Datenstrukturen und Algorithmen

Vorlesung 5: Rekursionsgleichungen (K4)

Joost-Pieter Katoen

Lehrstuhl für Informatik 2
Software Modeling and Verification Group

http://www-i2.rwth-aachen.de/i2/dsal10/

30. April 2010,  Königinnentag

Übersicht

Übersicht

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?

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.

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:

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 $\texttt{K} < \texttt{E[mid]}$

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

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[mid]`
2. suche in der rechten Hälfte von `E`, falls `K` > `E[mid]`

Fazit:

Wir halbieren den Suchraum in jedem Durchlauf.

Binäre Suche – Beispiel

Binäre Suche

```
bool binSearch(int E[], int n, int K) {
   int left = 0, right = n - 1;
   while (left <= right) {
     int mid = floor((left + right) / 2); // runde ab
     if (E[mid] == K) { return true; }
     if (E[mid] > K) { right = mid - 1; }
     if (E[mid] < K) { left = mid + 1; }
   }
   return false;
}
```

Binäre Suche – Analyse

Lemma

Sei $r \in \mathbb{R}$ und $n \in \mathbb{N}$. Dann gilt:

Binäre Suche – Analyse

Lemma

Sei $r \in \mathbb{R}$ und $n \in \mathbb{N}$. Dann gilt:

1. $\lfloor r + n \rfloor \;=\; \lfloor r \rfloor + n$

Binäre Suche – Analyse

Lemma

Sei $r \in \mathbb{R}$ und $n \in \mathbb{N}$. Dann gilt:

1. $\lfloor r + n \rfloor \; = \; \lfloor r \rfloor + n$
2. $\lceil r + n \rceil \; = \; \lceil r \rceil + n$

Binäre Suche – Analyse

Lemma

Sei $r \in \mathbb{R}$ und $n \in \mathbb{N}$. Dann gilt:

1. $\lfloor r + n \rfloor \ = \ \lfloor r \rfloor + n$
2. $\lceil r + n \rceil \ = \ \lceil r \rceil + n$
3. $\lfloor -r \rfloor \ = \ -\lceil r \rceil$

Binäre Suche – Analyse

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

Binäre Suche – Analyse

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

Größe des undurchsuchten Arrays

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

Binäre Suche – Analyse

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

Größe des undurchsuchten Arrays

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

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

Binäre Suche – Analyse

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

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:

- $m - l \;=\; \lfloor (l + r)/2 \rfloor - l \;=\; \lfloor (r - l)/2 \rfloor \;=\; \lfloor (n - 1)/2 \rfloor$

Binäre Suche – Analyse

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

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

Binäre Suche – Analyse

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

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

Binäre Suche – Analyse

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

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:

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

Rekursionsgleichung für Binäre Suche

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

Rekursionsgleichung für Binäre Suche

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

Wir erhalten die Rekursionsgleichung:

Rekursionsgleichung für Binäre Suche

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

Wir erhalten die Rekursionsgleichung:

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

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

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 suchen eine geschlossene Formel für $S(n)$

Lösen der Rekursionsgleichung

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

Lösen der Rekursionsgleichung

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

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

Lösen der Rekursionsgleichung

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

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

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

Lösen der Rekursionsgleichung

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

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

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

Lösen der Rekursionsgleichung

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

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

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

Lösen der Rekursionsgleichung

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

Da die maximale 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:

$S(2^k{-}1) \; = \; 1{+}S(2^{k-1}{-}1)$

Lösen der Rekursionsgleichung

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

Da die maximale 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:

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

Lösen der Rekursionsgleichung

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

Da die maximale 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:

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

Binäre Suche – Analyse

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

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

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 $2^{k-1} \leqslant n < 2^k$.

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 $2^{k-1} \leqslant n < 2^k$.

Oder: falls $k - 1 \leqslant \log n < k$.

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 $2^{k-1} \leqslant n < 2^k$.

Oder: falls $k - 1 \leqslant \log n < k$.

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

Binäre Suche – Analyse

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

Binäre Suche – Analyse

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

Induktion über n:

Binäre Suche – Analyse

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

Induktion über n:

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

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:

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:

$$S(n) \; = \; 1 + S(\lceil (n-1)/2 \rceil) \; =$$

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:

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

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

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 $n > 0$.

Binäre Suche – Analyse

Theorem

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

Vergleich der Suchalgorithmen

Algorithmus	Zeitkomplexität	Vorteil	Nachteil
Lineare Suche	$O(n)$	einfach	langsam
Bilineare Suche	$O(n)$	einfach / elegant	langsam
Binäre Suche	$O(\log n)$	schnell	sortiertes Array ($O(n \cdot \log n)$ Initialisierungsaufwand)

Übersicht

Rekursionsgleichungen

Rekursionsgleichung

Für rekursive Algorithmen wird die Laufzeit meistens durch Rekursionsgleichungen beschrieben.

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.

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

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.*

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:

$$\begin{aligned} Fib(0) &= 0 \\ Fib(1) &= 1 \\ Fib(n+2) &= Fib(n+1) + Fib(n) \qquad \text{für } n \geqslant 0. \end{aligned}$$

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:

$$\begin{aligned} Fib(0) &= 0 \\ Fib(1) &= 1 \\ Fib(n+2) &= Fib(n+1) + Fib(n) \qquad \text{für } n \geqslant 0. \end{aligned}$$

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

Naiver, rekursiver Algorithmus

Rekursiver Algorithmus

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

Naiver, rekursiver Algorithmus

Rekursiver Algorithmus

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

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

$$
\begin{aligned}
T_{fibRec}(0) &= 0 \\
T_{fibRec}(1) &= 0 \\
T_{fibRec}(n+2) &= T_{fibRec}(n+1) + T_{fibRec}(n) + 3 \quad \text{für } n \geqslant 0.
\end{aligned}
$$

Naiver, rekursiver Algorithmus

Rekursiver Algorithmus

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

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

$$
\begin{aligned}
T_{fibRec}(0) &= 0 \\
T_{fibRec}(1) &= 0 \\
T_{fibRec}(n+2) &= T_{fibRec}(n+1) + T_{fibRec}(n) + 3 \qquad \text{für } n \geqslant 0.
\end{aligned}
$$

Zur Ermittlung der Zeitkomplexitätsklasse von `fibRec` löst man diese Gleichung.

Analyse: Anwendung der „Substitutionsmethode“

Problem

$$T_{fibRec}(0) = 0$$
$$T_{fibRec}(1) = 0$$
$$T_{fibRec}(n+2) = T_{fibRec}(n+1) + T_{fibRec}(n) + 3 \qquad \textit{für } n \geqslant 0.$$

Rekursionsgleichungen

Analyse: Anwendung der „Substitutionsmethode“

Problem

$$
\begin{aligned}
T_{fibRec}(0) &= 0\\
T_{fibRec}(1) &= 0\\
T_{fibRec}(n+2) &= T_{fibRec}(n+1) + T_{fibRec}(n) + 3 \qquad \text{für } n \geqslant 0.
\end{aligned}
$$

Lösung (mittels vollständiger Induktion)

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

Analyse: Anwendung der „Substitutionsmethode“

Problem

$$
\begin{aligned}
T_{fibRec}(0) &= 0 \\
T_{fibRec}(1) &= 0 \\
T_{fibRec}(n+2) &= T_{fibRec}(n+1) + T_{fibRec}(n) + 3 \qquad \text{für } n \geqslant 0.
\end{aligned}
$$

Lösung (mittels vollständiger Induktion)

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

Fakt

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

Analyse: Anwendung der „Substitutionsmethode“

Problem

$$
\begin{aligned}
T_{fibRec}(0) &= 0 \\
T_{fibRec}(1) &= 0 \\
T_{fibRec}(n+2) &= T_{fibRec}(n+1) + T_{fibRec}(n) + 3 \qquad \textit{für } n \geqslant 0.
\end{aligned}
$$

Lösung (mittels vollständiger Induktion)

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

Fakt

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

Damit ergibt sich:

$T_{fibRec}(n) \in \Theta(2^n)$,

Analyse: Anwendung der „Substitutionsmethode“

Problem

$$
\begin{aligned}
T_{fibRec}(0) &= 0 \\
T_{fibRec}(1) &= 0 \\
T_{fibRec}(n+2) &= T_{fibRec}(n+1) + T_{fibRec}(n) + 3 \qquad \textit{für } n \geqslant 0.
\end{aligned}
$$

Lösung (mittels vollständiger Induktion)

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

Fakt

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

Damit ergibt sich:

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

Ein iterativer Algorithmus

Iterativer Algorithmus

```
int fibIter(int n) {
  int f[n];
  f[0] = 0; f[1] = 1;
  for (int i = 2; i <= n; i++) {
      f[i] = f[i-1] + f[i-2];
      }
  return f[n];
}
```

Ein iterativer Algorithmus

Iterativer Algorithmus

```
int fibIter(int n) {
  int f[n];
  f[0] = 0; f[1] = 1;
  for (int i = 2; i <= n; i++) {
      f[i] = f[i-1] + f[i-2];
      }
  return f[n];
}
```

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

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

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

Ein iterativer Algorithmus

Iterativer Algorithmus

```
int fibIter(int n) {
  int f[n];
  f[0] = 0; f[1] = 1;
  for (int i = 2; i <= n; i++) {
      f[i] = f[i-1] + f[i-2];
      }
  return f[n];
}
```

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

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

Damit ergibt sich:

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

Ein iterativer Algorithmus (2)

Jedoch: der `fibIter` Algorithmus hat eine Speicherkomplexität 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]`.

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.

 Rekursionsgleichungen

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

```
int fibIter2(int n) {
  int a = 0; int b = 1;
  for (int i = 2; i <= n; i++) {
    c = a + b;
    a = b;
    b = c;
  }
  return b;
}
```

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

```
int fibIter2(int n) {
  int a = 0; int b = 1;
  for (int i = 2; i <= n; i++) {
    c = a + b;
    a = b;
    b = c;
  }
  return b;
}
```

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 $n > 0$:

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

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:

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

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:

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

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

Praktische Konsequenzen

Beispiel

Größte lösbare Eingabelänge für angenommene 1 µs 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 unabhängig.

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.

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.

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.

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.

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

Einige Vereinfachungen

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

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

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.:

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

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.:

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

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 Wachstumgrad bleibt unverändert.

Lösen von Rekursionsgleichungen

Einfache Fälle

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

$$
\begin{aligned}
T(0) &= k \\
T(n{+}1) &= c \cdot T(n) \qquad \text{für } n \geqslant 0
\end{aligned}
$$

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

Lösen von Rekursionsgleichungen

Einfache Fälle

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

$$\begin{aligned} T(0) &= k \\ T(n{+}1) &= c \cdot T(n) \qquad \text{für } n \geqslant 0 \end{aligned}$$

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

Und die Rekursionsgleichung:

$$\begin{aligned} T(0) &= k \\ T(n{+}1) &= T(n) + f(n) \qquad \text{für } n \geqslant 0 \end{aligned}$$

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

Lösen von Rekursionsgleichungen

Einfache Fälle

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

$$\begin{aligned} T(0) &= k \\ T(n{+}1) &= c \cdot T(n) \quad \text{für } n \geqslant 0 \end{aligned}$$

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

Und die Rekursionsgleichung:

$$\begin{aligned} T(0) &= k \\ T(n{+}1) &= T(n) + f(n) \quad \text{für } n \geqslant 0 \end{aligned}$$

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

Allgemeine Format der Rekursionsgleichung

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

Lösen von Rekursionsgleichungen

Allgemeine Format der Rekursionsgleichung

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

Der typischer 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.

Lösen von Rekursionsgleichungen

Allgemeine Format der Rekursionsgleichung

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

Der typischer 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

Lösen von Rekursionsgleichungen

Allgemeine Format der Rekursionsgleichung

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

Der typischer 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 $\frac{n}{c}$

Lösen von Rekursionsgleichungen

Allgemeine Format der Rekursionsgleichung

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

Der typischer 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

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 Rekursionsbaum

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 Rekursionsbaum
2. Vollständige Induktion um die Konstanten zu finden und zu zeigen, dass die Lösung funktioniert.

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 Rekursionsbaum
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 angewendet werden in den Fallen in denen es relativ einfach ist, die Form der Lösung zu erraten.

Die Substitutionsmethode: Beispiel

Beispiel

Betrachte folgende Rekursionsgleichung:

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

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

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 $T(n) \leqslant c \cdot n \cdot \log n$ zeigen, für geeignete $c > 0$.

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 $T(n) \leqslant c \cdot n \cdot \log n$ zeigen, für geeignete $c > 0$.
- Bestimme ob für eine geeignete n_0, für $n \geqslant n_0$, $T(n) \leqslant c \cdot n \cdot \log n$ gilt.

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 $T(n) \leqslant c \cdot n \cdot \log n$ zeigen, für geeignete $c > 0$.
- Bestimme ob für eine geeignete n_0, für $n \geqslant n_0$, $T(n) \leqslant c \cdot n \cdot \log n$ gilt.
- Stelle fest, dass $T(1) = 1 \leqslant c \cdot 1 \cdot \log 1 = 0$ verletzt ist.

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 $T(n) \leqslant c \cdot n \cdot \log n$ zeigen, für geeignete $c > 0$.
- Bestimme ob für eine geeignete n_0, für $n \geqslant n_0$, $T(n) \leqslant c \cdot n \cdot \log n$ gilt.
- Stelle fest, dass $T(1) = 1 \leqslant c \cdot 1 \cdot \log 1 = 0$ verletzt ist.
- Es gilt: $T(2) = 4 \leqslant c \cdot 2 \log 2$ und $T(3) = 5 \leqslant c \cdot 3 \log 3$ für $c > 1$

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 $T(n) \leqslant c \cdot n \cdot \log n$ zeigen, für geeignete $c > 0$.
- Bestimme ob für eine geeignete n_0, für $n \geqslant n_0$, $T(n) \leqslant c \cdot n \cdot \log n$ gilt.
- Stelle fest, dass $T(1) = 1 \ \leqslant \ c \cdot 1 \cdot \log 1 = 0$ verletzt ist.
- Es gilt: $T(2) = 4 \leqslant c \cdot 2 \log 2$ und $T(3) = 5 \leqslant c \cdot 3 \log 3$ für $c > 1$
- Überprüfe dann durch Substitution und Induktion (s. nächste Folie)

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 $T(n) \leqslant c{\cdot}n{\cdot} \log n$ zeigen, für geeignete $c > 0$.
- Bestimme ob für eine geeignete n_0, für $n \geqslant n_0$, $T(n) \leqslant c{\cdot}n{\cdot} \log n$ gilt.
- Stelle fest, dass $T(1) = 1 \ \leqslant \ c{\cdot}1{\cdot} \log 1 = 0$ verletzt ist.
- Es gilt: $T(2) = 4 \leqslant c{\cdot}2 \log 2$ und $T(3) = 5 \leqslant c{\cdot}3 \log 3$ für $c > 1$
- Überprüfe dann durch Substitution und Induktion (s. nächste Folie)
- Damit gilt für jedes $c > 1$ und $n \geqslant n_0 > 1$, dass $T(n) \leqslant c{\cdot}n{\cdot} \log n$.

Die Substitutionsmethode: Beispiel

Beispiel

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

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

Die Substitutionsmethode: Feinheiten

Einige wichtige Hinweise

1. Die asymptotische Schranke ist korrekt erraten, kann aber manchmal nicht mit der vollständigen Induktion bewiesen werden.

   Das Problem ist gewöhnlich, dass die Induktionsannahme nicht streng genug ist.

Die Substitutionsmethode: Feinheiten

Einige wichtige Hinweise

1. Die asymptotische Schranke ist korrekt erraten, kann aber manchmal nicht mit der vollständigen Induktion bewiesen werden.

   Das Problem ist gewöhnlich, dass die Induktionsannahme nicht streng genug ist.

2. Manchmal ist eine Variablentransformation vernünftig um zu einer Lösung zu geraten:

Die Substitutionsmethode: Variablentransformation

Beispiel

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

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