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                          ≤ c_1
 3:    for j=i to n-1 do
 4:       if A[j] < A[minIdx] then         ≤ c_2
 5:          minIdx = j
 6:    swap(A[i], A[minIdx])               ≤ c_3
```

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

$x(n)$

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

Selection Sort: Analyse

```
SelectionSort(A):
 1: for i=0 to n-2 do
 2:    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])
```

$\leq c_1$

$\geq c_2'$

$\leq c_2$

$\leq c_3$

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

$T(n)$ $\qquad$ $x(n) \leq n^2$

Laufzeit $T(n) \leq c \cdot n^2$ $\qquad$ $T(n) \geq c_2' \cdot n^2$

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")

- 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: $\boldsymbol{T(n) \in O(n^2)}$, $\boldsymbol{T(n) \in \Omega(n^2)}$ $\Rightarrow$ $\boldsymbol{T(n) \in \Theta(n^2)}$

Landau-Symbole : Definitionen

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

- 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 $n \geq n_0$

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

- 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 $n \geq n_0$

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

- 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

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

- 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)

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

- 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:

$$f(n) \in o\big(g(n)\big) \quad \Rightarrow \quad f(n) \in O\big(g(n)\big)$$

$$f(n) \in \omega\big(g(n)\big) \quad \Rightarrow \quad f(n) \in \Omega\big(g(n)\big)$$

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

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

- $f(n)$ " $<$ " $g(n)$, 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 $g(n)$ "vernünftige" wachsende Funktionen sind, gilt:

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

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

Definition über Grenzwerte (vereinfacht)

Folgende Definitionen gelten falls der Grenzwert definiert ist:

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

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

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

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

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

Landau-Notation : Bemerkungen

Schreibweise:

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

Beispiele:

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

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

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

Landau-Notation : Bemerkungen

Schreibweise:

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

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

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

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

- Wir verwenden die 1. Definition
- Definitionen sind äquivalent falls $\lim_{n\to\infty} f(n)/g(n)$ definiert ist.

Landau-Notation : Beispiele

$\log_a x = \frac{\log_b x}{\log_b a}$

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:

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

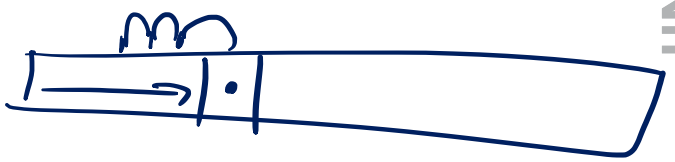
$$\lim_{n\to\infty} \frac{e^n}{n^{100}} \to \infty$$

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

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

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

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 $\in O(\underbrace{\#\text{Iter. der while-Schleife}}_{X(n)})$

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

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

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…

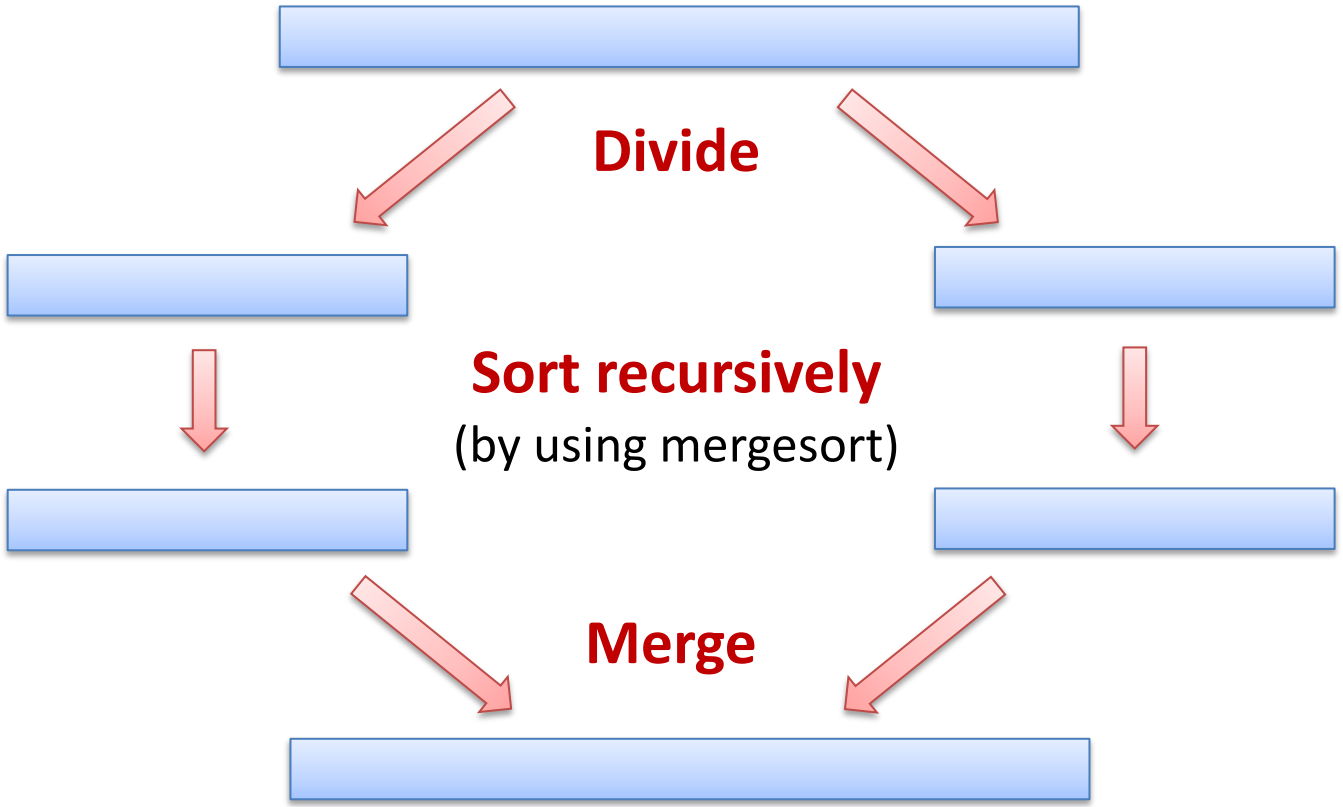
Beispielrechnung:

- 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!

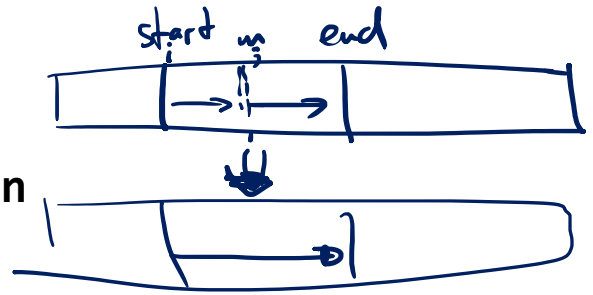
Analyse Merge Sort



- 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]
```



$k := end - start$

#Schleifenop.:

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

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

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

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

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

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

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

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

$\vdots$

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

$\overset{(k=\log_2 n)}{=} n \cdot T(1) + b \cdot n \cdot \log_2 n \leq bn(1 + \log_2 n)$

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

Schritt: (Induktionsvoraussetzung: Vermutung gilt für alle Werte $< n$)

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

Ind.-voraussetzung: $T(n/2) \leq b \cdot \frac{n}{2}\left(1 + \log_2 \frac{n}{2}\right) = b \cdot \frac{n}{2} \cdot \log_2 n$ ($\log_2 n - \log_2 2$, $\log_2 2 = 1$)

$$\leq 2\left[b \cdot \frac{n}{2} \cdot \log_2 n\right] + b \cdot n$$

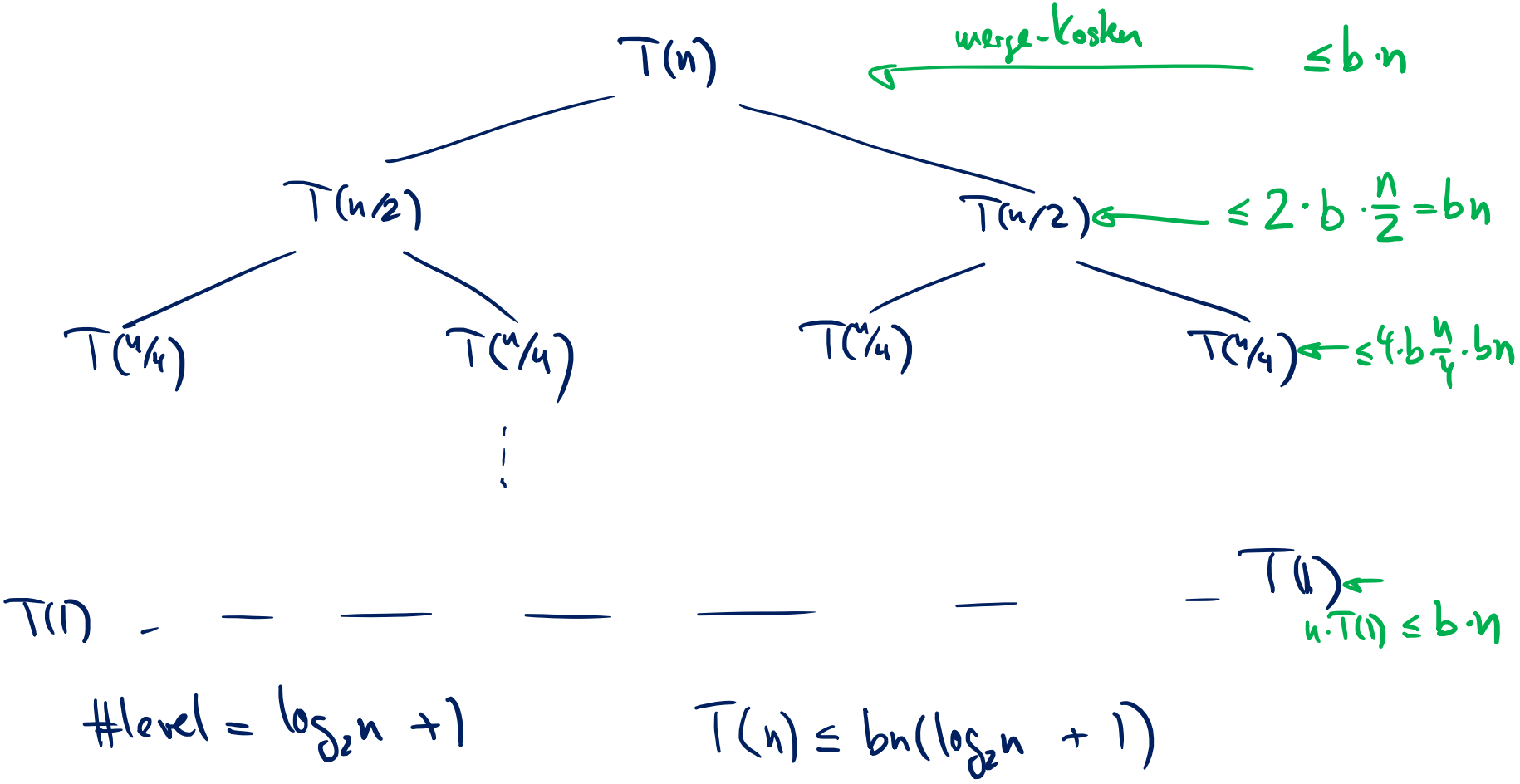
$$= bn \cdot (1 + \log_2 n).$$ ✓ □

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

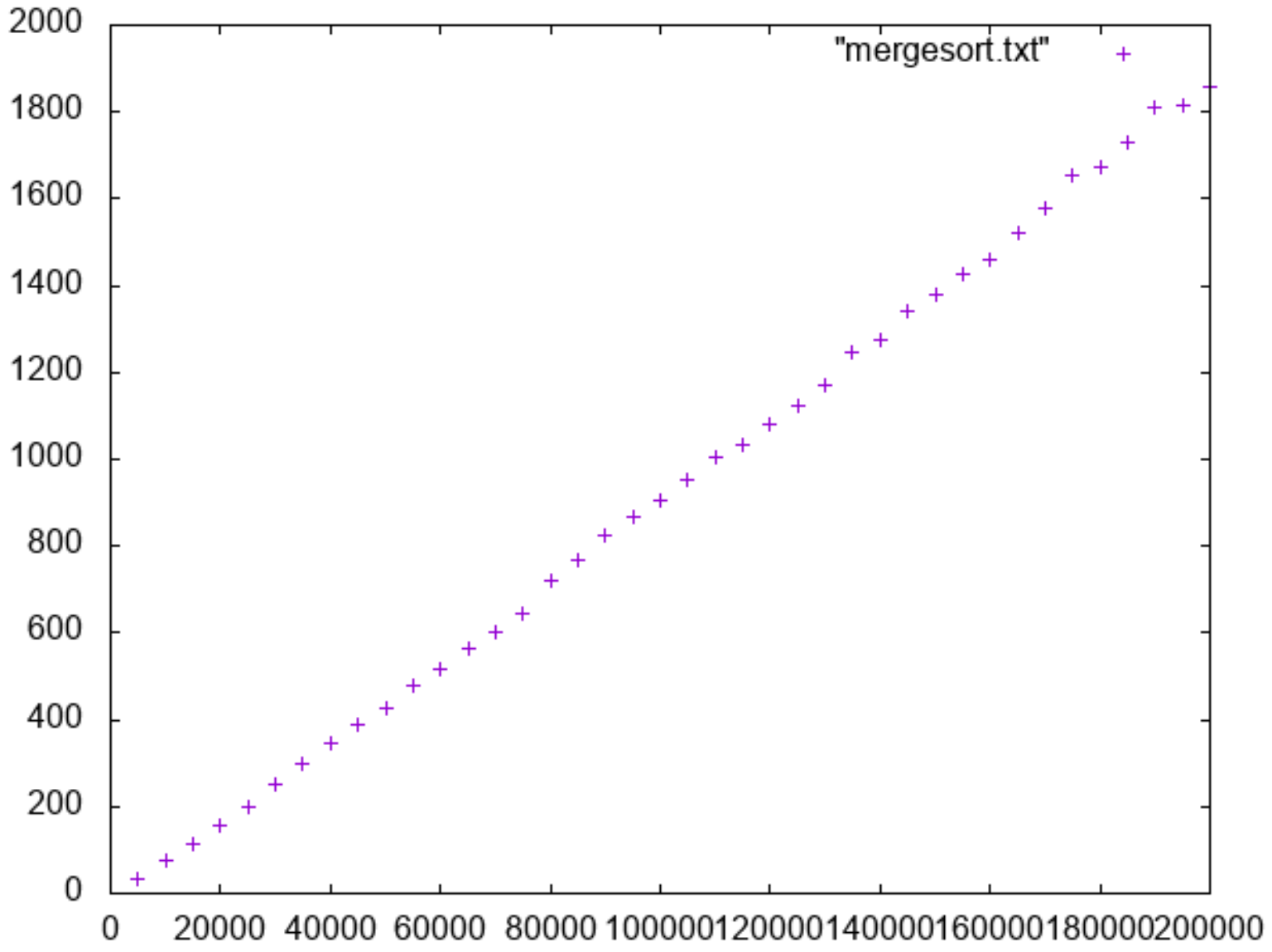
Alternative Analyse Merge Sort

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

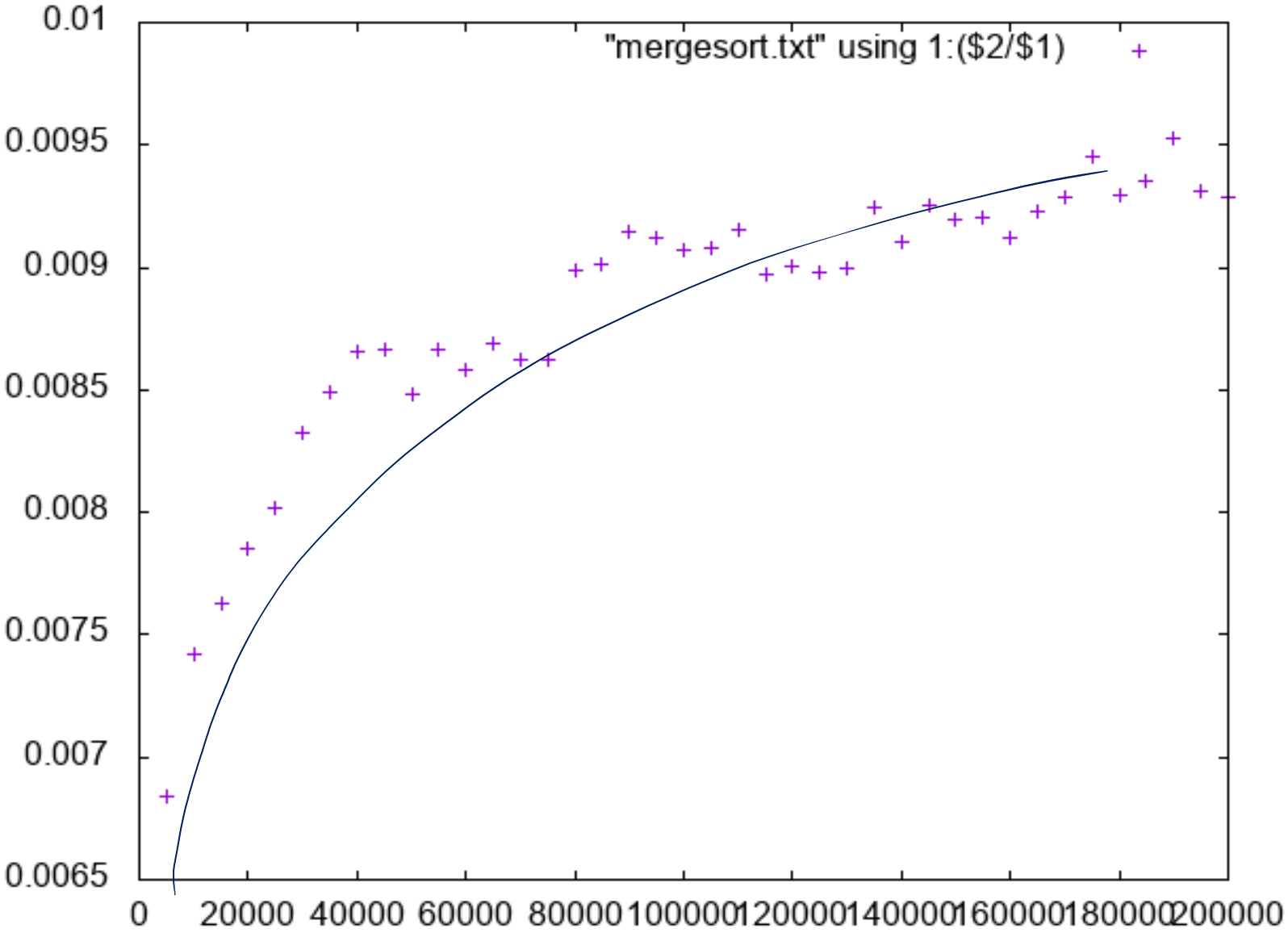
Betrachten wir den Rekursionsbaum:



Merge Sort Messungen



Merge Sort Messungen



Zusammenfassung Analyse Merge Sort

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	$\approx 4\text{KB}$	$10 \cdot 10 \cdot 2^{10} \cdot 10^{-9}\ \text{s} \approx 0.1\ \text{ms}$	$1\ \text{ms}$
$2^{20} \approx 10^6$ 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