Tutoriumsvorbereitung 4

Prof. Dr. Olaf Hellwich und Mitarbeiter

14.05.2018 - 18.05.2018

Inhaltsverzeichnis

0 Organisatorisches

Der abschließende Online-Test wird am **Montag, den 14.05.2018 um 10:00 Uhr** freigeschaltet und läuft bis **Sonntag, den 20.05.2018 um 23:59 Uhr**.

1 Sortieren

Da die binäre Suche ein sortiertes Array erfordert, stellt sich nun die Frage nach geeigneten Sortieralgorithmen.

1.1 Stabilität von Sortieralgorithmen

Definition:
Ein Sortierverfahren heißt stabil, wenn es die Reihenfolge der Datensätze, deren Sortierschlüssel gleich sind, bewahrt.
Betrachte folgende Menge von Elementen, die nach den Sortierschlüsseln sortiert werden soll:

2:Birne, 1:Clementine, 2:Kiwi, 2:Dattel 3:Apfel

- Ein stabiler Sort liefert: 1:Clementine, 2:Birne, 2:Kiwi, 2:Dattel, 3:Apfel
- Ein instabiler Sort liefert z.B.:
  - 1:Clementine, 2:Kiwi, 2:Birne, 2:Dattel, 3:Apfel
  - 1:Clementine, 2:Kiwi, 2:Dattel, 2:Birne, 3:Apfel
  - 1:Clementine, 2:Birne, 2:Dattel, 2:Kiwi, 3:Apfel
- Hinweis: Die Reihenfolge der Elemente wird allein von den Sortierschlüsseln (hier: den Zahlen vor den Früchten) bestimmt. Die alphabetische Reihenfolge ist irrelevant!
- Das in dieser Veranstaltung vorgestellte Selectionsort und allgemein Quicksort sortieren nur mit Glück stabil (je nach Konfiguration der Elemente), sind i.A. also nicht stabil (selbst ausprobieren!).
- Insertionsort sortiert (wenn nicht ungünstig implementiert) stabil.

1.2 In-place (in-situ) und Out-of-place (ex-situ)

- Wenn zusätzlicher Speicher benötigt wird, um z.B. Teilarrays nach einem Rekursionsschritt zu speichern, arbeitet der Sortieralgorithmus ex-situ.
- Wenn die Sortierung komplett im Originalarray passiert, arbeitet der Sortieralgorithmus in-situ.
- Viele Sortieralgorithmen lassen sich sowohl in-situ, als auch ex-situ implementieren.
- Selectionsort, Insertionsort und Quicksort können in-situ implementiert werden.

1.3 Selectionsort

Algorithmus

- Wir beginnen mit dem ersten Element (Index 0) des Arrays und tauschen dieses mit dem kleinsten Element des Arrays.
- Ist dieses erste Element bereits das Minimum des Arrays, bleibt es an seiner Stelle.
- Dann nehmen wir uns das zweite Element und tauschen dieses mit dem nächstkleineren Element.
- Das Element am Index 0 ist bereits abgearbeitet und wird bei der Suche nach dem kleinsten Element nicht mehr betrachtet.

- So durchlaufen wir das Array und durchsuchen jedes mal das Array nach dem kleinsten Element und in jedem Schritt wird das noch unsortierte Teilarray einen Slot kürzer.

Handsimulation

Aufgabe: Führen wir nun Selectionsort auf folgendem Array aus:

5	3	12	1	6	9

Lösung:

5	3	12	1	6	9
1	**3**	12	5	6	9
1	3	**12**	5	6	9
1	3	5	**12**	6	9
1	3	5	6	**12**	9
1	3	5	6	9	**12**
1	3	5	6	9	12

Legende:

- grün: kleinstes Element
- fett gedruckt: Element, mit dem getauscht wird
- grau: abgearbeitet

1.4 Insertionsort

Algorithmus

- Das Array wird Element für Element durchlaufen.
- Dabei wird jedes Element an seine korrekte Position im sortierten Teilarray (links) gesetzt.
- Dies geschieht, indem ein Vergleich mit dem linken Nachbarn im Array durchgeführt wird.
- Alle größeren Elemente müssen wiederum nach rechts verschoben werden.
- Der Algorithmus endet, sobald das Array vollständig durchlaufen wurde.

Handsimulation

Aufgabe: Führen wir nun Insertionsort auf folgendem Array aus:

5	3	12	4	10

Lösung:

5	3	12	4	10
3	**5**	12	4	10
3	5	**12**	4	10
3	**5**	4	12	10
3	4	5	12	10
3	4	5	**12**	10
3	4	**5**	10	12
3	4	5	10	12

Legende:

- grün: aktuelles Element
- fett gedruckt: Element, mit dem verglichen und ggf. getauscht wird
- grau: Bereich des sortierten Teilarrays

Umsetzung in Java

Herleitung:

- Rückgabetyp: void (Es gibt keine sinnvolle Rückgabe)
- Parameter: Es wird entweder nichts übergeben, da das Array als Attribut vorliegt, oder das Array übergeben
- Lokale Variablen: Wir brauchen eine Variable, die anzeigt, welches Element als nächstes nach links verschoben wird, bis es richtig einsortiert ist. Eine weitere, die bei den vorgenommenen Linksverschiebungen mitzählt (sich den Index merkt). Dazu eine Hilfsvariable für die Tauschaktionen.
- Zwei ineinander verschachtelte Schleifen: die innere zur Verschiebung nach links bis das Element passend eingeordnet ist, die äußere, um dies für jedes Element zu tun.

Lösung:

```
1  public class Insertionsort {
2     public static void sort(Comparable[] f) {
3
4        for (int nextToInsert = 0; nextToInsert < f.length; nextToInsert++) {
5
6           int cursor = nextToInsert;
7           while (f[cursor] < f[cursor-1] && cursor > 0) { // falls kleiner als
                linker Nachbar dann tauschen
8              Comparable tmp = f[cursor-1];
9              f[cursor-1] = f[cursor];
10             f[cursor] = tmp;
11             cursor--; // Positionsaktualisierung
12          }
13       }
14    }
15 }
```

这里应该没有 VL 的方法好，因为如果 cursor - 1 < 0 此时程序会崩掉吧

1.5 Quicksort

1.5.1 Motivation

- Selectionsort ist für viele Anwendungen ausreichend, hat im Average-Case aber eine quadratische Komplexität.
- Selectionsort hat ebenfalls im Best- und Worst-Case eine Komplexität $O(n^2)$.
- D.h. im Mittel braucht das Verfahren etwa genauso lange wie im schlechtesten Fall, das geht besser!
- Man wünscht sich einen Algorithmus, dessen Laufzeit selbst bei zunehmend vielen zu verarbeitenden Daten nur langsam ansteigt; Quicksort ist so ein Algorithmus.

1.5.2 Quicksort

1.5.3 Allgemeine Eigenschaften des Quicksort-Algorithmus

Quicksort

- ist ein rekursiver Sortieralgorithmus, basierend auf dem *Teile und Herrsche* Prinzip (wie auch Mergesort).
- ist i.A. nicht stabil (siehe Stabilität von Sortieralgorithmen).
- ist im Average-Case einer der schnellsten Sortieralgorithmen.
- ist in der Effizienz sehr von der Wahl des Pivot-Elements abhängig.

Das *Teile und Herrsche*-Prinzip (divide et impera) ist ein weit verbreitetes Konzept zur Lösung von Problemen in der Informatik. Der Lösungsaufwand vieler Probleme (insbesondere Suchprobleme) sinkt bei der Aufteilung des ursprünglichen Problems in mehrere kleinere Teilprobleme, die dann wiederum rekursiv aufgeteilt werden. Letztlich wird die Lösung des Gesamtproblems aus den vielen Lösungen der kleineren Probleme konstruiert. Quicksort ist sowohl in-situ als auch ex-situ anwendbar, im Folgenden werden beide Methoden vorgestellt.

1.5.4 Funktionsweise ex-situ Quicksort

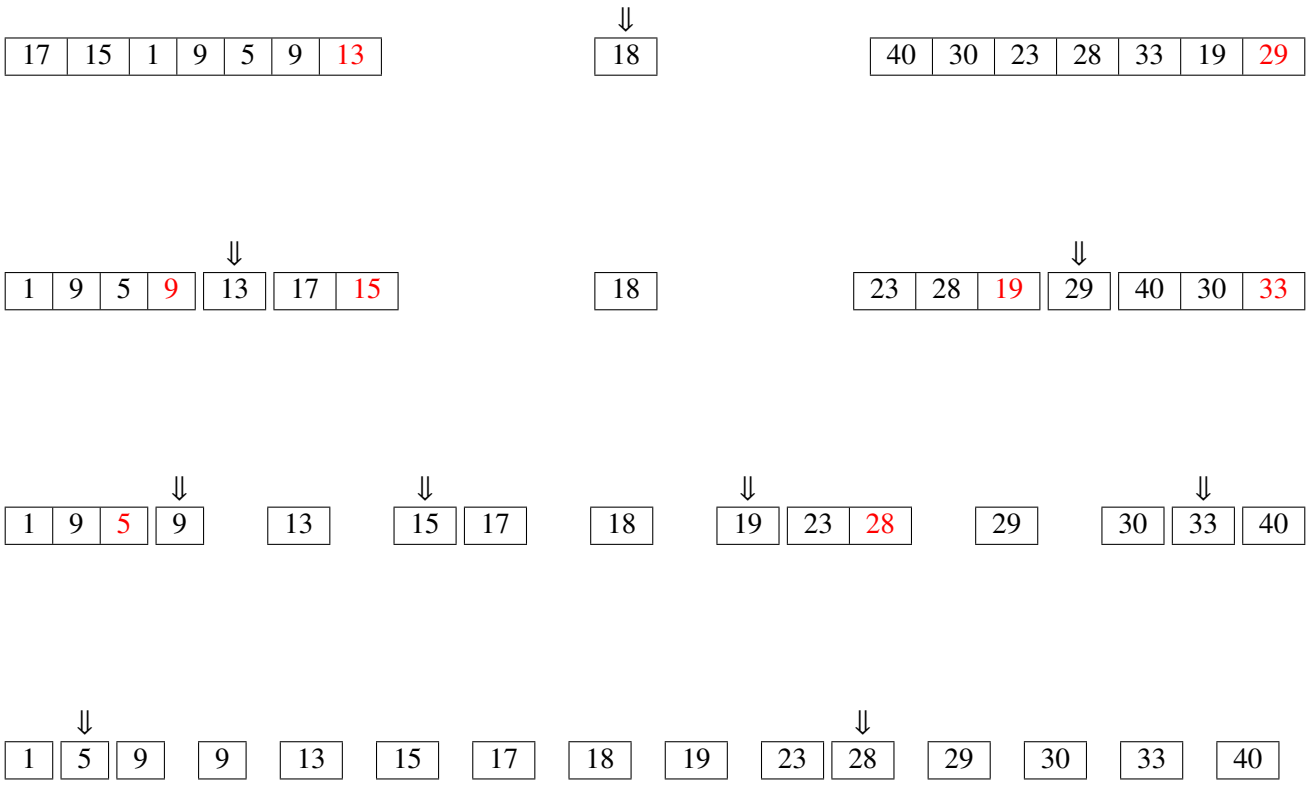
- Wähle aus den im Array befindlichen Elementen ein so genanntes **Pivot-Element**.
- Alle Elemente im Array, die kleiner sind als der Wert dieses Pivot-Element, kommen (zunächst unsortiert) in ein erstes Teilarray.
- Alle Elemente im Array, die größer sind als das Pivot-Element, kommen ins zweite Teilarray.
- Für beide Teilarrays wird nun ebenfalls wieder Schritt 1 angewandt. Dabei können wir bedenken, dass alle Elemente des ersten Teilarrays garantiert kleiner (oder gleich) sind als alle Elemente des zweiten Teilarrays.
- Die Rekursion muss terminieren, wenn ein Teilarray nur noch aus einem Element besteht.

Bestenfalls ist das Pivot-Element (*in jedem Rekursionsschritt*!) *so* zu wählen, dass beide Teilarrays in jedem Rekursionsschritt exakt gleich lang sind.

1.5.5 Handsimulation ex-situ

Im Folgenden wird willkürlich stets das letzte Element des Arrays als Pivot-Element benutzt.
Soll als Pivotelement ein beliebiges anderes Element gewählt werden, sollte dies jeweils als erster Schritt mit dem Element am Ende des Arrays getauscht werden, so dass das Pivotelement immer am Arrayende steht.
Wir betrachten folgende Zahlenfolge:

$$40, 17, 30, 23, 15, 28, 33, 1, 9, 5, 19, 29, 9, 13, 18$$



Fertig! Das sortierte Array ist also:

1	5	9	9	13	15	17	18	19	23	28	29	30	33	40

1.5.6 Funktionsweise in-situ Quicksort

- wähle: letztes Element = Pivot-Element
- soll als Pivotelement ein beliebiges anderes Element gewählt werden, sollte dies jeweils als erster Schritt mit dem Element am Ende des Arrays getauscht werden, so dass das Pivotelement immer am Arrayende steht
- definiere zwei Zähler: Laufindex i, Grenzindex g
- i,g zeigen initial auf erstes Element
- wenn array[i] < array[Pivot] Tausche array[i] und array[g], i++, g++
- wenn array[i] > array[Pivot] kein Tausch, i++
- wenn das gesamte Array durchlaufen wurde, tausche array[g] und array[Pivot] miteinander
- definiere: linkes Teilarray = alle Elemente links von array[g], rechtes Teilarray = alle Elemente rechts von array[g]
- wende diesen Algorithmus rekursiv auf jeweils beide Teilarrays an
- terminiere, wenn nur noch ein Element im (Teil-)Array vorhanden ist

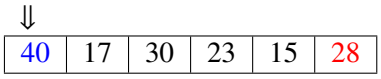
1.5.7 Handsimulation in-situ

Betrachte folgende Zahlenfolge:

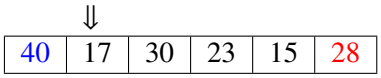
$$40, 17, 30, 23, 15, 28$$

Legende:

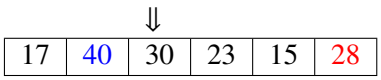
- Pfeil: Laufindex i
- rot: Pivot-Element
- blau: aktuelles „Grenzelement “ mit Index g
- fett: alle Elemente zwischen zwei fett gedruckten Elementen bilden zusammen ein Teilarray, dass noch sortiert werden muss



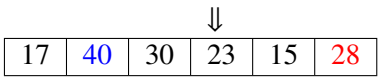
40>28 keine Tauschoperation, i++



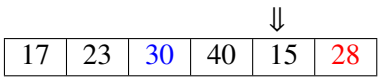
17<28 tausche 17 mit 40, g++, i++



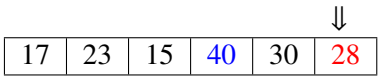
30>28 keine Tauschoperation, i++



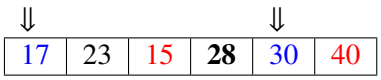
23<28 tausche 23 mit 40, g++, i++

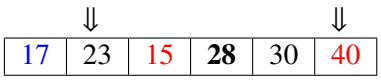


15<28 tausche 15 mit 30, g++, i++

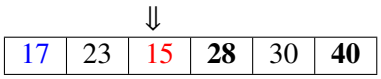


tausche Pivotelement 28 mit Grenzelement 40, dann Rekursionsabstieg

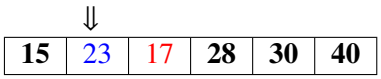


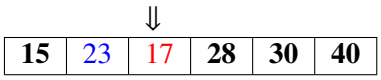


Rekursionsabstieg beim rechten Teilarray

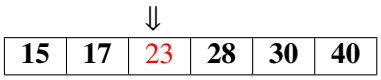


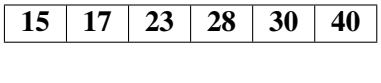
Rekursionsabstieg beim linken Teilarray





Rekursionsabstieg





Fertig

Das fertig sortierte Array ist also: