Karlsruhe Institute of Technology

Karlsruher Institut für Technologie

1. Zusatztutorium

$\mathcal{O}$-Notation, Rekurrenzen, Amortisierte Analyse

Algorithmen I, 1. Zusatztutorium

Henriette Färber | July 31, 2023

Themen für heute

Wo sind wir?

Was ist überhaupt ein Beweis?

Wir haben: Hypothesen

- Wissen über Datenstrukturen, Arithmetik, Graphen, ...
- Szenario (Bsp.: Wir betrachten einen Baum T)
- Garantien (Bsp.: Wir können in $\Theta(1)$ überprüfen, ob zwei Knoten benachbart sind)
- ...

Wir wollen: Eine Behauptung zeigen

- Eine Datenstruktur hat eine schöne Eigenschaft
- Ein Algorithmus funktioniert (nicht)
- Ein Algorithmus hat eine schöne Laufzeit
- ...

Dazu: Neue Erkenntnisse anhand von logischen Umformungen auf bestehendem Wissen

Korrektheitsbeweise I

Ziel: Zeige, dass ein Algorithmus das leistet, was wir erwarten

Plan:

- Betrachte eine beliebige (gültige) Eingabe
- Leite her, dass der Algorithmus eine korrekte Lösung berechnet

Korrektheitsbeweise I

Ziel: Zeige, dass ein Algorithmus das leistet, was wir erwarten

Plan:

- Betrachte eine beliebige (gültige) Eingabe
- Leite her, dass der Algorithmus eine korrekte Lösung berechnet

Beispiel: Binäre Suche

```
1:  BINSEARCH(haystack : [ℝ; n], needle : ℝ, l, r ∈ ℕ): ℕ
2:      if l = r then
3:          if haystack[l] = needle then
4:              return l
5:          return ⊥
6:      m: ℕ = l + ⌊(r − l)/2⌋
7:      if needle ≤ haystack[m] then
8:          return BINSEARCH(haystack, needle, l, m)
9:      else
10:         return BINSEARCH(haystack, needle, m, r)
```

Korrektheitsbeweise I

Ziel: Zeige, dass ein Algorithmus das leistet, was wir erwarten

Beispiel: Binäre Suche

```
1:  BINSEARCH(haystack : [ℝ; n], needle : ℝ, l, r ∈ ℕ): ℕ
2:      if l = r then
3:          if haystack[l] = needle then
4:              return l
5:          return ⊥
6:      m: ℕ = l + ⌊(r − l)/2⌋
7:      if needle ≤ haystack[m] then
8:          return BINSEARCH(haystack, needle, l, m)
9:      else
10:         return BINSEARCH(haystack, needle, m, r)
```

Was müssen wir zeigen?
Welche Fälle müssen wir beachten?

Korrektheitsbeweise I

Ziel: Zeige, dass ein Algorithmus das leistet, was wir erwarten

Was müssen wir zeigen?
Welche Fälle müssen wir beachten?

1. `needle` in `haystack` enthalten
   $\Rightarrow$ BINSEARCH liefert (kleinsten) Index von `needle`
2. sonst: BINSEARCH liefert $\bot$

Beispiel: Binäre Suche

```
1:  BINSEARCH(haystack : [ℝ; n], needle : ℝ, l, r ∈ ℕ): ℕ
2:      if l = r then
3:          if haystack[l] = needle then
4:              return l
5:          return ⊥
6:      m: ℕ = l + ⌊(r − l)/2⌋
7:      if needle ≤ haystack[m] then
8:          return BINSEARCH(haystack, needle, l, m)
9:      else
10:         return BINSEARCH(haystack, needle, m, r)
```

Korrektheitsbeweise II

Wir wollen zeigen:

1. `needle` in `haystack` enthalten
   $\Rightarrow$ BINSEARCH liefert (kleinsten) Index von `needle`
2. sonst: BINSEARCH liefert $\perp$

Dabei nutzen wir unser Wissen über die Eingabe aus:

- `haystack` ist sortiert
- initial wird BINSEARCH mit $l = 0, r = n - 1$ aufgerufen

Korrektheitsbeweise II

Wir wollen zeigen:

1. `needle` in `haystack` enthalten
   $\Rightarrow$ BINSEARCH liefert (kleinsten) Index von `needle`
2. sonst: BINSEARCH liefert $\perp$

Dabei nutzen wir unser Wissen über die Eingabe aus:

- `haystack` ist sortiert
- initial wird BINSEARCH mit $l = 0, r = n - 1$ aufgerufen

Fall 1: $\exists i \in \{0, \ldots, n-1\}$: `needle` $=$ `haystack`$[i]$

- Initial: `haystack`$[l] \leq$ `needle` $\leq$ `haystack`$[r]$
- Also: wenn $l = r$, dann `haystack`$[l] =$ `needle`
- Sonst: Wenn `needle` $\leq$ `haystack`$[m]$, dann `needle` $\leq$ `haystack`$[k]$ für alle $k \in \{m, \ldots, n\}$
  $\Rightarrow i \in \{l, \ldots, r\}$

Korrektheitsbeweise II

Wir wollen zeigen:

1. `needle` in `haystack` enthalten
   $\Rightarrow$ BINSEARCH liefert (kleinsten) Index von `needle`
2. sonst: BINSEARCH liefert $\perp$

Dabei nutzen wir unser Wissen über die Eingabe aus:

- `haystack` ist sortiert
- initial wird BINSEARCH mit $l = 0, r = n - 1$ aufgerufen

Fall 2: `needle` ist nicht in `haystack`

- trotzdem ein Vergleich mit `needle` pro Aufruf, untersuchter Bereich verkleinert sich
- wenn $l = r$, dann muss `haystack`$[l] \neq$ `needle` sein

Vollständige Induktion I

Ziel: Zeige eine Aussage, die für alle natürlichen Zahlen gelten soll

Plan: Zeige Aussage **nicht** explizit für jedes Element, sondern

- Zeige Aussage für eine erste Zahl
- Führe die Aussage für eine beliebige Zahl auf die Aussage für ihren Vorgänger zurück

Vollständige Induktion I

Ziel: Zeige eine Aussage, die für alle natürlichen Zahlen gelten soll

Plan: Zeige Aussage **nicht** explizit für jedes Element, sondern

- Zeige Aussage für eine erste Zahl
- Führe die Aussage für eine beliebige Zahl auf die Aussage für ihren Vorgänger zurück

Beispiel: Gaußsche Summenformel

$$\forall n \in \mathbb{N} : \sum_{i=0}^{n} k = \frac{n \cdot (n+1)}{2}$$

Vollständige Induktion I

Beispiel: Gaußsche Summenformel

$$\forall n \in \mathbb{N} : \sum_{i=0}^{n} k = \frac{n \cdot (n+1)}{2}$$

Was müssen wir zeigen? Wo fangen wir an?

Vollständige Induktion I

Beispiel: Gaußsche Summenformel

$$\forall n \in \mathbb{N} : \sum_{i=0}^{n} k = \frac{n \cdot (n+1)}{2}$$

Was müssen wir zeigen? Wo fangen wir an?

- Wähle einen **Induktionsanfang** (i.d.R. 0) und zeige die Aussage explizit
- Für jede andere der natürlichen Zahlen zeige:
  - Unter der Annahme, dass die Aussage für ihren Vorgänger gilt (**Induktionsvoraussetzung**)...
  - ...gilt die Aussage auch für diese Zahl (**Induktionsschritt**)

Vollständige Induktion II

IA: $n = 0$ $\quad \sum_{i=0}^{0} k = 0 = (0 \cdot 1)/2 \quad \checkmark$

Vollständige Induktion II

IA: $n = 0 \qquad \sum_{i=0}^{0} k = 0 = (0 \cdot 1)/2 \qquad \checkmark$

IV: Es gelte die Aussage für ein beliebiges aber festes $n \in \mathbb{N}$

Vollständige Induktion II

IA: $n = 0 \qquad \sum_{i=0}^{0} k = 0 = (0 \cdot 1)/2 \qquad \checkmark$

IV: Es gelte die Aussage für ein beliebiges aber festes $n \in \mathbb{N}$

IS: $n \rightsquigarrow n+1$

$$\begin{aligned}\sum_{i=0}^{n+1} k &= \sum_{i=0}^{n} k + (n+1) \\ &\overset{\text{IV}}{=} \frac{n \cdot (n+1)}{2} + (n+1) \\ &= \frac{n \cdot (n+1) + 2 \cdot (n+1)}{2} \\ &= \frac{(n+1) \cdot (n+2)}{2}\end{aligned}$$

□

Vollständige Induktion III

Beachtet:

- Im Induktionsschritt muss die Induktionsvoraussetzung auch benutzt werden!
- Verschiedene Variationen:
  - Mehrere Induktionsanfänge, Bezug auf mehr als einen Vorgänger
  - Anderer Induktionsanfang als 0
  - Erweiterung auf $\mathbb{Z}$
- Außerdem: strukturelle Induktion
  - Nicht auf $\mathbb{N}$, sondern auf strukturierten Daten
  - Für uns insbesondere relevant: Graphen

Widerspruchsbeweise

Ziel: Widerlege eine Aussage (bzw. ihr Negat)

Plan:

- Nimm an, die Aussage gilt nicht
- Zeige, dass dadurch Widersprüche zu anderen Aussagen entstehen, von denen wir wissen, dass sie korrekt sind

Widerspruchsbeweise

Ziel: Widerlege eine Aussage (bzw. ihr Negat)

Plan:

- Nimm an, die Aussage gilt nicht
- Zeige, dass dadurch Widersprüche zu anderen Aussagen entstehen, von denen wir wissen, dass sie korrekt sind

Beispiel: Kürzeste Pfade
Sei $G = (V, E)$ ein Graph und sei $p = (v_0, v_1, \ldots, v_{i-1}, v_i)$ ein kürzester Pfad zwischen v_0 und v_i in G. Dann gilt: Für alle $j, k \in \{0, \ldots, i\}$ ist ein kürzester Pfad zwischen v_j und v_k ein Teilpfad von p.

Widerspruchsbeweise

Beispiel: Kürzeste Pfade
Sei $G = (V, E)$ ein Graph und sei $p = (v_0, v_1, \ldots, v_{i-1}, v_i)$ ein kürzester Pfad zwischen v_0 und v_i in G. Dann gilt: Für alle $j, k \in \{0, \ldots, i\}$ ist ein kürzester Pfad zwischen v_j und v_k ein Teilpfad von p.

Was müssen wir zeigen? Was wollen wir hier widerlegen?

Widerspruchsbeweise

Beispiel: Kürzeste Pfade
Sei $G = (V, E)$ ein Graph und sei $p = (v_0, v_1, \ldots, v_{i-1}, v_i)$ ein kürzester Pfad zwischen v_0 und v_i in G. Dann gilt: Für alle $j, k \in \{0, \ldots, i\}$ ist ein kürzester Pfad zwischen v_j und v_k ein Teilpfad von p.

Was müssen wir zeigen? Was wollen wir hier widerlegen?

- Ziel: Aussage beweisen
- Also: Negat widerlegen
- Zeige: Angenommen, die Aussage gilt nicht, dann entsteht ein Widerspruch

Widerspruchsbeweise

Beispiel: Kürzeste Pfade
Sei $G = (V, E)$ ein Graph und sei $p = (v_0, v_1, \ldots, v_{i-1}, v_i)$ ein kürzester Pfad zwischen v_0 und v_i in G. Dann gilt: Für alle $j, k \in \{0, \ldots, i\}$ ist ein kürzester Pfad zwischen v_j und v_k ein Teilpfad von p.

- Annahme: es gibt $j, k \in \{0, \ldots, i\}$ so, dass der kürzeste Pfad q zwischen v_j und v_k nicht Teilpfad von p ist
- O.B.d.A.: j < k, also $q = (v_j, u_1, u_2, \ldots, u_l, v_k)$
- Dann: $p' = (v_0, \ldots, v_j, u_1, \ldots, u_l, v_k, \ldots, v_i)$ ist kürzer als p
- Widerspruch zu: p ist kürzester Pfad

Gegenbeispiele

Ziel: Widerlege eine Aussage, die auf allen Elementen einer Menge M gelten sollte

Plan: Zeige für ein bestimmtes Element, dass die Aussage nicht hält.

Gegenbeispiele

Ziel: Widerlege eine Aussage, die auf allen Elementen einer Menge M gelten sollte

Plan: Zeige für ein bestimmtes Element, dass die Aussage nicht hält.

Beispiel:
“Alle Graphen mit n Knoten und $n - 1$ Kanten sind kreisfrei.”

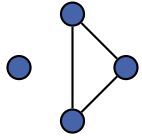
Gegenbeispiele

Ziel: Widerlege eine Aussage, die auf allen Elementen einer Menge M gelten sollte

Plan: Zeige für ein bestimmtes Element, dass die Aussage nicht hält.

Beispiel:
“Alle Graphen mit n Knoten und $n - 1$ Kanten sind kreisfrei.”

Diese Aussage gilt nicht, der folgende Graph ist ein Gegenbeispiel:



Gegenbeispiele

Ziel: Widerlege eine Aussage, die auf allen Elementen einer Menge M gelten sollte

Plan: Zeige für ein bestimmtes Element, dass die Aussage nicht hält.

Beispiel:
“Alle Graphen mit n Knoten und $n - 1$ Kanten sind kreisfrei.”

Gegenbeispiele sind toll!

Sie sind eine sehr kompakte und stichhaltige Art, eine Aussage zu widerlegen.
Wenn ihr sie nutzen könnt – tut es!

Wo sind wir?

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

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

Der böse Pseudo-Pseudocode

Beispiel 1: Iterieren über ein Array

A: $[\mathbb{R};\ n]$
`counter`: $\mathbb{N} = 0$
...
for $i \in \{0, \dots, n-1\}$ **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: $[\mathbb{R};\ n]$
`counter`: $\mathbb{N} = 0$
...
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 iterieren wir mit einer for-Schleife über das Array 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`$\colon \mathbb{N} = 0$
...
for $i \in \{0, \ldots, n-1\}$ **do**
  if $A[i] = 42$ **then**
    `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 $i < n-1 \wedge A[i] < \texttt{pivot}$ **do**
    $i = i + 1$
  while $i > 0 \wedge A[j] \geq \texttt{pivot}$ **do**
    $j = j - 1$
  if $i < j$ **then**
    SWAP$(A[i], A[j])$
SWAP$(A[i], A[n-1])$

Der böse Pseudo-Pseudocode

Beispiel 2: Swapping

$A\colon [\mathbb{R};\ n]$
`pivot`$\colon \mathbb{R} = A[n-1]$
$i, j\colon \mathbb{N} = 0, n-1$
while $i < j$ **do**
  while $i < n-1 \wedge A[i] <$ `pivot` **do**
    $i = i+1$
  while $i > 0 \wedge A[j] \geq$ `pivot` **do**
    $j = j-1$
  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

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

Wir wählen $A[n-1]$ [illegible] Pivot[illegible]ment. Nun legen wir zwei Variablen i = [illegible] j [illegible]-2 an und treten in eine while-Schleife ein, [illegible] solange läuft, wie $i < j$ ist. In jeder Iteration [illegible] solange erhöht, bis $A[i] \geq A[n-1]$ und [illegible] solange dekrementiert, bis $A[j] < A[n-$ [illegible]ilt da[illegible] $< j$, tauschen wir $A[i]$ und $A[j]$. N[illegible] Austritt a[illegible]s der while-Schleife tauschen wir [illegible] 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 = i + 1
    while i > 0 ∧ A[j] ≥ pivot do
        j = j − 1
    if i < j then
        SWAP(A[i], A[j])
SWAP(A[i], A[n − 1])
```

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

Wo sind wir?

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 $f\colon \mathbb{N}_0 \to \mathbb{R}_0^+$. 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."

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

"g wächst asymptotisch mindestens so schnell wie f."

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

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

$\mathcal{O}$-Notation

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

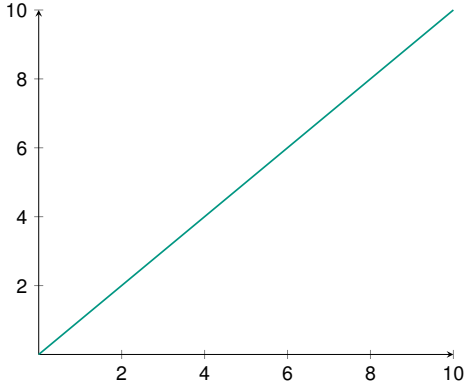
$$\begin{aligned} o(f) &= \{g : \mathbb{N}_0 \to \mathbb{R}_0^+ \mid \forall c \in \mathbb{R}_+ : \exists n_0 \in \mathbb{N}_0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\} \\ &= \mathcal{O}(f) \setminus \Theta(f) \end{aligned}$$

"g wächst asymptotisch langsamer als f."

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

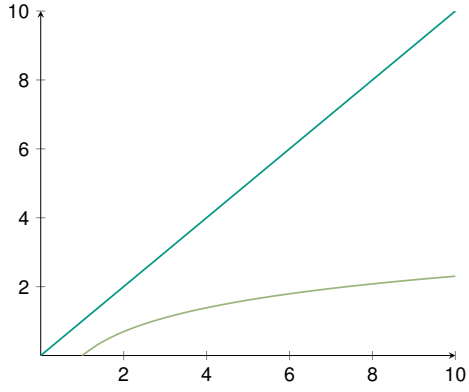
"g wächst asymptotisch schneller als f."

Elemetare Funktionen



“Nice.”

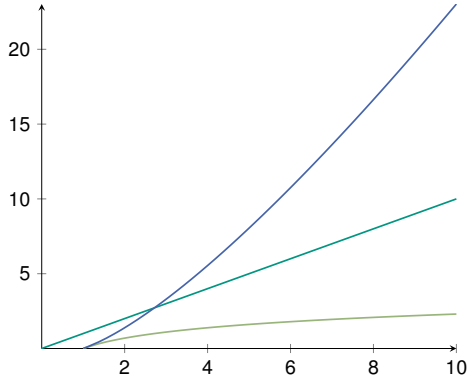
Elemetare Funktionen



“Very nice.”

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

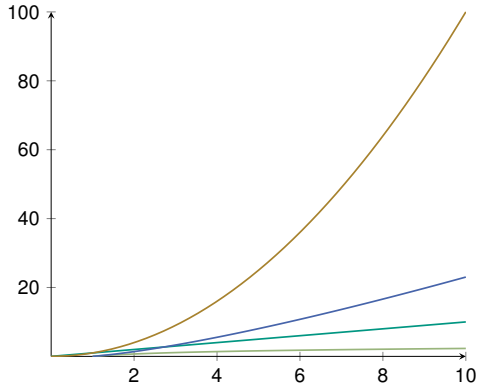
Elemetare Funktionen



“Nice-ish.”

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

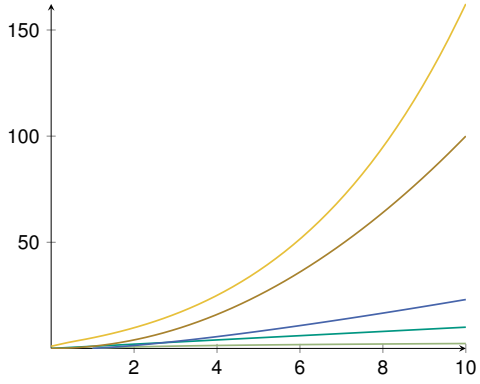
Elemetare Funktionen



“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: $n^{\log(n)}$

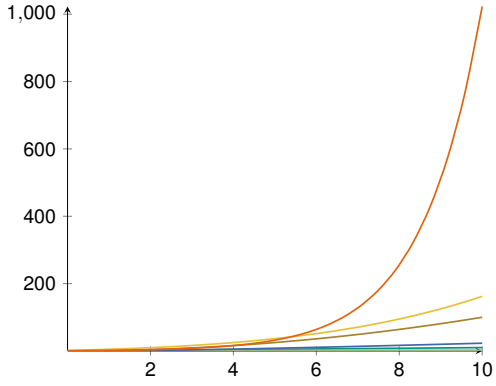
Elemetare Funktionen



“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: $n^{\log(n)}, 2^{\sqrt{n}}$

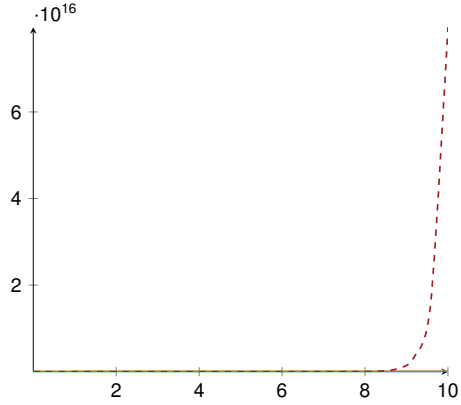
Elemetare Funktionen



“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: $2^n, 3^n, 4^n$

Elemetare Funktionen



- 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: $n!, 2^{n^2}$

“Oh god, oh no.”

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

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

Alles klar soweit?

Wo sind wir?

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.

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.

Warum sind rekursive Algorithmen nützlich?

Divide and Conquer

Idee: Kleine Probleme sind einfacher zu bewältigen als große.

Divide-and-Conquer-Algorithmus

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

Divide and Conquer

Idee: Kleine Probleme sind einfacher zu bewältigen als große.

Divide-and-Conquer-Algorithmus

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

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

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

Rekurrenzbäume

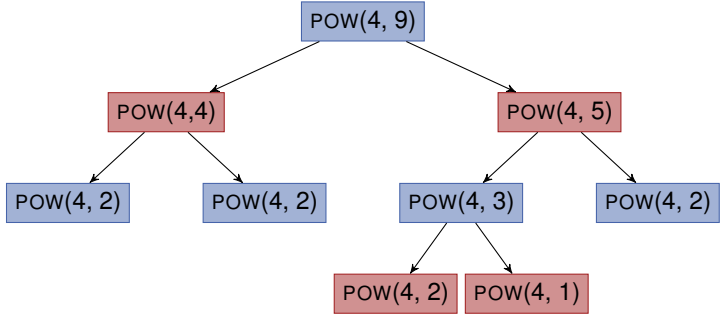
Wir können die Aufrufe / Instanzen eines rekursiven Algorithmus (und ihre Zusammenhänge) in einem Rekurrenzbaum darstellen:

```
1: POW(x, y : ℕ): ℕ
2:     if y = 1 then
3:         return x
4:     if y = 2 then
5:         return x · x
6:     y_1 : ℕ = ⌊y/2⌋
7:     y_2 : ℕ = ⌈y/2⌉
8:     return POW(x, y_1) · POW(x, y_2)
```

Wie sieht der Rekurrenzbaum für POW(4, 9) aus?

Rekurrenzbäume

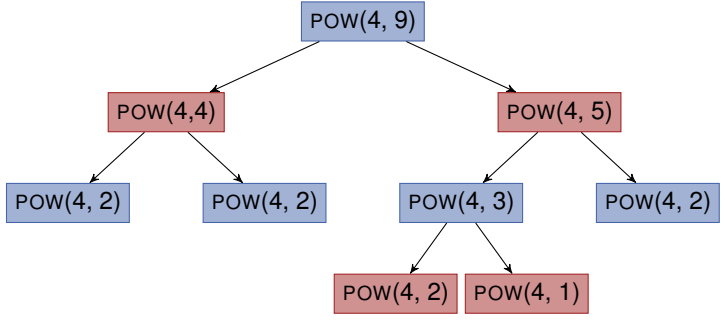
Wir können die Aufrufe / Instanzen eines rekursiven Algorithmus (und ihre Zusammenhänge) in einem Rekurrenzbaum darstellen:



Rekurrenzbäume

Wir können die Aufrufe / Instanzen eines rekursiven Algorithmus (und ihre Zusammenhänge) in einem Rekurrenzbaum darstellen:

- Wurzel: Initialer Aufruf
- Innere Knoten: Zerteilung
- Blätter: Abbruchbedingung erfüllt



Rekurrenzen aufstellen I

Wie können wir nun die tatsächliche Laufzeit eines rekursiven Algorithmus bestimmen?

Rekurrenzen aufstellen I

Wie können wir nun die tatsächliche Laufzeit eines rekursiven Algorithmus bestimmen?

Idee: Wir analysieren die Struktur des zugehörigen Rekurrenzbaums. Dazu stellen wir uns die folgenden Fragen:

1. Wie viele Lagen hat der Baum?
2. Wie viele Knoten sind auf der i-ten Lage?
3. Wie groß ist das n auf der i-ten Lage?
4. Wie viel Zeit kostet ein Knoten in der i-ten Lage?

Dann können wir die Kosten aller Lagen im Rekurrenzbaum bestimmen und aufsummieren.

Rekurrenzen aufstellen II

Beispiel: Wir analysieren POW

```
1: POW(x, y : ℕ): ℕ
2:     if y = 1 then
3:         return x
4:     if y = 2 then
5:         return x · x
6:     y_1 : ℕ = ⌊y/2⌋
7:     y_2 : ℕ = ⌈y/2⌉
8:     return POW(x, y_1) · POW(x, y_2)
```

Wie viele Lagen hat der Rekurrenzbaum im Allgemeinen?

Rekurrenzen aufstellen II

Beispiel: Wir analysieren POW

```
1: POW(x, y : ℕ): ℕ
2:     if y = 1 then
3:         return x
4:     if y = 2 then
5:         return x · x
6:     y_1 : ℕ = ⌊y/2⌋
7:     y_2 : ℕ = ⌈y/2⌉
8:     return POW(x, y_1) · POW(x, y_2)
```

Wie viele Lagen hat der Rekurrenzbaum im Allgemeinen?

Die Problemgröße n wird in jeder Lage um den Faktor $b = 2$ verkleinert $\Rightarrow \log_2 n$

Rekurrenzen aufstellen II

Beispiel: Wir analysieren POW

```
1: POW(x, y : ℕ): ℕ
2:     if y = 1 then
3:         return x
4:     if y = 2 then
5:         return x · x
6:     y_1 : ℕ = ⌊y/2⌋
7:     y_2 : ℕ = ⌈y/2⌉
8:     return POW(x, y_1) · POW(x, y_2)
```

Wie viele Knoten sind auf der i-ten Lage?

Rekurrenzen aufstellen II

Beispiel: Wir analysieren POW

```
1: POW(x, y : ℕ): ℕ
2:     if y = 1 then
3:         return x
4:     if y = 2 then
5:         return x · x
6:     y_1 : ℕ = ⌊y/2⌋
7:     y_2 : ℕ = ⌈y/2⌉
8:     return POW(x, y_1) · POW(x, y_2)
```

Wie viele Knoten sind auf der i-ten Lage?

Jeder innere Knoten hat $a = 2$ Kinder $\Rightarrow$ (bis zu) 2^i

Rekurrenzen aufstellen II

Beispiel: Wir analysieren POW

```
1: POW(x, y : ℕ): ℕ
2:     if y = 1 then
3:         return x
4:     if y = 2 then
5:         return x · x
6:     y_1 : ℕ = ⌊y/2⌋
7:     y_2 : ℕ = ⌈y/2⌉
8:     return POW(x, y_1) · POW(x, y_2)
```

Wie groß ist das n auf der i-ten Lage?

Rekurrenzen aufstellen II

Beispiel: Wir analysieren POW

```
1: POW(x, y : ℕ): ℕ
2:     if y = 1 then
3:         return x
4:     if y = 2 then
5:         return x · x
6:     y_1 : ℕ = ⌊y/2⌋
7:     y_2 : ℕ = ⌈y/2⌉
8:     return POW(x, y_1) · POW(x, y_2)
```

Wie groß ist das n auf der i-ten Lage?

Die Problemgröße n wird in jeder Lage um den Faktor $b = 2$ verkleinert $\Rightarrow \frac{n}{2^i}$

Rekurrenzen aufstellen II

Beispiel: Wir analysieren POW

```
1: POW(x, y : ℕ): ℕ
2:     if y = 1 then
3:         return x
4:     if y = 2 then
5:         return x · x
6:     y_1 : ℕ = ⌊y/2⌋
7:     y_2 : ℕ = ⌈y/2⌉
8:     return POW(x, y_1) · POW(x, y_2)
```

Wie viel Zeit kostet ein Knoten in der i-ten Lage?

Rekurrenzen aufstellen II

Beispiel: Wir analysieren POW

```
1: POW(x, y : ℕ): ℕ
2:     if y = 1 then
3:         return x
4:     if y = 2 then
5:         return x · x
6:     y_1 : ℕ = ⌊y/2⌋
7:     y_2 : ℕ = ⌈y/2⌉
8:     return POW(x, y_1) · POW(x, y_2)
```

Wie viel Zeit kostet ein Knoten in der i-ten Lage?

In jedem Aufruf wird Arbeit mit konstantem Zeitaufwand verrichtet $\Rightarrow f(n) \in \Theta(1)$

Rekurrenzen lösen

Beispiel: Wir analysieren POW

Insgesamt haben wir:

- $\log_2 n$ Lagen
- 2^i Knoten mit Instanzgröße $\frac{n}{2^i}$ auf Lage i
- Konstante Kosten pro Knoten

Rekurrenzen lösen

Beispiel: Wir analysieren POW

Insgesamt haben wir:

- $\log_2 n$ Lagen
- 2^i Knoten mit Instanzgröße $\frac{n}{2^i}$ auf Lage i
- Konstante Kosten pro Knoten

$$\sum_{i=0}^{\text{Anzahl Lagen}} \text{Anzahl Knoten auf Lage } i \cdot \text{ Kosten für Knoten auf Lage } i = \sum_{i=0}^{\log_2 n} 2^i \cdot 1$$
$$= 2^{\log_2 n + 1} - 1$$
$$= 2n - 1$$
$$\in \Theta(n)$$

Alles klar soweit?

```
1:  BINSEARCH(haystack : [ℝ; n], needle : ℝ, l, r ∈ ℕ): ℕ
2:      if l = r then
3:          if haystack[l] = needle then
4:              return l
5:          return ⊥
6:      m: ℕ = l + ⌊(r − l)/2⌋
7:      if needle ≤ haystack[m] then
8:          return BINSEARCH(haystack, needle, l, m)
9:      else
10:         return BINSEARCH(haystack, needle, m, r)
```

Wo sind wir?

Amortisierte Analyse ist schlussendlich das Wegargumentieren von schlechten Laufzeiten.

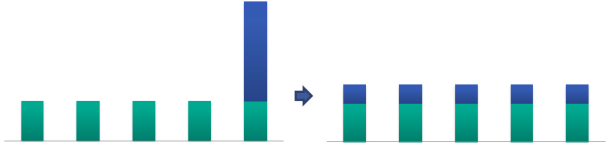
Grundlagen

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

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

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 zahl 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 zahl 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

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

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 = $1 + (n - 1) - n = 0 \in \Theta(1)$
- FLUSH(): amortisierte Kosten = $n + 0 - n = 0 \in \Theta(1)$

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

Analyse:

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

Das war es für heute!

Gibt es Fragen?