

ADS: Algorithmen und Datenstrukturen

Teil 4

Uwe Quasthoff

Institut für Informatik
Abteilung Automatische Sprachverarbeitung
Universität Leipzig

7.11.2017

[Letzte Aktualisierung: 30/11/2017, 12:02]

- 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 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, ...

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.

Allgemeine Problemstellung:

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

Sortierverfahren:

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

- 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.

Wie schnell kann man sortieren?

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 $\geq \log_2(n!)$.
- **Lemma.** $(n/2)^{n/2} \leq n! \leq n^n$



Zeigen Sie das mittels vollständiger Induktion!

- $\log_2(n!) \in \Omega(n \log n)$



warum?

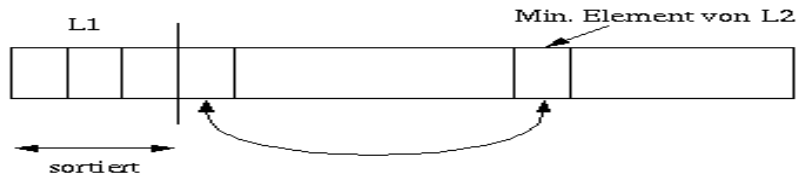


Was heißt allgemeiner Fall? Wann geht es schneller?

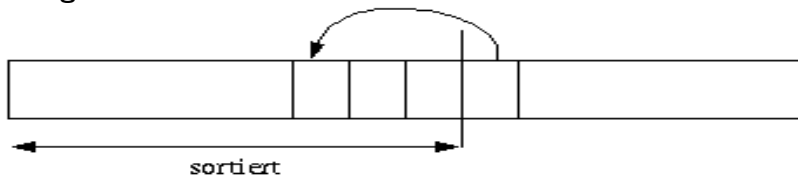
Klassifizierung von Sortiertechniken

Sortieren durch ...

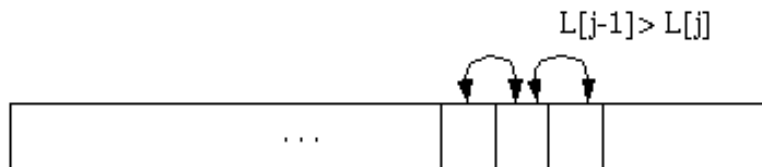
1 Auswählen



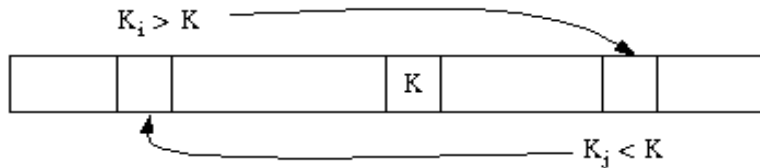
2 Einfügen



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



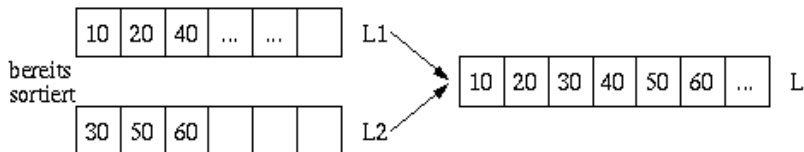
- ④ **Nicht-Lokales Austauschen**



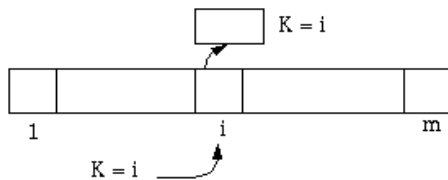
Klassifizierung von Sortiertechniken II

Sortieren durch ...

5 Mischen ("Reißverschlussverfahren")



6 Streuen & Sammeln

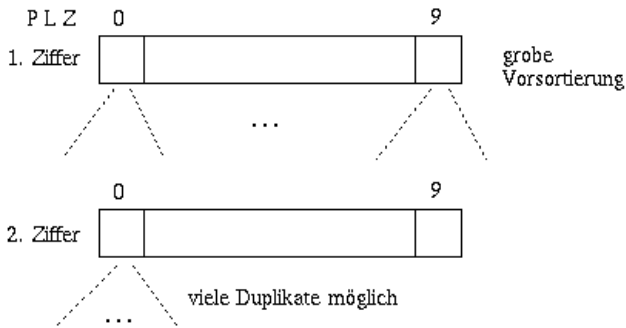


- begrenzter Schlüsselbereich m , z. B. 1 – 1000
- relativ dichte Schlüsselbelegung $n \leq m$
- Duplikate möglich ($n > m$)
- **lineare Sortierkosten !!!!**

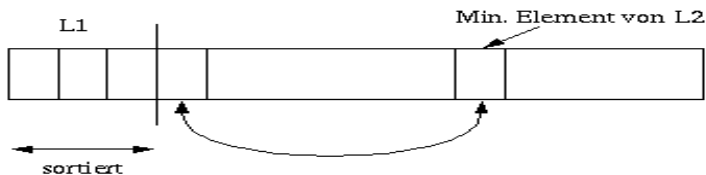
Klassifizierung von Sortiertechniken III

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



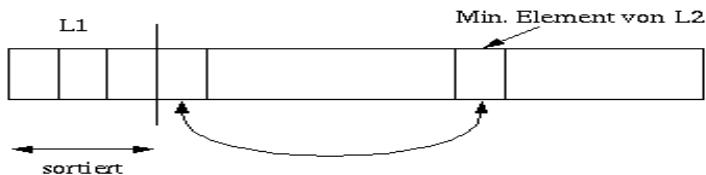
Algorithmus-Idee

- 1 L1 = Sortiert, L2 = Unsortiert
- 2 **WHILE**(L2 enthält mehr als ein Element):
 - 1 Wähle Minimum x in L2
 - 2 Vertausche x mit erstem Element in L2
 - 3 Vergrößere L1 um x
 - 4 Verkleinere L2 um x

Beispiel:

L1: | L2: 12 3 99 12 75

Sortieren durch Auswählen (Selection Sort) I



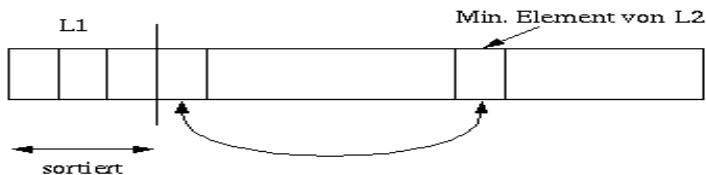
Algorithmus-Idee

- 1 L1 = Sortiert, L2 = Unsortiert
- 2 **WHILE**(L2 enthält mehr als ein Element):
 - 1 Wähle Minimum x in L2
 - 2 Vertausche x mit erstem Element in L2
 - 3 Vergrößere L1 um x
 - 4 Verkleinere L2 um x

Beispiel:

L1: | L2: 12 3 99 12 75

Sortieren durch Auswählen (Selection Sort) I



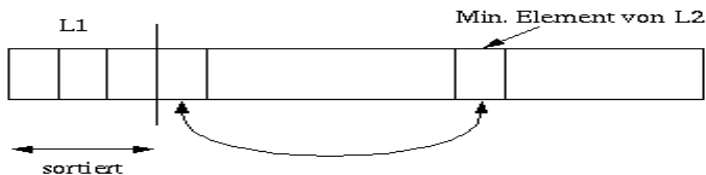
Algorithmus-Idee

- 1 L1 = Sortiert, L2 = Unsortiert
- 2 **WHILE**(L2 enthält mehr als ein Element):
 - 1 Wähle Minimum x in L2
 - 2 Vertausche x mit erstem Element in L2
 - 3 Vergrößere L1 um x
 - 4 Verkleinere L2 um x

Beispiel:

L1: 3 | L2: 12 99 12 75

Sortieren durch Auswählen (Selection Sort) I



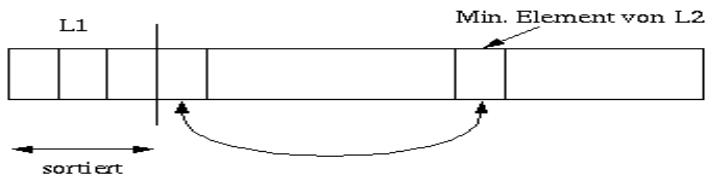
Algorithmus-Idee

- 1 L1 = Sortiert, L2 = Unsortiert
- 2 **WHILE**(L2 enthält mehr als ein Element):
 - 1 Wähle Minimum x in L2
 - 2 Vertausche x mit erstem Element in L2
 - 3 Vergrößere L1 um x
 - 4 Verkleinere L2 um x

Beispiel:

L1: 3 12 | L2: 99 12 75

Sortieren durch Auswählen (Selection Sort) I



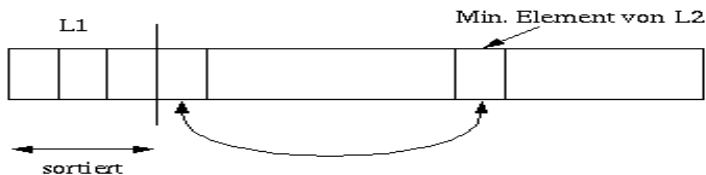
Algorithmus-Idee

- 1 L1 = Sortiert, L2 = Unsortiert
- 2 **WHILE**(L2 enthält mehr als ein Element):
 - 1 Wähle Minimum x in L2
 - 2 Vertausche x mit erstem Element in L2
 - 3 Vergrößere L1 um x
 - 4 Verkleinere L2 um x

Beispiel:

L1: 3 12 12 | L2: 99 75

Sortieren durch Auswählen (Selection Sort) I



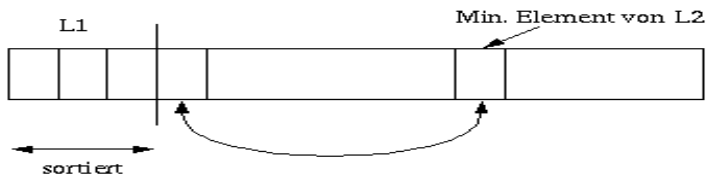
Algorithmus-Idee

- 1 L1 = Sortiert, L2 = Unsortiert
- 2 **WHILE**(L2 enthält mehr als ein Element):
 - 1 Wähle Minimum x in L2
 - 2 Vertausche x mit erstem Element in L2
 - 3 Vergrößere L1 um x
 - 4 Verkleinere L2 um x

Beispiel:

L1: 3 12 12 75 | L2: 99

Sortieren durch Auswählen (Selection Sort) I



Algorithmus-Idee

- 1 L1 = Sortiert, L2 = Unsortiert
- 2 **WHILE**(L2 enthält mehr als ein Element):
 - 1 Wähle Minimum x in L2
 - 2 Vertausche x mit erstem Element in L2
 - 3 Vergrößere L1 um x
 - 4 Verkleinere L2 um x

Beispiel:

L1: 3 12 12 75 99 | L2:

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

```
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]
```

Beispiel (i, min):

L: 12 3 99 12 75

Eigenschaften:

- Anzahl Schlüsselvergleiche: $C_{\min}(n) = C_{\max}(n) = n(n-1)/2$
- Anzahl Satzbewegungen (durch Swap):
 $M_{\min}(n) = M_{\max}(n) = 3(n-1)$
- *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}

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

```
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]
```

Beispiel (i, min):

L: 3 12 99 12 75

Eigenschaften:

- Anzahl Schlüsselvergleiche: $C_{\min}(n) = C_{\max}(n) = n(n-1)/2$
- Anzahl Satzbewegungen (durch Swap):
 $M_{\min}(n) = M_{\max}(n) = 3(n-1)$
- *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}

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

```
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]
```

Beispiel (i, min):

L: 3 12 99 12 75

Eigenschaften:

- Anzahl Schlüsselvergleiche: $C_{\min}(n) = C_{\max}(n) = n(n-1)/2$
- Anzahl Satzbewegungen (durch Swap):
 $M_{\min}(n) = M_{\max}(n) = 3(n-1)$
- *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}

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

```
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]
```

Beispiel (i, min):

L: 3 12 12 99 75

Eigenschaften:

- Anzahl Schlüsselvergleiche: $C_{\min}(n) = C_{\max}(n) = n(n-1)/2$
- Anzahl Satzbewegungen (durch Swap):
 $M_{\min}(n) = M_{\max}(n) = 3(n-1)$
- *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}

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

```
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]
```

Beispiel (i, min):

L: 3 12 12 75 99

Eigenschaften:

- Anzahl Schlüsselvergleiche: $C_{\min}(n) = C_{\max}(n) = n(n-1)/2$
- Anzahl Satzbewegungen (durch Swap):
 $M_{\min}(n) = M_{\max}(n) = 3(n-1)$
- *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”

12 3 99 12 75 65

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

Bubble Sort I



Idee:

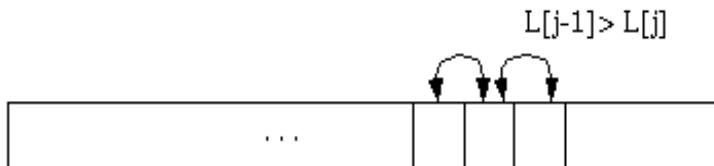
- Durchlaufe L und vertausche jeweils benachbarte Elemente, die nicht in Sortierordnung sind
(dabei steigen große Element immer weiter auf, “wie Blasen”)
- Wiederhole dieses Durchlaufen bis keine Vertauschung mehr nötig sind

Methode: “Sortieren durch lokale Vertauschung”

3 12 99 12 75 65

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

Bubble Sort I



Idee:

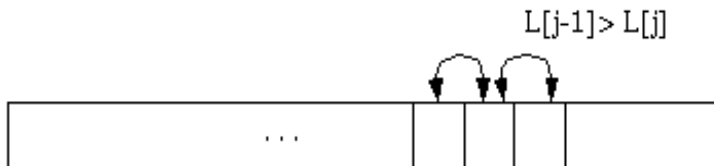
- Durchlaufe L und vertausche jeweils benachbarte Elemente, die nicht in Sortierordnung sind
(dabei steigen große Element immer weiter auf, “wie Blasen”)
- Wiederhole dieses Durchlaufen bis keine Vertauschung mehr nötig sind

Methode: “Sortieren durch lokale Vertauschung”

3 12 99 12 75 65

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

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”

3 12 12 99 75 65

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

Bubble Sort I



Idee:

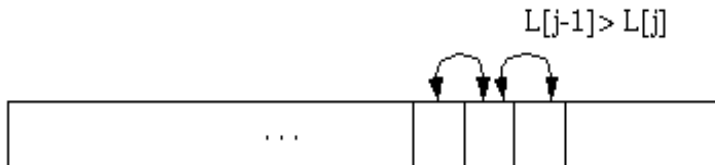
- Durchlaufe L und vertausche jeweils benachbarte Elemente, die nicht in Sortierordnung sind
(dabei steigen große Element immer weiter auf, “wie Blasen”)
- Wiederhole dieses Durchlaufen bis keine Vertauschung mehr nötig sind

Methode: “Sortieren durch lokale Vertauschung”

3 12 12 75 99 65

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

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”

3 12 12 75 65 99

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

Bubble Sort I



Idee:

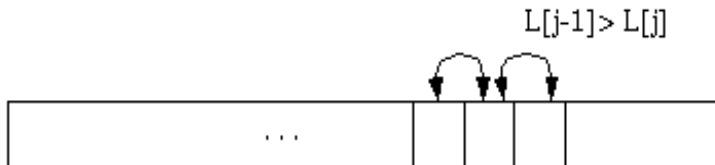
- Durchlaufe L und vertausche jeweils benachbarte Elemente, die nicht in Sortierordnung sind
(dabei steigen große Element immer weiter auf, “wie Blasen”)
- Wiederhole dieses Durchlaufen bis keine Vertauschung mehr nötig sind

Methode: “Sortieren durch lokale Vertauschung”

3 12 12 75 65 99

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

Bubble Sort I



Idee:

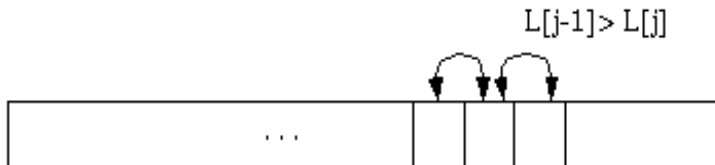
- Durchlaufe L und vertausche jeweils benachbarte Elemente, die nicht in Sortierordnung sind
(dabei steigen große Element immer weiter auf, “wie Blasen”)
- Wiederhole dieses Durchlaufen bis keine Vertauschung mehr nötig sind

Methode: “Sortieren durch lokale Vertauschung”

3 12 12 75 65 99

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

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”

3 12 12 75 65 99

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

Bubble Sort I



Idee:

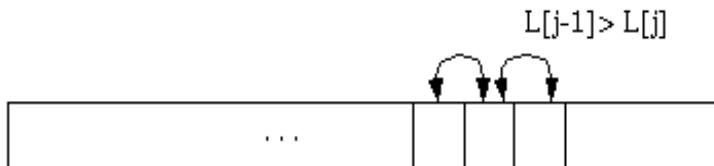
- Durchlaufe L und vertausche jeweils benachbarte Elemente, die nicht in Sortierordnung sind
(dabei steigen große Element immer weiter auf, “wie Blasen”)
- Wiederhole dieses Durchlaufen bis keine Vertauschung mehr nötig sind

Methode: “Sortieren durch lokale Vertauschung”

3 12 12 75 65 99

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

Bubble Sort I



Idee:

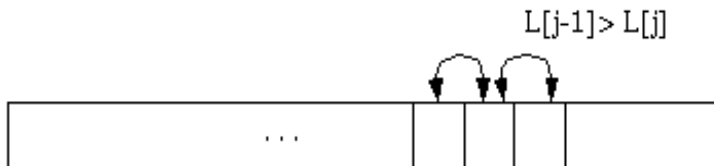
- Durchlaufe L und vertausche jeweils benachbarte Elemente, die nicht in Sortierordnung sind
(dabei steigen große Element immer weiter auf, “wie Blasen”)
- Wiederhole dieses Durchlaufen bis keine Vertauschung mehr nötig sind

Methode: “Sortieren durch lokale Vertauschung”

3 12 12 65 75 99

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

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)

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:

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

Anzahl Satzbewegungen (durch Swap):

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

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):

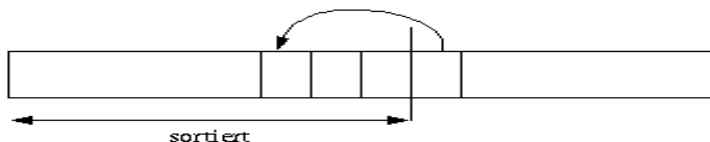
$$M_{\min}(n) = 0, M_{\max}(n) = 3n(n - 1)/2$$

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;
```


Sortieren durch Einfügen (Insertion Sort) I

27 | 75 99 3 45 12 87



Eigenschaften ?

Sortieren durch Einfügen (Insertion Sort) I

27 75 | 99 3 45 12 87



Eigenschaften ?

Sortieren durch Einfügen (Insertion Sort) I

27 75 99 | 3 45 12 87



Eigenschaften ?

Sortieren durch Einfügen (Insertion Sort) I

3 27 75 99 | 45 12 87



Eigenschaften ?

Sortieren durch Einfügen (Insertion Sort) I

3 27 45 75 99 | 12 87



Eigenschaften ?

Sortieren durch Einfügen (Insertion Sort) I

3 12 27 45 75 99 | 87



Eigenschaften ?

Sortieren durch Einfügen (Insertion Sort) I

3 12 27 45 75 87 99 |



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 $h_1 > h_2 > \dots > h_t = 1$. 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							

- 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 $t = \lceil \log_2 n \rceil - 1$
Dann: Aufwand $O(n^{1.2})$
- Andere Vorschläge erreichen $O(n \log n)$
mit Inkrementen der Form $2^p 3^q$