Algorithmen und Komplexität

Robin Rausch, Florian Maslowski

2. Juli 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:

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

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 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 Greedy Algorithmen

Greedy-Algorithmen zeichnen sich dadurch aus, dass sie zum aktuellen Zustand t den besten Weg einschlagen und nicht voraus $(t+1)$ planen. Anders gesagt: Ein Greedy-Algorithmus entscheidet sich immer für denjenigen Schritt, der ihn dem Ziel am nachsten bringt.

1.6 Rekurrenzen

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

1.7 Komplexität von rekursiven Algorithmen

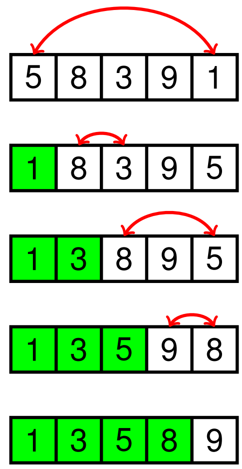
Rekursion? –> Kein Plan was ich hierzu reinschreiben soll? @flo

2 Einfache Sortierverfahren

2.1 Selectionsort

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

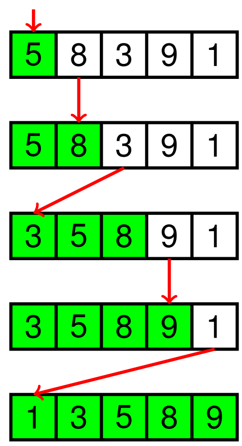
1. Finde kleinstes Element in Folge$(a_0, ...a_{k-1})$
2. Vertausche a_{min} mit a_0
3. finde kleinstes Element in Folge$(a_1, ...a_{k-1})$
4. Vertausche a_{min} mit a_1
5. ...



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

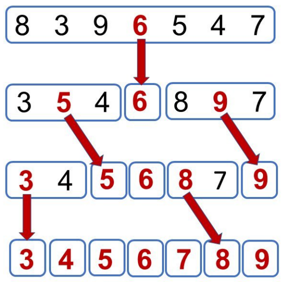
2.3 Bubblesort

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

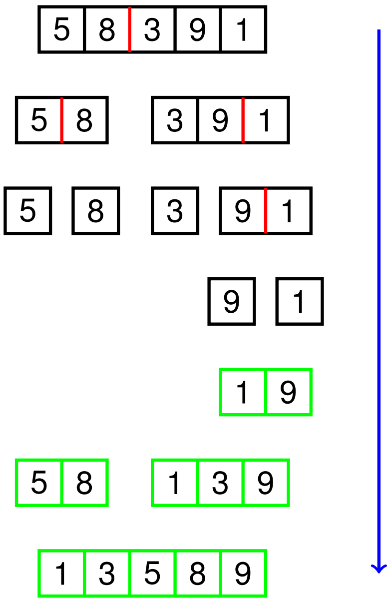


Warum ist Quicksort so effizient?
Da zuerst grob sortiert und dann immer feiner sortiert wird. Ebenso werden Elemente zwischen denen einmal ein Pivot lag nie wieder miteinander verglichen.

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

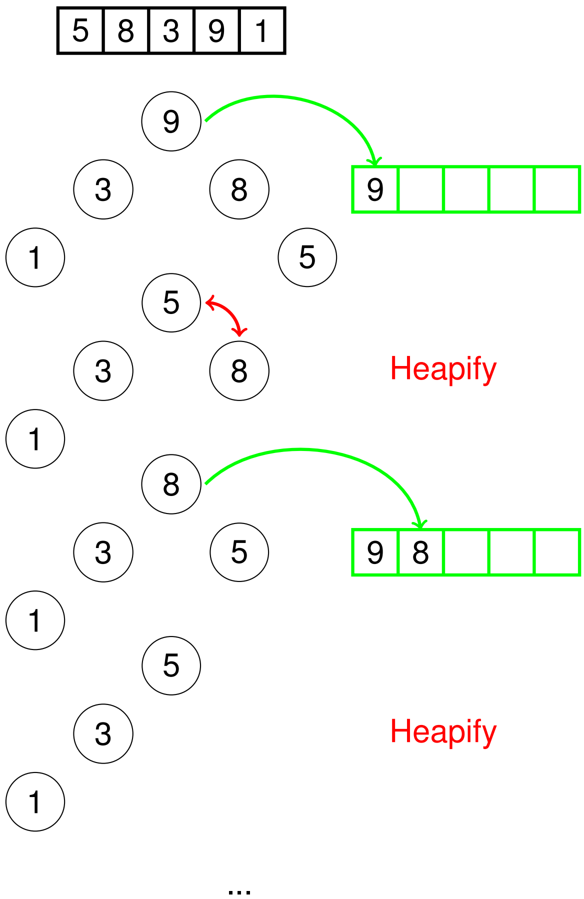


4 Heap Sortierverfahren

4.1 Heapsort

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

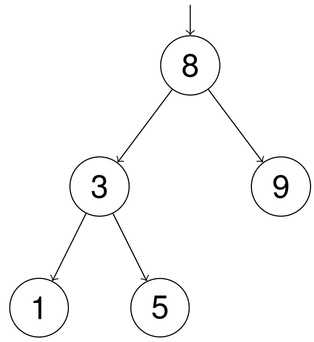
1. Liste in Baum übertragen
2. Solange Baum nicht leer:
   (a) Heapify(Größte nach oben)
   (b) Oberste Element in die Liste einfügen



5 Binäre Suchbäume

Ein binärer Suchbaum ist ein Binärbaum mit folgende Eigenschaften:

1. Die Knoten des Baums sind mit Schlüsseln aus einer geordneten Menge *K* beschriftet
2. Für jeden Knoten gilt:
   (a) Alle Schlüssel im linken Teilbaum von *N* sind kleiner als der Schlüssel von *N*
   (b) Alle Schlüssel im rechten Teilbaum von *N* sind größer als der Schlüssel von *N*



5.1 Suchen

Komplexität: $\mathcal{O}(log(n))$

1. Gegeben: Baum *B* mit Wurzel *W*, Schlüssel *k*
2. Algorithmus: Suche *k* in *B*
   (a) Wenn *B* leer ist: Ende, *k* ist nicht in *B*
   (b) Wenn *W.key* $>$ *k*: Suche im linken Teilbaum von B
   (c) Wenn *W.key* $<$ *k*: Suche im rechten Teilbaum von B
   (d) Sonst: *W.key* = *k*; Ende, gefunden

5.2 Einfügen

Komplexität: $\mathcal{O}(log(n))$

1. Gegeben: Baum *B* mit Wurzel *W*, Schlüssel *k*
2. Gesucht: Baum *B'*, der aus *B* entsteht, wenn *k* eingefügt wird
3. Idee:
   (a) Suche nach *k*
   (b) Falls *k* nicht in *B* ist, setze es an der Stelle ein, an der es gefunden worden wäre
4. Implementierung z.B. funktional:
   (a) Wenn *B* leer ist, dann ist ein Baum mit einem Knoten mit Schlüssel *k* der gesuchte Baum
   (b) Ansonsten:
      i. Wenn *W.key* $>$ *k*: Ersetze den linken Teilbaum von *B* durch den Baum, der entsteht, wenn man *k* in ihn einfügt
      ii. Wenn *W.key* $<$ *k*: Ersetze den rechten Teilbaum von *B* durch den Baum, der entsteht, wenn man *k* in ihn einfügt
      iii. Ansonsten: *k* ist schon im Baum

5.3 Löschen

Beim Löschen in Binärbäumen muss man beachten, dass der zu löschende Knoten auch Nachfolger haben kann. Aus diesem Grund gibt es hierbei eine Fallunterscheidung:

Komplexität: $\mathcal{O}(log(n))$

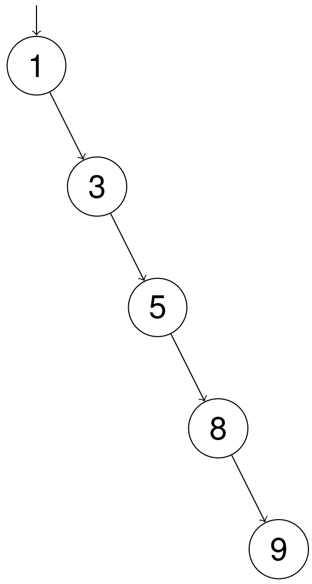
1. Problem: Entferne einen Knoten *K* mit gegebenen Schlüssel *k* aus dem Suchbaum
   (a) ... und erhalte die Binärbaumeigenschaft
   (b) ... und erhalte die Suchbaumeigenschaft
2. Fallunterscheidung:
   (a) Fall 1: Knoten hat keinen Nachfolger
      i. Lösung: Schneide Knoten ab
      ii. Korrektheit: Offensichtlich
   (b) Fall 2: Knoten hat einen Nachfolger
      i. Lösung: Ersetze Knoten durch seinen einzigen Nachfolger
      ii. Korrektheit: Alle Knoten in diesem Baum sind größer(bzw. kleiner) als die Knoten im Vorgänger des gelöschten Knotens
   (c) Fall 3: Knoten hat zwei Nachfolger
      i. Lösung:
         A. Suche größten Knoten *G* im linken Teilbaum
         B. Tausche *G* und *K* (oder einfacher: Ihre Schlüssel/Werte)
         C. Lösche rekursiv *k* im linken Teilbaum von (nun) *G*

5.4 Ausgabe

Komplexität: $\mathcal{O}(n)$

1. Gegeben: Baum *B* mit Wurzel *W*
2. Algorithmus: Gib alle *W* in *B* aus
   (a) Wenn *B* leer ist: Ende
   (b) Ausgabe(*W*) ist rekursiv:
      i. Wenn *W.linkes Kind* existiert: Ausgabe(*W.linker Teilbaum*)
      ii. Gib *W* aus
      iii. Wenn *W.rechtes Kind* existiert: Ausgabe(*W.rechter Teilbaum*)

5.5 Balacierte und Unbalancierte Bäume



Binärbaume können entarten, indem man zum Beispiel eine sortierte Liste in einen Binärbaum umwandelt. In diesem Fall entsteht solch ein unbalacierter Baum:

Diese entarteten Binärbäume sind sehr ineffizient und sollten rebalanciert werden. Die Rebalancierung zu Balancierten Binärbäumen geschieht zum Beispiel bei AVL-Bäumen.

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

6.1 Rotieren

6.2 Balancieren

6.3 Einfügen

6.4 Suchen

6.5 Löschen

7 Hashing und Hashtabellen

8 Master-Theorem

9 Master-Theorem nach Landau

10 Graph-Algorithmen

10.1 Minimale Spannbäume & Algorithmus von Prim

10.2 Kürzeste Wege/Dijkstra