Programmieren 1 Übung: Arrays: Sortieren

Klaus Kusche

Diejenigen Tätigkeiten, mit denen Computer im Schnitt die meiste Zeit verbringen, sind Suchen und Sortieren.

Es gibt in der Informatik 4 Gruppen von Sortierverfahren:

- Die *einfachen, aber langsamen Verfahren:*

  Bubblesort (Sortieren durch *Vertauschen*)
  Insertion Sort (Sortieren durch *Einfügen*)
  Selection Sort (Sortieren durch *Auswählen*)

  Sie brauchen im Schnitt in etwa n^2 *viele Vergleiche* und / oder Zuweisungen, um *n Werte* zu sortieren, und bestehen alle aus *zwei ineinander geschachtelten Schleifen*.

- Die *verbesserten einfachen, etwas schnelleren Verfahren:*

  Shellsort, **Combsort**, ...

  Ihr Rechenaufwand wächst mit einer kleineren Potenz als n^2, also z.B. mit $n^{1,5}$.

  Sie arbeiten meist ähnlich wie der Bubble- oder Insertion Sort,
  aber beseitigen zuerst einmal die grobe Unordnung (auf weite Distanzen)
  und bringen erst dann lokal benachbarte Elemente in die richtige Reihenfolge.

- Die *schnellen (aber auch ziemlich komplizierten) Sortierverfahren:*

  Quicksort, **Heapsort**, ...

  Sie brauchen zum Sortieren von n Zahlen im Mittel rund n*log(n) viele Vergleiche und Zuweisungen, was für große n sehr viel besser als n^2 ist (rechne nach!).

- Die *Verfahren für Spezialzwecke:*

  Mergesort, **Radixsort**, ...

  Der Mergesort ist primär für das Sortieren von Dateien statt von Daten in Arrays gedacht, und der Radixsort ist gut geeignet, wenn die Werte, nach denen sortiert wird, aus wenigen einzelnen Zeichen oder Ziffern bestehen (z.B. Autokennzeichen).

Wir wollen uns mit den einfachen Verfahren (2 geschachtelte Schleifen) beschäftigen.
Um **n** Zahlen mit Index **0** bis **n–1** zu sortieren, geht man wie folgt vor:

- **Selectionsort:** Schleife 1 macht **n–1** Durchläufe mit **pos** von **0** bis **n–2**.

  In jedem Durchlauf wählen wir das *kleinste Element* aus allen Elementen *hinter* der Stelle **pos** (dort stehen alle noch *unsortierten* Elemente) und *vertauschen* es mit dem Element an Stelle **pos**.

  - Dazu merken wir uns zuerst einmal den Wert des Elementes **pos** als vorläufig kleinsten Wert und die Position **pos** als Position des kleinsten Wertes (gibt es dahinter keinen kleineren Wert, so wird das Element an Stelle **pos** "mit sich selbst vertauscht").
  - Dann vergleichen wir in Schleife 2 jedes Element an Stelle **pos+1** bis **n–1** mit dem aktuellen kleinsten Wert. Ist es kleiner, speichern wir es als neuen kleinsten Wert und merken uns auch seine Position.
  - Schließlich kopieren wir das Element an Stelle **pos** an die gemerkte Position des kleinsten Elementes und den gespeicherten kleinsten Wert an die Stelle **pos**.

- **Insertionsort:** Schleife 1 macht **n–1** Durchläufe mit **pos** von **1** bis **n–1**.

  In jedem Durchlauf fügen wir das Element an Stelle **pos** an die *richtige Stelle* in die (schon *sortierten*) Elemente *davor* ein:

  - Wir merken uns zuerst einmal das Element an Stelle **pos** in einer Hilfsvariable **tmp**.
  - Dann gehen wir in Schleife 2 mit **i** von Stelle **pos** (incl.) *abwärts*, solange **i** größer **0** ist *und* das Element an Stelle **i–1** größer als **tmp** ist, und *verschieben* jedesmal das Element **i–1** in das Element **i**.
  - Sobald **i** an der richtigen Stelle ist (entweder gleich **0**, d.h. ganz vorne, oder hinter einem Element, das kleinergleich **tmp** ist), speichern wir **tmp** im Element **i**.

- **Bubblesort:** Schleife 1 macht **n–1** Durchläufe mit **pos** von **n–1** bis **1** *abwärts*.

  In jedem Durchlauf gehen wir in Schleife 2 mit **i** alle Elemente von **0** bis **pos–1** einmal durch (die Elemente ab Stelle **pos+1** sind nämlich schon richtig sortiert) und *vergleichen* das Element an Stelle **i** mit seinem *unmittelbaren Nachfolger* an Stelle **i+1**. Ist der Nachfolger kleiner, *vertauschen* wir die beiden Elemente.

  Hinweis: Es gibt zwei Optimierungen für Bubblesort (siehe z.B. Wikipedia):

  - Hat Schleife 2 *keine einzige Vertauschung* mehr gemacht, kann man sich die restlichen Durchläufe von Schleife 1 sparen und sofort *aufhören*.
  - Man muss mit Schleife 2 nicht bis **pos–1** gehen: Es reicht, wenn man bis zu jenem Index geht, bei dem im vorigen Durchlauf von Schleife 1 die *hinterste Vertauschung* in Schleife 2 stattfand.

  Im durchschnittlichen Fall (zufällig gemischte Elemente) bringen die Optimierungen aber nur wenig, Bubblesort ist trotzdem das im Schnitt *langsamste* Verfahren.

Und damit wir auch eine gute Vorstellung bekommen, wie diese Verfahren arbeiten, schreiben wir ein Programm mit schöner Grafik-Ausgabe.
Ich stelle dafür ein *halbfertiges Programm* zur Verfügung, das du ergänzen kannst.

- Wir sortieren ein Array von so vielen **`int`**-Werten, wie unser Fenster hoch ist (**`SDL_Y_SIZE`**).
- Als erstes *füllen* wir das Array mit den Zahlen von **`1`** bis **`SDL_Y_SIZE`**.
- Dann *mischen* wir die Zahlen gut:

  Wir ermitteln *zwei Zufallszahlen* **`x`** und **`y`** zwischen **`0`** und **`SDL_Y_SIZE-1`**
  (sorge dafür, dass die Zufallszahlen bei jedem Programmlauf *andere* sind!)
  und *vertauschen* die Array-Elemente an den Stellen **`x`** und **`y`**
  (erinnere dich: Was muss man tun, um zwei Array-Elemente zu vertauschen?).
  Das *wiederholen* wir 3000 Mal.

  (Es gibt *viel bessere* Verfahren, um eine zufällige Umordnung von **`n`** Zahlen zu erzeugen, aber das ist nicht das Thema dieser Übung.)
- Als nächstes *geben wir das Array aus*. Das soll eine *eigene Funktion* machen, der Code dafür steht schon im gegebenen Programm.
- Schließlich *sortieren* wir das Array mit einem der oben beschriebenen Verfahren. Auch das Sortieren soll in einer *eigenen Funktion* erledigt werden (Parameter: Array und Anzahl der Elemente; returnwert: Keiner).

  Der Sortier-Code soll am *Ende jedes einzelnen Durchlaufes von Schleife 1* (d.h. unten in der äußeren Schleife) wieder unsere *Ausgabe-Funktion* aufrufen, damit man dem Sortieren "bei der Arbeit zusehen" kann.

Zusatzaufgabe 1:

Versuche, bei allen Schleifen in deiner Sortierfunktion mit einem *Pointer* statt mit einem **`int`** durch das Array zu laufen!

Dein Programm sollte danach *keine Index-Zugriffe* mit **`[…]`** mehr enthalten.

Zusatzaufgabe 2:

Die Mutigen dürfen auch den **Shellsort** probieren:

Der Shellsort ist eine *Erweiterung des Insertionsorts*.
Die Idee ist, beim Einfügen des aktuellen Elementes in die schon sortierten Werte davor nicht unmittelbar benachbarte Werte zu vergleichen und zu verschieben,
sondern zuerst einmal die "grobe Unordnung" zu beheben,
indem man *Werte über große Distanzen vergleicht und verschiebt*.

Man macht also *mehrmals einen kompletten Insertionsort*
und *verringert* bei jedem Insertionsort den *Abstand zwischen den zu vergleichenden und zu vertauschenden Elementen*:

- Nimm dein Programm für den *Insertionsort als Ausgangsbasis*.
- Leg ein *Array für die Vergleichs- und Vertauschungsabstände* an
  und initialisiere es mit den Werten **385**, **124**, **40**, **13**, **4** und **1**
  (diese Werte wurden von der Wissenschaft als ziemlich optimal ermittelt).
- Bau um deine Schleife 1 *außen herum* eine *neue Schleife*, die dieses Array durchgeht
  und die Variable **`abst`** der Reihe nach mit den Werten aus dem Array belegt.
- Die bisherige Schleife 1 beginnt nicht bei **1**, sondern bei **`abst`** zu zählen.
- Die bisherige Schleife 2 zählt mit **`i`** beginnend von **`pos`** mit Schrittweite **`abst`**
  (statt **1**) abwärts, und zwar solange **`i`** größergleich **`abst`** ist
  und das Element an Stelle **`i-abst`** größer **`tmp`** ist.
- In Schleife 2 wird nicht das Element **`i-1`**,
  sondern das soeben verglichene Element von Stelle **`i-abst`**
  in das Element an Stelle **`i`** verschoben.

Der Shellsort sieht grafisch langsamer aus als die anderen Verfahren, aber das täuscht:
Wir *zeichnen* die Grafik beim Shellsort *viel öfter* neu als bei den anderen Verfahren,
deshalb braucht es länger (weil nach jedem Grafik-Update gewartet wird),
und man sieht mehr Zwischeschritte.
Aber *zwischen zwei Updates* der Grafik macht der Shellsort intern
sehr viel weniger Vergleiche und Vertauschungen als die anderen Verfahren
(zur Kontrolle kannst du ja bei allen Verfahren mitzählen und am Ende ausgeben,
wie oft zwei Array-Elemente verglichen bzw. verschoben werden).

Zusatzaufgabe 3:

Und die ganz Mutigen können auch den **Quicksort** probieren:

Quicksort ist ein *rekursives* Verfahren:
Die Sortier-Funktion *ruft sich selbst* immer wieder *für immer kleinere Teile des Arrays* auf, bis der zu sortierende Teil nur mehr aus *keinem oder einem einzigen* Element besteht (dann ist nichts mehr zu tun: Ein einzelnes Element für sich allein ist sortiert).

- Die *rekursive Sortierfunktion* hat *4 Parameter*:
  - Das *Array* und dessen *Größe*.
  - Den *Index* des *ersten* (linkesten) und des *letzten* (rechtesten) Elementes jenes Arrayteils, der von diesem Aufruf sortiert werden soll.

  Sie hat *keinen Returnwert*.

  - Wenn der linke Index *größergleich* dem rechten Index ist, dann hat der zu sortierende Arrayteil nur mehr 0 oder 1 Elemente. In diesem Fall kehrt die Funktion sofort zurück, ohne etwas zu tun.
  - Sonst ruft sie die *Partitionierungsfunktion* auf (siehe unten). Danach gibt sie das Array durch Aufruf der *Display-Funktion* aus und ruft sich selbst rekursiv *einmal für den linken Teil* (vom linken Rand bis eins vor dem Teilungselement) und *einmal für den rechten Teil* (von eins hinter dem Teilungselement bis zum rechten Rand) auf.
- Die vom Hauptprogramm aufgerufene *Sortier-Funktion* ruft nur die *rekursive Funktion für das gesamte Array* (alle Elemente von ganz links bis ganz rechts) auf.
- Die eigentliche Arbeit geschieht in der *Partitionierungsfunktion* (Aufteilfunktion): Sie hat die Aufgabe, sich *irgendeinen Wert* aus dem Array *auszusuchen*, und dann alle Elemente des Arrays, die *kleiner* als dieser Wert sind, *nach links* zu stellen, und alle Elemente, die *größergleich* sind, *nach rechts*.

  Wenn die Funktion zurückkehrt, besteht das Array daher *aus 3 Teilen*:
  - Dem *linken* Teil, der alle Elemente enthält, die *kleiner* als der Teilungswert sind.
  - Dem *Teilungselement selbst*: Seinen *Index* liefert die Funktion als *Returnwert*.
  - Dem *rechten* Teil mit allen Elementen, die *größergleich* dem Teilungswert sind.

  Aufgerufen wird die Funktion mit *3 Parametern*: Dem *Array* und den *Indices des ersten und des letzten Elementes* von dem Bereich, der aufgeteilt werden soll. Als *Returnwert* liefert sie den *Index der Grenze* zwischen den beiden Teilen.

  Für diese Funktion gibt es mehrere Vorgehensweisen.
  Die beste arbeitet mit *zwei Positionen*, die *von beiden Rändern* aufeinander zulaufen.
  Für unsere Zwecke reicht aber die *einfachste Variante* (deutlich langsamer):

  - Als *Teilungswert* nehmen wir das *Element in der Mitte* zwischen linkem und rechtem Rand: *Nimm es aus dem Array*, d.h. speichere es in einer Hilfsvariable und *ersetze* es im Array durch das Element von ganz rechts.
  - Weiters brauchst Du eine *Hilfsvariable* für die *aktuelle Grenze* zwischen linkem und rechtem Teil des Arrays.
    Am Anfang steht diese Grenze auf dem Element *ganz links*.

- Dann kommt eine *Schleife über alle Elemente*, die aufgeteilt werden sollen (**Achtung:**
  Sie geht nur bis zur Position **`rechts-1`**, denn das ganz rechte Element haben wir ja statt dem Teilungswert nach vor kopiert, es ist daher leer).
- *In der Schleife* vergleichen wir das aktuelle Element mit dem Teilungswert.
  - Ist es *kleiner*, *vertauschen* wir es mit dem Element an der Grenze und schieben dann die *Grenze eins nach rechts*.
  - Ist es größergleich, ist nichts zu tun.
- *Nach der Schleife* kopieren wir zuerst das *Element genau an der Grenze* in das *ganz rechte* (derzeit leere) Element unseres Array-Bereiches und dann den *Teilungswert* aus der Hilfsvariable in des *Element an der Grenze*.