Übung Algorithmen und Datenstrukturen

Lucas Heimberg

Sommersemester 2017

Die Folien basieren zu großen Teilen auf Folien von Marc Bux und Patrick Schäfer

Termine

Vorlesung

Montag	11–13 Uhr	Ulf Leser	RUD 26, 0'115
Mittwoch	11–13 Uhr	Ulf Leser	RUD 26, 0'115

Übungen

Montag	13–15 Uhr	Berit Grußien	RUD 26, 1'303	Gruppe 1
Montag	13–15 Uhr	Marc Bux	RUD 26, 1'305	Gruppe 2
Dienstag	13–15 Uhr	Marc Bux	RUD 26, 0'313	Gruppe 3
Dienstag	13–15 Uhr	Berit Grußien	RUD 26, 1'303	Gruppe 4
Mittwoch	13–15 Uhr	Patrick Schäfer	RUD 26, 0'313	Gruppe 5
Mittwoch	15–17 Uhr	Patrick Schäfer	RUD 26, 0'313	Gruppe 6
Donnerstag	09–11 Uhr	Lucas Heimberg	RUD 26, 0'313	Gruppe 7
Freitag	09–11 Uhr	Lucas Heimberg	RUD 26, 1'303	Gruppe 8

Klausur

7. August und 29. September 2017, jeweils 11–15 Uhr

Webseiten

Übung	`http://hu.berlin/algodat17`
Vorlesung	`http://hu.berlin/vl_algodat17`
Moodle	`http://moodle.hu-berlin.de`

Zeitplan

Kalenderwoche	Vorbereitung	Abgabe	Vorrechnen
17	Blatt 1 (1 & 2)		
18	Blatt 1 (3 & 4)		
19	Blatt 2	Blatt 1	
20			Blatt 1
21	Blatt 3	Blatt 2	
22			Blatt 2
23	Blatt 4	Blatt 3	
24			Blatt 3
25	Blatt 5	Blatt 4	
26			Blatt 4
27	Blatt 6	Blatt 5	
28			Blatt 5
29		Blatt 6	Blatt 6

Vorrechnen

- Freiwillige vor, ansonsten wird gelost (seid vorbereitet!)
- Kopie der eigenen Abgabe machen!

Übungsaufgaben

- Veröffentlichung alle 2 Wochen in Moodle und auf Webseite
- Jedes Aufgabenblatt muss bearbeitet werden

Schriftliche Aufgaben

- Jede Aufgabe separat auf Papier abgeben,
  Blätter einer Aufgabe zusammentackern
- Abgabe vor der Vorlesung bis 11:10 Uhr
  oder bis 10:45 Uhr im Briefkasten bei Raum RUD25, 3.321
- vor Abgabe kopieren (für Vorrechnen in Übung)

Programmieraufgaben (Java 8)

- vorher auf `gruenau2` testen
- gleicher Termin wie schriftliche Aufgaben

Auf jedes Blatt und in jede Java-Datei jeder Abgabe:

- Namen
- CMS-Benutzernamen
- Moodle-Arbeitsgruppe
- gewünschten Rückgabetermin (eindeutig: Wochentag, Uhrzeit, Dozent)

Übungsschein

50% der Punkte (150) → Übungsschein = Prüfungszulassung

Anmeldung in Agnes

Keine Zulassungsgarantie zum Wunschtermin

Anmeldung in Moodle

- Einschreibeschlüssel

  `Do.qc56lx` bzw. `Fr.fqjlj7`

- Abgabegruppen aus 2 (bis max. 3) Studenten
  von beliebigen Übungsterminen
- wichtige Nachrichten werden über Moodle(-Foren) versendet

Abzüge

- Abgaben ohne Anmeldung in Moodle: 0 Punkte
- Abgaben mit falscher Gruppengröße: 0 Punkte
- bei vermutetem Abschreiben: 0 Punkte
- nicht ausführbare Programmieraufgaben: 0 Punkte

To do

Jetzt:

- Wer hat den Übungsschein bereits?
- Wer hat noch keinen Übungspartner?

Für nächste Woche:

- Kein bestätigter Übungstermin in Agnes?
  $\rightarrow$ Termin auswählen und Berit Grußien anschreiben
- Kein Übungspartner? $\rightarrow$ Übungspartner finden
  (z.B. über den entsprechenden Post im Moodle-Forum)
- in Moodle anmelden und Abgabegruppe bilden
- über Agnes und Moodle versandte Nachrichten regelmäßig lesen

Inhaltlich:

- $\mathcal{O}$-Tutorial auf Übungswebseite durchlesen
- Grenzwerte, Ableitungen, Potenz- und Logarithmengesetze wiederholen
- Aufgabenblatt 1 bearbeiten

Übung 1

Landau-Notation

Algorithmus M(A)

Input: Array A der Länge $|A| = n$ mit $n \geq 2$
Output: Zahl x

```
1: x := 0;
2: for i := 1 to n do
3:     for j := i + 1 to n do
4:         if x < |A[i] − A[j]| then
5:             x := |A[i] − A[j]|;
6:         end if
7:     end for
8: end for
9: return x ;
```

Landau-Notation

Ziel

Abschätzung der Anzahl notwendiger Operationen eines Algorithmus

- als Funktion der Eingabe
- i.A. im Worst Case, d.h., bei ungünstigster Eingabe
- unabhängig von Hardware und Implementierung

Grundidee

Welche Terme bestimmen die Laufzeit des Algorithmus, wenn die Eingabe sehr groß wird?

- Konstante Summanden und Faktoren sind unerheblich
- Nur der Term mit dem größten Wachstum ist interessant

f wächst höchstens so schnell wie g

Seien $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$.
f wächst höchstens so schnell wie g bzw. g ist obere Schranke für $f \iff$

- $f(n) \leq c \cdot g(n)$ für irgendein $c > 0$, und
- für alle $n \geq n_0$, für irgendein $n_0 > 0$.

Anders formuliert: $f \in \mathcal{O}(g)$, wobei

$$\begin{aligned} \mathcal{O}(g) := \quad \{f : \mathbb{N} \to \mathbb{R}_{\geq 0} \mid & \exists c > 0 \\ & \exists n_0 \geq 0 \\ & \forall n \geq n_0 : f(n) \leq c \cdot g(n)\} \end{aligned}$$

Gilt für die folgenden Funktionen $f \in \mathcal{O}(g)$?

- $f(n) = k$ für $k > 0$, $\quad g(n) = 1$
- $f(n) = 1000n$, $\quad g(n) = n^2$
- $f(n) = 3n^5 + 4n^3 + 15$, $\quad g(n) = n^5$

Wichtige Komplexitätsklassen

$\mathcal{O}(1)$	konstant	Array-Zugriff
$\mathcal{O}(\log n)$	logarithmisch	Binäre Suche
$\mathcal{O}(n)$	linear	Sequentielle Suche
$\mathcal{O}(n \log n)$	linear-logarithmisch	Merge Sort
$\mathcal{O}(n^2)$	quadratisch	Bubble Sort
$\mathcal{O}(n^k)$	polynomiell	
$\mathcal{O}(2^n)$	exponentiell	Traveling Salesman

f wächst mindestens so schnell wie g

Seien $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$.
f wächst mindestens so schnell wie $g \iff$

- $f(n) \geq c \cdot g(n)$ für irgendein $c > 0$, und
- für alle $n \geq n_0$, für irgendein $n_0 > 0$.

Anders formuliert: $f \in \Omega(g)$, wobei

$$\begin{aligned} \Omega(g) := \quad \{f : \mathbb{N} \to \mathbb{R}_{\geq 0} \mid & \exists c > 0 \\ & \exists n_0 \geq 0 \\ & \forall n \geq n_0 : f(n) \geq c \cdot g(n)\} \end{aligned}$$

Gilt für die folgenden Funktionen $f \in \Omega(g)$?

- $f(n) = k$ für $k > 0, \quad g(n) = 1$
- $f(n) = 1000n, \quad g(n) = n^2$
- $f(n) = 3n^5 + 4n^3 + 15, \quad g(n) = n^5$

Zusammenhang zwischen $\mathcal{O}$ und Ω

Seien $f, g\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$. Dann gilt:

$$f \in \mathcal{O}(g) \iff g \in \Omega(f)$$

f wächst ungefähr genau so schnell wie g

Seien $f, g\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$.
f wächst ungefähr genau so schnell wie $g \iff$

- f höchstens so schnell wächst wie g, d.h., $f \in \mathcal{O}(g)$, und
- f mindestens so schnell wächst wie g, d.h., $f \in \Omega(g)$.

Anders formuliert: $f \in \Theta(g)$, wobei

$$\Theta(g) \;:=\; \mathcal{O}(g) \cap \Omega(g)$$

Gilt für die folgenden Funktionen $f \in \Theta(g)$?

- $f(n) = k$ für $k > 0, \quad g(n) = 1$
- $f(n) = 1000n, \quad g(n) = n^2$
- $f(n) = 3n^5 + 4n^3 + 15, \quad g(n) = n^5$

f wächst langsamer als g

Seien $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$.
f wächst langsamer als $g \iff$

- für jedes beliebig kleine $c > 0$ existiert ein $n_0 \geq 0$,
- so dass $f(n) < c \cdot g(n)$ für alle $n \geq n_0$.

Anders formuliert: $f \in o(g)$, wobei

$$o(g) := \{f : \mathbb{N} \to \mathbb{R}_{\geq 0} \mid \forall c > 0 \; \exists n_0 \geq 0 \; \forall n \geq n_0 : f(n) < c \cdot g(n)\}$$

Gilt für die folgenden Funktionen $f \in o(g)$?

- $f(n) = k$ für $k > 0$, $\quad g(n) = 1$
- $f(n) = 1000n$, $\quad g(n) = n^2$
- $f(n) = 3n^5 + 4n^3 + 15$, $\quad g(n) = n^5$

f wächst schneller als g

Seien $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$.
f wächst schneller als g $\iff$

- für jedes beliebig kleine $c > 0$ existiert ein $n_0 \geq 0$,
- so dass $f(n) > c \cdot g(n)$ für alle $n \geq n_0$.

Anders formuliert: $f \in \omega(g)$, wobei

$$\begin{aligned} \omega(g) := \quad \{f : \mathbb{N} \to \mathbb{R}_{\geq 0} \mid & \forall c > 0 \\ & \exists n_0 \geq 0 \\ & \forall n \geq n_0 : f(n) > c \cdot g(n)\} \end{aligned}$$

Zusammenhänge zwischen $\mathcal{O}, \Omega, \Theta, o, \omega$

Seien $f, g \colon \mathbb{N} \to \mathbb{R}_{\geq 0}$. Dann gilt:

$$
\begin{aligned}
&(1) && f \in o(g) && \Longrightarrow && f \in \mathcal{O}(g) \\
&(2) && f \in \omega(g) && \Longrightarrow && f \in \Omega(g) \\
&(3) && f \in \mathcal{O}(g) && \Longleftrightarrow && g \in \Omega(f) \\
&(4) && f \in o(g) && \Longleftrightarrow && g \in \omega(f) \\
&(5) && f \in o(g) && \Longrightarrow && f \notin \Omega(g) \\
&(6) && f \in \omega(g) && \Longrightarrow && f \notin \mathcal{O}(g).
\end{aligned}
$$

Beispiel

Seien $a, b \in \mathbb{N}_{\geq 1}$. Ist $a^{n+b} \in \Theta(a^n)$?

Zusammenfassung

Sei $g : \mathbb{N} \to \mathbb{R}_{\geq 0}$.

$$
\begin{aligned}
\mathcal{O}(g) &:= \{f : \mathbb{N} \to \mathbb{R}_{\geq 0} \mid \exists c > 0 \exists n_0 \geq 0 \forall n \geq n_0 : f(n) \leq c \cdot g(n)\} \\
\Omega(g) &:= \{f : \mathbb{N} \to \mathbb{R}_{\geq 0} \mid \exists c > 0 \exists n_0 \geq 0 \forall n \geq n_0 : f(n) \geq c \cdot g(n)\} \\
\Theta(g) &:= \mathcal{O}(g) \cap \Omega(g) \\
o(g) &:= \{f : \mathbb{N} \to \mathbb{R}_{\geq 0} \mid \forall c > 0 \exists n_0 \geq 0 \forall n \geq n_0 : f(n) < c \cdot g(n)\} \\
\omega(g) &:= \{f : \mathbb{N} \to \mathbb{R}_{\geq 0} \mid \forall c > 0 \exists n_0 \geq 0 \forall n \geq n_0 : f(n) > c \cdot g(n)\}
\end{aligned}
$$

Wachstum einer Funktion abschätzen

- Obige Definitionen direkt anwenden
- Zusammenhänge zwischen $\mathcal{O}, \Omega, \Theta, o, \omega$ nutzen
- Hinreichende Kritieren (auf den nächsten Folien)
- Ausnutzen von Transitivität von $\mathcal{O}$ und Monotonie (nächste Woche)

Wiederholung: Grenzwerte

Sei $a\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ eine Folge und sei $c \in \mathbb{R}_{\geq 0}$.

c ist ein Grenzwert von a

$\iff$ für jedes beliebig kleine $\epsilon > 0$ existiert ein $n_0 \geq 0$, so dass

$$|a(n) - c| < \epsilon \qquad \text{für alle } n \geq n_0.$$

Beispiele

- $a(n) = 3$ hat den Grenzwert 3
- $a(n) = \frac{1}{1+n}$ hat den Grenzwert 0
- $a(n) = n^2$ hat keinen Grenzwert
  (und divergiert bestimmt gegen ∞)
- $a(n) = \sin(n)$ hat keinen Grenzwert
  (und divergiert nicht bestimmt)

Limes

Sei $a\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ eine Folge.

$$\lim_{n\to\infty} a(n) \in \mathbb{R} \cup \{\infty\}$$

hat den folgenden Wert:

- $c \in \mathbb{R}_{\geq 0}$, wenn c Grenzwert von $a(n)$ ist.
- ∞, wenn $a(n)$ gegen ∞ bestimmt divergiert.
- $< \infty$, wenn $a(n)$ Grenzwert in $\mathbb{R}_{\geq 0}$ hat.
- $> c \in \mathbb{R}_{\geq 0}$, wenn $a(n)$ gegen ∞ bestimmt divergiert oder einen Grenzwert $> c$ hat.

Beispiele

- $\lim_{n\to\infty} 3 = 3$
- $\lim_{n\to\infty} \frac{1}{1+n} = 0$
- $\lim_{n\to\infty} n^2 = \infty$
- $\lim_{n\to\infty} \sin(n)$ ist nicht definiert

Hinreichendes Kriterium

Seien $f, g\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$. Dann gilt

(1) $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} < \infty \implies f \in \mathcal{O}(g)$ (2) $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = 0 \iff f \in o(g)$

(3) $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} > 0 \implies f \in \Omega(g)$ (4) $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = \infty \iff f \in \omega(g)$.

Gilt für die folgenden Funktionen $f \in \mathcal{O}(g)$?

- $f(n) = 23, \quad g(n) = \log\log n$
- $f(n) = n, \quad g(n) = \sqrt{n}$
- $f(n) = n^2 + n, \quad g(n) = 3n^2$

Gelten auch die Umkehrungen der Implikationen (1) und (3)?

Was tun wenn f und g bestimmt divergieren?

Sei $f(n) = \log n$ und $g(n) = \sqrt{n}$.

$$\lim_{n \to \infty} \frac{\log n}{\sqrt{n}} \quad = \quad ?$$

Ähnlich: Was tun wenn f und g den Grenzwert 0 haben?

Abschätzungen mit Hilfe des Satzes von l'Hôpital

Seien $f, g\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ differenzierbar und

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

Falls der Grenzwert $\lim_{n\to\infty} \frac{f'(n)}{g'(n)}$ existiert oder $\frac{f'(n)}{g'(n)}$ gegen ∞ bestimmt divergiert, dann ist

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \quad = \quad \lim_{n\to\infty} \frac{f'(n)}{g'(n)}.$$

Gilt für die folgenden Funktionen $f \in o(g)$?

- $f(n) = \log n, \quad g(n) = \sqrt{n}$
- $f(n) = n^2 + 2n + 3, \quad g(n) = 2n^2 + n - 2$

Übung 2

Zum Aufwärmen

Sei

- $g(n) = n^5$
- $h(n) = (n+1)^6$.

Zeige, dass $g \in \mathcal{O}(h)$ mit

- Definition von $\mathcal{O}$,
- Grenzwert-Kriterium.

Gibt es Funktionen $f, g \colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ so dass $f \notin \mathcal{O}(g)$ und $g \notin \mathcal{O}(f)$?

Transitivität

Seien $f, g, h\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$. Dann gilt:

$$f \in \mathcal{O}(g), g \in \mathcal{O}(h) \quad \Longrightarrow \quad f \in \mathcal{O}(h)$$

Beispiel

- $f(n) = 3n^5 + 4n^3 + 15$
- $g(n) = n^5$
- $h(n) = (n+1)^6$

Zeige, dass $f \in \mathcal{O}(h)$.

Monotonie

Seien $f, g, h\colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$. Sei f streng monoton wachsend. Dann gilt:

$$\exists n_0 \geq 0\ \forall n \geq n_0 \colon g(n) \leq h(n)$$

$$\iff \exists n_0' \geq 0\ \forall n \geq n_0' \colon f(g(n)) \leq f(h(n))$$

Beispiel

Zeige, dass $2^{\log n} \in \mathcal{O}(2^{\sqrt{n}})$.

Logarithmen

Erinnerung:

$$\log n = \log_2 n, \quad \ln n = \log_e n$$

Bereits genutzt:

$$\log n = \frac{\ln n}{\ln 2} \qquad \text{allgemeiner: } \log_a n = \frac{\log_b n}{\log_b a} \quad \text{f. a. } a, b \in \mathbb{R}_{>0}.$$

Anwendung

Seien $a, b \in \mathbb{R}_{>0}$ beliebig. Dann ist

$$\log_a n \in \Theta(\log_b n).$$

Übungsaufgaben

Beweisen oder widerlegen Sie für die Teilaufgaben (a) bis (d) die folgenden Aussagen:

$$f \in \mathcal{O}(g), f \in \Omega(g).$$

	$f(n)$	$g(n)$
(a)	$\sqrt{n}$	$n^{\frac{3}{4}}$
(b)	$\sqrt{n} + \log n$	$2\sqrt{n}$
(c)	2^n	2^{n+1}
(d)	2^n	$n!$

Pseudocode-Analyse (1)

Algorithmus M(A)

Input: Array A der Länge $|A| = n$ mit $n \geq 2$
Output: Zahl x

```
1: x := 0;
2: for i := 1 to n do
3:     for j := i + 1 to n do
4:         if x < |A[i] − A[j]| then
5:             x := |A[i] − A[j]|;
6:         end if
7:     end for
8: end for
9: return x ;
```

- Was berechnet der Algorithmus **M**?
- Analysieren Sie die Laufzeit des Algorithmus in Abhängigkeit von n.
- Entwickeln Sie einen optimalen Algorithmus

Nützliche Abschätzungen in der Laufzeitanalyse

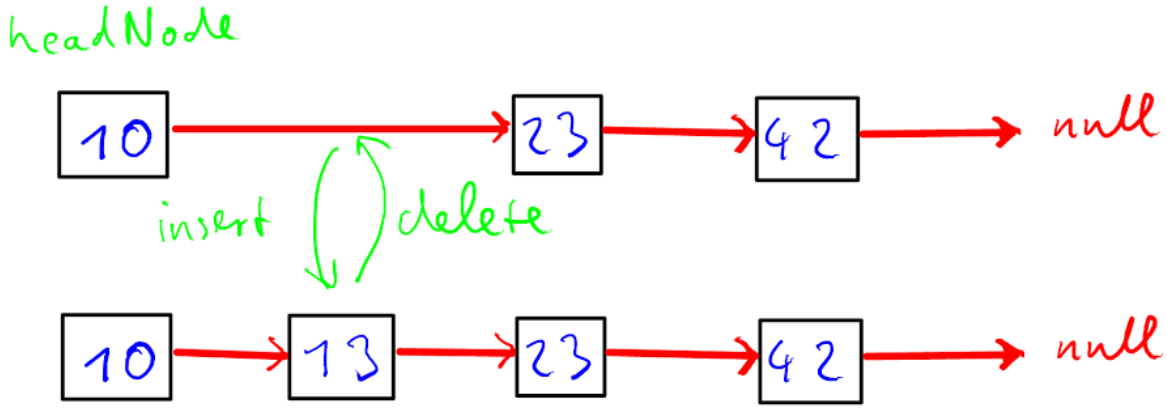
Seien $f, g \colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ und $c > 0$. Dann gilt

$$\begin{aligned} \mathcal{O}(c + f) &\subseteq \mathcal{O}(f) \\ \mathcal{O}(c \cdot f) &\subseteq \mathcal{O}(f) \\ \mathcal{O}(f) + \mathcal{O}(g) &\subseteq \mathcal{O}(\max\{f, g\}) \\ \mathcal{O}(f) \cdot \mathcal{O}(g) &\subseteq \mathcal{O}(f \cdot g) \end{aligned}$$

Übung 3

- (Sortierte) Listen
- Stacks & Queues
- Rekursion und vollständige Induktion
- Divide & Conquer

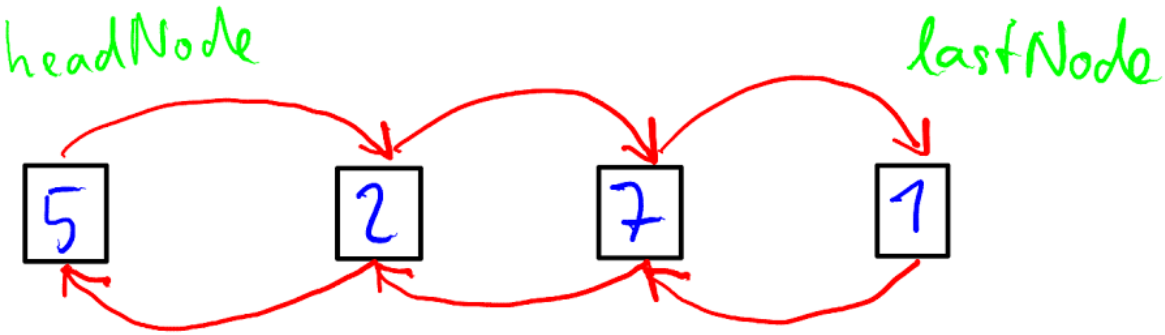
(Sortierte) Liste




```
public class SortedList {
  public ListNode headNode;

  private static class ListNode {
    private int value;
    private ListNode next;
    ...
  }
  ...
}
```

Doppelt verkettete Listen

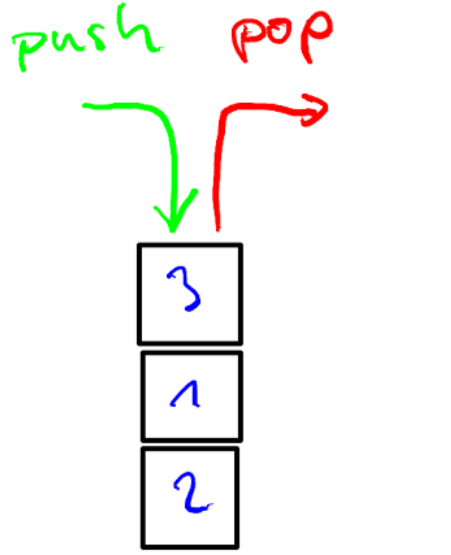



```
public class DoubleLinkedList {
  public ListNode headNode, lastNode;
  private static class ListNode {
    private int value;
    private ListNode prev, next;
    public void insertAfter(int value) { ... };
    public void deleteThis() { ... };
  }
  public void insertAfter(int pos, int value) { ... };
  public void deleteThis(ListNode node) { ... };
  public void append(DoubleLinkedList list) { ... };
}
```

Stacks und Queues

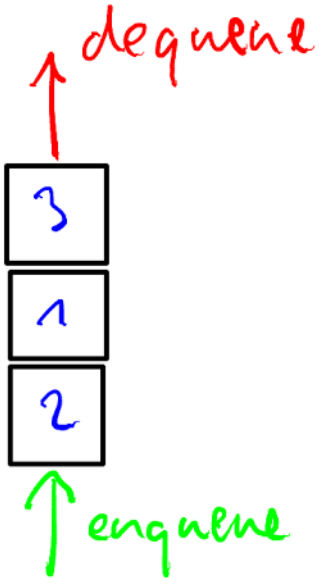
Stack (Stapel) - LIFO (last-in, first-out)

- `push(value)`: Wert `value` auf den Stapel
- `pop()`: Obersten Wert vom Stapel zurückgeben und entfernen
- `top()`: Obersten Wert vom Stapel betrachten
- `isEmpty()`: Ist der Stapel leer?



Queue (Warteschlange) - FIFO (first-in, first-out)

- `enqueue(value)`: Wert `value` an das Ende der Queue
- `dequeue()`: Wert an der Spitze der Queue zurückgeben und entfernen
- `head()`: Wert an der Spitze der Queue betrachten
- `isEmpty()`: Ist die Queue leer?



Wie kann man Stack und Queue mit einer doppelt verketteten Liste implementieren?

Aufgabe

- Implementieren Sie eine Queue unter Verwendung von zwei Stacks und keiner anderen Datenstrukturen!
- Analysieren Sie die Laufzeit der implementierten Operationen `enqueue(value)`, `dequeue()`, `head()`, `isEmpty()`.

Stacks

Aufgabe

Entwerfen Sie einen Algorithmus `ABC(w)`, der bei Eingabe eines Worts

$$w = a^k b^l c^m$$

für $k, l, m \geq 0$ testet, ob $k = l = m$.
Das Wort w ist als Stack aus Characters gegeben.

Mit welchen Datenstrukturen ist dies möglich?

Nutzen Sie außer Booleschen Werten und Charakters nur

- zwei weitere Stacks
- einen weiteren Stack

Rekursion und vollständige Induktion

Gegeben Sei der folgende Algorithmus:

Algorithmus M(n)

Input: $n \in \mathbb{N}$
Output: eine natürliche Zahl

```
1: if n = 0 then
2:     return n;
3: else
4:     return n · M(n − 1);
5: end if
```

1. Analysieren Sie die Laufzeit des Algorithmus.
2. Was berechnet der Algorithmus M bei Eingabe einer Zahl $n \in \mathbb{N}$? Beweisen Sie Ihre Aussage.

Vollständige Induktion

- Beweismethode für den Beweis einer Aussage $A(n)$ für alle $n \in \mathbb{N}$.

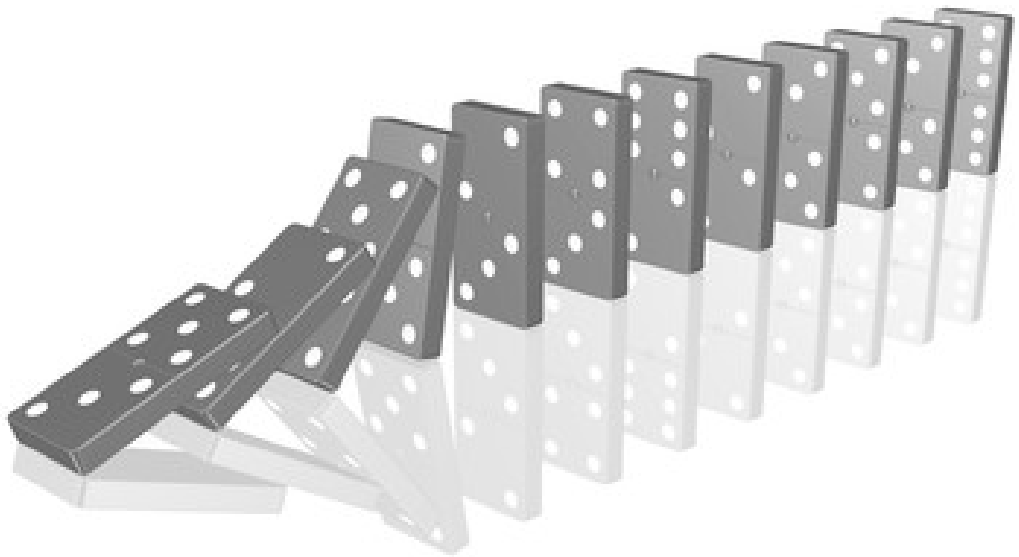
Induktionsanfang (I.A.)

Beweise, dass $A(0)$ wahr ist.

Induktionsschritt (I.S.)

- Induktionsvoraussetzung (I.V.): Die Aussage $A(n)$ ist wahr für ein $n \in \mathbb{N}$.
- Induktionsbehauptung (I.B.): Die Aussage $A(n+1)$ ist wahr.

Beweise, dass wenn (I.V.) wahr ist, auch (I.B.) wahr ist.



Rekursion und vollständige Induktion

Algorithmus P(A, i, j)

Input: Array A der Länge $|A| = n$ von Zahlen aus $\{0, 1, \ldots, 9\}$,
natürliche Zahlen i, j mit $1 \leq i \leq n$
Output: eine natürliche Zahl

```
1: if  i = 1  then
2:     return 10j + A[n];
3: else
4:     return P(A, i − 1, 10j + A[n−i+1]);
5: end if
```

- Was berechnet der Algorithmus **P** bei Eingabe des Arrays $[1, 2, 3]$ sowie $i = 3$ und $j = 0$?
- Was berechnet der Algorithmus **P** allgemein bei Eingabe eines Arrays A mit n Zahlen aus $\{0, 1, \ldots, 9\}$ sowie $i = n$ und $j = 0$?

Divide & Conquer

Idee

- Teile Problem in Teilprobleme auf (divide)
- Löse Teilprobleme rekursiv (conquer)
- Setze aus Lösungen für Teilprobleme die Lösung für das Gesamtproblem zusammen (merge)

Beispiele

- binäre Suche
- Quicksort, Mergesort
- Finden des Majority-Elements

Finden des Majority-Elements

Gegeben: Array A mit n Objekten

- Objekte können auf Gleichheit geprüft werden
- Objekte können nicht geordnet oder sortiert werden

Aufgabe:

Entscheide, ob es ein Element x in A gibt, dass mehr als $\frac{n}{2}$ mal vorkommt (formal: $H_x(A) > \frac{n}{2}$)

Beispiele

- $[2, 3, 7, 2, 5, 2, 9, 3, 3, 2, 2, 5, 2, 4, 2, 2]$ $\quad H_2(A_1) = 8 \leq \frac{n}{2}$
- $[2, 3, 7, 7, 2, 5, 2, 2, 3, 2, 2, 5, 2, 4, 2, 2]$ $\quad H_2(A_2) = 9 > \frac{n}{2}$
- $[2, 3, 3, 2, 3, 2, 3, 3, 3, 2, 2, 3, 2, 3, 2, 2]$ $\quad H_2(A_3) = H_3(A_3) = 8$

Naive Lösung

Algorithmus Major(A)

Input: Array A der Länge $n > 1$

Output: Majority-Element von A; ansonsten null

```
1:  if  n = 1  then
2:     return A[1];
3:  else
4:     for  i := 1 to ⌈n/2⌉ do
5:        c := 1 ;
6:        for  j := i + 1 to n  do
7:           if  A[i] = A[j]  then
8:              c := c + 1;
9:           end if
10:       end for
11:    end for
12:    if c > n/2 then
13:       return A[i];
14:    end if
15: end if
16: return null ;
```

Divide & Conquer-Lösung

Algorithmus MajorDC(A)

Input: Array A der Länge $n > 1$
Output: Majority-Element von A; ansonsten null

```
 1: if n = 1 then
 2:     return A[1];
 3: else
 4:     halbiere A in A_1 und A_2;
 5:     m_1 := MajorDC(A_1);
 6:     m_2 := MajorDC(A_2);
 7:     if m_1 = m_2 then
 8:         return m_1;
 9:     else if m_1 ≠ null then
10:         c := Anzahl Vorkommen von m_1 in A ;
11:         if c > n/2 then
12:             return m_1;
13:         end if
14:     else if m_2 ≠ null then
15:         c := Anzahl Vorkommen von m_2 in A ;
16:         if c > n/2 then
17:             return m_2;
18:         end if
19:     else
20:         return null ;
21:     end if
22: end if
```

Übung 4

Besprechung von Übungsblatt 1

Übung 5

Sortierverfahren

- Mergesort
- Quicksort
- Bucketsort
- Stabilität
- Untere Schranke für allgemeine Sortierverfahren

Mergesort

Aufgabe

Sortieren Sie das Array

$$A = [x, a, b, o, k, j, c, r, g]$$

mit dem Algorithmus Mergesort.

Algorithmus mergesort(A, l, r)

```
1: if l < r then
2:     m := ⌈(r − l)/2⌉
3:     mergesort(A, l, l + m)
4:     mergesort(A, l + m + 1, r)
5:     merge(A, l, l + m, r)
6: end if
```

Algorithmus merge(A, l, m, r)

```
1: B := array[1 . . . r − l + 1]
2: i := l
3: j := m + 1
4: k := 1
5: while i ≤ m and j ≤ r do
6:     if A[i] ≤ A[j] then
7:         B[k] := A[i]
8:         i := i + 1
9:     else
10:        B[k] := A[j]
11:        j := j + 1
12:    end if
13:    k := k + 1
14: end while
15: if i > m then
16:    copy A[j . . . r] to B[k . . . k + r − j]
17: else
18:    copy A[i . . . m] to B[k . . . k + m − i]
19: end if
20: copy B[1 . . . r − l + 1] to A[1 . . . r]
```

Quicksort

Aufgabe

Sortieren Sie das Array

$$A = [2, 10, 6, 7, 13, 4, 1, 12, 5, 9]$$

mit dem Algorithmus Quicksort.
(Pivotelement: das Element am rechten Rand des aktuellen Teilarrays)

Algorithmus quicksort(A, l, r)

```
1: if r > l then
2:     p := divide(A, l, r)
3:     quicksort(A, l, p − 1)
4:     quicksort(A, p + 1, r)
5: end if
```

Algorithmus divide(A, l, r)

```
1: v := A[r]
2: i := l
3: j := r − 1
4: repeat
5:     while A[i] ≤ v and i < r do
6:         i := i + 1
7:     end while
8:     while A[j] ≥ v and j > l do
9:         j := j − 1
10:    end while
11:    if i < j then
12:        swap(A[i], A[j])
13:    end if
14: until i ≥ j
15: swap(A[i], A[r])
16: return i
```

Bucketsort

Aufgabe

Sei $\Sigma = \{0, 1, 2, 3\}$ und $m = 3$. Sortieren Sie das Array

$$A = [103, 202, 101, 231, 022, 031, 030, 233, 201]$$

mit dem Algorithmus Bucketsort.

Algorithmus Bucketsort(A, m)

```
 1: B := Array leerer Queues mit |B| = |Σ|
 2: for i := m downto 1 do {die m-te Stelle ist die Stelle ganz rechts}
 3:    for j := 1 to |A| do
 4:       a := A[j]
 5:       k := findBucket(a, i)
 6:       B[k].enqueue(a)
 7:    end for
 8:    j := 1
 9:    for k := 1 to |B| do
10:       while not B[k].isEmpty() do
11:          A[j] := B[k].dequeue()
12:          j := j + 1
13:       end while
14:    end for
15: end for
16: return A
```

Linearsort

Algorithmus Linearsort(A)

Input: Array A von ganzen Zahlen aus $[1, z]$
Output: Array A aufsteigend sortiert

```
1:  B := Array der Länge z, initialisiert mit 0
2:  n := |A|
3:  C := Array der Länge n
4:  for i := 1 to n do
5:      B[A[i]] := B[A[i]] + 1
6:  end for
7:  for j := 2 to z do
8:      B[j] := B[j] + B[j − 1]
9:  end for
10: for i := 1 to n do
11:     C[B[A[i]]] := A[i]
12:     B[A[i]] := B[A[i]] − 1
13: end for
14: return C
```

Fragen

- Wie funktioniert Linearsort?
- Welche Laufzeit hat Linearsort?

Stabilität

- Wir sortieren Elemente $e = (e.\text{key}, e.\text{val})$, bestehend aus einem Wert $e.\text{val} \in \mathbb{V}$ und einem Schlüssel $e.\text{key} \in \mathbb{K}$ anhand ihres Schlüssels.
- Ein Sortieralgorithmus heißt stabil, wenn Elemente mit gleichem Schlüssel nach dem sortieren ihre relative Position zueinander behalten.

Beispiel

Eingabe: $[(3, c), (2, b), (3, a)]$

stabil: $[(2, b), (3, c), (3, a)]$

nicht stabil: $[(2, b), (3, a), (3, c)]$

Frage

Ist Linearsort stabil?

Linearsort (stabil)

Algorithmus LinearsortStabil(A)

Input: Array A von ganzen Zahlen aus $[1, z]$
Output: Array A aufsteigend sortiert

```
1:  B := Array der Länge z, initialisiert mit 0
2:  n := |A|
3:  C := Array der Länge n
4:  for i := 1 to n do
5:      B[A[i]] := B[A[i]] + 1
6:  end for
7:  for j := 2 to z do
8:      B[j] := B[j] + B[j − 1]
9:  end for
10: for i := n downto 1 do
11:     C[B[A[i]]] := A[i]
12:     B[A[i]] := B[A[i]] − 1
13: end for
14: return C
```

Untere Schranke der Worst-Case-Komplexität von allg. Sortierverfahren

Allgemeines Sortierverfahren

kann lediglich Elemente anhand ihres Schlüssels vergleichen.
Gegenbeispiel: Bucketsort ist kein allgemeines Sortierverfahren.

Untere Schranke für Anzahl der Vergleiche

$\Longrightarrow$ Untere Schranke für Zeitkomplexität

Array $[a_1, \ldots, a_n]$ der Länge n: wieviele Vergleiche sind mindestens nötig?

- es gibt $n!$ verschiedene Anordnungen (Permutationen) von $a_1, \ldots, a_n$
- Sortierverfahren muss alle $n!$ Permutationen unterscheiden (und zwar nur durch Vergleiche von Elementen!)
- Vergleiche führen zu einem Entscheidungsbaum
  - Form des Entscheidungsbaums von Sortierverfahren abhängig
  - In jedem Fall $n!$ Blätter (für jede Permutation eines)

Tiefe des Entscheidungsbaums $\cong$ Untere Schranke für Anzahl Vergleiche

Tiefe des Entscheidungsbaums

Beobachtung

- Entscheidungsbaum ist binär
- Jeder Binärbaum mit $n!$ Blättern hat Tiefe $\geq \log n!$

$$n! \geq \left(\frac{n}{2}\right)^{\frac{n}{2}} \implies \log n! \geq \log \left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2} \cdot \log \frac{n}{2} = \frac{n}{2} \cdot (\log n - 1)$$

$\implies$ Untere Schranke $\Omega(n \cdot \log n)$

Entscheidungsbaum für Insertionsort

Algorithmus InsertionSort(A)

```
1: for i := 2 to |A| do
2:     j := i
3:     key := A[j]
4:     while A[j − 1] > key and (j > 1) do
5:         A[j] := A[j − 1]
6:         j := j − 1
7:     end while
8:     A[j] := key
9: end for
```

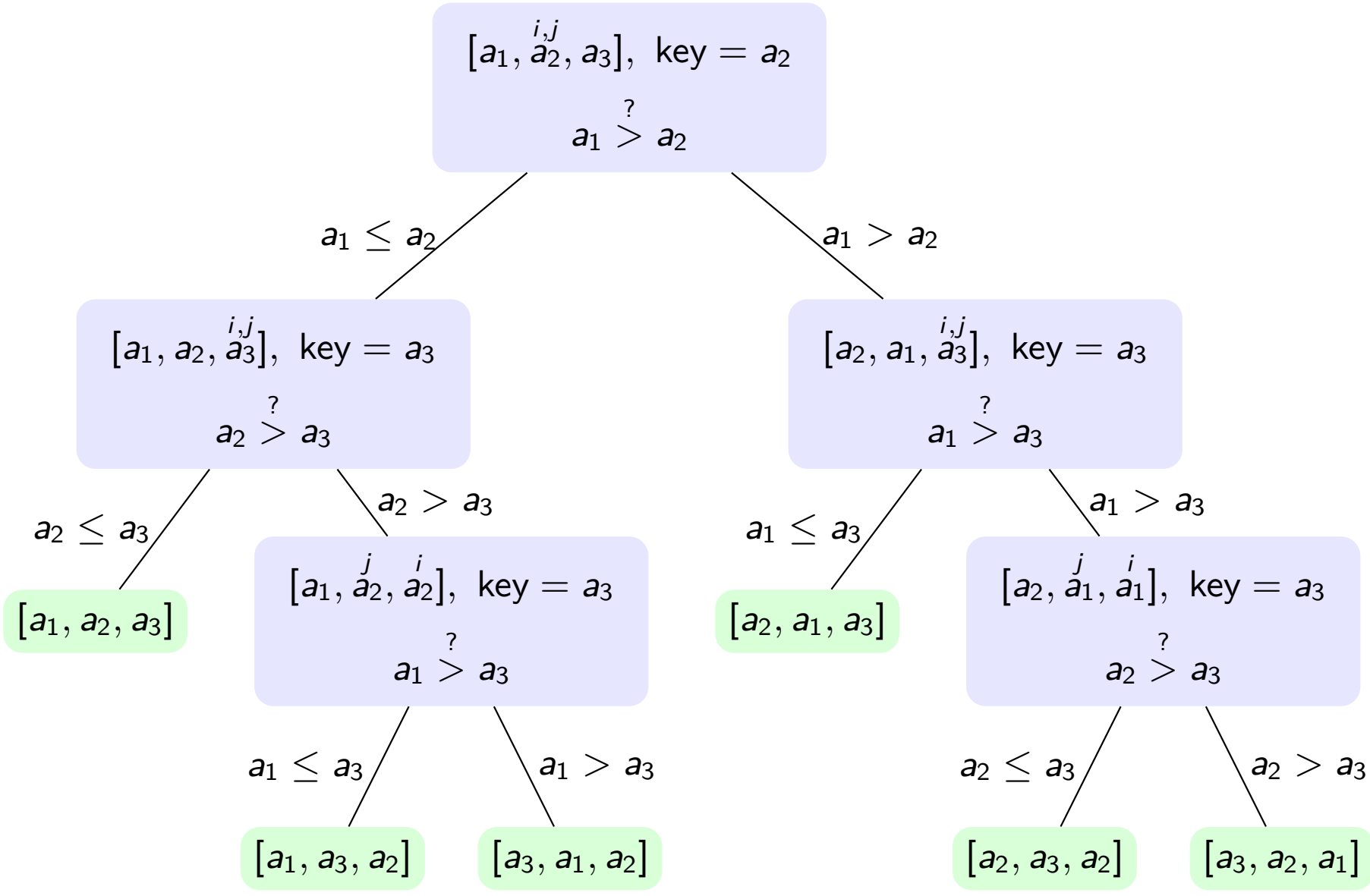
Eingabe: $A = [4, 2, 1, 3]$

Eingabe: $A = [a_1, a_2, a_3]$

a_1, a_2, a_3 sind beliebig

$\Longrightarrow$ **InsertionSort**(A) kann irgendeine Permutation von $[a_1, a_2, a_3]$ ausgeben!

Entscheidungsbaum für Insertionsort



Sortierung spezieller Arrays

Frage 1

Wir betrachten nur Arrays A, in denen ein Anfangsstück der Länge $\lceil \frac{3}{4} \cdot |A| \rceil$ bereits vorsortiert ist.

- Gibt es ein allgemeines Sortierverfahren, welches ein solches Array in Zeit $O(|A|)$ sortiert?

Frage 2

Wir betrachten Arrays, die jeweils nur natürliche Zahlen aus $1, \ldots, |A|$ enthalten.

- Gibt es ein (allgemeines) Sortierverfahren, welches ein solches Array in Zeit $O(|A|)$ sortiert?

Übung 7

- Suchverfahren
- Amortisierte Analyse

Suchverfahren

Der Eintrag 27 soll in dem Array

$$A = [2, 4, 5, 8, 12, 18, 27, 35, 40, 52, 68, 70]$$

gesucht werden. Verwenden Sie dazu die ...

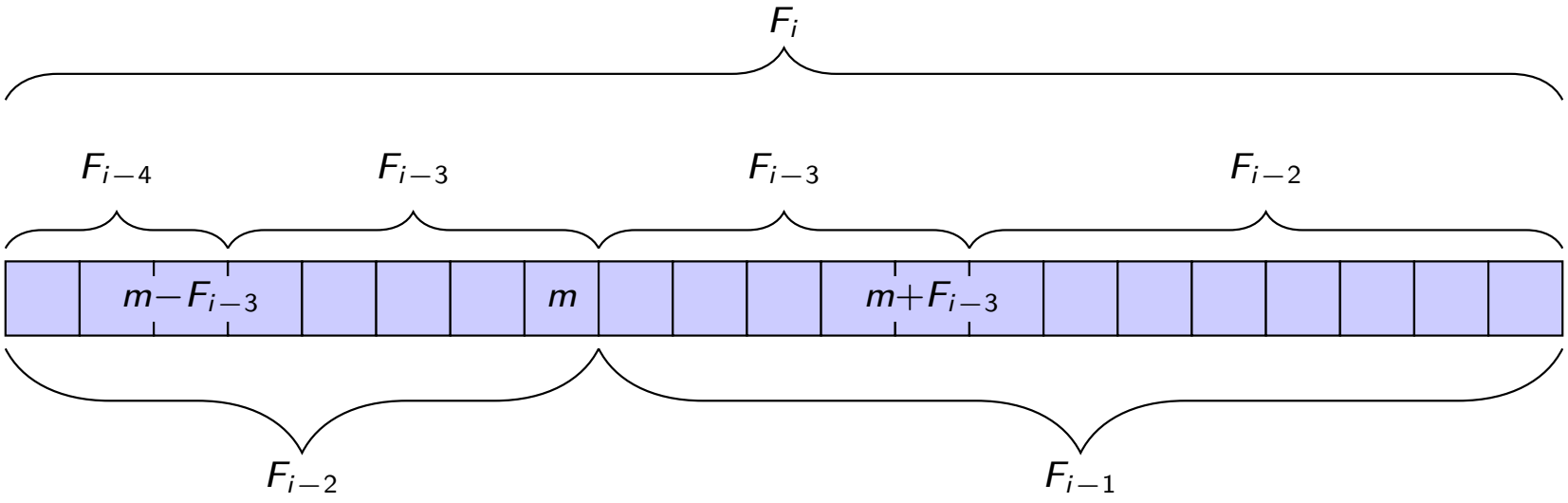
- Binäre Suche
- Fibonacci-Suche
- Interpolationssuche

Binäre Suche

Algorithmus BinSearch(A, c, l, r)

```
1:  if l > r then
2:      return false
3:  else
4:      m := l + ⌊(r − l)/2⌋
5:      if c < A[m] then
6:          return BinSearch(A, c, l, m − 1)
7:      else if c > A[m] then
8:          return BinSearch(A, c, m + 1, r)
9:      else
10:         return true
11:     end if
12: end if
```

Fibonacci-Suche



Erinnerung

$F_0 := 0, \quad F_1 := 1, \quad F_2 := 1 \quad$ und für $i > 2$,

$$F_i := F_{i-2} + F_{i-1}.$$

Fibonacci-Suche (vereinfacht)

Algorithmus FibonacciSearch(A, c, m, i)

```
if c > A[m] then
  if i = 3 then
    return false
  else
    return fibSearch(A, c, m + F_{i-3}, i − 1)
  end if
else if c < A[m] then
  if i = 4 then
    return false
  else
    return fibSearch(A, c, m − F_{i-3}, i − 2)
  end if
else
  return true
end if
```

Die Eingabe i ist das kleinste i, so dass $F_i > |A|$, und $m := F_{i-2}$.

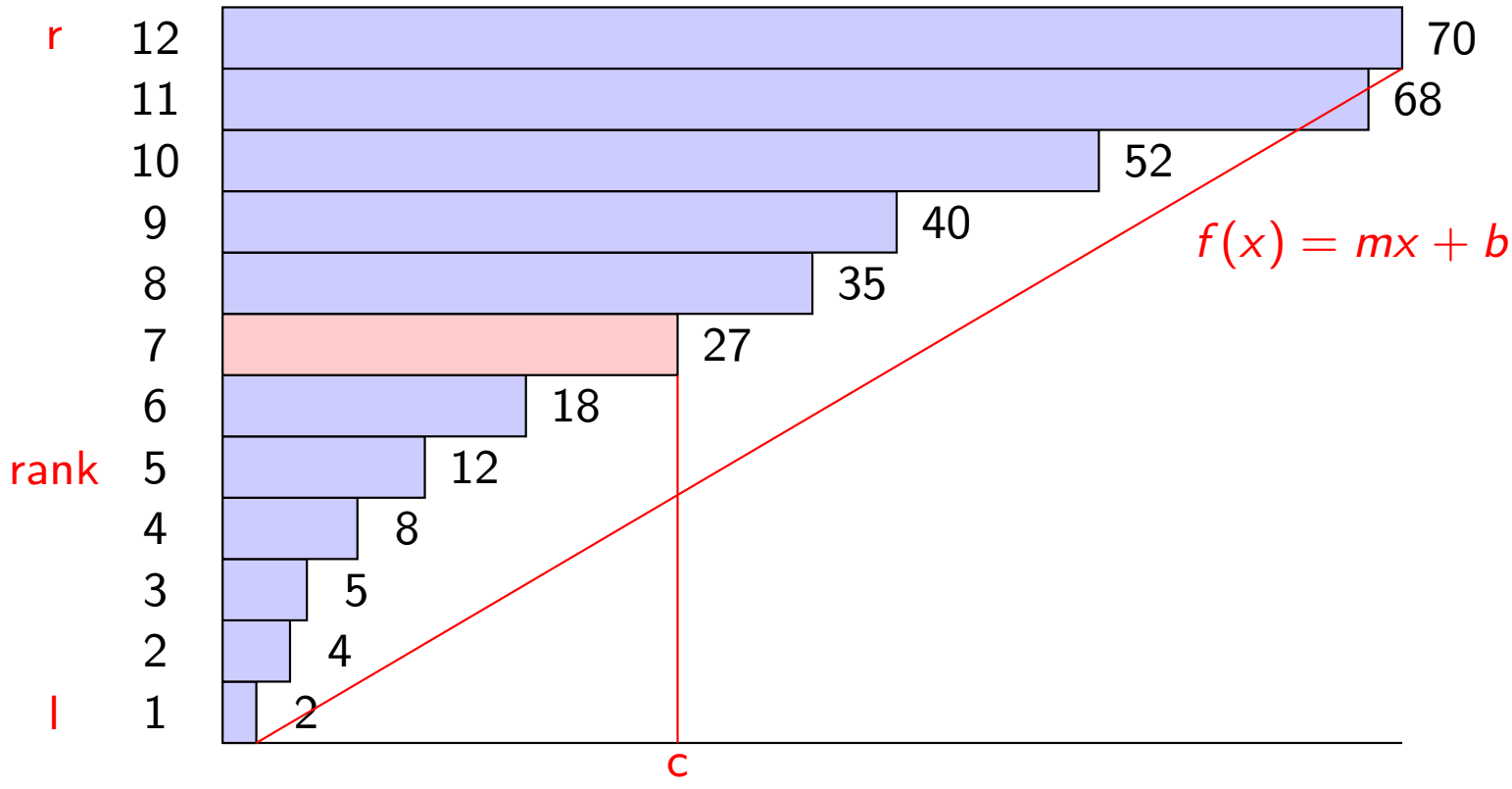
F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	$\cdots$
0	1	1	2	3	5	8	13	21	$\cdots$

Fibonacci-Suche

Algorithmus FibonacciSearch(A, c)

```
i := kleinstes i ≥ 1 so dass F_i > |A|
fib3 := F_{i-3};  fib2 := F_{i-2};  m := fib2
repeat
  if c > A[m] then
    if fib3 = 0 then
      return false
    else
      m := m + fib3;  tmp := fib3;  fib3 := fib2 − fib3;  fib2 := tmp
    end if
  else if c < A[m] then
    if fib2 = 1 then
      return false
    else
      m := m − fib3;  fib2 := fib2 − fib3;  fib3 := fib3 − fib2;
    end if
  else
    return true
  end if
until false
```

Interpolationssuche



$$\underbrace{\text{rank}}_{f(x)} = \left\lfloor \underbrace{\frac{r-l}{A[r]-A[l]}}_{m} \cdot \underbrace{(c - A[l])}_{x} + \underbrace{l}_{b} \right\rfloor$$

Binärzähler

Gegeben:

k-Bit Binärzähler, d.h. Array von Bits

$$[\ b_{k-1},\ b_{k-2}, \ldots, b_1,\ b_0\ ]$$

(entspricht Dezimalzahl $\sum_{i=0}^{k-1} b_i \cdot 2^i$)

- Operation: Zahl inkrementieren (um 1 erhöhen)
- Kosten: Anzahl der Bitänderungen (jedes Bit kostet 1)

n	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	Bitwechsel (Kosten)
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	2
3	0	0	0	0	0	0	1	1	1
4	0	0	0	0	0	1	0	0	3
5	0	0	0	0	0	1	0	1	1
6	0	0	0	0	0	1	1	0	2
7	0	0	0	0	0	1	1	1	1
8	0	0	0	0	1	0	0	0	4

Binärzähler – Kostenabschätzung

Gesucht:

Kosten für n Inkrement-Operationen (von 0 aus beginnend)

Kosten für eine Inkrement-Operation:

- Best Case?
- Worst Case?

Gesamtkosten:

?

Ist dies eine gute Abschätzung?

Binärzähler – Vergleich mit tatsächlichen Bitwechseln

n	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	$\sum$ Bitwechsel	$k \cdot n$
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1	8
2	0	0	0	0	0	0	1	0	3	16
3	0	0	0	0	0	0	1	1	4	24
4	0	0	0	0	0	1	0	0	7	32
5	0	0	0	0	0	1	0	1	8	40
6	0	0	0	0	0	1	1	0	10	48
7	0	0	0	0	0	1	1	1	11	56
8	0	0	0	0	1	0	0	0	15	64

Abschätzung ist zu pessimistisch!

Amortisierte Analyse

Amortisation: Anfängliche Aufwendungen für spätere Erträge

Gegeben:
Datenstruktur D, Folge Q von Operationen

Wann sollte amortisierte Analyse angewendet werden?

- Kosten aufeinanderfolgender Operationen schwanken stark
- teure Operationen erfordern vorangehende günstige

Ziel:
Bessere obere Schranke für Zeit- oder Speicherkomplexität im Worst Case.

Grundidee:
Betrachte amortisierte Kosten für Operationen

$$\sum_{\text{Operationen}} \text{reale Kosten} \quad \leq \quad \sum_{\text{Operationen}} \text{amortisierte Kosten}$$

Amortisierte Analyse – Verschiedene Perspektiven

Aggregatmethode

Obere Schranke für aufsummierte Gesamtkosten $T(n)$ von n Operationen

$$\text{amortisierte Kosten} = \frac{T(n)}{n}$$

Guthabenmethode

Operationen zahlen Guthaben auf "Konto" ein, welches später von anderen Operationen aufgebraucht werden kann

Potentialmethode

Potentialfunktion weist jedem Zustand der Datenstruktur ein Potential zu; Operationen verändern das Potential

Aggregatmethode

Beobachtung:

b_0 wechselt bei jedem Inkrement,
b_1 bei jedem zweiten Inkrement,
b_2 bei jedem vierten Inkrement, ... allgemein:

k-niedrigstes Bit ändert sich bei jedem 2^{k-1}ten Inkrement.

Aggregatmethode

Beobachtung:

b_0 wechselt bei jedem Inkrement,
b_1 bei jedem zweiten Inkrement,
b_2 bei jedem vierten Inkrement, ... allgemein:

k-niedrigstes Bit ändert sich bei jedem 2^{k-1}ten Inkrement.

Gesamtkosten für n Inkremente:

$$
\begin{aligned}
T(n) &= n + \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{4} \right\rfloor + \cdots + \left\lfloor \frac{n}{2^{k-1}} \right\rfloor \\
&\leq n \cdot \sum_{i=0}^{k-1} \frac{1}{2^i} \leq n \sum_{i=0}^{\infty} \frac{1}{2^i} = n \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i = n \cdot \frac{1}{1-\frac{1}{2}} \\
&= 2n \in O(n)
\end{aligned}
$$

Aggregatmethode

Beobachtung:

b_0 wechselt bei jedem Inkrement,
b_1 bei jedem zweiten Inkrement,
b_2 bei jedem vierten Inkrement, ... allgemein:

k-niedrigstes Bit ändert sich bei jedem 2^{k-1}ten Inkrement.

Gesamtkosten für n Inkremente:

$$\begin{aligned} T(n) &= n + \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{4} \right\rfloor + \cdots + \left\lfloor \frac{n}{2^{k-1}} \right\rfloor \\ &\leq n \cdot \sum_{i=0}^{k-1} \frac{1}{2^i} \leq n \sum_{i=0}^{\infty} \frac{1}{2^i} = n \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i = n \cdot \frac{1}{1-\frac{1}{2}} \\ &= 2n \in O(n) \end{aligned}$$

Amortisierte Kosten pro Operation:

$$\frac{T(n)}{n} \leq 2 \in O(1)$$

Potentialmethode

$D_1, D_2, D_3, \ldots$: Zustände der Datenstruktur nach Operation $1, 2, 3, \ldots$

Potentialfunktion $\Phi\colon D_i \to \mathbb{R}$

ordnet jedem Zustand der Datenstruktur ein Potential zu

Amortisierte Kosten:

$$d_i = \underbrace{c_i}_{\text{Reale Kosten}} + \underbrace{\Phi(D_i) - \Phi(D_{i-1})}_{\text{Potentialzuwachs}}$$

Amortisierte Gesamtkosten:

$$\sum_{i=1}^{n} d_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \Phi(D_n) - \Phi(D_0) + \sum_{i=1}^{n} c_i$$

Eigenschaften von Potentialfunktion:

- $\Phi(D_i)$ muss von Eigenschaft von D_i abhängen
- $\Phi(D_i) \geq \Phi(D_0)$
- d_i kann berechnet werden

$\to$ amortisierte Gesamtkosten $\geq$ tatsächliche Gesamtkosten

Potentialmethode: Binärzähler

Potentialfunktion $\phi(D_i)$: Anzahl der 1er im Binärzähler

Potentialmethode: Binärzähler

Potentialfunktion $\phi(D_i)$: Anzahl der 1er im Binärzähler

Reale Kosten:

In i-ter Operation springen t_i Bits auf 0 und ≤ 1 Bit auf 1

$$c_i \leq t_i + 1$$

Potentialmethode: Binärzähler

Potentialfunktion $\phi(D_i)$: Anzahl der 1er im Binärzähler

Reale Kosten:

In i-ter Operation springen t_i Bits auf 0 und ≤ 1 Bit auf 1

$$c_i \leq t_i + 1$$

Beobachtung:

$$\Phi(D_i) \leq \Phi(D_{i-1}) - t_i + 1 \qquad \text{für alle } i \geq 1$$

Potentialmethode: Binärzähler

Potentialfunktion $\phi(D_i)$: Anzahl der 1er im Binärzähler

Reale Kosten:

In i-ter Operation springen t_i Bits auf 0 und ≤ 1 Bit auf 1

$$c_i \leq t_i + 1$$

Beobachtung:

$$\Phi(D_i) \leq \Phi(D_{i-1}) - t_i + 1 \qquad \text{für alle } i \geq 1$$

Also gilt:

$$\begin{aligned} d_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq t_i + 1 + \Phi(D_{i-1}) - t_i + 1 - \Phi(D_{i-1}) \\ &\leq 2 \end{aligned}$$

$\Phi(D_i)$ hängt von einer Eigenschaft von D_i ab, $\Phi(D_i) \geq \Phi(D_0)$, und $d_i \leq 2$.

Potentialmethode: Binärzähler

Potentialfunktion $\phi(D_i)$: Anzahl der 1er im Binärzähler

Reale Kosten:

In i-ter Operation springen t_i Bits auf 0 und ≤ 1 Bit auf 1

$$c_i \leq t_i + 1$$

Beobachtung:

$$\Phi(D_i) \leq \Phi(D_{i-1}) - t_i + 1 \qquad \text{für alle } i \geq 1$$

Also gilt:

$$\begin{aligned} d_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq t_i + 1 + \Phi(D_{i-1}) - t_i + 1 - \Phi(D_{i-1}) \\ &\leq 2 \end{aligned}$$

$\Phi(D_i)$ hängt von einer Eigenschaft von D_i ab, $\Phi(D_i) \geq \Phi(D_0)$, und $d_i \leq 2$.

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} d_i \leq 2n \in O(n)$$