Algorithmen und Datenstrukturen

Vorlesung 2

Laufzeitanalyse, Sortieren II

UNI FREIBURG

Fabian Kuhn

Algorithmen und Komplexität

Laufzeitanalyse I

UNI FREIBURG

- 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:

Grundoperation

UNI FREIBURG

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 Modell

UNI FREIBURG

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…

Laufzeitanalyse II

UNI FREIBURG

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:    minIdx = i                      ← ≤ c1
 3:    for j=i to n-1 do
 4:       if A[j] < A[minIdx] then  ┐
 5:          minIdx = j             ┘  ← ≤ c2
 6:    swap(A[i], A[minIdx])           ← ≤ c3
```

Annotations: line 2 $\leq c_1$; lines 4–5 $\leq c_2$; line 6 $\leq c_3$

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

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

Selection Sort: Analyse

UNI FREIBURG

```
SelectionSort(A):
 1: for i=0 to n-2 do
 2:    minIdx = i
 3:    for j=i to n-1 do
 4:       if A[j] < A[minIdx] then
 5:          minIdx = j
 6:    swap(A[i], A[minIdx])
```

Zeile 2: $\leq c_1$

Zeilen 4–5: $\leq c_2$, $\geq c_2'$

Zeile 6: $\leq c_3$

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

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

Selection Sort: Obere Schranke

UNI FREIBURG

$\boldsymbol{T(n)}$**:** Anzahl Grundop. von Selection Sort bei Arrays der Länge n

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

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

Laufzeitanalyse

UNI FREIBURG

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")

UNI FREIBURG

- 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: $\left.\begin{array}{l}\boldsymbol{T(n) \in O(n^2)} \\ \boldsymbol{T(n) \in \Omega(n^2)}\end{array}\right\} \boldsymbol{T(n) \in \Theta(n^2)}$

Landau-Symbole : Definitionen

UNI FREIBURG

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

- 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 $n \geq n_0$

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

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

- 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 $n \geq n_0$

$$\boldsymbol{\Theta}(\boldsymbol{g}(\boldsymbol{n})) \coloneqq \boldsymbol{O}(\boldsymbol{g}(\boldsymbol{n})) \cap \boldsymbol{\Omega}(\boldsymbol{g}(\boldsymbol{n}))$$

- 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

UNI FREIBURG

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

- 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)

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

- 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:

$$f(n) \in o(g(n)) \quad \Longrightarrow \quad f(n) \in O(g(n))$$

$$f(n) \in \omega(g(n)) \quad \Longrightarrow \quad f(n) \in \Omega(g(n))$$

Landau-Symbole : Intuitiv

UNI FREIBURG

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

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

Landau-Symbole : Intuitiv

UNI FREIBURG

$\boldsymbol{f(n) \in o(g(n))}$:

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

$\boldsymbol{f(n) \in \omega(g(n))}$:

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

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

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

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

"$f(n) < g(n)$" $\Longleftrightarrow$ "$f(n) \not\geq g(n)$"

Definition über Grenzwerte (vereinfacht)

UNI FREIBURG

Folgende Definitionen gelten für monoton wachsende Funktionen

$$f(n) \in O\big(g(n)\big), \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

$$f(n) \in \Omega\big(g(n)\big), \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$$

$$f(n) \in \Theta\big(g(n)\big), \qquad 0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

$$f(n) \in o\big(g(n)\big), \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) \in \omega\big(g(n)\big), \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Landau-Notation : Bemerkungen

UNI FREIBURG

Schreibweise:

- $O(g(n)), \Omega(g(n)), \ldots$ 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 $T(n) \in \Omega(n^2)$
- $f(n) = n^2 + O(n)$:

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

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

Landau-Notation : Bemerkungen

Schreibweise:

- $O(g(n)), \Omega(g(n)), \ldots$ 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 $x \to 0$
- z.B. Taylor-Reihen: $e^x = 1 + x + O(x^2)$, bzw. $e^x = 1 + x + o(x)$

Alternative Definition für $\boldsymbol{\Omega(g(n))}$:

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

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

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

- Wir verwenden die 1. Definition
- Macht nur bei nicht-monotonen Funktionen einen Unterschied

Landau-Notation : Beispiele

UNI FREIBURG

Selection Sort:

- Laufzeit $T(n)$, es gibt Konstanten $c_1, c_2 : c_1 n^2 \leq T(n) \leq c_2 n^2$

$$T(n) \in O(n^2), \qquad T(n) \in \Omega(n^2), \qquad T(n) \in \Theta(n^2)$$

- $T(n)$ wächst schneller als linear: $T(n) \in \omega(n)$

Weitere Beispiele:

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

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

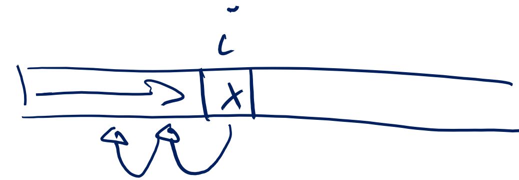
$$\frac{f(n)}{g(n)} = \frac{\sqrt{n}}{\log_2 n} = \frac{2^{t/2}}{t}$$

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

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

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(# Iterationen der while-Schleife)

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, best case, average case

UNI FREIBURG

Worst Case Analyse

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

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?

UNI FREIBURG

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

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

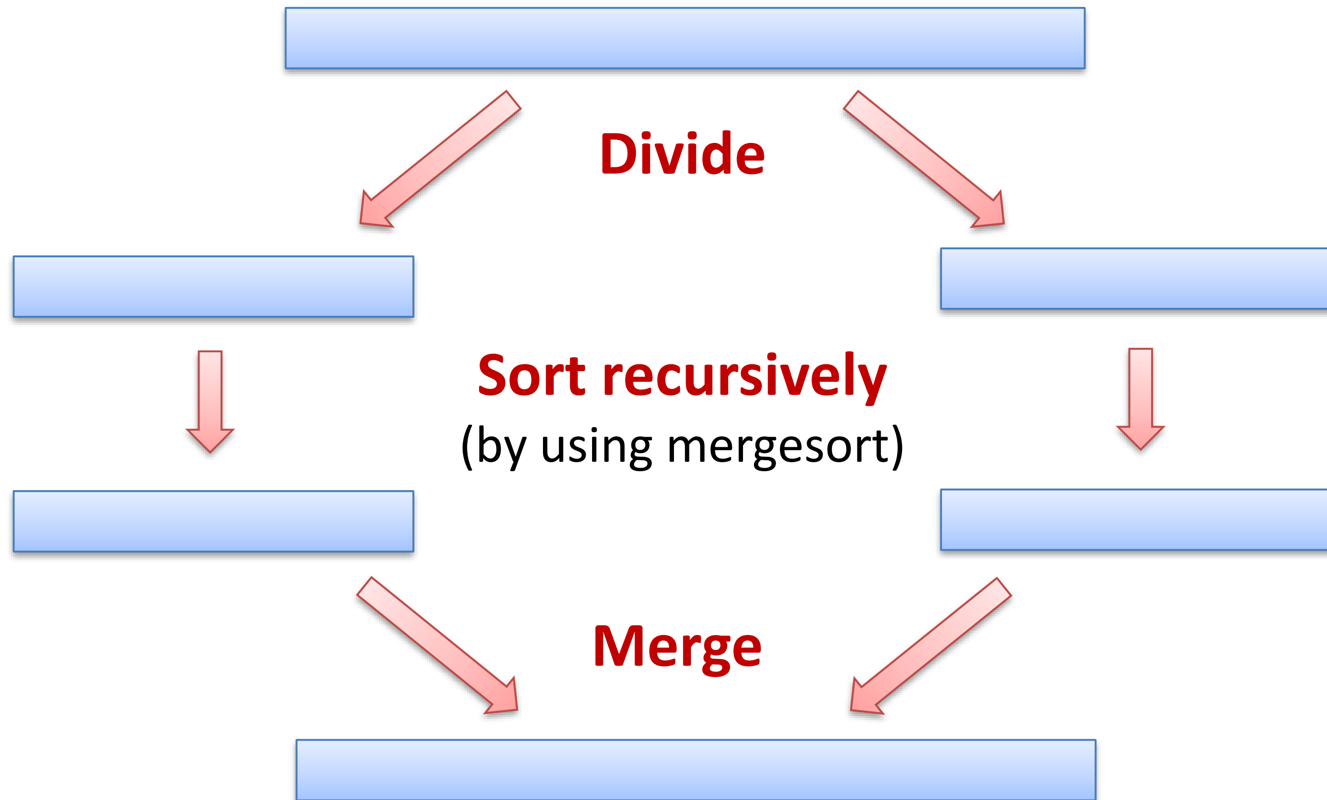
Beispielrechnung:

- 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!

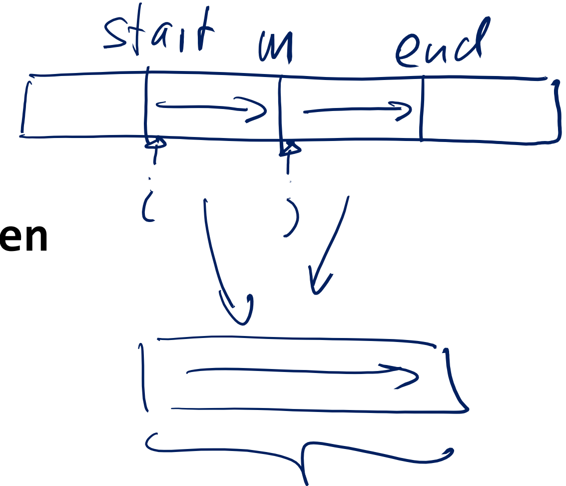
Analyse Merge Sort



- 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]
```



$k := end - start$

Schleifeniterationen:
while : k for : k

Laufzeit: $O(k)$

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

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

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

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

Analyse Merge Sort

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

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

$$T(n) \leq 2 \cdot T(n/2) + b \cdot n \qquad \left[T(n/2) \leq 2 \cdot T(n/4) + b \cdot \frac{n}{2} \right]$$

$$\leq 4 \cdot T(n/4) + b \cdot n + b \cdot n$$

$$= 4 \cdot T(n/4) + 2b \cdot n$$

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

$$= 8\,T(n/8) + 3 \cdot bn$$

$$\vdots$$

$$\leq 2^k T(n/2^k) + k \cdot b \cdot n$$

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

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: $n = 1$ $\quad T(1) \leq b \cdot 1 \, (1 + \log_2 1) = b$ ✓ (with $\log_2 1 = 0$)

Induktionsschritt:

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

$$T(n) \leq 2 \cdot T(n/2) + b \cdot n$$

$$\overset{(I.V.)}{\leq} 2 \cdot b \cdot \frac{n}{2} \cdot \left(1 + \log_2 \frac{n}{2}\right) + b \cdot n$$

($1 + \log_2 \frac{n}{2} = \log_2 n$)

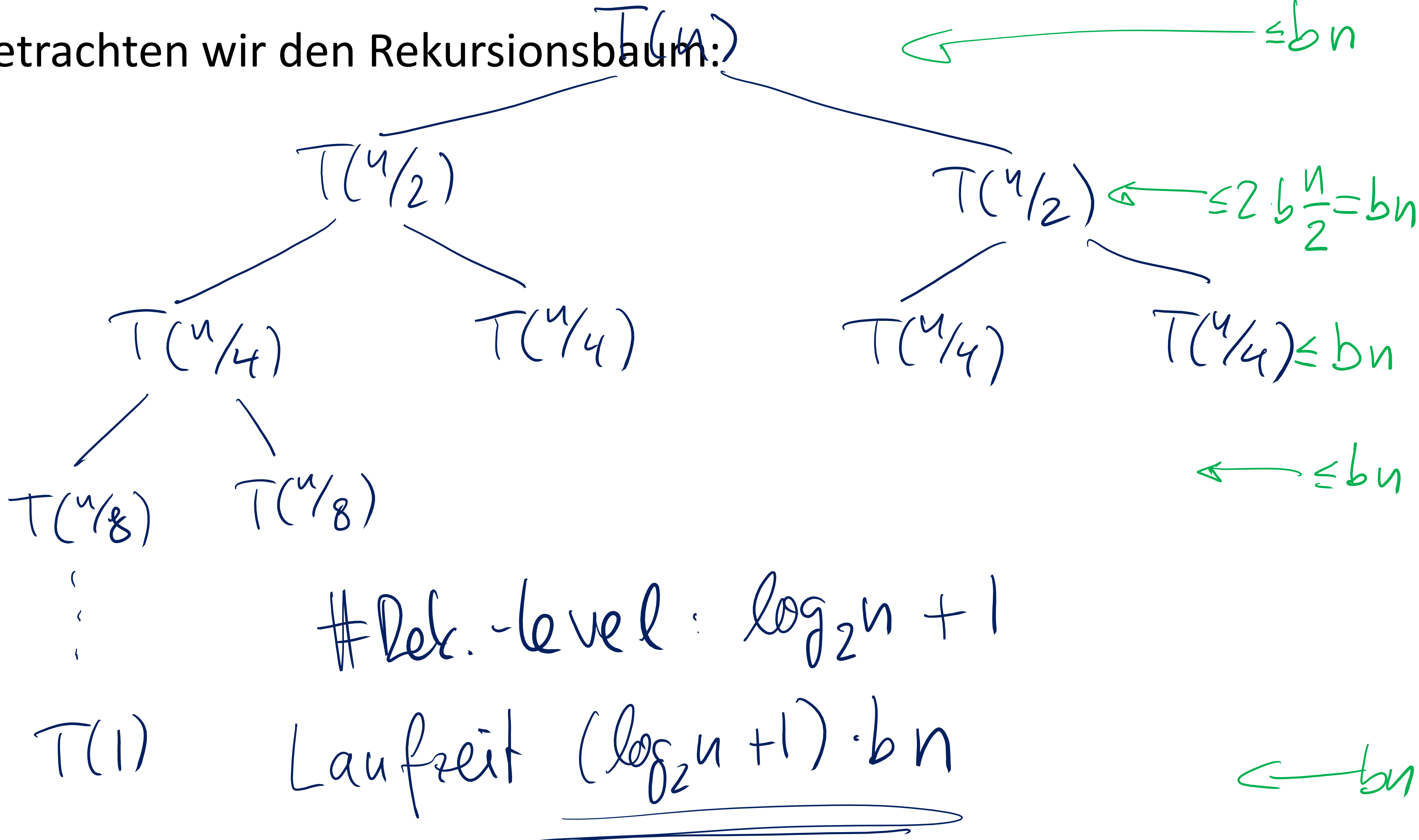
$$= bn\,(\log_2 n + 1)$$ ✓ □

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

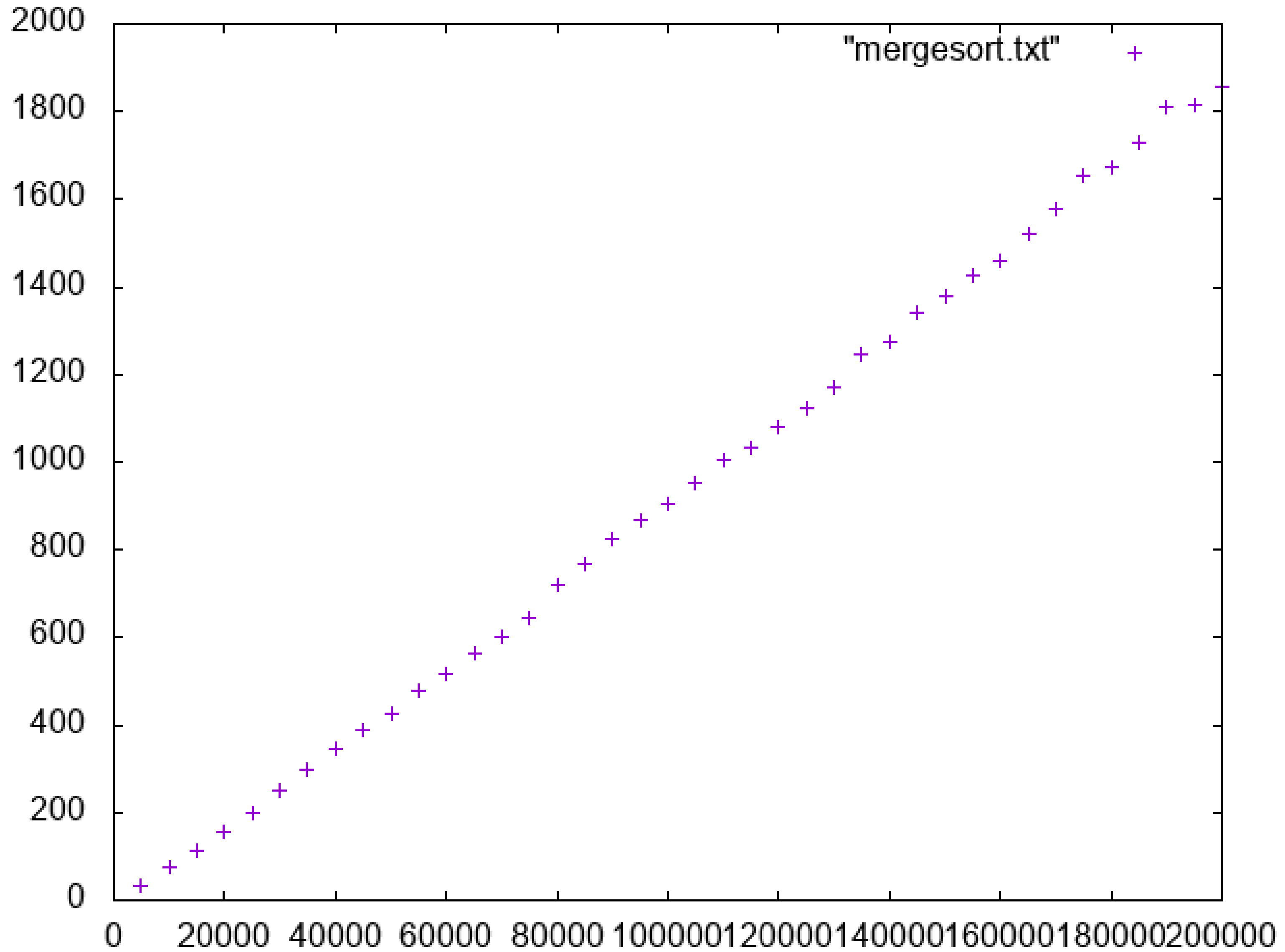
Alternative Analyse Merge Sort

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

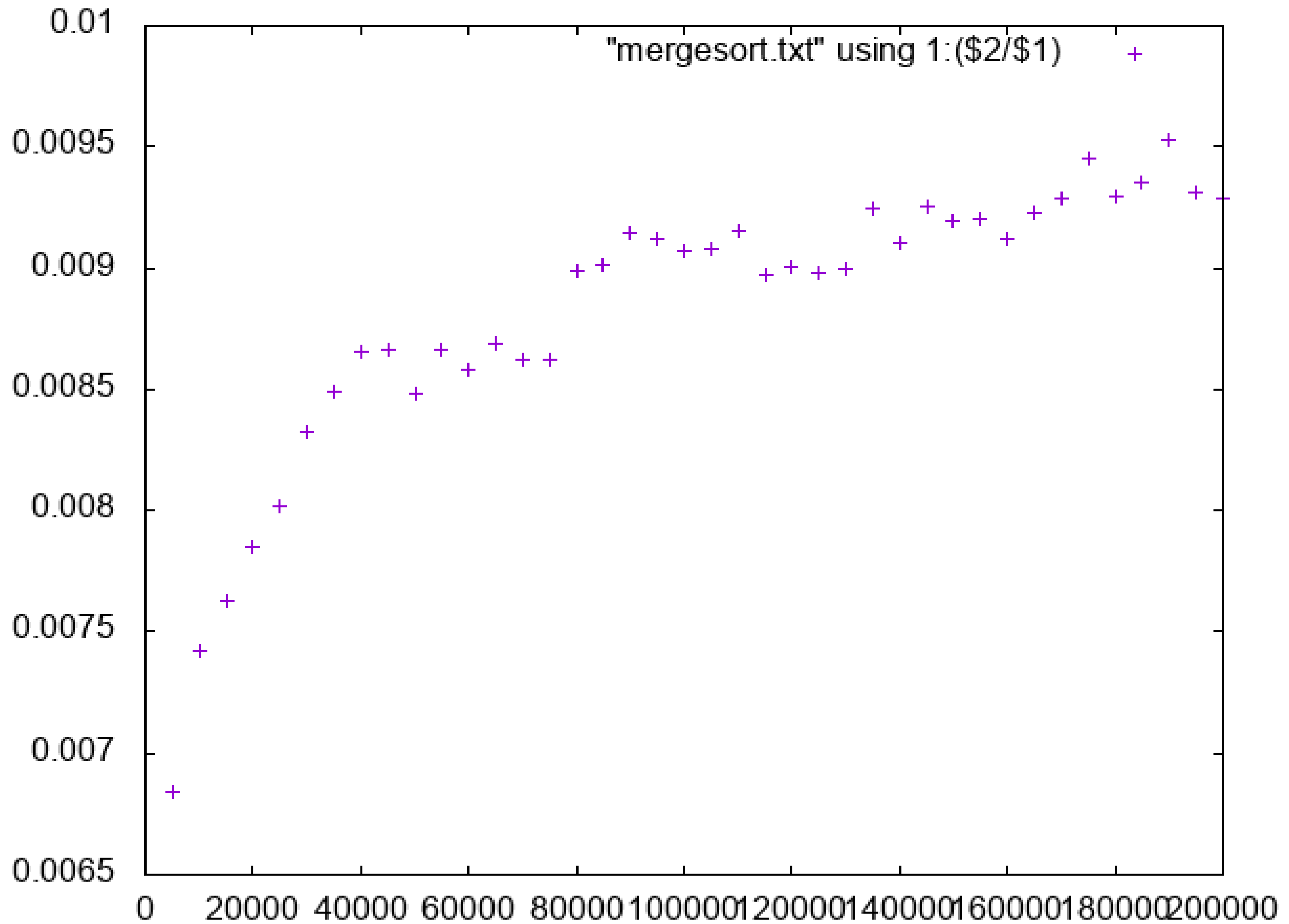
Betrachten wir den Rekursionsbaum:



Merge Sort Messungen



Merge Sort Messungen



Zusammenfassung Analyse Merge Sort

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

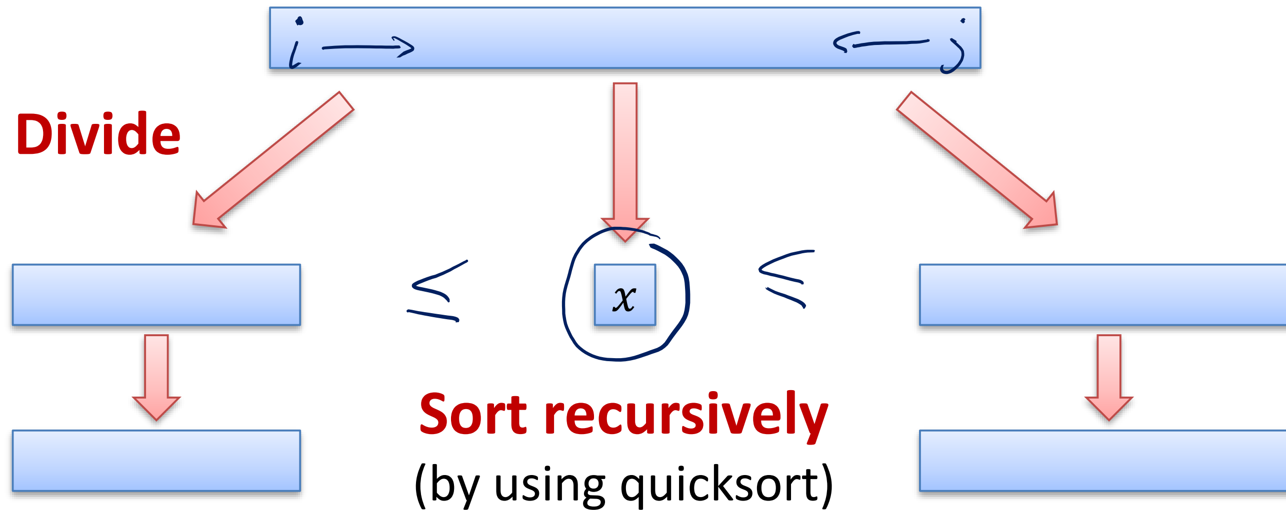
- wächst fast linear mit der Grösse der Eingabe…

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	$\approx 4\text{KB}$	$10 \cdot 10 \cdot 2^{10} \cdot 10^{-9}\ \text{s} \approx 0.1\ \text{ms}$	$1\ \text{ms}$
$2^{20} \approx 10^6$ Zahlen	$\approx 4\text{MB}$	$10 \cdot 20 \cdot 2^{20} \cdot 10^{-9}\ \text{s} \approx 0.2\ \text{s}$	$16.7\ \text{min}$
$2^{30} \approx 10^9$ Zahlen	$\approx 4\text{GB}$	$10 \cdot 30 \cdot 2^{30} \cdot 10^{-9}\ \text{s} \approx 5.4\ \text{min}$	31.7 Jahre
$2^{40} \approx 10^{12}$ Zahlen	$\approx 4\text{TB}$	$10 \cdot 40 \cdot 2^{40} \cdot 10^{-9}\ \text{s} \approx 122\ \text{h}$	$> 10^7$ 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:

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

- **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

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 $T(n) \in \Theta(n^2)$:

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

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

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

Schritt:

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

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

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

n/2

n/4

n/4

n

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:

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

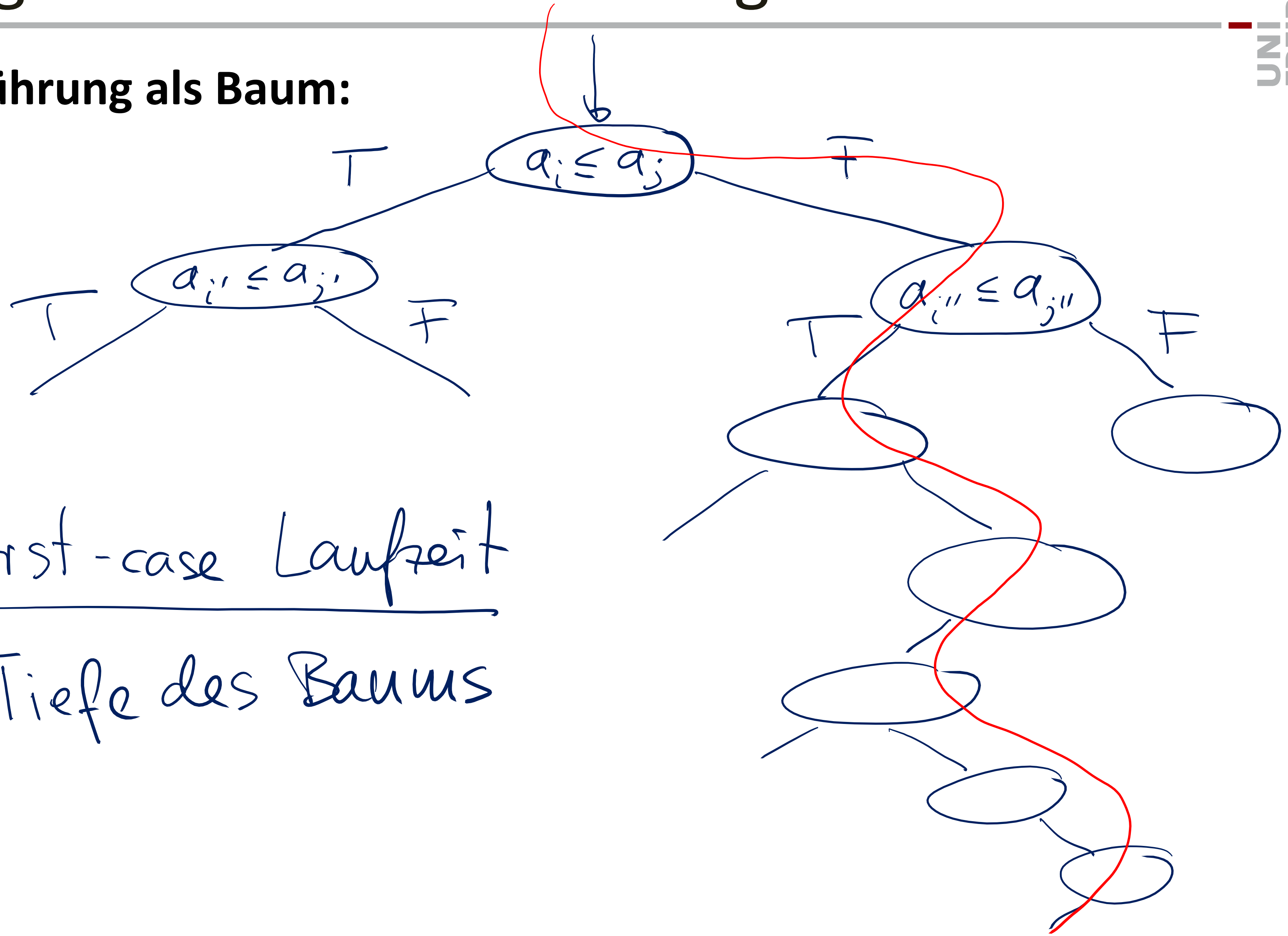
- In jeder Ausführung eines Algorithmus, induzieren die Resultate dieser Vergleiche eine Abfolge von T/F (true/false) Werten:

$$\mathbf{TFFTTTFTFFTTFFFFFTFTTT}\ \ldots$$

- 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):

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

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

Anzahl T/F-Folgen der Länge $\leq k$:

Länge $= k$: 2^k

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

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

Laufzeit $\leq T$

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

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

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

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

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

$$\log_2(n!) = \Theta(n \cdot \log n)$$

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

UNI FREIBURG

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
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++
```

$O(k)$

$O(n)$

$O(k+n)$