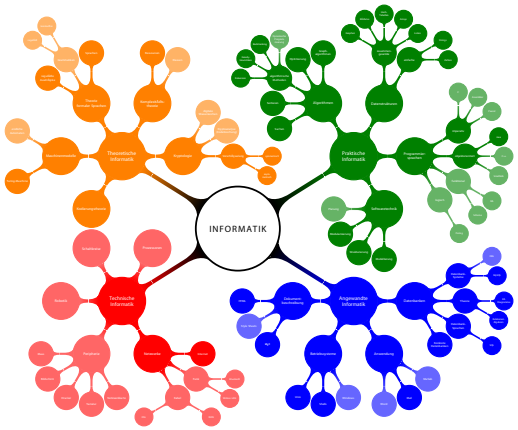
UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR
THEORETISCHE INFORMATIK

Kapitel 18

Komplexität von Problemen

Werden wir die Lösung in einer Sekunde oder in einem Jahr haben?

Vorlesung Einführung in die Informatik 1 vom 7. Januar 2014 von Till Tantau



Lernziele von Kapitel 18

1. Das Laufzeitverhalten der wichtigsten Algorithmen kennen
2. Das Laufzeitverhalten einfacher Programme abschätzen können
3. Die Gruppierung von Problemen nach groben Laufzeitklassen verstehen

Gliederung von Kapitel 18

Wie kann man die Rechenzeit von Algorithmen vergleichen?

Fragestellung

Sollte man lieber lineare Suche oder binäre Suche einsetzen?

Probleme bei der Beantwortung

- Je nach *Art der Eingabedaten* ist die Rechenzeit erheblich unterschiedlich:
  - Ist das zu suchende Element sehr nahe am Anfang, so ist lineare Suche schneller.
  - Ist das zu suchende Element in der Mitte, so ist binäre Suche schneller.
- Je nach *verwendeter Hardware* ist die Rechenzeit erheblich unterschiedlich.
- Je nach *Können des Programmierers* ist die Rechenzeit erheblich unterschiedlich.

Schnellere Computer oder schnellere Algorithmen?

- Benutzt man *lineare Suche*, so dauert das Suchen nach einem Element schlimmstenfalls $t_1 \cdot n$ Sekunden, wobei t_1 von der Hardware abhängig ist.
- Benutzt man *binäre Suche*, so dauert das Suchen nach einem Element schlimmstenfalls $t_2 \cdot \lceil \log_2 n \rceil$ Sekunden, wobei t_2 natürlich auch von der Hardware abhängig ist.

Zur Diskussion

Typischerweise ist t_2 größer als t_1. Was ist nun besser – lineare oder binäre Suche?

Vergleich der Laufzeiten von linearer und binärer Suche.

- Programmierer Anton implementiert lineare Suche in Maschinensprache auf einem Pentium 4 GHz.
  Dann ist grob $t_1 = 0{,}25$ ns.
- Programmiererin Beatrice implementiert binäre Suche in Java auf ihrem Handy mit 100 MHz.
  Dann ist grob $t_2 = 2\,\mu s = 2000$ ns.

Eingabegröße	*lineare Suche*	*binäre Suche*
10	2,5 ns	8 μs
100	25 ns	14 μs
1.000	250 ns	20 μs
10.000	2,5 μs	28 μs
100.000	25 μs	34 μs
1.000.000	250 μs	40 μs

Moral

- Ein guter Algorithmus schlägt einen schlechten Algorithmus, selbst wenn der gute Algorithmus schlecht implementiert wird und der schlechte gut implementiert wird.
- Die Terme n und $\log n$ in den Laufzeiten sind viel wichtiger als die Konstanten t_1 und t_2.

Der Begriff der Laufzeit.

Definition (Maschinenmodell)

Ein *Maschinenmodell* legt fest, wie lange eine Addition dauert, wie lange eine Multiplikation dauert, wie lange eine Zuweisung dauert und so weiter.

Definition (Laufzeiten)

Für ein Programm und eine Eingabe x bezeichnet

- $T(x)$ die Anzahl der Rechenschritte, die das Programm bei Eingabe x verbraucht;
- $T(l)$ die Anzahl Rechenschritte, die das Programm bei Eingaben der Länge l höchstens braucht.

Die *Eingabelänge* ist

- bei Strings die Länge des Strings und
- bei Arrays die Anzahl der Elemente.

Beispiel einer Laufzeitbestimmung.


```
static boolean containsZero (String s) {
  for (int i = 0; i < s.length(); i++) {
    if (s.charAt(i) == '0') {
      return true;
    }
  }
  return false;
}
```

Die verschiedenen Befehle benötigen unterschiedlich lange. Seien t_1 bis t_6 die Zeiten, die benötigt werden 1) für eine Zuweisung, 2) für einen Vergleich, 3) von der Methode `length`, 4) für die Berechnung von `i++`, 5) von der Methode `charAt` und 6) vom `return`-Befehl.

- $T(\texttt{ABCD05}) = t_1 + 10t_2 + 5t_3 + 4t_4 + 5t_5 + t_6$.
- $T(\texttt{ABCDEF}) = t_1 + 13t_2 + 7t_3 + 6t_4 + 6t_5 + t_6$.
- $T(6) = t_1 + 13t_2 + 7t_3 + 6t_4 + 6t_5 + t_6$.
- $T(l) = t_1 + (2l + 1)t_2 + (l + 1)t_3 + lt_4 + lt_5 + t_6$.

Wie genau sollten wir Laufzeiten messen?

- Die Laufzeit das Programm lautete $T(l) = t_1 + (2l + 1)t_2 + (l + 1)t_3 + lt_4 + lt_5 + t_6$.
- Wir hatten schon gesehen, dass die *genauen Werte* der Konstanten *eher unerheblich* sind.
- Wir führen deshalb eine spezielle Notation ein, mit der wir uns *auf das Wesentliche konzentrieren* können: Die *O-Klassen*.

Die zentralen Ideen sind:

1. Wir ignorieren *Konstanten*, mit denen *multipliziert wird*.
2. Uns interessiert nur Laufzeiten bei *großen Eingaben*.

Definition der O-Klassen.

Definition (O-Klasse)

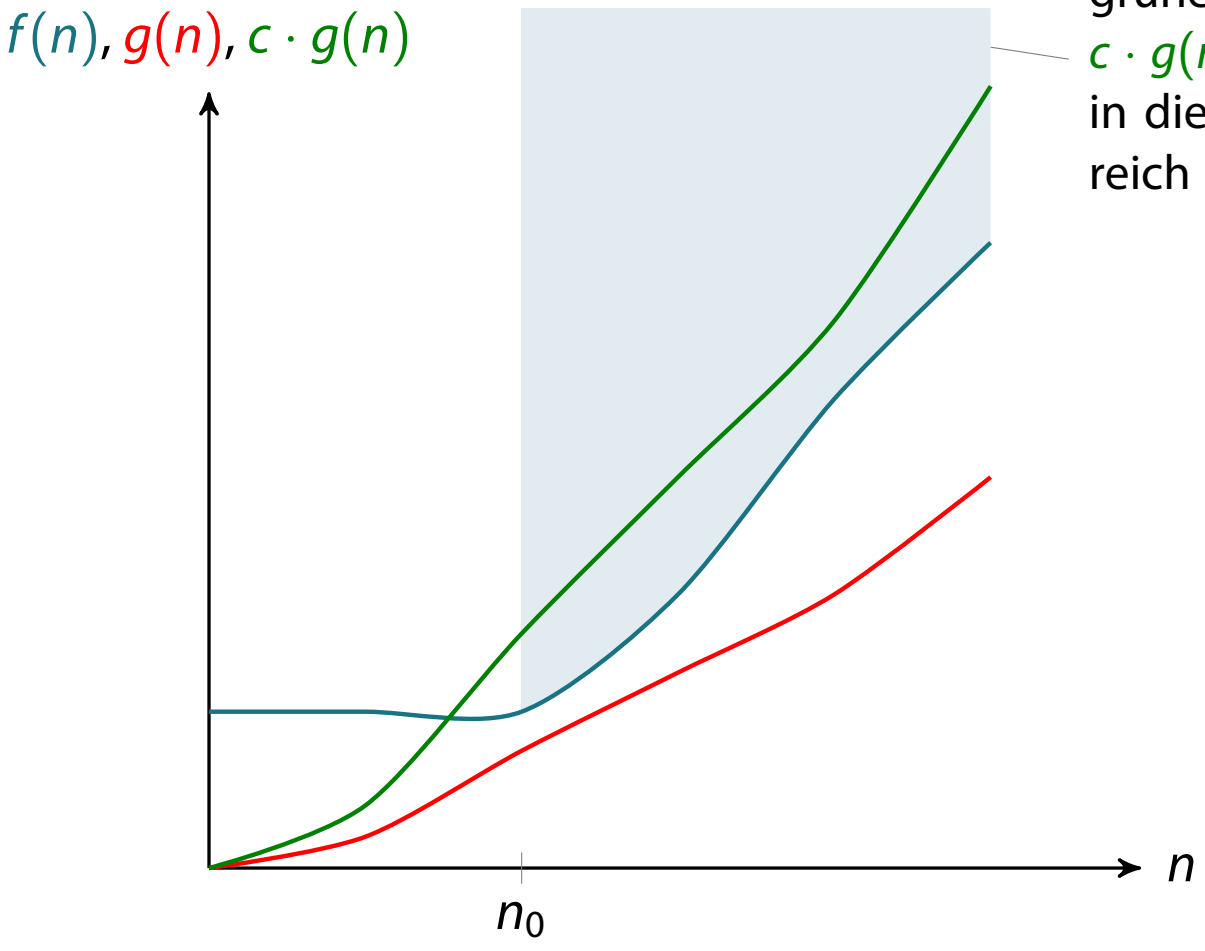
Sei $g : \mathbb{N} \to \mathbb{N}$ eine Funktion. Dann ist die O-Klasse $O(g)$ die Menge aller Funktionen $f : \mathbb{N} \to \mathbb{N}$, für die

- es eine Konstante c und
- eine Konstante n_0 gibt, so dass
- für alle $n > n_0$ gilt $f(n) \leq c \cdot g(n)$.

Merke

$f \in O(g)$ bedeutet, dass für *genügend große n* und einen *genügend großen Faktor* gilt, dass $g(n)$ mal diesen Faktor $f(n)$ dominiert.

Veranschaulichung von $f \in O(g)$



Beispiele zu O-Klassen.

Beispiel

Sei $g(n) = n^2$ und $f(n) = 3 + 17n^2$. Dann ist $f \in O(g)$.
(Wähle $c = 1000$ und $n_0 = 1000$. Dann ist sicherlich
$3 + 17n^2 \leq cn^2$.)

Beispiel

Sei $g(n) = n^2$ und $f(n) = 1000\sqrt{n}$. Dann ist $f \in O(g)$.
(Wähle $c = 1000$ und $n_0 = 0$.)

Beispiel

Sei $g(n) = n$ und $f(n) = n^2$. Dann ist $f \notin O(g)$.

Beispiel

Sei $g(n) = \log_2 n$ und $f(n) = \log_2(n^2)$. Dann ist $f \in O(g)$.
(Es gilt $\log_2(n^2) = 2 \log_2 n$.)

5-Minuten-Aufgabe

Zur Übung

Geben Sie für folgende f und g an, ob $f \in O(g)$ gilt.

1. $f(n) = n^4$ und $g(n) = n^5$.
2. $f(n) = n^5$ und $g(n) = n^4$.
3. $f(n) = n^5$ und $g(n) = n^4 \log n$.
4. $f(n) = 3n^5$ und $g(n) = 2n^5$.
5. $f(n) = \log_2 n$ und $g(n) = 1$.
6. $f(n) = \log_2 n$ und $g(n) = \log_3 n$.
7. $f(n) = \log_3 n$ und $g(n) = \log_2 n$.
8. $f(n) = 2^n$ und $g(n) = 3^n$.
9. $f(n) = 3^n$ und $g(n) = 2^n$.

Zur Schreibweise von O-Klassen.

- Man schreibt einfach $O(n^2)$ für »$O(g)$, wobei g die Funktion $g(n) = n^2$ ist«.
- Man schreibt auch $O(g)$, wenn g von mehreren Parametern abhängt.
  So bedeutet $O(n^2 m)$ »die Laufzeit ist für hinreichend große n und m höchstens cn^2m für eine Konstante c«.
- Man schreibt auch $f(n) = O(n^2)$ statt $f \in O(n^2)$.

Die wichtigsten O-Klassen.

Die wichtigsten O-Klassen und ihre Inklusionsbeziehungen:

$$\begin{aligned} O(1) &\subsetneq O(\log n) \subsetneq O(\sqrt{n}) \\ &\subsetneq O(n) \subsetneq O(n \log n) \subsetneq O(n \log^2 n) \\ &\subsetneq O(n^2) \subsetneq O(n^3) \\ &\subsetneq O(2^n) \subsetneq O(3^n). \end{aligned}$$

Wiederholung der Laufzeitbestimmung.


```
boolean containsZero (String s) {
  for (int i = 0; i < s.length(); i++) {
    if (s.charAt(i) == '0') {
      return true;
    }
  }
  return false;
}
```

- $T(l) = t_1 + (2l + 1)t_2 + (l + 1)t_3 + lt_4 + lt_5 + t_6$.
- Mit Hilfe der O-Notation können wir kurz sagen $T \in O(l)$ oder auch $T(n) = O(n)$.
- Der Aufwand ist also *linear*.

5-Minuten-Aufgabe

Zur Übung

1. Bestimmen Sie die *O*-Klasse des Zeitaufwands von folgendem Programm:

```
void prefix_sums (double[] vector) {
  int sum = 0;
  for (int i = 0; i < vector.length; i++) {
    sum = sum + vector[i];
    vector[i] = sum;
  }
}
```

2. Bestimmen Sie die *O*-Klasse des Zeitaufwands von folgendem Programm (schwierig):

```
double puzzle (double[] vector) {
  int i = 1;
  while (3*i < vector.length) {
    i = i*3;
  }
  return vector[i];
}
```

Was sind einfache Probleme?

- 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.

Beispiele (Effizient lösbare Probleme)

- Binäre und lineare Suche
- Sortieren
- Addieren, Multiplizieren, Dividieren
- Matrix-Multiplikation und Matrix-Invertierung
- Kürzeste Wege finden in Graphen
- Berechnen eines Bildes in einer Animation

Was sind schwierige Probleme?

- Manche Probleme gelten als *praktisch nicht gut lösbar*.
- Für solche Probleme ist *kein Algorithmus mit einer Laufzeit von* $O(n^k)$ bekannt.

Beispiel (Das Partitionsproblem)

Gegeben sind n Zahlen. Kann man sie so in zwei Teilmengen aufteilen, dass die Summen gleich sind?

- Es gibt 2^n Möglichkeiten, die Zahlen aufzuteilen,
- was einen Algorithmus mit Laufzeit $O(2^n)$ liefert.
- Was ist die Laufzeit des *besten Algorithmus* für das Partition-Problem?

Wir mögen keine exponentielle Laufzeit.

Programm 2010

Programmiererin Ada hat den Algorithmus für das Partition-Problems in Maschinensprache implementiert. Auf einem Pentium 4GHz kann sie damit in einer Stunde Eingaben bis $n \approx \log_2(360 \cdot 10^9) \approx 38$ bearbeiten.

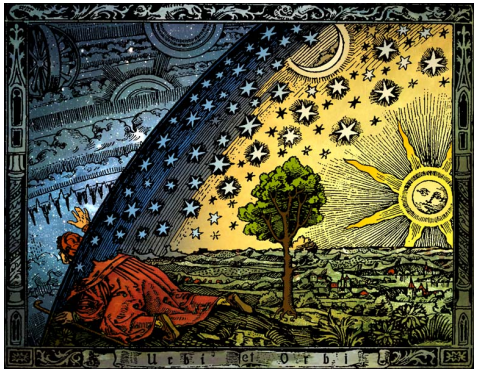
Programm 2020

10 Jahre später gibt es einen Quad-Core Rechner mit 40GHz. Nun kann sie in einer Stunde Eingaben bis $n \approx \log_2(4 \cdot 3600 \cdot 10^9) \approx 44$ bearbeiten.

Programm 2030

Wieder 10 Jahre später bekommt sie einen Supercomputer mit 10.000 Prozessoren mit je 1000 GHz. Nun kann sie in einer Stunde Eingaben bis $n \approx \log_2(3600 \cdot 10^{15}) \approx 62$ bearbeiten.

Wir mögen wirklich keine exponentielle Laufzeit.



Original C. Flammarion, public domain. Copyright coloring by Hugo Heikenwaelder, Creative Commons Attribution Sharealike License

Programm 2500

Ada baut einen großen Rechner. Jedes Atom im Universum ($\approx 10^{80}$) rechnet in jeder Planck-Zeit ($\approx 5{,}39 \cdot 10^{-43}$s) einen Rechenschritt. Nun kann sie in einer Stunde Eingaben bis
$n \approx \log_2(5{,}39 \cdot 3600 \cdot 10^{80+43}) \approx 423$
bearbeiten.

Moral

Eingaben mit 500 Zahlen wird man *niemals (!)* mit dem Algorithmus für das Partition-Problem lösen können.

Die Komplexitätsklassen P und NP.

- In der *Theoretischen Informatik*, speziell in der *Komplexitätstheorie*, untersucht man, welche Probleme einfach und welche schwierig sind.
- Die *einfachen Probleme* (solche, die sich in Zeit $O(n^k)$ lösen lassen) bilden die *Klasse P*.
- Nimmt man noch eine Reihe *anscheinend schwieriger Probleme* hinzu, so erhält man die *Klasse NP*.
- Die Frage, ob P $=$ NP gilt, ist die wichtigste ungelöste Frage der Theoretischen Informatik.

Zusammenfassung

O-Klasse

Die *O-Klasse $O(g)$* ist die Menge aller Funktionen $f : \mathbb{N} \to \mathbb{N}$, für die

- es eine Konstante c und
- eine Konstante n_0 gibt, so dass
- für alle $n > n_0$ gilt $f(n) \leq c \cdot g(n)$.

Man ignoriert also *konstante Faktoren* und *(zu) kurze Eingaben.*

Die wichtigsten O-Klassen und ihre Inklusionsbeziehungen

$$
\begin{aligned}
O(1) \subsetneq O(\log_3 n) = O(\log_2 n) = O(\log n) \subsetneq O(\sqrt{n}) \\
\subsetneq O(n) \subsetneq O(n \log n) \subsetneq O(n \log^2 n) \\
\subsetneq O(n^2) \subsetneq O(n^3) \\
\subsetneq O(2^n) \subsetneq O(3^n).
\end{aligned}
$$

Hierbei gilt alles bis $O(n^3)$ als »noch vertretbar«.