

# Algorithmen und Datenstrukturen

SoSe 2008 in Trier

Henning Fernau

Universität Trier

fernau@uni-trier.de

# Algorithmen und Datenstrukturen

## Gesamtübersicht

- Organisatorisches / Einführung
- Grundlagen: RAM,  $\mathcal{O}$ -Notation, Rekursion, Datenstrukturen
- Sortieren
- Wörterbücher und Mengen
- Graphen und Graphalgorithmen

## Rekursive Algorithmen / Prozeduren

Prozeduren kann man bei ihrem Aufruf *Parameter* übergeben, und bei ihrer Beendigung können Prozeduren auch Werte an ihren Aufrufer zurückgeben.

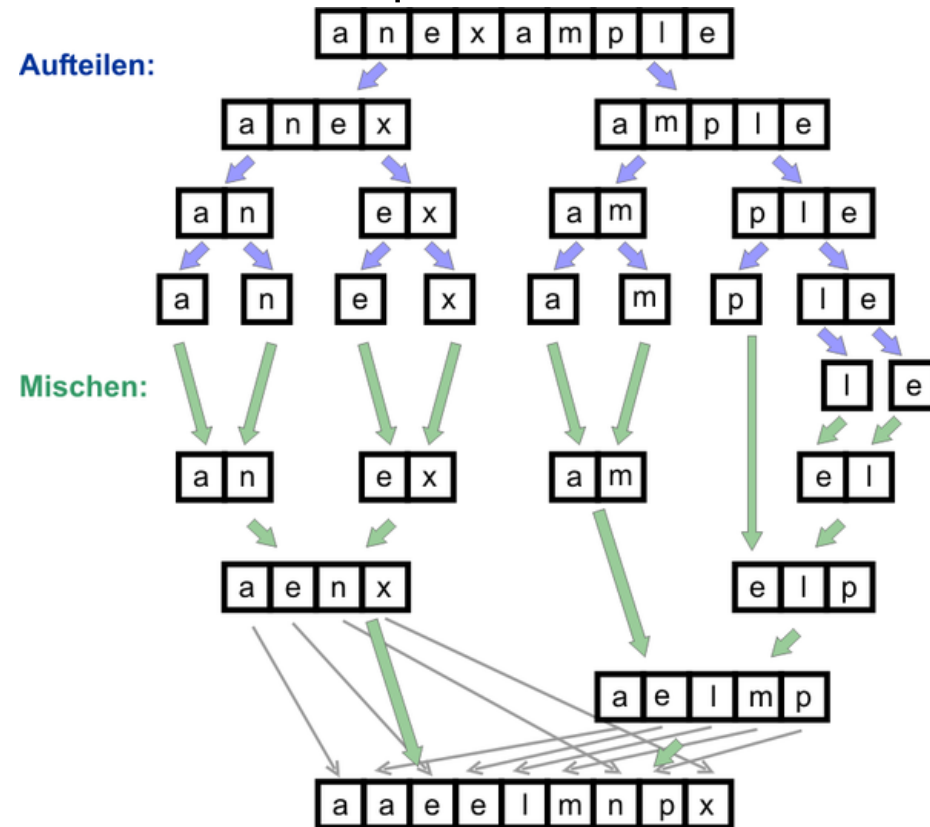
Diese Eigenschaften laden geradezu dazu ein, dass Prozeduren in ihrem Inneren wiederum “sich selbst” aufrufen; solche Prozeduraufrufe heißen auch *rekursiv*. Prozeduren mit rekursiven Aufrufen heißen auch selbst schlicht *rekursiv*, und ebenso Algorithmen, die mit rekursiven Prozeduren dargestellt werden.

## **Rekursion** am Beispiel: Sortieren durch Mischen (Mergesort)

```
funktion mergesort(feld);  
  falls (Größe von feld <= 1) dann antworte feld  
  sonst  
    halbiere bestmöglich feld in linkes, rechtes  
    antworte merge(mergesort(linkes), mergesort(rechtes));
```

Ich wechsele teilweise bewusst ab und an Darstellungen von Algorithmen, so wie Sie es auch in der Literatur vorfinden; hier ähnlich zu Wikipedia.

## Sortieren durch Mischen: Ein Beispiel



## Laufzeitabschätzung

Es sei  $T(n)$  eine obere Schranke für die Laufzeit von `mergesort` auf Feldern der Größe  $n$ . Auf Feldern der Größe 1 braucht der Algorithmus offenbar konstante Zeit, d.h.,  $T(1) = c$ . Wenn wir annehmen, `merge` brauche Linearzeit ebenso wie das (implizite) Feldaufspalten (Halbieren), so gilt für  $n > 1$ :

$$T(n) \leq a \cdot n + T(h_1(n)) + T(h_2(n))$$

für eine geeignete Konstante  $a \geq 1$ . Hierbei seien  $h_1(n)$  und  $h_2(n)$  die Größen der beiden “Hälften”, also gilt stets  $h_1(n) + h_2(n) = n$ , und  $h_i(n) = n/2$ , falls  $n$  gerade ist; sonst muss geeignet auf- bzw. abgerundet werden.

Sind wir nur am größenordnungsmäßigen Wachstum von  $T$  interessiert, können wir  $c = a = 1$  setzen.

Außerdem können wir  $h_1(n) = h_2(n) = \lceil n/2 \rceil$  setzen (stets aufrunden). (Warum?)

Wie löst man aber eine *Rekursionsgleichung* der Form  $T(n) \leq n + 2T(\lceil n/2 \rceil)$ ?

## Analyse von Rekursionen

Bei der Ressourcen-Analyse von rekursiven Algorithmen entstehen oft Rekursionsgleichungen, die gelöst werden müssen. In einer Rekursionsgleichung für eine Funktion  $f$  wird zur Berechnung der Funktion für den Wert  $n$  die Funktion für andere, kleinere Werte verwendet. Allgemein hat eine Rekursionsgleichung die Form  $f(n) = G(f(1); f(2); \dots; f(n - 1))$ .

Eine Rekursionsgleichung definiert eine Funktionenschar (alle Funktionen die die Gleichung erfüllen).

Aus der Funktionenschar wird eine Funktion durch Festlegen weiterer Parameter spezifiziert, z.B. durch Lösen der *Anfangswertbedingungen*  $f(1) = a_1$ ,  $f(2) = a_2$ , ...,  $f(k) = a_k$ .

**Beispiel:** Betrachte die Rekursionsgleichung  $T(n) = 2T(n - 1)$  und  $T(0) = 1$ . Offenbar gilt:  $T(n) = 2^n$  (*explizite Darstellung*).

## Lösen von Rekursionsgleichungen

**Ziel:** Überführen in eine geschlossene, nichtrekursive Form

**Induktive Einsetzungsmethode:** Anhand der Rekursionsgleichung wird eine Lösung geraten und durch Induktion bestätigt. Oft enthält die vermutete Lösung noch zu spezifizierende Parameter, diese ergeben sich oft durch einen Koeffizientenvergleich der für den Induktionsbeweis notwendig wird. (konstruktive Induktion).

Bei der **Iterationsmethode** wird die Funktion auf der rechten Seite immer wieder durch die Rekursionsgleichung ersetzt. Dann wird versucht die rechte Seite in eine geschlossenen Form zu bringen.



## Rekursive Algorithmen — *Divide and Conquer*: Teile und herrsche

Viele Algorithmen zerlegen das zu lösende Problem einer Größe in 2 oder mehrere Teilprobleme mit einer geringeren Größe.

Die Teilprobleme werden (rekursiv) berechnet und die Lösung aus den Ergebnissen zusammengesetzt. Die Laufzeit für eine Eingabe der Länge  $n$  setzt sich also zusammen aus der Laufzeit für die kleineren Eingaben plus der Laufzeit die benötigt, wird die Eingabe zu zerlegen und die Lösung zusammenzufügen.

```
Procedure: DivideAndConquer(x:Eingabe)
IF |x| = 1 THEN
    berechne Lösung l für x direkt.
ELSE zerlege x in zwei (gleichgroße) Teile x1 und x2
    l1 = DivideAndConquer(x1)
    l2 = DivideAndConquer(x2)
    berechne Lösung l für x aus l1 und l2.
RETURN l
```

Beispiel: Mergesort

## Rekursive Algorithmen — Divide and Conquer Analyse

Der Aufwand für das Zerlegen und Zusammenfügen sei  $cn + d$ .

Die Komplexität ergibt sich damit zu:  $T(n) \leq 2T(n/2) + cn + d$ :

Wir versuchen eine Lösung zu der Rekursionsgleichung zu finden und diese zu verifizieren.

Bei jedem Aufruf halbiert sich die Problemgröße, nach  $\log_2(n)$  vielen Schritten wird die Größe 1 und damit das Ende der Rekursion erreicht. Da sich die Anzahl der Teilprobleme jedesmal verdoppelt, summiert sich die Anzahl der Schritte (vermutlich) jeweils wieder zu  $cn + d$ .

Wir vermuten daher eine Lösung der Form  $T(n) = \alpha + \beta n + \delta n \log_2 n$ .

Die vermutete Lösung  $\alpha + \beta n + \delta n \log_2 n$  setzen wir induktiv ein:

$$\begin{aligned} T(n) &= cn + d + 2f(n/2) \\ &= cn + d + 2(\alpha + \beta(n/2) + \delta(n/2) \log_2(n/2)) \\ &= cn + d + 2\alpha + \beta n + \delta n \log_2 n - \delta n \end{aligned}$$

Koeffizientenvergleich liefert  $\alpha = -d$  und  $\delta = c$ . Dies ergibt Funktionen der Form

$$\{-d + \beta n + cn \log_2 n \mid \beta \in \mathbb{R}\}.$$

Der Wert von  $\beta$  ergibt sich aus der Anfangswertbedingung.

Für  $T(1) = 13$  z.B. ergibt sich aus  $-d + \beta + c \log_2 1 = 13$ , dass  $\beta = d + 13$ .

**Beispiel:** Iterationsmethode für  $f(n) = n + 3f(n/4)$ .

Iteriertes Einsetzen und Ersetzen der Summe ergibt:

$$\begin{aligned} f(n) &= n + 3f(n/4) = n + 3(n/4 + 3f(n/16)) \\ &= n + 3n/4 + 9(n/16 + 3f(n/64)) \leq n + 3n/4 + 9n/16 + \dots \\ &\leq n \cdot \sum_{i \geq 0} (3/4)^i = n \cdot \frac{1}{1 - 3/4} = 4n = \mathcal{O}(n) \end{aligned}$$

Die Anfangswertbedingung  $f(1) = a$  wirkt sich nur auf die in der  $\mathcal{O}$ -Notation verborgenen Konstanten aus.

## Hauptsatz der Laufzeitfunktionen (engl.: master theorem of recursion)

Betrachte folgende Laufzeitfunktion:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n), \quad \text{mit } T(1) = \Theta(1)$$

Hierbei sind  $a \geq 1$  und  $b > 1$  Konstanten.

$f(n)$  bezeichnet eine von  $T(n)$  unabhängige und nicht negative Funktion.

**Interpretation der Rekurrenz  $T(n)$ :**

$a$ : Anzahl der Unterprobleme in der Rekursion

$b$ : Anzahl Teile des Originalproblems, welches wiederum durch alle Unterprobleme repräsentiert wird

$f(n)$ : Kosten (Aufwand) die durch die Aufspaltung des Problems und der Kombination der Teillösungen entstehen

## Hauptsatz der Laufzeitfunktionen (engl.: master theorem of recursion)

	Erster Fall	Zweiter Fall	Dritter Fall
<b>Allgemein</b> Falls gilt:	$f(n) \in O(n^{\log_b a - \epsilon})$ für ein $\epsilon > 0$	$f(n) \in \Theta(n^{\log_b a})$	$f(n) \in \Omega(n^{\log_b a + \epsilon})$ für ein $\epsilon > 0$ , und ebenfalls für ein $c$ mit $0 < c < 1$ und hinreichend große $n$ gilt: $af(\frac{n}{b}) \leq cf(n)$
Dann folgt:	$T(n) \in \Theta(n^{\log_b a})$	$T(n) \in \Theta(n^{\log_b a} \log(n))$	$T(n) \in \Theta(f(n))$
<b>Beispiel</b>	$T(n) = 8T(\frac{n}{2}) + 1000n^2$	$T(n) = 2T(\frac{n}{2}) + 10n$	$T(n) = 2T(\frac{n}{2}) + n^2$
Aus der Formel ist folgendes abzulesen:	$a = 8, b = 2$ $f(n) = 1000n^2$ $\log_b a = \log_2 8 = 3$	$a = 2, b = 2$ $f(n) = 10n$ $\log_b a = \log_2 2 = 1$	$a = 2, b = 2$ $f(n) = n^2$ $\log_b a = \log_2 2 = 1$
1. Bedingung:	$f(n) \in O(n^{\log_b a - \epsilon})$ für ein $\epsilon > 0$	$f(n) \in \Theta(n^{\log_b a})$	$f(n) \in \Omega(n^{\log_b a + \epsilon})$ für ein $\epsilon > 0$
Werte einsetzen:	$1000n^2 \in O(n^{3-\epsilon})$	$10n \in \Theta(n^1)$	$n^2 \in \Omega(n^{1+\epsilon})$
Wähle $\epsilon > 0$ :	$1000n^2 \in O(n^2)$ mit $\epsilon = 1$ ✓	$10n \in \Theta(n)$ ✓	$n^2 \in \Omega(n^2)$ mit $\epsilon = 1$ ✓
2. Bedingung: (nur im 3. Fall)			$af(\frac{n}{b}) \leq cf(n)$ Setze auch hier obige Werte ein: $2(\frac{n}{2})^2 \leq cn^2 \Leftrightarrow \frac{1}{2}n^2 \leq cn^2$ Wähle $c = \frac{1}{2}$ : $\forall n \geq 1 : \frac{1}{2}n^2 \leq \frac{1}{2}n^2$ ✓
Ist die Bedingung erfüllt, so gilt:	$T(n) \in \Theta(n^{\log_b a})$	$T(n) \in \Theta(n^{\log_b a} \log(n))$	$T(n) \in \Theta(f(n))$
Damit gilt für die Laufzeitfunktion:	$T(n) \in \Theta(n^3)$	$T(n) \in \Theta(n \log(n))$	$T(n) \in \Theta(n^2)$

**Hauptsatz der Laufzeitfunktionen** (engl.: master theorem of recursion)

... meistert nicht alles:

Betrachte  $T(n) = 2T(n/2) + n \log_2(n)$ , also  $f(n) = n \log_2(n)$ , sowie  $a = b = 2$ .

Fall 1: Gilt  $f(n) = \mathcal{O}(n^{1-\epsilon})$  ? Nein.

Fall 2: Gilt  $f(n) = \Theta(n)$  ? Nein.

Fall 3: Gilt  $f(n) = \Omega(n^{1+\epsilon})$  ? Nein.

Das "Master Theorem" ist also nicht anwendbar.

## Hauptsatz der Laufzeitfunktionen (allgemeinere Form)

Sei  $T : \mathbb{N} \rightarrow \mathbb{N}$  die zu untersuchende Abbildung der Form

$$T(n) = \sum_{i=1}^m T(\alpha_i n) + f(n),$$

wobei  $\alpha_i \in \mathbb{R}: 0 < \alpha_i < 1$ ,  $m \in \mathbb{N}: m \geq 1$  und  $f(n) \in \Theta(n^k)$  mit  $k \in \mathbb{N}: k \geq 0$ .

$T$  wird hierfür implizit durch  $T(x) := T(\lfloor x \rfloor)$  oder  $T(\lceil x \rceil)$  für  $x \in \mathbb{R}^+$  fortgesetzt.

$$\leadsto T(n) \in \begin{cases} \Theta(n^k) & \text{falls } \sum_{i=1}^m (\alpha_i^k) < 1 \\ \Theta(n^k \log n) & \text{falls } \sum_{i=1}^m (\alpha_i^k) = 1 \\ \Theta(n^c) \text{ mit } \sum_{i=1}^m (\alpha_i^c) = 1 & \text{falls } \sum_{i=1}^m (\alpha_i^k) > 1 \end{cases}$$

Iterationsmethode für den **Hauptsatz der Laufzeitfunktionen**:

$$\text{Lemma: } T(n) = \Theta(n^{\log_b(a)}) + \overbrace{\sum_{j=0}^{\log_b(n)-1} [a^j \cdot f(n/b^j)]}^{g(n)}.$$

Beweis: Es gilt:

$$\begin{aligned} T(n) &= f(n) + T(n/b) \\ &= a^0(f(n/b^0)) + a^1(f(n/b^1) + a^1(T(n/b^2))) \\ &= a^0(f(n/b^0)) + a^1(f(n/b^1)) + a^2(T(n/b^2)) \\ \dots &= a^0(f(n/b^0)) + a^1(f(n/b^1)) + a^2(f(n/b^2)) + \dots + a^{\log_b(n)-1}(T(n/b^{\log_b(n)-1})) + a^{\log_b(n)} \cdot T(1) \end{aligned}$$

Strenggenommen gilt dies nur, falls  $n$  eine Potenz von  $b$  ist...



**Zum Beweis** des Hauptsatzes müsste man noch  $g(n)$  diskutieren.

Wir erörtern nur einen Fall.

Die anderen beiden Fälle sind ähnlich (nur etwas schwieriger).

Gilt  $f(n) = \Theta(n^{\log_b(a)})$ , so folgt:

$$\begin{aligned}g(n) &= \sum_{j=0}^{\log_b(n)-1} [a^j \cdot f(n/b^j)] \\&= \sum_{j=0}^{\log_b(n)-1} [a^j \cdot \Theta((n/b^j)^{\log_b(a)})] \\&= \Theta\left(\sum_{j=0}^{\log_b(n)-1} [a^j \cdot (n/b^j)^{\log_b(a)}]\right) \\&= \Theta(n^{\log_b(a)} \cdot \sum_{j=0}^{\log_b(n)-1} (a/b^{\log_b(a)})^j) \\&= \Theta(n^{\log_b(a)} \cdot \sum_{j=0}^{\log_b(n)-1} 1) \\&= \Theta(n^{\log_b(a)} \cdot \log(n))\end{aligned}$$