Informatik II
Komplexität von Algorithmen

G. Zachmann
Clausthal University, Germany
zach@tu-clausthal.de

Leistungsverhalten von Algorithmen

- Speicherplatzkomplexität: Wird primärer & sekundärer Speicherplatz effizient genutzt?
- Laufzeitkomplexität: Steht die Laufzeit im akzeptablen / vernünftigen / optimalen Verhältnis zur Aufgabe?
- Theorie: liefert untere Schranke, die für jeden Algorithmus gilt, der das Problem löst
- Spezieller Algorithmus liefert obere Schranke für die Lösung des Problems
- Erforschung von oberen und unteren Schranken: Algorithmik (praktische Informatik) und Komplexitätstheorie (theoretischen Informatik)

Laufzeit

- Definition:
  Die Laufzeit $T(x)$ eines Algorithmus A bei Eingabe x ist definiert als die Anzahl von Basisoperationen, die Algorithmus A zur Berechnung der Lösung bei Eingabe x benötigt
- Ziel: Laufzeit = Funktion der Größe der Eingabe

Basisoperationen und deren Kosten

- Annahme: computational model = real RAM (s. Kapitel 1)
- *Definition* : Als Basisoperationen bezeichnen wir
  - Arithmetische Operationen – Addition, Multiplikation,Division, Ab-, Aufrunden, auf Zahlen fester Länger (z.B 64 Bit = Double);
  - Datenverwaltung – Laden, Speichern, Kopieren von Datensätzen fester Größe;
  - Kontrolloperationen – Verzweigungen, Sprünge, Funktionsaufrufe.
- Kosten: Zur Vereinfachung nehmen wir an, daß jede dieser Operationen bei allen Operanden gleich viel Zeit benötigt (im Einheitskostenmaß)
  - Überwiegend unabhängig von der verwendeten Programmiersprache
  - Ablesbar aus Pseudocode oder Programmstück
  - Exakte Definition ist nicht bedeutend

Beispiele

- Einen Ausdruck auswerten
- Einer Variablen einen Wert zuweisen
- Indizierung in einem Array
- Aufrufen einer Methode / Funktion mit Parametern
- Verlassen einer Methode / Funktion

Laufzeitanalyse

- Sei P ein gegebenes Programm und x Eingabe für P, |x| Länge von x, und $T_P(x)$ die Laufzeit von P auf x
- Ziel: beschreibe den Aufwand eines Algorithmus als Funktion der Größe des Inputs (kann verschieden gemessen werden):
  $T_p(n)$ = Laufzeit des Programms P für Eingaben der Länge *n*
- Der beste Fall (best case): Oft leicht zu bestimmen, kommt in der Praxis jedoch selten vor:
  $$T_P(n) = \inf\{T_P(x) \mid |x| = n, x \text{ Eingabe für P}\}$$
- Der schlechteste Fall (worst case): Liefert garantierte Schranken, meist *relativ* leicht zu bestimmen. Oft zu pessimistisch:
  $$T_P(n) = \sup\{T_P(x) \mid |x| = n, x \text{ Eingabe für P}\}$$

- Bemerkung: Die exakte Größe der Eingabe (in Bytes) ist abhängig vom Problem!
  - Das ist aber i.A. kein Problem, da wir fast immer nur am Verhalten der Laufzeit in Relation zur Eingabegröße interessiert sind
  - Außer, die sog. "verborgenen" Konstanten sind sehr groß / unterschiedlich ... (später)

Kostenmaße

- Einheitskostenmaß: Annahme, jedes "primitive" Datenelement belegt, unabhängig von seinem Wert, denselben Speicherplatz (z.B. 4 Bytes)
  - Damit: Größe der Eingabe bestimmt durch Anzahl der Datenelemente
  - Beispiel: Sortierproblem
- Logarithmisches Kostenmaß (Bit-Komplexität): Annahme, jedes Datenelement belegt einen von seiner Größe (logarithmisch) abhängigen Platz
  - Größe der Eingabe ist bestimmt durch die Summe der tatsächlichen Größen der Elemente
  - Erinnerung: für Integer n ist die # Bits zur Darstellung $= \lceil \log_2(n+1) \rceil$
  - Beispiel: Zerlegung einer gegebenen großen Zahl in Primfaktoren
- Ab jetzt immer Einheitskostenmaß

Beispiel: Minimum-Suche

- Eingabe : Array von *n* Zahlen (a_0, a_2, … , a_{n-1}).
- Ausgabe : Index *i*, so dass $a_i \le a_j$ für alle Indizes $1 \le j \le n$.
- Beispiel:
  - Eingabe: 31, 41, 59, 26, 51, 48
  - Ausgabe: 3

```
def min(A):
    min = 0
    for j in range( 1, len(A) ):
        if A[j] < A[min]:
            min = j
  return min
```


Code	Kosten	max. Anzahl
`def min(A):`		
`min = 0`	c_1	1
`for j in range( 1, len(A) ):`	c_2	n - 1
`if A[j] < A[min]:`	c_3	n - 1
`min = j`	c_4	n - 1

- Zusammen: Zeit

$$T(n) = c_1 + (n-1)\cdot(c_2 + c_3 + c_4) \le c_5 n + c_1$$

n = Größe des Arrays

Weiteres Beispiel für eine Aufwandsberechnung

- Wir betrachten folgenden Algorithmus, der die Funktion

$$f(n) = 1! \cdot 2! \cdots (n-2)! \cdot (n-1)!$$

berechnet:

```
def f(n):
  r = 1
  while n > 0 :
      i = 1
      while i < n:
          r *= i
          i += 1
      n -= 1
  return r
```


- Anzahl Mult M(n)

$$M(n) = (n-1) + M(n-1) = (n-1) + (n-2) + M(n-2)$$
$$= \sum_{k=1}^{n-1} k = \frac{(n-1)(n-2)}{2}$$

- Anzahl der Inkrementierungen: $I(n) = n + M(n+1)$ woraus folgt:

$$I(n) = \frac{n(n+1)}{2}$$

- Die Anzahl der Vergleiche

$$V(n) = I(n+1) = \frac{(n+1)(n+2)}{2}$$

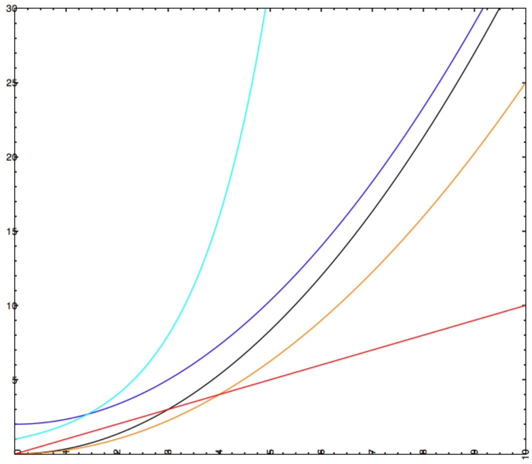
- Die Anzahl benötigter Zuweisungen Z(n) ist gleich

$$Z(n) = 1 + n + I(n) = 1 + \frac{n(n+3)}{2}$$

```
r = 1
while n > 0 :
    i = 1
    while i < n:
        r *= i
        i += 1
    n -= 1
return r
```

Beispiel für das Wachstum von Funktionen

$$f_1(n) = \frac{1}{3}n^2,$$
$$f_2(n) = \frac{1}{4}n^2,$$
$$f_3(n) = n,$$
$$f_4(n) = \frac{1}{3}n^2 + 2,$$
$$f_5(n) = 2^n.$$



Funktionenklassen

- Ziel: konstante Summanden und Faktoren sollen bei der Aufwandsbestimmung vernachlässigt werden
- Gründe:
  - Man ist am asymptotischem Verhalten für große Eingaben interessiert
  - Genaue Analyse ist technisch oft sehr aufwendig oder unmöglich
  - Lineare Beschleunigungen sind immer möglich (schnellere Hardware)
- Idee:
  - Komplexitätsmessungen mit Hilfe von Funktionenklassen
  - Alle Funktionen, die im Prinzip "gleich schnell" wachsen, sollen in einer Funktionenklasse sein
- Groß-O-Notation:
  Mit O-, Ω- und Θ-Notation sollen obere, untere bzw. genaue Schranken für das Wachstum von Funktionen beschrieben werden.

"Groß-O"

- Sei $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$
- Definition Groß-O: Die Ordnung von *f* (*the order of f*) ist die Menge von Funktionen

$$O\big(f(n)\big) = \{t : \mathbb{N} \to \mathbb{R}_{\geq 0} \mid \exists c \in \mathbb{R}_{\geq 0}\, \exists n_0 \in \mathbb{N}\, \forall n \geq n_0 : t(n) \leq c \cdot f(n)\}$$

- Definition Groß-Omega: die Menge Ω ist definiert als

$$\Omega(f(n)) = \{t : \mathbb{N} \to \mathbb{R}_{\geq 0} \mid \exists c \in \mathbb{R}_{\geq 0}\, \exists n_0 \in \mathbb{N}\, \forall n \geq n_0 : t(n) \geq c \cdot f(n)\}$$

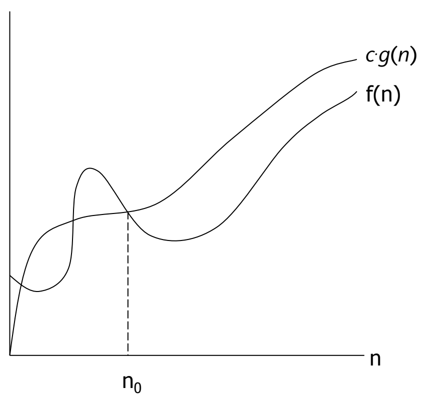
- Definition Groß-Theta: Die exakte Ordnung Θ von *f(n)* ist definiert als

$$\Theta(f(n)) = O\big(f(n)\big) \cap \Omega(f(n))$$

- Terminologie: O, Ω, Θ, heißen manchmal auch Landau'sche Symbole

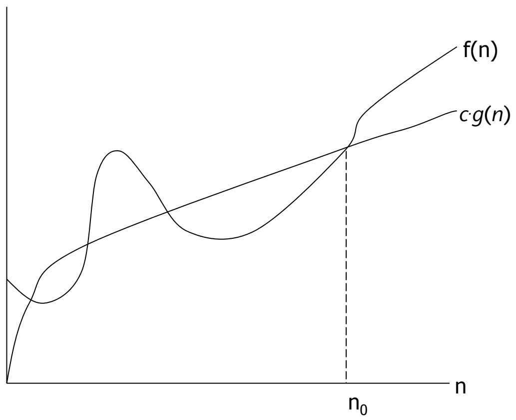
Veranschaulichung der O-Notation

- Die Funktion $f \in O(g)$, wenn es positive Konstanten c und n_0 gibt, so daß $f(n)$ ab n_0 unterhalb $c{\cdot}g(n)$ liegt



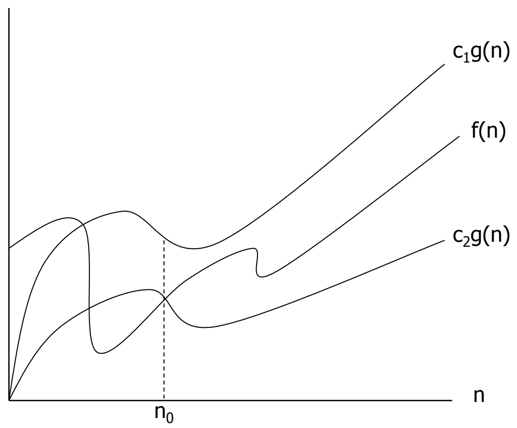
Veranschaulichung der Ω -Notation

- Die Funktion $f \in \Omega(g)$, wenn es positive Konstanten c und n_0 gibt, so dass $f(n)$ ab n_0 oberhalb $c \cdot g(n)$ liegt



Veranschaulichung der Θ-Notation

- Die Funktion $f \in \Theta(g)$, wenn es positive Konstanten c_1, c_2, und n_0 gibt, so dass $f(n)$ ab n_0 zwischen $c_1 g(n)$ und $c_2 g(n)$ "eingepackt" werden kann



Bemerkungen zu den O-Notationen

- In manchen Quellen findet man leicht abweichende Definitionen, etwa

$$O(g) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid \text{es gibt positive Konstanten a und b mit } f(n) \le ag(n) + b \text{ für alle n}\}$$

- Für die relevantesten Funktionen f (etwa die monoton steigenden f nicht kongruent 0) sind diese Definitionen äquivalent
- Minimalität: Die angegebene Größenordnung muss nicht minimal gewählt sein
- Asymptotik: Wie groß c und n_0 sind bleibt unklar (kann sehr groß sein!)
- "Verborgene Konstanten" (*hidden constants*): Die Konstanten c und n_0 haben für "kleine" n großen Einfluss!

- Schreibweise ist (leider) oft : $f = O(g)$ statt $f \in O(g)$
  - Manche (z.B. Knuth) bezeichnen diese Schreibweise als "*Einweg-Gleichung*"
  - OK ist $O(n) = O(n^2)$, nicht OK ist $O(n^2) = O(n)$
  - Unbedingt beachten, sonst kann man sinnlose Dinge damit ableiten!
- Mathematiker schreiben gerne asymptotische Ausdrücke der Art

$$e^x = 1 + x + \frac{1}{2}x^2 + O(x^3)$$

Aus Jeff Ericson's Lecture Notes seiner Algorithmen-Vorlesung

This sometimes leads to long sequences of results that sound like an obscure version of "Name that Tune":

Lennes: "I can triangulate that polygon in $O(n^2)$ time."

Shamos: "I can triangulate that polygon in $O(n \log n)$ time."

Tarjan: "I can triangulate that polygon in $O(n \log \log n)$ time."

Seidel: "I can triangulate that polygon in $O(n \log^* n)$ time."
[Audience gasps]

Chazelle: "I can triangulate that polygon in $O(n)$ time."
[Audience gasps and applauds]

Beispiel Min-Search

- Behauptung: unser Minimum-Search-Algo besitzt Laufzeit $\Theta(n)$
- Erinnerung:

$$T(n) \approx c_5 n + c_1$$

```
def min(A):
  min = 0
  for j in range (1, len(A)):
    if A[j] < A[min]:
      min = j
```

- Zum Beweis ist zu zeigen:
  1. Es gibt ein c_2 und n_2, so dass die Laufzeit von Min-Search bei allen Eingaben der Größe $n \geq n_2$ immer höchstens $c_2 n$ ist. (Groß–O)
  2. Es gibt ein c_1 und n_1, so dass für alle $n \geq n_1$ eine Eingabe der Größe n existiert, bei der Min–Search mindestens Laufzeit $c_1 n$ besitzt. (Omega)

Beispiele zu Funktionsklassen

- Ist $n^2 \in O(n^3)$?
  - Gesucht: $c \in \mathbb{R}^+$ und $n_0 \in \mathbb{N}$, so dass die Bedingung erfüllt ist,
  - also $\forall n > n_0 : n^2 \leq cn^3$

    $$\Leftrightarrow \quad n \cdot c \geq 1$$
  - Wähle $c = 1, n_0 = 1$
- Ist $n^3 \in O(n^2)$?
  - Gesucht: $c \in \mathbb{R}^+, n_0 \in \mathbb{N}$, so dass $\forall n > n_0 : n^3 \leq cn^2$
  - $\Leftrightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} \; \forall n > n_0 : n \leq c$
  - Widerspruch!
- Analog zeigt man z.B. dass $n \log n \notin O(n)$

Der Groß-O-Kalkül

- Zunächst ein paar einfache "Rechen"-Regeln:
- Sei $f \in O(f)$
- Dann gilt:

$$O(O(f)) = O(f)$$

$$k \cdot O(f) = O(k \cdot f) = O(f) \text{ für konstantes k}$$

$$O(f) + k = O(f + k) = O(f) \text{ für konstantes k}$$

Additionsregel

- Lemma, Teil 1: Für beliebige Funktionen f und g gilt:

$$f + g \in O(f+g) = O(\max(f,g))$$

- Zu beweisen: nur das rechte "="
- Zu beweisen: jede der beiden Mengen ist jeweils in der anderen Menge enthalten

"$\subseteq$": Sei $t(n) \in O(f(n)+g(n)) \Rightarrow$

$$\exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}\ \forall n \geq n_0 : t(n) \leq c \cdot (f(n)+g(n))$$

Abschätzung nach oben:

$$c \cdot (f(n)+g(n)) \leq c \cdot 2 \cdot max\{f(n), g(n)\}$$

Mit $\bar{c} = 2 \cdot c$ gilt $t(n) \leq \bar{c} \cdot max\{f(n), g(n)\}$

Also ist $t(n) \in O(max\{f(n), g(n)\})$

"$\supseteq$": Sei $t(n) \in O(\max\{f(n), g(n)\})$

Also gilt

$$\exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : t(n) \leq c \cdot \max\{f(n), g(n)\}$$

Abschätzung nach oben:

$$max\{f(n), g(n)\} \leq f(n) + g(n)$$

Also ist Bedingung für $t \in O(f(n)+g(n))$
mit denselben c, n_0 erfüllt

- Lemma, Teil 2: Für beliebige Funktionen f und g gilt:

$$O(f) + O(g) = O(f+g)$$

- Additionsregel findet Anwendung bei der Berechnung der Komplexität, wenn Programmteile hintereinander ausgeführt werden.

Multiplikationsregel

- Lemma: Für beliebige Funktionen f und g gilt:

$$f \cdot g \in O(f \cdot g) = O(f) \cdot O(g)$$

- Multiplikationsregel findet Anwendung bei der Berechnung der Komplexität, wenn Programmteile ineinander geschachtelt werden (Schleifen)

Teilmengenbeziehungen

- Lemma: Es gelten die folgenden Aussagen:
  1. $O(f) \subseteq O(g) \Leftrightarrow f \in O(g)$
  2. $O(f) = O(g) \Leftrightarrow f \in O(g)$ und $g \in O(f)$
  3. $O(f) \subset O(g) \Leftrightarrow f \in O(g)$ und $g \notin O(f)$
- Beweis von Teil 1:
  1. " $\Rightarrow$ " : trivial
  2. " $\Leftarrow$ " : $f(n) \in O(g(n))$.

     Also $\exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}\ \forall n \geq n_0 : f(n) \leq c \cdot g(n)$

     Zu zeigen: jedes $t(n) \in O(f(n))$ ist auch in $O(g(n))$

     Sei also $t(n) \in O(f(n))$.

     Per Def gilt $\exists c' \in \mathbb{R}^+, n'_0 \in \mathbb{N}\ \forall n \geq n'_0 : t(n) \leq c' f(n)$

     Wähle $\bar{n_0} = \max\{n_0, n'_0\}$ und $\bar{c} = c \cdot c'$

     Damit gilt $t(n) \leq \bar{c} \cdot g(n)$ für alle $n \geq \bar{n_0}$

     Also $t(n) \in O(g(n))$

- Teil 2 & 3 : analog

Transitivität von Groß–O

- Lemma:
  Falls $f \in O(g)$ und $g \in O(h)$, dann ist $f \in O(h)$.

- Beweis:
  Sei $f(n) \in O(g(n))$ und $g(n) \in O(h(n))$
  $\Rightarrow \quad \exists c', c'' \in \mathbb{R}^+, n_0', n_0'' \in \mathbb{N} \; \forall n \geq \max\{n_0', n_0''\}$:
  $f(n) \leq c' \cdot g(n) \leq c' \cdot c'' \cdot h(n)$
  $\Rightarrow \quad$ Behauptung

Einfache Beziehungen

- Lemma: Für alle $m \in \mathbb{N}$ gilt
  $$O(n^m) \subseteq O(n^{m+1}).$$
  - Beweis: Übung
- Satz: Sei $p(n) := a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0$ wobei $a_i \in \mathbb{R}_{\geq 0}$ für $0 \leq i \leq m$.
  Dann gilt
  $$p(n) \in O(n^m).$$
- Insbesondere: Es seien p_1 und p_2 Polynome vom Grad d_1 bzw. d_2, wobei die Koeffizienten vor n^{d_1} und n^{d_2} positiv sind.
  Dann gilt:
  $$a) \; p_1 \in \Theta(p_2) \Leftrightarrow d_1 = d_2$$
  $$b) \; p_1 \in O(p_2) \Leftrightarrow d_1 \leq d_2$$
  $$c) \; p_1 \in \Omega(p_2) \Leftrightarrow d_1 \geq d_2$$

- Für alle *k*, *k* fest, gilt: $n^k \in O(2^n)$
- Für alle *k* > 0 und *ε* > 0 gilt: $\log^k n \in O(n^\epsilon)$
- Es gilt: $2^{n/2} \in O(2^n)$
- Für beliebige positive Zahlen *a, b* ≠ 1 gilt:

$$f(n) = \log_a n \in O(\log_b n)$$

  - Insbesondere:

$$O(\log_b n) = O(\log_2 n)$$

- Beweise: Übungsaufgabe. (Gleichzeitig ein Beleg, dass die Analysis-Vorlesung Anwendung hat ☺)