Datenstrukturen und Algorithmen

Vorlesung 3: Rekursionsgleichungen (K4)

Prof. Dr. Erika Ábrahám

Theorie Hybrider Systeme
Informatik 2

http://ths.rwth-aachen.de/teaching/ss-14/datenstrukturen-und-algorithmen/

Diese Präsentation verwendet in Teilen Folien von Joost-Pieter Katoen.

24. April 2014

RWTH AACHEN UNIVERSITY

Übersicht

1. Beispiele für rekursive Algorithmen
2. Rekursionsgleichungen
3. Lösen von Rekursionsgleichungen
   - Die Substitutionsmethode
   - Rekursionsbäume
   - Mastertheorem
4. Nachtrag Rekursion

Übersicht

1. Beispiele für rekursive Algorithmen

2. Rekursionsgleichungen

3. Lösen von Rekursionsgleichungen
   - Die Substitutionsmethode
   - Rekursionsbäume
   - Mastertheorem

4. Nachtrag Rekursion

Die Teile-und-Beherrsche-Methode

Teile-und-Beherrsche (divide-and-conquer)

- Teile das Problem in eine Anzahl von Teilproblemen auf
- Beherrsche die Teilprobleme
  - kleine Teilprobleme: direktes Lösen
  - große Teilprobleme: Teile-und-Beherrsche
- Verbinde die Lösungen der Teilprobleme zur Lösung des Ausgangsproblems

Summenberechnung

Iterative Berechnung ($n \in \mathbb{N}$)

$$sum(n) = \sum_{i=0}^{n} i$$

Summenberechnung

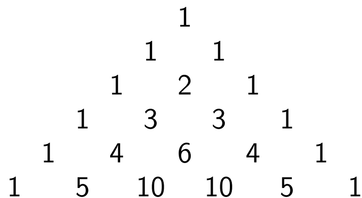
Iterative Berechnung ($n \in \mathbb{N}$)

$$sum(n) = \sum_{i=0}^{n} i$$

Rekursive Berechnung ($n \in \mathbb{N}$)

$$sum(n) = \begin{cases} 0 & \textit{falls } n = 0 \\ sum(n-1) + n & \textit{sonst.} \end{cases}$$

Pascalsches Dreieck



Pascalsches Dreieck

					1					
				1		1				
			1		2		1			
		1		3		3		1		
	1		4		6		4		1	
1		5		10		10		5		1

Rekursive Berechnung ($n, m \in \mathbb{N},\ m \leqslant n$)

$$pas(n, m) = \begin{cases} 1 & \textit{falls } m = 0 \lor m = n \\ pas(n-1, m-1) + pas(n-1, m) & \textit{sonst.} \end{cases}$$

Multiplikation von Binärzahlen der Länge n

- Bisher: Addition und Multiplikation haben Zeitaufwand $\mathcal{O}(1)$
- Addition: $\mathcal{O}(n)$ Bitoperationen
- Multiplikation: Auch $\mathcal{O}(n)$?

Multiplikation von Binärzahlen der Länge n

- Bisher: Addition und Multiplikation haben Zeitaufwand $\mathcal{O}(1)$
- Addition: $\mathcal{O}(n)$ Bitoperationen
- Multiplikation: Auch $\mathcal{O}(n)$?

1. Multiplikationsalgorithmus

$$
\begin{array}{rl}
(a_1 \ldots a_n) \cdot (b_1 \ldots b_n) = & (a_1 \ldots a_n) \cdot b_1 \\
 & \quad (a_1 \ldots a_n) \cdot b_2 \\
 & \qquad \ldots \\
 & \qquad\quad (a_1 \ldots a_n) \cdot b_n \\
\hline
 & \textit{Summenbildung}
\end{array}
$$

$n - 1$ Additionen von Zahlen der Länge $\leqslant 2n \quad \Rightarrow \quad \mathcal{O}(n^2)$!

Multiplikation von Binärzahlen der Länge n

Beobachtung

Seien a, b Binärzahlen mit $n = 2^m$ bits.

$$a = \underbrace{a_1 \ldots a_{n/2}}_{a'} \underbrace{a_{n/2+1} \ldots a_n}_{a''} \qquad b = \underbrace{b_1 \ldots b_{n/2}}_{b'} \underbrace{b_{n/2+1} \ldots b_n}_{b''}$$

Dann

$$\begin{aligned} a \cdot b &= (a' \cdot 2^{\frac{n}{2}} + a'') \cdot (b' \cdot 2^{\frac{n}{2}} + b'') \\ &= a' \cdot b' \cdot 2^n + a' \cdot b'' \cdot 2^{\frac{n}{2}} + a'' \cdot b' \cdot 2^{\frac{n}{2}} + a'' \cdot b'' \end{aligned}$$

Multiplikation von Binärzahlen der Länge n

Beobachtung

Seien a, b Binärzahlen mit $n = 2^m$ bits.

$$a = \underbrace{a_1 \dots a_{n/2}}_{a'} \underbrace{a_{n/2+1} \dots a_n}_{a''} \qquad b = \underbrace{b_1 \dots b_{n/2}}_{b'} \underbrace{b_{n/2+1} \dots b_n}_{b''}$$

Dann

$$\begin{aligned} a \cdot b &= (a' \cdot 2^{\frac{n}{2}} + a'') \cdot (b' \cdot 2^{\frac{n}{2}} + b'') \\ &= a' \cdot b' \cdot 2^n + a' \cdot b'' \cdot 2^{\frac{n}{2}} + a'' \cdot b' \cdot 2^{\frac{n}{2}} + a'' \cdot b'' \end{aligned}$$

2. Multiplikationsalgorithmus

$$mul(a, b, n) = \begin{cases} a \cdot b & \textit{falls } n = 1 \\ mul(a', b', \frac{n}{2}) \cdot 2^n + mul(a', b'', \frac{n}{2}) \cdot 2^{\frac{n}{2}} + & \\ mul(a'', b', \frac{n}{2}) \cdot 2^{\frac{n}{2}} + mul(a'', b'', \frac{n}{2}) & \textit{sonst.} \end{cases}$$

Übersicht

1 Beispiele für rekursive Algorithmen

2 Rekursionsgleichungen

3 Lösen von Rekursionsgleichungen
- Die Substitutionsmethode
- Rekursionsbäume
- Mastertheorem

4 Nachtrag Rekursion

Rekursionsgleichungen

Rekursionsgleichung

- Eine Rekursionsgleichung ist eine Gleichung oder eine Ungleichung, die eine Funktion durch ihre eigenen Funktionswerte für kleinere Eingaben beschreibt.
- Wir benutzen Rekursionsgleichungen um die Laufzeit von rekursiven Algorithmen zu beschreiben.
- Zur Ermittlung der Laufzeit lösen wir diese Gleichungen.

Rekursionsgleichungen für $\mathcal{O}$(Worst-Case-Laufzeit)

Rekursionsgleichung für eine obere Grenze der Worst-Case Laufzeit $T_A(n)$ eines Algorithmus A mit Parameterlänge n aufstellen

- Elementare Operationen und ihre Kosten werden bestimmt.
- Bei sequentieller Komposition werden die Kosten addiert.
- Bei bedingter Verzweigung wird das Maximum der beiden Zweige genommen.
- Der Aufruf eines Unterprogrammes `B` mit Parameterlänge $f(n)$ wird mit den Kosten $T_B(f(n))$ der Ausführung des Unterprogrammes versehen.
- Auch rekursive Aufrufe werden wie Unterprogramm-Aufrufe behandelt.

Rekursionsgleichungen für Worst-Case-Laufzeit

Beispiel: Teile-und-Beherrsche (divide-and-conquer)

- Teile das Problem in eine Anzahl von Teilproblemen auf
- Beherrsche die Teilprobleme
  - kleine Teilprobleme: direktes Lösen
  - große Teilprobleme: Teile-und-Beherrsche
- Verbinde die Lösungen der Teilprobleme zur Lösung des Ausgangsproblems

$$Laufzeit = Laufzeit_{Teilen} + Laufzeit_{Beherrschen} + Laufzeit_{Verbinden}$$

Summenberechnung

Rekursive Berechnung ($n \in \mathbb{N}$)

$$sum(n) = \begin{cases} 0 & \textit{falls } n = 0 \\ sum(n-1) + n & \textit{sonst.} \end{cases}$$

Laufzeit

$$T_{sum}(n) = \begin{cases} 0 & \textit{falls } n = 0 \\ 1 + T_{sum}(n-1) + 1 & \textit{sonst.} \end{cases}$$

Pascalsches Dreieck

Rekursive Berechnung ($n, m \in \mathbb{N},\ m \leqslant n$)

$$pas(n,m) = \begin{cases} 1 & \textit{falls } m = 0 \lor m = n \\ pas(n-1, m-1) + pas(n-1, m) & \textit{sonst.} \end{cases}$$

Laufzeit

$$T_{pas}(n,m) =$$

$$\begin{cases} 0 & \textit{falls } m = 0 \lor m = n \\ 3 + T_{pas}(n-1, m-1) + T_{pas}(n-1, m) + 1 & \textit{sonst.} \end{cases}$$

Multiplikation von Binärzahlen der Länge n

2. Multiplikationsalgorithmus

$$mul(a,b,n) = \begin{cases} a \cdot b & \textit{falls } n = 1 \\ mul(a', b', \frac{n}{2}) \cdot 2^n + mul(a', b'', \frac{n}{2}) \cdot 2^{\frac{n}{2}} + & \\ mul(a'', b', \frac{n}{2}) \cdot 2^{\frac{n}{2}} + mul(a'', b'', \frac{n}{2}) & \textit{sonst.} \end{cases}$$

Laufzeit

$$T_{mul}(n) = \begin{cases} 1 & \textit{falls } n = 1 \\ d_1 \cdot n + 4 \cdot T_{mul}(\frac{n}{2}) + d_2 \cdot n & \textit{sonst.} \end{cases}$$

Übersicht

1. Beispiele für rekursive Algorithmen
2. Rekursionsgleichungen
3. Lösen von Rekursionsgleichungen
   - Die Substitutionsmethode
   - Rekursionsbäume
   - Mastertheorem
4. Nachtrag Rekursion

Ansätze zur Lösung von Rekursionsgleichungen

Lösungsansätze

- Substitutionsmethode: Lösung raten, per Induktion Korrektheit beweisen
- Rekursionsbaum-Methode: Rekursionsbaum aufstellen, Summenformeln beschränken
- Mastertheorem: Liefert Lösungen für Rekursionsgleichungen einer bestimmten Form, einfach aber nicht immer anwendbar

Die Substitutionsmethode

Substitutionsmethode

Die Substitutionsmethode besteht aus zwei Schritten:

1. Rate die Form der Lösung, durch z.B. durch
   - scharfes Hinsehen oder
   - kurze Eingaben ausprobieren und einsetzen.

Die Substitutionsmethode

Substitutionsmethode

Die Substitutionsmethode besteht aus zwei Schritten:

1. Rate die Form der Lösung, durch z.B. durch
   - scharfes Hinsehen oder
   - kurze Eingaben ausprobieren und einsetzen.
2. Vollständige Induktion um die Konstanten zu finden und zu zeigen, dass das Geratene eine Lösung ist.

Die Substitutionsmethode

Substitutionsmethode

Die Substitutionsmethode besteht aus zwei Schritten:

1. Rate die Form der Lösung, durch z.B. durch
   - scharfes Hinsehen oder
   - kurze Eingaben ausprobieren und einsetzen.
2. Vollständige Induktion um die Konstanten zu finden und zu zeigen, dass das Geratene eine Lösung ist.

Bemerkungen

- Diese Methode ist sehr leistungsfähig, aber
- kann nur angewendet werden wenn es relativ einfach ist, die Form der Lösung zu erraten.

Beispiel Substitutionsmethode: Summenberechnung

Laufzeit

$$T_{sum}(n) = \begin{cases} 0 & \textit{falls } n = 0 \\ T_{sum}(n-1) + 2 & \textit{sonst.} \end{cases}$$

Beispiel

- Vermutung: $T_{sum}(n) \in \mathcal{O}(n)$
- Zeige: $\exists c > 0.\ \exists n_0 \geqslant 0.\ \forall n \geqslant n_0.\ T_{sum}(n) \leqslant c \cdot n$
- Induktionsanfang $n_0 = 0$: $T_{sum}(0) = 0 \leqslant c \cdot 0$ for all $c > 0$.
- Induktionsannahme: $T_{sum}(k) \leqslant c \cdot k$ für alle $0 \leqslant k < n$
- Induktionsschritt:

  $T_{sum}(n) = T_{sum}(n-1) + 2 \leqslant c \cdot (n-1) + 2 = c \cdot n + (2-c) \leqslant c \cdot n$ für $c \geqslant 2$.

Die Substitutionsmethode: Annahmenverstärkung

Induktiv vs. invariant

Eine asymptotische Schranke kann nicht immer direkt mit der vollständigen Induktion bewiesen werden.

$$\textit{induktiv} \overset{\rightarrow}{\underset{\not\leftarrow}{}} \textit{invariant}$$

Beispiel

- $T(n) = T\left(\lceil n/2 \rceil\right) + T\left(\lfloor n/2 \rfloor\right) + 1$
- $T(n) \in \mathcal{O}(n)$
- Zu zeigen: $\exists c > 0.\ \exists n_0 \geqslant 0.\ \forall n \geqslant n_0.\ T(n) \leqslant c \cdot n$
- $T(n) = T\left(\lceil n/2 \rceil\right) + T\left(\lfloor n/2 \rfloor\right) + 1 \leqslant c\lceil n/2 \rceil + c\lfloor n/2 \rfloor + 1 = c \cdot n \boxed{+1}$

Die Substitutionsmethode: Annahmenverstärkung

Beispiel

- $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 1$
- $T(n) \in \mathcal{O}(n)$
- Zu zeigen: $\exists c > 0.\ \exists n_0 \geqslant 0.\ \forall n \geqslant n_0.\ T(n) \leqslant c \cdot n$
- $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 1 \leqslant c\lceil n/2 \rceil + c\lfloor n/2 \rfloor + 1 = c \cdot n \boxed{+\ 1}$

Die Substitutionsmethode: Annahmenverstärkung

Beispiel

- $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 1$
- $T(n) \in \mathcal{O}(n)$
- Zu zeigen: $\exists c > 0.\ \exists n_0 \geqslant 0.\ \forall n \geqslant n_0.\ T(n) \leqslant c \cdot n$
- $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 1 \leqslant c\lceil n/2 \rceil + c\lfloor n/2 \rfloor + 1 = c \cdot n \boxed{+ 1}$

- Verstärke die Invariante und zeige $T(n) \in \mathcal{O}(n-1)$ und $n-1 \in \mathcal{O}(n)$
- Zu zeigen: $\exists c > 0.\ \exists n_0 \geqslant 0.\ \forall n \geqslant n_0.\ T(n) \leqslant c \cdot (n-1)$

$$\begin{aligned} T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 1 \\ &\leqslant c(\lceil n/2 \rceil - 1) + c(\lfloor n/2 \rfloor - 1) + 1 \\ &= c \cdot (n-1) + (1-c) \\ &\leqslant c \cdot (n-1) \end{aligned}$$

für $c \geqslant 1$.

$\varphi \quad (O(n))$ $\qquad\qquad\qquad\qquad$ φ invariant

$\big\downarrow$ Verstärkung $(\psi \to \varphi)$ $\qquad\qquad\qquad\qquad$ $\big\uparrow$

$\psi \quad (O(n-1)) \quad \leadsto \quad \psi$ induktiv

Die Substitutionsmethode: Variablentransformation

Manchmal hilft eine Variablentransformation beim Lösen.

Beispiel

$T(n) = 2 \cdot T\left(\lfloor\sqrt{n}\rfloor\right) + \log n$ für $n > 0$
(Im Folgenden vernachlässigen wir die Rundung von Werten.)

$$
\begin{array}{rlcll}
 & T(n) & = & 2 \cdot T\left(\sqrt{n}\right) + \log n & \mid \text{ Variablentransformation } m = \log n \\
\Leftrightarrow & T(2^m) & = & 2 \cdot T\left(2^{m/2}\right) + m & \mid \text{ Umbenennung } T(2^m) = S(m) \\
\Leftrightarrow & S(m) & = & 2 \cdot S(m/2) + m & \mid \text{ Substitutionsmethode} \\
\Leftrightarrow & S(m) & \leqslant & c \cdot m \cdot \log m & \\
\Leftrightarrow & S(m) & \in & \mathcal{O}\left(m \cdot \log m\right) & \mid m = \log n \\
\Leftrightarrow & T(n) & \in & \mathcal{O}\left(\log n \cdot \log \log n\right) &
\end{array}
$$

Übersicht

1 Beispiele für rekursive Algorithmen

2 Rekursionsgleichungen

3 Lösen von Rekursionsgleichungen
- Die Substitutionsmethode
- Rekursionsbäume
- Mastertheorem

4 Nachtrag Rekursion

Raten der Lösung durch Rekursionsbäume

Grundidee

Stelle das Ineinander-Einsetzen als Baum dar, indem man Buch über das aktuelle Rekursionsargument und die nichtrekursiven Kosten führt.

Raten der Lösung durch Rekursionsbäume

Grundidee

Stelle das Ineinander-Einsetzen als Baum dar, indem man Buch über das aktuelle Rekursionsargument und die nichtrekursiven Kosten führt.

Rekursionsbaum

1. Jeder Knoten stellt die Kosten eines Teilproblems dar.

Raten der Lösung durch Rekursionsbäume

Grundidee

Stelle das Ineinander-Einsetzen als Baum dar, indem man Buch über das aktuelle Rekursionsargument und die nichtrekursiven Kosten führt.

Rekursionsbaum

1. Jeder Knoten stellt die Kosten eines Teilproblems dar.
   - Die Wurzel stellt die zu analysierenden Kosten $T(n)$ dar.

Raten der Lösung durch Rekursionsbäume

Grundidee

Stelle das Ineinander-Einsetzen als Baum dar, indem man Buch über das aktuelle Rekursionsargument und die nichtrekursiven Kosten führt.

Rekursionsbaum

1. Jeder Knoten stellt die Kosten eines Teilproblems dar.
   - Die Wurzel stellt die zu analysierenden Kosten $T(n)$ dar.
   - Die Blätter stellen die Kosten der Basisfälle dar, z.B. $T(0)$ oder $T(1)$.

Raten der Lösung durch Rekursionsbäume

Grundidee

Stelle das Ineinander-Einsetzen als Baum dar, indem man Buch über das aktuelle Rekursionsargument und die nichtrekursiven Kosten führt.

Rekursionsbaum

1. Jeder Knoten stellt die Kosten eines Teilproblems dar.
   - Die Wurzel stellt die zu analysierenden Kosten $T(n)$ dar.
   - Die Blätter stellen die Kosten der Basisfälle dar, z.B. $T(0)$ oder $T(1)$.
2. Wir summieren die Kosten innerhalb jeder Ebene des Baumes.

Raten der Lösung durch Rekursionsbäume

Grundidee

Stelle das Ineinander-Einsetzen als Baum dar, indem man Buch über das aktuelle Rekursionsargument und die nichtrekursiven Kosten führt.

Rekursionsbaum

1. Jeder Knoten stellt die Kosten eines Teilproblems dar.
   - Die Wurzel stellt die zu analysierenden Kosten $T(n)$ dar.
   - Die Blätter stellen die Kosten der Basisfälle dar, z.B. $T(0)$ oder $T(1)$.
2. Wir summieren die Kosten innerhalb jeder Ebene des Baumes.
3. Die Gesamtkosten := summieren über die Kosten aller Ebenen.

Raten der Lösung durch Rekursionsbäume

Grundidee

Stelle das Ineinander-Einsetzen als Baum dar, indem man Buch über das aktuelle Rekursionsargument und die nichtrekursiven Kosten führt.

Rekursionsbaum

1. Jeder Knoten stellt die Kosten eines Teilproblems dar.
   - Die Wurzel stellt die zu analysierenden Kosten $T(n)$ dar.
   - Die Blätter stellen die Kosten der Basisfälle dar, z.B. $T(0)$ oder $T(1)$.
2. Wir summieren die Kosten innerhalb jeder Ebene des Baumes.
3. Die Gesamtkosten $:=$ summieren über die Kosten aller Ebenen.

Hinweis

Ein Rekursionsbaum ist sehr nützlich, um eine Lösung zu raten, die dann mit Hilfe der Substitutionsmethode überprüft werden kann.

Raten der Lösung durch Rekursionsbäume

Grundidee

Stelle das Ineinander-Einsetzen als Baum dar, indem man Buch über das aktuelle Rekursionsargument und die nichtrekursiven Kosten führt.

Rekursionsbaum

1. Jeder Knoten stellt die Kosten eines Teilproblems dar.
   - Die Wurzel stellt die zu analysierenden Kosten $T(n)$ dar.
   - Die Blätter stellen die Kosten der Basisfälle dar, z.B. $T(0)$ oder $T(1)$.
2. Wir summieren die Kosten innerhalb jeder Ebene des Baumes.
3. Die Gesamtkosten $:=$ summieren über die Kosten aller Ebenen.

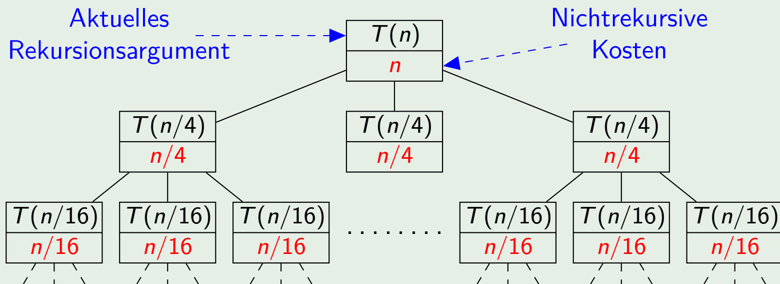
Hinweis

Ein Rekursionsbaum ist sehr nützlich, um eine Lösung zu raten, die dann mit Hilfe der Substitutionsmethode überprüft werden kann.
Der Baum selber reicht jedoch meistens nicht als Beweis.

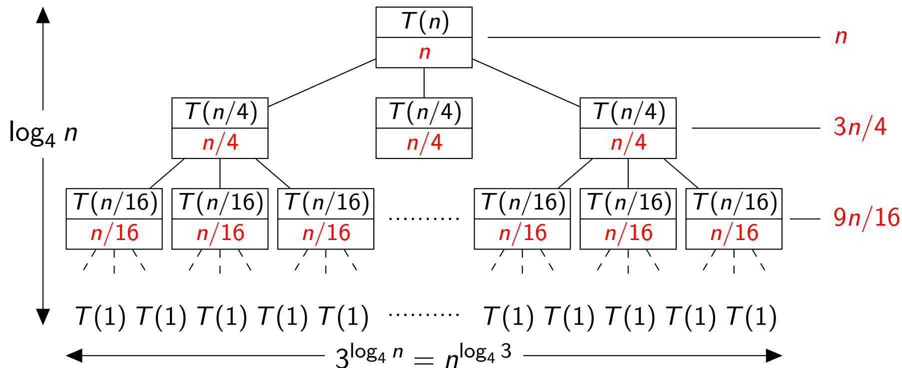
Rekursionsbaum: Beispiel

Beispiel

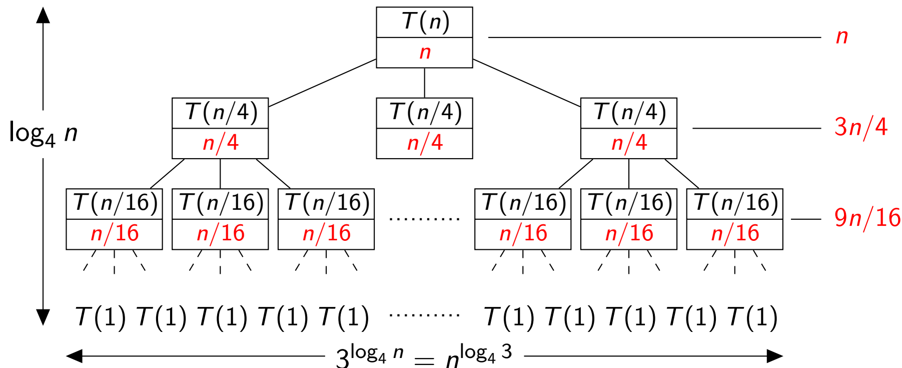
Der Rekursionsbaum von $T(n) = 3 \cdot T(n/4) + n$ sieht etwa so aus:



Rekursionsbaum: Beispiel



Rekursionsbaum: Beispiel



$$T(n) = \underbrace{\sum_{i=0}^{\log_4 n - 1}}_{\text{Summe über alle Ebenen}} \underbrace{\left(\frac{3}{4}\right)^i \cdot n}_{\text{Kosten pro Ebene}} + \underbrace{c \cdot n^{\log_4 3}}_{\text{Gesamtkosten für die Blätter mit } T(1) = c}$$

Rekursionsbaum: Beispiel

Eine obere Schranke für die Komplexität erhält man nun folgendermaßen:

$$T(n) = \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{4}\right)^i \cdot n + c \cdot n^{\log_4 3}$$ | Vernachlässigen kleinerer Terme

$$< \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i \cdot n + c \cdot n^{\log_4 3}$$ | Geometrische Reihe

$$= \frac{1}{1-(3/4)} \cdot n + c \cdot n^{\log_4 3}$$ | Umformen

$$= 4 \cdot n + c \cdot n^{\log_4 3}$$ | Asymptotische Ordnung bestimmen setze ein, dass $\log_4 3 < 1$

$T(n) \in \mathcal{O}(n)$.

Korrektheit

Wir können die Substitutionsmethode benutzen, um die Vermutung zu bestätigen dass:

$T(n) \in \mathcal{O}(n)$ eine obere Schranke von $T(n) = 3 \cdot T(n/4) + n$ ist.

Korrektheit

Wir können die Substitutionsmethode benutzen, um die Vermutung zu bestätigen dass:

$T(n) \in \mathcal{O}(n)$ eine obere Schranke von $T(n) = 3 \cdot T(n/4) + n$ ist.

$$
\begin{aligned}
T(n) &= 3 \cdot T(n/4) + n && \mid \text{ Induktionshypothese} \\
&\leqslant 3c \cdot n/4 + n \\
&= \frac{3}{4}c \cdot n + n \\
&= \left(\frac{3}{4}c + 1\right) \cdot n && \mid \text{ für } c \geqslant 4 \text{ folgt:} \\
&\leqslant c \cdot n
\end{aligned}
$$

Korrektheit

Wir können die Substitutionsmethode benutzen, um die Vermutung zu bestätigen dass:

$T(n) \in \mathcal{O}(n)$ eine obere Schranke von $T(n) = 3 \cdot T(n/4) + n$ ist.

$$
\begin{aligned}
T(n) &= 3 \cdot T(n/4) + n && \mid \text{ Induktionshypothese} \\
&\leqslant 3c \cdot n/4 + n \\
&= \frac{3}{4}c \cdot n + n \\
&= \left(\frac{3}{4}c + 1\right) \cdot n && \mid \text{ für } c \geqslant 4 \text{ folgt:} \\
&\leqslant c \cdot n
\end{aligned}
$$

Und wir stellen fest, dass es ein n_0 gibt, sodass $T(n_0) \leqslant c \cdot n_0$ ist.

Übersicht

1. Beispiele für rekursive Algorithmen
2. Rekursionsgleichungen
3. Lösen von Rekursionsgleichungen
   - Die Substitutionsmethode
   - Rekursionsbäume
   - Mastertheorem
4. Nachtrag Rekursion

Mastertheorem

Allgemeine Form der Rekursionsgleichung

Eine Rekursionsgleichung für die Komplexitätsanalyse sieht meistens folgendermaßen aus:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

wobei $a > 0$, $b > 1$ gilt und $f(n)$ eine gegebene Funktion ist.

Mastertheorem

Allgemeine Form der Rekursionsgleichung

Eine Rekursionsgleichung für die Komplexitätsanalyse sieht meistens folgendermaßen aus:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

wobei $a > 0$, $b > 1$ gilt und $f(n)$ eine gegebene Funktion ist.

Intuition:

- Das zu analysierende Problem teilt sich jeweils in a Teilprobleme auf

Mastertheorem

Allgemeine Form der Rekursionsgleichung

Eine Rekursionsgleichung für die Komplexitätsanalyse sieht meistens folgendermaßen aus:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

wobei $a > 0$, $b > 1$ gilt und $f(n)$ eine gegebene Funktion ist.

Intuition:

- Das zu analysierende Problem teilt sich jeweils in a Teilprobleme auf
- Jedes dieser Teilprobleme hat die Größe $\frac{n}{b}$

Mastertheorem

Allgemeine Form der Rekursionsgleichung

Eine Rekursionsgleichung für die Komplexitätsanalyse sieht meistens folgendermaßen aus:

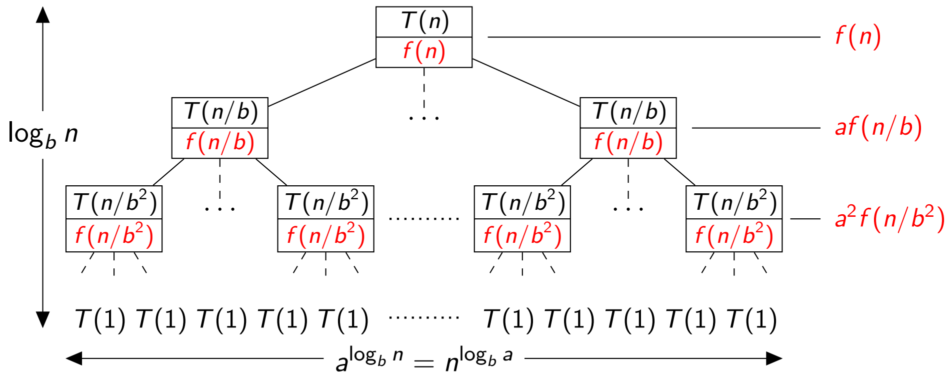
$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

wobei $a > 0$, $b > 1$ gilt und $f(n)$ eine gegebene Funktion ist.

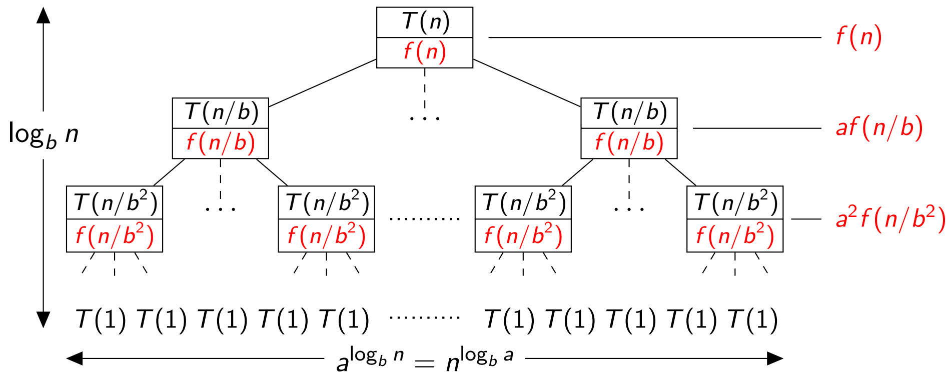
Intuition:

- Das zu analysierende Problem teilt sich jeweils in a Teilprobleme auf
- Jedes dieser Teilprobleme hat die Größe $\frac{n}{b}$
- Die Kosten für das Aufteilen eines Problems und Kombinieren der Teillösungen sind $f(n)$.

Erinnerung: Rekursionsbaum $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$



Erinnerung: Rekursionsbaum $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$



$$T(n) = \underbrace{\sum_{i=0}^{\log_b n - 1}}_{\text{Summe über alle Ebenen}} \underbrace{a^i f\left(n/b^i\right)}_{\text{Kosten pro Ebene}} + \underbrace{c \cdot n^{\log_b a}}_{\text{Gesamtkosten für die Blätter mit } T(1) = c}$$

Das Mastertheorem

$$\boxed{T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)} \text{ mit } a \geqslant 1 \text{ und } b > 1.$$

- Anzahl der Blätter im Rekursionsbaum: n^E mit $E = \log_b a$.

Das Mastertheorem

$$\boxed{T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)} \text{ mit } a \geqslant 1 \text{ und } b > 1.$$

- Anzahl der Blätter im Rekursionsbaum: n^E mit $E = \log_b a$.

Mastertheorem

	Wenn	Dann
1.	$f(n) \in \mathcal{O}(n^{E-\varepsilon})$ für ein $\varepsilon > 0$	$T(n) \in \Theta(n^E)$

Das Mastertheorem

$$\boxed{T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)} \text{ mit } a \geqslant 1 \text{ und } b > 1.$$

- Anzahl der Blätter im Rekursionsbaum: n^E mit $E = \log_b a$.

Mastertheorem

	Wenn	Dann
1.	$f(n) \in \mathcal{O}(n^{E-\varepsilon})$ für ein $\varepsilon > 0$	$T(n) \in \Theta(n^E)$
2.	$f(n) \in \Theta(n^E)$	$T(n) \in \Theta(n^E \cdot \log n)$

Das Mastertheorem

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n) \quad \text{mit } a \geqslant 1 \text{ und } b > 1.$$

- Anzahl der Blätter im Rekursionsbaum: n^E mit $E = \log_b a$.

Mastertheorem

	Wenn	Dann
1.	$f(n) \in \mathcal{O}(n^{E-\varepsilon})$ für ein $\varepsilon > 0$	$T(n) \in \Theta(n^E)$
2.	$f(n) \in \Theta(n^E)$	$T(n) \in \Theta(n^E \cdot \log n)$
3.	$f(n) \in \Omega(n^{E+\varepsilon})$ für ein $\varepsilon > 0$ und $a \cdot f(n/b) \leqslant d \cdot f(n)$ für ein $d < 1$ und n hinreichend groß	$T(n) \in \Theta(f(n))$

Das Mastertheorem

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$ mit $a \geqslant 1$ und $b > 1$.

- Anzahl der Blätter im Rekursionsbaum: n^E mit $E = \log_b a$.

Mastertheorem

	Wenn	Dann
1.	$f(n) \in \mathcal{O}(n^{E-\varepsilon})$ für ein $\varepsilon > 0$	$T(n) \in \Theta(n^E)$
2.	$f(n) \in \Theta(n^E)$	$T(n) \in \Theta(n^E \cdot \log n)$
3.	$f(n) \in \Omega(n^{E+\varepsilon})$ für ein $\varepsilon > 0$ und $a \cdot f(n/b) \leqslant d \cdot f(n)$ für ein $d < 1$ und n hinreichend groß	$T(n) \in \Theta(f(n))$

- Bemerke, dass das Mastertheorem nicht alle Fälle abdeckt.

Das Mastertheorem verstehen

In jedem der 3 Fälle wird die Funktion $f(n)$ mit $n^E = n^{\log_b a}$ verglichen.

Das Mastertheorem verstehen

In jedem der 3 Fälle wird die Funktion $f(n)$ mit $n^E = n^{\log_b a}$ verglichen.

Mastertheorem: Intuition

	Wenn	Dann
1.	$f(n)$ polynomiell kleiner ist als n^E	$T(n) \in \Theta(n^E)$

Das Mastertheorem verstehen

In jedem der 3 Fälle wird die Funktion $f(n)$ mit $n^E = n^{\log_b a}$ verglichen.

Mastertheorem: Intuition

	Wenn	Dann
1.	$f(n)$ polynomiell kleiner ist als n^E	$T(n) \in \Theta(n^E)$
2.	$f(n)$ und n^E die gleiche Größe haben	$T(n) \in \Theta(n^E \cdot \log n)$

Das Mastertheorem verstehen

In jedem der 3 Fälle wird die Funktion $f(n)$ mit $n^E = n^{\log_b a}$ verglichen.

Mastertheorem: Intuition

	Wenn	Dann
1.	$f(n)$ polynomiell kleiner ist als n^E	$T(n) \in \Theta(n^E)$
2.	$f(n)$ und n^E die gleiche Größe haben	$T(n) \in \Theta(n^E \cdot \log n)$
3.	$f(n)$ ist polynomiell größer als n^E und erfüllt $a \cdot f(n/b) \leqslant d \cdot f(n)$	$T(n) \in \Theta(f(n))$

Nicht abgedeckte Fälle:

1. $f(n)$ ist kleiner als n^E, jedoch nicht polynomiell kleiner.
2. $f(n)$ ist größer als n^E, jedoch nicht polynomiell größer.
3. $f(n)$ ist polynomiell größer als n^E, erfüllt nicht $a \cdot f(n/b) \leqslant d \cdot f(n)$.

Anwendung des Mastertheorems

Beispiel

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

Anwendung des Mastertheorems

Beispiel

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

- Somit: $a = 4$, $b = 2$ und $f(n) = n$; $E = \log 4 / \log 2 = 2$.

Anwendung des Mastertheorems

Beispiel

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

- Somit: $a = 4$, $b = 2$ und $f(n) = n$; $E = \log 4 / \log 2 = 2$.
- Da $f(n) = n \in \mathcal{O}(n^{2-\varepsilon})$, gilt Fall 1: $\boxed{T(n) \in \Theta(n^2)}$

Anwendung des Mastertheorems

Beispiel

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

- Somit: $a = 4$, $b = 2$ und $f(n) = n$; $E = \log 4 / \log 2 = 2$.
- Da $f(n) = n \in \mathcal{O}(n^{2-\varepsilon})$, gilt Fall 1: $\boxed{T(n) \in \Theta(n^2)}$

Beispiel

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

Anwendung des Mastertheorems

Beispiel

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

- Somit: $a = 4$, $b = 2$ und $f(n) = n$; $E = \log 4 / \log 2 = 2$.
- Da $f(n) = n \in \mathcal{O}(n^{2-\varepsilon})$, gilt Fall 1: $\boxed{T(n) \in \Theta(n^2)}$

Beispiel

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

- Somit: $a = 4$, $b = 2$ und $f(n) = n^2$; $E = \log 4 / \log 2 = 2$.

Anwendung des Mastertheorems

Beispiel

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

- Somit: $a = 4$, $b = 2$ und $f(n) = n$; $E = \log 4 / \log 2 = 2$.
- Da $f(n) = n \in \mathcal{O}(n^{2-\varepsilon})$, gilt Fall 1: $\boxed{T(n) \in \Theta(n^2)}$

Beispiel

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

- Somit: $a = 4$, $b = 2$ und $f(n) = n^2$; $E = \log 4 / \log 2 = 2$.
- Da $f(n) = n^2 \notin \mathcal{O}(n^{2-\varepsilon})$, gilt Fall 1 nicht.

Anwendung des Mastertheorems

Beispiel

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

- Somit: $a = 4$, $b = 2$ und $f(n) = n$; $E = \log 4 / \log 2 = 2$.
- Da $f(n) = n \in \mathcal{O}(n^{2-\varepsilon})$, gilt Fall 1: $\boxed{T(n) \in \Theta(n^2)}$

Beispiel

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

- Somit: $a = 4$, $b = 2$ und $f(n) = n^2$; $E = \log 4 / \log 2 = 2$.
- Da $f(n) = n^2 \notin \mathcal{O}(n^{2-\varepsilon})$, gilt Fall 1 nicht.
- Aber weil $f(n) = n^2 \in \Theta(n^2)$, gilt Fall 2: $\boxed{T(n) \in \Theta(n^2 \cdot \log n)}$

Anwendung des Mastertheorems

Beispiel

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

Anwendung des Mastertheorems

Beispiel

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

- Somit: $a = 4$, $b = 2$ und $f(n) = n^3$; $E = \log 4 / \log 2 = 2$.

Anwendung des Mastertheorems

Beispiel

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

- Somit: $a = 4$, $b = 2$ und $f(n) = n^3$; $E = \log 4 / \log 2 = 2$.
- Wegen $E = 2$ gelten Fälle 1 und 2 offenbar nicht.

Anwendung des Mastertheorems

Beispiel

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

- Somit: $a = 4$, $b = 2$ und $f(n) = n^3$; $E = \log 4 / \log 2 = 2$.
- Wegen $E = 2$ gelten Fälle 1 und 2 offenbar nicht.
- Da $f(n) = n^3 \in \Omega(n^{2+\varepsilon})$ für $\varepsilon = 1$, könnte Fall 3 gelten.

Anwendung des Mastertheorems

Beispiel

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

- Somit: $a = 4$, $b = 2$ und $f(n) = n^3$; $E = \log 4 / \log 2 = 2$.
- Wegen $E = 2$ gelten Fälle 1 und 2 offenbar nicht.
- Da $f(n) = n^3 \in \Omega(n^{2+\varepsilon})$ für $\varepsilon = 1$, könnte Fall 3 gelten.
- Überprüfe: gilt $f(n/2) \leqslant \frac{d}{4} \cdot f(n)$ für ein $d < 1$ und hinreichend große n?

Anwendung des Mastertheorems

Beispiel

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

- Somit: $a = 4$, $b = 2$ und $f(n) = n^3$; $E = \log 4 / \log 2 = 2$.
- Wegen $E = 2$ gelten Fälle 1 und 2 offenbar nicht.
- Da $f(n) = n^3 \in \Omega(n^{2+\varepsilon})$ für $\varepsilon = 1$, könnte Fall 3 gelten.
- Überprüfe: gilt $f(n/2) \leqslant \frac{d}{4} \cdot f(n)$ für ein $d < 1$ und hinreichend große n?
- Dies liefert $\frac{1}{8} n^3 \leqslant \frac{d}{4} \cdot n^3$, und dies gilt für alle $\frac{1}{2} \leqslant d < 1$ (und n)

Anwendung des Mastertheorems

Beispiel

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

- Somit: $a = 4$, $b = 2$ und $f(n) = n^3$; $E = \log 4 / \log 2 = 2$.
- Wegen $E = 2$ gelten Fälle 1 und 2 offenbar nicht.
- Da $f(n) = n^3 \in \Omega(n^{2+\varepsilon})$ für $\varepsilon = 1$, könnte Fall 3 gelten.
- Überprüfe: gilt $f(n/2) \leqslant \frac{d}{4} \cdot f(n)$ für ein $d < 1$ und hinreichend große n?
- Dies liefert $\frac{1}{8} n^3 \leqslant \frac{d}{4} \cdot n^3$, und dies gilt für alle $\frac{1}{2} \leqslant d < 1$ (und n)
- Somit gilt Fall 3 tatsächlich und wir folgern: $T(n) \in \Theta(n^3)$

Das Mastertheorem ist nicht immer anwendbar

Beispiel

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

Das Mastertheorem ist nicht immer anwendbar

Beispiel

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

- Also gilt: $a = 4$, $b = 2$ und $f(n) = n^2 / \log n$; $E = 2$.

Das Mastertheorem ist nicht immer anwendbar

Beispiel

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

- Also gilt: $a = 4$, $b = 2$ und $f(n) = n^2 / \log n$; $E = 2$.

Fall 1 ist nicht anwendbar:

$$n^2 / \log n \notin \mathcal{O}(n^{2-\varepsilon}), \text{ da } f(n)/n^2 = (\log n)^{-1} \notin \mathcal{O}(n^{-\varepsilon}).$$

Das Mastertheorem ist nicht immer anwendbar

Beispiel

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

- Also gilt: $a = 4$, $b = 2$ und $f(n) = n^2/\log n$; $E = 2$.

Fall 1 ist nicht anwendbar:

$$n^2/\log n \notin \mathcal{O}(n^{2-\varepsilon}), \text{ da } f(n)/n^2 = (\log n)^{-1} \notin \mathcal{O}(n^{-\varepsilon}).$$

Fall 2 ist nicht anwendbar: $n^2/\log n \notin \Theta(n^2)$.

Das Mastertheorem ist nicht immer anwendbar

Beispiel

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

- Also gilt: $a = 4$, $b = 2$ und $f(n) = n^2/\log n$; $E = 2$.

Fall 1 ist nicht anwendbar:

$$n^2/\log n \notin \mathcal{O}(n^{2-\varepsilon}), \text{ da } f(n)/n^2 = (\log n)^{-1} \notin \mathcal{O}(n^{-\varepsilon}).$$

Fall 2 ist nicht anwendbar: $n^2/\log n \notin \Theta(n^2)$.

Fall 3 ist nicht anwendbar:

$$f(n) \notin \Omega(n^{2+\varepsilon}), \text{ da } f(n)/n^2 = (\log n)^{-1} \notin \mathcal{O}(n^{+\varepsilon}).$$

Das Mastertheorem ist nicht immer anwendbar

Beispiel

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

- Also gilt: $a = 4$, $b = 2$ und $f(n) = n^2 / \log n$; $E = 2$.

Fall 1 ist nicht anwendbar:

$$n^2 / \log n \notin \mathcal{O}(n^{2-\varepsilon}), \text{ da } f(n)/n^2 = (\log n)^{-1} \notin \mathcal{O}(n^{-\varepsilon}).$$

Fall 2 ist nicht anwendbar: $n^2 / \log n \notin \Theta(n^2)$.

Fall 3 ist nicht anwendbar:

$$f(n) \notin \Omega(n^{2+\varepsilon}), \text{ da } f(n)/n^2 = (\log n)^{-1} \notin \mathcal{O}(n^{+\varepsilon}).$$

$\Longrightarrow$ Das Mastertheorem hilft hier überhaupt nicht weiter!

- Durch Substitution erhält man: $\boxed{T(n) \in \Theta(n^2 \cdot \log \log n)}$

Übersicht

1. Beispiele für rekursive Algorithmen
2. Rekursionsgleichungen
3. Lösen von Rekursionsgleichungen
   - Die Substitutionsmethode
   - Rekursionsbäume
   - Mastertheorem
4. Nachtrag Rekursion

Recall: Worst-Case Rekursionsgleichungen

Rekursions(un)gleichung für eine obere Grenze der Worst-Case Laufzeit $T_A(n)$ eines Algorithmus A mit Parameterlänge n aufstellen

- Elementare Operationen und ihre Kosten werden bestimmt.
- Bei sequentieller Komposition werden die Kosten addiert.
- Bei bedingter Verzweigung werden die Kosten zur Berechnung der Verzweigungsbedingung plus das Maximum der beiden Zweige genommen.
- Der Aufruf eines Unterprogrammes B mit Parameterlänge $f(n)$ wird mit den Kosten zur Berechnung der Parameter plus die Kosten $T_B(f(n))$ der Ausführung des Unterprogrammes versehen.
- Auch rekursive Aufrufe werden wie Unterprogramm-Aufrufe behandelt.

Recall: Worst-Case Rekursionsgleichungen

Rekursions(un)gleichung für eine obere Grenze der Worst-Case Laufzeit $T_A(n)$ eines Algorithmus A mit Parameterlänge n aufstellen

- Elementare Operationen und ihre Kosten werden bestimmt.
- Bei sequentieller Komposition werden die Kosten addiert.
- Bei bedingter Verzweigung werden die Kosten zur Berechnung der Verzweigungsbedingung plus das Maximum der beiden Zweige genommen.
- Der Aufruf eines Unterprogrammes `B` mit Parameterlänge $f(n)$ wird mit den Kosten zur Berechnung der Parameter plus die Kosten $T_B(f(n))$ der Ausführung des Unterprogrammes versehen.
- Auch rekursive Aufrufe werden wie Unterprogramm-Aufrufe behandelt.

Maximum bei bedingter Verzweigung

- Vorsicht bei Rekursion in den Zweigen!

Beispiel: Fibonacci-Folge

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

Beispiel: Fibonacci-Folge

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

- Mit Maximum: $T_{fib}(n) = \max(0, T_{fib}(n-1) + T_{fib}(n-2) + 3)$

Beispiel: Fibonacci-Folge

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

- Mit Maximum: $T_{fib}(n) = \max(0, T_{fib}(n-1) + T_{fib}(n-2) + 3)$
  Ok, wenn wir annehmen, dass $T_{fib}(n)$ für kleine n konstant ist.

Beispiel: Fibonacci-Folge

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

- Mit Maximum: $T_{fib}(n) = \max(0, T_{fib}(n-1) + T_{fib}(n-2) + 3)$

  Ok, wenn wir annehmen, dass $T_{fib}(n)$ für kleine n konstant ist.
- Aber besser:

$$T_{fib}(n) = \begin{cases} 0 & \textit{wenn } n \leqslant 1 \\ T_{fib}(n-1) + T_{fib}(n-2) + 3 & \textit{sonst.} \end{cases}$$

Beispiel: Sinnlose Methode

```
int m(int n) {
   if (false) return m(n+1);
   else return n;
}
```

Beispiel: Sinnlose Methode

```
int m(int n) {
   if (false) return m(n+1);
   else return n;
}
```

- Laufzeit in $\Theta(1)$

Beispiel: Sinnlose Methode

```
int m(int n) {
   if (false) return m(n+1);
   else return n;
}
```

- Laufzeit in $\Theta(1)$
- Rekursionsgleichung mit Maximum: $T_m(n) = \max(T_m(n+1) + 1, 0)$

Beispiel: Sinnlose Methode

```
int m(int n) {
   if (false) return m(n+1);
   else return n;
}
```

- Laufzeit in $\Theta(1)$
- Rekursionsgleichung mit Maximum: $T_m(n) = \max(T_m(n+1) + 1, 0)$
- Sie hat keine Lösung (vom Typ $T_m : \mathbb{N} \to \mathbb{R}_{\geqslant 0}$)

(Nicht)terminierung

```
int m(int n) {
   return m(n+1);
}
```

(Nicht)terminierung

```
int m(int n) {
  return m(n+1);
}
```

Was passiert eigentlich, wenn die Rekursion nicht terminiert?

(Nicht)terminierung

```
int m(int n) {
   return m(n+1);
}
```

Was passiert eigentlich, wenn die Rekursion nicht terminiert?
Dann ist die Laufzeit doch unendlich...

(Nicht)terminierung

```
int m(int n) {
  return m(n+1);
}
```

Was passiert eigentlich, wenn die Rekursion nicht terminiert?
Dann ist die Laufzeit doch unendlich...

$$T_m(n) = T_m(n{+}1) + 1$$

(Nicht)terminierung

```
int m(int n) {
   return m(n+1);
}
```

Was passiert eigentlich, wenn die Rekursion nicht terminiert?
Dann ist die Laufzeit doch unendlich...

$$T_m(n) = T_m(n+1) + 1$$

- Diese Rekursionsgleichung hat keine Lösung vom Typ $T_m : \mathbb{N} \rightarrow \mathbb{R}_{\geqslant 0}$

(Nicht)terminierung

```
int m(int n) {
   return m(n+1);
}
```

Was passiert eigentlich, wenn die Rekursion nicht terminiert?
Dann ist die Laufzeit doch unendlich...

$$T_m(n) = T_m(n+1) + 1$$

- Diese Rekursionsgleichung hat keine Lösung vom Typ $T_m : \mathbb{N} \to \mathbb{R}_{\geqslant 0}$
- Nur eine vom Typ $T_m : \mathbb{N} \to \mathbb{R}_{\geqslant 0} \cup \{\infty\}$

(Nicht)terminierung

```
int m(int n) {
   return m(n+1);
}
```

Was passiert eigentlich, wenn die Rekursion nicht terminiert?
Dann ist die Laufzeit doch unendlich...

$$T_m(n) = T_m(n{+}1) + 1$$

- Diese Rekursionsgleichung hat keine Lösung vom Typ $T_m : \mathbb{N} \to \mathbb{R}_{\geqslant 0}$
- Nur eine vom Typ $T_m : \mathbb{N} \to \mathbb{R}_{\geqslant 0} \cup \{\infty\}$

Annahme

Wir betrachten nur Algorithmen, die für jede Eingabe terminieren.

Vorteile und Nachteile der Rekursion

Vorteile

- Eleganz: Einfache mathematische Formulierung
- Problemreduktion: Führt das Lösen eines Problems auf das Lösen einfacherer Probleme zurück

Nachteile

- Laufzeiteffizienz: Man läuft leicht Gefahr, den gleichen Term mehrfach aus zu werten
- Speichereffizienz: Jeder Methodenaufruf hat seine eigene lokale Variablen

Rekursion kann eliminiert werden durch die Verwendung von Schleifen.

Beispiel: Fibonacci-Folge

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

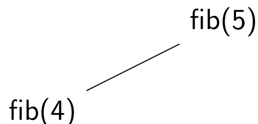
Beispiel: Fibonacci-Folge

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

fib(5)

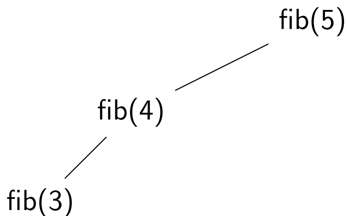
Beispiel: Fibonacci-Folge

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



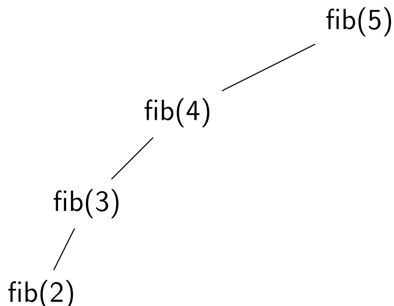
Beispiel: Fibonacci-Folge

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



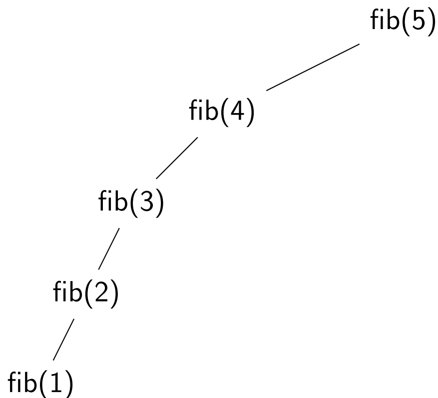
Beispiel: Fibonacci-Folge

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



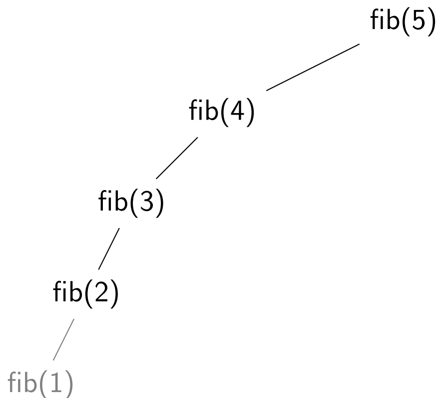
Beispiel: Fibonacci-Folge

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



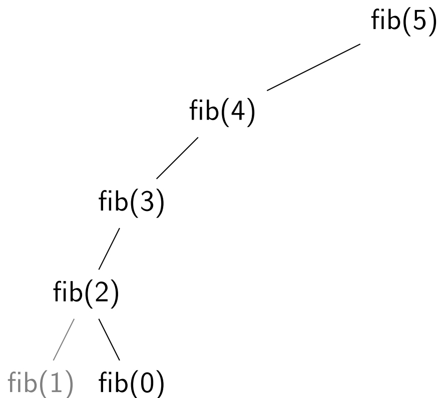
Beispiel: Fibonacci-Folge

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



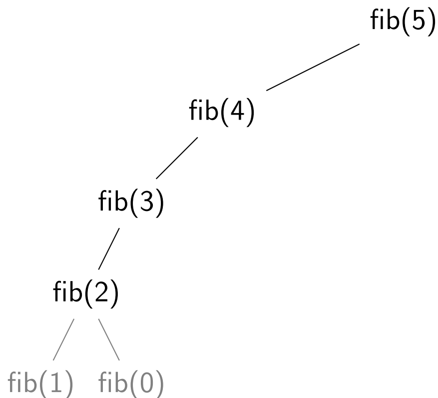
Beispiel: Fibonacci-Folge

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



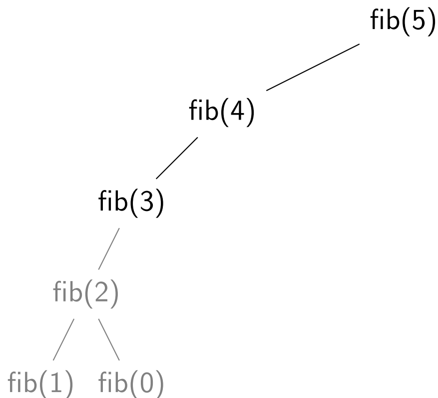
Beispiel: Fibonacci-Folge

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



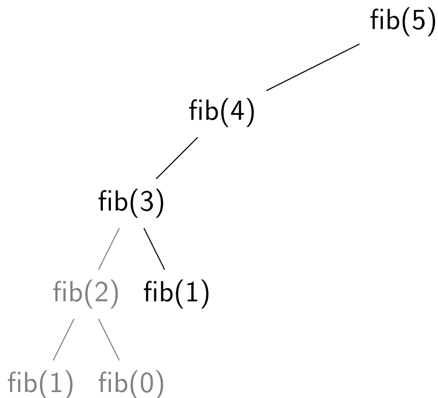
Beispiel: Fibonacci-Folge

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



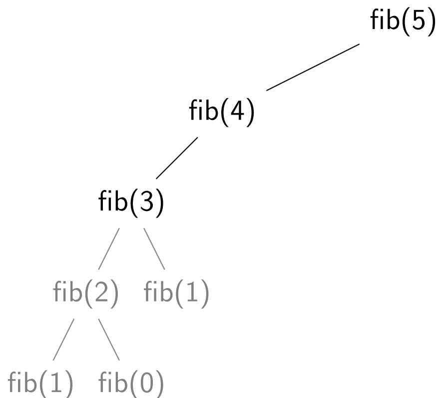
Beispiel: Fibonacci-Folge

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



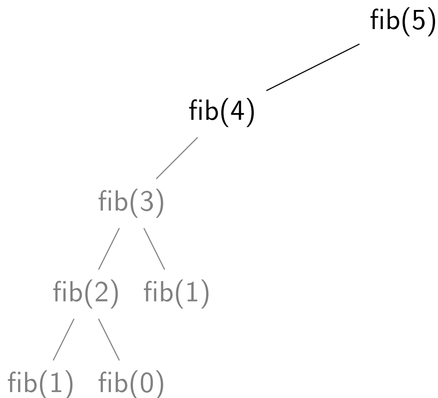
Beispiel: Fibonacci-Folge

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



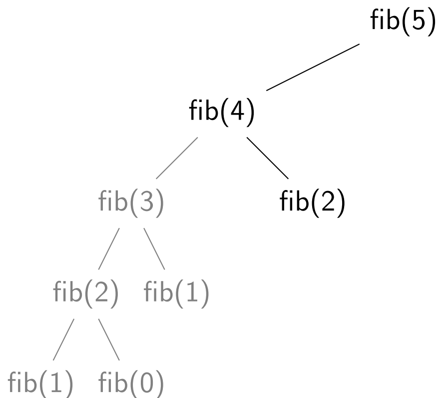
Beispiel: Fibonacci-Folge

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



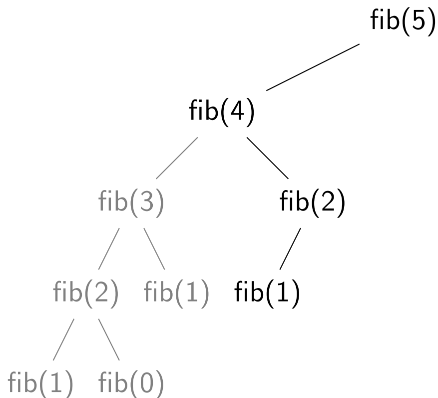
Beispiel: Fibonacci-Folge

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



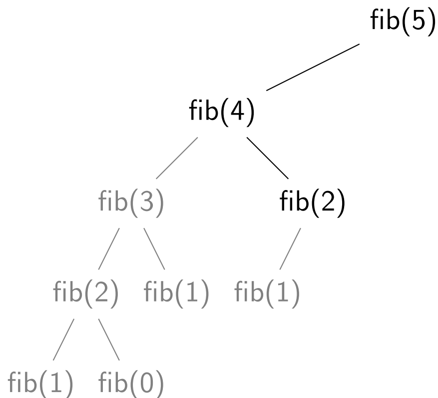
Beispiel: Fibonacci-Folge

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



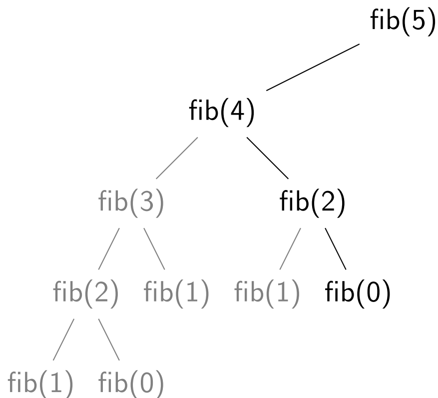
Beispiel: Fibonacci-Folge

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



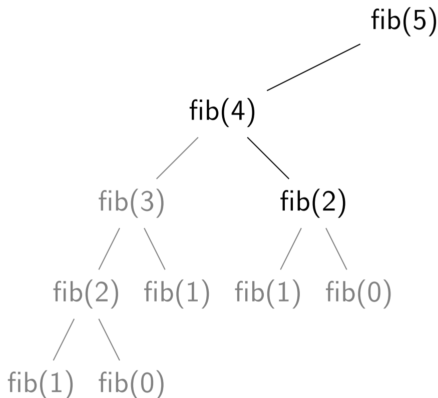
Beispiel: Fibonacci-Folge

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



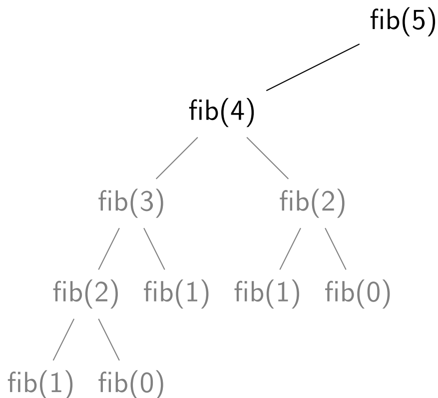
Beispiel: Fibonacci-Folge

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



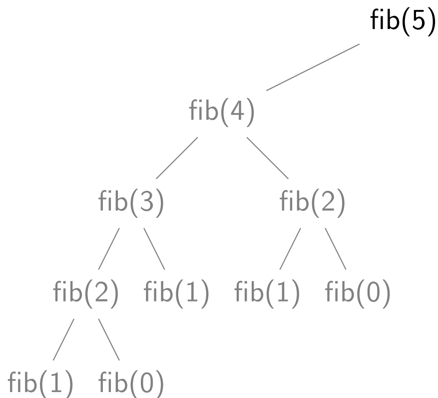
Beispiel: Fibonacci-Folge

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



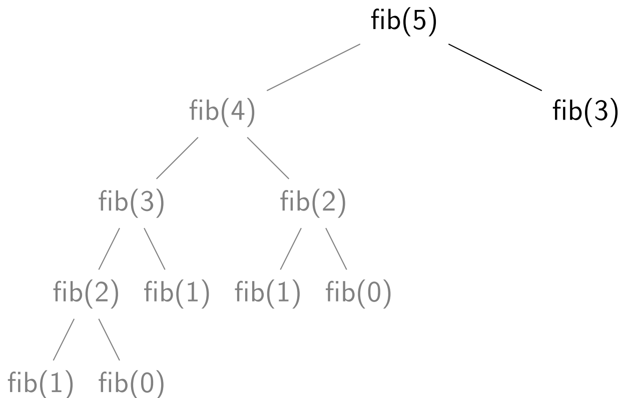
Beispiel: Fibonacci-Folge

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



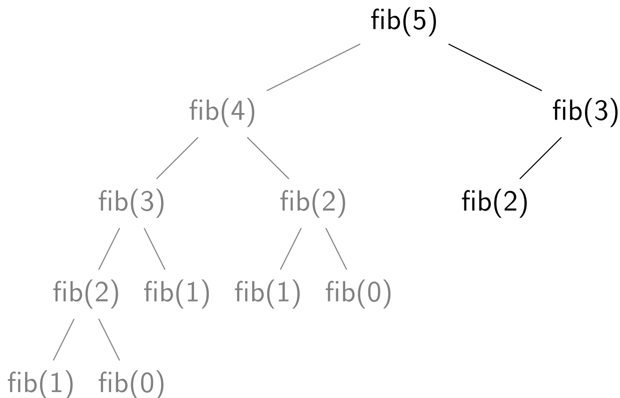
Beispiel: Fibonacci-Folge

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



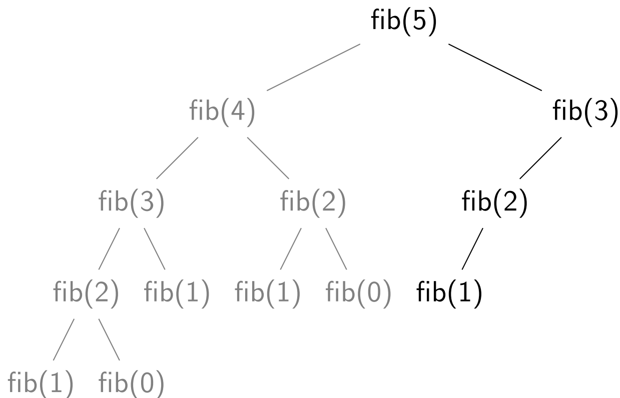
Beispiel: Fibonacci-Folge

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



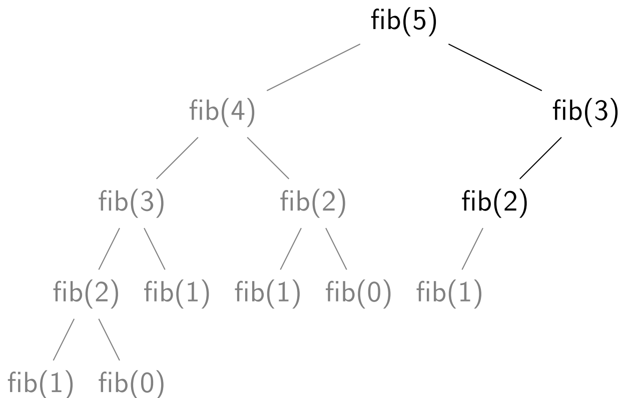
Beispiel: Fibonacci-Folge

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



Beispiel: Fibonacci-Folge

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



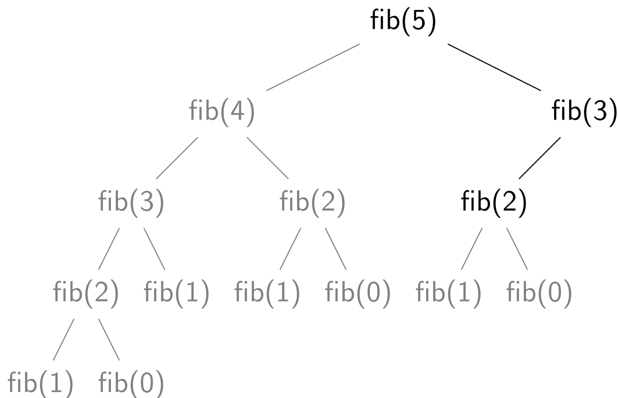
Beispiel: Fibonacci-Folge

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

fib(5)
fib(4) fib(3)
fib(3) fib(2) fib(2)
fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)
fib(1) fib(0)

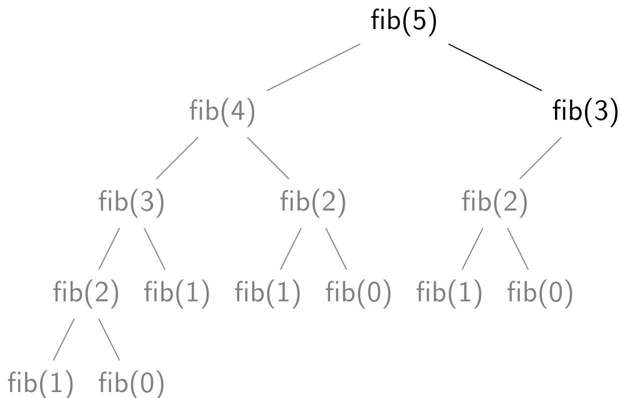
Beispiel: Fibonacci-Folge

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



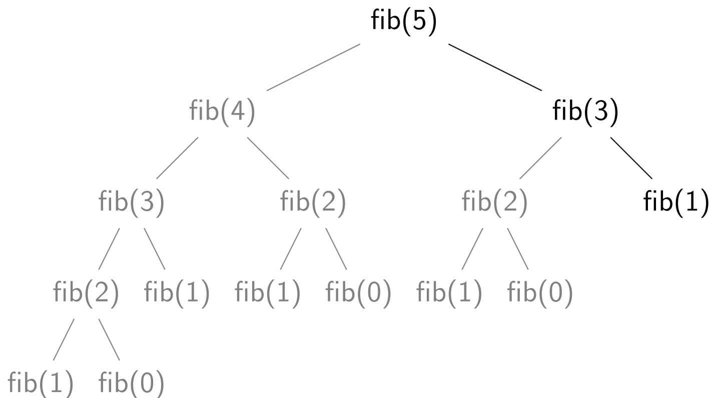
Beispiel: Fibonacci-Folge

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



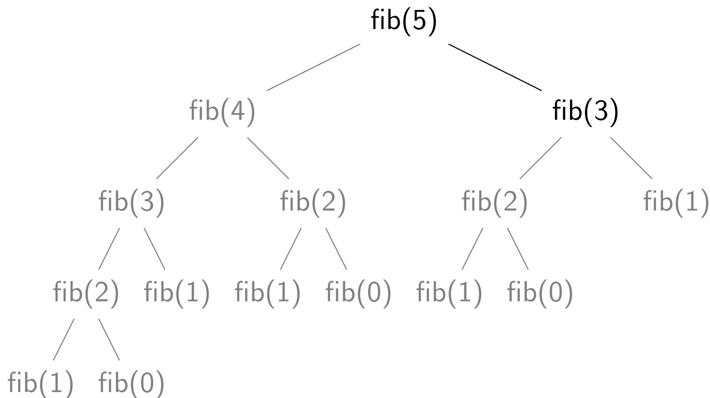
Beispiel: Fibonacci-Folge

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



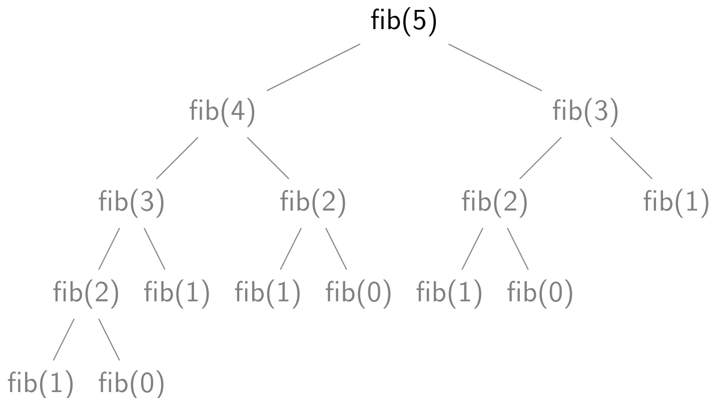
Beispiel: Fibonacci-Folge

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



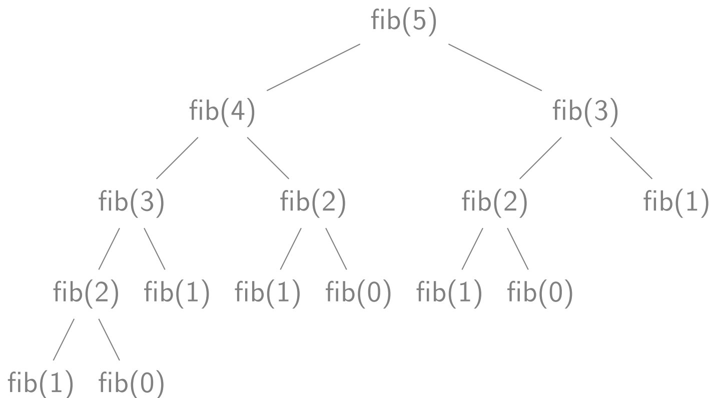
Beispiel: Fibonacci-Folge

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



Beispiel: Fibonacci-Folge

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



Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```

Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```

Beispiel: Fakultätsrechnung

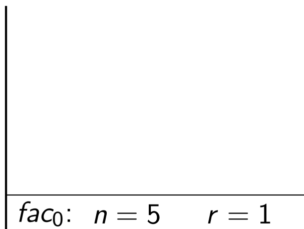
```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

fac_0: $n = 5$

```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```

Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```




```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```

Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

fac_1: $n = 4$
fac_0: $n = 5$ $r = 1$

```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```

Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

fac_1:	$n = 4$	$r = 1$
fac_0:	$n = 5$	$r = 1$

```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```

Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

fac_2:	$n = 3$	
fac_1:	$n = 4$	$r = 1$
fac_0:	$n = 5$	$r = 1$

```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```

Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

fac_2:	$n = 3$	$r = 1$
fac_1:	$n = 4$	$r = 1$
fac_0:	$n = 5$	$r = 1$

```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```

Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

fac_3:	$n = 2$	
fac_2:	$n = 3$	$r = 1$
fac_1:	$n = 4$	$r = 1$
fac_0:	$n = 5$	$r = 1$

```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```

Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

fac_3:	$n = 2$	$r = 1$
fac_2:	$n = 3$	$r = 1$
fac_1:	$n = 4$	$r = 1$
fac_0:	$n = 5$	$r = 1$

```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```

Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

fac_4:	$n = 1$	
fac_3:	$n = 2$	$r = 1$
fac_2:	$n = 3$	$r = 1$
fac_1:	$n = 4$	$r = 1$
fac_0:	$n = 5$	$r = 1$

```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```

Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

fac_4:	$n = 1$	$r = 1$
fac_3:	$n = 2$	$r = 1$
fac_2:	$n = 3$	$r = 1$
fac_1:	$n = 4$	$r = 1$
fac_0:	$n = 5$	$r = 1$

```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```

Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

fac_4:	1	
fac_3:	$n = 2$	$r = 1$
fac_2:	$n = 3$	$r = 1$
fac_1:	$n = 4$	$r = 1$
fac_0:	$n = 5$	$r = 1$

```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```

Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

fac_3:	$n = 2$	$r = 2$
fac_2:	$n = 3$	$r = 1$
fac_1:	$n = 4$	$r = 1$
fac_0:	$n = 5$	$r = 1$

```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```

Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

fac_3:	2	
fac_2:	$n = 3$	$r = 1$
fac_1:	$n = 4$	$r = 1$
fac_0:	$n = 5$	$r = 1$

```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```

Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

fac_2:	$n = 3$	$r = 6$
fac_1:	$n = 4$	$r = 1$
fac_0:	$n = 5$	$r = 1$

```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```

Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

fac_2:	6	
fac_1:	$n = 4$	$r = 1$
fac_0:	$n = 5$	$r = 1$

```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```

Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

fac_1:	$n = 4$	$r = 24$
fac_0:	$n = 5$	$r = 1$

```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```

Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

fac_1: 24
fac_0: $n = 5$ $r = 1$

```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```

Beispiel: Fakultätsrechnung

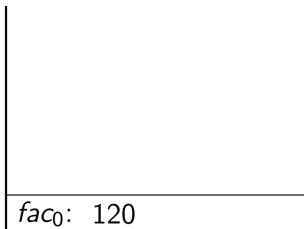
```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

fac_0:	$n = 5$	$r = 120$

```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```

Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```




```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```

Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```

Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```

Beispiel: Fakultätsrechnung

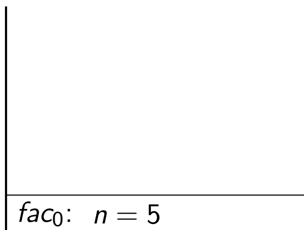
```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```

Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

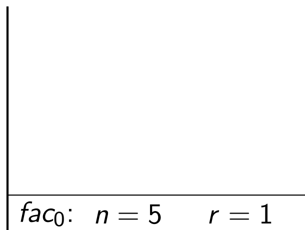
```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```



Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

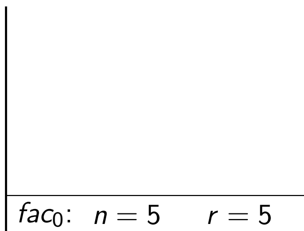
```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```



Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

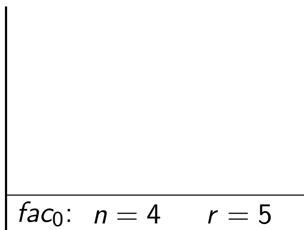
```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```



Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

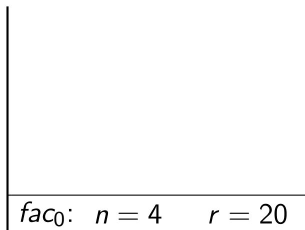
```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```



Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

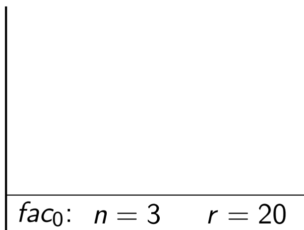
```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```



Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

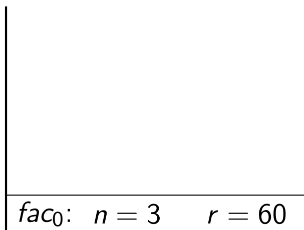
```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```



Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

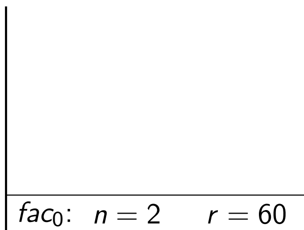
```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```



Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

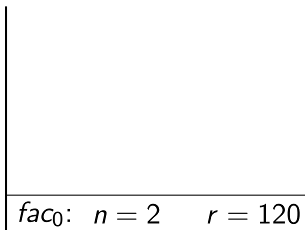
```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```



Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

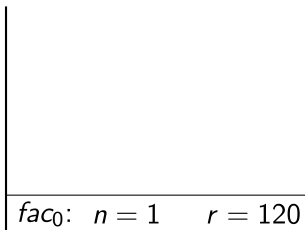
```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```



Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

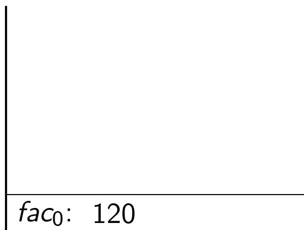
```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```



Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```



Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```

Beispiel: Fakultätsrechnung

```
int fac(int n) {
   int r = 1;
   if (n > 1)
      r = n * fac(n-1);
   return r;
}
```

```
int fac(int n) {
   int r = 1;
   while (n > 1){
      r = n * r;
      n--;
   }
   return r;
}
```