Komplexitätsanalyse von Algorithmen

Ziel: Effizienzuntersuchung, Entwicklung effizienter Algorithmen

rechnerunabhängig

Wahl des Rechnermodells hat Konsequenzen: formale Rechnermodelle (Turing-maschine, Random access machines) hier: gewöhnlicher Rechner, d.h. Befehle eines Programms werden sequenziell abgearbeitet, Kosten abhängig von Anzahl der Operationen

Voraussetzung: RAM random access memory - Speicher mit wahlfreiem Zugriff, d.h. beliebiges Objekt in fest vorgegebener Zeit zugreifbar

Analyse von Alg., erste Aufgabe: welche Operationen, wie hoch sind ihre Kosten (Grundrechenarten, Vergleiche, Zuweisungen, Funktionsaufrufe)

elementare Operationen auf einfache Daten benötigen nach oben beschränkte Zeit (nicht mehr richtig für strukturierte Daten , wie etwa Listen, Felder, Vektoren)

D.h.: Ausführungszeit proportional zu Anzahl der Operationen

Analysearten

- a-priori (rechnerunabh.) Abschätzungen über die Anzahl der Operationen (und damit Ausführungszeit) in abhängigkeit von der Größe des Inputs
  - worst-case: obere Schranke
  - obere Schranke für die mittlere Komplexität unter Annahme bestimmter Wahrscheinlichkeiten über das Auftreten
  - untere Komplexitätsschranken

  ideal: untere/obere schranke dicht beisammen
- a-posteriori (rechnerabh.): testen einer Implementierung des Alg. mit großen datensätzen, um "alle" fälle zu berücksichtigen, ggfs. statistische aussagen

Bsp. a-priori: geg.: Feld (Vektor) $x = (x_i) \in \mathbb{R}^n$ und $c \in \mathbb{R}$. Gesucht: i mit $x_i = c$. Es sei bekannt, dass ein solches i existiert. Betrachten wir die Anzahl der Vergleiche: worst-case n, im Mittel $n/2$, untere Schranke: 1.

Aufgabe
Bestimme die Anzahl der Operationen bei $\sum_{i=1}^{k} \frac{1}{i(i+1)}$.
Was ist die relevante Dimension des Inputs? (k)
Welche Arten von Operationen treten auf? Multiplikationen, Additionen
worst-case, mittlere, untere Schranken? Multiplikationen k (immer), Additionen $k-1+k+k-1 = 3k-2$ (oder nur $k+k-1 = 2k-1$ und k Zuweisungen, abhängig vom Alg.)

Es geht um die Größenordnungen, dazu gibt es folgende Bezeichnungen.

Asymptotische Notation/Landau-Symbole

Obere Schranken:

Definition 1 (Groß-Oh) *Seien* $f, g : \mathbb{N} \to \mathbb{R}$.

$f(n) = O(g(n)) \quad :\Longleftrightarrow \quad \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}: \quad |f(n)| \leq c|g(n)| \quad \forall n \geq n_0.$

Andere Schreibweise: $f(n) \in O(g(n))$.

n kann sein: Anzahl der Ein-/Ausgabewerte,

f Rechenzeit, (rechnerabhängig, bestimme g so dass $f = O(g(n))$.

Bsp. $f_1(n) = 2n^2, f_2(n) = 3n^2$, $f_1(n) = O(n^2), f_2(n) = O(n^2)$, Motivation für $\in$-Schreibweise.

Satz 1 *$f \in \Pi_m$, d.h. $f(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0$ gilt $f(n) = O(n^m)$.*

Beweis:

$$\begin{aligned} |f(n)| &\leq |a_m| n^m + \ldots + |a_1| n + |a_0| \\ &= \left(|a_m| + \frac{|a_{m-1}|}{n} + \ldots + \frac{|a_0|}{n^m} \right) n^m \\ &\leq \left(|a_m| + |a_{m-1}| + \ldots + a_0 | \right) n^m. \end{aligned}$$

□

Konstanten und Terme niedriger Ordnung werden vernachlässigt

- für große n Terme höchster Ordnung entscheidend
- Konstanten, Terme niederer Ordnung sind oft maschinenabhängig

$O(1)$ = beschränkt

gängigste Größenordnungen sind:

$$\begin{aligned} O(1) &< O(\log n) < O(n) < O(n^2) \\ &< O(n^k) \text{ für } k \geq 3 \text{ fest} \\ &< O(n^{\log n}) < O(2^n). \end{aligned}$$

polynomial, exponentiell,

Bsp.: $2^n \notin O(n^m)$ für alle m.

Vergleich Wachstum:

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

Aufgabe
Sei $f(n)$ Laufzeit einer Berechnung, n_0 mit jetziger Technologie in bestimmter Zeit T (z.B. 1 Tag) berechenbare Dimension des Problems. Sei z.B. $n_0 = 10^6 = 1Mio.$
Wir kaufen einen Rechner, der 10 mal so schnell ist. Gesucht ist die Dimension n_1 des Problems, die wir in gleicher Zeit T mit dem neuen Rechner behandeln können.

- Laufzeit polynomial: $f(n) = n^k$. Es gilt $n_0^k = T$ mit dem alten, $n_0^k = T/10$ und $n_1^k = T$ mit dem neuen Rechner, also $10n_0^k = n_1^k$ bzw. $n_1 = \sqrt[k]{10}n_0$, bei $k = 2$ etwa 3-mal so große Probleme.
- exponentiell $f(n) = 2^n$. analog: $2^{n_0} = T/10$ und $2^{n_1} = T$ mit dem neuen Rechner, also $10 \cdot 2^{n_0} = 2^{n_1}$ bzw. $n_1 = n_0 + \log_2 10 \approx n_0 + 3$.

Definition 2 (Klein-Oh) *Seien* $f, g : \mathbb{N} \to \mathbb{R}$.

$$f(n) = o(g(n)) \quad :\Longleftrightarrow \quad f(n) = O(g(n)) \ \textit{und} \ g(n) \neq O(f(n))$$

Man sagt dann: f ist von echt kleinerer Größenordnung als g.

Es gilt:

$$f(n) = o(g(n)) \quad \Longleftarrow \quad \forall \varepsilon \in \mathbb{R}^+ \, \exists n_0 \in \mathbb{N} : \quad |f(n)| \leq \varepsilon |g(n)| \quad \forall n \geq n_0 \Longleftrightarrow \lim_{n\to\infty} \frac{|f(n)|}{|g(n)|} = 0.$$

Beweis s. Übungsblatt

Hinweis: Wie verneint man die Aussage $g(n) = O(f(n))$:

$$g(n) \neq O(f(n)) \quad \Leftrightarrow \quad \forall C \in \mathbb{R}^+, n_0 \in \mathbb{N} \, \exists n \geq n_0 : \quad |g(n)| \quad > \quad C|f(n)|$$

Untere Schranken

Definition 3 *Seien* $f, g : \mathbb{N} \to \mathbb{R}$.

$f(n) = \Omega(g(n)) \quad :\Longleftrightarrow \quad \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} : \quad |f(n)| \geq c|g(n)| \quad \forall n \geq n_0.$

$f(n) = \Theta(g(n)) \quad :\Longleftrightarrow \quad f(n) = O(g(n)) \ \textit{und} \ f(n) = \Omega(g(n)), \ \textit{also}$

$f(n) = \Theta(g(n)) \quad :\Longleftrightarrow \quad \exists c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N} : \quad c_1|g(n)| \leq |f(n)| \leq c_2|g(n)| \quad \forall n \geq n_0.$

Andere Schreibweise: $f(n) \in \Omega, \Theta(g(n))$.

Bemerkung: $f = O(g) \Leftrightarrow g = \Omega(f)$.

Bsp.: $f(n)$ Anzahl Vergleiche bei Suche in einem Feld (Vektor) Länge n, $f(n) = \Theta(n)$.

Bsp.: Matrizenmultiplikation: Berechnung eines Elements der Produktmatrix: n Mult., $4n-1$ Add., insgesamt n^2 Einträge, d.h. $n^2(2n-1) = O(n^3)$. Mindestens n^2 Einträge, d.h. $\Omega(n^2)$. Komplexitätslücke.

Lemma 1 $f(n) = o(g(n))$ *genau dann, wenn* $[f(n) = O(g(n)) \wedge f(n) \neq \Omega(g(n)]$.

Beweis: "⇒":
Sei $f(n) = o(g(n)$. Nach Definition ist dann $f(n) = O(g(n))$ und $g(n) \neq O(f(n))$. Zu zeigen ist also noch: $f(n) \neq \Omega(g(n))$. Angenommen, dies sei doch der Fall. Dann folgt

$$f(n) = \Omega(g(n)) \quad \overset{Def.}{\Rightarrow} \quad \exists c_1 > 0, n_1 \in \mathbb{N} : f(n) \geq c_1 g(n) \quad \forall n \geq n_1$$

$$
\begin{aligned}
&\Rightarrow \quad \exists c_1 > 0, n_1 \in \mathbb{N} : g(n) \leq \frac{1}{c_1} f(n) \quad \forall n \geq n_1 \\
&\Rightarrow \quad \exists c := \frac{1}{c_1} > 0, n_0 := n_1 \in \mathbb{N} : g(n) \leq c f(n) \quad \forall n \geq n_0 \\
&\overset{Def.}{\Rightarrow} \quad g(n) = O(f(n)).
\end{aligned}
$$

Dies ist ein Widerspruch dazu, dass $g(n) \neq O(f(n))$.
"$\Leftarrow$":
Sei $f(n) = O(g(n))$ und $f(n) \neq \Omega(g(n))$. Zu zeigen ist $f(n) = o(g(n))$, d.h. (nach Definition) $f(n) = O(g(n))$ (was nach Voraussetzung erfüllt ist) und $g(n) \neq O(f(n))$.
Angenommen, dies sei doch der Fall. Dann folgt

$$
\begin{aligned}
g(n) = O(f(n)) \quad &\overset{Def.}{\Rightarrow} \quad \exists c_2 > 0, n_2 \in \mathbb{N} : g(n) \leq c_2 f(n) \quad \forall n \geq n_2 \\
&\Rightarrow \quad \exists c_2 > 0, n_2 \in \mathbb{N} : f(n) \geq \frac{1}{c_1} g(n) \quad \forall n \geq n_2 \\
&\Rightarrow \quad \exists c := \frac{1}{c_2} > 0, n_0 := n_2 \in \mathbb{N} : f(n) \geq c g(n) \quad \forall n \geq n_0 \\
&\overset{Def.}{\Rightarrow} \quad f(n) = \Omega(g(n))
\end{aligned}
$$

im Widerspruch zur Voraussetzung $f(n) \neq \Omega(g(n))$. □

A posteriori

Zeitmessung. Man unterscheidet:

- cputime
- Systemzeit
- wall-clock-time (Gesamtzeit)

Was braucht man?

- Möglichkeiten zur Laufzeitmessung von Programmen:
  - vom Betriebssystem: Unix: `time`
  - in Programmiersprache: C++
- Statistik (s. dort)