RUB

Vorlesung Datenstrukturen

Einleitung und Grundlagen

Maike Buchin

18.4.2017

Verantwortliche

RUB

Dozentin

- Maike Buchin
- Maike.Buchin@rub.de
- Raum NA 1/70
- Sprechzeiten: Termin per Mail vereinbaren

Organisation der Übungen

- Stef Sijben
- Stef.Sijben@rub.de
- Raum NA 1/71
- Sprechzeiten: Donnerstag 13 bis 14 Uhr

Übungsleiter

- Lars Schlieper, Stef Sijben

Korrekteure

- Lars Schlieper, Lea Thiel

Informationen

Webseite www.rub.de\lmi\lehre\ds_ss17\

Vorlesungen

- Di 14:15-15:45 Uhr, HNC 30
- Do 14:15-15:45 Uhr, HNC 30

Übungen

RUB

Termine
- Di 10-12 Uhr, NB 02/99, Stef Sijben
- Di 12-14 Uhr, NB 3/99, Stef Sijben
- Di 16-18 Uhr, NA 2/99, Lars Schlieper

  Anmeldung via Blackboard erforderlich!

Zettel
- Freitag online
- Dienstag Präsenzübung
- Dienstag Abgabe bis 10:00 Uhr
- Dienstag Rückgabe

Inhalt
- Lösung vorheriger Zettel
- Präsenzübung aktueller Zettel

Abgabe
- zu dritt
- Kästen auf NA 02 gegenüber von Raum 257

1. **Zettel** am Freitag 21.4. online
1. **Übung** am Dienstag 25.4.

Bewertung

Klausur: am Di 25.7. von 14 bis 16 Uhr
Nachklausur am Ende des WiSe 17/18

mündliche Prüfung für Hauptfach Mathe BSc.: am Di 25.7.
oder zu Beginn des WiSe 17/18

Bonuspunkte:

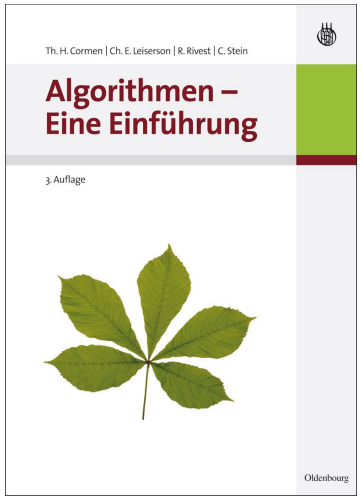
10 % Punkte auf den Übungszetteln = 1 Punkt auf der Klausur
bzw. 60% = -0,3 in der mündlichen Prüfung

Bonuspunkte zählen nur in der ersten Klausur!

Literatur

Dietzfelbinger, Mehlhorn, Sanders:
Algorithmen und Datenstrukturen - Die Grundwerkzeuge

über den OPAC der RUB innerhalb des Campusnetzes als pdf erhältlich

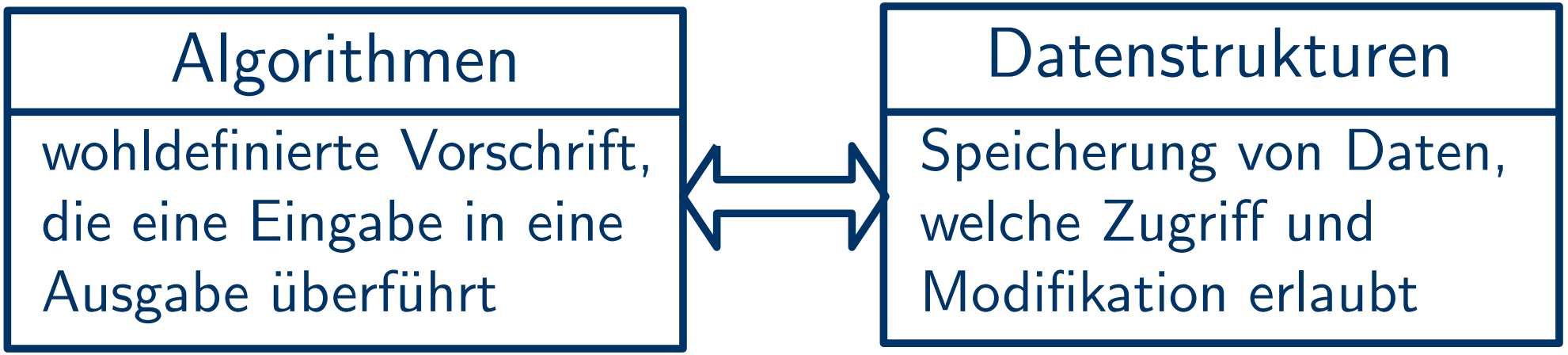


Cormen, Leiserson, Rivest, Stein:
Algorithmen - Eine Einführung

.. und viele weitere gute Bücher zu Algorithmen und Datenstrukturen

Inhalt des Kurses

RUB



- Korrektheit
- Laufzeit
- Speicherbedarf

Effizienz

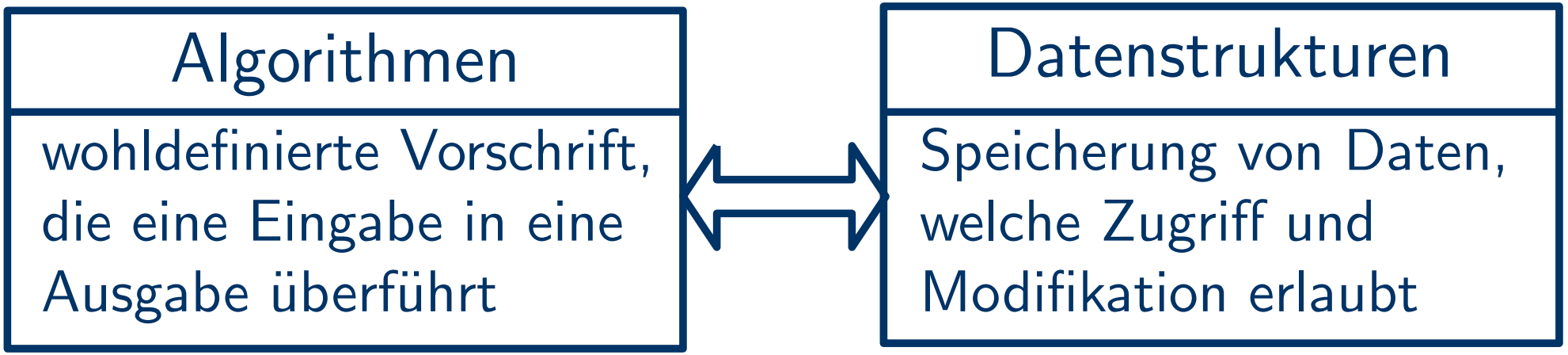
- Aufbau
- Speicherbedarf
- Operationen

Wir interessieren uns v.a. für die theoretische Analyse dieser!

Die praktische Implementierbarkeit kennen Sie ggfs. aus anderen Veranstaltungen (Informatik 2, Einführung in die Programmierung).

Inhalt des Kurses

RUB



Wir werden sehen:

- grundlegende Datenstrukturen
- grundlegende Algorithmenparadigmen

Themen:

- Einleitung
- Suchen & Sortieren
- Graphen

2. Einleitung [Grundlagen]

2.1 Asymptotische Notation

RUB

Laufzeitanalyse von Algorithmen:

- Anzahl von elementaren Rechenschritten
- Laufzeit in Abhängigkeit der Größe der Eingabe

  $\text{time}(I) = \#$ Rechenschritte eines Algorithmus auf einer Eingabe I

  Analyse im schlechtesten Fall:

$$T(n) = \max\{\text{time}(I) \mid \text{size}(I) = n\}$$

  Alternativ:
  Analyse im besten Fall:

$$T(n) = \min\{\text{time}(I) \mid \text{size}(I) = n\}$$

  Analyse im mittleren Fall:

$$T(n) = \frac{1}{|\{I|\text{size}(I) = n\}|} \sum_{\{I|\text{size}(I)=n\}} \text{time}(I)$$

- Rechenschritte asymptotisch zählen mit der Oh-Notation

Oh-Notation

$$O(f(n)) = \{g(n) \mid \exists c > 0\ \exists n_0 \geq 1\ \forall n > n_0 : g(n) \leq c \cdot f(n)\}$$

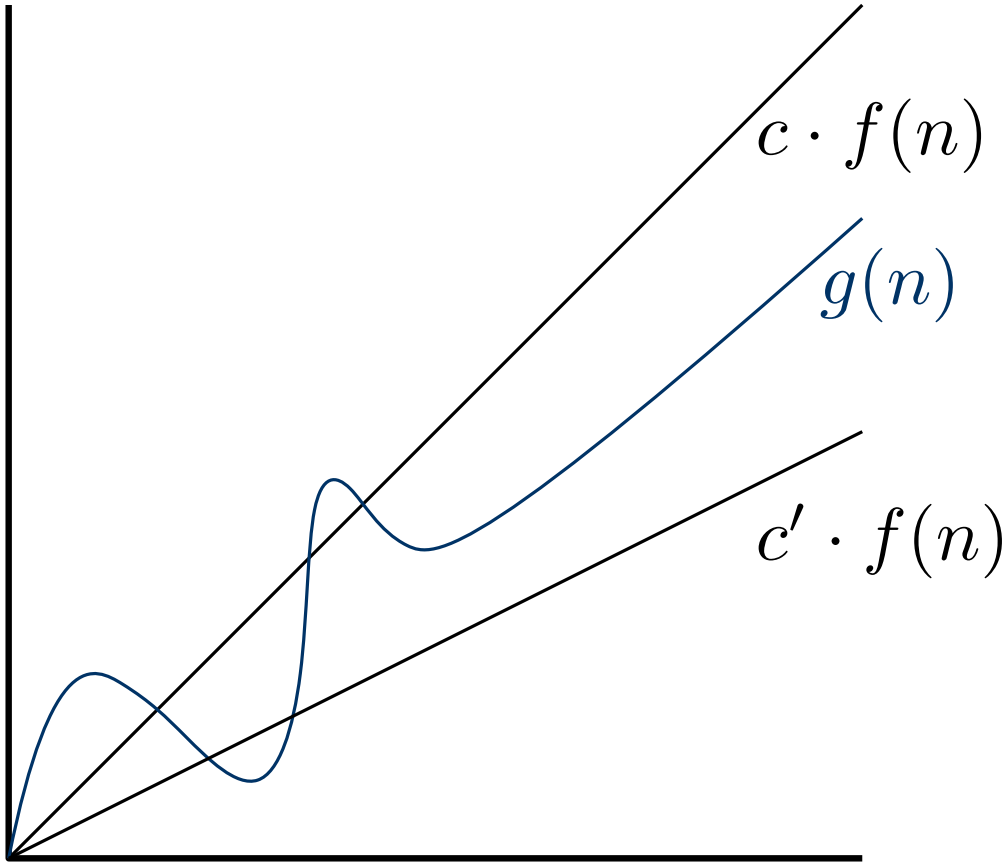
$$\Omega(f(n)) = \{g(n) \mid \exists c > 0\ \exists n_0 \geq 1\ \forall n > n_0 : g(n) \geq c \cdot f(n)\}$$

$$\Theta(f(n)) = \Omega(f(n)) \cap O(f(n))$$

Notation: $g(n) = O(f(n))$ statt $g(n) \in O(f(n))$

Lemma:

Sei $p(n) = \sum_{i=0}^{k} a_i n^i$ ein Polynom mit reellen Koeffizienten a_i und $a_k > 0$. Dann ist $p(n) = \Theta(n^k)$.



Oh-Notation

RUB

$$O(f(n)) = \{g(n) \mid \exists c > 0\ \exists n_0 \geq 1\ \forall n > n_0 : g(n) \leq c \cdot f(n)\}$$

$$\Omega(f(n)) = \{g(n) \mid \exists c > 0\ \exists n_0 \geq 1\ \forall n > n_0 : g(n) \geq c \cdot f(n)\}$$

$$\Theta(f(n)) = \Omega(f(n)) \cap O(f(n))$$

Notation: $g(n) = O(f(n))$ statt $g(n) \in O(f(n))$

Lemma:

- $cf(n) = \Theta(f(n))\ \forall c > 0$
- $f(n) + g(n) = \Omega(f(n))$
- $f(n) + g(n) = O(f(n))$ falls $g(n) = O(f(n))$
- $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$

Beispiele:

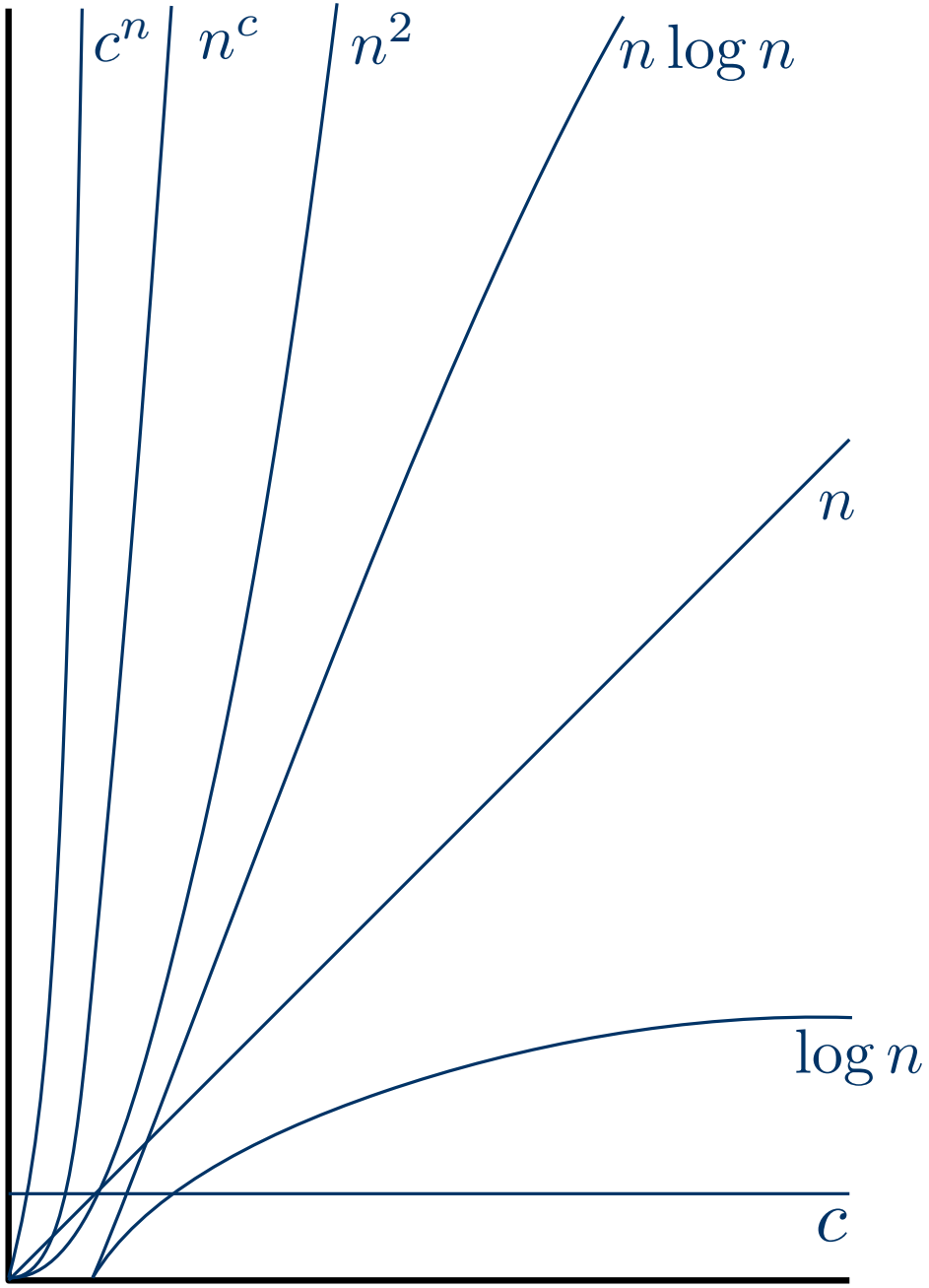
- $7n^2 + 10^6 n \quad = O(n^2)$
- $3n \log n + 5n = \Omega(n)$
- $99n + 3 \log n \ = O(n)$

Oh-Notation

RUB

Wichtige Klassen:

- $O(c)$ konstant
- $O(\log n)$ logarithmisch
- $O(n)$ linear
- $O(n \log n)$ loglinear
- $O(n^2)$ quadratisch
- $O(n^c)$ polynomiell
- $O(c^n)$ exponentiell



2.2 Maschinenmodell

RUB

RAM = random access machine (dt. Registermaschine)

Einfaches Maschinenmodell eines sequentiellen Rechners

- CPU – Recheneinheit
- Register – endlich viele Register $R_1, \ldots, R_k$
- Speicher – unendlich viele Speicherzellen $S[0], S[1], \ldots$

Speicherzellen und Register enthalten *'kleine' ganze Zahlen*, d.h. die Größe einer Zahl ist beschränkt durch ein Polynom in der Größe der Eingabe

- die Anzahl der bits zur Darstellung einer Zahl ist also logarithmisch in der Größe der Eingabe
- Zeit und Speicherbedarf wachsen also maximal um einen log-Faktor

2.2 Maschinenmodell

RUB

RAM = random access machine (dt. Registermaschine)

Einfaches Maschinenmodell eines sequentiellen Rechners

- CPU – Recheneinheit
- Register – endlich viele Register $R_1, \ldots, R_k$
- Speicher – unendlich viele Speicherzellen $S[0], S[1], \ldots$

Speicherzellen und Register enthalten *'kleine' ganze Zahlen*, d.h. die Größe einer Zahl ist beschränkt durch ein Polynom in der Größe der Eingabe

Ohne diese Annahme wäre es z.B. möglich, durch n Quadrierungen die Zahl 2^{2^n} zu berechnen.

Alternativ verwendet man das *logarithmische Kostenmaß* statt dem *uniformen Kostenmaß*.

RAM-Programm

RUB

besteht aus durchnummerierter Liste von **Befehlen**

- Laden/Schreiben von Registern aus/in den Speicher
- Operationen auf Registern
  - arithmetisch $(+, -, \times,$ div, mod$)$
  - Vergleiche $(\leq, <, >, \geq)$
  - logisch $(\wedge, \vee, \neg)$
- Sprung-Anweisungen
  - bedingt `if` $R_i = 0$ `jump to k`
  - unbedingt `jump to k`

Ein **Programm**

- startet in Zeile 1
- falls kein Sprung, gehe zur nächsten Zeile
- endet falls Zeile ausserhalb Programm
- Ein- und Ausgabe stehen in festgelegten Speicherzellen

RAM-Programm

RUB

Beispiel: finde das Minimum von zwei Zahlen

Eingabe: x, y in $S[0], S[1]$
Ausgabe: $\min(x, y)$ in $S[0]$

1. $R_1 = S[0]$
2. $R_2 = S[1]$
3. $R_3 = R_1 \leq R_2$
4. if $R_3 = 0$ jump to 6
5. $S[0] = R_2$

Maschinenmodell

RUB

Zeitkomplexität:

ein Rechenschritt entspricht einem Maschinenbefehl

RAM vs. realer Rechner:

- endlicher Speicher
- feste Anzahl bits
- nicht jeder Befehl verursacht gleiche Kosten

Bezug zur Berechenbarkeitstheorie:

- RAM führt im Prinzip GOTO-Programm aus
- Turingmaschinen besser geeignet für untere Schranken
- RAM besser geeignet für obere Schranken

2.3 Pseudocode

RUB

- besser geeignet zur Beschreibung von Algorithmen als RAM-Programme
- Abstraktion und Vereinfachung imperativer Programmiersprachen

Enthält

- Variablen verschiedener Typen (Zahlen, Folgen, Mengen, ...)
- Zuweisungen und Schleifen
  - `if .. then .. else ..`
  - `while .. do ..`
- Prozeduren und Funktionen als Unterprogramme

Zeitkomplexität:

Elementare Pseudocode-Befehle benötigen konstante Zeit; Prozedur- und Funktionsaufrufe benötigen konstante Zeit plus die Zeit für die Ausführung ihres Rumpfes.

Pseudocode

Beispiel: Sieb des Eratosthenes

$a = \langle 1, \ldots, 1 \rangle : Array\ [2..n]\ \textbf{of}\ \{0,1\}$
// Referenzparameter; am Ende: $a[i] = 1 \Leftrightarrow i$ ist Primzahl
for $i := 2$ **to** $\lfloor \sqrt{n} \rfloor$ **do**
  if $a[i]$ **then for** $j := 2i$ **to** n **step** i **do** $a[j] := 0$
    // Wenn $a[i] = 1$, ist i prim, Vielfache von i dagegen nicht
for $i := 2$ **to** n **do** **if** $a[i]$ **then** output("$\boxed{i}$ ist Primzahl")

Eigenschaften:

- kompakt und genau
- Abstraktion erlaubt auch Details zu "vertuschen"

2.4 Korrektheit von Algorithmen

RUB

Korrektheit lässt sich häufig zeigen mit Hilfe von:

- **Zusicherungen:** Vor- und Nachbedingungen die während der Ausführung eines Programmes gelten
- **Schleifeninvarianten:** Eigenschaften, die vor und nach jedem Schleifendurchlauf gelten
- **Datenstrukturinvarianten:** Eigenschaften, die unmittelbar nach Konstruktion gelten, sowie Vor- und Nachbedingung für alle Operationen darauf sind

Function $power(a : \mathbb{R}; n_0 : \mathbb{Z}) : \mathbb{R}$
  assert $n_0 \geq 0$ // negative Exponenten können nicht bearbeitet werden
  $p = a : \mathbb{R}; \quad r = 1 : \mathbb{R}; \quad n = n_0 : \mathbb{N}$ // es gilt $p^n r = a^{n_0}$
  while $n > 0$ **do**
    invariant $p^n r = a^{n_0}$
    if n *ist ungerade* **then** $n--; r := r \cdot p$ // zwischen Zuweisungen ist Invariante verletzt
    else $(n, p) := (n/2, p \cdot p)$ // parallele Zuweisung erhält Invariante aufrecht
  assert $r = a^{n_0}$ // Folgerung aus der Invarianten und $n = 0$
  return r

2.4 Korrektheit von Algorithmen

RUB

Korrektheit lässt sich häufig zeigen mit Hilfe von:

- **Zusicherungen:** Vor- und Nachbedingungen die während der Ausführung eines Programmes gelten
- **Schleifeninvarianten:** Eigenschaften, die vor und nach jedem Schleifendurchlauf gelten
- **Datenstrukturinvarianten:** Eigenschaften, die unmittelbar nach Konstruktion gelten, sowie Vor- und Nachbedingung für alle Operationen darauf sind
- **Zertifikate** zur Überprüfung von Zusicherungen.
  Z.B. ist ein Teiler einer Zahl ein Zeuge dafür, dass diese nicht prim ist.

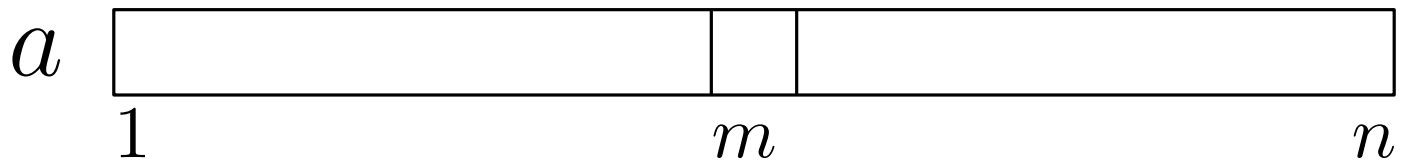
2.5 Binäre Suche

RUB

effiziente Art, um in einer geordneten Menge zu suchen

Gegeben: ein geordnetes Array $a[1 \ldots n]$ mit paarweise verschiedenen Einträgen, d.h. $a[1] < \ldots < a[n]$, sowie ein Element x.
Gefragt: Ist x in A und der Index k mit $a[k-1] < x \leq a[k]$.
Dabei fassen wir $a[0] = -\infty$ und $a[n+1] = \infty$ auf.

Teile-und-Herrsche Prinzip:



Vergleiche x und $a[m]$

- falls $x = a[m] \rightarrow$ gebe m zurück
- falls $x < a[m] \rightarrow$ suche weiter in $a[1 \ldots m-1]$
- falls $x > a[m] \rightarrow$ suche weiter in $a[m+1 \ldots n]$

2.5 Binäre Suche

Realisierung dieser Idee mit zwei Zeigern ℓ und r für die gilt

Invariante: $0 \leq \ell < r \leq n+1$ und $a[\ell] < x < a[r]$ (I)

$(\ell, r) := (0, n+1)$
while *true* **do**
  invariant (I) // d. h. Invariante (I) gilt hier
  if $\ell + 1 = r$ **then return** ("$(a[\boxed{\ell}] < x < a[\boxed{\ell+1}]$")
  $m := \lfloor (r+\ell)/2 \rfloor$ // $\ell < m < r$
  $s := compare(x, a[m])$ // -1 falls $x < a[m]$, 0 falls $x = a[m]$, $+1$ falls $x > a[m]$
  if $s = 0$ **then return** ("x steht in $a[\boxed{m}]$")
  if $s < 0$
    then $r := m$ // $a[\ell] < x < a[m] = a[r]$
    else $\ell := m$ // $a[\ell] = a[m] < x < a[r]$

Korrektheit: folgt aus der Invariante

Laufzeit: ist $O(\log n)$, denn

- in jedem Durchlauf (ausser dem letzten) halbiert sich die Größe des zu durchsuchenden Arrays
- ein Durchlauf benötigt konstante Zeit