Institut für Theoretische Physik
der Universität zu Köln

Prof. Dr. Simon Trebst
Michael Becker, Peter Bröcker

Computerphysik

Übungsblatt 3

SS 2014

Website: http://www.thp.uni-koeln.de/trebst/Lectures/2014-CompPhys.shtml
Abgabedatum: Montag, 28. April 2014 vor Beginn der Vorlesung

11. N-dimensionale Wörterbücher *Programmiertechniken*

Ein Experiment oder eine Simulation besteht in den allermeisten Fällen darin, dass man ein vorgeschriebenes Messprotokoll für eine Reihe von Parametern durchführt. Dies kann zum Beispiel eine zeitaufgelöste Messreihe sein oder auch das Finden von Nullstellen eines komplexen Polynoms für eine vorgegebene Menge von Anfangspunkten, wie auf dem vorherigen Übungsblatt.

Um bei dieser Arbeit die Übersicht zu behalten, ist es von großem Nutzen passende Datenstrukturen zum Speichern **mehrdimensionaler Daten** zur Verfügung zu haben. Neben der bereits eingeführten **numpy** arrays bietet Python weitere sehr nützliche Strukturen, die wir in dieser Aufgabe kennenlernen wollen.

Der erste Datentyp, den wir vorstellen möchten, ist die `list`. Eine `list` nimmt Daten beliebigen Typs auf und weist diesen einen Index zu. Über den Index können wir dann wieder auf die gespeicherten Daten zugreifen. Zu den möglichen Typen zählt natürlich auch wieder die `list` selbst, so dass man auf diese Weise auch zweidimensionale Listen erstellen kann. Diese Funktionen sind in dem folgenden Codeausschnitt zusammengefasst:

```
l = []
l.append(3)
l.append('Hello World')
l.append(0.14)
l.append([])
l[3].append('Nested list')
print l[1]
```

Oft ist es allerdings alleine schon für die Lesbarkeit und Übersicht besser, wenn man die Elemente einer Menge nicht in der Reihenfolge in der sie hinzugefügt wurden, indiziert werden, sondern mit einem selbst gewählten Schlüssel. Für diesen Zwecke gibt es in Python den Datentyp `dict`, gekennzeichnet durch geschweifte Klammern. Das Hinzufügen von Elementen geschieht durch die Zuweisung eines Elements zu einem Schlüssel. Welche Schlüssel bereits vorhanden sind, lässt sich mit Hilfe der Funktion `keys()` ermitteln.

```
d = {}
d['a float'] = 3.14
d['a list'] = [7, 'hello world']
d['a list in a dict in a dict'] = {'a list in a dict':[42, 23, 9.29]}
```

```
print d.keys()
```

Zuguterletzt möchten wir noch einmal auf das Beispiel des numpy Arrays `ndarray` verweisen. Dieser ist besonders praktisch, wenn im Voraus bekannt ist, wie viele Daten für welche Anzahl Parameter aufgenommen werden sollen. Die gewünschte Größe wird durch ein Tupel mittels der Funktion `shape` übergeben. Anders als bei mehrdimensionalen Listen, die per Hand konstruiert wurden, stehen die Indizes eines Eintrags innerhalb einer einzelnen, eckigen Klammer:

```
import numpy as np
a = np.ndarray(shape=(9, 2, 5))
a[6,1,2] = 3.14
```

Weitere, ausfürliche Informationen inklusive vieler Beispiele erhalten Sie auf den Seiten der jeweiligen Datentypen, `list`, `dict` und `ndarray`.

12. Ordnung muss sein

5 Punkte

In dieser Aufgabe wollen wir die **Effizienz von Algorithmen** untersuchen. Dazu wollen wir verschiedene Algorithmen betrachten, eine Menge natürlicher Zahlen der Größe nach zu sortieren.

1. Als Eingabe werden wir Listen von zufällig erzeugten natürlichen Zahlen verwenden:

   ```
   import numpy as np
   Zahlen=np.random.randint(100,size=50) #50 Zufallszahlen ∈ [0 .. 99]
   ```

   Laden Sie sich den Sortier-Algorithmus **selection-sort.py** herunter und führen Sie diesen aus. Sie sehen als Ausgabe eine sortierte Liste von 50 Zahlen. Beschreiben Sie kurz, wie der Algorithmus funktioniert.

2. Der Code hat außerdem einen Zähler, der bestimmt, **wie viele Iterationsschritte** (Schleifendurchläufe) zur vollständigen Sortierung benötigt wurden.

   Ermitteln Sie die nötigen Iterationsschritte für Listen verschiedener Längen, indem Sie die Zeile

   ```
   N=[50]
   ```

   entsprechend verändern. Leiten Sie (auch mit Hilfe des Programmcodes) eine Formel ab, die die Anzahl der Durchläufe in Abhängigkeit von der Länge n der eingegebenen Liste abschätzt. Reduzieren Sie den erhaltenen Term auf seine **führende Ordnung**. Sehen Sie, wie Sie diese führende Ordnung auf den ersten Blick aus dem Programmcode ableiten können?

3. Die benötigte Laufzeit in Abhängigkeit der Länge der Eingabe nennt man die **Komplexität** eines Algorithmus. Jeder Algorithmus lässt sich in eine sogenannte Komplexitätsklasse einordnen, die mit z.B. $\mathcal{O}(n^4)$ notiert wird (auch O-Notation genannt).

   Der gegebene Programmcode verfügt auch über eine äußere Schleife, die je eine Sortierung von Zufallszahlenfolgen der Längen von 2 bis 500 vornimmt und die jeweils benötigte Laufzeit ermittelt. Anschließend wird die Laufzeit **graphisch** in Abhängigkeit von der Eingabelänge dargestellt.

   Kommentieren Sie dazu die betreffenden Zeilen ein, kommentieren Sie im Gegenzug die print-Anweisungen aus und starten Sie sie das Programm erneut.

   Können Sie die in 2. ermittelte Komplexitätsklasse bestätigen?

4. Betrachten Sie nun den Sortieralgorithmus **merge-sort.py**. Versuchen Sie nachzuvollziehen, wie dieser Algorithmus funktioniert. Nehmen Sie die auskommentierten print-Befehle zu Hilfe und überlegen Sie, warum er **merge sort** heißt.

   Aktivieren Sie wie in 3. die graphische Darstellung der Geschwindigkeit und bestimmen Sie die Komplexitätsklasse. (Tipp: Betrachten Sie den zweiten Graphen den der Code plottet.) Warum unterliegen diese Graphen einem **Rauschen**, im Gegensatz zum selection sort? Bei welchen Listenlängen n kommt es zu deutlich erkennbaren **Sprüngen**? Warum?

   Auf Kosten wovon wurde bei diesem Algorithmus der **Geschwindigkeitsvorteil** erkauft? (Tipp: Auf welche Kennzahl achten Sie üblicherweise beim Laptop-/Computerkauf neben der Prozessorgeschwindigkeit?)

13. Cache-Effekte *5 Punkte*

In der Vorlesung haben wir anhand einfacher physikalischer Überlegungen untersucht, wie schnell sich Informationen im Computer (maximal) ausbreiten können und was dies für die geometrische Anordnung etwa von CPU und Speicher bedeutet.

Illustrieren Sie diese Überlegungen für einen CPU Chip von 3cm Kantenlänge: Wie hoch darf die Taktfrequenz einer solchen CPU höchstens sein, damit die beiden am weitesten voneinander entferntesten Punkte des Chips während eines Taktes noch Informationen austauschen können, also Information von einem Punkt zum anderen und wieder zurück übertragen können? Bedenken Sie dabei, dass die Leiterbahnen auf einem CHIP nicht diagonal sondern in Manhattan-Form (rechtwinklig) verlegt sind.

Um diesen elementaren Flaschenhals zu vermeiden, werden in modernen Chips verschiedene **Speicher-Ebenen** (sogenannte **Cache-Level**) verbaut, die in unmittelbarer Nähe zu den Rechen-Cores auf dem Chip angesiedelt werden.

In dieser Aufgabe wollen wir untersuchen, ob wir mit einem recht elementaren Programm diese Speicherstruktur sichtbar machen können. Untersuchen Sie dazu den folgenden von uns verlinkten **C++ code**. Lesen Sie diesen stark kommentierten Quellcode, welchen Sie allein mit Ihren Python-Kenntnissen verstehen können sollten, und beschreiben Sie in Worten den Ablauf des Programms. Skizzieren Sie – ähnlich zu der in der Vorlesung verwandten Notation – einen Flussablauf des Programms.

In Abbildung 1 ist die Ausgabe des Programms graphisch aufgetragen. Beschreiben Sie den Verlauf der Datenkurve, ihre charakteristischen Merkmale und interpretieren Sie letztere. Können Sie sich erklären, warum in der Summenschleife des Programms der Speicher nicht sequentiell abgearbeitet wird?

Zusatzaufgabe (ohne Punkte).– Kompilieren Sie den Code auf einem Rechner Ihrer Wahl und lassen Sie ihn dort laufen. Unter Linux/Unix/OS X können Sie dies in der Shell mit den folgenden Kommandos machen:

```
g++ cache.cpp -o cache_test
./cache_test
```

Versuchen Sie, mit Hilfe der gewonnenen Daten die Größe der verschiedenen Cache-Speicher

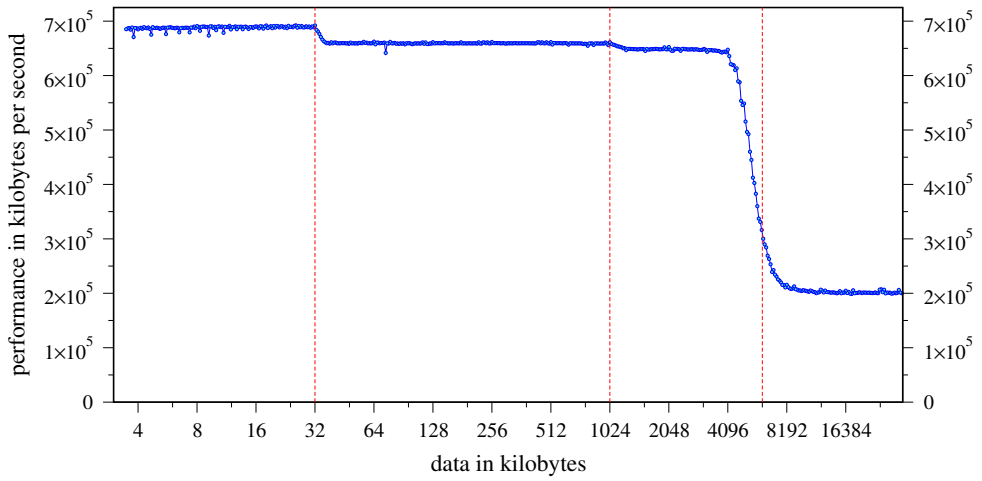

Abbildung 1: Graphische Darstellung der Ausgabe des obigen Programms

auf Ihrem System zu identifizieren. Anschließend vergleichen Sie (wenn möglich) mit den tatsächlichen Systemdaten. Auf einer Linux/Unix-Maschine können Sie letztere über den Befehl

```
cat /proc/cpuinfo
```

ausgeben.

Untersuchen Sie dabei auch, welche Ausmaße die CPU in Ihrem Rechner hat.

14. Flappy Bird

optionale Aufgabe – 6 Punkte
Abgabe am Montag, 5. Mai

In dieser Aufgabe wollen wir ein kleines **Arcade-Spiel** programmieren, in dem es darum geht, einen leicht behäbigen Vogel zum Fliegen zu bringen. Alles was Sie dafür brauchen werden, sind elementare Kenntnisse der klassischen Mechanik und Ihre bisherigen Python-Kenntnisse.

Die Spielidee sei folgende: Ein Vogel der Masse $m = 1$ fällt ballistisch im Schwerefeld der Erde. Durch einen Mausklick kann ihm ein Kraftstoß nach oben gegeben werden. Das Ziel ist, genau wie beim Hochhalten eines Fußballs, den Vogel nicht auf den Boden fallen zu lassen und dabei noch dem einen oder anderen am Horizont auftauchenden Hindernis auszuweichen.

Sie haben zwei unterschiedliche Möglichkeiten, die Spielphysik umzusetzen, die von der Umsetzung her vergleichbar komplex sind, allerdings die Handhabung des Spiels verändern.

- **Ansatz 1: Delta-Kraftstoß**
  Bei diesem Ansatz erzeugt der Mausklick einen momentanen Krafstoß von einer konstanten Stärke f. Die (zeitabhängige) Kraft, die auf den Vogel wirkt, ist damit

$$F(t) = -g + f \sum_i \delta(t - t_i), \quad t_i < t, \tag{1}$$

  wobei die t_i's die Zeitpunkte der bisher erfolgten Kraftstöße sind. Wenn wir damit die Newtonschen Bewegungsgleichungen integrieren, finden wir für die Höhe des Vogels

folgende Funktion

$$y(t) = -\frac{gt^2}{2} + f\sum_i (t - t_i) + y_0. \tag{2}$$

- **Ansatz 2: Angepasster Kraftstoß**
  Bei diesem Ansatz erzeugt der Mausklick auch einen momentanen Kraftstoß, welcher allerdings so angepasst ist, dass der Vogel direkt danach immer die gleiche Geschwindigkeit nach oben hat. Die physikalische Modellierung ist komplizierter, wir können aber folgende simple Überlegung nutzen: Die allgemeine Funktion für die Höhe des Vogels ist

  $$y(t) = -\frac{g(t-t_0)^2}{2} + v(t-t_0) + y_0, \tag{3}$$

  mit einer positiven Geschwindigkeit v. Bei jedem Klick wird $t \mapsto t_0$ und $y_0 \mapsto y(t_0)$ gesetzt. Damit erreichen wir, dass der Vogel nach jedem Klick mit Geschwindigkeit v fliegt.

Generell lässt sich sagen, dass der erste Ansatz physikalisch besser motiviert ist, der zweite der Spielmechanik allerdings zuträglicher. Das als iPhone-Spiel bekannt gewordene "Flappy Birds" verfolgt Ansatz 2.

Hindernisse
Unser Vogel fliegt nicht auf freier Strecke! Er muss über diverse Mauern fliegen, die ihm im Weg stehen. Da wir uns ins Bezugssystem des Vogels setzen, kommen die Hindernisse auf uns zu: In regelmäßigen Abständen bewegt sich ein Hindernis von rechts ins Bild hinein und nach links wieder hinaus. Die Höhe des Hindernisses soll zufällig im Bereich von 50 bis 150 Pixel liegen. Sobald die x-Koordinaten der Mauer und des Vogels in gewissen Grenzen übereinstimmen, müssen Sie die Flughöhe des Vogels überprüfen: Fliegt er zu tief, kollidiert er mit der Mauer und das Programm sollte abbrechen. Ebenso sollte ihr Programm abbrechen, sobald der Vogel auf den Boden fällt.

Tipps:

- Für den grafischen Teil der Aufgabe können Sie sich stark an der Aufgabe zum zellulären Automaten orientieren. Modellieren Sie die Bildlfläche als ein Array der Größe 400×400 pixel. Der Vogel kann in einer ersten Version als einfaches Quadrat oder ähnliches dargestellt werden. Es macht Sinn, den Vogel nicht mittig fliegen zu lassen, sondern eher am linken Rand. Somit bleibt mehr Zeit, sich auf das Überwinden des nächsten Hindernisses vorzubereiten.
- Der Hauptteil des Programms sollte eine Schleife sein, welche solange läuft, bis eine Abbruchbedingung erreicht ist (z.B. "Vogel ist auf den Boden gefallen"). In dieser Schleife sollte die Zeitvariable t stetig grösser werden. Die Geschwindigkeit des Programms hängt natürlich stark vom verwendeten System ab, insofern ist es sinnvoll, wenn Sie den Zeitfluss anpassbar machen, damit man das Spiel auf schnelleren System gegebenenfalls verlangsamen kann.
- Der Mausklick kann in Pyplot sehr leicht abgefangen werden. Sie müssen zunächst eine Funktion definieren, welche bei jedem Klick aufgerufen wird. Danach muss diese Funktion mit dem Input-Handler der matplotlib verknüpft werden. Ein einfacher Beispielcode:

```
import matplotlib.pyplot as plt
```

```
# Die aufzurufende Funktion
def onClick(event):
    print 'Klick!'

fig = plt.figure()
# Verknuepfe die Funktion mit dem Mausklick-Event
fig.canvas.mpl_connect('button_press_event', onClick)
plt.show()
```