ADS: Algorithmen und Datenstrukturen

Teil $\lceil \pi \rceil$

Prof. Peter F. Stadler & Dr. Christian Höner zu Siederdissen

Bioinformatik/IZBI
Institut für Informatik
& Interdisziplinäres Zentrum für Bioinformatik
Universität Leipzig

2.11.2015

[Letzte Aktualisierung: 26/10/2015, 01:13]

Sortierverfahren

- Elementare Sortierverfahren
  - Sortieren durch direktes Auswählen (Straight Selection Sort)
  - Sortieren durch Vertauschen (Bubble Sort)
  - Sortieren durch Einfügen (Insertion Sort)
- Shell-Sort (Sortieren mit abnehmenden Inkrementen)
- Quick-Sort Auswahl-Sortierung (Turnier-Sortierung, Heap-Sort)
- Sortieren durch Streuen und Sammeln
- Merge-Sort
- Externe Sortierverfahren
  - Ausgeglichenes k-Wege-Merge-Sort
  - Auswahl-Sortierung mit Ersetzung (Replacement Selection)

Sortieren: Anwendungen

Sortieren ist **fundamentale Operation** in vielen System- und Anwendungsprogrammen

dominiert Programmlaufzeit bei großen Datenmengen

Beispiele:

- Wörter in Lexikon
- Bibliothekskatalog
- Kontoauszug (geordnet nach Datum der Kontobewegung)
- Sortieren von Studenten nach Namen, Notenschnitt, Semester, ...
- Sortieren von Adressen / Briefen nach Postleitzahlen, Ort, Straße,...
- Teilaufgabe komplexer Anwendungen: Datenbanken, Datamining, Information Retrieval, Bioinformatik, ...

Anwendung “Duplikaterkennung”

Naive Implementierung: Vergleiche jeden Datensatz mit jedem anderen. Algorithmus für Duplikate in Feld L der Länge n (im Folgenden sei L 1-basiert, also L=L[1..n]):

```
for (i = 1..n-1)
  for ( j = i+1..n)
     if ( L[i] == L[j] ) mark_as_duplicate( L[j] )
```

Aufwand: $O(n^2)$

Zweiter Versuch:

- Liste sortieren (danach stehen Duplikate unmittelbar hintereinander).
- Liste einmal linear durchlaufen und Duplikate markieren.

Falls wir besser als in $O(n^2)$ sortieren können, ist das insgesamt effizienter.

Beschreibung von Sortierverfahren I

Allgemeine Problemstellung:

- Gegeben: Folge von Datensätzen $S_1, \ldots S_n$ mit Schlüsseln $K_1, \ldots K_n$.
- Gesucht: Permutation der Zahlen von 1 bis n, welche aufsteigende Schlüsselreihenfolge ergibt: $K_{\pi(1)} \leq K_{\pi(2)} \leq \ldots \leq K_{\pi(n)}$.

Sortierverfahren:

- *intern* (im Hauptspeicher)
  - $O(n^\alpha)$
  - $O(n \log n)$
- *extern* (Nutzung von Externspeicher)
  - Mischen von vorsortierten (Teil-)Folgen

Beschreibung von Sortierverfahren II

- allgemeine vs. spezielle Sortierverfahren
  — Annahmen über Datenstrukturen (z.B. Array) oder Schlüsseleigenschaften
- Wünschenswert: stabile Sortierverfahren, bei denen die relative Reihenfolge gleicher Schlüsselwerte bei der Sortierung gewahrt bleibt
- Speicherplatzbedarf am geringsten für Sortieren "in place" (*in situ*)
- Weitere Kostenmaße:
  \# Schlüsselvergleiche C ($C_{\min}$, $C_{\max}$, C_{avg})
  \# Satzbewegungen M ($M_{\min}$, $M_{\max}$, M_{avg})

Voraussetzung für jedes Sortieren: Ordnung

Auf den zu sortierenden Objekten muss eine Ordnung definiert sein.

Sei X eine Menge, und $\leq$ eine Relation auf X. Betrachte die folgenden Eigenschaften, die für alle $x, y, z \in X$ gelten mögen:

(O1) $x \leq x$ (reflexiv)

(O2) $x \leq y$ und $y \leq x$ impliziert $x = y$ (anti-symmetrisch)

(O3) $x \leq y$ und $y \leq z$ impliziert $x \leq z$ (transitiv)

(O4) $x \leq y$ oder $y \leq x$ (total)

$(X, \leq)$ ist eine *Halbordnung* wenn (O1), (O2) und (O3) gelten.
$(X, \leq)$ ist eine *Ordnung* wenn alle 4 Axiome gelten.

$x < y$ ist definiert als "$x \leq y$ und $x \neq y$".
$x \geq y$ und $x > y$ sind definiert als "$y \leq x$" bzw. "$y < x$".

Wie schnell kann man sortieren?

Satz. Jedes allgemeine Sortierverfahren, welches zum Sortieren nur Vergleichsoperationen zwischen Schlüsseln verwendet, benötigt sowohl im mittleren als auch im schlechtesten Fall wenigstens $\Omega(n \log n)$ Schlüsselvergleiche.

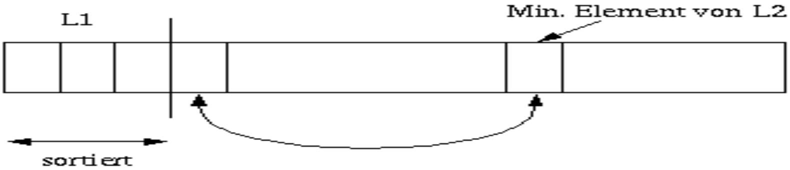
Beweisskizze:

- Sortieren muss unter $n!$ möglichen Permutationen auswählen.
- Dazu benötigt man $\log_2(n!)$ viele bits Information.
- Jeder Vergleich liefert höchstens ein zusätzliches bit.
- Also benötigt Sortieren $\geq \log_2(n!)$ Vergleiche, das ist in $\Omega(n \log n)$.

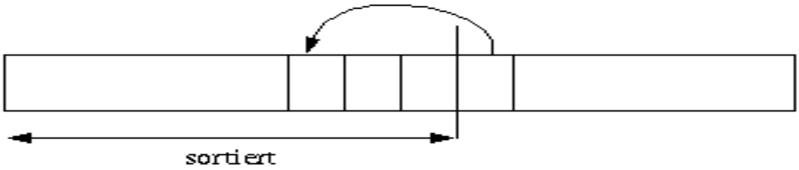
Was heißt allgemeiner Fall? Wann geht es schneller?
$\log_2(n!) \in \Omega(n \log n)$ **rechnen sie selbst nach!**

Sortieren durch ...

1. **Auswählen**

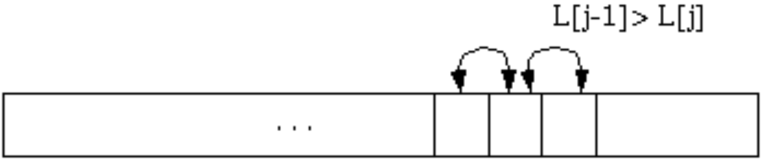


2. **Einfügen**

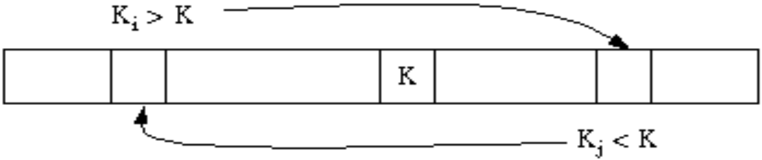


Klassifizierung von Sortiertechniken II

3. **Lokales Austauschen**: vertausche lokale Fehlordnungen $x > y$

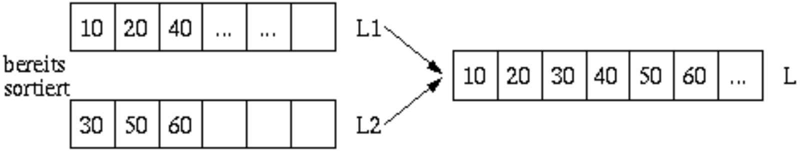


4. **Nicht-Lokales Austauschen**

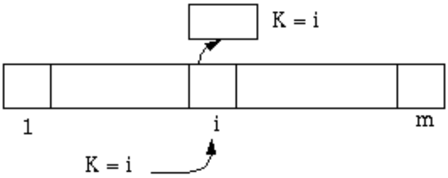


Sortieren durch ...

5. Mischen ("Reißverschlussverfahren")



6. Streuen & Sammeln



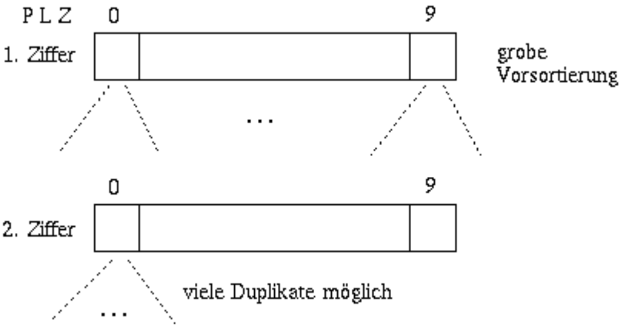
- begrenzter Schlüsselbereich m, z. B. 1 – 1000
- relativ dichte Schlüsselbelegung $n \leq m$
- Duplikate möglich $(n > m)$
- lineare Sortierkosten !!!!

Klassifizierung von Sortiertechniken III

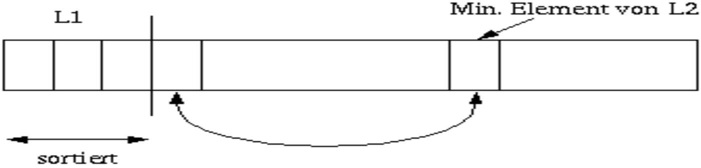
Sortieren durch ...

7. Fachverteilen (z.B. Poststelle) zum Beispiel: Postleitzahlen in 10 Fächer nach der 1. Stelle

   dann Sortierung der "Fächer"



Sortieren durch Auswählen (Selection Sort) I



Algorithmus-Idee

1. **Start:** L1=leer, die unsortierte Teilliste L2 ist die ganze Liste
2. **WHILE**(Liste L2 enthält mehr als ein Element):
    3. Wähle kleinstes Element x in L2
    4. Hänge x an L1 an und lösche x aus L2
       (dazu: Tausche x mit erstem Element von L2)

Beispiel: 12 3 99 12 75 → Tafel

Selection Sort II

Algorithmus (Selection Sort von L[1..n])

```
for i = 1 .. n-1
  min = i
  for j = i+1 .. n
     if (L[j] < L[min]) min = j
  swap L[i] and L[min]
```

Eigenschaften:

- Anzahl Schlüsselvergleiche: $C_{\text{min}}(n) = C_{\text{max}}(n) = n(n-1)/2$
- Anzahl Satzbewegungen (durch Swap): $M_{\text{min}}(n) = M_{\text{max}}(n) = 3(n-1)$
- *In-situ*-Verfahren
- nicht stabil (Beispiel: $2_1 2_2 1 \rightarrow 1 2_2 2_1$)

1 Swap = 3 Satz-Bewegungen, "swap x and y": {t=x; x=y; y=t}

Bubble Sort I

Idee:

- Durchlaufe L und vertausche jeweils benachbarte Elemente, die nicht in Sortierordnung sind
  (dabei steigen große Element immer weiter auf, "wie Blasen")
- Wiederhole dieses Durchlaufen bis keine Vertauschung mehr nötig sind

Methode: "Sortieren durch lokale Vertauschung"

Variation: Shaker-Sort (Durchlaufrichtung wechselt bei jedem Durchgang)

Bubble Sort II

Algorithmus 1 (Bubble Sort von L[1..n])

```
for i=n-1 .. 1
   for j=1 .. i
       if ( L[j] > L[j+1] )
         swap L[j] and L[j+1]
```

Eigenschaften:
Anzahl Schlüsselvergleiche:

$$C_{\min}(n) = C_{\max}(n) = C_{avg}(n) = n(n-1)/2$$

Anzahl Satzbewegungen (durch Swap):

$$M_{\min}(n) = 0, M_{\max}(n) = 3n(n-1)/2, M_{avg}(n) = M_{\max}/2$$

in situ, stabil, Vorsortierung kann teilweise genutzt werden

Bubble Sort III

Algorithmus 2 (Bubble Sort von L[1..n] mit Abbruchkontrolle)

```
i=n;
do {
   swapped = false;
   for j = 1 .. i-1
       if ( L[j] > L[j+1] ) {
             swap L[j] and L[j+1];
             swapped = true
       }
   i=i-1;
} while( swapped )
```

Eigenschaften:
Anzahl Schlüsselvergleiche: $C_{\min}(n) = n - 1$, $C_{\max}(n) = n(n-1)/2$
Anzahl Satzbewegungen (durch Swap):

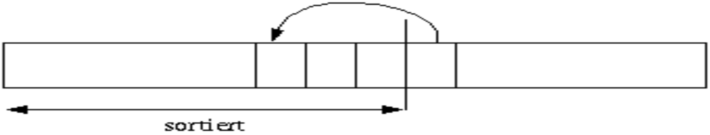
$$M_{\min}(n) = 0, M_{\max}(n) = 3n(n-1)/2$$

in situ, stabil, Vorsortierung kann teilweise genutzt werden

Sortieren durch Einfügen (Insertion Sort) I

Idee:

- i-tes Element der Liste x (1. Element der unsortierten Teilliste) wird an der richtigen Stelle der bereits sortierten Teilliste (1 bis $i-1$) eingefügt
- Elemente in sortierter Teilliste mit höherem Schlüsselwert als x werden verschoben



Algorithmus (Insertion Sort von L[1..n])

```
for i = 2 .. n
   temp = L[i]; j = i-1;
   while (j>0 and L[j]>temp) { L[j+1] = L[j]; j--; }
   L[j+1] = temp;
```

Beispiel 27 75 99 3 45 12 87 → TAFEL

Eigenschaften

Vergleich der einfachen Sortierverfahren

Schlüsselvergleiche

Algorithm	$C_{\min}$	C_{avg}	$C_{\max}$
Selection	$n(n-1)/2$	$n(n-1)/2$	$n(n-1)/2$
Bubble	$n(n-1)/2$	$n(n-1)/2$	$n(n-1)/2$
Insertion	$n-1$	$\approx n(n-1)/4 + n - \ln(n)$	$n(n-1)/2$

Satzbewegungen

Algorithm	$M_{\min}$	M_{avg}	$M_{\max}$
Selection	$3(n-1)$	$3(n-1)$	$3(n-1)$
Bubble	0	$3n(n-1)/4$	$3n(n-1)/2$
Insertion	$2(n-1)$	$(n-1) + (n^2+3n-4)/4$	$2(n-1) + n(n-1)/2$ $= (n^2+3n-4)/2$

Sortieren von großen Datensätzen (Indirektes Sortieren)

Indirektes Sortieren erlaubt, Kopieraufwand für jedes Sortierverfahrens auf lineare Kosten $O(n)$ zu beschränken.

Methode:
Führen eines Hilfsfeldes von Indizes auf das eigentliche Listenfeld

- Liste: $L[1..n]$
- Pointerfeld $P[1..n]$ (Initialisierung $P[i] = i$)
- Schlüsselzugriff: statt $L[i]$ immer $L[P[i]]$
- Austausch: statt swap von $L[i]$ und $L[j]$ nur swap von $P[i]$ und $P[j]$

Sortierung erfolgt lediglich auf Indexfeld

abschließend: linearer Durchlauf zum Umkopieren der Sätze

Shell Sort:
Sortieren mit abnehmenden Inkrementen (Shell, 1957)

Idee

- Sortierung in mehreren Stufen “von grob bis fein”
- Vorsortierung reduziert Anzahl von Tauschvorgängen

Vorgehensweise:

- Festlegung von t Inkrementen (Elementabständen) h_i mit $h_1 > h_2 > ... > h_t = 1$. Wir bezeichnen als h_i-Folge eine Folge von Elementen aus der Liste mit Abstand h_i
- Im i-ten Schritt erfolgt unabhängiges Sortieren aller h_i-Folgen (mit Insertion Sort)
- Eine Elementbewegung bewirkt Sprung um h_i Positionen
- Im letzten Schritt erfolgt “normales” Sortieren durch Einfügen

Shell-Sort: Beispiel

BESPIEL: $h_1 = 4,\ h_2 = 2,\ h_3 = 1$

Original	27	75	99	3	45	12	87
4-Folgen	a	b	c	d	a	b	c
	27	12	87	3	45	75	99
2-Folgen	p	q	p	q	p	q	p
	27	3	45	12	87	75	99
1-Folge							

Shell-Sort II: Komplexität

- wesentlich abhängig von Wahl der Anzahl und Art der Schrittweiten
- Insertion Sort ist Spezialfall ($t = 1$)
- Knuth [Kn73] empfiehlt:
  Schrittweitenfolge von 1, 3, 7, 15, 31 ...
  mit $h_{i-1} = 2h_i + 1$, $h_t = 1$ und $t = [\log_2 n] - 1$
  Dann: Aufwand $O(n^{1.2})$
- Andere Vorschläge erreichen $O(n \log n)$
  mit Inkrementen der Form $2^p 3^q$