Prof. Dr. Stefan Göller
Florian Bruse
Dr. Norbert Hundeshagen

Einführung in die Informatik

WS 2018/2019

Übungsblatt 5
22.11.2018-29.11.2018

Abgabe: Bis zum 29.11. 18:00 Uhr über moodle. Reichen Sie pro Aufgabe, die Sie bearbeitet haben, genau eine Textdatei mit dem Namen `aufgabe_i.py`, (wobei i die Aufgabennummer ist) ein, welche die Lösung ihrer Gruppe enthält.

Achtung: Halten Sie sich an die vorgebenen Dateiformate und Funktionsköpfe. Abweichungen werden mit Punktabzug sanktioniert.

Aufgabe 1 (lambda, filter, map) (4*5=20 Punkte):

a) Schreiben Sie eine Funktion `filter_palindrom(liste)` aus einer gegebenen Liste aus Zeichenketten alle Palindrome herausfiltert und die Restliste zurückgibt. Ihre Funktion darf (neben dem Kopf) nur eine Zeile Programmcode enthalten. Ein Palindrom ist eine Zeichenkette die vorwärts und rückwärts gelesen gleich ist. Zum Beispiel `"abba"`.

   Hinweis: In der Vorlesung wurde gezeigt, wie mit Hilfe des Zeichenketten-Slicings `[start:stop:schritt]` eine Zeichenkette gespiegelt werden kann. Dies kann benutzt werden um Palindrome zu erkennen.

b) Schreiben Sie eine Funktion `liste_kleiner(liste1, liste2)` welche aus zwei gegebenen gleichlangen Integer-Listen eine neue Liste erzeugt, die jeweils elementweise die kleinere Zahl der beiden Listen enthält, und dann diese Liste zurückgibt..

   Beispiel: Bei Aufruf `liste_kleiner([5,7,-3,-1], [2,10,-3,5])` soll von Ihrer Funktion die Liste `[2,7,-3,-1]` erzeugt werden.

   Hinweis: Ihre Funktion muss nur mit zwei gleichlangen Integer-Listen funktionieren.

c) Schreiben Sie eine Funktion `custom_op(liste)` der eine Liste übergeben werden soll. Um zu beschreiben, was die Funktion leisten soll, schauen wir uns zunächst deren Einsatzbereich an. Die Funktion soll innerhalb der `map`-Operation wie folgt benutzt werden:

```
list(map(custom_op(L),L))
```

und folgendes leisten:

- überprüfen, ob es sich bei der Liste `L` um (1) eine Liste nur aus Integern, (2) eine Liste nur aus Zeichenkettenor (3) eine andere Liste handelt

  und

zu (1) ermöglichen, dass durch die `map`-Operation alle Listenelemente in `L` verdoppelt werden

zu (2) ermöglichen, dass durch die `map`-Operation alle Listenelemente in `L` gespiegelt (siehe Hinweis Aufgabenteil a)) werden

zu (3) die Liste unverändert lassen

Beispiel: Bei Eingabe `L=[1,2,3]` soll `map(custom_op(L),L)` die Liste `[2,4,6]` zurückliefern. Bei Eingabe `L=["ab","3a5",]` soll `map(custom_op(L),L)` die Liste `["ba","5a3"]` zurückliefern.

d) Gegeben ist folgende Funktion `sortiere(kleiner, L)`, welche als Parameter eine Funktion `kleiner` und eine Liste `L` besitzt. Die Funktion gibt die nach `kleiner` sortierte Liste `L` zurück.

```
def sortiere(kleiner,L):
        trenner = 0
        while trenner != len(L):
                for i in range(trenner,len(L)):
                        if kleiner(L[i],L[trenner]):
                                L[trenner], L[i] = L[i], L[trenner]
                trenner += 1
        return L
```

Schreiben Sie nun einen lambda-Ausdruck zur längenlexikographischen Ordnung zweier Zeichenketten. Dieser soll dann dem Parameter `kleiner` übergeben werden, um eine Liste aus Zeichenketten längenlexikographisch zu sortieren. Dabei ist die längenlexikographische Ordnung wie folgt definiert: Seien w_1 und w_2 zwei Zeichenketten, w_1 ist **längenlexikographisch kleiner** als w_2, wenn

- entweder w_1 kürzer ist als w_2,
- oder im Fall das $\texttt{len}(w_1) = \texttt{len}(w_2)$, w_1 lexikographisch vor w_2 kommt.

Testen Sie Ihren lambda-Ausdruck indem Sie `sortiere` auf der Liste `["z","aaa","1","11111","ccv","22"]` aufrufen.

Hinweis: Lexikographisch lassen sich zwei Zeichenketten in Python einfach mit `<` vergleichen.

Aufgabe 2 (Das erweiterte Streichholzspiel) (2+3+20=25 Punkte):

In der Vorlesung wurde das *Streichholzspiel* eingeführt. Dabei war ein Haufen von n Streichholzern gegeben. Ein Zug eines Spielers bestand darin entweder ein, zwei oder drei Streichhölzer vom Haufen zu entfernen. Derjenige Spieler, der den Haufen zuerst leert, hatte verloren.

Das *erweiterte Streichholzspiel* hat sieben nichtleere Haufen. Ein *Zug* besteht darin von genau einem nichtleeren Haufen ein oder mehrere (womöglich alle) Streichhölzer zu entfernen. Verloren hat derjenige Spieler, der den letzten nichtleeren Haufen leert. Erweitern Sie den Backtracking-Algorithmus mittels Memoization zur Berechnung der Gewinnstrategie im Streichholzspiel (aus der Vorlesung) auf das erweiterte Streichholzspiel. Befolgen Sie dazu folgende Punkte:

a) Stellen Sie dazu den aktuellen Spielstand als 7-Tupel mit Elementen vom Typ `int` dar, wobei der Eintrag an Position i beschreibt wieviel Streichhölzer auf Haufen i noch übrig sind (die Haufen haben daher, der Einfachheit halber, Indizes 0 bis 6).

b) Schreiben Sie eine Initialisierungfunktion `init(x)`, die einen Maximalwert x entgegennimmt und jeden Haufen mit einem Zufallswert aus `range`$(1, x+1)$ initialisiert. Die Funktion soll dann diesen Spielstand (wie bei Teilaufgabe a) beschrieben) zurück geben. Verwenden Sie das in der Vorlesung eingeführte Paket `random`.

c) Schreiben Sie, ähnlich wie in der Vorlesung, eine Funktion `strategie`, die einen Spielstand und ein geeignetes Memoisierungswörterbuch übergeben bekommt (so dass bereits berechnete Zwischenwerte nicht nochmal neu berechnet werden) und

   - -1 zurückgibt, falls der anziehende Spieler von diesem Spielstand aus keine Gewinnstrategie hat,
   - ein Paar (i, j) zurückgibt, falls der anziehende Spieler eine Gewinnstratie hat und ein möglicher Gewinnzug darin besteht, vom Haufen i genau j Streichhölzer zu entfernen.

Hinweis. Ihr Memoisierungswörterbuch soll als Schlüsselmenge Spielstände und als Werte

entweder Tupel der Form (i, j) oder -1 annehmen.

Aufgabe 3 (Rekursion, Backtracking, Memoization) (3+3+6+3=15 Punkte):

Es soll ein Programm geschrieben werden, welches für einen Springer auf einem Schachbrett den kürzesten Weg zu jedem anderen Schachbrettfeld berechnet. Informieren Sie sich dazu zunächst über die erlaubten Züge eines Springers, z.B. auf

https://de.wikipedia.org/wiki/Springer_(Schach).

Das Programm soll nun im Folgenden schrittweise erarbeitet werden:

a) Das Programm soll mit beliebig großen **quadratischen Schachbrettern** arbeiten können. Schreiben Sie daher zunächst den Programmrahmen der folgendes leisten soll:

- Einlesen der Größe n des Schachbretts,
- Einlesen der Position `pos` des Springers auf dem Schachbrett und Speichern als Tupel (i, j), wobei i für die Zeile und j für die Spalte steht,
- Erstellen einer Liste `brett` der Länge n, die selbst n Listen der Länge n enthält (diese bezeichnen wir auch als $n \times n$-Liste), welche alle mit -1 gefüllt sind, bis auf die Zelle die der Position des Springers entspricht, welche mit 0 gefüllt ist.

  Hinweis: Obige Liste kann sehr einfach mit **List-Comprehensions** erzeugt werden.

Beispiel: Für `n=4` und `pos=(2,1)` soll `brett` also die Liste

```
[[-1, -1, -1, -1], [-1, -1, -1, -1], [-1, 0, -1, -1], [-1, -1, -1, -1]]
```

enthalten.

Als Schachbrett kann man sich diese Datenstruktur also wie folgt vorstellen. Der Einfachheit halber ist hier noch die Zeilen und Spaltennummerierung mit angegeben.

	0	1	2	3
0	-1	-1	-1	-1
1	-1	-1	-1	-1
2	-1	0	-1	-1
3	-1	-1	-1	-1

b) Schreiben Sie eine Funktion `possible_moves( tupel, n)`, welche das Tupel der Position des Springers sowie die Größe des Schabretts übergeben bekommt und eine Liste von Tupeln der möglichen Position die der Springer in einem Zug erreichen

kann, zurückgibt. Achten Sie darauf, dass nur Positionen auf dem Schachbrett zurück gegeben werden.

c) Schreiben Sie nun eine Funktion `min_moves(brett, position, n)`, welche ein Schachbrett, die aktuelle Position des Springers und die Anzahl der bis zur aktuellen Position benötigten Schritte, übergeben wird.

   Die Funktion soll dann rekursiv die minimale Schrittzahl für jedes Feld des Schachbretts berechnen und diese in das jeweilige Feld eintragen. Sie können dabei wie folgt vorgehen:

   - überprüfen, ob man durch den nächsten möglichen Schritt in ein Feld gelangt, für das man bisher mehr Schritte benötigt hat,
   - falls ja, trage die aktuelle Schrittanzahl in das Feld ein und ziehe weiter von diesem Feld,
   - falls nein, probiere den nächsten möglichen Zug.

   Eine geeignete Abbruchbedingung für den rekursiven Aufruf dieser Funktion ist die Situation in der sich von einer Position aus keine Änderung mehr auf dem Schachbrett ergeben. D.h. von dieser Position aus gibt es keinen kürzeren Weg mehr zu den benachbarten Feldern.

d) Setzen Sie obige Programmteile zu einem fertigen Programm zusammen, welches das Schachbrett mit den enthaltenen Zahlen für die Anzahl der minimalen Züge bis zu diesem Feld ausgibt.

   Beispielausgabe:

```
[[1, 4, 1, 5], [2, 3, 2, 1], [3, 0, 3, 2], [4, 3, 2, 1]]
```