

# Datenstrukturen und Algorithmen

## Vorlesung 5: Rekursionsgleichungen (K4)

Joost-Pieter Katoen

Lehrstuhl für Informatik 2  
Software Modeling and Verification Group

<https://moves.rwth-aachen.de/teaching/ss-18/dsa1/>

24. April 2015



## Übersicht

- 1 Binäre Suche
  - Was ist binäre Suche?
  - Worst-Case Analyse von Binärer Suche
- 2 Rekursionsgleichungen
  - Fibonacci-Zahlen
  - Ermittlung von Rekursionsgleichungen
- 3 Lösen von Rekursionsgleichungen
  - Die Substitutionsmethode
  - Rekursionsbäume

## Übersicht

- 1 Binäre Suche
  - Was ist binäre Suche?
  - Worst-Case Analyse von Binärer Suche
- 2 Rekursionsgleichungen
  - Fibonacci-Zahlen
  - Ermittlung von Rekursionsgleichungen
- 3 Lösen von Rekursionsgleichungen
  - Die Substitutionsmethode
  - Rekursionsbäume

## Binäre Suche

### Suchen in einem sortierten Array

**Eingabe:** *Sortiertes* Array  $E$  mit  $n$  Einträgen, und das gesuchte Element  $K$ .  
**Ausgabe:** Ist  $K$  in  $E$  enthalten?

### Idee

Da  $E$  sortiert ist, können wir das gesuchte Element  $K$  schneller suchen.  
Liegt  $K$  nicht in der Mitte von  $E$ , dann:

1. suche in der linken Hälfte von  $E$ , falls  $K < E[\text{mid}]$
2. suche in der rechten Hälfte von  $E$ , falls  $K > E[\text{mid}]$

### Fazit:

Wir **halbieren** den Suchraum in jedem Durchlauf.

## Binäre Suche

```

1 bool binSearch(int E[], int n, int K) {
2   int left = 0, right = n - 1;
3   while (left <= right) {
4     int mid = floor((left + right) / 2); // runde ab
5     if (E[mid] == K) { return true; }
6     if (E[mid] > K) { right = mid - 1; }
7     if (E[mid] < K) { left = mid + 1; }
8   }
9   return false;
10 }

```

## Rekursionsgleichung für Binäre Suche

Sei  $S(n)$  die maximale Anzahl der Schleifendurchläufe bei einer erfolglosen Suche.

Wir erhalten die **Rekursionsgleichung**:

$$S(n) = \begin{cases} 0 & \text{falls } n = 0 \\ 1 + S(\lceil (n-1)/2 \rceil) & \text{falls } n > 0 \end{cases}$$

Die ersten Werten sind:

$n$	0	1	2	3	4	5	6	7	8
$S(n)$	0	1	2	2	3	3	3	3	4

Wir haben letztes Mal abgeleitet:  $S(n) = \lfloor \log(n) \rfloor + 1$ .

## Binäre Suche – Analyse

Abkürzungen:  $m = \text{mid}$ ,  $r = \text{right}$ ,  $l = \text{left}$

### Größe des undurchsuchten Arrays

Im nächsten Durchlauf ist die Größe des Arrays  $m - l$  oder  $r - m$ .

Hierbei ist  $m = \lfloor (l + r)/2 \rfloor$ .

Die neue Größe ist also:

- ▶  $m - l = \lfloor (l + r)/2 \rfloor - l = \lfloor (r - l)/2 \rfloor = \lfloor (n - 1)/2 \rfloor$   
oder
- ▶  $r - m = r - \lfloor (l + r)/2 \rfloor = \lceil (r - l)/2 \rceil = \lceil (n - 1)/2 \rceil$

Im schlimmsten Fall ist die neue Größe des Arrays also:

$$\lceil (n - 1)/2 \rceil$$

## Lösen der Rekursionsgleichung

Betrachte den **Spezialfall**  $n = 2^k - 1$ .

Da die maximale neue Größe des Arrays  $\lceil (n-1)/2 \rceil$  ist, leiten wir her:

$$\left\lceil \frac{(2^k - 1) - 1}{2} \right\rceil = \left\lceil \frac{2^k - 2}{2} \right\rceil = \lceil 2^{k-1} - 1 \rceil = 2^{k-1} - 1.$$

Daher gilt für  $k > 0$  nach der Definition  $S(n) = 1 + S(\lceil (n-1)/2 \rceil)$ , dass:

$$S(2^k - 1) = 1 + S(2^{k-1} - 1) \quad \text{und damit} \quad S(2^k - 1) = k + \underbrace{S(2^0 - 1)}_{=0} = k.$$

## Binäre Suche – Analyse

$n$	0	1	2	3	4	5	6	7	8
$S(n)$	0	1	2	2	3	3	3	3	4

Vermutung:  $S(2^k) = 1 + S(2^{k-1})$ .

$S(n)$  steigt monoton, also  $S(n) = k$  falls  $2^{k-1} \leq n < 2^k$ .

Oder: falls  $k - 1 \leq \log(n) < k$ .

Dann ist  $S(n) = \lfloor \log(n) \rfloor + 1$ .

## Binäre Suche – Analyse

### Theorem

Die Worst Case Zeitkomplexität der binären Suche ist  $W(n) = \lfloor \log(n) \rfloor + 1$ .

## Binäre Suche – Analyse

Wir vermuten  $S(n) = \lfloor \log(n) \rfloor + 1$  für  $n > 0$

**Induktion über  $n$ :**

**Basis:**  $S(1) = 1 = \lfloor \log(1) \rfloor + 1$

**Induktionsschritt:** Sei  $n > 1$ . Dann:

$$S(n) = 1 + S(\lceil (n-1)/2 \rceil) = 1 + \lfloor \log(\lceil (n-1)/2 \rceil) \rfloor + 1$$

Man kann zeigen (Hausaufgabe):  $\lfloor \log(\lceil (n-1)/2 \rceil) \rfloor + 1 = \lfloor \log(n) \rfloor$ .

Damit:  $S(n) = \lfloor \log(n) \rfloor + 1$  für  $n > 0$ .

## Übersicht

- 1 Binäre Suche
  - Was ist binäre Suche?
  - Worst-Case Analyse von Binärer Suche
- 2 Rekursionsgleichungen
  - Fibonacci-Zahlen
  - Ermittlung von Rekursionsgleichungen
- 3 Lösen von Rekursionsgleichungen
  - Die Substitutionsmethode
  - Rekursionsbäume

# Rekursionsgleichungen

## Rekursionsgleichung

Für rekursive Algorithmen wird die Laufzeit meistens durch **Rekursionsgleichungen** beschrieben.

Eine **Rekursionsgleichung** ist eine Gleichung oder eine Ungleichung, die eine Funktion durch ihre eigenen Funktionswerte für kleinere Eingaben beschreibt.

## Beispiele

- ▶  $T(n) = T(n-1) + 1$  Lineare Suche
- ▶  $T(n) = T(\lceil (n-1)/2 \rceil) + 1$  Binäre Suche
- ▶  $T(n) = T(n-1) + n - 1$  Bubblesort
- ▶  $T(n) = 2 \cdot T(n/2) + n - 1$  Mergesort
- ▶  $T(n) = 7 \cdot T(n/2) + c \cdot n^2$  Strassen's Matrixmultiplikation

# Naiver, rekursiver Algorithmus

## Rekursiver Algorithmus

```

1 int fibRec(int n) {
2   if (n == 0 || n == 1) {
3     return n;
4   }
5   return fibRec(n - 1) + fibRec(n - 2);
6 }

```

Die zur Berechnung von  $\text{fibRec}(n)$  benötigte Anzahl arithmetischer Operationen  $T_{\text{fibRec}}(n)$  ist durch folgende **Rekursionsgleichung** gegeben:

$$T_{\text{fibRec}}(0) = 0$$

$$T_{\text{fibRec}}(1) = 0$$

$$T_{\text{fibRec}}(n+2) = T_{\text{fibRec}}(n+1) + T_{\text{fibRec}}(n) + 3 \quad \text{für } n \geq 0.$$

Zur Ermittlung der Zeitkomplexitätsklasse von  $\text{fibRec}$  **löst** man diese Gleichung.

# Fibonacci-Zahlen

## Problem

Betrachte das Wachstum einer Kaninchenpopulation:

- ▶ Zu Beginn gibt es ein Paar geschlechtsreifer Kaninchen.
- ▶ Jedes neugeborene Paar wird im zweiten Lebensmonat geschlechtsreif.
- ▶ Jedes geschlechtsreife Paar wirft pro Monat ein weiteres Paar.
- ▶ Sie sterben nie und hören niemals auf.

## Lösung

Die Anzahl der Kaninchenpaare lässt sich wie folgt berechnen:

$$\text{Fib}(0) = 0$$

$$\text{Fib}(1) = 1$$

$$\text{Fib}(n+2) = \text{Fib}(n+1) + \text{Fib}(n) \quad \text{für } n \geq 0.$$

$n$	0	1	2	3	4	5	6	7	8	9	...
$\text{Fib}(n)$	0	1	1	2	3	5	8	13	21	34	...

# Analyse: Anwendung der „Substitutionsmethode“

## Problem

$$T_{\text{fibRec}}(0) = 0$$

$$T_{\text{fibRec}}(1) = 0$$

$$T_{\text{fibRec}}(n+2) = T_{\text{fibRec}}(n+1) + T_{\text{fibRec}}(n) + 3 \quad \text{für } n \geq 0.$$

## Lösung (mittels vollständiger Induktion)

$$T_{\text{fibRec}}(n) = 3 \cdot \text{Fib}(n+1) - 3.$$

## Fakt

$$2^{(n-2)/2} \leq \text{Fib}(n) \leq 2^{n-2} \quad \text{für } n > 1.$$

## Damit ergibt sich:

$$T_{\text{fibRec}}(n) \in \Theta(2^n), \text{ oft abgekürzt dargestellt als } \text{fibRec}(n) \in \Theta(2^n).$$

## Ein iterativer Algorithmus

### Iterativer Algorithmus

```

1 int fibIter(int n) {
2   int f[n];
3   f[0] = 0; f[1] = 1;
4   for (int i = 2; i <= n; i++) {
5     f[i] = f[i-1] + f[i-2];
6   }
7   return f[n];
8 }

```

Die benötigte Anzahl arithmetischer Operationen  $T_{fibIter}(n)$  ist:

$$T_{fibIter}(0) = 0 \quad \text{und} \quad T_{fibIter}(1) = 0$$

$$T_{fibIter}(n+2) = 3 \cdot (n+1) \quad \text{für } n \geq 0.$$

### Damit ergibt sich:

$T_{fibIter}(n) \in \Theta(n)$ , oder als Kurzschreibweise  $fibIter(n) \in \Theta(n)$ .

## Ein iterativer Algorithmus (2)

Jedoch: der fibIter Algorithmus hat eine Speicherkomplexität in  $\Theta(n)$ .

Beobachtung: jeder Durchlauf "benutzt" nur die Werte  $f[i-1]$  und  $f[i-2]$ .

Zwei Variablen reichen also aus, um diese Werte zu speichern.

### Iterativer Algorithmus

```

1 int fibIter2(int n) {
2   int a = 0; int b = 1;
3   for (int i = 2; i <= n; i++) {
4     c = a + b;
5     a = b;
6     b = c;
7   }
8   return b;
9 }

```

Der fibIter2 Algorithmus hat eine Speicherkomplexität in  $\Theta(1)$  und  $T_{fibIter2}(n) \in \Theta(n)$ .

## Ein Matrixpotenz-Algorithmus

### Matrixdarstellung der Fibonacci-Zahlen

Es gilt für  $n > 0$ :

$$\begin{pmatrix} Fib(n+2) \\ Fib(n+1) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} Fib(n+1) \\ Fib(n) \end{pmatrix}$$

Damit lässt sich  $Fib(n+2)$  durch Matrixpotenzierung berechnen:

$$\begin{pmatrix} Fib(n+2) \\ Fib(n+1) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^2 \cdot \begin{pmatrix} Fib(n) \\ Fib(n-1) \end{pmatrix} = \dots = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \cdot \begin{pmatrix} Fib(2) \\ Fib(1) \end{pmatrix}$$

- ▶ Wie können wir Matrixpotenzen effizient berechnen?
- ▶ Dies betrachten wir hier nicht im Detail; geht in  $\Theta(\log(n))$

## Praktische Konsequenzen

### Beispiel

Größte lösbare Eingabelänge für angenommene 1  $\mu$ s pro Operation:

Verfügbare Zeit	Rekursiv	Iterativ	Matrix
1 ms	14	500	$10^{12}$
1 s	28	$5 \cdot 10^5$	$10^{12\,000}$
1 m	37	$3 \cdot 10^7$	$10^{700\,000}$
1 h	45	$1,8 \cdot 10^9$	$10^{10^6}$

Lösbare Eingabelänge

Vereinfachende Annahmen:

- ▶ Nur arithmetische Operationen wurden berücksichtigt.
- ▶ Die Laufzeit der arithmetischen Operationen ist fix, also nicht von ihren jeweiligen Argumenten abhängig.

## Rekursionsgleichungen von Programmcode ableiten

### Rekursionsgleichung im Worst-Case

Zur Ermittlung der Worst-Case Laufzeit  $T(n)$  zerlegen wir das Programm:

- ▶ Die Kosten aufeinanderfolgender Blöcke werden **addiert**.
- ▶ Von alternativen Blöcken wird das **Maximum** genommen.
- ▶ Beim Aufruf von Unterprogrammen (etwa `sub1()`) wird  $T_{sub1}(f(n))$  genommen, wobei  $f(n)$  die Länge der Parameter beim Funktionsaufruf —abhängig von der Eingabelänge  $n$  des Programms— ist.
- ▶ Rekursive Aufrufe werden mit  $T(g(n))$  veranschlagt;  $g(n)$  gibt wieder die von  $n$  abgeleitete Länge der Aufrufparameter an.

## Einige Vereinfachungen

- ▶ Wenn wir Rekursionsgleichungen aufstellen und lösen, vernachlässigen wir häufig **das Runden** auf ganze Zahlen, z. B.:

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 3 \quad \text{wird} \quad T(n) = 2T(n/2) + 3.$$

- ▶ Manchmal wird angenommen, daß  $T(n)$  **für kleine  $n$  konstant ist** anstatt genau festzustellen was  $T(0)$  und  $T(1)$  ist. Also z. B.:

$$T(0) = c \text{ und } T(1) = c' \quad \text{statt} \quad T(0) = 4 \text{ und } T(1) = 7.$$

- ▶ Wir nehmen an, dass die Funktionen nur **ganzzahlige** Argumente haben, z. B.:

$$T(n) = T(\sqrt{n}) + n \quad \text{bedeutet} \quad T(n) = T(\lfloor \sqrt{n} \rfloor) + n.$$

- ▶ **Grund:** die Lösung wird typischerweise nur um einen konstanten Faktor verändert, aber **der Wachstumsgrad** bleibt unverändert.

## Übersicht

- 1 Binäre Suche
  - Was ist binäre Suche?
  - Worst-Case Analyse von Binärer Suche
- 2 Rekursionsgleichungen
  - Fibonacci-Zahlen
  - Ermittlung von Rekursionsgleichungen
- 3 Lösen von Rekursionsgleichungen
  - Die Substitutionsmethode
  - Rekursionsbäume

## Lösen von Rekursionsgleichungen

### Einfache Fälle

Für einfache Fälle gibt es **geschlossene** Lösungen, z. B. für  $k, c \in \mathbb{N}$ :

$$T(0) = k$$

$$T(n+1) = c \cdot T(n) \quad \text{für } n \geq 0$$

hat die eindeutige Lösung  $T(n) = c^n \cdot k$ .

Und die Rekursionsgleichung:

$$T(0) = k$$

$$T(n+1) = T(n) + f(n) \quad \text{für } n \geq 0$$

hat die eindeutige Lösung  $T(n) = T(0) + \sum_{i=1}^n f(i)$ .

Bei der Zeitkomplexitätsanalyse treten solche Fälle jedoch **selten** auf.

## Lösen von Rekursionsgleichungen

### Allgemeines Format der Rekursionsgleichung

Im allgemeinen Fall – **der hier häufig auftritt** – gibt es keine geschlossene Lösung.

Der typische Fall sieht folgendermaßen aus:

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

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

### Intuition:

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

## Die Substitutionsmethode: Beispiel

### Beispiel

Betrachte folgende Rekursionsgleichung:

$$T(1) = 1$$

$$T(n) = 2 \cdot T(n/2) + n \quad \text{für } n > 1.$$

- ▶ Wir vermuten als Lösung  $T(n) \in O(n \cdot \log(n))$ .
- ▶ Dazu müssen wir  $T(n) \leq c \cdot n \cdot \log(n)$  zeigen, für ein geeignetes  $c > 0$ .
- ▶ Bestimme, ob für ein geeignetes  $n_0$  und für  $n \geq n_0$  gilt, dass  $T(n) \leq c \cdot n \cdot \log(n)$ .
- ▶ Stelle fest, dass  $T(1) = 1 \leq c \cdot 1 \cdot \log(1) = 0$  **verletzt** ist.
- ▶ Es gilt:  $T(2) = 4 \leq c \cdot 2 \log(2)$  und  $T(3) = 5 \leq c \cdot 3 \log(3)$  für  $c \geq 2$
- ▶ **Überprüfe** dann durch Substitution und Induktion (s. nächste Folie)
- ▶ Damit gilt für jedes  $c \geq 2$  und  $n \geq n_0 > 1$ , dass  $T(n) \leq c \cdot n \cdot \log(n)$ .

## Die Substitutionsmethode

### Substitutionsmethode

Die Substitutionsmethode besteht aus zwei Schritten:

1. **Rate** die Form der Lösung, durch z. B.:
  - ▶ Scharfes Hinsehen, kurze Eingaben ausprobieren und einsetzen
  - ▶ Betrachtung des Rekursionsbaums
2. **Vollständige Induktion**, um die Konstanten zu finden und zu zeigen, dass die Lösung funktioniert.

### Einige Hinweise

- ▶ Diese Methode ist sehr leistungsfähig, aber
- ▶ kann nur in den Fällen angewendet werden, in denen es relativ einfach ist, die Form der Lösung zu erraten.

## Die Substitutionsmethode: Beispiel

### Beispiel

$$T(n) = 2 \cdot T(n/2) + n \quad \text{für } n > 1, \text{ und } T(1) = 1$$

$$\begin{aligned} T(n) &= 2 \cdot T(n/2) + n && | \text{ Induktionshypothese} \\ &\leq 2(c \cdot n/2 \cdot \log(n/2)) + n \\ &= c \cdot n \cdot \log(n/2) + n && | \text{ log-Rechnung: } (\log \equiv \log_2) \\ & && | \log(n/2) = \log(n) - \log(2) \\ &= c \cdot n \cdot \log(n) - c \cdot n \cdot \log(2) + n \\ &\leq c \cdot n \cdot \log(n) - c \cdot n + n && | \text{ mit } c > 1 \text{ folgt sofort:} \\ &\leq c \cdot n \cdot \log(n) \end{aligned}$$

## Die Substitutionsmethode: Feinheiten

### Einige wichtige Hinweise

- Die asymptotische Schranke ist korrekt erraten, kann aber manchmal nicht mittels vollständiger Induktion bewiesen werden.  
Das Problem ist gewöhnlich, dass die Induktionsannahme **nicht streng genug** ist.
- Manchmal ist eine Variablentransformation hilfreich, um zu einer Lösung zu geraten:

## Die Substitutionsmethode

### Substitutionsmethode

Die Substitutionsmethode besteht aus zwei Schritten:

- Rate** die Form der Lösung, durch z. B.:
  - Scharfes Hinsehen, kurze Eingaben ausprobieren und einsetzen
  - Betrachtung des Rekursionsbaums
- Vollständige Induktion**, um die Konstanten zu finden und zu zeigen, dass die Lösung funktioniert.

Wir betrachten nun detaillierter, wie man die Form der Lösung raten kann.

## Die Substitutionsmethode: Variablentransformation

### Beispiel

$$T(n) = 2 \cdot T(\sqrt{n}) + \log(n) \text{ für } n > 0$$

$$T(n) = 2 \cdot T(\sqrt{n}) + \log(n) \quad | \text{ Variablentransformation } m = \log(n)$$

$$\Leftrightarrow T(2^m) = 2 \cdot T(2^{m/2}) + m \quad | \text{ Umbenennung } T(2^m) = S(m)$$

$$\Leftrightarrow S(m) = 2 \cdot S(m/2) + m \quad | \text{ Lösung des vorherigen Beispiels}$$

$$\Leftrightarrow S(m) \leq c \cdot m \cdot \log(m)$$

$$\Leftrightarrow S(m) \in O(m \cdot \log(m)) \quad | m = \log(n)$$

$$\Leftrightarrow T(n) \in O(\log(n) \cdot \log(\log(n)))$$

## Raten der Lösung durch Iteration

### Grundidee

Wiederholtes Einsetzen der Rekursionsgleichung in sich selbst, bis man ein Muster erkennt.

### Beispiel

$$T(n) = 3 \cdot T(n/4) + n \quad | \text{ Einsetzen}$$

$$= 3 \cdot (3 \cdot T(n/16) + n/4) + n \quad | \text{ Nochmal einsetzen}$$

$$= 9 \cdot (3 \cdot T(n/64) + n/16) + 3 \cdot n/4 + n \quad | \text{ Vereinfachen}$$

$$= 27 \cdot T(n/64) + \left(\frac{3}{4}\right)^2 \cdot n + \left(\frac{3}{4}\right)^1 \cdot n + \left(\frac{3}{4}\right)^0 \cdot n$$

Wir nehmen  $T(1) = c$  an und erhalten:  $T(n) = \sum_{i=0}^{\log_4(n)-1} \left(\frac{3}{4}\right)^i \cdot n + c \cdot n^{\log_4(3)}$

Diese Aussage kann mit Hilfe der Substitutionsmethode gezeigt werden.

# Raten der Lösung durch Rekursionsbäume

## Grundidee

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

## Rekursionsbaum

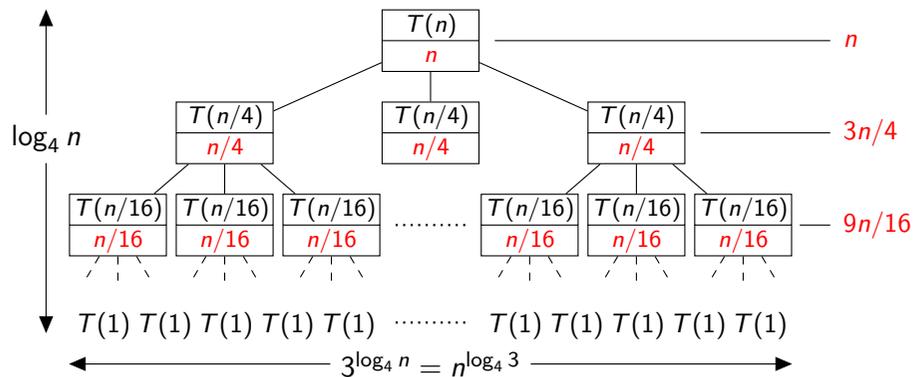
- Jeder **Knoten** stellt die Kosten eines Teilproblems dar.
  - Die Wurzel stellt die zu analysierenden Kosten  $T(n)$  dar.
  - Die Blätter stellen die Kosten der Basisfälle dar, z. B.  $T(0)$  oder  $T(1)$ .
- Wir summieren die Kosten innerhalb jeder **Ebene** des Baumes.
- Die **Gesamtkosten** :=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

## Wichtiger 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

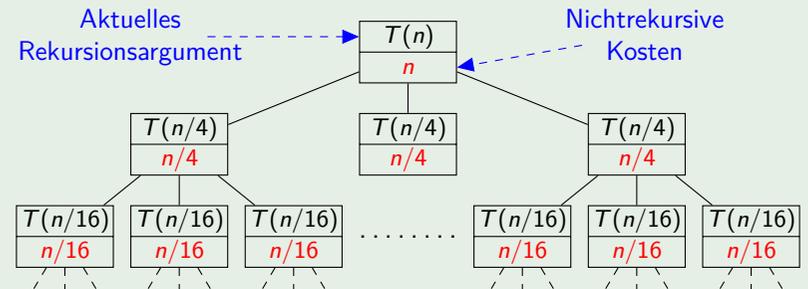


$$T(n) = \underbrace{\sum_{i=0}^{\log_4(n)-1} \left(\frac{3}{4}\right)^i \cdot n}_{\text{Summe über alle Ebenen}} + \underbrace{c \cdot n^{\log_4(3)}}_{\text{Gesamtkosten für die Blätter mit } T(1) = c}$$

# Rekursionsbaum: Beispiel

## Beispiel

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



# Rekursionsbaum: Beispiel

Eine obere Schranke für die Komplexität erhält man nun folgendermaßen:

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

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

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

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

$$T(n) \in O(n).$$

## Korrektheit

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

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

$$\begin{aligned} T(n) &= 3 \cdot T(n/4) + n && | \text{ Induktionshypothese} \\ &\leq 3d \cdot n/4 + n \\ &= \frac{3}{4}d \cdot n + n \\ &= \left(\frac{3}{4}d + 1\right) \cdot n && | \text{ mit } d \geq 4 \text{ folgt sofort:} \\ &\leq d \cdot n \end{aligned}$$

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

## Nächste Vorlesung

### Nächste Vorlesung

Montag 7. Mai, 08:30 (Hörsaal H01). Bis dann!

### Nächste Frontalübung

Freitag 4. Mai, 13:15 (Hörsaal H01) statt am 1. Mai.