

Algorithmen und Datenstrukturen

Vorlesung 2

Laufzeitanalyse, Sortieren II



**UNI
FREIBURG**

Fabian Kuhn

Algorithmen und Komplexität

- Wie können wir die Laufzeit des Algorithmus analysieren?
 - Ist auf jedem Computer unterschiedlich...
 - Hängt vom Compiler, Programmiersprache, etc. ab
- Wir benötigen ein **abstraktes Mass**, um die Laufzeit zu messen
- **Idee:** Zähle **Anzahl (Grund-)Operationen**
 - Anstatt direkt die Zeit zu messen
 - Ist unabhängig von Computer, Compiler
 - Ein gutes Mass für die Laufzeit, falls alle Grundoperationen etwa gleich lange brauchen:

Was ist eine Grundoperation?

- Einfache arithmetische Operationen
 - $+$, $-$, $*$, $/$, $\%$ (mod), ...
- Ein Speicherzugriff
 - Variable auslesen, Variablenzuweisung
 - Ist das wirklich eine Grundoperation?
- Ein Funktionsaufruf
 - Natürlich nur das Springen in die Funktion
- **Intuitiv:** eine Zeile Programmcode
- **Besser:** eine Zeile Maschinencode
- **Noch besser (?):** ein Prozessorzyklus

- **Wir werden sehen:** Es ist nur wichtig, dass die Anzahl Grundoperation ungefähr proportional zur Laufzeit ist.

RAM = Random Access Machine

- **Standardmodell**, um Algorithmen zu analysieren!
- **Grundoperationen** (wie “definiert”) benötigen alle **eine Zeiteinheit**
- Insbesondere sind alle Speicherzugriffe gleich teuer:

Jede Speicherzelle (1 Maschinenwort) kann in 1 Zeiteinheit gelesen, bzw. beschrieben werden

- ignoriert insbesondere Speicherhierarchien
- Ist aber in den meisten Fällen eine vernünftige Annahme
- Alternative abstrakte Modelle existieren:
 - um Speicherhierarchien explizit abzubilden
 - bei riesigen Datenmengen (vgl. «Buzzword» Big Data)
 - z.B.: Streaming-Modelle: Speicher muss sequentiell gelesen werden
 - für verteilte/parallele Architekturen
 - Speicherzugriff kann lokal oder über’s Netzwerk sein...

Bisher: Anzahl Grundoperationen ist proportional zur Laufzeit

- Das können wir auch erreichen, ohne die Anzahl Grundoperationen genau zu zählen!

Vereinfachung 1: Wir berechnen nur eine **obere Schranke** (bzw. eine untere Schranke) an die Anzahl Grundoperationen

– So, dass die obere/untere Schranke immer noch proportional ist...

- Anz. Grundop. kann von div. Eigenschaften der Eingabe abhängen
 - Länge der Eingabe, aber auch z.B. bei Sortieren: zufällig, vorsortiert, ...

Vereinfachung 2: Wichtigster Parameter ist Grösse der Eingabe n

Wir betrachten daher die **Laufzeit $T(n)$ als Funktion von n .**

– Und ignorieren weitere Eigenschaften der Eingabe

Selection Sort: Analyse

SelectionSort(A):

1: **for** $i=0$ **to** $n-2$ **do**

2: $\text{minIdx} = i$ $\longleftarrow \leq c_1$

3: **for** $j=i$ **to** $n-1$ **do**

4: **if** $A[j] < A[\text{minIdx}]$ **then** $\left. \vphantom{A[j] < A[\text{minIdx}]} \right\} \longleftarrow \leq c_2$

5: $\text{minIdx} = j$

6: $\text{swap}(A[i], A[\text{minIdx}])$ $\longleftarrow \leq c_3$

#Grundop. $\leq c \cdot$ #Schleifeniterationen der inneren for-Schleife

$x(n)$

$$x(n) = \sum_{i=0}^{n-2} (n-i) = \sum_{h=2}^n h \leq \sum_{h=1}^n h = \frac{n(n+1)}{2} \leq n^2$$

Selection Sort: Analyse

SelectionSort(A):

1: **for** $i=0$ **to** $n-2$ **do**

2: $\text{minIdx} = i$ $\longleftarrow \leq c_1$

3: **for** $j=i$ **to** $n-1$ **do**

4: **if** $A[j] < A[\text{minIdx}]$ **then** $\left. \begin{array}{l} \longleftarrow \leq c_2 \\ \nearrow \geq c'_2 \end{array} \right\}$

5: $\text{minIdx} = j$

6: $\text{swap}(A[i], A[\text{minIdx}])$ $\longleftarrow \leq c_3$

$\underbrace{\# \text{Grundop.}}_{T(n)} \leq c \cdot \underbrace{\# \text{Schleifeniterationen der inneren for-Schleife}}_{x(n) \leq n^2}$

$T(n)$

$x(n) \leq n^2$

Laufzeit $T(n) \leq c \cdot n^2$

Selection Sort: Obere Schranke

$T(n)$: Anzahl Grundop. von Selection Sort bei Arrays der Länge n

Lemma: *Es gibt eine **Konstante** $c_U > 0$, so dass $T(n) \leq c_U \cdot n^2$*

Lemma: *Es gibt eine **Konstante** $c_L > 0$, so dass $T(n) \geq c_L \cdot n^2$*

Zusammenfassung

- Wir können nur eine Grösse berechnen, welche proportional zur Laufzeit ist
- Wir wollen auch gar nichts anderes berechnen:
 - Analyse sollte unabhängig von Computer / Compiler / etc. sein
 - Wir wollen Aussagen, welche auch in 10/100/... Jahren noch Gültigkeit haben

- Wir werden immer Aussagen der folgenden Art haben:

Es gibt eine Konstante C , so dass

$$T(n) \leq C \cdot f(n) \quad \text{oder} \quad T(n) \geq C \cdot f(n)$$

- Um dies zu vereinfachen / verallgemeinern gibt's die O-Notation...

Landau-Symbole (“O-Notation”)

- Formalismus, um das asymptotische Wachstum von Funktionen zu beschreiben.
 - Formale Definitionen: siehe nächste Folie...

- Es gibt eine Konst. $C > 0$, so dass $T(n) \leq C \cdot f(n)$ wird zu:

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

- Es gibt eine Konst. $C > 0$, so dass $T(n) \geq C \cdot g(n)$ wird zu:

$$T(n) \in \Omega(g(n))$$

- Bei Selection Sort: $T(n) \in O(n^2)$
 $T(n) \in \Omega(n^2)$ } $T(n) \in \Theta(n^2)$

Landau-Symbole : Definitionen

$$O(g(n)) := \{f(n) \mid \exists c, n_0 > 0 \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$$

- Funktion $f(n) \in O(g(n))$, falls es Konstanten c und n_0 gibt, so dass $f(n) \leq c \cdot g(n)$ für alle $n \geq n_0$

$$f(n) = n^2 + 100n$$

$$\Omega(g(n)) := \{f(n) \mid \exists c, n_0 > 0 \forall n \geq n_0 : f(n) \geq c \cdot g(n)\}$$

- Funktion $f(n) \in \Omega(g(n))$, falls es Konstanten c und n_0 gibt, so dass $f(n) \geq c \cdot g(n)$ für alle $n \geq n_0$

$$\Theta(g(n)) := O(g(n)) \cap \Omega(g(n))$$

- Funktion $f(n) \in \Theta(g(n))$, falls es Konstanten c_1, c_2 und n_0 gibt, so dass $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ für alle $n \geq n_0$, resp. falls $f(n) \in O(n)$ und $f(n) \in \Omega(n)$

Landau-Symbole : Definitionen

$$o(g(n)) := \{f(n) \mid \forall c > 0 \exists n_0 > 0 \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$$

- Funktion $f(n) \in o(g(n))$, falls für alle Konstanten $c > 0$ gilt, dass $f(n) \leq c \cdot g(n)$ (für genug grosse n , abhängig von c)

$$\omega(g(n)) := \{f(n) \mid \forall c > 0 \exists n_0 > 0 \forall n \geq n_0 : f(n) \geq c \cdot g(n)\}$$

- Funktion $f(n) \in \omega(g(n))$, falls für alle Konstanten $c > 0$ gilt, dass $f(n) \geq c \cdot g(n)$ (für genug grosse n , abhängig von c)

Insbesondere gilt:

$$f(n) \in o(g(n)) \implies f(n) \in O(g(n))$$

$$f(n) \in \omega(g(n)) \implies f(n) \in \Omega(g(n))$$

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

- $f(n) \leq g(n)$, 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 $g(n)$

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

- $f(n) \ll g(n)$, asymptotisch gesehen...
- $f(n)$ wächst asymptotisch langsamer als $g(n)$

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

- $f(n) \gg g(n)$, asymptotisch gesehen...
- $f(n)$ wächst asymptotisch schneller als $g(n)$

monoton wachsen und sich "vernünftig" verhalten

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

$$f(n) \in o(g(n)) \iff f(n) \notin \Omega(g(n))$$

$$f(n) \in \omega(g(n)) \iff f(n) \notin O(g(n))$$

$$\underline{f(n) \ll g(n)} \iff \underline{f(n) \gg g(n)}$$

Definition über Grenzwerte (vereinfacht)

Folgende Definitionen gelten für ~~monoton wachsende Funktionen~~

falls der Grenzwert $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$ definiert ist

$$f(n) \in O(g(n)), \quad \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$$

$$f(n) \in \Omega(g(n)), \quad \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0$$

$$f(n) \in \underline{\Theta}(g(n)), \quad 0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$$

$$f(n) \in o(g(n)), \quad \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) \in \omega(g(n)), \quad \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

Landau-Notation : Bemerkungen

Schreibweise:

- $O(g(n)), \Omega(g(n)), \dots$ sind Mengen (von Funktionen)
- Korrekte Schreibweise ist deshalb eigentlich: $f(n) \in O(g(n))$
- Sehr verbreitete Schreibweise: $f(n) = O(g(n))$

Beispiele:

- $T(n) = O(n^2)$ statt $T(n) \in O(n^2)$
- $T(n) = \Omega(n^2)$ statt $T(n) \in \Omega(n^2)$

- $f(n) = n^2 + O(n)$:

$$f(n) \leq n^2 + c \cdot n$$

$$f(n) \in \{g(n) : \exists h(n) \in O(n) \text{ s.t. } g(n) = n^2 + h(n)\}$$

- $a(n) = (1 + o(1)) \cdot b(n)$

$$\forall \varepsilon > 0 : \exists n_0, \forall n > n_0 : a(n) \leq (1 + \varepsilon)b(n)$$

Schreibweise:

- $O(g(n)), \Omega(g(n)), \dots$ sind Mengen (von Funktionen)
- Korrekte Schreibweise ist deshalb eigentlich: $f(n) \in O(g(n))$
- Sehr verbreitete Schreibweise: $f(n) = O(g(n))$

Asymptotisches Verhalten für allgemeine Grenzwerte:

- gleiche Schreibweise auch für Verhalten von z.B. $f(x)$ für $x \rightarrow 0$
- z.B. Taylor-Reihen: $e^x = 1 + x + O(x^2)$, bzw. $e^x = 1 + x + o(x)$

Alternative Definition für $\Omega(g(n))$:

$$g(n) = n^2, f(n) = \begin{cases} n^2, & n \text{ gerade} \\ 1, & n \text{ ungerade} \end{cases}$$

$$\Omega(g(n)) := \{f(n) \mid \exists c, n_0 > 0 \forall n \geq n_0 : f(n) \geq c \cdot g(n)\} \leftarrow$$

$$\Omega(g(n)) := \{f(n) \mid \exists c > 0 \forall n_0 > 0 \exists n \geq n_0 : f(n) \geq c \cdot g(n)\}$$

– Wir verwenden die 1. Definition

– *Macht keinen Unterschied falls der Grenzwert $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$ definiert ist*

Landau-Notation : Beispiele

Selection Sort:

- Laufzeit $T(n)$, es gibt Konstanten $c_1, c_2 : c_1 n^2 \leq T(n) \leq c_2 n^2$
 $T(n) \in O(n^2), \quad T(n) \in \Omega(n^2), \quad T(n) \in \Theta(n^2)$
- $T(n)$ wächst schneller als linear: $T(n) \in \omega(n)$

Weitere Beispiele:

- $f(n) = 10n^3, g(n) = n^3/1000 : f(n) \in \Theta(g(n))$
- $f(n) = e^n, g(n) = n^{100} : f(n) \in \omega(g(n))$
- $f(n) = n/\log_2 n, g(n) = \sqrt{n} : f(n) \in \omega(g(n))$
- $f(n) = n^{1/256}, g(n) = 10 \ln n : f(n) \in \omega(g(n))$
- $f(n) = \log_{10} n, g(n) = \log_2 n : f(n) \in \Theta(g(n))$
- $f(n) = n^{\sqrt{n}}, g(n) = 2^n : f(n) \in o(g(n))$

$$\lim_{n \rightarrow \infty} \frac{e^n}{n^{100}} \rightarrow \infty$$

$$\frac{f(n)}{g(n)} = \frac{\sqrt{n}}{\log_2 n} = \frac{2^{t/2}}{t}$$

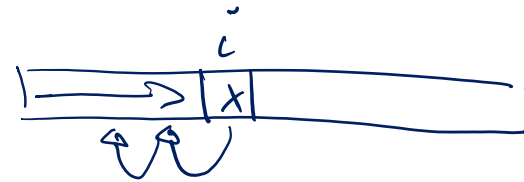
$$\log_{10} n = \frac{\log_2 n}{\log_2 10}$$

$$\log(n^{\sqrt{n}}) = \sqrt{n} \cdot \log n, \log(2^n) = n$$

Analyse Insertion Sort

InsertionSort(A):

- 1: **for** $i = 1$ **to** $n-1$ **do**
- 2: *// prefix $A[1..i]$ is already sorted*
- 3: $pos = i$
- 4: **while** $(pos > 0)$ **and** $(A[pos] < A[pos-1])$ **do**
- 5: $swap(A[pos], A[pos-1])$
- 6: $pos = pos - 1$



Laufzeit = $O(\underbrace{\# \text{ Iterationen der while-Schleife}}_{X(n)})$

$$X(n) \leq \sum_{i=1}^{n-1} i = O(n^2)$$

$$X(n) \geq \sum_{i=1}^{n-1} 1 = \Omega(n)$$

Worst Case Analyse

- Analysiere Laufzeit $T(n)$ für eine schlechtestmögliche Eingabe der Grösse n
- Wichtigste / Standard- Art der Algorithmenanalyse

Best Case Analyse

- Analysiere Laufzeit $T(n)$ für eine bestmögl. Eingabe der Grösse n
- Meistens uninteressant...

Average Case Analyse

- Analysiere Laufzeit $T(n)$ für eine typische Eingabe der Grösse n
- Problem: was ist eine typische Eingabe?
 - Standardansatz: zufällige Eingabe
 - nicht klar, wie nahe tatsächliche Instanzen bei uniform zufälligen sind...
 - eine mögl. Alternative: smoothed analysis (werden wir nicht anschauen)

Wie gut ist quadratische Laufzeit?

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

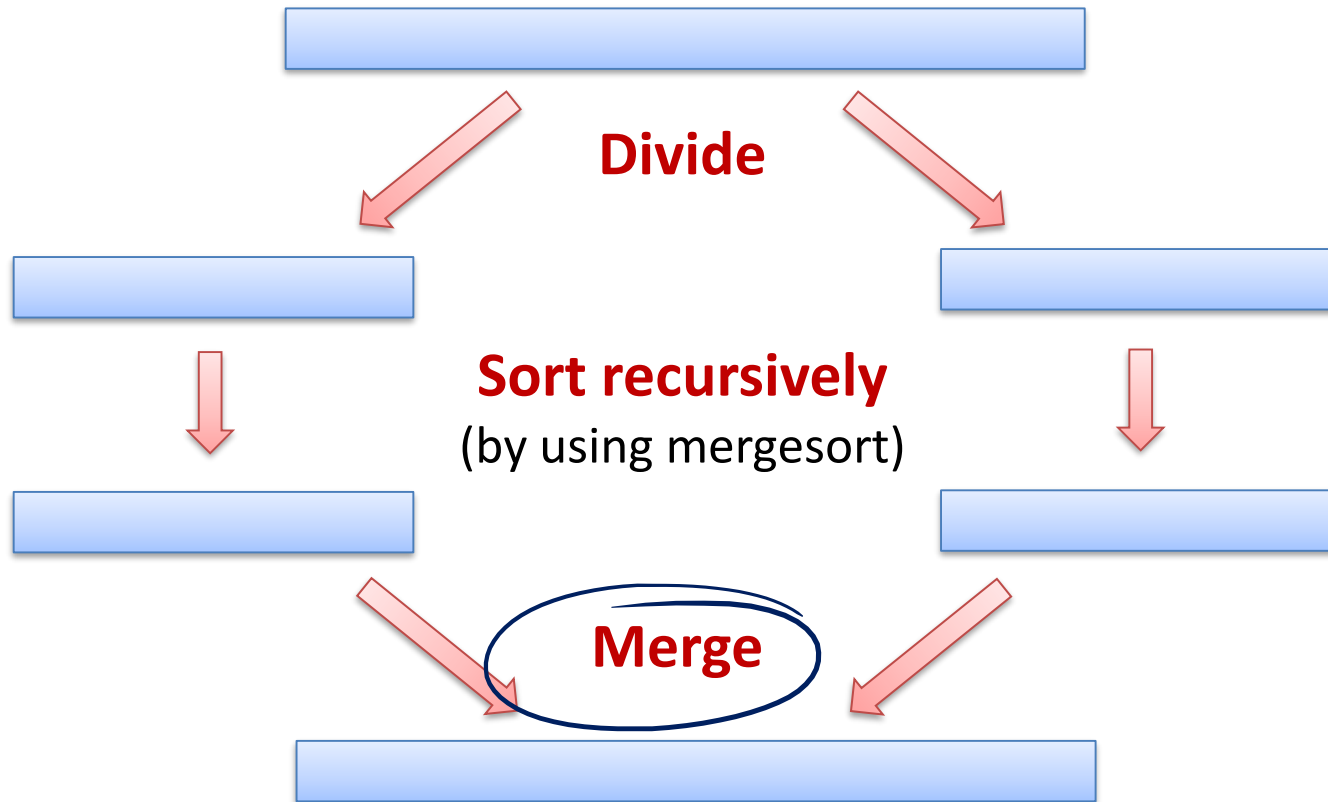
– das wächst für grosse n schon ziemlich schnell...

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!

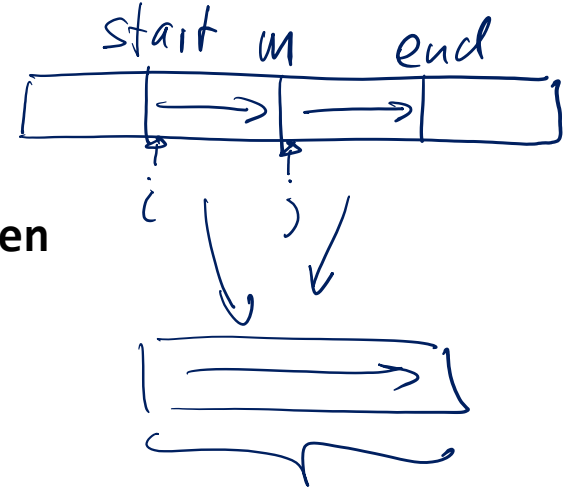


- Divide ist trivial \rightarrow Kosten: $O(1)$
- Rekursives Sortieren: Werden wir gleich noch anschauen...
- Merge: Das werden wir uns zuerst anschauen...

Analyse Merge-Schritt

```
MergeSortRecursive(A, start, end, tmp)
:
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]
```

// sort A[start..end-1]



$k := \text{end} - \text{start}$

Schleifeniterationen:
while: k for: k

Laufzeit: $O(k)$

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

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

- Wir machen uns das Leben ein bisschen einfacher und ignorieren das Auf- und Abrunden: *Annahme: n Zweierpotenz*

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

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...

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

Vermutung
↓

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: $n=1$ $T(1) \leq b \cdot 1 \cdot (1 + \log_2 1) = b \checkmark$

Induktionsschritt:

Ind.-voraussetzung: Vermutung gilt für Werte $< n$

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

$$\stackrel{\text{(i.v.)}}{\leq} 2 \cdot b \cdot \frac{n}{2} \cdot \underbrace{\left(1 + \log_2 \frac{n}{2}\right)}_{\log_2 n} + b \cdot n$$

$$= b n (\log_2 n + 1) \checkmark$$

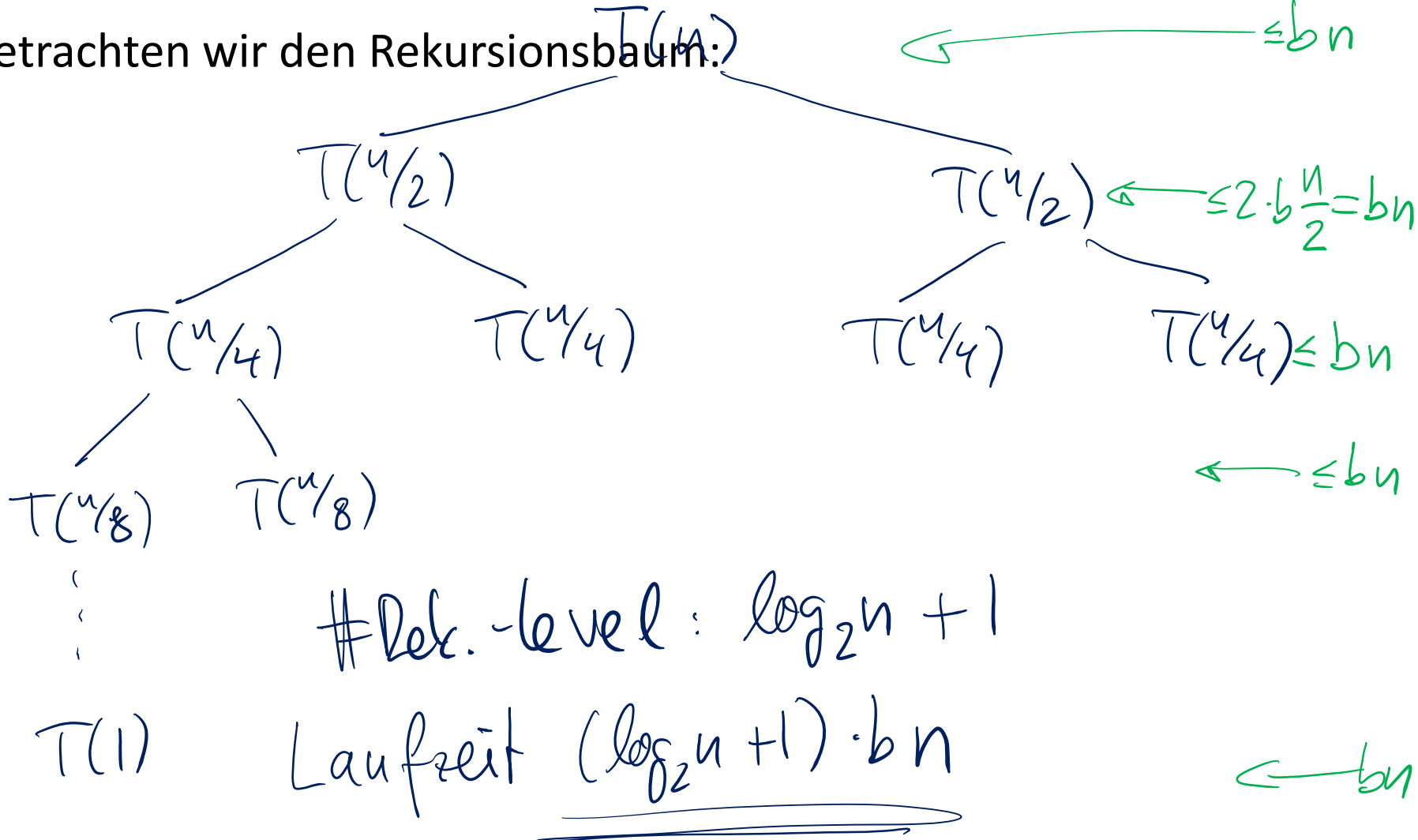
$$T(n) = O(n \cdot \log n)$$

□

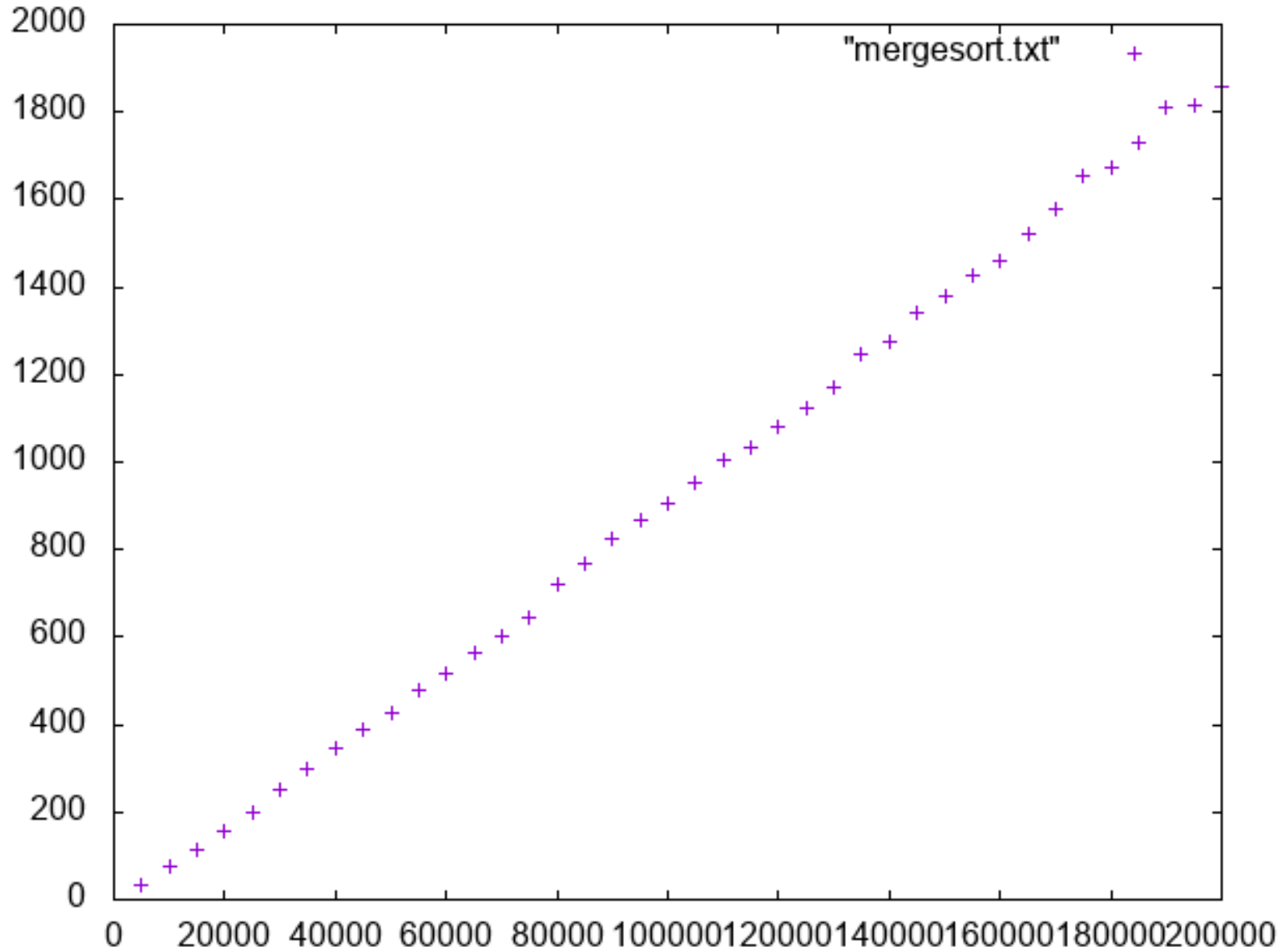
Alternative Analyse Merge Sort

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

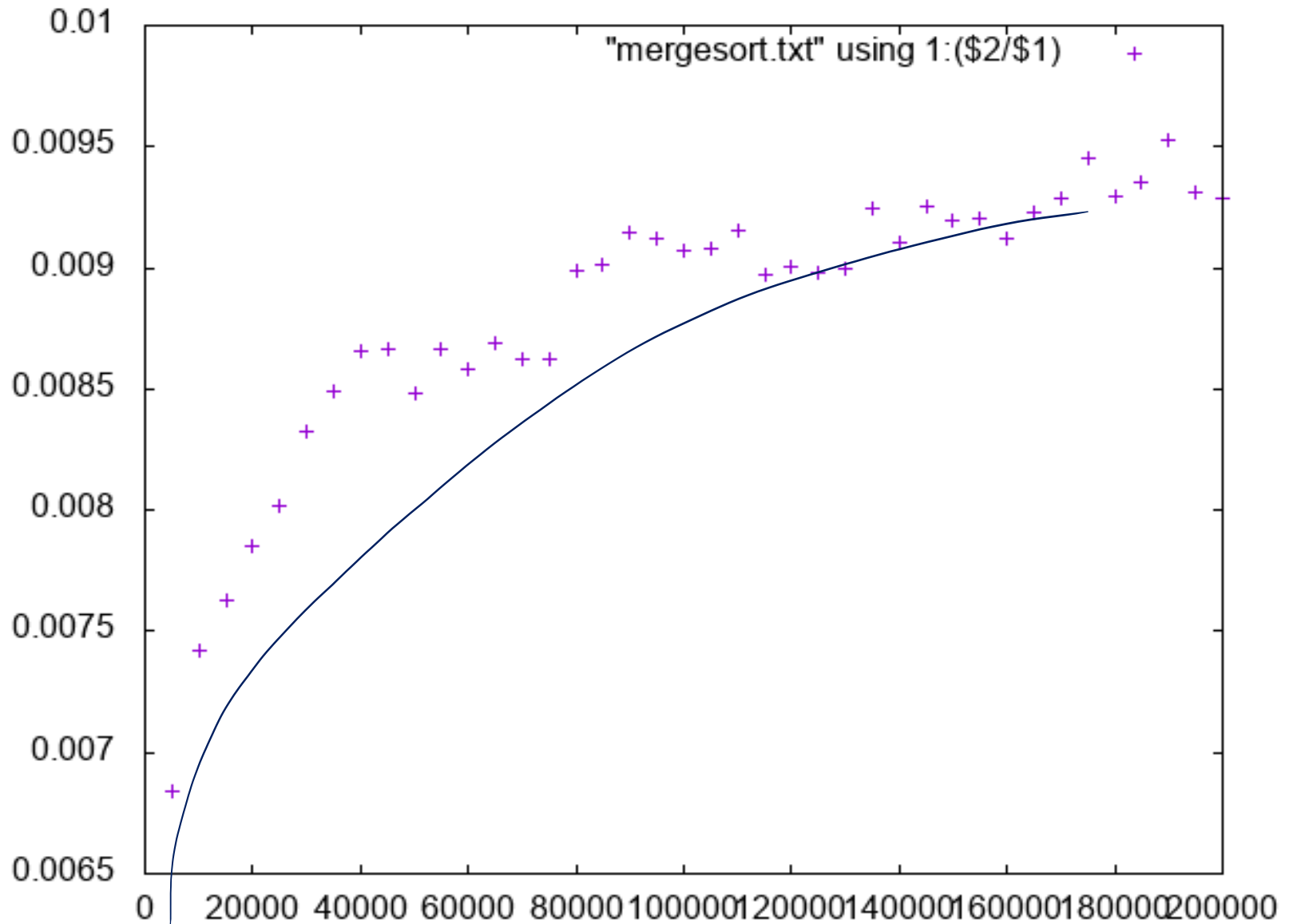
Betrachten wir den Rekursionsbaum:



Merge Sort Messungen



Merge Sort Messungen



Zusammenfassung Analyse Merge Sort

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

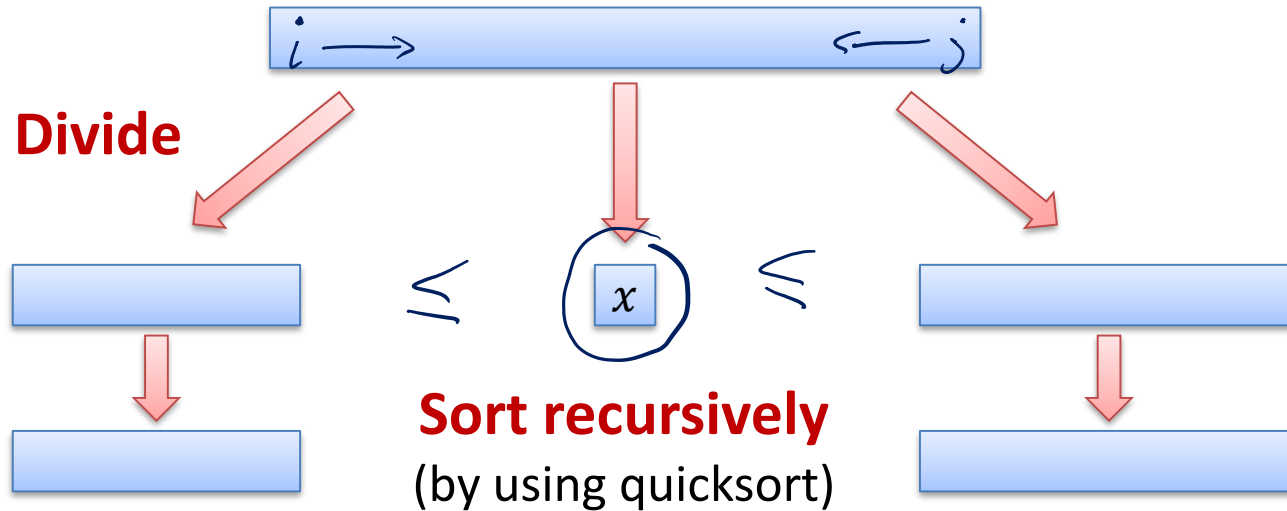
- wächst fast linear mit der Grösse der Eingabe...

Wie gut ist das?

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

$$T(n) = \underline{10} \cdot n \log n$$

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



- 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:

$$\underline{T(n)} = \underline{T(\lambda n)} + \underline{T((1 - \lambda)n)} + \underbrace{\text{"Pivotsuche + Divide"}}_{O(n)}$$

- **Divide:**
 - Wir gehen einmal von beiden Seiten über's Array mit konstanten Kosten pro Schritt \rightarrow 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

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

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

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

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



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

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

Quick Sort : Worst Case Analyse

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

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

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

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

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

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

Schritt:

$$\begin{aligned} T(n) &\leq T(n-1) + b \cdot n \\ &\stackrel{(IV)}{\leq} b \frac{(n-1)n}{2} + bn \\ &= b \frac{n(n+1)}{2} \quad \checkmark \end{aligned}$$

Quick Sort mit zufälligem Pivot

Aufteilung bei zufälligem Pivot:

- Laufzeit $T(n) = O(n \log n)$ 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

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

The diagram shows a horizontal bar representing an array of size n . A pivot is chosen, and the array is partitioned into three parts: a left part of size $n/4$, a middle part (the pivot and its duplicates) of size $n/2$, and a right part of size $n/4$. The total size n is indicated by a bracket below the bar.

Aufteilung bei zufälligem Pivot:

- Laufzeit $T(n) = O(n \log n)$ 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:

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

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

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:

if ($a_i \leq a_j$) **then ...**

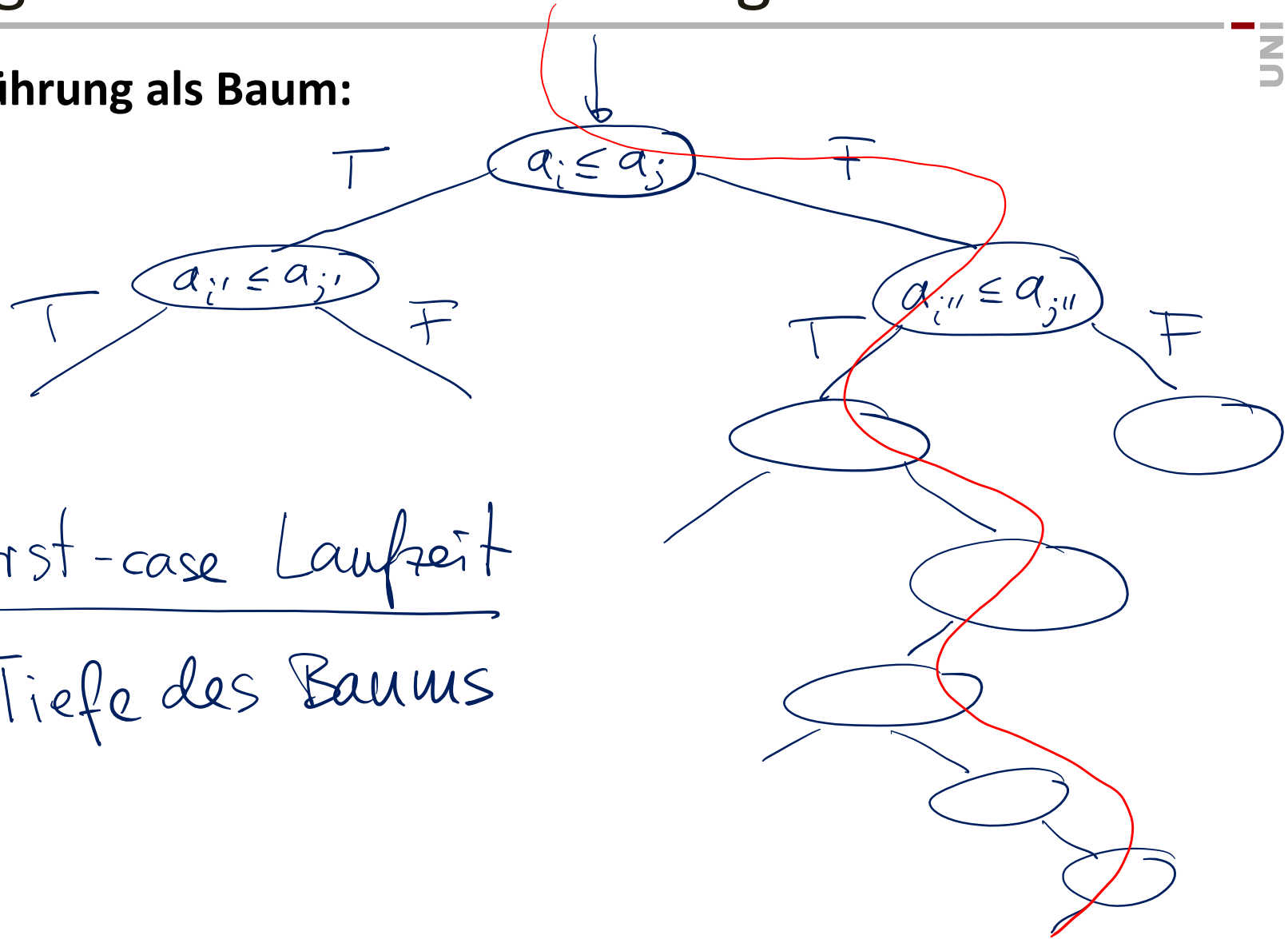
- In jeder Ausführung eines Algorithmus, induzieren die Resultate dieser Vergleiche eine Abfolge von T/F (true/false) Werten:

TFFTTTFFTFFFTTFFFFFTFTTT ...

- 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:



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, ..., 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 \rightarrow wir benötigen so viele T/F-Folgen...

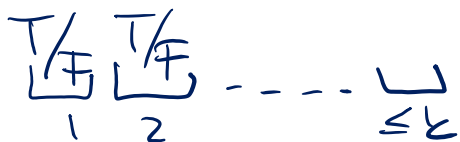
Vgl.-Basiertes Sortieren : Untere Schranke I

Anzahl Mögliche Eingaben (Anfangsreihenfolgen):

$$\underline{n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1}$$

$$\left[\log(a^b) = b \cdot \log a \right]$$

Anzahl T/F-Folgen der Länge $\leq k$: Länge = k : 2^k



$$\underline{2^k + 2^{k-1} + \dots + 1 \leq 2^{k+1}}$$

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

Laufzeit $\leq T$

$$\underline{\frac{2^{T+1}}{T+1} \geq \frac{n!}{\log_2(n!)}}$$
$$\underline{T \in \Omega(n \log n)}$$

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

$$\frac{n}{2} \cdot \log_2 \left(\frac{n}{2} \right) \leq \log_2(n!) \leq n \cdot \log_2(n)$$

$$\underline{\log_2(n!) = \Theta(n \cdot \log n)}$$

- 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

Aufgabe:

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

Algorithmus:

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

} O(k + n)