Übung 2 Abgabe 15.5.2020

Aufgabe 0 – Debuggen von Übung 1

Studieren Sie die Musterlösung zu Übung 1 auf Moodle und vergleichen Sie mit Ihrer eigenen Lösung. Fügen Sie Kommentare in Ihre ursprüngliche Abgabe ein, welche Fehler Sie gemacht haben und wie die Lösung korrigiert werden muss. Kommentieren Sie auch Dinge, die bei Ihnen zwar korrekt, aber umständlich oder ineffizient gelöst sind. Je besser Ihre Kommentare sind, desto weniger Punkte werden abgezogen. Insbesondere werden keine Punkte für Folgefehler abgezogen, wenn Sie die Grundursache eines Problems identifiziert und korrigiert haben. In Quellcode-Dateien kennzeichnen Sie Ihre Kommentare mit

```
# comment: <Ihr Kommentar>
```

In PDF-Dateien verwenden Sie eine geeignete Farbe. Wichtig ist, dass die Tutoren die Kommentare leicht von der ursprünglichen Abgabe unterscheiden können.

Benennen Sie die Dateien mit kommentierten Lösungen in "`kommentiert_<original-filename>`" um und geben Sie sie im zip-File für Übung 2 ab.

Aufgabe 1 – Eigenschaften von Sortierverfahren 20 Punkte

a) Um Daten sortieren zu können, muss auf den Elementen eine Relation '≤' definiert sein, die die Eigenschaften einer *totalen Ordnung* (siehe https://de.wikipedia.org/wiki/Ordnungsrelation) hat. Zeigen Sie, dass dadurch auch die anderen Relationen '<', '>', '≥', '==' und '!=' festgelegt sind, indem Sie diese so definieren, dass in den Definitionen nur die '≤' Relation und logische Operationen (not, and, or) vorkommen. *3 Punkte*

b) Nach dem Sortieren eines Arrays a der Größe N ist das folgende Axiom erfüllt: Für alle Paare von Indizes j und k mit $j < k$ sind die zugehörigen Arrayelemente sortiert, d.h. es gilt $a[j] \le a[k]$. Das sind $N(N-1)/2$ Bedingungen (eine für jedes mögliche Paar $j < k$), die man alle prüfen muss, um die Korrektheit der Sortierung nachzuweisen. Zeigen Sie mit Hilfe der Eigenschaften der '≤' Relation, dass dieser Aufwand unnötig ist und man auch mit $(N-1)$ Tests auskommt. Welche Tests sollte man ausführen, und warum sind die übrigen dann automatisch erfüllt? *5 Punkte*

c) Implementieren Sie `insertion_sort()` und `quick_sort()` (mit geschickter Wahl des Pivots) so, dass die Gesamtzahl der Vergleiche von Arrayelementen gezählt und am Ende zurückgegeben wird (andere Vergleiche, z.B. von Schleifenindizes, werden nicht mitgezählt): *7 Punkte*

```
count1 = insertion_sort(a)
count2 = quick_sort(a)
```

Führen Sie diese Funktionen mit zufällig sortierten Arrays verschiedener Größe aus. Ein zufällig sortiertes Array erzeugen Sie folgendermaßen:

```
import random        # Python's Zufallszahlenmodul
N = ...              # Arraygröße
a = list(range(N))   # erzeuge sortiertes Array
random.shuffle(a)    # verwürfle das Array
```

Lassen Sie die Sortierfunktion für jedes N mehrmals laufen und bestimmen Sie die Mittelwerte c_1 (insertion sort) und c_2 (quick sort) der Zähler über die zufälligen Arrays der gleichen Größe. Benutzen Sie das Modul "`matplotlib`", um die Anzahl der Vergleiche in Abhängigkeit von N zu plotten[1]. Prüfen Sie die Aussage aus der Vorlesung, dass die Kurven für große N durch die Funk-

[1] Siehe https://matplotlib.org/gallery/lines_bars_and_markers/simple_plot.html. Um mehrere Kurven ins selbe Diagramm zu zeichnen, ruft man `ax.plot(...)` einfach mehrmals auf. `matplotlib` wird am einfachsten mittels Anaconda installiert.

tionen $c_1 \approx d_1 N^2$ (für insertion sort) bzw. $c_2 \approx d_2 N \log N$ (für quick sort) mit geeigneten Konstanten d_1, d_2 approximiert werden. Plotten Sie die Approximationsfunktionen ins gleiche Diagramm. (Zur Verbesserung der Fits können Sie auch die Funktionen $d_1\, N^2 + e_1\, N + f_1$ bzw. $d_2\, N \log N + e_2\, N + f_2$ verwenden, aber die Genauigkeit muss zur Lösung der Aufgabe nicht sonderlich hoch sein.)

Machen Sie das entsprechende Experiment außerdem mit bereits sortierten Arrays und vergleichen Sie die Ergebnisse und Kurven. Vergleichen Sie `quick_sort()` auf sortierten Arrays außerdem mit einer neuen Implementation `quick_sort_2()`, die das Pivot ungeschickt wählt.

d) Testen Sie nun, ob die Anzahl der Vergleiche aus c) eine gute Vorhersage der tatsächlichen Laufzeit erlaubt. Messen Sie dazu die Laufzeiten t_1 (insertion sort) und t_2 (quick sort) mit dem `timeit`-Modul (eine Anleitung finden Sie am Ende des Übungsblattes). Berechnen Sie das Verhältnis c_1/t_1 von insertion sort und überprüfen Sie, dass es für große *N* gegen eine Konstante konvergiert. Wiederholen Sie dasselbe für das Verhältnis c_2/t_2 von quick sort. *5 Punkte*

Der Code für die gesamte Aufgabe soll im File "`sortieren.py`" bzw. "`sortieren.ipynb`" abgegeben werden.

Aufgabe 2 – Zelluläre Automaten 20 Punkte

Algorithmen werden wesentlich durch die Menge der elementaren Operationen geprägt, aus denen sie aufgebaut sind. *Zelluläre Automaten* sind interessant, weil sie aus sehr einfachen elementaren Operationen erstaunlich komplexes Verhalten generieren. Das bekannteste Beispiel ist Conway's *Game of Life* (de.wikipedia.org/wiki/Conways_Spiel_des_Lebens).

Jeder zelluläre Automat besteht aus unabhängigen *Zellen*, die auf einem regelmäßigen Gitter angeordnet sind und *K* vorher festgelegte Zustände annehmen können. Die Zustände werden meist durch druckbare Zeichen wie " " (Leerzeichen), "*", "#" etc. symbolisiert. Der Automat entwickelt sich in diskreten Zeitschritten, und der Zustand jeder Zelle zum Zeitpunkt *T*+1 ergibt sich deterministisch aus den Zuständen dieser Zelle sowie ihrer unmittelbaren Nachbarn zum Zeitpunkt *T*. Die Menge der erlaubten Zustandsübergänge in Abhängigkeit der möglichen Nachbarschafts-Konstellationen wird als *Regel* des Automaten bezeichnet. Die Regel wird einmal festgelegt und bleibt dann für alle Zeitschritte gleich.

a) Wir beschränken uns in dieser Übung auf 1-dimensionale zelluläre Automaten. Diese kann man bequem als Strings einer vorher festgelegten Länge darstellen, sodass jedes Zeichen im String eine Zelle repräsentiert. Der neue Zustand einer Zelle *j* ergibt sich dann aus dem alten Zustand dieser Zelle und dem Zustand ihrer linken und rechten Nachbarzelle (Zellen *j*-1 und *j*+1), also jeweils aus Teilstrings der Länge 3. Deshalb kann man die Regel bequem in einem Python-Dictionary darstellen: Strings der Länge 3 bilden die Schlüssel, und die zugehörigen Werte geben den neuen Zustand der mittleren Zelle an, z.B.: *5 Punkte*

```
rule = {"   ": " ",    # drei Leerzeichen => Leerzeichen
        " * ": "*",    # Stern zwischen zwei Leerzeichen => Stern
        ...            # weitere Zustandsübergänge
       }
```

Implementieren Sie im File "`cellular_atomata.py`" bzw. "`cellular_atomata.ipynb`" eine Funktion

```
ca2 = ca_step(ca1, rule)
```

der der String `ca1` (Zustand zur Zeit *T*) sowie die Regel übergeben werden und die daraus den neuen String `ca2` (mit gleicher Länge wie `ca1`) als Zustand zur Zeit *T*+1 berechnet. Definieren Sie dabei eine geeignete Randbehandlung für die beiden Enden von `ca1`, also für die Zellen, die keinen linken bzw. rechten Nachbarn haben. Diese Funktion wird in b) und c) benutzt.

b) *Elementare* zelluläre Automaten haben nur zwei Zustände " " und "*". Wie viele verschiedene Regeln kann man daraus aufbauen? Manche dieser Regeln führen zu interessantem Verhalten. *5 Punkte*

Experimentieren Sie mit den Regeldefinitionen (indem Sie die Dictionaries variieren) und berechnen Sie jeweils die ersten 30 Zeitschritte, wobei der Anfangszustand immer ein String der Länge 71 ist, bei dem sich nur das Element 35 im Zustand "*" befindet, alle übrigen sind im Zustand " ". Geben Sie mit `print` den aktuellen Zustand nach jeder Iteration aus. Implementieren Sie vier Funktionen "`elementary1()`" bis "`elementary4()`" (die jeweils eine Regeldefinition und die Schleife enthalten) im File "`cellular_atomata.py`", die nette Muster erzeugen.

c) Implementieren Sie jetzt eine Funktion `ping_pong()`, deren Regel eine Art Ping-Pong-Spiel realisiert. Es gibt hier neben dem Zustand " " einen Zustand "#", der die beiden Spielfeldränder angibt, sowie einen Ball, der immer zwischen den beiden Rändern hin- und herfliegt und automatisch reflektiert wird. Der Ball wird durch zwei benachbarte Zellen symbolisiert, die die Zustände "o-" haben, wenn der Ball gerade nach links fliegt, und "-o" wenn er nach rechts fliegt. Der Anfangszustand ist der String "# o- #" (der Abstand zwischen den beiden "#" beträgt 12 Zeichen und soll unverändert bleiben). Implementieren Sie die zeitliche Entwicklung des Automaten als Endlosschleife (diese kann man in Python durch die Tastenkombination "Ctrl-C" abbrechen, wenn man lange genug zugeschaut hat). Um die Animation besser beobachten zu können, sollten Sie in jeder Iteration eine gewisse Wartezeit einbauen (Funktion "`sleep()`" aus dem Modul "`time`"). Beachten Sie, dass die Regel während des Ablaufs fix bleibt und *nicht* verändert werden darf (um z.B. die Flugrichtung des Balls umzukehren). *10 Punkte*

Verwendung des timeit-Moduls

Das `timeit`-Modul dient zur Messung der Laufzeit in Python. Der Code, dessen Laufzeit gemessen werden soll, wird der `Timeit`-Klasse bei der Initialisierung als (mehrzeiliger) String übergeben. In einem zweiten String kann man außerdem Initialisierungscode angeben, der zuvor ausgeführt werden muss, dessen Zeit aber nicht in die Messung eingehen soll. Für die Sortier-Aufgabe sieht dies so aus:

```
import timeit

code_to_be_measured = '''
quick_sort(a)
'''

initialisation = '''
N = 1000
a = list(range(N))
random.shuffle(a)
'''

t = timeit.Timer(code_to_be_measured, initialisation)
```

Die eigentliche Zeitmessung erfolgt dann mit der Methode `t.timeit(M)`, wobei M angibt, wie häufig das Programm ausgeführt werden soll, sodass man eine Durchschnittszeit berechnen kann:

```
M = 100
time = t.timeit(M)    # run 'code_to_be_measured' M times
print("average execution time:", time / M)
```

Bitte laden Sie Ihre Lösung bis zum 15.5.2020 um 14:00 Uhr auf Moodle hoch.