TU Berlin, Fakultät IV
Blankertz/Röhr/Stahl
MAR 4-3 und MAR 5-1

Aufgabenblatt 3

Backtracking, JUnit Tests & Greedy Algorithmen

Zur Erinnerung

- **Erinnerung Prüfungsanmeldung**: (für die meisten: in QISPOS) Deadline ist am **25.05.2019**. Ohne Prüfungsanmeldung können Sie nicht an der Klausur teilnehmen und bekommen keine Prüfungsleistungen angerechnet.
- Alle Übungen sind in Einzelarbeit zu erledigen. Kopieren Sie niemals Code und geben Sie Code in keiner Form weiter. Die Hausaufgaben sind Teil Ihrer Prüfungsleistung. Finden wir ein Plagiat (wir verwenden Plagiatserkennungssoftware), führt das zum Nichtbestehen des Kurses für alle Beteiligten.
- Wenn Ihre Abgabe nicht im richtigen Ordner liegt, nicht kompiliert, unerlaubte packages oder imports enthält oder zu spät abgegeben wird, gibt es **0 Punkte** auf diese Abgabe.

Abgabe (bis 21.05.2019 19:59 Uhr)

Die folgenden Dateien müssen für eine erfolgreiche Abgabe im git Ordner eingecheckt sein:

Geforderte Dateien:	
`Blatt03/src/PermutationTest.java`	Aufgabe 4

Als Abgabe wird jeweils nur die letzte Version im git gewertet.

Aufgabe 1: Greedy Algorithmen (Tut)

Gehen Sie davon aus, dass Sie möglichst viele Apps auf Ihrem Handy installieren wollen und noch 1 GB Speicher nutzen können.

1.1 Diskutieren Sie an dem folgenden Beispiel, wie ein Greedy Algorithmus vorgehen würde: App1 (400MB), App2 (200MB), App3 (140MB), App4 (160MB), App5 (100MB), App6 (110)

1.2 Wird mit einem Greedy Algorithmus der Speicherplatz optimal (ggf. vollständig) ausgenutzt?

1.3 Wird mit einem Greedy Algorithmus die maximale Anzahl an Programmen installiert?

Aufgabe 2: JUnit Tests (Tut)

Betrachten Sie das folgende Programm:

```java
public class StringGenome {
        private String s = "";

        public void addNucleotide(char c) {
                if (c == 'A'  c == 'C'  c == 'G'  c == 'T')
                        s = s + c;
                else
                        throw new RuntimeException("Illegal nucleotide");
        }

        public char nucleotideAt(int i) {
                if (i < s.length())
                        return s.charAt(i);
                else
                        throw new RuntimeException("Genome out of bounds");
        }

        public int length() {
                return s.length();
        }

        public String toString() {
                return s;
        }

        @Override
        public boolean equals(Object obj) {
                StringGenome i = (StringGenome) obj;
                return i.s.equals(this.s);
        }

        @Override
        public int hashCode() {
                return this.s.hashCode();
        }
}
```

2.1 Welche Aspekte der Klasse würden Sie in einem JUnit Test testen?

2.2 Wie wird eine Testvorlage in IDEA importiert?

2.3 Wie könnte man diesen Test implementieren? Welches sind die wichtigsten Befehle, die in JUnit-Tests benutzt werden?

Aufgabe 3: Backtracking (Tut)

Das N Queen Problem beschreibt die Aufgabe N Königinnen auf ein NxN Schachfeld zusetzen, ohne dass sie sich gegenseitig schlagen können.

3.1 Wie würden sie Backtracking in diesem Fall anwenden?

3.2 Führen sie für den Fall $N = 4$ eine Handsimulation durch.

Aufgabe 4: JUnit Tests for Backtracking (Hausaufgabe)

Das Ziel dieser Aufgabe ist die Entwicklung von JUnit Tests. Die zu testende Klasse benutzt *Backtracking*, um alle fixpunktfreien Permutationen (oder Derangements) einer Zahlenfolge zu bestimmen.

Definition 1: Fixpunktfreie Permutation bzw. Derangement

Eine Permutation ist fixpunktfrei (bzw. ein Derangement), wenn sie keine Elemente an ihrem Platz lässt. Für die Permutationsabbildung π gilt dann also $\pi(i) \neq i$ für alle i. Für die Folge $[2, 4, 6, 8]$ ist $[8, 6, 2, 4]$ ein Derangement, während $[8, 2, 6, 4]$ es nicht es, da die 6 an ihrem Platz bleibt.

Es sind zwei Klassen gegeben, die von der Klasse `PermutationVariation` erben. Die Klassen sollen jeweils die Menge aller fixpunktfreien Permutationen bestimmen. Eine der beiden Klassen funktioniert, die andere nicht. Ihre Aufgabe ist es die JUnit-Testklasse `PermutationTest` zu schreiben, welche die beiden gegebenen Klassen und Ihnen nicht vorliegende Testfälle (also weitere Unterklassen von `PermutationVariation`) auf Korrektheit testet.

4.1 Konstruktor (35 Punkte)

Schreiben Sie den JUnit-Test

```
        void testPermutation()
```

die den Konstruktor der Permutationsklassen testet. Die Variable `original` muss im Konstruktur mit einer Folge der vorgegeben Länge initialisiert werden, in der keine Zahl doppelt vorkommt. Desweiteren muss `allDerangements` mit einer leeren Liste initialisiert werden.

4.2 Derangements (35 Punkte)

Schreiben Sie den JUnit-Test

```
        void testDerangements()
```

der die `derangments()`-Methode der Permutationsklassen testet. Diese Methode erzeugt alle fixpunktfreien Permutationen und speichert sie in `allDerangements`. In diesem Test soll überprüft werden, dass die Anzahl der erzeugten Derangements korrekt ist, und, dass jedes Derangement die Eigenschaft der Fixpunktfreiheit erfüllt. (Ob es sich tatsächlich um Permutationen handelt wird separat in c) überprüft.)

4.3 Elemente (30 Punkte)

Schreiben Sie den JUnit-Test

```
        void testsameElements()
```

der überprüft, ob alle von der `derangments()`-Methode erzeugten Folgen tatsächlich Permutationen des Originalarrays `original` sind. Wenn keine Permutationen berechnet wurden, sollte dieser Test fehlschlagen.

Hinweise:

- Wenden Sie Ihren Test auf die beiden gegebenen Klassen `Permutation` und `Permutation1` an. Um auszuwählen, welche Klasse getestet wird, ändern Sie den Initialisierungswert von `cases` in `PermutationTest` (0 oder 1).
- Die Tests sollen beliebige Unterklassen von `PermutationVariation` testen. In der Vorgabe werden zwei Objekte von (Unterklassen von) `PermutationVariation` erzeugt, nämlich `p1` und `p2` durch die Klasse `Cases` mit den Argumenten `n1` und `n2` und `cases`.
- Sie sollten nach Möglichkeit Code-Verdopplung vermeiden. Dazu ist es zweckmäßig Hilfsmethoden zu schreiben (ohne `@Test` Annotation, damit sie nicht als eigenständiger Test ausgeführt werden). Dies ist nur eine Empfehlung und wird nicht von den Korrekturtests überprüft.
- Beispiele für eine solche Hilfsmethoden sind `initialize()` und `fixConstructor()`. Letztere repariert einen ggf. falschen Konstruktor, damit die anderen Tests davon unabhängig laufen können.
- Zur Berechnungen der Anzahl fixpunktfreier Permutation: Zum Potenzieren wird die Funktion `Math.pow(a, b)` verwendet (`^` ist in Java ein bitweiser Operator).