Karlsruhe Institute of Technology

1. Zusatztutorium

$\mathcal{O}$-Notation, Rekurrenz, Mastertheorem, Amortisierung

Algorithmen I, Zusatz-Tutorium

Joshua Jantzen | 28. Juli 2024

FAKULTÄT FÜR INFORMATIK

Roadmap

Roadmap

Stilblüten - Allgemein

- wir wollen möglichst effiziente Algorithmen
  $\rightsquigarrow$ Fragt euch immer: Ginge es besser? Was könnte weggelassen werden?

Stilblüten - Allgemein

- wir wollen möglichst effiziente Algorithmen
  $\rightsquigarrow$ Fragt euch immer: Ginge es besser? Was könnte weggelassen werden?
- Teilaufgaben wollen euch helfen!
  - können Ideen für die späteren Teilaufgaben liefern
  - können euch helfen einen Datenstruktur / einen Algorithmus zu verstehen

  $\Rightarrow$ Überlegt, wie ihr Erkenntnisse aus vorigen Teilaufgaben nutzen könnt!

Stilblüten - Allgemein

- wir wollen möglichst effiziente Algorithmen
  $\rightsquigarrow$ Fragt euch immer: Ginge es besser? Was könnte weggelassen werden?
- Teilaufgaben wollen euch helfen!
  - können Ideen für die späteren Teilaufgaben liefern
  - können euch helfen einen Datenstruktur / einen Algorithmus zu verstehen

  $\Rightarrow$ Überlegt, wie ihr Erkenntnisse aus vorigen Teilaufgaben nutzen könnt!
- führt eure Variablen ein (Name, Typ, im Fließtext: Nutzen!)
- nutzt die Variablen, die ihr einführt

Stilblüten - Allgemein

- wir wollen möglichst effiziente Algorithmen
  $\rightsquigarrow$ Fragt euch immer: Ginge es besser? Was könnte weggelassen werden?
- Teilaufgaben wollen euch helfen!
  - können Ideen für die späteren Teilaufgaben liefern
  - können euch helfen einen Datenstruktur / einen Algorithmus zu verstehen

  $\Rightarrow$ Überlegt, wie ihr Erkenntnisse aus vorigen Teilaufgaben nutzen könnt!
- führt eure Variablen ein (Name, Typ, im Fließtext: Nutzen!)
- nutzt die Variablen, die ihr einführt
- mehr ist nicht besser: vermeidet Füllwörter (so viel wie nötig, so wenig wie möglich)
- besonders ungünstig: vage Füllwörter / Verwässern eurer Beschreibung («könnte» , «eventuell» etc.)

Der böse Pseudo-Pseudocode

Wie war das nochmal mit den Abstraktionsebenen?

Ob wir einen Algorithmus in Fließtext oder in Pseudocode angeben, macht einen großen Unterschied. Dabei geht es nicht nur um die Form, sondern auch um den inhaltlichen Anspruch:

Der böse Pseudo-Pseudocode

Wie war das nochmal mit den Abstraktionsebenen?

Ob wir einen Algorithmus in Fließtext oder in Pseudocode angeben, macht einen großen Unterschied. Dabei geht es nicht nur um die Form, sondern auch um den inhaltlichen Anspruch:

- Fließtext: **Was** leistet der Algorithmus?
- Pseudocode: **Wie** funktioniert der Algorithmus?

bzw.

- Fließtext: **high** level Betrachtung
- Pseudocode: **low** level Betrachtung

Der böse Pseudo-Pseudocode

Wie war das nochmal mit den Abstraktionsebenen?

Ob wir einen Algorithmus in Fließtext oder in Pseudocode angeben, macht einen großen Unterschied. Dabei geht es nicht nur um die Form, sondern auch um den inhaltlichen Anspruch:

- Fließtext: **Was** leistet der Algorithmus?
- Pseudocode: **Wie** funktioniert der Algorithmus?

bzw.

- Fließtext: **high** level Betrachtung
- Pseudocode: **low** level Betrachtung

Wenn ihr einen Algorithmus in Fließtext angeben sollt, dann solltet ihr darüber nachdenken und euch **nicht** an Implementierungsdetails aufhängen.

Der böse Pseudo-Pseudocode

Beispiel 1: Iterieren über ein Array

Der böse Pseudo-Pseudocode

Beispiel 1: Iterieren über ein Array

$A\colon [\mathbb{R};\ n]$
$\texttt{counter}\colon \mathbb{N} = 0$
$\ldots$
for $i \in \{0, \ldots, n-1\}$ **do**
    if $A[i] = 42$ **then**
        $\texttt{counter}{+}{+}$

Der böse Pseudo-Pseudocode

Beispiel 1: Iterieren über ein Array

$A\colon [\mathbb{R};\ n]$
$\texttt{counter}\colon \mathbb{N} = 0$
$\ldots$
for $i \in \{0, \ldots, n-1\}$ **do**
  if $A[i] = 42$ **then**
    `counter++`

Wir legen eine Zählvariable `counter` an, die zu 0 initialisiert wird. Nun laufen wir mit einer for-Schleife über das Array A und inkrementieren `counter`, wenn das aktuell betrachtete Element von A den Wert 42 hat.

Der böse Pseudo-Pseudocode

Beispiel 1: Iterieren über ein Array

$A\colon [\mathbb{R};\ n]$
`counter`$\colon \mathbb{N} = 0$
$\ldots$
for $i \in \{0, \ldots, n-1\}$ **do**
    if $A[i] = 42$ **then**
        `counter++`

Wir legen eine Zählvariable `counter` an, die zu 0 initialisiert wird. Nun laufen wir mit einer for-Schleife über das Array A und inkrementieren `counter`, wenn das aktuell betrachtete Element von A den Wert 42 hat.

Der böse Pseudo-Pseudocode

Beispiel 1: Iterieren über ein Array

$A\colon [\mathbb{R};\ n]$
$\texttt{counter}\colon \mathbb{N} = 0$
...
for $i \in \{0, \ldots, n-1\}$ **do**
    if $A[i] = 42$ **then**
        $\texttt{counter}{+}{+}$

Wir zählen, wie oft der Wert 42 in A vorkommt.

Der böse Pseudo-Pseudocode

Beispiel 2: Swapping

Der böse Pseudo-Pseudocode

Beispiel 2: Swapping

$A\colon [\mathbb{R};\ n]$
$\texttt{pivot}\colon \mathbb{R} = A[n-1]$
$i, j\colon \mathbb{N} = 0, n-1$
while $i < j$ **do**
  while $i < n-1 \land A[i] < \texttt{pivot}$ **do**
    i++
  while $i > 0 \land A[j] \geq \texttt{pivot}$ **do**
    j−
  if $i < j$ **then**
    SWAP$(A[i], A[j])$
SWAP$(A[i], A[n-1])$

Der böse Pseudo-Pseudocode

Beispiel 2: Swapping

$A\colon [\mathbb{R};\ n]$
$\texttt{pivot}\colon \mathbb{R} = A[n-1]$
$i, j\colon \mathbb{N} = 0, n-1$
while $i < j$ **do**
  while $i < n-1 \land A[i] < \texttt{pivot}$ **do**
    i++
  while $i > 0 \land A[j] \geq \texttt{pivot}$ **do**
    j−
  if $i < j$ **then**
    SWAP$(A[i], A[j])$
SWAP$(A[i], A[n-1])$

Wir wählen $A[n-1]$ als Pivotelement. Nun legen wir zwei Variablen $i = 0, j = n-2$ an und treten in eine while-Schleife ein, die solange läuft, wie $i < j$ ist. In jeder Iteration wird i solange erhöht, bis $A[i] \geq A[n-1]$ und j wird so lange dekrementiert, bis $A[j] < A[n-1]$. Gilt dann $i < j$, tauschen wir $A[i]$ und $A[j]$. Nach Austritt aus der while-Schleife tauschen wir $A[i]$ mit $A[n-1]$

Der böse Pseudo-Pseudocode

Beispiel 2: Swapping

```
A: [ℝ; n]
pivot: ℝ = A[n − 1]
i, j: ℕ = 0, n − 1
while i < j do
    while i < n − 1 ∧ A[i] < pivot do
        i++
    while i > 0 ∧ A[j] ≥ pivot do
        j−
    if i < j then
        swap(A[i], A[j])
swap(A[i], A[n − 1])
```

Wir wählen $A[n-1]$ als Pivotelement. Nun legen wir zwei Variablen $i = 0$, $j = n - 1$ an und treten in eine äußere Schleife ein, die solange läuft wie $i < j$ ist. In jeder Iteration wird i solange erhöht, bis $A[i] \geq A[n-1]$ und j wird so lange dekrementiert, bis $A[j] < A[n-1]$. Gilt dann $i < j$, tauschen wir $A[i]$ und $A[j]$. Nach Austritt aus der while-Schleife tauschen wir $A[i]$ mit $A[n-1]$

Der böse Pseudo-Pseudocode

Beispiel 2: Swapping

```
A: [ℝ; n]
pivot: ℝ = A[n − 1]
i, j: ℕ = 0, n − 1
while i < j do
    while i < n − 1 ∧ A[i] < pivot do
        i++
    while i > 0 ∧ A[j] ≥ pivot do
        j−
    if i < j then
        swap(A[i], A[j])
swap(A[i], A[n − 1])
```

Wir partitionieren A anhand des Pivots $A[n-1]$.

Roadmap

Was war das nochmal?

Das $\mathcal{O}$-Kalkül

…ist ein Werkzeug, um das **asymptotische** Verhalten einer Funktion bzw. der Laufzeit eines Algorithmus anzugeben.

Was war das nochmal?

Das $\mathcal{O}$-Kalkül

…ist ein Werkzeug, um das **asymptotische** Verhalten einer Funktion bzw. der Laufzeit eines Algorithmus anzugeben.

Wozu brauchen wir das?

- Genaue Analyse ist kompliziert, bringt aber nur wenig relevante Einsicht
- **Ziel**: grobe Abschätzung, Blick auf das Wesentliche
- Wir wollen: Hilfsmittel, welches einfach über die Güte einer Laufzeit urteilen lässt

$\mathcal{O}$-Notation

Sei $f : \mathbb{N}_0 \to \mathbb{R}_0^+$. Dann ist

$$\mathcal{O}(f) = \{g : \mathbb{N}_0 \to \mathbb{R}_0^+ \mid \exists c \in \mathbb{R}_+ : \exists n_0 \in \mathbb{N}_0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

«g wächst asymptotisch höchstens so schnell wie f.»

$$\Omega(f) = \{g : \mathbb{N}_0 \to \mathbb{R}_0^+ \mid \exists c \in \mathbb{R}_+ : \exists n_0 \in \mathbb{N}_0 : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$$

«g wächst asymptotisch mindestens so schnell wie f.»

$$\begin{aligned}\Theta(f) &= \mathcal{O}(f) \cap \Omega(f) \\ &= \{g : \mathbb{N}_0 \to \mathbb{R}_0^+ \mid \exists c, c' \in \mathbb{R}_+ : \exists n_0 \in \mathbb{N}_0 : \\ &\forall n \geq n_0 : cf(n) \leq g(n) \leq c'f(n)\}\end{aligned}$$

Man sagt, g wächst asymptotisch genau so schnell wie f.

$\mathcal{O}$-Notation

Sei $f : \mathbb{N}_0 \to \mathbb{R}_0^+$. Dann ist

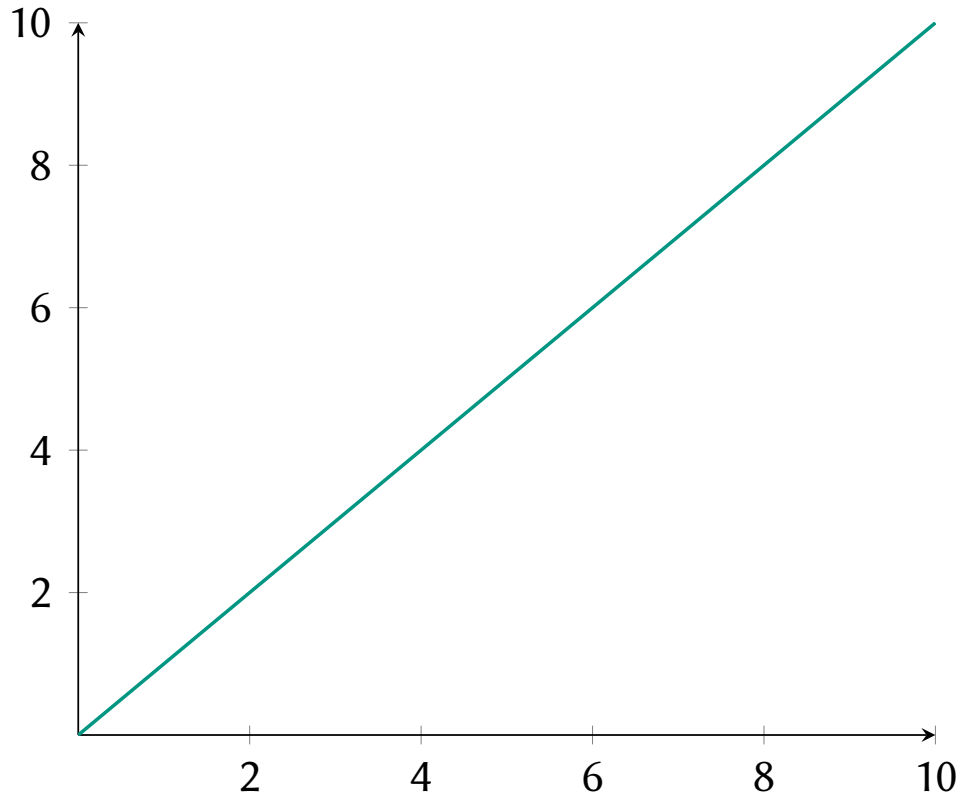
$$o(f) = \{g : \mathbb{N}_0 \to \mathbb{R}_0^+ \mid \forall c \in \mathbb{R}_+ : \exists n_0 \in \mathbb{N}_0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$
$$= \mathcal{O}(f) \setminus \Theta(f)$$

«g wächst asymptotisch langsamer als f.»

$$\omega(f) = \{g : \mathbb{N}_0 \to \mathbb{R}_0^+ \mid \forall c \in \mathbb{R}_+ : \exists n_0 \in \mathbb{N}_0 : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$$
$$= \Omega(f) \setminus \Theta(f)$$

«g wächst asymptotisch schneller als f.»

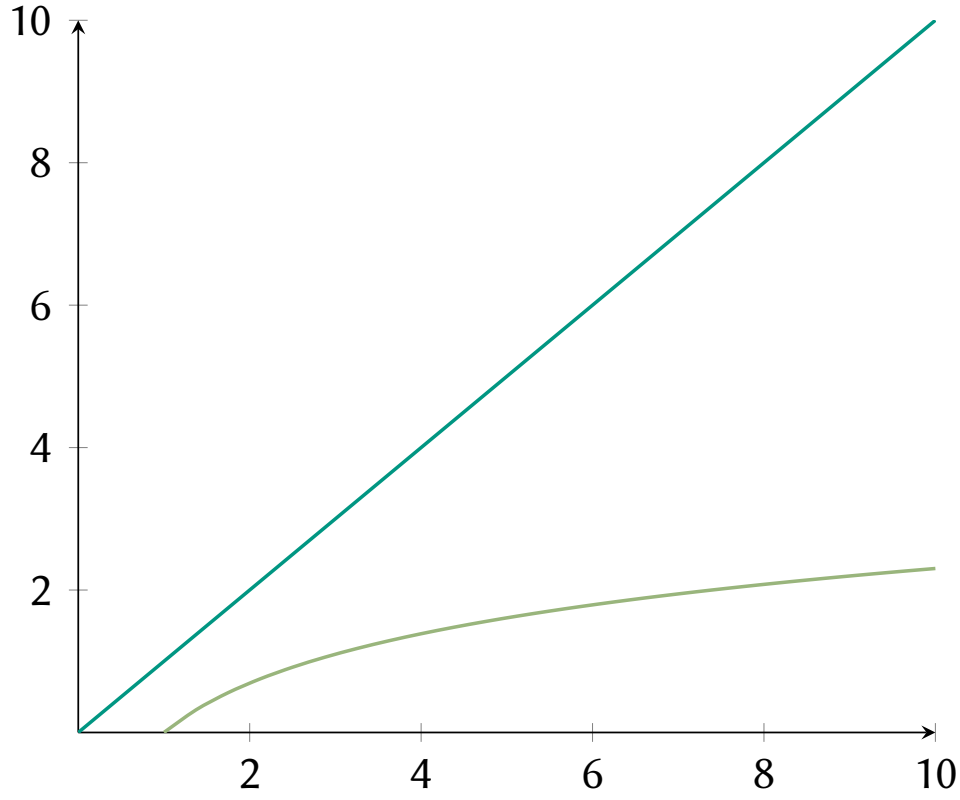
Elementare Funktionen



«Nice.»

linear: n

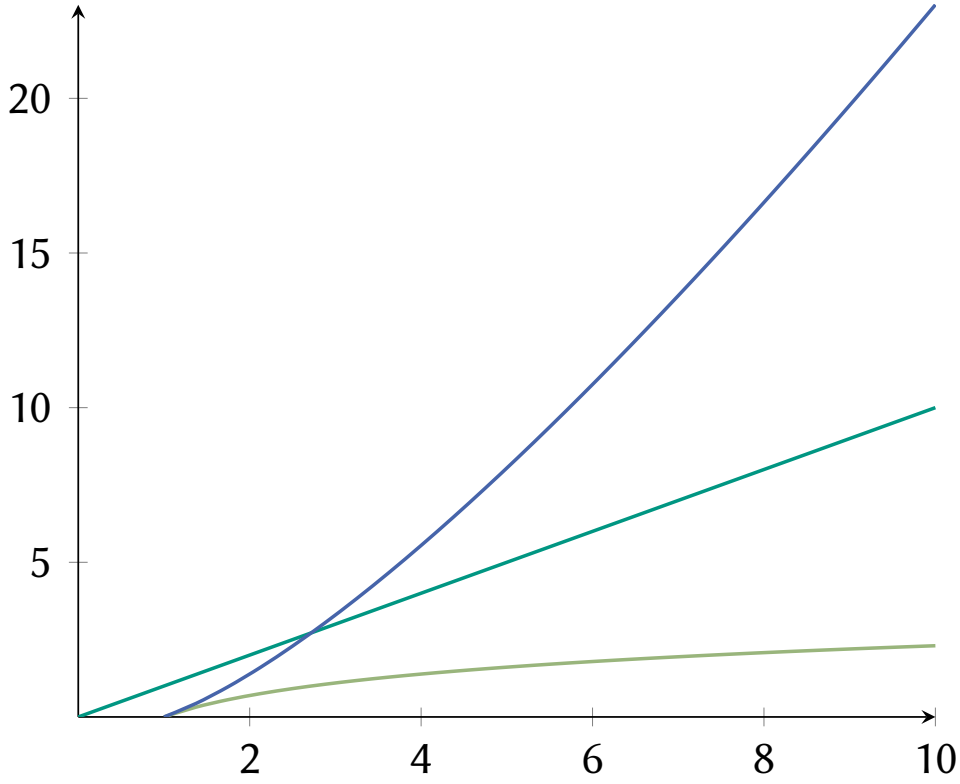
Elementare Funktionen



«Very nice.»

- sub-linear: $1, \log(n), \log^*(n), \sqrt[3]{n}, \sqrt{n}$
- linear: n

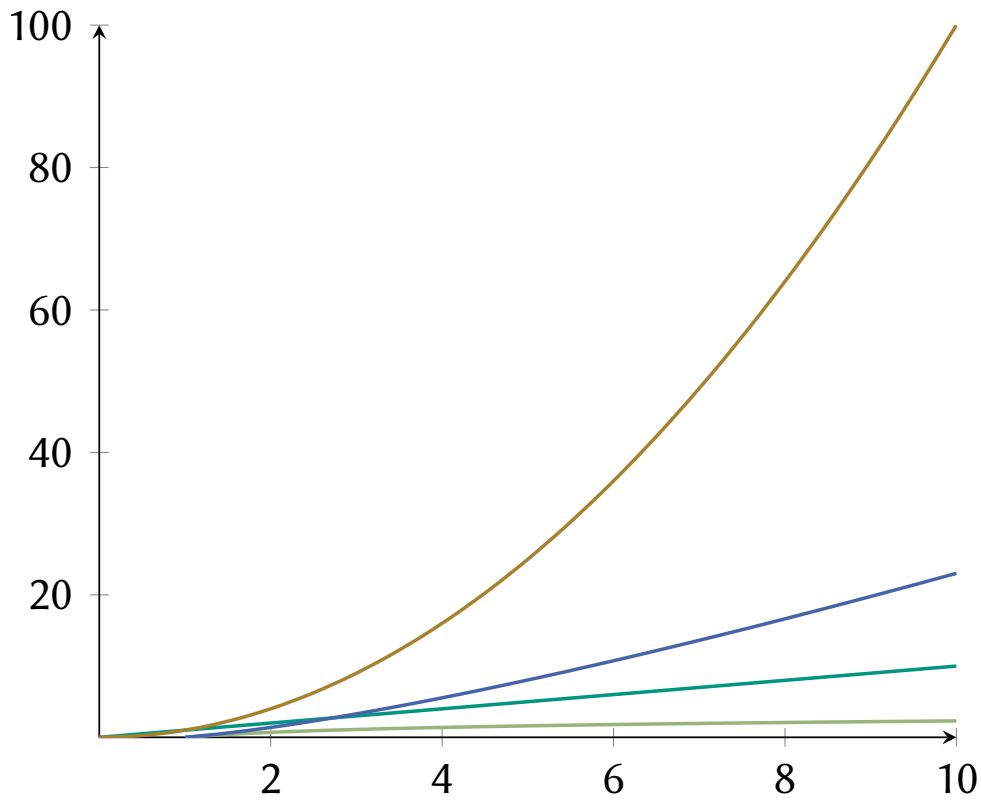
Elementare Funktionen



«Nice-ish.»

- sub-linear: $1, \log(n), \log^*(n), \sqrt[3]{n}, \sqrt{n}$
- linear: n
- quasi-linear: $n \log(n)$

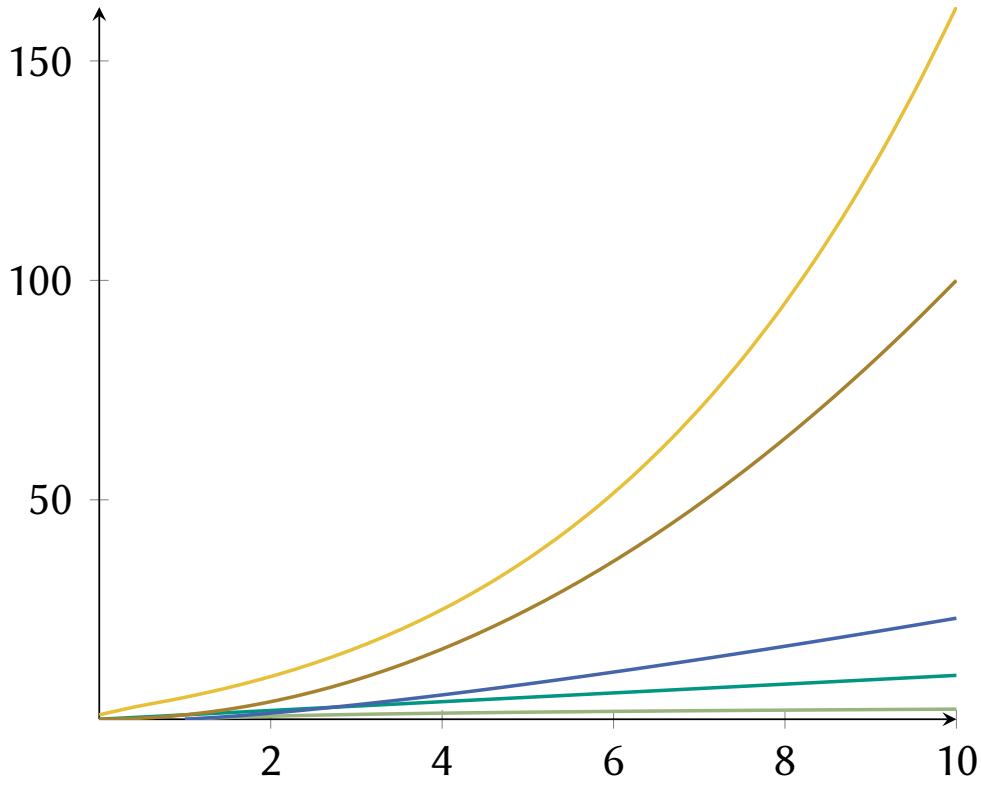
Elementare Funktionen



«OK.»

- sub-linear: $1, \log(n), \log^*(n), \sqrt[3]{n}, \sqrt{n}$
- linear: n
- quasi-linear: $n \log(n)$
- polynomiell: n^2, n^3
- quasi-polynomiell: $n^{\log(n)}$

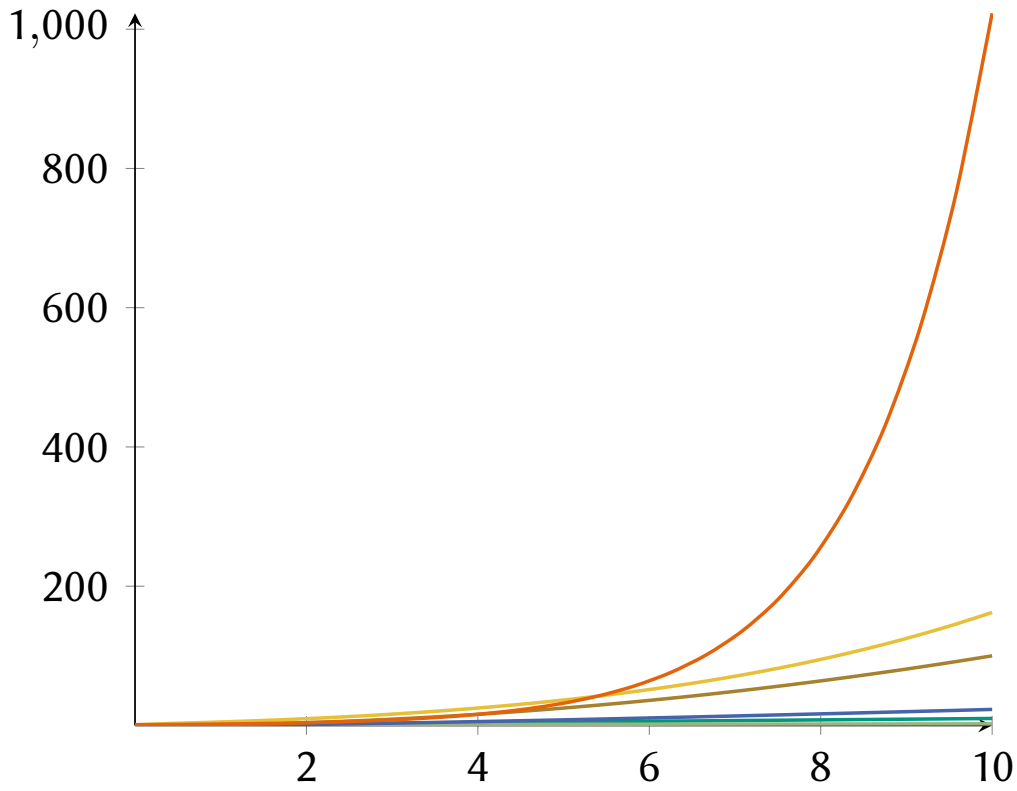
Elementare Funktionen



«Not OK.»

- sub-linear: $1, \log(n), \log^*(n), \sqrt[3]{n}, \sqrt{n}$
- linear: n
- quasi-linear: $n \log(n)$
- polynomiell: n^2, n^3
- quasi-polynomiell: $n^{\log(n)}$
- super-polynomiell, sub-exponentiell: $n^{\log(n)}, 2^{\sqrt{n}}$

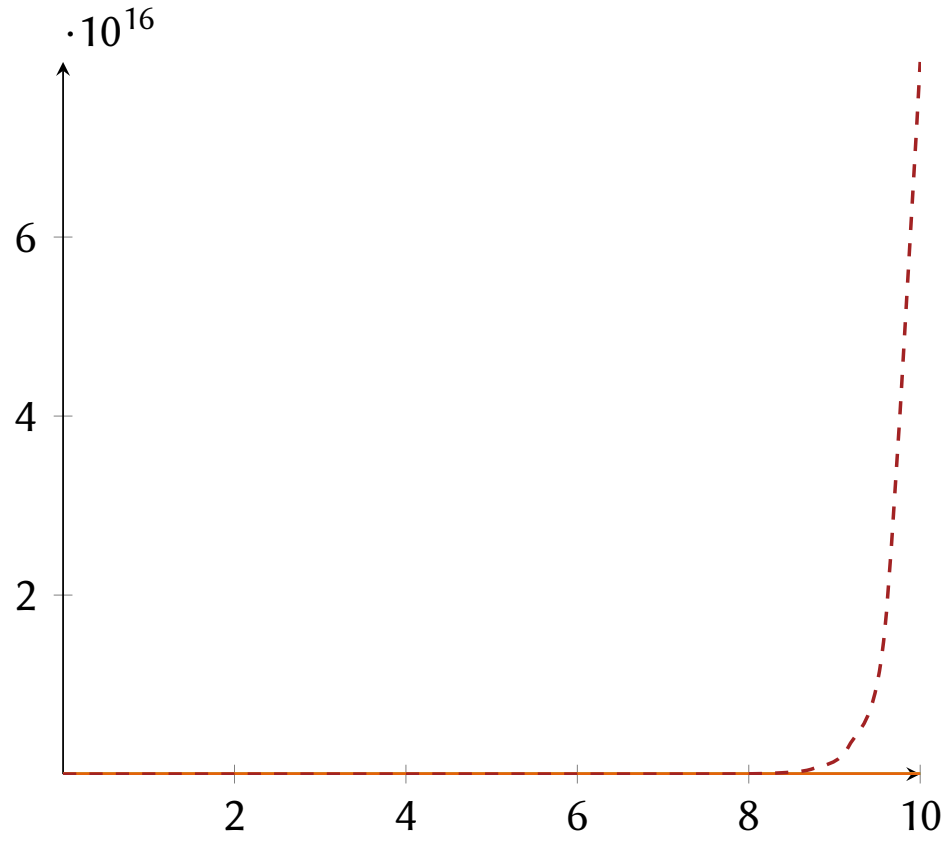
Elementare Funktionen



«Eeeeeeeew.»

- sub-linear: $1, \log(n), \log^*(n), \sqrt[3]{n}, \sqrt{n}$
- linear: n
- quasi-linear: $n \log(n)$
- polynomiell: n^2, n^3
- quasi-polynomiell: $n^{\log(n)}$
- super-polynomiell, sub-exponentiell: $n^{\log(n)}, 2^{\sqrt{n}}$
- exponentiell: $2^n, 3^n, 4^n$

Elementare Funktionen



«Oh god, oh no.»

- sub-linear: $1, \log(n), \log^*(n), \sqrt[3]{n}, \sqrt{n}$
- linear: n
- quasi-linear: $n \log(n)$
- polynomiell: n^2, n^3
- quasi-polynomiell: $n^{\log(n)}$
- super-polynomiell, sub-exponentiell: $n^{\log(n)}, 2^{\sqrt{n}}$
- exponentiell: $2^n, 3^n, 4^n$
- super-exponentiell: $n!, 2^{n^2}$

Grenzwertbetrachtung I

Seien $f, g\colon \mathbb{N}_0 \to \mathbb{R}_0^+$. Dann gilt:

$$f \in \mathcal{O}(g) \Leftrightarrow 0 \leq \limsup_{n\to\infty} \frac{f}{g} < \infty$$

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

$$f \in \Omega(g) \Leftrightarrow 0 < \liminf_{n\to\infty} \frac{f}{g} \leq \infty$$

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

$$f \in \omega(g) \Leftrightarrow \limsup_{n\to\infty} \frac{f}{g} = \infty$$

Grenzwertbetrachtung II

Die Regel von de l'Hospital

Seien $f, g\colon I \to \mathbb{R}$ differenzierbare Funktionen mit

$$\lim_{x\to x_0} f(x) = \lim_{x\to x_0} g(x) = 0 \qquad \text{oder} \qquad \lim_{x\to x_0} f(x) = \lim_{x\to x_0} g(x) = \infty$$

Dann gilt:

$$\text{Wenn } \lim_{x\to x_0} \frac{f'(x)}{g'(x)} \text{ existiert, dann ist } \lim_{x\to x_0} \frac{f(x)}{g(x)} = \lim_{x\to x_0} \frac{f'(x)}{g'(x)}$$

Beweise mittels Induktion

Um asymptotische Abschätzungen mit vollständiger Induktion zu beweisen, verwendet man die formale Definition des entsprechenden Landau-Symbols:

Bsp.: Beweise $n^3 \in \Omega(n^2)$

Beweise mittels Induktion

Um asymptotische Abschätzungen mit vollständiger Induktion zu beweisen, verwendet man die formale Definition des entsprechenden Landau-Symbols:

Bsp.: Beweise $n^3 \in \Omega(n^2)$

- Definition:
  $\Omega(n^2) = \{g \colon \mathbb{N}_0 \to \mathbb{R}_0^+ \mid \exists c \in \mathbb{R}_+ : \exists n_0 \in \mathbb{N}_0 : \forall n \geq n_0 : g(n) \geq cn^2\}$
- Wähle **ein** $c \in \mathbb{R}_+$ und **ein** $n_0 \in \mathbb{N}_0$ und zeige, dass $n^3 \geq c \cdot n^2$ für alle $n \geq n_0$
- Wie sähe das z.B. für $c = 1, n_0 = 1$ aus?

Beweise mittels Induktion

Um asymptotische Abschätzungen mit vollständiger Induktion zu beweisen, verwendet man die formale Definition des entsprechenden Landau-Symbols:

Bsp.: Beweise $n^3 \in \Omega(n^2)$

IA: $n = n_0 = 1$: $\quad n^3 = 1^3 \geq 1^2 = n^2 \quad \checkmark$

IV: Es gelte $n^3 \geq n^2$ für ein beliebiges aber festes $n \geq n_0$

IS: $n \rightsquigarrow n+1$

$$
\begin{aligned}
(n+1)^3 &= n^3 + 3n^2 \cdot 1 + 3n \cdot 1^2 + 1^3 \\
&\overset{\text{IV}}{\geq} 4n^2 + 3n + 1 \\
&\overset{n \geq n_0}{\geq} n^2 + 2n + 1 \\
&= (n+1)^2
\end{aligned}
$$

□

Merkregeln für den Logarithmus

- $\log(ab) = \log(a) + \log(b)$
- $\log(\frac{a}{b}) = \log(a) - \log(b)$
- $a^{\log_a(b)} = b$
- $a^x = e^{\ln(a) \cdot x}$
- $\log(a^b) = b \cdot \log(a)$
- $\log_b(n) = \frac{\log_a(n)}{\log_a(b)}$

Anmerkungen

- Θ entspricht **nicht** dem average case!
- Nur die dominante, d.h. am stärksten wachsende Komponente, ist relevant
- Konstante Faktoren werden nicht beachtet
- $\mathcal{O}(\log n)$, aber zu welcher Basis?
  - In der Informatik: üblicherweise Basis 2
  - Kann in $\mathcal{O}$-Notation vernachlässigt werden, denn

$$\mathcal{O}(\log_b n) = \mathcal{O}(\frac{\log_a n}{\log_a b}) = \mathcal{O}(\log_a n)$$

Roadmap

Rekursive Algorithmen

Eine rekursiver Algorithmus ruft sich im Laufe seiner Ausführung selbst erneut auf.

Abbruchbedingung

Ganz besonders bei der Modellierung eines rekursiven Algorithmus zu beachten ist die Abbruchbedingung. Sie wird der Rekursion vorangestellt und bricht somit, wie der Name schon sagt, ein tieferes Absteigen in die Rekursion ab, wenn sie erfüllt wird. Hat ein rekursiver Algorithmus kein Abbruchbedingung kommt dies einer Endlosschleife gleich.

Rekursive Algorithmen

Eine rekursiver Algorithmus ruft sich im Laufe seiner Ausführung selbst erneut auf.

Abbruchbedingung

Ganz besonders bei der Modellierung eines rekursiven Algorithmus zu beachten ist die Abbruchbedingung. Sie wird der Rekursion vorangestellt und bricht somit, wie der Name schon sagt, ein tieferes Absteigen in die Rekursion ab, wenn sie erfüllt wird. Hat ein rekursiver Algorithmus kein Abbruchbedingung kommt dies einer Endlosschleife gleich.

Warum sind rekursive Algorithmen nützlich?

Divide and Conquer

Idee: Kleine Probleme sind einfacher zu bewältigen als große.

→ "Teile und herrsche"

→ Wir teilen ein Problem in mehrere Teilprobleme auf, lösen diese und setzen die Teillösungen zu einer Gesamtlösung zusammen.

Divide and Conquer

Idee: Kleine Probleme sind einfacher zu bewältigen als große.

→ "Teile und herrsche"

→ Wir teilen ein Problem in mehrere Teilprobleme auf, lösen diese und setzen die Teillösungen zu einer Gesamtlösung zusammen.

Divide-and-Conquer-Algorithmus

Ein Algorithmus der mit «Devide and Conquer» ein Problem der Größe **n** löst, geht typischerweise wie folgt vor:

- Zerteile das Problem in **a** Teilprobleme der Größe $\frac{\mathbf{n}}{\mathbf{b}}$ $\mathcal{O}(f(n))$
- löse alle Teilprobleme
- setze Teillösungen zusammen $\mathcal{O}(f(n))$

Divide and Conquer

Idee: Kleine Probleme sind einfacher zu bewältigen als große.

→ "Teile und herrsche"

→ Wir teilen ein Problem in mehrere Teilprobleme auf, lösen diese und setzen die Teillösungen zu einer Gesamtlösung zusammen.

Divide-and-Conquer-Algorithmus

Ein Algorithmus der mit «Devide and Conquer» ein Problem der Größe **n** löst, geht typischerweise wie folgt vor:

- Zerteile das Problem in **a** Teilprobleme der Größe $\frac{\mathbf{n}}{\mathbf{b}}$ $\mathcal{O}(f(n))$
- löse alle Teilprobleme
- setze Teillösungen zusammen $\mathcal{O}(f(n))$

Um die Laufzeit eines solchen Algorithmus zu lösen, muss daher eine Rekurrenz der folgenden Form gelöst werden:

$$T(n) = a \cdot T(\frac{n}{b}) + f(n)$$

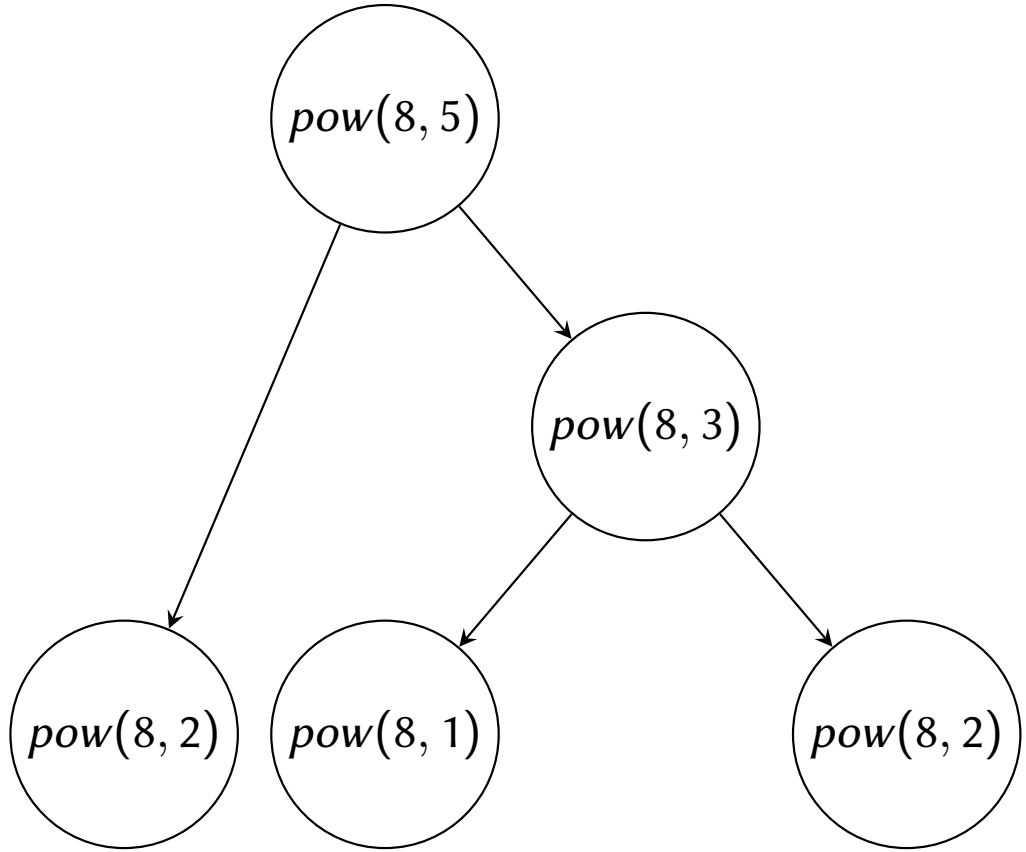
Rekurrenzbaum

Der Rekurrenzbaum beschreibt den Aufruf des rekursiven Algorithmus:

Initale Aufruf: Wurzel

Teilungsschritt: innere Knoten

Abbruchbedingung: Blätter



Roadmap

Mastertheorem

Problemstellung:

Gegeben sei eine rekursiv definierte Funktion T.
Frage: Welche Laufzeit hat T?

Beispiel:

$$T(n) = 8T\left(\frac{n}{2}\right) + 1000n^2$$

$\Rightarrow$ Mastertheorem

Mastertheorem

Def.: Mastertheorem

Seien $a \geq 1$ und $b > 1$ Konstanten, $f : \mathbb{N} \to \mathbb{R}_0^+$ und $T(n)$ eine Laufzeitfunktion der Form

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^c)$$

Dann gilt nach dem **Mastertheorem**:

- **Fall 1:**
  Wenn $a < b^c$ ist, dann ist $T \in \Theta(n^c)$.
- **Fall 2:**
  Wenn $a = b^c$ ist, dann ist $T \in \Theta(n^c \log n)$.
- **Fall 3:**
  Wenn $a > b^c$ ist, dann ist $T \in \Theta(n^{\log_b a})$.

Mastertheorem

Beispiel zum 1. Fall

Sei $T(n) = 2T\left(\frac{n}{2}\right) + n^2$.

- Aus der Formel lässt sich ablesen:
  $a = 2, b = 2, c = 2$
- b^c bestimmen:
  $b^c = 2^2 = 4$
- b^c mit a vergleichen: $2 < 4$?
  Ja.
- Mit dem Mastertheorem folgt:
  $T(n) \in \Theta(n^c) = \Theta(n^2)$

Mastertheorem

Beispiel zum 2. Fall

Sei $T(n) = 2T\left(\frac{n}{2}\right) + 10n$.

- Aus der Formel lässt sich ablesen:
  $a = 2, b = 2, c = 1$
- b^c bestimmen:
  $2^1 = 2$
- b^c mit a vergleichen: $2 = 2$?
  Ja!
- Mit dem Mastertheorem folgt:
  $T(n) \in \Theta(n^1 \log n) = \Theta(n \log n)$

Mastertheorem

Beispiel zum 3. Fall

Sei $T(n) = 8T\left(\frac{n}{2}\right) + 1000n^2$.

- Aus der Formel lässt sich ablesen:
  $a = 8, b = 2, f(n) = 1000n^2 \Rightarrow c = 2$
- b^c bestimmen:
  $b^c = 2^2 = 4$
- b^c mit a vergleichen: $8 > 4$?
  Ja.
- Mit dem Mastertheorem folgt:
  $T(n) \in \Theta(n^{\log_b(a)}) = \Theta(n^{\log_2(8)}) = \Theta(n^3)$

Mastertheorem

Sonderfall Mastertheorem

Wenn $f(n) = cn$ für ein $c \in \mathbb{R}$, dann vereinfacht sich das Mastertheorem zu:

Seien $a \geq 1$ und $b > 1$ Konstanten, $f : \mathbb{N} \to \mathbb{R}_0^+$ und $T(n)$ eine Laufzeitfunktion der Form

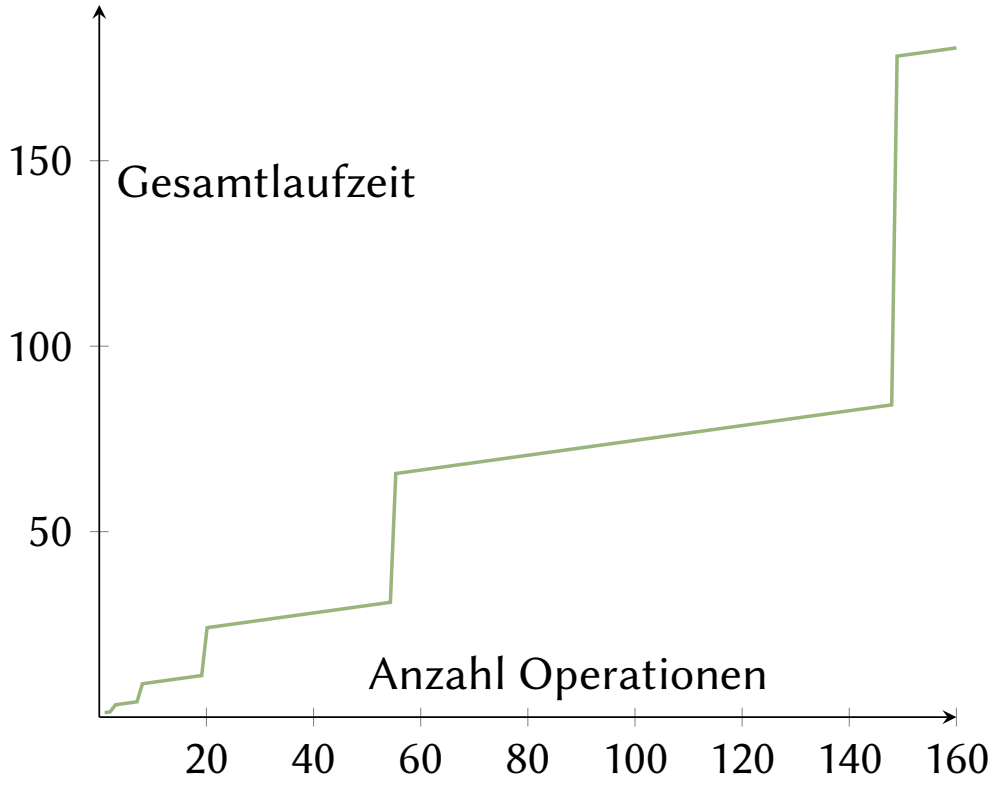
$$T(n) = aT\left(\frac{n}{b}\right) + cn$$

Dann gilt:

$$T(n) \in \begin{cases} \Theta(n) & \text{falls } a < b \\ \Theta(n \log n) & \text{falls } a = b \\ \Theta(n^{\log_b a}) & \text{falls } a > b \end{cases}$$

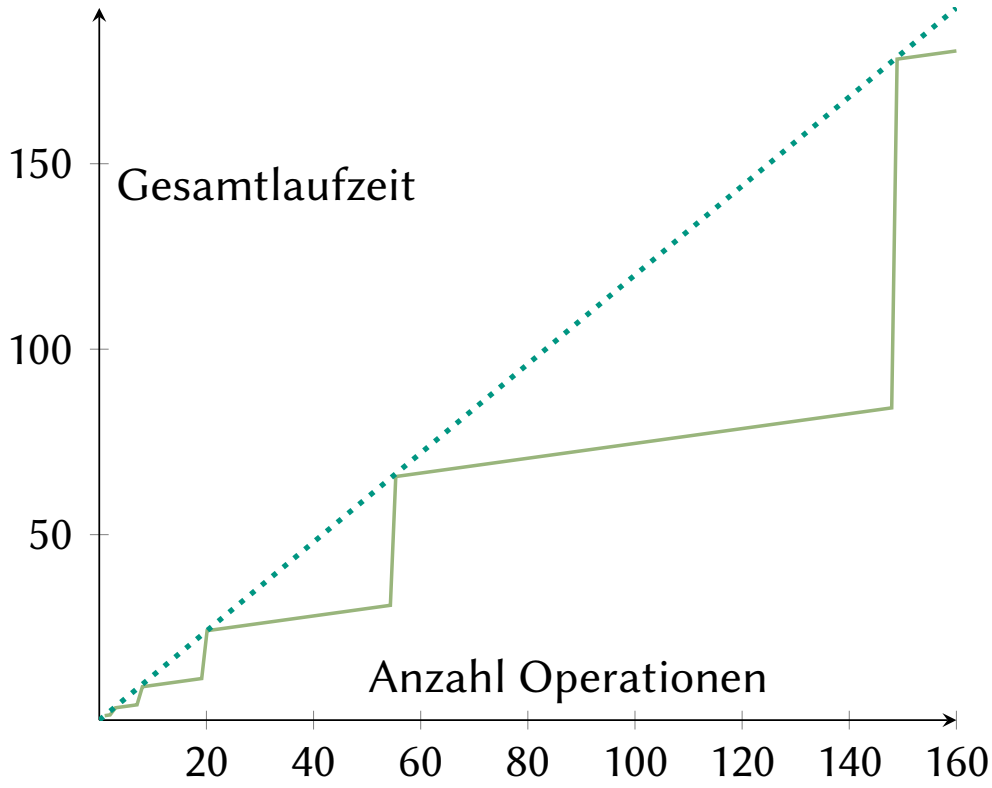
Roadmap

Intuition



- Sprünge, also seltene, hier linear teure Aufrufe

Intuition



- Sprünge, also seltene, hier linear teure Aufrufe
- **In der Summe** immer langsamer, einzelne Operationen konstant

Grundlagen

Amortisierte Analyse ist schlussendlich das Wegargumentieren von schlechten Laufzeiten.

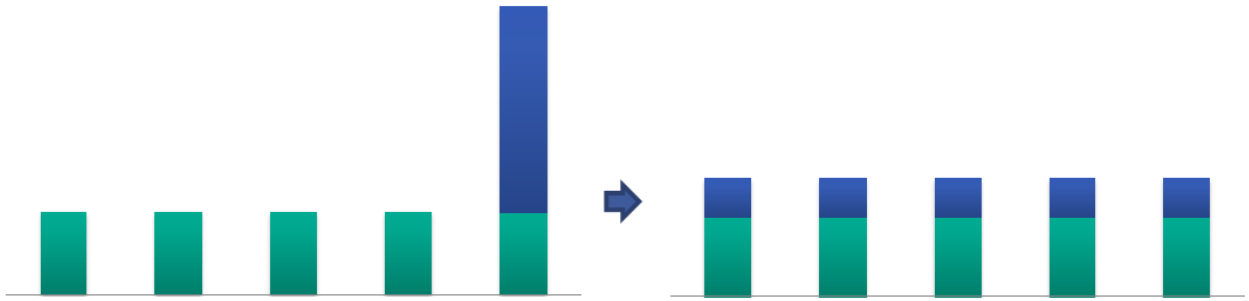
Grundlagen

Amortisierte Analyse ist schlussendlich das Wegargumentieren von schlechten Laufzeiten.

Idee

Statt eine teure Operation isoliert zu betrachten, schauen wir uns den Kontext an, in dem sie aufgerufen wird.

- Abfolge von Operationen statt einer Operation
- Gesamtlaufzeit statt Summe von einzelnen Laufzeiten
- «Umverteilen» / der Differenz von Soll- und Ist-Laufzeit



Grundlagen

Amortisierte Analyse ist schlussendlich das Wegargumentieren von schlechten Laufzeiten.

Idee

Statt eine teure Operation isoliert zu betrachten, schauen wir uns den Kontext an, in dem sie aufgerufen wird.

- Abfolge von Operationen statt einer Operation
- Gesamtlaufzeit statt Summe von einzelnen Laufzeiten
- «Umverteilen» / der Differenz von Soll- und Ist-Laufzeit

Grundlegendes Szenario:

- Wir wissen, wie günstige und teure Operationen voneinander abhängen
- Schlechte Laufzeit tritt selten oder nur nach vielen günstigen Operationen auf
- Wir wissen, wann eine Operation teuer ist

Begleitendes Beispiel

Wir betrachten einen sog. **FlushStack**:

- Operationen PUSH(), POP(), TOP() alle in $\Theta(1)$
- Feste Kapazität `capacity` $\in \mathbb{N}$
- Anzahl Elemente auf dem Stack: $n \in \mathbb{N}$
- Vereinfachende Annahme: kein POP()auf leerem Stack, kein PUSH()auf vollem Stack

```
1: FLUSH
2:     while n > 0 do
3:         POP()
```

Begleitendes Beispiel

Wir betrachten einen sog. **FlushStack**:

- Operationen PUSH(), POP(), TOP() alle in $\Theta(1)$
- Feste Kapazität `capacity` $\in \mathbb{N}$
- Anzahl Elemente auf dem Stack: $n \in \mathbb{N}$
- Vereinfachende Annahme: kein POP()auf leerem Stack, kein PUSH()auf vollem Stack

```
1: FLUSH
2:     while n > 0 do
3:         POP()
```

Ziel

Wir wollen zeigen, dass FLUSH() amortisierte Laufzeit in $\Theta(1)$ hat.

Aggregationsmethode

Idee

Summiere die Kosten von $m \in \mathbb{N}$ Operationen auf und teile das Ergebnis durch m, um die amortisierten Kosten für eine einzelne Operation zu erhalten.

Aggregationsmethode

Idee

Summiere die Kosten von $m \in \mathbb{N}$ Operationen auf und teile das Ergebnis durch m, um die amortisierten Kosten für eine einzelne Operation zu erhalten.

Vorteile:

- Einfach nachzuvollziehen
- Einfache Berechnung

Nachteile:

- Nur schwer anzuwenden, wenn Reihenfolge der Operationen die Kosten verändert
- Nicht immer möglich!

Aggregationsmethode für FLUSH()

Beobachtungen:

- k-mal PUSH() und l-mal POP() $\Rightarrow n = k - l$
- Also: tatsächliche Kosten von FLUSH(): $(k - l) \cdot \Theta(1)$
- Alle günstigen Operationen haben konstante Laufzeit

Wie sieht eine sinnvolle Folge von Operationen für die amortisierte Analyse von FLUSH() aus?

Aggregationsmethode für FLUSH()

Beobachtungen:

- k-mal PUSH() und l-mal POP() $\Rightarrow n = k - l$
- Also: tatsächliche Kosten von FLUSH(): $(k - l) \cdot \Theta(1)$
- Alle günstigen Operationen haben konstante Laufzeit

Wie sieht eine sinnvolle Folge von Operationen für die amortisierte Analyse von FLUSH() aus?

- FLUSH() kommt nur **genau ein Mal** FLUSH() vor
- Die FLUSH()-Operation ist die letzte Operation der Folge

Aggregationsmethode für FLUSH()

Beobachtungen:

- k-mal PUSH() und l-mal POP() $\Rightarrow n = k - l$
- Also: tatsächliche Kosten von FLUSH(): $(k - l) \cdot \Theta(1)$
- Alle günstigen Operationen haben konstante Laufzeit

Analyse:

$$\sum_{i=1}^{m-1} \Theta(1) + \sum_{i=1}^{n} \Theta(1) = \Theta(m-1) + \Theta(n) \overset{n \leq m}{=} \Theta(m)$$

Damit:

- Laufzeit pro Operation ist in $\Theta(m/m) = \Theta(1)$
- Amortisierte Laufzeit von FLUSH() ist in $\Theta(1)$

Charging

Idee

- Fasse «Einheit» von konstanter Laufzeit als Token auf
- Tokens können von teuren auf günstige Operationen umverteilt werden
- Einschränkung: Umverteilung nur auf bereits ausgeführte Operationen möglich

Charging

Idee

- Fasse «Einheit» von konstanter Laufzeit als Token auf
- Tokens können von teuren auf günstige Operationen umverteilt werden
- Einschränkung: Umverteilung nur auf bereits ausgeführte Operationen möglich

Vorteile:

- Tokens sind «greifbar»
- Entspricht Intuition vom Umverteilen
- Detaillierter als Aggregationsmethode

Nachteile:

- Ggf. schwierig, die Anzahl Tokens zu bestimmen, die auf eine Operation entfallen (Abhängigkeit von der Reihenfolge, der Typen der Operationen etc.)

Charging für FLUSH()

Beobachtungen:

- k-mal PUSH() und l-mal POP() $\Rightarrow n = k - l$
- Also: tatsächliche Kosten von FLUSH(): $(k - l) \cdot \Theta(1)$
- Alle günstigen Operationen haben konstante Laufzeit

Wie viele Tokens fallen pro Operation tatsächlich an? Auf wie viele Tokens wollen wir kommen?

Charging für FLUSH()

Beobachtungen:

- k-mal PUSH() und l-mal POP() $\Rightarrow n = k - l$
- Also: tatsächliche Kosten von FLUSH(): $(k - l) \cdot \Theta(1)$
- Alle günstigen Operationen haben konstante Laufzeit

Wie viele Tokens fallen pro Operation tatsächlich an? Auf wie viele Tokens wollen wir kommen?

- Alle günstigen Operationen kosten 1 Token
- FLUSH() kostet n Token
- Ziel: allen Operationen werden konstant viele Tokens zugewiesen

Charging für FLUSH()

Beobachtungen:

- k-mal PUSH() und l-mal POP() $\Rightarrow n = k - l$
- Also: tatsächliche Kosten von FLUSH(): $(k - l) \cdot \Theta(1)$
- Alle günstigen Operationen haben konstante Laufzeit

Analyse:

- vor jedem Aufruf von FLUSH() : mindestens n Aufrufe von 1-Token-Operation
- Damit: Umverteilung von je 1 Token auf die n vorigen Operationen möglich

Charging für FLUSH()

Beobachtungen:

- k-mal PUSH() und l-mal POP() $\Rightarrow n = k - l$
- Also: tatsächliche Kosten von FLUSH(): $(k - l) \cdot \Theta(1)$
- Alle günstigen Operationen haben konstante Laufzeit

Analyse:

- vor jedem Aufruf von FLUSH() : mindestens n Aufrufe von 1-Token-Operation
- Damit: Umverteilung von je 1 Token auf die n vorigen Operationen möglich
- Nach Charging: jeder Operation wurden maximal 2 Tokens zugewiesen
- Damit: Amortisierte Laufzeit aller Operationen ist in $\Theta(1)$

Kontomethode

Idee

- Fasse «Einheit» von konstanter Laufzeit als Token auf
- Tokens können von günstigen Operationen auf ein Konto eingezahlt werden
- teure Operationen können (vorhandene) Tokens vom Konto abheben
- Damit: Umverteilung von bereits ausgeführten auf spätere Operation
- **Wichtig:** Kontostand darf niemals negativ werden

Kontomethode – Vor- und Nachteile

Vorteile: wie bei Charging

- Tokens sind «greifbar»
- Entspricht Intuition vom Umverteilen
- Detaillierter als Aggregationsmethode

Nachteile: Umgekehrt zu Charging

- Ggf. schwierig, die Anzahl Tokens zu bestimmen, die eine bestimmte Operation zahlen muss (Abhängigkeit von der Reihenfolge, der Typen der Operationen etc.)

Kontomethode für FLUSH()

Beobachtungen:

- k-mal PUSH() und l-mal POP() $\Rightarrow n = k - l$
- Also: tatsächliche Kosten von FLUSH(): $(k - l) \cdot \Theta(1)$
- Alle günstigen Operationen haben konstante Laufzeit
- Annahme: «illegale» POP() und PUSH() betrachten wir nicht!

Wie viele Tokens sollte welche Operation einzahlen? Auf wie viele Tokens wollen wir kommen?

Kontomethode für FLUSH()

Beobachtungen:

- k-mal PUSH() und l-mal POP() $\Rightarrow n = k - l$
- Also: tatsächliche Kosten von FLUSH(): $(k - l) \cdot \Theta(1)$
- Alle günstigen Operationen haben konstante Laufzeit
- Annahme: «illegale» POP() und PUSH() betrachten wir nicht!

Wie viele Tokens sollte welche Operation einzahlen? Auf wie viele Tokens wollen wir kommen?

Wie schon bei Charging:

- Alle günstigen Operationen kosten 1 Token
- FLUSH() kostet n Token
- Ziel: allen Operationen werden konstant viele Tokens zugewiesen

Kontomethode für FLUSH()

Beobachtungen:

- k-mal PUSH() und l-mal POP() $\Rightarrow n = k - l$
- Also: tatsächliche Kosten von FLUSH(): $(k - l) \cdot \Theta(1)$
- Alle günstigen Operationen haben konstante Laufzeit
- Annahme: «illegale» POP() und PUSH() betrachten wir nicht!

Analyse:

- **Jede** Operation zahlt ein Token auf das Konto ein
- FLUSH() hebt n Tokens ab
- Da zuvor mindestens n Tokens eingezahlt wurden: Kontostand wird nicht negativ

Kontomethode für FLUSH()

Beobachtungen:

- k-mal PUSH() und l-mal POP() $\Rightarrow n = k - l$
- Also: tatsächliche Kosten von FLUSH(): $(k - l) \cdot \Theta(1)$
- Alle günstigen Operationen haben konstante Laufzeit
- Annahme: «illegale» POP() und PUSH() betrachten wir nicht!

Analyse:

- **Jede** Operation zahlt ein Token auf das Konto ein
- FLUSH() hebt n Tokens ab
- Da zuvor mindestens n Tokens eingezahlt wurden: Kontostand wird nicht negativ
- Amortisierte Kosten der günstigen Operationen: 2 Tokens, also in $\Theta(1)$
- Amortisierte Kosten von FLUSH(): $(n + 1) - n = 1$ Token, also auch in $\Theta(1)$

Potentialmethode

Idee

- Kontostand nicht in Abhängigkeit von Operationen, sondern von (Zustand der) Datenstruktur selbst
- Potentialfunktion $\Phi(A)$ gibt Kontostand der Datenstruktur A an
- Zustand von A vor / nach einer Operation: A_{vor} bzw. A_{nach}
- Amortisierte Kosten = tatsächliche Kosten + $\Phi(A_{nach}) - \Phi(A_{vor})$
- **Wichtig:** $\Phi(A) \geq 0$ muss immer gelten

Potentialmethode – Vor- und Nachteile

Vorteile:

- «globale» Sichtweise auf die gesamte Datenstruktur
- Operationen müssen aber nur für sich allein betrachtet werden
- Einfach nachzurechnen bei korrekter Potentialfunktion

Nachteile:

- Finden der korrekten Potentialfunktion kann knifflig sein
- Evtl. weniger intuitiv als andere Methoden

Potentialmethode für FLUSH()

Beobachtungen:

- k-mal PUSH() und l-mal POP() $\Rightarrow n = k - l$
- Also: tatsächliche Kosten von FLUSH(): $(k - l) \cdot \Theta(1)$
- Alle günstigen Operationen haben konstante Laufzeit

Woran sollte der Kontostand (also wie teuer die nächste teure Operation ist) des Stacks gemessen werden? Wie sieht das vor / nach einer Operation aus?

Potentialmethode für FLUSH()

Beobachtungen:

- k-mal PUSH() und l-mal POP() $\Rightarrow n = k - l$
- Also: tatsächliche Kosten von FLUSH(): $(k - l) \cdot \Theta(1)$
- Alle günstigen Operationen haben konstante Laufzeit

Woran sollte der Kontostand (also wie teuer die nächste teure Operation ist) des Stacks gemessen werden? Wie sieht das vor / nach einer Operation aus?

$\Phi(A)$ sollte die tatsächlichen Kosten von FLUSH() «ausgleichen» , also sollte $\Phi(A_{vor}) \in \Theta(n)$ und $\Phi(A_{nach}) \in \Theta(1)$ gelten.

Potentialmethode für FLUSH()

Beobachtungen:

- k-mal PUSH() und l-mal POP() $\Rightarrow n = k - l$
- Also: tatsächliche Kosten von FLUSH(): $(k - l) \cdot \Theta(1)$
- Alle günstigen Operationen haben konstante Laufzeit

Analyse:

- Hypothese: $\Phi(A) = A.size() = n$ erfüllt die Bedingungen
- PUSH(): amortisierte Kosten = $1 + (n + 1) - n = 2 \in \Theta(1)$
- POP(): amortisierte Kosten = $1 + (n - 1) - n = 0 \in \mathcal{O}(1)$
- FLUSH(): amortisierte Kosten = $n + 0 - n = 0 \in \mathcal{O}(1)$

Potentialmethode für FLUSH()

Beobachtungen:

- k-mal PUSH() und l-mal POP() $\Rightarrow n = k - l$
- Also: tatsächliche Kosten von FLUSH(): $(k - l) \cdot \Theta(1)$
- Alle günstigen Operationen haben konstante Laufzeit

Karlsruhe Institute of Technology

Fragen?

Credits

An der Erstellung des Foliensatzes haben mitgewirkt:

Daniel Schild
Erik Eitel
Ferdinand Heinen
Henriette Färeber
Kilian Wurm
Lukas Freudenmann
Max Ackermann
Moritz Laupichler
Philip Scherer
Stephan Bohr
Tobias Knorr