Übung 2 Abgabe 3.5.2012

Aufgabe 1 – Vergleichen und Testen von Sortierverfahren 24 Punkte

a) Implementieren Sie die Algorithmen selection sort, merge sort und quick sort aus der Vorlesung (Abgabe als File "`sort.py`"). Fügen Sie dabei Zählvariablen ein, die bei jedem Aufruf die Anzahl der Vergleiche zwischen Arrayelementen während des Sortierens zählen:

```
>>> a1 = [...]                                  # the array to be sorted
>>> a2 = copy.deepcopy(a1)                      # a copy of the array
>>> a3 = copy.deepcopy(a1)                      # another copy of the array
>>> a1, comparisonCount1 = selectionSort(a1) # return sorted array
                                                # and comparison counter[1]
>>> a2, comparisonCount2 = mergeSort(a2)     # likewise
>>> a3, comparisonCount3 = quickSort(a3)     # likewise
```

Messen Sie für die drei Algorithmen die Anzahl der Vergleiche in Abhängigkeit von der Arraygröße und stellen Sie die Ergebnisse in einem Diagramm dar.[2] Suchen Sie geeignete Konstanten *a...i*, so dass die Kurven durch Funktionen

$$\begin{aligned} &a\,N^2 + b\,N + c && \text{(selection sort)}\\ &d\,N\log N + e\,N + f && \text{(merge sort)}\\ &g\,N\log N + h\,N + i && \text{(quick sort)} \end{aligned}$$

gut approximiert werden (die Fits müssen nicht sehr genau sein).

b) Die Laufzeit von Algorithmen kann in Python mit Hilfe des "timeit"-Moduls gemessen werden (siehe docs.python.org/library/timeit.html). Messen Sie mit diesem Modul die Laufzeit der drei Algorithmen (wieder in Anhängigkeit von der Arraygröße), stellen Sie die Ergebnisse graphisch dar und vergleichen Sie mit den Ergebnissen von a). Ist die funktionale Form aus a) – mit anderen Konstanten *a...i* – auch für diese Messung geeignet?

c) Wir haben in der Vorlesung drei Nachbedingungen für die Korrektheit eines Sortieralgorithmus behandelt: die Arrays müssen davor und danach die gleiche Größe haben, die gleichen Elemente enthalten, und das Ergebnis muss sortiert sein. Entwickeln Sie einen Algorithmus, der diese Bedingungen prüft (dieser Algorithmus muss nicht effizient sein), und begründen Sie dessen Vorgehen. Denken Sie dabei daran, dass Zahlen mehrfach vorkommen können ([3,2,3,1]→[1,1,2,3] ist ein Fehler!). Implementieren Sie den Algorithmus als Pythonfunktion

```
>>> correct = checkSorting(arrayBefore, arrayAfter)  # return True or False
```

und geben Sie die Implementation ebenfalls in "`sort.py`" ab.

Aufgabe 2 – Gewinnalgorithmus für ein Spiel 16 Punkte

Früher war folgendes Spiel recht beliebt: Auf dem Tisch liegen N=100 Streichhölzer. Alice und Bob nehmen abwechselnd mindestens ein und höchstens P=10 Streichhölzer weg. Wie viele sie in jedem Zug nehmen, steht ihnen frei, aber sie müssen auf jeden Fall ziehen. Wer das letzte Streichholz bekommt, gewinnt das Spiel. Alice beginnt.

a) Wie muss Alice ziehen, damit sie immer gewinnt?

---

[1] Eine Python-Funktion kann mehrere Werte zurückgeben, indem man "`return r1, r2`" schreibt.

[2] Die Wahl des Werkzeugs zur Erstellung von Diagrammen ist freigestellt. Wer z.B. MS Excel oder Matlab beherrscht, kann dies verwenden. Handarbeit auf Millimeterpapier ist ebenso zulässig. Der Standard unter Python ist matplotlib (download unter matplotlib.sourceforge.net), das ich deshalb in erster Linie empfehle. Ebenfalls gut und frei erhältlich ist Gnuplot (www.gnuplot.info). Man übergibt hier die zu zeichnenden Daten in Form von Textfiles, die man zuvor in Python erstellt, oder man benutzt das Modul gnuplot.py (gnuplot-py.sourceforge.net) für eine direkte Pythoneinbindung.

b) Verallgemeinern Sie die Strategie aus a) für beliebige P und N (mit P < N) und geben Sie Formeln an, wie Alice jeweils optimal ziehen muss. Beweisen Sie, dass Alice den Gewinn immer erzwingen kann, oder geben Sie Gegenbeispiele (also ungünstige Paare (P, N)) an, falls dies nicht möglich ist.

c) Beantworten Sie die Frage b) für das umgekehrte Spiel: Wer das letzte Streichholz ziehen muss, *verliert* das Spiel.

Bonusaufgabe – Dynamisches Array mit verringertem Speicherverbrauch 10 Punkte

In Übungsblatt 1 haben wir das dynamische Array betrachtet, das eine effiziente `append()`-Funktion unterstützt, die bei Bedarf den internen Speicher verdoppelt. Man kann gegen diese Implementation einwenden, dass mitunter ziemlich viel Speicher „verschwendet“ wird: unmittelbar nach einer Verdoppelung befinden sich (capacity/2 + 1) Elemente im Array, der übrige Speicher für (capacity/2 - 1) Elemente, also fast die Hälfte, ist unbenutzt.

Der Speichererbrauch kann verringert werden, indem man den internen Speicher auf P Teilarrays der Länge Q aufteilt, wobei capacity=P*Q gilt. Am besten wählt man P=Q, so dass capacity immer eine Quadratzahl sein muss (wir haben dann keine exakte Verdoppelung der Kapazitäten mehr, sondern z.B. die Folge 4, 9, 16, 36, 64, …, aber das spielt für die Effizienz keine Rolle). Um auf die einzelnen Teilarrays schnell zugreifen zu können, speichert man dieselben wiederum in einem Array, insgesamt erhält man also ein „Array of Arrays“. Stellt man sich die Teilarrays als Zeilen einer Matrix vor, wird die Matrix durch `append()` zeilenweise gefüllt. Den Speicherverbrauch minimiert man, indem man jedes Teilarray erst anlegt, wenn es tatsächlich benötigt wird.

a) Zeigen Sie, dass die Größe des unbenutzten Speichers jetzt < $2\sqrt{\text{capacity}}$ ist (statt < capacity/2 wie beim normalen dynamischen Array).

b) Geben Sie die Formeln an, mit denen man (1) die zum Index k gehörende Speicherzelle findet (dazu brauchen Sie den Modulo-Operator), (2) entscheidet, ob ein neues Teilarray begonnen werden muss, und (3) das nächste P berechnet, wenn die Kapazität der gesamten Matrix erschöpft ist.

c) Implementieren Sie die Datenstruktur (d.h. die Funktionen `__init__`, `__getitem__`, und `append`) und führen Sie ähnliche Experimente durch wie mit dem dynamischen Array in Übung 1.

d) Mit welchen Nachteilen wird der verringerte Speicherverbrauch erkauft?