37. Bundeswettbewerb Informatik

1. Runde

01.09.2018 - 26.11.2018

Aufgabe 3 - Voll Daneben

Nico Jeske (48312)

Team: ITler (00140)

Ich versichere hiermit, die vorliegende Arbeit selbstständig und unter den Wettbewerbsregeln des Bundeswettbewerbs Informatik, ohne fremde Hilfe und ohne die Verwendung anderer, als der in den Quellen angegebenen Hilfsmitteln, angefertigt zu haben.

Nico Jeske, den 12.10.2018

1 Lösungsidee

1.1 Problembeschreibung

An dem Gewinnspiel von Al Capone Junior nehmen n Personen teil. Jeder Teilnehmer zahlt hierbei einen Einsatz von 25€. Der gesamte Einsatz E beläuft sich bei n Personen also auf $E = n * 25$ Euro. Nun wählt jeder Teilnehmer eine zufällige Zahl x für die gilt: $1 \leq x \leq 1000$. Diese gewählten Zahlen der Teilnehmer benenne ich im folgenden als die Menge Z_T Nun wählt Al 10 Zahlen $y_1, ..., y_{10}$ aus für die gilt: $1 \leq y \leq 1000$. Diese Zahlen benenne ich im folgenden als Menge Z_A. Nun wird für jede von den Teilnehmern gewählte Zahl $x \in Z_T$ die Zahl $y_k \in Z_A$ mit dem geringsten Abstand $d_x = |x - y_k|$ gewählt. Der Auszuzahlende Betrag A berechnet sich wie folgt:

$$A = \sum_{i=0}^{n} d_{x_i} \tag{1}$$

Der gesamte Gewinn G – oder auch Verlust – für Al berechnet sich nun wie folgt:

$$G = E - A \tag{2}$$

Das Ziel ist es nun also, y_1 *bis* y_{10} so zu bestimmen, dass G maximal ist. Dafür betrachte ich im folgenden mehrere Lösungsansätze.

1.2 Gleichmäßige Verteilung

Betrachen wir beispielsweise die gewählten Zahlen aus der zweiten Beispielaufgabe und stellen diese auf einem Zahlenstrahl dar:

Figure 1: zweite Beispieldatei als Zahlenstrahl

Um nun die Werte für y_1 *bis* y_{10} zu wählen wird dieser Zahlenstrahl in 10 Abschnitte unterteilt. Dabei wird der Zahlenstrahl nicht in zehn genau gleich große Teile gespalten. Stattdessen werden die 10 Abschnitte so bestimmt, dass jeder Abschnitt die gleiche Menge an ausgewählten Zahlen enthält. Die benötigte Menge m pro Abschnitt lässt sich mit $m = \frac{n}{10}$ bestimmen. n stellt dabei die Anzahl der gewählten Nummern dar. Im falle eines Restes werden die letzten Abschnitte dementsprechend angepasst, so dass diese auch mehr oder weniger Zahlen als die restlichen Abschnitte haben können.

Zeichnet man diese Abschnitte nun in den oben Zahlenstrahl ein, so kommt man auf folgendes Ergebnis:

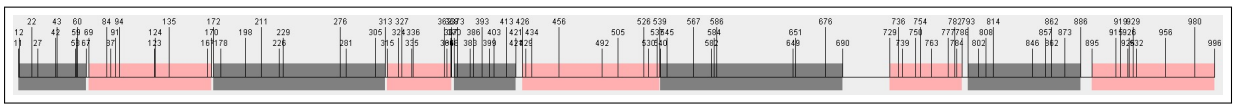

Figure 2: eingezeichnete Abschnitte

Jeder Abschnitt kann nun also als Menge der eingeschlossenen Zahlen $x_i \in A_i$ betrachtet werden. Nun soll für jeden dieser Abschnitte eine Zahl z gefunden werden, die die aufkommenden Auszahlungen von Al minimiert. Die benötigte Auszahlung $G_{auszahlung}$ in diesem Abschnitt lässt sich hierbei durch die Summe aus der Differenz jeder Zahl x_i zu z bestimmen. Es ergibt sich also folgende Formel:

$$\sum_{k=0}^{|A_i|} |x_k - z| = G_{auszahlung} \tag{3}$$

Somit lässt sich für jeden einzelnen Abschnitt eine Zahl für Al finden, mit der die Auszahlung für die Teilnehmer in diesem Abschnitt minimal ist. Allerdings führt diese Methode nicht immer zum optimalen Ergebnis, ist allerdings sehr nahe dran.

Um nun ein besseres Ergebnis zu finden werden mehrere Schritte durch geführt. Im ersten Schritt werden nun erstmal die gewählten Zahlen durch die Teilnehmerzahl des Abschnittes mit dem geringsten Abstand ersetzt. Wurden nun schließlich für jeden Abschnitt eine Zahl bestimmt, werden noch ein paar nahegelegene Möglichkeiten ausprobiert. Dafür werden alle Kombinationen aus der letzten und der nächsten Teilnehmerzahl für jede Teilnehmerzahl ausprobiert.

Betrachten wir beispielsweise folgende Teilnehmerzahlen: [3, 5, 7, 10, 13, 17, 20, 22] und wählen 5, 10 und 20 als Al's Zahlen, so würde jede Kombination folgender Listen ausprobiert werden: [[3, 5, 7], [7, 10, 13], [17, 20, 22]]. Dabei gilt dann schließlich die beste einer dieser Kombinationen als unser Endergebnis.

Diese Methode ist natürlich nicht optimal, reicht aber aus, um ein nahezu optimales Ergebnis zu erreichen.

2 Implementierung

Die Aufgabe wurde in Java implementiert. Die Zahlen der Teilnehmer werden mit einem BufferedReader eingelesen und in einer List¡Integer¿ gespeichert. Diese wird schließlich sortiert und es wird gleichzeitig eine Version der Teilnehmerzahlen ohne Duplikate generiert. Diese Liste ohne Duplikate wird später dann für das finden der nächstkleineren oder größeren Teilnehmerzahl verwendet.

Nachdem nun also die Teilnehmerzahlen eingelesen wurden, werden sie mithilfe der Funktion *Solver.orderdSplit(numbers, parts)* in 10 gleichgroße Abschnitte unterteilt. Dabei wird die ursprüngliche Reihenfolge beibehalten.

Um nun die erste grobe Lösung zu finden, wird für jeden Abschnitt nach der in der Lösungsidee genannten Formel eine Zahl gesucht, für die die Auszahlung in diesem Bereich minimal wären. Statt dieser Zahl wird dann allerdings die Teilnehmerzahl die dieser am nähesten ist als vorzeitige Lösung gespeichert.

Nun, wo eine vorzeitige Lösung gefunden wurde, werden, wie in der Lösungsidee beschrieben, ein paar Kombinationen um die gewählten Zahlen herum getestet, um ein besseres Ergebnis zu erreichen.

Die einzigen Teillisten wie z.B. [[3, 5, 7], [7, 10, 13], [17, 20, 22]] aus der Lösungsidee werden hierbei von den gewählten Alzahlen mit der Funktion *Solver.getPossibleVariances* generiert. Die einzelnen Möglichkeiten aus diesen Listen werden schließlich mithilfe des Cartesischen Produktes generiert. Verwendet wird hierbei die Implementation aus der Guava-Bibliothek von Google.

Diese Varianten werden nun nach und nach mit der Helferfunktion *Solver.calcExpenses(Teilnehmerzahlen, Alzahlen)* danach bewertet, wie viel Al auszahlen muss. Das finale Ergebnis ist schließlich die Kombination, die die geringste benötigte Auszahlung aufweist.

3 Beispiele

Datei	Al's Zahlen	Gewinn
beispiel1.txt	[50, 145, 245, 345, 445, 545, 645, 745, 845, 945]	25€
beispiel2.txt	[42, 91, 211, 335, 393, 505, 584, 763, 857, 926]	198€
beispiel3.txt	[100, 240, 340, 400, 460, 520, 620, 720, 820, 920]	240€

4 Quellcode

Listing 1: Klasse Solver

```
1  package jeske;
2
3  import com.esotericsoftware.minlog.Log;
4  import com.google.common.collect.Lists;
5  import jeske.GUI.Draw;
6
7  import java.util.ArrayList;
8  import java.util.Collections;
9  import java.util.HashSet;
10 import java.util.List;
11 import java.util.concurrent.atomic.AtomicInteger;
12
13
14 /**
15  * Solver for the task
16  */
17 public class Solver {
18   private List<Integer> numbers;
19   private Draw draw;
```

```
20
21  /**
22   * Creates a new Instance of the Solver
23   *
24   * @param numbers chosen numbers
25   * @param draw    GUI for rendering
26   */
27  Solver(List<Integer> numbers, Draw draw) {
28    this.numbers = numbers;
29    this.draw = draw;
30  }
31
32  /**
33   * Splits an list into n mostly equal parts
34   *
35   * @param list list to split
36   * @param n    number of result lists
37   * @param <T>  type
38   * @return list, split into n parts
39   * @throws NullPointerException     input list null
40   * @throws IllegalArgumentException n <= 0 OR n < list.size
41   */
42  private static <T> List<List<T>> orderedSplit(List<T> list, int n)
43  throws NullPointerException, IllegalArgumentException {
44    if (list == null) {
45      throw new NullPointerException("list is null.");
46    }
47
48    if (n <= 0) {
49      throw new IllegalArgumentException("division with 0");
50    }
51
52    if (list.size() < n) {
53      throw new IllegalArgumentException("less elements than asked parts.");
54    }
55
56    List<List<T>> result = new ArrayList<>(n);
57
58    int listsSize = list.size();
59    int remainder = list.size();
60
61    int index = 0;
62    int remainderAccess = 0;
63    int from = index * listsSize + remainderAccess;
64    int to = (index + 1) * listsSize + remainderAccess;
65
66    while (n > index) {
67
68      if (remainder != 0) {
69        result.add(list.subList(from, to + 1));
70        remainder--;
71        remainderAccess++;
72      } else {
73        result.add(list.subList(from, to));
74      }
75
76      index++;
77      from = index * listsSize + remainderAccess;
78      to = (index + 1) * listsSize + remainderAccess;
79    }
80
81    return result;
82  }
83
84  /**
85   * Solves the task
86   */
87  void solve() {
88    //Sort the numbers
89    Collections.sort(numbers);
90
91    //Removing duplicate numbers
92    List<Integer> numbersWithoutDuplicates = new ArrayList<>(new HashSet<>(numbers));
93    //Sort numbers in ascending order
94    Collections.sort(numbersWithoutDuplicates);
95
96    //The entire stake
97    int paidMoney = 25 * numbers.size();
98
99    //Column the numbers into ten parts
100   List<List<Integer>> list = orderedSplit(numbers, 10);
101
102   //Finding a first solution. For each of the ten parts,
103   //an number z is searched for which the needed payment is minimal.
104   //The nearest number to z is then saved for the solution.
105   List<Integer> aiList = new ArrayList<>();
106   for (List<Integer> lis : list) {
```

```
107       int small = lis.get(0);
108       int height = lis.get(lis.size() - 1);
109
110       int min = 999999;
111       int minZ = -1;
112
113       for (int z = small; z <= height; z++) {
114         int sum = 0;
115         for (int i = 0; i < lis.size() - 1; i++) {
116           int currNumber = lis.get(i);
117           sum += Math.abs(currNumber - z);
118         }
119
120         if (sum <= min) {
121           min = sum;
122           minZ = z;
123           //System.out.printf("Min: %s Z: %s \n", min, z);
124         }
125       }
126
127       aiList.add(nearestElement(lis, minZ));
128     }
129
130     draw.setAreas(list);
131     draw.setOldSoulution(aiList);
132     draw.repaint();
133
134     Log.info("First Solution: " + aiList + " -> " + (paidMoney - calcExpenses(numbers, aiList)) + " gain.");
135
136
137     //For each AI number, take the AI number itself, as well as the number before and after it.
138     //e.g. from aiList [5,10,20] -> e.g. [[3,5,7], [7,10,13], [17,20]]
139     List<List<Integer>> variances = getPossibleVariances(numbersWithoutDuplicates, aiList);
140     List<List<Integer>> allPossibilitys = Lists.cartesianProduct(variances);
141
142     //Test all possible solutions and choose the best one.
143     int lowest = Integer.MAX_VALUE;
144     List<Integer> best = null;
145     for (List<Integer> possibility : allPossibilitys) {
146       int calced = calcExpenses(numbers, possibility);
147       if (calced < paidMoney) {
148         if (calced < lowest) {
149           lowest = calced;
150           best = possibility;
151         }
152       }
153     }
154
155     draw.setSolution(best);
156     draw.repaint();
157     Log.info("Final solution: " + best + " with " + (paidMoney - lowest) + " gain");
158   }
159
160   /**
161    * For each number in the aiList, take the AI number itself, as well as the number before and after it.
162    * e.g. from aiList [5,10,20] -> e.g. [[3,5,7], [7,10,13], [17,20]]
163    *
164    * @param numbersWithoutDuplicates chosen numbers
165    * @param aiList                   ai numbers
166    * @return list of possible variances
167    */
168   private List<List<Integer>> getPossibleVariances(List<Integer> numbersWithoutDuplicates, List<Integer> aiList) {
169     List<List<Integer>> variances = new ArrayList<>();
170     for (Integer anAiList : aiList) {
171       List<Integer> currList = new ArrayList<>();
172       int ai = anAiList;
173       int index = numbersWithoutDuplicates.indexOf(ai);
174
175       if (index >= 1) {
176         int beforeNumber = numbersWithoutDuplicates.get(index - 1);
177         currList.add(beforeNumber);
178       }
179
180       currList.add(ai);
181
182       if (index < numbersWithoutDuplicates.size() - 1) {
183         int afterNumber = numbersWithoutDuplicates.get(index + 1);
184         currList.add(afterNumber);
185       }
186
187       variances.add(currList);
188     }
189     return variances;
190   }
191
192   /**
193    * Finds the element in the list that is closest to the given number.
```

```
194     *
195     * @param numbers list of numbers
196     * @param avg     number
197     * @return closest element in the list
198     */
199    static int nearestElement(List<Integer> numbers, double avg) {
200      if (numbers.size() == 1)
201        return numbers.get(0);
202
203      for (int i = 0; i < numbers.size() - 1; i++) {
204        int aiNumber = numbers.get(i);
205        int nextNumber = numbers.get(i + 1);
206
207        if (aiNumber == Math.round(avg))
208          return aiNumber;
209
210        if (nextNumber == Math.round(avg))
211          return nextNumber;
212
213        if (aiNumber <= avg && avg < nextNumber) {
214          double distDown = Math.abs(avg - aiNumber);
215          double distUp = Math.abs(avg - nextNumber);
216
217          if (distDown == distUp)
218            return nextNumber;
219
220          if (Math.min(distDown, distUp) == distDown) {
221            return aiNumber;
222          } else {
223            return nextNumber;
224          }
225        }
226
227      }
228
229      //The code should not reach this point
230      throw new UnsupportedOperationException();
231    }
232
233    /**
234     * Calculates the average of all numbers in the list.
235     *
236     * @param numbers list
237     * @return average
238     */
239    static double avg(List<Integer> numbers) {
240      double sum = 0;
241      double numberOfElements = numbers.size();
242      for (int number : numbers) {
243        sum += number;
244      }
245      return sum / numberOfElements;
246    }
247
248    /**
249     * For a given AI selection, the cost of this will be calculated
250     *
251     * @param numbers   numbers
252     * @param aiNumbers ai numbers
253     * @return cost
254     */
255    public static int calcExpenses(List<Integer> numbers, List<Integer> aiNumbers) {
256      AtomicInteger payments = new AtomicInteger(0);
257
258      for (int number : numbers) {
259
260        for (int i = 0; i < aiNumbers.size() - 1; i++) {
261          int aiNumber = aiNumbers.get(i);
262          int nextNumber = aiNumbers.get(i + 1);
263
264          if (aiNumber <= number && number < nextNumber) {
265            int distDown = Math.abs(number - aiNumber);
266            int distUp = Math.abs(number - nextNumber);
267            payments.addAndGet(Math.min(distDown, distUp));
268            break;
269          }
270          //choosen > highest AI number
271          else if (number > aiNumbers.get(aiNumbers.size() - 1)) {
272            payments.addAndGet(Math.abs(number - aiNumbers.get(aiNumbers.size() - 1)));
273            break;
274          } else if (number < aiNumbers.get(0)) {
275            payments.addAndGet(aiNumbers.get(0) - number);
276            break;
277          }
278        }
279      }
280      return payments.get();
```

```
281     }
282 }
```

5 Quellen

- Guava: https://github.com/google/guava
- minlog: https://github.com/EsotericSoftware/minlog