# Kap. 3: Sortieren

Professor Dr. Petra Mutzel

Lehrstuhl für Algorithm Engineering, LS11

Fakultät für Informatik, TU Dortmund

4. VO DAP2 SS 2009 23. April 2009

# Überblick

- Einführung in das Sortierproblem

- Insertion-Sort
- Selection-Sort
- Merge-Sort

# Motivation

„Warum soll ich hier bleiben?"

Sortierverfahren sind WICHTIG!!!

„Ich kann doch schon sortieren."

ABER ES GEHT SCHNELLER!

# „Unser“ Sortierproblem

**Eingabe:** Folge von Datensätzen $<s_1, s_2, \ldots, s_n>$ mit Schlüsseln $k_1, k_2, \ldots, k_n$, auf denen eine Ordnungsrelation „$\leq$“ definiert ist.

**Ausgabe:** Permutation $\pi : \{1,2,\ldots,n\} \xrightarrow{1:1} \{1,2,\ldots,n\}$, so dass die Umordnung der Datensätze gemäß $\pi$ die Schlüssel in aufsteigende Reihenfolge bringt:
$k_{\pi(1)} \leq k_{\pi(2)} \leq \ldots \leq k_{\pi(n)}$

Speicherung in Feld: $A[1],\ldots,A[n]$

Ansprechbar: Schlüssel: $A[i].key$

Informationsfeld: $A[i].info$

# Laufzeitmessung

- Anzahl der durchgeführten Schlüsselvergleiche („Comparisons") für Best-Case, Worst-Case und Average-Case:
- $C_{best}(n)$, $C_{worst}(n)$, $C_{avg}(n)$

- Anzahl der durchgeführten Bewegungen („Movements", Kopieren) von Datensätzen für Best-Case, Worst-Case und Average-Case:
- $M_{best}(n)$, $M_{worst}(n)$, $M_{avg}(n)$

# Eigenschaften von Sortierverfahren

- **Intern/Extern:** Geht das Verfahren davon aus, dass alle Daten im Hauptspeicher sind, dann → **intern**
- Manchmal müssen Daten aus Platzgründen ausgelagert werden (Platte); Verfahren, die hier gut geeignet sind → **extern**

- **In situ:** Benötigt ein Sortieralgorithmus zusätzlich zur Eingabe höchstens konstant viel zusätzlichen Speicherplatz, dann → **in situ**

# Eigenschaften von Sortierverfahren

- **Adaptiv:** Laufzeit abhängig von dem Grad der Vorsortierung der Daten; falls besser für vorsortierte Daten, dann → **adaptiv**

- **Stabil:** gleiche Reihenfolge von Datensätzen mit gleichem Schlüssel vor und nach dem Sortieren, dann → **stabil**

# 3.1 Allgemeine Sortierverfahren

- **Voraussetzung:** je zwei Schlüssel $k_i$ und $k_j$ sind vergleichbar, also entweder gilt $k_i \leq k_j$ oder $k_j \leq k_i$.

# 3.1.1 Insertion-Sort / Analyse

- **Anzahl der Schlüsselvergleiche:**

$C_{best}(n)=$

# InsertionSort(ref A)

Eingabe/Ausgabe: Zahlenfolge in Feld A[1..n]

```
for k:=2,…,n {
  key:=A[k]
  i:=k
  while i>1 and A[i−1]>key {
    A[i]:=A[i−1]
    i:=i−1
  }
  A[i]:=key
}
```

# 3.1.1 Insertion-Sort / Analyse

- **Anzahl der Schlüsselvergleiche (Z. 4):**

  $C_{best}(n)=\Theta(n)$ und $C_{avg}(n)=C_{worst}(n)=\Theta(n^2)$

- **Anzahl der Datenbewegungen (Z. 2,5,8):**

  $M_{best}(n)=\Theta(n)$ und $M_{avg}(n)=M_{worst}(n)=\Theta(n^2)$

- **Eigenschaften:**
  - in situ? ☺
  - adaptiv? ☺
  - stabil? ☺

# Inversionen

In einer Permutation $\pi = <\pi_1, \pi_2, \ldots, \pi_n>$ heißt ein Paar $(\pi_i, \pi_j)$ eine **Inversion**, wenn gilt: $i < j$ und $\pi_i > \pi_j$ .

- Die Anzahl der Inversionen einer Folge $\pi$ heißt **Inversionszahl** und ist ein Maß für die Vorsortierung einer Folge.

- **Es gilt:** Eine Folge ist sortiert g.d.w. die Anzahl ihrer Inversionen gleich 0 ist.
- Im schlimmsten Fall besitzt eine Folge $\sum_{i=1..n-1}(n-i)=\Theta(n^2)$ viele Inversionen.

# Beziehung zu InsertionSort(ref A)?

$s_k$:Anzahl der Durchführungen von (4)

```
(1) for k:=2,…,n {
(2)   key:=A[k]
(3)   i:=k
(4)   while i>1 and A[i−1]>key {
(5)      A[i]:=A[i−1]
(6)      i:=i−1
(7)   }
(8)   A[i]:=key
(9) }
```

| Zeit | Wie oft? |
|---|---|
| $t_1$ | n |
| $t_2$ | n-1 |
| $t_3$ | n-1 |
| $t_4$ | $\sum s_k$ |
| $t_5$ | $\sum(s_k-1)$ |
| $t_6$ | $\sum(s_k-1)$ |
| $t_7$ | $\sum(s_k-1)$ |
| $t_8$ | n-1 |
| $t_9$ | n-1 |

# Inversionen

- Die Anzahl der Schlüsselvergleiche und Datenbewegungen (und damit der Laufzeit) in InsertionSort hängt eng mit der Anzahl der Inversionen der Folge zusammen:

- Die Anzahl der Inversionen der Folge ist gleich $\sum_{k=2..n} (s_k-1)$.

- Die Laufzeit von Insertion Sort hängt direkt von der Anzahl der Inversionen der Folge ab.

# 3.1.2 Selection-Sort

**Idee** von „Sortieren durch Auswahl“:

- Bestimme Position $i_1 \in \{1,2,\ldots,n\}$ zu der das Element mit **minimalem** Schlüssel auftritt; vertausche $A[1]$ mit $A[i_1]$;
- Bestimme $i_2 \in \{2,\ldots,n\}$ …, vertausche A[2] mit $A[i_2]$; etc.

Beispiel: s. VO und Skript

# SelectionSort(ref A)

Eingabe/Ausgabe: Zahlenfolge in Feld A[1..n]

```
for j:=1,2,…,n-1 {
   minpos:=j
   for i:=j+1,…,n {
      if A[i].key < A[minpos].key then
         minpos:=i
   }
   if minpos > j then
      Vertausche A[minpos] mit A[j]
}
```

# Analyse von SelectionSort

- **Anzahl der Schlüsselvergleiche (Z. 4):**

  $C_{best}(n)=C_{avg}(n)=C_{worst}(n)=\Theta(n^2)$

denn:

$$C_{best}(n) = \sum_{j=1}^{n-1}(n-j) = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} = \theta(n^2)$$

- **Anzahl der Datenbewegungen (Z. 8):**

  $M_{best}(n)=0$

  $M_{avg}(n)=M_{worst}(n)=\Theta(n)$

# Eigenschaften von SelectionSort

- **Eigenschaften:**
  - in situ ?
  - adaptiv ?
  - stabil ?

- **Einsatz von SelectionSort, wenn:**
  - Bewegungen von Datensätzen teuer
  - Vergleiche zwischen Schlüsseln billig

# 3.1.3 Merge-Sort

Idee folgt dem „Divide and Conquer“-Prinzip („Teile und Erobere“):

- **Teile** das Problem in Teilprobleme auf.
- **Erobere** die Teilprobleme durch rekursives Lösen. Wenn sie klein genug sind, löse sie direkt.
- **Kombiniere** die Lösungen der Teilprobleme zu einer Lösung des Gesamtproblems.

wichtiges Algorithmen-Entwurfsprinzip!

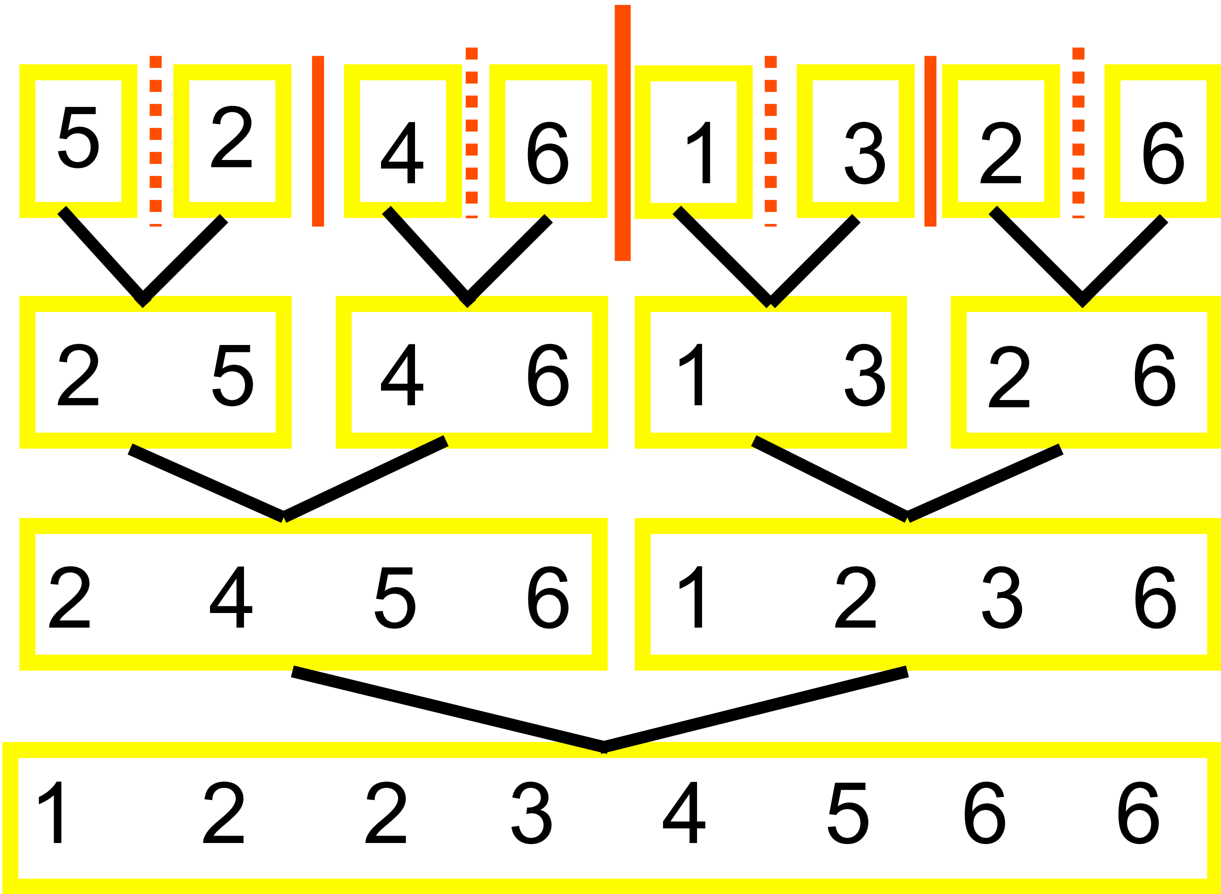
# 3.1.3 Merge-Sort

Idee von „Sortieren durch Mischen“:

- **Teile:** Teile die Folge in der Mitte in zwei Teilfolgen.
- **Erobere:** Sortiere beide Teilfolgen rekursiv. Für 1-elementige Teilfolgen ist nichts zu tun.
- **Kombiniere:** Verschmelze die sortierten Teilfolgen zu einer Gesamtfolge.

**Merge Sort:** älteste für den Computer entwickelte Sortieralgorithmus (John von Neumann 1945)

# Ablauf von MergeSort



# MergeSort(ref A,l,r)

```
procedure MergeSort(ref A,l,r)
•     var Index m
•     if l < r then {
(1)     m:=⌊(l+r)/2⌋
(2)     MergeSort(A,l,m)
(3)     MergeSort(A,m+1,r)
(4)     Merge(A,l,m,r)
(5) }
```

Aufruf: MergeSort(A,1,n)

# Merge(ref A,l,m,r)

```
procedure Merge(ref A,l,m,r)
  i:=l; j:=m+1
  for k:=l,…,r {
     if (i>m) or ((j≤r) and (A[i].key>A[j].key))
        then
          B[k]:=A[j]; j:=j+1
     else
          B[k]:=A[i]; i:=i+1
  }
Schreibe sort. Folge zurück von B nach A
```

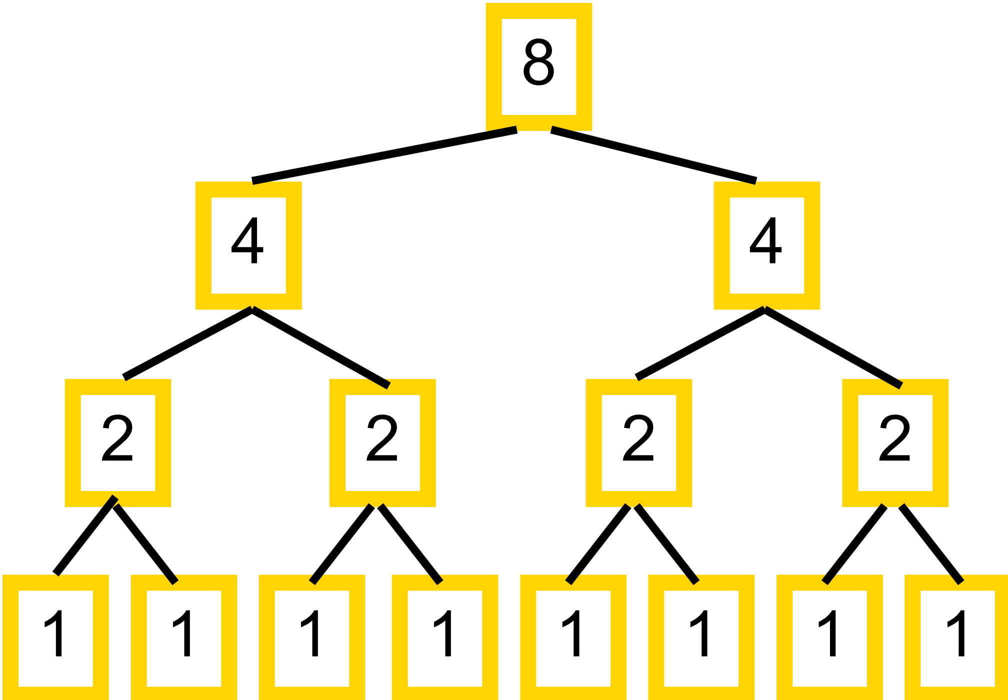
Rekursive Aufrufe unseres Beispiels

MergeSort(A,1,8)
- MergeSort(A,1,4)
  - MergeSort(A,1,2)
    - MergeSort(A,1,1)
    - MergeSort(A,2,2)
    - Merge(A,1,1,2)
  - MergeSort(A,3,4)
    - MergeSort(A,3,3)
    - MergeSort(A,4,4)
    - Merge(A,3,3,4)
  - Merge(A,1,4,2) ….u.s.w.

# Herleitung der Laufzeitfunktion

Sei $n=2^k$ für ein beliebiges $k$

weiter: s. nächste




| Anzahl der Instanzen | Zeit pro Instanz | Gesamtzeit |
|---|---|---|
| 1 | 8 | 8 |
| 2 | 4 | 8 |
| 4 | 2 | 8 |
| 8 | 1 | 8 |

Aufwand in jeder Stufe gleich $n=2^k$.

Es gibt $k+1=\log n + 1$ solcher Stufen

Gesamtaufwand:
$T(n) = n\,(1 + \log n) =$
$= n + n \log n = \Theta(n \log n)$