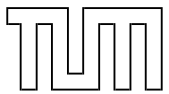
Grundlagen: Algorithmen und Datenstrukturen

Prof. Dr. Hanjo Täubig

Lehrstuhl für Effiziente Algorithmen
(Prof. Dr. Ernst W. Mayr)
Institut für Informatik
Technische Universität München

Sommersemester 2010



Übersicht

Rechenregeln für $\mathcal{O}$-Notation

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

- $\Omega(f(n) + g(n)) = \Omega(f(n))$ *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!

Rechenregeln für $\mathcal{O}$-Notation

Lemma

Seien f und g stetig und 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 $1/n \in \mathcal{O}(1)$.
    $\Rightarrow \log n \in \mathcal{O}(n)$

von-Neumann-Architektur

Quelle: http://wikipedia.org/

1945 J. von Neumann:
Entwurf eines Rechnermodells

Programm und Daten teilen sich einen gemeinsamen Speicher

Besteht aus

- Arithmetic Logic Unit (ALU): Rechenwerk / Prozessor
- Control Unit: Steuerwerk (Befehlsinterpreter)
- Memory: eindimensional adressierbarer Speicher
- I/O Unit: Ein-/Ausgabewerk

Random Access Machine

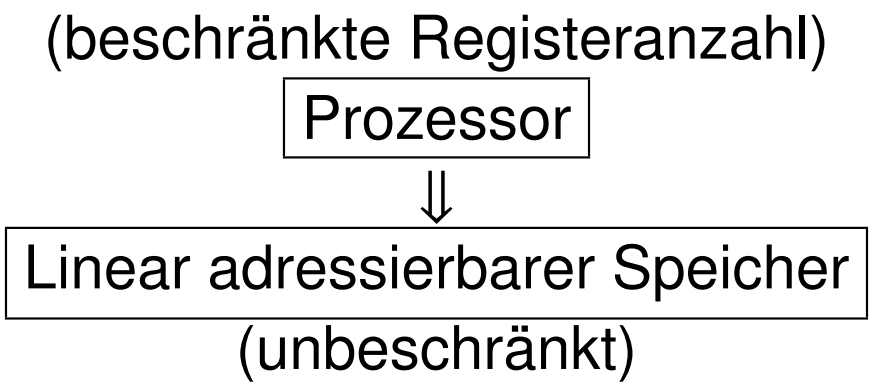
Was ist eigentlich ein Rechenschritt?

$\Rightarrow$ Abstraktion mit Hilfe von Rechnermodellen

Bsp. Turing-Maschine:
kann aber nicht auf beliebige Speicherzellen zugreifen,
sondern nur an der aktuellen Position

1963 J. Shepherdson, H. Sturgis (u.a.):

Random Access Machine (RAM)



Random Access Machine

Speicher:

- Unbeschränkt viele Speicherzellen (words) $s[0], s[1], s[2], \ldots$, von denen zu jedem Zeitpunkt nur endlich viele benutzt werden
- Jede Speicherzelle kann eine polynomiell in n (Eingabegröße) beschränkte Zahl speichern (dafür $\mathcal{O}(\log n)$ Bits)
- Grund: Speicherung von Array-Indizes
- aber: Speicherung beliebig großer Zahlen würde zu unrealistischen Algorithmen führen

Begrenzter Parallelismus:

- Verknüpfung logarithmisch vieler Bits in konstanter Zeit

Random Access Machine

Prozessor:

- Beschränkte Anzahl an Registern $R_1, \ldots, R_k$
- Instruktionszeiger zum nächsten Befehl im Speicher
- Befehlssatz (pro Instruktion eine Zeiteinheit):

$R_i := s[R_j]$	lädt Inhalt von $s[R_j]$ in R_i
$s[R_j] := R_i$	speichert Inhalt von R_i in $s[R_j]$
$R_i := c$	Registerzuweisung für eine Konstante c
$R_i := R_j \; op \; R_k$	binäre Rechenoperation
	$op \in \{+, -, \cdot, \oplus, /, \%, \wedge, \vee, \ldots\}$ oder
	$op \in \{<, \leq, =, \geq, >\}$: 1$\hat{=}$wahr, 0$\hat{=}$falsch
$R_i := op \; R_j$	unäre Rechenoperation
	$op \in \{-, \neg\}$

Random Access Machine

Prozessor (Fortsetzung):

- Befehlssatz (pro Instruktion eine Zeiteinheit):

jump x	Springe zu Position x
jumpz x R_i	Falls $R_i = 0$, dann springe zu Position x
jumpi R_j	Springe zur Adresse aus R_j

entspricht Assembler-Code einer realen Maschine!

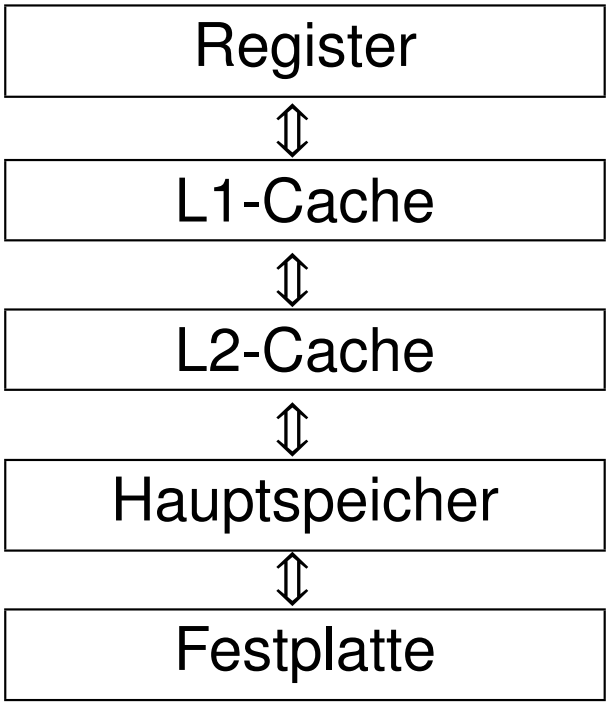
Maschinenmodell

RAM-Modell

- Grundlage für die ersten Computer
- Prinzip gilt auch heute noch

Aber: Einführung von Speicherhierarchien und Multicore-Prozessoren erfordern Anpassung des RAM-Modells
$\Rightarrow$ Herausforderungen an Algorithm Engineering

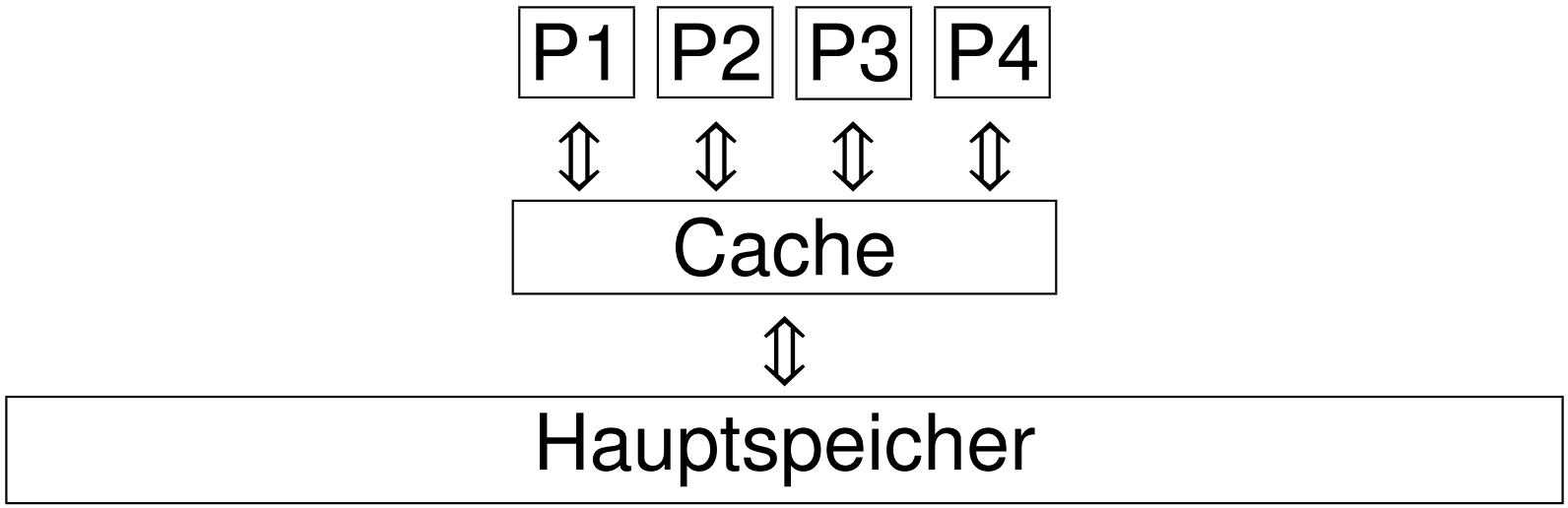
Speicherhierarchie



schneller, kleiner

langsamer, größer

Multicore-Prozessoren



Java

Maschinencode / Assembler umständlich
$\Rightarrow$ besser: Programmiersprache wie Java

Variablendeklarationen:

T v;	Variable v vom Typ T
T v=x;	initialisiert mit Wert x

Variablentypen:

- int, boolean, char, double, ...
- Klassen T, Interfaces I
- T[n]: Feld von Elementen vom Typ T (indexiert von 0 bis $n-1$)

Java-Programme

Allokation und Deallokation von Speicherobjekten:

- v = new T($v_1, \ldots, v_k$); // implizit wird Konstruktor für T aufgerufen

Sprachkonstrukte: (C: Bedingung, I,J: Anweisungen)

- $v = A$; // Ergebnis von Ausdruck A wird an Variable v zugewiesen
- **if** (C) I **else** J
- **do** I **while** (C); **while**(C) I
- **for**($v = a$; $v < e$; v++) I
- **return** v;

Laufzeitanalyse

Was wissen wir?

- $\mathcal{O}$-Kalkül
  ($\mathcal{O}(f(n))$, $\Omega(f(n))$, $\Theta(f(n))$, ...)
- RAM-Modell
  (load, store, jump, ...)
- Java
  (if-else, while, new, ...)

Wie analysieren wir damit Programme?

worst case

Berechnung der worst-case-Laufzeit:

- $T(I)$ sei worst-case-Laufzeit für Konstrukt I
- $T(\text{elementare Zuweisung}) = \mathcal{O}(1)$
- $T(\text{elementarer Vergleich}) = \mathcal{O}(1)$
- $T(\textbf{return } x) = \mathcal{O}(1)$
- $T(\textbf{new } \text{Typ}(\ldots)) = \mathcal{O}(1) + \mathcal{O}(T(\text{Konstruktor}))$
- $T(I_1; I_2) = T(I_1) + T(I_2)$
- $T(\textbf{if } (C)\ I_1\ \textbf{else}\ I_2) = \mathcal{O}(T(C) + \max\{T(I_1), T(I_2)\})$
- $T(\textbf{for}(i = a; i < b; i++)\ I) = \mathcal{O}\left(\sum_{i=a}^{b-1}(1 + T(I))\right)$
- $T(e.m(\ldots)) = \mathcal{O}(1) + T(ss)$, wobei ss Rumpf von m

Beispiel: Vorzeichenausgabe

Algorithmus 6 : `signum`(x)

Input : Zahl $x \in \mathbb{R}$

Output : -1,0 bzw. 1 entsprechend dem Vorzeichen von x

```
if x < 0 then
  return -1
if x > 0 then
  return 1
return 0
```

Wir wissen:

$T(x < 0) = \mathcal{O}(1)$

$T(\textbf{return } \textit{-1}) = \mathcal{O}(1)$

$T(\textbf{if } (C)\ I) = \mathcal{O}(T(C) + T(I))$

Also: $T(\textbf{if } (x < 0) \textbf{ return } -1) = \mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(1)$

Beispiel: Vorzeichenausgabe

Algorithmus 7 : `signum`(x)

Input : Zahl $x \in \mathbb{R}$

Output : -1,0 bzw. 1 entsprechend dem Vorzeichen von x

if $x < 0$ **then**	
⌊ **return** *-1*	$\mathcal{O}(1)$
if $x > 0$ **then**	
⌊ **return** *1*	$\mathcal{O}(1)$
return *0*	$\mathcal{O}(1)$

$$\mathcal{O}(1 + 1 + 1) = \mathcal{O}(1)$$

Beispiel: Minimumsuche

Algorithmus 8 : `minimum`(A, n)

Input : Zahlenfolge in $A[0], \ldots, A[n-1]$
n: Anzahl der Zahlen
Output : Minimum der Zahlen

min $= A[0]$;	$\mathcal{O}(1)$
for $(i = 1; i < n; i{+}{+})$ **do**	$\mathcal{O}(\sum_{i=1}^{n-1}(1 + T(I)))$
⌊ **if** $A[i] <$ min **then** min $= A[i]$	$\mathcal{O}(1)$ ↗
return min	$\mathcal{O}(1)$

$$\mathcal{O}\left(1 + \left(\sum_{i=1}^{n-1} 1\right) + 1\right) = \mathcal{O}(n)$$

Beispiel: Sortieren

Algorithmus 9 : `BubbleSort`(A, n)

Input : Zahlenfolge in $A[0], \ldots, A[n-1]$
n: Anzahl der Zahlen

Output : Sortierte Zahlenfolge A

for $(i = 0; i < n-1; i{+}{+})$ **do**	$O(\sum_{i=0}^{n-2} T(I))$
for $(j = n-2; j \geq i; j{-}{-})$ **do**	$O(\sum_{j=i}^{n-2} T(I))$
if $A[j] > A[j+1]$ **then**	$O(1 + T(I))$
$x = A[j];$	$O(1)$
$A[j] = A[j+1];$	$O(1)$
$A[j+1] = x;$	$O(1)$

$$O\left(\sum_{i=0}^{n-2}\sum_{j=i}^{n-2} 1\right)$$

Beispiel: Sortieren

$$
\begin{aligned}
\mathcal{O}\left(\sum_{i=0}^{n-2}\sum_{j=i}^{n-2} 1\right) &= \sum_{i=0}^{n-2}(n-i-1) \\
&= \sum_{i=1}^{n-1} i \\
&= \frac{n(n-1)}{2} \\
&= \frac{n^2}{2} - \frac{n}{2} \\
&= \mathcal{O}\left(n^2\right)
\end{aligned}
$$

Beispiel: Binäre Suche

Algorithmus 10 : `BinarySearch`(A, n, x)

Input : Zahlenfolge in $A[0], \ldots, A[n-1]$
n: Anzahl der Zahlen
x: gesuchte Zahl

Output : Index der gesuchten Zahl

$\ell = 0;$	$\mathcal{O}(1)$
$r = n - 1;$	$\mathcal{O}(1)$
while $(\ell \leq r)$ **do**	$\mathcal{O}(\sum_{i=1}^{k} T(l))$
$\quad m = \lfloor (r + \ell)/2 \rfloor;$	$\quad \mathcal{O}(1)$ $\uparrow$
$\quad$ **if** $A[m] == x$ **then return** $m;$	$\quad \mathcal{O}(1)$
$\quad$ **if** $A[m] < x$ **then** $\ell = m + 1;$	$\quad \mathcal{O}(1)$
$\quad$ **else** $r = m - 1;$	$\quad \mathcal{O}(1)$
return -1	$\mathcal{O}(1)$

$$\mathcal{O}\left(\sum_{i=1}^{k} 1\right) = \mathcal{O}(k)$$

Beispiel: Binäre Suche

Aber: Wie groß ist die Anzahl der Schleifendurchläufe k?

Größe des verbliebenen Suchintervalls $(r - \ell + 1)$ nach Iteration i:

$$\begin{aligned} s_0 &= n \\ s_{i+1} &\leq \lfloor s_i/2 \rfloor \end{aligned}$$

Bei $s_i < 1$ endet der Algorithmus.

$$\Rightarrow \quad k \leq \log_2 n$$

Gesamtkomplexität: $\mathcal{O}(\log n)$