```java
package view;

import model.Land;

import java.io.IOException;
import java.nio.file.Path;

public interface IO {

    /**
     *
     * @param f Die Datei, in welche geschrieben werden soll
     * @param model - The solution
     */
    void writeSolutionToFile(Path f, Land model) throws IOException;

    Land readFromFile(Path f) throws IOException;
    String getDescription();
    String getDescriptionNameOnly();
}

```

```java
package view;

import exceptions.InputFormatException;
import model.Land;
import model.Punkt;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Collections;
import java.util.List;
import java.util.function.Predicate;
import java.util.regex.Pattern;

/**
 * Die normale Schnittstelle für Ein- und Ausgaben, wie durch IHK gefordert
 */
public class DefaultInOut implements IO {

    public final char VALUE_SEP = ' ';
    public final char COMMENT = ';';
    public final char DESCRIPTION = '*';
    private String description;
    private String inputFileName;

    public String getDescription() {
        if (description != null) {
            String stars = "*".repeat(25);
            return "%s\n%s\n%s".formatted(stars, description, stars);
        } else {
            return null;
        }
    }

    public String getDescriptionNameOnly() {
        return description;
    }

    public void print(Land model) {
        String description = getDescription() != null ? getDescription() + "\n" : "";
        System.out.println(description + model.toString());
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public void writeSolutionToFile(Path relative, Land model) throws IOException {
        String description = getDescription() != null ? getDescription() + "\n" : "";
        String result = description + model.toString();
```

```
52          Path directory = relative.resolve(inputFileName);
53          if (!Files.exists(directory)) {
54              Files.createDirectory(directory);
55          }
56          Path path = directory.resolve("%s.txt".formatted(inputFileName));
57          Files.write(path, Collections.singleton(result));
58      }
59
60      /**
61       * @param relative Der Pfad in dem die Dateien erstellt werden
62       * @param model    Das Datenmodell mit Lösung (Antenna Positions)
63       * @throws IOException falls das Schreiben der Datei verhindert wurde
64       */
65      public void generateGnuPlotFiles(Path relative, Land model) throws
    IOException {
66          String descriptionName = "Unbekannt";
67          if (getDescriptionNameOnly() != null) {
68              descriptionName = getDescriptionNameOnly();
69          }
70          List<Punkt> antenna = model.getAntenna();
71          float[][] data = model.getFeld();
72          Path directory = relative.resolve(inputFileName); // stripDescName ist der
    Dateiname
73          if (!Files.exists(directory)) {
74              Files.createDirectory(directory);
75          }
76          writeDatFile(directory, data);
77          writeAntennaFiles(directory, antenna);
78          writeDemFile(directory, antenna, descriptionName);
79      }
80
81      private void writeAntennaFiles(Path relative, List<Punkt> antenna) throws
    IOException {
82          StringBuilder builder = new StringBuilder();
83          for (int i = 0; i < antenna.size(); i++) {
84              Punkt p = antenna.get(i);
85              builder.setLength(0);
86              builder.append("%.0f %.0f %d\n".formatted(p.x, p.y, (int) p.z));
87              builder.append("%.0f %.0f %.1f".formatted(p.x, p.y, p.z));
88              Path path = relative.resolve("%sAntenne%d.dat".formatted(inputFileName,
    i + 1));
89              Files.write(path, Collections.singleton(builder.toString()));
90          }
91      }
92
93      private void writeDatFile(Path relative, float[][] data) throws IOException {
94          StringBuilder builder = new StringBuilder();
95          for (int i = 0; i < data.length; i++) {
96              for (int j = 0; j < data[i].length; j++) {
97                  builder.append("%d %d %.0f\n".formatted(i, j, data[i][j]));
98              }
```

```
 99            builder.append("\n");
100        }
101        Path path = Paths.get(relative.toString() + "\\%s.dat".formatted(inputFileName
       ));
102        Files.write(path, Collections.singleton(builder.toString()));
103    }
104
105    private void writeDemFile(Path relative, List<Punkt> antenna, String
       descriptionName) throws IOException {
106        StringBuilder builder = new StringBuilder();
107        builder.append("# %s\n".formatted(descriptionName));
108        builder.append("set title \"%s\"\n".formatted(descriptionName));
109        builder.append("set hidden3d\n");
110        builder.append("set key outside top\n");
111        builder.append("splot '%s.dat' w line , \\\n".formatted(inputFileName));
112        for (int i = 0; i < antenna.size() - 1; i++) {
113            builder.append("'%sAntenne%d.dat' w linesp , \\\n".formatted(
       inputFileName, i + 1));
114        }
115        builder.append("'%sAntenne%d.dat' w linesp\n".formatted(inputFileName,
       antenna.size()));
116        builder.append("pause -1 \"Hit return to continue\"");
117        Path path = Paths.get(relative.toString() + "\\%s.dem".formatted(
       inputFileName));
118        Files.write(path, Collections.singleton(builder.toString()));
119    }
120
121    @Override
122    public Land readFromFile(Path file) throws IOException {
123        inputFileName = file.getFileName().toString().split(".txt")[0];
124        List<String> lines = Files.readAllLines(file);
125        if (lines.size() <= 3) {
126            throw new InputFormatException("Die Datei muss mindestens 3 Zeilen
       enthalten");
127        }
128        description = readDescriptionIfExists(lines);
129        float[][] feld = readSizeAndHeights(lines);
130        return new Land(feld);
131    }
132
133    /**
134     * Liest von allen Zeilen, ohne Kommentare, die Werte in ein float[][]-Feld ein
135     *
136     * @param lines - Die Zeilen der Datei
137     * @return Eingabe umgewandelt in
138     */
139    private float[][] readSizeAndHeights(final List<String> lines) {
140        float[][] field = null;
141        List<String> noComments = lines.stream().filter(s -> !s.startsWith(";")).toList();
142        boolean sizeFlag = true;
143        int countValueLines = 0;
```

```
144
145         //Dieses Pattern schlägt an, wenn negative oder dezimale Werte in einer Zeile
    auftauchen
146         Predicate<String> predicate = Pattern.compile("^.*(-[0-9]+[,.]?[0-9]*)|([0-9
    ]+[,.][0-9]*)$").asPredicate();
147         boolean negativeOrDecimals = noComments.stream().anyMatch(predicate);
148         if (negativeOrDecimals) {
149             throw new InputFormatException("Negative oder dezimale Werte sind
    nicht erlaubt");
150         }
151         for (int n = 0; n < noComments.size(); n++) {
152             String line = noComments.get(n);
153             if (line.isBlank()) {
154                 boolean hasMoreLinesWithContent = noComments.subList(n,
    noComments.size()).stream().anyMatch(s -> !s.isEmpty());
155                 if (hasMoreLinesWithContent) {
156                     throw new InputFormatException("Leere Zeilen dürfen nur am Ende
    der Datei stehen");
157                 } else {
158                     break;
159                 }
160             }
161             if (sizeFlag) {
162                 sizeFlag = false;
163                 String[] gebiet = line.split("" + VALUE_SEP);
164                 if (gebiet.length != 2) {
165                     throw new InputFormatException("Die Angabe des Gebiets muss 2
    Einträge haben");
166                 }
167                 int spalte = Integer.parseInt(gebiet[0]);
168                 int zeile = Integer.parseInt(gebiet[1]);
169                 if (zeile <= 1 || spalte <= 1) {
170                     throw new InputFormatException("Das Gebiet muss positiv und
    größer 0 definiert werden");
171                 }
172                 field = new float[zeile][spalte];
173             } else {
174                 predicate = Pattern.compile("^.*[7-9]+|[0-9]{2,}.*$").asPredicate();
175                 if (predicate.test(line)) {
176                     throw new InputFormatException("Höhenwerte");
177                 }
178                 String[] row = line.split("" + VALUE_SEP);
179                 if (row.length > field[0].length) {
180                     throw new InputFormatException("Zeile %s enthält zu viele Einträge
    ".formatted(lines.indexOf(line) + 1));
181                 }
182                 float recent = -1;
183                 for (int i = 0; i < row.length; i++) {
184                     try {
185                         String entry = row[i];
186                         recent = Float.parseFloat(entry);
```

```
187                     field[countValueLines][i] = recent;
188                 } catch (NumberFormatException e) {
189                     throw new InputFormatException("Ungültiger Wert in Zeile [%s]".
    formatted(line));
190                 }
191             }
192             if (row.length < field[0].length) {
193                 for (int distance = field[0].length - row.length; distance > 0; distance--) {
194                     field[countValueLines][field[0].length - distance] = recent;
195                 }
196             }
197             countValueLines++;
198         }
199     }
200     if (countValueLines < field.length) {
201         throw new InputFormatException("Nicht genügend Höhenwerte
    angegeben. Offset %d Zeilen".formatted(field.length - countValueLines));
202     }
203     return field;
204 }
205
206 /**
207  * Liest die ersten Zeilen und befüllt <code>description</code> falls eine existiert.
208  * Sonst: <code>description</code> ist null
209  *
210  * @param lines - eingelesene Zeilen des Programms
211  * @return description
212  */
213 private String readDescriptionIfExists(final List<String> lines) {
214     boolean descFlag = false;
215     String desc = null;
216     List<String> onlyComments = lines.stream().filter(s -> s.startsWith("" +
    COMMENT)).toList();
217     for (String s : onlyComments) {
218         if (s.charAt(1) == DESCRIPTION) {
219             descFlag = !descFlag;
220             continue;
221         }
222         if (descFlag) {
223             desc = s.replace("" + COMMENT, "").trim();
224             break;
225         }
226     }
227     return desc;
228 }
229
230 }
231
```

```java
package model;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class Land {

    public final float[][] feld;
    public final int rows;
    public final int columns;
    private final List<Punkt> antenna;
    private final Set<Ebene> plains;
    private List<Punkt> points;

    public Land(float[][] feld) {
        this.feld = feld;
        this.antenna = new ArrayList<>();
        this.plains = new HashSet<>();
        initializePoints();
        initializePlains();
        rows = feld.length;
        columns = feld[0].length;
    }

    /**
     * Copy-Konstruktor um ein unabhängiges Land zu erzeugen.
     * Biegt Referenzen
     *
     * @param model von dem die Kopie erstellt werden soll
     */
    public Land(Land model) {
        float[][] temp = new float[model.feld.length][model.feld[0].length];
        for (int i = 0; i < model.feld.length; i++) {
            System.arraycopy(model.feld[i], 0, temp[i], 0, model.feld[0].length);
        }
        this.feld = temp;
        this.antenna = new ArrayList<>();
        for (Punkt antenna : model.antenna) {
            this.antenna.add(new Punkt(antenna));
        }
        this.plains = Set.copyOf(model.plains);
        this.points = new ArrayList<>();
        for (Punkt point : model.points) {
            this.points.add(new Punkt(point));
        }
        rows = feld.length;
        columns = feld[0].length;
    }

```

```
52      public Set<Ebene> getPlains() {
53          return plains;
54      }
55
56      private void initializePoints() {
57          points = new ArrayList<>();
58          for (int i = 0; i < feld.length; i++) {
59              for (int j = 0; j < feld[0].length; j++) {
60                  Punkt here = new Punkt(i, j, feld[i][j]);
61                  points.add(here);
62              }
63          }
64      }
65
66      public List<Punkt> getAntenna() {
67          return antenna;
68      }
69
70      public float[][] getFeld() {
71          return feld;
72      }
73
74      /**
75       * Fügt eine Antenne hinzu
76       *
77       * @param punkt Punkt
78       * @return false, wenn sie bereits existiert
79       */
80      public boolean addAntenna(Punkt punkt) {
81          if (!antenna.contains(punkt)) {
82              antenna.add(punkt);
83              return true;
84          }
85          return false;
86      }
87
88      /**
89       * Sucht einen Punkt über x und y Koordinate
90       */
91      public Punkt getByXY(float x, float y) {
92          for (Punkt p : points) {
93              if (Float.compare(x, p.x) == 0 && Float.compare(y, p.y) == 0) {
94                  return p;
95              }
96          }
97          return null;
98      }
99
100     /**
101      * Erstellt für jede innere Spalte und Zeile jeweils eine Ebene
102      */
```

```
103     private void initializePlains() {
104         for (int row = 1; row < feld.length - 1; row++) {
105             Punkt a = new Punkt(row, 0, feld[row][0]);
106             Punkt b = new Punkt(row, 1, feld[row][1]);
107             Ebene e = new Ebene(a, b);
108             plains.add(e);
109         }
110         for (int column = 1; column < feld[0].length - 1; column++) {
111             Punkt a = new Punkt(0, column, feld[0][column]);
112             Punkt b = new Punkt(1, column, feld[1][column]);
113             Ebene e = new Ebene(a, b);
114             plains.add(e);
115         }
116     }
117 
118     /**
119      * Gibt eine Liste an Punkten zurück, die weder die Antennen noch gesehene
    Punkte enthält
120      *
121      * @return relevante Punkte
122      */
123     public ArrayList<Punkt> getRelevantPoints() {
124         ArrayList<Punkt> relevant = new ArrayList<>();
125         for (Punkt point : points) {
126             if (!point.seen && !antenna.contains(point)) {
127                 relevant.add(point);
128             }
129         }
130         return relevant;
131     }
132 
133     /**
134      * Gibt die Anzahl an gesehenen Punkten zurück
135      *
136      * @return Anzahl
137      */
138     public int countSeenPoints() {
139         return (int) points.stream().filter(p -> p.seen).count();
140     }
141 
142     public String toString() {
143         StringBuilder builder = new StringBuilder();
144         builder.append("Ausdehnung in X: %d\n".formatted(columns))
145                 .append("Ausdehnung in Y: %d\n".formatted(rows))
146                 .append("Eingelesene Höhen\n");
147         for (float[] row : feld) {
148             for (float val : row) {
149                 builder.append("%.0f ".formatted(val));
150             }
151             builder.append("\n");
152         }
```

```
153         builder.append("Benötigte Antennen: %d\n".formatted(antenna.size()));
154         for (int i = 0; i < antenna.size(); i++) {
155             Punkt ant = antenna.get(i);
156             builder.append("Antenne %d: %.0f %.0f\n".formatted(i + 1, ant.y, ant.x));
157         }
158         return builder.toString();
159     }
160 }
161
```

```java
package model;

public class Ebene {
    public final Punkt aufpunkt;
    public final Punkt r1;
    public final Punkt r2;
    public final Punkt normVektor;
    public final float rechteSeite;

    /**
     * Erstellt eine Ebene mit {@code a, b} und {@code a} mit a.z = 0.
     * Initialisiert einen Normalenvektor und rechte Seite
     *
     * @param a Aufpunkt der Ebene
     * @param b Zweiter Punkt
     */
    public Ebene(Punkt a, Punkt b) {
        this.aufpunkt = a;
        this.r1 = a.vectorized(b);
        this.r2 = a.vectorized(new Punkt(a.x, a.y, 0));
        normVektor = r1.cartProd(r2);
        rechteSeite = normVektor.x * a.x + normVektor.y * a.y + normVektor.z * a.z;
    }

    public Ebene(Punkt a, Punkt b, Punkt c) {
        this.aufpunkt = a;
        this.r1 = a.vectorized(b);
        this.r2 = a.vectorized(c);
        normVektor = r1.cartProd(r2);
        rechteSeite = normVektor.x * a.x + normVektor.y * a.y + normVektor.z * a.z;
    }


    /**
     * Ermittelt den Schnittpunkt mit einer Geraden
     *
     * @param gerade Gerade
     * @return Schnittpunkt, falls existiert, sonst null
     */
    public Punkt getSchnittpunkt(Gerade gerade) {
        if (isParallel(gerade)) return null;
        float ganzzahl = rechteSeite - gerade.aufpunkt.x * normVektor.x
                - gerade.aufpunkt.y * normVektor.y
                - gerade.aufpunkt.z * normVektor.z;
        float r = gerade.richtung.x * normVektor.x
                + gerade.richtung.y * normVektor.y
                + gerade.richtung.z * normVektor.z;
        r = ganzzahl / r;
        float xKoord = gerade.aufpunkt.x + r * gerade.richtung.x;
        float yKoord = gerade.aufpunkt.y + r * gerade.richtung.y;
        float zKoord = gerade.aufpunkt.z + r * gerade.richtung.z;
```

```
52          return new Punkt(xKoord, yKoord, zKoord);
53      }
54
55      /**
56       * Ermittelt, ob eine Gerade parallel zur Ebene liegt
57       *
58       * @param gerade Gerade
59       * @return true/false
60       */
61      public boolean isParallel(Gerade gerade) {
62          float skalar = normVektor.dot(gerade.richtung.normalized());
63          return Float.compare(skalar, 0) == 0;
64      }
65  }
66
```

File - C:\DEV\GroProSIM\Code\GroProCode\src\main\java\model\Punkt.java

```java
package model;

import java.util.Objects;

/**
 * Kann für Punkte und Vektoren verwendet werden
 */
public class Punkt implements Cloneable {
    public final float x;
    public final float y;
    public final float z;
    /**
     * <code>true</code> wenn Punkt von einer Antenne gesehen wird
     */
    public boolean seen;

    public Punkt(float x, float y, float z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    public Punkt(float x, float y, float z, boolean seen) {
        this.x = x;
        this.y = y;
        this.z = z;
        this.seen = seen;
    }

    public Punkt(Punkt p) {
        this(p.x, p.y, p.z, p.seen);
    }

    /**
     * Berechnet das Skalarprodukt zweier Vektoren
     *
     * @param p der andere Vektor
     */
    public float dot(Punkt p) {
        return x * p.x + y * p.y + z * p.z;
    }

    /**
     * Berechnet das kartesische Produkt (Kreuzprodukt) zweier Vektoren
     */
    public Punkt cartProd(Punkt p) {
        float a = y * p.z - z * p.y;
        float b = z * p.x - p.z * x;
        float c = x * p.y - y * p.x;
        return new Punkt(a, b, c);
    }
```

```
52
53      /**
54       * Erstellt einen Richtungsvektor aus this und {@code b}
55       *
56       * @param b anderer Vektor
57       */
58      public Punkt vectorized(Punkt b) {
59          return new Punkt(b.x - x, b.y - y, b.z - z);
60      }
61
62      /**
63       * Addiert die Koordinaten von this und {@code p}
64       *
65       * @param p anderer Punkt/Vektor
66       * @return neuer Punkt
67       */
68      public Punkt add(Punkt p) {
69          return new Punkt(this.x + p.x, this.y + p.y, this.z + p.z);
70      }
71
72      /**
73       * Berechnet einen normierten Vektor von {@code this}
74       *
75       * @return neuen Vektor normiert
76       */
77      public Punkt normalized() {
78          float norm = norm();
79          float newX = 1 / norm * x;
80          float newY = 1 / norm * y;
81          float newZ = 1 / norm * z;
82          return new Punkt(newX, newY, newZ);
83      }
84
85      /**
86       * Berechnet die Norm eines Vektors mit allen Koordinaten
87       *
88       * @return norm
89       */
90      public float norm() {
91          return (float) Math.sqrt(x * x + y * y + z * z);
92      }
93
94      /**
95       * Berechnet die Norm eines Vektors ohne Berücksichtigen der Z-Koordinate
96       *
97       * @return norm
98       */
99      public float xyNorm() {
100         return (float) Math.sqrt(x * x + y * y);
101     }
102
```

```
103     /**
104      * Vergleicht zwei Punkte
105      */
106     @Override
107     public boolean equals(Object o) {
108         if (this == o) return true;
109         if (!(o instanceof Punkt punkt)) return false;
110         return Math.abs(punkt.x - x) < 0.00001 && Math.abs(punkt.y - y) < 0.00001;
111     }
112 
113     /**
114      * Schaut, ob der Punkt auf dem Feld liegt und positiv in seiner Höhe ist
115      *
116      * @param rows    Anzahl
117      * @param columns Anzahl
118      * @return true/false
119      */
120     public boolean isLegal(int rows, int columns) {
121         if (x >= 0 && x <= rows) {
122             if (y >= 0 && y <= columns) {
123                 return z >= 0;
124             }
125         }
126         return false;
127     }
128 
129     @Override
130     public int hashCode() {
131         return Objects.hash(x, y);
132     }
133 
134     @Override
135     public String toString() {
136         return "{%.2f, %.2f, %.2f} seen=%s".formatted(x, y, z, seen);
137     }
138 
139     @Override
140     public Object clone() {
141         Punkt clone;
142         try {
143             clone = (Punkt) super.clone();
144         } catch (CloneNotSupportedException e) {
145             clone = new Punkt(x, y, z, seen);
146         }
147         return clone;
148     }
149 }
150 
```

```java
package model;

public class Gerade {

    public final Punkt aufpunkt;
    public final Punkt richtung;

    /**
     * Erstellt eine Gerade aus zwei Punkten
     *
     * @param a Aufpunkt
     * @param b zweiter Punkt
     */
    public Gerade(Punkt a, Punkt b) {
        this.aufpunkt = a;
        this.richtung = a.vectorized(b);
    }

    /**
     * Skaliert den Richtungsvektor mit factor und gibt den Punkt an der Stelle zurück
     *
     * @param factor Faktor
     * @return Punkt auf der Geraden
     */
    public Punkt scale(float factor) {
        float x, y, z;
        x = aufpunkt.x + factor * richtung.x;
        y = aufpunkt.y + factor * richtung.y;
        z = aufpunkt.z + factor * richtung.z;
        return new Punkt(x, y, z);
    }

    /**
     * Ermittelt, ob {@code p} oberhalb oder unterhalb der Geraden liegt
     *
     * @param p Punkt
     * @return true wenn oberhalb
     */
    public boolean isPointAbove(Punkt p) {
        float r;
        if (Float.compare(richtung.x, 0) == 0) {
            r = (p.y - aufpunkt.y) / richtung.y;
        } else {
            r = (p.x - aufpunkt.x) / richtung.x;
        }
        float z = scale(r).z;
        return z <= p.z;
    }

    /**
     * Ermittelt, ob ein Schnittpunkt {@code sp} im relevanten Bereich der Geraden ist
```

```
52      * (zwischen Antenne und untersuchtem Punkt
53      *
54      * @param sp Schnittpunkt mit Ebene
55      * @return true, wenn es ein Schnittpunkt im relevanten Bereich der Geraden ist
56      */
57     public boolean isImportantSchnittpunkt(Punkt sp) {
58         float r = probe(sp);
59         Punkt vec = aufpunkt.vectorized(sp);
60         Punkt end = aufpunkt.add(richtung);
61         return r > 0 && richtung.xyNorm() >= vec.xyNorm() && !sp.equals(aufpunkt
    ) && !sp.equals(end);
62     }
63
64     /**
65      * Gibt den Skalierungsfaktor zurück
66      *
67      * @param p Punkt
68      * @return Faktor oder -99 wenn nicht auf Geraden
69      */
70     private float probe(Punkt p) {
71         float r1 = (p.x - aufpunkt.x) / richtung.x;
72         float r2 = (p.y - aufpunkt.y) / richtung.y;
73
74         return r1 == r2 ? r1 : -99;
75     }
76 }
77
```

```java
package controller;

import model.Land;

public interface Algorithm {

    /**
     * Löst das Rätsel nach eigenem Ansatz
     * @param model enthält die eingelesenen Daten
     * @return Land mit Liste der Position der Antennen
     */
    Land solve(final Land model);
}

```

```java
package controller;

import exceptions.InputFormatException;
import model.Land;
import view.DefaultInOut;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class Controller {

    public static void main(String[] args) throws IOException {
        if (args.length != 1) {
            throw new InputFormatException("Es muss ein Dateiname oder ein relativer Dateipfad angegeben werden");
        }
        String fileName = args[0];
        DefaultInOut inOut = new DefaultInOut();
        Path p = Paths.get(fileName);
        Land l;
        try {
            l = inOut.readFromFile(p);
        } catch (FileNotFoundException e) {
            throw new InputFormatException("Pfad %s konnte nicht gefunden werden ".formatted(p.toAbsolutePath()));
        }
        BruteMeinenForce algo = new BruteMeinenForce();
        long msStart = System.currentTimeMillis();
        Land solution = algo.solve(l);
        long msEnd = System.currentTimeMillis();
        System.out.println(msEnd - msStart);

        inOut.print(solution);
        //Für jar out, sonst target
        Path out = Paths.get("Testbeispiele/out");
        if(!Files.exists(out)) {
            Files.createDirectory(out);
        }
        inOut.generateGnuPlotFiles(out, solution);
        inOut.writeSolutionToFile(out, solution);
    }
}

```

```java
package controller;

import model.Ebene;
import model.Gerade;
import model.Land;
import model.Punkt;

import java.util.ArrayList;

public class BruteMeinenForce implements Algorithm {

    /**
     * Das Land enthält eine bisher ungetestete Antenne und wird für die Überprüfung genutzt
     */
    private Land pseudo;

    /**
     * Löst das Rätsel mit einer BruteForce-Implementierung
     *
     * @param model enthält die eingelesenen Daten
     * @return Lösung des Problems mit Antennen
     */
    @Override
    public Land solve(Land model) {
        Land returnLand = new Land(model);
        Land lastRoundsMax = new Land(returnLand);
        while (returnLand.countSeenPoints() != returnLand.rows * returnLand.columns) {
            //Teste für neue Antenne jede mögliche Position
            for (int i = 0; i < model.getFeld().length; i++) {
                for (int j = 0; j < model.getFeld()[i].length; j++) {
                    Punkt antennaHere = new Punkt(i, j, model.getFeld()[i][j] + 0.1f);
                    pseudo = new Land(returnLand);
                    if (pseudo.addAntenna(antennaHere)) {
                        pseudo.getByXY(antennaHere.x, antennaHere.y).seen = true;
                    } else {
                        continue;
                    }
                    ArrayList<Punkt> relevant = pseudo.getRelevantPoints();
                    for (Punkt punkt : relevant) {
                        if (punkt.seen) continue;
                        Gerade g = new Gerade(antennaHere, punkt);
                        if (Float.compare(g.richtung.xyNorm(), (float) Math.sqrt(2)) <= 0) {
                            punkt.seen = true;
                        } else {
                            //Wenn Punkt horizontal zu Antenne
                            if (Float.compare(g.richtung.x, 0) == 0) {
                                horizontalPoint(punkt, g);
                                //Wenn Punkt vertikal zu Antenne
                            } else if (Float.compare(g.richtung.y, 0) == 0) {
```

```
50                          verticalPoint(punkt, g);
51                      } else if (Math.abs(g.richtung.x) - Math.abs(g.richtung.y) == 0) {
52                          diagonalPoint(punkt, g);
53                      } else {
54                          unalignedPoint(punkt, g);
55                      }
56                  }
57              }
58              //Setze das
59              if (pseudo.countSeenPoints() > lastRoundsMax.countSeenPoints()) {
60                  lastRoundsMax = pseudo;
61              }
62          }
63      }
64      if (lastRoundsMax.countSeenPoints() > returnLand.countSeenPoints()) {
65          returnLand = lastRoundsMax;
66      }
67  }
68  //      checkForRedundancy();
69      return returnLand;
70  }
71
72  //  private void checkForRedundancy() {
73  //
74  //  }
75
76  /**
77   * Behandelt einen Punkt der horizontal zur Antenne liegt und setzt ihn auf true,
    falls er gesehen wird
78   *
79   * @param punkt der zu überprüfende Punkt
80   * @param g     die Gerade von Antenne zu {@code punkt}
81   */
82  public void horizontalPoint(Punkt punkt, Gerade g) {
83      boolean noDisturber = true;
84      for (int k = 1; k < Math.abs((int) g.richtung.y); k++) {
85          int schritt;
86          schritt = g.richtung.y > 0 ? k : -k;
87          Punkt zwischenPunkt = new Punkt(punkt.x,
88                  punkt.y - schritt,
89                  pseudo.feld[(int) (punkt.x)][(int) (punkt.y - schritt)]);
90          if (g.isPointAbove(zwischenPunkt)) {
91              noDisturber = false;
92              break;
93          }
94      }
95      if (noDisturber) {
96          punkt.seen = true;
97      }
98  }
99
```

```
100     /**
101      * Behandelt einen Punkt der Vertikal zur Antenne liegt und setzt ihn auf true, falls
        er gesehen wird
102      *
103      * @param punkt der zu überprüfende Punkt
104      * @param g     die Gerade von Antenne zu {@code punkt}
105      */
106     public void verticalPoint(Punkt punkt, Gerade g) {
107         boolean noDisturber = true;
108         for (int k = 1; k < Math.abs((int) g.richtung.x); k++) {
109             int schritt;
110             schritt = g.richtung.x > 0 ? k : -k;
111             Punkt zwischenPunkt = new Punkt(punkt.x - schritt,
112                     punkt.y,
113                     pseudo.feld[(int) (punkt.x - schritt)][(int) (punkt.y)]);
114             if (g.isPointAbove(zwischenPunkt)) {
115                 noDisturber = false;
116                 break;
117             }
118         }
119         if (noDisturber) {
120             punkt.seen = true;
121         }
122     }
123
124     /**
125      * Behandelt einen Punkt der diagonal zur Antenne liegt und setzt ihn auf true,
        falls er gesehen wird
126      *
127      * @param punkt der zu überprüfende Punkt
128      * @param g     die Gerade von Antenne zu {@code punkt}
129      */
130     public void diagonalPoint(Punkt punkt, Gerade g) {
131         boolean noDisturber = true;
132         for (int k = 1; k < Math.abs((int) g.richtung.x); k++) {
133             int schrittX = g.richtung.x > 0 ? k : -k;
134             int schrittY = g.richtung.y > 0 ? k : -k;
135             Punkt zwischenPunkt = new Punkt(punkt.x - schrittX,
136                     punkt.y - schrittY,
137                     pseudo.feld[(int) (punkt.x - schrittX)][(int) (punkt.y - schrittY)]);
138             if (g.isPointAbove(zwischenPunkt)) {
139                 noDisturber = false;
140                 break;
141             }
142         }
143         if (noDisturber) {
144             punkt.seen = true;
145         }
146     }
147
148     /**
```

```
149     * Behandelt einen windschiefen Punkt und setzt ihn auf true, falls er gesehen
    wird
150     *
151     * @param punkt der zu überprüfende Punkt
152     * @param g     die Gerade von Antenne zu {@code punkt}
153     */
154    public void unalignedPoint(Punkt punkt, Gerade g) {
155        boolean collision = false;
156        for (Ebene plain : pseudo.getPlains()) {
157            Punkt schnitt = plain.getSchnittpunkt(g);
158            if (schnitt != null && schnitt.isLegal(pseudo.rows, pseudo.columns) && g.
    isImportantSchnittpunkt(schnitt)) {
159                if (schnitt.x == (int) schnitt.x) {
160                    float y1 = (float) Math.floor(schnitt.y);
161                    float y2 = (float) Math.ceil(schnitt.y);
162                    Punkt p1 = new Punkt(schnitt.x,
163                            y1,
164                            pseudo.feld[(int) schnitt.x][(int) y1]);
165                    Punkt p2 = new Punkt(schnitt.x,
166                            y2,
167                            pseudo.feld[(int) schnitt.x][(int) y2]);
168                    Gerade topEdge = new Gerade(p1, p2);
169                    if (!topEdge.isPointAbove(schnitt)) {
170                        collision = true;
171                        break;
172                    }
173                } else {
174                    float x1 = (float) Math.floor(schnitt.x);
175                    float x2 = (float) Math.ceil(schnitt.x);
176                    Punkt p1 = new Punkt(x1,
177                            schnitt.y,
178                            pseudo.feld[(int) x1][(int) schnitt.y]);
179                    Punkt p2 = new Punkt(x2,
180                            schnitt.y,
181                            pseudo.feld[(int) x2][(int) schnitt.y]);
182                    Gerade topEdge = new Gerade(p1, p2);
183                    if (!topEdge.isPointAbove(schnitt)) {
184                        collision = true;
185                        break;
186                    }
187                }
188            }
189        }
190        if (!collision) {
191            punkt.seen = true;
192        }
193    }
194 }
195
```

File - C:\DEV\GroProSIM\Code\GroProCode\src\main\java\exceptions\InputFormatException.java

```java
package exceptions;

public class InputFormatException extends RuntimeException {
    public InputFormatException(String msg) {
        super(msg);
    }
    public InputFormatException() {
        super();
    }
}

```