SS 2021

Algorithmen und Datenstrukturen

1. Kapitel

Einführung und Grundlagen

Martin Dietzfelbinger

April/Mai 2021

Algorithmen!

Beispiel: **Sortieren**.

Vorher: $(\binom{\texttt{Pam}}{f}, \binom{\texttt{Ida}}{f}, \binom{\texttt{Al}}{m}, \binom{\texttt{Liz}}{f}, \binom{\texttt{Lex}}{m}, \binom{\texttt{Tom}}{m}, \binom{\texttt{Sue}}{f}, \binom{\texttt{Ed}}{m})$

Nachher: $(\binom{\texttt{Al}}{m}, \binom{\texttt{Ed}}{m}, \binom{\texttt{Ida}}{f}, \binom{\texttt{Lex}}{m}, \binom{\texttt{Liz}}{f}, \binom{\texttt{Pam}}{f}, \binom{\texttt{Sue}}{f}, \binom{\texttt{Tom}}{m})$

Allgemeine **Aufgabe:** Für jede beliebige gegebene Folge

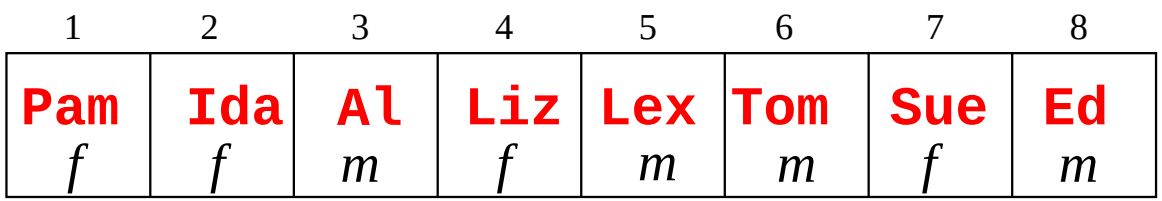
$$(a_1, a_2, \ldots, a_n)$$

von Objekten, die mit einem „Namen“ („Schlüssel“) aus einem total geordneten Bereich $(U, <)$ (Zahlen, Strings/Zeichenreihen, usw.) versehen sind,

sortiere $(a_1, a_2, \ldots, a_n)$, d. h. bringe die Objekte in aufsteigende Reihenfolge.

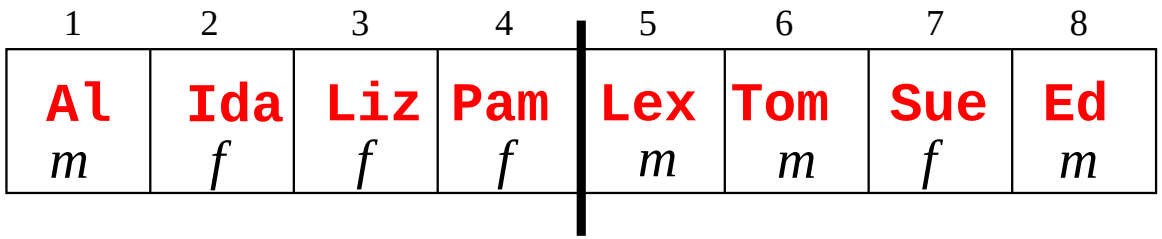
Notation: `x.key`: Schlüssel des Objekts `x`.

Sortieren durch Einfügen (Insertionsort)



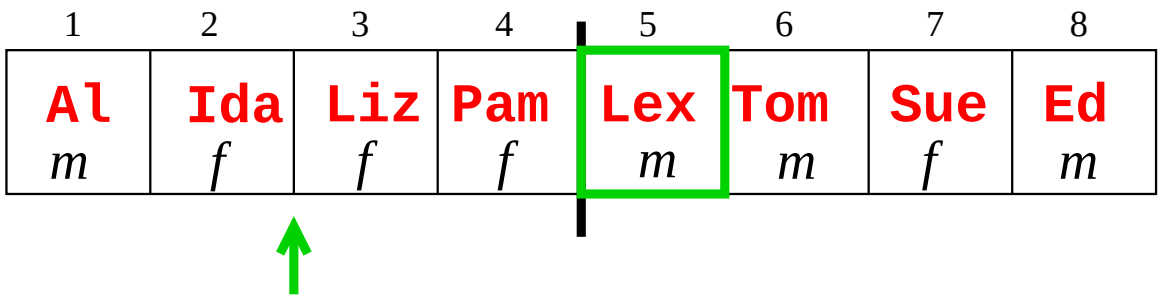
Zu Beginn, vor Runde 1.

Sortieren durch Einfügen (Insertionsort)



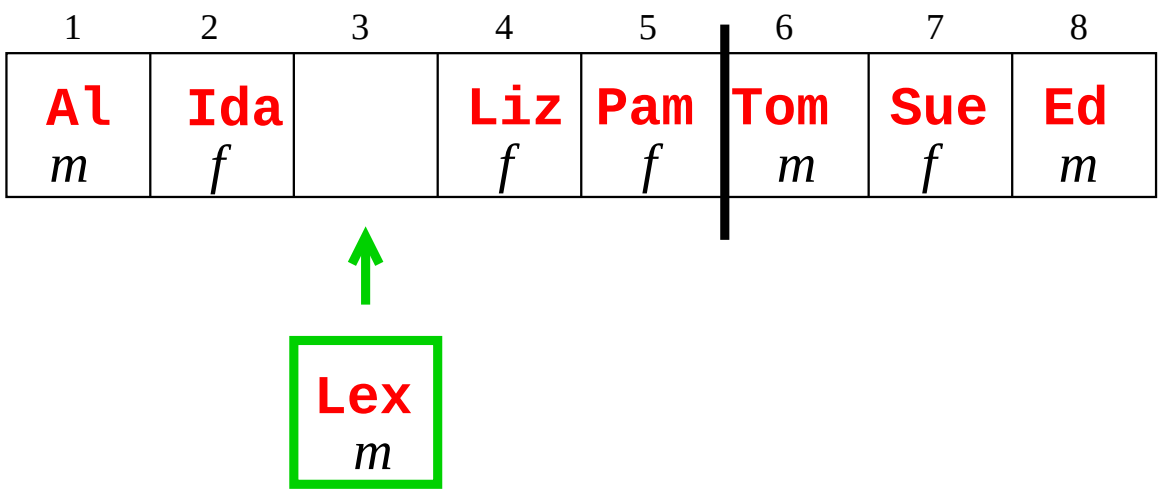
Vor Runde 5: `A[1..4]` sortiert.

Sortieren durch Einfügen (Insertionsort)



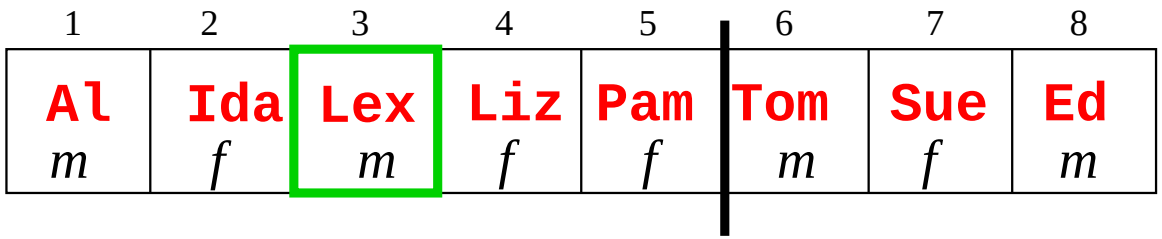
Runde 5: Bestimme Position von `A[5]` in `A[1..4]`.

Sortieren durch Einfügen (Insertionsort)



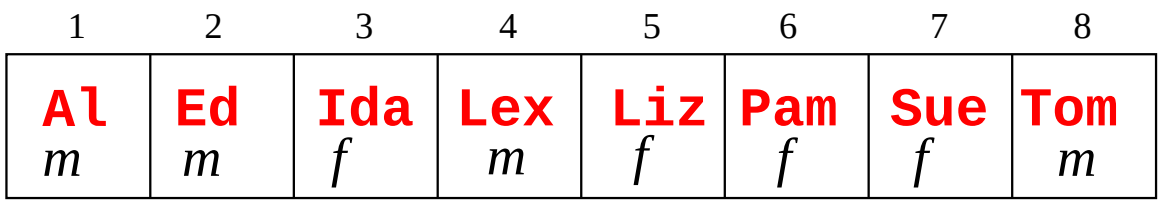
Runde 4: Entnehme `A[5]`, verschiebe `A[4]` und `A[3]`, um für `A[5]` Platz zu machen.

Sortieren durch Einfügen (Insertionsort)



Nach Runde 5: `A[1..5]` sortiert.

Sortieren durch Einfügen (Insertionsort)



Am Ende: `A[1..8]` sortiert.

Sortieren durch Einfügen (Insertionsort)

Pseudocode:

Algorithmus Insertionsort(`A[`$1..n$`]`**)**

```
(1)  for i from 2 to n do                  // Runden i = 2, ..., n
(2)    x ← A[i] ;                          // entnehme A[i]
(3)    j ← i ;
(4)    while j > 1 and* x.key < A[j-1].key do
(5)      A[j] ← A[j-1] ;                   // Einträge größer als A[i] werden einzeln
(6)      j ← j-1 ;                         // um 1 Position nach rechts verschoben
(7)    A[j] ← x ;                          // Einfügen
(8)  return A[1..n] .                      // Ausgabe
```

*: sequenzielles **and**

Was ist ein Algorithmus?

Erklärung:

Ein **Algorithmus** $\mathcal{A}$ ist eine durch einen endlichen Text gegebene Verarbeitungsvorschrift, die zu jeder gegebenen Eingabe x aus einer Menge $\mathcal{I}$ (der Menge der möglichen Eingaben) eindeutig eine Abfolge von auszuführenden Schritten („Berechnung“) angibt.
Wenn diese Berechnung nach endlich vielen Schritten anhält, wird ein Ergebnis $\mathcal{A}(x)$ aus einer Menge $\mathcal{O}$ (der Menge der möglichen Ausgaben) ausgegeben.

Beschreibungsformen: Umgangssprache, Pseudocode,
Programm in Programmiersprache, Programm für formales Maschinenmodell,
Programm in Maschinensprache, Turingmaschinenprogramm, . . .

Merkmale eines Algorithmus

- Endlicher Text
- **Ein** Algorithmus für eine **große** (oft: unendliche!) Menge $\mathcal{I}$ von Eingaben x

  (Beachte: **Zeitliche Trennung** der Formulierung des
  Algorithmus bzw. der Programmierung und der Anwendung auf eine Eingabe.)
- Eindeutig* vorgeschriebene Abfolge von Schritten,
  wenn Eingabe x vorliegt *(im Rahmen des benutzten Modells)
- **Wenn** Verarbeitung anhält: Ausgabe $\mathcal{A}(x) \in \mathcal{O}$ ablesbar

Nicht verlangt: Terminierung auf allen $x \in \mathcal{I}$.

Erweiterungen des Algorithmusbegriffs

- **Online-Algorithmen**: Die Eingabe wird nicht am Anfang der Berechnung, sondern nach und nach zur Verfügung gestellt, Ausgaben müssen nach Teileingabe erfolgen
  $\rightarrow$ Häufige Situation bei **Datenstrukturen**
  $\rightarrow$ Standard bei **Betriebssystemen**, **Systemprogrammen**, **Anwendungsprogrammen** mit **Benutzerdialog** usw.
- **Randomisierung**: Algorithmus führt **Zufallsexperimente** durch, keine Eindeutigkeit der Berechnung gegeben.
- Verteilte Algorithmen*: Mehrere Rechner arbeiten zusammen, kommunizieren, zeitliche Abfolge nicht vorhersagbar

* in dieser Vorlesung nicht oder nur am Rande diskutiert

Algorithmus Insertionsort (InsS)

```
(1)  for i from 2 to n do
(2)     x ← A[i] ;
(3)     j ← i ;
(4)     while j > 1 and x.key < A[j-1].key do
(5)        A[j] ← A[j-1] ;
(6)        j ← j-1 ;
(7)     A[j] ← x;
(8)  return A[1..n] .
```

Fragen:

(a) Ist dieser Algorithmus „**korrekt**“?

(b) **Wie lange** dauern Berechnungen mit diesem Algorithmus?

(a) Korrektheitsbeweise

Damit die Frage nach einem Beweis der Korrektheit überhaupt sinnvoll ist, müssen wir zunächst die zu lösende Aufgabe

spezifizieren.

Bemerkung: Aus einer **vagen Problemstellung** eine **präzise Spezifikation** des zu lösenden Problems zu gewinnen ist ein in der Praxis extrem wichtiger Schritt, dessen Bedeutung oft unterschätzt wird.

Berechnungsprobleme

Erklärung: Ein **Berechnungsproblem $\mathcal{P}$** besteht aus

(i) einer Menge $\mathcal{I}$ von möglichen *Eingaben* („*Inputs*“, „*Instanzen*“)
($\mathcal{I}$ kann unendlich sein!)

(ii) einer Menge $\mathcal{O}$ von möglichen *Ausgaben*

(iii) einer Funktion $f : \mathcal{I} \to \mathcal{O}$

Ein Algorithmus $\mathcal{A}$ **„löst $\mathcal{P}$“**, wenn $\mathcal{A}$ genau die Eingaben aus $\mathcal{I}$ verarbeitet und zu Eingabe $x \in \mathcal{I}$ Ausgabe $\mathcal{A}(x) = f(x)$ liefert.

Spezialfall $\mathcal{O} = \{1, 0\}$ „=“ $\{\mathit{true}, \mathit{false}\}$ „=“ $\{\text{Ja}, \text{Nein}\}$:
$\mathcal{P}$ heißt **Entscheidungsproblem**.

Varianten

- Zu einer Eingabe $x \in \mathcal{I}$ könnte es mehrere legale Ergebnisse geben.

  Formal: In (iii) haben wir statt einer Funktion f eine **Relation $R \subseteq \mathcal{I} \times \mathcal{O}$** mit $\forall x \in \mathcal{I}\, \exists y \in \mathcal{O}\colon R(x, y)$.

  $\mathcal{A}$ löst $\mathcal{P}$, wenn $\mathcal{A}$ genau die Eingaben aus $\mathcal{I}$ verarbeitet und $R(x, \mathcal{A}(x))$ für jedes $x \in \mathcal{I}$ gilt.

  Beispiel „nicht-stabiles Sortieren“: Beim Sortieren von Objekten/Datensätzen, wobei Schlüssel wiederholt vorkommen dürfen, kann *jede* sortierte Anordnung der Objekte eine legale Lösung sein. Dann ist die Lösung nicht eindeutig.

- Online-Situation: Eingabe wird „in Raten“ geliefert, Zwischenausgaben werden verlangt.

Spezifikation des Sortierproblems

Parameter: Die Grundbereiche (Klassen), aus denen die zu sortierenden Objekte und die Schlüssel stammen, sind Parameter der Spezifikation.

$(U, <)$: total geordnete Menge (endlich/unendlich, Elemente heißen „Schlüssel“).

Beispiele: Zahlenmengen $\{1, \ldots, n\}$, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$; Mengen von Strings.

D: Menge (Elemente heißen „Datensätze“).

$key : D \to U$ gibt zu jedem Datensatz einen „Sortierschlüssel“ an.

(i), (ii)

$$\mathcal{I} = \mathcal{O} = \mathbf{Seq}(D) := D^{<\infty} := \bigcup_{n \geq 0} D^n$$
$$= \{(a_1 \ldots, a_n) \mid n \in \mathbb{N}, a_1, \ldots, a_n \in D\},$$

(die Menge aller endlichen Folgen von Datensätzen)

Beispiel: $U = \mathbb{N}$, $D = \mathbb{N} \times \{\mathtt{a}, \ldots, \mathtt{z}\}$

$key\colon D \ni (i, b) \mapsto i \in U$

Ein möglicher Input:

$$x = ((5, \mathtt{r}), (3, \mathtt{s}), (11, \mathtt{t}), (6, \mathtt{e}), (9, \mathtt{r}), (5, \mathtt{t}), (4, \mathtt{o}), (5, \mathtt{i})) \in \mathsf{Seq}(D).$$

Das Ergebnis beim Sortieren sollte z. B. sein:

$$y = ((3, \mathtt{s}), (4, \mathtt{o}), (5, \mathtt{r}), (5, \mathtt{t}), (5, \mathtt{i}), (6, \mathtt{e}), (9, \mathtt{r}), (11, \mathtt{t})) \in \mathsf{Seq}(D).$$

(iii) Wir definieren eine geeignete Relation $R \subseteq \mathcal{I} \times \mathcal{O}$.

Vorher/Nachher dieselbe Objektmenge: Ausgabe ist **Permutation** der Eingabe.

$x = (a_1, \ldots, a_n) \in \mathcal{I}$ und $y = (b_1, \ldots, b_m) \in \mathcal{O}$ stehen in Relation R, d. h. $R(x, y)$ gilt

$:\Leftrightarrow$

$n = m$ und es gibt eine Permutation π von $\{1, \ldots, n\}$ mit

$b_i = a_{\pi(i)}$, $1 \leq i \leq n$, und $key(b_1) \leq key(b_2) \leq \cdots \leq key(b_n)$.

Beispiel:

Für $x = ((5, \mathtt{r}), (3, \mathtt{s}), (11, \mathtt{t}), (6, \mathtt{e}), (9, \mathtt{r}), (5, \mathtt{t}), (4, \mathtt{o}), (5, \mathtt{i}))$ und
$y = ((3, \mathtt{s}), (4, \mathtt{o}), (5, \mathtt{r}), (5, \mathtt{t}), (5, \mathtt{i}), (6, \mathtt{e}), (9, \mathtt{r}), (11, \mathtt{t}))$ ist z. B.

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 2 & 7 & 1 & 6 & 8 & 4 & 5 & 3 \end{pmatrix}$$

passend. ($\pi(2) = 7$: Ausgabepos. $2 \;\widehat{=}\;$ Eingabepos. 7.)

Alternative Formulierung des Sortierproblems:

Die Ausgabe ist nicht die sortierte Folge, sondern die Permutation π, die die Eingabefolge in die richtige Reihenfolge bringt, wie oben beschrieben.

$$\mathcal{O}' = \{\pi \mid \pi \text{ Permutation einer Menge } \{1, \ldots, n\}, n \in \mathbb{N}\}$$

Definition:

$(a_1, \ldots, a_n)$ **wird durch π sortiert** $:\Leftrightarrow key(a_{\pi(1)}) \leq key(a_{\pi(2)}) \leq \cdots \leq key(a_{\pi(n)})$.

$x = (a_1, \ldots, a_n) \in \mathcal{I}$ und $\pi \in \mathcal{O}'$ **stehen in der Relation R'**, d. h.:

$R'(x, \pi)$ gilt $:\Leftrightarrow$ x wird durch π sortiert.

Dann ist $(\mathcal{I}, \mathcal{O}', R')$ ebenfalls eine Spezifikation des Sortierproblems.

Löst Insertionsort das Sortierproblem?

Gilt für jede Eingabe $(a_1, \ldots, a_n)$ in `A[1..n]`, dass nach Ausführung von Insertionsort dieselben Elemente im Array stehen, aber nach Schlüsseln aufsteigend sortiert?

Korrektheitsbeweis an *Beispiel*:

Durch **vollständige Induktion** über $i = 1, 2, \ldots, n$ zeigt man die folgende (Induktions-)Behauptung:

(IB_i)

> Nach Durchlauf Nummer i (in `i`) der äußeren Schleife (1)–(7) stehen in `A[1..i]` dieselben Elemente wie am Anfang, aber aufsteigend sortiert.

Nach Durchlauf Nummer n, also am Ende, gilt dann (IB_n). Diese Behauptung sagt gerade, dass die Einträge aus der ursprünglichen Eingabe nun in aufsteigender Reihenfolge in `A[1..n]` stehen.

Korrektheitsbeweis für Insertionsort

Zunächst erledigen wir zwei (triviale) Randfälle: Wenn $n = 0$ ist, also das Array leer ist, oder wenn $n = 1$ ist, es also nur einen Eintrag gibt, dann tut der Algorithmus gar nichts, und die Ausgabe ist korrekt.

Sei nun $n \geq 2$ beliebig, und der Input sei im Array `A[1..n]` gegeben.
Wir beweisen die Induktionsbehauptung (IB_i) durch Induktion über $i = 1, 2, \ldots, n$.

I.A.: $i = 1$: „Nach Durchlauf für $i = 1$“ ist vor Beginn.
Anfangs ist das Teilarray `A[1..1]` (nur ein Eintrag) sortiert, also gilt (IB_1).

I.V.: Wir nehmen an: $1 < i \leq n$ und (IB_{i-1}) gilt, d. h. `A[1..i − 1]` ist sortiert, enthält dieselben Objekte wie am Anfang, und Schleifendurchlauf i beginnt.

I.-Schritt: Betrachte Schleifendurchlauf i (Zeilen (2)–(7)).
$x :=$ `A[`i`]` ist das aktuelle Objekt; es wird in Variable `x` kopiert.
Position `A[`i`]` kann man sich jetzt als leer vorstellen.

Sei $m = m_i$ die größte Zahl in $\{0, \ldots, i-1\}$, so dass $\boxed{key(\texttt{A[}j\texttt{]}) \leq key(x) \text{ für } 1 \leq j \leq m}$.

(Wenn es gar kein $j \leq i-1$ mit $key(\texttt{A}[j]) \leq key(x)$ gibt, ist $m = 0$.
Wenn für alle $1 \leq j \leq i-1$ die Ungleichung erfüllt ist, ist $m = i-1$.)

Weil nach I.V. $\texttt{A}[1..i-1]$ sortiert ist, gilt: $\boxed{key(x) < key(\texttt{A}[j]) \text{ für } m+1 \leq j \leq i-1}$.
Die innere Schleife (4)–(7) mit der Initialisierung in Zeile (3) testet, durch direkten Vergleich in Zeile (4), die Zahlen $j-1 = i-1, i-2, i-3, \ldots, 0$ nacheinander darauf, ob $j-1 = m$ ist. Entweder wird dabei $j = 1$ erreicht, dann ist $m = 0 = j-1$, oder man stellt für ein $j \geq 2$ fest, dass $key(\texttt{A}[j-1]) \leq key(x)$ gilt. Dann ist offenbar ebenfalls $m = j-1$.

Schon während der Suche nach m werden (in Zeile (5)) die Einträge
$\texttt{A}[i-1]$,$\texttt{A}[i-2]$,. . . ,$\texttt{A}[m+1]$, in dieser Reihenfolge, um eine Position nach rechts verschoben, ohne die Sortierung dieser Elemente zu ändern.
Nun stehen $\texttt{A}[1], \ldots, \texttt{A}[m]$ an ihrer ursprünglichen Position, $\texttt{A}[m+1]$,$\texttt{A}[m+2]$,. . . ,$\texttt{A}[i-1]$ (vorher) stehen jetzt in $\texttt{A}[m+2..i]$.
In Zeile (7) wird x (aus x) an die Stelle $\texttt{A}[m+1]$ geschrieben. Nach den Feststellungen in den Boxen und aus der I.V. ergibt sich, dass nun $\texttt{A}[1..i]$ aufsteigend sortiert ist. Also gilt (IB_i).

Nach Durchlaufen der äußeren Schleife Schleife für $i = 2, \ldots, n$ gilt die Aussage (IB_n), also ist nach Abschluss des Ablaufs das Array sortiert.

Methoden für Korrektheitsbeweise

- **Vollständige Induktion** (wie soeben gesehen)
  Damit wird die Korrektheit von **Schleifen** bewiesen.
  Meist ist dabei die Formulierung einer geeigneten **Induktionsbehauptung** zentral, die dann **„Schleifeninvariante“** genannt wird.
- **Verallgemeinerte Induktion** (d.h. man schließt von (IB_j) *für alle* $j < i$ auf (IB_i)).
  Damit werden **rekursive Prozeduren** behandelt. (Siehe: Divide-and-Conquer.)
- **Diskrete Mathematik**: Ad-hoc-Beweise.
  (Beispiele werden wir bei Graphalgorithmen sehen.)

PAUSE

(b) Laufzeitanalysen

Sprechweise: Einen Algorithmus **analysieren** heißt, Kosten, Aufwand, „Rechenzeit“ (synonym!) zu bestimmen oder zumindest abzuschätzen, die sich einstellt, wenn man eine Implementierung des Algorithmus $\mathcal{A}$ auf Eingabe x ablaufen lässt.

Natürlich geht es schneller, kurze Arrays zu sortieren als lange. Daher ist ein zentraler Parameter einer solchen Analyse die

Größe der Eingabe $x \in \mathcal{I}$, bezeichnet mit $|x|$ oder $\mathbf{size}(x)$ (oft, nicht immer: n).

Die Bedeutung von $\mathrm{size}(x)$ ist abhängig vom Problem $\mathcal{P}$.
Beispiele: Beim Sortierproblem ist $\mathrm{size}((a_1, \ldots, a_n)) = n$.
Wenn $x \in \mathcal{I}$ ein String $a_1 \ldots a_n$ ist: $\mathrm{size}(x) = n = \#(\text{Buchstaben in } x)$.
Wenn Eingabe $G = (V, E)$ ein Graph mit $n = |V|$ Knoten (Ecken) und m Kanten ist:
Oft $\mathrm{size}(G) = n + m$.

$$\mathcal{I}_n := \{x \in \mathcal{I} \mid \mathrm{size}(x) = n\} \text{ – „Größenklasse“.}$$

Kosten – Rechenzeiten

Kosten für Elementaroperationen in Programm:

Addition, Vergleich, Test, Sprung, Zuweisung,
Auswertung eines Attributs, Zeiger-/Indexauswertung

Jede Elementaroperation op ist mit einer Konstanten c_{op} bewertet, abhängig von **Programmiersprache**, **Compiler**, **Prozessor**, **Hauptspeichertyp**, **Cachestruktur**, . . .

Idee: op benötigt **höchstens** „Zeit“ c_{op} (Takte oder ns oder μs oder . . .)

Unabhängig vom konkreten Input x.

Die Konstanten c_{op} werden nie explizit genannt;
vereinfachend oft sogar: $\boldsymbol{c_{\text{op}} = 1}$ („1 Maschinenschritt“, „1 Operation“).

Kosten – Rechenzeiten

Wir schreiben

$$O(1)$$

für: „ein Wert, nicht größer als eine (von x unabhängige) Konstante c“
und wollen Laufzeit des Algorithmus anhand von Pseudocode o. ä. analysieren.
Technische Besonderheiten von konkreten Rechnern
(Cachearchitekturen, Pipelining, Mehrkernprozessoren, Coprozessoren, . . .),
Implementierungen, Compileroptimierungen, usw.
zu berücksichtigen wäre viel zu komplex!

Wir definieren als Kosten/Rechenzeit von Algorithmus $\mathcal{A}$ auf Eingabe x:

$t_{\mathcal{A}}(x) := \#$(Operationen, die $\mathcal{A}$ auf Input x ausführt). (# bedeutet „Anzahl“)

Beispiel: Insertionsort auf Input $(a_1, \ldots, a_n)$, gegeben in `A[1..n]`

Zeilen (2)–(3) und (7): Werden $(n-1)$-mal ausgeführt.
Es gibt n Schleifenende-Tests (Zeile (1)) $\Rightarrow$

Kosten von Zeilen (1)–(3), (7): $n \cdot \boldsymbol{O}(1) =$ „$\boldsymbol{O}(n)$“ (genaue Erklärung später).

Zeilen(4)–(6) für ein festes i (in `i`): Entscheidend: Anzahl der Tests in (4).

Der Test „`j` > 1“ wird zwischen 1-mal und i-mal durchgeführt,
der Test „`x.key` $<$ `A[j-1].key`“ zwischen 1-mal und $(i-1)$-mal, **inputabhängig**.

Setze* $\boldsymbol{k_i} := k_i(x) := \#\{l \mid 1 \leq l \leq i-1 \text{ und } key(a_i) < key(a_l)\}$ („verkehrt herum“).
(Nicht schwer zu sehen: $k_i = i - 1 - m_i$ mit m_i aus dem Korrektheitsbeweis von Insertionsort.)

Kosten für Zeilen (4)–(6) bei Schleifendurchlauf für i: $(k_i+1) \cdot \boldsymbol{O}(1) =$ „$\boldsymbol{O}(k_i+1)$“.

---

* **Notation:** $\#A$ oder $|A|$: Kardinalität von A.

Kosten insgesamt: Summiere über $2 \leq i \leq n$.

$$\boldsymbol{O}(n) + (k_2 + 1) \cdot \boldsymbol{O}(1) + (k_3 + 1) \cdot \boldsymbol{O}(1) + \cdots + (k_n + 1) \cdot \boldsymbol{O}(1) =$$

$$= \boldsymbol{O}(n + \sum_{2 \leq i \leq n} k_i).$$

(„Ausklammern des O“ und unkonventionelle Benutzung von „$=$“ wird später gerechtfertigt.)

Typisch für solche Analysen: Nicht jede Operation zählen, sondern Zurückziehen auf das Zählen von **„charakteristischen Operationen“**, die den Rest bis auf einen konstanten Faktor dominieren.

Hier: Vergleiche zwischen Schlüsseln (Zeile (4)), Anzahl $\leq k_i + 1$.

(Oder: Zuweisungen von Elementen (Zeilen (5) und (7)), Anzahl exakt $k_i + 1$.)

Definition:

$$\boldsymbol{F_x} := \sum_{2 \leq i \leq n} k_i$$
$$= \#\{(l,i) \mid 1 \leq l < i \leq n, key(a_l) > key(a_i)\}$$

Anzahl der **„Fehlstände“** in $x = (a_1, \ldots, a_n)$

Z. B. gibt es in $x = (4, 7, 5, 3)$ vier Fehlstände,
nämlich $(1, 4), (2, 3), (2, 4), (3, 4)$.

Gesamtaufwand von Insertionsort auf Eingabe x der Länge n ist „$\boldsymbol{O(n + F_x)}$“.

„Worst case“ (Schlechtester Fall):

Wegen $F_x \leq \sum_{2 \leq i \leq n}(i-1) = \sum_{j=1}^{n-1} j \overset{\text{(Mathem.)}}{=} \frac{n(n-1)}{2} < \frac{n^2}{2}$ gilt:

Gesamtaufwand/-rechenzeit/-kosten für Insertionsort auf Inputs mit n Komponenten beträgt „$\boldsymbol{O}(n + n^2/2)$“ = „$\boldsymbol{O}(n^2)$“.

Es können auch solch große Kosten auftreten:
$k_i = i - 1$ für alle i gilt für absteigend sortierte Eingaben, z. B.
$x = ((11, \mathtt{a}), (9, \mathtt{q}), (8, \mathtt{p}), (7, \mathtt{a}), (5, \mathtt{z}), (4, \mathtt{u}), (2, \mathtt{q}))$.

Für solche Inputs sind die Kosten mindestens $n + \frac{1}{2}n(n-1) > \frac{1}{2}n^2$.

Wir schreiben: Worst-case-Kosten sind „$\boldsymbol{\Theta}(n^2)$“. (Details zur Notation später).

Worst-case-Kosten, allgemein

Sei $\mathcal{A}$ Algorithmus für Inputs aus $\mathcal{I}$. Wir hatten: $\mathcal{I}_n = \{x \in \mathcal{I} \mid \mathsf{size}(x) = n\}$.

Wir definieren **worst-case-Kosten** oder **Kosten im schlechtesten Fall** auf Inputs der Größe n:

$$\boldsymbol{T_{\mathcal{A}}(n)} := \max\{t_{\mathcal{A}}(x) \mid x \in \mathcal{I}_n\}.$$

Wenn man betonen will, dass es um die worst-case-Kosten geht: $\boldsymbol{T_{\mathcal{A},\mathsf{worst}}(n)}$.

Im *Beispiel* Insertionsort haben wir: $T_{\mathsf{IS},\mathsf{worst}}(n) =$ „$\boldsymbol{\Theta}(n^2)$“.

Best-case-Kosten, allgemein

Wir definieren **best-case-Kosten** oder **Kosten im besten Fall**:

$$T_{\mathcal{A},\text{best}}(n) := \min\{t_{\mathcal{A}}(x) \mid x \in \mathcal{I}_n\}.$$

Beispiel Insertionsort:
$k_i = 0$ für alle i gilt für aufsteigend sortierte Eingabe x, z. B.
$x = ((2, \mathtt{q}), (4, \mathtt{u}), (5, \mathtt{z}), (7, \mathtt{a}), (8, \mathtt{p}), (9, \mathtt{q}), (11, \mathtt{a}))$.
Für solche Inputs ist $F_x = 0$, also $n + F_x = n + 0 = n$.
Für jeden Input gilt $n + \sum_{2 \le i \le n} k_i \ge n$.
Gesamtaufwand für Insertionsort im besten Fall ist daher: $T_{\text{IS,best}}(n) = \boldsymbol{O}(n + 0) = \boldsymbol{O}(n)$.

„Average case“ (Mittlerer Fall):

Wir betrachten wieder das Beispiel Insertionsort, mit Eingabe $X = (a_1, \ldots, a_n)$.

Vereinfachende Annahme: Die Schlüssel $key(a_i)$, $1 \leq i \leq n$, sind verschieden.

Annahme: Jede der $n!$ möglichen Anordnungen der n Objekte in der Eingabe $(a_1, \ldots, a_n)$ tritt mit derselben Wahrscheinlichkeit $1/n!$ auf.

(Vgl.: Laplace'sches Indifferenzprinzip, `https://de.wikipedia.org/wiki/Indifferenzprinzip`)

Beispiel: Schlüssel $2, 3, 7$:
Anordnungen $(2,3,7), (2,7,3), (3,2,7), (3,7,2), (7,2,3), (7,3,2)$, jede mit W.-keit $\frac{1}{6}$

Bilde **Mittelwert/Erwartungswert** für $F_x = \sum_{2 \leq i \leq n} k_i$ und berechne:

$$\mathbf{E}(F_x) \overset{(*)}{=} \frac{1}{2} \cdot \binom{n}{2} = \frac{1}{2} \cdot \frac{n(n-1)}{2} = \frac{1}{4} n(n-1) < \frac{n^2}{4}.$$

Zu $(*)$: Wir betrachten der Einfachheit halber Inputs, die aus den Zahlen $1, \ldots, n$ bestehen. Dann ist jeder mögliche Input x einfach eine Permutation $(a_1, \ldots, a_n)$ von $\{1, \ldots, n\}$, und $\mathcal{I}_n$ ist die Menge aller dieser Inputs. Es gibt $n!$ viele davon. Nun betrachten wir die Menge P_n aller Paare $(x, (l, i))$, für x einen solchen Input und $1 \leq l < i \leq n$. Wir haben offenbar $|P_n| = n! \cdot \binom{n}{2}$. Uns interessiert

$$P_n^{\times} := \{(x, (l, i)) \in P_n \mid (l, i) \text{ ist Fehlstand in } x\}.$$

Zu Input $x = (a_1, \ldots, a_n)$ und Paar (l, i) betrachten wir den Input $x^{(l,i)}$, der sich durch Vertauschen von a_l und a_i in x ergibt. (*Beispiel*: $(5, \underline{2}, 3, 1, \underline{4})^{(2,5)}$ ist $(5, \underline{4}, 3, 1, \underline{2})$.)
Dann ist folgendes offensichtlich: Wenn wir in $x^{(l,i)}$ wieder Positionen l und i vertauschen, erhalten wir x zurück. Und: Von Inputs x und $x^{(l,i)}$ hat genau einer bei (l, i) einen Fehlstand. Das heißt, dass es in P_n genau $\frac{1}{2}|P_n| = \frac{1}{2}n! \cdot \binom{n}{2}$ viele Paare $(x, (l, i))$ gibt, für die x bei (l, i) einen Fehlstand hat. Kurz: $\left|P_n^{\times}\right| = \frac{1}{2}n! \cdot \binom{n}{2}$.

Also:

$$\sum_{x \in \mathcal{I}_n} F_x = \sum_{x \in \mathcal{I}_n} \sum_{\substack{1 \leq l < i \leq n \\ (l,i) \text{ ist Fehlstand in } x}} 1 \quad = \left|P_n^{\times}\right| = \frac{1}{2}n! \cdot \binom{n}{2}.$$

Durch Mittelung erhalten wir die mittlere/erwartete Anzahl von Fehlständen in einem $x \in \mathcal{I}_n$ als

$$\mathbf{E}(F_x) = \frac{1}{n!} \cdot \frac{1}{2} n! \cdot \binom{n}{2} = \frac{1}{2}\binom{n}{2} = \frac{n(n-1)}{4}.$$

Also ist der Aufwand von Insertionsort **im mittleren Fall**, auf Inputs der Länge n: $\boldsymbol{O}(n^2)$, aber auch $\boldsymbol{\Omega}(n^2)$. Zusammengefasst: $\boldsymbol{\Theta}(n^2)$. (Notation wird später erklärt.)

Average-case-Kosten, allgemein

Average-case-Kosten oder **durchschnittliche Kosten**:

Gegeben sei, für jedes $n \in \mathbb{N}$, eine Wahrscheinlichkeitsverteilung $\mathbf{Pr}_n$ auf $\mathcal{I}_n$.

D.h.: Für jedes $x \in \mathcal{I}_n$ ist eine Wahrscheinlichkeit $\mathbf{Pr}_n(x)$ festgelegt.

(Woher kommt $\mathbf{Pr}_n$? Aus empirischer Beobachtung, oder per Festlegung, z. B. auf der Basis des Laplace'schen Indifferenzprinzips.) Dann definieren wir:

$$T_{\mathcal{A},\mathsf{av}}(n) := \mathbf{E}(t_{\mathcal{A}}) = \sum_{x \in \mathcal{I}_n} \mathbf{Pr}_n(x) \cdot t_{\mathcal{A}}(x).$$

(**E**: Erwartungswert, berechnet mit $\mathbf{Pr}_n$; „Mittelwert".)

Beispiel Insertionsort: $T_{\mathsf{IS},\mathsf{av}}(n) = \text{„}\mathbf{\Theta}(n + \frac{1}{4}n(n-1))\text{“} = \text{„}\mathbf{\Theta}(n^2)\text{“}$.

Analyse von Kosten/Rechenzeit

Feststellung:

Selbst für feste Implementierung eines Algorithmus $\mathcal{A}$ auf fester Maschine und feste

„Eingabegröße“ $n = \text{size}(x) = |x|$

kann der Berechnungsaufwand auf verschiedenen Inputs sehr unterschiedlich sein.

D. h. der Begriff „**Die** Laufzeit von Algorithmus $\mathcal{A}$ auf Eingaben x vom Umfang $n = \text{size}(x)$“

ist oft sinnlos.

Sinnvoll: **Schlechtesten** und besten Fall, eventuell **durchschnittlichen** Fall betrachten.

Analyse von Kosten/Rechenzeit

In allen drei Situationen (worst case, best case, average case):

Konstante Faktoren ignorieren!

Technisch:

Obere Schranken (d. h. „Kosten sind höchstens . . . “) $\rightarrow$ „O-Notation“

Untere Schranken (d. h. „Kosten sind mindestens . . . “) $\rightarrow$ „Ω-Notation“ (s. unten)

Asymptotisches Verhalten, **präzise**
(d. h. „Kosten sind bis auf konstanten Faktor gleich . . . “) $\rightarrow$ „Θ-Notation“

Ende 1. Vorlesung

Größenordnungen von Funktionen

Notation:

$\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ (bei Bedarf: $\mathbb{N}^+ = \{n \in \mathbb{N} \mid n > 0\} = \{1, 2, 3, \ldots\}$)

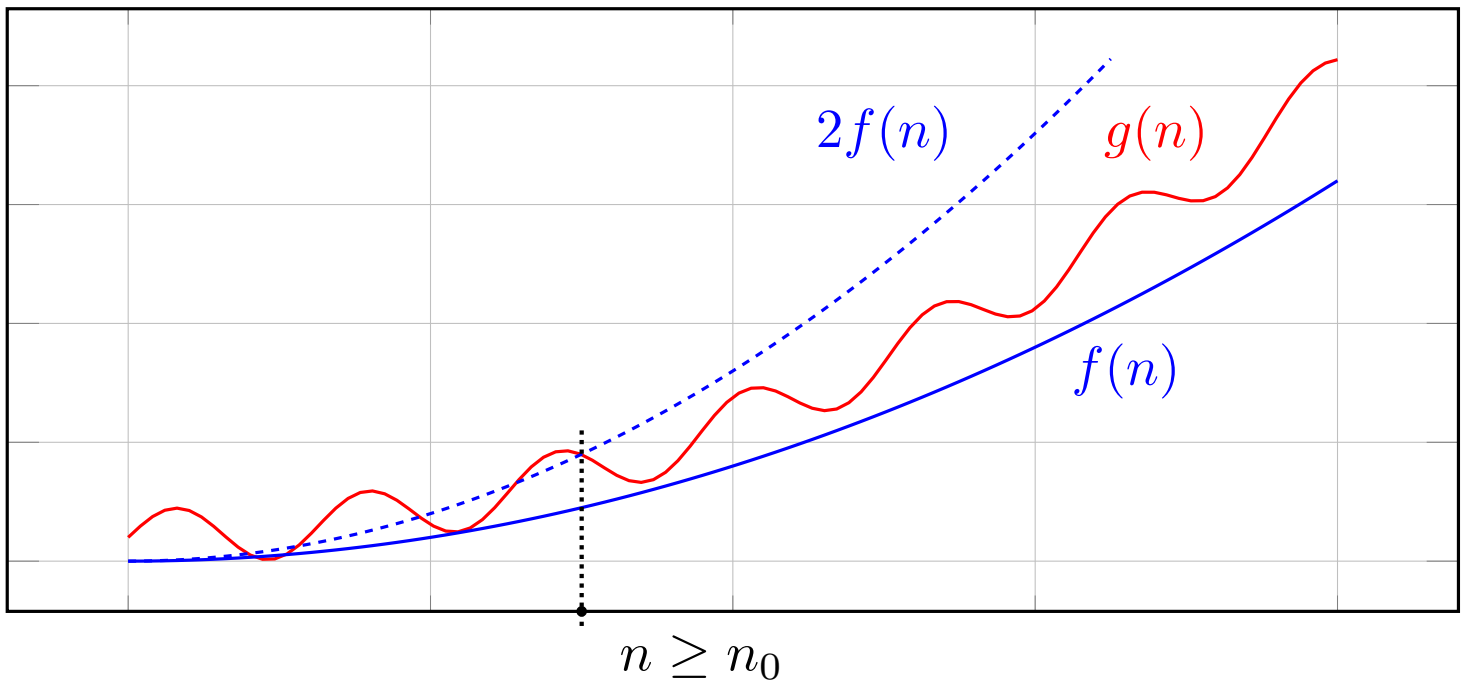
$\mathbb{R}_0^+ = \{x \in \mathbb{R} \mid x \geq 0\}$ und $\mathbb{R}^+ = \{x \in \mathbb{R} \mid x > 0\}$

Wir betrachten Funktionen $f : \mathbb{N} \to \mathbb{R}$. Das Argument $n \in \mathbb{N}$ ist oft die Eingabegröße, $f(n)$ schätzt eine Rechenzeit oder eine Wahrscheinlichkeit ab. Uns interessieren praktisch nur Funktionen, die für alle genügend großen Argumente n positiv oder nichtnegativ sind. – Abkürzungen:

$$\mathcal{F}^+ := \{f \mid f : \mathbb{N} \to \mathbb{R}, \exists n_0 \forall n \geq n_0 : f(n) > 0\}.$$

$$\mathcal{F}_0^+ := \{f \mid f : \mathbb{N} \to \mathbb{R}, \exists n_0 \forall n \geq n_0 : f(n) \geq 0\}.$$

„g **wächst asymptotisch höchstens so schnell wie** f“ soll heißen:
Es gibt eine Konstante $c > 0$, so dass für alle genügend großen n gilt: $g(n) \leq c \cdot f(n)$.



Horizontale Achse: n in $\mathbb{N}$. – Im Bild ist $g(n) \leq c \cdot f(n)$ für $n \geq n_0$ und $c = 2$.

$O(f)$: Funktionenmenge

Wir fassen alle Funktionen g, die asymptotisch höchstens so schnell wie f wachsen, in einer Menge zusammen.

Definition Für $f \in \mathcal{F}^+$ sei

$$\boldsymbol{O(f)} := \{\, g \in \mathcal{F}_0^+ \mid \exists n_0 \in \mathbb{N}\ \ \exists c > 0\ \ \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}.$$

Beispiel: Für $g(n) = 2n + 7\log n$ und $f(n) = n$ gilt $g \in O(f)$.
Wir wählen dazu $c = 4$ und $n_0 = 16$. Für $n \geq n_0$ gilt (weil $\frac{\log n}{n}$ monoton fällt):

$$\begin{aligned} g(n) = 2n + 7\log n &= 2n + \tfrac{7\log n}{n} \cdot n \leq 2n + \tfrac{7\log n_0}{n_0} \cdot n \\ &= 2n + \tfrac{7\cdot 4}{16} \cdot n < 2n + 2n = 4n = c \cdot f(n). \end{aligned}$$

$O(f)$: Funktionenmenge

Beispiele: $g \in O(f)$ gilt für folgende Paare (f, g):

(a) $g(n) = 2n + 7 \log n$ und $f(n) = n$.

(b) $g(n) = 2n + 7 \log n$ und $f(n) = n^2$.

(c) $g(n) = 4n^3 - 13n^2 + 10$ und $f(n) = n^3 \log n$.

(d) $g(n) = 3 + 2 \sin n$ und $f(n) = 1$.

(e) $g(n) = 2n$ für gerade n und $g(n) = 2/n$ für ungerade n, und $f(n) = n$.

Beispiel (b) zeigt, dass es nicht verboten ist, eine „viel zu große“ Funktion $f(n)$ zu verwenden. Beispiel (d) zeigt, dass die konstante Funktion $f(n) = 1$ für $n \in \mathbb{N}$ auch zugelassen ist.

Übung: Finde für Fälle (b)–(e) passende Zahlen c und n_0. (Müssen nicht so klein wie möglich sein.)

Notation „$\cdots = O(f(n))$“

Schreib- und Sprechweisen:

(i) „$g(n) = O(f(n))$“ hat exakt dieselbe Bedeutung wie „$g \in O(f)$“.

(ii) Man spricht: „$g(n)$ **ist (in) Groß-Oh von** $f(n)$“. Achtung: **Nicht** „. . . ist gleich . . . “

(!!) „$=$“ bedeutet **nicht Gleichheit**, die Relation ist nicht symmetrisch.

(iii) Spezialfall: „$g(n) = O(1)$“. (1 steht für f mit $f(n) = 1$ für $n \in \mathbb{N}$.)

Hilfreiche Intuition:

$O(f(n))$ steht für **irgendein Element** von $O(f)$. (Eine anonyme Funktion $g \in \mathcal{F}_0^+$, von der man **nur** weiß, dass sie für genügend große n durch $c \cdot f(n)$ beschränkt ist.)

Damit ist z. B. auch Folgendes sinnvoll: $3n^3 + 2n^2 - 5n + 4 = 3n^3 + O(n^2)$.

Notation „$\cdots = O(f(n))$“

Beispiele:

(a) $2n + 7 \log n = O(n)$.

(b) Für jede Funktion g mit $0 \leq g(n) \leq 4n^2 / \log n$ für alle $n \geq 10$ gilt: $g(n) = O(n^2)$.

(c) $2n^2 - 30n + 3 = O(n^2)$.

(d) $4 \cdot 2^n + 3 \cdot n^2 = 4 \cdot 2^n + O(n^2) = O(2^n)$.

(e) $4n^2 \log n + 2n(\log n)^3 = 4n^2 \log n + O(n^2) = O(n^3)$.

(f) $4n^2 \log n + 2n(\log n)^3 = O(n^2 \log n)$.

(g) $3 + 2 \sin n = O(1)$.

Übung: Finde für Fälle (b)–(f) passende Zahlen c und n_0.

Verwendung der Notation „$g(n) = O(f(n))$“ im Kontext der Algorithmenanalyse:

$g(n) = T_{\mathcal{A}}(n)$ für einen Algorithmus $\mathcal{A}$. Oft sind solche Funktionen in ihren Details gar nicht zu ermitteln. Daher sucht man eine Abschätzung nach oben in der Form $T_{\mathcal{A}}(n) = O(f(n))$. Angenehm dabei: Man muss sich nicht um die konstanten Faktoren kümmern, also auch nicht um die Faktoren, die möglicherweise in der Definition von $T_{\mathcal{A}}(n)$ vorkommen.
Die Funktion $f(n)$ sollte einerseits eine einfache Gestalt haben, andererseits auch $T_{\mathcal{A}}(n)$ nicht unnötig überschätzen.

Konvention:

Innerhalb des $O(\ldots)$-Terms verwendet man möglichst **kleine** und/oder **möglichst einfache** Funktionen.

$17n^2/(\log n)^3 + 13n^{3/2} = O(n^2/(\log n)^3)$ (klein) oder

$17n^2/(\log n)^3 + 13n^{3/2} = O(n^2)$ (besonders einfach)

O-Notation ist auch für Funktionen definiert, die für große n eher klein sind: Konstante Funktionen, sogar Funktionen, die gegen 0 gehen.

Verwendung: Abschätzung von „kleinen“ Fehlertermen (absolut oder relativ), Abschätzung von Wahrscheinlichkeiten.

Beispiele:

(a) $\sum_{i=0}^{n} c^i < \frac{1}{1-c}$ (geometrische Reihe), also $\sum_{i=0}^{n} c^i = O(1)$, falls $0 < c < 1$ fest.

(b) $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \ln n + O(1)$. (Die n-te „harmonische Zahl“.)

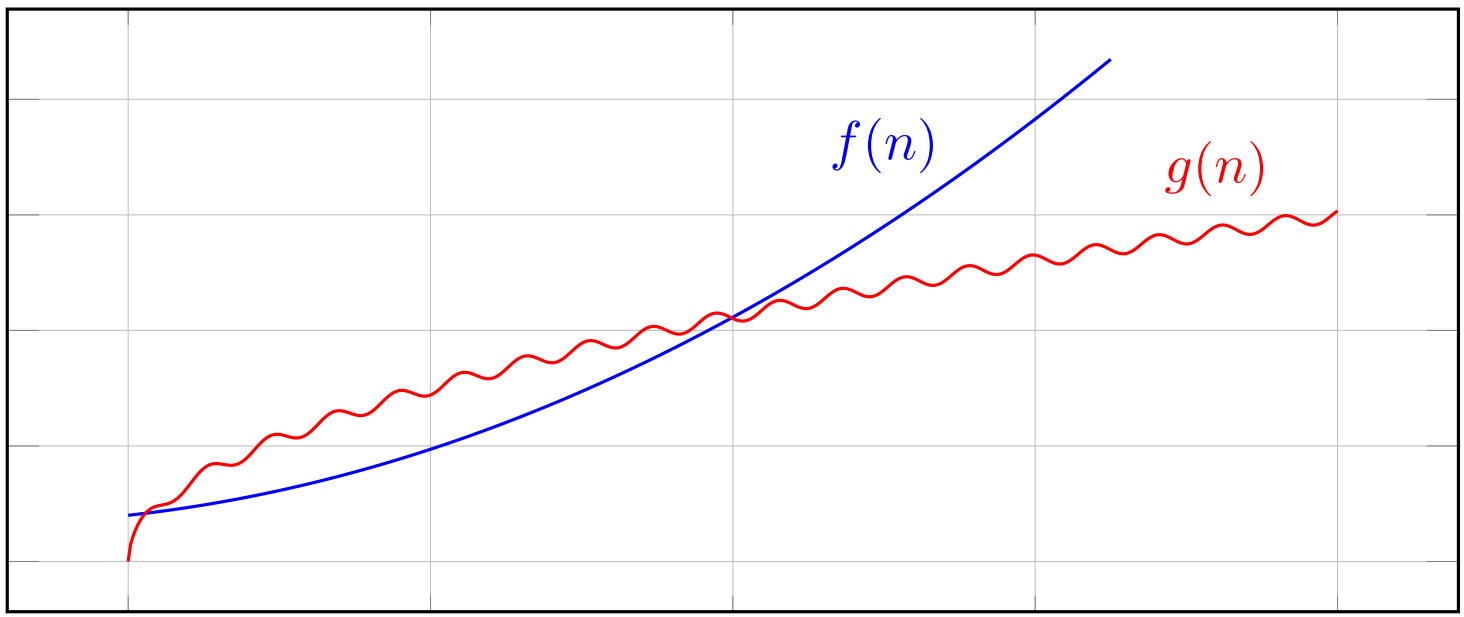
(c) $\binom{n}{2}^{-1} = \frac{2}{n(n-1)} = O(1/n^2)$.

(d) $2/n^2 + 3/n^3 = O(1/n^2)$.

(e) $2^{-n} + 3^{-n} = O(2^{-n})$.

„$o(f)$“ und „$\cdots = o(f(n))$“

„g **wächst asymptotisch strikt langsamer als** f“



Horizontale Achse: n aus $\mathbb{N}$.

„$o(f)$“ und „$\cdots = o(f(n))$“

Definition Für $f \in \mathcal{F}^+$ sei

$$o(f) := \left\{ g \in \mathcal{F}_0^+ \;\middle|\; \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0 \right\}$$

Schreibe „$g(n) = o(f(n))$“ für $g \in o(f)$. (Sprich: „$g(n)$ **ist (in) klein-oh von** $f(n)$“.)

Beispiele:

(a) $n/\log n = o(n)$.

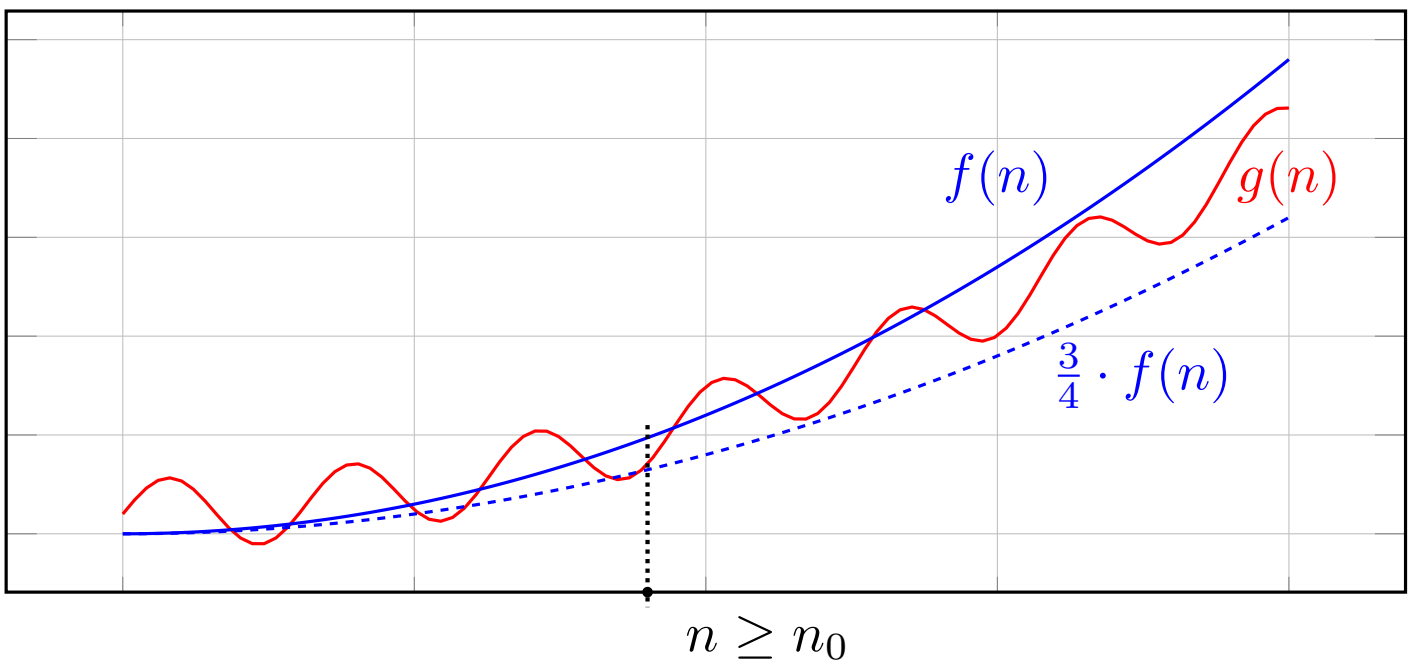
(b) $5n^2 \log n + 7n^{2.1} = o(n^{5/2})$.

(c) $n^{10} = o(2^n)$.

(d) $1/(n \log n) = o(1/n)$.

„$\Omega(f)$“ und „$\cdots = \Omega(f(n))$“

„g **wächst asymptotisch mindestens so schnell wie** f“ heißt: Es gibt eine Konstante $d > 0$, so dass $g(n) \geq d \cdot f(n)$ gilt, für genügend große n. – Im Bild: $g(n) \geq d \cdot f(n)$ für $d = \frac{3}{4}$, $n \geq n_0$.



„$\Omega(f)$“ und „$\cdots = \Omega(f(n))$“

Definition Für $f \in \mathcal{F}^+$ sei

$$\Omega(f) := \{g \in \mathcal{F}^+ \mid \exists n_0 \in \mathbb{N}\, \exists d > 0\, \forall n \geq n_0 \colon g(n) \geq d \cdot f(n)\}$$

Schreibe „$g(n) = \Omega(f(n))$“ für $g \in \Omega(f)$. (Sprich: „$g(n)$ **ist Groß-Omega von** $f(n)$“.)

Beispiele:

(a) $0.3n \log n - 20n = \Omega(n \log n)$.

(b) $\frac{1}{10}n^2 - 20n + 1 = \Omega(n^2)$.

(c) $(1 - 1/n)^n = \Omega(1)$.

(d) $\frac{1}{3n^2} - \frac{5}{n^3} = \Omega(1/n^2)$.

Nur manchmal wird benutzt: Für $f \in \mathcal{F}^+$:

$\boldsymbol{\omega(f)} := \{g \in \mathcal{F}^+ \mid \lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty\}$

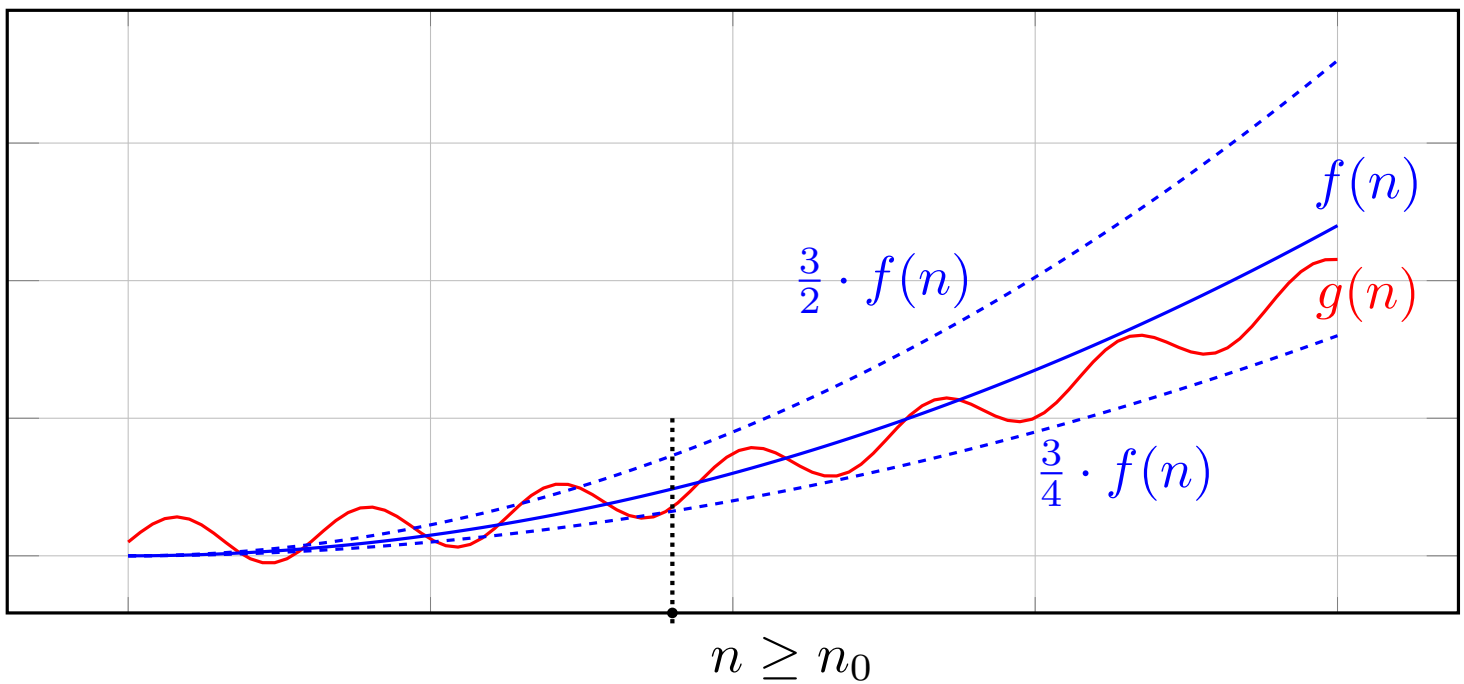
„$g(n)$ **wächst asymptotisch echt schneller als** $f(n)$.“

Schreibe „$\boldsymbol{g(n) = \omega(f(n))}$“ für $g \in \omega(f)$.
(Sprich: „$g(n)$ ist **klein-omega von** $f(n)$“.)

Klar: Für $f, g \in \mathcal{F}^+$ gilt: $g \in \omega(f) \Leftrightarrow f \in o(g)$.

„$\Theta(f)$“ und „$\cdots = \Theta(f(n))$“

„g **ist asymptotisch von derselben Größenordnung wie** f“.



„$\Theta(f)$“ und „$\cdots = \Theta(f(n))$“

Für $f \in \mathcal{F}^+$ definiere:

$\Theta(f) := O(f) \cap \Omega(f)$, d.h.

$\Theta(f) = \{g \in \mathcal{F}^+ \mid \exists n_0 \in \mathbb{N}\ \exists c, d > 0\ \forall n \geq n_0 \colon d \cdot f(n) \leq g(n) \leq c \cdot f(n)\}$

Beobachtung: $g \in \Theta(f) \Leftrightarrow O(g) = O(f)$.

Schreibe „$g(n) = \Theta(f(n))$“ für $g \in \Theta(f)$. (Sprich: „$g(n)$ **ist Theta von** $f(n)$“.)

Beispiele:

(a) $\binom{n}{2} = \Theta(n^2)$.

(b) $n \log_2 n = \Theta(n \log_{10} n) = \Theta(n \ln n)$.

(c) $(1 + 1/n)^n = \Theta(1)$.

Nur manchmal benutzt: Für $f \in \mathcal{F}^+$ definiere:

$\boldsymbol{\omega(f)} := \{g \in \mathcal{F}^+ \mid \lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty\}$

Schreibe „$\boldsymbol{g(n) = \omega(f(n))}$“ für $g \in \omega(f)$. (Sprich: „$g(n)$ ist **klein-omega von** $f(n)$“.)

Bedeutung: „$g(n)$ **wächst asymptotisch echt schneller als** $f(n)$.“

Klar: Für $f, g \in \mathcal{F}^+$ gilt: $g \in \omega(f) \Leftrightarrow f \in o(g)$.

PAUSE

Rechenregeln

Lemma 1 (Grenzwertregel)

Wenn $f \in \mathcal{F}^+$, $g \in \mathcal{F}_0^+$ und $\lim_{n\to\infty} \frac{g(n)}{f(n)} = c$ für ein $c > 0$, dann ist $g(n) = \Theta(f(n))$.

Beweis von Lemma 1: Aus $\lim_{n\to\infty} \frac{g(n)}{f(n)} = c$ folgt, dass es ein n_0 gibt, so dass für alle $n \geq n_0$ gilt: $c/2 \leq \frac{g(n)}{f(n)} \leq 2c$.

Also: $\frac{1}{2}f(n) \leq g(n) \leq 2f(n)$ für $n \geq n_0$, also $g(n) = \Omega(f(n))$ und $g(n) = O(f(n))$.

Beispiel: $g(n) = 2n + 7\log n$ und $f(n) = n$.
Es ist $\lim_{n\to\infty} \frac{2n+7\log n}{n} = \lim_{n\to\infty}(2 + \frac{7\log n}{n}) = 2$, also $2n + 7\log n = \Theta(n)$.

Beachte auch den etwas anders gelagerten Fall $c = 0$:

$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$ bedeutet: $g(n) = o(f(n))$.

(Daraus folgt $g(n) = O(f(n))$, aber es gilt **nicht** $g(n) = \Theta(f(n))$).

[Variante für Spezialist/inn/en: Wenn $\sup_{n \in \mathbb{N}} \frac{g(n)}{f(n)} < \infty$, dann folgt $g(n) = O(f(n))$.]

Rechenregeln

Wichtiger Spezialfall: Wenn g ein **Polynom** ist:

$$g(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$$

für $a_k, \ldots, a_1, a_0 \in \mathbb{R}$, wobei $a_k > 0$, und $f(n) = n^k$, dann

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \left(a_k + \frac{a_{k-1}}{n} + \cdots + \frac{a_1}{n^{k-1}} + \frac{a_0}{n^k} \right) = a_k,$$

also gilt $g(n) = \Theta(n^k)$.

Beispiel: Man erhält „durch bloßes Hinschauen“: $20n^5 - 30n^3 - 210n^2 + 15 = \Theta(n^5)$.

Rechenregeln

Lemma 2 (Transitivität)

Es seien $f, g \in \mathcal{F}^+$. Dann gilt für jedes $h \in \mathcal{F}_0^+$:

Wenn $h(n) = O(g(n))$ und $g(n) = O(f(n))$, dann gilt $h(n) = O(f(n))$.

D.h.: Wenn $g \in O(f)$, dann ist $O(g) \subseteq O(f)$.

Kurz: $h(n) = O(g(n)) = O(f(n))$; man kann schrittweise abschätzen und vereinfachen.

Achtung: In der $O(\cdot)$-Notation bedeutet „$=$“ keine Gleichheit,
die Relation ist **nicht symmetrisch**. Ihr Verhalten ist eher dem „$\leq$“ ähnlich!

Rechenregeln

Beweis von Lemma 2:
Wähle $c_1 > 0$ und n_1 so groß, dass gilt: $\forall n \geq n_1 : g(n) \leq c_1 \cdot f(n)$.
Wähle $c_2 > 0$ und n_2 so groß, dass gilt: $\forall n \geq n_2 : h(n) \leq c_2 \cdot g(n)$.
Setze $n_0 := \max\{n_1, n_2\}$.
Dann gilt für $n \geq n_0$: $h(n) \leq c_2 g(n) \leq c_2(c_1 f(n)) = (c_1 c_2) f(n)$.
Also: $h(n) = O(f(n))$.

Rechenregeln

Lemma 3 („Konstante Faktoren weglassen“)

Ist $g(n) = O(d \cdot f(n))$ und $d > 0$ beliebig, dann gilt $g(n) = O(f(n))$.

D.h.: Wenn $d > 0$, dann ist $O(d \cdot f) = O(f)$ (für die Funktionenmengen).

Analoges gilt bei $o(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$, $\omega(\cdot)$.

Beispiele:

- Nicht ~~$g(n) = O(3n \log_{2.8} n)$~~ schreiben, sondern $g(n) = O(n \log n)$;
- nicht ~~$g(n) = O(n/3)$~~ schreiben, sondern $g(n) = O(n)$;
- nicht ~~$g(n) = O(2.5)$~~ schreiben, sondern $g(n) = O(1)$.

Achtung: Bei 2^n und 3^n kommt es auf die unterschiedliche Basis an!
($2^n = o(3^n)$, also ist 3^n **nicht** in $O(2^n)$, und $O(2^n) \neq O(3^n)$.)

Rechenregeln

Beweis von Lemma 3:
Wähle $c_0 > 0$ und n_0 so groß, dass gilt: $\forall n \geq n_0 : g(n) \leq c_0 \cdot f(n)$.
Dann gilt für alle $n \geq 0$ auch $g(n) \leq (c_0/d) \cdot (d \cdot f(n))$,
d. h. $g(n) = O(d \cdot f(n))$.

Rechenregeln

Lemma 4 (Linearität und Maximumsregel)

Ist $g_1(n) = O(f_1(n))$ und $g_2(n) = O(f_2(n))$, so folgt

$$g_1(n) + g_2(n) = O(f_1(n) + f_2(n)) = O(\max\{f_1(n), f_2(n)\}).$$

Hintereinanderausführung von Algo'en: Rechenzeiten addieren, auch in O-Notation.

Kosten teurerer Teile dominieren die Kosten billigerer Teile. Analog bei $o(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$, $\omega(\cdot)$.

Beispiele: $10n \log n + 5n^2 = O[\Theta, \Omega](n \log n + n^2) = O[\Theta, \Omega](n^2)$, und
$5n/\log n + 10\sqrt{n} = o(n) + o(n) = o(n)$.

Rechenregeln

Beweis von Lemma 4:

Für $i = 1, 2$: Wähle $c_i > 0$ und n_i so groß, dass gilt: $\forall n \geq n_i : g_i(n) \leq c_i \cdot f_i(n)$.

Setze $n_0 := \max\{n_1, n_2\}$ und $c_0 := \max\{c_1, c_2\}$. Dann gilt für $n \geq n_0$:

$g_1(n) + g_2(n) \leq c_1 f_1(n) + c_2 f_2(n) \leq c_0(f_1(n) + f_2(n))$,
also $g_1(n) + g_2(n) = O(f_1(n) + f_2(n))$.

Zudem für $n \geq n_0$: $f_1(n) + f_2(n) \leq 2 \max\{f_1(n), f_2(n)\}$.

Also $g_1(n) + g_2(n) \leq 2c_0 \max\{f_1(n), f_2(n)\}$.

Rechenregeln

Lemma 5 (Weglassen von Termen „kleinerer Ordnung“)

Ist $f \in \mathcal{F}^+$ und $f(n) = \Theta(h(n))$ und $|g(n)| = o(f(n))$, so ist $f(n)+g(n) = \Theta(h(n))$.

Analog bei $O(\cdot)$, $\Omega(\cdot)$, $o(\cdot)$, $\omega(\cdot)$.

Fazit: In Summen suche den „dominierenden Term“; lasse Rest weg.

Beispiele:

$5n^2 - 10n \log n = \Theta(n^2)$.
$n^2/\log n - 10n = o(n^2)$ und $n^2/\log n - 10n = \omega(n)$.
$1/(2n) - 10/n^2 = \Theta(1/n)$.

Rechenregeln

Beweis von Lemma 5:

Wähle $C > c > 0$ und n_1 so groß, dass gilt: $\forall n \geq n_1 : c \cdot h(n) \leq f(n) \leq C \cdot h(n)$.

Wähle n_2 so groß, dass gilt: $\forall n \geq n_2 : |g(n)| \leq \frac{1}{2} \cdot f(n)$.

Setze $n_0 := \max\{n_1, n_2\}$.

Dann gilt für $n \geq n_0$: $\frac{c}{2} \cdot h(n) \leq \frac{1}{2} \cdot f(n) \leq f(n) + g(n) \leq \frac{3}{2} \cdot f(n) \leq \frac{3C}{2} \cdot h(n)$.

Rechenregeln

Lemma 6 (Multiplikationsregel)

Ist $g_1(n) = O(f_1(n))$ und $g_2(n) = O(f_2(n))$, so folgt $g_1(n)g_2(n) = O(f_1(n)f_2(n))$.

Analog bei $o(\cdot)$, $\Omega(\cdot)$, $\omega(\cdot)$, $\Theta(\cdot)$.

(Bei $o(\cdot)$ genügt als Voraussetzung: $g_1(n) = O(f_1(n))$ und $g_2(n) = o(f_2(n))$.)

Anwendung bei Algorithmenanalyse: Sei $g_1(n) = O(f_1(n))$ und $g_2(n) = O(f_2(n))$.

Wird eine **Schleife** $g_1(n)$-mal ausgeführt, mit $g_1(n) = O(f_1(n))$,
wobei die Kosten für die einmalige Ausführung höchstens $g_2(n) = O(f_2(n))$ betragen,
dann sind die Gesamtkosten $O(f_1(n)f_2(n))$.

Rechenregeln

Beweis von Lemma 6:

Für $i = 1, 2$: Wähle $c_i > 0$ und n_i so groß, dass gilt: $\forall n \geq n_i : g_i(n) \leq c_i \cdot f_i(n)$.

Setze $n_0 := \max\{n_1, n_2\}$ und $c_0 = c_1 c_2$. Dann gilt für $n \geq n_0$:

$g_1(n)g_2(n) \leq (c_1 \cdot f_1(n))(c_2 \cdot f_2(n)) \leq c_0(f_1(n)f_2(n))$,
also $g_1(n)g_2(n) = O(f_1(n)f_2(n))$.

Nun betrachte den Fall $g_1(n) = O(f_1(n))$ und $g_2(n) = o(f_2(n))$.
Das heißt: $g_1(n) \leq c_1 \cdot f_1(n)$ für $n \geq n_1$ und $\lim_{n\to\infty} \frac{g_2(n)}{f_2(n)} = 0$.
Dann haben wir: $\frac{g_1(n)g_2(n)}{f_1(n)f_2(n)} \leq c_1 \frac{g_2(n)}{f_2(n)} \to 0$, also $g_1(n)g_2(n) = o(f_1(n)f_2(n))$.

Rechenregeln

O-Notation in **Summen**

Beispiel: Wir wollen die Rechenzeit einer Schleife analysieren, wobei der Schleifenrumpf bei verschiedenen Durchläufen unterschiedlich lange braucht, etwa:

Input der Größe n erzeugt $\leq n$ Durchläufe, Kosten beim i-ten Durchlauf: $a_{n,i} \leq \frac{10n}{i}$.

Maximum der Kosten: $\leq 10n/1 = O(n)$ im ersten Durchlauf.

Wenn wir einheitlich $a_{n,i} = O(n)$ abschätzen, liefert die Multiplikationsregel Gesamtkosten $O(n) \cdot O(n) = O(n \cdot n) = O(n^2)$.

Aber: $\frac{n}{1} + \frac{n}{2} + \cdots + \frac{n}{n} = n \cdot (\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n}) = nH_n$, für $H_n = n$-te harmonische Zahl.

Wir werden später sehen: $H_n \leq 1 + \ln n = O(\log n)$.

Daher Vermutung: Gesamtkosten sind nur $O(n \log n)$.

Rechenregeln

Lemma 7 (O-Notation in Summen)

Sei $g\colon \mathbb{N} \times \mathbb{N} \to \mathbb{R}_0^+$. Für jedes $n \in \mathbb{N}$ sei $a_{n,1}, \ldots, a_{n,t(n)}$ Zahlenfolge mit

$$\exists \boldsymbol{c} > 0\ \exists n_0\ \forall n \geq n_0\ \forall 1 \leq i \leq t(n):\ 0 \leq a_{n,i} \leq \boldsymbol{c} \cdot g(n,i).$$

(„$a_{n,i}$ ist **uniform** durch ein Vielfaches von $g(n,i)$ beschränkt“; kurz: $a_{n,i} = O(g(n,i))$.)

Dann gilt (die Variable ist n):

$$S(n) := \sum_{1 \leq i \leq t(n)} a_{n,i} = O\Big(\sum_{1 \leq i \leq t(n)} g(n,i) \Big).$$

Beispiel: Im oben diskutierten Beispiel hat bei Eingabegröße n der i-te Schleifendurchlauf Kosten $a_{n,i} \leq 10n/i$, für $1 \leq i \leq t(n) = n$. Wir wählen $c = 10$, $n_0 = 1$ und $g(n,i) = n/i$. Dann sind nach der Summationsformel die Gesamtkosten

$$\sum_{1 \leq i \leq t(n)} a_{n,i} = O\Big(\sum_{1 \leq i \leq n} \frac{n}{i}\Big) = O\Big(n \cdot \Big(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}\Big)\Big) = O(n \log n).$$

Achtung! Wenn es für jedes $n \geq n_0$ ein (eigenes) c_n gibt mit

$$\forall 1 \leq i \leq t(n) : 0 \leq a_{n,i} \leq c_n \cdot g(n,i),$$

($a_{n,i}$ **nicht-uniform** beschränkt),dann kann man **nicht** folgern, dass gilt:

$$\sum_{1 \leq i \leq t(n)} a_{n,i} = O\Big(\sum_{1 \leq i \leq t(n)} g(n,i)\Big).$$

PAUSE

Verwendung der O-Notation bei der Analyse eines Algorithmus

Beispiel Insertionsort, Eingabe $x = (a_1, \ldots, a_n)$, Größe n:

Mit Multiplikationsregel: Im Durchlauf für i hat die Schleife in Zeilen (4)–(6) Kosten $\Theta(1) + k_i \cdot \Theta(1) = \Theta(1 + k_i)$ (wobei $k_i = \#\{(l, i) \mid 1 \leq l < i \leq n, key(a_l) > key(a_i)\}$).

Mit Additionsregel: Zeilen (1)–(7) für den Durchlauf für i haben Kosten $\Theta(1) + \Theta(1 + k_i) + \Theta(1) = \Theta(1 + k_i)$.

Mit Summationsregel: Der gesamte Ablauf hat Kosten

$$t_{\mathsf{IS}}(x) = \sum_{2 \leq i \leq n} \Theta(1 + k_i) = \Theta(n + \sum_{\mathbf{2 \leq i \leq n}} \boldsymbol{k_i}) = \Theta(n + \boldsymbol{F_x}) = O(n^2).$$

$\boldsymbol{F_x}$: Anzahl der „Fehlstände“ (l, i), wo $l < i$ und $key(a_l) > key(a_i)$. Wissen schon: $F_x \leq \binom{n}{2}$.

Ignorieren konstanter Faktoren: Bedeutung

Beispiel:

Ein anderer Sortieralgorithmus ist **Mergesort** (siehe AuP-Vorlesung).

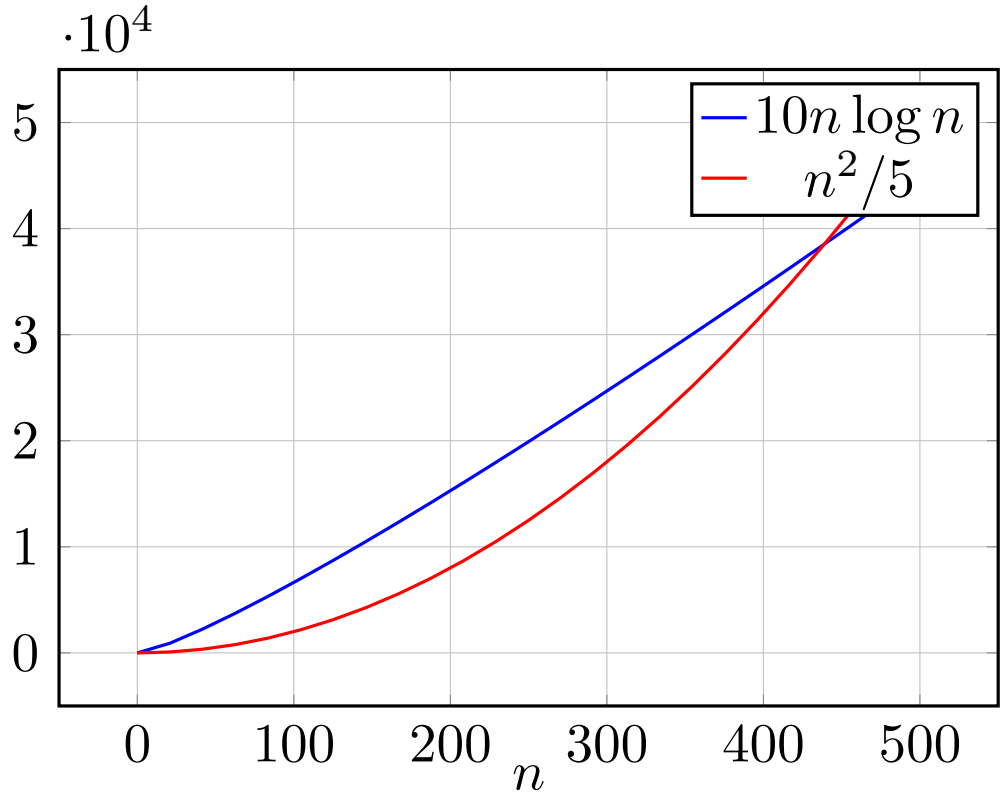
Rechenzeit/Kosten: $T_{\text{Mergesort}}(n) = \Theta(n \log n)$.

Auch wenn wir die Konstanten nicht kennen, können wir Folgendes sagen:

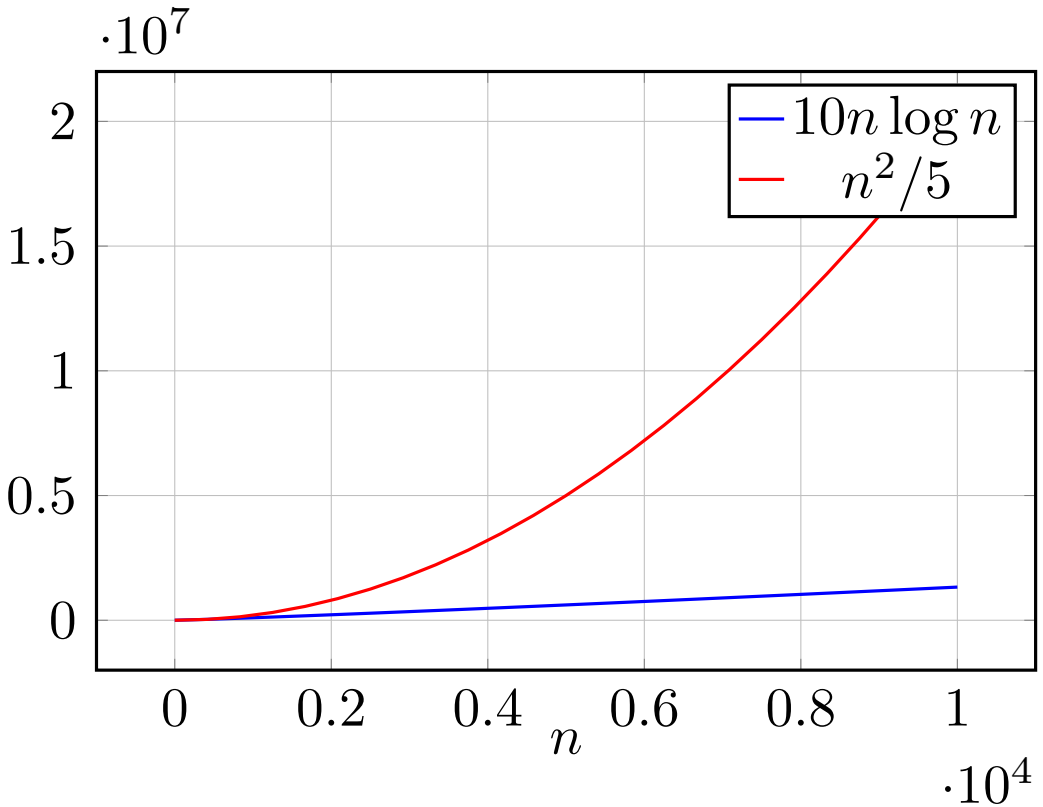
Mergesort wird „für genügend große Inputs“ schneller sein
als Insertionsort (mit $T_{\text{IS,worst}}(n) = \Theta(n^2)$).

(Die Praxis zeigt: Für viele Algorithmenpaare tritt ein solcher „asymptotischer“ Unterschied schon für nicht allzugroße Werte von n zu Tage.)

Die folgenden Bilder vergleichen Funktionen in $\Theta(n^2)$ und $\Theta(n \log n)$ für bestimmte versteckte Konstanten. Für kleine n scheint die $\Theta(n^2)$-Funktion im Vorteil zu sein, aber für größere n gewinnt die $\Theta(n \log n)$-Funktion klar. Der Überschneidungspunkt hängt von den versteckten Konstanten ab.



Für $n \leq 430$ ist $n^2/5$ „besser als“ $10n \log n$.



Für n in $[10^3, 10^4]$ übernimmt das asymptotische Verhalten: $10n \log n$ **viel besser** als $n^2/5$.

Wachstumsordnungen

„Einfache“ Funktionen mit unterschiedlich schnellem Wachstum bilden **Hierarchie**.
Beispiele (s. Übung):

$$
\begin{array}{lll}
f_0(n) = 1, & & \\
f_1(n) = \log\log n, & f_2(n) = \log n, & f_3(n) = (\log n)^2, \\
f_4(n) = \sqrt{n}, & f_5(n) = \frac{n}{\log n}, & f_6(n) = n, \\
f_7(n) = n \log n, & f_8(n) = n(\log n)^2, & f_9(n) = n^{3/2}, \\
f_{10}(n) = n^2, & f_{11}(n) = n^3, & f_{12}(n) = 2^n, \\
f_{13}(n) = n^5 \cdot 2^n, & f_{14}(n) = e^n, & f_{15}(n) = 3^n \\
f_{16}(n) = (n/e)^n, & f_{17}(n) = n!, & f_{18}(n) = n^n.
\end{array}
$$

Es gilt: $\lim_{n\to\infty} f_i(n)/f_{i+1}(n) = 0$, also $f_i(n) = o(f_{i+1}(n))$, für $0 \leq i \leq 17$.

Mit $f_{-i}(n) := 1/f_i(n)$, für $1 \leq i \leq 18$, gilt dies sogar für $-18 \leq i \leq 17$.

Vergleich einiger Wachstumsordnungen

	n							
$t_{\mathcal{A}}(n)$	10	100	1000	10^4	10^5	10^6	10^7	10^8
$\log n$	33ns	66ns	0.1μs	0.1μs	0.2μs	0.2μs	0.2μs	0.3μs
$\sqrt{n}$	32ns	0.1μs	0.3μs	1μs	3.1μs	10μs	31μs	0.1ms
n	100ns	1μs	10μs	0.1ms	1ms	10ms	0.1s	1s
$n \log n$	0.3μs	6.6μs	0.1ms	1.3ms	16ms	0.2s	2.3s	27s
$n^{3/2}$	0.3μs	10μs	0.3ms	10ms	0.3s	10s	5.2m	2.7h
n^2	1μs	0.1ms	10ms	1s	1.7m	2.8h	11d	3.2y
n^3	10μs	10ms	10s	2.8h	115d	317y	$3.2{\cdot}10^5$y	
1.1^n	26ns	0.1ms	$7.8{\cdot}10^{25}$y					
2^n	10μs	$4.0{\cdot}10^{14}$y						
$n!$	36ms	$3.0{\cdot}10^{142}$y						
n^n	1.7m	$3.2{\cdot}10^{184}$y						

Zeit für eine (Elementar-)Operation: 10 ns

Hilft ein schnellerer Rechner?

Wie wächst die maximal mögliche Eingabegröße, wenn der Rechner um den Faktor 10 schneller wird?

$T_{\mathcal{A}}(n)$	Max_{alt}	Max_{NEU}
$\log n$	n	n^{10}
$\sqrt{n}$	n	$100n$
n	n	$10n$
$n \log n$	n	$10n \cdot \frac{\log n}{\log n + \log 10}$
$n^{3/2}$	n	$4.64n$
n^2	n	$3.16n$
n^3	n	$2.15n$
n^k	n	$10^{1/k}n$
c^n	n	$n + \log_c 10$
$n!$	um von n auf $n+1$ zu kommen, braucht man einen um den Faktor n schnelleren Rechner	
n^n		

Hilft ein schnellerer Rechner?

Angetrieben durch die höheren **Rechengeschwindigkeiten** und die noch rascher wachsenden **Speicherkapazitäten** wächst in vielen Bereichen die Menge der zu bewältigenden Daten noch schneller als die Rechnergeschwindigkeit.

Beispiele:
Indizes der Internet-Suchmaschinen.
Routenplaner, Fahrplan-, Ticketsysteme der Bahn.
Computergraphik (Szenen aus zig Milliarden von Elementen!)

Um die steigende Rechnergeschwindigkeit wenigstens einigermaßen ausnutzen zu können, sind Algorithmen mit polynomieller Laufzeit (kleine Exponenten) wesentlich.

Unser Ziel in dieser Vorlesung: Bekannte hocheffiziente Algorithmen kennenlernen.

Techniken für den Entwurf und die Analyse effizienter Algorithmen bereitstellen.

Ende 2. Vorlesung