ETH zürich

Felix Friedrich

Datenstrukturen und Algorithmen

Vorlesung am D-MATH der ETH Zürich

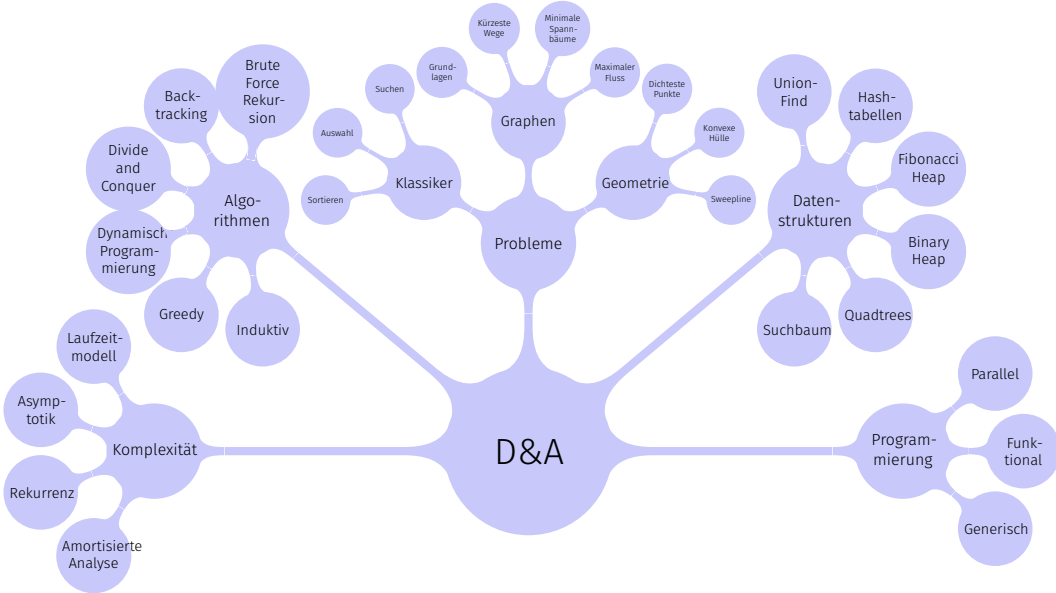
Frühjahr 2022

1. Einführung

Überblick, Algorithmen und Datenstrukturen, Korrektheit, erstes Beispiel

Ziele des Kurses

- Verständnis des Entwurfs und der Analyse grundlegender Algorithmen und Datenstrukturen.
- Vertiefter Einblick in ein modernes Programmiermodell (mit C++).
- Wissen um Chancen, Probleme und Grenzen des parallelen und nebenläufigen Programmierens.


D&A
Algo-rithmen
Brute Force Rekur-sion
Back-tracking
Divide and Conquer
Dynamisch. Program-mierung
Greedy
Induktiv
Probleme
Graphen
Kürzeste Wege
Minimale Spann-bäume
Maximaler Fluss
Dichteste Punkte
Grund-lagen
Suchen
Klassiker
Auswahl
Sortieren
Geometrie
Konvexe Hülle
Sweepline
Daten-strukturen
Union-Find
Hash-tabellen
Fibonacci Heap
Binary Heap
Quadtrees
Suchbaum
Komplexität
Laufzeit-modell
Asymp-totik
Rekurrenz
Amortisierte Analyse
Program-mierung
Parallel
Funk-tional
Generisch

1.2 Algorithmen

[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) $(a_1, a_2, \ldots, a_n)$
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"**.

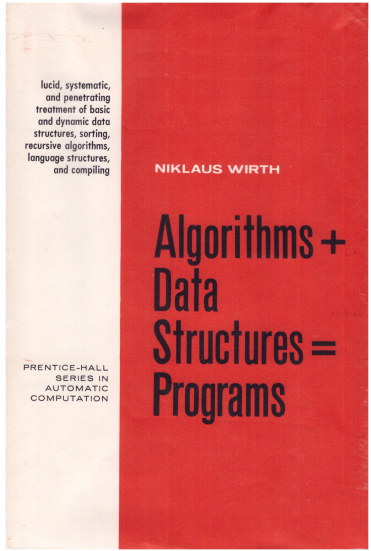
Mögliche Lösung

Wie oft werden die Zeilen jeweils ausgeführt?

```
void sort(std::vector<int>& a){
  unsigned n = a.size()
  for (unsigned i = 0; i<n ; ++i){
    for (unsigned j = i+1; j<n; ++j){
      if (a[j] < a[i]){
        std::swap(a[i],a[j])
      }
    }
  }
}
```

Datenstrukturen

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



Typische Schritte beim Algorithmendesign: Beispiel

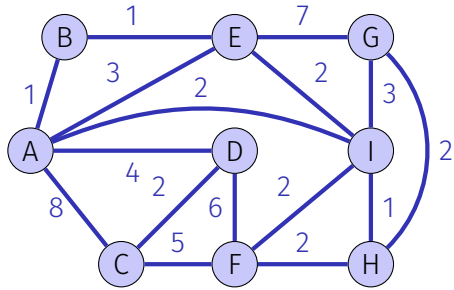
Routenplanung

Typische Design-Schritte

1. Spezifikation des Problems: Finde beste Route (kürzeste Zeit) von A nach B
2. Abstraktion: Graph aus Knoten, Kanten und Kantengewichten
3. Idee (Heureka!): Dijkstra
4. Datenstrukturen und Algorithmen: z.B. Min-Heap, Adjazenzmatrix / Adjazenzliste, Hash-Tabelle ...
5. Laufzeitanalyse: $\mathcal{O}((n+m) \cdot \log n)$
6. Implementation: Wahl der Repräsentation (z.B. Adjazenzmatrix/ Adjazenzliste/ Objekte

Schwieriges Problem: Der Handlungsreisende

Gegeben: Graph (Landkarte) aus Knoten (Städte) und gewichteten Kanten (Wege mit Weglänge)
Gesucht: Rundweg durch alle Städte, wobei jede Stadt genau einmal besucht wird (Hamilton-Kreis) mit kürzester Weglänge.



Der beste bekannte Algorithmus hat eine Laufzeit die mit der Anzahl der Knoten (Städte) exponentiell ansteigt.

Schon das Finden eines Hamilton-Kreises ist im allgemeinen Fall ein schwieriges Problem. Das Problem, einen Eulerkreis zu finden, nämlich einen Kreis, der jede *Kante* einmal besucht, ist hingegen ein Problem, welches in polynomieller Laufzeit gelöst werden kann.

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.**

Effizienz

Ressourcen sind beschränkt und nicht umsonst:

- Rechenzeit $\rightarrow$ Effizienz
- Speicherplatz $\rightarrow$ Effizienz

Eigentlich geht es in diesem Kurs fast nur um Effizienz.

2. Effizienz von Algorithmen

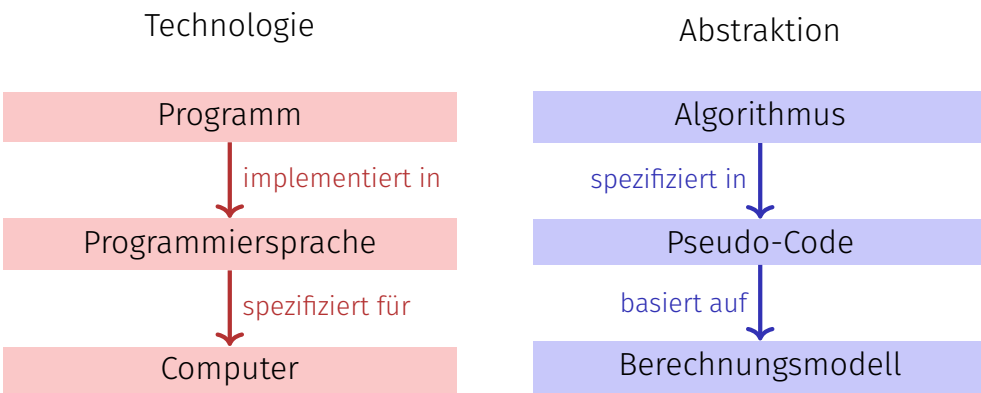
Effizienz von Algorithmen, Random Access Machine Modell, Funktionenwachstum, Asymptotik [Cormen et al, Kap. 2.2,3,4.2-4.4 | Ottman/Widmayer, Kap. 1.1]

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.
- 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.

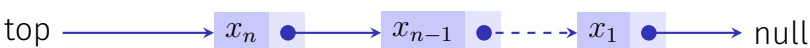
Grösse der Eingabedaten

- Typisch: Anzahl Eingabeobjekte (von fundamentalem Typ).
- Oftmals: Anzahl Bits für eine *vernünftige / kostengünstige* Repräsentation der Daten.
- Annahme: fundamentale Typen passen in Machinenwort (*word*) mit Grösse : $w \geq \log(\text{sizeof(mem)})$ Bits.

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.

Algorithmen, Programme und Laufzeit

Programm: Konkrete Implementation eines Algorithmus.

Laufzeit des Programmes: messbarer Wert auf einer konkreten Maschine. Kann sowohl nach oben, wie auch nach unten abgeschätzt werden.

Example 1

Rechner mit 3 GHz. Maximale Anzahl Operationen pro Taktzyklus (z.B. 8). $\Rightarrow$ untere Schranke.

Einzelne Operation dauert mit Sicherheit nie länger als ein Tag $\Rightarrow$ obere Schranke.

Hinsichtlich des *asymptotischen Verhaltens* des Programmes spielen die Schranken keine Rolle.

2.2 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]

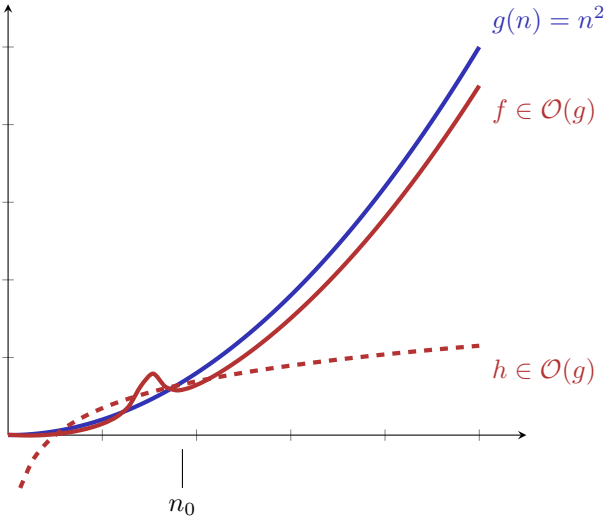
$$\begin{aligned}\mathcal{O}(g) = \{f : \mathbb{N} \to \mathbb{R} | \\ \exists\, c > 0, \exists n_0 \in \mathbb{N} : \\ \forall\, n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\}\end{aligned}$$

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

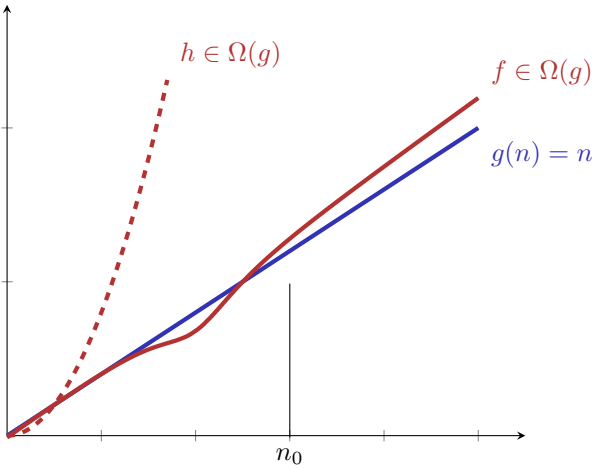


Umkehrung: Asymptotische untere Schranke

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



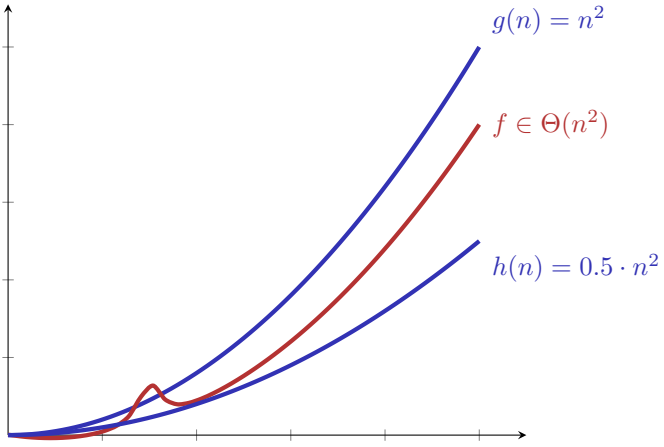
Asymptotisch scharfe Schranke

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

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

Einfache, geschlossene Form: Übung.

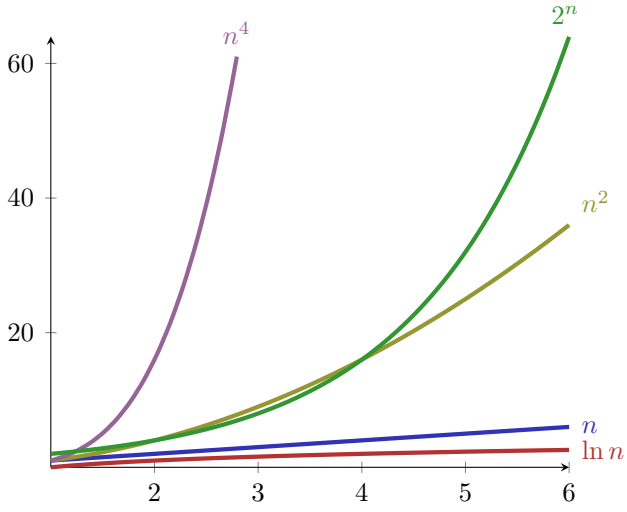
Beispiel



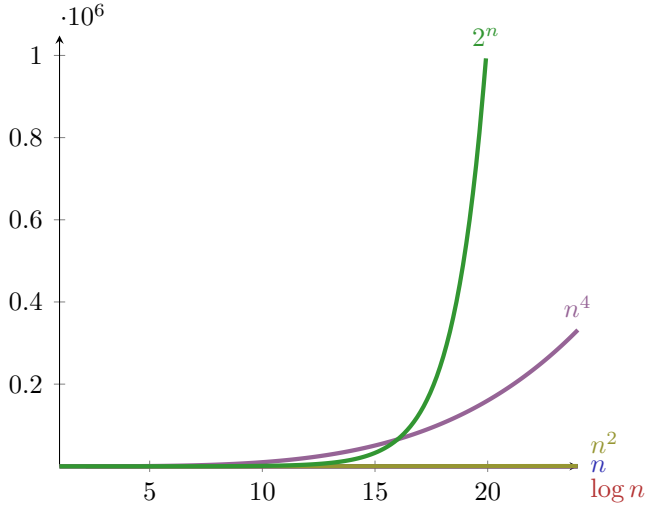
Wachstumsbezeichnungen

$\mathcal{O}(1)$	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
$\mathcal{O}(c^n)$	exponentiell	Travelling Salesman Dynamic Programming
$\mathcal{O}(n!)$	faktoriell	Travelling Salesman naiv

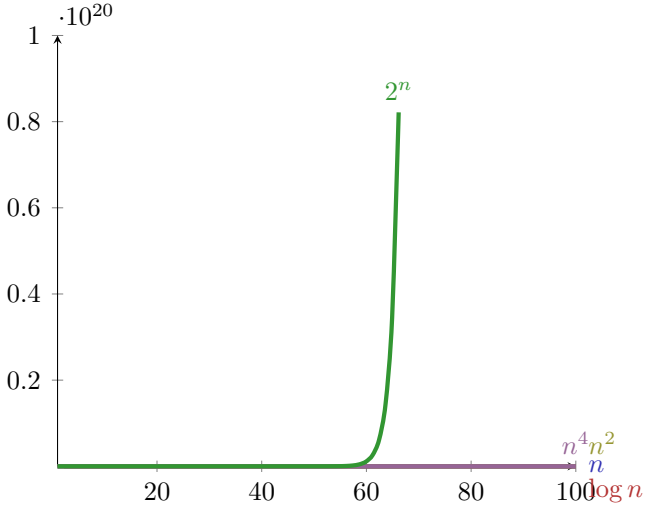
Kleine n



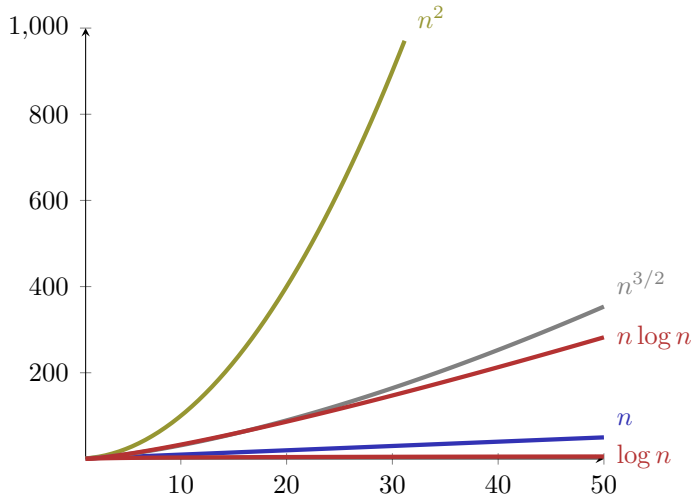
Grössere n



„Grosse” n



Logarithmen!



Zeitbedarf

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

Problemgrösse	1	100	10000	10^6	10^9
$\log_2 n$	$1\mu s$	$7\mu s$	$13\mu s$	$20\mu s$	$30\mu s$
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.	$\approx \infty$	$\approx \infty$	$\approx \infty$

Nützliches

Theorem 2

Seien $f, g : \mathbb{N} \to \mathbb{R}^+$ *zwei Funktionen. Dann gilt:*

1. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in \mathcal{O}(g)$, $\mathcal{O}(f) \subsetneq \mathcal{O}(g)$.
2. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = C > 0$ (C *konstant*) $\Rightarrow f \in \Theta(g)$.
3. $\frac{f(n)}{g(n)} \underset{n\to\infty}{\to} \infty \Rightarrow g \in \mathcal{O}(f)$, $\mathcal{O}(g) \subsetneq \mathcal{O}(f)$.

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: Effizienz: Arrays vs. Verkettete Listen

- Speicher: Unser **`avec`** belegt ungefähr n ints (Vektorgrösse n), unser **`llvec`** ungefähr $3n$ ints (ein Zeiger belegt i.d.R. 8 Byte)

- Laufzeit (mit **`avec`** = **`std::vector`**, **`llvec`** = **`std::list`**):

Asymptotische Laufzeiten

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

Typische Asymptotische Laufzeiten (Vorgriff!)

Datenstruktur	Wahlfreier Zugriff	Einfügen	Nächstes	Einfügen nach Element	Suchen
`std::vector`	$\Theta(1)$	$\Theta(1)\,A$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
`std::list`	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
`std::set`	–	$\Theta(\log n)$	$\Theta(\log n)$	–	$\Theta(\log n)$
`std::unordered_set`	–	$\Theta(1)\,P$	–	–	$\Theta(1)\,P$

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

Komplexität

Komplexität eines Problems P

Minimale (asymptotische) Kosten über alle Algorithmen A, die P lösen.

Komplexität der Elementarmultiplikation zweier Zahlen der Länge n ist $\Omega(n)$ und $\mathcal{O}(n^{\log_3 2})$ (Karatsuba Ofman).

Komplexität

Problem	Komplexität	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\Omega(n \log n)$
		$\Uparrow$	$\Uparrow$	$\Uparrow$	$\Downarrow$
Algorithmus	Kosten[2]	$3n-4$	$\mathcal{O}(n)$	$\Theta(n^2)$	$\Omega(n \log n)$
		$\Downarrow$	$\Updownarrow$	$\Updownarrow$	$\Downarrow$
Programm	Laufzeit	$\Theta(n)$	$\mathcal{O}(n)$	$\Theta(n^2)$	$\Omega(n \log n)$

[2] Anzahl Elementaroperationen