

# Aufgabenblatt 3

## *Backtracking & Greedy Algorithmen*

### Zur Erinnerung

- Melden Sie sich für die Prüfung auf MOSES an.
- Alle Übungen sind in Einzelarbeit zu erledigen. Kopieren Sie niemals Code und geben Sie Code in keiner Form weiter.
- 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 20.05.2024 23:59 Uhr)

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

#### Geforderte Dateien:

Blatt03/src/PermutationTest.java    Aufgabe 4

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

### Aufgabe 1: Greedy Algorithmen (Klausurvorbereitung)

Gehen Sie davon aus, dass Sie möglichst viele Apps auf Ihrem Handy installieren wollen und noch 1 GB Speicher nutzen können. Die einzige Information, die Sie zu den Apps haben ist der benötigte Speicherplatz.

- 1.1 Was wäre eine sinnvolle Greedy Auswahlstrategie um die größtmögliche Anzahl an Apps zu installieren? Gehen Sie an folgendem Beispiel durch, wie so ein Greedy Algorithmus vorgehen würde:  
App1 (400MB), App2 (200MB), App3 (140MB), App4 (160MB), App5 (100MB), App6 (110)  
In welcher Reihenfolge werden die Apps ausgewählt?
- 1.2 Wird mit dem Greedy Algorithmus der Speicherplatz optimal (ggf. vollständig) ausgenutzt?
- 1.3 Wird mit einem Greedy Algorithmus die maximale Anzahl an Programmen installiert?

### Aufgabe 2: Backtracking (Klausurvorbereitung)

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

- 2.1 Wie würden sie Backtracking in diesem Fall anwenden?
- 2.2 Führen sie für den Fall  $N = 4$  eine Handsimulation durch. Zu dieser Aufgabe gibt es ein Video-tutorium.

## Aufgabe 3: JUnit Tests

Betrachten Sie das folgende Programm, welches Sie auch im git finden:

```
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();
    }
}
```

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

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

Sie finden eine Testvorlage ebenfalls im git. Damit Sie JUnit nutzen können, müssen Sie ggf. JUnit5 zum classpath hinzufügen.

## 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. Genutzt wird JUnit5.

### Definition 1: Fixpunktfreie Permutation bzw. Derangement

Eine Permutation ist fixpunktfrei (bzw. ein Derangement), wenn sie keine Elemente an ihrem Platz lässt. Für die Permutationsabbildung  $\pi$  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 ist, 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 für Derangements der Länge  $> 1$  und  $< 10$ .

#### 4.1 Konstruktor (35 Punkte)

Schreiben Sie den JUnit-Test

```
void testPermutation()
```

der den Konstruktor der Permutationsklassen testet. Die Variable `original` muss im Konstruktor mit einer Folge der vorgegebenen 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 der nächsten Teilaufgabe 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`. Ihre Tests sollen unter anderem also auf eben jenen `p1` und `p2` laufen. Es kann hilfreich sein, den Ursprung und die Definition von `p1/p2` im Code nachzuverfolgen, um genauer zu verstehen, wie die Daten aussehen.
- 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 Permutationen: Zum Potenzieren wird die Funktion `Math.pow(a, b)` verwendet (^ ist in Java ein bitweiser Operator).

### **Was Sie nach diesem Blatt wissen sollten:**

- Wie treffen Greedy Algorithmen Entscheidungen? Welche Informationen werden für die Entscheidungen genutzt?
- Was ist Rekursion? Was sind rekursive Aufrufe?
- Wie funktioniert Backtracking und wie wird Rekursion dabei genutzt?
- Für welche Probleme ist Backtracking geeignet?
- Was ist der Baum der Teillösungen? Und wie können Backtracking und Greedy Algorithmen anhand dessen erklärt werden?
- Was sind JUnit Tests und wie schreibt man sie?
- Welche Arten von Fehlern werden über JUnit Tests überprüft?