Team: < 04 >, < Schäfer & Ahmad >

Aufgabenaufteilung:

1. <Aufgaben, für die Teammitglied 1 verantwortlich ist>, <Dateien, die komplett/zum Teil von Teammitglied 1 implementiert/bearbeitet wurden>
2. <Aufgaben, für die Teammitglied 2 verantwortlich ist>, <Dateien, die komplett/zum Teil von Teammitglied 2 implementiert/bearbeitet wurden>

Quellenangaben: <

http://de.wikipedia.org/wiki/Quicksort

>

Begründung für Codeübernahme: < Es wurde kein Code übernommen. >

Bearbeitungszeitraum: <

Entwurfszeit: 2 Stunden 05.11.14

>

Aktueller Stand: < Entwurf fertig >

Änderungen im Entwurf: < KEINE >

Allgemeines zum Quicksort-Algorithmus

Quicksort ist ein schneller und rekursiver Sortieralgorithmus, der nach dem Prinzip *Teile und herrsche* arbeitet. Er wurde ca. 1960 von C. Antony R. Hoare in seiner Grundform entwickelt und seitdem von vielen Forschern verbessert.

Der Algorithmus hat den Vorteil, dass er über eine sehr kurze innere Schleife verfügt (was die Ausführungsgeschwindigkeit stark erhöht) und ohne zusätzlichen Speicherplatz auskommt (abgesehen von dem für die Rekursion zusätzlichen benötigten Platz auf dem Aufruf-Stack).

Quicksort-Prinzip

Zunächst wird die zu sortierende Liste in zwei Teillisten („linke“ und „rechte“ Teilliste) getrennt. Dazu wählt Quicksort ein sogenanntes Pivotelement aus der Liste aus.

Das **Pivotelement** ist dasjenige Element einer Zahlenmenge, welches als Erstes von einem Algorithmus ausgewählt wird, um bestimmte Berechnungen durchzuführen.

Alle Elemente, die kleiner als das Pivotelement sind, kommen in die linke Teilliste, und alle, die größer sind, in die rechte Teilliste. Die Elemente, die gleich dem Pivotelement sind, können sich beliebig auf die Teillisten verteilen. Nach der Aufteilung sind die Elemente der linken Liste kleiner oder gleich den Elementen der rechten Liste.

Anschließend muss man also nur noch jede Teilliste in sich sortieren, um die Sortierung zu vollenden. Dazu wird der Quicksort-Algorithmus jeweils auf der linken und auf der rechten Teilliste ausgeführt. Jede Teilliste wird dann wieder in zwei Teillisten aufgeteilt und auf diese jeweils wieder der Quicksort-Algorithmus angewandt, und so fort. Diese Selbstaufrufe werden als Rekursion bezeichnet. Wenn eine Teilliste der Länge eins oder null auftritt, so ist diese bereits sortiert und es erfolgt der Abbruch der Rekursion.

Quicksort-Pseudocode

Die Implementierung der Teilung:
Die Elemente werden nicht in zusätzlichen Speicher kopiert, sondern nur innerhalb der Liste vertauscht.
Dafür wird ein Verfahren verwendet, das als *teilen* oder auch *partitionieren* bezeichnet wird.
Danach sind die beiden Teillisten gleich in der richtigen Position.
Sobald die Teillisten in sich sortiert wurden, ist die Sortierung der Gesamtliste beendet.

Der folgende Pseudocode illustriert die Arbeitsweise des Algorithmus, wobei `daten` die zu sortierende Liste mit *n* Elementen ist.
Bei jedem Aufruf von `quicksort()` gibt `links` den Index des ersten Elements in der Teilliste an und `rechts` den des letzten.
Beim ersten Aufruf (oberste Rekursionsebene) ist `links = 0` und `rechts = n-1`.

Die übergebene Liste wird dabei rekursiv immer weiter geteilt, bis sie nur noch einen Wert enthält.

```
funktion quicksort(links, rechts)
    falls links < rechts dann
        teiler := teile(links, rechts)
        quicksort(links, teiler-1)
        quicksort(teiler+1, rechts)
    ende
ende
```

Die folgende Implementierung der Funktion `teile` teilt das Feld so, dass sich das Pivotelement an seiner endgültigen Position befindet und alle kleineren Elemente davor stehen, während alle größeren danach kommen:

```
funktion teile(links, rechts)
    i := links
    j := rechts - 1
   // Pivot Element an linkester Stelle auswählen
   pivot := daten[links]

    wiederhole
        // Suche von links ein Element, welches größer als das
        //Pivotelement ist
        wiederhole solange daten[i] ≤ pivot und i < rechts
            i := i + 1
        ende

        // Suche von rechts ein Element, welches kleiner als das
        //Pivotelement ist
        wiederhole solange daten[j] ≥ pivot und j > links
             j := j - 1
        ende

        falls i < j dann
            tausche daten[i] mit daten[j]
        ende

      // solange i an j nicht vorbeigelaufen ist
    solange i < j

    // Tausche Pivotelement (daten[links]) mit neuer endgültiger
    //Position (daten[i])
    falls daten[j] < pivot dann
            tausche daten[j] mit daten[links]
    ende

    // gib die Position des Pivotelements zurück
    antworte j
ende
```

Implementieren sie die Funktion quickSortRekursiv.

Diese soll den oben beschriebenen Quicksortalgorithmus implementieren.

Als Besonderheit ist zu beachten, dass wenn in einer Teilliste weniger als 12 Elemente zu sortieren sind (während der Rekursion), diese an einen der in Aufgabe 2 implementierten Sortieralgorithmen weiter zu leiten und diese Elemente von ihm sortieren zu lassen sind (Verzweigung).

Außerdem muss das Pivotelement an linkester Stelle ausgewählt werden.

Implementieren sie die Funktion quickSortRandom.

Diese soll den oben beschriebenen Quicksortalgorithmus implementieren.

Als Besonderheit ist zu beachten, dass wenn in einer Teilliste weniger als 12 Elemente zu sortieren sind (während der Rekursion), diese an einen der in Aufgabe 2 implementierten Sortieralgorithmen weiter zu leiten und diese Elemente von ihm sortieren zu lassen sind (Verzweigung).

Außerdem muss das Pivotelement randomisiert ausgewählt werden.

Für beide Sortieralgorithmen gilt:

Zeitmessung:

Als erstes lassen sie den Algorithmus laufen und er protokolliert nur wie lange er zum sortieren gebraucht hat.

Aufwandberechnung:

Anschließend lassen sie ihn noch einmal laufen und er protokolliert den Aufwand für die Vergleiche und die Verschiebung.

Speicherung:

Der sortierte Array soll in die sortiert.dat Datei gespeichert werden.

Ausführung

Die Algorithmen sind 100-mal auszuführen, wenn möglich mit jeweils unterschiedlichen Zahlen:
- 80-mal Zufallszahlen
- 10-mal "best case"
- 10-mal "worst case"

Aus den Zeitmessungen sind anzugeben:

1. Anzahl eingelesener Elemente
2. Name des Algorithmus
3. Benötigte Zeit(Mittelwert), sowie maximal und minimal benötigteZeit.
4. Anzahl Vergleiche(Mittelwert), sowie maximale Anzahl und minimale Anzahl an Vergleichen.
5. Anzahl Verschiebungen(Mittelwert), sowie maximale Anzahl und minimale Anzahl an Verschiebungen.

Die Daten sind in einer Datei messung.log zu speichern. Zusätzlich, zur statistischen Aufbereitung, können auch *.csv dateien erzeugt werden.

Tipps:

- Das abspeichern von Zufallszahlen kann realisiert werden durch ein Zusatz Modul wie z.B. myIO
- Speichert die generierten Zufallszahlen in einer Erlang Datenstruktur, somit kann man das Modul weiter geben und verwenden, beim raus holen aus der zahlen.dat, kann man den Inhalt wieder in die Erlang Datenstruktur konvertieren.
- Ausführung: Dieses empfehlen wir als eigenes Modus zu implementieren.