

# 13.

## Komplexität von Algorithmen

---

Buch Mark Weiss „Data Structures & Problem Solving Using Java“ siehe:  
- 223-227, 237-243, 248-250

# Lernziele Kapitel 13 Komplexität von Algorithmen

- O-Notation für die asymptotische Zeitkomplexität beherrschen
- Typische Vertreter der wichtigsten Komplexitätsklassen kennen

## Thema / Inhalt

Algorithmen benötigen Rechenzeit und brauchen Speicherplatz. Diese Ressourcen sind oft kritisch: Sie kosten etwas oder sind a priori limitiert. Um dies analytisch behandeln zu können, muss man den Ressourcenbedarf quantifizieren. Dieser hängt typischerweise von der Grösse des durch den Algorithmus bearbeiteten Problems ab – beispielsweise dürfte der Ressourcenbedarf um so grösser sein, desto mehr Zahlen zu sortieren sind. Die „**Problemgrösse**“, die den Ressourcenbedarf wesentlich bestimmt (also z.B. die Anzahl der zu sortierenden Elemente), wird mit  $n$  bezeichnet. Die Ausführung von Algorithmen, eine „Berechnung“, geschieht in aufeinanderfolgenden Rechenschritten. Oft nimmt man vereinfachend an, dass jeder Schritt gleich viel Zeit benötigt. Dann ist hinsichtlich der Zeitkomplexität interessant, wie viele Schritte (in etwa) bei einer Problemgrösse von  $n$  vom Algorithmus ausgeführt werden. Die **Zeitkomplexität** eines Problemlösungsalgorithmus ist damit eine Funktion  $f(n)$ . Diese Funktion wächst typischerweise monoton – ein umfangreicheres Problem benötigt zur Lösung länger als der gleiche Problemtyp in kleinerer Instanz.

Nun kann sich ein gewisses Problem (Multiplikation von Zahlen, Sortieren von Zahlen, Binärsuche etc.) auch bei fester Problemgrösse  $n$  unterschiedlich darstellen (zu sortierende Zahlen liegen absteigend sortiert oder zufällig angeordnet vor; bei der Binärsuche liegt das zu suchende

# Thema / Inhalt (2)

Element ganz links oder genau in der Mitte etc.), was bei manchen Algorithmen unterschiedlichen Aufwand verursacht. Daher ist man meist am Durchschnitt vieler Fälle (average case), manchmal auch an den Extremfällen (best / worst case) interessiert.

Oft will man gar nicht den genauen Wert der Komplexitätsfunktion  $f(n)$  kennen, sondern nur wissen, wie schnell sie im Prinzip wächst. Vereinfachend ausgedrückt, werden dazu alle im Wesentlichen gleich schnell wachsenden Funktionen zu **Komplexitätsklassen** zusammengefasst. Mit  **$O(f(n))$**  werden solche Funktionenklassen bezeichnet, wo das Wachstum nicht schneller als beim aufgeführten Repräsentanten  $f$  erfolgt, beispielsweise  $O(\log n)$  oder  $O(n^2)$ . Im ersten Fall spricht man von logarithmischer Komplexität (dann kann man auch noch recht grosse Probleme in vernünftiger Zeit, d.h. mit wenigen Berechnungsschritten, ausführen), im zweiten Fall von quadratischer Komplexität. Ist letzteres der Fall, muss man aufpassen: Wenn das Problem 8 Mal grösser wird, dann wird bei quadratischem Wachstum etwa 64 Mal mehr Rechenzeit benötigt! Es gibt aber noch schneller wachsende und daher in der Praxis „unangenehmere“ Komplexitätsklassen; schnellere Prozessoren verbessern die Situation hinsichtlich des bewältigbaren Problemumfangs zugehöriger Algorithmen dann kaum!

Spannend wird es, wenn man zeigen kann, dass ein bestimmtes algorithmisches Problem (z.B. das Sortieren) einen **inhärenten Mindestaufwand** zur Lösung verursacht. Einerseits kann man dann die Hoffnung begraben, gewisse Probleme eines nennenswerten Umfangs jemals lösen zu können. Andererseits kann man, ja muss man vielleicht, kreativ nach Auswegen suchen. Genügt vielleicht auch eine approximative Lösung, die sich eventuell viel leichter finden lässt? Und vor allem kann man sich die Tatsache, dass etwas nicht effizient geht, umgekehrt auch zunutze machen – ein grosser Teil der Kryptographie, und damit der Informationssicherheit, beruht gerade darauf, dass Schlapphüte und andere Malefikanten, aber auch Hinz und Kunz, aus einer abgehörten Kommunikation eben gerade nicht den Geheimschlüssel einfach berechnen können!


# Ist mein Algorithmus zu langsam? Oder evtl. das zu lösende Problem inhärent schwierig?

- Fragen sind so noch etwas naiv formuliert
- Aber berechtigt und relevant!
- Wir hatten uns ja bereits bei der **altägyptischen Multiplikation** gefragt, wie „effizient“ der Algorithmus ist
- Und wie „komplex“ ist die Multiplikation als solche?



# Aufwand von Algorithmen

- Algorithmen verbrauchen **Rechenzeit**
- Benutzte Datenstrukturen benötigen **Speicher**
- Wichtiges Ziel: Diesen **Ressourcenbedarf minimieren**
  - Beispiel einer typischen Frage: Wie hoch ist der Ressourcenbedarf beim Sortieren von  $n$  Elementen (bei einem konkreten Algorithmus)?
  - Frage präzisieren!  
→ Im **Durchschnitt**? Im **worst case**? Unter welchen Bedingungen?



„Mathematiker sind von Haus aus trainierte Pessimisten: Bewiesen ist nur, was unter keinen Umständen schief gehen kann. Also wird immer das Schlimmste angenommen.“  
-- Bernhard Steffen

# Begriffe bei der **Aufwandsanalyse**

- **Problemumfang** bzw. -grösse wird meist mit **n** bezeichnet
  - Oft: Anzahl der Eingabewerte
  - Manchmal aber auch zweckmässig: Gesamtzahl der Bits der Eingabe
  - **Beispiele:**
    - Zahl der *Sportvereine*, für die ein Saisonspielplan mit gewissen Nebenbedingungen gefunden werden soll
    - Zahl der *Städte*, für die paarweise der kürzeste Weg berechnet werden soll
    - Zahl der *Bits* bei der Primfaktorzerlegung von grossen Zahlen
    - Zahl der *Kanten*  $x$  und *Knoten*  $y$  (der Aufwand ist z.B. proportional zu  $x \cdot \sqrt{y}$  bei manchen Graphenproblemen)
- Der **Aufwand** (Bedarf an Zeit bzw. Speicherplatz in sinnvollen Einheiten) ist typischerweise von  $n$  abhängig
  - Wird daher als Funktion  $f(n)$  angegeben,  $f: \mathbb{N} \rightarrow \mathbb{R}^+$
  - Typischerweise wächst  $f$  monoton (oft sogar recht schnell)
- **Zeitkomplexität** typw. als Anzahl von **Berechnungsschritten**
  - Dabei muss klar sein, was genau ein elementarer Schritt sein soll, welches Modell einer Maschine man also annimmt

# Begriffe bei der Aufwandsanalyse (2)

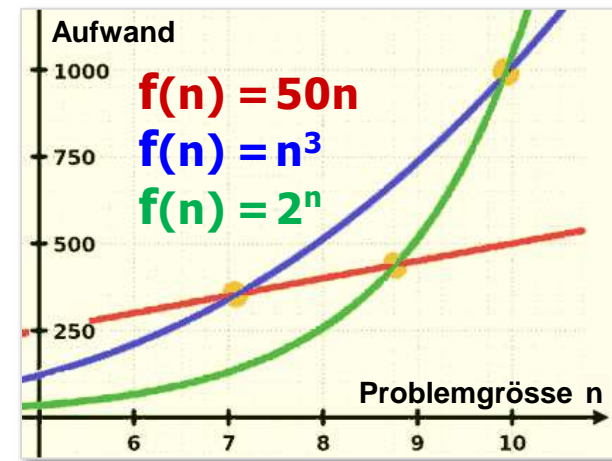
- Beim Zeitaufwand wird im Allgemeinen von **konstanten Faktoren** (und asymptotisch irrelevanten additiven Termen) **abstrahiert**
  - Also z.B.:  **$f(n) = n \log(n)$**  statt genauer  **$3 + 5n \log(n)$**
  - Grund: Aussagen zur Komplexität von Algorithmen sollen möglichst **allgemeingültig sein**, d.h. unabhängig von speziellen Eigenschaften einer Maschine, einer Implementierung oder technologischen Details
- Oft ist der Aufwand eines Algorithmus nicht nur von der Problemgröße  $n$ , sondern von den **konkreten Eingabewerten** (oder auch deren Reihenfolge) abhängig, daher unterscheidet man:
  - **günstigster** Aufwand („best case“)
  - **mittlerer** Aufwand („average case“)
  - **ungünstigster** Aufwand („worst case“)


Für eine angenommene Verteilung der Eingabewerte



# Begriffe bei der Aufwandsanalyse (3)

- Oft ist man nicht am exakten Wertverlauf der Komplexitätsfunktion  $f$  interessiert, sondern nur an der „Wachstumstendenz“, also am (oft leichter bestimmbareren) **asymptotischen Aufwand** (für  $n \rightarrow \infty$ )
  - Beachte: für „kleine“  $n$  kann ein Algorithmus mit schlechterem asymptotischen Aufwand dennoch besser sein ( $\rightarrow$  break even point)!



- **Komplexität eines Problems**  = geringstmöglicher Aufwand, der mit dem dafür besten Lösungsalgorithmus erreicht werden kann
  - Manche Probleme sind **inhärent** aufwendig / schwierig / „komplex“ (etwa: „kann mit keinem Algorithmus schneller als quadratisch gehen“)
  - Im positiven Sinn genutzt etwa bei sicheren kryptographischen Verfahren (deswegen brauchen z.B. die Bitcoin-Miner so unverschämt viel Energie!)



# Größen-**O**rdnungen

Verschiedene  
Geschwindigkeitsklassen



Letztendlich sind alle Autos schneller als alle Velos

# Komplexitätsgrößenordnung

- Zweck: Angabe der **Größenordnung** der (Zeit)komplexität eines Algorithmus als Funktion der Eingabegröße („n“)
- Idee: **Von unwesentlichen Konstanten abstrahieren**
  - Technologieparameter, Implementierungsdetails, Zeiteinheiten,...
- Schreibweise:
  - $O(\log n)$  logarithmische
  - $O(n)$  lineare
  - $O(n \log n)$  linear-logarithmische
  - $O(n^2)$  quadratische
  - $O(n^k)$  (mit  $k \in \mathbb{Q}^+$ ) polynomielle
  - $O(c^n)$  exponentielle
  - $O(1)$  konstante  
(d.h., unabhängig von n)

Komplexität

In der Praxis gelten alle Probleme als gut lösbar, die sich in Zeit bis  $O(n^3)$  lösen lassen. Die „versteckten Konstanten“ sollten allerdings nicht zu gross sein. In der Theorie gelten alle Probleme als gut lösbar, die sich in Zeit  $O(n^k)$  lösen lassen für irgendein  $k$ . -- Till Tantau

# Die O-Notation

- Man sagt, die Komplexität eines Algorithmus ist von der Grössenordnung  $O(f(n))$ , wenn für die „wirkliche“ Komplexität  $g(n)$  des Algorithmus gilt:

$$\exists n_0, c > 0 : \forall n \geq n_0 : g(n) \leq c f(n)$$

$g$  wird ab einem gewissen  $n_0$  majorisiert

$c \cdot f$  ist also eine asymptotisch obere Schranke für  $g$

- Beispiel: Für  $7 + 1.5n + 3n^2$  schreibe  $O(n^2)$

$O(n^3)$  oder  $O(2^n)$  wäre auch richtig, aber „übertrieben“

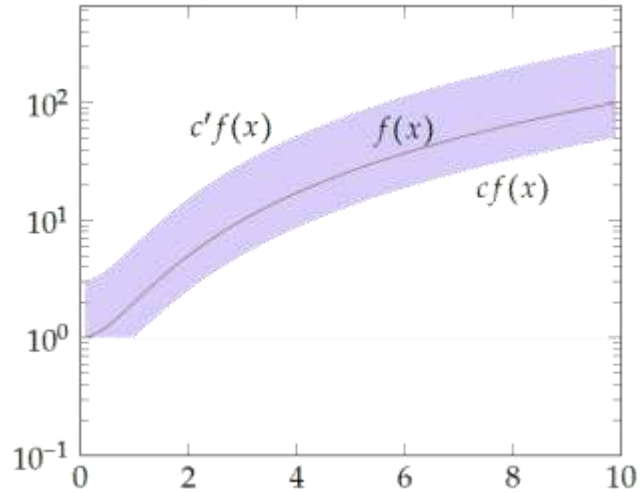
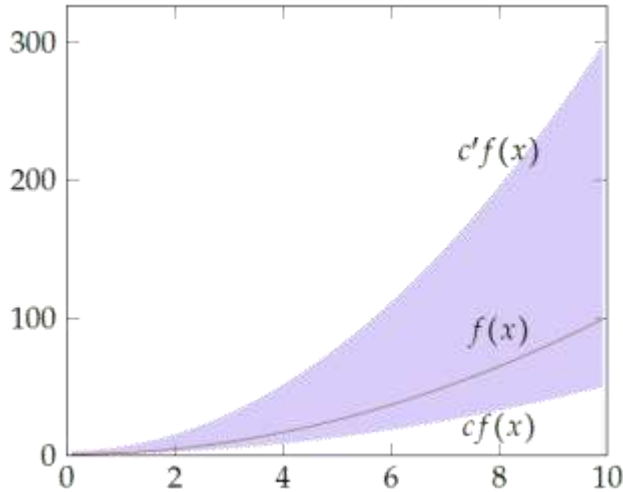
- Informelle Interpretation:** Die Laufzeit wird „im wesentlichen“ durch  $O(\dots)$  beschränkt

Interessant ist hier im Allgemeinen die „kleinste“ Funktion  $f$ , die  $g$  majorisiert

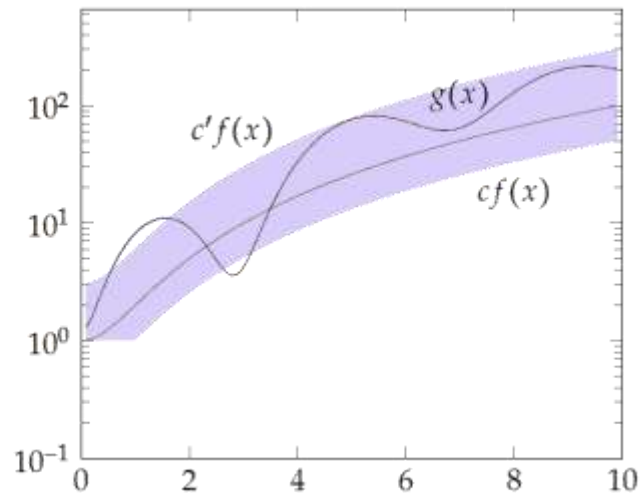
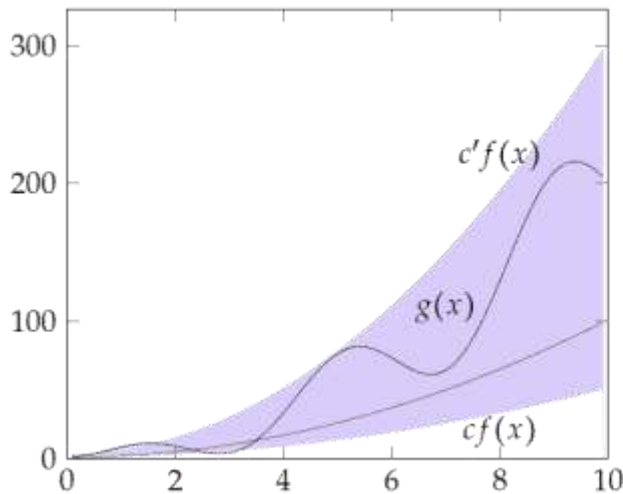
- Oder oft noch unpräziser: „ist etwa proportional zu“

- Oft kann man zumindest die Grössenordnung in O-Notation angeben, ohne die wirkliche Komplexität exakt zu kennen
  - Sinngemäss: „Gehört zur Klasse der Velos, nicht zu den Flugzeugen!“

# Schranken und Schläuche

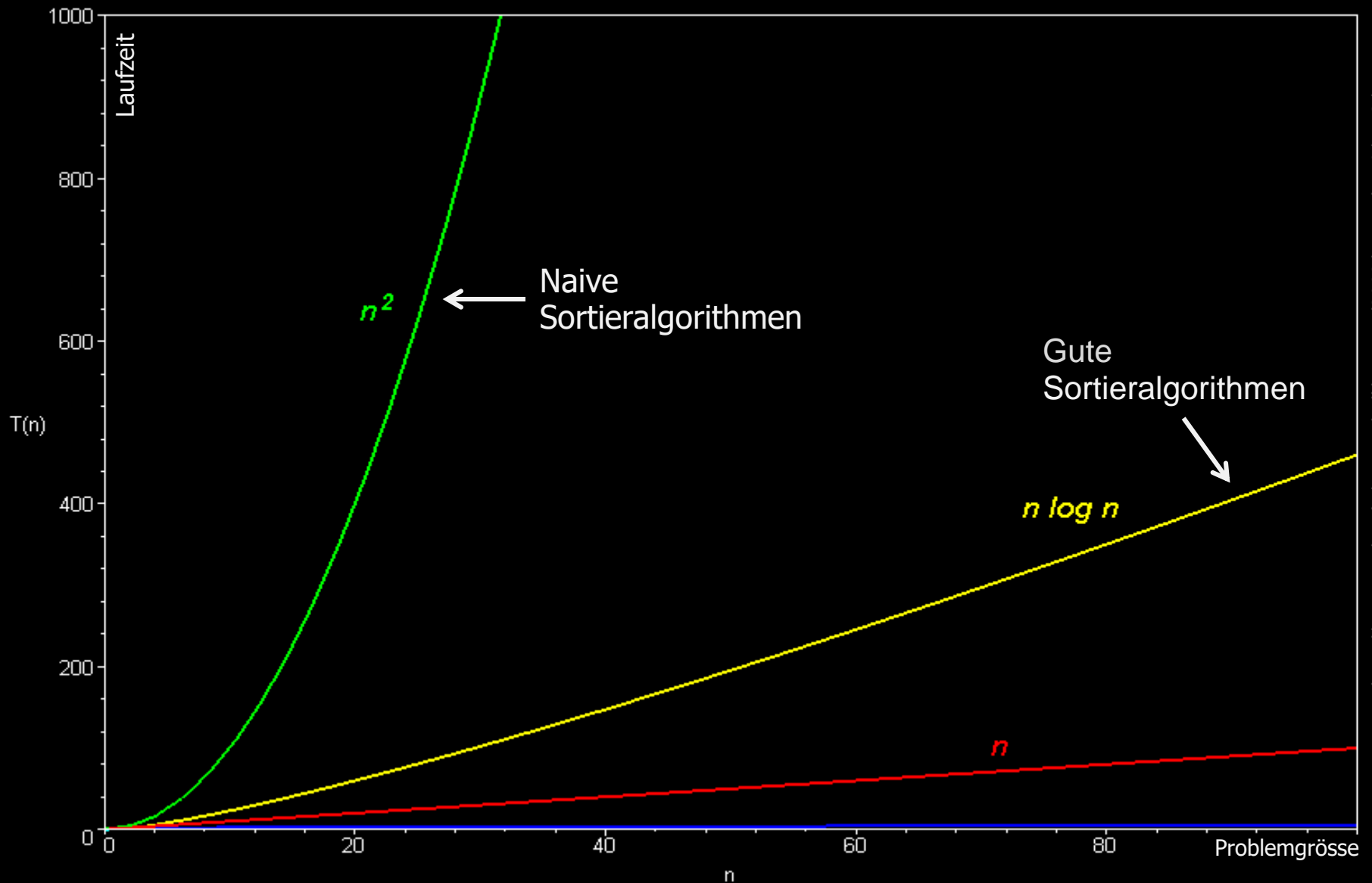


Zweimal die selbe Funktion  $f(x)$  sowie zwei Schranken  $c f(x)$  und  $c' f(x)$ ; linke Darstellung mit linear skaliertem y-Achse, rechte Darstellung mit logarithmisch skaliertem y-Achse.

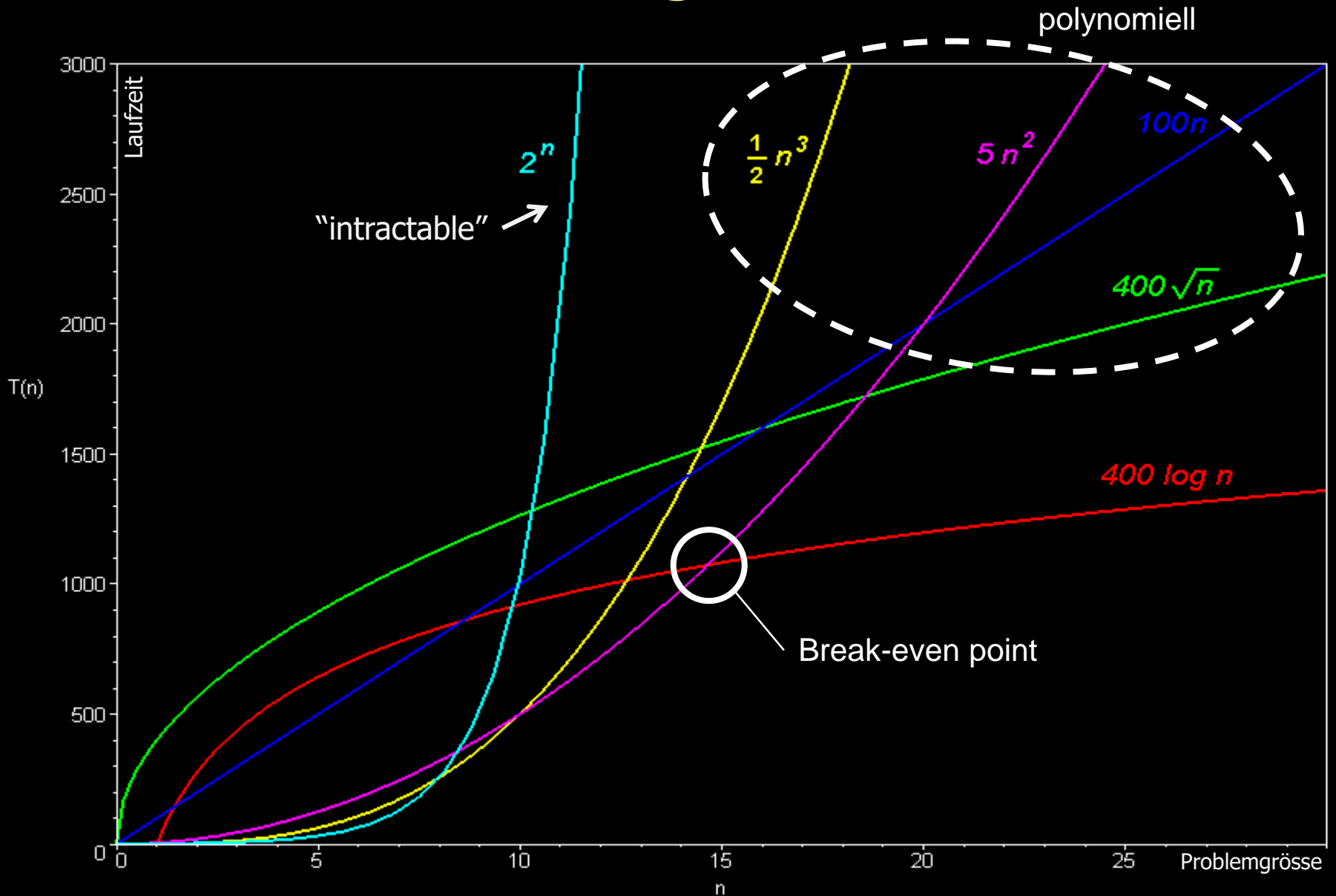


Zweimal die selbe Funktion  $g(x)$ , die für  $n > 4$  durch  $c f(n)$  und  $c' f(n)$  beschränkt ist. Der Graph von  $g$  liegt fast überall innerhalb des blauen Schlauches, der in der halblogarithmischen Darstellung besonders ausgeprägt ist.

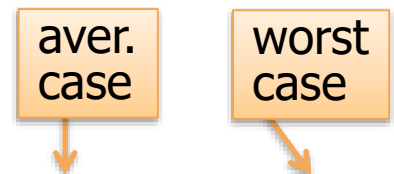
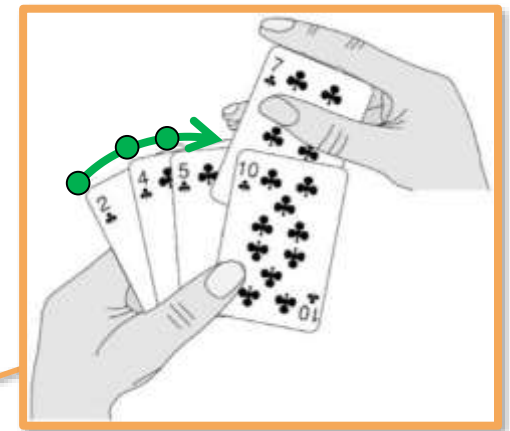
# Laufzeiten und Problemgrößen



# Laufzeiten und Problemgrößen



# Asymptotische Zeitkomplexität: Beispiele aus der Vorlesung



- Ägypt\_mult(m,n), ggT(m,n):  $O(\log n)$
- Türme von Hanoi:  $O(2^n)$
- Exchange sort, insertion sort:  $O(n^2)$
- Quicksort sowie Sortieren mit Suchbäumen:  $O(n \log n) \dots O(n^2)$
- Mergesort:  $O(n \log n)$
- Binärsuche:  $O(\log n)$  } average und worst case
- Umwandlung Infix-Ausdruck  $\rightarrow$  Postfix-Ausdruck:  $O(n)$   
(mit  $n$  = Ausdruckslänge: Anzahl Operanden, Operatoren, Klammern)
- Push bzw. pop auf einen Stack mit  $n$  Elementen:  $O(1)$

Nicht in der Vorlesung behandelt:

Einfacher Primzahltest:  $O(\sqrt{n})$ ; Travelling Salesman (naiv):  $O(n!)$



# Maximaler Problemumfang bei gegebener Zeit

- Beispiel: Bei Zeitbedarf von  $10 \mu\text{s}$  für  $n=1$ :

	1 Sek.	1 Min.	1 Stunde
$n$	100000	$6 \cdot 10^6$	$3.6 \cdot 10^8$
$n \log n$	7740	327491	15 095 829
$n^2$	316	2449	18974
$n^3$	46	181	711
$2^n$	16	22	28
$n!$	8	10	11

*Beispiel:* Wenn ein Schritt  $10 \mu\text{s}$  benötigt, dann brauchen  $181^3 = 5929741$  Schritte ca.  $10 * 6000000 \mu\text{s} = 1 \text{ Min}$

# Zeitbedarf bei vergrössertem Problemumfang

- Beispiel: Bei  $1 \mu\text{s}$  für  $n=1$ :

	n=1	n=100	n=10 000	n=100 000
log n	$1 \mu\text{s}$	$7 \mu\text{s}$	$13 \mu\text{s}$	$17 \mu\text{s}$
n	$1 \mu\text{s}$	$100 \mu\text{s}$	$0.01 \text{ s}$	$0.1 \text{ s}$
$n^2$	$1 \mu\text{s}$	$0.01 \text{ s}$	$1.7 \text{ min}$	$2.8 \text{ Std}$
$2^n$	$1 \mu\text{s}$	$10^{14} \text{ Jahrh.}$	$\approx \infty$	$\approx \infty$

# Bewältigbarer Problemumfang bei schnellerer Maschine

Wenn derzeit Problemumfang  $M$  bewältigbar ist, dann könnte eine **10 Mal schnellere Maschine** (in gleicher Zeit) folgenden Problemumfang  $M'$  bewältigen:

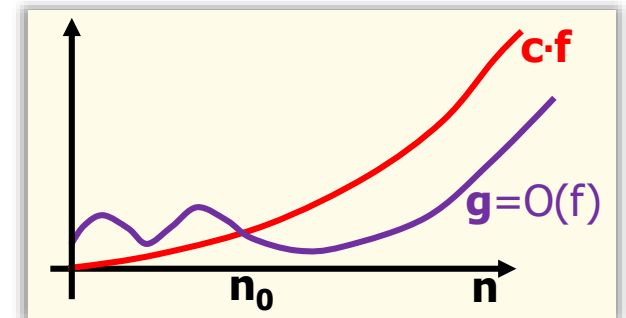
$f(n)$	$M'$
$\log n$	$2^{10 \log M}$
$n$	$10 M$
$n \log n$	fast $10 M$
$n^2$	$3.16 M$
$n^3$	$2.15 M$
$2^n$	$M + 3.3$

*Beispiel:* In  $2^M$  Berechnungsschritten wird ein Problem der Grösse  $M$  bewältigt. Mit der schnelleren Maschine kann man (in gleicher Zeit) 10 Mal mehr Schritte, also  $10 \times 2^M$  Schritte, ausführen. Welche Problemgrösse  $M'$  kann mit so vielen Schritten bewältigt werden? Wir setzen  $M'$  als unbekannte, zu bestimmende Variable in die Laufzeitformel  $f(n) = 2^n$  ein:  $2^{M'} = 10 \times 2^M$ . Logarithmieren (zur Basis 2) beider Seiten liefert  $M' = \log_2 10 + M \approx 3.3 + M$ .

Anzahl Berechnungsschritte („Laufzeit“) bei Problemumfang  $n$

# Komplexitätsklassen $O(f)$

- Mit  **$O(f)$**  wird jeweils eine ganze Funktionsklasse bezeichnet
  - Genauer:  $O(f(n)) = \{ g(n) \mid \exists n_0, c > 0: \forall n \geq n_0: g(n) \leq c f(n) \}$
- Statt  $g \in O(f)$  schreibt man oft salopper (aber falsch!):  $g = O(f)$ 
  - Notation dann z.B.:  $g(n) = O(n^2)$   
oder kürzer  $g = O(n^2)$
  - Interpretation:  $g$  wächst **höchstens so schnell** wie  $f$
- $f = O(1)$  bedeutet:  $f$  ist **beschränkt**
  - Überschreitet einen bestimmten konstanten Wert nicht
- Relevant ist auch die Klasse der **polynomiellen** Funktionen:
 
$$\text{POLY}(n) = \bigcup_{k>0} O(n^k)$$
  - Falls  $f \in \text{POLY}(n)$ , dann spricht man von **polynomiellem Aufwand** (wesentlicher Gegensatz dazu: *exponentieller Aufwand*  $O(c^n)$ )



Denkübung: Gilt  $O(\log n) \subset O(n) \subset O(n^2) \subset O(c^n)$ ? Oder ist es evtl. genau umgekehrt?

# Effiziente und ineffiziente Algorithmen

POLY  
vs.  
EXPO

Wir zitieren [http://mathwiki.cs.ut.ee/asymptotics/05\\_polynomial\\_complexity](http://mathwiki.cs.ut.ee/asymptotics/05_polynomial_complexity):

The main aim of computer science is develop efficient algorithms for different problems. However, the question **what is efficient** is not so straightforward. Engineers who program low-level operating system routines try to shave off any assembler level instruction [...]. The most liberal definition of efficiency is used by theoretical computer scientists, who consider **all polynomial time algorithms** as efficient.

Many widely used algorithms have polynomial time complexity. Examples of algorithms with **non-polynomial time complexity** are all kinds of **brute-force algorithms** that look through all possible configurations. For example, looking through **all the subsets** of a set of size  $n$  takes time  $O(2^n)$ , looking through **all permutations** of  $n$  elements takes time  $O(n!)$

The decision to declare polynomial algorithms as efficient is **motivated by following reasons**. **First**, there is a vast difference between running times of polynomial and exponential algorithms. **Secondly**, programming tasks seem to have either a polynomial-time algorithms or require exponential  $O(2^n)$  time brute-force algorithms. **Thirdly**, polynomial-time algorithms are closed under superposition. In layman's terms, if an algorithm makes polynomial number of calls to a function that is implemented as a polynomial algorithm, the resulting algorithm has also polynomial time-complexity. This greatly simplifies theoretical analysis of algorithms – you do not have to keep track what is the individual complexity of sub-routines as long as they are polynomial.

**In practice**, this rule for separating efficient algorithms form inefficient ones is often true, but not always. If your program uses  $n^{100}$  seconds or  $10^{100} n$  seconds then it runs in polynomial time but it is too slow for any practical purposes even for  $n = 2$ ; on the other hand, an algorithm which takes  $1.00000001^n$  seconds does not run in polynomial time but finishes within a second even for  $n = 1,000,000$ . However, determining whether an algorithm has polynomial time complexity is usually much easier than calculating the precise number of steps it makes or estimating the number of seconds it might spend on certain hardware.

# Beispiel: Komplexität von Mergesort

- Wir zeigen, dass die „vom Himmel gefallene“ Formel  $t(n) = n + n \log n$  die Zahl der Schritte angibt

- Beweis** ~~trivial~~, Ansatz mit folgender **Rekurrenzgleichung**  
*induktiv*

$$\begin{aligned}t(n) &= n + 2 t(n/2) \\&= n + 2 (n/2 + n/2 \log (n/2)) \\&= n + n + n \log (n/2) \\&= n + n - n + n \log n \\&= n + n \log n \\&\in \underline{\underline{O(n \log n)}}$$

$n$  = Aufwand für „merge“ von  $n$  Elementen

Aufwand für rekursives Sortieren von 2 Mal der Hälfte

Induktionsannahme für  $n/2$ :  
 $t(n/2) = n/2 + n/2 \log (n/2)$

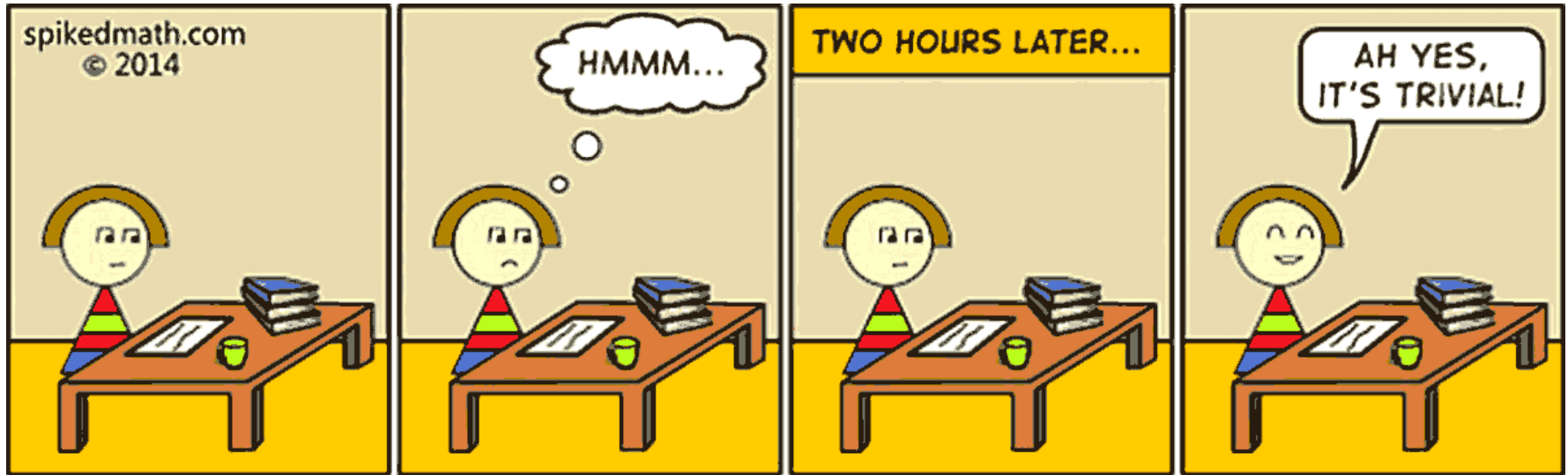
Da  $\log (n/2) = (-1 + \log n)$   
[Logarithmus zur Basis 2]

Damit alles „glatt“ geht, nimmt man zweckmässigerweise zunächst an, dass  $n$  eine **Zweierpotenz** ist. (Ansonsten Abschätzung durch nächst grössere Zweierpotenz oder mit einem Ansatz  $t(n) = n + t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil)$  arbeiten.)

Denkübung:  
Induktionsanfang?

# Triviale Beweise...

La qualité essentielle d'une démonstration est de forcer à croire. -- Pierre de Fermat

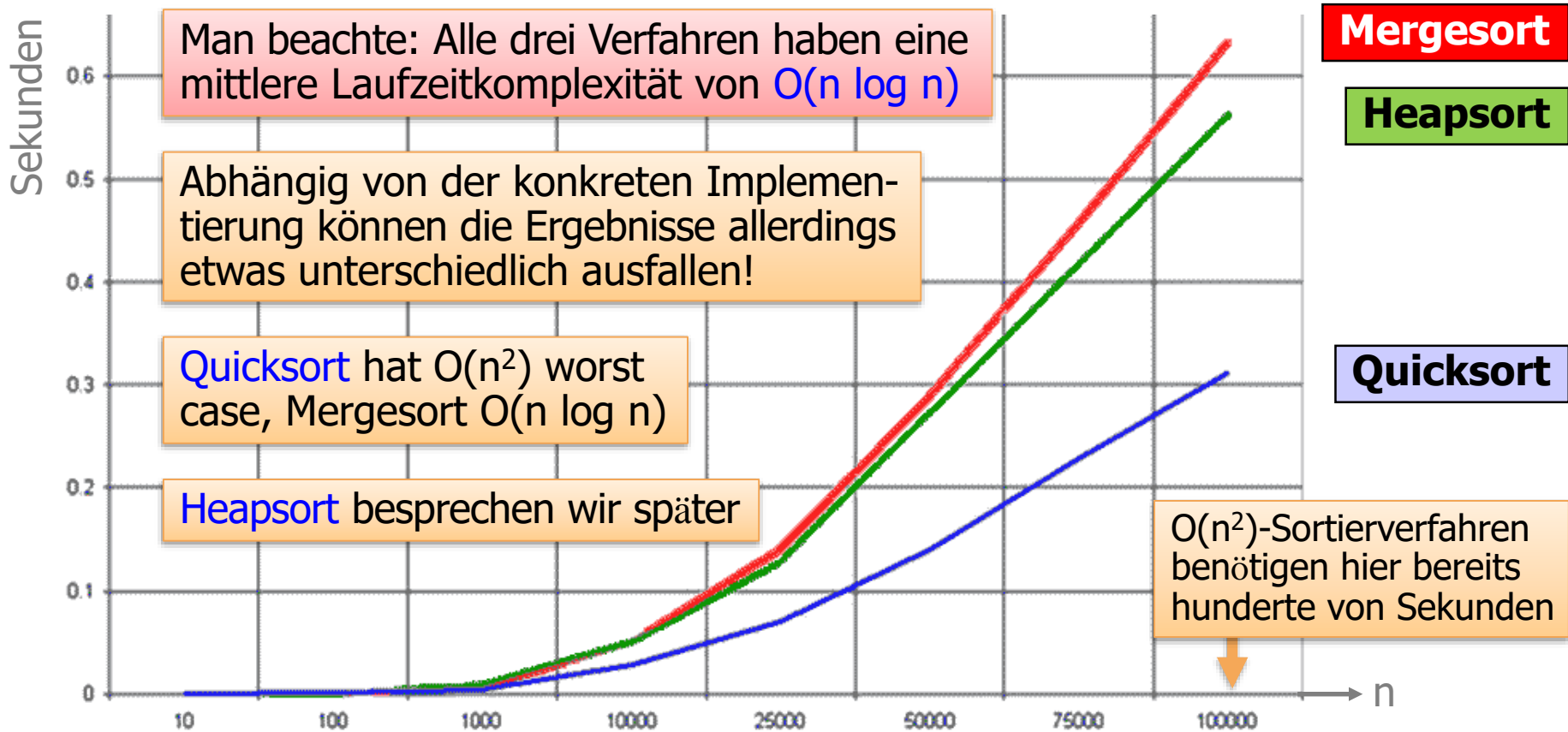


<http://spikedmath.com/comics/557-Proofs-Part-1.png>

“Professor Łojasiewicz from Jagiellonian University (Poland) once showed a proof of one implication in two hours. It was less than a minute until the end of lecture, so he looked at the theorem and said: ‘And the other implication is trivial’. The students were too shy to ask the professor why it is trivial, although they didn't see it themselves. Days passed and at an oral exam Prof. Łojasiewicz asked a student to prove that theorem. The student quickly proved one implication and then said: ‘and the other implication is trivial’. Łojasiewicz was sitting still for 15 minutes, thinking intensively. Finally he said: ‘You're right, it is trivial’ and he gave the student the highest possible mark.” [<http://spikedmath.com/557.html>]



# Empirischer Vergleich dreier Sortierverfahren



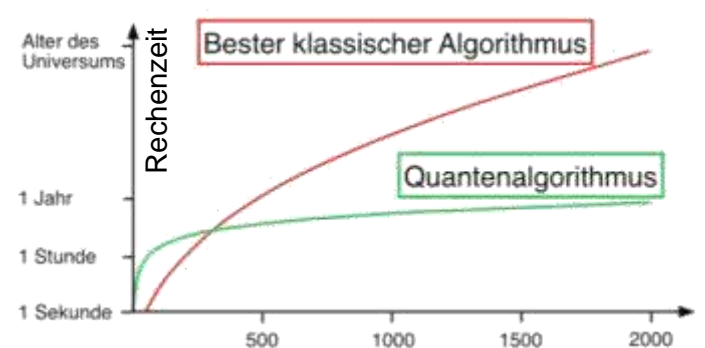
„...möchte ich nicht verschweigen, dass einige Informatiker mit einem Augenzwinkern behaupten, es gäbe keinen Unterschied zwischen  $O(n)$  und  $O(n \log n)$ . Die Argumentation ist recht einfach: Unsere Erde hat geschätzt  $6 \times 10^{49}$  Atome. Mehr Objekte werden ganz sicher nie zu sortieren sein. Der Zweierlogarithmus dieser wahrlich astronomischen Zahl beträgt trotzdem nur knapp schnöde 166. Das liesse sich doch quasi als Konstante ansehen.“ – Jens Gallenbacher

# Komplexität von Sortieren

- Es gilt (hier ohne Beweis), dass das **Sortieren** von  $n$  Elementen ein **Problem** mit **Zeitkomplexität  $O(n \log n)$**  ist
  - Unter gewissen allgemeinen Annahmen (dass z.B. über den Wertebereich der Elemente a priori gar nichts bekannt ist)
- Das heisst, dass z.B. **Mergesort** (grössenordnungsmässig) ein **optimaler Algorithmus** für das Sortierproblem ist
  - → Es gibt keinen „wesentlich“ besseren Sortieralgorithmus!

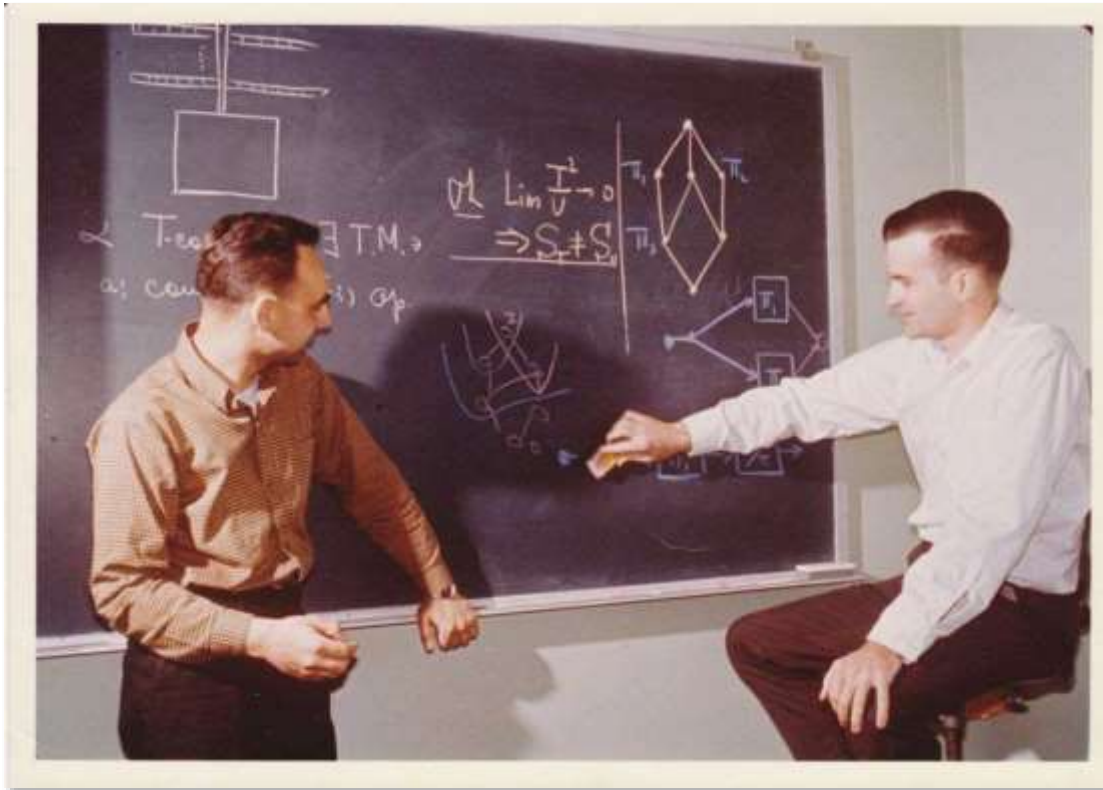
- 
- Aber vielleicht hilft ja **Parallelität**?
  - Oder ein **Quantencomputer**??

Aber wie ein Quantencomputer funktioniert, das lernen wir hier nicht. Und ob man jemals einen brauchbaren bauen wird, das wissen wir nicht.



# Komplexitätstheorie

Dass viele kombinatorische Probleme offenbar oft nicht viel besser als durch ein **Backtracking**-Verfahren mit **exponentieller Komplexität** gelöst werden können, war eine Beobachtung, die irritierte: Könnte man das beweisen? Oder geht es vielleicht doch irgendwie effizienter?



<http://blog.computationalcomplexity.org/2015/05/fiftieth-anniversary-of-publication-of.html>

**Juris Hartmanis** und **Richard Stearns** 1963; ihr wegweisender Artikel "On the Computational Complexity of Algorithms" von 1965 gab dem Gebiet seinen Namen; die Autoren erhielten 1993 den Turing Award.

**Gauß** fand für das 8-Damen-Problem keinen effizienteren Ansatz, aber erst der Logiker **Kurt Gödel** (1906 – 1978) formulierte Mitte der 1950er-Jahre (in einem Brief an John von Neumann) dies als konkrete Frage („wie stark im allgemeinen bei finiten kombinatorischen Problemen die Anzahl der Schritte gegenüber dem blossen Probieren verringert werden kann“), was später als „**P-NP-Problem**“ formaler ausgedrückt wurde. Die intensive Beschäftigung mit dieser Frage und damit verwandter Aspekte begründete ab den späten 1960er-Jahren die **Komplexitätstheorie** und machte sie zu einem Hauptbereiche der Theoretischen Informatik.

# Komplexitätstheorie (2)

Auf die fundamentalen Erkenntnisse und Fragen der Komplexitätstheorie können wir hier nicht eingehen; wir beschließen das Kapitel mit einem Zitat aus dem zuvor schon erwähnten Buch „Planet der Algorithmen“ von Sebastian Stiller, der das zugrundeliegende Phänomen charmant als „**algorithmische Schwerkraft**“ bezeichnet. Sein Hinweis auf die innewohnende „Macht“ ist nicht übertrieben, wenn man sich vergegenwärtigt, dass die moderne Kryptographie, und damit die Informationssicherheit, wesentlich auf der Komplexitätstheorie beruht.

„Komplexitätstheorie ist die Frage, wie viel Umstände es jeden Algorithmus, jedes präzise Schlussfolgern kostet, eine Frage zu lösen, weil dieser Aufwand nicht eine Schwäche des Algorithmus, sondern **essenziell für das Problem** ist. Die Komplexität eines Problems zu verstehen, geht Hand in Hand mit der Forschung an passenden Algorithmen. Wer versteht, was zu hoffen und was unmöglich ist, kann besser nach dem richtigen Algorithmus suchen.

**Gibt es schwierige Fragen?** Oder ist jede Frage nur eine Frage des Aufwands und des Genies, mit dem man sich ihr nähert? Sind manche Fragen in einem allgemeinen Sinn schwieriger als andere? Gibt es Fragen, die niemand herausbekommt – kein Genie, kein Algorithmus, kein Supercomputer? **Gibt es eine Grenze**, jenseits derer Rätsel **unlösbar** werden? Eine Grenze nicht nur hier und heute, nicht nur für mich als einzelnen Menschen, sondern für das, was ich tue, für mein Schlussfolgern? [...]

Eine wirklich schwierige Frage formulieren zu können, etwas erschaffen zu können, das kein anderer lösen kann, ist eine äusserst praktische Fähigkeit. **Darin liegt Macht.**“

# Resümee des Kapitels

- Aufwand von Algorithmen
  - Asymptotische Zeitkomplexität
  - O-Notation: Komplexitätsklassen  $O(f(n))$
  - $O(\dots)$ -Beispiele aus Algorithmen der Vorlesung
  - Mergesort: Induktiver Beweis der  $O(n \log n)$ –Zeitkomplexität



SOMETIMES IT'S JUST SIMPLE

<http://geek-and-poke.com>