Informatik II - SS 2016

(Algorithmen & Datenstrukturen)

Vorlesung 4 (29.4.2016)

Asymptotische Analyse, Sortieren IV

UNI FREIBURG

Fabian Kuhn

Algorithmen und Komplexität

Erfahrungen 1. Übung

UNI FREIBURG

- Einige haben viel Zeit für SVN, daphne, flake8, Makefiles, etc. gebraucht
  - Die Übung war unter anderem dazu da, dass Sie sich mit all den Tools auseinandersetzten und “anfreunden” können
- Gewisse verstehen nicht, was SVN und Jenkins sind
  - Alex Saukh bietet heute um 16:15 im Raum 106-00-007 ein kurzes Tutorial an, um diese Dinge nochmals vorzuzeigen
- Durchnittliche Bearbeitungszeit $\approx 7h$
  - ca 9-10 Stunden pro Übungsblatt sind normal
  - **Aber:** Wenden Sie sich bei Problemen frühzeitig ans Forum!
- Zeitmessung und Schaubilder waren zeitaufwendig
  - Sehe ich ein, das war vermutlich eine einmalige Sache

Erfahrungen 1. Übung

UNI FREIBURG

Probleme bei der Implementierung von Quicksort

- Das richtige Abbruchkriterium beim Divide zu finden war tricky

Problem mit Python

- RuntimeError: maximum recursion depth exceeded in comparison
  - Trat bei Quicksort auf, wenn man das erste Element als Pivot nimmt und das Array absteigend vorsortiert ist.
  - Lösung:
    import sys
    sys.setrecursionlimit(10000)

Bemerkungen 1. Übung

UNI FREIBURG

Hinweise zu Daphne, etc. auf Webseite:

http://ac.informatik.uni-freiburg.de/teaching/ss_16/info2/InfoUebungen.php

- Einfachste Lösung: Vorlage aus der Vorlesung übernehmen und anpassen
  - Makefile / build.xml einfach übernehmen und jedesmal verwenden
  - Für Java: build.properties jeweils kurz anpassen
  - Für Übung 1 dann einfach Quicksort-Code hinzufügen und Main-File anpassen
  - Wir werden im public-Folder eine Musterlösung bereitstellen:
    Auch wenn Sie die Aufgabe nicht gelöst haben, versuchen Sie die Vorlesungsbeispiele und die Musterlösung aus dem SVN (public-Folder) auszuchecken und bei auszuprobieren

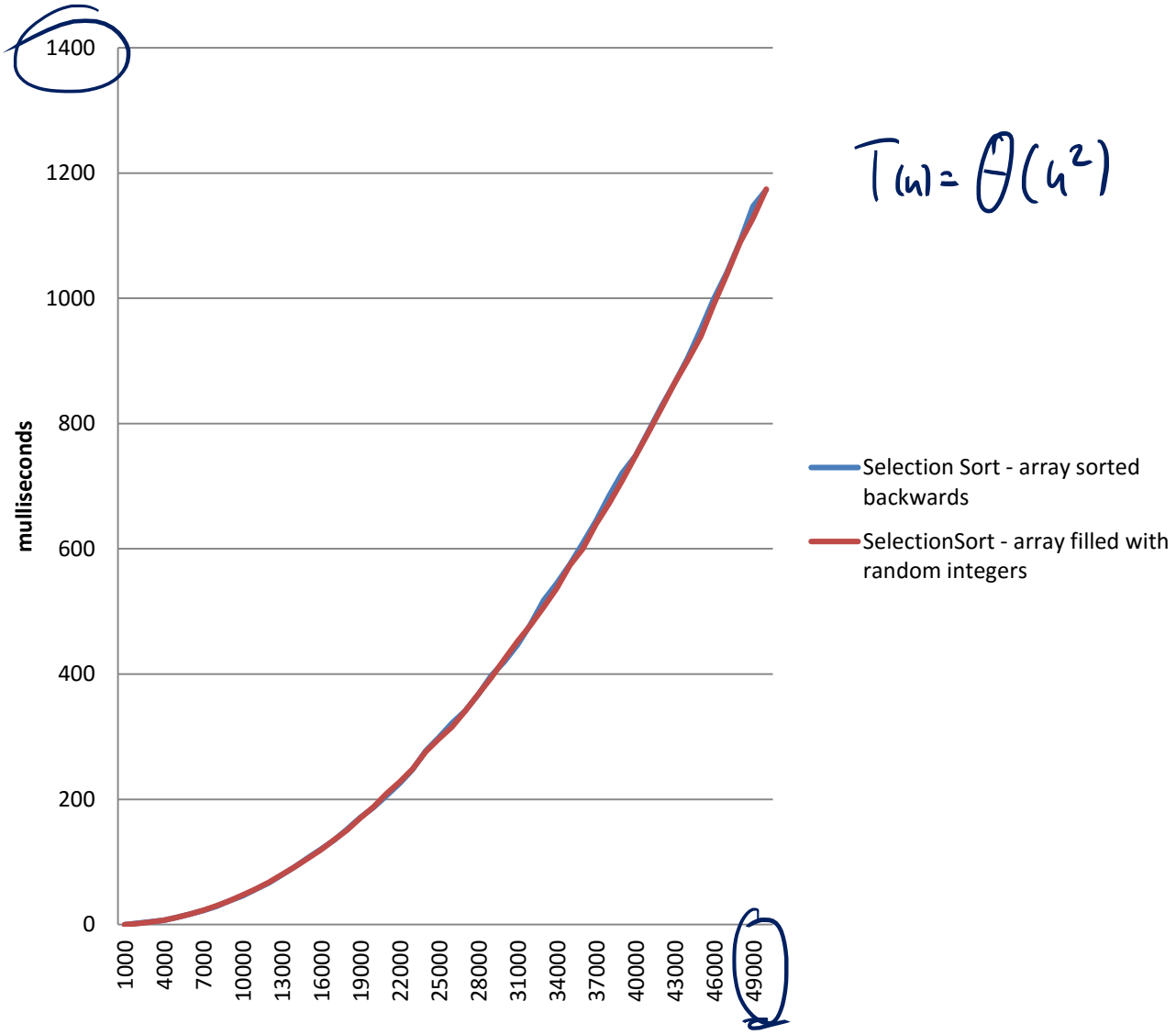
Nächste Übungen

- Übung 2 (online) ist rein theoretisch...
- Übung 3 wird mind. zur Hälfte theoretisch sein...

Weitere Programmierübungen:

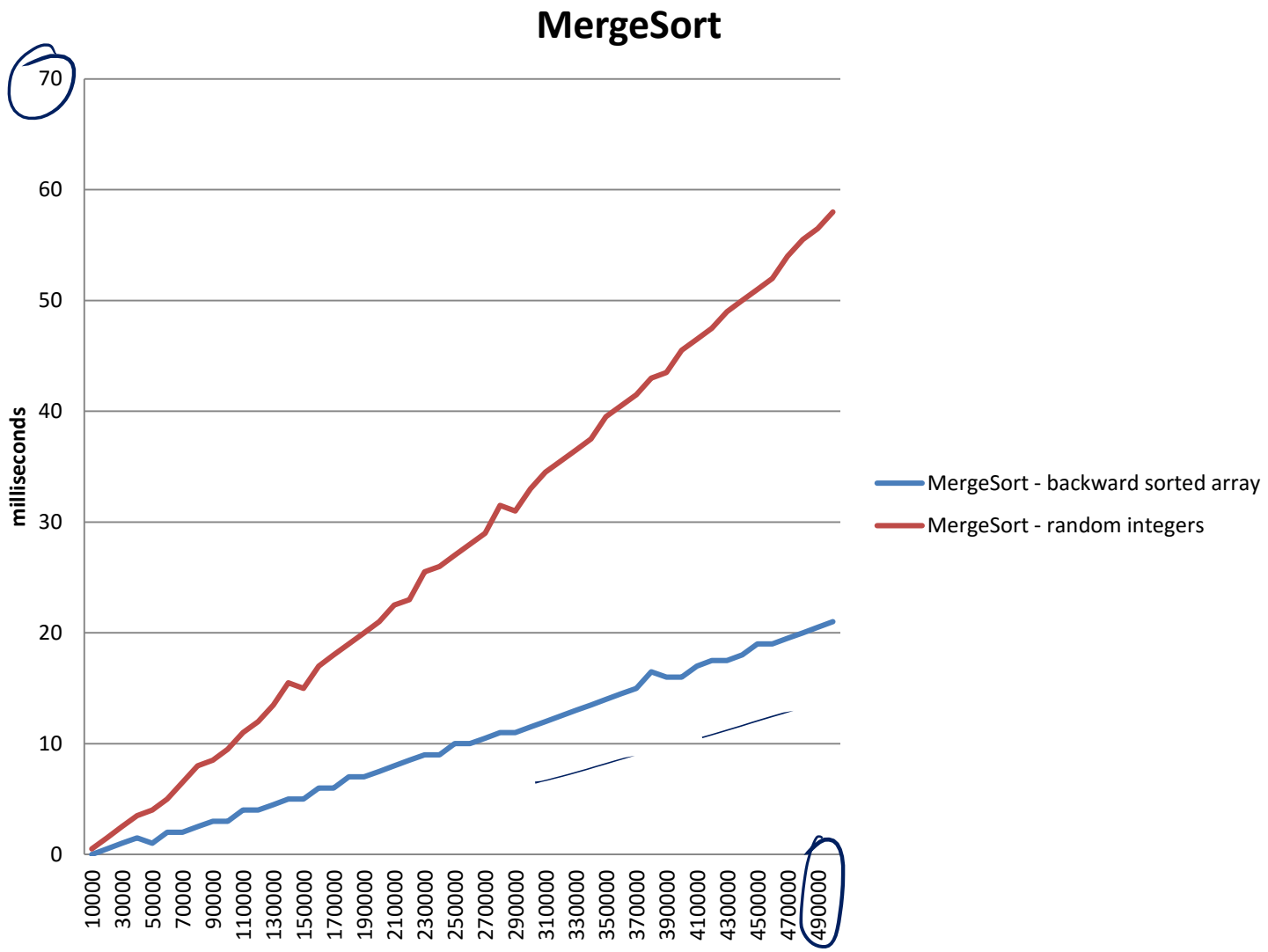
- Wir werden versuchen, jeweils Python-Grundgerüst zu geben, welches Sie dann entsprechend ergänzen können
- Es ist normal, dass die Dinge nicht immer auf Anhieb funktionieren...
  - Verschwenden Sie allerdings nicht zu viel Zeit, ohne Fortschritt zu machen, sondern wenden Sie sich **frühzeitig** mit Ihren Fragen ans Forum
  - Wenn möglich nicht erst am Donnerstag Abend / Nacht!

Messungen Selection Sort

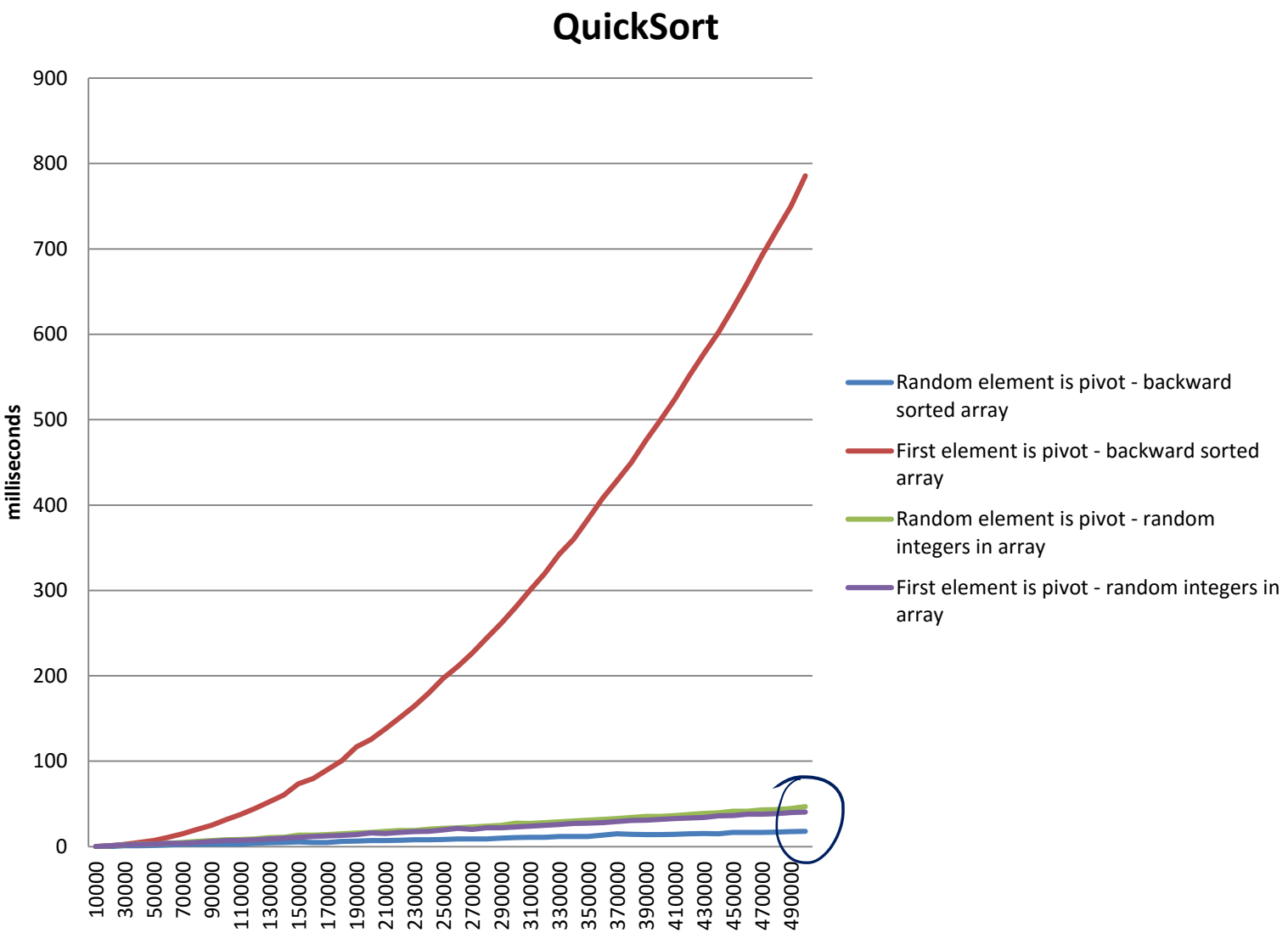


$T(n) = \Theta(n^2)$

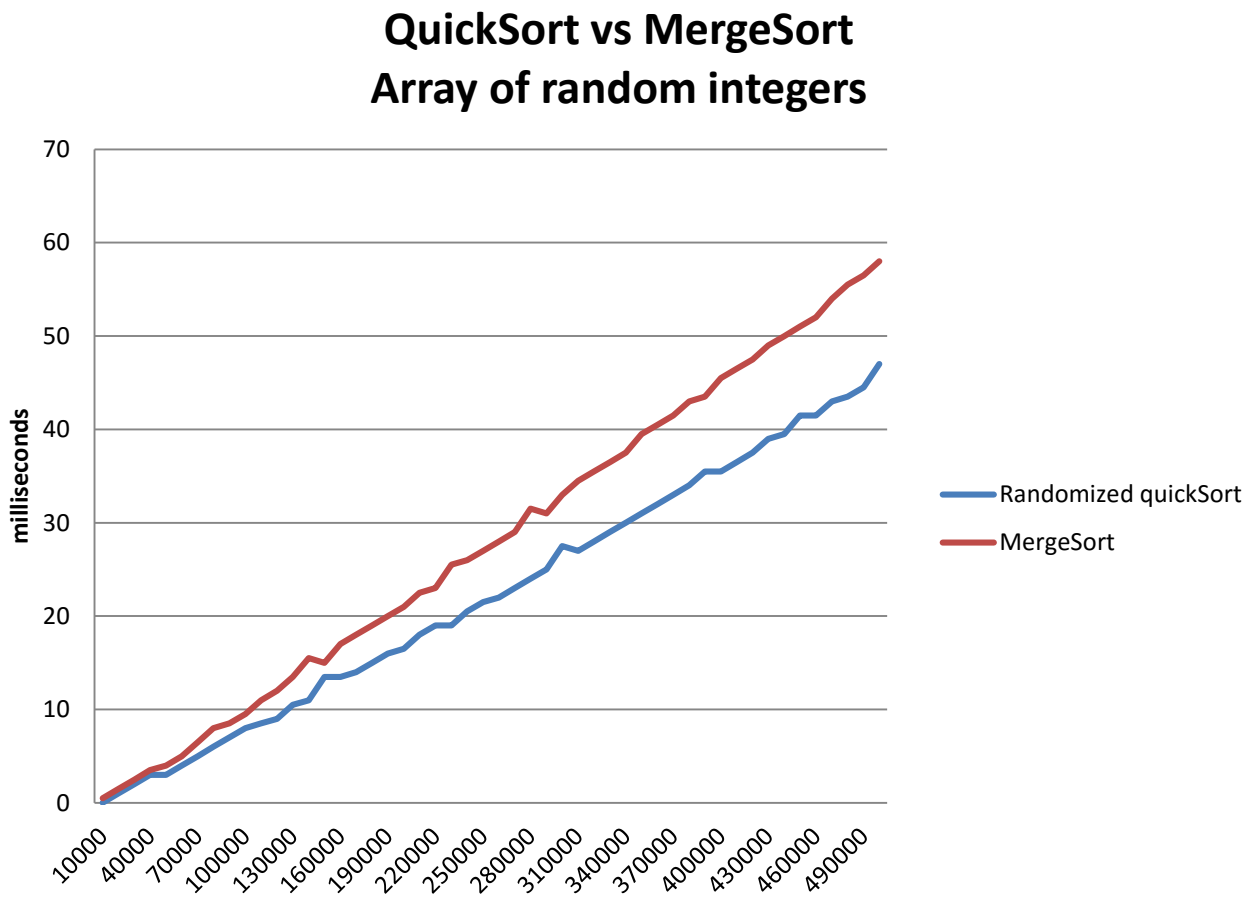
Messungen Merge Sort



Messungen Quick Sort



Messungen – Merge Sort vs. Quick Sort



Landau-Symbole : Intuitiv

$f(n) \in O(g(n))$: $f(n) \in O(g(n)) \iff g(n) \in \Omega(f(n))$

- $f(n)$ " $\leq$ " $g(n)$, asymptotisch gesehen…
- $f(n)$ wächst asymptotisch nicht schneller als $g(n)$

$f(n) \in \Omega(g(n))$:

- $f(n)$ " $\geq$ " $g(n)$, asymptotisch gesehen…
- $f(n)$ wächst asymptotisch mindestens so schnell, wie $g(n)$

$f(n) \in \Theta(g(n))$:

- $f(n)$ " $=$ " $g(n)$, asymptotisch gesehen…
- $f(n)$ wächst asymptotisch gleich schnell, wie $g(n)$

Landau-Symbole : Intuitiv

UNI FREIBURG

$f(n) \in o(g(n))$:

- $f(n)$ " $<$ " $g(n)$, asymptotisch gesehen…
- $f(n)$ wächst asymptotisch langsamer als $g(n)$

$f(n) \in \omega(g(n))$:

- $f(n)$ " $>$ " $g(n)$, asymptotisch gesehen…
- $f(n)$ wächst asymptotisch schneller als $g(n)$

Falls $f(n)$ und $g(n)$ monoton wachsen, gilt:

$$f(n) \in o(g(n)) \quad \Leftrightarrow \quad f(n) \notin \Omega(g(n))$$

$$f(n) \in \omega(g(n)) \quad \Leftrightarrow \quad f(n) \notin O(g(n))$$

Wie gut ist quadratische Laufzeit?

UNI FREIBURG

Quadratisch = 2x so grosse Eingabe → 4x so grosse Laufzeit

- das wächst für grosse n schon ziemlich schnell…

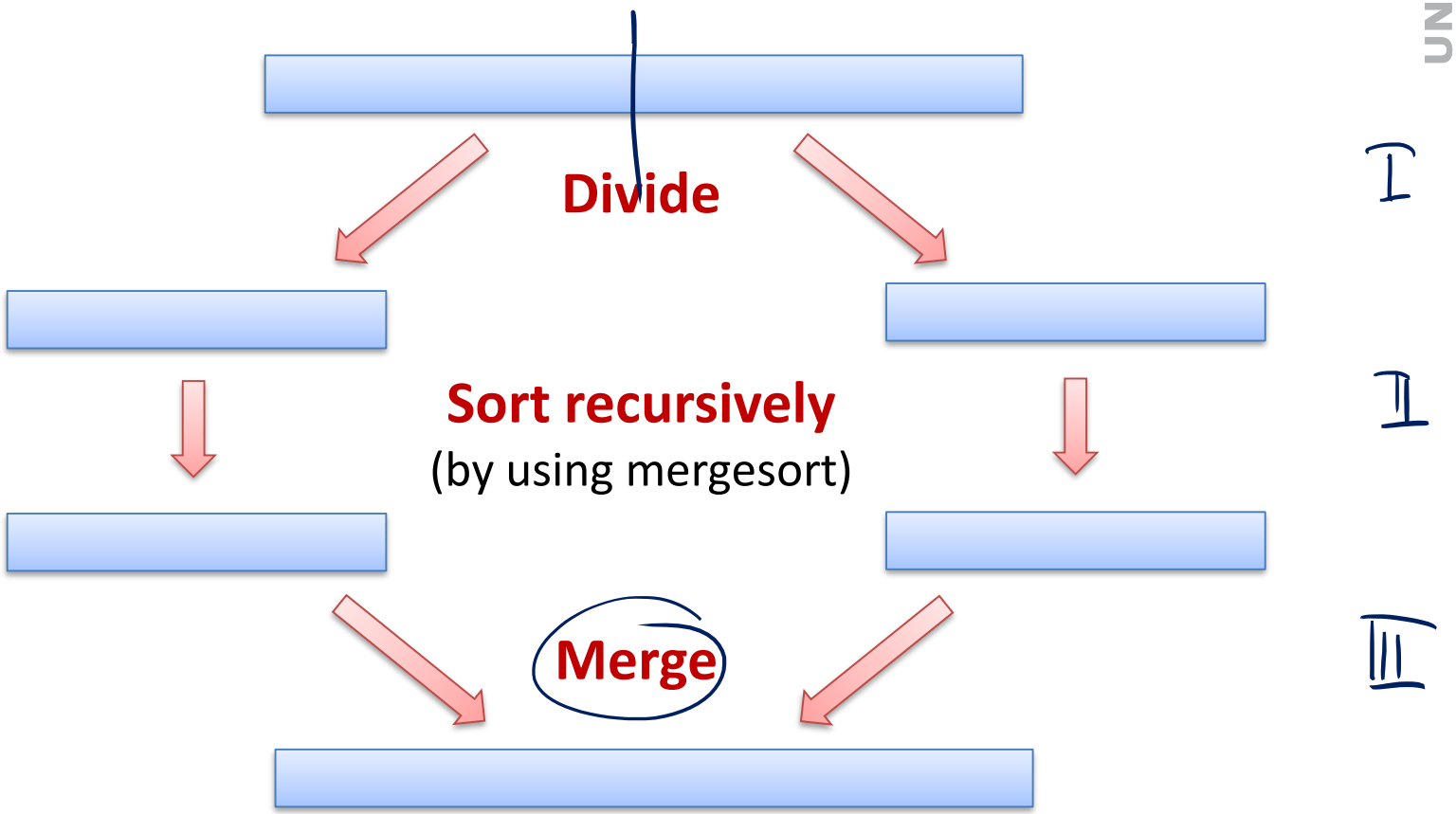
Beispielrechnung:

- Nehmen wir an, Anz. Grundop. $T(n) = n^2$
- Nehmen wir zudem an, 1 Grundop. pro Rechnerzyklus
- Bei einem 1Ghz-Rechner gibt das 1 ns pro Grundop.

Eingabegrösse n	4 Bytes pro Zahl	Laufzeit $T(n)$
10^3 Zahlen	$\approx 4\text{KB}$	$10^{3\cdot 2} \cdot 10^{-9}\ \text{s} = 1\ \text{ms}$
10^6 Zahlen	$\approx 4\text{MB}$	$10^{6\cdot 2} \cdot 10^{-9}\ \text{s} = 16.7\ \text{min}$
10^9 Zahlen	$\approx 4\text{GB}$	$10^{9\cdot 2} \cdot 10^{-9}\ \text{s} = 31.7\ \text{Jahre}$

für grosse Probleme zu langsam!

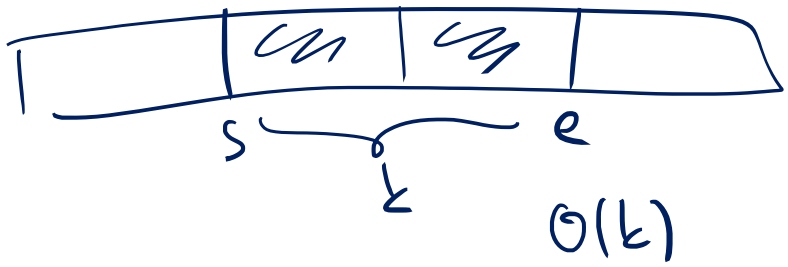
Analyse Merge Sort



- Divide ist trivial → Kosten: $O(1)$
- Rekursives Sortieren: Werden wir gleich noch anschauen...
- Merge: Das werden wir uns zuerst anschauen...

Analyse Merge-Schritt

```
MergeSortRecursive(A, start, end, tmp)          // sort A[start..end-1]
    ⋮
 5:   pos = start; i = start; j = middle
 6:   while (pos < end) do
 7:     if (i < middle) and (A[i] < A[j]) then
 8:       tmp[pos] = A[i]; pos++; i++
 9:     else
10:       tmp[pos] = A[j]; pos++; j++
11:   for i = start to end-1 do A[i] = tmp[i]
```



Analyse Merge Sort

bu

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

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

- 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

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

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

$T(n) \leq 2T(n/2) + b \cdot n$ $\quad (T(n/2) \leq 2 \cdot T(n/4) + b\frac{n}{2})$

$\leq 4T(n/4) + bn + bn$

$= 4T(n/4) + 2bn$

$\leq 4\left(2T(n/8) + b\frac{n}{4}\right) + 2bn$

$= 8T(n/8) + 3bn$

$\vdots \qquad \vdots$

$\leq 2^k T\left(n/2^k\right) + k \cdot bn \qquad (n/2^k \doteq 1)$

$= n \cdot T(1) + bn \cdot \log_2 n \leq b \cdot n(1 + \log_2^2 n)$

Vermutung:

$T(n) \leq bn(1 + \log_2 n)$

Analyse Merge Sort

asympt. $T(n) = O(n \cdot \log n)$

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: $n = 1$ $\quad T(1) \leq b \cdot 1 \cdot (1 + \log 1) = b$ ✓

Induktionsschritt:

Ind.-voraussetzung: Vermutung gilt für alle Werte $\leq n-1$

$$T(n) \leq 2T(n/2) + bn$$

$$\overset{IV.}{\leq} 2\left(b \cdot \frac{n}{2}\left(1 + \underbrace{\log_2 \frac{n}{2}}_{\log_2 n}\right)\right) + bn$$

$$\log_2 \frac{n}{2} = \log_2 n - \underbrace{\log_2 2}_{=1}$$

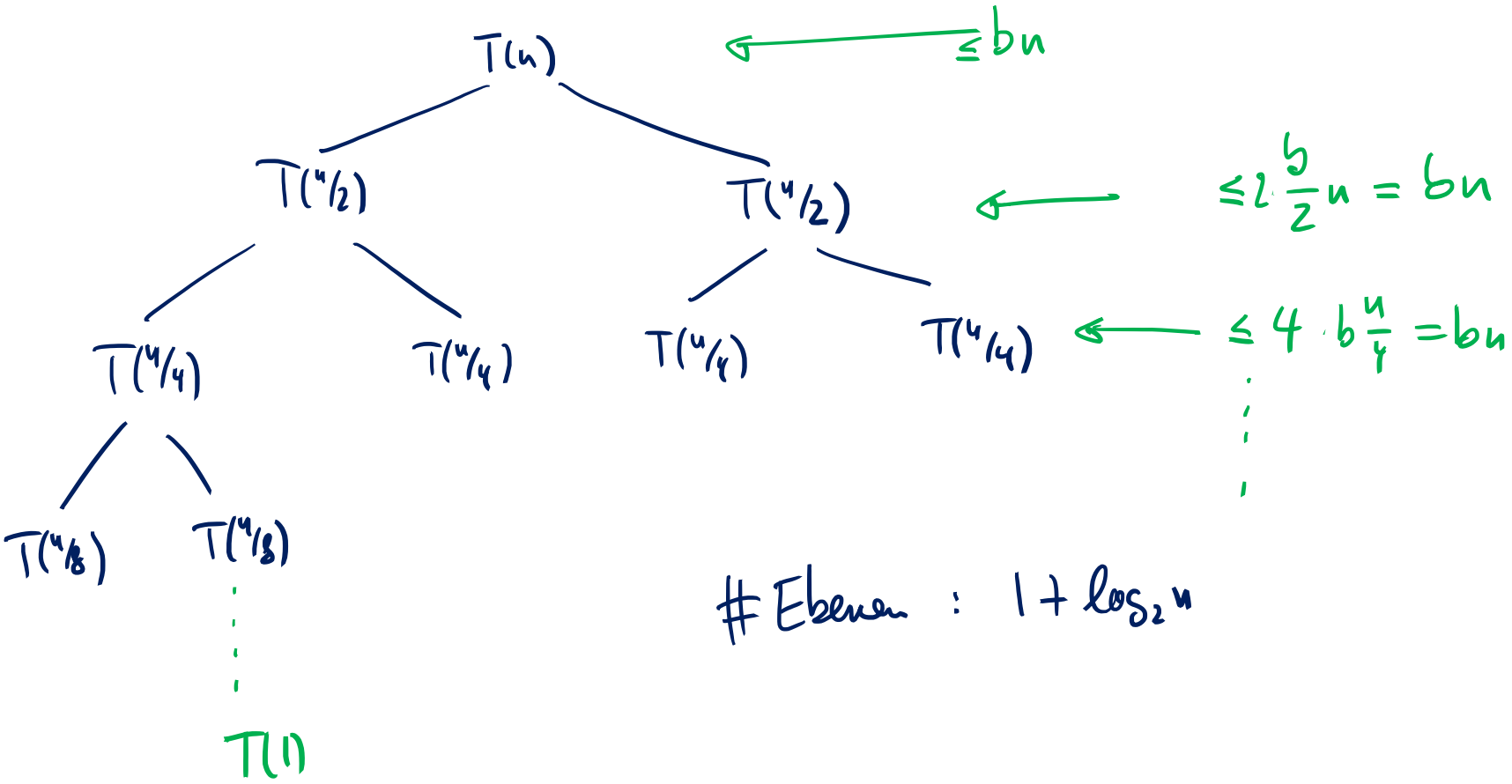
$$= b \cdot n \log_2 n + bn = bn(1 + \log_2 n)$$ ✓

□

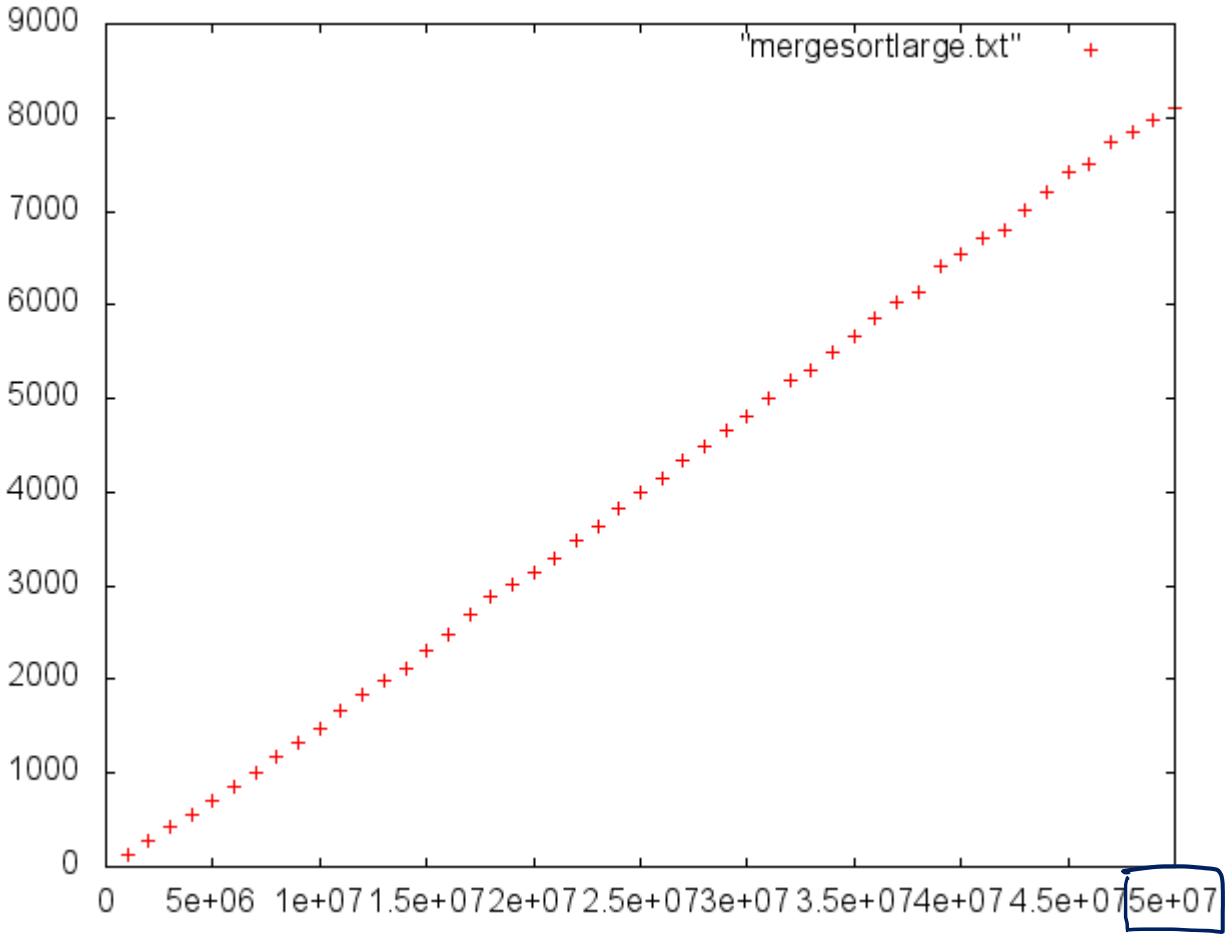
Alternative Analyse Merge Sort

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 für grössere n

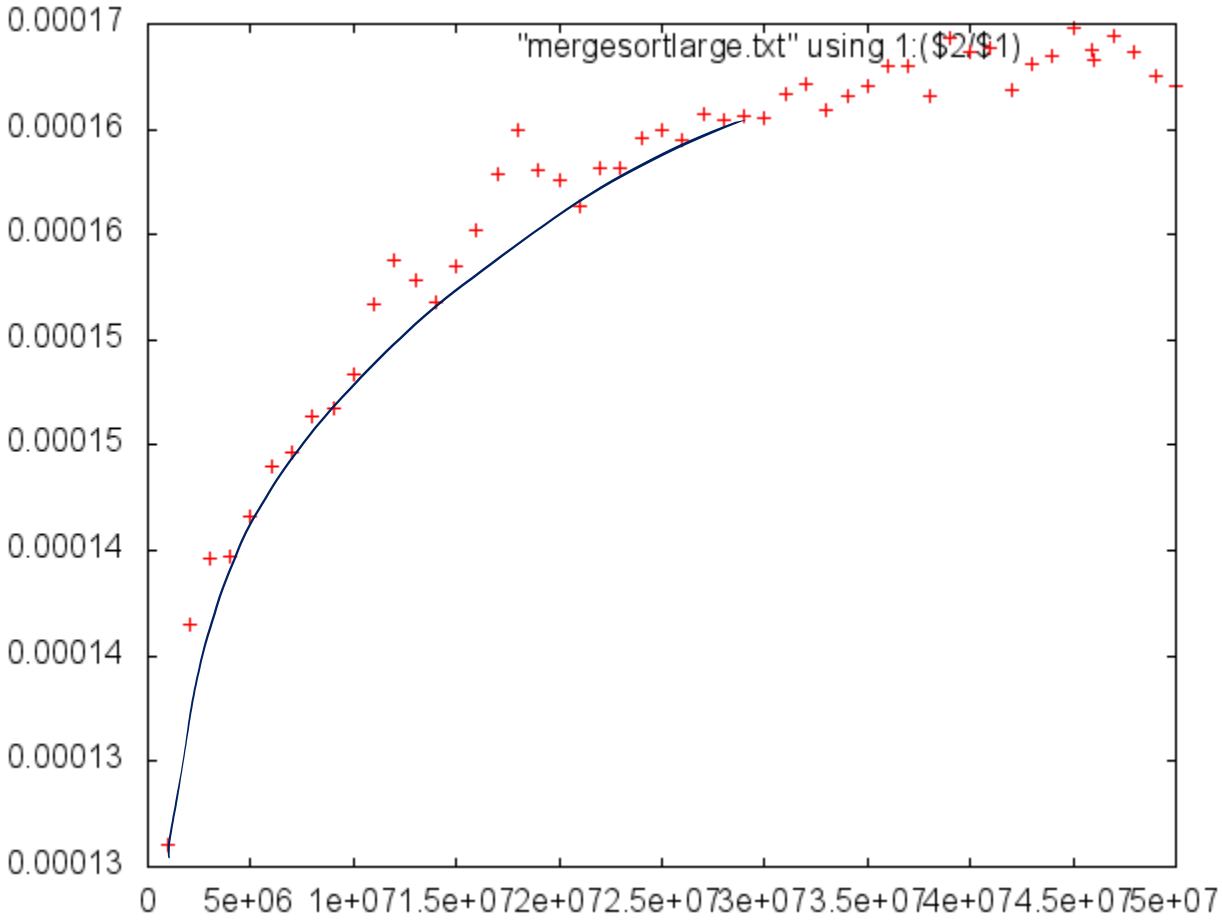


T(n)

Gnuplot:

```
set term png
set output "mergesort_1.png"
plot "mergesortlarge.txt"
```

Merge Sort Messungen für grössere n



$T(n)/n$

Gnuplot:

```
set term png
set output "mergesort_2.png"
plot "mergesortlarge.txt" using 1:($2:$1)
```

Zusammenfassung Analyse Merge Sort

UNI FREIBURG

Die Laufzeit von Merge Sort ist $T(n) \in O(n \cdot \log n)$.

- wächst fast linear mit der Grösse der Eingabe…

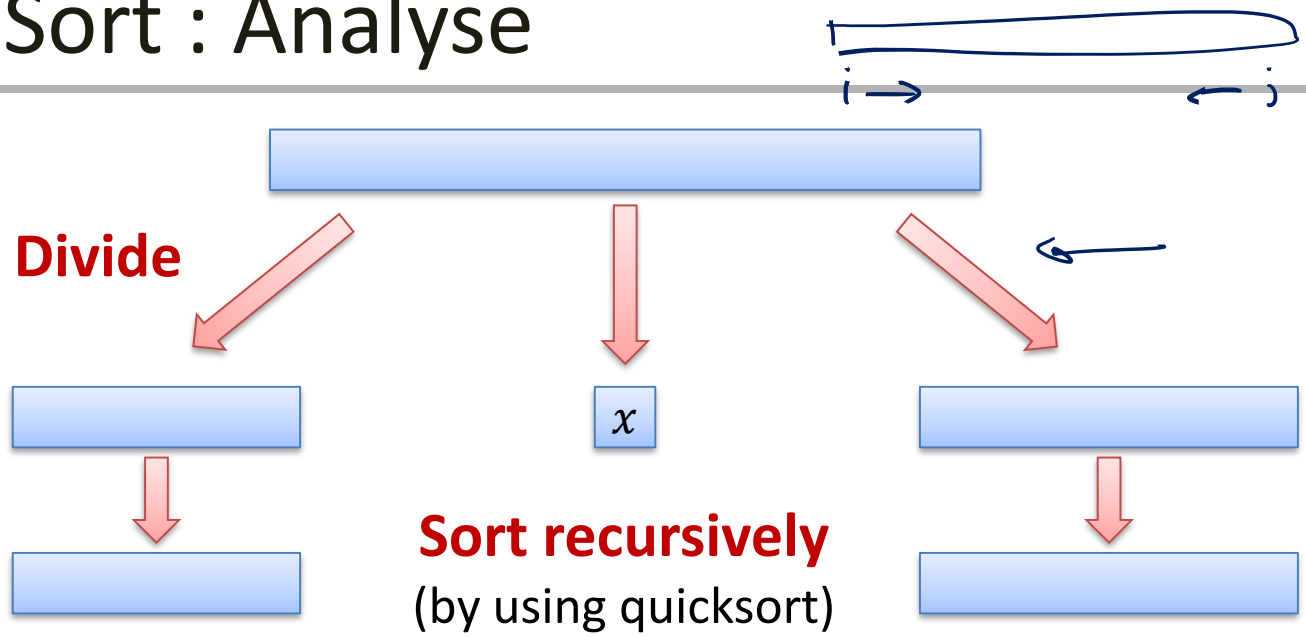
Wie gut ist das?

- Beispielrechnung:
  - Nehmen wir wieder an, 1 Grundop. = 1 ns
  - Wir sind aber ein bisschen konservativer als vorher und nehmen

$$T(n) = 10 \cdot n \log n$$

Eingabegrösse n	4 Bytes p. Zahl	Laufzeit $T(n) = 10 \cdot n \log n$	n^2
$2^{10} \approx 10^3$ Zahlen	$\approx$ 4KB	$10 \cdot 10 \cdot 2^{10} \cdot 10^{-9}$ s $\approx$ 0.1 ms	1 ms
$2^{20} \approx 10^6$ Zahlen	$\approx$ 4MB	$10 \cdot 20 \cdot 2^{20} \cdot 10^{-9}$ s $\approx$ 0.2 s	16.7 min
$2^{30} \approx 10^9$ Zahlen	$\approx$ 4GB	$10 \cdot 30 \cdot 2^{30} \cdot 10^{-9}$ s $\approx$ 5.4 min	31.7 Jahre
$2^{40} \approx 10^{12}$ Zahlen	$\approx$ 4TB	$10 \cdot 40 \cdot 2^{40} \cdot 10^{-9}$ s $\approx$ 122 h	$> 10^7$ Jahre

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:

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

$\mathcal{O}(n)$

- **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 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):

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

$T(n) \in O(n \log n)$

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

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

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

$T(n) \leq T(n-1) + T(1) + bn$

Quick Sort : Worst Case Analyse

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

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

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

$$T(n) \leq T(n-1) + bn$$
$$\leq T(n-2) + bn + b(n-1)$$
$$\leq T(n-3) + bn + b(n-1) + b(n-2)$$
$$\vdots$$
$$\leq T(n-k) + \dots$$
$$\vdots$$
$$\leq T(1) + b\underbrace{(n + n-1 + n-2 + \dots + 2)}_{\frac{n(n+1)}{2} - 1}$$
$$\leq b \cdot \frac{n(n+1)}{2}$$

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

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

Schritt:

$$T(n) \leq T(n-1) + bn$$
$$\overset{\text{I.V.}}{\leq} b \cdot \frac{(n-1)n}{2} + bn$$
$$= b\frac{n(n+1)}{2}$$ ✓

□

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:

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

In der Übung...

- ... müssen Sie zeigen, dass man $O(n \log n)$ bekommt, falls die Pivotwahl immer mindestens eine $(1/4, 3/4)$-Aufteilung ergibt.

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:

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

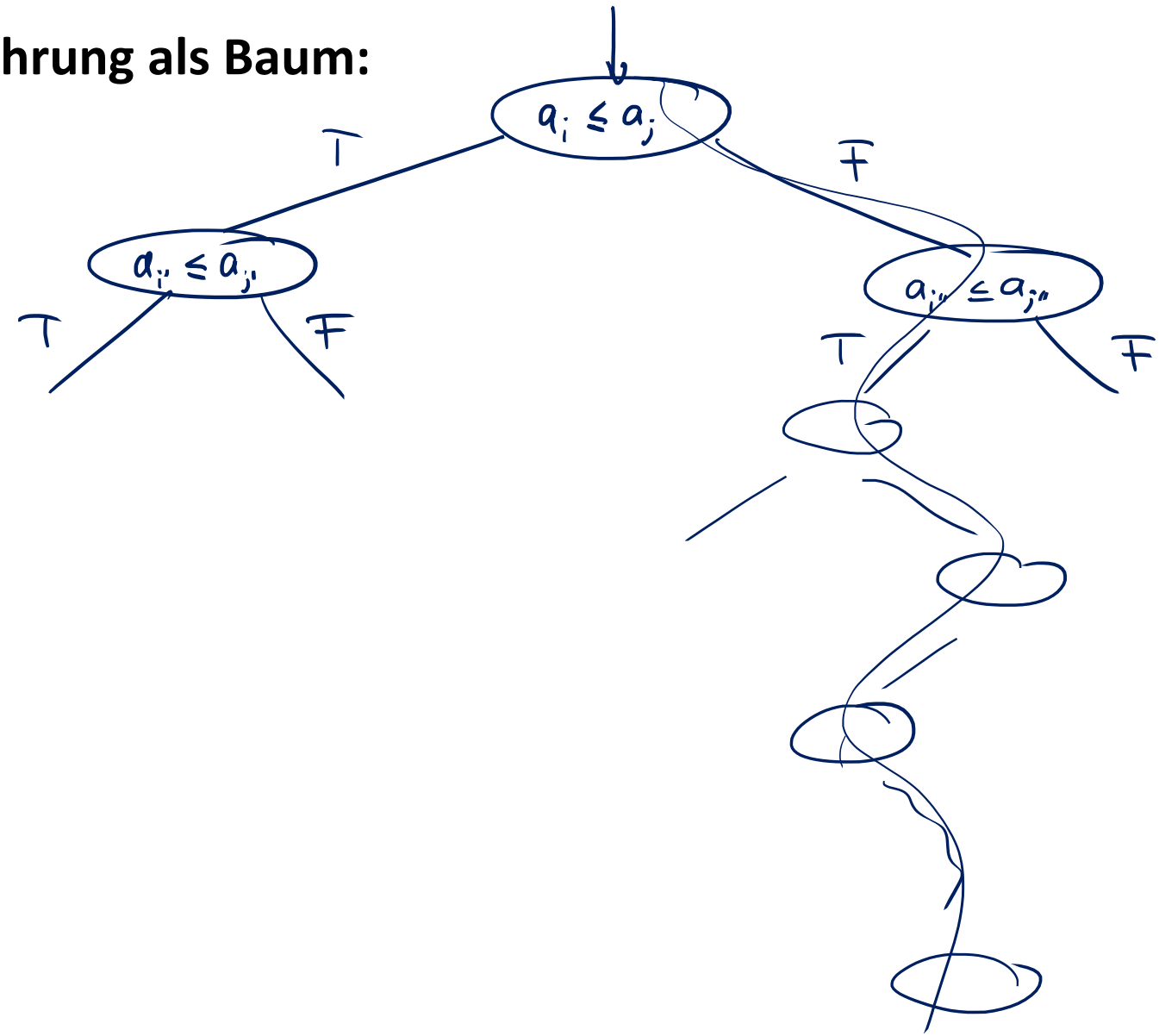
- In jeder Ausführung eines Algorithmus, induzieren die Resultate dieser Vergleiche eine Abfolge von T/F (true/false) Werten:

$$\mathbf{TFFTTTFTFFTTFFFFFTFTTT} \ldots$$

- 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

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):

$n! = 1 \cdot 2 \cdot \ldots \cdot n$

Anzahl T/F-Folgen der Länge $\leq k$:

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

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

$\log(\frac{n}{2}) = \log n - 1$

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

$n! \leq 2^{T+1}$ (Laufzeit $\leq T$)

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

$T+1 \in \Omega(n \log n)$

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

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

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

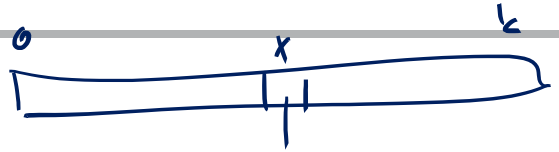
Sortieren in Linearzeit

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

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 = 0 to counts[j] do
7:       A[i] = j; i++
```

counts[A[i]] += 1 ∈ {0, .., k}