Thema, Ziele: Sortieralgorithmen und Komplexitätstheorie

Aufgabe 1: Zeitmessung bei Fibonacci-Implementationen

Sie erhalten als Vorgabe die beiden Dateien `fiboit.c` und `fiborek.c`. Sie beinhalten eine iterative Implementation der Fibonacci-Zahlen, respektive eine rekursive Implementation, jeweils in der Programmiersprache C.

a) Implementieren Sie die beiden Programme je in C++ (sie müssen nicht objektorientiert sein).

b) Sie haben gesehen, dass die Komplexität der rekursiven Implementation exponentiell ist. Verifizieren Sie die Theorie, indem Sie Zeitmessungen für die Berechnung vornehmen.

Hinweis zu den Zeitmessungen:

Mit einfachen Mitteln ist eine genaue absolute Zeitmessung nicht durchführbar. Für unsere Zwecke genügt hier eine relative Zeitmessung, die genügend genau ist. Wählen Sie eine der folgenden Varianten:

Variante 1: Messung der Standardzeit

Die Funktion `clock()` aus `<ctime>` liefert die abgelaufene CPU-Zeit in Clockticks seit Programmstart. Wenn diese Grösse durch `CLOCKS_PER_SEC` geteilt wird, erhält man eine Zeit in Sekunden.

```
#include <ctime>
clock_t start = clock();
// do something
clock_t end = clock();
cout << "Ticks: " << end-start << endl;
cout << "Time: " << static_cast<double>(end-start) / CLOCKS_PER_SEC << " sec" << endl;
```

Variante 2: Native Zeitmessung von Linux (geht auch mit Cygwin)

```
#include <sys/resource.h>
#include <sys/types.h>
rusage tp;
double start;   // Startzeit in Millisekunden
double end;     // Endzeit
getrusage(RUSAGE_SELF, &tp);
start = static_cast<double>(tp.ru_utime.tv_sec) +
        static_cast<double>(tp.ru_utime.tv_usec)/1E6;
// do something
getrusage(RUSAGE_SELF, &tp);
end = static_cast<double>(tp.ru_utime.tv_sec) +
      static_cast<double>(tp.ru_utime.tv_usec)/1E6;
cout << "Dauer: " << end-start << " sec" << endl;
```

Aufgabe 2: Klasse für Stoppuhr

Implementieren Sie eine Klasse `StopWatch`, die sich bei der Gründung eines Objekts die Zeit merkt. Bei jedem Aufruf der Methode `elapsed()` wird die bis dahin abgelaufene Zeit in Sekunden zurückgegeben.

Beispiel:

```
StopWatch t;
double d = t.elapsed();
```

Aufgabe 3: Komplexität

Es gilt die Annahme, dass ein gewisses Programm jeden Abend exakt eine Stunde Rechenzeit bekommt. Sie haben herausgefunden, dass das Programm n=1'000'000 Datensätze verarbeiten kann. Nun wird ein neuer Rechner angeschafft, der 100 Mal schneller als der alte ist.
Wie viele Datensätze kann Ihr Programm nun in einer Stunde verarbeiten, wenn wir die folgende Zeitkomplexität mit den Konstanten k_i annehmen?

a) $k_1 \cdot n$

b) $k_2 \cdot n \cdot \log_{10} n$

c) $k_3 \cdot n^2$

d) $k_4 \cdot n^3$

e) $k_5 \cdot 10^n$

Hinweis: Verwenden Sie den Ansatz, dass der schnelle Rechner in einer Stunde gleich viel leistet, wie der langsame in 100 Stunden.

Aufgabe 4: Sortieralgorithmen

Implementieren Sie einen rekursiven Quicksort-Algorithmus, der für Teildaten mit weniger als M Elementen zu Insertionsort (Direktes Einfügen) übergeht.

Bestimmen Sie empirisch den optimalen Wert von M. Nehmen Sie einen Array mit ungefähr fünf Millionen zufällig erzeugten Ganzzahlen an und messen Sie, mit welchem M die Sortierung am schnellsten abgearbeitet wird. Betrachten Sie dabei sowohl das Sortieren der unsortierten als auch der bereits sortierten Liste.

Hinweis: Im Verzeichnis `./Vorgabe/QuickSortFast` finden Sie ein Eclipse-Projekt als Vorlage. Verwenden Sie dieses Projekt bei Bedarf.