2. Grundlagen

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

Pseudocode

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

Beispiel Minimum-Suche

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

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 $(a_1, a_2, \ldots, a_n)$ 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
```

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                          ▷ Zuweisung
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 = 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

Zuweisung

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

$min = 1$, $j = 4$, $A[4] < A[1]$? Ja $\Longrightarrow$ $min = 4$

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

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

$min = 4$

Invarianten

Definition 2.1 Eine Invariante ist eine Aussage, die über die Ausführung bestimmter Programmbefehle hinweg gilt.

Beispiel: betrachte die Invariante I(x)=„x ist gerade“

$x \leftarrow 0;$
▷ I(x)
$y \leftarrow 3;\ z \leftarrow y+x;$
▷ I(x)

I(x) gilt vor und nach diesen Befehlen

Eine Schleifeninvariante ist eine Sonderform der Invariante, die vor und nach einer Schleife und jedem Durchlauf der Schleife gilt.

Invarianten

Definition 2.1 Eine Invariante ist eine Aussage, die über die Ausführung bestimmter Programmbefehle hinweg gilt. Eine Schleifeninvariante ist eine Sonderform der Invariante, die vor und nach einer Schleife und jedem Durchlauf der Schleife gilt.

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

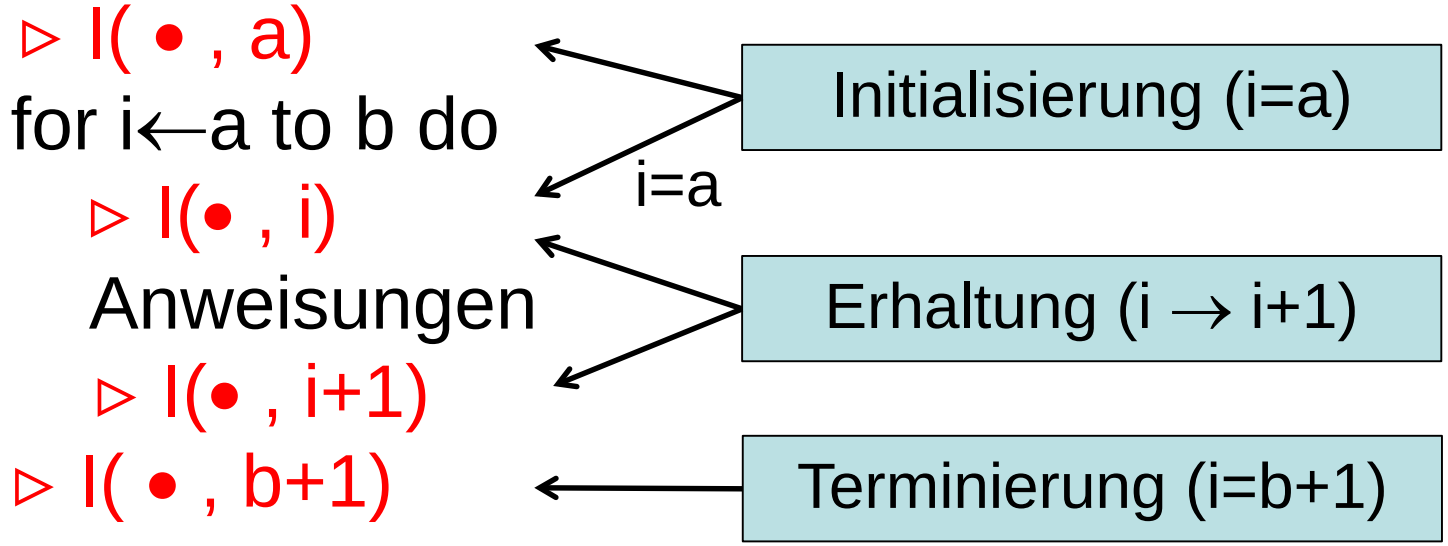
Invarianten und Korrektheit

Für die Korrektheit der Schleifeninvariante muss gezeigt werden, dass

- die Invariante direkt vor Ausführung der Schleife und damit auch am Anfang des ersten Schleifendurchlaufs gilt (Initialisierung),
- falls die Invariante am Anfang eines Schleifendurchlaufs erfüllt ist, sie dann auch am Ende erfüllt ist (Erhaltung), und
- sie direkt nach Beendigung der Schleife gilt (Terminierung).

Invarianten und Korrektheit

Beispiel für eine for-Schleife:



Beweis ähnlich zu vollständiger Induktion.

Invariante bei Min-Search

Min - Search(A)

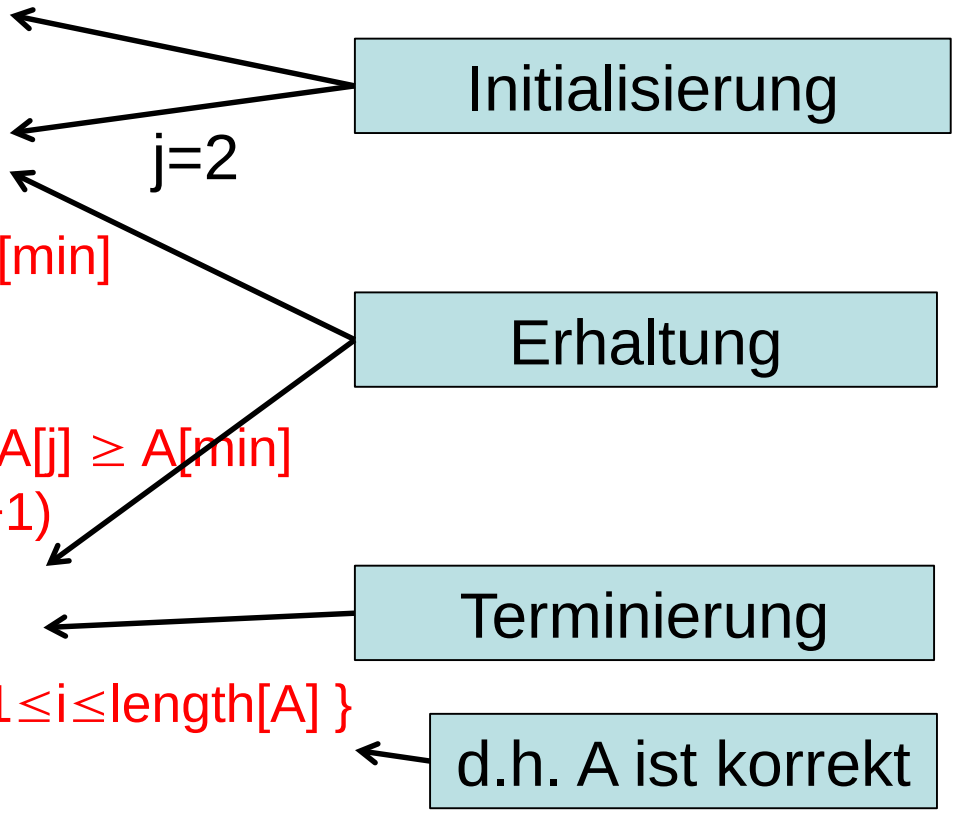
```
1  min  ← 1
2  for j ← 2 to length[A]
3      do if A[j] < A[min]
4             then min ← j
5  return min
```

Invariante I(min,j): A[min] = min{ A[i] | 1≤i≤j-1}

Invariante bei Min-Search

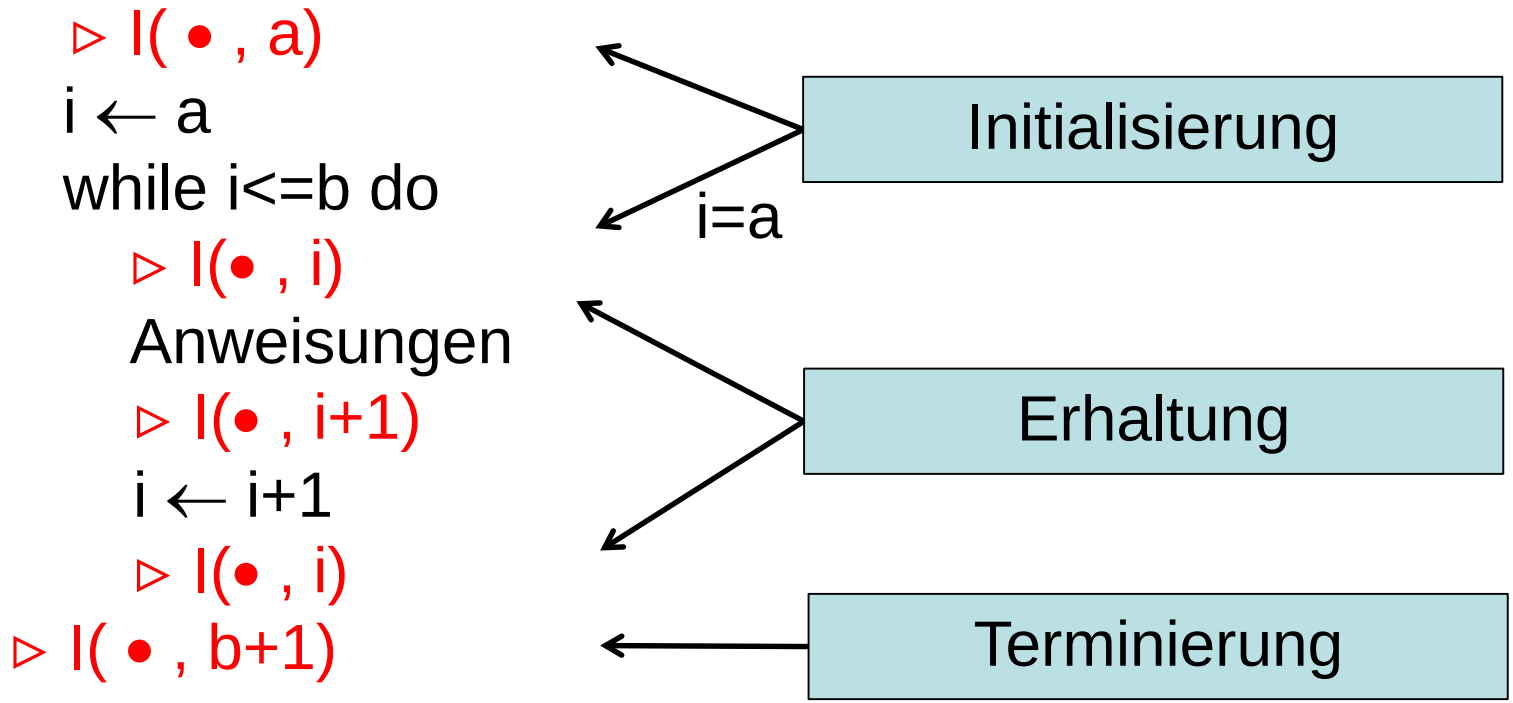
Min-Search(A) ▷ Invariante I(min,j): A[min] = min{ A[i] | 1≤i≤j-1}

```
1  min←1
   ▷ I(min,2)
2  for j←2 to length[A]
   |   ▷ I(min,j)
3  |   if A[j] < A[min] then
   |   |   ▷ I(min,j) ∧ A[j] < A[min]
4  |   |   min←j
   |   |   ▷ I(min,j+1)
   |   |   ▷  sonst: I(min,j) ∧ A[j] ≥ A[min]
   |   |   ▷            ⇒ I(min,j+1)
   |   ▷ I(min,j+1)
   ▷  I(min,length[A]+1)
   ▷  ⇒ A[min] = min{ A[i] | 1≤i≤length[A] }
5  return min
```



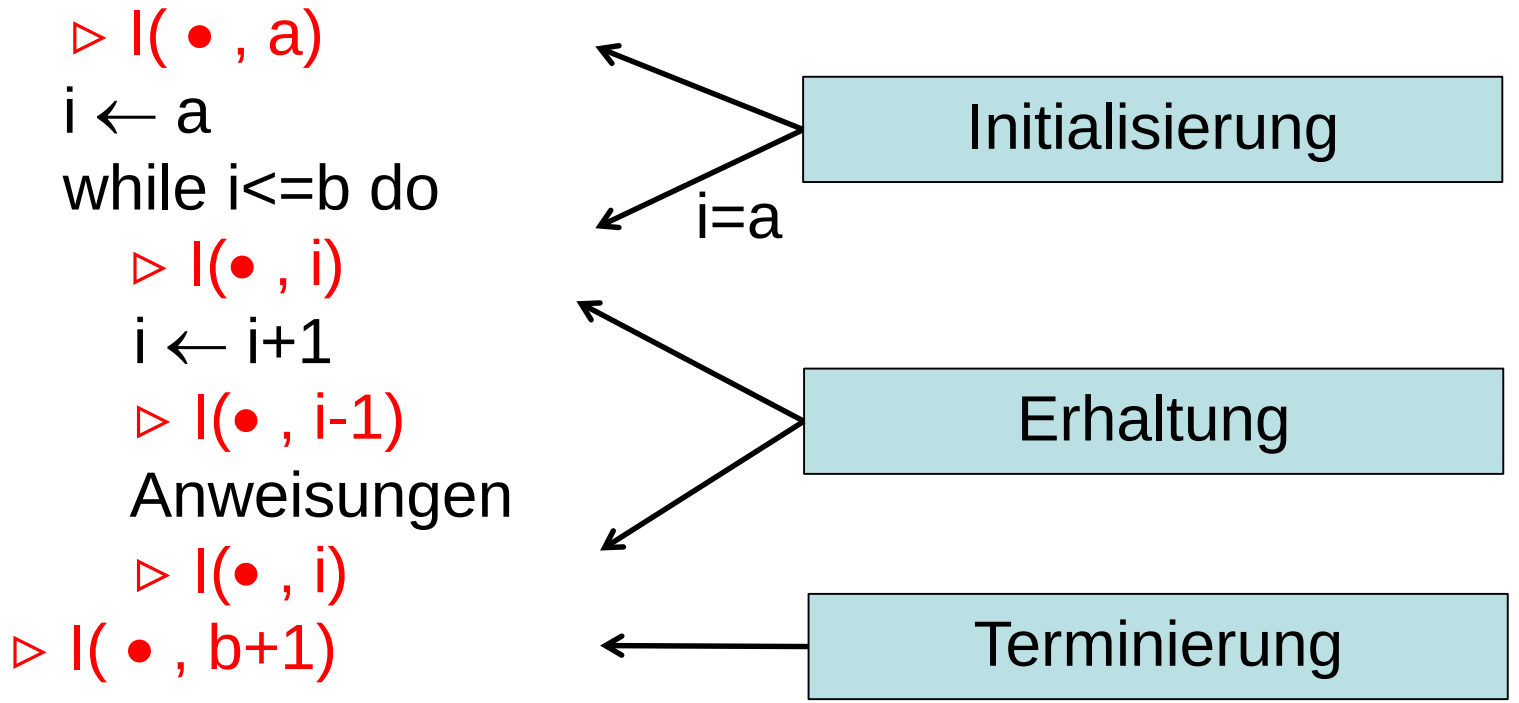
Invarianten und Korrektheit

Beispiel für eine while-Schleife:



Invarianten und Korrektheit

Beispiel für eine while-Schleife:

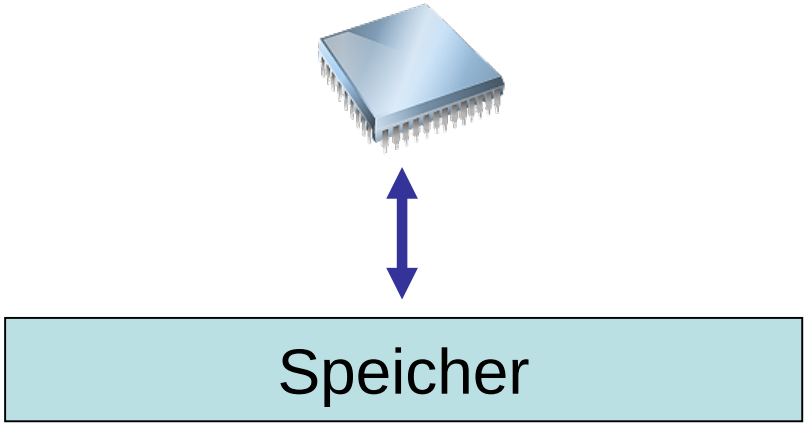


Laufzeitanalyse und Rechenmodell

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

- Welche Basisoperationen 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).



Basisoperationen – Kosten

Definition 2.2: Als Basisoperationen bezeichnen wir

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

Kosten: Zur Vereinfachung nehmen wir an, dass jede dieser Operationen bei allen Operanden gleich viel Zeit benötigt (d.h. 1 Zeiteinheit).

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 Minimumsuche: *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)	Kosten	mal
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 Mimimumswechsel.

Es gilt $t \leq n-1$.

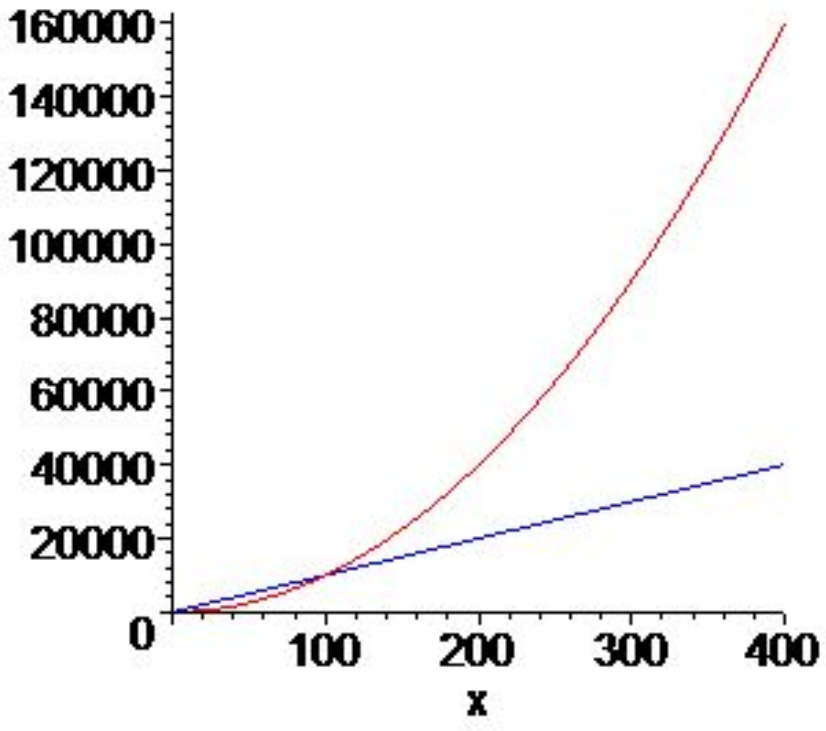
O-Notation

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

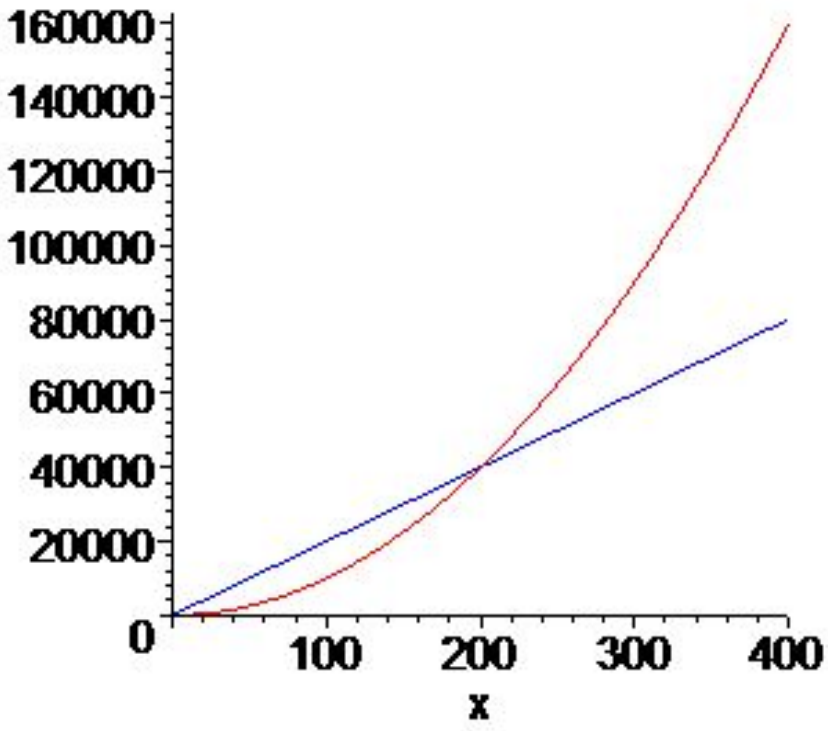
$$O(g(n)) := \left\{ f(n) : \begin{array}{l} \text{Es existieren Konstanten } c, n_0 > 0 \\ \text{so dass für alle } n \geq n_0 \text{ gilt} \\ 0 \leq f(n) \leq cg(n). \end{array} \right\}$$

- O($g(n)$) formalisiert: Die Funktion $f(n)$ wächst asymptotisch nicht schneller als $g(n)$.
- Statt $f(n) \in O(g(n))$ in der Regel $f(n) = O(g(n))$

Illustration von O(*g*(*n*))



$g(x)=x^2$
$f(x)=100x$



$g(x)=x^2$
$f(x)=200x$

Ω-Notation

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

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

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

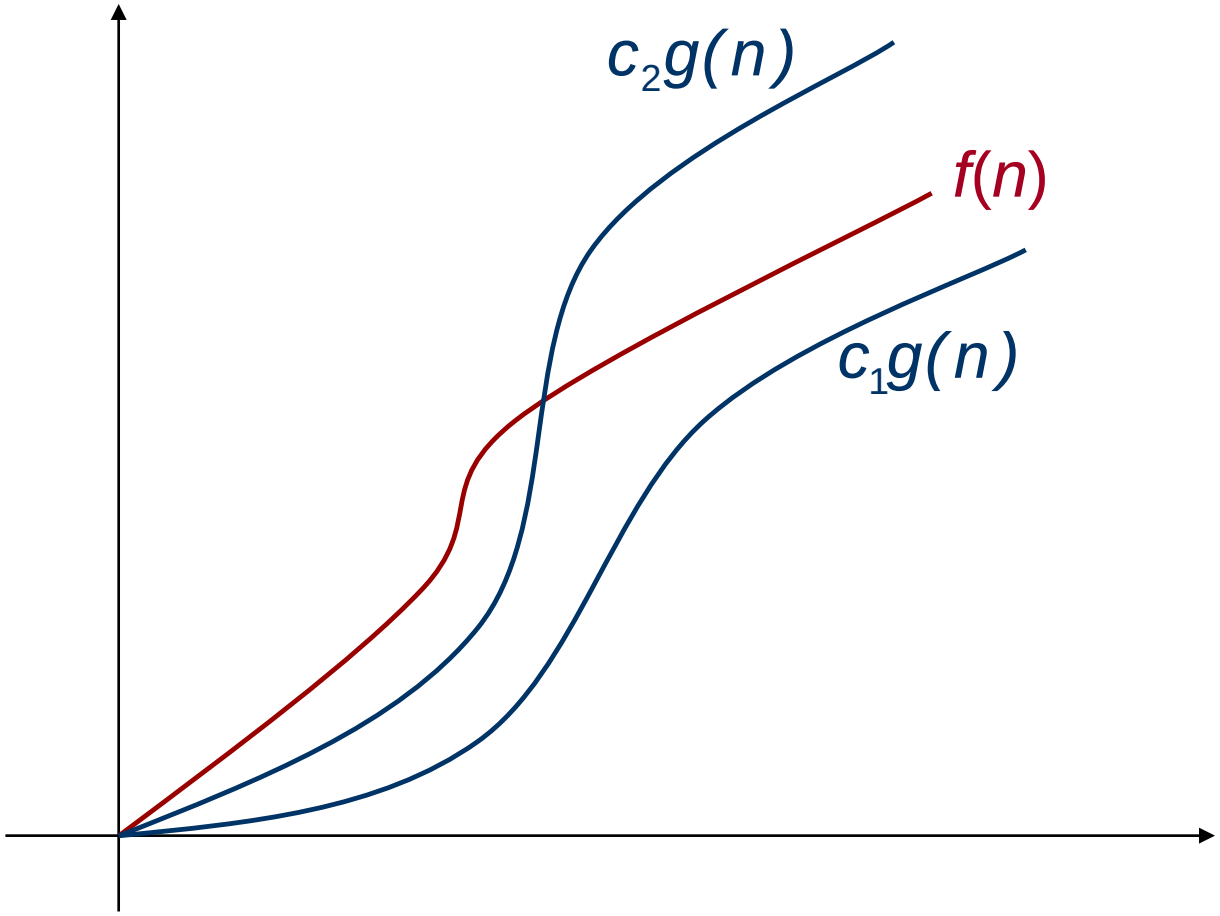
Θ-Notation

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

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

- $\Theta(g(n))$ 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 $\Theta(g(n))$



o-Notation

Definition 2.9.: Sei $g: \mathbb{N} \to \mathbb{R}^+$ eine Funktion. Dann bezeichnen wir mit $o(g(n))$ die folgende Menge von Funktionen

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

- o($g(n)$) formalisiert: Die Funktion $f(n)$ wächst asymptotisch weniger schnell als $g(n)$.

- Analog zu $\Omega(g(n))$: $\omega(g(n))$ ist die Menge aller Funktionen, die asymptotisch schneller steigen als $g(n)$.

Übersicht über Kalküle

- $\exists$: Existenzquantor (“es existiert”)
- $\forall$: Allquantor (“für alle”)

Kalküle für asymptotisches Verhalten:

- $O(g(n))=\{ f(n) \mid \exists\, c>0\ \exists\, n_0>0\ \forall\, n \geq n_0: f(n) \leq c \cdot g(n) \}$
- $\Omega(g(n))=\{ f(n) \mid \exists\, c>0\ \exists\, n_0>0\ \forall\, n \geq n_0: f(n) \geq c \cdot g(n) \}$
- $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$
- $o(g(n))=\{ f(n) \mid \forall\, c>0\ \exists\, n_0>0\ \forall\, n \geq n_0: f(n) \leq c \cdot g(n) \}$
- $\omega(g(n))=\{ f(n) \mid \forall\, c>0\ \exists\, n_0>0\ \forall\, n \geq n_0: f(n) \geq c \cdot g(n) \}$

Nur Funktionen f(n) (bzw. g(n)) mit $\exists N>0\ \forall n>N: f(n)>0$!
Diese sollen schließlich Zeit-/Speicherschranken sein.

Übersicht über Kalküle

- $\limsup_{n\to\infty} x_n$: $\lim_{n\to\infty} (\sup_{m\geq n} x_m)$ (sup: Maximum)
- $\liminf_{n\to\infty} x_n$: $\lim_{n\to\infty} (\inf_{m\geq n} x_m)$ (inf: Minimum)

Alternative Schreibweise für Kalküle:

- $O(g(n))=\{ f(n) \mid \exists\ c>0\ \limsup_{n\to\infty} f(n)/g(n) \leq c \}$
- $\Omega(g(n))=\{ f(n) \mid \exists\ c>0\ \liminf_{n\to\infty} f(n)/g(n) \geq c \}$
- $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

- $o(g(n))=\{ f(n) \mid \limsup_{n\to\infty} f(n)/g(n) = 0 \}$
- $\omega(g(n))=\{ f(n) \mid \liminf_{n\to\infty} g(n)/f(n) = 0 \}$

Exkurs über Grenzwerte

Sei $f:\mathbb{N}\to\mathbb{R}$ eine Funktion und $a,b\in\mathbb{R}$.

- f hat in ∞ den Grenzwert b, falls es zu jedem $\varepsilon>0$ ein $k>0$ gibt mit $|f(z)-b|<\varepsilon$ für alle $z\in\mathbb{R}$ mit $z>k$. In diesem Fall schreiben wir

$$\lim_{z\to\infty} f(z) = b$$

- f hat in ∞ den Grenzwert ∞ , falls es zu jedem $c>0$ ein $k>0$ gibt mit $f(z)>c$ für alle $z\in\mathbb{R}$ mit $z>k$. Wir schreiben dann

$$\lim_{z\to\infty} f(z) = \infty$$

Als Elemente der erweiterten reellen Zahlen $\mathbb{R}\cup\{-\infty, \infty\}$ existieren $\liminf_{n\to\infty}$ und $\limsup_{n\to\infty}$ für jede Folge $(x_n)_{n\in\mathbb{N}}$ reeller Zahlen, was für $\lim_{n\to\infty}$ nicht garantiert ist, da es Folgen gibt, die nicht gegen einen Grenzwert gemäß obiger Definition streben. Die Existenz ergibt sich aus dem Satz von Bolzano-Weierstraß.

Regeln für Kalküle

O-Notation als Platzhalter für eine Funktion:

- statt $g(n) \in O(f(n))$ schreiben wir gewöhnlich $g(n) = O(f(n))$
- Für $f(n)+g(n)$ mit $g(n)=o(h(n))$ schreiben wir auch $f(n)+g(n) = f(n)+o(h(n))$
- Statt $O(f(n)) \subseteq O(g(n))$ schreiben wir auch $O(f(n)) = O(g(n))$

Beispiel: $n^3+n = n^3 + o(n^3) = (1+o(1))n^3 = O(n^3)$

O-Notationsgleichungen sollten nur von links nach rechts verstanden werden!

Regeln für Kalküle

- O- und Ω- bzw. o- und ω-Kalkül sind komplementär zueinander, d.h.:

$$f(n) = O(g(n)) \Rightarrow g(n) = \Omega(f(n))$$

$$f(n) = \Omega(g(n)) \Rightarrow g(n) = O(f(n))$$

$$f(n) = o(g(n)) \Rightarrow g(n) = \omega(f(n))$$

$$f(n) = \omega(g(n)) \Rightarrow g(n) = o(f(n))$$

Beweis: folgt aus Definition der Kalküle

Regeln für Kalküle - Reflexivität

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

$$f(n) = O(f(n))$$

$$f(n) = \Omega(f(n))$$

$$f(n) = \Theta(f(n))$$

- Θ-Kalkül ist symmetrisch, d.h.

$f(n) = \Theta(g(n))$ genau dann, wenn $g(n) = \Theta(f(n))$.

Regeln für Kalküle - Transitivität

Satz 2.10: Das O-, Ω- und Θ-Kalkül sind transitiv, d.h.:

- Aus $f(n) = O(g(n))$ und $g(n) = O(h(n))$ folgt $f(n) = O(h(n))$.
- Aus $f(n) = \Omega(g(n))$ und $g(n) = \Omega(h(n))$ folgt $f(n) = \Omega(h(n))$.
- Aus $f(n) = \Theta(g(n))$ und $g(n) = \Theta(h(n))$ folgt $f(n) = \Theta(h(n))$.

Beweis:
Über Definition der Kalküle.

Transitivität gilt auch für o- und ω-Kalkül.

Regeln für Kalküle - Transitivität

Beweis: (Punkt 1)

$f(n) = O(g(n)) \Leftrightarrow$ es gibt c', n'_0, so dass

$f(n) \leq c' g(n)$ für alle $n > n'_0$.

$g(n) = O(h(n)) \Leftrightarrow$ es gibt c'', n''_0, so dass

$g(n) \leq c'' h(n)$ für alle $n > n''_0$.

Sei $n_0 = \max\{n'_0, n''_0\}$ und $c = c' \cdot c''$. Dann gilt für $n > n_0$:

$$f(n) \leq c' \cdot g(n) \leq c' \cdot c'' \cdot h(n) = c \cdot h(n).$$

Regeln für Kalküle - Transitivität

Beweis: (Punkt 2)

$f(n) = \Omega(g(n)) \Leftrightarrow$ es gibt c', n'_0 , so dass

$f(n) \geq c' g(n)$ für alle $n > n'_0$.

$g(n) = \Omega(h(n)) \Leftrightarrow$ es gibt c'', n''_0 , so dass

$g(n) \geq c'' h(n)$ für alle $n > n''_0$.

Sei $n_0 = \max\{n'_0, n''_0\}$ und $c = c' \cdot c''$. Dann gilt für $n > n_0$:

$$f(n) \geq c' \cdot g(n) \geq c' \cdot c'' \cdot h(n) = c \cdot h(n).$$

Regeln für Kalküle - Transitivität

Beweis: (Punkt 3)

$f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$ und $f(n) = \Omega(g(n))$

$g(n) = \Theta(h(n)) \Rightarrow g(n) = O(h(n))$ und $g(n) = \Omega(h(n))$

$f(n) = O(g(n))$ und $g(n) = O(h(n))$ impliziert
$f(n) = O(h(n))$

$f(n) = \Omega(g(n))$ und $g(n) = \Omega(h(n))$ impliziert
$f(n) = \Omega(h(n))$

$\downarrow$

$f(n) = \Theta(h(n))$.

Regeln für Kalküle

Satz 2.11: Sei $p(n) = \sum_{i=0}^{k} a_i \cdot n^i$ mit $a_k > 0$. Dann ist $p(n) = \Theta(n^k)$.

Beweis:

Zu zeigen: $p(n) = O(n^k)$ und $p(n) = \Omega(n^k)$.

$p(n) = O(n^k)$: Für alle $n \geq 1$ gilt

$$p(n) \leq \sum_{i=0}^{k} |a_i| \, n^i \leq n^k \sum_{i=0}^{k} |a_i|$$

Also ist Definition von $O()$ mit $c = \sum_{i=0}^{k} |a_i|$ und $n_0 = 1$ erfüllt.

$p(n) = \Omega(n^k)$: Für alle $n \geq 2k \cdot A/a_k$ mit $A = \max_i |a_i|$ gilt

$$p(n) \geq a_k \cdot n^k - \sum_{i=0}^{k-1} A \cdot n^i \geq a_k \, n^k - k \cdot A \, n^{k-1} \geq a_k \, n^k / 2$$

Also ist Definition von $\Omega()$ mit $c = a_k/2$ und $n_0 = 2kA/a_k$ erfüllt.

Regeln für Kalküle

Satz 2.12:

Seien $f_1(n) = O(g_1(n))$ und $f_2(n)=O(g_2(n))$.

Dann gilt:

(a) $f_1(n)+f_2(n) = O(g_1(n)+g_2(n))$

(b) $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$

Ausdrücke auch korrekt für Ω , o, ω und Θ.

Beweis: folgt aus Definition des O-Kalküls.

Folgerung aus (b): ist $f(n)=O(g(n))$, dann ist auch $f(n)^k=O(g(n)^k)$ für alle $k \in \mathbb{N}$

Regeln für Kalküle

Satz 2.13: (Rechnung mit Platzhaltern)

(a) $c \cdot f(n) = \Theta(f(n))$ für jede Konstante $c>0$

(b) $O(f(n))+O(g(n)) = O(f(n)+g(n))$

(c) $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$

(d) $O(f(n)+g(n)) = O(f(n))$ falls $g(n)=O(f(n))$

Ausdrücke auch korrekt für Ω statt O.

Vorsicht bei induktiver Anwendung von (d)!

Regeln für Kalküle

Behauptung: $\sum_{i=1}^{n} i = O(n)$

"Beweis": Sei $f(n) = n+f(n-1)$ und $f(1)=1$.

Induktionsanfang: $f(1)=O(1)$.

Induktionsschluss: $f(n-1)=O(n-1)$ gezeigt.

Dann gilt:

$$f(n) = n + f(n-1) = n + O(n-1) = O(n)$$

Also ist $f(n) = \sum_{i=1}^{n} i = O(n)$ natürlich falsch!

Also Vorsicht mit (d) in Induktionsbeweisen!

Regeln für Kalküle

Satz 2.14: Seien f und g stetig und differenzierbar. Dann gilt:

(a) Falls $f'(n) = O(g'(n))$, dann auch $f(n)=O(g(n))$

(b) Falls $f'(n) = \Omega(g'(n))$, dann auch $f(n)=\Omega(g(n))$

(c) Falls $f'(n) = o(g'(n))$, dann auch $f(n)=o(g(n))$

(d) Falls $f'(n) = \omega(g'(n))$, dann auch $f(n)=\omega(g(n))$

Der Umkehrschluss gilt im Allg. nicht!

Regeln für Kalküle

Satz 2.15: Sei $f:\mathbb{N}\to\mathbb{R}^+$ mit $f(n)\geq 1$ für alle $n>n_0$.
Weiter seien $k,l\geq 0$ und $k\geq l$. Dann gilt

(a) $f(n)^l = O(f(n)^k)$

(b) $f(n)^k = \Omega(f(n)^l)$

Satz 2.16: Seien $\varepsilon,k>0$ beliebig. Dann gilt

(a) $\ln(n)^k = O(n^\varepsilon)$

(b) $n^\varepsilon = \Omega(\ln(n)^k)$

Beweis – Satz 2.16

(b) folgt wegen Folie 41 direkt aus (a). Um (a) zu zeigen, beweisen wir zunächst, dass $\ln n = O(n)$ ist.

Beweis über Satz 2.14:

- Wir wissen: $(\ln n)' = 1/n$, $(n)' = 1$.
- Da offensichtlich $1/n=O(1)$ ist, folgt aus Satz 2.14 (a), dass $\ln n = O(n)$ ist.

Beweis über Regel von L´Hospital:

Beweis – Satz 2.16 (2)

Regel von L´Hospital (Spezialfall):

Seien $f,g:\mathbb{N}\to\mathbb{R}^+$ mit $\lim_{x\to\infty} f(x) = \lim_{x\to\infty} g(x) \in \{0,\infty\}$.

Falls $g(x)\neq 0$ für alle $x\in\mathbb{N}$ und

$$\lim_{x\to\infty} \frac{f'(x)}{g'(x)} \in \mathbb{R}\cup\{\infty\}$$

dann gilt

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}$$

Beweis – Satz 2.16 (3)

Die Regel von L´Hospital impliziert mit $f(x)=x$ und $g(x)=\log(x)$, dass

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)} = \infty$$

- Also gibt es laut Definition von lim zu jedem $c>0$ ein $m>0$ mit $x/\ln(x)>c$ für alle $x>m$, oder anders gesagt, $\ln(x) < c \cdot x$ für alle $x>m$.
- Laut Definition des O-Kalküls ist damit $\ln(n) = O(n)$ (und genau genommen sogar $\ln(n) = o(n)$).

Beweis – Satz 2.16 (4)

- Substitution von x durch $\ln(n)$ ergibt:
  Für alle $c>0$ gibt es ein $m(c)>0$, so dass für alle $\ln(n)>m(c)$,

$$\ln(n) > c \ln \ln(n)$$

- Damit gilt:

$$\ln(n)^k = e^{(k/\varepsilon)\cdot(\varepsilon/k)\, k \ln \ln(n)} \leq e^{(\varepsilon/k)\, k \ln(n)} = n^{\varepsilon}$$

  für alle $\ln(n)>m(k/\varepsilon)$.
- Dabei haben wir ausgenutzt, dass

$$\ln(n) > (k/\varepsilon) \ln \ln(n) \text{ für alle } \ln(n) > m(k/\varepsilon)$$

- Also ist $\ln(n)^k = O(n^{\varepsilon})$.

Laufzeitanalyse mithilfe des O-Kalküls

Berechnung der worst-case Laufzeit:

- T(I) sei worst-case Laufzeit für Instruktion I
- T(el. Zuweisung) = O(1), T(el. Vergleich) = O(1)
- T(return x) = O(1)
- T(I;I´) = T(I) + T(I´)
- T(if C then I else I´) = T(C) + max{T(I),T(I´)}
- T(for i:=a to b do I) = $\sum_{i=a}^{b} T(I)$
- T(repeat I until C) = $\sum_{i=1}^{k} (T(C)+T(I))$
  (k: Anzahl Iterationen)
- T(while C do I) = $\sum_{i=1}^{k} (T(C)+T(I))$

Beispiel: Vorzeichenausgabe

Gegeben: Zahl $x \in \mathbb{R}$

Algorithmus Signum(x):

if $x<0$ then return -1

if $x>0$ then return 1

return 0

Wir wissen:

$T(x<0) = O(1)$

$T(\text{return } -1) = O(1)$

$T(\text{if } B \text{ then } I) = O(T(B)+ T(I))$

Satz 2.12(d)

Also ist $T(\text{if } x<0 \text{ then return } -1) = O(1+1) = O(1)$

Beispiel: Vorzeichenausgabe

Gegeben: Zahl $x \in \mathbb{R}$

Algorithmus Signum(x):

if $x<0$ then return -1 $O(1)$

if $x>0$ then return 1 $O(1)$

return 0 $O(1)$

Satz 2.12(b)

Gesamtlaufzeit: $O(1+1+1)=O(1)$

Satz 2.12(d)

Beispiel: Minimumsuche

Gegeben: Zahlenfolge in A[1],…,A[n]

Algorithmus Minimum(A):

min ← A[1]	O(1)
for i←2 to n do	$\sum_{i=1}^{n} T(I)$
if A[i]<min then min:=A[i]	O(1)
return min	O(1)

Satz 2.12(b),(d)

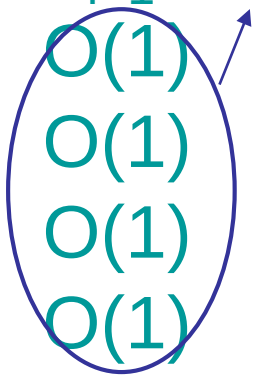
Laufzeit: $O(1 + (\sum_{i=1}^{n} 1) + 1) = O(n)$

Beispiel: Binäre Suche

Gegeben: Zahl x und sortiertes Array A[1],…,A[n]

Algorithmus BinäreSuche(A,x):

```
l←1; r←n                          O(1)
while l < r do                    Σ_{i=1}^{k} T(l)
    m← (r+l) div 2                O(1)
    if A[m] = x then return m     O(1)
    if A[m] < x then l←m+1        O(1)
                else r←m-1        O(1)
return l                          O(1)
```



$O(\sum_{i=1}^{k} 1) = O(k)$

Beispiel: Binäre Suche

Gegeben: Zahl x und sortiertes Array $A[1],\ldots,A[n]$

Algorithmus BinäreSuche(A,x):

```
l←1; r←n
while l < r do
    m← (r+l) div 2
    if A[m] = x then return m
    if A[m] < x then l←m+1
                else r←m-1
return l
```

$O(\sum_{i=1}^{k} 1) = O(k)$

Was ist k ?? →Zeuge

$s_i = (r-l+1)$ in Iteration i

$s_1 = n$, $s_{i+1} \leq s_i/2$

$s_i < 1$: fertig

Also ist $k \leq \log n + 1$

Beispiel: Bresenham Algorithmus

```
(x,y)←(0,R)                                                    O(1)
F←1-R                                                          O(1)
plot(0,R); plot(R,0); plot(0,-R); plot(-R,0)                   O(1)
while x<y do                                                   ∑_{i=1}^{k} T(I)
    x←x+1
    if F<0 then                                                alles
        F←F+2·x + 1
    else                                                       O(1)
        y←y-1
        F←F+2·(x-y)
    plot(x,y); plot(y,x); plot(-x,y); plot(y,-x)
    plot(x,-y); plot(-y,x); plot(-y,x); plot(-x,-y)
```

$O(\sum_{i=1}^{k} 1) = O(k)$

Beispiel: Bresenham Algorithmus

```
(x,y)←(0,R)
F←1-R
plot(0,R); plot(R,0); plot(0,-R); plot(-R,0)
while x<y do
    x←x+1
    if F<0 then
        F←F+2·x + 1
    else
        y←y-1
        F←F+2·(x-y)
    plot(x,y); plot(y,x); plot(-x,y); plot(y,-x)
    plot(x,-y); plot(-y,x); plot(-y,x); plot(-x,-y)
```

Zeuge:

$\phi(x,y) = y-x$

Monotonie: verringert sich um ≥ 1 pro while-Runde

Beschränktheit: while-Bed.

Beispiel: Bresenham Algorithmus

```
(x,y)←(0,R)
F←1-R
plot(0,R); plot(R,0); plot(0,-R); plot(-R,0)
while x<y do
    x←x+1
    if F<0 then
        F←F+2·x + 1
    else
        y←y-1
        F←F+2·(x-y)
    plot(x,y); plot(y,x); plot(-x,y); plot(y,-x)
    plot(x,-y); plot(-y,x); plot(-y,x); plot(-x,-y)
```

Zeuge:

$\phi(x,y) = y-x$

Anzahl Runden:

$\phi_0(x,y) = R$, $\phi(x,y)>0$

$\rightarrow$ maximal R Runden

Beispiel: Fakultät

Gegeben: natürliche Zahl n

Algorithmus Fakultät(n):

if n=1 then return 1 O(1)

else return n · Fakultät(n-1) O(1) + ??

Laufzeit:

- T(n): Laufzeit von Fakultät(n)
- T(n) = T(n-1) + O(1), T(1) = O(1)

Anwendung auf Laufzeiten

- 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.