

# Tutorium Grundlagen: Algorithmen und Datenstrukturen

## Übungsblatt Woche 5

# Aufgabe 5.1

## Laufzeitanalyse: Mergesort

# Aufgabe 5.1

In dieser Aufgabe machen wir eine Laufzeitanalyse von Mergesort auf drei verschiedene Weisen.

- a) Zeigen Sie argumentativ, wie sich die Laufzeit von Mergesort verhält, indem Sie folgende Fragen beantworten: Wie viele Rekursionsebenen gibt es im Allgemeinen bei Mergesort (wobei wir den initialen Aufruf von Mergesort nicht als eigene Rekursionsebene zählen)? In welcher Größenordnung liegt asymptotisch der Aufwand für jede Rekursionsebene? Was ist damit der asymptotische Aufwand für den gesamten Algorithmus?

In der Vorlesung wurde die Laufzeit rekursiv formuliert und das Mastertheorem verwendet, um zu zeigen, dass die Laufzeit von Mergesort in  $O(n \log n)$  liegt.

Die rekursive Formulierung der Laufzeit lautet

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \Theta(n)$$

$$T(1) = \Theta(1)$$

Zeigen Sie ohne Verwendung des Master-Theorems, dass die Worst-Case-Laufzeit von Mergesort angewandt auf Zahlenfolgen, deren Längen Zweierpotenzen sind, in  $\mathcal{O}(n \log n)$  liegt.

- b) Verwenden Sie dazu die Methode des iterativen Einsetzens.  
c) Verwenden Sie dazu vollständige Induktion (Abschätzung durch z.B.  $T(n) \leq dn \log(dn)$ ).

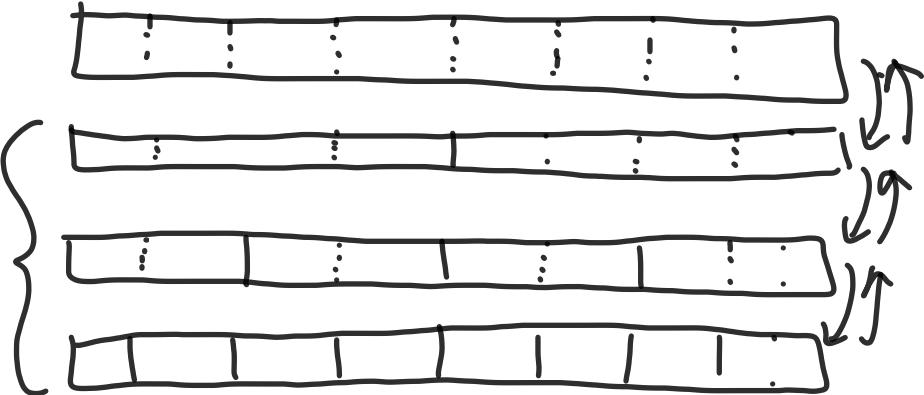
# Aufgabe 5.1 (a)

Bsp:  $n=8 = \underbrace{2}_{}^3$

Init:



3 Ebenen



- Bis nur noch Teilarrays mit Länge 1 übrig sind,  
Sind  $\lceil \log_2(n) \rceil$  Teilungen nötig (aufrunden für nicht-Zweierpotenzen)  
 $\Rightarrow \lceil \log_2(n) \rceil$  Rekursionsebenen
- In jeder Ebene benötigt das Erstellen der neuen Teilarrays, sowie das spätere Verschmelzen einen Aufwand von  $c \cdot n$  (Abhängigkeit von  $n$ )  
 $\Rightarrow$  Gesamtaufwand:  $O(\lceil \log_2(n) \rceil \cdot c \cdot n) = O(n \cdot \log_2(n))$

# Aufgabe 5.1 (b)

- $$\begin{aligned} T(x) &= T\left(\frac{x}{2}\right) + T\left(\frac{x}{2}\right) + \Theta(x) \\ &= 2 \cdot T\left(\frac{x}{2}\right) + c \cdot x \end{aligned}$$

- Variable durch  $x$  ersetzt zur besseren Verständlichkeit
- Nur  $2^i \ i \in \mathbb{N}$  als Eingaben  
⇒ kein Runden nötig
- $c$  konstant (unbekannt)

⇒ Jetzt mit Eingabegröße  $n$

Ersetze  $T(n)$  mit Definition

$$T(n) \hookrightarrow 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n$$

Ersetze  $T\left(\frac{n}{2}\right)$  durch Definition  
Vereinfachen

$$\hookrightarrow 2 \cdot \left(2 \cdot T\left(\frac{n}{4}\right) + c \cdot \frac{n}{2}\right) + c \cdot n$$

Ersetze  $T\left(\frac{n}{4}\right)$  durch Definition

$$\hookrightarrow 4 \cdot T\left(\frac{n}{4}\right) + 2 \cdot c \cdot n$$

Vereinfachen

$$\hookrightarrow 4 \cdot \left(2 \cdot T\left(\frac{n}{8}\right) + c \cdot \frac{n}{4}\right) + 2 \cdot c \cdot n$$

$$\hookrightarrow 8 \cdot T\left(\frac{n}{8}\right) + 3 \cdot c \cdot n$$

$$\begin{aligned} &\vdots \\ &= 2^m \cdot T\left(\frac{n}{2^m}\right) + m \cdot c \cdot n \quad \leftarrow \text{Nach allen } m \text{ Rekursions-} \\ &= 2^{\log_2(n)} \cdot T\left(\frac{n}{2^{\log_2(n)}}\right) + \log_2(n) \cdot c \cdot n \quad \text{ebenen} \\ &= n \underbrace{T\left(\frac{n}{n}\right)}_{\Theta(1)} + c \cdot n \cdot \log_2(n) \quad \text{Wir wissen} \\ &= \underbrace{n \cdot \Theta(1)}_{\Theta(n)} + \underbrace{c \cdot n \cdot \log_2(n)}_{\Theta(n \cdot \log_2(n))} \quad \text{aber } m = \log_2(n) \\ &\in \Theta(n \cdot \log_2(n)) \rightarrow \text{enthält } \mathcal{O}(n \cdot \log_2(n)) \end{aligned}$$

# Aufgabe 5.1 (c)

- Wir wissen  $d \cdot n \cdot \log_2(d \cdot n) \in O(n \cdot \log(n)) \rightarrow d \text{ konstant}$
- Zeigen wir  $T(n) \leq d \cdot n \cdot \log_2(d \cdot n)$  so ist auch  $T(n) \in O(n \cdot \log(n))$

→ Wir betrachten nur Zweierpotenzen als Eingabe → Induktions schritt verdoppelt Eingabe

IB:  $n=1 : T(1) = a \text{ (konstant)} \quad a \leq d \cdot \log_2(d) \quad \vee \quad \text{für } d \geq \max\{a, 2\}$

IA:  $T(x) \leq dx \cdot \log_2(dx)$  ist wahr für alle Zweierpotenzen zwischen 1 und  $n-1$  ( $m > 0 ; n = 2^m$ )

IS:  $T(n) = 2T\left(\frac{n}{2}\right) + cn \leq 2 \cdot \left(d \frac{n}{2} \cdot \log_2\left(d \cdot \frac{n}{2}\right)\right) + cn = d \cdot n \cdot \log_2\left(d \cdot n \cdot \frac{1}{2}\right) + cn$   
 $\downarrow \quad \downarrow$   
Rek Def. IA  
 $= d \cdot n \cdot \log_2\left(d \cdot n \cdot 2^{-1}\right) + cn \quad | \quad \log(x \cdot y) = \log(x) + \log(y)$   
 $= d \cdot n \cdot (\log_2(d \cdot n) + \log_2(2^{-1})) + cn \quad | \quad \log_b(b^x) = x$   
 $= dn \cdot (\log_2(dn) - 1) + cn$   
 $= dn \cdot \log_2(dn) - dn + cn \leq dn \cdot \log_2(dn)$

$\left. \begin{array}{l} d \geq \max\{a, 2\} \\ d \geq c \end{array} \right\} d \geq \max\{2, a, c\}$   
• Konstante  $d$  ist mindestens so groß

# Aufgabe 5.2

## Betrunkener Übungsleiter

# Aufgabe 5.2

Wir betrachten einen torkelnden Übungsleiter an einer Kletterwand, der die folgenden beiden Operationen durchföhren kann:

- **hoch**
- **runter(int  $k$ )**

Die Starthöhe des Übungsleiters beträgt 0 Meter.

Durch die Operation **hoch** steigt der Übungsleiter von seiner aktuellen Position aus genau einen Meter höher. Diese Operation hat die Laufzeit 1.

Durch die Operation **runter(int  $k$ )** fällt der Übungsleiter von seiner aktuellen Position aus exakt  $\min\{h, k\}$  Meter nach unten, wobei  $h$  die aktuelle Höhe des Übungsleiters ist. (Das bedeutet, dass der Übungsleiter nie tiefer als seine Starthöhe sinken kann). Wir nehmen hierbei an, dass  $k$  stets eine natürliche Zahl (nicht-negativ) ist. Die Operation **runter(int  $k$ )** hat die Laufzeit  $\min\{h, k\}$ .

Zeigen Sie mithilfe der Bankkonto-Methode, dass die amortisierten Laufzeiten der Operationen **hoch** und **runter(int  $k$ )** in  $\mathcal{O}(1)$  liegen.

# Aufgabe 5.2

- Wir suchen  $\Delta(\text{hoch})$  und  $\Delta(\text{runter}(k))$  für amortisierte Laufzeiten  $\mathcal{O}(1)$ .
- $T(\text{hoch})$  ist bereits konstant, also könnte ein konstanter Betrag eingezahlt werden ohne  $\mathcal{O}(1)$  zu verletzen.
- $T(\text{runter}(k))$  ist abhängig von  $\min\{k, h\} \rightarrow$  um konstant zu werden liegt ein  $\Delta$  von  $-\min\{k, h\}$  nahe  
(Summe  $T(\dots) + \Delta(\dots) = 0$  ist konstant)

$$\Rightarrow \Delta(\text{hoch}) = 1$$

$$\Delta(\text{runter}(h, k)) = -\min\{h, k\}$$

$$\Rightarrow A(\text{hoch}) = T(\text{hoch}) + \Delta(\text{hoch}) = 1 + 1 = 2 \in \mathcal{O}(1) \checkmark$$

$$\Rightarrow A(\text{runter}(k)) = T(\text{runter}(k)) + \Delta(\text{runter}(k)) = \min\{h, k\} - \min\{h, k\} = 0 \in \mathcal{O}(1) \checkmark$$

- Laufzeiten somit erreicht Validität noch zu prüfen:

→ Konto darf nie negativ sein  
→ Annahme: Kontostand vor Operation gleich der Höhe  $h$   
Basecase:  
 $h=0$  Konto=0 (Höhe ist nicht-negativ)  
hoch: Erhöht Höhe um 1 } gleich  
Erhöht Konto um 1 }  
runter(k): Verringert Höhe um  $\min\{k, h\}$   
Verringert Konto um  $\min\{k, h\}$   
⇒ Kontostand immer gleich Höhe  $\rightarrow \geq 0$

# Aufgabe 5.3

# Stapelschlange

# Aufgabe 5.3

In dieser Aufgabe geht es darum, einen Algorithmus, der eine Queue für Integer-Zahlen mittels zweier Stacks implementiert, hinsichtlich seiner Laufzeit zu untersuchen.

- Geben Sie die Laufzeitklasse der Worst-Case-Laufzeit eines Aufrufs der ***dequeue()***-Methode in Abhängigkeit der aktuellen Größe der Queue  $n$  in Landau-Notation an.
- Überlegen Sie sich ein Amortisierungsschema, welches die amortisierte Laufzeit der einzelnen Operationen minimiert. Nennen Sie die amortisierten Laufzeitklassen der ***enqueue(int v)***- und ***dequeue()***-Methode, die sich aus Ihrem Amortisierungsschema ergeben. Zeigen Sie die Richtigkeit Ihres Amortisierungsschemas (das Tokenkonto darf niemals negativ werden) und der resultierenden Laufzeitklassen.

```
1 class Stapelschlaenge {  
2     private Stack s1 = new Stack();  
3     private Stack s2 = new Stack();  
4  
5     public void enqueue(int v) {  
6         s1.push(v);  
7     }  
8  
9     public int dequeue() {  
10        if(s2.isEmpty())  
11            while(!s1.isEmpty())  
12                s2.push(s1.pop());  
13        return s2.pop();  
14    }  
15}
```

Methode	Beschreibung	Laufzeitklasse
<b>void push(int v)</b>	legt eine Zahl auf den Stack	$\mathcal{O}(1)$
<b>int pop()</b>	nimmt die oberste Zahl vom Stack	$\mathcal{O}(1)$
<b>boolean isEmpty()</b>	prüft, ob der Stack leer ist	$\mathcal{O}(1)$

# Aufgabe 5.3 (a)

Dequeue - Worst - Case : 

- In diesem Fall muss befinden sich alle  $n$  Elemente der Queue auf Stack S1.
- Alle Elemente werden dann eins nach dem anderen von S1 gepoppt und auf S2pusht.
- Nach der Operation enthält S2 alle Elemente der Queue mit dem ältesten ganz oben und dem neuesten ganz unten

$\Rightarrow$  1. Check isEmpty + 1. Return (konstant)

$\Rightarrow$   $n$  Pop- und  $n$  Push-Befehle

$\left( \begin{array}{l} \text{Push : } O(1) \\ \text{Pop : } O(1) \\ \text{isEmpty : } O(1) \end{array} \right)$

$\Rightarrow T(\text{dequeue}) \in O(n)$

# Aufgabe 5.3 (b)

Worst-Case Laufzeiten :  $T(\text{enqueue}) \in O(1)$  (nur 1 push)

$$T(\text{dequeue}) \in \begin{cases} O(1) + c \cdot n & \text{wenn } s2.\text{isEmpty}() \\ O(1) & \text{sonst} \end{cases} \quad (\text{check isEmpty + return + umschichten von } s1 \text{ auf } s2, n \cdot \text{konstanter Aufwand})$$

- Wir wissen, jedes Element muss vor dem Verlassen der Queue einmal von  $s1$  zu  $s2$  geschoben werden (Kosten  $c$  pro Element)
- Idee: Beim Einfügen bezahlt ein Element schon für sein Verschieben ein ( $c$  Tokens)  
Dies kompensiert dann die Kosten beim Verschieben.

$$\Delta(\text{enqueue}) = c$$

$$\Delta(\text{dequeue}) = \begin{cases} -c \cdot n & \text{wenn } s2.\text{isEmpty}() \\ 0 & \text{sonst} \end{cases}$$

\* Anmerkung: Für  $\text{!s2.isEmpty}()$  wird weder verschoben noch hinzugefügt, darum keine Änderung hier

$$\Rightarrow A(\text{enqueue}) = T(\text{enqueue}) + \Delta(\text{enqueue}) = O(1) + c \in O(1)$$

$$A(\text{dequeue}) = T(\text{dequeue}) + \Delta(\text{dequeue}) = \begin{cases} O(1) + cn - cn \in O(1) & \text{wenn } s2.\text{isEmpty}() \\ O(1) + 0 \in O(1) & \text{sonst} \end{cases}$$

$\Rightarrow$  Wenn  $s2$  leer sind  $n$  Elemente in  $s1$ . Per Definition wurden somit  $c \cdot n$  Tokens eingezahlt. Dequeue reduziert dies auf 0. Und der Zyklus startet von vorne.  $\Rightarrow$  Nie negativ