4. Algorithmen und Datenstrukturen

Algorithmen und Datenstrukturen, Übersicht
[Cormen et al, Kap. 1; Ottman/Widmayer, Kap. 1.1]

Algorithmus

Algorithmus

Wohldefinierte Berechnungsvorschrift, welche aus Eingabedaten (**input**) Ausgabedaten (**output**) berechnet.

Beispielproblem: Sortieren

Input: Eine Folge von n Zahlen (vergleichbaren Objekten) $(a_1, a_2, \ldots, a_n)$

Beispielproblem: Sortieren

Input: Eine Folge von n Zahlen (vergleichbaren Objekten) $(a_1, a_2, \ldots, a_n)$

Output: Eine Permutation $(a'_1, a'_2, \ldots, a'_n)$ der Folge $(a_i)_{1 \leq i \leq n}$, so dass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Beispielproblem: Sortieren

Input: Eine Folge von n Zahlen (vergleichbaren Objekten) $(a_1, a_2, \ldots, a_n)$
Output: Eine Permutation $(a'_1, a'_2, \ldots, a'_n)$ der Folge $(a_i)_{1 \leq i \leq n}$, so dass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Mögliche Eingaben

$(1, 7, 3)$, $(15, 13, 12, -0.5)$, $(999, 998, 997, 996, \ldots, 2, 1)$, (1), $()$ …

Beispielproblem: Sortieren

Input: Eine Folge von n Zahlen (vergleichbaren Objekten) $(a_1, a_2, \ldots, a_n)$
Output: Eine Permutation $(a'_1, a'_2, \ldots, a'_n)$ der Folge $(a_i)_{1 \leq i \leq n}$, so dass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Mögliche Eingaben

$(1, 7, 3)$, $(15, 13, 12, -0.5)$, $(999, 998, 997, 996, \ldots, 2, 1)$, (1), $()$ …

Jedes Beispiel erzeugt eine **Probleminstanz**.

Die Performanz (Geschwindigkeit) des Algorithmus hängt üblicherweise ab von der Probleminstanz. Es gibt oft „gute" und "schlechte" Instanzen.

Daher betrachten wir Algorithmen manchmal **„im Durchschnitt"** und meist **„im schlechtesten Fall"**.

Beispiele für Probleme in der Algorithmik

- Tabellen und Statistiken: Suchen, Auswählen und Sortieren

Beispiele für Probleme in der Algorithmik

- Tabellen und Statistiken: Suchen, Auswählen und Sortieren
- Routenplanung: Kürzeste Wege Algorithmus, Heap Datenstruktur

Beispiele für Probleme in der Algorithmik

- Tabellen und Statistiken: Suchen, Auswählen und Sortieren
- Routenplanung: Kürzeste Wege Algorithmus, Heap Datenstruktur
- DNA Matching: Dynamic Programming

Beispiele für Probleme in der Algorithmik

- Tabellen und Statistiken: Suchen, Auswählen und Sortieren
- Routenplanung: Kürzeste Wege Algorithmus, Heap Datenstruktur
- DNA Matching: Dynamic Programming
- Auswertungsreihenfolge: Topologische Sortierung

Beispiele für Probleme in der Algorithmik

- Tabellen und Statistiken: Suchen, Auswählen und Sortieren
- Routenplanung: Kürzeste Wege Algorithmus, Heap Datenstruktur
- DNA Matching: Dynamic Programming
- Auswertungsreihenfolge: Topologische Sortierung
- Autovervollständigung: Wörterbücher/ Bäume

Beispiele für Probleme in der Algorithmik

- Tabellen und Statistiken: Suchen, Auswählen und Sortieren
- Routenplanung: Kürzeste Wege Algorithmus, Heap Datenstruktur
- DNA Matching: Dynamic Programming
- Auswertungsreihenfolge: Topologische Sortierung
- Autovervollständigung: Wörterbücher/Bäume
- Schnelles Nachschlagen : Hash-Tabellen

Beispiele für Probleme in der Algorithmik

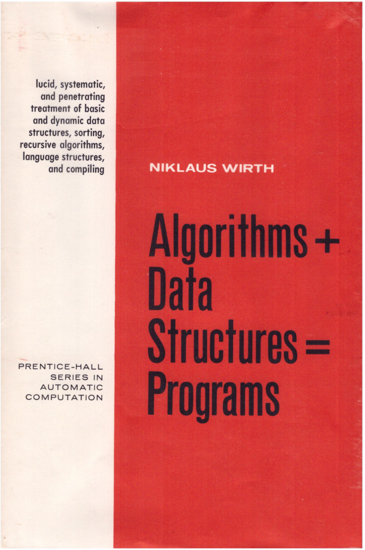
- Tabellen und Statistiken: Suchen, Auswählen und Sortieren
- Routenplanung: Kürzeste Wege Algorithmus, Heap Datenstruktur
- DNA Matching: Dynamic Programming
- Auswertungsreihenfolge: Topologische Sortierung
- Autovervollständigung: Wörterbücher/ Bäume
- Schnelles Nachschlagen : Hash-Tabellen
- Der Handlungsreisende: Dynamische Programmierung, Minimal aufspannender Baum, Simulated Annealing,

Charakteristik

- Extrem grosse Anzahl potentieller Lösungen
- Praktische Anwendung

Datenstrukturen

- Eine Datenstruktur **organisiert Daten** so in einem Computer, dass man sie (in den darauf operierenden Algorithmen) **effizient nutzen** kann.
- Programme = Algorithmen + Datenstrukturen.



Effizienz

- Wären Rechner unendlich schnell und hätten unendlich viel Speicher …
- … dann bräuchten wir die Theorie der Algorithmen (nur) für Aussagen über Korrektheit (incl. Terminierung).

Effizienz

- Wären Rechner unendlich schnell und hätten unendlich viel Speicher …
- … dann bräuchten wir die Theorie der Algorithmen (nur) für Aussagen über Korrektheit (incl. Terminierung).

Realität: Ressourcen sind beschränkt und nicht umsonst:

- Rechenzeit $\to$ Effizienz
- Speicherplatz $\to$ Effizienz

Eigentlich geht es in diesem Kurs nur um Effizienz.

Schwierige Probleme

- NP-vollständige Probleme: Keine bekannte effiziente Lösung (Existenz einer effizienten Lösung ist zwar sehr unwahrscheinlich – es ist aber unbewiesen, dass es keine gibt!)
- Beispiel: Travelling Salesman Problem

In diesem Kurs beschäftigen wir uns *hauptsächlich* **mit Problemen, die effizient (in Polynomialzeit) lösbar sind.**

5. Effizienz von Algorithmen

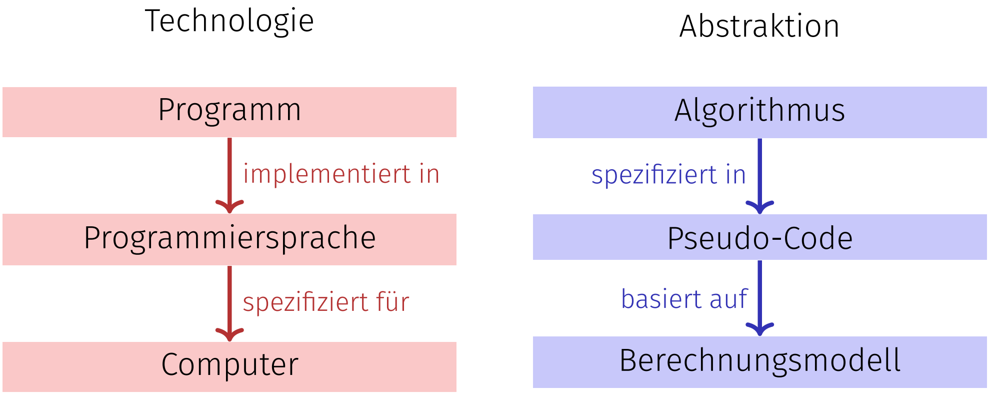
Effizienz von Algorithmen, Random Access Machine Modell, Funktionenwachstum, Asymptotik [Cormen et al, Kap. 2.2,3,4.2-4.4 | Ottman/Widmayer, Kap. 1.1]

Effizienz von Algorithmen

Ziele

- Laufzeitverhalten eines Algorithmus maschinenunabhängig quantifizieren.
- Effizienz von Algorithmen vergleichen.
- Abhängigkeit von der Eingabegrösse verstehen.

Programme und Algorithmen



Technologiemodell

Random Access Machine (RAM) Model

- Ausführungsmodell: Instruktionen werden der Reihe nach (auf einem Prozessorkern) ausgeführt.

Technologiemodell

Random Access Machine (RAM) Model

- Ausführungsmodell: Instruktionen werden der Reihe nach (auf einem Prozessorkern) ausgeführt.
- Speichermodell: Konstante Zugriffszeit (grosses Array)

Technologiemodell

Random Access Machine (RAM) Model

- Ausführungsmodell: Instruktionen werden der Reihe nach (auf einem Prozessorkern) ausgeführt.
- Speichermodell: Konstante Zugriffszeit (grosses Array)
- Elementare Operationen: Rechenoperation $(+,-,\cdot,\ldots)$, Vergleichsoperationen, Zuweisung / Kopieroperation auf Maschinenworten (Registern), Flusskontrolle (Sprünge)

Technologiemodell

Random Access Machine (RAM) Model

- Ausführungsmodell: Instruktionen werden der Reihe nach (auf einem Prozessorkern) ausgeführt.
- Speichermodell: Konstante Zugriffszeit (grosses Array)
- Elementare Operationen: Rechenoperation $(+,-,\cdot,\ldots)$, Vergleichsoperationen, Zuweisung / Kopieroperation auf Maschinenworten (Registern), Flusskontrolle (Sprünge)
- Einheitskostenmodell: elementare Operation hat Kosten 1.

Technologiemodell

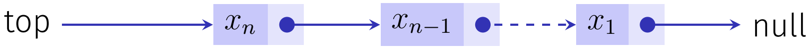
Random Access Machine (RAM) Model

- Ausführungsmodell: Instruktionen werden der Reihe nach (auf einem Prozessorkern) ausgeführt.
- Speichermodell: Konstante Zugriffszeit (grosses Array)
- Elementare Operationen: Rechenoperation $(+,-,\cdot,\ldots)$, Vergleichsoperationen, Zuweisung / Kopieroperation auf Maschinenworten (Registern), Flusskontrolle (Sprünge)
- Einheitskostenmodell: elementare Operation hat Kosten 1.
- Datentypen: Fundamentaltypen wie grössenbeschränkte Ganzzahl oder Fliesskommazahl.

Für dynamische Datenstrukturen

Pointer Machine Modell

- Objekte beschränkter Grösse können dynamisch erzeugt werden in konstanter Zeit 1.
- Auf Felder (mit Wortgrösse) der Objekte kann in konstanter Zeit 1 zugegriffen werden.



Asymptotisches Verhalten

Genaue Laufzeit eines Algorithmus lässt sich selbst für kleine Eingabedaten kaum voraussagen.

- Betrachten das asymptotische Verhalten eines Algorithmus.
- Ignorieren alle konstanten Faktoren.

Eine Operation mit Kosten 20 ist genauso gut wie eine mit Kosten 1. Lineares Wachstum mit Steigung 5 ist genauso gut wie lineares Wachstum mit Steigung 1.

5.2 Funktionenwachstum

$\mathcal{O}$, Θ, Ω [Cormen et al, Kap. 3; Ottman/Widmayer, Kap. 1.1]

Oberflächlich

Verwende die asymptotische Notation zur Kennzeichnung der Laufzeit von Algorithmen

Wir schreiben $\Theta(n^2)$ und meinen, dass der Algorithmus sich für grosse n wie n^2 verhält: verdoppelt sich die Problemgrösse, so vervierfacht sich die Laufzeit.

Genauer: Asymptotische obere Schranke

Gegeben: Funktion $g : \mathbb{N} \to \mathbb{R}$.

Definition:[1]

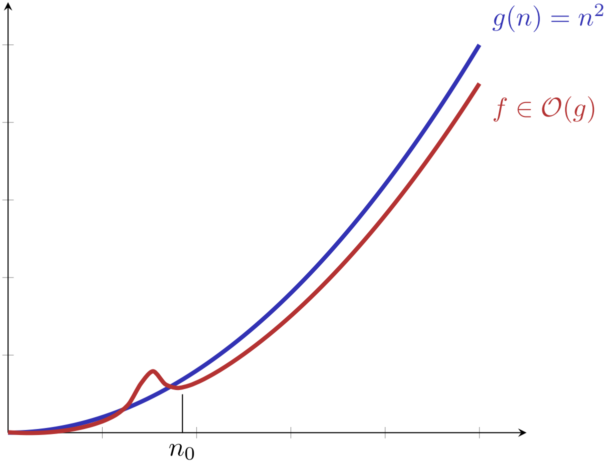
$$\begin{aligned} \mathcal{O}(g) = \{f : \mathbb{N} \to \mathbb{R} | \\ \exists\, c > 0, \exists n_0 \in \mathbb{N} : \\ \forall\, n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\} \end{aligned}$$

Schreibweise:

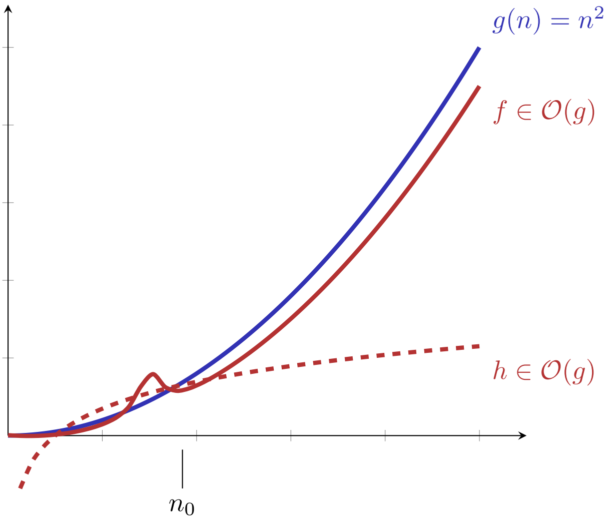
$$\mathcal{O}(g(n)) := \mathcal{O}(g(\cdot)) = \mathcal{O}(g).$$

[1]Ausgesprochen: Menge aller reellwertiger Funktionen $f : \mathbb{N} \to \mathbb{R}$ für die gilt: es gibt ein (reellwertiges) $c > 0$ und ein $n_0 \in \mathbb{N}$ so dass $0 \leq f(n) \leq n \cdot g(n)$ für alle $n \geq n_0$.

Anschauung



Anschauung



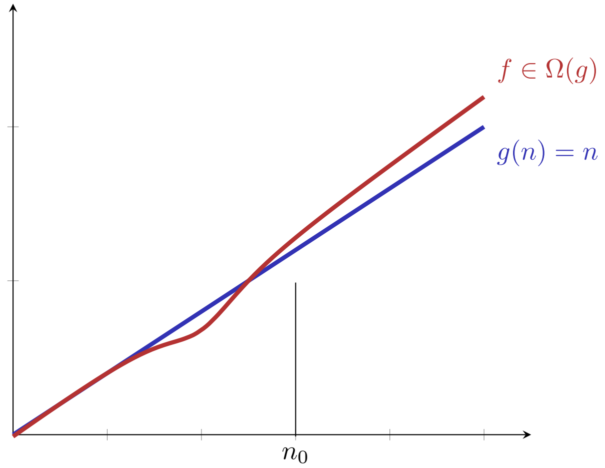
Umkehrung: Asymptotische untere Schranke

Gegeben: Funktion $g : \mathbb{N} \to \mathbb{R}$.

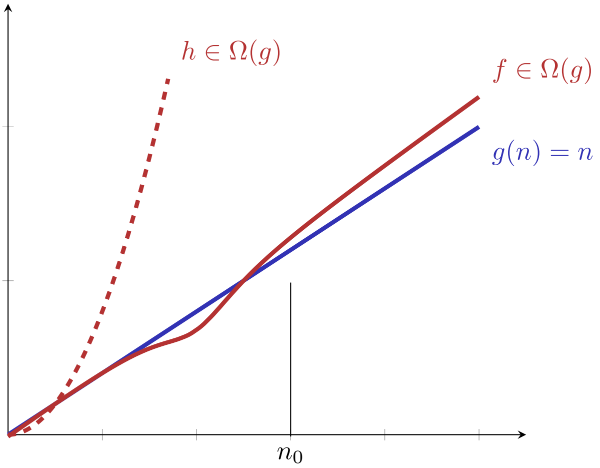
Definition:

$$\begin{aligned}\Omega(g) = \{f : \mathbb{N} \to \mathbb{R} | \\ &\exists\, c > 0, \exists n_0 \in \mathbb{N} : \\ &\forall\, n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n)\}\end{aligned}$$

Beispiel



Beispiel



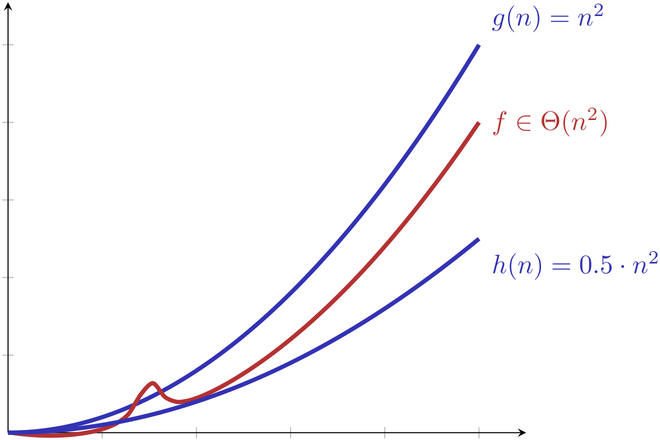
Asymptotisch scharfe Schranke

Gegeben Funktion $g : \mathbb{N} \to \mathbb{R}$.
Definition:

$$\Theta(g) := \Omega(g) \cap \mathcal{O}(g).$$

Einfache, geschlossene Form: Übung.

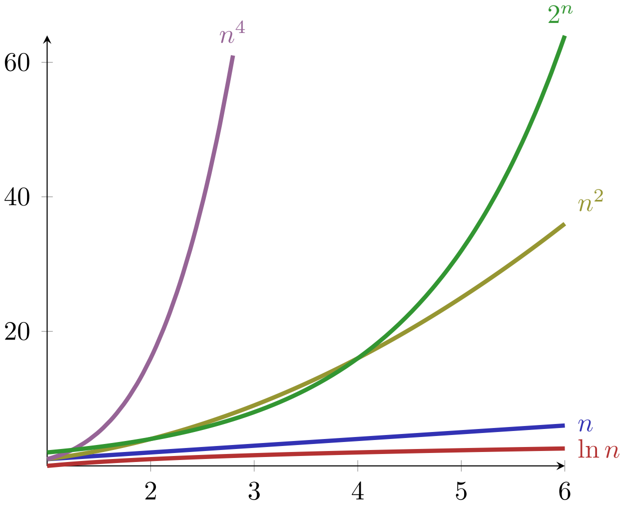
Beispiel



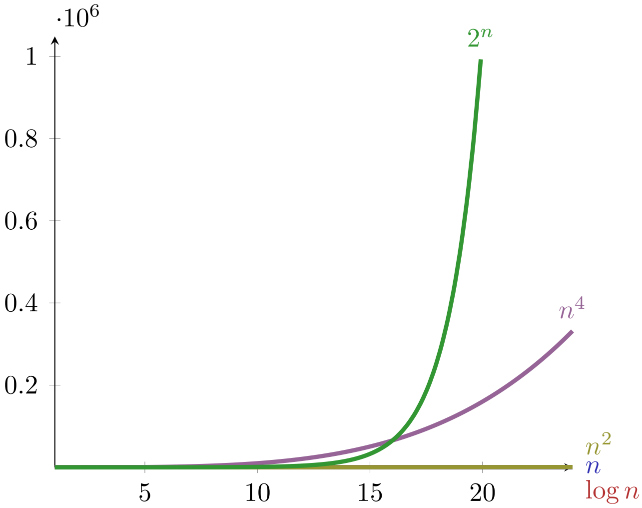
Wachstumsbezeichnungen

$\mathcal{O}(1)$	beschränkt	Array-Zugriff
$\mathcal{O}(\log\log n)$	doppelt logarithmisch	Binäre sortierte Suche interpoliert
$\mathcal{O}(\log n)$	logarithmisch	Binäre sortierte Suche
$\mathcal{O}(\sqrt{n})$	wie die Wurzelfunktion	Primzahltest (naiv)
$\mathcal{O}(n)$	linear	Unsortierte naive Suche
$\mathcal{O}(n\log n)$	superlinear / loglinear	Gute Sortieralgorithmen
$\mathcal{O}(n^2)$	quadratisch	Einfache Sortieralgorithmen
$\mathcal{O}(n^c)$	polynomial	Matrixmultiplikation
$\mathcal{O}(2^n)$	exponentiell	Travelling Salesman Dynamic Programming
$\mathcal{O}(n!)$	faktoriell	Travelling Salesman naiv

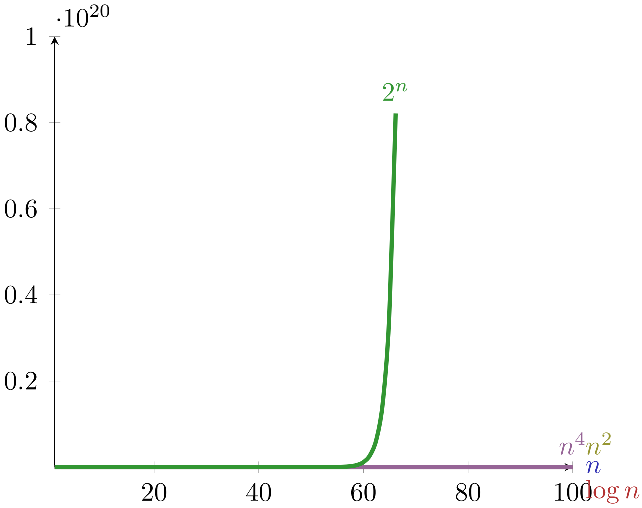
Kleine n



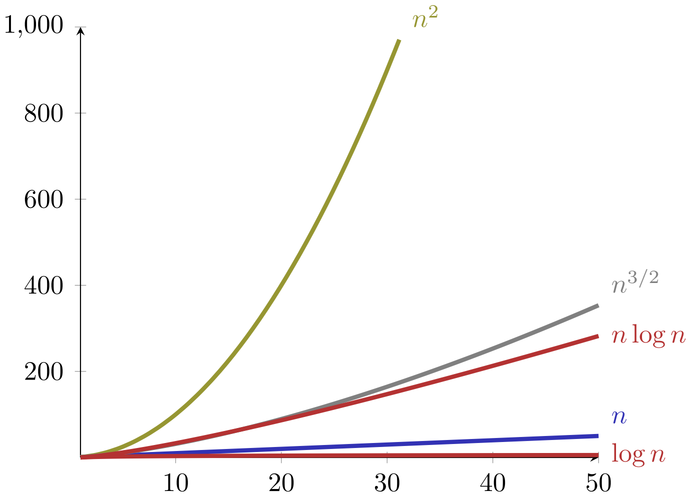
Grössere n



“Grosse” n



Logarithmen!



Zeitbedarf

Annahme: 1 Operation = $1\mu s$.

Problemgrösse	1	100	10000	10^6	10^9
$\log_2 n$	$1\mu s$				
n	$1\mu s$				
$n \log_2 n$	$1\mu s$				
n^2	$1\mu s$				
2^n	$1\mu s$				

Zeitbedarf

Annahme: 1 Operation = $1\mu s$.

Problemgrösse	1	100	10000	10^6	10^9
$\log_2 n$	$1\mu s$				
n	$1\mu s$	$100\mu s$	$1/100s$	$1s$	17 Minuten
$n \log_2 n$	$1\mu s$				
n^2	$1\mu s$				
2^n	$1\mu s$				

Zeitbedarf

Annahme: 1 Operation = $1\mu s$.

Problemgrösse	1	100	10000	10^6	10^9
$\log_2 n$	$1\mu s$				
n	$1\mu s$	$100\mu s$	$1/100s$	$1s$	17 Minuten
$n \log_2 n$	$1\mu s$				
n^2	$1\mu s$	$1/100s$	1.7 Minuten	11.5 Tage	317 Jahrhund.
2^n	$1\mu s$				

Zeitbedarf

Annahme: 1 Operation = $1\mu s$.

Problemgrösse	1	100	10000	10^6	10^9
$\log_2 n$	$1\mu s$	$7\mu s$	$13\mu s$	$20\mu s$	$30\mu s$
n	$1\mu s$	$100\mu s$	$1/100s$	$1s$	17 Minuten
$n \log_2 n$	$1\mu s$				
n^2	$1\mu s$	$1/100s$	1.7 Minuten	11.5 Tage	317 Jahrhund.
2^n	$1\mu s$				

Zeitbedarf

Annahme: 1 Operation = $1\mu s$.

Problemgrösse	1	100	10000	10^6	10^9
$\log_2 n$	$1\mu s$	$7\mu s$	$13\mu s$	$20\mu s$	$30\mu s$
n	$1\mu s$	$100\mu s$	$1/100s$	$1s$	17 Minuten
$n \log_2 n$	$1\mu s$	$700\mu s$	$13/100\mu s$	$20s$	8.5 Stunden
n^2	$1\mu s$	$1/100s$	1.7 Minuten	11.5 Tage	317 Jahrhund.
2^n	$1\mu s$				

Zeitbedarf

Annahme: 1 Operation = $1\mu s$.

Problemgrösse	1	100	10000	10^6	10^9
$\log_2 n$	$1\mu s$	$7\mu s$	$13\mu s$	$20\mu s$	$30\mu s$
n	$1\mu s$	$100\mu s$	$1/100s$	$1s$	17 Minuten
$n \log_2 n$	$1\mu s$	$700\mu s$	$13/100\mu s$	$20s$	8.5 Stunden
n^2	$1\mu s$	$1/100s$	1.7 Minuten	11.5 Tage	317 Jahrhund.
2^n	$1\mu s$	10^{14} Jahrh.	$\approx \infty$	$\approx \infty$	$\approx \infty$

Zur Notation

Übliche informelle Schreibweise

$$f = \mathcal{O}(g)$$

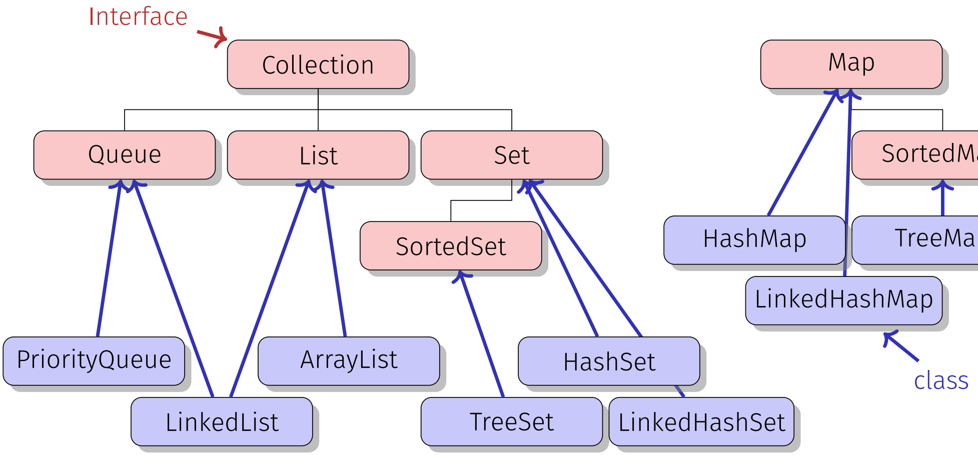
ist zu verstehen als $f \in \mathcal{O}(g)$.
Es gilt nämlich

$$f_1 = \mathcal{O}(g), f_2 = \mathcal{O}(g) \not\Rightarrow f_1 = f_2!$$

$n = \mathcal{O}(n^2), n^2 = \mathcal{O}(n^2)$ aber natürlich $n \neq n^2$.

Wir vermeiden die informelle „=" Schreibweise, wo sie zu Mehrdeutigkeiten führen könnte.

Erinnerung: Java Collections / Maps

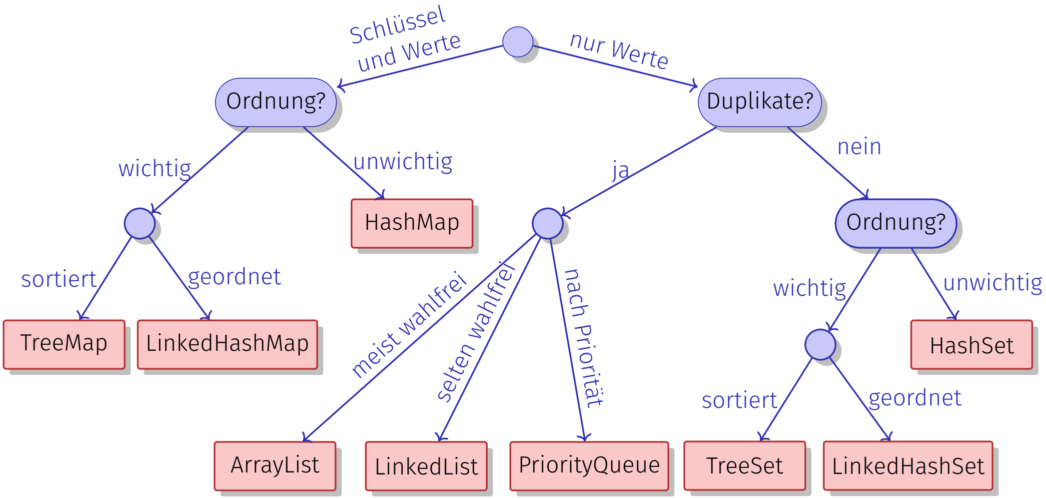


`ArrayList` versus `LinkedList`

Laufzeitmessungen für 10000 Operationen (auf [code]expert)

	`ArrayList`	`LinkedList`
Einfügen am Ende	$469\mu s$	$1787\mu s$
Einfügen am Anfang	$37900\mu s$	$761\mu s$
Iterieren	$1840\mu s$	$2050\mu s$
Wahlfreier Zugriff	$426\mu s$	$110600\mu s$
Einfügen in der Mitte	31ms	301ms
Enthält (erfolgreich)	38ms	141ms
Enthält (erfolglos)	228ms	1080ms
Entfernen am Ende	$648\mu s$	$757\mu s$
Entfernen am Anfang	$58075\mu s$	$609\mu s$

Erinnerung: Entscheidungshilfe



Asymptotische Laufzeiten (Java)

Mit unserer neuen Sprache $(\Omega, \mathcal{O}, \Theta)$ können wir das **Verhalten der Datenstrukturen und ihrer Algorithmen präzisieren.**

Asymptotische Laufzeiten (Vorgriff!)

Datenstruktur	Wahlfreier Zugriff	Einfügen	Nächstes	Einfügen nach Element	Suchen
`ArrayList`	$\Theta(1)$	$\Theta(1)\ A$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
`LinkedList`	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
`TreeSet`	–	$\Theta(\log n)$	$\Theta(\log n)$	–	$\Theta(\log n)$
`HashSet`	–	$\Theta(1)\ P$	–	–	$\Theta(1)\ P$

A= amortisiert, P= erwartet, sonst schlechtester Fall („worst case”)

Asymptotische Laufzeiten (Python)

Asymptotische Laufzeiten

Datenstruktur	Wahlfreier Zugriff	Einfügen	Iteration	Einfügen nach Element	Suchen **`x in S`**
`list`	$\Theta(1)$	$\Theta(1)\ A$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
`set`	–	$\Theta(1)\ P$	$\Theta(n)$	–	$\Theta(1)\ P$
`dict`	–	$\Theta(1)\ P$	$\Theta(n)$	–	$\Theta(1)\ P$

A= amortisiert, P= erwartet, sonst schlechtester Fall („worst case”)