

19. Sortierverfahren

19.1 Allgemeines

19.2 Einfache Sortierverfahren

19.3 Effizientes Sortieren: Quicksort

19.4 Zusammenfassung

19. Sortierverfahren

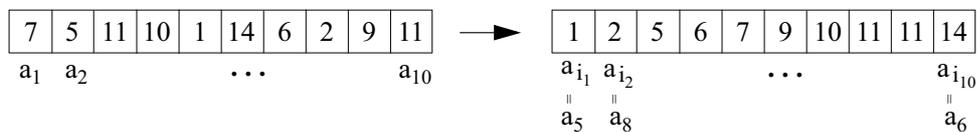
19.1 Allgemeines

19.2 Einfache Sortierverfahren

19.3 Effizientes Sortieren: Quicksort

19.4 Zusammenfassung

- Sortieren ist eine wichtige Operation auf Mengen von Objekten.
- Sortierte Mengen erlauben deutlich effizienteren Zugriff auf einzelne Elemente (z.B. durch binäre Suche).
- Problemstellung:
 - Gegeben: n Objekte a_1, \dots, a_n mit ihren Schlüsseln k_1, \dots, k_n , anhand derer die Objekte sortiert werden sollen.
 - Gesucht: Eine Anordnung a_{i_1}, \dots, a_{i_n} mit
 - (i_1, \dots, i_n) ist eine Permutation von $(1, \dots, n)$
 - $k_{i_1} \leq k_{i_2} \leq \dots \leq k_{i_n}$ (Sortierkriterium).
- Beispiel:



- Der Typ der Schlüssel k_i muss entweder primitiv sein, oder das Interface `Comparable` implementieren.
- Im folgenden betrachten wir folgende Vereinfachungen:
 - wir verwenden die Schreibweisen $k_i < k_j$, $k_i > k_j$, usw., bzw. $k_i = k_j$ unabhängig davon, ob k_i einen primitiven Datentyp oder einen Objekttyp besitzt (im letzteren Fall müssten wir eigentlich mit den Methoden `compareTo` bzw. `equals` arbeiten),
 - wir nehmen an, dass die Datenstruktur zur Verwaltung der Menge ein einfaches Array ist.
- Allgemeiner Sortieralgorithmus:


```
while  $\exists(i, j) : (i < j) \wedge (k_i > k_j)$  do vertausche  $a_i$  und  $a_j$ ;
```
- Problem: Algorithmus nicht deterministisch.
- Daher: speziellere Algorithmen nötig.

Definition (Stabilität von Sortierverfahren)

Ein Sortierverfahren heisst *stabil*, wenn die relative Ordnung von Elementen mit gleichen Schlüsselwerten beim Sortieren erhalten bleibt, d.h. wenn für die sortierte Menge k_{i_1}, \dots, k_{i_n} gilt: $k_{i_j} = k_{i_l}$ und $j < l \Rightarrow i_j < i_l$.

Definition (in situ Sortierverfahren)

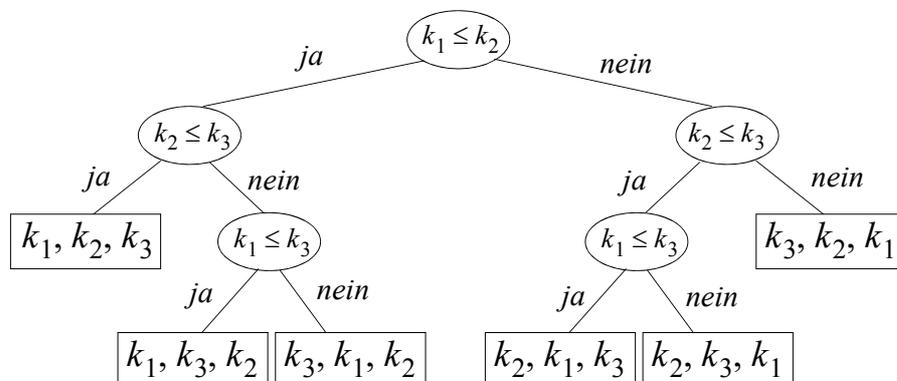
Ein Sortierverfahren heisst *in situ*, wenn zusätzlich zu dem zu sortierenden Array *kein* weiterer Speicherplatz benötigt wird. Dabei wird angenommen, dass die zu sortierenden Objekte die Indexpositionen $1, \dots, n$ belegen und das Feld mit Index 0 für Vertauschungen genutzt werden kann.

- Es gibt verschiedene Kriterien, um Sortieralgorithmen zu klassifizieren, z.B. ihre Stabilität oder ihren Speicherplatzbedarf.
- Das in dieser Vorlesung wesentliche Kriterium ist das Laufzeitverhalten.
- Um das Laufzeitverhalten unterschiedlicher Verfahren miteinander zu vergleichen, zählen wir die Anzahl der bei einer Sortierung von n Objekten durchzuführenden Operationen (in Abhängigkeit von n).

- Die beiden wesentlichen Operationen der meisten Sortierverfahren sind:
 - Vergleiche* von Schlüsselwerten um Informationen über die vorliegende Ordnung zu erhalten. Im folgenden bezeichnen wir die Anzahl dieser Schlüssel-Vergleiche mit $C(n)$.
 - Zuweisungsoperationen*, z.B. Vertauschungsoperationen oder Tranpositionen von Objekten (i.a. innerhalb eines Arrays) zur Herstellung der Sortierordnung. Im folgenden bezeichnen wir die Anzahl dieser Zuweisungsoperationen mit $M(n)$.
- Wir beschränken uns hier auf Algorithmen, die nur Schlüsselvergleiche und Transpositionen verwenden.

Wie schnell kann man sortieren?

- Gesucht: eine untere Schranke für die Anzahl $C_{\max}(n)$ von Schlüsselvergleichen, die im schlechtesten Fall notwendig sind, um n Objekte zu sortieren.
- Entscheidungsbaum für 3 Schlüssel k_1, k_2 und k_3 :



- In einem Entscheidungsbaum ohne redundante Vergleiche entspricht jedes Blatt einer der $n!$ verschiedenen Permutationen.
- Da der Entscheidungsbaum ein binärer Baum ist, hat er für n Elemente eine minimale Höhe von $\lceil \log(n!) \rceil$.

- Damit gilt für einen beliebigen Sortieralgorithmus:

$$C_{max}(n) \geq O(n \cdot \log n), \text{ denn}$$

$$C_{max}(n) \geq \lceil \log(n!) \rceil \quad \text{und}$$

$$n! \geq n \cdot (n-1) \cdot \dots \cdot \lceil n/2 \rceil \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

$$\Rightarrow \log_2(n!) \geq \frac{n}{2} \cdot \log_2\left(\frac{n}{2}\right) = O(n \cdot \log n).$$

- Resultat: Sortierverfahren haben mindestens eine Laufzeit von $O(n \cdot \log n)$.

19. Sortierverfahren

19.1 Allgemeines

19.2 Einfache Sortierverfahren

19.3 Effizientes Sortieren: Quicksort

19.4 Zusammenfassung

- Einfache Sortierverfahren besitzen meist eine Laufzeit von $O(n^2)$.
- Für wenige Objekte (kleine Werte von n) ist dies meist noch akzeptabel, für große Mengen (z.B. $n > 100$) allerdings nicht mehr.
- In den Übungen haben wir bereits eine Reihe einfacher Sortieralgorithmen kennen gelernt, von denen wir einige im folgenden noch einmal kurz besprechen.
- Was die Java-Programme anbetrifft, wird hier auf die Übungen verwiesen.

- Prinzip:
Der j -te Schlüssel der sortierten Folge ist größer als $j - 1$ der übrigen Schlüssel. Die Position eines Schlüssels in der sortierten Folge kann damit durch Abzählen der kleineren Schlüssel ermittelt werden.
- Vorgehen:
Für jedes Objekt o im ursprünglichen Array: zähle die Objekte o' mit $o' < o$.
- Laufzeit:
$$C(n) = (n - 1) + (n - 2) + \dots + 1 = \frac{n \cdot (n - 1)}{2} = O(n^2)$$
$$M(n) = C(n) = O(n^2)$$
- Das Verfahren ist nicht in situ.
- Das Verfahren ist stabil (wird instabil falls Test $o' \leq o$).

- **Prinzip/Vorgehen:**
Suche aus allen n Elementen das kleinste und setze es an die erste Stelle. Wiederhole dieses Verfahren für die verbleibenden Elemente. Nach $n - 1$ Durchläufen ist die Folge sortiert.
- **Laufzeit:**
$$C(n) = (n - 1) + (n - 2) + \dots + 1 = \frac{n \cdot (n-1)}{2} = O(n^2)$$
$$M_{\emptyset}(n) = O(n \log n) \text{ (ohne Beweis)}$$
- Das Verfahren ist in situ.
- Je nach Implementierung ist das Verfahren stabil.

- **Prinzip:**
Vertausche die relative Reihenfolge benachbarter Elemente, sodass kleine Elemente nach vorne und größere Elemente nach hinten wandern.
- **Vorgehen:**
Im ersten Durchlauf werden die Paare $(a_{n-1}, a_n), (a_{n-2}, a_{n-1}), \dots$ bearbeitet. Dadurch wandert das kleinste Element an die erste Position im Array. Nach $n - 1$ Durchläufen ist die Sortierung abgeschlossen.
- **Laufzeit:**
$$C(n) = (n - 1) + (n - 2) + \dots + 1 = \frac{n \cdot (n-1)}{2} = O(n^2)$$
$$M_{\emptyset}(n) = O(n^2) \text{ (ohne Beweis)}$$
- Das Verfahren ist in situ.
- Je nach Implementierung ist das Verfahren stabil.

19. Sortierverfahren

19.1 Allgemeines

19.2 Einfache Sortierverfahren

19.3 Effizientes Sortieren: Quicksort

19.4 Zusammenfassung

- Analyse: einfache Sortierverfahren reduzieren die Größe des noch zu sortierenden Arrays in jedem Schritt lediglich um eins.
- Idee: Reduziere das noch zu sortierende Array in jedem Schritt um die Hälfte. Dadurch wird eine Laufzeit von $O(n \log n)$ ermöglicht.
- Das allgemeine Algorithmus-Prinzip von Quicksort heißt *Divide-and-Conquer* (teile-und-beherrsche).
- Divide-and-Conquer-Algorithmen zerlegen ein gegebenes Problem solange in kleinere Probleme, bis diese beherrschbar sind. Die globale Lösung ergibt sich durch Verschmelzen der einzelnen (Teil-)Lösungen.

- Allgemeines Schema eines Divide-and-Conquer-Sortieralgorithmus:

ALGORITHM Divide-and-Conquer-Sort

IF Objektmenge klein genug

THEN loese das Problem direkt;

ELSE

DIVIDE: Zerlege die Menge in Teilmengen moeglichst gleicher Groesse;

CONQUER: Loese das Problem fuer jede der Teilmengen;

MERGE: Berechne aus den Teilloesungen die Gesamtloesung;

ENDIF

END

- Wichtige Eigenschaft: Jedes Divide-and-Conquer-Sortierverfahren, dessen *Divide-* und *Merge-*Schritt in $O(n)$ Zeit durchgeführt werden können und das eine balancierte Unterteilung des Problems garantiert, besitzt eine Laufzeit von $O(n \log n)$.

- Ein (prinzipiell beliebiger) Schlüssel x aus dem Array wird ausgewählt (das sog. *Pivot-Element*).
- *Divide*-Schritt: das Array wird in Schlüssel $\geq x$ und Schlüssel $< x$ zerlegt.
- *Conquer*-Schritt: die beiden resultierenden Teilarrays werden rekursiv bis auf Elementebene in gleicher Weise behandelt.
- *Merge*-Schritt: entfällt, durch entsprechende Speicherung der Teilarrays innerhalb des Arrays.

```

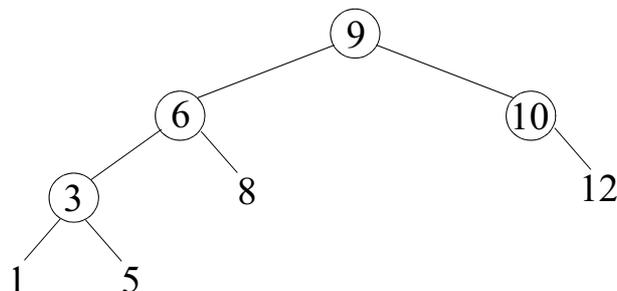
ALGORITHM QuickSort(ARRAY S)
  IF S.length = 1
  THEN RETURN S;
  ELSE
    // DIVIDE:
    Waehle ein Pivot-Element x aus S;
    Initialisiere zwei Teilfolgen S1 und S2;
    FOR EACH y IN S DO
      IF y < x
      THEN fuege x zu S1 hinzu;
      ELSE fuege x zu S2 hinzu;
      ENDIF
    ENDFOR
    // CONQUER:
    S1' = QuickSort(S1);
    S2' = QuickSort(S2);
    // MERGE:
    RETURN Konkatination aus S1' und S2';
  ENDIF
END

```

- Arbeit im *Divide*-Schritt: $O(n)$.
- Arbeit im *Merge*-Schritt: $O(1)$.

9	5	1	12	8	10	6	3
---	---	---	----	---	----	---	---

- Rekursive Unterteilung des Array bzgl. des Pivot-Elements ergibt einen Binärbaum über den Array-Elementen (innere Knoten entsprechen den einzelnen Pivot-Elementen).



- Beobachtung: Höhe des Baumes entspricht Rekursionstiefe des Verfahrens.
- Erwünscht: balancierter binärer Baum \Rightarrow Rekursionstiefe $O(\log n)$.

- Ausgangspunkt des Divide-Schritt: angenommen das k -te Element der Sortierreihenfolge wird als Pivot gewählt und dann an Position k positioniert.
- Im Zerlegungsprozess wird jedes Element betrachtet: $C(n) = n$.
- Die Anzahl der Vertauschungsoperationen ist abhängig von k :

$$M(n) = (\text{Anz. Elemente} < k) \cdot (\text{WS für eine Vertauschung})$$

$$M(n) = (\text{Anz. Elemente} < k) \cdot (\text{WS, dass Element im rechten Teil})$$

$$M(n) = (k - 1) \cdot \frac{n - (k - 1)}{n}.$$

- Mittelwert $\bar{M}(n)$ der Vertauschungen über alle $k = 1, \dots, n$ für den ersten Zerlegungsschritt:

$$\begin{aligned} \bar{M}(n) &= \frac{1}{n} \sum_{k=1}^n (k - 1) \cdot \frac{n - (k - 1)}{n} \\ &= \frac{1}{n^2} \cdot \sum_{k=0}^{n-1} k \cdot (n - k) \\ &= \frac{n \cdot (n - 1)}{2 \cdot n} - \frac{2 \cdot n^2 - 3 \cdot n + 1}{6 \cdot n} = \frac{n - 1/n}{6} \\ &= O(n). \end{aligned}$$

Der günstigste Fall

Das Array wird in jeweils zwei Teilarrays etwa gleicher Größe geteilt. Dann folgt für den Gesamtzeitbedarf $T(n)$:

$$\begin{aligned}T_{best}(n) &= c \cdot n + 2T\left(\frac{n}{2}\right) = c \cdot n + 2\left(c \cdot \frac{n}{2} + 2T\left(\frac{n}{4}\right)\right) \\ &= 2cn + 4T\left(\frac{n}{4}\right) = \dots \\ &= \log n \cdot c \cdot n + n \cdot T(1) = O(n \log n)\end{aligned}$$

Quicksort: Laufzeitanalyse (Gesamt/average case)

Der durchschnittliche Fall

Das Array wird jeweils bezüglich eines zufällig ausgewählten Pivot-Elements zerlegt.

Nach Analysen ist dieser Fall nur um den Faktor $2 \cdot \ln(2) \approx 1,39$ (ca. 40%) schlechter.

$$\Rightarrow T_{\emptyset}(n) = O(n \log n).$$

Der schlechteste Fall

Degenerierung zur linearen Liste: stets wird das größte/kleinste Element als Pivot gewählt.

$$T_{\text{worst}}(n) = (n - 1) + (n - 2) + \dots + 1 = \frac{n \cdot (n - 1)}{2} = O(n^2)$$

Beachte hierbei auch die hohe Rekursionstiefe n .

Quicksort: Schlussbemerkungen

- Austausch von Schlüsseln über große Distanzen \Rightarrow Array ist schnell “nahezu” sortiert.
- Wie alle komplexen Sortierverfahren ist Quicksort schlecht für kleine n .
- Es gibt verschiedene Methoden, die Wahrscheinlichkeit des schlechtesten Falls zu vermindern; dennoch bleibt Quicksort immer eine Art “Glückspiel”, denn es bleibt

$$T_{\text{worst}}(n) = O(n^2).$$

- Quicksort war lange Zeit das im Durchschnittsverhalten beste bekannte Sortierverfahren.
- Die statischen Methoden `java.util.Arrays.sort` implementieren Quicksort.

19. Sortierverfahren

19.1 Allgemeines

19.2 Einfache Sortierverfahren

19.3 Effizientes Sortieren: Quicksort

19.4 Zusammenfassung

Sie kennen jetzt

- die untere Grenze der Laufzeit von Sortierverfahren,
- die Idee und die Laufzeit von einfachen Sortierverfahren,
- die Idee sowie Vor- und Nachteile des Sortierverfahrens Quicksort.

- Vertiefung der objektorientierten Programmierung:
 - Programmierpraktikum (3. Semester)
 - diverse Hauptstudiumsvorlesungen
- Vertiefung Datenstrukturen und Algorithmen:
 - Effiziente Algorithmen (4. Semester)
 - Index- und Speicherstrukturen (Hauptstudium)
 - Spatial-Temporal-Multimedia-Database-Systems (Hauptstudium)