

ALGORITHMEN UND DATENSTRUKTUREN

TEIL 1: EINFÜHRUNG

Martin Hofmann

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

16. April 2016



- ① ÜBERBLICK ÜBER DIE VORLESUNG
- ② SORTIEREN DURCH EINFÜGEN
 - Die Sortieraufgabe
 - Sortieren durch Einfügen
 - Laufzeitanalyse
- ③ SORTIEREN DURCH MISCHEN
 - Teile und Herrsche
 - Sortieren durch Mischen
 - Laufzeitanalyse
- ④ O-NOTATION
 - Theta, Omega, o, omega
 - O-Notation bei Induktionsbeweisen
- ⑤ MASTER-METHODE
 - Matrizenmultiplikation nach Strassen

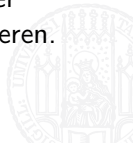


WAS SIND ALGORITHMEN, WIESO ALGORITHMEN?

Ein **Algorithmus** ist eine genau festgelegte **Berechnungsvorschrift**, die zu einer Eingabe oder Problemstellung eine wohldefinierte Ausgabe oder Antwort berechnet.

Ein Algorithmus kann durch ein Programm in einer Programmiersprache angegeben werden, aber typischerweise lässt sich ein und derselbe Algorithmus auf verschiedene Arten und in verschiedenen Programmiersprachen implementieren.

Neben Programmiermethodiken (wie Modularisierung, Objektorientierung, etc.) bilden Algorithmen den *Schlüssel zu erfolgreicher Softwareentwicklung*. Durch Verwendung effizienter Algorithmen lassen sich oft spektakuläre Laufzeitgewinne realisieren.



GIBT ES ALLES SCHON FERTIG?

Für viele Programmiersprachen gibt es eine Menge von Algorithmen bereits fertig in Form von **Bibliotheken**; jedoch nicht für alle Programmiersprachen und/oder oft ist es der Fall, dass eine Bibliotheksfunktion nicht genau passt.

Die **Algorithmik** ist eine lebendige Disziplin. *Neue Problemstellungen* (Bioinformatik, Internet, Mobilität) erfordern neuartige Algorithmen. Für alte Problemstellungen werden bessere Algorithmen entwickelt (kombinatorische Optimierung, Planung). **Schnellere Hardware** rückt bisher unlösbare Probleme in den Bereich des Machbaren (Grafik, Simulation, Lernen, Programmverifikation). **Neue Architekturen** (Parallelisierung, Vernetzung, beschränkte Ressourcen) erfordern neue algorithmische Verfahren.



WAS SIND DATENSTRUKTUREN, WIESO DATENSTRUKTUREN?

Methoden, Daten so anzuordnen, dass Zugriffe möglichst schnell erfolgen.

Konkurrierende Anforderungen: Platzbedarf, verschiedene Arten von Zugriffen.

BEISPIEL: STAPEL (KELLER)

Ist die Größe von vornherein beschränkt, so bietet sich ein Array an. Anderenfalls eine verkettete Liste. Diese verbraucht aber mehr Platz und Zeit. Will man nicht nur auf das oberste Element zugreifen, so benötigt man eine doppelt verkettete Liste oder gar einen Baum, ...

Datenstrukturen und Algorithmen gehen Hand in Hand. Oft bildet eine geeignete Datenstruktur den Kern eines Algorithmus.

BEISPIEL: Heapsort.



INHALT DER VORLESUNG

- KAPITEL I Einführung:** Sortieren durch Einfügen, Sortieren durch Mischen, Laufzeitabschätzungen, Lösen von Rekurrenzen.
- KAPITEL II Sortieren und Suchen:** Heapsort, Quicksort, Median.
- KAPITEL III Datenstrukturen:** , Prioritätsschlangen, Balancierte Binärbäume, Hashtabellen, (*union find*).
- KAPITEL IV Allgemeine Entwurfsmethoden:** Greedy, Backtracking, Branch & Bound, dynamische Programmierung, (Amortisierte Komplexität).
- KAPITEL V Graphalgorithmen:** transitive Hülle, Spannbäume, kürzeste Pfade, (Fluss in Netzwerken).



SORTIEREN

DIE SORTIERAUFGABE

EINGABE: Eine Folge von n Zahlen $\langle a_1, a_2, \dots, a_n \rangle$.

AUSGABE: Eine Umordnung $\langle a_{\pi 1}, a_{\pi 2}, \dots, a_{\pi n} \rangle$, sodass $a_{\pi 1} \leq a_{\pi 2} \leq \dots \leq a_{\pi n}$. Hierbei muss $\pi : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ eine Bijektion sein.

BEISPIEL: $\langle 31, 41, 59, 26, 41, 58 \rangle \mapsto \langle 26, 31, 41, 41, 58, 59 \rangle$.

Allgemeiner: „Zahlen“ \mapsto „Objekte“, „ \leq “ \mapsto „Ordnungsrelation“.

BEISPIELE:

- Zeichenketten, lexikographische Ordnung.
- Autos, Baujahr.
- Geometrische Objekte, Entfernung.
- Transaktionen, Auftragszeitpunkt oder Abwicklungszeitpunkt



SORTIEREN DURCH EINFÜGEN

ALGORITHMUS INSERTION-SORT

Insertion-Sort(A)

1 **for** $j \leftarrow 2$ **to** $\text{length}[A]$

2 **do** $\text{key} \leftarrow A[j]$

3 ▷ Einfügen von $A[j]$ in die sortierte Folge $A[1..j - 1]$

4 $i \leftarrow j - 1$

5 **while** $i > 0$ und $A[i] > \text{key}$

6 **do** $A[i + 1] \leftarrow A[i]$

7 $i \leftarrow i - 1$

8 $A[i + 1] \leftarrow \text{key}$



PSEUDOCODE

Um von Implementierungsdetails zu abstrahieren, verwenden wir zur Angabe von Algorithmen **Pseudocode**:

PSEUDOCODE

Eine prozedurale Sprache, welche

- Neben formellen Anweisungen und Kontrollstrukturen auch **Umgangssprache** enthalten kann,
- Verallgemeinerte Datentypen (wie Mengen, Graphen, etc.) bereitstellt,
- Blockstruktur auch durch **Einrückung** kennzeichnet (wie in Python).
- Achtung: Arrayindizes fangen im Pseudocode bei 1 statt bei 0 wie in Java.

BEISPIEL

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|----|----|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|----|----|----|

| | | | | | | | | | | | |
|---|----|----------|---|---|----|---|---|----|----|---|---|
| 3 | 14 | <u>4</u> | 7 | 1 | 11 | 6 | 8 | 10 | 12 | 5 | 2 |
|---|----|----------|---|---|----|---|---|----|----|---|---|

| | | | | | | | | | | | |
|---|---|----|----------|---|----|---|---|----|----|---|---|
| 3 | 4 | 14 | <u>7</u> | 1 | 11 | 6 | 8 | 10 | 12 | 5 | 2 |
|---|---|----|----------|---|----|---|---|----|----|---|---|

| | | | | | | | | | | | |
|---|---|---|----|----------|----|---|---|----|----|---|---|
| 3 | 4 | 7 | 14 | <u>1</u> | 11 | 6 | 8 | 10 | 12 | 5 | 2 |
|---|---|---|----|----------|----|---|---|----|----|---|---|

| | | | | | | | | | | | |
|---|---|---|---|----|-----------|---|---|----|----|---|---|
| 1 | 3 | 4 | 7 | 14 | <u>11</u> | 6 | 8 | 10 | 12 | 5 | 2 |
|---|---|---|---|----|-----------|---|---|----|----|---|---|

etc.

Gezeigt ist jeweils das Array zu Beginn der Schleife. *key* ist unterstrichen.

Siehe auch diverse Animationen im Internet.



LAUFZEITANALYSE

Wir wollen die **Laufzeit** eines Algorithmus als **Funktion der Eingabegröße** ausdrücken.

Manchmal auch den Verbrauch an anderen Ressourcen wie Speicherplatz, Bandbreite, Prozessorzahl.

Laufzeit bezieht sich auf ein bestimmtes Maschinenmodell, hier RAM (*random access machine*).

Laufzeit kann neben der Größe der Eingabe auch von deren Art abhängen (*worst case, best case, average case*). Meistens *worst case*.



LAUFZEIT VON Insertion-Sort

| Insertion-Sort(A) | Zeit | Wie oft? |
|-----------------------------------------------------|-------|--------------------------|
| 1 for $j \leftarrow 2$ to $length[A]$ | c_1 | n |
| 2 do $key \leftarrow A[j]$ | c_2 | $n - 1$ |
| 3 ▷ Insert $A[j]$ into $A[1..j - 1]$ | | |
| 4 $i \leftarrow j - 1$ | c_4 | $n - 1$ |
| 5 while $i > 0$ and $A[i] > key$ | c_5 | $\sum_{j=2}^n (t_j + 1)$ |
| 6 do $A[i + 1] \leftarrow A[i]$ | c_6 | $\sum_{j=2}^n t_j$ |
| 7 $i \leftarrow i - 1$ | c_7 | $\sum_{j=2}^n t_j$ |
| 8 $A[i + 1] \leftarrow key$ | c_8 | $n - 1$ |

$t_j =$ Anzahl der Durchläufe der **while**-Schleife im j -ten Durchgang.

$c_1 - c_8 =$ un spezifizierte Konstanten.

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5 \sum_{j=2}^n (t_j + 1) + (c_6 + c_7) \sum_{j=2}^n t_j$$

BESTER FALL: ARRAY BEREITS SORTIERT

Ist das Array bereits aufsteigend sortiert, so wird die **while**-Schleife jeweils übersprungen: $t_j = 0$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5 \sum_{j=2}^n (t_j + 1) + (c_6 + c_7) \sum_{j=2}^n t_j$$

$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

Also ist $T(n)$ eine **lineare Funktion** der Eingabegröße n .



SCHLECHTESTER FALL: ARRAY ABSTEIGEND SORTIERT

Ist das Array bereits absteigend sortiert, so wird die `while`-Schleife maximal oft durchlaufen: $t_j = j - 1$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5 \sum_{j=2}^n (t_j + 1) + (c_6 + c_7) \sum_{j=2}^n t_j$$

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

$$\sum_{j=2}^n (j - 1) = \frac{n(n-1)}{2}$$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5 \left(\frac{n(n+1)}{2} - 1 \right) + (c_6 + c_7) \frac{n(n-1)}{2}$$

Also ist $T(n)$ eine **quadratische Funktion** der Eingabegröße n .

WORST-CASE UND AVERAGE-CASE

Meistens geht man vom worst case aus.

- **worst case** Analyse liefert **obere Schranken**
- In vielen Fällen ist der worst case die Regel
- Der (gewichtete) Mittelwert der Laufzeit über alle Eingaben einer festen Länge (**average case**) ist oft bis auf eine multiplikative Konstante nicht besser als der worst case.
- Belastbare Annahmen über die Verteilung von Eingaben sind oft nicht verfügbar.
- Manchmal zeigt sich allerdings, dass in der Praxis der worst case extrem selten auftritt (Beispiel: Simplexverfahren, Typinferenz in ML). Abhilfe *parametrisierte Komplexität*, *smoothed analysis*.



GRÖSSENORDNUNGEN

Um Rechnungen zu vereinfachen und da Konstanten wie c_1, \dots, c_8 sowieso willkürlich sind, beschränkt man sich oft darauf, die **Größenordnung** der Laufzeit anzugeben:

$$10,5n^2 + 100n + 3 = O(n^2)$$

$$50n + 30,45n^{-1} = O(n)$$

$$2^n + n^{10000} = O(2^n)$$

$O(f(n))$ bezeichnet alle Funktionen der **Größenordnung** höchstens $f(n)$. Dies wird später formal definiert.

Laufzeit von Insertion-Sort im schlechtesten Fall ist $O(n^2)$.

Es gibt auch gemischte Notationen wie $2n^2 + O(n)$, also $2n^2$ plus eine Funktion, die höchstens linear wächst.



TEILE UND HERRSCHE

Das Entwurfsverfahren **divide-and-conquer** (Teile und Herrsche, *divide et impera*) dient dem Entwurf **rekursiver Algorithmen**. Die Idee ist es, ein Problem der Größe n in mehrere gleichartige aber kleinere **Teilprobleme** zu zerlegen (*divide*). Aus rekursiv gewonnenen Lösungen der Teilprobleme gilt es dann, eine Gesamtlösung für das ursprüngliche Problem zusammzusetzen (*conquer*).

SORTIEREN DURCH MISCHEN (**MERGE SORT**)

Teile n -elementige Folge in zwei Teilfolgen der Größe $n/2 (+1)$.
Sortiere die beiden Teilfolgen rekursiv.
Füge die nunmehr sortierten Teilfolgen zusammen durch Reißverschlussverfahren.

SORTIEREN DURCH MISCHEN

ALGORITHMUS MERGE-SORT

Merge-Sort(A, p, r)

▷ Sortiere $A[p..r]$,

▷ Lasse den Rest von A unverändert.

1 **if** $p < r$

2 **then** $q \leftarrow (p + r)/2$

3 Merge-Sort(A, p, q)

4 Merge-Sort($A, q + 1, r$)

5 Merge(A, p, q, r)



BEISPIEL

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|----|----|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|----|----|----|

| | | | | | | | | | | | |
|---|----|---|---|---|-----|---|---|----|----|---|---|
| 3 | 14 | 4 | 7 | 1 | 11] | 6 | 8 | 10 | 12 | 5 | 2 |
|---|----|---|---|---|-----|---|---|----|----|---|---|

| | | | | | | | | | | | |
|---|----|----|---|---|-----|---|---|----|----|---|---|
| 3 | 14 | 4] | 7 | 1 | 11] | 6 | 8 | 10 | 12 | 5 | 2 |
|---|----|----|---|---|-----|---|---|----|----|---|---|

| | | | | | | | | | | | |
|---|---|-----|---|---|-----|---|---|----|----|---|---|
| 3 | 4 | 14] | 1 | 7 | 11] | 6 | 8 | 10 | 12 | 5 | 2 |
|---|---|-----|---|---|-----|---|---|----|----|---|---|

| | | | | | | | | | | | |
|---|---|---|---|----|-----|---|---|----|----|---|---|
| 1 | 3 | 4 | 7 | 11 | 14] | 6 | 8 | 10 | 12 | 5 | 2 |
|---|---|---|---|----|-----|---|---|----|----|---|---|

| | | | | | | | | | | | |
|---|---|---|---|----|-----|---|---|---|---|----|----|
| 1 | 3 | 4 | 7 | 11 | 14] | 2 | 5 | 6 | 8 | 10 | 12 |
|---|---|---|---|----|-----|---|---|---|---|----|----|

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|----|----|----|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 10 | 11 | 12 | 14 |
|---|---|---|---|---|---|---|---|----|----|----|----|

Gezeigt sind (nicht aufeinanderfolgende) Zwischenzustände. “]” gibt die Werte von q an.



PROZEDUR Merge

HILFSPROZEDUR MERGE

Merge(A, p, q, r)

- ▷ Sortiere $A[p..r]$ unter der Annahme,
- ▷ dass $A[p..q]$ und $A[q + 1..r]$ sortiert sind.
- ▷ Lasse den Rest von A unverändert.

1 $i \leftarrow p; j \leftarrow q + 1$

2 **for** $k \leftarrow 1$ **to** $r - p + 1$ **do**

3 **if** $j > r$ **or** $(i \leq q$ **and** $A[i] \leq A[j])$

4 **then** $B[k] \leftarrow A[i]; i \leftarrow i + 1$ **else** $B[k] \leftarrow A[j]; j \leftarrow j + 1$

5 **for** $k \leftarrow 1$ **to** $r - p + 1$ **do** $A[k + p - 1] \leftarrow B[k]$



ANALYSE VON Merge-Sort

Sei $T(n)$ die Laufzeit von Merge-Sort.

Das **Aufteilen** braucht $O(1)$ Schritte.

Die **rekursiven Aufrufe** brauchen $2T(n/2)$ Schritte.

Das **Mischen** braucht $O(n)$ Schritte.

ALSO: $T(n) = 2T(n/2) + O(n)$, wenn $n > 1$.

NB $T(1)$ ist irgendein fester Wert.

SATZ

Sei $T : \mathbb{N} \rightarrow \mathbb{R}$ und gelte für n ab einer gewissen Schranke (hier 1)

$$T(n) = 2T(n/2) + O(n)$$

dann ist $T(n) = O(n \log(n))$.

Für große n ist das besser als $O(n^2)$ trotz des Aufwandes für die Verwaltung der Rekursion.

NB: $\log(n)$ bezeichnet den Zweierlogarithmus

MOTIVATION DER LÖSUNG

INTUITIV: Rekursionstiefe: $\log(n)$, auf dem Tiefenniveau k hat man 2^k Teilprobleme der Größe jeweils $g_k := n/2^k$, jedes verlangt einen Mischaufwand von $O(g_k) = O(n/2^k)$. Macht also $O(n)$ auf jedem Niveau: $O(n \log(n))$ insgesamt.

DURCH FORMALISMUS: Sei $T(n)$ für Zweierpotenzen n definiert durch

$$T(n) = \begin{cases} 0, & \text{wenn } n = 1 \\ 2T(n/2) + n, & \text{wenn } n = 2^k, k \geq 1 \end{cases}$$

Dann gilt $T(2^k) = k2^k$, also $T(n) = n \log(n)$ für $n = 2^k$.
Beweis durch Induktion über k (oder n).

Dass es dann für Nicht-Zweierpotenzen auch gilt, kann man beweisen.



ASYMPTOTIK: DEFINITION VON O DEFINITION VON $O(g)$

Seien $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$ Funktionen. (NB Geht auch für andere Definitionsbereiche und auch negative Werte.)

$$O(g) = \{f \mid \exists c > 0, n_0 \in \mathbb{N}. \forall n \geq n_0. f(n) \leq cg(n)\}$$

$O(g)$ ist also eine **Menge von Funktionen**: die Funktionen der **Größenordnung** höchstens g .

Um zu zeigen, dass $f \in O(g)$ ist, muss man eine Schranke $n_0 \in \mathbb{N}$ und einen Faktor $c > 0$ angeben und dann argumentieren, dass $f(n) \leq cg(n)$ für alle $n \geq n_0$ erfüllt ist.

SATZ

Wenn $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c < \infty$, dann ist $f(n) = O(g(n))$.

Umkehrung gilt i.A. nicht.

KONVENTIONEN BEI O

In Ermangelung **guter Notation** für Funktionen schreibt man z.B. $O(n^2)$ und meint damit $O(g)$ wobei $g(n) = n^2$.

Bei $O(m + n^2)$ wird's schwierig. Da muss aus dem *Zusammenhang* klar werden, auf welche Variable sich das O bezieht.

Noch schlimmer sind Notationen wie $O(1 + \alpha)$, wobei $\alpha = n/m$, kommt aber vor. . .

Man schreibt in der Regel $f = O(g)$ anstelle von $f \in O(g)$.

Man verallgemeinert die O -Notation auch auf Funktionen, die bis zu einer gewissen Schranke n_0 negative Werte haben, oder gar undefiniert sind.



BEISPIELE

$$4n^2 + 10n + 3 = O(n^2)$$

Für $n \geq 10$ gilt $4n^2 + 10n + 3 \leq 4n^2 + n^2 + n^2 = 6n^2$.

Also gilt: $4n^2 + 10n + 3 = O(n^2)$ mit $n_0 = 10$ und $c = 6$.

$$an + b = O(n)$$

Für $n \geq b$ gilt $an + b \leq an + n$.

Also gilt $an + b = O(n)$ mit $n_0 \geq b$ und $c \geq a$.

$$n^k = O(b^n) \text{ FÜR } b > 1$$

$\lim_{n \rightarrow \infty} \frac{n^k}{b^n} = \lim_{n \rightarrow \infty} \frac{k!}{\ln(b)^k b^n} = 0$ (L'Hospital), also muss b^n die Funktion n^k irgendwann übertreffen.

Es folgt daher $n^k = O(b^n)$ mit $c = 1$ und n_0 "entsprechend gewählt".

ASYMPTOTIK: AUSDRÜCKE MIT O

Kommt $O(g)$ *in einem Ausdruck* vor, so bezeichnet dieser die Menge aller Funktionen, die man erhält, wenn man die Elemente von $O(g)$ für $O(g)$ *einsetzt*.

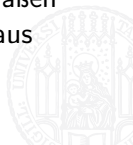
Z.B. ist $n^2 + 100n + 2 \log(n) = n^2 + O(n)$ aber $2n^2 \neq n^2 + O(n)$.

Manchmal kommt O sowohl links, als auch rechts des Gleichheitszeichens vor. Man meint dann eine Inklusion der entsprechenden Mengen. M.a.W. jede Funktion links kommt auch rechts vor.

Z.B.

$$2n^2 + O(n) = O(n^2)$$

Die Verwendung von „ $=$ “ statt „ \in “ oder „ \subseteq “ ist zugegebenermaßen etwas unbefriedigend, z.B. **nicht symmetrisch**, hat sich aber aus *praktischen Gründen* durchgesetzt.



ASYMPTOTIK: $\Theta, \Omega, o, \omega$ DEFINITION VON Ω UND Θ

Seien wieder $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$.

$$\begin{aligned}\Omega(g) &= \{f \mid g \in O(f)\} = \\ &\quad \{f \mid \exists c > 0, n_0 \in \mathbb{N}. \forall n \geq n_0. f(n) \geq cg(n)\} \\ \Theta(g) &= O(g) \cap \Omega(g)\end{aligned}$$

- $f = \Omega(g)$ heißt: g ist eine asymptotische **untere Schranke** für f .
- $f = \Theta(g)$ heißt: f ist von derselben Größenordnung wie g .
- Oft wird O im Sinne von Θ gebraucht.

SATZ

Es gilt: $f = \Omega(g)$ genau dann, wenn $g = O(f)$.

KLEINES o UND ω DEFINITION VON o

$$o(g) = \{f \mid \forall c > 0. \exists n_0 \in \mathbb{N}. \forall n \geq n_0. f(n) \leq cg(n)\}$$

- $f = o(g)$ heisst: g ist eine asymptotische **obere Schranke** für f und f ist nicht asymptotisch proportional zu g . f ist gegenüber g verschwindend gering.
- Es gilt $f = o(g) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$.
- Man schreibt auch $f \in \omega(g)$ für $g \in o(f)$, also $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$.
- Wenn $f = o(g)$ so folgt $f = O(g)$.

BEISPIELE

$$2n^2 = O(n^2), \text{ aber } 2n^2 \neq o(n^2)$$

$$1000n = O(n^2) \text{ und sogar } 1000n = o(n^2)$$

$$n^k = o(b^n) \text{ für } b > 1$$



WEITERE BEISPIELE FÜR ASYMPTOTIK

- $n^2 = o(n^3)$
- $10^{10}n^2 = O(10^{-10}n^2)$ aber nicht mit o statt O .
- $\Theta(\log_b(n)) = \Theta(\log_c(n))$.
Grund: $\log_b(n) = \frac{\log_c(n)}{\log_c(b)}$ (alle Logarithmen sind proportional).
NB: Cormen schreibt $\lg(x)$ für $\log_2(x) = \log(x)$. Allgemein schreibt man $\ln(x)$ für $\log_e(x)$.
- *Achtung:* $4^n \notin O(2^n)$.



KOMPLIZIERTERE BEISPIELE

$f = O(g)$ UND $g = O(h)$ IMPLIZIERT $f = O(h)$

Sei $f = O(g)$ mit n_1, c_1 und $g = O(h)$ mit n_2, c_2 .

Für $n \geq \max(n_1, n_2)$ gilt $f(n) \leq c_1 g(n) \leq c_1 c_2 h(n)$, also $f = O(h)$ mit $n_0 = \max(n_1, n_2)$ und $c = c_1 c_2$.

NB: Ähnliches gilt für $\Theta, \Omega, o, \omega$.

$\log n = o(n^\epsilon)$ FÜR $\epsilon > 0$

$$\lim_{n \rightarrow \infty} \frac{\log n}{n^\epsilon} = \lim_{n \rightarrow \infty} \frac{1}{\epsilon n^{\epsilon-1}} = \lim_{n \rightarrow \infty} \frac{1}{\epsilon n^\epsilon} = 0$$

$\log n! = \Theta(n \log n)$

“O”: $\log n! = \log 1 + \log 2 + \dots + \log n \leq n \log n$.

“Ω”: $\log n! \geq \log(n/2) + \log(n/2 + 1) + \dots + \log(n) \geq n/2 \log(n/2) \geq 1/4 \cdot n \log(n)$

ASYMPTOTIK UND INDUKTION

Will man eine asymptotische Beziehung durch Induktion beweisen, so muss man die Konstante c ein für alle Mal wählen und darf sie nicht während des Induktionsschritts abändern:

FALSCHER BEWEIS VON $\sum_{i=1}^n i = O(n)$

Sei $S(n) = \sum_{i=1}^n i$, also $S(n) = n(n+1)/2$.

$S(1) = O(1)$.

Sei $S(n) = O(n)$, dann gilt

$S(n+1) = S(n) + (n+1) = O(n) + O(n) = O(n)$.



ASYMPTOTIK UND INDUKTION

RICHTIGER BEWEIS VON $S(n) = \Omega(n^2)$

Wir versuchen die Induktion mit einem noch zu bestimmenden c durchzukriegen und leiten Bedingungen an diese Konstante ab.

$S(1) = 1$, daraus $c \leq 1$.

Sei $S(n) \geq cn^2$. Es ist

$S(n+1) = S(n) + n + 1 \geq cn^2 + n + 1 \geq cn^2 + 2cn + c = c(n+1)^2$
falls $c \leq 1/2$.

Also funktioniert's mit $c \leq 1/2$, insbesondere $c = 1/2$.

Die Induktion fing bei $n = 1$ an, also können wir $n_0 = 1$ nehmen.



LÖSEN VON REKURRENZEN BEI DIVIDE-AND-CONQUER

Bei der Analyse von **divide-and-conquer** Algorithmen stößt man auf Rekurrenzen der Form:

$$T(n) = aT(n/b) + f(n)$$

Das passiert dann, wenn ein Problem der Größe n in a Teilprobleme der Größe n/b zerlegt wird und der Aufwand für das Aufteilen und Zusammenfassen der Teilresultate Aufwand $f(n)$ erfordert.

BEMERKUNG: n/b steht hier für $\lfloor n/b \rfloor$ oder $\lceil n/b \rceil$.

MEHR NOCH: $aT(n/b)$ kann man hier sogar als $a_1 T(\lfloor n/b \rfloor) + a_2 T(\lceil n/b \rceil)$ mit $a_1 + a_2 = a$ lesen.

Die **Master-Methode** liefert eine kochrezeptartige Lösung für derartige Rekurrenzen.



HAUPTSATZ DER MASTER-METHODE

MASTER-THEOREM

Seien $a \geq 1, b > 1$ Konstanten, $f, T : \mathbb{N} \rightarrow \mathbb{R}$ Funktionen und gelte

$$T(n) = aT(n/b) + f(n)$$

Dann erfüllt T die folgenden Größenordnungsbeziehungen:

- 1 Wenn $f(n) = O(n^{\log_b a - \epsilon})$ für ein $\epsilon > 0$ und $f = \Omega(1)$, so gilt $T(n) = \Theta(n^{\log_b a})$.
- 2 Wenn $f(n) = \Theta(n^{\log_b a})$, so gilt $T(n) = \Theta(n^{\log_b a} \log(n))$.
- 3 Wenn $f(n) = \Omega(n^{\log_b a + \epsilon})$ für ein $\epsilon > 0$ und außerdem $af(n/b) \leq cf(n)$ für ein $c < 1$ und genügend großes n , so gilt $T(n) = \Theta(f(n))$.

BEACHTEN: Ist $f(n) = n^{\log_b(a)+\epsilon}$, so gilt

$$af(n/b) = an^{\log_b(a)+\epsilon} / b^{\log_b(a)+\epsilon} = af(n)/a/b^\epsilon \leq cf(n)$$

mit $c = b^{-\epsilon} < 1$.

ANMERKUNGEN

- Beachte: $\log_b(a) = \log(a) / \log(b)$.
- Wichtigster Spezialfall: Wenn $T(n) = aT(n/a) + O(n)$, dann $T(n) = O(n \log n)$.
- Beispiel einer Funktion, die zwar $\Omega(n)$ ist, aber nicht die Regularitätsbedingung (z.B. für $a = 1, b = 2$) erfüllt:
$$f(n) = 2^{2^{\lceil \log(\log(n)) \rceil}}$$



BEISPIELE FÜR DIE MASTER METHODE

- Die Laufzeit $T(n)$ von Merge-Sort genügt der Beziehung:
 $T(n) = 2T(n/2) + \Theta(n)$ somit
 $T(n) = \Theta(n^{\log_2(2)} \log(n)) = \Theta(n \log(n))$.
- $T(n) = 2T(n/3) + n \Rightarrow T(n) = \Theta(n)$. Hier $\log_b(a) = 0,63$.
- $T(n) = 2T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2)$. Hier $\log_b(a) = 1$.
- $T(n) = 4T(n/2) + n \Rightarrow T(n) = \Theta(n^2)$. Hier $\log_b(a) = 2$.
- Die Rekurrenz $T(n) = 2T(n/2) + n \log n$ kann man mit der Master-Methode nicht lösen. Die Lösung ist hier
 $T(n) = \Theta(n \log^2(n))$.



BEWEISIDEE

- Rekursionstiefe: $\log_b(n)$.
- Auf Tiefenniveau i gibt es $(a/b)^i$ Mischaufgaben der Größe jeweils $f(n/b^i)$.
- Gesamtarbeit: $\sum_{i=0}^{\log_b(n)} (a/b)^i f(n/b^i)$.
- Daraus die drei Fälle mit geometrischer Reihe und elementaren Rechengesetzen, wie z.B. $a^{\log_b(n)} = n^{\log_b(a)}$.



MATRIZENMULTIPLIKATION

Seien $A = (a_{ik}), B = (b_{jk})$ zwei $n \times n$ Matrizen. Das Produkt $C = AB$ ist definiert durch

$$c_{ik} = \sum_{j=1}^n a_{ij} b_{jk}$$

Das macht $\Theta(n^3)$ Additionen und Multiplikationen.
Matrizen kann man **blockweise** multiplizieren:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

Die "kleinen" Produkte AB sind hier selbst wieder Matrizenprodukte.

Die blockweise Multiplikation führt auf dasselbe Ergebnis, wie die direkte Verwendung der Definition.



MATRIZENMULTIPLIKATION MIT *divide-and-conquer*

$T(n)$ = Anzahl der Operationen erforderlich zur Multiplikation zweier $n \times n$ Matrizen

Es ist mit **divide-and-conquer**:

$$T(n) = 8T(n/2) + \Theta(n^2)$$

Es ist $\log_2(8) = 3$ und $\Theta(n^2) = O(n^{3-\epsilon})$, z.B. mit $\epsilon = 1$, also $T(n) = \Theta(n^3)$.

Keine Verbesserung, da auch die direkte Verwendung der Definition ($\Theta(n^3)$ Rechenoperationen ($\Theta(n^3)$ Multiplikationen, $\Theta(n^2)$ Additionen.) verbraucht.



STRASSENS ERSTAUNLICHER ALGORITHMUS

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} R & S \\ T & U \end{pmatrix}$$

wobei (V. Strassens geniale Idee):

$$R = P_5 + P_4 - P_2 + P_6$$

$$S = P_1 + P_2$$

$$T = P_3 + P_4$$

$$U = P_5 + P_1 - P_3 - P_7$$

$$P_1 = A(F - H)$$

$$P_2 = (A + B)H$$

$$P_3 = (C + D)E$$

$$P_4 = D(G - E)$$

$$P_5 = (A + D)(E + H)$$

$$P_6 = (B - D)(G + H)$$

$$P_7 = (A - C)(E + F)$$

Nunmehr ist

$$T(n) = 7T(n/2) + \Theta(n^2)$$

Also: $T(n) = \Theta(n^{\log_2(7)}) = O(n^{2,81})$.

