Informatik II - SS 2018

(Algorithmen & Datenstrukturen)

Vorlesung 4 (30.4.2018)

Sortieren IV

UNI FREIBURG

Fabian Kuhn

Algorithmen und Komplexität

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:*

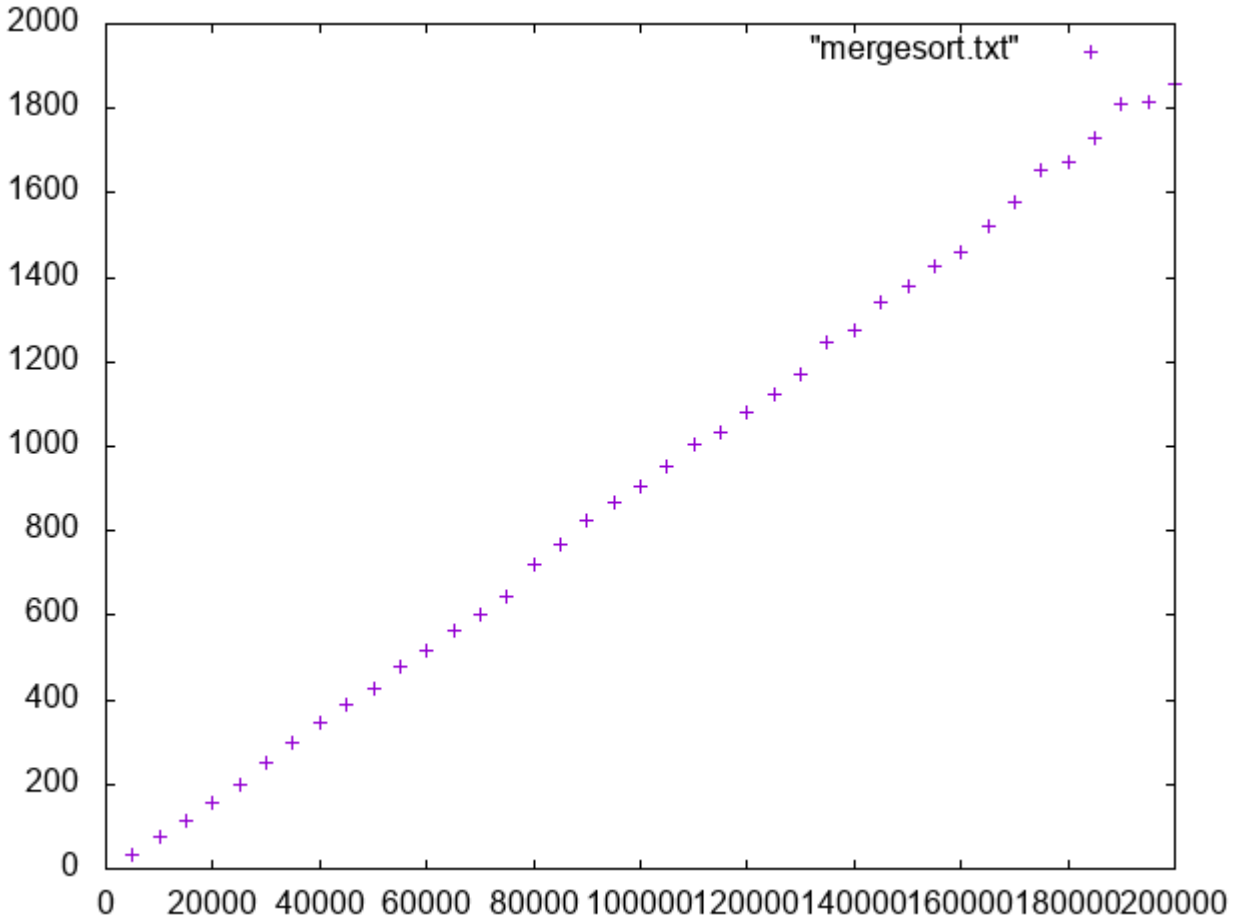
$$\begin{aligned} T(n) &\leq 2T\left(\frac{n}{2}\right) + bn \\ &\leq 2 \cdot \left(b \cdot \frac{n}{2} \cdot \left(1 + \log_2 \frac{n}{2}\right)\right) + bn \\ &= bn \cdot \log_2 n + bn \\ &= bn \cdot (1 + \log_2 n) \end{aligned}$$

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:

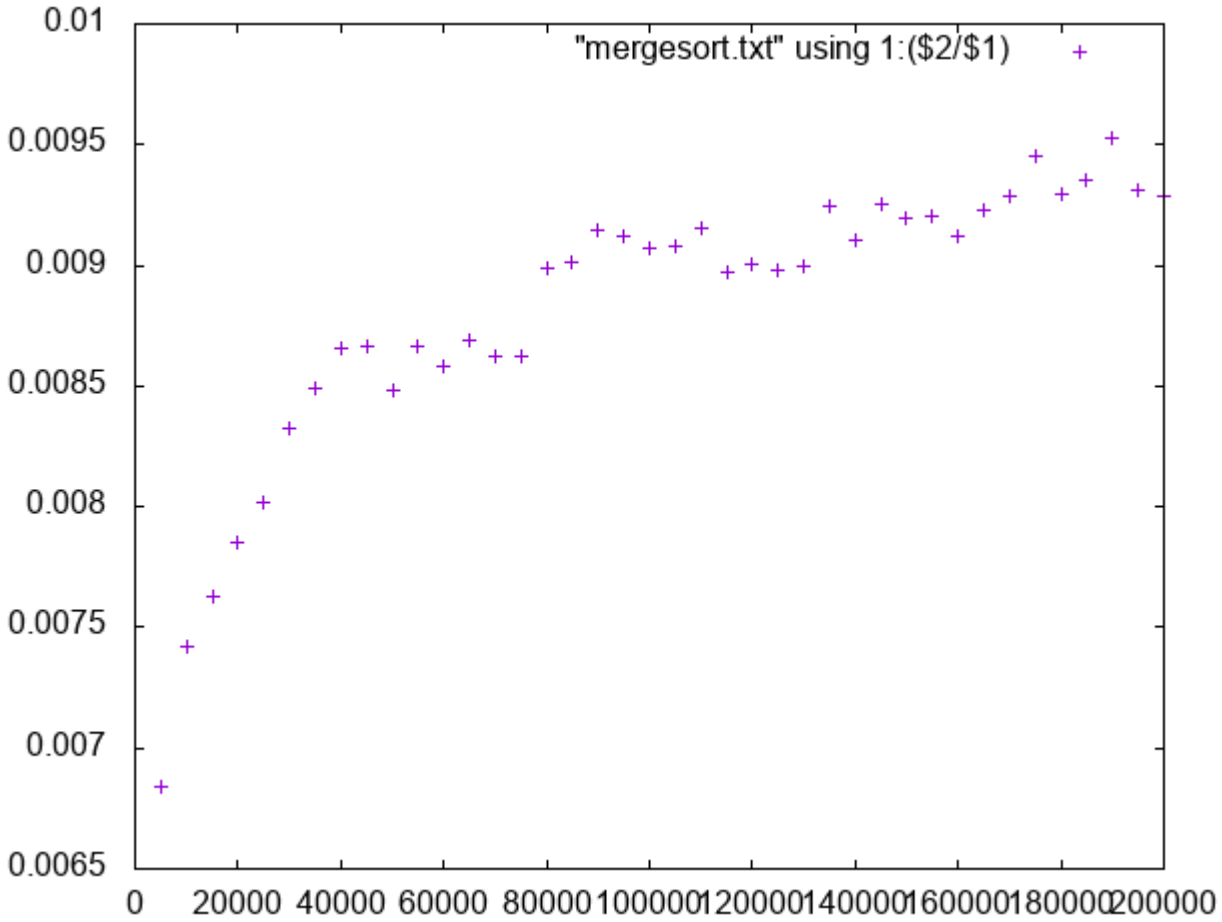
Merge Sort Messungen



Gnuplot:

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

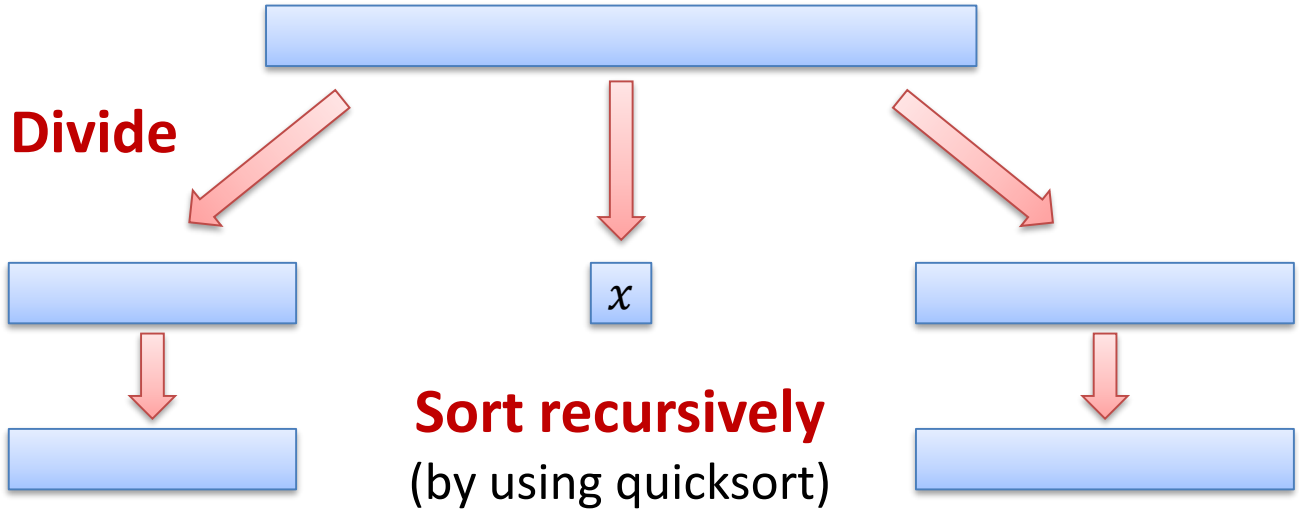
Merge Sort Messungen



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 λn und $(1-\lambda)n$ partitioniert:

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

- **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 λn und $(1-\lambda)n$ unterteilt:

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

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

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

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

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

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

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

Quick Sort : Worst Case Analyse

UNI FREIBURG

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

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

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

Quick Sort mit zufälligem Pivot

UNI FREIBURG

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

UNI FREIBURG

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

UNI FREIBURG

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:

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

- In jeder Ausführung eines Algorithmus, induzieren die Resultate dieser Vergleiche eine Abfolge von T/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 T/F-Sequenzen führen!

Vergleichsbasierte Sortieralgorithmen

UNI FREIBURG

Ausführung als Baum:

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 T/F-Folgen
- Laufzeit einer Ausführung $\geq$ Länge der erzeugten T/F-Folge
- Worst-Case Laufzeit $\geq$ Länge der längsten T/F-Folge:
  - Wir wollen eine untere Schranke
  - Zählen der Anz. mögl. Eingaben → wir benötigen so viele T/F-Folgen...

Vgl.-Basiertes Sortieren : Untere Schranke I

Anzahl Mögliche Eingaben (Anfangsreihenfolgen):

Anzahl T/F-Folgen der Länge $\leq k$:

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

Sortieren in Linearzeit

UNI FREIBURG

- 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 $i \in \{0, \dots, n\}$, $A[i] \in \{0, \dots, k\}$

Algorithmus:

```
1: counts = new int[k+1]          // new int array of length k
2: for i = 0 to k do counts[i] = 0
3: for i = 0 to n-1 do counts[A[i]]++
4: i = 0;
5: for j = 0 to k do
6:    for l = 1 to counts[j] do
7:       A[i] = j; i++
```

Analyse Counting Sort

```
1: counts = new int[k+1]
2: for i = 0 to k do counts[i] = 0
3: for i = 0 to n-1 do counts[A[i]]++
4: i = 0;
5: for j = 0 to k do
6:   for l = 1 to counts[j] do
7:     A[i] = j; i++
```

Stabiles Sortieren

UNI FREIBURG

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”

Stabiles Sortieren

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

Stabiles Sortieren: Anwendung

UNI FREIBURG

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

Sortieren in Linearzeit II

UNI FREIBURG

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 0 und $n^2 - 1$?

Sortieren in Linearzeit II

UNI FREIBURG

Sortiere Werte zwischen 0 und $n^2 - 1$:

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

UNI FREIBURG

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 $a_{i,k}$

Zusammenfassung Sortieralgorithmen

UNI FREIBURG

- **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: $\Omega(n \log n)$