FAU, Informatik 2, AUD-Team
aud@i2.cs.fau.de

# 5. Übung

Abgabe bis 28.11.2016, 10:00 Uhr

## Einzelaufgabe 5.1: Induktionsbeweis — 14 EP

Gegeben sei folgende Java-Methode:

```
1  public static long hokuspokus(long n) {
2      if (n > 2) {
3          return 9 * hokuspokus(n - 2) - 16;
4      } else {
5          return 2 * n + 1;
6      }
7  }
```

**a)** Beweisen Sie folgenden Zusammenhang mittels vollständiger Induktion:

$$\forall n \geq 1 : \texttt{hokuspokus(n)} = 3^n - 4 \cdot \sum_{i=0}^{n-2} 3^i$$

Zur Vereinfachung nehmen Sie bitte an, dass der Wertebereich des Rückgabetyps `long` unbeschränkt ist, dass also kein Überlauf auftreten kann. Bitte strukturieren Sie Ihren Beweis so, dass er leicht nachvollziehbar ist – geben Sie insbesondere alle Induktionsanfänge bzw. Induktionsvoraussetzungen explizit an.

**b)** Beweisen Sie, dass die Methode immer terminiert. Geben Sie dazu eine Terminierungsfunktion an und begründen Sie kurz Ihre Wahl.

Geben Sie Ihre Lösung als `Induktionsbeweis.pdf` über EST ab.

## Einzelaufgabe 5.2: Seiteneffekte — 7 EP

In der Vorlesung haben Sie gelernt, dass ein formaler Parameter beim Methodenaufruf als „lokale Variable“ angelegt wird, welcher anschließend der jeweils aktuelle Parameter „zugewiesen“ wird. Auch haben Sie gelernt, dass diese lokalen Variablen nach dem Beenden der Methode vom Aufrufstapel gelöscht/entfernt werden.
Sie sollen sich diesen Mechanismus an folgendem Programm nochmal verdeutlichen. Dabei sei insbesondere auf das *unterschiedliche Verhalten* hingewiesen, das sich ergibt, wenn der formale Parameter keinen primitiven Datentyp hat, sondern z.B. eine Reihung ist. Im Falle primitiver Typen wirken sich Änderungen an den Variablen nur innerhalb der jeweiligen Methode aus – die neuen Werte gehen verloren, sobald der Methodenblock beendet wird. Bei Reihungen in Java ist das aber anders …

```java
    public static final long[] FLUECHTIGES_FINALES_FELD = { 0, 0, 0 };

    public static void aendere_etwas(long etwas) {
        etwas = etwas + 42;
    }

    public static void aendere_etwas_anderes(long[] etwas_anderes) {
        etwas_anderes[1] = etwas_anderes[1] + 42;
    }

    public static void aendere_alles() {
        long etwas = 47;
        aendere_etwas(etwas);
        System.out.println("etwas = " + etwas);
        // ------------------------------
        long anderes = 11;
        aendere_etwas(anderes);
        System.out.println("anderes = " + anderes);
        // ------------------------------
        long[] etwas_anderes = { 0, 8, 15 };
        aendere_etwas_anderes(etwas_anderes);
        System.out.println("etwas_anderes = " + java.util.Arrays.toString(etwas_anderes));
        // ------------------------------
        long[] anderes_etwas = { 0, 8, 15 };
        aendere_etwas_anderes(anderes_etwas);
        System.out.println("anderes_etwas = " + java.util.Arrays.toString(anderes_etwas));
        // ------------------------------
        long[] aber_was = { 6, 6, 6 };
        aendere_etwas(aber_was[1]);
        System.out.println("aber_was = " + java.util.Arrays.toString(aber_was));
        // ------------------------------
        long a = 47, u = 11, d = 666;
        long[] aud = { a, u, d };
        aendere_etwas_anderes(aud);
        System.out.println("a = " + a + ", u = " + u + ", d = " + d);
        // ------------------------------
        aendere_etwas_anderes(FLUECHTIGES_FINALES_FELD);
        System.out.println("FLUECHTIGES_FINALES_FELD = " + java.util.Arrays.toString(FLUECHTIGES_FINALES_FELD));
    }
```

Führen Sie die Methode `aendere_alles()` „gedanklich“ in einem Schreibtischlauf aus. Überlegen Sie sich, welche Werte die Variablen *nach* den Methodenaufrufen von `aendere_etwas` und `aendere_etwas_anderes` haben. Geben Sie die **Ausgabe** des Programms als `Seiteneffekte.pdf` über EST ab.

### Einzelaufgabe 5.3: Fibonacci – Verallgemeinert — 6 EP

Eine der (vielen) möglichen Verallgemeinerungen der Fibonacci-Reihe wird wie folgt definiert:

$$F_n = \begin{cases} n, & \text{falls } 0 \leq n < c \\ a \cdot F_{n-1} + F_{n-c}, & \text{falls } n \geq c \text{ und gerade} \\ b \cdot F_{n-1} + F_{n-c}, & \text{sonst} \end{cases}$$

dabei sind $a$ und $b$ jeweils ungleich 0, sowie $c$ positiv und ganzzahlig. In dieser Aufgabe soll die **rekursive** Berechnung dieser Zahlenfolge implementiert werden. Vervollständigen Sie dazu die Methode `fibonacciVerallgemeinert` der Klasse `FibonacciVerallgemeinert` derart, dass sie für die als Parameter gegebene nicht-negative ganze Zahl `n` das $n$-te Glied der Folge zurückgibt. Reichen Sie Ihre Lösung als `FibonacciVerallgemeinert.java` über EST ein.

**Gruppenaufgabe 5.4: Veggie-Wahn** **11 GP**

Das Küchenpersonal der Mensa hat ein Problem: In einer Umfrage haben sich die Studierenden über das angeblich fade Essen beschwert und zusätzlich gefordert, dass jedes Gericht auch vegan angeboten werden muss. Hier nun der Plan des Studentenwerks:

- Es werden mindestens so viele Gewürze $g$ angeschafft, wie es vegane Grundgerichte $v$ gibt, aus Kostengründen aber maximal 123 verschiedene Gewürze ($1 \leq v \leq g \leq 123$).
- Jedes Gericht soll mit *mindestens* einem Gewürz abgeschmeckt werden.
- Jedes Gewürz soll für *genau* ein Gericht verwendet werden.
- Alle veganen Grundgerichte schmecken ungewürzt gleich und sehen auch identisch aus.
- Bei nicht-veganen Gerichten wird einfach zusätzlich etwas Hackfleisch reingekippt.

Die Verteilung der $g$ Gewürze auf die $v$ veganen Grundgerichte soll möglichst abwechslungsreich gestaltet werden. Für die Erstellung des wöchentlichen Speiseplans muss das Studentenwerk bestimmen, wie viele *unterschiedliche Würzmischungen* $W_{v,g}$ auf diesem Wege überhaupt möglich sind und schließlich wie viele verschiedene Essen $E_{v,g}$ im Speiseplan stehen werden:

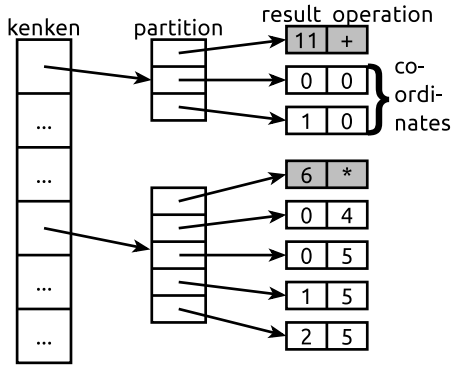
- Bei gleich vielen veganen Gerichten wie Gewürzen ($v = g$) gibt es eine Würzung $W_{v,g} = 1$: Jedes vegane Gericht bekommt genau eines der Gewürze.
- Bei nur einem veganen Gericht ($v = 1$) gibt ebenfalls genau eine Würzmöglichkeit $W_{v,g} = 1$: Alle Gewürze werden in das eine Gericht zusammengekippt.
- Andernfalls gibt es weitere Kochtricks: Man nehme das erstbeste Gewürz $G$ zur Hand und ...
  - entweder gibt es ein Gericht, das nur mit diesem einen Gewürz $G$ abgeschmeckt wird, dann verteilen sich die restlichen $g - 1$ Gewürze auf die anderen $v - 1$ Gerichte;
  - oder das Gewürz $G$ kommt zusammen mit anderen Gewürzen in das gleiche Gericht, dann werden zuerst die anderen $g-1$ Gewürze auf alle $v$ Gerichte verteilt (ergibt $W_{v,g-1}$ Würzmischungen) und anschließend wird das aktuelle Gewürz $G$ zu einem beliebigen der $v$ Gerichte hinzugefügt (wofür es $v$ Möglichkeiten gibt).

  $\rightsquigarrow$ Das macht dann insgesamt $W_{v,g} = W_{v-1,g-1} + v \cdot W_{v,g-1}$ vegane Würzmischungen.
- Die $v$ veganen Grundgerichte und die $W_{v,g}$ Würzmischungen werden schließlich sowohl mit als auch ohne Hackfleisch zu $E_{v,g} = 2 \cdot W_{v,g}$ unterschiedliche Essen zusammengerührt.

Laden Sie die Vorlage `VeggieWahn.java` von der AuD-Homepage herunter und ergänzen Sie die Methode `essen` so, dass sie die oben beschriebene Anzahl $E_{v,g}$ ermittelt. Verwenden Sie zur Implementierung *unbedingt dynamische Programmierung* mittels *Memoization*. Sie dürfen davon ausgehen, dass das Ergebnis Ihrer Rechnung in einen `long` passt.
Geben Sie Ihre Lösung als `VeggieWahn.java` über EST ab.

## Gruppenaufgabe 5.5: *KenKen*™ — 22 GP

Eine Variante von *Sudoku* heißt *KenKen*. Das Rätsel besteht aus einer $(n \times n)$-Matrix $\mathcal{M}$, die durch dickere Umrandung zu unterschiedlichen Partitionen $\mathcal{P}_i$ zusammengefasst werden. Jedem $\mathcal{P}_i$ ist eine Operation $O_i \in \{+, -, *, /, \texttt{' '}\}$ und deren Endergebnis $R_i$ zugeordnet. Das *KenKen* ist gelöst, wenn in jeder Spalte bzw. jeder Zeile jeweils alle Zahlen von 1 bis $n$ genau einmal so vorkommen, dass sich bei Anwenden von $O_i$ auf alle Zahlen in $\mathcal{P}_i$ das angegebene $R_i$ ergibt.



`int[][][] kenken`

Jede der im Folgenden zu entwickelnden Methoden wird mit einem 3D-`int[][][]`-Feld aufgerufen, das ein *KenKen*-Problem repräsentieren soll. Eine korrekte Datenstruktur hat exemplarisch nebenstehenden Aufbau: Jeder Eintrag enthält die Beschreibung einer Partition $\mathcal{P}_i$ als 2D-`int[][]`-Feld. Der erste Eintrag für jede einzelne Partition $\mathcal{P}_i$ enthält wiederum ein 1D-`int[]`-Feld der Länge 2 mit dem $(R_i, O_i)$-Paar. Alle weiteren Einträge von $\mathcal{P}_i$ sind ebenfalls 1D-`int[]`-Felder der Länge 2 und enthalten die Koordinaten eines zu $\mathcal{P}_i$ gehörenden Zahlenfeldes der Matrix $\mathcal{M}$ in der Form *(Zeile, Spalte)*. Das Zahlenfeld „oben/links“ in $\mathcal{M}$ hat die Koordinaten $(0, 0)$.

Erstellen Sie eine Klasse `KenKen` und implementieren Sie darin die einzelnen Methoden gemäß ihrer `API` wie folgt:

**a)** Ergänzen Sie zunächst die Methode `checkIntegrity()`. Sie soll überprüfen, ob das übergebene 3D-Array strukturell als *KenKen*-Problem (wie oben beschrieben und im Beispiel-Bild gezeigt) interpretiert werden kann.

**b)** Implementieren Sie nun die Methode `checkValidity()`. Sie soll überprüfen, ob das übergebene 3D-Array semantisch als *KenKen*-Problem interpretiert werden kann, d.h. ob die Partitionen überlappungsfrei sind und insgesamt genau ein quadratisches *KenKen* abdecken (die Partitionen selbst müssen dabei nicht zwangsweise zusammenhängend sein).

**c)** Programmieren Sie schließlich ein rekursives Verfahren zur Lösung eines *KenKen*-Problems in `solve()`. Falls es mindestens eine Lösung gibt, dann soll `solve()` eine beliebige davon als quadratisches Array zurückgeben.

Geben Sie Ihre vollständige Datei `KenKen.java` über EST ab.

**27 EP + 33 GP = 60 Punkte**