

Datenstrukturen und Algorithmen

SS 2008

Robert Elsässer

Organisatorisches

Vorlesung:

- Mo 11:15 - 12:45
- Fr 11:15 - 12:45

Zentralübung:

- Do 13:00 – 13:45
- Beginn: nächste Woche

Übungen:

- Beginn: diese Woche Donnerstag

Organisatorisches

Heimübungen:

- Jede Woche ein Übungsblatt
- 1. Blatt Donnerstag, diese Woche
- Abgabe: Donnerstag bis 10:00 Uhr, D3-Flur
- Bonuspunkte
 - 50% - 75% : 1/3 Note
 - 75% - 90% : 2/3 Note
 - > 90% : 1.0 Note
- Erste Abgabe: nächste Woche
- Erste bewertete Abgabe: übernächste Woche
- Vorstellung *einer* Musterlösung in der Zentralübung

Organisatorisches

Übungsgruppen:

- StudInfo (erreichbar über www.upb.de/cs/elsa/DuA.htm)
 - IMT-Login und Password
 - Bis zu 30 Teilnehmer in einer Gruppe
- Freischaltung heute um 13:00 Uhr
- Insgesamt 14 Gruppen
- Präsenzübungen

Organisatorisches

Klausur:

- Eine Korrelation mit den Übungsaufgaben ist zu erwarten
- Es gab in der Vergangenheit einen direkten Zusammenhang zwischen Übungsteilnahme/abgabe und gutem Abschneiden bei Klausuren

Sprechzeiten:

- Do 16:15 – 17:00 (Raum F2.315; Fürstenallee)
- Fragen/Anmerkungen zunächst an den **eigenen Tutor**
- Webseite der Vorlesung: www.upb.de/cs/elsa/DuA.htm

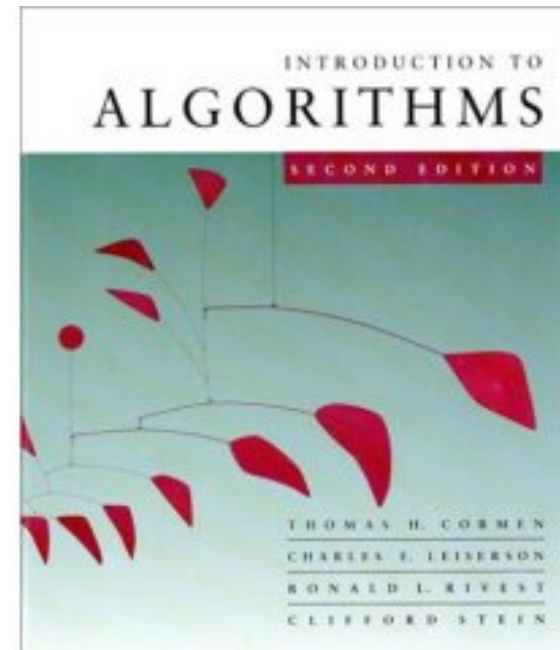
Organisatorisches

Literatur:

Cormen, Leiserson, Rivest, Stein:
Introduction to Algorithms, 2nd ed.

MIT Press/McGraw-Hill

ISBN 0-262-53196-8



1. Einführung

- Was ist ein Algorithmus (eine Datenstruktur)?
- Welche Probleme kann man damit lösen?
- Warum betrachten wir (effiziente) Algorithmen?
- Wie beschreiben wir Algorithmen?
- Nach welchen Kriterien beurteilen wir Algorithmen?
- Welche Algorithmen betrachten wir?
- Wie passt *Datenstrukturen und Algorithmen* ins Informatikstudium?

Was ist ein Algorithmus?

Definition 1.1: Ein *Algorithmus* ist eine eindeutige Beschreibung eines Verfahrens zur Lösung einer bestimmten Klasse von Problemen.

Genauer: Ein Algorithmus ist eine Menge von Regeln für ein Verfahren, um aus gewissen Eingabegrößen bestimmte Ausgabegrößen herzuleiten. Dabei muss

1. Das Verfahren in einem endlichen Text beschreibbar sein.
2. Jeder Schritt des Verfahrens auch tatsächlich ausführbar sein.
3. Der Ablauf des Verfahrens zu jedem Zeitpunkt eindeutig definiert sein.

Beispiel Sortieren

Eingabe bei Sortieren: Folge von n Zahlen (a_1, a_2, \dots, a_n) .

Ausgabe bei Sortieren: Umordnung (b_1, b_2, \dots, b_n) der Eingabefolge, so dass $b_1 \leq b_2 \leq \dots \leq b_n$.

Sortieralgorithmus: Verfahren, das zu jeder Folge (a_1, a_2, \dots, a_n) sortierte Umordnung (b_1, b_2, \dots, b_n) berechnet.

Eingabe : (31,41,59,26,51,48)

Ausgabe : (26,31,41,48,51,59)

Was ist eine Datenstruktur?

Definition 1.2: Eine *Datenstruktur* ist eine bestimmte Art Daten im Speicher eines Computers anzuordnen, so dass Operationen wie z.B. *Suchen*, *Einfügen*, *Löschen* einfach zu realisieren sind.

Einfache Beispiele:

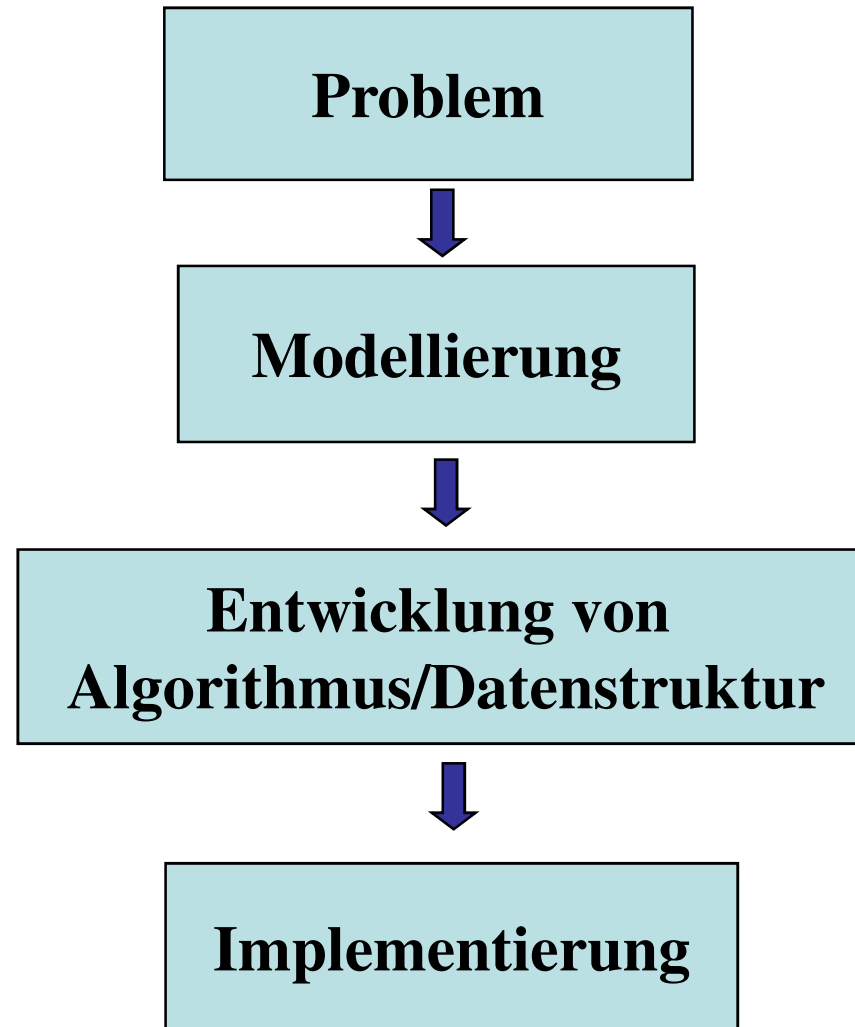
- *Listen*
- *Arrays*

Datenstrukturen und Algorithmen sind eng verbunden:
Gute Datenstrukturen häufig unerlässlich für gute Algorithmen.

Beispielprobleme

- Wie findet ein Navigationssystem gute Verbindungen zwischen zwei Orten? Wie werden im Internet Informationen geroutet?
- Wie berechnet ein Unternehmen eine möglichst gute Aufteilung seiner Ressourcen, um seinen Gewinn zu maximieren?
- Wie werden etwa in Google Informationen schnell gefunden?
- Wie werden Gleichungssysteme der Form $Ax=b$ gelöst?

Softwareentwicklung



Kriterien für Algorithmen

- Algorithmen müssen korrekt sein.
 - ⇒ Benötigen *Korrektheitsbeweise*.
- Algorithmen sollen *Zeit- und Speichereffizient* sein.
 - ⇒ Benötigen Analysemethoden für Zeit-/Speicherbedarf.
- Analyse basiert *nicht auf* empirischen Untersuchungen, sondern auf mathematischen Analysen. Nutzen hierfür *Pseudocode* und *Basisoperationen*.

Algorithmenentwurf

Zum Entwurf eines Algorithmus gehören

1. *Beschreibung des Algorithmus*
2. *Korrektheitsbeweis*
3. *Zeit- bzw. Speicherbedarfsanalyse.*

Beschreibung von Algorithmen

- Zunächst informell (mathematisches) Verfahren zur Lösung.
- Dann Präzisierung durch *Pseudocode*.
- *Keine* Beschreibung durch Programmcode in Java oder C, C++.

Algorithmenentwurf

Warum mathematische Korrektheitsbeweise?

- Fehler können fatale Auswirkungen haben (Steuerungssoftware in Flugzeugen, Autos, AKWs)
- Fehler können selten auftreten („Austesten“ funktioniert nicht)

Der teuerste algorithmische Fehler?

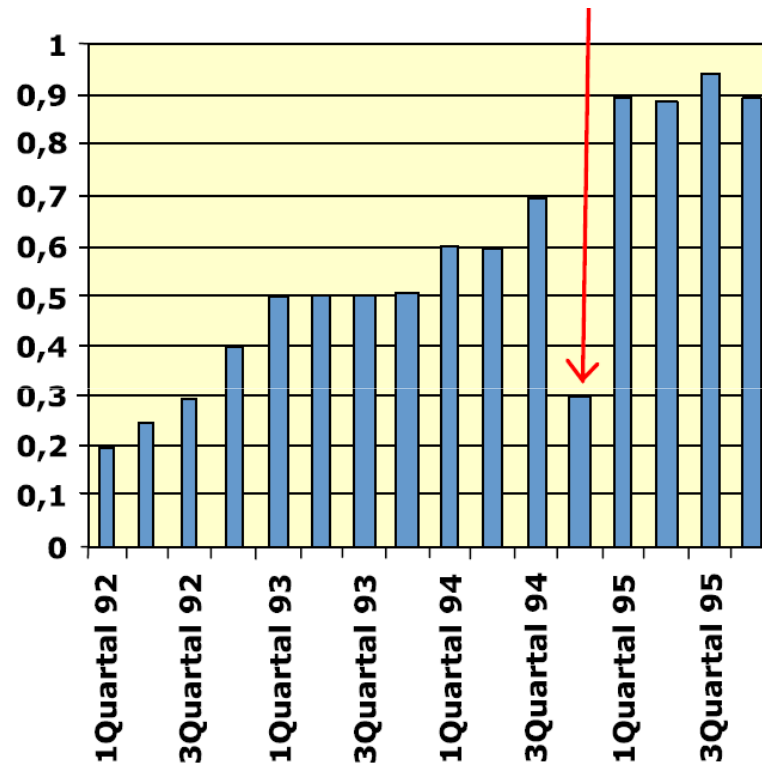
- Pentium bug (\approx \$500 Mio.)
- Enormer Image Schaden
- Trat relativ selten auf

Algorithmenentwurf

Die Berechnung von
 $z = x - (x/y)y$
sollte $z=0$ ergeben.

Der fehlerhafte
Pentium-Prozessor
berechnete statt dessen
mit den Werten
 $x=4195835$ und
 $y=3145727$
den Wert $z=256$

\$480 Mio. entgangener Gewinn



Quartalsergebnisse der Firma Intel

Wozu gute Algorithmen?

- Computer werden zwar immer schneller und Speicher immer größer und billiger, aber
- Geschwindigkeit und Speicher werden immer begrenzt sein.
- Daher muss mit den Ressourcen *Zeit* und *Speicher* geschickt umgegangen werden.
- Außerdem werden in Anwendungen immer größere Datenmengen verarbeitet. Diese wachsen schneller als Geschwindigkeit oder Speicher von Computern.

Effekt guter Algorithmen - Sortieren

Insertion - Sort :

Sortiert n Zahlen mit $c_1 n^2$ Vergleichen.

Computer A :

10^9 Vergleiche/Sek.

$c_1 = 2, n = 10^6$.

Dann benötigt A

$$\frac{2 \cdot (10^6)^2}{10^9} = 2000 \text{ Sek.}$$

Merge - Sort :

Sortiert n Zahlen mit $c_2 n \log(n)$ Vergleichen.

Computer B :

10^7 Vergleiche/Sek.

$c_2 = 50, n = 10^6$.

Dann benötigt B

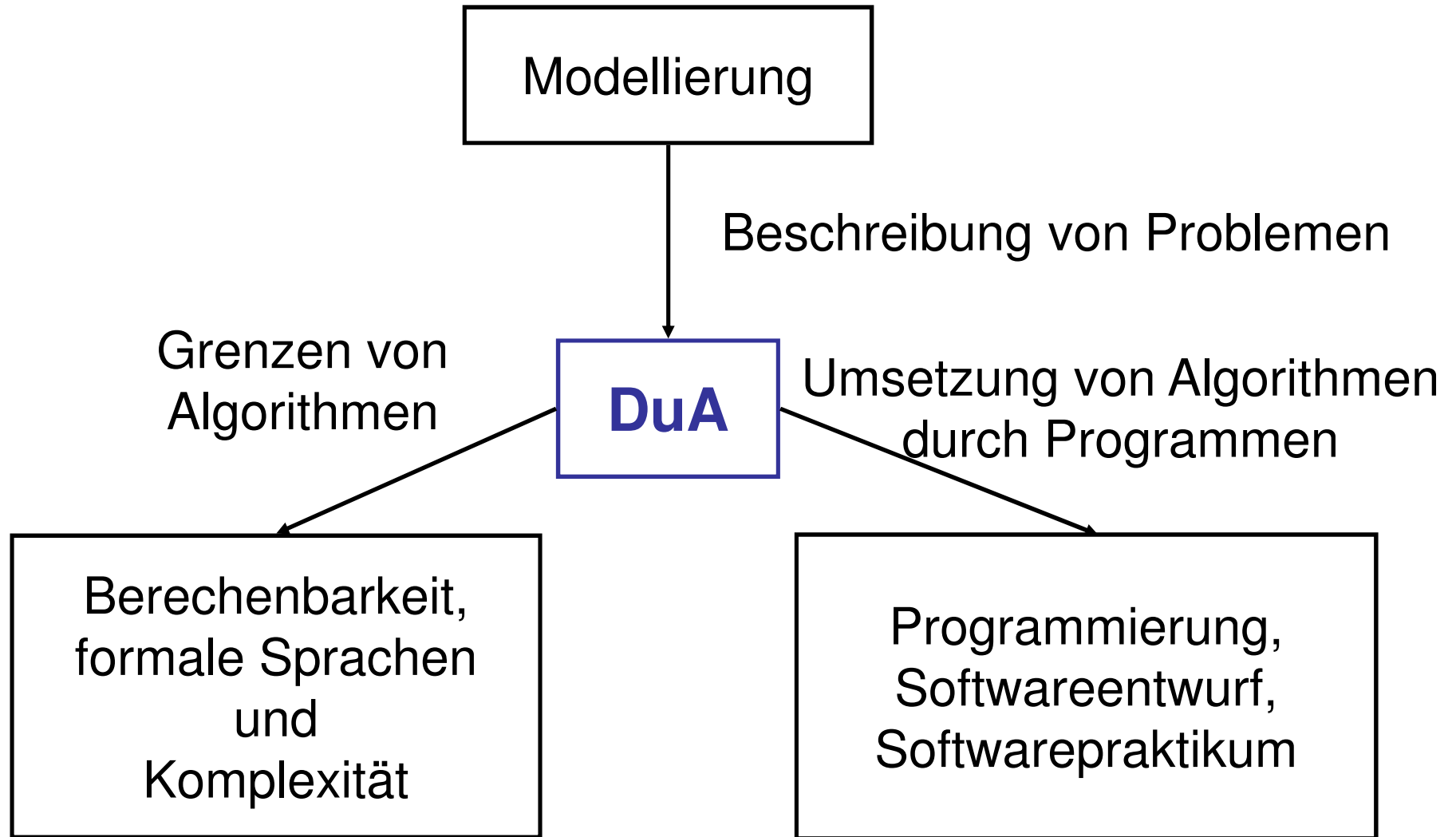
$$\frac{50 \cdot (10^6) \log(10^6)}{10^7} \approx 100 \text{ Sek.}$$

Probleme, Algorithmen, Ziele

Ziel der Vorlesung ist es

1. Wichtige Probleme, Algorithmen und Datenstrukturen kennen zu lernen.
2. Wichtige Algorithmentechniken und Entwurfsmethoden kennen und *anwenden* zu lernen.
3. Wichtige Analysemethoden kennen und *anwenden* zu lernen.
4. Das Zusammenspiel von Datenstrukturen und Algorithmen zu verstehen.

DuA + Info-Studium



2. Grundlagen

- **Beschreibung von Algorithmen durch *Pseudocode*.**
- **Korrektheit von Algorithmen durch *Invarianten*.**
- **Laufzeitverhalten beschreiben durch *O-Notation*.**

Beispiel Minimum-Suche

Eingabe bei Minimum - Suche : Folge von n Zahlen
 (a_1, a_2, \dots, a_n) .

Ausgabe bei Minimum - Suche : Index i , so dass $a_i \leq a_j$ für
alle Indizes $1 \leq j \leq n$.

Minimumalgorithmus : Verfahren, das zu jeder Folge
 (a_1, a_2, \dots, a_n) Index eines kleinsten
Elements berechnet.

Eingabe : (31,41,59,26,51,48)

Ausgabe : 4

Min-Search in Pseudocode

Min - Search(*A*)

```
1 min ← 1
2 for j ← 2 to length[A]
3     do if A[j] < A[min]
4         then min ← j
5 return min
```

Pseudocode (1)

- **Schleifen** (for, while, repeat)
- **Bedingtes Verzweigen** (if – then – else)
- **(Unter-)Programm aufruf/Übergabe** (return)
- **Zuweisung** durch ←
- **Kommentar** durch ▷
- **Daten** als Objekte mit einzelnen Feldern oder Eigenschaften (z.B. $length(A) :=$ Länge des Arrays A)
- **Blockstruktur** durch Einrückung

Min-Search

Min - Search(*A*)

```
1 min ← 1
2 for j ← 2 to length[A]
3   do if A[j] < A[min]
4     then min ← j
5 return min
```

Eingabe : (31,41,59,26,51,48)

Min-Search

Min - Search(*A*)

```
1 min ← 1
2 for j ← 2 to length[A]
3   do if A[j] < A[min]
4     then min ← j
5 return min
```

▷ **Zuweisung**

Eingabe : (31,41,59,26,51,48)

min = 1

Min-Search

Min - Search(*A*)

```
1 min ← 1
2 for j ← 2 to length[A]
3   do if A[j] < A[min]
4     then min ← j
5 return min
```

} **Schleife**

Eingabe : (31, 41, 59, 26, 51, 48)

min = 1, *j* = 2

Min-Search

Min - Search(*A*)

```
1 min ← 1
2 for j ← 2 to length[A]
3   do if A[j] < A[min]
4     then min ← j
5 return min
```

} Verzweigung

Eingabe: (31, 41, 59, 26, 51, 48)

min = 1, *j* = 2, *A*[2] < *A*[1] ? **Nein**

Min-Search

Min - Search(*A*)

```
1 min ← 1
2 for j ← 2 to length[A]
3     do if A[j] < A[min]
4         then min ← j
5 return min
```

} **Schleife**

Eingabe: (31, 41, 59, 26, 51, 48)

min = 1, *j* = 3

Min-Search

Min - Search(*A*)

```
1 min ← 1
2 for j ← 2 to length[A]
3 do if A[j] < A[min]
4 then min ← j
5 return min
```

} Verzweigung

Eingabe: (31, 41, 59, 26, 51, 48)

min = 1, *j* = 3, *A*[3] < *A*[1] ? **Nein**

Min-Search

Min - Search(*A*)

```
1 min ← 1
2 for j ← 2 to length[A]
3   do if A[j] < A[min]
4     then min ← j
5 return min
```

} **Schleife**

Eingabe : (31, 41, 59, 26, 51, 48)

min = 1, *j* = 4

Min-Search

Min - Search(*A*)

```
1 min ← 1
2 for j ← 2 to length[A]
3   do if A[j] < A[min]
4     then min ← j
5 return min
```

} Verzweigung

Eingabe: (31,41,59,26,51,48)

min = 1, *j* = 4, *A*[4] < *A*[1] ? Ja \Rightarrow *min* = 4

Min-Search

Min - Search(*A*)

```
1 min ← 1
2 for j ← 2 to length[A]
3   do if A[j] < A[min]
4     then min ← j
5 return min
```

} **Schleife**

Eingabe : (31,41,59,26,51,48)

min = 4, *j* = 5

Min-Search

Min - Search(*A*)

```
1 min ← 1
2 for j ← 2 to length[A]
3   do if A[j] < A[min]
4     then min ← j
5 return min
```

} Verzweigung

Eingabe : (31,41,59,26,51,48)

min = 4, *j* = 5, *A*[5] < *A*[4] ? **Nein**

Min-Search

Min - Search(*A*)

```
1 min ← 1
2 for j ← 2 to length[A]
3   do if A[j] < A[min]
4     then min ← j
5 return min
```

} **Schleife**

Eingabe : (31,41,59,26,51,48)

min = 4, *j* = 6 = *length*[*A*]

Min-Search

Min - Search(*A*)

```
1 min ← 1
2 for j ← 2 to length[A]
3   do if A[j] < A[min]
4     then min ← j
5 return min
```

} Verzweigung

Eingabe : (31,41,59,26,51,48)

min = 4, *j* = 6, *A*[6] < *A*[4] ? **Nein**

Min-Search

Min - Search(A)

```
1  $min \leftarrow 1$   
2 for  $j \leftarrow 2$  to  $length[A]$   
3   do if  $A[j] < A[min]$   
4     then  $min \leftarrow j$   
5 return  $min$    ▷ Ausgabe
```

Eingabe : (31,41,59,26,51,48)

$min = 4$

Invarianten

Definition 2.1 Eine (Schleifen-) **Invariante** ist eine Eigenschaft eines Algorithmus, die vor und nach jedem Durchlaufen einer Schleife erhalten bleibt.

- **Invarianten dienen dazu, die Korrektheit von Algorithmen zu beweisen.**
- **Sie werden in der Vorlesung immer wieder auftauchen und spielen eine große Rolle.**

Invarianten und Korrektheit

Invariante und Korrektheit von Algorithmen wird bewiesen, indem gezeigt wird, dass

- **Die Invarianten vor dem ersten Schleifendurchlauf erfüllt ist (**Initialisierung**).**
- **Die Eigenschaft bei jedem Schleifendurchlauf erhalten bleibt (**Erhaltung**).**
- **Die Invariante nach Beendigung der Schleife etwas über die Ausgabe des Algorithmus aussagt, Algorithmus korrekt ist (**Terminierung**).**

Invariante bei Min-Search

Invariante: Vor Schleifendurchlauf mit Index i ist $A[\mathit{min}]$ kleinstes Element in $A[1..i-1]$.

Initialisierung: Der kleinste Index für die Schleife ist $i=2$. Davor ist $A[\mathit{min}]=A[1]$.

Erhaltung: if-Abfrage mit then in Zeilen 3 und 4 ersetzt korrekt Minimum, wenn zusätzlich $A[i]$ betrachtet wird.

Terminierung: Vor Durchlauf mit $i=n+1$ ist $A[\mathit{min}]$ das Minimum der Zahlen in $A[1..n]$.

Laufzeitanalyse und Rechenmodell

Für eine präzise mathematische Laufzeitanalyse benötigen wir ein Rechenmodell, das definiert

- Welche Operationen zulässig sind.
- Welche Datentypen es gibt.
- Wie Daten gespeichert werden.
- Wie viel Zeit Operationen auf bestimmten Daten benötigen.

Formal ist ein solches Rechenmodell gegeben durch die **Random Access Maschine (RAM)**.

RAMs sind Idealisierung von 1- Prozessorrechner mit einfachem aber unbegrenzt großem Speicher.

Basisoperationen – Kosten

Definition 2.2: Als Basisoperationen bezeichnen wir

- **Arithmetische Operationen** – Addition, Multiplikation, Division, Ab-, Aufrunden.
- **Datenverwaltung** – Laden, Speichern, Kopieren.
- **Kontrolloperationen** – Verzweigungen, Programmaufrufe, Wertübergaben.

Kosten: Zur Vereinfachung nehmen wir an, dass jede dieser Operationen bei allen Operanden gleich viel Zeit benötigt.

In weiterführenden Veranstaltungen werden Sie andere und häufig realistischere Kostenmodelle kennen lernen.

Eingabegröße - Laufzeit

Definition 2.3: Die Laufzeit $T(I)$ eines Algorithmus A bei Eingabe I ist definiert als die Anzahl von Basisoperationen, die Algorithmus A zur Berechnung der Lösung bei Eingabe I benötigt.

Definition 2.4: Die (worst-case) Laufzeit eines Algorithmus A ist eine Funktion $T : \mathbb{N} \rightarrow \mathbb{R}^+$, wobei

$$T(n) := \max\{T(I) : I \text{ hat Eingabegröße } \leq n\}.$$

Eingabegröße – Laufzeit (2)

- **Laufzeit** angegeben als Funktion der Größe der Eingabe.
- **Eingabegröße** abhängig vom Problem definiert.
- **Eingabegröße Minimumssuche** = Größe des Arrays.
- **Laufzeit bei Minimumssuche:** A Array, für das Minimum bestimmt werden soll.

$T(A) :=$ Anzahl der Operationen, die zur Bestimmung des Minimums in A benötigt werden.

Satz 2.5: Algorithmus Min-Search hat Laufzeit $T(n) \leq an+b$ für Konstanten a, b .

Minimum-Suche

Min - Search(A)	cost	times
1 $min \leftarrow 1$	c_1	1
2 for $j \leftarrow 2$ to $length[A]$	c_2	n
3 do if $A[j] < A[min]$	c_3	$n-1$
4 then $min \leftarrow j$	c_4	t
5 return min	c_5	1

Hierbei ist t die Anzahl der Minimumswechsel.

Es gilt $t \leq n-1$

O-Notation

Definition 2.6. : Sei $g : \mathbf{N} \rightarrow \mathbf{R}^+$ eine Funktion. Dann bezeichnen wir mit $\mathbf{O}(g(n))$ die folgende Menge von Funktionen

$$\mathbf{O}(g(n)) := \left\{ \begin{array}{l} \text{Es existieren Konstanten } c > 0, n_0, \\ f(n) : \text{ so dass f\u00fcr alle } n \geq n_0 \text{ gilt} \\ 0 \leq f(n) \leq cg(n). \end{array} \right\}$$

- $\mathbf{O}(g(n))$ formalisiert: Die Funktion $f(n)$ w\u00e4chst asymptotisch nicht schneller als $g(n)$.
- Statt $f(n)$ in $\mathbf{O}(g(n))$ in der Regel $f(n) = \mathbf{O}(g(n))$

Ω -Notation

Definition 2.7: Sei $g : \mathbf{N} \rightarrow \mathbf{R}^+$ eine Funktion. Dann bezeichnen wir mit $\Omega(g(n))$ die folgende Menge von Funktionen

$$\Omega(g(n)) := \left\{ \begin{array}{l} \text{Es existieren Konstanten } c > 0, n_0, \\ f(n) : \text{ so dass f\u00fcr alle } n \geq n_0 \text{ gilt} \\ 0 \leq cg(n) \leq f(n). \end{array} \right\}$$

- $\Omega(g(n))$ formalisiert: Die Funktion $f(n)$ w\u00e4chst asymptotisch mindestens so schnell wie $g(n)$.
- Statt $f(n)$ in $\Omega(g(n))$ in der Regel $f(n) = \Omega(g(n))$

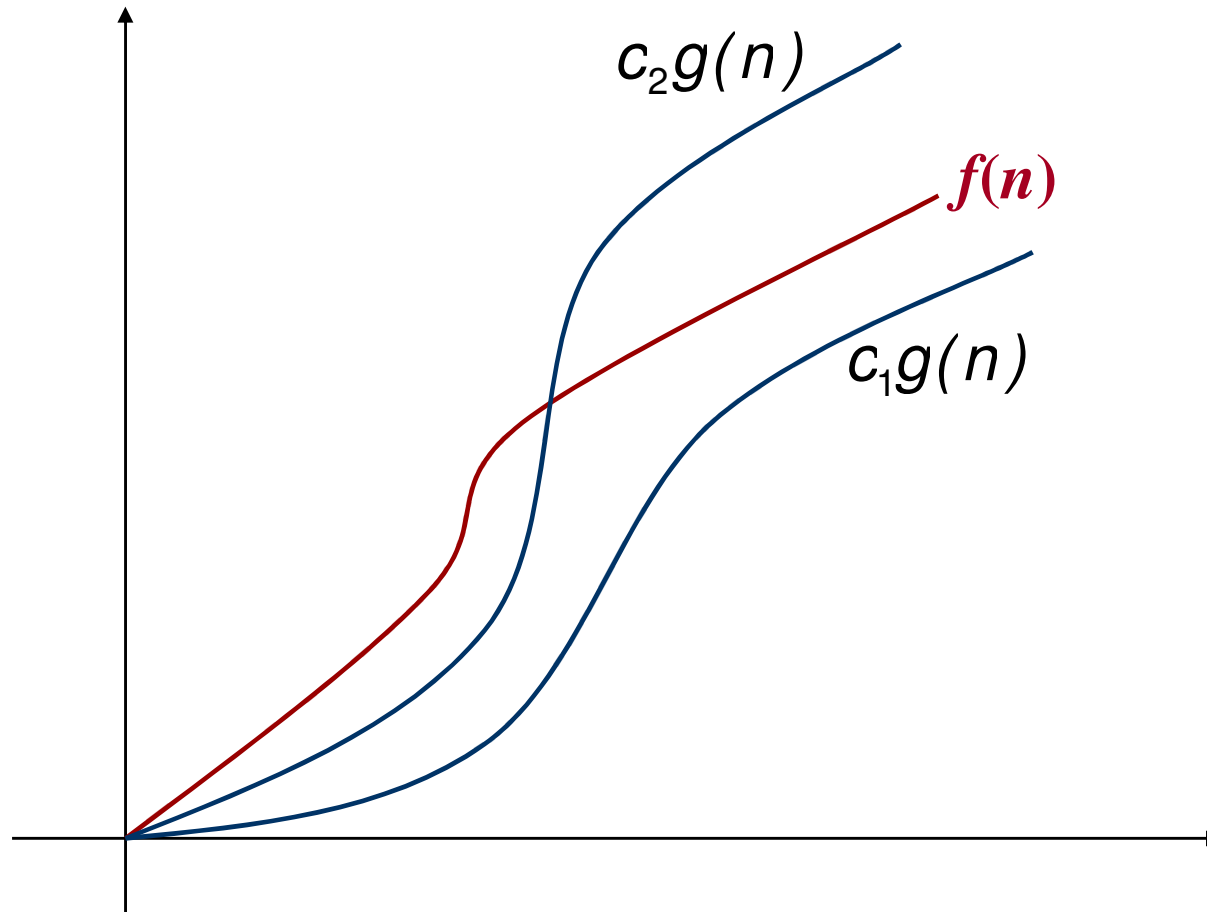
Θ -Notation

Definition 2.8: Sei $g : \mathbf{N} \rightarrow \mathbf{R}^+$ eine Funktion. Dann bezeichnen wir mit $\Theta(g(n))$ die folgende Menge von Funktionen

$$\Theta(g(n)) := \left\{ \begin{array}{l} \text{Es existieren Konstanten } c_1 > 0, c_2, n_0, \\ f(n) : \text{ so dass für alle } n \geq n_0 \text{ gilt} \\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n). \end{array} \right\}$$

- $\Theta(g(n))$ formalisiert: Die Funktion $f(n)$ wächst asymptotisch genau so schnell $g(n)$.
- Statt $f(n)$ in $\Theta(g(n))$ in der Regel $f(n) = \Theta(g(n))$

Illustration von $\Theta(g(n))$



Regeln für Kalküle - Transitivität

O-, **Ω**- und **Θ**-Kalkül sind **transitiv**, d.h.:

➤ Aus $f(n) = \mathbf{O}(g(n))$ und $g(n) = \mathbf{O}(h(n))$ folgt $f(n) = \mathbf{O}(h(n))$.

➤ Aus $f(n) = \mathbf{\Omega}(g(n))$ und $g(n) = \mathbf{\Omega}(h(n))$ folgt $f(n) = \mathbf{\Omega}(h(n))$.

➤ Aus $f(n) = \mathbf{\Theta}(g(n))$ und $g(n) = \mathbf{\Theta}(h(n))$ folgt $f(n) = \mathbf{\Theta}(h(n))$.

Regeln für Kalküle - Reflexivität

- **O**-, **Ω**- und **Θ**-Kalkül sind **reflexiv**, d.h.:

$$f(n) = \mathbf{O}(f(n))$$

$$f(n) = \mathbf{\Omega}(f(n))$$

$$f(n) = \mathbf{\Theta}(f(n))$$

- **Θ**-Kalkül ist **symmetrisch**, d.h.

$$f(n) = \mathbf{\Theta}(g(n)) \text{ genau dann, wenn } g(n) = \mathbf{\Theta}(f(n)).$$

Regeln für Kalküle

Satz 2.10: Sei $f : \mathbb{N} \rightarrow \mathbb{R}^+$ mit $f(n) \geq 1$ für alle n . Weiter sei $k, l \geq 0$ mit $k \geq l$. Dann gilt

1. $f(n)^l = O(f(n)^k)$.
2. $f(n)^k = \Omega(f(n)^l)$.

Satz 2.11: Seien $\varepsilon, k > 0$ beliebig. Dann gilt

1. $\log(n)^k = O(n^\varepsilon)$.
2. $n^\varepsilon = \Omega(\log(n)^k)$.

Anwendung auf Laufzeiten

Min-Search

Satz 2.12: Minimum-Search besitzt Laufzeit $\Theta(n)$.

Zum Beweis ist zu zeigen:

1. Es gibt ein c_2 , so dass die Laufzeit von Min - Search bei allen Eingaben der Größe n immer höchstens c_2n ist.
2. Es gibt ein c_1 , so dass für alle n eine Eingabe I_n der Größe n existiert bei der Min - Search mindestens Laufzeit c_1n besitzt.

Anwendung auf Laufzeiten (2)

- **O-Notation** erlaubt uns, Konstanten zu ignorieren.
- Wollen uns auf **asymptotische Laufzeit** konzentrieren.
- Werden in Zukunft Laufzeiten immer mit Hilfe von **O-, Ω -, Θ -Notation** angeben.