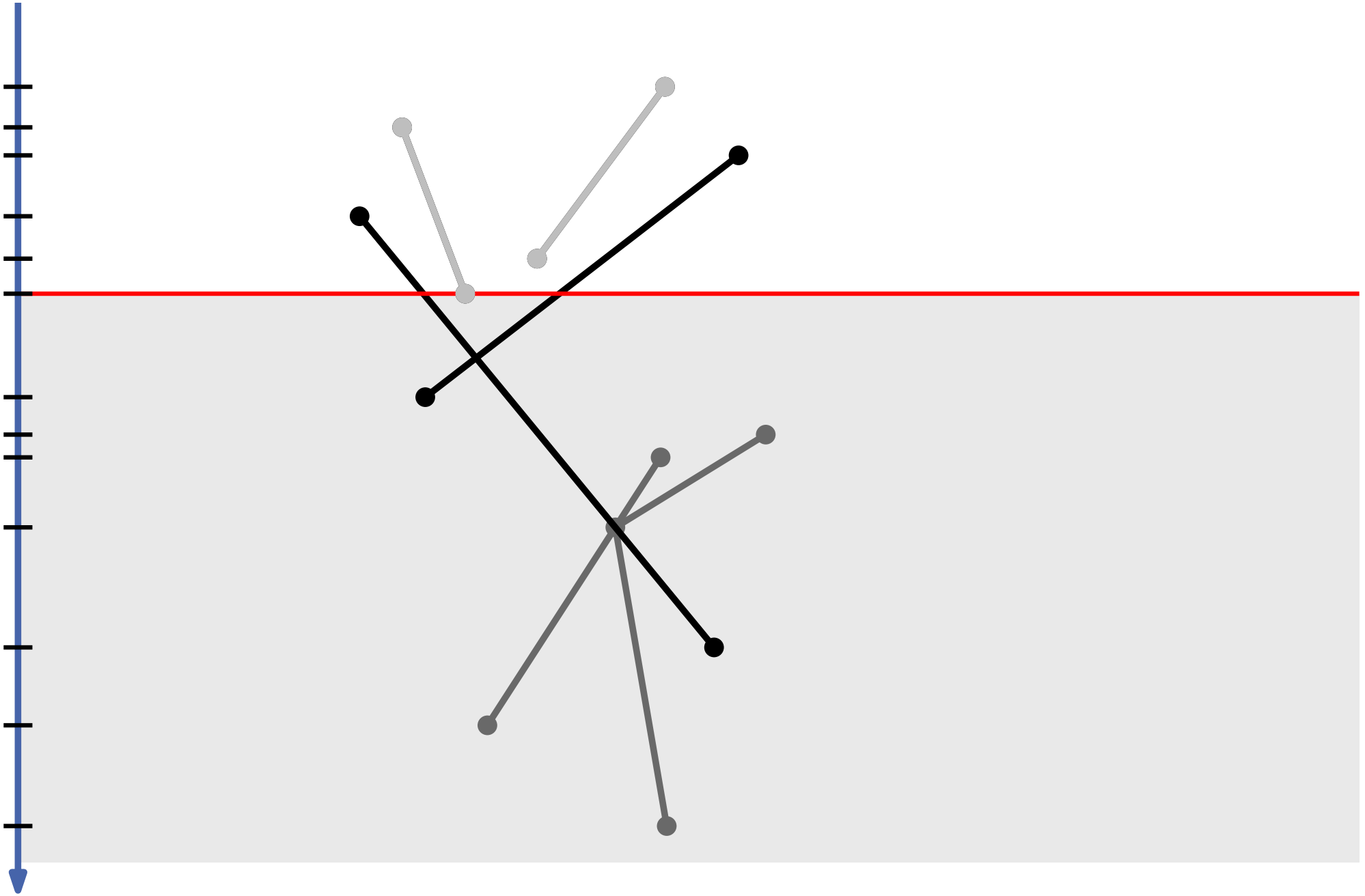
# Das Sweep-Line Verfahren: Beispiel



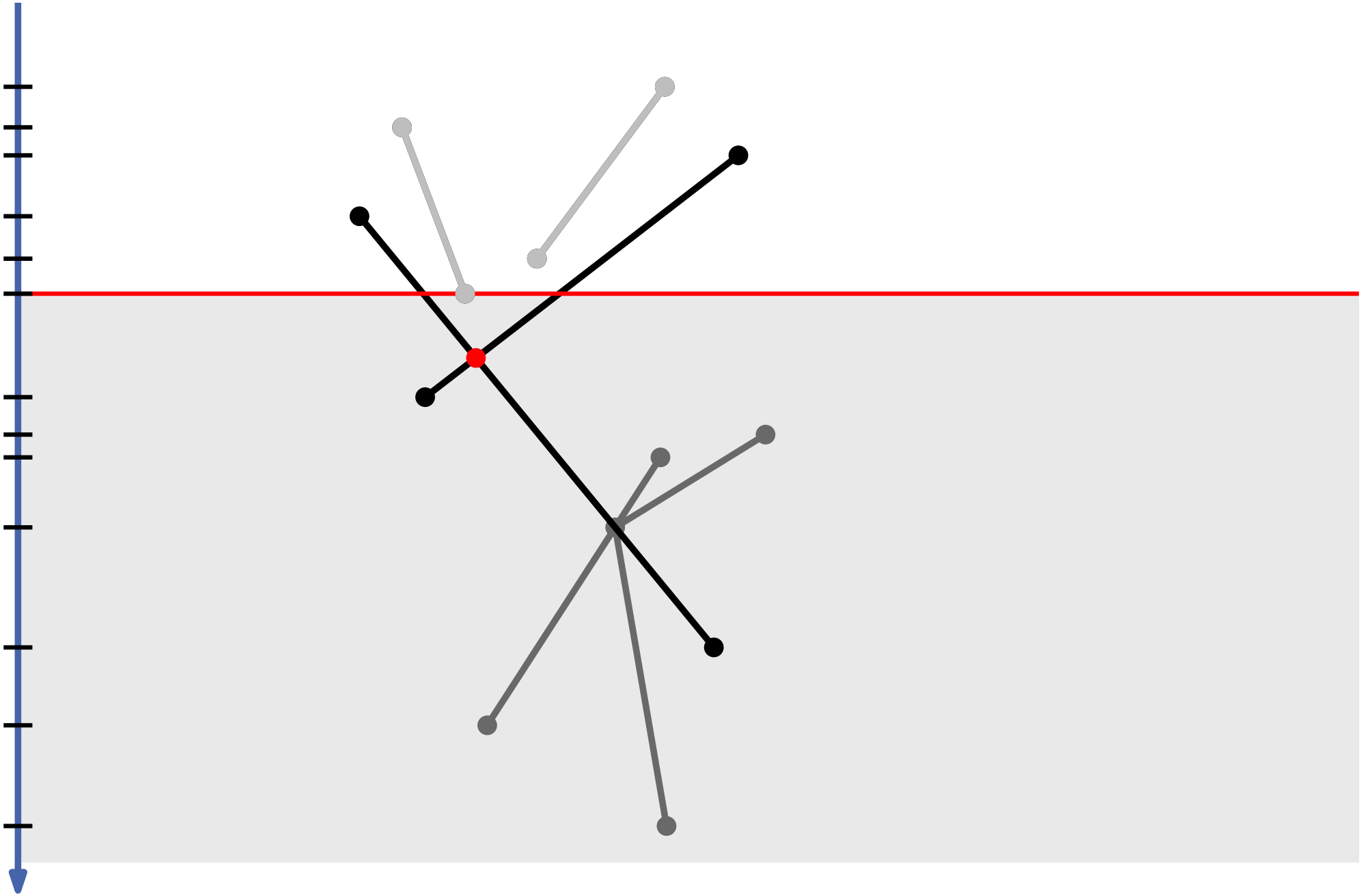
# Das Sweep-Line Verfahren: Beispiel



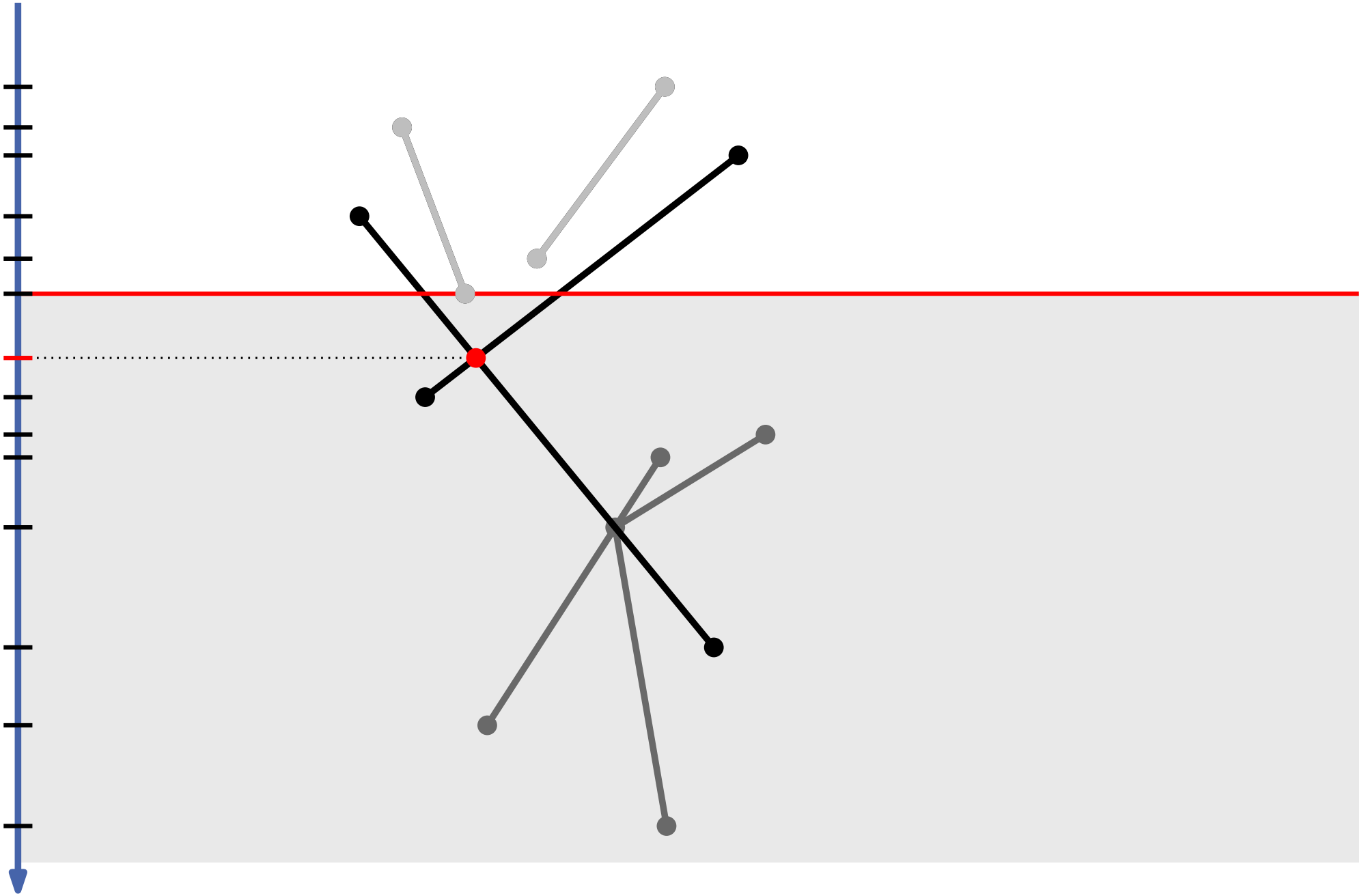
# Das Sweep-Line Verfahren: Beispiel



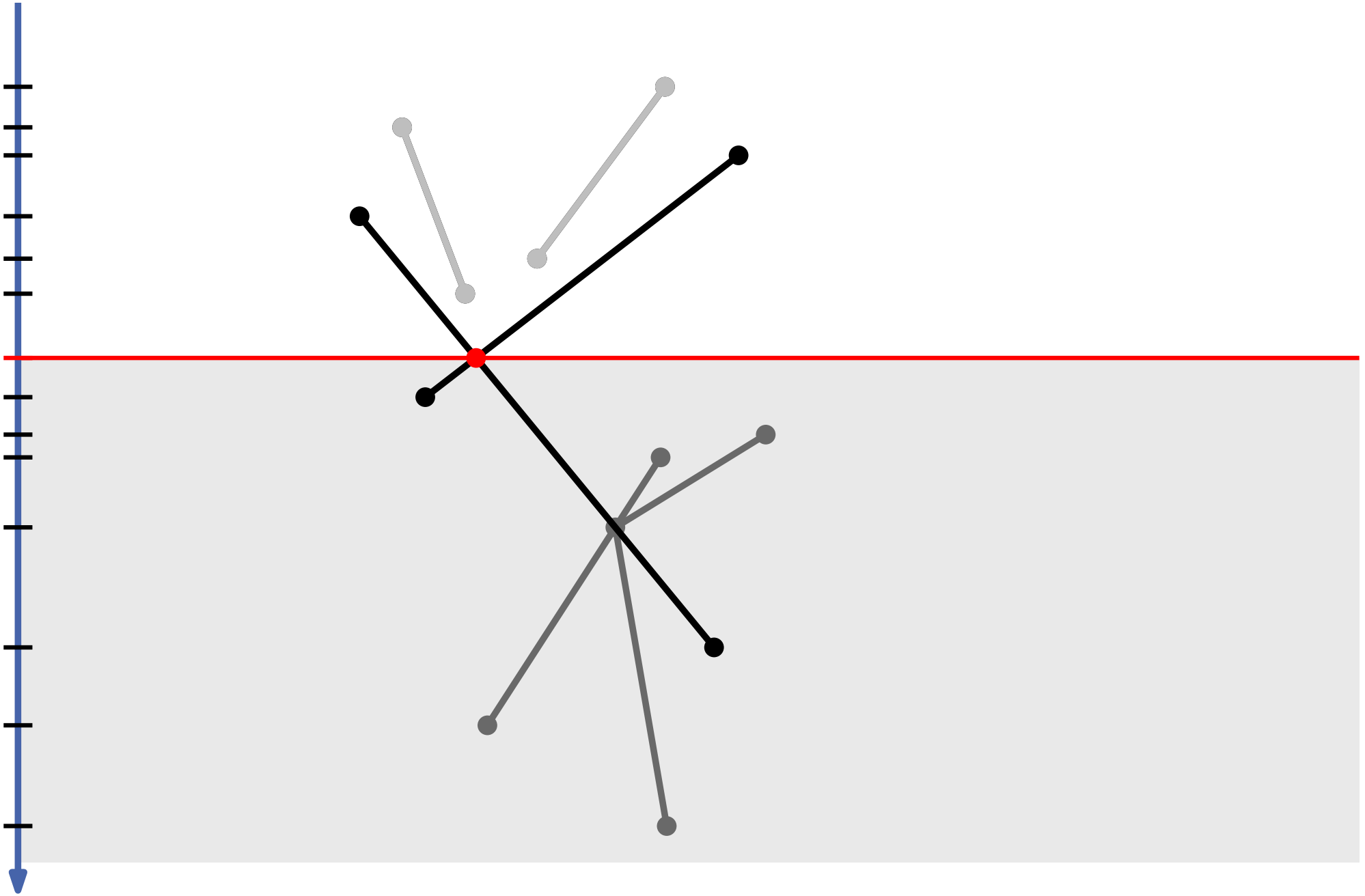
# Das Sweep-Line Verfahren: Beispiel



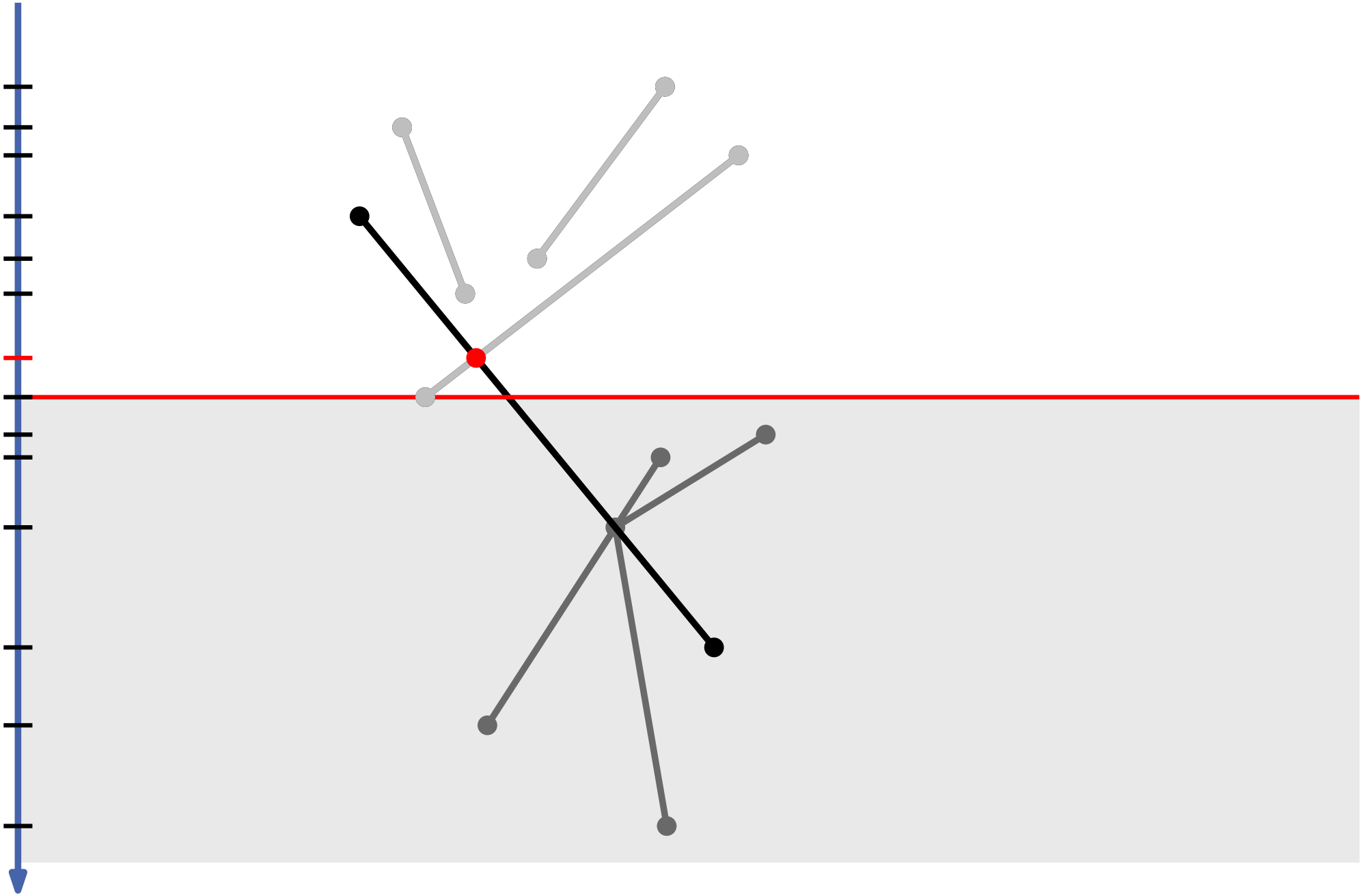
# Das Sweep-Line Verfahren: Beispiel



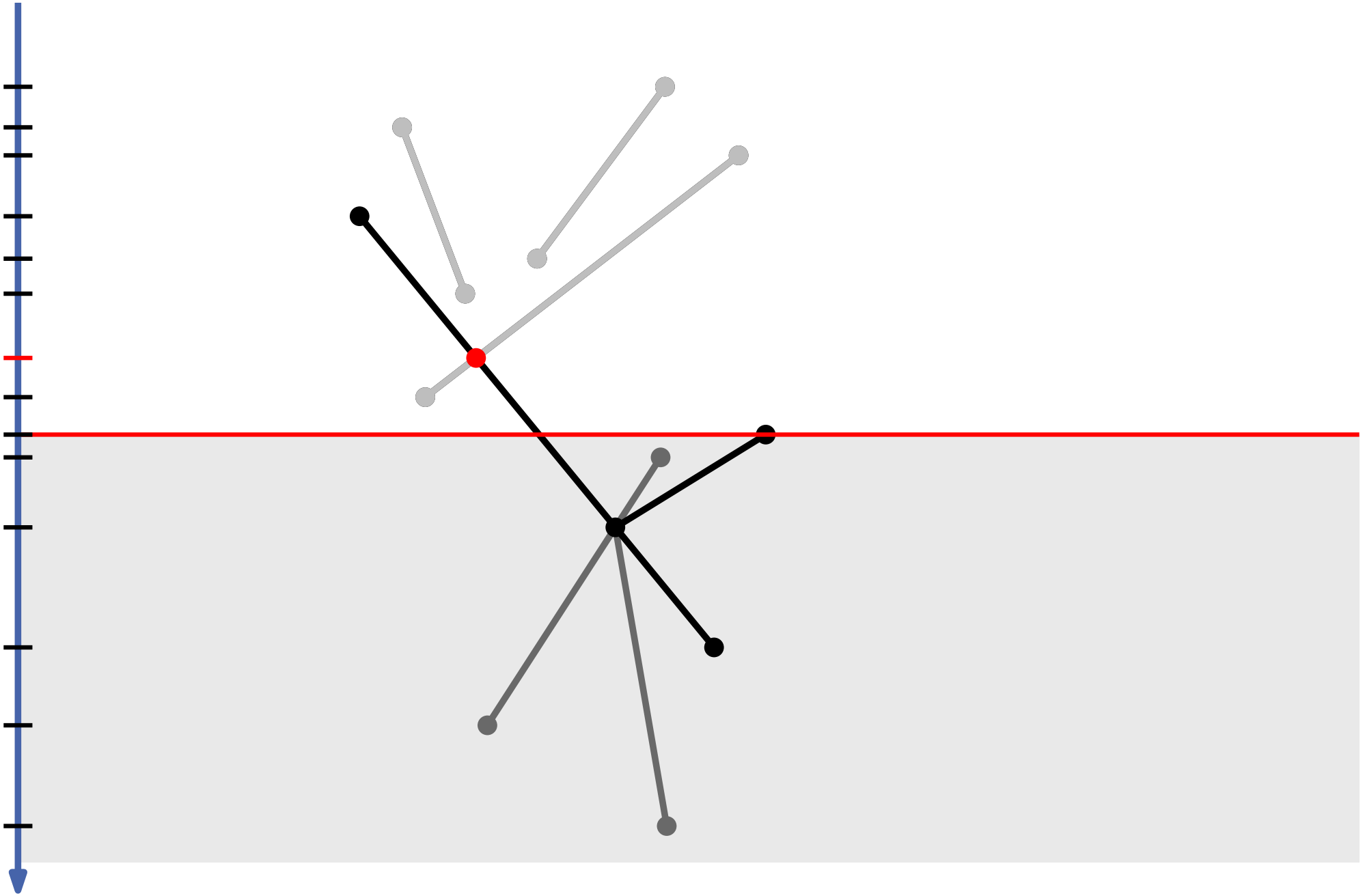
# Das Sweep-Line Verfahren: Beispiel



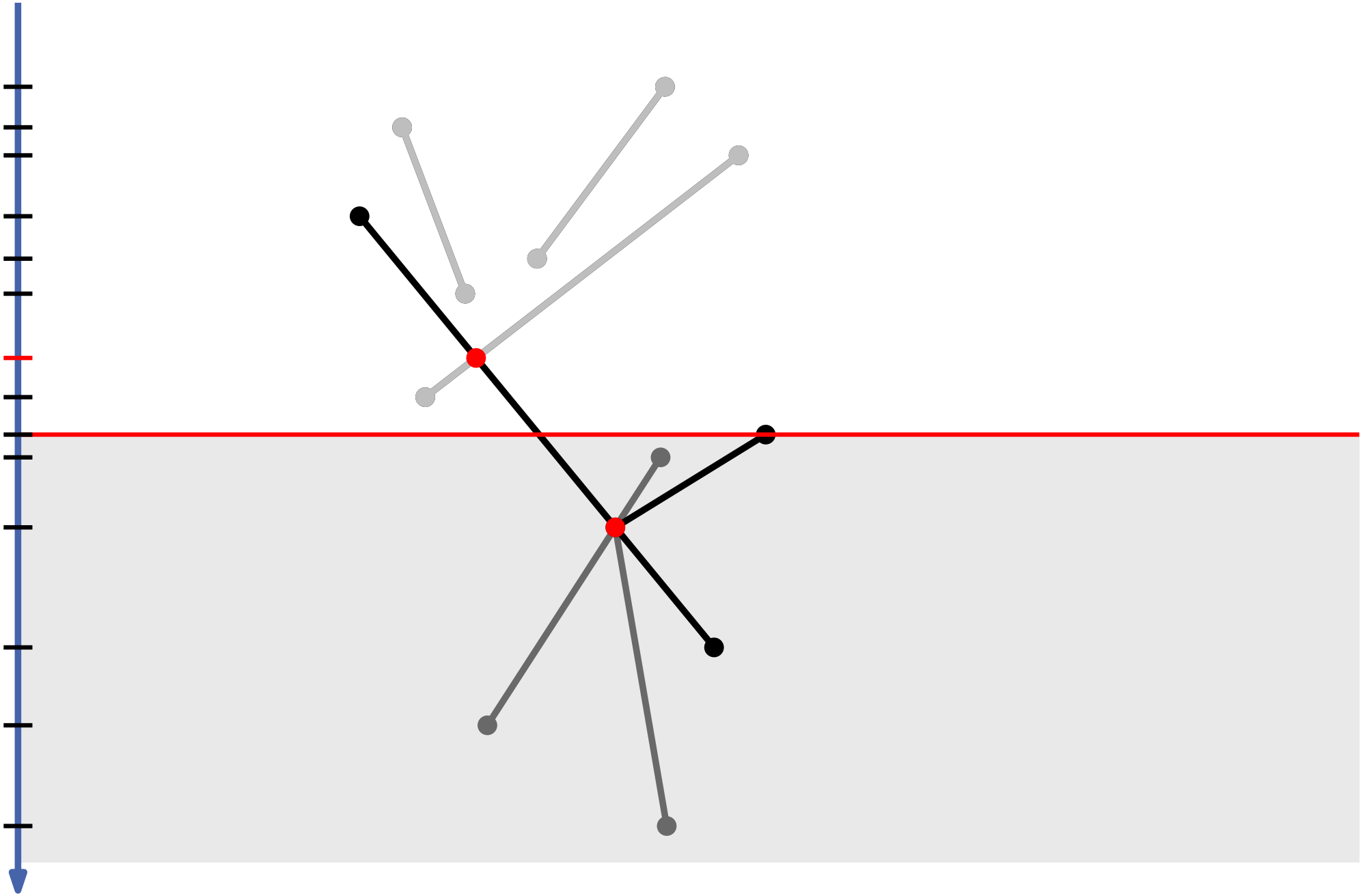
# Das Sweep-Line Verfahren: Beispiel



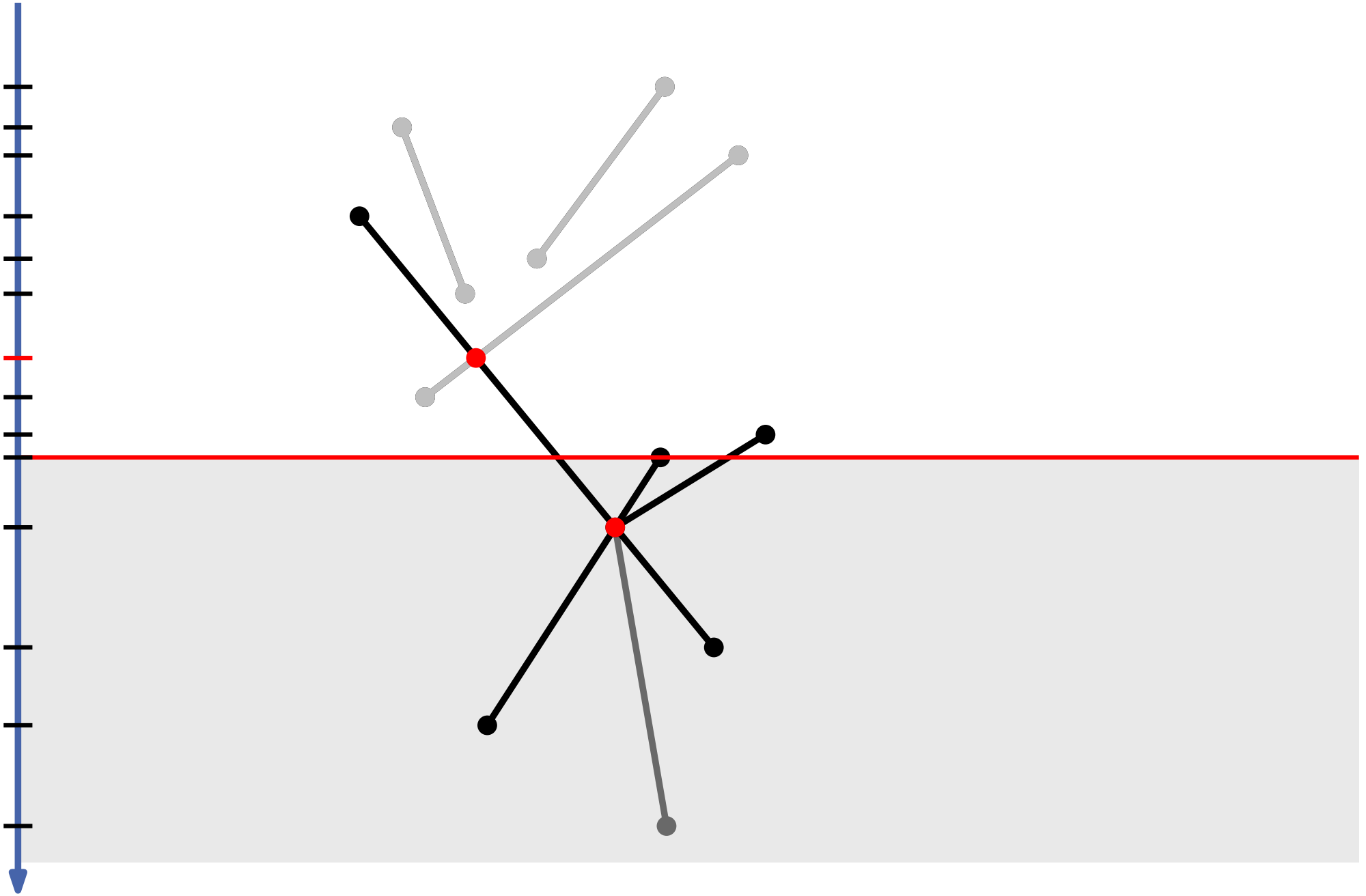
# Das Sweep-Line Verfahren: Beispiel



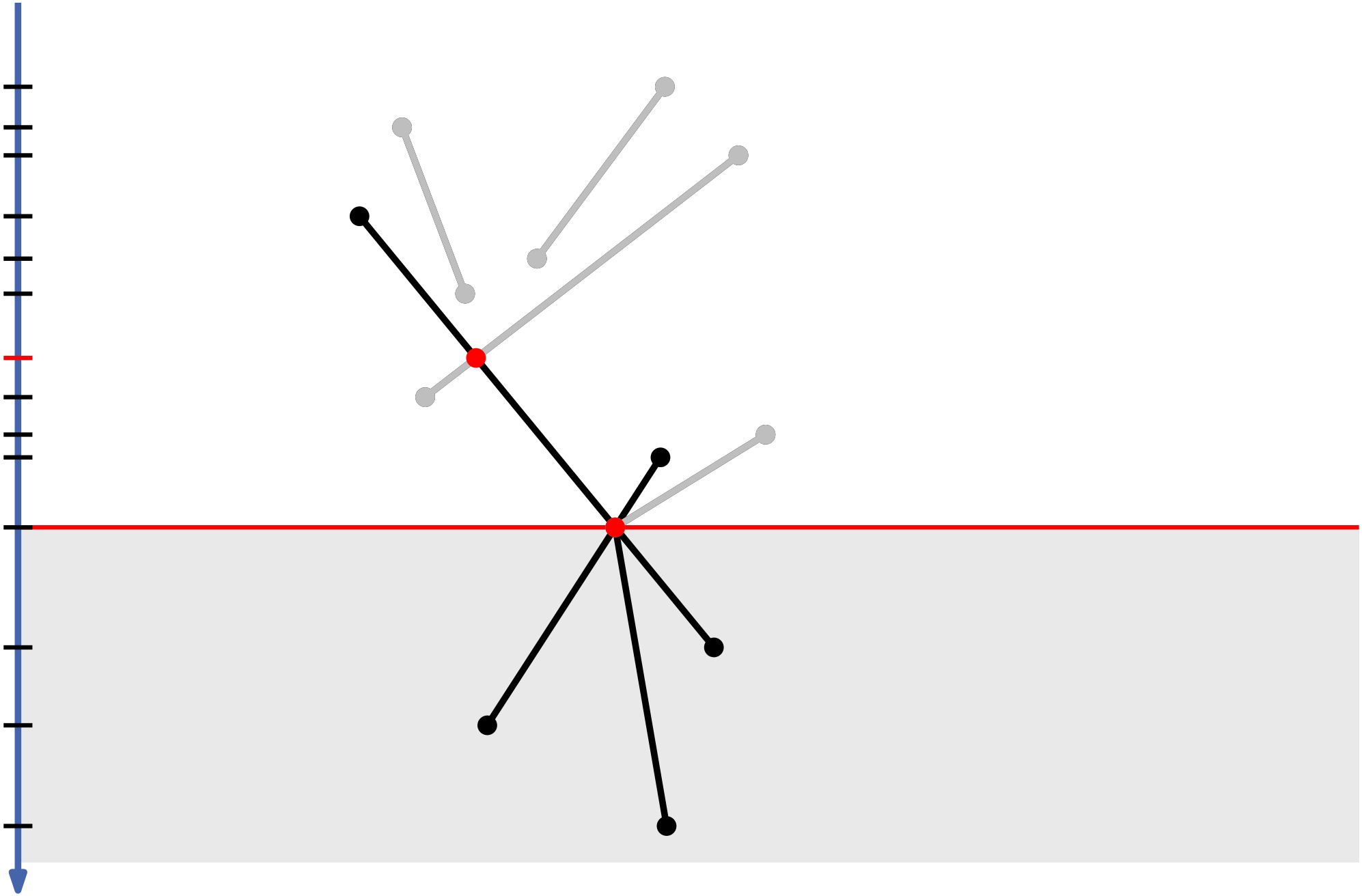
# Das Sweep-Line Verfahren: Beispiel



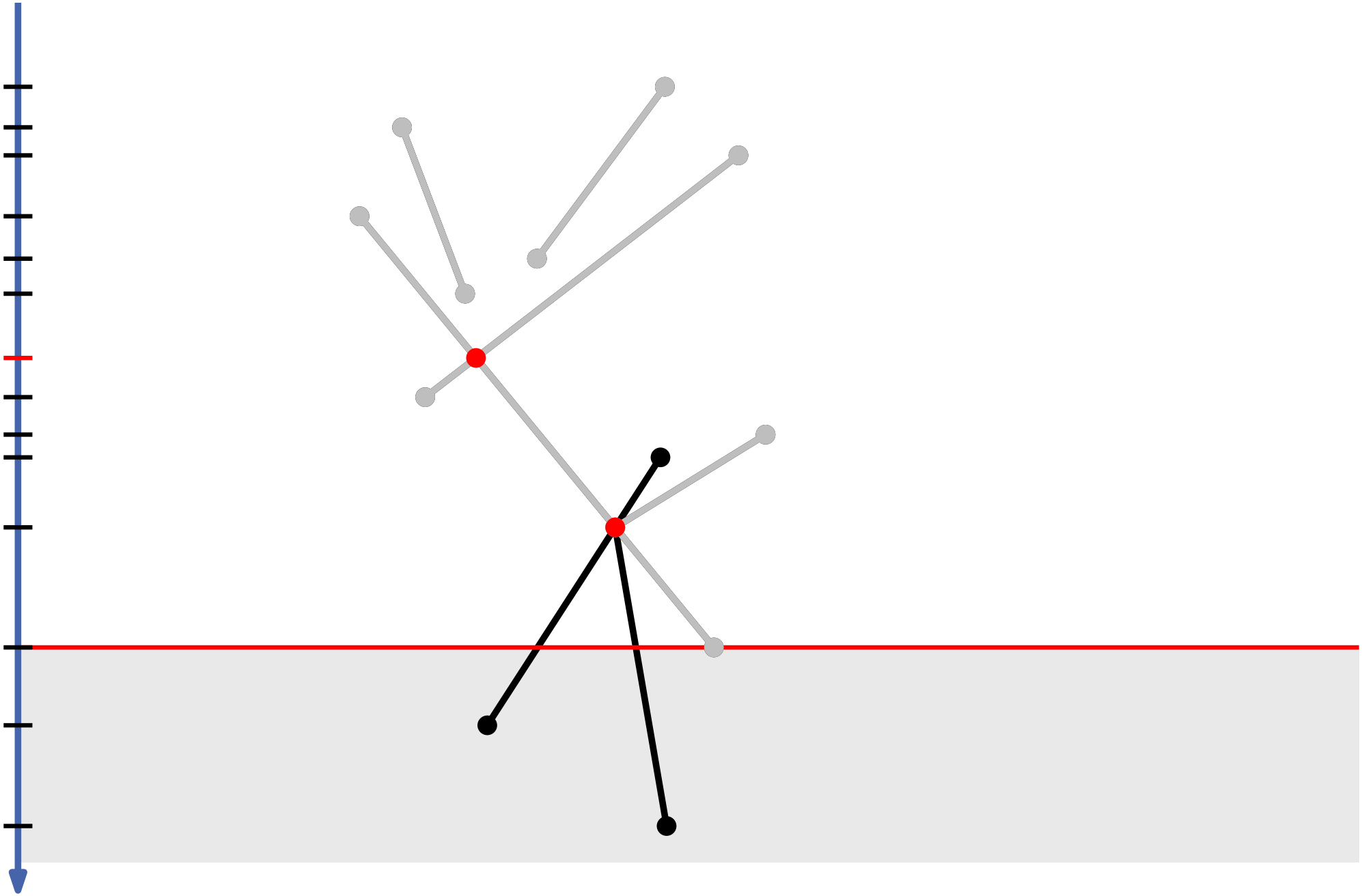
# Das Sweep-Line Verfahren: Beispiel



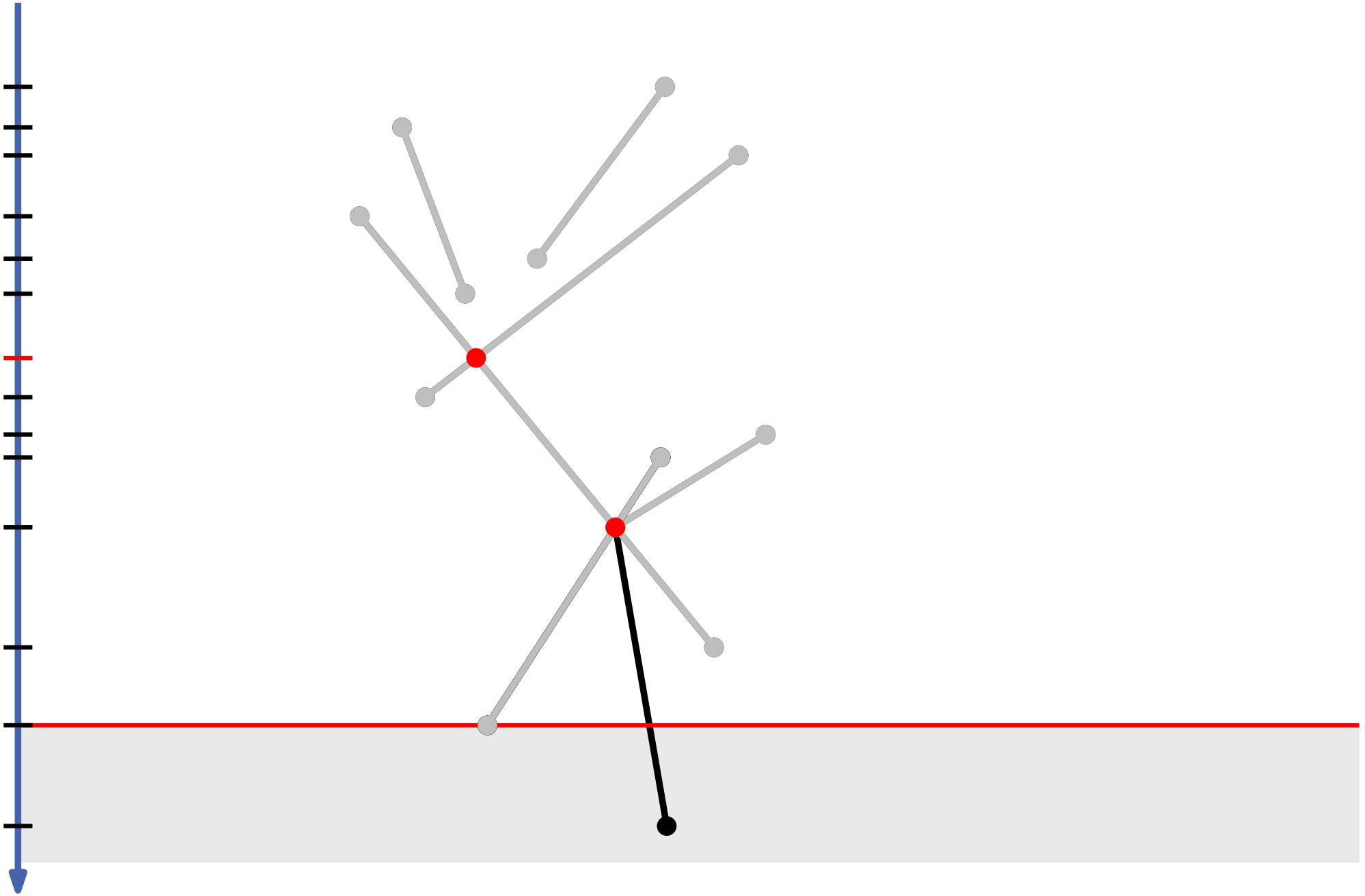
# Das Sweep-Line Verfahren: Beispiel



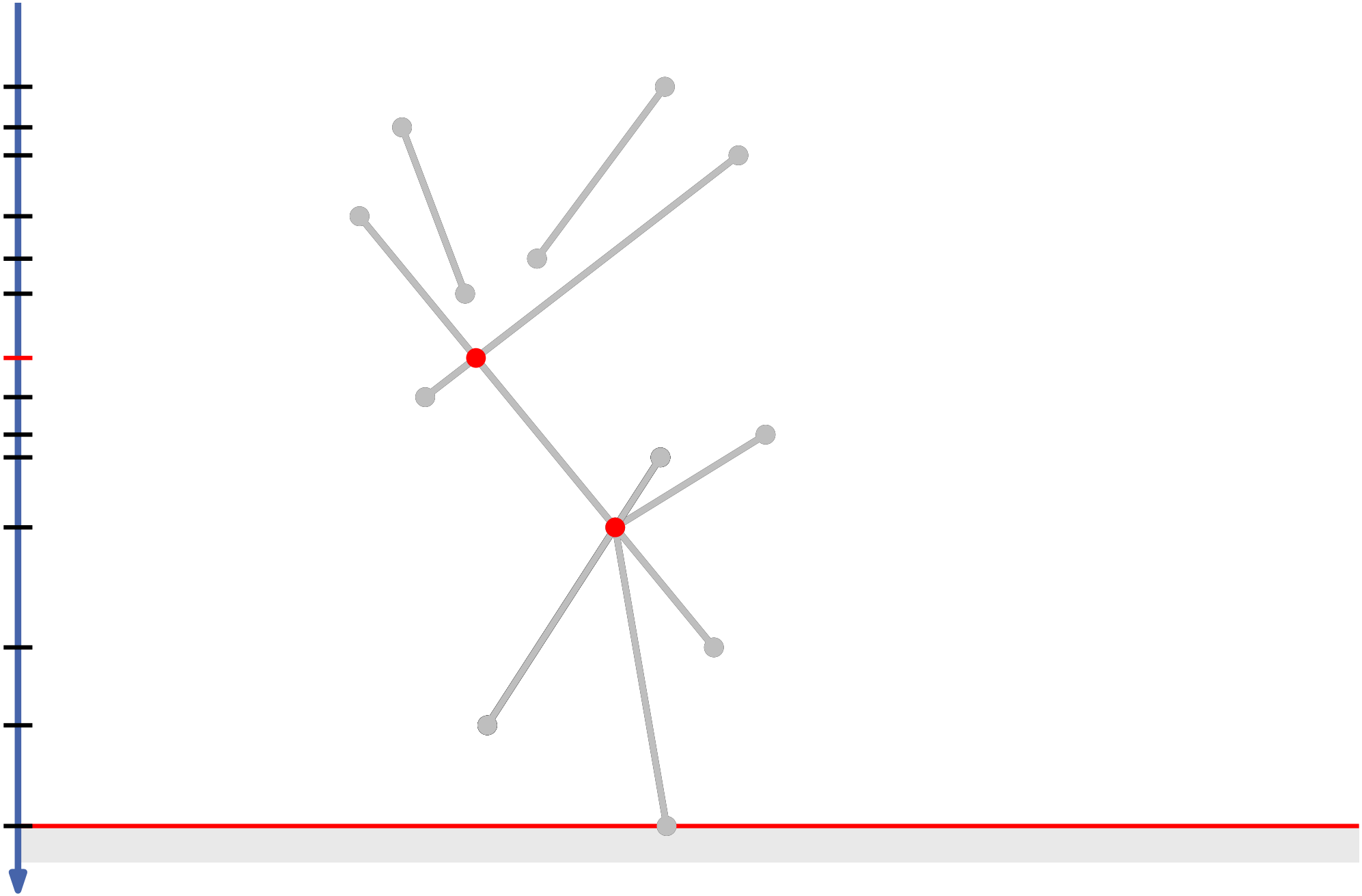
# Das Sweep-Line Verfahren: Beispiel



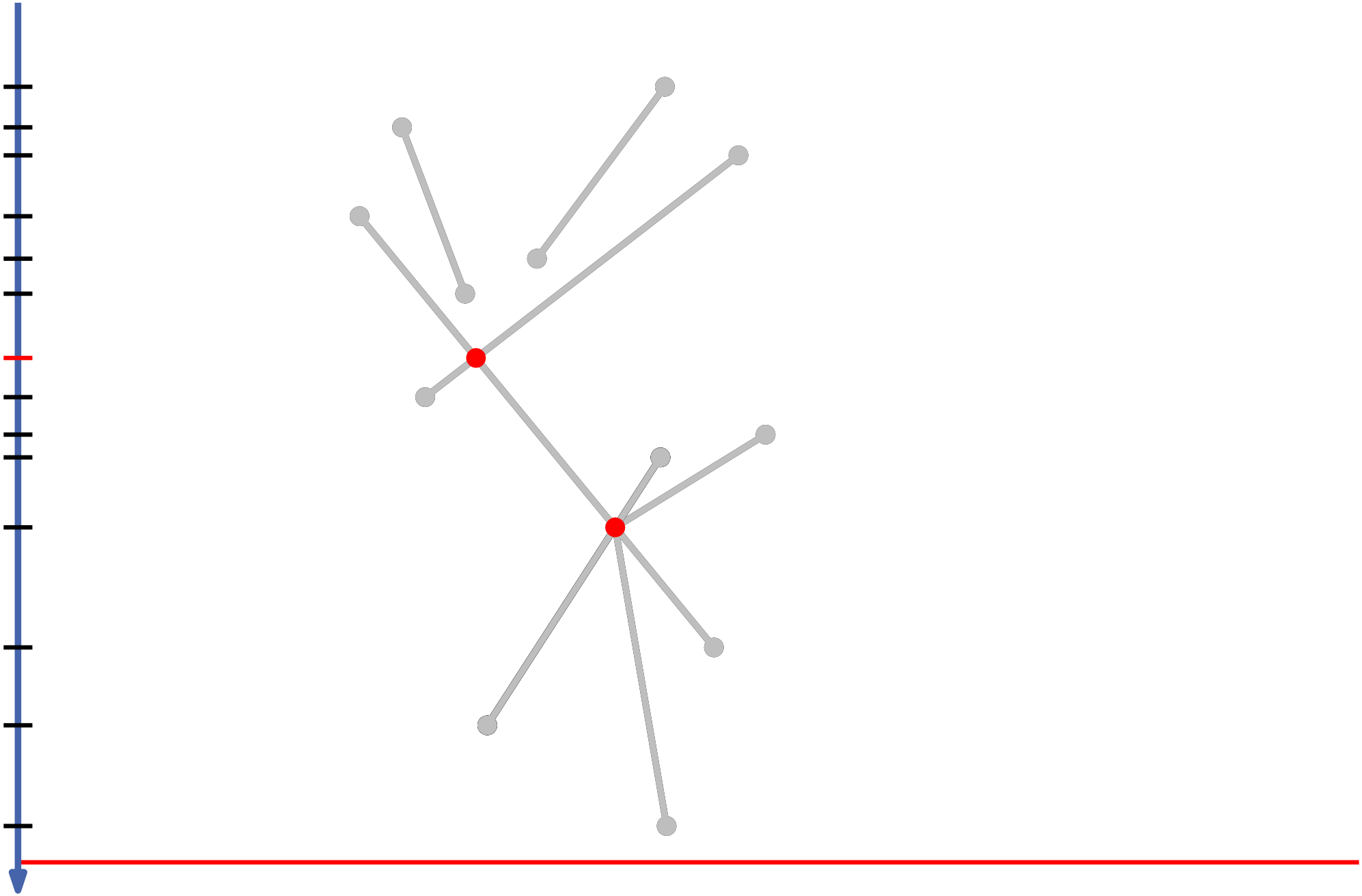
# Das Sweep-Line Verfahren: Beispiel



# Das Sweep-Line Verfahren: Beispiel



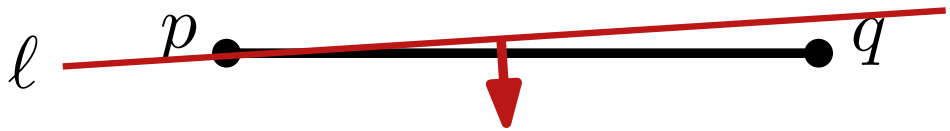
# Das Sweep-Line Verfahren: Beispiel



# Datenstrukturen

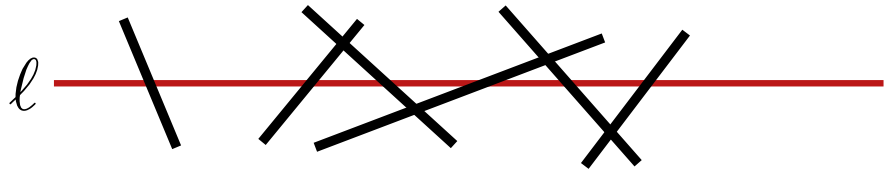
## 1.) Event Queue $\mathcal{Q}$

- definiere $p \prec q \quad \Leftrightarrow_{\text{def.}} \quad y_p > y_q \vee (y_p = y_q \wedge x_p < x_q)$




- speichere Events sortiert nach $\prec$ in **balanciertem binärem Suchbaum**
  $\rightarrow$ AVL-Baum, Rot-Schwarz-Baum, ...
- Operationen `insert`, `delete` und `nextEvent` in $O(\log |\mathcal{Q}|)$ Zeit

## 2.) Sweep-Line Status $\mathcal{T}$




- speichere von $\ell$ geschnittene Strecken geordnet von links nach rechts
- benötigte Operationen `insert`, `delete`, `findNeighbor`
- ebenfalls balancierter binärer Suchbaum mit Strecken in den Blättern!

# Algorithmus

FindIntersections($S$)

**Input**: Menge $S$ von Strecken

**Output**: Menge aller Schnittpunkte mit zugeh. Strecken

$\mathcal{Q} \leftarrow \emptyset$; $\mathcal{T} \leftarrow \emptyset$

**foreach** $s \in S$ **do**

    $\mathcal{Q}$.insert(upperEndPoint($s$))

    $\mathcal{Q}$.insert(lowerEndPoint($s$))

**while** $\mathcal{Q} \neq \emptyset$ **do**

    $p \leftarrow \mathcal{Q}$.nextEvent()

    $\mathcal{Q}$.deleteEvent($p$)

    handleEvent($p$)

# Algorithmus

handleEvent($p$)

$U(p) \leftarrow$ Strecken mit $p$ oberer Endpunkt
$L(p) \leftarrow$ Strecken mit $p$ unterer Endpunkt
$C(p) \leftarrow$ Strecken mit $p$ innerer Punkt
**if** $|U(p) \cup L(p) \cup C(p)| \geq 2$ **then**
    gebe $p$ und $U(p) \cup L(p) \cup C(p)$ aus
entferne $L(p) \cup C(p)$ aus $\mathcal{T}$
füge $U(p) \cup C(p)$ in $\mathcal{T}$ ein
**if** $U(p) \cup C(p) = \emptyset$ **then** //$s_l$ und $s_r$ Nachbarn von $p$ in $\mathcal{T}$
    $\mathcal{Q} \leftarrow$ prüfe $s_l$ und $s_r$ auf Schnitt unterhalb $p$
**else** //$s'$ und $s''$ linkeste und rechteste Strecke in $U(p) \cup C(p)$
    $\mathcal{Q} \leftarrow$ prüfe $s_l$ und $s'$ auf Schnitt unterhalb $p$
    $\mathcal{Q} \leftarrow$ prüfe $s_r$ und $s''$ auf Schnitt unterhalb $p$

# Übungsblatt 2 - Aufgabe 2

Karlsruhe Institute of Technology

**VL:**

Laufzeit: $\mathcal{O}((n + I) \log n)$

Speicherplatz: $\mathcal{O}(n + I)$

**Gesucht:**

Algorithmus der nur linearen Speicher benötigt.

**Frage:**

Welche Datenstruktur ist problematisch?

# Übungsblatt 2 - Aufgabe 2

Karlsruhe Institute of Technology

**VL:**

Laufzeit: $\mathcal{O}((n + I) \log n)$

Speicherplatz: $\mathcal{O}(n + I)$

**Gesucht:**

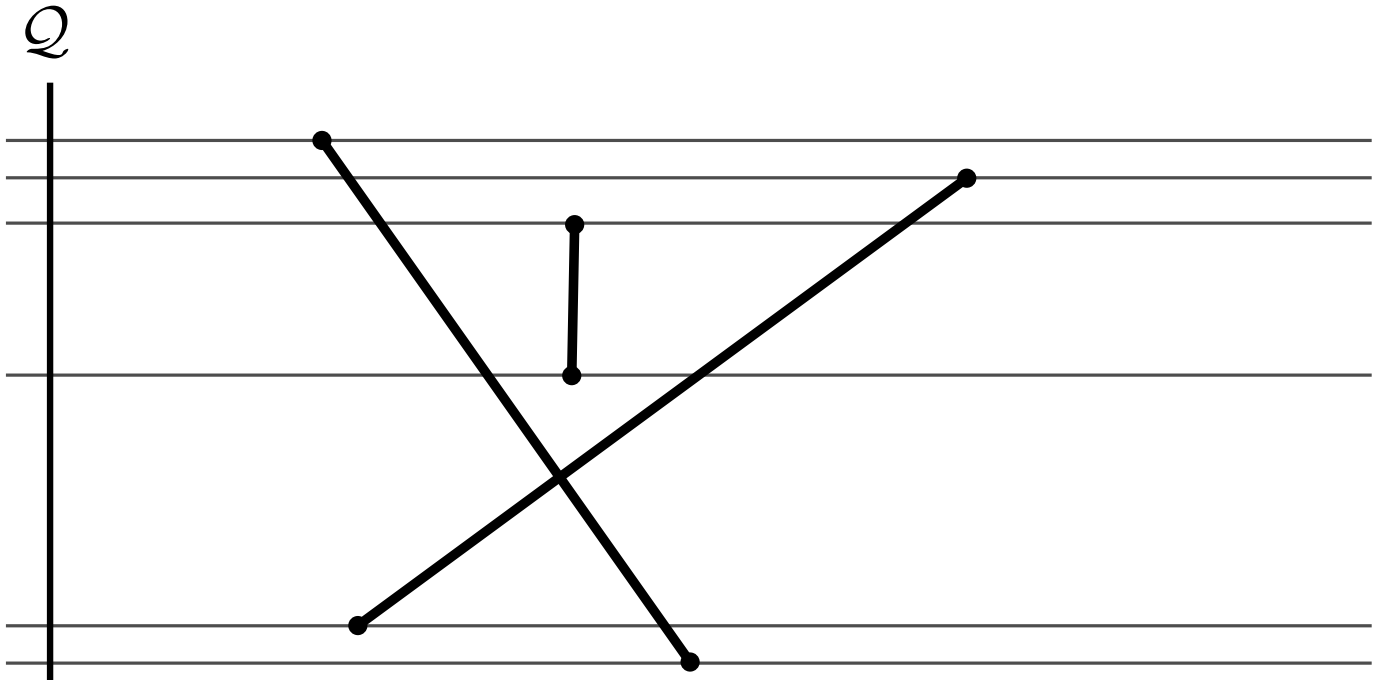
Algorithmus der nur linearen Speicher benötigt.

**Frage:**

Welche Datenstruktur ist problematisch?

Die Event-Queue enthält am Ende $2n + I$ viele Events, wobei $I \in \Omega(n^2)$ im schlimmsten Fall.

# Übungsblatt 2 - Aufgabe 2

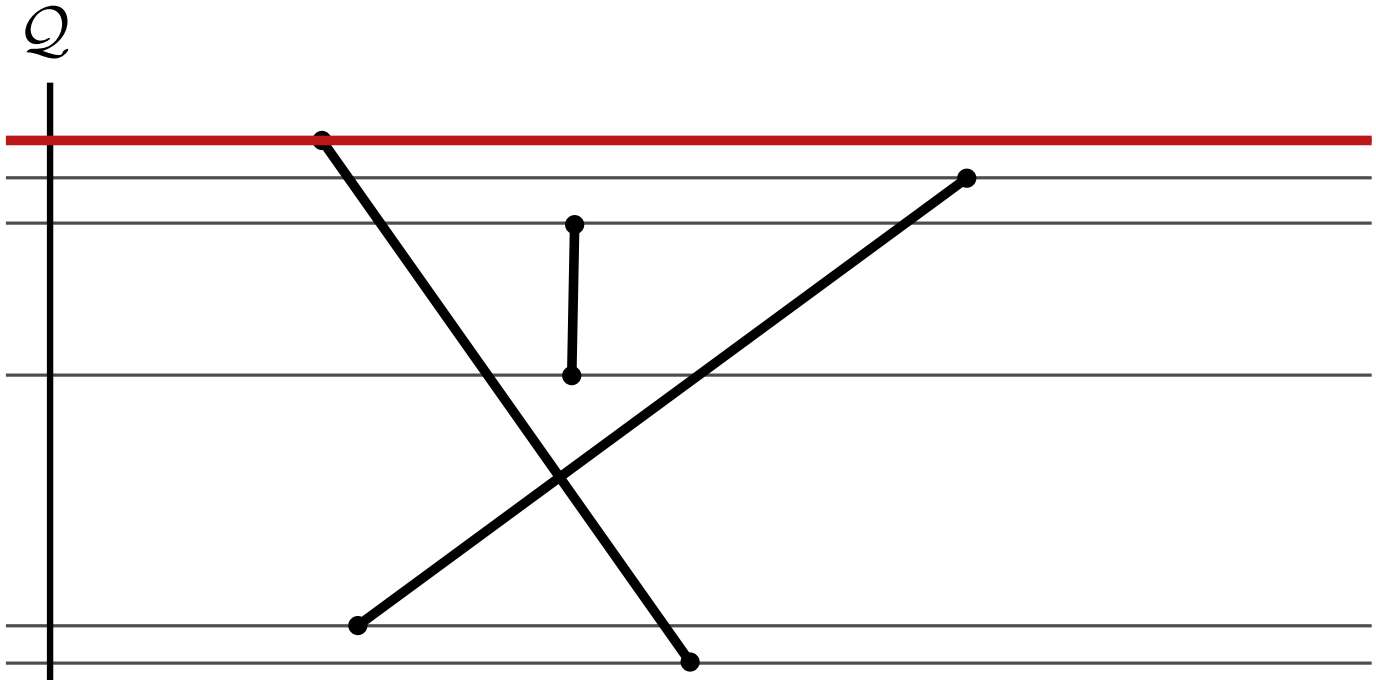



**Idee:** Speichere ausschließlich Schnittpunkte von Strecken, die gerade in $\mathcal{T}$ benachbart sind.

**Beob.:** Pro Zeitpunkt gibt es nur $O(n)$ viele solcher Schnittpunkte.

**Vorgehen:** Wenn Strecken in $\mathcal{T}$ Nachbarschaft verlieren, entferne entsprechende Schnittpunkte aus $\mathcal{Q}$

# Übungsblatt 2 - Aufgabe 2

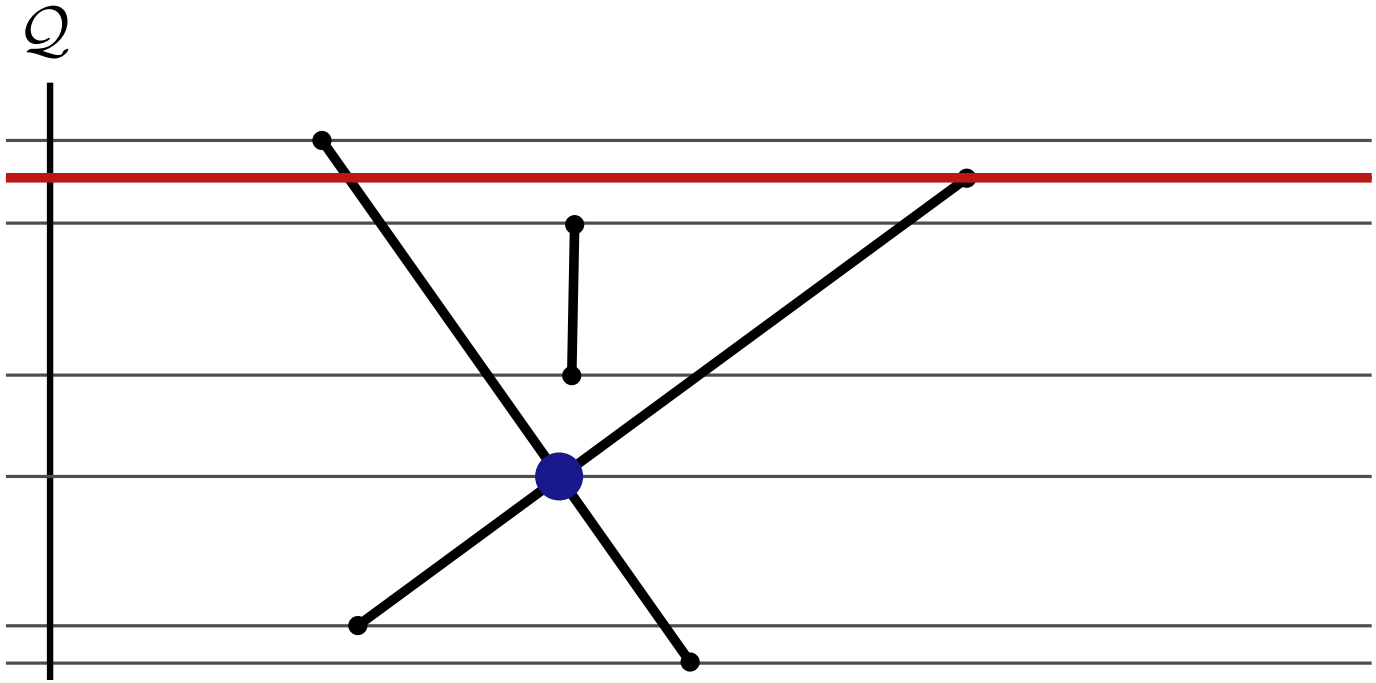



**Idee:** Speichere ausschließlich Schnittpunkte von Strecken, die gerade in $\mathcal{T}$ benachbart sind.

**Beob.:** Pro Zeitpunkt gibt es nur $O(n)$ viele solcher Schnittpunkte.

**Vorgehen:** Wenn Strecken in $\mathcal{T}$ Nachbarschaft verlieren, entferne entsprechende Schnittpunkte aus $\mathcal{Q}$

# Übungsblatt 2 - Aufgabe 2

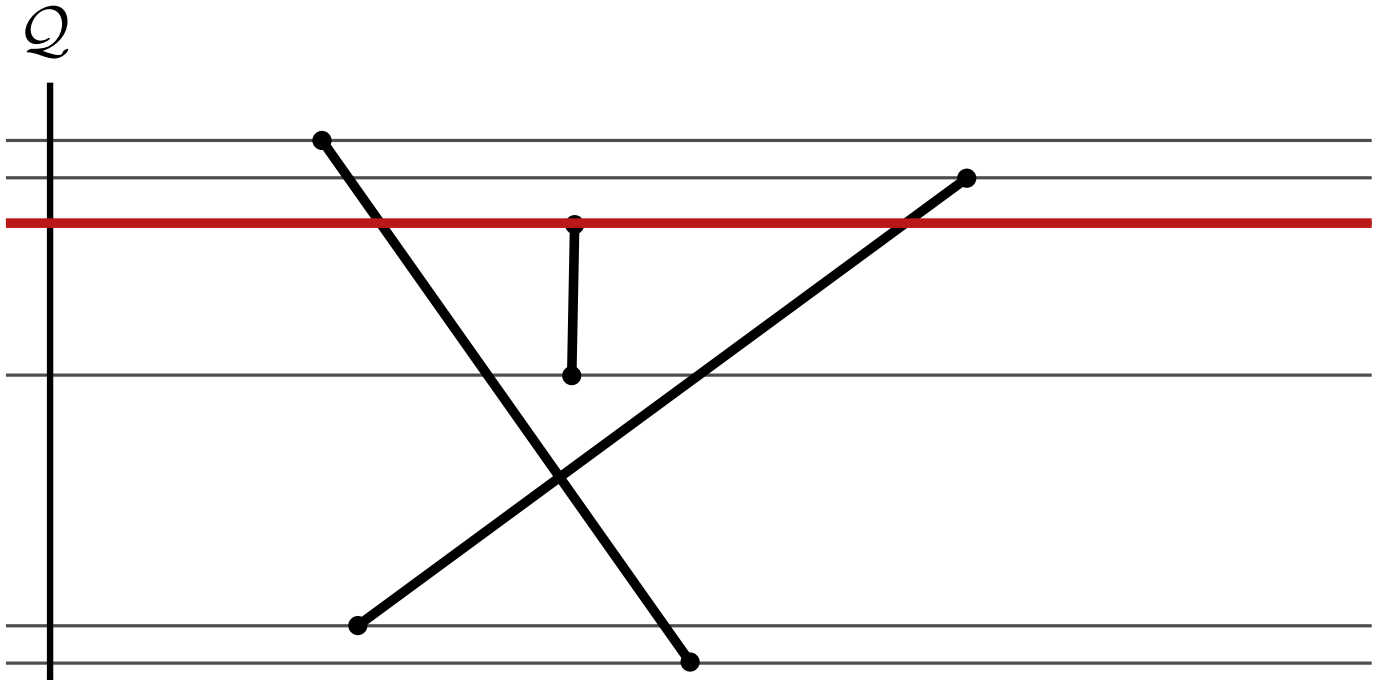



**Idee:** Speichere ausschließlich Schnittpunkte von Strecken, die gerade in $\mathcal{T}$ benachbart sind.

**Beob.:** Pro Zeitpunkt gibt es nur $O(n)$ viele solcher Schnittpunkte.

**Vorgehen:** Wenn Strecken in $\mathcal{T}$ Nachbarschaft verlieren, entferne entsprechende Schnittpunkte aus $\mathcal{Q}$

# Übungsblatt 2 - Aufgabe 2

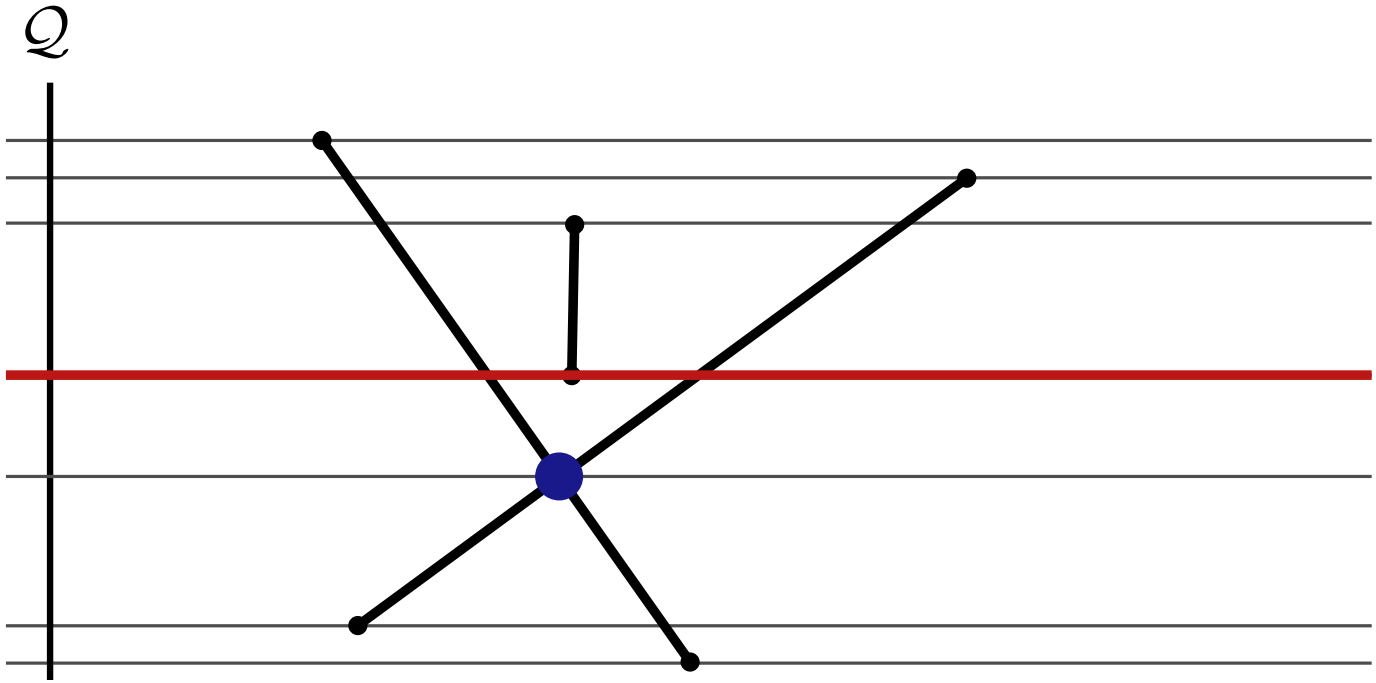



**Idee:** Speichere ausschließlich Schnittpunkte von Strecken, die gerade in $\mathcal{T}$ benachbart sind.

**Beob.:** Pro Zeitpunkt gibt es nur $O(n)$ viele solcher Schnittpunkte.

**Vorgehen:** Wenn Strecken in $\mathcal{T}$ Nachbarschaft verlieren, entferne entsprechende Schnittpunkte aus $\mathcal{Q}$

# Übungsblatt 2 - Aufgabe 2




**Idee:** Speichere ausschließlich Schnittpunkte von Strecken, die gerade in $\mathcal{T}$ benachbart sind.

**Beob.:** Pro Zeitpunkt gibt es nur $O(n)$ viele solcher Schnittpunkte.

**Vorgehen:** Wenn Strecken in $\mathcal{T}$ Nachbarschaft verlieren, entferne entsprechende Schnittpunkte aus $\mathcal{Q}$

# Übungsblatt 2 - Aufgabe 3

Karlsruhe Institute of Technology

**Gegeben:**

Endliche Punktmenge $P$

**Def.:**

*größter rechter obere Bereich (groB)* von $p \in P$: Vereinigung aller offenen achsenparallelen Quadrate, die p mit ihrer linken unteren Ecke berühren und keinen Punkt aus $P$ in ihrem Inneren enthalten.

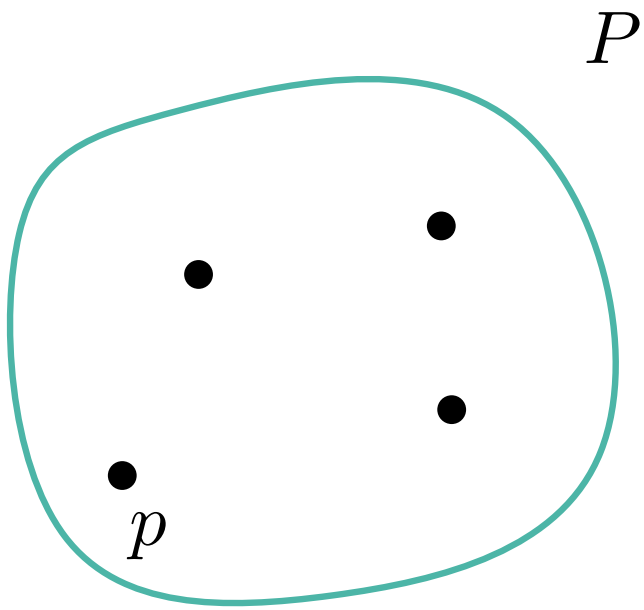
# Übungsblatt 2 - Aufgabe 3

**Gegeben:**

Endliche Punktmenge $P$

**Def.:**

*größter rechter obere Bereich (groB)* von $p \in P$: Vereinigung aller offenen achsenparallelen Quadrate, die p mit ihrer linken unteren Ecke berühren und keinen Punkt aus $P$ in ihrem Inneren enthalten.



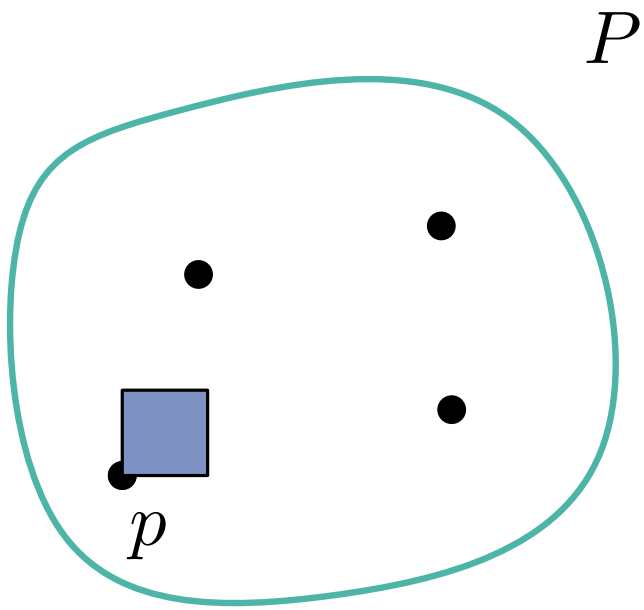
# Übungsblatt 2 - Aufgabe 3

**Gegeben:**

Endliche Punktmenge $P$

**Def.:**

*größter rechter obere Bereich (groB)* von $p \in P$: Vereinigung aller offenen achsenparallelen Quadrate, die p mit ihrer linken unteren Ecke berühren und keinen Punkt aus $P$ in ihrem Inneren enthalten.



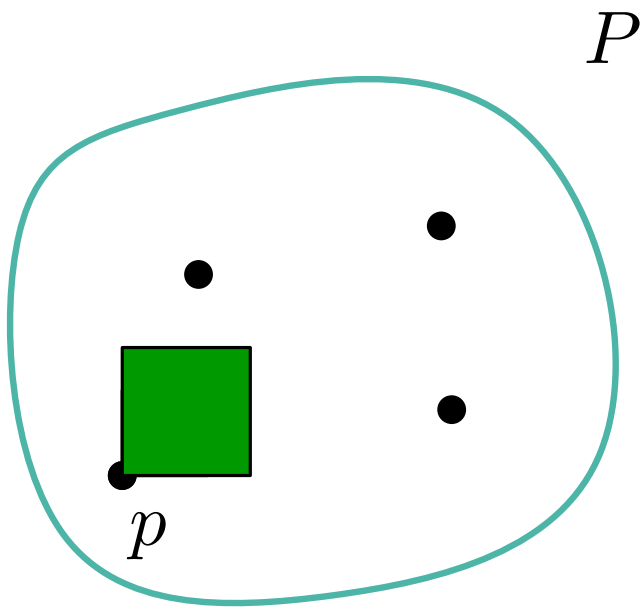
# Übungsblatt 2 - Aufgabe 3

**Gegeben:**

Endliche Punktmenge $P$

**Def.:**

*größter rechter obere Bereich (groB)* von $p \in P$: Vereinigung aller offenen achsenparallelen Quadrate, die p mit ihrer linken unteren Ecke berühren und keinen Punkt aus $P$ in ihrem Inneren enthalten.



# Übungsblatt 2 - Aufgabe 3

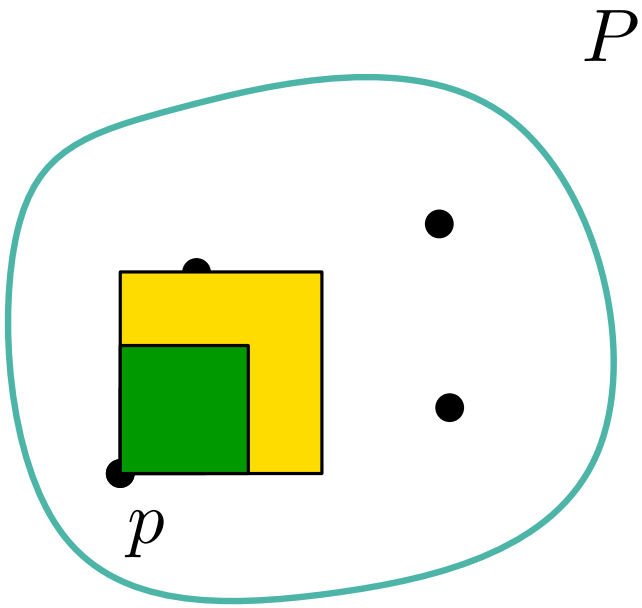
Karlsruhe Institute of Technology

**Gegeben:**

Endliche Punktmenge $P$

**Def.:**

*größter rechter obere Bereich (groB)* von $p \in P$: Vereinigung aller offenen achsenparallelen Quadrate, die p mit ihrer linken unteren Ecke berühren und keinen Punkt aus $P$ in ihrem Inneren enthalten.



# Übungsblatt 2 - Aufgabe 3

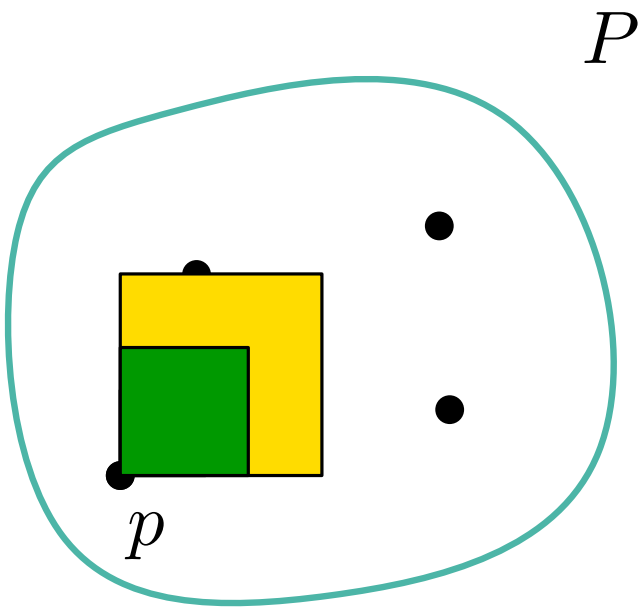
**Gegeben:**

Endliche Punktmenge $P$

**Def.:**

*größter rechter obere Bereich (groB)* von $p \in P$: Vereinigung aller offenen achsenparallelen Quadrate, die p mit ihrer linken unteren Ecke berühren und keinen Punkt aus $P$ in ihrem Inneren enthalten.

a) Zeige, dass *groB* entweder Quadrat oder Schnitt zweier offener Halbebenen



# Übungsblatt 2 - Aufgabe 3

**Gegeben:**

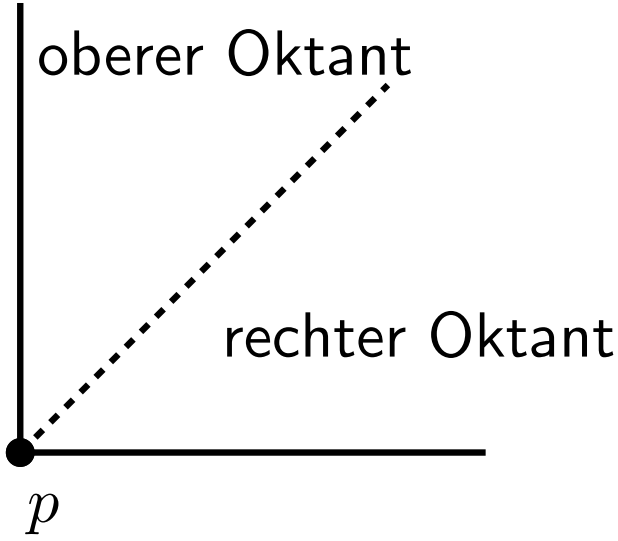
Endliche Punktmenge $P$

**Def.:**

*größter rechter obere Bereich (groB)* von $p \in P$: Vereinigung aller offenen achsenparallelen Quadrate, die p mit ihrer linken unteren Ecke berühren und keinen Punkt aus $P$ in ihrem Inneren enthalten.

a) Zeige, dass *groB* entweder Quadrat oder Schnitt zweier offener Halbebenen

b) Welche Punkte im rechten und im oberen Oktanten schränken den *groB* am stärksten ein?



# Übungsblatt 2 - Aufgabe 3

**Gegeben:**

Endliche Punktmenge $P$

**Def.:**

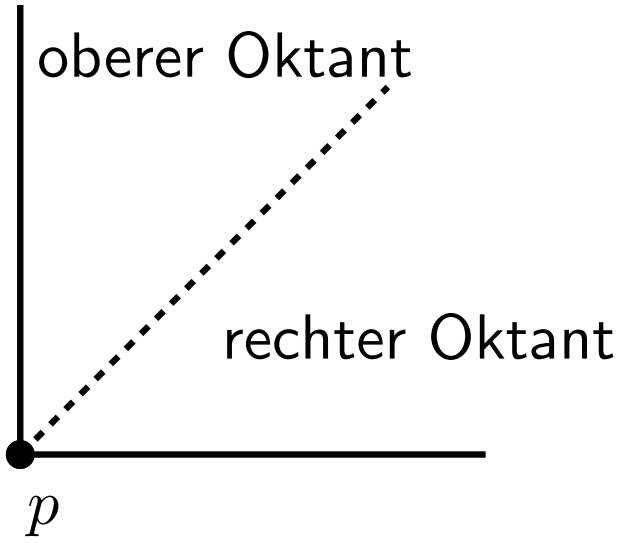
*größter rechter obere Bereich (groB)* von $p \in P$: Vereinigung aller offenen achsenparallelen Quadrate, die p mit ihrer linken unteren Ecke berühren und keinen Punkt aus $P$ in ihrem Inneren enthalten.

a) Zeige, dass *groB* entweder Quadrat oder Schnitt zweier offener Halbebenen

b) Welche Punkte im rechten und im oberen Oktanten schränken den *groB* am stärksten ein?

$o(p) \in P$: Punkt im oberen Oktant mit geringsten vert. Abstand zu $p$.

$r(p) \in P$: Punkt im rechten Oktant mit geringsten horz. Abstand zu $p$.



# Übungsblatt 2 - Aufgabe 3

**Gegeben:**

Endliche Punktmenge $P$

**Def.:**

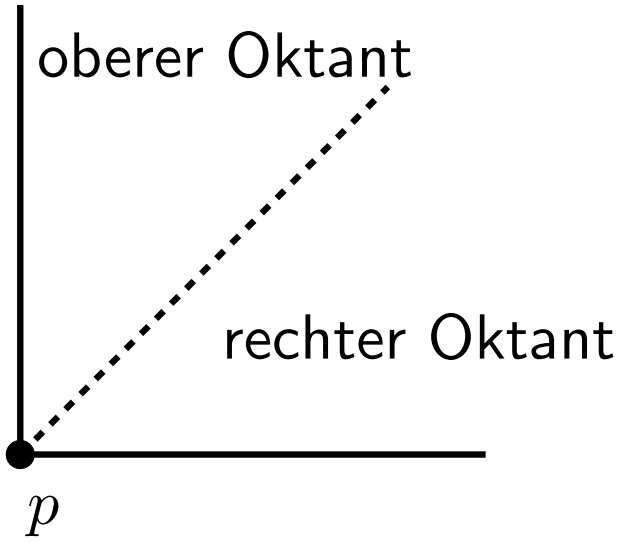
*größter rechter obere Bereich (groB)* von $p \in P$: Vereinigung aller offenen achsenparallelen Quadrate, die p mit ihrer linken unteren Ecke berühren und keinen Punkt aus $P$ in ihrem Inneren enthalten.

a) Zeige, dass *groB* entweder Quadrat oder Schnitt zweier offener Halbebenen

b) Welche Punkte im rechten und im oberen Oktanten schränken den *groB* am stärksten ein?

$o(p) \in P$: Punkt im oberen Oktant mit geringsten vert. Abstand zu $p$.

$r(p) \in P$: Punkt im rechten Oktant mit geringsten horz. Abstand zu $p$.




c) Algorithmus der für alle Punkte in $P$ den *groB* in $\mathcal{O}(n \log n)$ berechnet.

# Aufgabe 3

**Idee:** Bestimme für jeden Punkt $p$ den Punkt $o(p)$ (analog $r(p)$)

**Sweepline:** von unten nach oben

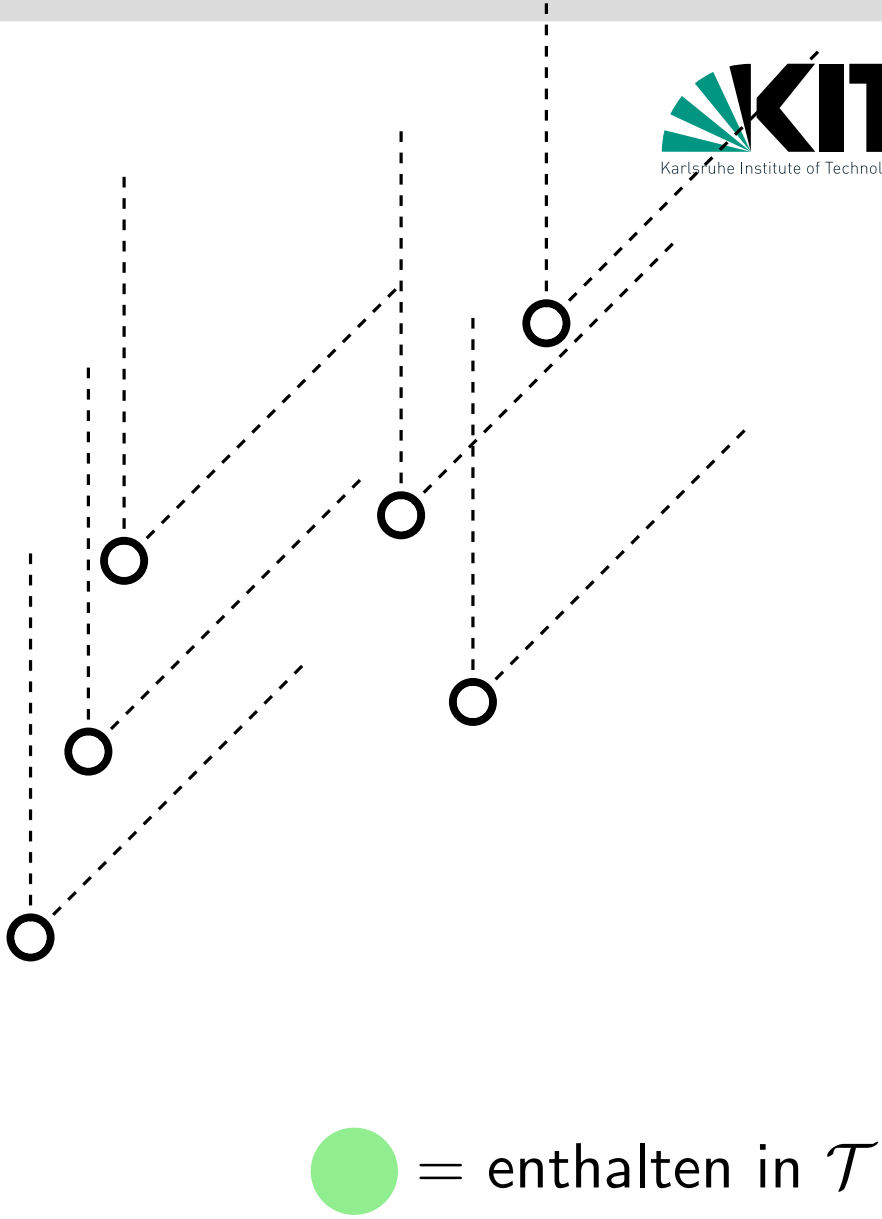
**Events:** Punkte aus $P$

**Eventbehandlung:**

1. Füge $p$ in $\mathcal{T}$ ein.
2. Finde Pkt. $p' \in \mathcal{T}$ direkt links von $p$:

**Falls** $p$ im oberen Oktant von $p'$ :

- $o(p') \leftarrow p$, entferne $p'$ aus $\mathcal{T}$
- wiederhole Schritt 2



= enthalten in $\mathcal{T}$

**Datenstruktur:**

Binärer Suchbaum $\mathcal{T}$ über $P$, mit Punkt $p \in \mathcal{T}$ , falls

1. $p$ liegt unterhalb der Sweep-Line
2. $o(p)$ wurde noch nicht bestimmt.

$\mathcal{T}$ ist inital leer und Punkte in $\mathcal{T}$ nach $x$-Koordinate sortiert.

# Aufgabe 3

**Idee:** Bestimme für jeden Punkt $p$ den Punkt $o(p)$ (analog $r(p)$)

**Sweepline:** von unten nach oben

**Events:** Punkte aus $P$

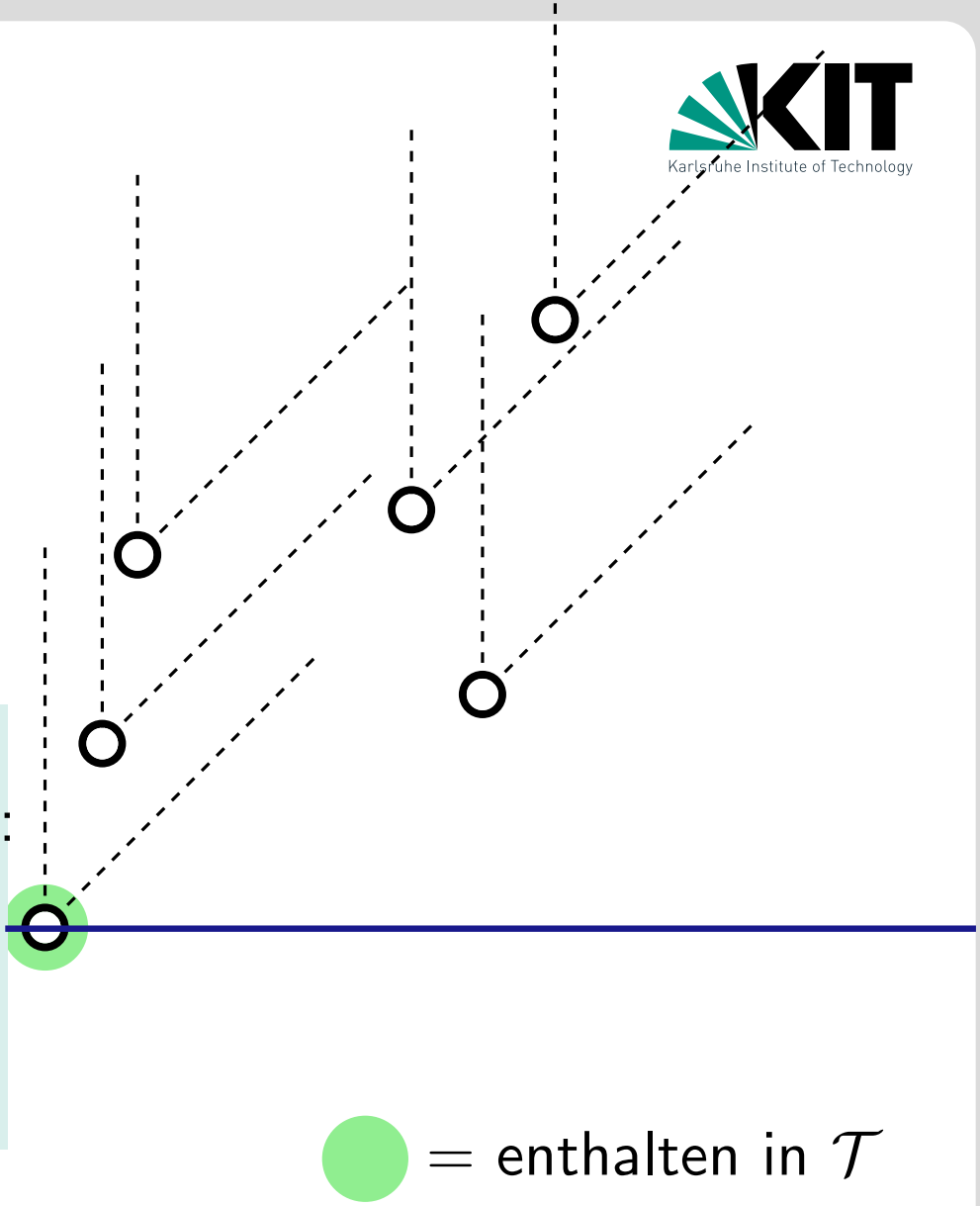
**Eventbehandlung:**

1. Füge $p$ in $\mathcal{T}$ ein.
2. Finde Pkt. $p' \in \mathcal{T}$ direkt links von $p$:

**Falls** $p$ im oberen Oktant von $p'$ :

$o(p') \leftarrow p$, entferne $p'$ aus $\mathcal{T}$

wiederhole Schritt 2



= enthalten in $\mathcal{T}$

**Datenstruktur:**

Binärer Suchbaum $\mathcal{T}$ über $P$, mit Punkt $p \in \mathcal{T}$ , falls

1. $p$ liegt unterhalb der Sweep-Line
2. $o(p)$ wurde noch nicht bestimmt.

$\mathcal{T}$ ist inital leer und Punkte in $\mathcal{T}$ nach $x$-Koordinate sortiert.

# Aufgabe 3

**Idee:** Bestimme für jeden Punkt $p$ den Punkt $o(p)$ (analog $r(p)$)

**Sweepline:** von unten nach oben

**Events:** Punkte aus $P$

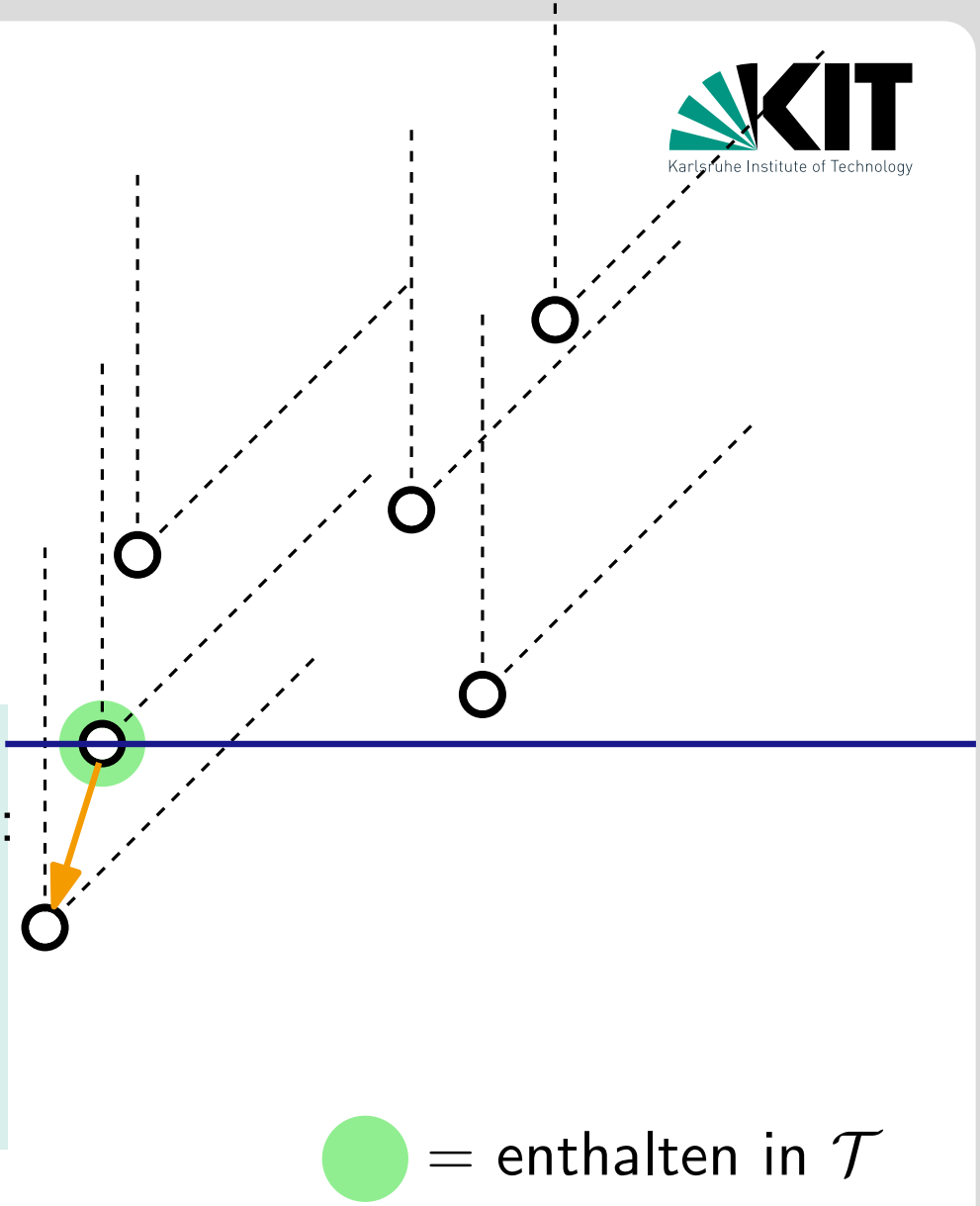
**Eventbehandlung:**

1. Füge $p$ in $\mathcal{T}$ ein.
2. Finde Pkt. $p' \in \mathcal{T}$ direkt links von $p$:

**Falls** $p$ im oberen Oktant von $p'$ :

$o(p') \leftarrow p$, entferne $p'$ aus $\mathcal{T}$

wiederhole Schritt 2



= enthalten in $\mathcal{T}$

**Datenstruktur:**

Binärer Suchbaum $\mathcal{T}$ über $P$, mit Punkt $p \in \mathcal{T}$ , falls

1. $p$ liegt unterhalb der Sweep-Line
2. $o(p)$ wurde noch nicht bestimmt.

$\mathcal{T}$ ist inital leer und Punkte in $\mathcal{T}$ nach $x$-Koordinate sortiert.

# Aufgabe 3

**Idee:** Bestimme für jeden Punkt $p$ den Punkt $o(p)$ (analog $r(p)$)

**Sweepline:** von unten nach oben

**Events:** Punkte aus $P$

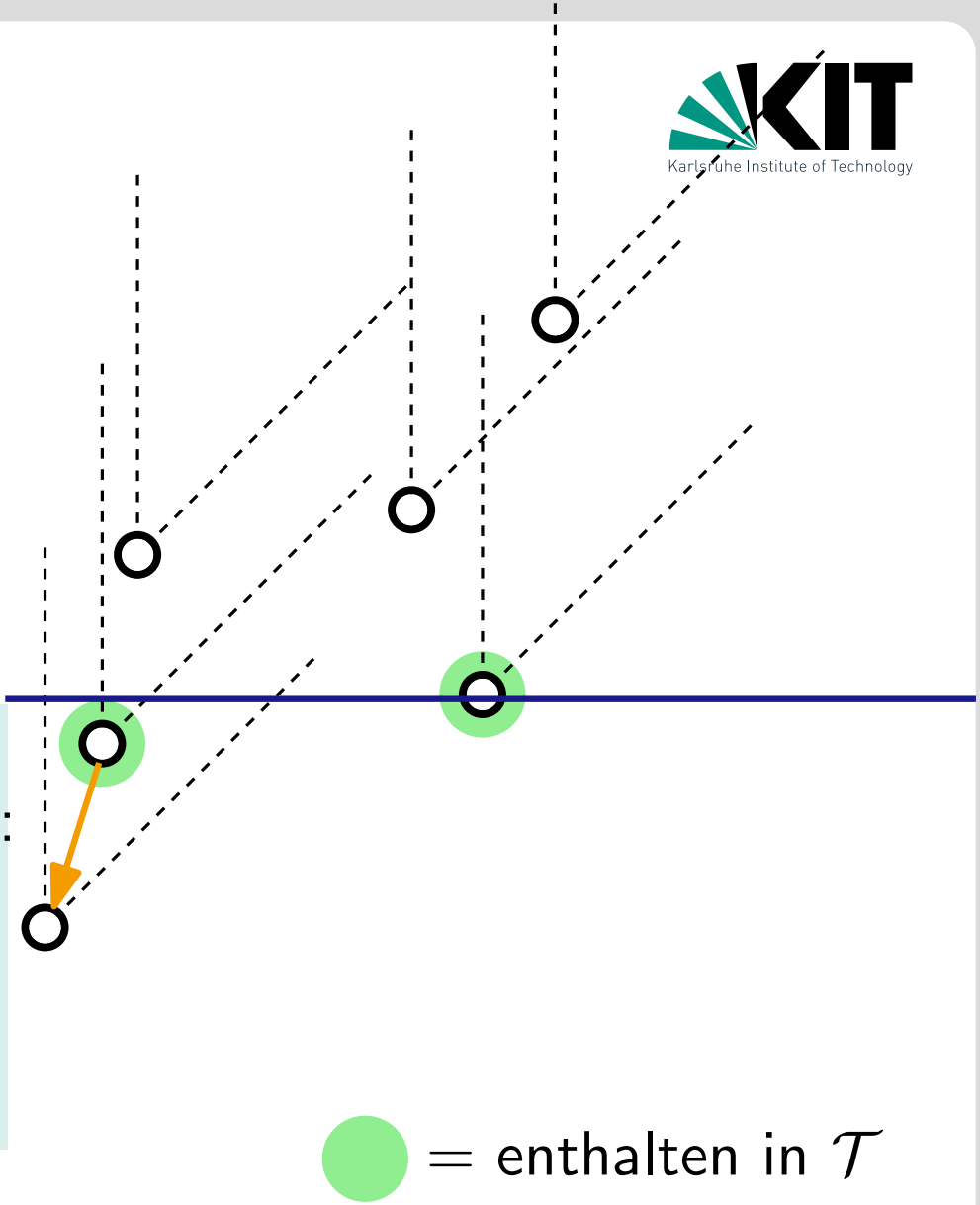
**Eventbehandlung:**

1. Füge $p$ in $\mathcal{T}$ ein.
2. Finde Pkt. $p' \in \mathcal{T}$ direkt links von $p$:

**Falls** $p$ im oberen Oktant von $p'$ :

$o(p') \leftarrow p$, entferne $p'$ aus $\mathcal{T}$

wiederhole Schritt 2



= enthalten in $\mathcal{T}$

**Datenstruktur:**

Binärer Suchbaum $\mathcal{T}$ über $P$, mit Punkt $p \in \mathcal{T}$ , falls

1. $p$ liegt unterhalb der Sweep-Line
2. $o(p)$ wurde noch nicht bestimmt.

$\mathcal{T}$ ist inital leer und Punkte in $\mathcal{T}$ nach $x$-Koordinate sortiert.

# Aufgabe 3

**Idee:** Bestimme für jeden Punkt $p$ den Punkt $o(p)$ (analog $r(p)$)

**Sweepline:** von unten nach oben

**Events:** Punkte aus $P$

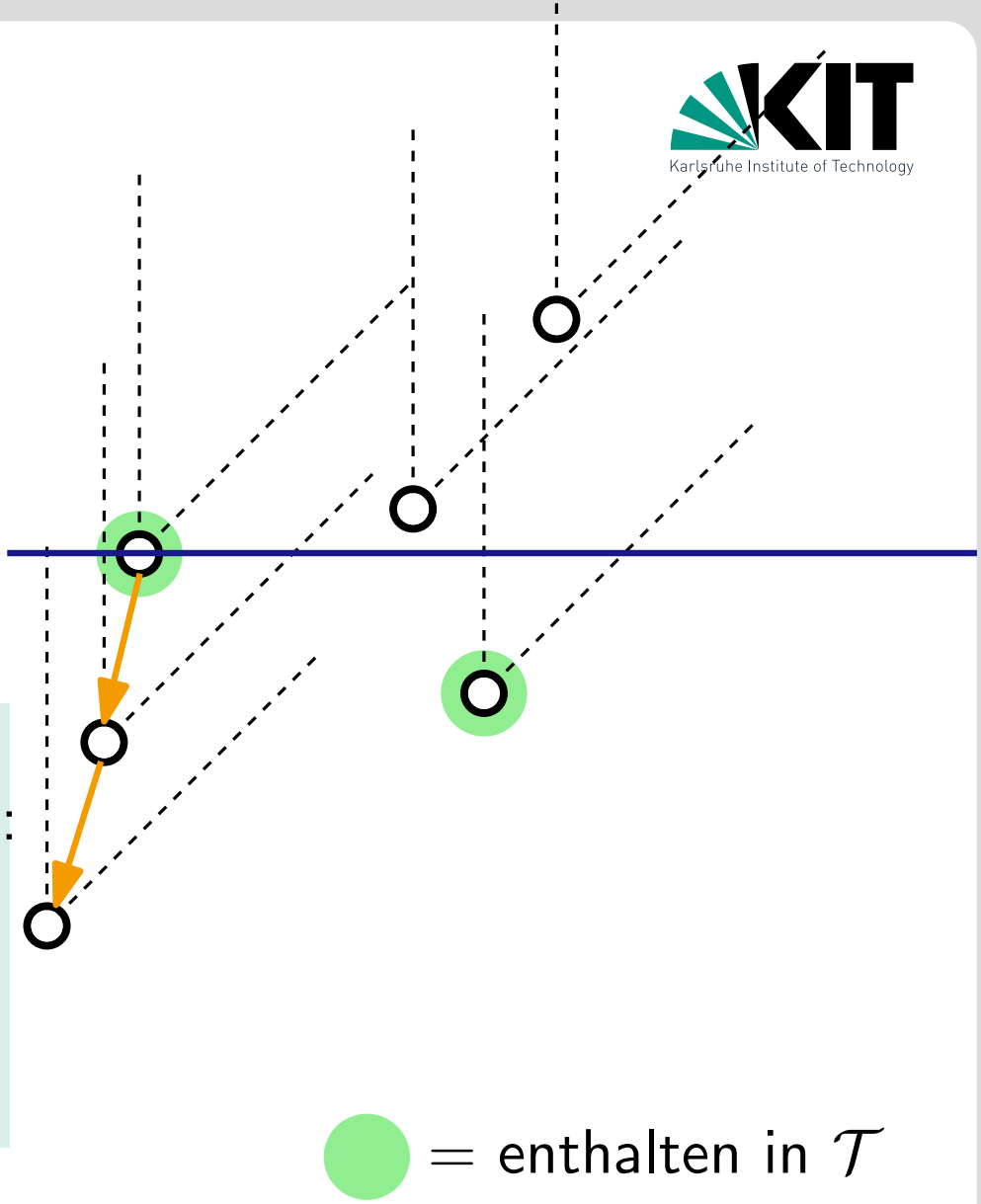
**Eventbehandlung:**

1. Füge $p$ in $\mathcal{T}$ ein.
2. Finde Pkt. $p' \in \mathcal{T}$ direkt links von $p$:

**Falls** $p$ im oberen Oktant von $p'$ :

$o(p') \leftarrow p$, entferne $p'$ aus $\mathcal{T}$

wiederhole Schritt 2



= enthalten in $\mathcal{T}$

**Datenstruktur:**

Binärer Suchbaum $\mathcal{T}$ über $P$, mit Punkt $p \in \mathcal{T}$ , falls

1. $p$ liegt unterhalb der Sweep-Line
2. $o(p)$ wurde noch nicht bestimmt.

$\mathcal{T}$ ist inital leer und Punkte in $\mathcal{T}$ nach $x$-Koordinate sortiert.

# Aufgabe 3

**Idee:** Bestimme für jeden Punkt $p$ den Punkt $o(p)$ (analog $r(p)$)

**Sweepline:** von unten nach oben

**Events:** Punkte aus $P$

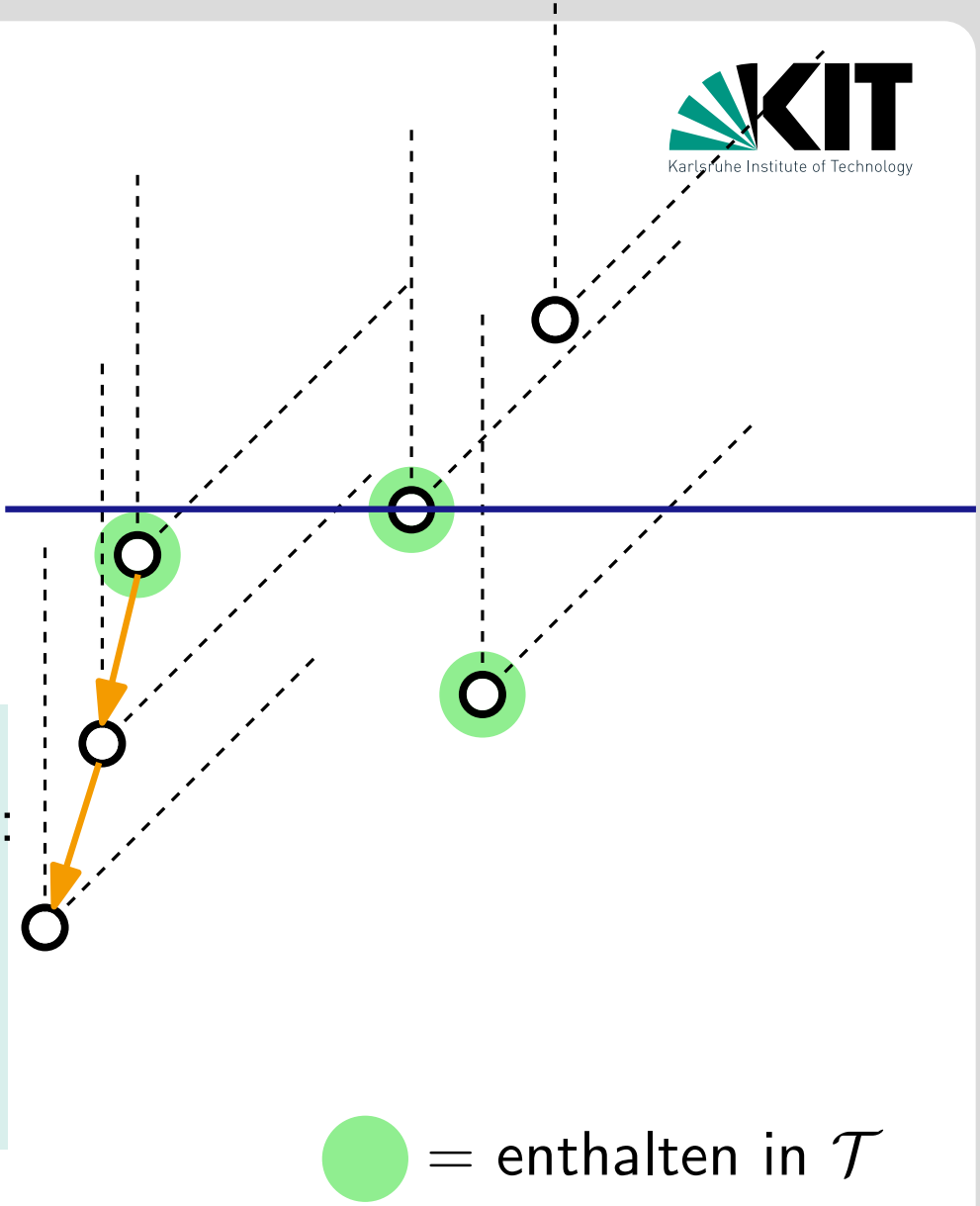
**Eventbehandlung:**

1. Füge $p$ in $\mathcal{T}$ ein.
2. Finde Pkt. $p' \in \mathcal{T}$ direkt links von $p$:

**Falls** $p$ im oberen Oktant von $p'$ :

$o(p') \leftarrow p$, entferne $p'$ aus $\mathcal{T}$

wiederhole Schritt 2




**Datenstruktur:**

Binärer Suchbaum $\mathcal{T}$ über $P$, mit Punkt $p \in \mathcal{T}$ , falls

1. $p$ liegt unterhalb der Sweep-Line
2. $o(p)$ wurde noch nicht bestimmt.

$\mathcal{T}$ ist inital leer und Punkte in $\mathcal{T}$ nach $x$-Koordinate sortiert.

# Aufgabe 3

**Idee:** Bestimme für jeden Punkt $p$ den Punkt $o(p)$ (analog $r(p)$)

**Sweepline:** von unten nach oben

**Events:** Punkte aus $P$

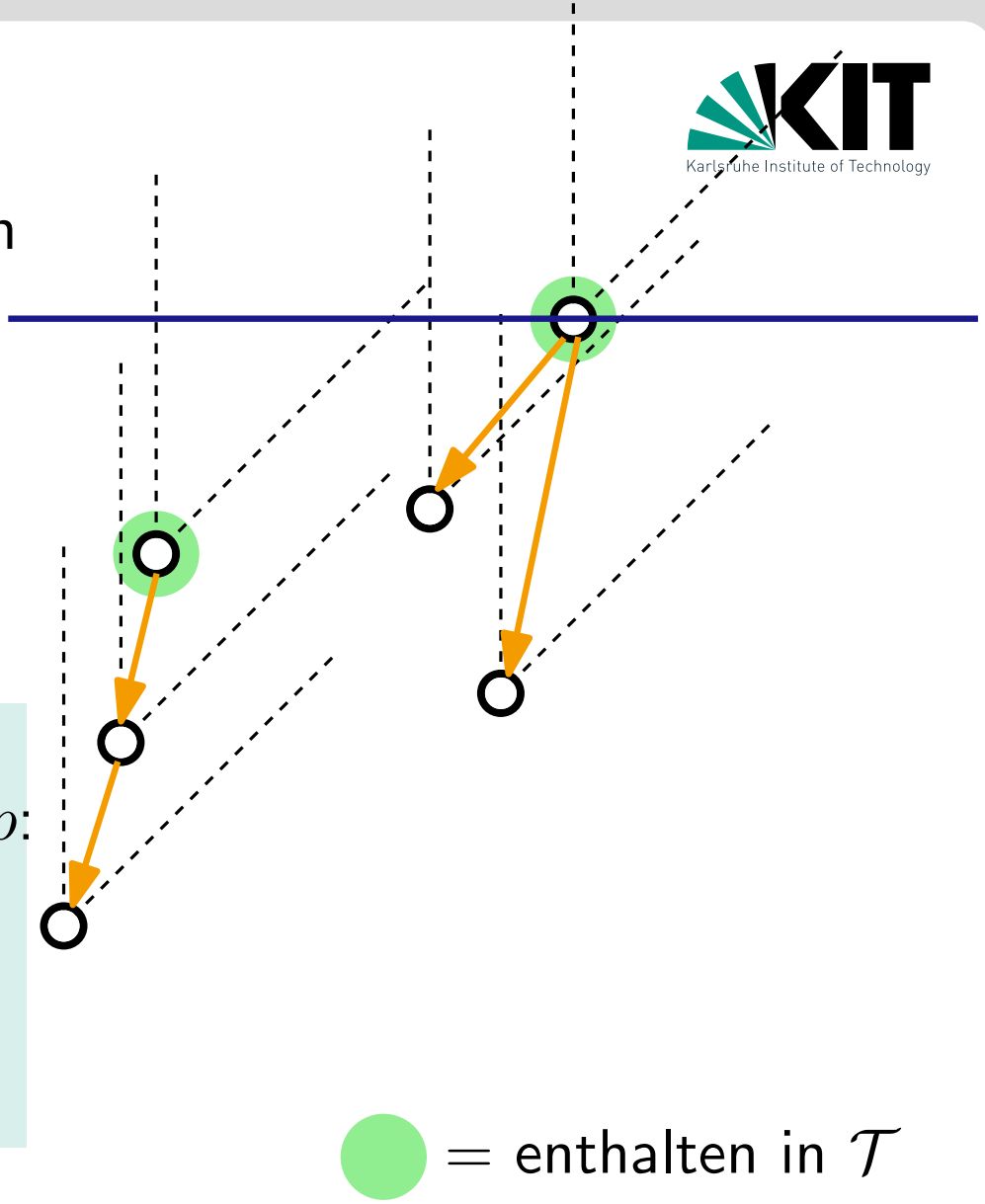
**Eventbehandlung:**

1. Füge $p$ in $\mathcal{T}$ ein.
2. Finde Pkt. $p' \in \mathcal{T}$ direkt links von $p$:

**Falls** $p$ im oberen Oktant von $p'$ :

$o(p') \leftarrow p$, entferne $p'$ aus $\mathcal{T}$

wiederhole Schritt 2



= enthalten in $\mathcal{T}$

**Datenstruktur:**

Binärer Suchbaum $\mathcal{T}$ über $P$, mit Punkt $p \in \mathcal{T}$ , falls

1. $p$ liegt unterhalb der Sweep-Line
2. $o(p)$ wurde noch nicht bestimmt.

$\mathcal{T}$ ist inital leer und Punkte in $\mathcal{T}$ nach $x$-Koordinate sortiert.