33. Bundeswettbewerb Informatik

Zahlenspiel (Junioraufgabe 2)

Johannes Heinrich, Marian Dietz

Lösungsidee

Die Bedingungen für die Aufgaben zum Kürzen von Brüchen schränken die möglichen Brüche, welche generiert werden können, stark ein. Aus diesem Grund können, ohne dass es hier eine große Verzögerung gibt, zunächst einmal alle bereits gekürzten möglichen Brüche generiert werden. Sollten also beispielsweise für den Schwierigkeitsgrad *leicht* Brüche benötigt werden, werden zunächst alle möglichen Brüche in der Form $\frac{p}{q}$ generiert. Hierbei muss selbstverständlich die Bedingung $p + q \leq 10$ eingehalten werden. Außerdem muss $p \neq q$, denn $a \neq b$ und $\frac{a}{b} = \frac{p}{q}$. Da a und b voneinander verschieden sind, müssen demnach auch p und q voneinander verschieden sein. Die Liste, welche alle möglichen gekürzten Brüche enthält, stellen wir als mathematische Menge $K = \{\frac{1}{2}, \frac{1}{3}, ..., \frac{8}{1}, \frac{9}{1}\}$ dar. In diesem Fall beinhaltet K alle gekürzten Brüche der Schwierigkeitsstufe *leicht*. Der Bruch $\frac{8}{2}$ hält zwar alle Bedingungen ein, kann aber noch weiter gekürzt werden, und ist deshalb nicht in der Liste enthalten. Dies trifft auch auf einige weitere Brüche zu.

Sobald dies getan ist, muss die gewünschte Anzahl von Aufgaben mithilfe von K generiert werden. Dazu wird zunächst für jede zu generierende Aufgabe ein zufälliger Bruch aus K ausgewählt. Für diesen Bruch werden wiederum alle Zahlen ermittelt, welche jeweils mit p und q multipliziert werden können. Eine solche Zahl nennen wir im Folgenden Z. Daraus entsteht dann der Bruch $\frac{a}{b}$, welcher auf den Bruch $\frac{p}{q}$ gekürzt werden kann. Z kann jedoch nur eingesetzt werden, wenn der daraus entstehende Bruch $\frac{a}{b}$ auch die Länge hat, welche für die Schwierigkeitsstufe benötigt wird. Außerdem muss $Z \geq 2$, da der Bruch ansonsten nicht tatsächlich gekürzt werden könnte (denn es gilt: $q < b$ und $Z \in \mathbb{Z}$).

Nachdem für $\frac{p}{q}$ alle validen Zahlen Z zur Multiplikation berechnet wurden, kann aus dieser Liste bzw. Menge zufällig ein Z ausgewählt werden, welches verwendet wird. Nehmen wir an, es wurde aus der Liste mit allen gekürzten Brüchen für die Schwierigkeitsstufe *leicht* der Bruch $\frac{1}{6}$ zufällig ausgewählt. Mögliche Zahlen Z wären demnach $\{10, 11, 12, 13, 14, 15, 16\}$. Aus dieser Menge wird zufällig die Zahl 11 ausgewählt. Der ungekürzte Bruch lautet dann $\frac{11}{66}$.

Es kann der Fall eintreten, dass so viele Brüche gewünscht sind, sodass in der Menge K nicht genügend Brüche enthalten sind. In diesem Fall werden bei unserer Lösung einfach alle bereits verwendeten Brüche „erneut verwendet“, wobei aber in diesem Fall eine neue Zahl Z verwendet werden muss. Bereits vorher genutzte Zahlen Z sind nicht erneut verwendbar. Dadurch könnten, sollte die Anzahl von gewünschten Brüchen groß genug sein, theoretisch also im oben genannten

Beispiel höchstens sieben Aufgaben mit $\frac{1}{6}$ als Startbruch generiert werden, da es sieben valide Zahlen als Z geben kann.

Umsetzung

Das Programm wurde objektorientiert in der Programmiersprache Java geschrieben. Ausgeführt werden kann das Programm in einer Konsole / einem Terminal mit dem Befehl „java -jar [Pfad zur .jar-Datei]“. Dabei sind folgende Klassen enthalten:

- `Stage` — Schwierigkeitsstufe
- `Fracture` — Bruch, besteht aus `x` (Zähler) und `y` (Nenner).
- `ReduceFractureExercise` — Aufgabe zum Kürzen von Brüchen, besteht aus zwei `Fracture`s: `normal` (ungekürzter Bruch) und `reduced` (gekürzter Bruch).
- `FractureGenerator` — Generiert `ReduceFractureExercise`s anhand der angegebenen `Stage` und der angegebenen Anzahl an Aufgaben (`int`).
- `Zahlenspiel` — Hauptklasse, liest Schwierigkeitsstufe und Anzahl an Aufgaben ein und lässt `FractureGenerator` die Aufgaben generieren.
- `NoFracturesFoundException` — `Exception`, welche verwendet wird, wenn kein Bruch generiert werden konnte.

Der wichtigste Teil, also das Generieren der Brüche, geschieht in der Klasse `FractureGenerator`:

- Mögliche gekürzte Brüche werden in der Methode `generatePossibleReducedFractures()` erstellt.
- Danach werden sämtliche Aufgaben in der Methode `generateExercises()` generiert. Sollte der Fall eintreten, dass bereits alle möglichen Aufgaben verwendet wurden, wird hier schon vorher abgebrochen.
- `generateExercises()` ruft wiederum x-mal (wobei x die Anzahl der zu generierenden Aufgaben ist) die Methode `generateExercise()` auf, welche schließlich eine Aufgabe generiert.
- Um zu überprüfen, ob ein Bruch gekürzt werden kann, wird die Methode `canBeShortened()` der Klasse `Fracture` verwendet. Diese durchläuft alle Zahlen von 2 bis zur kleineren der beiden Zahlen `x` und `y` und überprüft, ob `x` und `y` durch eine dieser Zahlen teilbar ist. Ist dies der Fall, kann der Bruch gekürzt werden. Dadurch wird der größte gemeinsame Teiler ermittelt. Diese Methode wird in `generatePossibleReducedFractures()` verwendet, um nur Brüche, die nicht gekürzt werden können, der Liste K hinzuzufügen.
- Um einen größeren Bruch eines bereits gekürzten Bruches zu generieren, hat die Klasse `Fracture` eine weitere Methode `createBiggerFracture()`. Bereits für die Multiplikation genutzte Zahlen Z werden in der Liste `usedMultiplications` gespeichert und sind später nicht mehr nutzbar.

Beispiele

Beispiel 1

```
Stufe (leicht / mittel / schwer) eingeben:
schwer
Anzahl an Aufgaben (≥ 1) eingeben:
3
90/140 = 9/14
99/209 = 9/19
112/49 = 16/7
```

In diesem Beispiel werden 3 Aufgaben in der Schwierigkeitsstufe *schwer* erstellt. Für alle generierten Aufgaben stimmen die Bedingungen für die angegebene Stufe:

- Länge von $\frac{a}{b}$: 5
- $20 < p + q \leq 30$
- $q < b$
- $a \neq b$

Beispiel 2

```
Stufe (leicht / mittel / schwer) eingeben:
leicht
Anzahl an Aufgaben (≥ 1) eingeben:
3
50/30 = 5/3
63/27 = 7/3
26/91 = 2/7
```

Hier werden wieder 3 Aufgaben erstellt, diesmal in der Schwierigkeitsstufe *leicht*. Die dafür geänderten Bedingungen sind:

- Länge von $\frac{a}{b}$: 4
- $p + q \leq 10$

Diese werden alle eingehalten, genauso wie die oben genannten allgemeinen Bedingungen $q < b$ und $a \neq b$.

Beispiel 3

```
Stufe (leicht / mittel / schwer) eingeben:
mittel
Anzahl an Aufgaben (≥ 1) eingeben:
100000
378/27 = 14/1
24/432 = 1/18
238/85 = 14/5
68/578 = 2/17
```

Bei diesem Beispiel tritt ein Sonderfall ein: die angegebene Anzahl an Aufgaben (100000) ist viel zu hoch, deshalb wird nach dem 2314. Durchlauf abgebrochen. Alle möglichen Aufgaben, die zu dem Schwierigkeitsgrad *mittel* existieren können, sind schon generiert worden. Das bedeutet, dass

nur 2314 Aufgaben existieren, die der Schwierigkeitsstufe *mittel* zugehören. Zum Vergleich: für Stufe *schwer* gibt es 2180 verschiedene Aufgaben, für *leicht* lediglich 446.

Quellcode

Stage

```
public class Stage {

    private int forEnd,
            minSum, maxSum,
            fractureLength;

    /**
     * @param stage der Schwierigkeitsgrad
     * @throws java.lang.IllegalArgumentException wenn {@code stage} nicht 1,
     * 2 oder 3 beträgt
     */
    public Stage(byte stage) {
        switch (stage) {
            case 1:
                forEnd = 9;
                minSum = 0;
                maxSum = 10;
                fractureLength = 4;
                break;
            case 2:
                forEnd = 19;
                minSum = 11;
                maxSum = 20;
                fractureLength = 5;
                break;
            case 3:
                forEnd = 29;
                minSum = 21;
                maxSum = 30;
                fractureLength = 5;
                break;
            default:
                throw new IllegalArgumentException("stage must be 1, 2 or 3");
        }
    }

    public int getForEnd() {
        return forEnd;
    }

    public int getMinSum() {
        return minSum;
    }

    public int getMaxSum() {
        return maxSum;
    }

    public int getFractureLength() {
        return fractureLength;
    }

}
```

NoFracturesFoundException

```
public class NoFracturesFoundException extends Exception {

}
```

ReduceFractureExercise

```java
public class ReduceFractureExercise {

    /**
     * normal ist der ungekürzte Bruch, reduced der gekürzte.
     */
    private Fracture normal, reduced;

    public ReduceFractureExercise(Fracture normal, Fracture reduced) {
        this.normal = normal;
        this.reduced = reduced;
    }

    @Override
    public String toString() {
        return normal + " = " + reduced;
    }
}
```

Fracture

```java
/**
 * Diese Klasse repräsentiert einen Bruch, welcher aus x (Zähler) und y (Nenner)
 * besteht.
 */
public class Fracture {

    private int x, y;
    private List<Integer> usedMultiplications;

    /**
     * @param x der Zähler
     * @param y der Nenner
     */
    public Fracture(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public String toString() {
        return x + "/" + y;
    }

    // ...

}
```

Fracture.canBeShortened()

```java
/**
 * Diese Methode prüft, ob der Bruch aus den Zahlen x und y gekürzt werden
 * kann. Ein Bruch kann gekürzt werden, wenn beide Zahlen von mindestens einer Zahl
 * von 2 bis zu der kleineren Zahl teilbar ist. Die eins ist nicht in diesem
 * Bereich, da jede ganze Zahl durch eins teilbar ist.
 *
 * @return {@code true} wenn der Bruch gekürzt werden kann,
 * {@code false} wenn nicht
 */
public boolean canBeShortened() {
    int min = Math.min(x, y);

    // Bereich: 2 bis zur kleineren der beiden Zahlen (Math.min())
    for (int i = 2; i <= min; i++) {
        if (x % i == 0 && y % i == 0) // beide müssen durch i teilbar sein
            return true;
    }

    return false;
}
```

Fracture.createBiggerFracture()

```java
/**
 * Diese Methode erstellt einen neuen Bruch, welcher eine ungegürzte 'Variante'
 * dieses Bruches ist. Hierbei wird immer, wenn ein Bruch generiert wurde, die
 * Zahl, mit der multipliziert wurde, in die Liste {@link #usedMultiplications}
 * eingefügt. Alle Zahlen dieser Liste können dann künftig nicht mehr in dieser
 * Methode verwendet werden, um neue Brüche generieren zu lassen. Sollte es
 * demnach keine Zahl mehr geben, mit der multipliziert werden kann, kommt eine
 * NoFracturesFoundException zum Einsatz.
 *
 * @param length die 'Länge' des neuen Bruches
 * @param random eine Instanz der Klasse {@link java.util.Random},
 *               welche für zufällige Brüche verwendet werden soll.
 * @return der neue, ungekürzte Bruch
 * @throws NoFracturesFoundException wenn kein ungekürzter Bruch generiert
 *                                   werden konnte.
 */
public Fracture createBiggerFracture(int length, Random random) throws
        NoFracturesFoundException {

    if (usedMultiplications == null) // lazy initialization
        usedMultiplications = new ArrayList<Integer>();

    // Mögliche Zahlen, mit denen der ungekürzte Bruch multipliziert werden kann:
    List<Integer> possibleMultiplications = new ArrayList<Integer>();

    for (int i = 2; true; i++) { /* muss mindestens 2 sein, da '1 * x = x',
                                    und damit nicht sinnvoll ist */

        // Wenn bereits genutzt, darf die Zahl nicht erneut verwendet werden:
        if (usedMultiplications.contains(i))
            continue;

        // Länge des ungekürzten Bruches, also nach der  Multiplizierung mit i:
        int realLength = Integer.toString(x * i).length() +
                Integer.toString(y * i).length();

        if (realLength == length) // muss mit der angegebenen Länge übereinstimmen
            possibleMultiplications.add(i);

        // wenn realLength größer als length ist, kann diese nur noch steigen,
        // da i immer größer wird; demnach kann es also keine Zahl mehr geben,
        // mit der noch multipliziert werden kann
        else if (realLength > length)
            break;
    }

    // keine geeignete Zahl zum Multiplizieren gefunden:
    if (possibleMultiplications.size() == 0)
        throw new NoFracturesFoundException();
    else {
        int multiplication = possibleMultiplications
                .get(random.nextInt(possibleMultiplications.size()));

        // Wird nun verwendet; kann demnach nicht mehr genutzt werden:
        usedMultiplications.add(multiplication);

        return new Fracture(x * multiplication, y * multiplication);
    }
}
```

FractureGenerator

```java
public class FractureGenerator {

    private Stage stage;
    private int number;

    /**
     * Erstellt eine neue Instanz dieser Klasse.
     *
     * @param stage  die Stufe, für die die Brüche generiert werden soll
     * @param number die Anzahl an Brüchen, die generiert werden sollen
     */
    public FractureGenerator(Stage stage, int number) {
        this.stage = stage;
        this.number = number;
    }

    // ...

}
```

FractureGenerator.generate()

```java
/**
 * Generiert ein Set mit ReduceFractureExercises. Dazu werden der
 * Schwierigkeitsgrad und die Anzahl an Brüchen verwendet.
 *
 * @return die generierten Aufgaben
 */
public Set<ReduceFractureExercise> generate() {
    List<Fracture> possibleFractures = generatePossibleReducedFractures();
    return generateExercises(possibleFractures);
}
```

FractureGenerator.generatePossibleReducedFractures()

```java
/**
 * Generiert alle möglichen gekürzten Brüche für den zuvor definierten
 * Schwierigkeitsgrad.
 *
 * @return eine Liste mit allen möglichen passenden Brüchen
 */
private List<Fracture> generatePossibleReducedFractures() {
    List<Fracture> fractures = new ArrayList<Fracture>();

    for (int x = 1; x < stage.getForEnd(); x++) {
        for (int y = 1; y < stage.getForEnd(); y++) {
            // Bedingung a ≠ b und Einschränkung der Summe von p/q:
            if (x != y && x + y >= stage.getMinSum()
                    && x + y <= stage.getMaxSum()) {

                Fracture fracture = new Fracture(x, y);

                // p/q muss immer voll gekürzt sein:
                if (!fracture.canBeShortened())
                    fractures.add(fracture);
            }
        }
    }

    return fractures;
}
```

FractureGenerator.generateExercises()

```
/**
 * Generiert ein Set mit ReduceFractureExercises. Dazu wird die angegebene Liste
 * mit den bereits gekürzten Brüchen verwendet.
 *
 * @param possibleFractures die möglichen gekürzten Brüche
 * @return die generierten Aufgaben
 */
private Set<ReduceFractureExercise> generateExercises(List<Fracture>
                                                        possibleFractures) {
    //bereits genutzte Brüche:
    List<Fracture> usedFractures = new ArrayList<Fracture>();

    Random random = new Random();
    Set<ReduceFractureExercise> exercises = new
            HashSet<ReduceFractureExercise>();

    for (int i = 0; i < number; i++) {
        ReduceFractureExercise exercise =
                generateExercise(possibleFractures, usedFractures, random);

        if (exercise == null) // alle möglichen Aufgaben wurden schon generiert
            break;
        else
            exercises.add(exercise);
    }

    return exercises;
}
```

FractureGenerator.generateExercise()

```
/**
 * Generiert eine ReduceFractureExercise, wobei die angegebenen möglichen
 * gekürzten Brüche und die angegebenen bereits zum Generieren benutzten Brüche,
 * welche verwendet werden, sollten nicht mehr genug noch nicht verwendete
 * Brüche existieren, verwendet werden.
 *
 * @param possibleFractures die möglichen gekürzten Brüche
 * @param usedFractures     die bereits genutzten Brüche
 * @param random            die Instanz von Random, welches für zufällige
 *                          Operationen verwendet werden soll
 * @return eine Aufgabe zum Kürzen von Brüchen
 */
private ReduceFractureExercise generateExercise(
        List<Fracture> possibleFractures, List<Fracture> usedFractures,
        Random random) {

    // Der ungekürzte Bruch:
    Fracture fracture;
    // Der größere Bruch, der auf den oben genannten gekürzt werden soll:
    Fracture biggerFracture = null;

    do {
        // Es gibt gekürzte Brüche, welche noch nicht verwendet wurden:
        if (possibleFractures.size() > 0) {
            fracture = possibleFractures
                    .get(random.nextInt(possibleFractures.size()));

            // Wird jetzt verwendet und deshalb aus der Liste mit den möglichen
            // Brüchen entfernt und kommt zu den genutzten Brüchen hinzu:
            possibleFractures.remove(fracture);
            usedFractures.add(fracture);
        }

        // Alternativ weitere Brüche generieren, deren gekürzte Brüche schon
        // verwendet wurden:
        else if (usedFractures.size() > 0)
            fracture = usedFractures.get(random.nextInt(usedFractures.size()));
        else
            return null; // beide Listen sind leer, keine Brüche mehr möglich

        try {
            biggerFracture = fracture.createBiggerFracture(
                    stage.getFractureLength(), random);
        } catch (NoFracturesFoundException exception) {
            // Der gekürzte Bruch hat keinen weiteren größeren Bruch, der alle
            // Bedingungen einhält. Er wird also aus allen Listen entfernt:
            possibleFractures.remove(fracture);
            usedFractures.remove(fracture);
        }
    }
    // Sollte noch keiner generiert worden sein können, wird es erneut ausgeführt:
    while (biggerFracture == null);

    return new ReduceFractureExercise(biggerFracture, fracture);
}
```

Folgender Code wird in der Hauptklasse ausgeführt, nachdem die Daten eingelesen wurden. Dabei ist `stage` die eingelese Stufe und `number` ist die eingegebene Anzahl von Aufgaben.

```
Set<ReduceFractureExercise> exercises =
        new FractureGenerator(stage, number).generate();

for (ReduceFractureExercise exercise : exercises)
    System.out.println(exercise);
```