Algorithmen und Datenstrukturen

3. LAUFZEITANALYSEN

Komplexität

- Gegeben
  - Ein zu lösendes Problem
- Wünschenswert
  - Algorithmus zur Berechnung der Lösung mit möglichst **geringem Aufwand**
- Daher
  - Abschätzung des Aufwands von Algorithmen (Komplexität)
  - Mindestaufwand zur Lösung von Problemen einer bestimmten Klasse

Motivierendes Beispiel

Sequentielle Suche in Folgen

- Gegeben:
  - n Zahlen, z. B. Folge mit n Zahlen A[0 … n-1] , mit n > 0 und Zahlen sind verschieden
  - Zahl b
- Gesucht:
  - Index: i ∈ {0, …, n-1} mit b = A[i] falls Index existiert, sonst i = n
- Lösung für das Problem:

```
i = 0;
while (i < n  &&  b != A[i])
     i++;
```

Motivierendes Beispiel (2)

Aufwand der Suche

- Ergebnis hängt von der Eingabe ab, d.h. vom gewählten Wert n, den Zahlen A[0], …, A[n] und von b
  1. Erfolgreiche Suche: wenn b = A[i]: S = i+1 Schritte
  2. Erfolglose Suche: S = n+1 Schritte
- Problem mit 1. Aussage: hängt von zu vielen Parametern ab
  - Ziel: globale Aussage zu finden, die nur von einer einfachen Größe abhängt, z.B. die Länge ***n*** der Folge
- Fragen zur Komplexität:
  a) Wie groß ist S für gegebenes n im schlechtesten Fall?
  b) Wie groß ist S für gegebenes n im Mittel?

Analyse für erfolgreiche Suche - Frage a

- Im schlechtesten Fall
  - b wird erst im letzten Schritt gefunden: b = A[n-1]
    - S = n im schlechtesten Fall

Analyse für erfolgreiche Suche - Frage b

- Im Mittel
  - Wiederholte Anwendung mit verschiedenen Eingaben
  - Annahme über Häufigkeit:
    - Beobachten, wie oft b an erster, zweiter, … letzter Stelle gefunden wird.
  - Annahme Gleichverteilung:
    - Läuft Algorithmus k mal (k >>1), so wird b gleich oft an erster, zweiter, … , letzter Stelle gefunden.
      - Also k/n-mal an jeder Stelle

Analyse für erfolgreiche Suche - Frage b (2)

- Anzahl Schritte insgesamt (**für k** Suchvorgänge):

$$M = \frac{k}{n} \cdot 1 + \frac{k}{n} \cdot 2 + \ldots + \frac{k}{n} \cdot n$$

$$= \frac{k}{n} \cdot (1 + 2 + \ldots + n)$$

$$= \frac{k}{n} \cdot \frac{n \cdot (n+1)}{2} \qquad = k \cdot \frac{n+1}{2}$$

- Für **eine** Suche: $S = \frac{M}{k}$ Schritte

Also: $S = \frac{n+1}{2}$ im Mittel (bei Gleichverteilung)

Asymptotische Analyse

- Analyse der Komplexität → Angabe einer Funktion

$$f : \mathbb{N} \rightarrow \mathbb{N}$$

  als Maß für den Aufwand
- Bedeutung: f(n) = a
  - bei Problemen der Größe n
  - beträgt der Aufwand a
- **Problemgröße:** Umfang der Eingabe, wie z.B. Anzahl der zu sortierenden oder zu durchsuchenden Elemente
- **Aufwand:**
  - Rechenzeit (Abschätzung als Anzahl der Operationen, wie z.B. Vergleiche)
  - Speicherplatz

Aufwand für Schleifen

- Wie oft wird die Wertzuweisung „x = x + 1“ in folgenden Anweisungen ausgeführt?

```
x = x +1
```

1-mal

```
for (i = 1; i <= n; i++)
        x = x + 1;
```

n-mal

```
for (i = 1; i <= n; i++)
      for (j = 1; j <= n; j++)
            x = x + 1;
```

n^2-mal

Aufwandsfunktion

- Aufwandsfunktion $f : \mathbb{N} \to \mathbb{N}$ meist nicht exakt bestimmbar
- daher
  - Abschätzung des Aufwands im schlechtesten Fall
  - Abschätzung des Aufwands im Mittel
- und „ungefähres Rechnen in Größenordnungen“

Algorithmen und Datenstrukturen

3.1 O-NOTATION

O-Notation

- Angabe der asymptotischen oberen Schranke für Aufwandsfunktion → Wachstumsgeschwindigkeit bzw. Größenordnung

- Asymptote: Gerade, der sich eine Kurve bei immer größer werdender Entfernung vom Koordinatenursprung unbegrenzt nähert

- Einfache Vergleichsfunktion $f(n) \in O(g(n))$ für Aufwandsfunktion mit $g : \mathbb{N} \to \mathbb{N}$

O-Notation

Definition:

Für eine Funktion $f : \mathbb{N} \to \mathbb{N}$ ist die Menge $O(f(n))$ wie folgt definiert:

$$O(f(n)) = \{g : \mathbb{N} \to \mathbb{N} \mid \exists c \in \mathbb{R}^{>0}, \exists n_o \in \mathbb{N}\ \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

- Anschaulich: O(f(n)) ist die Menge aller durch f „nach oben“ beschränkten Funktionen
  - ➔ Asymptotische obere Schranke

O-Notation (2)

- Definition veranschaulicht:

$$g(n) \in O(f(n)) \Leftrightarrow \exists c > 0, \exists n_0 \forall n \geq n_0 : g(n) \leq c \cdot f(n)$$

„g wächst nicht schneller als f“

- Dies bedeutet: $\frac{g(n)}{f(n)}$ ist für genügend große n durch eine Konstante c nach oben beschränkt

Ω - Notation

Definition:

Für eine Funktion $f : \mathbb{N} \to \mathbb{N}$ ist die Menge $\Omega(f(n))$ wie folgt definiert:

$$\Omega(f(n)) = \{g : \mathbb{N} \to \mathbb{N} \mid \exists c \in \mathbb{R}^{>0}, \exists n_o \in N \ \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$$

- Anschaulich: Ω(f(n)) ist die Menge aller von f „nach unten“ beschränkten Funktionen
  - ➔ Asymptotische untere Schranke

Zum Vergleich:

$$O(f(n)) = \{g : \mathbb{N} \to \mathbb{N} \mid \exists c \in \mathbb{R}^{>0}, \exists n_o \in \mathbb{N} \ \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

Θ - Notation

Definition:

Die exakte Ordnung Θ von $f(n)$

ist definiert als:

$$\Theta(f(n)) = \{g : \mathbb{N} \to \mathbb{N} \mid \exists c_1 \in \mathbb{R}^{>0}, \exists c_2 \in \mathbb{R}^{>0}, \\ \exists n_o \in N \ \forall n \geq n_0 : c_1 \cdot f(n) \geq g(n) \geq c_2 \cdot f(n)\}$$

- Anschaulich: Menge aller durch f „nach unten und oben“ beschränkten Funktionen
  - Asymptotische untere und obere Schranke

Θ-Notation - Charakterisierung

Lemma:

Für jede Funktion $f(n)$ gilt:

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

Beweis

Zu zeigen:

Θ(f(n)) ⊆ O(f(n))∩Ω(f(n)) und Θ(f(n)) ⊇ O(f(n))∩Ω(f(n))

Θ(f(n)) ⊆ O(f(n))∩Ω(f(n)):

Zeige g(n)∈Θ(f(n)) ⇒g(n)∈O(f(n))∩Ω(f(n)).

- g(n)∈Θ(f(n)) ⇒ ∃ c_1,c_2,n_0: ∀ n≥n_0: c_1f(n) ≥ g(n) ≥ c_2f(n)

Θ-Notation - Charakterisierung

- $g(n) \in \Theta(f(n)) \Rightarrow \exists\, c_1, c_2, n_0: \forall\ n \geq n_0: c_1 f(n) \geq g(n) \geq c_2 f(n)$

$\Rightarrow$ $\exists\, c_1, n_0: \forall\ n \geq n_0: c_1 f(n) \geq g(n)$ **und**
$\exists\, c_2, n_0: \forall\ n \geq n_0: g(n) \geq c_2 f(n)$

$\Rightarrow$ $g(n) \in O(f(n))$ **und** $g(n) \in \Omega(f(n))$

$\Rightarrow$ $g(n) \in O(f(n)) \cap \Omega(f(n))$

$\Theta(f(n)) \supseteq O(f(n)) \cap \Omega(f(n))$: Übung □

Beispiele

- Frage: ist $n^2 \in O(n^3)$

  Bzw: ist n^3 eine obere Schranke für n^2?

- Antwort: Ja
- Begründung:

$$n_0 = 1,\ c = 1$$

$$\Rightarrow n^2 \leq n^3$$

$$\Rightarrow 1 \leq n \text{ für } n \geq 1$$

Beispiele

Frage: ist $n^3 \in O(n^2)$

Bzw: ist n^2 eine obere Schranke für n^3?

- Antwort: Nein
- Begründung: durch Widerspruch
  - Annahme:

$$\exists c, n_0 \in \mathbb{N} : n^3 \leq c \cdot n^2, \text{ für alle } n > n_0$$

$$n^3 \leq c \cdot n^2, \text{ für alle } n > n_0$$

$$\Rightarrow n \leq c, \text{ für alle } n > n_0$$

Wähle: $n = c + n_0$

$$\Rightarrow c + n_0 \leq c$$

Widerspruch!

Eigenschaften

Lemma:

Für beliebige Funktionen $f,\ g$ gilt:

$$O(f(n) + g(n)) = O(max(f(n), g(n)))$$

Beweis:
(zeige beide Richtungen)

$$t(n) \in O(f(n) + g(n)) \ \Rightarrow \ t(n) \in O(max(f(n), g(n)))$$

$$t(n) \in O(f(n) + g(n)) \ \Leftarrow \ t(n) \in O(max(f(n), g(n)))$$

Eigenschaften (Beweis)

- Beweis: „⇒“

$$t(n) \in O(f(n) + g(n)) \Rightarrow t(n) \in O(max(f(n), g(n)))$$

Es gilt:

$$\exists c, n_o \in \mathbb{N} : t(n) \leq c \cdot (f(n) + g(n)) \quad \forall n > n_o$$

$$\Rightarrow t(n) \leq 2 \cdot c \cdot max(f(n), g(n)) \quad \forall n > n_o$$

$$\Rightarrow t(n) \in O(max(f(n), g(n)))$$

Eigenschaften (Beweis)

- Beweis: „$\Leftarrow$“

$$t(n) \in O(f(n) + g(n)) \Leftarrow t(n) \in O(max(f(n), g(n)))$$

Es gilt:

$$\exists c, n_o \in \mathbb{N} : \; t(n) \leq c \cdot (max(f(n), g(n))) \quad \forall n > n_o$$

$$\Rightarrow \; t(n) \leq c \cdot (f(n) + g(n)) \quad \forall n > n_o$$

$$\Rightarrow \; t(n) \in O(f(n) + g(n))$$

Beispiel

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

$$O(n^4 + 4 \cdot n^3) = O(n^4)$$

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

Eigenschaften

Lemma:

1. $O(f(n)) \subseteq O(g(n))$ genau dann wenn
$$f(n) \in O(g(n))$$
2. $O(f(n)) = O(g(n))$ genau dann wenn
$$f(n) \in O(g(n)) \wedge g(n) \in O(f(n))$$
3. $O(f(n)) \subset O(g(n))$ genau dann wenn
$$f(n) \in O(g(n)) \wedge g(n) \notin O(f(n))$$

Eigenschaften

1. $O(f(n)) \subseteq O(g(n))$ genau dann wenn

$$f(n) \in O(g(n))$$

Beweis:
(zeige beide Richtungen)

„⇒“

$$O(f(n)) \subseteq O(g(n)) \Rightarrow f(n) \in O(g(n))$$

$$f(n) \in O(f(n)) \subseteq O(g(n)) \Rightarrow f(n) \in O(g(n))$$

Eigenschaften (Beweis zu 1.)

1. $O(f(n)) \subseteq O(g(n))$ genau dann wenn

$$f(n) \in O(g(n))$$

Beweis:

„$\Leftarrow$“

$$O(f(n)) \subseteq O(g(n)) \Leftarrow f(n) \in O(g(n))$$

Definition

$$f(n) \in O(g(n) \Rightarrow \exists c_0, n_o \in \mathbb{N} : f(n) \leq c_0 \cdot g(n) \forall n > n_o$$

und sei:

$$t(n) \in O(f(n)) \Rightarrow \exists c_1, n_1 \in \mathbb{N} : t(n) \leq c_1 \cdot f(n) \ \forall n > n_1$$

t(n) beliebiges Element der Menge O(f(n))

Definition

Eigenschaften (Beweis zu 1.)

$$\Rightarrow t(n) \leq c_1 \cdot f(n) \leq c_1 \cdot c_0 \cdot g(n) \quad \forall n > \max(n_0, n_1)$$

$$t(n) \in O(f(n)) \Rightarrow t(n) \in O(g(n))$$

$$O(f(n)) \subseteq O(g(n))$$

Definition Teilmenge, da t(n) beliebiges Element

Beispiele

$$O(n^2) = \{n^2,\ 2n^2 - 6,\ 3n^2 + 5,\ \frac{1}{2}n^2 + 8, \ldots\}$$

Damit: $(3n^2 + 5) \in O(n^2)$

$$O(3n^2 + 5) \subseteq O(n^2)$$

Wie ist c zu wählen?

$$O(3n^2 + 5) = \{n^2,\ 2n^2 - 6,\ 3n^2 + 5,\ \frac{1}{2}n^2 + 8, \ldots\}$$

Damit: $n^2 \in O(3n^2 + 5)$

$$O(n^2) \subseteq O(3n^2 + 5)$$

Wie ist c zu wählen?

Damit: $O(n^2) = O(3n^2 + 5)$

Weitere Eigenschaften

Lemma:

Falls $f(n) \in O(g(n))$ und $g(n) \in O(h(n))$,

dann ist auch $f(n) \in O(h(n))$.

$$f(n) \leq c_0 \cdot g(n) \; \forall n > n_0 \text{ und}$$

$$g(n) \leq c_1 \cdot h(n) \; \forall n > n_1 \text{ und}$$

$$\Rightarrow f(n) \leq c_0 \cdot g(n) \leq \underbrace{c_0 \cdot c_1}_{\text{Konstante}} \cdot h(n) \; \forall n \geq max(n_0, n_1)$$

Beispiele

$$O(n^2) = O(3n^2) = O(\frac{1}{2}n^2)$$

$$O(n^2) \subseteq O(n^{2,5}) \subseteq O(n^3)$$

$$O(n^2) \subset O(n^{2,5}) \subset O(n^3)$$

Aufwand und asymptotische Komplexität

Lemma:

Es gilt:

1. $lim_{n\to\infty}(f(n)/g(n)) = c, c > 0 \Rightarrow O(f(n)) = O(g(n))$
2. $lim_{n\to\infty}(f(n)/g(n)) = 0 \Rightarrow O(f(n)) \subset O(g(n))$

Aufwand und asymptotische Komplexität (2)

Häufiges Problem: Grenzwert der „Art“: $\frac{\infty}{\infty}$ oder $\frac{0}{0}$

➔ Ansatz: Regel von de l‘Hospital

Satz (Regel von de l‘Hospital) „x → ∞“:

Seien f und g auf dem Intervall $[a, \infty)$ differenzierbar.

Es gelte $lim_{x\to\infty} f(x) = lim_{x\to\infty} g(x) = 0$ (bzw. $= \infty$)

und es existiere $\lim\limits_{x\to\infty} \frac{f'(x)}{g'(x)}$.

Dann existiert auch $\lim\limits_{x\to\infty} \frac{f(x)}{g(x)}$ und es gilt:

$$\lim_{x\to\infty} \frac{f'(x)}{g'(x)} = \lim_{x\to\infty} \frac{f(x)}{g(x)}$$

Beispiele

1. $f(n) = 3n + 5,\ g(n) = n$

$$\lim_{n\to\infty} \frac{3n+5}{n} \Rightarrow \lim_{n\to\infty} \frac{3}{1} = 3 > 0 \Rightarrow O(3n+5) = O(n)$$

2. $f(n) = n^2 + 5,\ g(n) = n^3$

$$lim_{n\to\infty} \frac{n^2+5}{n^3} \Rightarrow lim_{n\to\infty} \frac{2n}{3n^2} \Rightarrow lim_{n\to\infty} \frac{2}{6n} = 0$$

Wiederholte Anwendung von de l'Hospital

$$\Rightarrow O(n^2+5) \subset O(n^3)$$

Weitere Eigenschaften

- Gibt es immer eine Ordnung zwischen Funktionen?

- Es gibt Funktionen f, g:

$$f(n) \notin O(g(n)) \text{ und } g(n) \notin O(f(n))$$

- Zum Beispiel: sin(n) und cos(n)

Asymptotische Komplexität

Lemma:

Für alle $m \in \mathbb{N}$ *gilt* : $O(n^m) \subseteq O(n^{m+1})$

Beweis: (durch Widerspruch)

Annahme: $s(n) \in O(n^k)$,

d.h. $\exists c, n_0, \forall n > n_o : \; s(n) \leq c \cdot n^k$

aber es muss auch gelten: $s(n) \notin O(n^{k+1})$

d.h. $\exists n > n_0 : \; s(n) > c \cdot n^{k+1}$

$\Rightarrow \exists n > n_0 : \;$ und $n < 1$

Aufwand und asymptotische Komplexität

- Aussage 1:

Sei $f(n) = a_m \cdot n^m + a_{m-1} \cdot n^{m-1} + \ldots + a_1 \cdot n + a_0$,

wobei $a_i \in \mathbb{R}^+$ für $0 \leq i \leq m$. Dann gilt $f(n) \in O(n^m)$.

- Wir sagen, ein Algorithmus mit Komplexität f(n) benötigt höchstens polynomielle Rechenzeit, falls es ein Polynom p(n) gibt, mit $f(n) \in O(p(n))$.
- Aussage 2:

Ein Algorithmus benötigt höchstens exponentielle Rechenzeit, falls es eine Konstante $a \in \mathbb{R}^+$ gibt, mit $f(n) \in O(a^n)$.

Komplexitätsklassen

$O(1)$	konstanter Aufwand (=Aufwand ist nicht abhängig von Eingabe)
$O(log\ n)$	logarithmischer Aufwand
$O(n)$	linearer Aufwand
$O(n \cdot log\ n)$	
$O(n^2)$	quadratischer Aufwand
$O(n^k)$ für $k \geq 0$	polynomialer Aufwand
$O(2^n)$	exponentieller Aufwand

Typische Problemklassen

Aufwand	Problemklasse
$O(1)$	einige Suchverfahren für Tabellen (Hashing)
$O(\log n)$	allgemeine Suchverfahren für Tabellen (Baum-Suchverfahren)
$O(n)$	sequenzielle Suche, Suche in Texten, syntaktische Analyse von Programmen (bei „guter“ Grammatik)
$O(n \cdot \log n)$	Sortieren

Aufwand	Problemklasse
$O(n^2)$	einige dynamische Optimierungsverfahren (z.B. optimale Suchbäume), Multiplikation Matrix-Vektor (einfach)
$O(n^3)$	Matrizen-Multiplikation (einfach)
$O(2^n)$	viele Optimierungsprobleme (z.B. optimale Schaltwerke), automatisches Beweisen (im Prädikatenkalkül 1. Stufe)

Algorithmen und Datenstrukturen

3.2 AUFWANDSANALYSE VON ITERATIVEN ALGORITHMEN

Aufwand iterativer Algorithmen

- Aufwand ≈ Anzahl durchlaufener Operationen (Zuweisungen, Vergleiche, ...) zur Lösung eines Problems
- Häufig: Aufwand ist abhängig von Eingabeparametern (=> Problemgröße)
- Aufwandsklasse: Wie wächst der Aufwand in Abhängigkeit von der Problemgröße?
- **Frage: Gegeben ein beliebiger Java-Code, wie kann ich die Aufwandsklasse bestimmen?**

Aufwand von Programmen ablesen

```
void alg1(int n){
     int m = 2;
     int i;
     int k = n;
     while (n > 0){
          i = k;
          while (i > 0) {
                m = m + i;
                i = i / 2;
          }
          n = n - 1;
     }
}
```

$\Theta(n \cdot \log n)$

Äußere Schleife: n-mal
Innere Schleife: log n-mal

Aufwand von Programmen ablesen

```
void alg1(int n) {
      int m = 1;
      int i = 0;
      while (m < n) {
        while (i < m)
          i = i + 1;
        m = m + i;
      }
}
```

Θ(n + log n)

In jedem Durchlauf der äußeren Schleife wird m verdoppelt, d.h. sie läuft log n Mal, die innere Schleife läuft bis n/2, aber nicht jedes Mal, weil i nur ein Mal auf 0 gesetzt wird.

Man könnte auch O(n) sagen, weil der Summand log n nicht ins Gewicht fällt.

Aufwand iterativer Algorithmen (I)

- Bestandteile iterativer Algorithmen:
  - elementare Anweisungen (Zuweisungen, Vergleiche, ...)
    - $\Theta(1)$
  - Sequenz: $\alpha_1 ; \alpha_2$
    - Obere Grenze: $O(f_{\alpha_1}(n)) + O(f_{\alpha_2}(n))$
    - Untere Grenze: $\Omega(f_{\alpha_1}(n)) + \Omega(f_{\alpha_2}(n))$
  - Selektion: $\text{if } (B) \{ \alpha_1 \} \text{ else } \{ \alpha_2 \}$
    - Obere Grenze: $O(f_B(n)) + O(max(f_{\alpha_1}(n), f_{\alpha_2}(n)))$
    - Untere Grenze: $\Omega(f_B(n)) + \Omega(min(f_{\alpha_1}(n), f_{\alpha_2}(n)))$

$f_{\alpha_1}(n)$: Aufwand, der bei der Ausführung von α_1 entsteht.

Aufwand iterativer Algorithmen (II)

- Iteration: $\text{while } (B)\,\{\,\alpha\,\}$
  - Obere Grenze: $\text{"\# Schleifendurchläufe"} \cdot (O(f_B(n)) + O(f_\alpha(n)))$
  - Untere Grenze: $\text{"\# Schleifendurchläufe"} \cdot (\Omega(f_B(n)) + \Omega(f_\alpha(n)))$

- **Frage: Wie ist der Aufwand für for-Schleifen?**
  - Beispiel: $\text{for } (\alpha_1; B; \alpha_2)\,\{\,\alpha_3\,\}$
  - Antwort: Abbildung auf while-Schleife:

$$
\begin{array}{l}
\alpha_1; \\
\text{while } (B)\,\{ \\
\quad \alpha_3; \\
\quad \alpha_2; \\
\}
\end{array}
$$

Beispiel: Sequenz

```
public int berechne(int n) {
  int x = 0;                    ⟸ Θ(1)
  x = x + 1;                    ⟸ Θ(1)
  return x;                     ⟸ Θ(1)
}
```

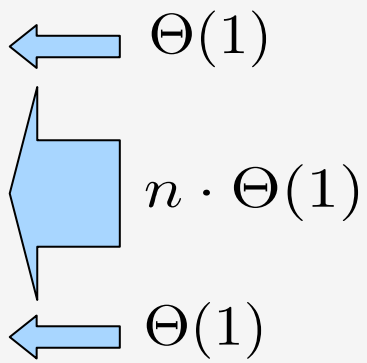
Fragen:

- **Wieviele Operationen werden durchlaufen?**
- **Ist die Anzahl abhängig von einem Eingabeparameter?**

- Aufwand: $f(n) = \Theta(1) + \Theta(1) + \Theta(1) = 3 \cdot \Theta(1)$
- Aufwandsklasse: $\Theta(f(n)) = \Theta(1)$

Beispiel: Schleifen (I)

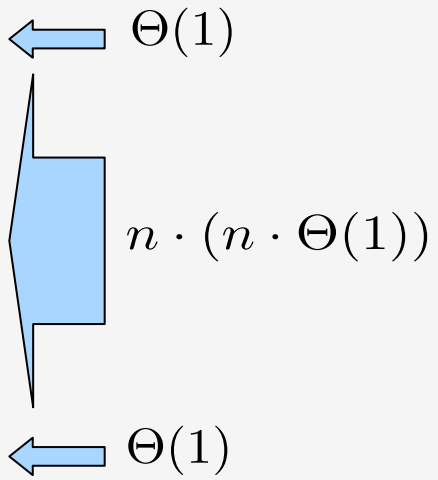
```
public int berechne(int n) {
  int x = 0;
  for (int i=0; i < n; i++) {
    x = x + 1;
  }
  return x;
}
```



- Aufwand: $f(n) = \Theta(1) + n \cdot \Theta(1) + \Theta(1) = 2 \cdot \Theta(1) + \Theta(n)$
- Aufwandsklasse: $\Theta(f(n)) = \Theta(n)$

Beispiel: Schleifen (II)

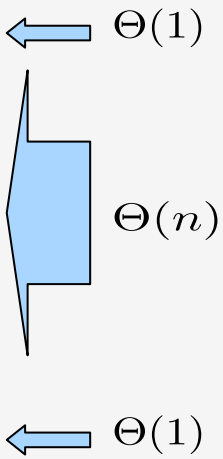
```
public int berechne(int n) {
  int x = 0;
  for (int i=0; i < n; i++) {
    for (int j=0; j < n; j++) {
      x = x + 1;
    }
  }
  return x;
}
```



- Aufwand: $f(n) = \Theta(1) + n^2 \cdot \Theta(1) + \Theta(1) = 2 \cdot \Theta(1) + \Theta(n^2)$
- Aufwandsklasse: $\Theta(f(n)) = \Theta(n^2)$

Beispiel: Selektion

```
public int berechne(int n) {
if (n % 2 == 0) {
     int x = 0;
     for (int i=0; i < n; i++) {
            x = x + 1;
     }
     return x;
  }else{
     return n;
  }
}
```



- Obere Grenze: $O(f(n)) = \Theta(1) + O(max(\Theta(n), \Theta(1)) = O(n)$
- Untere Grenze: $\Omega(f(n)) = \Theta(1) + \Omega(min(\Theta(n), \Theta(1)) = \Omega(1)$

Faustregeln

- Häufig verwendete Faustregeln:
  - Keine Schleife: Konstanter Aufwand
  - Eine Schleife: Linearer Aufwand
  - Zwei geschachtelte Schleifen: Quadratischer Aufwand

- Gelten die Faustregeln ohne Ausnahme? **Nein!**
  - Aufwandsbestimmung für Schleifen
  - Mehrere Eingabevariablen
  - Funktionsaufrufe
  - Rekursionen

Aufwandsbestimmung für Schleifen (I)

```
public int berechne(int n) {
   int x = 0;
   for (int i=0; i < 5; i++) {
      x = x + 1;
   }
   return x;
}
```

- Schleifenabbruch hängt nicht von Eingabeparameter ab!
- Aufwand: $f(n) = \Theta(1) + 5 \cdot \Theta(1) + \Theta(1) = 7 \cdot \Theta(1)$
- Aufwandsklasse: $\Theta(f(n)) = \Theta(1)$

Aufwandsbestimmung für Schleifen (II)

```
public int berechne(int n) {
   int x = 0;
   for (int i=1; i < n; i = 2*i) {
      x = x + 1;
   }
   return x;
}
```

- Laufvariable wächst nicht linear an!

- Aufwand: $f(n) = \Theta(1) + \log_2 n \cdot \Theta(1) + \Theta(1)$
- Aufwandsklasse: $\Theta(f(n)) = \Theta(\log n)$

Aufwandsbestimmung für Schleifen (III)

```
for (int i=1; i < n; i=2*i) {
   x = x + 1;
}
```

- **Frage 1: Wie entwickelt sich die Laufvariable?**
  - Startwert: i=1
  - Veränderung: i=2*i

- Entwicklung der Laufvariablen:
  - Nach 1. Durchlauf: $i = 1 \cdot 2 = 2^1$
  - Nach 2. Durchlauf: $i = (1 \cdot 2) \cdot 2 = 2^2$
  - Nach 3. Durchlauf: $i = ((1 \cdot 2) \cdot 2) \cdot 2 = 2^3$
  - Nach k-tem Durchlauf: $i = 2^k$

Aufwandsbestimmung für Schleifen (IV)

```
for (int i=1; i < n; i=2*i) {
   x = x + 1;
}
```

- Entwicklung der Laufvariablen: $i = 2^k$
- **Frage 2: Nach wie vielen Durchläufen wird abgebrochen?**
  - Abbruch, wenn i ≥ n

$$\begin{aligned} & i & \geq n & \quad | i = 2^k \\ \Leftrightarrow & 2^k & \geq n & \quad | \log_2 \\ \Leftrightarrow & k & \geq \log_2 n & \end{aligned}$$

  - Antwort: Abbruch nach $k = \lceil \log_2 n \rceil$ Durchläufen

Mehrere Eingabevariablen

```
public int berechne(int[] f1, int[] f2) {
  int result = 0;
  for (int i=0; i < f1.length; i++) {
    for (int j=0; j < f2.length; j++) {
      if (f1[i] == f2[j]) result++;
    }
  }
  return result;
}
```

- Problemgrößen: $n = f1.length$ und $m = f2.length$
- Aufwand: $f(n,m) = \Theta(1) + n \cdot (m \cdot (\Theta(1) + O(1))) + \Theta(1)$
- Aufwandsklasse: $\Theta(f(n,m)) = \Theta(n \cdot m)$

Algorithmen und Datenstrukturen

3.3 AUFWANDSANALYSE VON REKURSIVEN ALGORITHMEN

Rekursionen

```
public int fib(int n) {
if (n == 0 || n == 1) {
    return 1;
  } else {
    return fib(n-1) + fib(n-2);
  }
}
```

Rek.abbruch

Rek.fall

+

Rek.fall

- **Frage: Welcher Aufwand entsteht für Fibonacci?**
  - Für Rekursionsabbruch: $f(n) = \Theta(1) + \Theta(1)$
  - Für Rekursionsfall: $f(n) = \Theta(1) + ?????$

➔ Lösung: Rekursionsgleichungen!

Rekursionsgleichungen

- Definition: Eine Rekursionsgleichung ist eine Gleichung oder Ungleichung, die eine Funktion anhand ihrer Anwendung auf kleinere Werte beschreibt.
- Rekursionsgleichung für Fibonacci:

Aufwand für Rekursionsabbruch

$$T(n) = \begin{cases} \Theta(1) & \text{für } (n = 0 \vee n = 1) \\ \Theta(1) + T(n-1) + T(n-2) & \text{sonst} \end{cases}$$

Aufwand für Rekursionsfall

Lösung von Rekursionsgleichungen 2/2

Gegeben

$$T(n) = \begin{cases} \Theta(1) & \text{für } (n = 0 \vee n = 1) \\ \Theta(1) + T(n-1) + T(n-2) & \text{sonst} \end{cases}$$

Welche Aufwandsklasse hat T(n)?

- O(1)?
- O(log n)?
- O(n)?
- $O(n^2)$?
- ...

Lösung von Rekursionsgleichungen 2/2

- **Frage: Gegeben eine Rekursionsgleichung, welche Aufwandsklasse hat der dazugehörige Algorithmus?**

- Lösungsmethoden:
  - Vollständige Induktion
  - Master-Theorem

Algorithmen und Datenstrukturen

3.3.1 VOLLSTÄNDIGE INDUKTION

Vollständige Induktion

- Induktion:
  - Rekursive Beweistechnik aus der Mathematik
  - ➔ Gut geeignet, um Eigenschaften von rekursiv definierten Funktionen zu beweisen!

- Generelles Vorgehen:
  - Vermutung einer Eigenschaft (z.B. Aufwandsklasse einer Rekursionsgleichung)
  - Induktionsanfang: Eigenschaft hält für ein kleines n
  - Induktionsschritt:
    - Annahme: Wir haben es bereits für ein kleineres n gezeigt
    - Wenn Eigenschaft für kleinere n hält, dann hält sie auch für das nächstgrößere n!

Beispiel – Obere Grenze beweisen (I)

$$T(n) = \begin{cases} \Theta(1) & \text{für} n \leq 1 \\ 4 \cdot T(\frac{n}{2}) + \Theta(n^3) & \text{sonst} \end{cases}$$

$$T(n) \in O(n^3)$$

Siehe Definition der O-Notation

$$\exists n_0, c : \forall n \geq n_0 : T(n) \leq c \cdot n^3$$

$$n = 2^k$$

Brauchen keine Spezial-fälle behandeln, und im Induktionsschritt können wir von $\frac{n}{2}$ nach n gehen.

Beispiel – Obere Grenze beweisen (II)

- Induktionsvermutung: $T(\frac{n}{2}) \leq c \cdot \left(\frac{n}{2}\right)^3$
- Induktionsschritt: Wir beweisen von $\frac{n}{2}$ nach n

Rekursionsgleichung einsetzen

Zu zeigende obere Grenze

$$
\begin{aligned}
& T(n) && \leq c \cdot n^3 && \mid T(n) = 4 \cdot T(\tfrac{n}{2}) + n^3 \\
\Leftrightarrow\ & 4 \cdot T(\tfrac{n}{2}) + n^3 && \leq c \cdot n^3 && \mid T(\tfrac{n}{2}) \leq c \cdot \left(\tfrac{n}{2}\right)^3 \\
\Leftarrow\ & 4 \cdot c \cdot \left(\tfrac{n}{2}\right)^3 + n^3 && \leq c \cdot n^3 && \\
\Leftrightarrow\ & 4 \cdot c \cdot \tfrac{n^3}{8} + n^3 && \leq c \cdot n^3 && \mid -c \cdot n^3 \\
\Leftrightarrow\ & -\tfrac{1}{2} \cdot c \cdot n^3 + n^3 && \leq 0 && \mid : n^3 \\
\Leftrightarrow\ & -\tfrac{1}{2} \cdot c + 1 && \leq 0 && \mid + \tfrac{1}{2} \cdot c \\
\Leftrightarrow\ & 1 && \leq \tfrac{1}{2} \cdot c && \mid \cdot 2 \\
\Leftrightarrow\ & 2 && \leq c &&
\end{aligned}
$$

Induktionsvermutung einsetzen

Induktionsschritt ist erfolgreich, wenn $c \geq 2$

Beispiel – Obere Grenze beweisen (III)

- Induktionsanfang:
  - Zeige Induktionsvermutung für einen Anfangswert.
  - Am einfachsten: Zeige es für den Rekursionsa[illegible]

Zu zeigende obere Grenze

Rekursionsgleichung einsetzen

$$T(1) \leq c \cdot 1^3 \quad | \; T(1) = 1$$

$$\Leftrightarrow \quad 1 \leq c$$

Induktionsanfang ist erfolgreich, wenn $c \geq 1$

- Wann können wir zeigen, dass $T(n) \leq c \cdot n^3$?
  - Für $n_0 = 1$

Wert, für den wir Induktionsanfang gezeigt haben

  - Und wenn $(c \geq 2 \wedge c \geq 1) \Rightarrow c \geq 2$

Nötig laut Induktionsschritt

Nötig laut Induktionsanfang

Algorithmen und Datenstrukturen

3.3.2 MASTER-THEOREM

Mastermethode

- Die Mastermethode ist ein „Kochrezept“ zur Lösung von Rekursionsgleichungen der Form:

$$T(n) = a\,T(n/b) + f(n)$$

mit Konstanten $a \geq 1$ und $b > 1$,
$f(n)$ ist eine asymptotisch positive Funktion, d.h. $f(n) > 0 \quad \forall\ n > n_0$

Bemerkung

- a steht für die Anzahl der Unterprobleme
- n/b ist die Größe eines Unterproblems
- T(n/b) ist der Aufwand zum Lösen eines Unterproblems (der Größe n/b)
- f(n) ist der Aufwand für das Zerlegen und Kombinieren in bzw. von Unterproblemen

Mastermethode

- Schnelles Lösungsverfahren zur Bestimmung der Laufzeitklasse einer gegebenen rekursiv definierten Funktion
- 3 gängige Fälle
  - Fall 1: Obere Abschätzung
  - Fall 2: Exakte Abschätzung
  - Fall 3: Untere Abschätzung

- Lässt sich keiner dieser 3 Fälle anwenden, so muss die Komplexität anderweitig bestimmt werden
  - Wir müssen Voraussetzungen für Anwendung des Mastertheorems überprüfen.

Mastermethode - Grundlage

T(n) = a T(n/b) + f(n)

mit Konstanten a ≥ 1 und b > 1,
f(n) ist eine asymptotisch positive Funktion, d.h. f(n) > 0 $\forall$ n > n_0

Allgemein:
- Vergleichen $f(n)$ mit $n^{log_b a}$

- Wir verstehen n/b als $\lfloor n/b \rfloor$ oder $\lceil n/b \rceil$

Hinweis: verkürzte Notation im Folgenden

$log_2 n$ als $ld\ n$ (logarithmus dualis)

Fall 1

Wenn $f(n) \in O(n^{log_b a - \epsilon})$ für ein $\epsilon > 0$

- $f(n)$ wächst polynomiell langsamer als $n^{log_b a}$ um einen Faktor n^{ϵ}
- **Lösung:** $T(n) \in \Theta(n^{\log_b a})$

Fall 2

Wenn $f(n) \in \Theta(n^{log_b a} \cdot ld^k n)$ für ein $k \geq 0$

- $f(n)$ und $n^{log_b a} \cdot ld^k n$ *wachsen vergleichbar gleich schnell*
- **Lösung:** $T(n) \in \Theta(n^{\log_b a} \cdot ld^{k+1} n)$

Fall 3

Wenn $f(n) \in \Omega(n^{log_b a+\epsilon})$ für ein $\epsilon > 0$

und $\underbrace{a \cdot f(n/b) \leq c \cdot f(n)}$ für eine Konstante $c \in (0,1)$ und genügend große n

Regularitätsbedingung

- *f* (*n*) wächst polynomiell schneller als $n^{log_b a}$ um einen Faktor n^{ϵ}
- f(n) erfüllt sog. Regularitätsbedingung

- **Lösung:** $T(n) \in \Theta(f(n))$

Master-Theorem (Überblick)

Ist T(n) eine rekursiv definierte Funktion der Form

$T(n) = a\,T(n/b) + f(n)$ mit $a \geq 1$, $b > 1$, $\forall\ n > n_0 : f(n) > 0$

Dann gilt:

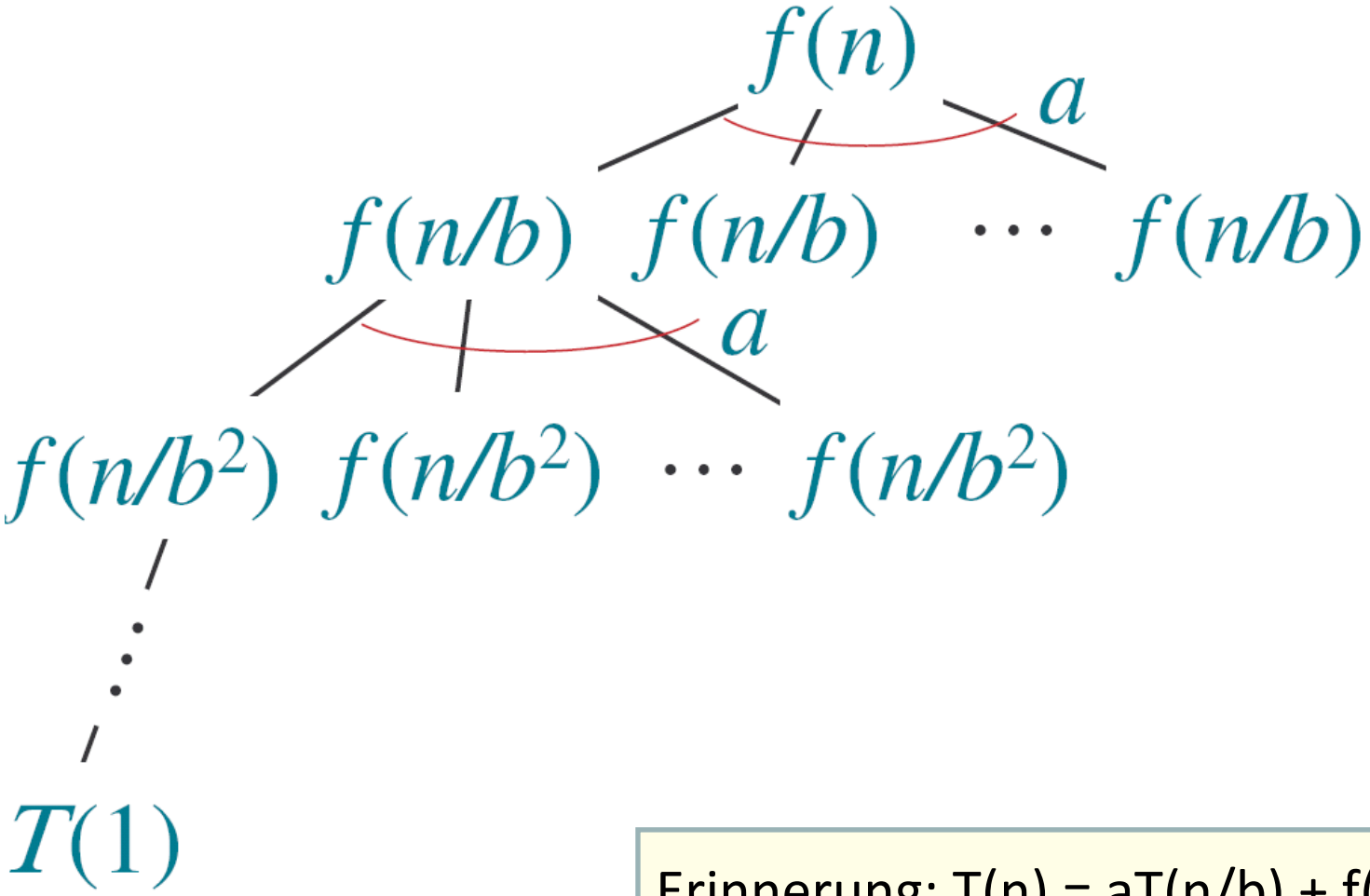
1. Fall: Wenn $f(n) \in O(n^{log_b a - \epsilon})$ für ein $\epsilon > 0$
   dann $T(n) \in \Theta(n^{\log_b a})$

2. Fall: Wenn $f(n) \in \Theta(n^{log_b a} \cdot ld^k n)$ für ein $k \geq 0$
   dann $T(n) \in \Theta(n^{\log_b a} \cdot ld^{k+1} n)$

3. Fall: Wenn $f(n) \in \Omega(n^{log_b a + \epsilon})$ für ein $\epsilon > 0$
   und $a \cdot f(n/b) \leq c \cdot f(n)$ für eine Konstante $c \in (0, 1)$ und genügend große n
   dann $T(n) \in \Theta(f(n))$

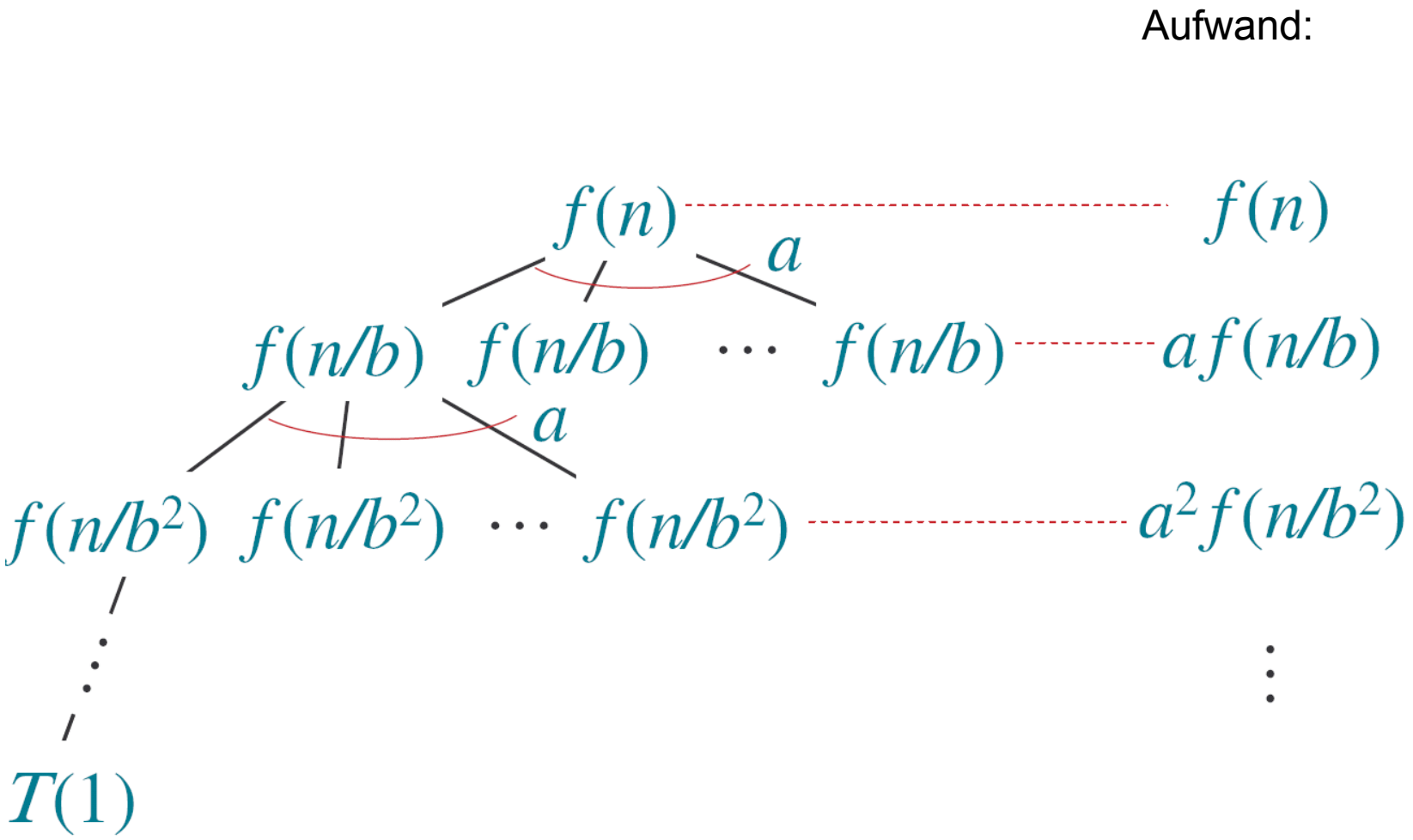
Mastertheorem – Idee

Illustration als „Rekursionsbaum“:

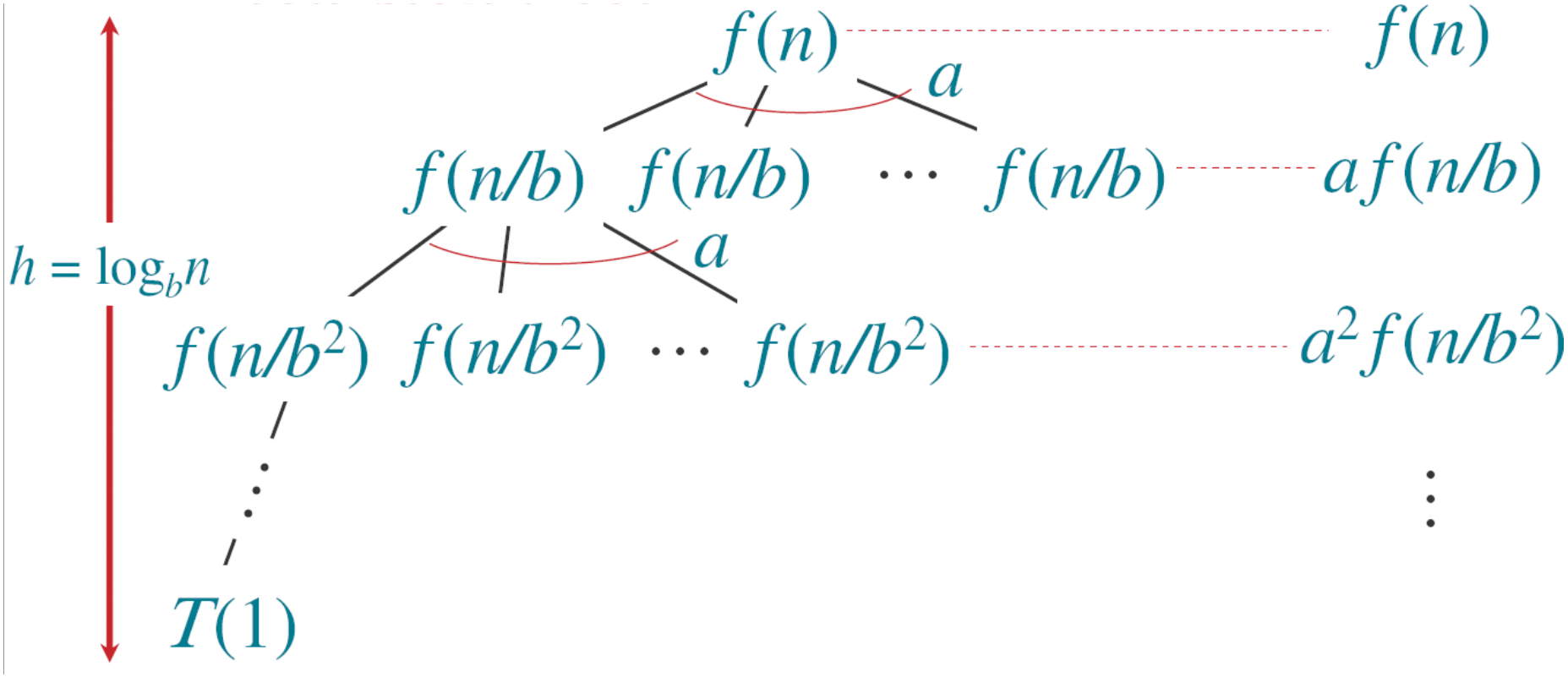


Erinnerung: T(n) = aT(n/b) + f(n)

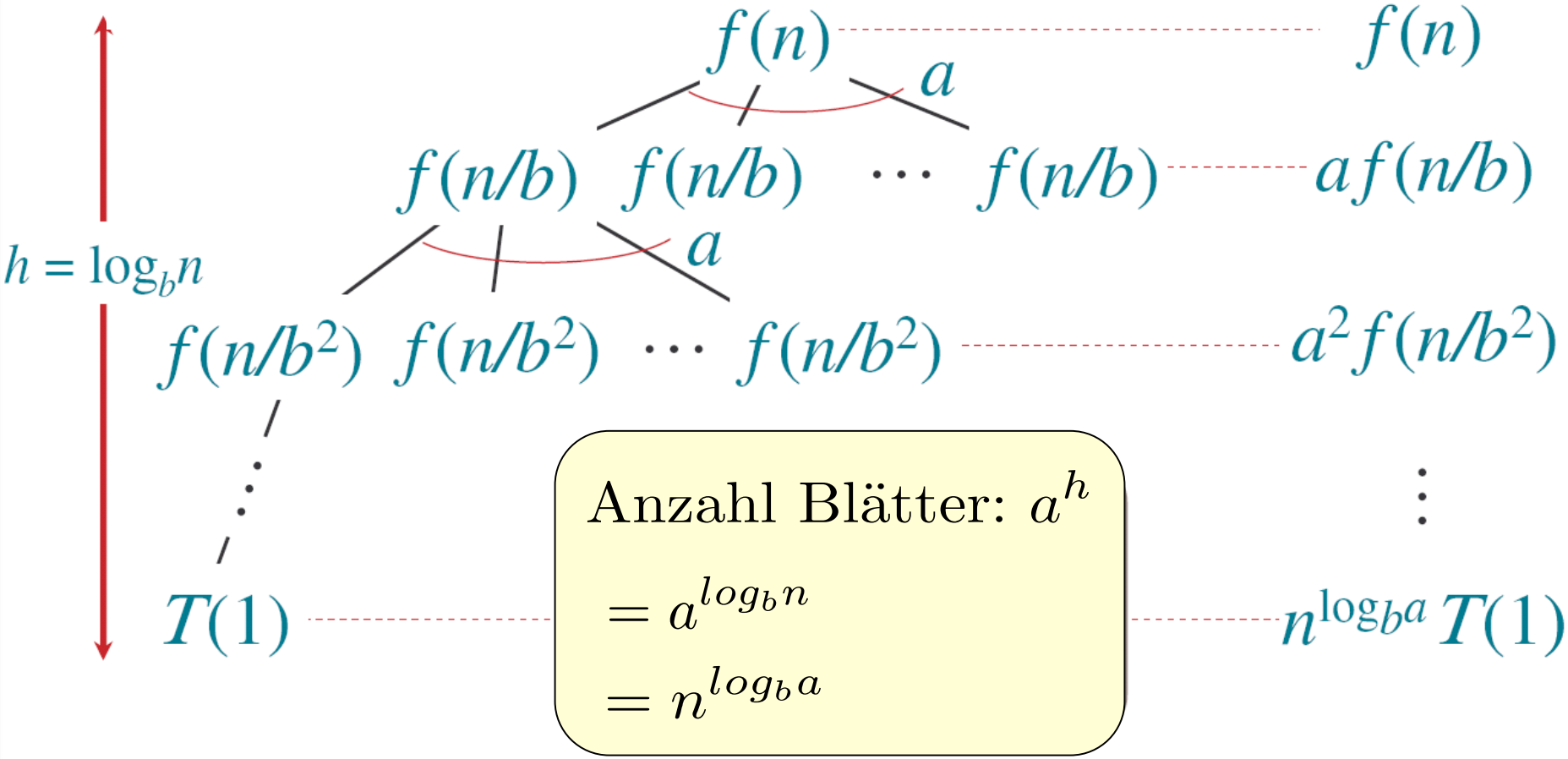
Mastertheorem – Idee



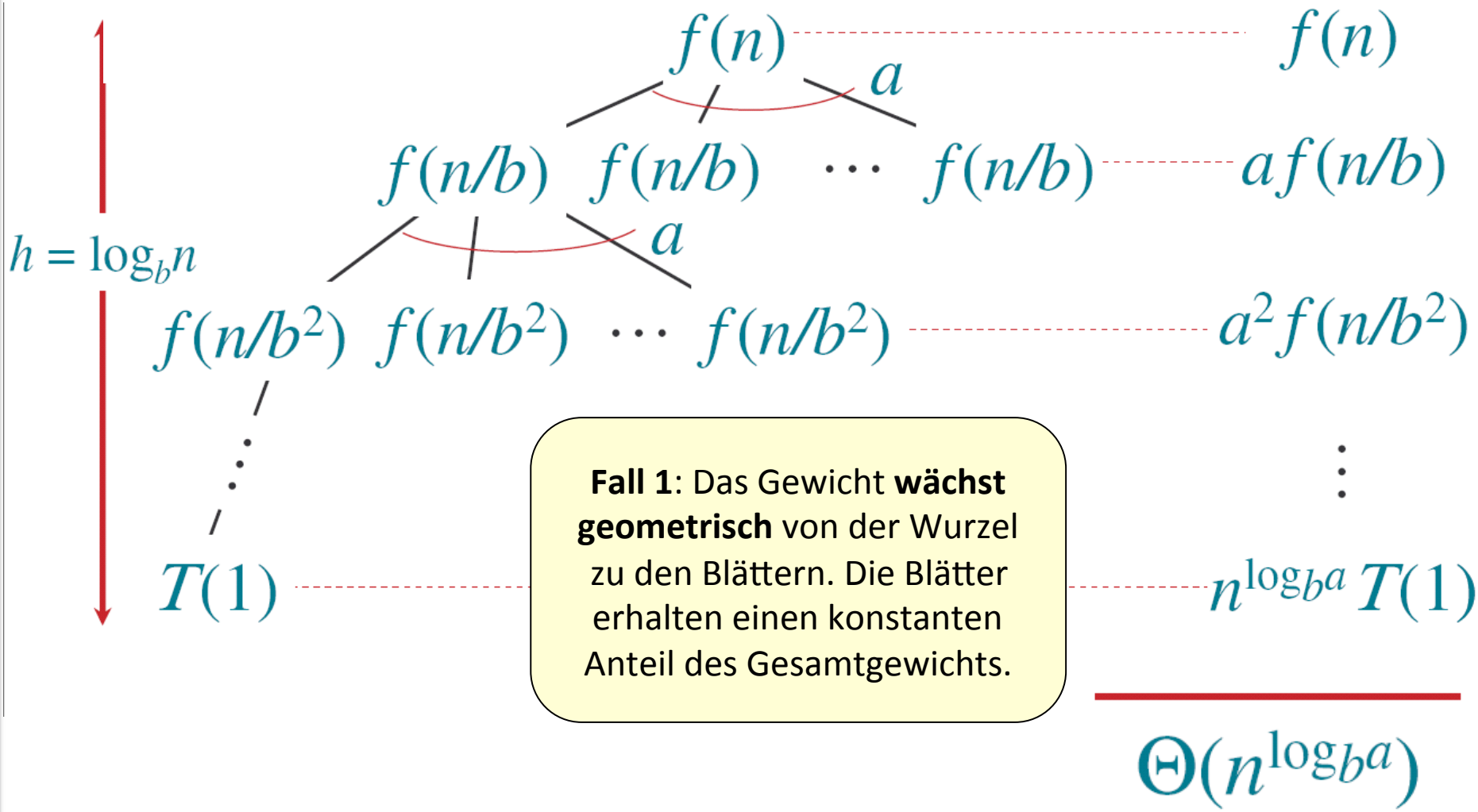
Mastertheorem – Idee



Mastertheorem – Idee

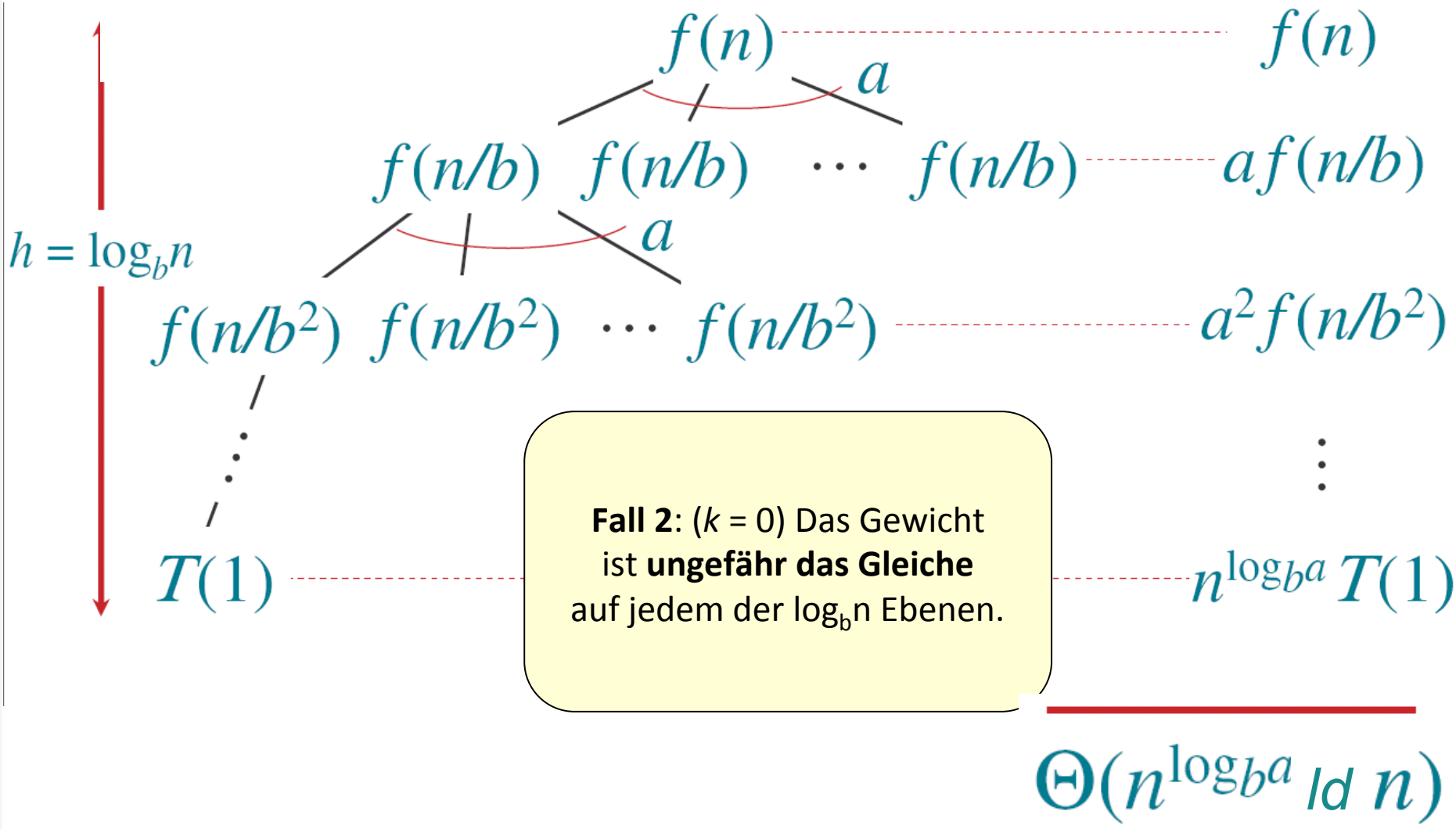


Mastertheorem – Idee



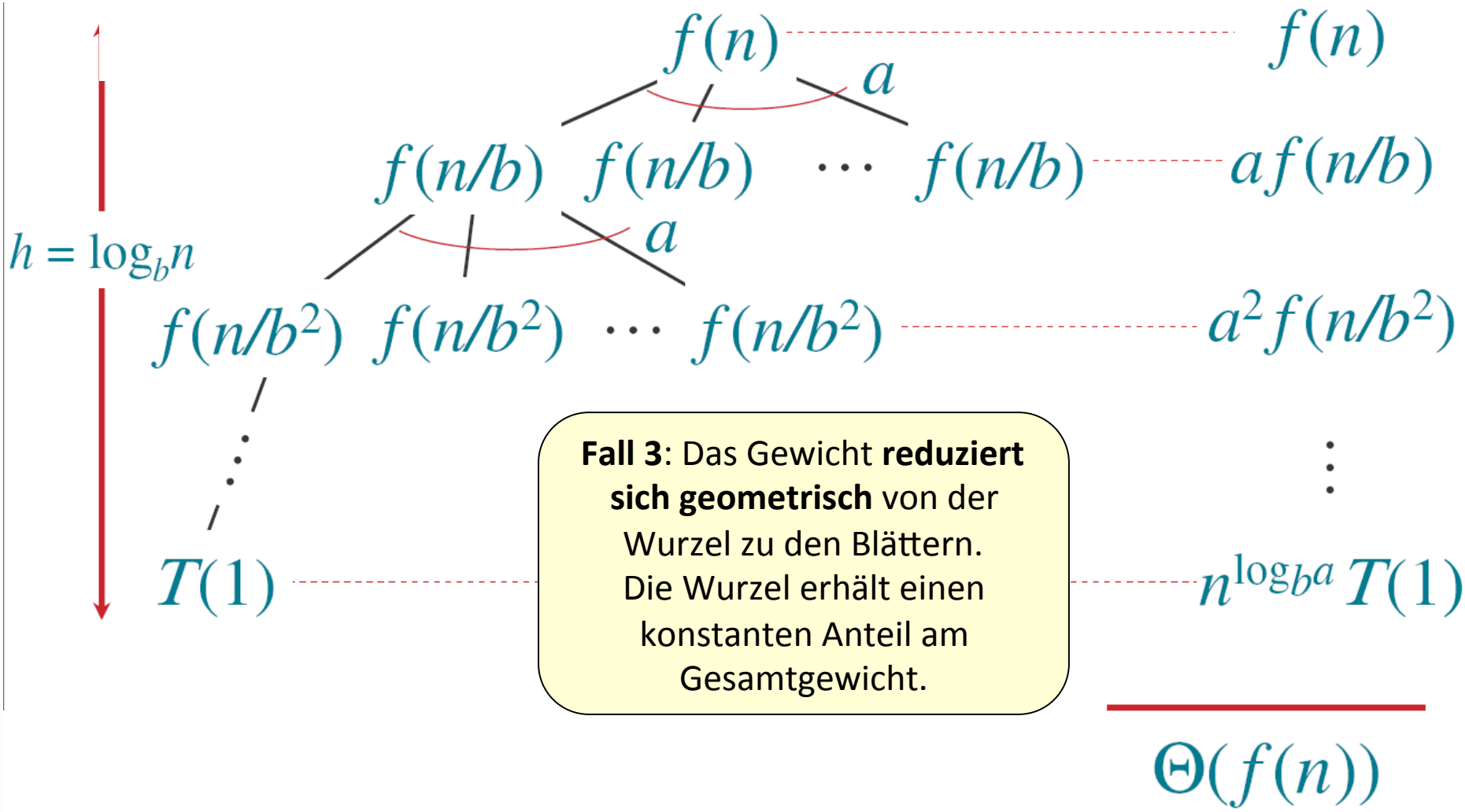
Fall 1: Das Gewicht **wächst geometrisch** von der Wurzel zu den Blättern. Die Blätter erhalten einen konstanten Anteil des Gesamtgewichts.

Mastertheorem – Idee



Fall 2: (k = 0) Das Gewicht ist **ungefähr das Gleiche** auf jedem der $\log_b$n Ebenen.

Mastertheorem – Idee



Fall 3: Das Gewicht **reduziert sich geometrisch** von der Wurzel zu den Blättern. Die Wurzel erhält einen konstanten Anteil am Gesamtgewicht.

Mastertheorem – Bedeutung

- In jedem Fall vergleichen wir $f(n)$ mit $n^{log_b a}$
- Intuitiv: die Lösung wird durch die größere Funktion bestimmt
  - Im zweiten Fall wachsen sie ungefähr gleich schnell
- Im ersten und dritten Fall muss $f(n)$ nicht nur kleiner oder größer als $n^{log_b a}$ sein, sondern poynomiell kleiner oder größer um einem Faktor n^{ϵ}
- Der dritte Fall kann nur angewandt werden, wenn die Regularitätsbedingung erfüllt ist

Ergänzung zur Regularitätsbedingung

- Wozu wird im dritten Fall die Regularitätsbedingung benötigt?
  - Erinnerung: im dritten Fall dominiert f(n) das Wachstum von T(n)

➔ Wir müssen sicherstellen, dass auch bei rekursivem Anwenden (d.h. wenn Argumente kleiner werden) T(n) von f(n) dominiert wird.

Ergänzung zur Regularitätsbedingung (2)

… veranschaulicht:

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

$$= a(aT(n/b^2) + f(n/b)) + f(n)$$

Wachstum dieses Teils muss durch f(n) dominiert werden.

$$= a^2T(n/b^2) + af(n/b) + f(n)$$

bereits berücksichtigt

Problem: darf f(n) nicht dominieren

→ für: $af(n/b) \leq cf(n)$ $(c \in (0,1))$

Beispiel

$$T(n) = 4T(n/2) + n$$

$$a = 4, \quad b = 2 \Rightarrow n^{log_b a} = n^{log_2 4} = n^2$$

$$f(n) = n$$

Fall 1: $f(n) \in O(n^{2-\epsilon})$ für $\epsilon > 0$ z.B. $\epsilon = 1$

$$\Rightarrow T(n) \in \Theta(n^2)$$

Beispiel (2)

$$T(n) = 4T(n/2) + n^2$$

$$a = 4, \quad b = 2 \Rightarrow n^{log_b a} = n^{log_2 4} = n^2$$

$$f(n) = n^2$$

Fall 2: $f(n) \in \Theta(n^2 ld^k\ n)$ $\qquad k = 0$

$$\Rightarrow T(n) \in \Theta(n^2 ld\ n)$$

Beispiel (3)

$$T(n) = 4T(n/2) + n^3$$

$$a = 4, \quad b = 2 \Rightarrow n^{log_b a} = n^{log_2 4} = n^2$$

$$f(n) = n^3$$

Fall 3: $f(n) \in \Omega(n^{2+\epsilon})$ für $\epsilon > 0$ z.B. $\epsilon = 1$

$$\text{und } 4(\frac{n}{2})^3 \leq cn^3 \quad \text{(Regularitätsbedingung)}$$

$$\text{für } c = \frac{1}{2}$$

$$\Rightarrow T(n) \in \Theta(n^3)$$

Beispiel (4)

$$T(n) = 4T(n/2) + \frac{n^2}{log\ n}$$

$$a = 4, \quad b = 2 \Rightarrow n^{log_b a} = n^{log_2 4} = n^2$$

$$f(n) = \frac{n^2}{log\ n}$$

Fall?

Mastertheorem kann nicht angewandt werden:

- 1. Fall: $f(n) \notin O(n^{2-\epsilon})$
- 2. Fall: $f(n) \notin \Theta(n^2 \cdot ld^k n) \quad (k \geq 0)$
- 3. Fall: $f(n) \notin \Omega(n^{2+\epsilon})$

Nützliche Hinweise

- Basisumrechnung $log_b x = \frac{log_a x}{log_a b} \Rightarrow O(log_b x) = O(log_a x)$
- de l'Hospital: $lim_{x \to \infty} \frac{f(x)}{g(x)} = lim_{x \to \infty} \frac{f'(x)}{g'(x)}$
- Vergleich Logarithmus vs. Polynom

$$lim_{x \to \infty} log_b x = \infty \quad lim_{x \to \infty} x^\epsilon = \infty \text{ für } \epsilon > 0$$

$$\lim_{x \to \infty} \frac{log_b x}{x^\epsilon} = \lim_{x \to \infty} \frac{(log_b x)'}{(x^\epsilon)'} = \lim_{x \to \infty} \frac{\frac{1}{x}}{\epsilon x^{\epsilon - 1}}$$

$$= \lim_{x \to \infty} \frac{1}{x \epsilon x^{\epsilon - 1}} = \lim_{x \to \infty} \frac{1}{\epsilon x^\epsilon} = 0 \text{ für } \epsilon > 0$$

Algorithmen und Datenstrukturen

3. LAUFZEITANALYSEN

ZUSAMMENFASSUNG

Zusammenfassung

- O-Notation

$$O(f(n)) = \{g : \mathbb{N} \to \mathbb{N} \mid \exists c \in \mathbb{R}^{>0}, \exists n_o \in \mathbb{N}\ \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

$$\Omega(f(n)) = \{g : \mathbb{N} \to \mathbb{N} \mid \exists c \in \mathbb{R}^{>0}, \exists n_o \in N\ \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$$

$$\Theta(f(n)) = \{g : \mathbb{N} \to \mathbb{N} \mid \exists c_1 \in \mathbb{R}^{>0}, \exists c_2 \in \mathbb{R}^{>0}, \exists n_o \in N\ \forall n \geq n_0 : c_1 \cdot f(n) \geq g(n) \geq c_2 \cdot f(n)\}$$

- Aufwandsanalyse von iterativen Algorithmen
  - Schleifenschachtelung maßgebend für Laufzeit
- Aufwandsanalyse von rekursiven Algorithmen
  - Rekursionsgleichungen
  - Vollständige Induktion
  - Master-Theorem