Aufgabe 4: Schrebergärten

Team-ID: 00848

Team-Name: WirSchlagenEuchZuWabbel!

Bearbeiter dieser Aufgabe:
Erik Klein

26. November 2018

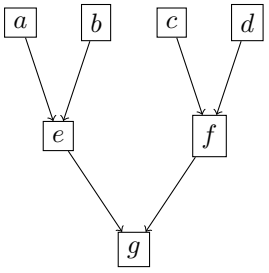
Inhaltsverzeichnis

1 Aufgabe

Gegeben ist eine Liste von Schrebergärten, Rechtecke mit Länge und Höhe, die so angeordnet werden müssen, dass sie in ein möglichst kleines Rechteck passen.

2 Lösungsidee

Meine Idee war es, das Problem so in Teilprobleme aufzuteilen, dass immer nur zwei Rechtecke zu einem grösseren Rechteck zusammengesetzt werden müssen.
Aus der Anfangsliste wollte ich Paare von Rechtecken bilden und diese dann zu neuen Rechtecken zusammensetzen, in denen beide Rechtecke platziert sind. Dadurch erhalte ich eine neue, halb so lange Liste. Diesen Prozess wollte ich rekursiv auf der resultierenden Liste wiederholen, wobei die Länge der Liste immer wieder halbiert wird, bis ich nur noch ein Rechteck, in dem alle Anfangsrechtecke enthalten sind, erhalte.
So wird beispielsweise auf die Liste mit den Rechtecken a, b, c und d verfahren:

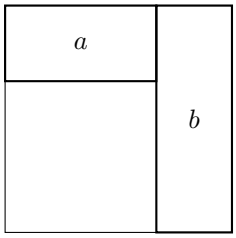


In der ersten Stufe werden a und b zu e und c und d zu f zusammengefügt. In der zweiten Stufe werden

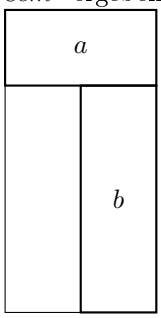
die beiden resultierten Rechtecke zu g zusammengefügt. Da die Liste jetzt nur noch die Länge eins hat, kann die rekursion abgebrochen werden.
Da sich die Länge auf jeder Stufe halbiert, braucht man für n Elemente $\lg n$ Stufen.

Um zwei zwei Rechtecke zu einem neuen zusammenzusetzen, gibt es zwei Möglichkeiten. Man kann die Rechtecke entweder nebeneinander oder übereinander legen. Wenn man sie nebeneinander legt, wird die Höhe h nicht grösser, sondern zur maximalen Höhe der beiden Rechtecke:
$h_{\text{res}} \leftarrow max(h_1, h_2)$
Durch das Nebeneinanderlegen addieren sich die Längen x der Rechtecke auf:
$x_{\text{res}} \leftarrow x_1 + x_2$
Beim Übereinanderlegen ist es genau anders herum, die Höhen addieren sich auf und die Längen werden zur maximalen Länge der beiden Rechtecke.

Ich wollte deshalb prüfen, in welchem der beiden Fälle der resultierende Flächeninhalt $h_{\text{res}} \times x_{\text{res}}$ kleiner ist und dann diesen Fall auswählen.
Beispielsweise sollen die Rechtecke a mit Grösse $2cm \times 1cm$ und b mit Grösse $1cm \times 3cm$ zusammengesetzt werden:



Durch Nebeneinanderlegen würde sich ein Rechteck der Grösse $3cm \times 3cm$ mit einem Flächeninhalt von $9cm^2$ ergeben.
Durch Übereinanderlegen würde sich ein Rechteck der Grösse $2cm \times 4cm$ mit einem Flächeninhalt von $8cm^2$ ergeben:



Da in diesem Fall durch das Übereinanderlegen ein kleinerer Flächeninhalt entsteht, würde man dieses wählen. Diese Vorgehensweise wird in folgender Methode ausgedrückt:

Algorithmus 1 : Zusammenfügen zweier Rechtecke

```
1 Function zusammenfügen(Rechteck_1, Rechteck_2):
2     if max(x_1, x_2) × (y_1 + y_2) ≥ (x_1 + x_2) × max(y_1, y_2) then
3         return new Rechteck(x_1 + x_2, max(y_1, y_2))
4     else
5         return new Rechteck(max(x_1, x_2), y_1 + y_2)
6     end
```

Damit der Flächeninhalt des finalen Rechtecks niedrig ist, müssen wir versuchen, die Lücken, also die Bereiche in den aus zwei Rechtecken zusammengesetzten Rechtecken, die nicht von diesen ausgefüllt werden, klein zu halten.
Beim Nebeineinanderlegen ergibt sichd er Flächeninhalt der Lücke, den wir minimieren wollen, als

$$A_{\text{lücke}} = \begin{cases} x_1 \times (h_2 - h_1), & \text{if } h_2 \geq h_1 \\ x_2 \times (h_1 - h_2), & \text{otherwise} \end{cases}$$

, beim Übereinanderlegen als

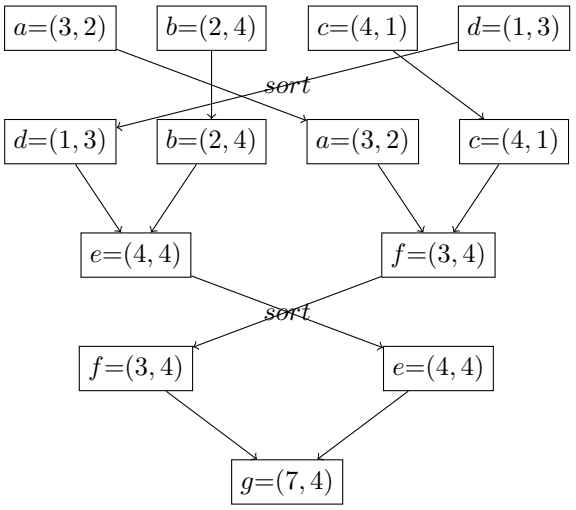
$$A_{\text{lücke}} = \begin{cases} h_1 \times (x_2 - x_1), & \text{if } x_2 \geq x_1 \\ h_2 \times (x_1 - x_2), & \text{otherwise} \end{cases}$$

Wenn wir für kleine Lücken sorgen wollen, brauchen wir also $x_1 \approx x_2$ und $h_1 \approx h_2$, die Rechtecke, die zusammengefügt werden, sollen also aus möglichst ähnliche Abmessungen haben. Meine Idee war daher, auf jeder Stufe, bevor die Paare von Rechtecken gebildet werden, die Rechtecke zu sortieren, sodass ähnliche Rechtecke nebeneinander liegen und dann jeweils zwei nebeneinanderliegende Rechtecke als Paar zu nehmen. Ich habe empirisch herausgefunden, dass sich eine Sortierung der Rechtecke nach mit erster Priorität der Länge und mit zweiter Priorität der Höhe am besten eignet.
Um diese Sortierung vorzunehmen, müssen wir eine Totalordnung für Rechtecke definieren. Dies erreicht folgende Methode:

Algorithmus 2 : Vergleichen zweier Rechtecke

```
1  Function compare(Rechteck_1, Rechteck_2):
2      if x_1 > x_2 then
3          return +1
4      if x_1 < x_2 then
5          return −1
6      if y_1 > y_2 then
7          return +1
8      if y_1 < y_2 then
9          return −1
10     return 0
```

Die Sortierung nach dieser Ordnung wollte ich in linearithmischer Laufzeit mit MergeSort vornehmen. Hier ist ein Beispiel für die Anwendung des Verfahren mit Sortierung:



Auf jeder Stufe wird also erst sortiert und dann jeweils zwei nebeneinanderliegende Rechtecke zu einem neuen zusammengefügt. Dieser Prozess wird rekursiv wiederholt, bis die Länge der Liste 1 ist. Dies ergibt Algorithmus 3.
Der Algorithmus hat eine Laufzeitkomplexität von $O(n \lg^2 n)$.

Algorithmus 3 : Berechnen der Anordnung der Rechtecke

```
1  Function pack(Rechtecke):
2      l ← len(Rechtecke);
3      if l = 1 then
4          return Rechtecke[0]
5      end
6      Rechtecke_neu ← new Rechteck[(l + 1)/2];
7      MergeSort(Rechtecke);
8      for i ← 0 to l/2 do
9          Rechtecke_neu[i] ← zusammenfügen(Rechtecke[2i], Rechtecke[2i + 1]);
10     end
11     if (l mod 2) = 1 then
12         Rechtecke_neu[l/2] ← Rechteck[l − 1]
13     end
14     return pack(Rechtecke_neu)
```

3 Umsetzung

Das Verfahren wurde in Java implementiert.
Zum Repräsentieren der Rechtecke wurde die Klasse Rechteck erstellt, welche die Attribute x und y für Länge und Höhe besitzt. Sie implementiert das Interface Comparable, in dem ich in der Methode *compareTo*() die Totalordnung aus Algorithmus 2 festgelegt habe. Dadurch kann ich die Rechtecke mit *Arrays.sort*() sortieren.

Um in Algorithmus 3 kein neues Array erstellen zu müssen, wird immer das gleiche Array wiederverwendet und der Index *hi*, bis zu welchem es verwendet wird, weitergegeben.

Die Eingabe für das Programm habe ich so festgelegt: In der ersten Zeile wird ein positiver Integer n erwartet, der die Anzahl der Rechtecke angibt. In den folgenden n Zeilen soll mit Leerzeichen getrennt jeweils die Länge und Höhe der Rechtecke stehen.
Die Rechtecke werden mithilfe eines Scanners eingelesen.

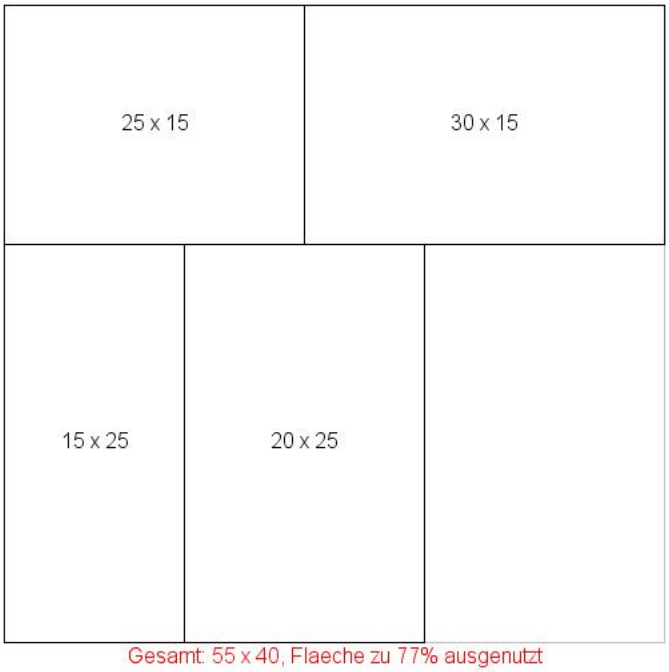
Die grafische Ausgabe wird mithilfe der Klasse StdDraw aus algs4.jar von Robert Sedgewick and Kevin Wayne durchgeführt.
Um die Rechtecke zu zeichnen, muss ich ihre Positionen kennen. Deshalb habe ich zur Klasse Rechteck die Attribute *posx* und *posy* hinzugefügt, mit denen ich die Positionen der Rechtecke zu speichern. Diese sind anfangs 0 und ändern sich in der Methode *zusammenfügen* wie folgt: Beim Übereinanderlegen eines $Rechtecks_1$ auf ein $Rechteck_2$ bleibt die Position von $Rechteck_2$ gleich, *posy* von $Rechteck_1$ ändert sich zu y_2. Beim Legen eines $Rechtecks_1$ neben ein $Rechteck_2$ bleibt die Position von $Rechteck_2$ gleich, *posx* von $Rechteck_1$ ändert sich zu x_2.

Wenn ein Rechteck seine Position verändert, ändert sich dadurch direkt die Position der beiden Rechtecke, welche in diesem enthalten sein können. Um die Position herauszufinden, an der ich ein Rechteck zeichnen muss, muss ich also die eigene Position und die Positionen aller Rechtecke, in denen das Rechteck enthalten ist, aufsummieren.
Deshalb enthält die Klasse Rechteck auch Links zu den beiden wohlmöglich im Rechteck enthaltenen Rechtecken. Diese sind anfangs *null* und werden in der Methode *zusammenfügen* gesetzt.
In der Methode *draw*(*Rechteck*) werden rekursiv die Links zu allen im Rechteck enthaltenen Rechtecken abgerufen, bis die Anfangsrechtecke, welche nicht aus kleineren Rechtecken zusammengefügt wurden und bei denen die Links *null* entsprechen, erreicht sind. Für jedes Anfangsrechteck wurden die Positionen der Rechtecke auf dem Weg zu diesem aufaddiert, weshalb das Anfangsrechteck dann an der richtigen Position gezeichnet werden kann.

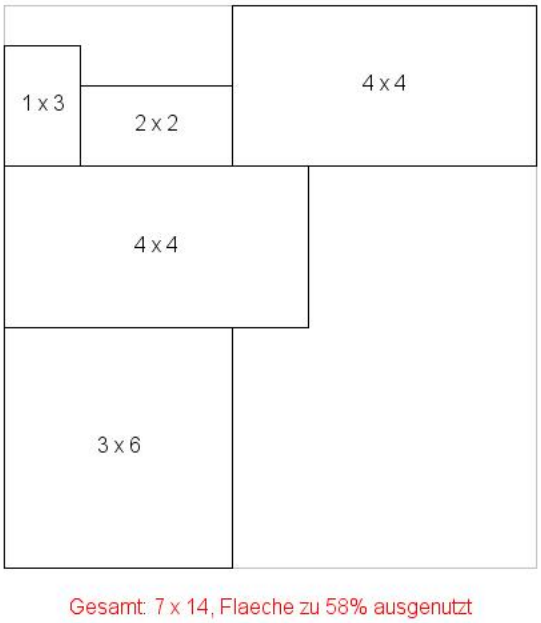
4 Beispiele

Die Abmessungen der Rechtecke aus den Beispielen werden in der Ausgabe angegeben, weshalb sich die Eingabe aus der Ausgabe ergibt. Der Lesbarkeit halber werde ich für die folgenden Beispiele die Eingaben nicht angeben.
Beispiele der BwInf-Website:

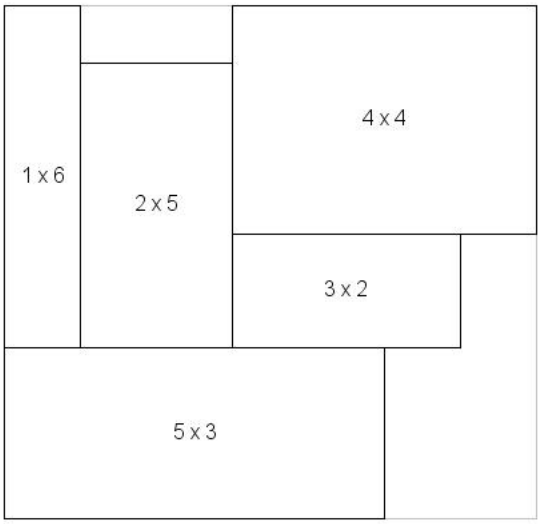
Beispiel 1:



Beispiel 2:

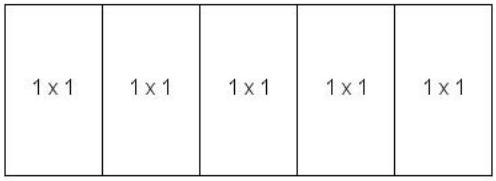


Beispiel 3:



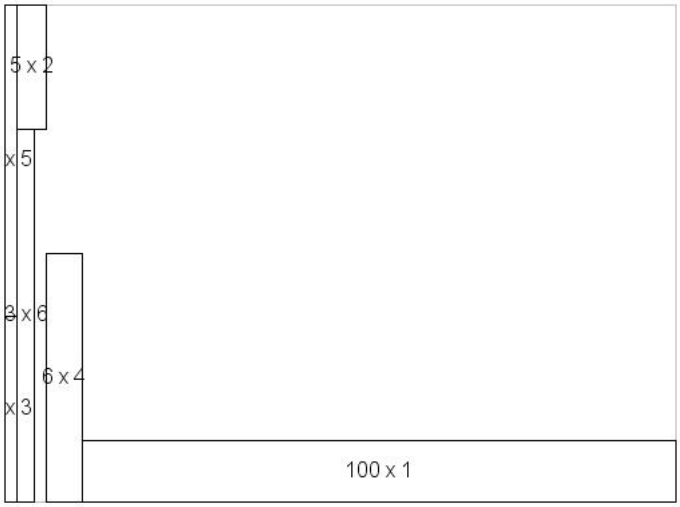
Gesamt: 7 x 9, Flaeche zu 84% ausgenutzt

Auf diesen Beispielen hat das Verfahren relativ effiziente Anordnungen berechnet.



Gesamt: 5 x 1, Flaeche zu 100% ausgenutzt

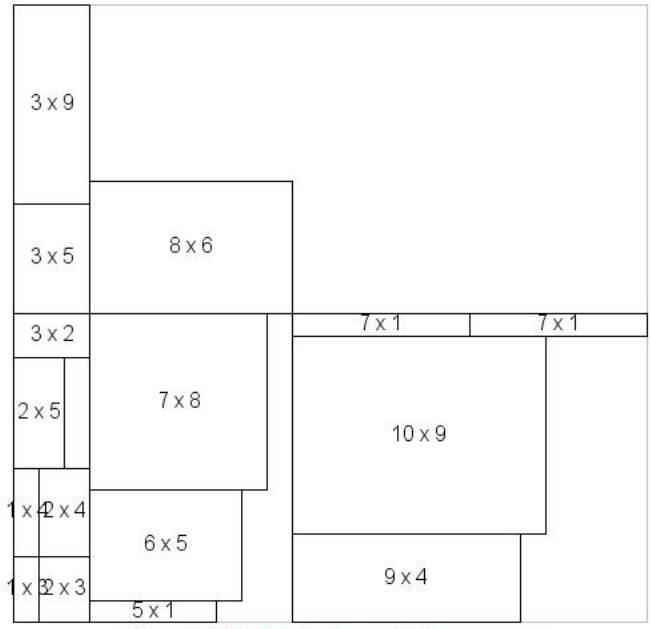
Es ist aber auch zum Erreichen einer prefekten fähig.



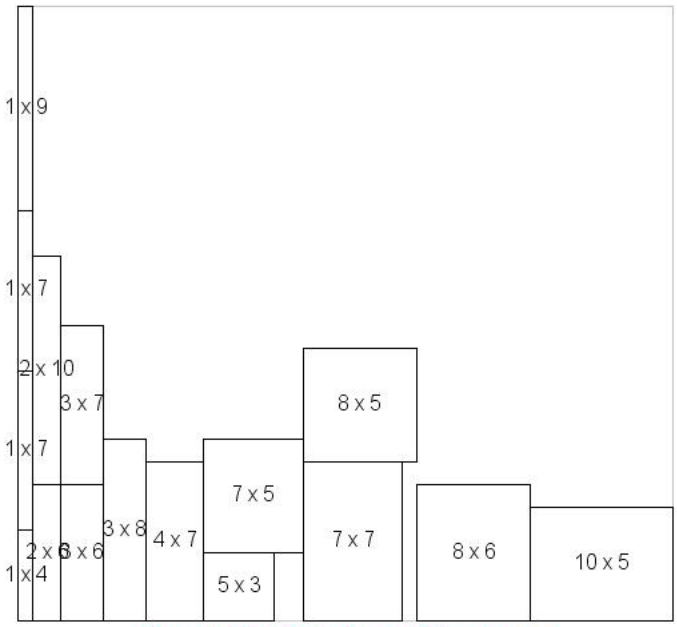
Gesamt: 113 x 8, Flaeche zu 18% ausgenutzt

Wie gut das Verfahren funktioniert, hängt auch von der gesamten Ähnlichkeit der Eingaberechtecke ab. Im obrigen Fall waren diese sehr verschieden, weshalb eine schlechte Packung erzielt wurde.

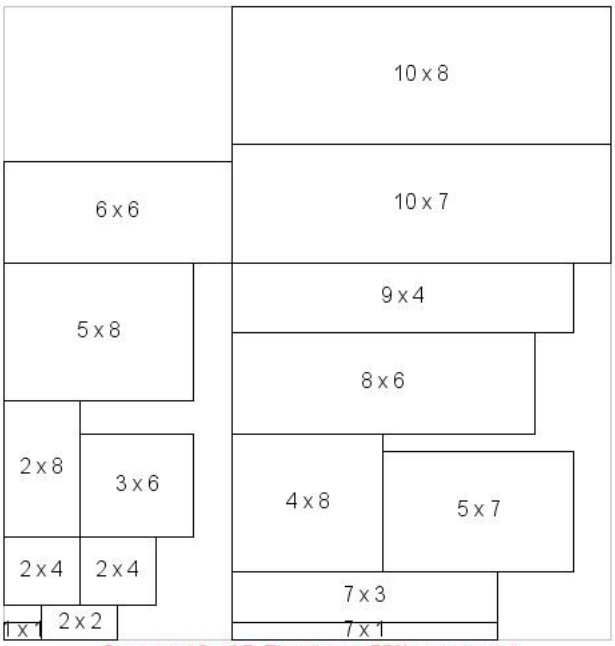
So reagiert das Verfahren auf zufällige Eingabewerte:



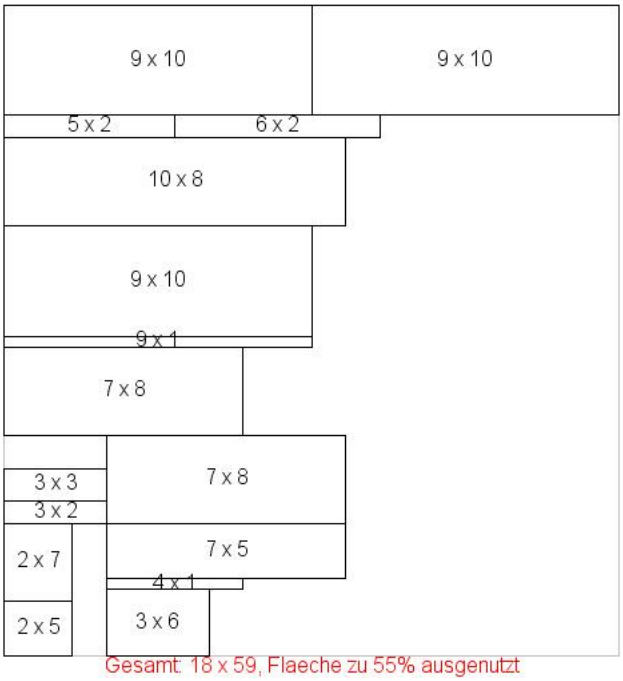
Gesamt: 25 x 28, Flaeche zu 51% ausgenutzt



Gesamt: 46 x 27, Flaeche zu 31% ausgenutzt



Gesamt: 16 x 37, Flaeche zu 77% ausgenutzt



Das Verfahren ist ein guter subquadratischer Näherungsalgorithmus für das Problem.

5 Quellcode

```
1
/*****************************************************************************
3  *  Kompilierung:  javac Schrebergaerten.java StdDraw.java
   *  Ausfuehrung:   java  Schrebergaerten < input.txt
5  *
   *  Loesung zu Aufgabe 4: "Schrebergaerten" der ersten Runde
7  *  des 37. Bundeswettbewerbs Informatik
   *
9  *  https://bwinf.de/bundeswettbewerb/biber-2018/1-runde/
   *
11 *  Eingabe:
   *
13 *      1. Zeile: Integer n fuer die Anzahl der Schrebergaerten
   *
15 *      folgende n Zeilen:
   *        jeweils Laenge und Breite des Schrebergartens, mit Leerzeichen getrennt
17 *
   *  @author  Erik Klein
19 *  @version 25.11.18
   *
21 *****************************************************************************/

23 import java.awt.Color;
   import java.util.Arrays;
25 import java.util.Scanner;

27 public class Schrebergaerten
   {
29
     public static void main(String[] args)
31   {
       int sum = 0;
33
       // Einlesen der Anzahl der Schrebergaerten
35     Scanner sc = new Scanner(System.in);
       int n = sc.nextInt();
37
       // Einlesen der Schrebergaerten in Array
39     Rechteck[] schrebergaerten = new Rechteck[n];
```

```
    for (int i = 0; i < n; i++)
    {
      int y = sc.nextInt(); // Breite
      int x = sc.nextInt(); // Laenge
      sum += x * y;
      schrebergaerten[i] = new Rechteck(x, y, null, null);
    }
    sc.close();

    // Berechnen der Anordnung
    Rechteck loesung = pack(schrebergaerten, n);

    // Zeichnen der Anordnung

    int x = loesung.x;
    int y = loesung.y;

    StdDraw.setXscale(-1, x + 1);
    StdDraw.setYscale(-2, y + 1);

    StdDraw.setPenColor(Color.RED);
    StdDraw.text(x / 2.0, -1.0, ("Gesamt:␣" + x + "␣x␣" + y + ",␣Flaeche␣zu␣" + Math.round(100 * sum /

    StdDraw.setPenColor(Color.LIGHT_GRAY);
    StdDraw.rectangle(x / 2.0, y / 2.0, x / 2.0, y / 2.0);

    StdDraw.setPenColor();
    draw(loesung, 0, 0);
  }

  // Methode zum rekursiven Berechnen der Anordnung
  public static Rechteck pack(Rechteck[] schrebergaerten, int hi)
  {
    // wenn nur noch ein Rechteck vorhanden, dieses zurueckliefern
    if (hi == 1)
      return schrebergaerten[0];

    // Sortieren der Rechtecke
    Arrays.sort(schrebergaerten, 0, hi);

    // immer zwei Rechtecke zu einem neuen zusammenfuegen
    for (int pos = 0; pos < hi / 2; pos++)
      schrebergaerten[pos] = zusammenfuegen(schrebergaerten[2 * pos], schrebergaerten[2 * pos + 1]);

    // Falls Anzahl der Rechtecke ungerade, bleibt in Rechteck uebrig
    if (hi % 2 == 1)
    {
      schrebergaerten[hi / 2] = schrebergaerten[hi - 1];
      hi++;
    }

    // Prozedur fuer errechnete Rechtecke wiederholen
    return pack(schrebergaerten, hi / 2);
  }

  // Methode zum Zusammenfuegen zweier Rechtecke
  public static Rechteck zusammenfuegen(Rechteck r1, Rechteck r2)
  {
    // Wenn Flaecheninhalt des resultierenden Rechtecks beim Uebereinanderlegen
    // kleiner
    // als beim Nebeneinenderlegen
    if ((r1.x + r2.x) * Math.max(r1.y, r2.y) > Math.max(r1.x, r2.x) * (r1.y + r2.y))
    {
      // uebereinender legen
      r2.posy += r1.y;
      return new Rechteck(Math.max(r1.x, r2.x), r1.y + r2.y, r1, r2);
      // sonst
    } else
    {
      // nebeneinander legen
      r2.posx += r1.x;
      return new Rechteck(r1.x + r2.x, Math.max(r1.y, r2.y), r1, r2);
    }
```

```
113   }
115   // Methode zum Zeichnen der Anordnung
      public static void draw(Rechteck r, int posx, int posy)
117   {
        // Wenn Rechteck nicht aus zwei kleineren Rechtecken zusammengesetzt
119     if (r.comp1 == null)
        {
121       // Rechteck zeichnen
          StdDraw.rectangle(posx + r.posx + r.x / 2.0, posy + r.posy + r.y / 2.0, r.x / 2.0, r.y / 2.0);
123
          // Groesse dazu schreiben
125       StdDraw.text(posx + r.posx + r.x / 2.0, posy + r.posy + r.y / 2.0, (r.x + "␣x␣" + r.y));
        } else
127     {
          // sonst Prozess fuer beide kleinere Rechtecke wiederholen
129       draw(r.comp1, posx + r.posx, posy + r.posy);
          draw(r.comp2, posx + r.posx, posy + r.posy);
131     }
      }
133
      // Klasse zum Repraesentieren von Schrebergaerten
135   public static class Rechteck implements Comparable<Rechteck>
      {
137     // Laenge, Breite, Position x, Position y
        private int x, y, posx, posy;
139
        // Rechtecke, aus denen das Rechteck zusammengesetzt wurde
141     private Rechteck comp1, comp2;

143     public Rechteck(int x, int y, Rechteck comp1, Rechteck comp2)
        {
145       this.x = x;
          this.y = y;
147       this.posx = 0;
          this.posy = 0;
149       this.comp1 = comp1;
          this.comp2 = comp2;
151     }

153     // Methode zum Vergleichen zweier Rechtecke, noetig fuer das Sortieren
        @Override
155     public int compareTo(Rechteck that)
        {
157       // Mit erster Prioritaet die Laenge, danach die Breite betrachten
          if (this.x > that.x)
159         return +1;
          if (this.x < that.x)
161         return -1;
          if (this.y > that.y)
163         return +1;
          if (this.y < that.y)
165         return -1;
          return 0;
167     }

169   }
    }
```

Schrebergaerten.java