

Datenstrukturen und Algorithmen

Vorlesung am D-Math (CSE) der ETH Zürich

Felix Friedrich

FS 2019

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.

Inhalte der Vorlesung

Datenstrukturen / Algorithmen

Begriff der Invariante, Kostenmodell, Landau Symbole
Algorithmenentwurf, Induktion
Suchen und Auswahl, Sortieren
Amortisierte Analyse
Dynamic Programming

Minimale Spannbäume, Fibonacci Heaps
Kürzeste Wege, Maximaler Fluss
Fundamentale Algorithmen auf Graphen
Wörterbücher: Hashing und Suchbäume, AVL
van-Emde-Boas-Bäume, Splay-Bäume

Programmieren mit C++

RAll, Move Konstruktion, Smart Pointers,
Templates und Generische Programmierung
Exceptions
Funktionen und Lambdas

Promises and Futures
Threads, Mutexs and Monitors

Parallel Programming

Parallelität vs. Concurrency, Speedup (Amdahl/Gustavson), Races, Memory Reordering, Atomic Registers, RMW (CAS, TAS), Deadlock/Starvation

Algorithmus

1.2 Algorithmen

[Cormen et al, Kap. 1; Ottman/Widmayer, Kap. 1.1]

Algorithmus: wohldefinierte Berechnungsvorschrift, welche aus Eingabedaten (*input*) Ausgabedaten (*output*) berechnet.

23

24

Beispielproblem

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

Mögliche Eingaben

$(1, 7, 3)$, $(15, 13, 12, -0.5)$, $(1) \dots$

Jedes Beispiel erzeugt eine *Probleminstanz*.

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

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,

25

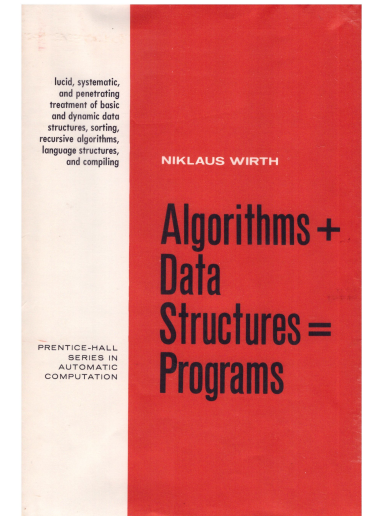
26

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.



27

28

Effizienz

Illusion:

- 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 → Effizienz
- Speicherplatz → 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.

29

30

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

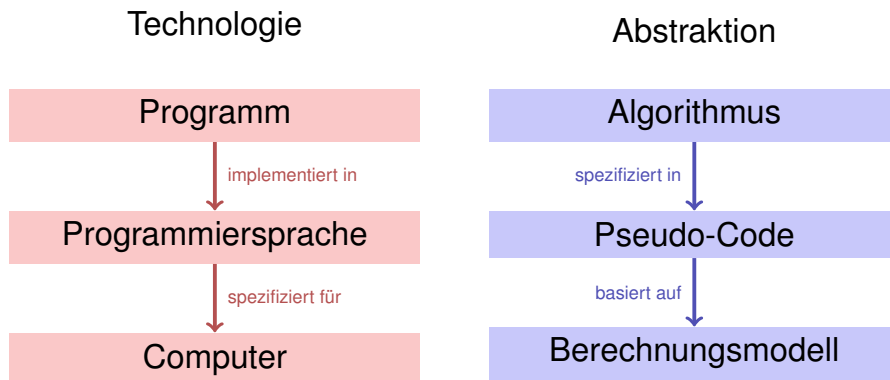
Ziele

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

31

32

Programme und Algorithmen



33

Technologiemodell

Random Access Machine (RAM)

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

34

Grösse der Eingabedaten

Typisch: Anzahl Eingabeobjekte (von fundamentalem Typ).

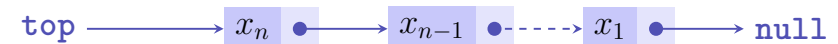
Oftmals: Anzahl Bits für eine *vernünftige / kostengünstige* Repräsentation der Daten.

Annahme: fundamentale Typen passen in Maschinenwort (*word*) mit Grösse : $w \geq \log(\text{sizeof(mem)})$ Bits.

Pointer Machine Modell

Wir nehmen an

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



35

36

Asymptotisches Verhalten

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

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

Beispiel

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.

Beispiel

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.

37

38

Oberflächlich

2.2 Funktionenwachstum

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

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.

39

40

Genauer: Asymptotische obere Schranke

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

Definition:¹

$$\mathcal{O}(g) = \{f : \mathbb{N} \rightarrow \mathbb{R} \mid \exists c > 0, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\}$$

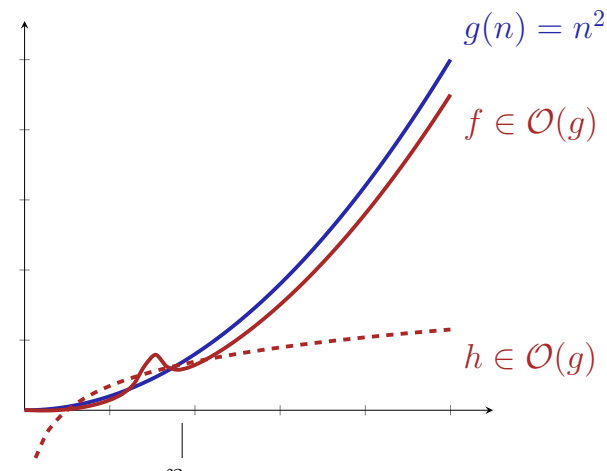
Schreibweise:

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

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

41

Anschauung



42

Beispiele

$$\mathcal{O}(g) = \{f : \mathbb{N} \rightarrow \mathbb{R} \mid \exists c > 0, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\}$$

$f(n)$	$f \in \mathcal{O}(?)$	Beispiel
$3n + 4$	$\mathcal{O}(n)$	$c = 4, n_0 = 4$
$2n$	$\mathcal{O}(n)$	$c = 2, n_0 = 0$
$n^2 + 100n$	$\mathcal{O}(n^2)$	$c = 2, n_0 = 100$
$n + \sqrt{n}$	$\mathcal{O}(n)$	$c = 2, n_0 = 1$

$$f_1 \in \mathcal{O}(g), f_2 \in \mathcal{O}(g) \Rightarrow f_1 + f_2 \in \mathcal{O}(g)$$

43

44

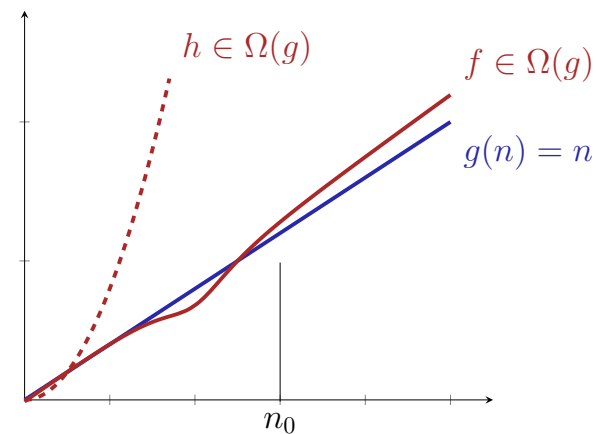
Umkehrung: Asymptotische untere Schranke

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

Definition:

$$\Omega(g) = \{f : \mathbb{N} \rightarrow \mathbb{R} \mid \exists c > 0, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n)\}$$

Beispiel



45

46

Asymptotisch scharfe Schranke

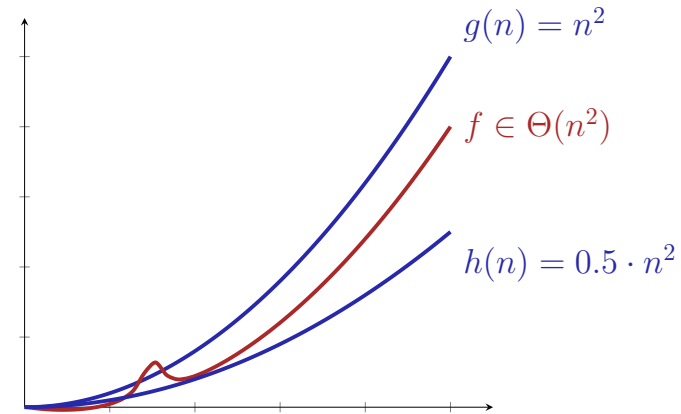
Gegeben Funktion $g : \mathbb{N} \rightarrow \mathbb{R}$.

Definition:

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

Einfache, geschlossene Form: Übung.

Beispiel



47

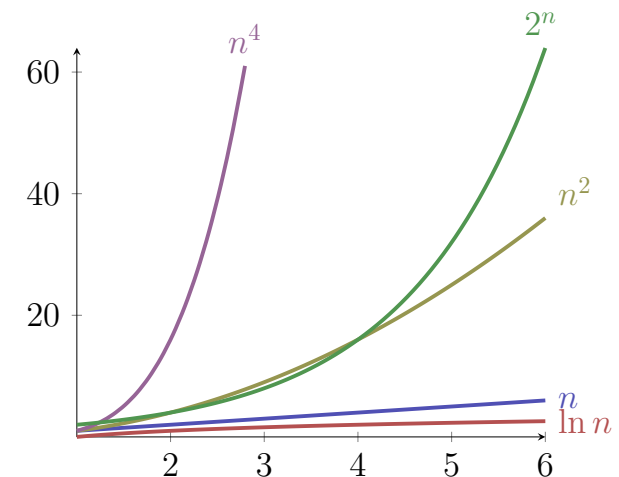
48

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

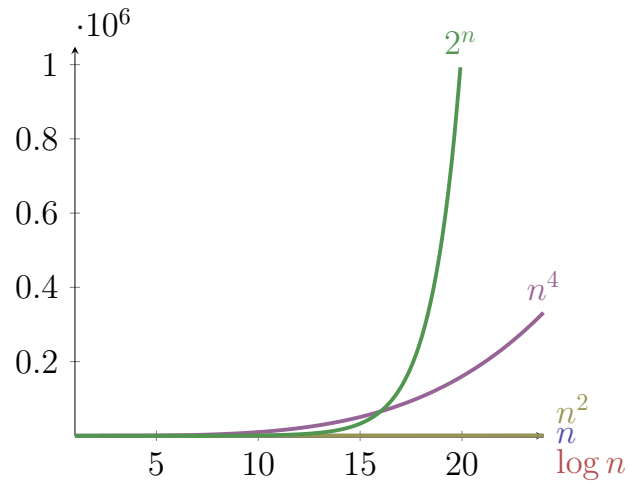
49

Kleine n



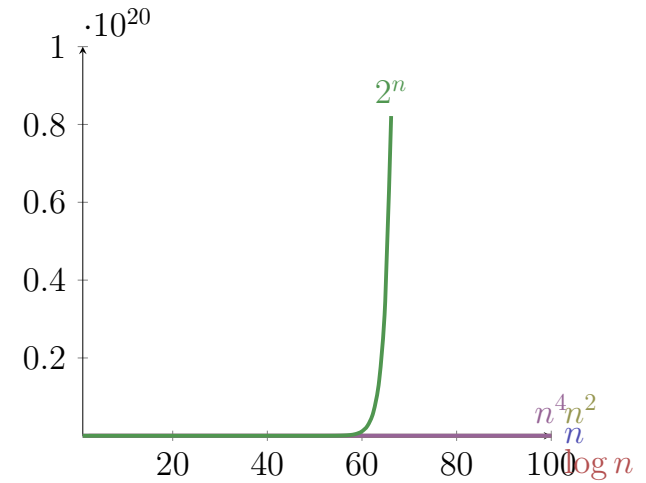
50

Grössere n



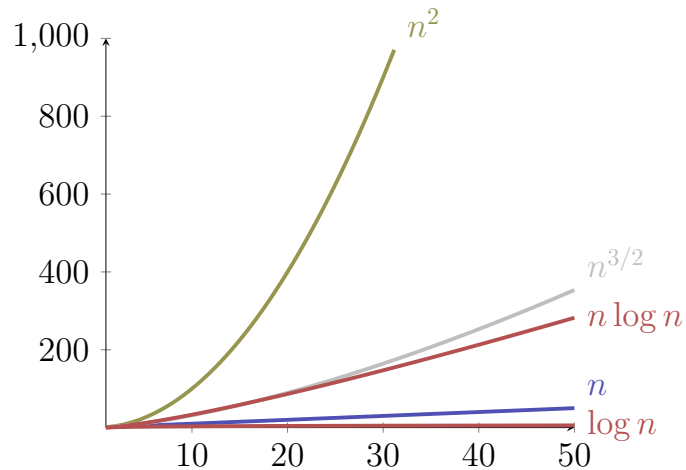
51

“Grosse” n



52

Logarithmen!



53

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.	$\approx \infty$	$\approx \infty$	$\approx \infty$

54

Nützliches

Theorem

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

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

Zur Notation

Übliche informelle Schreibweise

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

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

Es gilt nämlich

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

Beispiel

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

Wir vermeiden diese Schreibweise, wo sie zu Mehrdeutigkeiten führen könnte.

55

56

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`):

```

prepending (insert at front) [100,000x]:
  ▶ avec: 675 ms
  ▶ llvec: 10 ms
appending (insert at back) [100,000x]:
  ▶ avec: 2 ms
  ▶ llvec: 9 ms
removing first [100,000x]:
  ▶ avec: 675 ms
  ▶ llvec: 4 ms
removing last [100,000x]:
  ▶ avec: 0 ms
  ▶ llvec: 4 ms

removing randomly [10,000x]:
  ▶ avec: 3 ms
  ▶ llvec: 113 ms
inserting randomly [10,000x]:
  ▶ avec: 16 ms
  ▶ llvec: 117 ms
fully iterate sequentially (5000 elements) [5,000x]:
  ▶ avec: 354 ms
  ▶ llvec: 525 ms
    
```

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
<code>std::vector</code>	$\Theta(1)$	$\Theta(1) A$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
<code>std::list</code>	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
<code>std::set</code>	–	$\Theta(\log n)$	$\Theta(\log n)$	–	$\Theta(\log n)$
<code>std::unordered_set</code>	–	$\Theta(1) P$	–	–	$\Theta(1) P$

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

57

58

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).

Exemplarisch:

Problem	Komplexität	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
		↑	↑	↑
Algorithmus	Kosten ²	$3n - 4$	$\mathcal{O}(n)$	$\Theta(n^2)$
		↓	↕	↕
Programm	Laufzeit	$\Theta(n)$	$\mathcal{O}(n)$	$\Theta(n^2)$

²Anzahl Elementaroperationen