Programmieren C: Arrays & Funktionen: 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).

Damit wir eine gute Vorstellung bekommen, wie diese Verfahren arbeiten,
schreiben wir ein Programm mit schöner Grafik-Ausgabe.
Ich stelle dafür ein *halbfertiges Programm* zur Verfügung, in dem zwei Funktionen fehlen:

- Eine *Funktion* **`fuelle`** zum Befüllen des Arrays. Sie bekommt ein **`int`**-*Array* und dessen *Größe* **`anzahl`** als *Parameter* und hat *keinen Returnwert*.

  Zuerst *befüllen* wir das Array *der Reihe nach* mit den Zahlen **`1`** … **`anzahl`** (*nicht* **`0`** … **`anzahl-1`** !).

  Dann *mischen* wir die Zahlen gut:
  Wir ermitteln *zwei Zufallszahlen* **`x`** und **`y`** zwischen **`0`** und **`anzahl-1`**
  (sorge dafür, dass die Zufallszahlen bei jedem Programmlauf *andere* sind!)
  und *vertauschen* die Array-Elemente an den Positionen **`x`** und **`y`** .
  Das *wiederholen* wir 3000 Mal.

  (Es gibt *viel bessere* Verfahren, um eine zufällige Umordnung von **`n`** Werten zu erzeugen, aber das ist nicht das Thema dieser Übung.)

  Als nächstes *zeigen wir das Array an*: Wir rufen *der Reihe nach* für jedes Element die bereits vorhandene Funktion **`balken`** auf, und zwar mit dem *Array*, dem *Index* des Elementes, und **`farbe_normal`** als *Farbe*.

- Eine *Funktion* **`sort`** zum Sortieren des Arrays. Sie bekommt ebenfalls ein **`int`**-*Array* und dessen *Größe* als *Parameter* und hat *keinen Returnwert*.

  Diese Funktion soll jeweils ein *Sortierverfahren* implementieren (siehe unten).

  Die Funktion **`sort`** darf die Elemente des Arrays *nicht direkt ändern*:
  Wir wollen ja jede Änderung grafisch anzeigen!

  Dafür gibt es bereits eine *Funktion* **`tausche`**. Sie wird mit dem *Array* und den *Indices der beiden zu vertauschenden Elemente* aufgerufen und vertauscht die beiden angegebenen Elemente sowohl im Array als auch auf dem Bildschirm.
  `sort` darf das Array daher *nur* durch Aufruf von **`tausche`** ändern.

  Auch das *Vergleichen von zwei Array-Elementen* soll *grafisch angezeigt* werden:
  Dafür gibt es bereits eine *Funktion* **`kleiner`**. Sie wird wie **`tausche`** mit dem *Array* und den *Indices der beiden zu vergleichenden Elemente* aufgerufen und liefert als *Returnwert* **`true`**, wenn *das erste Element kleiner* ist, und sonst **`false`**.
  `sort` soll *für alle Vergleiche* von Array-Elementen *nur diese Funktion* benutzen.

Wir wollen uns zuerst 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 die Position **pos** als *Position des vorläufig kleinsten Wertes* (findet der nächste Schritt keinen kleineren Wert dahinter, 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 Element. Ist es kleiner, merken wir uns seine Position als *neue Position des kleinsten Elementes*.
  - *Nach der Schleife vertauschen* wir das Element an Stelle **pos** mit dem im vorigen Schritt ermittelten kleinsten Element.

- **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, indem wir es immer wieder *mit seinem Vorgänger vertauschen*, bis es richtig steht (das geht effizienter, aber dann sieht man den Effekt nicht so gut).

  Dazu gehen wir in *Schleife 2* mit **i** beginnend von Stelle **pos** *abwärts*, solange **i** größer **0** ist *und* das Element an Stelle **i−1** *größer* als das an Stelle **i** ist, und *vertauschen* jedesmal das Element **i−1** mit dem 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 Position **0** bis **pos−1** einmal durch (alle Elemente ab Stelle **pos+1** sind nämlich *schon fertig!*) 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.

  Beide Optimierungen lassen sich *kombinieren*:
  - Man merkt sich in einer *Hilfsvariable* bei jeder *Vertauschung* das **i**.
  - *Vor* Schleife 2 initialisiert man diese Hilfsvariable mit **0**.
  - *Nach* Schleife 2 enthält die Variable die *Position der letzten Vertauschung*.
  - Anstatt **pos** bei jedem Durchlauf von Schleife 1 um eins runterzuzählen, speichert man die Hilfsvariable als *Grenze für den nächsten Durchlauf* in **pos**.

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

Zusatzaufgabe 1:

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 **`301, 132, 57, 23, 10, 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 mit **`pos`** 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 dem Element an Stelle **`i`** ist.

  In Schleife 2 wird daher nicht das Element Nummer **`i-1`**,
  sondern das soeben verglichene Element Nummer **`i-abst`**
  mit dem Element an Stelle **`i`** vertauscht.

Zusatzaufgabe 2:

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 immer sortiert).

- Die *rekursive Quicksort-Funktion* hat *3 Parameter*:
  - Das *Array*.
  - 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 ruft sie 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.
- Unsere vom Hauptprogramm aufgerufene *Sortier-Funktion* **`sort`**
  ruft nur die *rekursive Quicksort-Funktion für das gesamte Array*
  (alle Elemente von ganz links bis ganz rechts) auf.

Die eigentliche Arbeit geschieht in der *Partitionierungsfunktion* (Aufteilfunktion).
Aufgerufen wird diese Funktion mit *3 Parametern*: Dem *Array* und den *Indices des ersten und des letzten Elementes* von dem Bereich, der aufgeteilt werden soll.

Die Funktion hat die Aufgabe, die Werte im Array so *umzuordnen*,
dass das Array danach *in zwei Teile geteilt* ist,
wobei alle Werte im linken Teil *kleiner* sind als alle Werte im rechten Teil
(gleiche Werte links und rechts sind für Quicksort auch erlaubt).
Innerhalb eines jeden Teils dürfen die Werte in *beliebiger* Reihenfolge durcheinander sein.

Dazu *wählt* die Funktion *irgendeinen Wert* aus dem Array aus,
und stellt dann alle Elemente des Arrays, die *kleiner* als dieser Wert sind, *nach links*,
und alle Elemente, die *größergleich* sind, *nach rechts*.

Wenn die Funktion zurückkehrt, besteht das Array daher *aus 3 Bereichen*:
- Dem *linken* Teil, der lauter Elemente enthält, die *kleinergleich* dem Teilungswert sind.
- Dem *Teilungselement selbst*: Seinen *Index* liefert die Funktion als *Returnwert*,
  er ist die *Grenze* zwischen den beiden Teilen.
- Dem *rechten* Teil mit lauter Elementen, die *größergleich* dem Teilungswert sind.

Wie groß die beiden Teile werden (wo die Grenze ist), lässt sich *nicht vorhersagen*:
Das hängt davon ab, wie groß oder wie klein das beliebig gewählte Teilungselement ist,
und ergibt sich *erst beim Aufteilen* (im Extremfall kann ein Teil sogar komplett *leer* sein,
wenn man zum Aufteilen zufällig das größte oder das kleinste Element ausgewählt hat).

Für den Quicksort wären *möglichst gleichgroße Teile optimal*.
Dazu müsste für die Aufteilung *genau der mittelgroße Wert im Bereich* ausgewählt werden.
Dieser Wert lässt sich aber *vorab nicht effizient ermitteln*.

Es haben sich *mehrere Verfahren* verbreitet, um in möglichst vielen Fällen einen möglichst *brauchbaren Wert* zu bekommen (z.B. *"schau die 3 Elemente ganz vorne, ganz hinten und genau in der Mitte des Bereiches an, und nimm davon das wertmäßig mittlere"*).

Da viele Partitionierungsverfahren das *gewählte Teilungselement* temporär mit dem *letzten Element des Bereiches vertauschen*, verzichten wir auf ein kluges Auswahlverfahren und nehmen einfach gleich das *hinterste Element des Bereiches als Teilungselement* (bei unserem zufälligen Array-Inhalt ist das eine akzeptable Wahl, aber bei *annähernd vorsortierten* Array-Daten wäre das eine *ganz schlechte Idee*!) Wir *ersparen* uns damit den *ersten Schritt* *"wähle ein Teilungselement und bringe es ganz nach hinten"*.

Im *zweiten Schritt* müssen wir die Elemente davor mit diesem Element vergleichen und *aufteilen*. Dazu gibt es *zwei weit verbreitete Verfahren*:

- Das *deutlich bessere* (aber etwas kompliziertere) Verfahren verwendet *zwei Indices* **`i`** und **`j`**, wobei **`i`** am *linken* Rand des Bereiches beginnt und **`j`** *eins vor dem rechten* Rand (weil ganz am rechten Rand steht ja unser Teilungselement).
  Diese beiden Indices *laufen aufeinander zu*.

  Im Detail passiert in einer *Schleife* immer wieder Folgendes:

  - Suche *von links* nach rechts ein *zu großes* Element:
    Geh mit **`i`** immer wieder *eins nach rechts*, solange Du *nicht am rechten Rand* bist und das Element Nummer **`i`** *nicht größer als das Teilungselement* ist.
  - Suche *von rechts* nach links ein *zu kleines* Element:
    Geh mit **`j`** immer wieder *eins nach links*, solange **`j`** *rechts* von **`i`** ist und das Element Nummer **`j`** *nicht kleiner als das Teilungselement* ist.
  - Wenn **`i`** und **`j`** *zusammengestoßen* oder gar *überkreuzt* sind, *endet* die Schleife:
    Der Werte sind richtig aufgeteilt, und bei **`i`** beginnt der größere Teil.
  - Sonst stehen die Elemente **`i`** und **`j`** falsch und werden *miteinander vertauscht*.
    Dann sucht die Schleife die nächsten beiden zu vertauschenden Elemente.
- Das etwas *einfachere*, aber *klar langsamere* Verfahren verwendet auch *zwei Indices* **`i`** und **`j`**, wobei *beide am linken Rand* beginnen und nach rechts laufen:
  `i` ist das *nächste zu prüfende Element* (läuft also voraus)
  und **`j`** ist die *aktuelle Grenze* zwischen den kleinen und den großen Elementen.

  Die Schleife läuft in diesem Fall mit **`i`** *bis eins vor dem rechten Rand*
  (unser Teilungselement ganz rechts spielt nicht mit)
  und *vergleicht* jedesmal das Element Nummer **`i`** mit dem Teilungselement.

  - Ist das Element **`i`** *kleiner*, so wird Element **`i`** und **`j`** *vertauscht*
    und **`j`** danach *um eins erhöht* (denn der *kleine Teil* ist gerade größer geworden).
  - Ist es *größergleich* dem Teilungselement, braucht man *nichts tun*.

  Nach dieser Schleife ist **`j`** die Nummer des ersten Elementes des größeren Teils.

Der *dritte Schritt* ist wieder bei beiden Verfahren gleich:
Vertausche das *Teilungselement* (ganz hinten am rechten Rand)
mit dem *ersten Element des größeren Teils* (genau an der Grenze)
und gib diesen Grenz-Index als Returnwert zurück.

Zusatzaufgabe 3:

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

Dein Code sollte danach *keine Index-Zugriffe* mit **`[`**…**`]`** mehr enthalten (außer eventuell zur Pointer-Berechnung mit **`&(arr[`**…**`])`**).

Ich habe dafür ein *zweites Rahmenprogramm* bereitgestellt, in dem die Parameter der Funktionen **`balken`**, **`kleiner`** und **`tausche`** entsprechend angepasst sind: Das *Array* als Ganzes fällt *weg*, und *statt den Indices* der betroffenen Elemente werden *Pointer* auf diese Elemente übergeben.

Natürlich kann man zur Übung auch die Funktion **`fuelle`** auf Pointer umschreiben, dort bringt es aber nicht so viel.