Algorithmen und Komplexität

Robin Rausch, Florian Maslowski

30. Juni 2022

Inhaltsverzeichnis

1 Komplexitaet

Der Begriff Komplexität beschreibt die Frage:

> Wie teuer ist ein Algorithmus? Voll Teuer!

Genauergesagt wird hierfür ermittelt, wie viele elementare Schritte eine Algorithmus im Durchschnitt und schlimmstenfalls braucht. Diese beiden Werte spiegeln die Komplexität wieder.

1.1 $\mathcal{O}$-Notation

Die $\mathcal{O}$-Notation ist eine obere Grenze einer Funktion. $\mathcal{O}(f)$ ist die Menge aller Funktionen, die langfristig nicht wesentlich schneller wachsen als f.
Einige Beispiele sind zum Beispiel:

- $n^2 \in \mathcal{O}(n^3)$
- $3n^3 + 2n^2 + 17 \in \mathcal{O}(n^3)$
- $n\sqrt{n} \in \mathcal{O}(n^2)$

Rechenregeln für $\mathcal{O}$-Notation:

Für jede Funktion f				
	$g \in \mathcal{O}(f)$	$\Rightarrow$	$c \cdot g \in \mathcal{O}(f)$	Konstanter Faktor
$g \in \mathcal{O}(f) \wedge h \in \mathcal{O}(f)$		$\Rightarrow$	$g + h \in \mathcal{O}(f)$	Summe
$g \in \mathcal{O}(f) \wedge h \in \mathcal{O}(g)$		$\Rightarrow$	$h \in \mathcal{O}(f)$	Transivität
$\lim_{n\to\infty} \frac{g(n)}{f(n)} \in \mathbb{R}$		$\Rightarrow$	$g \in \mathcal{O}(f)$	Grenzwert

1.1.1 Landau-Symbole

$g \in \Omega(f)$	g wächst mindestens so schnell wie f	$\lim_{x\to\infty} \frac{f(x)}{g(x)} = c \in \mathbb{R}$
$g \in \Theta(f)$	g wächst genau so schnell wie f	$\lim_{x\to\infty} \frac{g(x)}{f(x)} = c \in \mathbb{R}^{>0}$
$g \sim f$	g wächst genau so schnell wie f	$\lim_{x\to\infty} \frac{g(x)}{f(x)} = 1$

- Betrachten Sie folgende Funktionen:
  - $h_1(x) = x^2 + 100x + 3$
  - $h_2(x) = x^2$
  - $h_3(x) = \frac{1}{3}x^2 + x$
  - $h_4(x) = x^3 + x$

$g \in \mathcal{O}(f)$: $\lim_{x\to\infty} \frac{g(x)}{f(x)} = c \in \mathbb{R}$

$g \in \Omega(f)$: $\lim_{x\to\infty} \frac{f(x)}{g(x)} = c \in \mathbb{R}$

$g \in \Theta(f)$: $\lim_{x\to\infty} \frac{g(x)}{f(x)} = c \in \mathbb{R}^{>0}$

$g \sim f$: $\lim_{x\to\infty} \frac{g(x)}{f(x)} = 1$

Vervollständigen Sie die Tabelle. Zeile steht in Relation ... zu Spalte:

	h_1	h_2	h_3	h_4
h_1	$\mathcal{O}, \Omega, \Theta, \sim$	$\mathcal{O}, \Omega, \Theta, \sim$	$\mathcal{O}, \Omega, \Theta$	$\mathcal{O}$
h_2	$\mathcal{O}, \Omega, \Theta, \sim$	$\mathcal{O}, \Omega, \Theta, \sim$	$\mathcal{O}, \Omega, \Theta$	$\mathcal{O}$
h_3	$\mathcal{O}, \Omega, \Theta$	$\mathcal{O}, \Omega, \Theta$	$\mathcal{O}, \Omega, \Theta, \sim$	$\mathcal{O}$
h_4	Ω	Ω	Ω	$\mathcal{O}, \Omega, \Theta, \sim$

Zur Θ-Notation gibt es auch ein eigenes *Master-Theorem*.

1.2 Logarithmen

Der Logarithmus beschreibt die Umkehrfunktion zur Potenzierung:

$$\log_a a^b = b$$

Wir werden meist den Logarithmus zur Basis 2 brauchen.
Rechenregeln mit dem Logarithmus:

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

$$\log_a x = \frac{1}{\log_b a} \log_b x = c \log_b x$$

$$\mathcal{O}(\log_a x) = \mathcal{O}(c \log_b x) = \mathcal{O}(\log_b x) \Rightarrow \text{Die Basis ist für } \mathcal{O} \text{ irrelevant!}$$

1.3 Dynamisches Programmieren

Dynamisches Programmieren ist eine Optimierung der normalen Programmierung. Hierfür werden Probleme in kleinere Teilprobleme aufgeteilt. Die Teilprobleme werden gelöst und dann die Gesamtlösung aus den Teillösungen rekonstruiert.
Die Dynamische Programmierung versagt, wenn...

- Einzellösungen nicht wiederverwendet werden können
- die globale Lösung sich nicht einfach aus lokalen Lösungen zusammensetzen lässt

Ein Beispiel für die Verwendung der dynamischen Programmierung ist eine rekursive Fibonacci-Funktion.

1.4 Rekurrenzen

Rekursion? –> Kein Plan was ich hierzu reinschreiben soll? @flo

1.5 Divide & Conquer

Der Divide & Conquer-Ansatz teilt ein Problem in zwei gleich große Hälften und muss somit nur noch ein halb so großes Problem lösen.
Ein Algorithmus der...

- ein Problem in mehrere Teile aufspaltet
- die Teilprobleme (rekursiv) löst
- die Teillösungen zu einer Gesamtlösung kombiniert

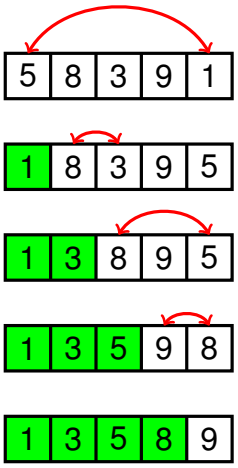
1.5.1 Komplexität von rekursiven Algorithmen

2 Einfache Sortierverfahren

2.1 Selectionsort

Speicher: In-place
Stabilität: Stabil
Komplexität: $\mathcal{O}(n^2)$

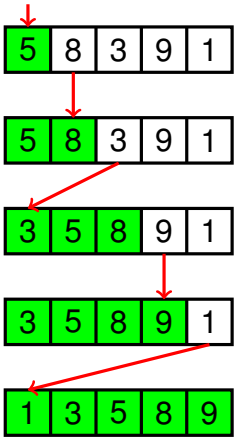
1. Finde kleinstes Element in Folge($a_0, ...a_{k-1}$)
2. Vertausche a_{min} mit a_0
3. finde kleinstes Element in Folge($a_1, ...a_{k-1}$)
4. Vertausche a_{min} mit a_1
5. ...



2.2 Insertionsort

Speicher: In-place
Stabilität: Stabil
Komplexität: $\mathcal{O}(n^2)$

1. Verwende erstes Element von In als erstes Element von Out
2. füge zweites Element von In an korrekte Position in Out
3. füge drittes Element von In an korrekte Position in Out
4. ...



Wen man den Insertionsort In-place verwenden will, sind In und Out gleich.

2.2.1 Indirektes Sortieren

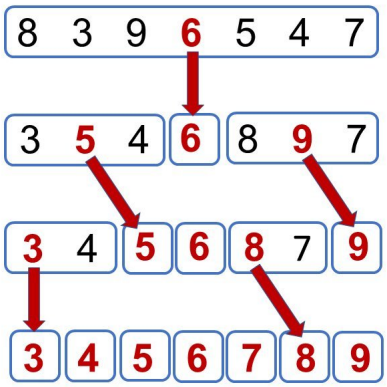
2.3 Bubblesort

3 Divide & Conquer Sortierverfahren

3.1 Quicksort

Speicher: In-place
Stabilität: Instabil
Komplexität: $\mathcal{O}(n \cdot \log n)$

1. Wenn $|S| \leqslant 1$: fertig
2. Wähle Pivot-Element $p \in S$
   (a) Pivot: Dreh- und Angelpunkt
   (b) idealerweise: Mittlere Größe
   (c) teile Folge in zwei Teilfolgen S< und S$\geqslant$
      i. $\forall_{a \in S<} : a < p$
      ii. $\forall_{a \in S\geqslant} : a \geqslant p$
3. Sortiere S< und S$\geqslant$ mittels Quicksort

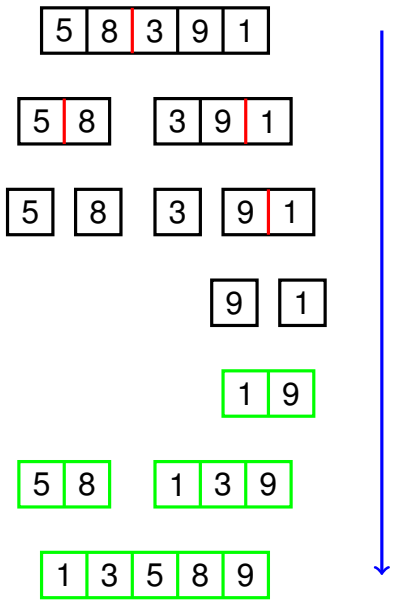


Warum ist Quicksort so effizient?
Da zuerst grob sortiert und dann immer feiner sortiert wird. Ebenso werden Elemente zwischen denen einmal ein Pivot lag nie wieder miteinander verglichen.

3.2 Mergesort(Top-Down)

Speicher: Out-of-place
Stabilität: Stabil
Komplexität: $\mathcal{O}(n^2)$

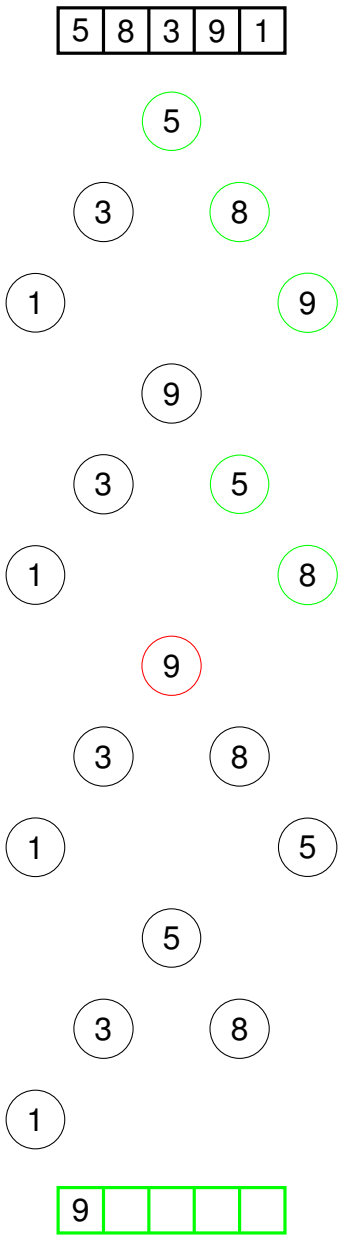
1. Wenn |S| $\leq$ 1: gib S zurück
2. Teile S in zwei gleich lange Folgen L und R
3. Sortieren L und R(rekursiv)
4. Vereinige L und R zu S':
   (a) solange L oder R nicht leer sind:
   (b) m:= min(l_1, r_1)
   (c) entferne m aus L bzw. R
   (d) hänge m an S' an
5. gib S' zurück



4 Heap Sortierverfahren

4.1 Heapsort

Speicher:	In-place
Stabilität:	Instabil
Komplexität:	$\mathcal{O}(n \cdot log(n))$



1. Liste in Binärbaum übertragen
2. Solange Binärbaum nicht leer:
   (a) Von unten nach oben Heapeigenschaften herstellen
   (b) Von oben nach unten Heapeigenschaften kontrollieren
   (c) Wurzel mit der letzten Stelle des Heaps (unterste -> rechteste) tauschen und in neuen (sortierten) Array schreiben, weil diese nun Sortiert ist

5 Binäre Suchbäume

6 AVL-Bäume

Binäre Suchbäume mit maximaler Höhendifferenz von 2.
Begriffe:

1. Höhe/Tiefe: Anzahl der Knoten auf dem längsten Ast
2. Gewicht: Anzahl der Knoten
3. Balance: Tiefenerschied ist nicht Höher als 2