- Variante 1

Tag 8: Haunted Wasteland

Tag 8 ist wieder verhältnismäßig einfach.

Die Aufgabe schreit eigentlich nach Rekursion. Leider wird dies bei der Verwendung von Java zu einem Stack-Overflow führen, da viele rekursive Aufrufe nötig sind. Java unterstützt wohl leider auch noch immer nicht die Optimierung bei Verwendung einer "Endrekursion" (wenn dies falsch ist, dann bitte diese Aussage korrigieren). Daher muss auf eine iterative Vorgehensweise zurückgegriffen werden.

Hilfestellung Teil 1

- Lies die erste Zeile als normalen String ein und speichere sie z. B. als `instructions`.
- Lies alle restlichen Zeilen jeweils als String-Array ein (entferne die dazwischenliegenden Zeichen, indem du mehrfach `split()` verwendest). Du weißt dann, dass pro Zeile bei Index 0 immer der 'Input' der Zeile ist, bei Index 1 der linke Nachfolger und bei Index 2 der rechte Nachfolger. Speichere dann alle Zeilen (String-Arrays) in einer ArrayList<String[]>.
- Halte in einer Variablen fest, welche das nächste zu suchende Wort ist (z. B. `nextWord`). Anfangs ist `nextWord` natürlich `"AAA"`.
- Setzte einen `instructionPointer` anfangs auf 0, um dir zu merken, welche nächste Instruktion (R oder L) ausgeführt werden muss. Du kannst dir dann mithilfe dieses Pointers jeweils den Character aus dem String `instructions` geben lassen mit `instructions.charAt(instructionPointer)`.
- Wiederhole nun solange bis `nextWord` dem gesuchten String `"ZZZ"` entspricht:
  - Suche diejenige Zeile, bei der der Input dem nextWord entspricht.
  - Je nachdem, ob die nächste Instruktion nun L oder R ist, wird nextWord auf den Index 1 oder 2 der aktuellen Zeile gesetzt.
  - Erhöhe den `instructionPointer` um eins und stelle sicher, dass er, wenn er das Ende aller Instruktionen erreicht hat, wieder bei 0 beginnt (→ Modulo-Operator %).
  - Zähle in der Schleife außerdem einen counter hoch für die Anzahl der Schritte.

Lösungsvorschlag

```
public int partOne() {
    String instructions = inputLines.get(0).trim();
    int instructionPointer = 0;

    ArrayList<String[]> nodes = new ArrayList<String[]>();
    for (int i = 2; i < inputLines.size(); i++) {
        String line = inputLines.get(i);
        String[] node = new String[3];
        node[0] = line.split("=")[0].trim();
        node[1] =
line.split("[=]")[1].split("[,]")[0].split("[(]")[1].trim();
        node[2] = line.split("=")[1].split(",")[1].split("[)]")[0].trim();
        nodes.add(node);
    }
```

```
    int steps = 0;
    String nextWord = "AAA";
    while (!nextWord.equals("ZZZ")) {
        for (String[] node: nodes) {
            if (node[0].equals(nextWord)) {
                if (instructions.charAt(instructionPointer) == 'L') {
                    nextWord = node[1];
                } else {
                    nextWord = node[2];
                }
                break;
            }
        }
        steps++;
        instructionPointer = (instructionPointer + 1) %
instructions.length();
    }

    return steps;
}
```

Hilfestellung Teil 2

- Tipp vorneweg: die Ergebniszahl wird riesig. Es ist also **keine** Lösung, einfach für alle Start-Wörter (die auf "A" enden) jeweils gleichzeitig einen Schritt zu berechnen und immer zu prüfen, ob nun alle Wörter mit "Z" enden. Dann würde der PC womöglich noch in einigen Monaten am Rechnen sein.
- Schaue dir besser in der Aufgabenerklärung noch mal genau den Verlauf für die beiden Start-Wörter an. Wann / mit welcher Häufigkeit findet jedes Wort sein End-Wort (mit "Z" am Ende)? Erkennst du da ein Muster?

Ja, ich habe das Muster erkannt

- Es genügt tatsächlich, für jedes Start-Wort nur das erste End-Wort zu finden und sich zu merken, wie viele Schritte das waren.
- Am Ende muss man von allen Schritten für alle Start-Wörter nur das kleinste gemeinsame Vielfache (kgV) finden. Da gibt es bei google genügend mögliche Implementationen zu finden.
- Die kgV-Methode kann immer nur zwei Zahlen verarbeiten. Du kannst ja aber in einer Schleife immer eine Zahl aus allen Schritten entfernen, und mit dem bisherigen kgV verrechnen.

Lösungsvorschlag

```
public long partTwo() {
    String instructions = inputLines.get(0).trim();

    ArrayList<String[]> nodes = new ArrayList<String[]>();
```

```
    for (int i = 2; i < inputLines.size(); i++) {
        String line = inputLines.get(i);
        String[] node = new String[3];
        node[0] = line.split("=")[0].trim();
        node[1] =
line.split("[=]")[1].split("[,]")[0].split("[(]")[1].trim();
        node[2] = line.split("=")[1].split(",")[1].split("[)]")[0].trim();
        nodes.add(node);
    }

    ArrayList<String> nextWords = new ArrayList<String>();
    for (String[] node: nodes) {
        if (node[0].endsWith("A")) {
            nextWords.add(node[0]);
        }
    }

    ArrayList<Integer> wordSteps = new ArrayList<Integer>();

    for (String nextWord: nextWords) {
        int steps = 0;
        int instructionPointer = 0;
        while (!nextWord.endsWith("Z")) {
            for (String[] node: nodes) {
                if (node[0].equals(nextWord)) {
                    if (instructions.charAt(instructionPointer) == 'L') {
                        nextWord = node[1];
                    } else {
                        nextWord = node[2];
                    }
                    break;
                }
            }
            steps++;
            instructionPointer = (instructionPointer + 1) %
instructions.length();
        }
        wordSteps.add(steps);
    }

    long result = wordSteps.remove(0);
    while (wordSteps.size() > 0) {
        result = kgv(result, wordSteps.remove(0));
    }
    return result;

}

private long ggt(long a, long b) {
    if (b == 0) {
        return a;
```

```
    } else {
        return ggt(b, a % b);
    }
}

private long kgv(long a, long b) {
    return (a * b) / ggt(a, b);
}
```