# Informatik II - SS 2018

## (Algorithmen & Datenstrukturen)

Vorlesung 4 (30.4.2018)

Sortieren IV

Fabian Kuhn

Algorithmen und Komplexität

UNI FREIBURG

# Analyse Merge Sort

Laufzeit $T(n)$ setzt sich zusammen aus:

- Divide und Merge: $O(n)$
- 2 rekursive Aufrufe zum Sortieren von $\lceil n/2 \rceil$ und $\lfloor n/2 \rfloor$ Elementen

**Rekursive Formulierung von $T(n)$:**

- Es gibt eine Konstante $b > 0$, so dass

$$T(n) \leq T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + b \cdot n, \qquad \mathrm{T}(1) \leq b$$

- Wir machen uns das Leben ein bisschen einfacher und ignorieren das Auf- und Abrunden:

$$\boldsymbol{T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + b \cdot n, \qquad T(1) \leq b}$$

# Analyse Merge Sort

$$\boldsymbol{T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + b \cdot n, \; T(1) \leq b}$$

Setzen wir einfach mal ein, um zu sehen, was rauskommt…

$$\begin{aligned}
T(n) &\leq 2T\left(\frac{n}{2}\right) + bn \leq 2\left(2T\left(\frac{n}{4}\right) + b\frac{n}{2}\right) + bn \\
&= 4T\left(\frac{n}{4}\right) + 2bn \\
&\leq 8T\left(\frac{n}{8}\right) + 3bn \\
&\vdots \\
&\leq 2^k T\left(\frac{n}{2^k}\right) + kbn \\
&\leq n \cdot T(1) + bn \cdot \log_2 n \\
&\leq bn \cdot (1 + \log_2 n)
\end{aligned}$$

# Analyse Merge Sort

**Rekursionsgleichung:** $T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + b \cdot n,\ T(1) \leq b$

**Vermutung:** $T(n) \leq b \cdot n \cdot (1 + \log_2 n)$

**Beweis durch vollständige Induktion:**

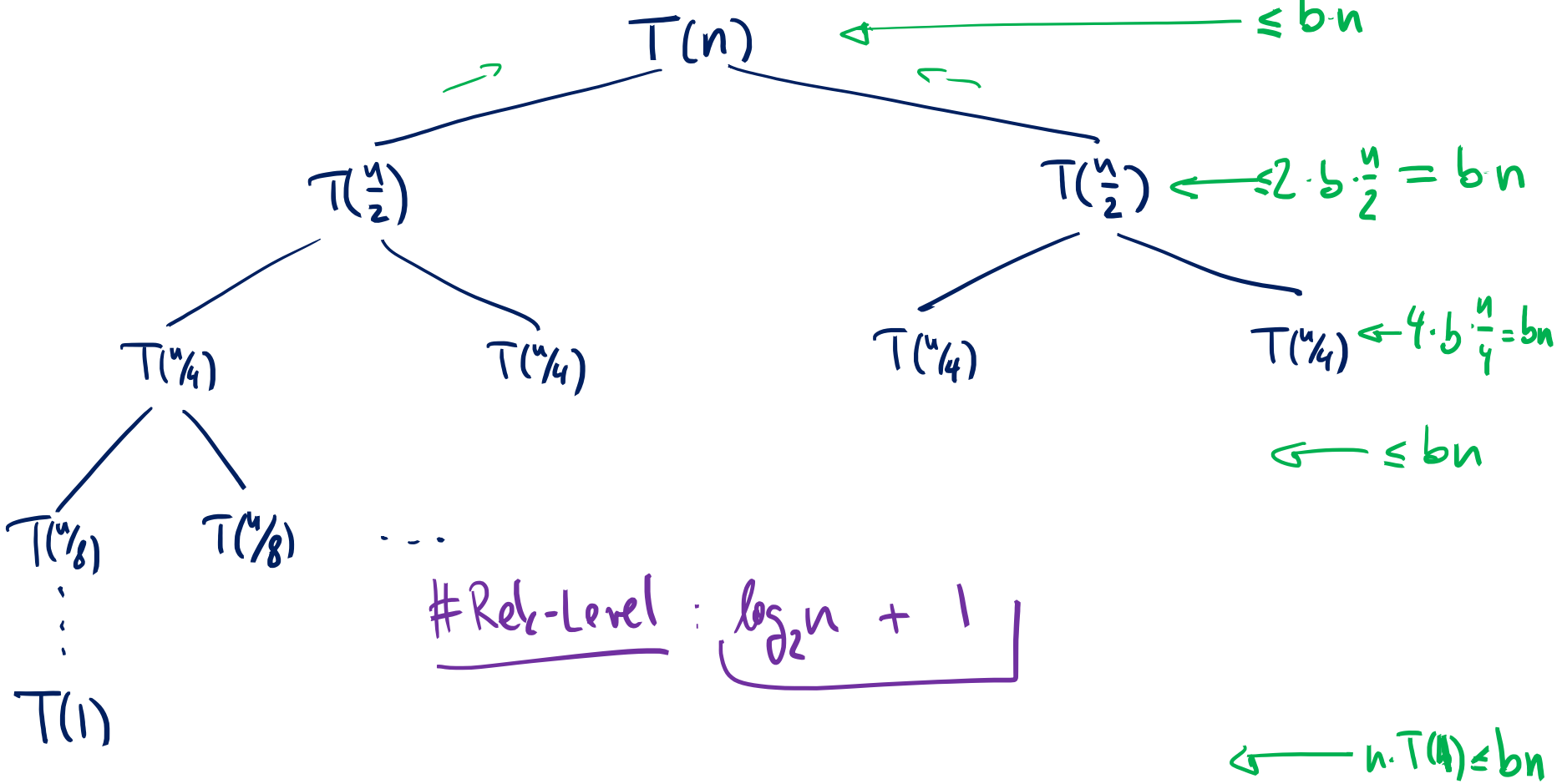
- *Verankerung:* $n = 1 \Longrightarrow T(1) \leq b \cdot 1 \cdot (1 + 0) = b$
- *Induktionsschritt:*

$$
\begin{aligned}
T(n) &\leq 2T\left(\frac{n}{2}\right) + bn \\
&\overset{(I.A.)}{\leq} 2 \cdot \left(b \cdot \frac{n}{2} \cdot \left(1 + \log_2 \frac{n}{2}\right)\right) + bn \\
&= bn \cdot \log_2 n + bn \\
&= bn \cdot (1 + \log_2 n)
\end{aligned}
$$

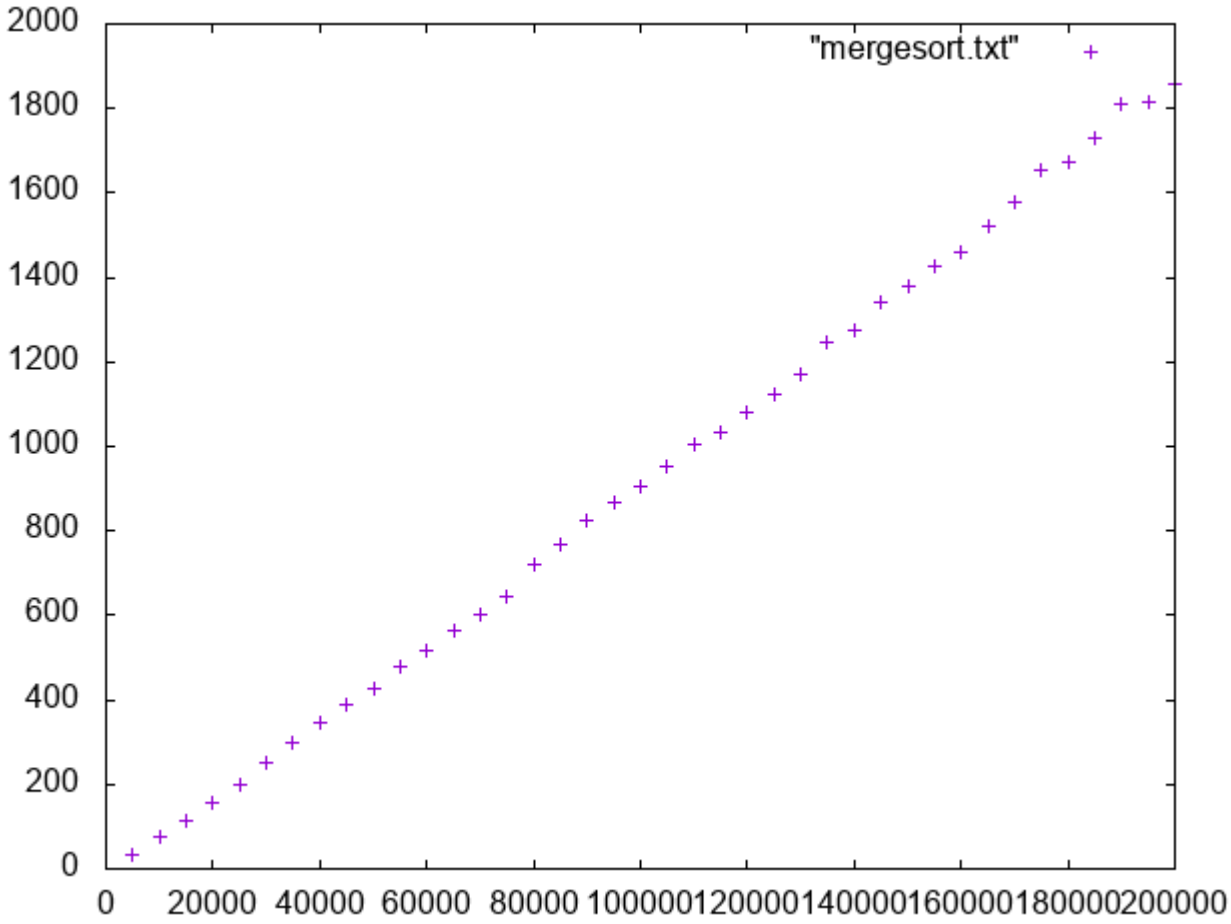
# Alternative Analyse Merge Sort

**Rekursionsgleichung:** $T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + b \cdot n, \; T(1) \leq b$

Betrachten wir den Rekursionsbaum:



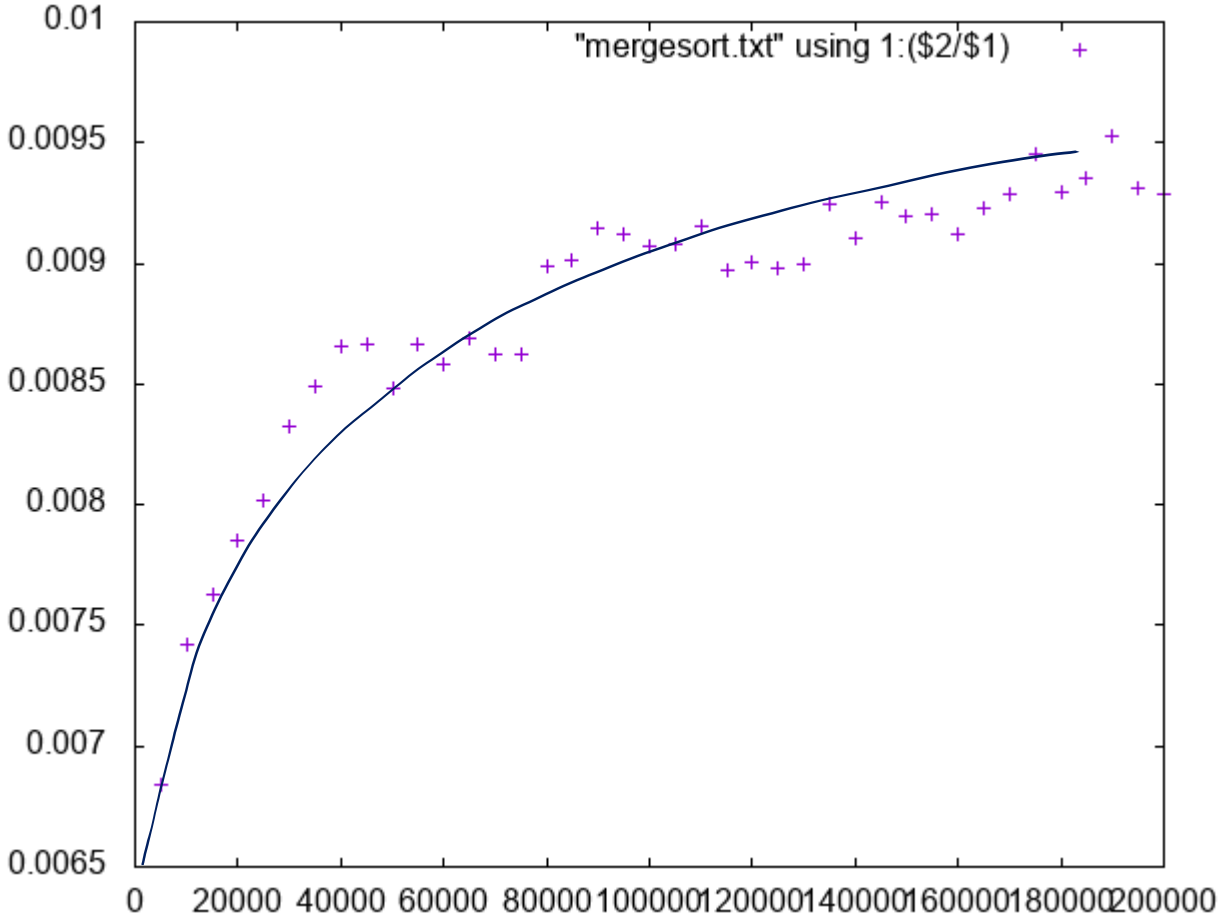
# Merge Sort Messungen




**Gnuplot:**

```
set term png
set output “mergesort.png”
plot “mergesort.txt”
```

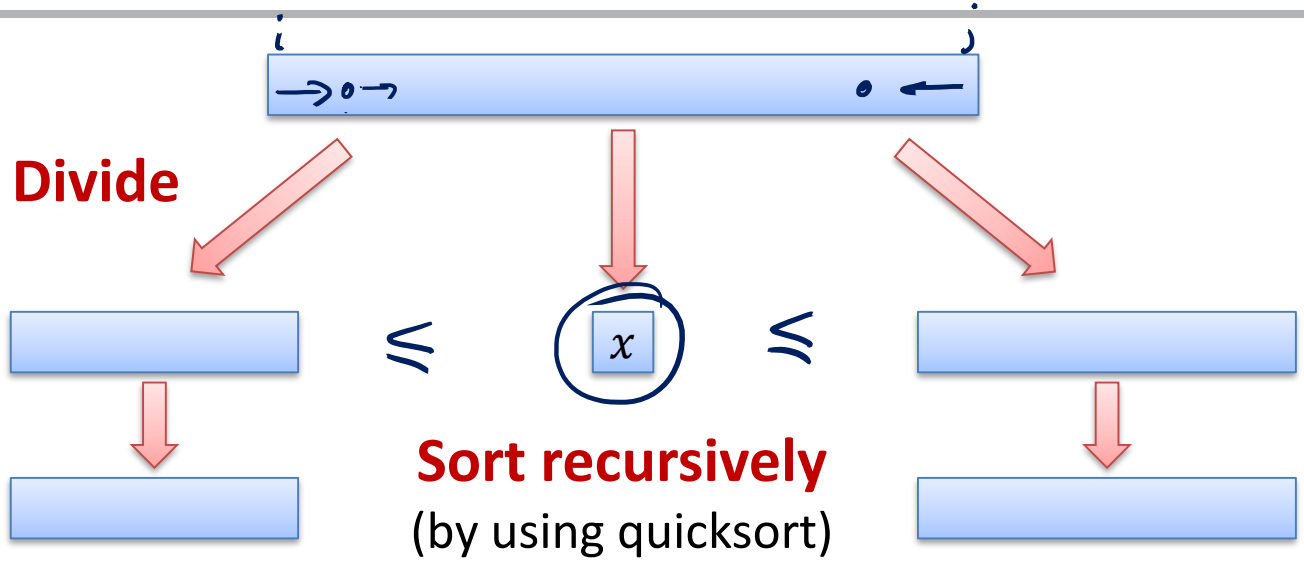
# Merge Sort Messungen




**Gnuplot:**

```
set term png
set output “mergesort2.png”
plot “mergesort.txt” using 1:($2:$1)
```

# Quick Sort : Analyse




- Laufzeit hängt davon ab, wie gut die Pivots sind
- Laufzeit, um Array der Länge $n$ zu sortieren, falls das Pivot in Teile der Grösse $\lambda n$ und $(1-\lambda)n$ partitioniert:

$$\boldsymbol{T(n) = T(\lambda n) + T\big((1-\lambda)n\big) + \text{"Pivotsuche + Divide"}}$$

$O(n)$

- **Divide:**
  - Wir gehen einmal von beiden Seiten über's Array mit konstanten Kosten pro Schritt → Zeit, um Array der Länge $n$ zu partitionieren: $O(n)$

# Quick Sort : Analyse

Falls wir in $O(n)$ Zeit ein Pivot finden können, welches das Array in Teile der Grösse $\lambda n$ und $(1-\lambda)n$ unterteilt:

- Es gibt eine Konstante $b > 0$, so dass

$$\boxed{T(n) \leq T(\lambda n) + T\big((1-\lambda)n\big) + b \cdot n, \qquad T(1) \leq b}$$

**Extremfall I)** $\lambda = 1/2$ (best case):

$$T(n) \leq 2T\left(\frac{n}{2}\right) + bn, \qquad T(1) \leq b \quad \Rightarrow T(n) = \Theta(n \log n)$$

- Wie bei Merge Sort: $T(n) \in O(n \log n)$

**Extremfall II)** $\lambda n = 1$, $(1-\lambda)n = n-1$ (worst case):

$$T(n) = T(n-1) + bn, \qquad T(1) \leq b$$

eigentlich Teile der Gr. 0 & n-1

# Quick Sort : Worst Case Analyse

**Extremfall II)** $\lambda n = 1, (1-\lambda)n = n-1$ (worst case):

$$T(n) = T(n-1) + bn, \qquad \underline{T(1) \leq b}$$

In dem Fall, ergibt sich $T(n) \in \Theta(n^2)$:

$$\begin{aligned}
T(n) &= T(n-1) + bn \\
&= T(n-2) + b(n-1) + bn \\
&= T(n-3) + b(n-2 + n-1 + n) \\
&\vdots \\
&= T(n-k) + b((n-k+1) + (n-k+2) + \ldots + n) \\
&\vdots \\
&= T(1) + b(2 + 3 + \ldots + n) \\
&\leq b(1 + \ldots + n) = \underbrace{b\frac{n(n+1)}{2}} = \Theta(n^2)
\end{aligned}$$

Vermutung $T(n) \leq b\frac{n(n+1)}{2}$

Verankerung $(n=1)$: $T(1) \leq b\frac{1 \cdot 2}{2} = b$ ✓

Schritt:

$$\begin{aligned}
T(n) &\leq T(n-1) + bn \\
&\overset{(I.A.)}{\leq} b\frac{(n-1)n}{2} + b \cdot n \\
&= b\frac{n(n+1)}{2} \quad ✓
\end{aligned}$$

# Quick Sort mit zufälligem Pivot

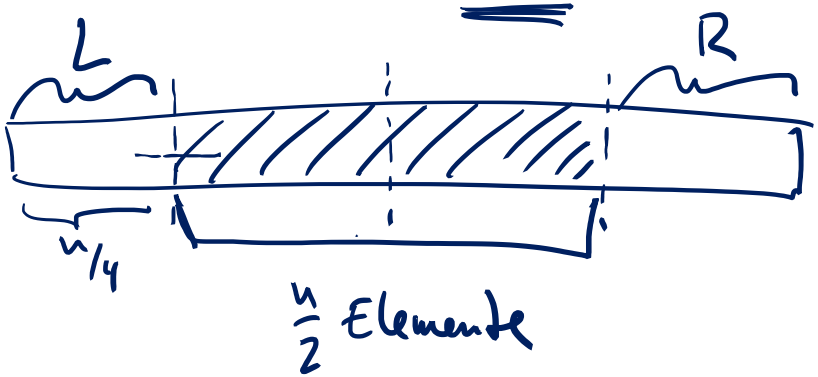
**Aufteilung bei zufälligem Pivot:**

- Laufzeit $\boldsymbol{T(n) = O(n \log n)}$ für alle Eingaben
  - allerdings nur im Erwartungswert, bzw. mit sehr grosser Wahrscheinlichkeit

**Intuition:**

- Mit Wahrscheinlichkeit $1/2$, haben die Teile Grösse $\geq n/4$, so dass

$$T(n) \quad \leq \quad T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + bn$$



# Quick Sort mit zufälligem Pivot

**Aufteilung bei zufälligem Pivot:**

- Laufzeit $\boldsymbol{T(n) = O(n \log n)}$ für alle Eingaben
  - allerdings nur im Erwartungswert, bzw. mit sehr grosser Wahrscheinlichkeit

**Analyse:**

- Werden wir hier nicht tun
  - siehe z.B. Cormen et al. oder die Algorithmentheorie-Vorlesung
- Mögl. Vorgehen, Rekursion mit Erwartungswerten hinschreiben:

$$\mathbb{E}[T(n)] \leq \mathbb{E}[T(N_L) + T(n - N_L)] + bn$$

# Sortieren : Untere Schranke

**Aufgabe:** Sortiere Folge $a_1, a_2, \dots, a_n$

- Ziel: benötigte (worst-case) Laufzeit nach unten beschränken

**Vergleichsbasierte Sortieralgorithmen**

- Vergleiche sind die einzige erlaubte Art, die relative Ordnung von Elementen zu bestimmen
- Das heisst, das Einzige, was die Reihenfolge der Elemente in der sortierten Liste beeinflussen kann, sind Vergleiche der Art

$$a_i = a_j, a_i \leq a_j, a_i < a_j, a_i \geq a_j, a_i > a_j$$

- Nehmen wir an, die Elemente sind paarweise verschieden, dann reichen Vergleiche der Art $a_i \leq a_j$
- 1 solcher Vergleich ist eine Grundoperation

# Vergleichsbasierte Sortieralgorithmen

UNI FREIBURG

**Alternative Sichtweise**

- Jedes Programm (für einen deterministischen, vgl.-basierten Sortieralg.) kann in eine Form gebracht werden, in welcher jede if/while/…-Bedingung von folgender Form ist:

$$\textbf{if } (a_i \leq a_j) \textbf{ then } \ldots$$

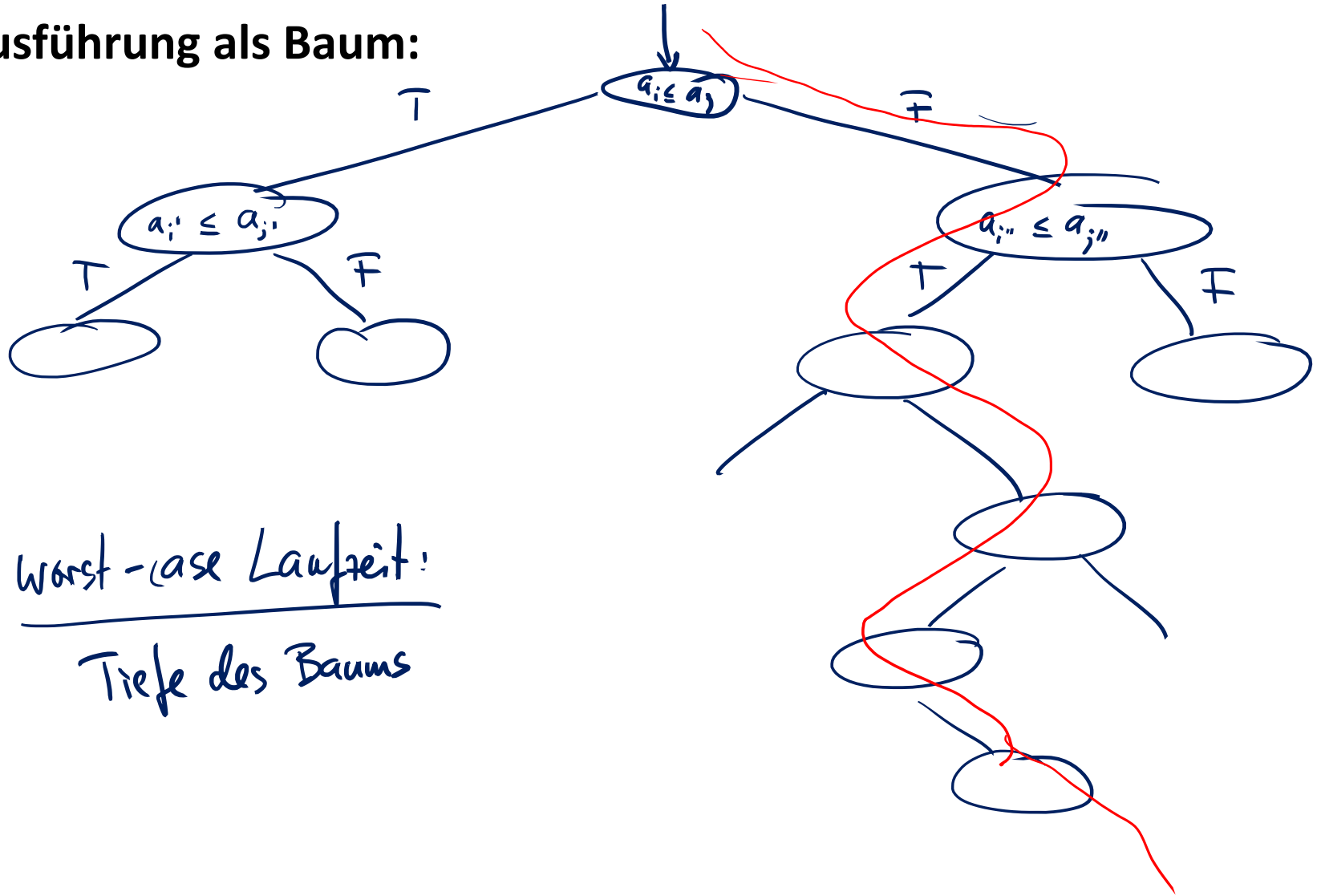
- In jeder Ausführung eines Algorithmus, induzieren die Resultate dieser Vergleiche eine Abfolge von $\mathrm{T}/\mathrm{F}$ (true/false) Werten:

$$\mathbf{TFFTTTFTFFTTFFFFFTFTTT} \ldots$$

- Diese Abfolge bestimmt in eindeutiger Weise, wie die Elemente umgeordnet werden.
- Unterschiedliche Eingaben der gleichen Werte, müssen daher zu unterschiedlichen $\mathrm{T}/\mathrm{F}$-Sequenzen führen!

# Vergleichsbasierte Sortieralgorithmen

**Ausführung als Baum:**




Worst-case Laufzeit:

Tiefe des Baums

# Vgl.-Basiertes Sortieren : Untere Schranke I

- Bei vergleichsbasierten Sortieralgorithmen hängt die Ausführung nur von der Ordnung der Werte in der Eingabe, nicht aber von den eigentlichen Werten ab
  - Wir beschränken und auf Eingaben, bei denen die Werte unterschiedlich sind.
- O.b.d.A. können wir deshalb annehmen, dass wir die Zahlen $1, \dots, n$ sortieren müssen.
- Verschiedene Eingaben müssen verschieden bearbeitet werden.
- Verschiedene Eingaben erzeugen verschiedene $\mathrm{T/F}$-Folgen
- Laufzeit einer Ausführung $\geq$ Länge der erzeugten $\mathrm{T/F}$-Folge
- Worst-Case Laufzeit $\geq$ Länge der längsten $\mathrm{T/F}$-Folge:
  - Wir wollen eine untere Schranke
  - Zählen der Anz. mögl. Eingaben → wir benötigen so viele $\mathrm{T/F}$-Folgen...

# Vgl.-Basiertes Sortieren : Untere Schranke I

Anzahl Mögliche Eingaben (Anfangsreihenfolgen):

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

$$n! \geq \underbrace{n \cdot (n-1) \cdot \ldots \cdot \lceil \tfrac{n}{2} \rceil}_{\geq \frac{n}{2} \text{ Faktoren}}$$

Anzahl T/F-Folgen der Länge $\leq k$:

Länge $= k$ : $2^k$

T/F T/F . . . . . .

1 2 k

$$2^k + 2^{k-1} + 2^{k-2} + \ldots + 1 \leq 2^{k+1}$$

**Theorem:** Jeder det. Vergleichs-basierte Sortieralgorithmus benötigt im Worst Case mindestens $\Omega(n \cdot \log n)$ Vergleiche.

$\log(a^b) = b \cdot \log a$

Laufzeit $\leq T$

$$2^{T+1} \geq n!$$

$$T+1 \geq \log_2(n!)$$

$$T \in \Omega(\log(n!))$$

$$\in \Omega(n \log n)$$

$$\left(\frac{n}{2}\right)^{\left(\frac{n}{2}\right)} \leq n! \leq n^n$$

$$\frac{n}{2} \cdot \log\left(\frac{n}{2}\right) \leq \log(n!) \leq n \cdot \log n$$

$$\log(n!) = \Theta(n \log n)$$

# Sortieren in Linearzeit

- Mit Vergleichs-basierten Algorithmen nicht möglich
  - Untere Schranke gilt auch mit Randomisierung...

- Manchmal geht's schneller
  - wenn wir etwas über die Art der Eingabe wissen und ausnützen können

- Beispiel: Sortiere $n$ Zahlen $a_i \in \{0,1\}$:
  1. Zähle Anzahl Nullen und Einsen in $O(n)$ Zeit
  2. Schreibe Lösung in Array in $O(n)$ Zeit

# Counting Sort

**Aufgabe:**

- Sortiere Integer-Array $A$ der Länge $n$
- Wir wissen, dass für alle $i \in \{0, \dots, n-1\}$, $A[i] \in \{0, \dots, k\}$

counts[x] += 1

**Algorithmus:**

```
counts = new int[k+1]        // new int array of length k
for i = 0 to k do counts[i] = 0
for i = 0 to n-1 do counts[A[i]]++
i = 0;
for j = 0 to k do
   for l = 1 to counts[j] do
      A[i] = j; i++
```

# Analyse Counting Sort

```
counts = new int[k+1]
for i = 0 to k do counts[i] = 0
for i = 0 to n-1 do counts[A[i]]++
i = 0;
for j = 0 to k do
   for l = 1 to counts[j] do
      A[i] = j; i++
```

$O(1)$, $O(k)$

$O(k)$

$O(n)$

$$\leq \sum_{j=0}^{k} c \cdot (counts[j] + 1)$$

$$= c \cdot (n + k + 1)$$

$$= O(n+k)$$

Laufzeit: $O(n+k)$

linear in $n$ falls $k = O(n)$

# Stabiles Sortieren

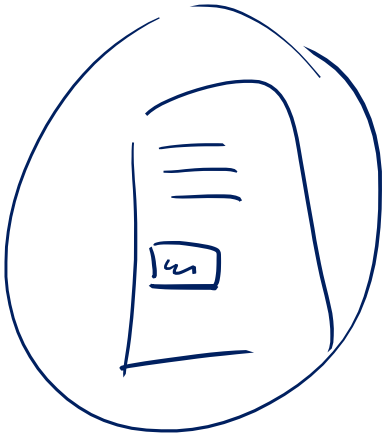
**Definition:** Ein Sortieralgorithmus ist stabil, falls Elemente mit gleichem Schlüssel in der gleichen Reihenfolge bleiben.

**Beispiel:**

Sortiere folgende Strings stabil nach Anfangsbuchstabe:

“tuv”, “adr”, “bbc”, “tag”, “taa”, “abc”, “sru”, “bcb”

adr, abc, bbc, bcb, sru, tuv, tag, taa



# Stabiles Sortieren

Welche der in der Vorlesung behandelten Sortieralgorithmen sind (so, wie wir sie besprochen haben) stabil?

selection sort X

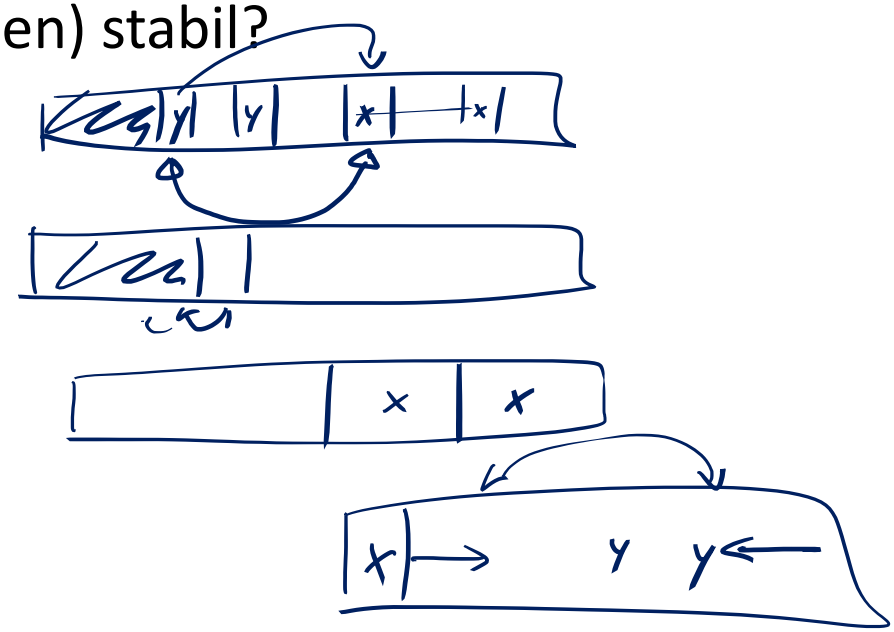
ins. sort ✓

bubble sort ✓

merge sort ✓

quick sort X

counting sort (✓)



# Stabiles Sortieren: Anwendung

**Gegeben:** Liste von Paaren

- z.B.: (7,5), (2,8), (3,4), (3,1), (2,4), (2,6), (7,9), (3,8), (7,1)

**Ziel:** Sortiere die Liste lexikographisch

- lexikographisch: zuerst nach der ersten Zahl und falls die erste Zahl gleich ist, nach der zweiten Zahl

2 x nacheinander sortieren

1) sortiere nach der zweiten Zahl

(3,1), (7,1), (3,4), (2,4), (7,5), (2,6), (2,8), (3,8), (7,9)

2) sortiere stabil nach der ersten Zahl

(2,4), (2,6), (2,8), (3,1), (3,4), (3,8), (7,1), (7,5), (7,9)

# Sortieren in Linearzeit II

**Counting Sort:**

- sortiert ganze Zahlen zwischen 0 und $O(n)$ in Linearzeit

Was, wenn die Zahlen aus einem grösseren Bereich sind?

- z.B.: ganze Zahlen zwischen $\underline{0}$ und $\underline{n^2 - 1}$?

$x = a \cdot n + b$, so dass $a, b \in \{0, \dots, n-1\}$

$x_1 = a_1 n + b_1$

$x_2 = a_2 n + b_2$

$x_1 < x_2 \iff (a_1, b_1) < (a_2, b_2)$

lexikogr.

# Sortieren in Linearzeit II

**Sortiere Werte zwischen $\mathbf{0}$ und $\boldsymbol{n^2 - 1}$:**

**Ziel:** Sortiere $x_0, x_1, x_2, \dots, x_{n-1}$, wobei $x_i \in \{0, \dots, n^2 - 1\}$

**Algorithmus:**

1. Schreibe alle Zahlen als $x_i = a_i \cdot n + b_i$ für $a_i, b_i \in \{0, \dots, n-1\}$
2. Sortiere Zahlen mit Counting Sort nach $b_i$
3. Sortiere Zahlen **stabil** mit Counting Sort nach $a_i$

# Radixsort

$x_i \geq 0$

**Verallgemeinerung der Idee**

- **Ziel:** sortiere (nichtnegative) ganze Zahlen $x_0, \dots, x_{n-1}$
- Schreibe Zahlen als $x_i = a_{i,0} + a_{i,1} \cdot n + a_{i,2} \cdot n^2 + \cdots + a_{i,k} \cdot n^k$

**Algorithmus:**

1. Sortiere mit Counting Sort nach $a_{i,0}$
2. Sortiere mit Counting Sort stabil nach $a_{i,1}$
3. Sortiere mit Counting Sort stabil nach $a_{i,2}$
4. …
5. Sortiere mit Counting Sort stabil nach $a_{i,k}$

# Radixsort

Sortiere $x_0, \dots, x_{n-1}$, mit $x_i = a_{i,0} + a_{i,1}n + a_{i,2}n^2 + \cdots + a_{i,k}n^k$

**Algorithmus:**

1. Sortiere mit Counting Sort nach $a_{i,0}$
2. Sortiere mit Counting Sort stabil nach $a_{i,1}$
3. …
4. Sortiere mit Counting Sort stabil nach $a_{i,k}$

max. Wert: $X$

Laufzeit: $O(k \cdot n)$, $k = \lceil \log_n X \rceil$

$$O(n \cdot \log_n X) = O\left(n \cdot \frac{\log X}{\log n}\right)$$

$$X \leq n^c$$

# Zusammenfassung Sortieralgorithmen

- **Selection Sort, Insertion Sort, Bubble Sort**
  worst case: $\Theta(n^2)$
- **Merge Sort**
  worst case (und best case): $\Theta(n \log n)$
- **Quick Sort**
  worst case (fixed pivot): $\Theta(n^2)$, average case: $O(n \log n)$
  worst case, randomized: $O(n \log n)$
- **Radix Sort** (für positive ganze Zahlen)
  worst case: $O(n \log_n M)$, wobei $M$ die grösste Zahl ist
- **Lower Bound**
  "comparison-based" Algorithmen: $\Omega(n \log n)$