Script generated by TTT

Title: Seidl: GAD (21.04.2015)

Date: Tue Apr 21 13:46:17 CEST 2015

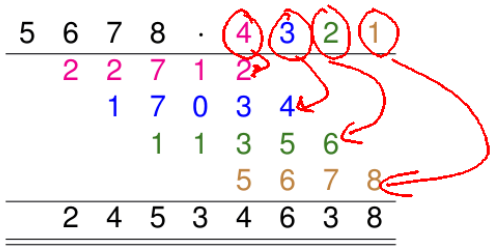
Duration: 117:15 min

Pages: 44

Multiplikation langer Zahlen

Schulmethode:

- gegeben Zahlen a und b
- multipliziere a mit jeder Ziffer von b
- addiere die Teilprodukte



Grundoperation

- Multiplikation von zwei Ziffern: $x \cdot y = ?$

  Das Ergebnis besteht aus (höchstens) zwei Ziffern u (Zehnerstelle) und v (Einerstelle), also

  $$x \cdot y = 10 \cdot u + v$$

- Addition von drei Ziffern: $x + y + z = ?$

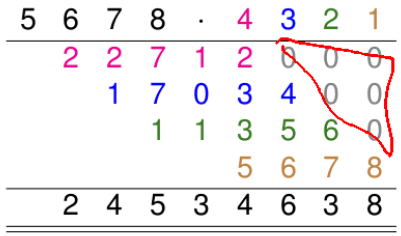
  Auch hier besteht das Ergebnis aus (höchstens) zwei Ziffern u (Zehnerstelle) und v (Einerstelle), also

  $$x + y + z = 10 \cdot u + v$$

  Wir benutzen hier drei Ziffern als Summanden, weil wir später Überträge berücksichtigen wollen.

Analyse des Produkts

- *Zahl mal Zahl:*



Zur Multiplikation zweier Zahlen mit jeweils n Ziffern brauchen wir

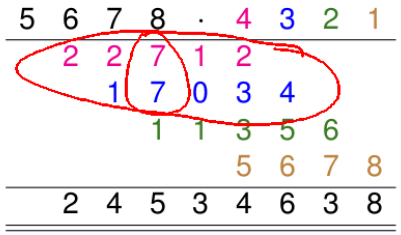
- n Multiplikationen einer n-Ziffern-Zahl mit einer Ziffer, also $n \cdot (2n[+1]) = 2n^2[+n]$ Grundoperationen
- Zwischenergebnisse sind nicht länger als das Endergebnis ($2n$ Ziffern), also $n-1$ Summen von Zahlen mit $2n$ Ziffern, also $(n-1) \cdot 2n = 2n^2 - 2n$ Grundoperationen

Insgesamt: $4n^2 - [2]n$ Grundoperationen

Multiplikation langer Zahlen

Schulmethode:

- gegeben Zahlen a und b
- multipliziere a mit jeder Ziffer von b
- addiere die Teilprodukte



Aufwand

- Wenn die Zahlen klein sind, ist der Aufwand ok.
- Aber wenn die Zahlen sehr lang sind, kann man das Produkt dann schneller ausrechnen als mit der Schulmethode?

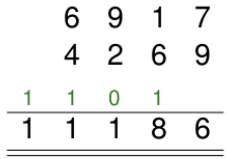
⇒ Wie wollen wir die Zeit oder den Aufwand überhaupt messen?

- Am besten nicht in Sekunden, die irgendein Rechner braucht, denn das könnte für einen anderen Rechner eine ganz andere Zahl sein.
- Außerdem werden die Computer ja von Generation zu Generation immer schneller und leistungsfähiger.

⇒ Wir zählen Grundoperationen: Operationen, die man in einem einzigen Schritt bzw. in einer konstanten Zeiteinheit ausführen kann.

Analyse der Addition

- *Zahl plus Zahl:*

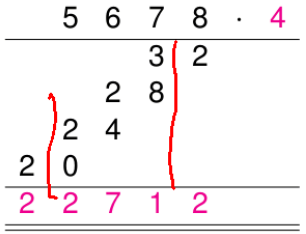


Zur Addition zweier Zahlen mit jeweils n Ziffern brauchen wir n Additionen von 3 Ziffern, also n Grundoperationen.

Ergebnis: Zahl mit $n+1$ Ziffern

Analyse des Teilprodukts

- *Zahl mal Ziffer:*



Zur Multiplikation einer Zahl bestehend aus n Ziffern mit einer einzelnen Ziffer brauchen wir

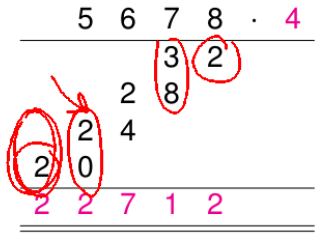
- n Multiplikationen von 2 Ziffern und
- $n+1$ Additionen von 3 Ziffern, wobei in der letzten Spalte eigentlich nichts addiert werden muss,

also $2n$ Grundoperationen.

Ergebnis: Zahl mit $n+1$ Ziffern

Analyse des Teilprodukts

- *Zahl mal Ziffer:*



Zur Multiplikation einer Zahl bestehend aus n Ziffern mit einer einzelnen Ziffer brauchen wir

- n Multiplikationen von 2 Ziffern und
- $n+1$ Additionen von 3 Ziffern, wobei in der letzten Spalte eigentlich nichts addiert werden muss,

also $2n[+1]$ Grundoperationen.

Ergebnis: Zahl mit $n+1$ Ziffern

Analyse des Produkts

- *Zahl mal Zahl:*

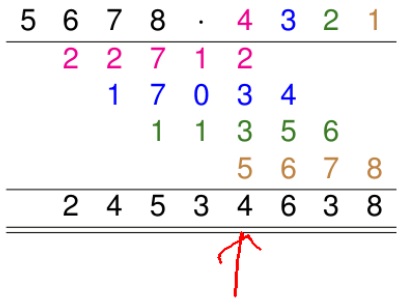
Zur Multiplikation zweier Zahlen mit jeweils n Ziffern brauchen wir

- n Multiplikationen einer n-Ziffern-Zahl mit einer Ziffer, also $n \cdot (2n[+1]) = 2n^2[+n]$ Grundoperationen
- Zwischenergebnisse sind nicht länger als das Endergebnis ($2n$ Ziffern), also $n-1$ Summen von Zahlen mit $2n$ Ziffern, also $(n-1) \cdot 2n = 2n^2 - 2n$ Grundoperationen

Insgesamt: $4n^2 - [2]n$ Grundoperationen

Analyse des Produkts

- *Zahl mal Zahl:*



Genauer:

- Beim Aufsummieren der Zwischenergebnisse muss man eigentlich jeweils nur Zahlen bestehend aus $n+1$ Ziffern addieren. Das ergibt $(n-1)(n+1) = n^2 - 1$ Grundoperationen.

Insgesamt hätte man damit $3n^2[-1]$ – 1 Grundoperationen.

Geht es besser?

Frage:

- Ist das überhaupt gut?
- Vielleicht geht es ja schneller?
- Was wäre denn überhaupt eine signifikante Verbesserung?
- Vielleicht irgendetwas mit $2n^2$?
- Das würde die Zeit auf ca. 2/3 des ursprünglichen Werts senken.
- Aber bei einer Verdoppelung der Zahlenlänge hätte man immer noch eine Vervierfachung der Laufzeit.

- Wir werden diese Frage später beantworten ...

Algorithmen-Beispiele

- Rolf Klein und Tom Kamphans:
  Der Pledge-Algorithmus: Wie man im Dunkeln aus einem Labyrinth entkommt
- Dominik Sibbing und Leif Kobbelt:
  Kreise zeichnen mit Turbo
- Arno Eigenwillig und Kurt Mehlhorn:
  Multiplikation langer Zahlen (schneller als in der Schule)
- Diese und weitere Beispiele:

Taschenbuch der Algorithmen (Springer, 2008)

Übersicht

Effizienzmessung

Ziel:

- Beschreibung der Performance von Algorithmen
- möglichst genau, aber in kurzer und einfacher Form

Exakte Spezifikation der Laufzeit eines Algorithmus
(bzw. einer DS-Operation):

- Menge $\mathcal{I}$ der Instanzen
- Laufzeit des Algorithmus T : $\mathcal{I} \mapsto \mathbb{N}$

Problem: T sehr schwer exakt bestimmbar bzw. beschreibbar

Lösung: Gruppierung der Instanzen (meist nach Größe)

Eingabekodierung

Bei Betrachtung der Länge der Eingabe:

Vorsicht bei der Kodierung!

Beispiel (Primfaktorisierung)

Gegeben: Zahl $x \in \mathbb{N}$

Gesucht: Primfaktoren von x (Primzahlen $p_1, \ldots, p_k$ mit $x = \prod_{i=1}^{k} p_i^{e_i}$)

Bekannt als hartes Problem (wichtig für RSA-Verschlüsselung!)

Eingabekodierung - Beispiel Primfaktorisierung

Beispiel (Primfaktorisierung)

Trivialer Algorithmus

Teste von $y = 2$ bis $\lfloor \sqrt{x} \rfloor$ alle Zahlen, ob diese x teilen und wenn ja, dann bestimme wiederholt das Ergebnis der Division bis die Teilung nicht mehr ohne Rest möglich ist

Laufzeit: $\sqrt{x}$ Teilbarkeitstests und höchstens $\log_2 x$ Divisionen

- Unäre Kodierung von x (x Einsen als Eingabe):
  Laufzeit polynomiell bezüglich der Länge der Eingabe
- Binäre Kodierung von x ($\lceil \log_2 x \rceil$ Bits):
  Laufzeit exponentiell bezüglich der Länge der Eingabe

Eingabekodierung

Betrachtete Eingabegröße:

- Größe von Zahlen: Anzahl Bits bei binärer Kodierung
- Größe von Mengen / Folgen: Anzahl Elemente

Beispiel (Sortieren)

Gegeben: Folge von Zahlen $a_1, \ldots, a_n \in \mathbb{N}$

Gesucht: sortierte Folge der Zahlen

Größe der Eingabe: n

Manchmal Betrachtung von mehr Parametern:

- Größe von Graphen: Anzahl Knoten und Anzahl Kanten

Effizienzmessung

Sei $\mathcal{I}_n$ die Menge der Instanzen der Größe n eines Problems.

Effizienzmaße:

- Worst case:
$$t(n) = \max\{T(i) : i \in \mathcal{I}_n\}$$
- Average case:
$$t(n) = \frac{1}{|\mathcal{I}_n|} \sum_{i \in \mathcal{I}_n} T(i)$$
- Best case:
$$t(n) = \min\{T(i) : i \in \mathcal{I}_n\}$$

(Wir stellen sicher, dass max und min existieren und dass $\mathcal{I}_n$ endlich ist.)

Vor- und Nachteile der Maße

- worst case:
  liefert Garantie für die Effizienz des Algorithmus,
  evt. aber sehr pessimistische Abschätzung
- average case:
  beschreibt durchschnittliche Laufzeit, aber nicht unbedingt übereinstimmend mit dem "typischen Fall" in der Praxis,
  ggf. Verallgemeinerung mit Wahrscheinlichkeitsverteilung
- best case:
  Vergleich mit worst case liefert Aussage über die Abweichung innerhalb der Instanzen gleicher Größe,
  evt. sehr optimistisch

Exakte Formeln für $t(n)$ sind meist sehr aufwendig bzw. nicht möglich!

$\Rightarrow$ betrachte asymptotisches Wachstum ($n \to \infty$)

Wachstumsrate / -ordnung

- $f(n)$ und $g(n)$ haben gleiche Wachstumsrate, falls für große n das Verhältnis durch Konstanten beschränkt ist:

$$\exists c, d \in \mathbb{R}_+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 : \quad c \leq \frac{f(n)}{g(n)} \leq d$$

- $f(n)$ wächst schneller als $g(n)$, wenn es für alle positiven Konstanten c ein n_0 gibt, ab dem $f(n) \geq c \cdot g(n)$ für $n \geq n_0$ gilt, d.h.,

$$\forall c \in \mathbb{R}_+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 : \quad f(n) \geq c \cdot g(n)$$

anders ausgedrückt: $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$

Beispiel

n^2 und $5n^2 - 7n$ haben gleiche Wachstumsrate, da für alle $n \geq 2$ $1 \leq \frac{5n^2-7n}{n^2} \leq 5$ gilt. Beide wachsen schneller als $n^{3/2}$.

Effizienz Effizienzmaße

Wachstumsrate / -ordnung

- $f(n)$ und $g(n)$ haben gleiche Wachstumsrate, falls für große n das Verhältnis durch Konstanten beschränkt ist:

$$\exists c, d \in \mathbb{R}_+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 : \quad c \leq \frac{f(n)}{g(n)} \leq d$$

- $f(n)$ wächst schneller als $g(n)$, wenn es für alle positiven Konstanten c ein n_0 gibt, ab dem $f(n) \geq c \cdot g(n)$ für $n \geq n_0$ gilt, d.h.,

$$\forall c \in \mathbb{R}_+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 : \quad f(n) \geq c \cdot g(n)$$

anders ausgedrückt: $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$

Beispiel

n^2 und $5n^2 - 7n$ haben gleiche Wachstumsrate, da für alle $n \geq 2$ $1 \leq \frac{5n^2-7n}{n^2} \leq 5$ gilt. Beide wachsen schneller als $n^{3/2}$.

H. Seidl (TUM) GAD SS'15 56

Asymptotische Notation

Mengen zur Formalisierung des asymptotischen Verhaltens:

$$\begin{aligned}
O(f(n)) &= \{g(n) : \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\} \\
\Omega(f(n)) &= \{g(n) : \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}
\end{aligned}$$

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

$$\begin{aligned}
o(f(n)) &= \{g(n) : \forall c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\} \\
\omega(f(n)) &= \{g(n) : \forall c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}
\end{aligned}$$

Funktionen sollen Laufzeit bzw. Speicherplatz beschreiben
$\Rightarrow$ Forderung: $\exists n_0 : \forall n \geq n_0 : f(n) > 0$
Manchmal auch: $\forall n : f(n) \geq 0$

Wachstumsrate / -ordnung

- Warum die Betrachtung der Wachstumsrate und die Forderung nur für genügend große n?

  Ziel effizienter Algorithmen: Lösung großer Probleminstanzen

  gesucht: Verfahren, die für große Instanzen noch effizient sind

  Für große n sind Verfahren mit kleinerer Wachstumsrate besser.

- Warum Verzicht auf konstante Faktoren?

  Unser Maschinenmodell ist nur eine Abstraktion von echten Computern und kann die reale Laufzeit sowieso nur bis auf konstante Faktoren bestimmen.

  Daher ist es meistens sinnvoll, Algorithmen mit gleicher Wachstumsrate erstmal als gleichwertig zu betrachten.

- außerdem: Laufzeitangabe durch einfache Funktionen

Asymptotische Notation

Mengen zur Formalisierung des asymptotischen Verhaltens:

$$
\begin{aligned}
O(f(n)) &= \{g(n) : \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\} \\
\Omega(f(n)) &= \{g(n) : \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}
\end{aligned}
$$

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

$$
\begin{aligned}
o(f(n)) &= \{g(n) : \forall c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\} \\
\omega(f(n)) &= \{g(n) : \forall c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}
\end{aligned}
$$

Funktionen sollen Laufzeit bzw. Speicherplatz beschreiben
$\Rightarrow$ Forderung: $\exists n_0 : \forall n \geq n_0 : f(n) > 0$
Manchmal auch: $\forall n : f(n) \geq 0$

Effizienz Effizienzmaße

Asymptotische Notation

Mengen zur Formalisierung des asymptotischen Verhaltens:

$O(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$
$\Omega(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$

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

$o(f(n)) = \{g(n) : \forall c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$
$\omega(f(n)) = \{g(n) : \forall c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$

Funktionen sollen Laufzeit bzw. Speicherplatz beschreiben
$\Rightarrow$ Forderung: $\exists n_0 : \forall n \geq n_0 : f(n) > 0$
Manchmal auch: $\forall n : f(n) \geq 0$

H. Seidl (TUM) GAD SS'15 57

Effizienz Effizienzmaße

Wachstumsrate / -ordnung

- Warum die Betrachtung der Wachstumsrate und die Forderung nur für genügend große n?

  Ziel effizienter Algorithmen: Lösung großer Probleminstanzen

  gesucht: Verfahren, die für große Instanzen noch effizient sind

  Für große n sind Verfahren mit kleinerer Wachstumsrate besser.
- Warum Verzicht auf konstante Faktoren?

  Unser Maschinenmodell ist nur eine Abstraktion von echten Computern und kann die reale Laufzeit sowieso nur bis auf konstante Faktoren bestimmen.

  Daher ist es meistens sinnvoll, Algorithmen mit gleicher Wachstumsrate erstmal als gleichwertig zu betrachten.
- außerdem: Laufzeitangabe durch einfache Funktionen

Effizienz Effizienzmaße

Asymptotische Notation

Mengen zur Formalisierung des asymptotischen Verhaltens:

$O(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$
$\Omega(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$

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

$o(f(n)) = \{g(n) : \forall c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$
$\omega(f(n)) = \{g(n) : \forall c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$

Funktionen sollen Laufzeit bzw. Speicherplatz beschreiben
$\Rightarrow$ Forderung: $\exists n_0 : \forall n \geq n_0 : f(n) > 0$
Manchmal auch: $\forall n : f(n) \geq 0$

H. Seidl (TUM) GAD SS'15 57

Effizienz Effizienzmaße

Asymptotische Notation

Mengen zur Formalisierung des asymptotischen Verhaltens:

$O(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$
$\Omega(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$

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

$o(f(n)) = \{g(n) : \forall c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$
$\omega(f(n)) = \{g(n) : \forall c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$

Funktionen sollen Laufzeit bzw. Speicherplatz beschreiben
$\Rightarrow$ Forderung: $\exists n_0 : \forall n \geq n_0 : f(n) > 0$
Manchmal auch: $\forall n : f(n) \geq 0$

H. Seidl (TUM) GAD SS'15 57

Wachstumsrate / -ordnung

- Warum die Betrachtung der Wachstumsrate und die Forderung nur für genügend große n?

  Ziel effizienter Algorithmen: Lösung großer Probleminstanzen

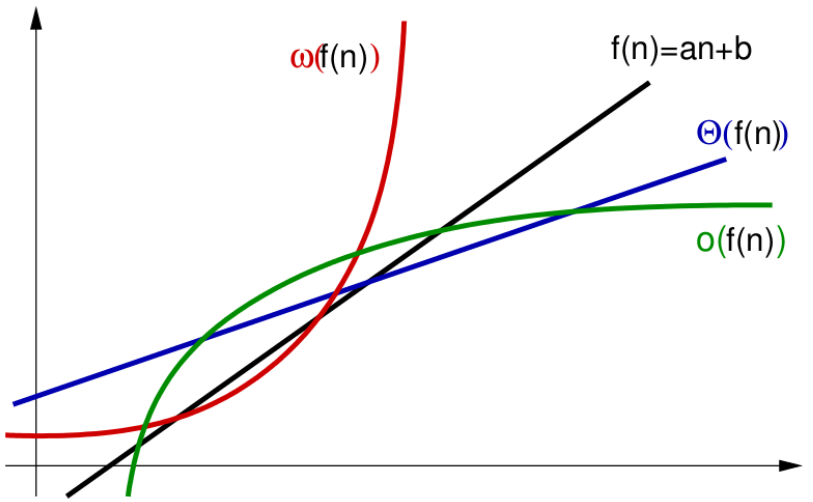
  gesucht: Verfahren, die für große Instanzen noch effizient sind

  Für große n sind Verfahren mit kleinerer Wachstumsrate besser.
- Warum Verzicht auf konstante Faktoren?

  Unser Maschinenmodell ist nur eine Abstraktion von echten Computern und kann die reale Laufzeit sowieso nur bis auf konstante Faktoren bestimmen.

  Daher ist es meistens sinnvoll, Algorithmen mit gleicher Wachstumsrate erstmal als gleichwertig zu betrachten.
- außerdem: Laufzeitangabe durch einfache Funktionen

Asymptotische Notation



Asymptotische Notation

Beispiel

- $5n^2 - 7n \in O(n^2)$, $n^2/10 + 100n \in O(n^2)$, $4n^2 \in O(n^3)$
- $5n^2 - 7n \in \Omega(n^2)$, $n^3 \in \Omega(n^2)$, $n \log n \in \Omega(n)$
- $5n^2 - 7n \in \Theta(n^2)$
- $\log n \in o(n)$, $n^3 \in o(2^n)$
- $n^5 \in \omega(n^3)$, $2^{2n} \in \omega(2^n)$

Asymptotische Notation als Platzhalter

- statt $g(n) \in O(f(n))$ schreibt man oft auch $g(n) = O(f(n))$
- für $f(n) + g(n)$ mit $g(n) \in o(h(n))$ schreibt man auch $f(n) + g(n) = f(n) + o(h(n))$
- statt $O(f(n)) \subseteq O(g(n))$ schreibt man auch $O(f(n)) = O(g(n))$

Beispiel

$$n^3 + n = n^3 + o(n^3) = (1 + o(1))n^3 = O(n^3)$$

O-Notations"gleichungen" sollten nur von links nach rechts gelesen werden!

Übersicht

Wachstumsrate von Polynomen

Lemma

Sei p ein Polynom der Ordnung k bzgl. der Variable n, also

$$p(n) = \sum_{i=0}^{k} a_i \cdot n^i \quad \textit{mit} \quad a_k > 0.$$

Dann ist

$$p(n) \in \Theta(n^k).$$

Wachstumsrate von Polynomen

Beweis.

Zu zeigen: $p(n) \in \mathcal{O}(n^k)$ und $p(n) \in \Omega(n^k)$

$p(n) \in \mathcal{O}(n^k)$:

Für $n \geq 1$ gilt:

$$p(n) \leq \sum_{i=0}^{k} |a_i| \cdot n^i \leq n^k \sum_{i=0}^{k} |a_i|$$

Also ist die Definition

$$\mathcal{O}(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

mit $c = \sum_{i=0}^{k} |a_i|$ und $n_0 = 1$ erfüllt.

Wachstumsrate von Polynomen

Beweis.

$p(n) \in \Omega(n^k)$:

$$A = \sum_{i=0}^{k-1} |a_i|$$

Für positive n gilt dann:

$$p(n) \geq a_k n^k - A n^{k-1} = \frac{a_k}{2} n^k + n^{k-1}\left(\frac{a_k}{2} n - A\right)$$

Also ist die Definition

$$\Omega(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$$

mit $c = a_k/2$ und $n_0 > 2A/a_k$ erfüllt. □

Rechenregeln für $\mathcal{O}$-Notation

Für Funktionen $f(n)$ (bzw. $g(n)$) mit $\exists n_0\ \forall n \geq n_0 : f(n) > 0$ gilt:

Lemma

- $c \cdot f(n) \in \Theta(f(n))$ *für jede Konstante* $c > 0$
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$
- $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$
- $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(f(n))$ *falls* $g(n) \in \mathcal{O}(f(n))$

Die Ausdrücke sind auch korrekt für Ω *statt* $\mathcal{O}$.

Vorsicht, der letzte heißt dann

- $\Omega(f(n) + g(n)) \subseteq \Omega(f(n))$ *falls* $g(n) \in \mathcal{O}(f(n))$

Aber: Vorsicht bei induktiver Anwendung!

Induktions"beweis"

Behauptung:

$$\sum_{i=1}^{n} i = \mathcal{O}(n)$$

"Beweis": Sei $f(n) = n + f(n-1)$ und $f(1) = 1$.

Ind.anfang: $f(1) = \mathcal{O}(1)$

Ind.vor.: Es gelte $f(n-1) = \mathcal{O}(n-1)$

Ind.schritt: Dann gilt

$$f(n) = n + f(n-1) = n + \mathcal{O}(n-1) = \mathcal{O}(n)$$

Also ist

$$f(n) = \sum_{i=1}^{n} i = \mathcal{O}(n)$$

FALSCH!

Ableitungen und $\mathcal{O}$-Notation

Lemma

Seien f *und* g *differenzierbar.*

Dann gilt

- *falls* $f'(n) \in \mathcal{O}(g'(n))$, *dann auch* $f(n) \in \mathcal{O}(g(n))$
- *falls* $f'(n) \in \Omega(g'(n))$, *dann auch* $f(n) \in \Omega(g(n))$
- *falls* $f'(n) \in o(g'(n))$, *dann auch* $f(n) \in o(g(n))$
- *falls* $f'(n) \in \omega(g'(n))$, *dann auch* $f(n) \in \omega(g(n))$

Umgekehrt gilt das im Allgemeinen nicht!

Rechenbeispiele für $\mathcal{O}$-Notation

Beispiel

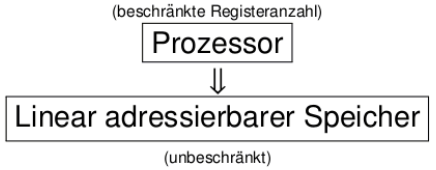
- 1. Lemma:
  - $n^3 - 3n^2 + 2n \in \mathcal{O}(n^3)$
  - $\mathcal{O}(\sum_{i=1}^{n} i) = \mathcal{O}(n^2/2 + n/2) = \mathcal{O}(n^2)$
- 2. Lemma:
  Aus $\log n \in \mathcal{O}(n)$ folgt $n \log n \in \mathcal{O}(n^2)$.
- 3. Lemma:
  - $(\log n)' = 1/n$, $(n)' = 1$ und $1/n \in \mathcal{O}(1)$.
    $\Rightarrow \log n \in \mathcal{O}(n)$

Übersicht

Abstraktion durch Maschinen-/Rechnermodelle

- 1936 Turing-Maschine: kann nicht auf beliebige Speicherzellen zugreifen, nur an der aktuellen Position des Lese-/Schreibkopfs
- 1945 J. von Neumann u.a.: Entwurf des Rechners EDVAC (Electronic Discrete Variable Automatic Computer)
  Programm und Daten teilen sich einen gemeinsamen Speicher
- 1963 John Shepherdson, Howard Sturgis (u.a.):

Random Access Machine (RAM)



RAM: Aufbau

Prozessor:
- beschränkte Anzahl an Registern $R_1, \ldots, R_k$
- Instruktionszeiger zum nächsten Befehl

Programm:
- nummerierte Liste von Befehlen
  (Adressen in Sprungbefehlen entsprechen dieser Nummerierung)

Eingabe:
- steht in Speicherzellen $S[1], \ldots, S[R_1]$

Modell / Reale Rechner:
- unendlicher / endlicher Speicher
- Abhängigkeit / Unabhängigkeit der Größe der Speicherzellen von der Eingabegröße

RAM: Speicher

- unbeschränkt viele Speicherzellen (words) $S[0], S[1], S[2], \ldots$, von denen zu jedem Zeitpunkt nur endlich viele benutzt werden
- beliebig große Zellen führen zu unrealistischen Algorithmen

$\Rightarrow$ Jede Speicherzelle darf bei Eingabelänge n eine Zahl mit $O(\log n)$ Bits speichern.
(Für konstant große Zellen würde man einen Faktor $O(\log n)$ bei der Rechenzeit erhalten.)

$\Rightarrow$ gespeicherte Werte stellen polynomiell in Eingabelänge n beschränkte Zahlen dar (sinnvoll für Array-Indizes; bildet auch geschichtliche Entwicklung $4 \rightarrow 8 \rightarrow 16 \rightarrow 32 \rightarrow 64$ Bit ab)

Begrenzter Parallelismus:
- sequentielles Maschinenmodell, aber
- Verknüpfung logarithmisch vieler Bits in konstanter Zeit

RAM: Befehle

Annahme:

- Jeder Befehl dauert genau eine Zeiteinheit.
- Laufzeit ist Anzahl ausgeführter Befehle

Befehlssatz:

- Registerzuweisung:
  $R_i := c$ (Konst. an Register), $R_i := R_j$ (Register an Register)
- Speicherzugriff:
  $R_i := S[R_j]$ (lesend), $S[R_j] := R_i$ (schreibend)
- Arithmetische / logische Operationen:
  $R_i := R_j \text{ op } R_k$ (binär: op $\in \{+, -, \cdot, \oplus, /, \%, \wedge, \vee, <, \leq, =, \geq, >\}$),
  $R_i := \text{op } R_j$ (unär: op $\in \{-, \neg\}$)
- Sprünge:
  jump x (zu Adresse x), jumpz x R_i (bedingt, falls $R_i = 0$),
  jumpi R_j (zu Adresse aus R_j)

Das entspricht Assembler-Code von realen Maschinen!