

### Übungsblatt 3

Julius Auer, Alexa Schlegel

---

**Aufgabe 1** (Graham-Scan):

*Implementieren Sie den Graham-Scan. Benutzen Sie dabei aus Gründen besserer Laufzeit und zur Vermeidung von Rundungsfehlern möglichst nur die Grundrechenarten  $+$ ,  $-$ ,  $\times$ ,  $\div$ . Eine Umrechnung in Polarkoordinaten, wie in der Vorlesung beschrieben, ist nicht nötig, um die Strahlen nach Steigung zu sortieren.*

Im Folgenden wird der implementierte Algorithmus mit Hilfe von Pseudocode beschreiben, dieser orientiert sich an der in *Introduction to Algorithms*<sup>1</sup> vorgestellten Lösung. Der Stack  $S$  wird verwendet, um mögliche Kandidaten der konvexen Hülle zu verwalten. Nach terminierung des Algorithmus enthält  $S$  die Knoten der konvexen Hülle entgegen dem Urzeigersinn.

---

**Algorithm 1** Graham-Scan( $Q$ )

---

```
1: sei  $p_0 \in Q$  mit minimaler  $y$ -Koordinate:
   bei gleicher  $y$ -Koordinate wird minimale  $x$ -Koordinate verwendet
2: sei  $p_1, p_2, \dots, p_n$  die restlichen Punkte aus  $Q$ :
   von  $p_0$  aus gesehen in Polarkoordinaten gegen den Urzeigersinn sortiert
   bei gleichem Winkel, nur Punkt mit größtem Abstand von  $p_0$  behalten
3: PUSH( $p_0, S$ )
4: PUSH( $p_1, S$ )
5: PUSH( $p_2, S$ )
6: for  $i = 3$  to  $n$  do
7:   while Winkel (NEXT-TO-TOP( $S$ ), TOP( $S$ ),  $p_i$ ) ist keine Linkskurve do
8:     POP( $S$ )
9:   end while
10:  PUSH( $p_i, S$ )
11: end for
12: return  $S$ 
```

---

Die Methode *grahamscan()* erhält als Eingabe eine Punktwolke und liefert ein Polygon zurück, welches der Konvexenhülle dieser Punktwolke entspricht.

Wir verwenden als Startpunkt des Algorithmus den Punkt der am weitestens Links liegt und die kleinsten  $y$ -Koordinate besitzt (*getMinY()*). (In der Vorlesung wird ein beliebiger Punkt in der Mitte der Punktwolke ermittelt.)

Die Sortierung der Punkte erfolgt nach ihrer Steigung. Damit die Punkte gegen den Urzeigersinn sortiert sind werden sie vorher an der Funktion  $f(x) = |x|$  gespiegelt und erst dann der Anstieg berechnet (*getRelevantPoints()*).

Um Links- und Rechtskurven zu prüfen, wird die Determinante einer  $2 \times 2$ -Matrix bestimmt, welche sich aus den zwei Vektoren der folgenden drei Punkte zusammensetzt.  $t$  ist der letzte Punkt auf dem Stack,  $ntt$  der vorletzte Punkt und  $ca$  der Punkt der aktuelle betrachtet wird. Es wird getestet ob  $\overline{t\ ca}$  links oder rechts von  $\overline{t\ ntt}$  liegt.

---

<sup>1</sup>Introduction to Algorithms, Second Edition, S. 949

```

1 public static Polygon grahamscan(Points pointcloud) {
2
3     // get minimum
4     Point min = pointcloud.getMinY();
5
6     // select only relevant points from point cloud
7     Map<MyAnstieg, Point> relevantPoints = pointcloud.getRelevantPoints();
8
9     // sort by ascent
10    Map<MyAnstieg, Point> sortedPoints = new TreeMap<MyAnstieg, Point>(Collections.
        reverseOrder());
11    for (MyAnstieg a : relevantPoints.keySet()) {
12        sortedPoints.put(a, relevantPoints.get(a));
13    }
14
15    // put first 3 points on stack
16    Stack<Point> stack = new Stack<Point>();
17    stack.push(min);
18
19    MyAnstieg ascent = ((TreeMap<MyAnstieg, Point>) sortedPoints).firstKey();
20    Point p = sortedPoints.get(ascent);
21    sortedPoints.remove(ascent);
22    stack.push(p);
23
24    ascent = ((TreeMap<MyAnstieg, Point>) sortedPoints).firstKey();
25    p = sortedPoints.get(ascent);
26    sortedPoints.remove(ascent);
27    stack.push(p);
28
29    // have a look at each point
30    for (MyAnstieg ca : sortedPoints.keySet()) { // ca ... current ascent
31
32        Point t = stack.pop(); // t ... top
33        Point ntt = stack.peek(); // ntt ... next-to-top
34        stack.push(t); // put top back on stack
35
36        // use determinant for testing left/right turns
37        Vector v1 = new Vector(ntt.getX()-t.getX(), ntt.getY() - t.getY());
38        Vector v2 = new Vector(sortedPoints.get(ca).getX() - t.getX(), sortedPoints.get(
            ca).getY() - t.getY());
39
40        Mat2x2 matrix = new Mat2x2(v1.get(0), v1.get(1), v2.get(0), v2.get(1));
41
42        // testing for not left-turns
43        while (! (matrix.getDeterminant() <= 0) ) {
44
45            // remove last point
46            stack.pop();
47
48            // have a look at the next point
49            t = stack.pop();
50            ntt = stack.peek();
51            stack.push(t);
52
53            v1 = new Vector(ntt.getX()-t.getX(), ntt.getY() - t.getY());
54            v2 = new Vector(sortedPoints.get(ca).getX() - t.getX(), sortedPoints.get(ca
                ).getY() - t.getY());
55
56            matrix = new Mat2x2(v1.get(0), v1.get(1), v2.get(0), v2.get(1));
57        }
58
59        // put current point on stack
60        stack.push(sortedPoints.get(ca));
61    }
62
63    // generate polygon from points on stack
64    Point[] points = new Point[stack.size()];
65    stack.toArray(points);
66
67    return new Polygon(points);

```

```
68     }
```

```
1  public Point getMinY() {
2      Point min = points[0];
3      for(Point p : points) {
4          if(min.getY() <= p.getY()) {
5              if(min.getY() == p.getY()) {
6                  // Compare x-values
7                  if(min.getX() > p.getX()) {
8                      min = p;
9                  }
10             }
11         }
12         else {
13             min = p;
14         }
15     }
16     return min;
17 }
```

```
1  public Map<MyAnstieg, Point> getRelevantPoints() {
2      Point min = getMinY();
3      Map<MyAnstieg, Point> relevantPoints = new HashMap<MyAnstieg, Point>();
4
5      for(Point p : points) {
6
7          if(p != min) {
8
9              Double d = p.toPosition().subtract(min.toPosition()).getMirroredAscent();
10             MyAnstieg m = new MyAnstieg(d);
11
12             if (Double.isInfinite(d)) {
13                 m = new MyAnstieg(Double.MAX_VALUE);
14             }
15
16             if(!relevantPoints.containsKey(m)) {
17                 relevantPoints.put(m, p);
18             } else {
19                 Point cp = relevantPoints.get(m);
20                 double distanceold = cp.getX()*cp.getX()+cp.getY()*cp.getY();
21                 double distancenew = p.getX()*p.getX()+p.getY()*p.getY();
22
23                 if(distancenew > distanceold) {
24                     relevantPoints.remove(m);
25                     relevantPoints.put(m, p);
26                 }
27             }
28         }
29     }
30     return relevantPoints;
31 }
```

Zur Veranschaulichung des Algorithmus gibt es eine Testklasse, wo die einzelnen Schritte visualisiert werden. Hier ein Beispiel mit 10 zufälligen Punkten:

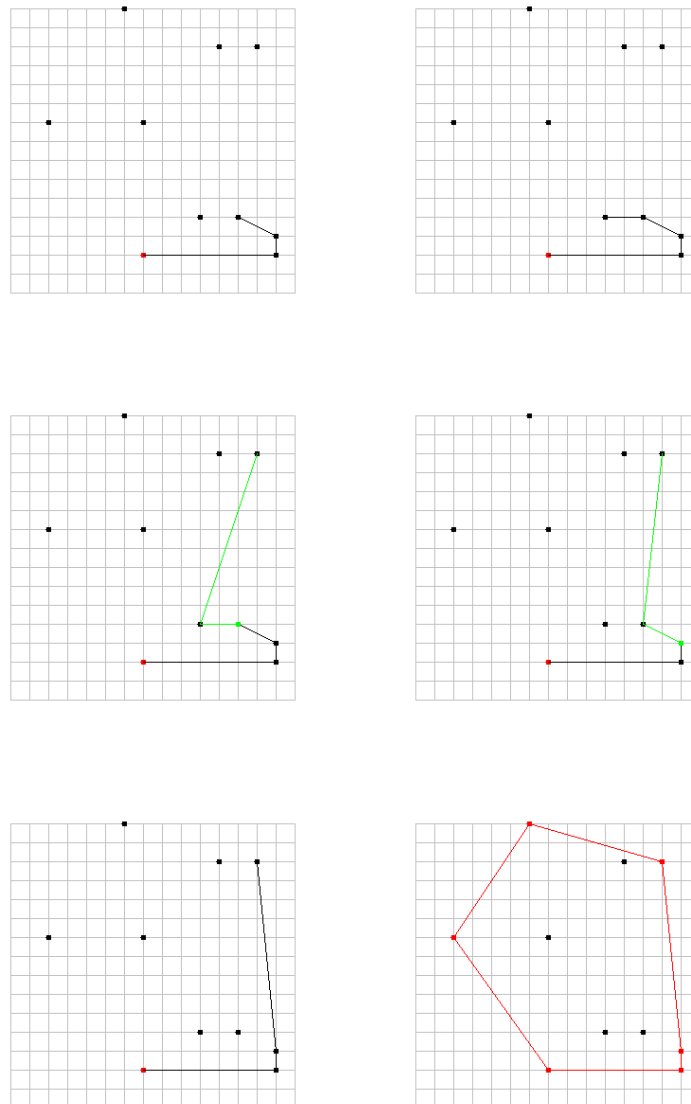


Abbildung 1: Die ersten 5 Schritte und die resultierende Konvexehülle.

**Aufgabe 2** (inkrementelle Konstruktion der konvexen Hülle):

Die ersten drei Punkte  $S = p_1, p_2, p_3$  werden zunächst nach X-Koordinate aufsteigend und dann nach Y-Koordinate absteigend sortiert. Die konvexe Hülle  $CH(S) = S$  ist somit ein Polygonzug der im Uhrzeigersinn verläuft. Ferner wird das Center of Mass  $com = ((x_1 + x_2 + x_3)/3, (y_1 + y_2 + y_3)/3)$  bestimmt. Außerdem wird als Datenstruktur ein binärer Suchbaum  $T$  angelegt, der  $search(p)$ ,  $insert(p)$  und  $delete(p)$  in  $O(\log n)$  garantiert (z.B. ein AVL-Baum) - in diesen werden  $p_1, p_2, p_3$  eingefügt, wobei die totale Ordnung gegeben ist durch

$$p_i < p_j \text{ g.d.w. } \frac{y_i - y_{com}}{x_i - x_{com}} < \frac{y_j - y_{com}}{x_j - x_{com}}$$

wenn also die Steigung zwischen  $com$  und  $p_i$  kleiner ist, als die von  $com$  und  $p_j$ . Diese Ordnung

gilt auch für zukünftige Operationen auf dem Baum.  $search(p)$  soll hierbei den Punkt aus  $CH(S)$  liefern, der die nächst kleinere Steigung im Vergleich zu  $p$  hat. Diese Initialisierung benötigt offensichtlich nur  $O(1)$  Zeit.

Für jeden weiteren Punkt  $p_n$  um den die konvexe Hülle  $S = p_1, \dots, p_{n-1}$  erweitert werden soll wird nun  $CH(S, p_n, T)$  aufgerufen, wobei  $left(p, \vec{q})$  die Funktion sei, die prüft ob  $p$  auf der linken Seite des Vektors  $\vec{q}$  liegt und  $S.insert(p_i, j)$  den Punkt  $p_i$  hinter dem Punkt  $p_j$  zum Polygonzug  $S$  der konvexen Hülle hinzufügt:

---

**Algorithm 2**  $CH(S, p_n, T)$

---

```

1:  $p_i = T.search(p_n)$ 
2: if  $left(p_n, \overrightarrow{p_i p_{i+1}})$  then
3:   return
4: end if
5:  $T.insert(p_n)$ 
6:  $S.insert(p_n, i)$ 
7:  $j = i$ 
8: while  $left(p_n, \overrightarrow{p_{j-1} p_j})$  do
9:    $T.remove(p_j)$ 
10:   $S.remove(p_j)$ 
11:   $j = j - 1$ 
12: end while
13: while  $left(p_n, \overrightarrow{p_i p_{i+1}})$  do
14:   $T.remove(p_i)$ 
15:   $S.remove(p_i)$ 
16:   $i = i + 1$ 
17: end while

```

---

Korrektheit:  
TODO

Speicherplatz:  
TODO

Laufzeit:  
TODO

**Aufgabe 3** (untere Schranke):

Angenommen die konvexe Hülle der Punkte  $S = \{(a_1, a_1^2), \dots, (a_n, a_n^2)\}$  ließe sich in weniger als  $\Omega(n \cdot \log n)$  bestimmen. Die konvexe Hülle hat hier stets die Form  $CH(S) = p_1, \dots, p_n$  mit  $x_i \geq x_{i+1}$  und  $y_i \geq y_{i+1}$  (vgl. auch Abb. 1).

Beweis (Induktion):

I.A.: Der Polygonzug  $p_1, p_2, p_3$  aus 3 Punkten beschreibt offensichtlich immer eine konvexe Hülle.

I.V.: Der Polygonzug  $p_1, \dots, p_n$  mit  $x_i \geq x_{i+1}$  beschreibt die konvexe Hülle von  $\{p_1, \dots, p_n\}$

$n \rightarrow n + 1$ : Alle Punkte  $p_1, \dots, p_n$  liegen rechts der Geraden  $\overrightarrow{p_{n+1} p_1}$ , da stets

$$\begin{aligned} \frac{y_1 - y_{n+1}}{x_1 - x_{n+1}} &> \frac{y_1 - y_n}{x_1 - x_n} \\ &= \frac{x_1^2 - x_{n+1}^2}{x_1 - x_{n+1}} > \frac{x_1^2 - x_n^2}{x_1 - x_n} \end{aligned}$$

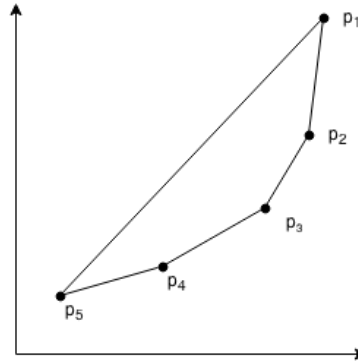


Abbildung 2: Konvexe Hülle (keine lineare Skala!)

(aufgrund der Tatsache, dass nach Voraussetzung  $x_n \geq x_{n+1}$ ) und nach I.V. alle  $p_i$  mit  $i < n$  schon vorher rechts der Geraden  $\overrightarrow{p_n p_1}$  lagen.  $p_n$  (und somit auch alle  $p_i$  mit  $i < n$ ) kann jedoch auf keinen Fall aus der konvexen Hülle "gestrichen" werden, da (nach derselben Rechnung) die Steigung von  $\overrightarrow{p_n p_{n-1}}$  steiler sein muss, als die von  $\overrightarrow{p_{n+1} p_n}$ . Damit muss die konvexe Hülle  $CH(p_1, \dots, p_{n+1}) = p_1, \dots, p_{n+1}$  sein.

Somit könnte man den Polygonzug in linearer Zeit ablaufen um die sortierte Folge  $a_n, \dots, a_1$  zu erhalten (da stets  $x_i \geq x_{i+1}$ ). Das Sortieren der Folge hätte somit nur soviel Zeit benötigt wie die Konstruktion der konvexen Hülle zzgl.  $O(n)$  für das Berechnen der Quadrate und noch einmal  $O(n)$  für das Ablaufen des Polygonzugs. Dies ist nach Annahme jedoch nicht möglich: die Konstruktion der konvexen Hülle muss folglich  $\Omega(n \cdot \log n)$  Zeit benötigen.