Sortieralgorithmen

Weiterführende Literatur:

- *Wikipedia-Artikel „Sortierverfahren“*

Klassifizierung der Sortieralgorithmen

Interne vs. externe Verfahren[1]

Bei *internen Sortierverfahren* ist stets ein *direkter Zugriff* auf *alle* zu sortierenden Elemente notwendig. Alle Elemente müssen *gleichzeitig* im Hauptspeicher liegen.

interner Sortierverfahren
direkter Zugriff
alle

Bei *externen Sortierverfahren* ist der Zugriff auf einen *Teil* der zu sortierenden Elemente beschränkt. Nur ein Teil der Daten muss gleichzeitig im *Hauptspeicher* liegen. Dieses Verfahren eignet sich für Sortierung von *Massendaten* auf *externen Speichermedien*.[2]

externen Sortierverfahren
Teil
Massendaten
externen Speichermedien

Vergleichsbasierte vs. Nicht-Vergleichsbasierte Verfahren[3]

Beim vergleichsbasierten Sortieren *vergleicht* der Algorithmus mehrfach jeweils *zwei Elemente* miteinander. Die Elementen werden aufgrund aufgrund ihrer *relativen Position* vertauscht. Beispiele: `QuickSort`, `MergeSort`

vergleicht
zwei Elemente

Beim nicht-vergleichsbasiertes Sortieren benötigt der Algorithmus *keinen direkten Vergleich* zwischen zwei Elementen, er *zählt* stattdessen die Werte oder betrachtet *„einzelne Stellen“* Beispiele: `CountingSort`, `RadixSort`

keinen direkten Vergleich
zählt
„einzelne Stellen“

Stabil vs. Instabil[4]

- stabiles Sortierverfahren:
  → Sortierverfahren, welches die Eingabereihenfolge von Elementen mit *gleichem Wert* beim Sortieren *bewahrt*

  Insbesondere dann wichtig, wenn hintereinander nach *mehreren Kriterien* sortiert wird.

In-Place vs. Out-Of-Place

- in-place (in situ)
  - Speicherverbrauch unabhängig von Eingabegröße
    → braucht nur eine konstante Menge an zusätzlichem Speicher
    → überschreibt im Allgemeinen die Eingabe- mit den Ausgabedaten
- out-of-place (ex situ)

[1] *Algorithmen und Datenstrukturen: Tafelübung 11, WS 2018/19*, Seite 34.
[2] Saake und Sattler, *Algorithmen und Datenstrukturen*, Seite 124.
[3] *Algorithmen und Datenstrukturen: Tafelübung 11, WS 2018/19*, Seite 35.
[4] *Algorithmen und Datenstrukturen: Tafelübung 11, WS 2018/19*, Seite 36.

- Speicherverbrauch abhängig von Eingabegröße
  $\rightarrow$ Speicherverbrauch steigt mit Anzahl der zu sortierenden Elemente

Achtung: Aufrufstapel *Rekursive Algorithmen*, deren Aufruftiefe von der Eingabegröße abhängt, arbeiten genaugenommen *out-of-place*, denn für die Funktionsschachteln auf dem Aufrufestapel wird Speicherplatz benötigt. Manchmal bezeichnet man aber auch solche Algorithmen mit einem Speicherverbrauch von $\mathcal{O}(log(n))$ als in-place.

Laufzeitkomplexität

Algorithmen und Datenstrukturen: Tafelübung 11, WS 2018/19, Seite 38

- für die Laufzeitkomplexität unterscheidet man verschiedene Fälle:
  - Best-Case
  - Average-Case
  - Worst-Case
- adaptive Sortierverfahren:
  - Laufzeit abhängig vom *Grad der Vorsortierung*
    $\rightarrow$ *schneller*, wenn Eingabe schon *„einigermaßen" sortiert* ist
    $\rightarrow$ Laufzeit in *Best-Case* und *Worst-Case unterschiedlich*
- *untere Schranken* für die Laufzeit (n: Anzahl an Elementen):
  - vergleichsbasiertes Sortieren: nicht besser möglich als $\mathcal{O}(log(n))$
  - nicht-vergleichsbasiertes Sortieren: lineare Laufzeit möglich

Vergleich der Sortieralgorithmen

Laufzeit[5]

	Best	Average	Worst
Binary Tree Sort	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n^2)$
Bubblesort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Vergleiche	$n-1$	$\sim\frac{n^2}{2}$	$\sim\frac{n^2}{2}$
Kopieraktionen	0	$\sim\frac{n^2}{4}$	$\sim\frac{n^2}{2}$
Heapsort	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$
Insertionsort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Vergleiche	$n-1$	$\sim\frac{n^2}{4}$	$\sim\frac{n^2}{2}$
Kopieraktionen	0	$\sim\frac{n^2}{4}$	$\sim\frac{n^2}{2}$
Mergesort	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$
Quicksort	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n^2)$
Selectionsort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Vergleiche	$\sim\frac{n^2}{2}$	$\sim\frac{n^2}{2}$	$\sim\frac{n^2}{2}$
Kopieraktionen	0	$3n$	$3n$

Implementation

	Kontrollstrukturen	Hilfsvariablen	Hilfsmethoden
Bubblesort	do while, for (bis vorletztes), if	t = getauscht	tausche
Insertionsort	for (ab zweitem), while	m = merker	
Selectionsort	while, for (ab zweitem)	m = markierung	tausche

```
3  /**
4   * Eine Sammlung von Sortier-Algorithmen, die in einer Klasse zusammengefügt
5   * sind. Sie enhalten keine Kommentare. Kommentierte Code-Beispiele gibt es in
6   * den einzelen Klassen, die auf einen Sortieralgorithmus spezialisiert sind. Es
7   * werden Ein-Buchstaben-Variablennamen verwendet, damit man die Code-Beispiele
8   * per Hand flüssig schreiben kann.
9   *
10  * <ul>
11  * <li>a: array (Ein Feld mit Integer-Zahlen, das sortiert werden soll)
12  * <li>i1, i2: index1, index2 (Eine Index-Nummer in einem Zahlen-Feld)
13  * <li>m: merker oder markierung (Eine Zahl wird in einer Hilfsvariablen
14  * bespeichert)
15  * </ul>
16  */
17 public class Sammlung {
```

[5] *Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen 2*, Seite 35.

```
18
19      /**
20       * Vertausche zwei Zahlen im einem Zahlen-Feld. Im Englischen heißt die Methode
21       * auch oft „swap".
22       *
23       * @param a  Ein Feld mit Zahlen.
24       * @param i1 Die Index-Nummer der ersten Zahl.
25       * @param i2 Die Index-Nummer der zweiten Zahl.
26       */
27      static void vertausche(int[] a, int i1, int i2) {
28        int tmp = a[i1];
29        a[i1] = a[i2];
30        a[i2] = tmp;
31      }
32
33      /**
34       * Sortiere mit Hilfe des Bubblesort-Algorithmus.
35       *
36       * <h2>Weitere Abkürzungen
37       *
38       * <ul>
39       * <li>t: getauscht
40       * </ul>
41       *
42       * @param a Ein Feld mit Zahlen, das sortiert werden soll.
43       */
44      static void bubblesort(int[] a) {
45        boolean t;
46        do {
47          t = false;
48          for (int i = 0; i < a.length - 1; i++) {
49            if (a[i] > a[i + 1]) {
50              vertausche(a, i, i + 1);
51              t = true;
52            }
53          }
54        } while (t);
55      }
56
57      static void insertionsort(int[] a) {
58        for (int i = 1; i < a.length; i++) {
59          int m = a[i];
60          int j = i;
61          while (j >= 1 && a[j - 1] > m) {
62            a[j] = a[j - 1];
63            j--;
64          }
65          a[j] = m;
66        }
67      }
68
69      static void selectionsort(int[] a) {
70        int m = a.length - 1;
71        while (m >= 0) {
72          int max = 0;
73          for (int i = 1; i <= m; i++) {
74            if (a[i] > a[max]) {
75              max = i;
76            }
77          }
78          vertausche(a, m, max);
79          m--;
```

```
80        }
81      }
82
83      /**
84       * Hilfsmethode zum Zerlegen der Zahlen-Folge für den Quicksort-Algorithmus.
85       * Diese Methode heißt im Englischen auch oft „partition".
86       *
87       * <h2>Weitere Abkürzungen
88       *
89       * <ul>
90       * <li>pn: pivotPositionNeu (Die Index-Nummer des neuen Pivot-Elements)
91       * <li>pw: pivotWert (Der Wert des (alten) Pivot-Elements)
92       * </ul>
93       *
94       * @param a Ein Feld mit Zahlen, das sortiert werden soll.
95       * @param u Die Index-Nummer der unteren Grenze.
96       * @param o Die Index-Nummer der oberen Grenze.
97       * @param p Die Index-Nummer der Pivot-Position.
98       *
99       * @return Die Index-Nummer der neuen Pivot-Position
100      */
101     static int quicksortPartition(int[] a, int u, int o, int p) {
102       int pn = u;
103       int pw = a[p];
104       vertausche(a, p, o);
105       for (int i = u; i < o; i++) {
106         if (a[i] <= pw) {
107           vertausche(a, pn++, i);
108         }
109       }
110       vertausche(a, o, pn);
111       return pn;
112     }
113
114     static void quicksort(int[] a, int u, int o) {
115       int p = (u + o) / 2;
116       if (o > u) {
117         int pn = quicksortPartition(a, u, o, p);
118         quicksort(a, u, pn - 1);
119         quicksort(a, pn + 1, o);
120       }
121     }
122
123     static void quicksort(int[] a) {
124       quicksort(a, 0, a.length - 1);
125     }
126
127     /**
128      * Eine Hilfsmethode für rekursives Sortieren durch Mischen des
129      * Mergesort-Algorithmus.
130      *
131      * <h2>Weitere Abkürzungen
132      *
133      * <ul>
134      * <li>i: Index links
135      * <li>j: Index rechts
136      * <li>k: Index
137      * <li>m: Index-Nummer der Mitte
138      * </ul>
139      *
140      * @param a Ein Feld mit Zahlen, das sortiert werden soll.
141      * @param l Die linke Grenze.Die Index-Nummer, ab der das Zahlen-Feld sortiert
```

```
142       *          werden soll.
143       * @param r Die rechte Grenze. Die Index-Nummer, bis zu der das Zahlen-Feld
144       *          sortiert werden soll.
145       * @param h Eine Hilfsfeld, in dem die Zahlen temporär zwischengespeichert
146       *          werden.
147       */
148     private static void mergesort(int[] a, int l, int r, int[] h) {
149       if (r <= l)
150         return;
151       int i, j, k;
152       int m = (r + l) / 2;
153       mergesort(a, l, m, h);
154       mergesort(a, m + 1, r, h);
155       for (k = l; k <= m; k++) {
156         h[k] = a[k];
157       }
158       for (k = m; k < r; k++) {
159         h[r + m - k] = a[k + 1];
160       }
161       i = l;
162       j = r;
163       for (k = l; k <= r; k++) {
164         if (h[i] < h[j]) {
165           a[k] = h[i++];
166         } else {
167           a[k] = h[j--];
168         }
169       }
170     }
171
172     public static void mergesort(int[] a) {
173       int h[] = new int[a.length];
174       mergesort(a, 0, a.length - 1, h);
175     }
176   }
```

Code-Beispiel auf Github ansehen: `src/main/java/org/bschlangaul/sortier/Sammlung.java`

Literatur

[1] *Algorithmen und Datenstrukturen: Tafelübung 11, WS 2018/19*. `https://www.studon.fau.de/file2567217_download.html`. FAU: Lehrstuhl für Informatik 2 (Programmiersysteme).

[2] *Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen 2. Sortieren, Suchen, Komplexität*. `https://www.studon.fau.de/file2566441_download.html`.

[3] Gunter Saake und Kai-Uwe Sattler. *Algorithmen und Datenstrukturen. Eine Einführung in Java*. 2014.

[4] *Wikipedia-Artikel „Sortierverfahren“*. `https://de.wikipedia.org/wiki/Sortierverfahren`.