**Informatik I: Einführung in die Programmierung**

Prof. Dr. Bernhard Nebel
Tim Schulte, Thorsten Engesser
Wintersemester 2017/2018

Universität Freiburg
Institut für Informatik

# Übungsblatt 5

**Abgabe: Freitag, 24. November 2017, 20:00 Uhr**

**Wichtige Hinweise:** Zur Bearbeitung der Übungsaufgaben legen Sie bitte ein neues Unterverzeichnis `sheet05` im Wurzelverzeichnis Ihrer Arbeitskopie des SVN-Repositories an. Ihre Lösungen werden dann in Dateien in diesem Unterverzeichnis erwartet. Beachten Sie bitte bei allen Aufgaben die *Hinweise zur Bearbeitung der Übungsaufgaben* unter der folgenden URL:

`http://gki.informatik.uni-freiburg.de/teaching/ws1718/info1/guide/hinweise.html`

Beachten Sie weiter, dass für Python-Abgaben ab sofort die Einhaltung des Style Guides vorausgesetzt wird. Achten Sie insbesondere darauf, dass ihr Code ordentlich dokumentiert ist. Siehe:

`http://gki.informatik.uni-freiburg.de/teaching/ws1718/info1/guide/styleguide.html`

Bewertet wird bei allen Aufgaben die letzte Version, die zur Deadline des Übungsblattes auf dem SVN-Server eingereicht ist.

**Aufgabe 5.1** (Permutationen; Datei: `permutation.py`; Punkte: 3+2+2)

Unter einer *n-Permutation* (für $n \in \mathbb{N}$) verstehen wir im Folgenden eine bijektive Abbildung $\sigma : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$. Intuitiv lassen sich Permutationen als Vertauschungsvorschrift interpretieren (siehe `https://de.wikipedia.org/wiki/Permutation`). Die *Identitätspermutation* ist hierbei eine besondere Permutation, die jedes Element aus der Menge $\{1, 2, \ldots, n\}$ auf sich selbst abbildet.
In den folgenden Teilaufgaben werden wir typische Operationen für Permutationen implementieren. Dazu werden wir $n$-Permutationen in der üblichen Tupelschreibweise als Python-Tupel der Länge $n$ repräsentieren: ein Python-Tupel $(k_0, k_1, \ldots, k_{n-1})$, in dem alle $k_i$ aus $\{1, \ldots, n\}$ sowie paarweise verschieden sind, repräsentiert die $n$-Permutation $\sigma$, die der Zahl 1 die Zahl $k_0$, der Zahl 2 die Zahl $k_1$, etc., zuordnet.

(a) Durch (einmalige) *Anwendung einer n-Permutation $\sigma$ auf eine Sequenz (der Länge $n$)* $\overline{x} = x_1, x_2, \ldots, x_n$ erhält man die Sequenz $\overline{x}^{\sigma} = x_{\sigma(1)}, x_{\sigma(2)}, \ldots, x_{\sigma(n)}$.

Implementieren Sie in der Datei `permutation.py` eine Funktion `apply(s, lst)`, die eine $n$-Permutation `s` auf eine Liste `lst` der Länge $n$ einmalig anwendet. Die Funktion soll die eingegebene Liste `lst` verändern und `None` zurückgegeben.

Betrachten Sie dazu folgendes Beispiel:

```
>>> sigma = (4, 3, 2, 1)
>>> spam = ['s', 'p', 'a', 'm']
>>> apply(sigma, spam)
>>> spam
['m', 'a', 'p', 's']
>>> sigma = (1, 3, 4, 2)
>>> spam = ['s', 'p', 'a', 'm']
>>> apply(sigma, spam)
>>> spam
['s', 'a', 'm', 'p']
```

(b) Die *Komposition* zweier $n$-Permutationen $\sigma$ und $\tau$ ist definiert als die $n$-Permutation, die zuerst $\tau$ und dann $\sigma$ ausführt; formal: $(\sigma \circ \tau)(x) := \sigma(\tau(x))$.

Implementieren Sie in der Datei `permutation.py` eine Funktion `compose(s, t)`, die die Komposition zweier $n$-Permutationen `s` und `t` zurückgibt. Betrachten Sie dazu folgende Beispiele:

```
>>> sigma = (1, 3, 4, 2)
>>> tau = (3, 1, 2, 4)
>>> compose(sigma, tau)
(4, 1, 3, 2)
>>> compose(tau, tau)
(2, 3, 1, 4)
```

(c) Die *Ordnung* einer $n$-Permutation $\sigma$ ist definiert als die kleinste natürliche Zahl $k > 0$, so dass die $k$-fache Komposition von $\sigma$ die Identitätspermutation $(1, 2, \ldots, n)$ ergibt. Die $k$-fache Komposition von $\sigma$ ist hierbei rekursiv wie folgt definiert: $\sigma^1 := \sigma$ und $\sigma^{k+1} := (\sigma \circ \sigma^k)$.

Implementieren Sie eine Funktion `order(s)`, die bei Eingabe einer $n$-Permutation `s` die Ordnung von `s` zurückgibt. Betrachten Sie dazu folgende Beispiele:

```
>>> sigma = (2, 1, 4, 3)
>>> order(sigma)
2
>>> sigma = (2, 3, 4, 1)
>>> order(sigma)
4
```

**Aufgabe 5.2** (Liste glätten; Datei: `flatten.py`; Punkte: 2+2)

Wir nennen eine Liste *verschachtelt*, wenn sie mindestens eine weitere Liste als Element enthält. Die geglättete Version einer (verschachtelten oder nicht-verschachtelten) Liste ergibt sich wie folgt: jedes Element $L$, das eine Liste ist, wird (unter Beibehaltung der Reihenfolge) durch die Elemente der geglätteten Version von $L$ ersetzt; jedes andere Element wird (unter Beibehaltung der Reihenfolge) übernommen.

*Hinweis:* Bei der Bearbeitung der folgenden Aufgaben sollen nur Hilfsmittel verwendet werden, die bisher in der Vorlesung eingeführt wurden. Lösungen, die zusätzliche Module importieren, werden nicht bewertet.

(a) Definieren Sie eine rekursive Funktion `flatten(lst)`, die die geglättete Version der übergebenen Liste `lst` berechnet und zurückgibt. Ihre Funktion soll dabei eine komplett neue Liste zurückgeben und darf die übergebene Liste bzw. alle betrachteten Teillisten nicht verändern. Betrachten Sie dazu das folgende Beispiel:

```
>>> egg = [3,4,[[5]]]
>>> spam = [[[1, 2, egg], (6, [7]), 8], 9, False]
>>> flatten(spam)
[1, 2, 3, 4, 5, (6, [7]), 8, 9, False]
>>> spam
[[[1, 2, [3, 4, [[5]]]], (6, [7]), 8], 9, False]
>>> egg
[3, 4, [[5]]]
```

(b) Definieren Sie eine rekursive Funktion `flatten_in_place(lst)`, die eine beliebige Liste `lst` übernimmt und die Liste selbst glättet (in Gegensatz zu (a)). Bei einem Aufruf von `flatten_in_place` sollen auch alle Teillisten geglättet werden. Betrachten Sie dazu folgendes Beispiel:

```
>>> egg = [3,4,[[5]]]
>>> spam = [[[1, 2, egg], (6, [7]), 8], 9, False]
>>> flatten_in_place(spam)
>>> spam
[1, 2, 3, 4, 5, (6, [7]), 8, 9, False]
>>> egg
[3, 4, 5]
```

**Aufgabe 5.3** (Wort-Baum; Datei: `words.py`; Punkte: 3+2+2)

In dieser Aufgabe geht es darum, eine Zeichenfolge (einen String) einzulesen und dabei eine Datenstruktur anzulegen, die es später erlaubt für ein gegebenes Wort zu entscheiden, ob und wie oft dieses Wort in der Zeichenkette vorkommt. Unter einem *Wort* verstehen wir im Folgenden jede endliche Folge von Buchstaben des deutschen Alphabets (also den Zeichen a, b, c, ..., z, A, B,..., Z, ä, Ä, ö, Ö, ü, Ü, ß) der Länge $\geq 1$. Je zwei Wörter in der Zeichenfolge werden durch eine nicht-leere, endliche Folge von Zeichen, die nicht zu diesen Buchstaben gehören (z.B., Leerzeichen, Satzzeichen, Zeilenumbrüche), getrennt.

Laden Sie das Template `words.py` von der Vorlesungswebsite herunter. Dieses enthält bereits eine Funktion `next_word(s)`, die angewendet auf einen String `s` ein Tupel `(word, rest)` zurückgibt, wobei `word` das erste Wort (im Sinne der Spezifikation) in `s` ist und `rest` die Zeichenfolge ist, die in `s` auf `word` folgt.

Es soll nun ein Suchbaum der in einem String `s` vorkommenden Wörter erzeugt werden: Jeder Knoten des Suchbaumes wird durch eine Liste

```
[ltree, word, n, rtree]
```

repräsentiert. Dabei ist `word` ein Wort, `n` die Anzahl der Vorkommnisse von `word` in `s`, `ltree` der linke und `rtree` der rechte Teilbaum. Als Ordnungsrelation verwenden wir Python's lexikographische Ordnung von Strings. Das heißt, ein Wort `w` wird im linken Teilbaum eines Knotens `[ltree, word, n, rtree]` eingefügt bzw. gesucht, falls der Vergleich `w < word` den Wert `True` zurückgibt, etc. In Blattknoten sind `ltree` und `rtree` jeweils der Wert `None`.

(a) Definieren Sie eine Funktion `word_tree(s)`, die aus dem übergebenen String `s` diesen Suchbaum erzeugt und zurückgibt. Natürlich kann Ihre Funktion eine selbst-definierte Hilfsfunktion verwenden.

(b) Definieren Sie eine Funktion `word_freq(tree, word)`, die für einen solchen Suchbaum `tree` und ein Wort `word`, die in `tree` hinterlegte Anzahl der Wortvorkommnisse von `word` zurückgibt. Falls das Wort in dem Baum nicht vorkommt, soll die Funktion den Wert `0` zurückgeben.

(c) Definieren Sie eine Funktion `print_tree(tree)`, die alle in `tree` abgelegten Wörter und die in `tree` jeweils hinterlegte Anzahl der jeweiligen Wortvorkommnisse zeilenweise (pro Zeile ein Wort und dessen Anzahl) ausgibt. Dabei soll der Baum in symmetrischer Reihenfolge (*In-Order*) traversiert werden.

Die Ausgabe könnte in etwa wie folgt aussehen:

```
>>> s = "spam eggs spam eggs ham spam hamham Spam eggs hamham"
>>> tr = word_tree(s)
>>> print_tree(tr)
Spam    : 1
eggs    : 3
ham     : 1
hamham  : 2
spam    : 3
```

**Aufgabe 5.4** (Erfahrungen; Datei: `erfahrungen.txt`; Punkte: 2)

Legen Sie im Unterverzeichnis `sheet05` eine Textdatei `erfahrungen.txt` an. Notieren Sie in dieser Datei kurz Ihre Erfahrungen beim Bearbeiten der Übungsaufgaben (Probleme, benötigter Zeitaufwand nach Teilaufgabe, Bezug zur Vorlesung, Interessantes, etc.).