Praktikum zur Vorlesung
GRUNDLAGEN DER INFORMATIK I
Studiengang Elektrotechnik
WS 02/03

AG Betriebssysteme FB3
Kirsten Berkenkötter

Ausgabe: 10.12./17.12.2002

Praktikum 4

1 Aufgabe 1: Sortieren mit Bubble-Sort

Einer der einfachsten Sortier-Algorithmen ist Bubble-Sort. In einer Schleife werden stets alle benachbarten Elemente durchlaufen und getauscht, falls das linke größer als das rechte ist. Dadurch wandert im *i-ten* Durchlauf der Schleife das *i-kleinste* Element nach vorn.

Ursprung	101	4	37	8	9	34	49	1	32	122	4	311	54	78	20
1. Durchl.	1	101	4	37	8	9	34	49	4	32	122	20	311	54	78
2. Durchl.	1	4	101	4	37	8	9	34	49	20	32	122	54	311	78
...															
Ende	1	4	4	8	9	20	32	34	37	49	54	78	101	122	311

Im Hauptprogramm soll ein Array mit 25 beliebigen Integer-Zahlen deklariert und initialisiert werden. Mit Hilfe der Funktion *bubbleSort* wird es dann der Größe nach aufsteigend sortiert. Um den Algorithmus zu überprüfen, soll das Array vor und nach dem Sortieren ausgegeben werden.

Die Funktion *bubbleSort* soll folgendes Aussehen haben:

*void bubbleSort(int *array, int laenge)*

Es werden zwei Schleifen benötigt, eine äußere, die den jeweiligen Durchlauf beim Sortieren angibt, und eine innere, welche die Elemente vergleicht und gegebenenfalls tauscht. Die innere Schleife startet stets beim letzten Element und läuft bis zu dem Bereich, der bereits getauscht wurde (im *i-ten* Durchlauf wird das *i-kleinste* Element ermittelt, d.h. in diesem Bereich ist das Array bereits sortiert!).

Beim Bubble-Sort werden stets $\frac{n*(n-1)}{2}$ Vergleiche benötigt, um das Array zu sortieren. Das Verfahren ist zwar einfach zu implementieren, aber auch nicht sehr effizient

2 Aufgabe 2: Sortieren mit Merge-Sort

Die Grundidee des Merge-Sort ist es, nicht ein großes Array zu sortieren, was sehr zeitaufwendig ist, sondern mehrere kleine. Die kleinen, sortierten Arrays werden dann wieder zu einem großem zusammengesetzt.

Im Hauptprogramm soll ein Array mit 25 beliebigen Integer-Zahlen deklariert und initialisiert werden. Mit Hilfe der Funktion *mergeSort* wird es dann der Größe nach aufsteigend sortiert. Um den Algorithmus zu überprüfen, soll das Array vor und nach dem Sortieren ausgegeben werden.

Die Funktion *mergeSort* soll folgendes Aussehen haben:

void mergeSort(int untereGrenze, int obereGrenze, int array[])

Dabei geben die ersten beiden Parameter an, welcher Bereich des Arrays sortiert werden soll und der dritte das zu sortierende Array. Die Angabe des Bereichs ist notwendig, da die Funktion sich selbst rekursiv aufrufen soll.

Der Merge-Sort funktioniert nun folgendermaßen: zunächst einmal wird die Mitte des zu sortierenden Bereichs ermittelt. Der Bereich von *untereGrenze* bis *mitte* ist das erste Teilarray und der Bereich von *mitte* bis *obereGrenze* das zweite. Für diese beiden Teilarrays wird *mergeSort* dann erneut aufgerufen, und zwar solange, bis *untereGrenze* und *obereGrenze* sich überschneiden. Zu guter Letzt müssen die Teilarrays wieder sortiert werden. Dies geschieht in der Funktion *mische.*

Die Funktion *mische* soll folgendermaßen aussehen:

void mische (int untereGrenze, int mitte, int obereGrenze, int array[])

Die drei Parameter geben an, wo die Grenzen des zu sortierenden Teilarrays und dessen Mitte liegen, der letzte ist das ursprünglich zu sortierende Array. Das erste Teilarray geht also von *untereGrenze* bis *mitte*, das zweite von *mitte* bis *obereGrenze.*

Beginnend mit dem ersten Element des ersten Teilarrays werden nun die beiden Teilarrays sortiert, d.h. die Elemente werden der Größe nach in ein Hilfsarray (lokale Variable der Funktion *mische*) geschrieben. Damit stehen die Teilarrays dann in sortierter Reihenfolge im Hilfsarray und können in das ursprüngliche Array übertragen werden.

Ähnlich wie Quick-Sort, das in der Vorlesung vorgestellt wurde, ist Merge-Sort ein sehr effektiver Sortier-Algorithmus. Im ungünstigsten Fall wäre Merge-Sort sogar besser als Quick-Sort, d.h. es würden weniger Operationen benötigen, um das Array zu sortieren. Allerdings benötigt Merge-Sort immer ein Hilfsarray, das genauso groß wie das zu sortierende ist, es kostet also Speicherplatz. Deshalb wird Quick-Sort in der Praxis häufiger eingesetzt.