Algorithmen und Datenstrukturen

Kapitel 2: Grundlagen

Prof. Dr. Peter Kling

Wintersemester 2020/21

Übersicht

1) Pseudocode

Beispiel: Minimum-Suche

Eingabe

- Folge von n Zahlen $(a_1, a_2, \ldots, a_n)$

Ausgabe

- Index *min* mit $a_{min} \leq a_j$ für alle Indizes $i \in \{1, 2, \ldots, n\}$

Beispiel

- **Eingabe:** $(31, 41, 59, 26, 51, 48)$
- **Ausgabe:** 4

Beschreibung von Algorithmen

Wie beschreibt man einen Algorithmus für die Minimum-Suche?

- Zahlenfolge $(a_1, a_2, \ldots, a_n)$ sei in Array A gespeichert
- d. h., der Array-Eintrag $A[i]$ enthält die Zahl a_i

Algorithmus 2.1: MinSearch(A)

```
min ← 1
for i ← 2 to length(A)
    if A[i] < A[min]
        min ← i
return min
```

$A = \langle\ 17\ ,\ 22\ ,\ 6\ ,\ 19\ ,\ 7\ \rangle$

$min = 3$

$i = 5$

2020-11-04

Algorithmen und Datenstrukturen
└─ Pseudocode
└─ Beschreibung von Algorithmen

Wie beschreibt man einen Algorithmus für die Minimum-Suche?

- Zahlenfolge $(a_1, a_2, \ldots, a_n)$ sei in Array A gespeichert
- d. h., der Array-Eintrag $A[i]$ enthält die Zahl a_i

Algorithmus 2.1: MinSearch(A)

```
min ← 1
for i ← 2 to length(A)
    if A[i] < A[min]
        min ← i
return min
```

$A = \langle\, 17 \,,\; 22 \,,\; 6 \,,\; 19 \,,\; 7 \,\rangle$

$min = \; 3$

$i = \; 5$

Algorithmen werden durch Pseudocode beschrieben

Bestandteile von Pseudocode

- **Zuweisung & Vertauschung**
- **Bedingte Ausführung** von Code-Blöcken
- **Schleifen** Wiederholung von Code-Blöcken
- **Einrückung** Beginn/Ende von Code-Blöcken
- **Kommentare** Anmerkungen zum Code
- **Aufrufe & Rückgabe** Kombinieren von Algorithmen
- **Daten** Objekte mit Eigenschaften

```
for i ← 1 to length(A) − 1            // outer loop
    k ← i
    for j ← i + 1 to length(A)        // inner loop
        if A[j] < A[k]
            k ← j
    A[i] ↔ A[k]
return
```

Bestandteile von Pseudocode

- **Zuweisung & Vertauschung**
- **Bedingte Ausführung** von Code-Blöcken
- **Schleifen** Wiederholung von Code-Blöcken
- **Einrückung** Beginn/Ende von Code-Blöcken
- **Kommentare** Anmerkungen zum Code
- **Aufrufe & Rückgabe** Kombinieren von Algorithmen
- **Daten** Objekte mit Eigenschaften

```
for i ← 1 to length(A) − 1          // outer loop
    k ← i
    for j ← i + 1 to length(A)      // inner loop
        if A[j] < A[k]
            k ← j
    A[i] ↔ A[k]
return
```

der gezeigte Algorithmus ist SelectionSort

2) Invarianten

Definition & Beispiel

Definition 2.1: Invariante

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

Beispiel: $I(x) =$ „x ist gerade“

1 $x \leftarrow 2 \cdot z$ // *$I(x)$ gilt*
2 $y \leftarrow 3$ // *$I(x)$ gilt*
3 $x \leftarrow x/2$ // *unklar ob $I(x)$ gilt*

Typischerweise einfach!

Definition 2.2: Schleifeninvariante

Eine Schleifeninvariante für eine Schleife ist eine Invariante, die vor jedem Durchlauf der Schleife sowie am Ende der Schleife gilt.

Invarianten...

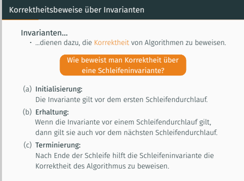
- ...dienen dazu, die Korrektheit von Algorithmen zu beweisen.

Wie beweist man Korrektheit über eine Schleifeninvariante?

(a) **Initialisierung:**
Die Invariante gilt vor dem ersten Schleifendurchlauf.

(b) **Erhaltung:**
Wenn die Invariante vor einem Schleifendurchlauf gilt, dann gilt sie auch vor dem nächsten Schleifendurchlauf.

(c) **Terminierung:**
Nach Ende der Schleife hilft die Schleifeninvariante die Korrektheit des Algorithmus zu beweisen.



- Beweis einer Schleifeninvariante im Wesentlichen per Induktion

Exemplarischer Beweis einer for-Schleifeninvariante

(a) formuliere Schleifeninvariante $I(\underset{\substack{\uparrow\\\text{Schleifenzähler}}}{i}\,,\overset{\substack{\text{weitere}\\\text{Abhängigkeiten}\\\downarrow}}{\bullet})$

(b) beweise Initialisierung

(c) beweise Erhaltung

(d) beweise Terminierung

```
// I(a, •)
for i ← a to b
    // I(i, •)
    ... Anweisungen ...
    // I(i + 1, •)
// I(b + 1, •)
```

Beispiel: MinSearch-Korrektheit via Schleifeninvariante

```
MinSearch(A)
min ← 1
for i ← 2 to length(A)
    if A[i] < A[min]
        min ← i
return min
```

Was ist gewünscht?

- *min* ist Index von minimalem Element in A
- also $A[min] = \min \{ A[i] \mid i \in \{ 1, 2, \ldots, \text{length}(A) \} \}$

Von was hängt die Schleifeninvariante ab?

- Schleifenzähler i und Variable *min*

Kandidat für $I(i, min)$

$$A[min] = \min \{ A[j] \mid j \in \{ 1, 2, \ldots, i - 1 \} \}$$

Beispiel: MinSearch-Korrektheit (Initialisierung)

```
min ← 1
// I(2, min)
for i ← 2 to length(A)
    // I(i, min)
    if A[i] < A[min]
        // I(i, min) ∧ A[i] < A[min]
        min ← i
        // I(i + 1, min)
    else
        // I(i, min) ∧ A[i] ≥ A[min]
        // I(i + 1, min)
    // I(i + 1, min)
// I(length(A) + 1, min)
return min
```

$I(i, min)$

$A[min] = \min \{ A[j] \mid j \in \{ 1, 2, \ldots, i-1 \} \}$

- $I(2, min) = I(2, 1)$
- $I(2, 1)$: "$A[1] = \min \{ A[1] \}$" ✓

Beispiel: MinSearch-Korrektheit (Erhaltung)

```
1   min ← 1
2   // I(2, min)
3   for i ← 2 to length(A)
4       // I(i, min)
5       if A[i] < A[min]
6           // I(i, min) ∧ A[i] < A[min]
7           min ← i
8           // I(i + 1, min)
9       else
10          // I(i, min) ∧ A[i] ≥ A[min]
11          // I(i + 1, min)
12      // I(i + 1, min)
13  // I(length(A) + 1, min)
14  return min
```

$I(i, min)$

$A[min] = \min \{ A[j] \mid j \in \{ 1, 2, \ldots, i-1 \} \}$

- **Annahme:** $I(i, min)$ in Zeile 4
- **Ziel:** $I(i+1, min)$ in Zeile 12
- in Zeile 6:

$$
\begin{aligned}
& A[i] < A[min] \\
&\quad = \min \{ A[1], \ldots, A[i-1] \} \\
\Longrightarrow\ & A[i] = \min \{ A[1], \ldots, A[i] \} \\
\Longrightarrow\ & I(i+1, min) \text{ nach Zeile 7}
\end{aligned}
$$

- Zeile 10 impliziert

$$
\begin{aligned}
& A[min] = \min \{ A[1], \ldots, A[i] \} \\
\Longrightarrow\ & I(i+1, min)
\end{aligned}
$$

- **also:** $I(i+1, min)$ in Zeile 12 ✓

Beispiel: MinSearch-Korrektheit (Terminierung)

1 $min \leftarrow 1$
2 // $I(2, min)$
3 **for** $i \leftarrow 2$ **to** length(A)
4   // $I(i, min)$
5   **if** $A[i] < A[min]$
6   // $I(i, min) \wedge A[i] < A[min]$
7         $min \leftarrow i$
8   // $I(i + 1, min)$
9   **else**
10   // $I(i, min) \wedge A[i] \geq A[min]$
11   // $I(i + 1, min)$
12   // $I(i + 1, min)$
13 // $I(\text{length}(A) + 1, min)$
14 **return** min

$I(i, min)$

$A[min] = \min \{ A[j] \mid j \in \{ 1, 2, \ldots, i - 1 \} \}$

- Ende letzter Schleifendurchlauf:

$$i = \text{length}(A)$$

- nach Zeile 12 gilt also

$$I(\text{length}(A) + 1, min)$$

- gilt folglich auch in Zeile 13 ✓

3) Laufzeit von Algorithmen

Wie misst man eigentlich Laufzeit?

Mathematische Laufzeitanalyse benötigt ein Rechenmodell!

- Können wir in konstanter Zeit addieren?
- Was ist mit multiplizieren? Dividieren? Potenzieren?
- Wie wäre es mit Sortieren? 🤨
- Wieviel Speicher?
- Präzision bei Kommazahlen? Reelle Zahlen?

konst. # Rechen-zyklen

Ein Rechenmodell definiert...

- welche Basisoperationen zulässig sind,
- welche elementare Datentypen es gibt und
- wie Daten gespeichert werden.

konst. Zeit

Laufzeitanalyse & Rechenmodell

Wie misst man eigentlich Laufzeit?

Mathematische Laufzeitanalyse benötigt ein Rechenmodell!

- Können wir in konstanter Zeit addieren?
- Was ist mit multiplizieren? Dividieren? Potenzieren?
- Wie wäre es mit Sortieren? 😕
- Wieviel Speicher?
- Präzision bei Kommazahlen? Reelle Zahlen?

konst. # Rechen-zyklen

Ein Rechenmodell definiert...

- welche Basisoperationen zulässig sind,
- welche elementare Datentypen es gibt und
- wie Daten gespeichert werden.

konst. Zeit

- potenzieren unüblich, aber Zweierpotenzen sind oft einfach
- Sortieren zu kompliziert als Basisoperation

Unser Rechenmodell: Random Access Machine (RAM)

- modelliert eine einfache 1-Prozessor Maschine
- erlaubt folgende Basisoperationen
  - arithmetische Operationen
    Addition, Multiplikation, Division, Ab-/Aufrunden
  - Datenverwaltung
    Laden, Speichern, Kopieren
  - Kontrolloperationen
    Verzweigung, Programmaufruf, Wertrückgabe

vereinfacht
1 Zeiteinheit
pro B-Operation

typi-
scher-
weise

Das ist eine Idealisierung!

- Mathematik schon so schwer genug… 😔
- realistischere Modelle in weiterführenden Veranstaltungen 🤓

Definition 2.3: Laufzeit bei Eingabe I

Die Laufzeit $T_A(I)$ eines Algorithmus A bei Eingabe I ist die Anzahl von Basisoperationen, die Algorithmus A zur Berechnung der Lösung zu Eingabe I benötigt.

- etwas unhandlich…
- hätten gerne einfaches mathematisches Objekt

Ver-
gleich-
barkeit

Wie vergleicht man Laufzeit von…

Sortier-
algorithmus A **vs** Sortier-
algorithmus B **vs** Algorithmus C
zur Wasserrohr-
optimierung

Laufzeiten als Funktion der Eingabegröße

Beobachtung

- Laufzeit typischerweise $\geq$ linear in Eingabegröße
- Eingaben gleicher Größe oft ähnliche Laufzeit

Warum?

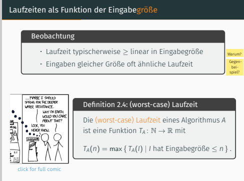
Gegenbeispiel?

click for full comic

Definition 2.4: (worst-case) Laufzeit

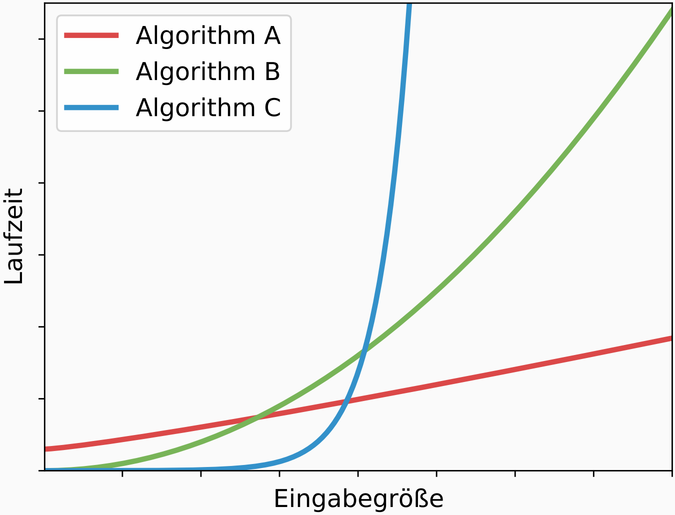
Die (worst-case) Laufzeit eines Algorithmus A ist eine Funktion $T_A : \mathbb{N} \to \mathbb{R}$ mit

$$T_A(n) = \max \{ T_A(I) \mid I \text{ hat Eingabegröße} \leq n \} .$$



- linear Abhängigkeit in der Eingabegröße, da Algorithmen sich (i. A.) die gesamte Eingabe anschauen sollten
- (sinnvoller) Sortieralgorithmus auf sortierter Zahlenfolge hat lineare Laufzeit
- (sinnvoller) Sortieralgorithmus auf invers sortierter Zahlenfolge hat superlineare Laufzeit

Vergleichbarkeit der worst-case Laufzeit



Wie definiert man Eingabegröße?

Aber was genau ist die Eingabegröße einer Eingabe I?

- Anzahl Bits zur Darstellung der Eingabe?
  - Manchmal ja, aber...
  - ...RAM unterstützt Arithmetik in konst. Zeit!
- $\rightsquigarrow$ Abhängig von Maschinenmodell und Problem!
- Beispiele für unser Setting:
  - **Sortieren:** Anzahl der zu sortierenden Zahlen
  - **Minimum-Suche:** Länge des Arrays A
  - **Multiplikation zweier Zahlen:** Anzahl Bits

unabh. von #Bits

Beispiel: Laufzeit von MinSearch

Theorem 2.1: Laufzeit von MinSearch

Der Algorithmus MinSearch hat worst-case Laufzeit $T(n) \leq a \cdot n + b$ für passende Konstanten $a, b \in \mathbb{N}$.

Beweis.

MinSearch(A)	Kosten	mal
$min \leftarrow 1$	c_1	1
for $i \leftarrow 2$ to length(A)	c_2	n
if $A[i] < A[min]$	c_3	$n-1$
$min \leftarrow i$	c_4	t
return min	c_5	1

- Anzahl t der Minimumswechsel ist maximal $n-1$

$\implies$ $T(n) \leq a \cdot n + b$ für passende Konstanten $a, b \in \mathbb{N}$ $\square$

Theorem 2.1: Laufzeit von MinSearch

Der Algorithmus MinSearch hat worst-case Laufzeit $T(n) \leq a \cdot n + b$ für passende Konstanten $a, b \in \mathbb{N}$.

Beweis.

MinSearch(A)	Kosten	mal
$min \leftarrow 1$	c_1	1
for $i \leftarrow 2$ to length(A)	c_2	n
if $A[i] < A[min]$	c_3	$n-1$
$min \leftarrow i$	c_4	t
return min	c_5	1

- Anzahl t der Minimumswechsel ist maximal $n-1$

$\Longrightarrow$ $T(n) \leq a \cdot n + b$ für passende Konstanten $a, b \in \mathbb{N}$ □

z. B. $a = c_2 + c_3 + c_4$ und $b = c_1 + c_5$

4) Asymptotisches Wachstum

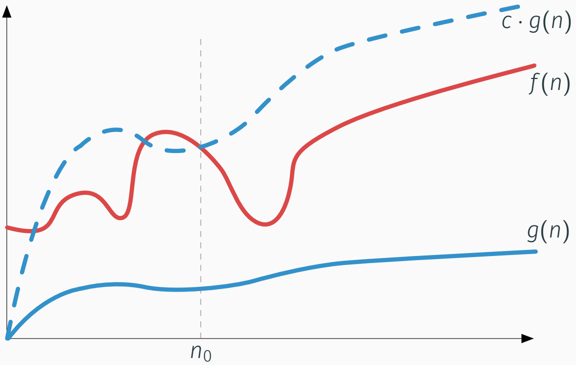
„≤“

Definition 2.5

Sei $g\colon \mathbb{R} \to \mathbb{R}$ eine Funktion. Die Menge $\mathrm{O}(g(n))$ ist definiert als

$$\mathrm{O}(g(n)) = \{ f\colon \mathbb{R} \to \mathbb{R} \mid \exists c > 0\ \exists n_0 > 0\ \forall n \geq n_0\colon |f(n)| \leq c \cdot |g(n)| \} .$$

Funktionen f die asymptotisch höchstens so schnell wachsen wie g.



Ω-Notation

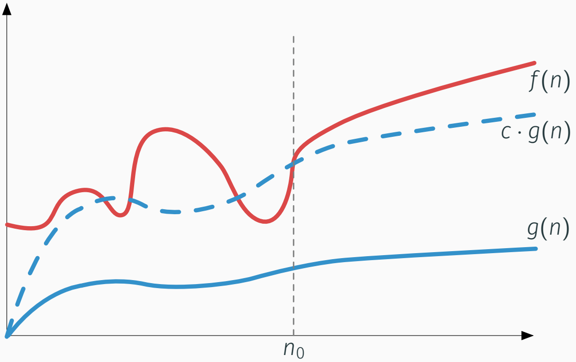
„$\geq$“

Definition 2.6

Sei $g\colon \mathbb{R} \to \mathbb{R}$ eine Funktion. Die Menge $\Omega(g(n))$ ist definiert als

$$\Omega(g(n)) = \{ f\colon \mathbb{R} \to \mathbb{R} \mid \exists c > 0 \; \exists n_0 > 0 \; \forall n \geq n_0 \colon |f(n)| \geq c \cdot |g(n)| \} \,.$$

Funktionen f die asymptotisch mindestens so schnell wachsen wie g.



Θ-Notation

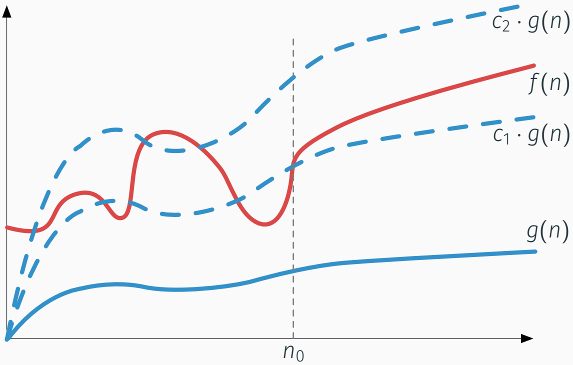
„="

Definition 2.7

Sei $g: \mathbb{R} \to \mathbb{R}$ eine Funktion. Die Menge $\Theta(g(n))$ ist definiert als

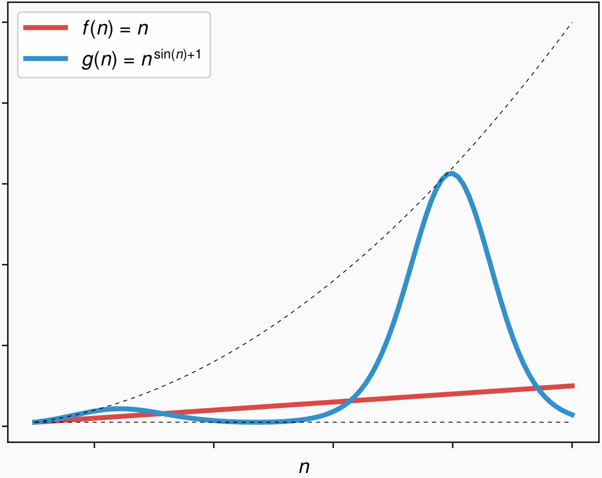
$$\Theta(g(n)) = \mathrm{O}(g(n)) \cap \Omega(g(n)).$$

Funktionen f die asymptotisch genau so schnell wachsen wie g.



Sind zwei Funktionen immer asymptotisch Vergleichbar?

Gilt immer $f(n) = O(g(n))$ oder $f(n) = \Omega(g(n))$?



Die „kleinen“ Geschwister: o- & ω-Notation

Definition 2.8

Sei $g\colon \mathbb{R} \to \mathbb{R}$ eine Funktion. Die Menge $o(g(n))$ ist definiert als

$$o(g(n)) = \{\, f\colon \mathbb{R} \to \mathbb{R} \mid \forall c > 0\ \exists n_0 > 0\ \forall n \geq n_0\colon |f(n)| \leq c \cdot |g(n)| \,\}\,.$$

Funktionen f, die asymptotisch echt langsamer wachsen als g.

„<“

Definition 2.9

Sei $g\colon \mathbb{R} \to \mathbb{R}$ eine Funktion. Die Menge $\omega(g(n))$ ist definiert als

$$\omega(g(n)) = \{\, f\colon \mathbb{R} \to \mathbb{R} \mid \forall c > 0\ \exists n_0 > 0\ \forall n \geq n_0\colon |f(n)| \geq c \cdot |g(n)| \,\}\,.$$

Funktionen f, die asymptotisch echt schneller wachsen als g.

„>“

Übersicht aller Landau-Symbole

$$\mathrm{O}\big(g(n)\big) = \{ f : \mathbb{R} \to \mathbb{R} \mid \exists c > 0\ \exists n_0 > 0\ \forall n \geq n_0 : |f(n)| \leq c \cdot |g(n)| \}$$

$$\mathrm{o}\big(g(n)\big) = \{ f : \mathbb{R} \to \mathbb{R} \mid \forall c > 0\ \exists n_0 > 0\ \forall n \geq n_0 : |f(n)| \leq c \cdot |g(n)| \}$$

$$\Omega\big(g(n)\big) = \{ f : \mathbb{R} \to \mathbb{R} \mid \exists c > 0\ \exists n_0 > 0\ \forall n \geq n_0 : |f(n)| \geq c \cdot |g(n)| \}$$

$$\omega\big(g(n)\big) = \{ f : \mathbb{R} \to \mathbb{R} \mid \forall c > 0\ \exists n_0 > 0\ \forall n \geq n_0 : |f(n)| \geq c \cdot |g(n)| \}$$

$$\Theta\big(g(n)\big) = \mathrm{O}\big(g(n)\big) \cap \Omega\big(g(n)\big)$$

Bemerkungen

- Definitionen in Literatur können leicht abweichen
- auch über Grenzwerte ($\lim$, $\liminf$, $\limsup$) definierbar, z. B.:
  - $f \in \mathrm{O}\big(g(n)\big) \iff \limsup_{n\to\infty} |f(n)/g(n)| < \infty$
  - $f \in \mathrm{o}\big(g(n)\big) \iff \limsup_{n\to\infty} |f(n)/g(n)| = 0$
- hier $f, g : \mathbb{N} \to \mathbb{N}$ für Laufzeiten bzw. Speicherverbrauch

Übersicht aller Landau-Symbole

$$\mathrm{O}(g(n)) = \{ f : \mathbb{R} \to \mathbb{R} \mid \exists c > 0\ \exists n_0 > 0\ \forall n \geq n_0 : |f(n)| \leq c \cdot |g(n)| \}$$
$$\mathrm{o}(g(n)) = \{ f : \mathbb{R} \to \mathbb{R} \mid \forall c > 0\ \exists n_0 > 0\ \forall n \geq n_0 : |f(n)| \leq c \cdot |g(n)| \}$$

$$\Omega(g(n)) = \{ f : \mathbb{R} \to \mathbb{R} \mid \exists c > 0\ \exists n_0 > 0\ \forall n \geq n_0 : |f(n)| \geq c \cdot |g(n)| \}$$
$$\omega(g(n)) = \{ f : \mathbb{R} \to \mathbb{R} \mid \forall c > 0\ \exists n_0 > 0\ \forall n \geq n_0 : |f(n)| \geq c \cdot |g(n)| \}$$

$$\Theta(g(n)) = \mathrm{O}(g(n)) \cap \Omega(g(n))$$

Bemerkungen

- Definitionen in Literatur können leicht abweichen
- auch über Grenzwerte ($\lim$, $\liminf$, $\limsup$) definierbar, z. B.:
  - $f \in \mathrm{O}(g(n)) \iff \limsup_{n\to\infty} |f(n)/g(n)| < \infty$
  - $f \in \mathrm{o}(g(n)) \iff \limsup_{n\to\infty} |f(n)/g(n)| = 0$
- hier $f, g : \mathbb{N} \to \mathbb{N}$ für Laufzeiten bzw. Speicherverbrauch

- im Fall von $f \in \mathrm{o}\big(g(n)\big)$ kann man auch $\lim$ statt $\limsup$ schreiben
- Definitionen über Grenzwerte in Praxis oft nützlicher/einfacher

Laundau-Symbole im algorithmischen Alltag

- als Platzhalter für eine beliebige Funktion aus der Klasse
  - z. B. anstelle von $\log(n)$, n, $n^{3/2}$ einfach $O(n^2)$...
  - ...oder $O(n^{17})$
- statt $g(n) \in O(f(n))$ schreibt man $g(n) = O(f(n))$
- statt $O(g(n)) \subseteq O(f(n))$ schreibt man $O(g(n)) = O(f(n))$
- wenn $g(n) = o(h(n))$, erlaubt uns diese Notation z. B.:

$$f(n) + g(n) = f(n) + o(h(n))$$

- konkreteres Beispiel:

$$n^3 + n = n^3 + o(n^3) = (1 + o(1)) \cdot n^3 = O(n^3)$$

abuse of notation

Solche „Gleichungen“ sind nicht symmetrisch!

$O(n) = O(n^2)$ ✓ $\qquad$ $O(n^2) = O(n)$ ✗

Theorem 2.2: Komplementarität

(a) $g(n) = O(f(n)) \iff f(n) = \Omega(g(n))$

(b) $g(n) = o(f(n)) \iff f(n) = \omega(g(n))$

Beweis.
Folgt leicht aus den Definitionen. $\square$

Theorem 2.3: Symmetrie

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

Beweis.
Folgt leicht aus den Definitionen. $\square$

Theorem 2.4: Reflexivität

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

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

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

Beweis.
Folgt leicht aus den Definitionen. $\square$

Rechenregeln: Transitivität

Theorem 2.5: Transitivität

Wenn $f(n) = O(g(n)) \wedge g(n) = O(h(n))$, dann gilt $f(n) = O(h(n))$. Dies gilt auch für die Ω-/o-/ω- und Θ-Notation.

Beweis.

- $f(n) = O(g(n)) \iff \exists c' > 0\ \exists n_0' > 0\ \forall n \geq n_0' : |f(n)| \leq c' \cdot |g(n)|$
- $g(n) = O(h(n)) \iff \exists c'' > 0\ \exists n_0'' > 0\ \forall n \geq n_0'' : |g(n)| \leq c'' \cdot |h(n)|$
- setze $n_0 = \max\{n_0', n_0''\}$ und $c = c' \cdot c''$

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

- damit haben wir $f(n) = O(h(n))$ gezeigt
- Aussagen für die restlichen Landau-Symbole folgen analog $\square$

Rechenregeln: Linearkombinationen von Potenzen

Theorem 2.6

Sei $p(n) = \sum_{i=0}^{k} c_i \cdot n^i$ für Konstanten c_i und sei $c_k > 0$. Dann gilt $p(n) = \Theta(n^k)$.

Beweis.

- **Zu zeigen:** $p(n) = O(n^k)$ und $p(n) = \Omega(n^k)$
- $\underline{p(n) = O(n^k):}$
  - für alle $n \geq 1$: $|p(n)| \leq \sum_{i=0}^{k} |c_i| \cdot n^i \leq n^k \cdot \sum_{i=0}^{k} |c_i|$

  $\Longrightarrow$ Definition von $O(\cdot)$ erfüllt für $c = \sum_{i=0}^{k} |c_i|$ und $n_0 = 1$
- $\underline{p(n) = \Theta(n^k):}$
  - setze $C = \max\{\, |c_i| \mid i \in \{0, 1, \ldots, k\}\,\}$
  - für alle $n \geq 2k \cdot C/c_k$:

    $|p(n)| \geq c_k \cdot n^k - \sum_{i=0}^{k-1} C \cdot n^i \geq c_k \cdot n^k - k \cdot C \cdot n^{k-1} \geq c_k \cdot n^k/2$

  $\Longrightarrow$ Definition von $\Omega(\cdot)$ erfüllt für $c = c_k/2$ und $n_0 = 2k \cdot C/c_k$ $\square$

Rechenregeln: Linearkombinationen von Potenzen

Theorem 2.6
Sei $p(n) = \sum_{i=0}^{k} c_i \cdot n^i$ für Konstanten c_i und sei $c_k > 0$. Dann gilt $p(n) = \Theta(n^k)$.

Beweis.

- Zu zeigen: $p(n) = O(n^k)$ und $p(n) = \Omega(n^k)$
- $p(n) = O(n^k)$:
  - für alle $n \geq 1$: $|p(n)| \leq \sum_{i=0}^{k} |c_i| \cdot n^i \leq n^k \cdot \sum_{i=0}^{k} |c_i|$

  $\Longrightarrow$ Definition von $O(\cdot)$ erfüllt für $c = \sum_{i=0}^{k} |c_i|$ und $n_0 = 1$
- $p(n) = \Theta(n^k)$:
  - setze $C = \max \{ |c_i| \mid i \in \{0, 1, \ldots, k\} \}$
  - für alle $n \geq 2k \cdot C / c_k$:
    $|p(n)| \geq c_k \cdot n^k - \sum_{i=0}^{k-1} C \cdot n^i \geq c_k \cdot n^k - k \cdot C \cdot n^{k-1} \geq c_k \cdot n^k / 2$

  $\Longrightarrow$ Definition von $\Omega(\cdot)$ erfüllt für $c = c_k/2$ und $n_0 = 2k \cdot C / c_k$ $\square$

n ist gerade so gewählt, dass die letzte Ungleichung korrekt ist

Theorem 2.7

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

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

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

Dies gilt auch für die Ω-/o-/ω- und Θ-Notation.

DIY-Beweis.

Ähnlich zu vorherigen Beweisen. $\square$

Korollar 2.1

Wenn $f(n) = O(g(n))$, dann gilt $(f(n))^k = O((g(n))^k)$.
Dies gilt auch für die Ω-/o-/ω- und Θ-Notation.

Rechenregeln: Verträglichkeit mit Addition & Multiplikation

Theorem 2.7

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

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

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

Dies gilt auch für die Ω-/o-/ω- und Θ-Notation.

DIY-Beweis.
Ähnlich zu vorherigen Beweisen. □

Korollar 2.1

Wenn $f(n) = O(g(n))$, dann gilt $(f(n))^k = O((g(n))^k)$.
Dies gilt auch für die Ω-/o-/ω- und Θ-Notation.

- tatsächlich sogar: $f_1(n) + f_2(n) = O(\max g_1(n), g_2(n))$
- Korollar 2.1 per Induktion aus Aussage (b) von Theorem 2.7

Theorem 2.8

(a) Für jede Konstante c gilt $c \cdot f(n) = O(f(n))$.

(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) Falls $g(n) = O(f(n))$, dann $O(f(n) + g(n)) = O(f(n))$.

Dies gilt auch für die Ω-/o-/ω- und Θ-Notation.

Beweis (hier nur Aussage (b)).

- wähle beliebige $h_1 = O(f(n))$ und $h_2 = O(g(n))$

$$\overset{\text{Theorem 2.7}}{\Longrightarrow} \quad h_1(n) + h_2(n) = O(f(n) + g(n))$$

$$\Longrightarrow \quad O(f(n)) + O(g(n)) = O(f(n) + g(n)) \qquad \square$$

Claim 2.1

Es gilt $\sum_{i=1}^{n} i = O(n)$.

„Beweis“.

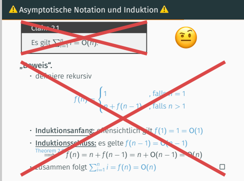
- definiere rekursiv

$$f(n) = \begin{cases} 1 & \text{, falls } n = 1 \\ n + f(n-1) & \text{, falls } n > 1 \end{cases}$$

- **Induktionsanfang:** offensichtlich gilt $f(1) = 1 = O(1)$
- **Induktionsschluss:** es gelte $f(n-1) = O(n-1)$

$$\xRightarrow{\text{Theorem 2.8}} f(n) = n + f(n-1) = n + O(n-1) = O(n)$$

- Zusammen folgt $\sum_{i=1}^{n} i = f(n) = O(n)$ $\square$



Das Problem ist, dass die Induktion n Konstanten akkumuliert, eine für jedes „$f(i) = O(i)$“. Die finale „Konstante“ für „$\sum_{i=1}^{n} i = O(n)$“ wäre die Summe dieser Konstanten. Diese Summe hängt aber von n ab und ist damit nicht konstant!

Theorem 2.9

Seien f und g stetig und differenzierbar mit $g(n) = \omega(1)$.
Falls $f'(n) = \mathrm{O}\big(g'(n)\big)$, dann gilt auch $f(n) = \mathrm{O}(g(n))$.
Dies gilt auch für die Ω-/o-/ω- und Θ-Notation.

DIY-Beweis.
Die Gleichung $f(n) - f(n_0) = \int_{n_0}^{n} f'(x)\,\mathrm{d}x$ könnte hilfreich sein. $\square$

Beachte

Der Umkehrschluss gilt im Allgemeinen nicht!

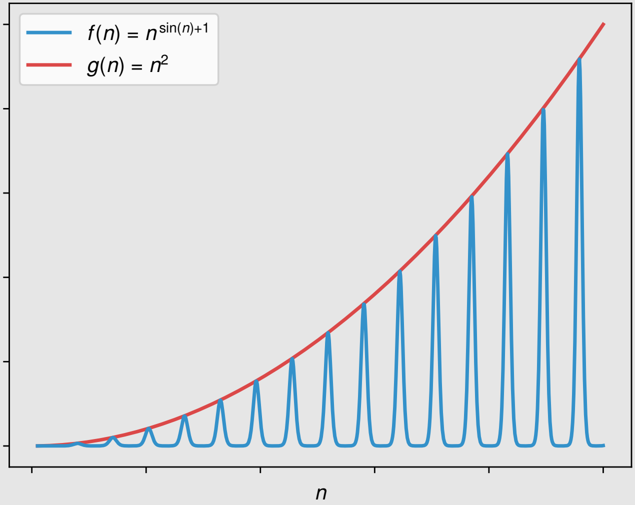
Theorem 2.9

Seien f und g stetig und differenzierbar mit $g(n) = \omega(1)$.
Falls $f'(n) = O(g'(n))$, dann gilt auch $f(n) = O(g(n))$.
Dies gilt auch für die Ω-/o-/ω- und Θ-Notation.

DIY-Beweis.
Die Gleichung $f(n) - f(n_0) = \int_{n_0}^{n} f'(x)\,\mathrm{d}x$ könnte hilfreich sein. □

Beachte

Der Umkehrschluss gilt im Allgemeinen nicht!



Theorem 2.10

Seien $\epsilon, k > 0$ beliebige Konstanten. Dann gilt $\left(\ln(n)\right)^k = O(n^\epsilon)$.

DIY-Beweis.
Ein möglicher Ansatz ist es zuerst $\ln x = o(x)$ zu zeigen und dann die Substitution $x \to \ln(n)$ zu benutzen. $\square$

5) Asymptotische Laufzeitanalyse

Worst-case Laufzeit $T(I)$ verschiedener Operationen

- $T(\; a \leftarrow b \;) = O(1)$
  $T(\; a \leftrightarrow b \;) = O(1)$
  $T(\; a \; R \; b \;) = O(1)$ für $R \in \{ <, >, \leq, \geq \}$
- $T(\; I; I' \;) = T(I) + T(I')$
  $T(\; \textit{if } C \textit{ then } I \textit{ else } I' \;) = T(C) + \max\{ T(I), T(I') \}$
  $T(\; \textit{return } x \;) = O(1)$
- $T(\; \textit{for } i \leftarrow a \textit{ to } b \textit{ do } I \;) = \sum_{i=a}^{b} T(I)$
- $T(\; \textit{repeat } I \textit{ until } C \;) = \sum_{i=1}^{k} (T(C) + T(I))$
  $T(\; \textit{while } C \textit{ do } I \;) = \sum_{i=1}^{k} (T(C) + T(I))$

$k =$ #Iterat.

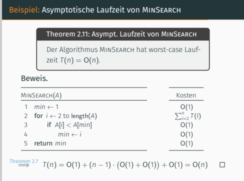
Theorem 2.11: Asympt. Laufzeit von MinSearch

Der Algorithmus MinSearch hat worst-case Laufzeit $T(n) = O(n)$.

Beweis.

MinSearch(A)	Kosten
$min \leftarrow 1$	$O(1)$
for $i \leftarrow 2$ to length(A)	$\sum_{i=2}^{n} T(I)$
if $A[i] < A[min]$	$O(1)$
$min \leftarrow i$	$O(1)$
return min	$O(1)$

$$\stackrel{\text{Theorem 2.7}}{\Longrightarrow} \quad T(n) = O(1) + (n-1) \cdot \big(O(1) + O(1)\big) + O(1) = O(n) \qquad \square$$



Theorem 2.11 folgt natürlich auch direkt aus Theorem 2.1.

Beispiel: Analyse einer while Schleife 1/2

Eingabe

- sortiertes Array A
- gesuchte Zahl x

Ausgabe

- Index l mit $A[l] = x$

Algorithmus 2.2: BINARYSEARCH(A, x)	Kosten
$l \leftarrow 1$; $r \leftarrow \text{length}(A)$	$O(1)$
while $l < r$	$\sum_{i=1}^{k} T(I)$
$m \leftarrow \lfloor (r + l)/2 \rfloor$	$O(1)$
if $A[m] = x$ **then return** m	$O(1)$
if $A[m] < x$ **then** $l \leftarrow m + 1$	$O(1)$
else $r \leftarrow m - 1$	$O(1)$
return l	$O(1)$

$$\overset{\text{Theorem 2.7}}{\Longrightarrow} \quad T(n) = O(1) + k \cdot O(1) + O(1) = O(k)$$

Was ist k?

Beispiel: Analyse einer while Schleife 2/2

Wie groß ist k, die Anzahl der Schleifendurchläufe?

BinarySearch(A, x)

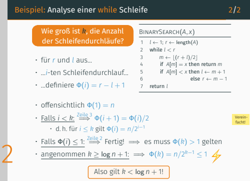
```
1  l ← 1; r ← length(A)
2  while l < r
3      m ← ⌊(r + l)/2⌋
4      if A[m] = x then return m
5      if A[m] < x then l ← m + 1
6                  else r ← m − 1
7  return l
```

- für r und l aus...
- ...i-ten Schleifendurchlauf...
- ...definiere $\Phi(i) = r - l + 1$

- offensichtlich $\Phi(1) = n$
- Falls $i < k$: $\overset{\text{Zeile 3}}{\Longrightarrow}$ $\Phi(i+1) = \Phi(i)/2$
  - d. h. für $i \leq k$ gilt $\Phi(i) = n/2^{i-1}$
- Falls $\Phi(i) \leq 1$: $\overset{\text{Zeile 2}}{\Longrightarrow}$ Fertig! $\Longrightarrow$ es muss $\Phi(k) > 1$ gelten
- angenommen $k \geq \log n + 1$: $\Longrightarrow$ $\Phi(k) = n/2^{k-1} \leq 1$

Verein-
facht!

$\boxed{\text{Also gilt } k < \log n + 1!}$



- wir nehmen vereinfachend an, dass n eine Zweierpotenz ist
- leicht zu verallgemeinern

Vorgehen für while & repeat Schleifen

1. finde eine Potentialfunktion Φ
2. sei $\phi_0 < \infty$ der initiale Wert von ϕ
3. <u>beweise:</u>
   (i) Φ sinkt (bzw. steigt) in jedem Schleifendurchlauf um $\geq \delta$
   (ii) Φ ist nach unten (bzw. oben) beschränkt durch B beschränkt

Warum reicht es nicht zu fordern, dass Φ streng monoton ist?

Laufzeit aus diesen Eigenschaften

für sinkendes Φ

- angenommen #Schleifendurchläufe $\ell > |\Phi_0 - B|/\delta$
- danach gilt $\Phi \leq \Phi_0 - \ell \cdot \delta < B$
- Widerspruch zur unteren Schranke $\Phi \geq B$! ⚡

- Monotonie reicht nicht, da Φ dann eventuell unendlich lange sinken kann, ohne die untere Schranke zu erreichen
- z. B., könnte sich eine positive Potentialfunktion Φ die nach unten durch 0 beschränkt ist in jedem Schleifendurchlauf halbieren, ohne jemals 0 zu erreichen

Ausblick: Analyse rekursiver Algorithmen

Eingabe

- natürliche Zahl $n \in \mathbb{N}$

Ausgabe

- die Fakultät $n!$ von n

Algorithmus 2.3: FACTORIAL(n)	Kosten
if $n = 1$ **then return** 1	$O(1)$
else return $n \cdot \text{FACTORIAL}(n-1)$	$O(1) + ?$

Wie berechnet man hier die Laufzeit?

- sei $T(n)$ die Laufzeit von FACTORIAL(n)
- es gilt $T(n) = \begin{cases} O(1) & \text{, falls } n = 1 \\ T(n-1) + O(1) & \text{, falls } n > 1 \end{cases}$

$\Longrightarrow$ **Rekursionsgleichung!** (Details folgen später...)

Takeaway zur asymptotischen Laufzeit

- Landau-Symbole erlauben uns konst. Faktoren zu ignorieren
- erleichtern Analyse auf Kosten der Genauigkeit

Entwurf von Algorithmen

1. entwerfe Algorithmus mit optimaler asymptotischer Laufzeit
2. optimiere Algorithmus um die Konstanten zu minimieren

Der Rest der Vorlesung konzentriert sich auf Punkt 1!

Fragen?

😴