Algorithmen und Datenstrukturen
Kapitel 2:
Korrektheit von Algorithmen und Laufzeitanalyse rekursiver Algorithmen (mittels Rekurrenzgleichungen)

Frank Heitmann
heitmann@informatik.uni-hamburg.de

21. Oktober 2015

Algorithmenanalyse

Um **Algorithmen zu bewerten** beschäftigen wir uns insb. mit dem **Zeit- und Platzbedarf** eines Algorithmus.

Wir behandeln nachfolgend *Zeit-* und *Platzkomplexität*. Eine (elementare) Anweisung zählt dabei als eine Zeiteinheit. Eine benutzte (elemantare) Variable als eine Platzeinheit.

Zeit- und Platzbedarf wird *abhängig von der Länge/der Kenngröße n der Eingabe* gezählt. Wir arbeiten hierfür mit Funktionen $f : \mathbb{N} \to \mathbb{N}$ bzw. $f : \mathbb{N} \to \mathbb{R}$.

O-Notation - Definition

Definition (O-Notation (und Verwandte) - Die wichtigen)

- $O(g(n)) = \{f \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : |f(n)| \leq c \cdot |g(n)|\}$
- $\Omega(g(n)) = \{f \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : |f(n)| \geq c \cdot |g(n)|\}$
- $\Theta(g(n)) = \{f \mid \exists c_1, c_2 \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : c_1 \cdot |g(n)| \leq |f(n)| \leq c_2 \cdot |g(n)|\}$

Definition (O-Notation (und Verwandte) - Die nicht ganz so wichtigen)

- $o(g(n)) = \{f \mid \forall c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : |f(n)| \leq c \cdot |g(n)|\}$
- $\omega(g(n)) = \{f \mid \forall c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : |f(n)| \geq c \cdot |g(n)|\}$

Bemerkung

f ist dabei stets eine Funktion von $\mathbb{N}$ nach $\mathbb{R}$, also $f : \mathbb{N} \to \mathbb{R}$.

O-Notation - Variante

Satz

1. $f(n) \in O(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$
2. $f(n) \in \Omega(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$
3. $f(n) \in \Theta(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = c \in \mathbb{R}^+$
4. $f(n) \in o(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
5. $f(n) \in \omega(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

Ergänzung: Der Satz von L'Hospital

Satz von L'Hospital

Seien f, g differenzierbar und sei $g'(n) \neq 0$ für alle n. Gehen $f(n)$ und $g(n)$ beide gegen 0 oder beide gegen Unendlich, wenn n gegen Unendlich geht, so gilt:

1. Wenn $\lim_{n\to\infty}\left(\frac{f'(n)}{g'(n)}\right)$ existiert, dann existiert auch $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right)$ und
2. $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = \lim_{n\to\infty}\left(\frac{f'(n)}{g'(n)}\right)$

Hinweis

Für unsere Funktion ist der Satz praktisch immer anwendbar. Siehe auch den kleinen Text auf der Webseite.

Literaturhinweis

Genauere Formulierung nachzulesen in Königsberger, Konrad: *Analysis 1.* Springer, 6. Auflage, 2004.

Ergänzung: Der Satz von L'Hospital

Beispiel

Als Beispiel wollen wir

$$\ln(n) \in O(\sqrt[3]{n})$$

unter Nutzung des Satzes von l'Hospital beweisen. Es folgt:

$$\lim_{n\to\infty}\left(\frac{\ln(n)}{\sqrt[3]{n}}\right)$$
$$= \lim_{n\to\infty}\left(\frac{3n^{\frac{2}{3}}}{n}\right) = \lim_{n\to\infty}\left(\frac{3}{n^{\frac{1}{3}}}\right) = \lim_{n\to\infty}\left(\frac{3}{\sqrt[3]{n}}\right) = 0.$$

(Mit $\sqrt[3]{n} = n^{\frac{1}{3}}$ und den Ableitungen $\ln'(n) = \frac{1}{n}$ und $(n^{\frac{1}{3}})' = \frac{1}{3}n^{-\frac{2}{3}} = \frac{1}{3n^{\frac{2}{3}}}$.)
Damit ist die Größenabschätzung (und sogar $\ln n \in o(\sqrt[3]{n})$) bewiesen.

Merkhilfe

Bemerkung

Eine kleine *Merkhilfe* (nicht mehr!)

- $f \in O(g) \approx f \leq g$
- $f \in \Omega(g) \approx f \geq g$
- $f \in \Theta(g) \approx f = g$
- $f \in o(g) \approx f < g$
- $f \in \omega(g) \approx f > g$

Wir sagen, dass f asymptotisch kleiner gleich, größer gleich, gleich, kleiner bzw. größer ist als g, wenn $f \in O(g), f \in \Omega(g), f \in \Theta(g), f \in o(g)$ bzw. $f \in \omega(g)$ gilt.

Vorgehen

Vorgehen, wenn wir Laufzeit/Platzbedarf eines Algorithmus analysieren:

1. Überlegen bzgl. welcher *Kenngröße* der Eingabe wir messen wollen. Diese kann sich von der Größe der Eingabe unterscheiden! (Beispiel: Anzahl Knoten eines Graphen).
2. Laufzeit/Speicherbedarf bzgl. dieser Kenngröße ausdrücken.
3. Nochmal überlegen, ob dies die Aussage des Ergebnisses verfälscht (so wie bei der Fakultätsberechnung oben).

⇒ I.A. sollte sich die eigentliche Eingabegröße leicht durch die Kenngröße ausdrücken lassen und der Unterschied sollte nicht zu groß sein.

- \+ Beim Graphen mit n Knoten ist die Adjazenmatrix in $O(n^2)$.
- – Ist eine Zahl n die Eingabe, so ist die Eingabegröße in $O(\log n)$. Eine Laufzeit von $O(n)$ wäre also exponentiell in der Eingabe!

Suchen

Definition (Das Suchproblem)

Eingabe: Eine Menge $\{a_1, a_2, \ldots, a_n\}$ von n Objekten und ein Objekt p alle vom gleichen Typ.
Gesucht: Gibt es ein a_i mit $a_i = p$?

Anmerkung

Es ist hier nicht genau spezifiziert, wie die Menge von Objekten gegeben ist, d.h. in welcher Datenstruktur sie abgelegt sind. Ebenso ist nicht angegeben, ob eine Ordnung auf den Objekten existiert.

Ein Suchalgorithmus kann dies (beides) ggf. ausnutzen!

Lineare Suche

Algorithmus 1 Lineare Suche

```
1: for i = 0 to n do
2:     if a[i] == max mustermann then
3:         return true
4:     end if
5: end for
6: return false
```


Binäre Suche

Algorithmus 2 Binäre Suche

```
1: while first ≤ last ∧ idx < 0 do
2:     m = first + ((last − first)/2)
3:     if a[m] < p then
4:         first = m + 1
5:     else if a[m] > p then
6:         last = m − 1
7:     else
8:         idx = m
9:     end if
10: end while
11: return idx
```


Zur Binären Suche

Anmerkung

Die lineare Suche stellt sehr wenige/keine Anforderungen.
Bei der binären Suche:

- müssen die Elemente bereits sortiert vorliegen,
- es muss eine Ordnung ($<$, $>$ etc.) existieren und
- ein direkter Zugriff auf einzelne Elemente ist nötig (d.h. das Element $a[m]$ muss direkt zugreifbar sein).

Sortieren

Definition (Das Sortierproblem)

Eingabe: Eine Sequenz $< a_1, a_2, \ldots, a_n >$ von n Zahlen.
Gesucht: Eine Permutation $< a'_1, a'_2, \ldots, a'_n >$ der Eingabesequenz mit $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Anmerkung

Ob die Reihenfolge zweier Elemente a_i, a_j $(i < j)$ mit $a_i = a_j$ beibehalten werden soll oder nicht, ist nicht gesagt. Bleibt sie erhalten, d.h. ist $a'_p = a_i$, $a'_q = a_j$ und $p < q$, so nennt man das Verfahren *stabil*.
Ferner heißt ein Sortierverfahren *in-place* (oder *in situ*), wenn der zusätzlich benötigte Speicherbedarf unabhängig von der gegebenen Sequenz ist.

Sortieren mit Maximumbestimmung (MaxSort)

Algorithmus 3 Sortieren mit Max

```
1: for i = n downto 1 do
2:     idx = max(A)
3:     B[i] = A[idx]
4:     A[idx] = 0
5: end for
6: return B
```

Algorithmus 4 Finde Maximum

```
1: max = 1
2: for i = 2 to n do
3:     if a[i] > a[max] then
4:         max = i
5:     end if
6: end for
7: return max;
```


Pause to ponder...

Geht das auch *in-place* ?

Min/MaxSort

Algorithmus 5 MaxSort($A[1 \ldots n]$)

```
1: for i = n downto 2 do
2:     idx = max(A[1..i])
3:     swap(A[i], A[idx])
4: end for
```

Algorithmus 6 max($A[1 \ldots n]$)

```
1: idxMax = 1
2: for i = 2 to n do
3:     if A[i] > A[idxMax] then
4:         idxMax = i
5:     end if
6: end for
7: return idxMax;
```

InsertionSort

Algorithmus 7 InsertionSort($A[1 \dots n]$)

```
1: for j = 2 to n do
2:     key = A[j]
3:     i = j − 1
4:     while i > 0 und A[i] > key do
5:         A[i + 1] = A[i]
6:         i = i − 1
7:     end while
8:     A[i + 1] = key
9: end for
```


Analyse

Zur Laufzeit:

- Lineare Suche in $O(n)$
- Binäre Suche in $O(\log n)$
- Maximumsbestimmung in $O(n)$
- MaxSort in $O(n^2)$
- InsertionSort in $O(n^2)$

Die Kenngröße n ist dabei stets die Anzahl der zu sortierenden Elemente.

Eine kleine Warnung zum Schluss

Wichtige Anmerkung

Die O-Notation 'verschluckt' Konstanten. Wenn die zu gross/klein sind, dann kann dies das Ergebnis verfälschen! In dem Fall ist dann eine genauere Analyse (ohne O-Notation) nötig. I.A. hat sich die O-Notation aber bewährt, weil Extreme wie $10^6 \cdot n$ und $10^{-10} \cdot 2^n$ in der Praxis kaum vorkommen.
Kurz: $O(2^n)$ ist nicht zwingend *immer* schlimmer als $O(n)$ aber auf lange Sicht (d.h. bei wachsenden Eingabelängen) auf jeden Fall und im Allgemeinen (und bei allem, was einem so i.A. in der Praxis begegnet) ist $O(2^n)$ eben doch schlimmer als $O(n)$.

Vorgehen bei der Analyse und Ausblick

Bisher haben wir die Laufzeiten meist nur oberflächlich begründet/bewiesen. Dies werden wir in der Zukunft ausführlicher machen! (Da die Algorithmen komplizierter werden.)

Zudem werden wir uns mehr auf die *Korrektheit* von Algorithmen konzentrieren.

In der Zukunft daher oft:

1. Gegeben ein **Problem**, einen **Algorithmus finden**, der es löst.
2. **Korrektheit** des Algorithmus beweisen.
3. **Laufzeit** des Algorithmus ermitteln und die Behauptung beweisen.

Beobachtungen und Fragen

Beobachtungen und Fragen bis hierher:

1. *Korrektheit* der Verfahren bzw. von Algorithmen allgemein?
2. Bisher nur auf Arrays gearbeitet. Geht da auch was anderes, d.h. mit anderen *Datenstrukturen*? (Einerseits, um es vielleicht schneller zu machen, andererseits, weil die Daten vielleicht nicht in einem Array sind!)
3. Können wir schneller sortieren als in $O(n^2)$?
4. Wie kann man die Laufzeit *rekursiver Algorithmen analysieren*?

Heute: 1. und 4.

Sortieren

Definition (Das Sortierproblem)

Eingabe: Eine Sequenz $< a_1, a_2, \ldots, a_n >$ von n Zahlen.
Gesucht: Eine Permutation $< a'_1, a'_2, \ldots, a'_n >$ der Eingabesequenz mit $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Anmerkung

Ob die Reihenfolge zweier Elemente a_i, a_j $(i < j)$ mit $a_i = a_j$ beibehalten werden soll oder nicht, ist nicht gesagt. Bleibt sie erhalten, d.h. ist $a'_p = a_i$, $a'_q = a_j$ und $p < q$, so nennt man das Verfahren *stabil*.
Ferner heißt ein Sortierverfahren *in-place* (oder *in situ*), wenn der zusätzlich benötigte Speicherbedarf unabhängig von der gegebenen Sequenz ist.

InsertionSort: Die Idee

Beim *Sortieren durch Einfügen* (InsertionSort) wird ähnlich wie beim Spielkarten sortieren vorgegangen:

- Starte mit der leeren linken Hand
- Nimm ein Karte und füge sie an der richtigen Position in der linken Hand ein. Dazu
  - Vergleiche diese neue Karte von rechts nach links mit den Karten, die schon auf der linken Hand sind.
  - Sobald eine kleinere Karte erreicht wird, füge ein.

$\Rightarrow$ Zu jedem Zeitpunkt sind die Karten auf der linken Hand sortiert.

$\Rightarrow$ Die Karten auf der linken Hand sind jeweils die obersten Karten des Haufens.

InsertionSort: Der Algorithmus

Algorithmus 8 InsertionSort($A[1 \ldots n]$)

```
1: for j = 2 to n do
2:     key = A[j]
3:     i = j − 1
4:     while i > 0 und A[i] > key do
5:         A[i + 1] = A[i]
6:         i = i − 1
7:     end while
8:     A[i + 1] = key
9: end for
```

InsertionSort: Korrektheit 1/4

Die Korrektheit zeigen wir mit folgender **Schleifeninvarianten**:

> *Zu Beginn jeder Iteration der* **for***-Schleife besteht das Teilfeld* $A[1..j-1]$ *aus den ursprünglich in* $A[1..j-1]$ *enthaltenden Elementen, alledings in geordneter Reihenfolge.*

Anmerkung

'Zu Beginn der Iteration' heißt hier *nach* Zuweisung des neuen Wertes an j und *vor* dem Test im Schleifenkopf.

InsertionSort: Korrektheit 2/4

Zu zeigen ist:

1. Initialisierung: Die Invariante ist vor der ersten Iteration wahr.
2. Fortsetzung: Wenn die Invariante vor der Iteration der Schleife wahr ist, so ist sie es auch vor der nächsten Iteration.
3. Terminierung: Die Schleife bricht irgendwann ab.

Wenn die Schleife abbricht, so sollte die Invariante eine nützliche Eigenschaft liefern, die uns hilft, die Korrektheit des Algorithmus zu zeigen.

InsertionSort: Korrektheit 3/4

1. Initialisierung: Vor der ersten Iteration ist $j = 2$ und das Teilfeld $A[1..j-1] = A[1]$ besteht aus nur einem Feld, das auch ursprünglich dort war und das sortiert ist.
2. Fortsetzung: Zu zeigen ist, dass die Invariante bei jeder Iteration erhalten bleibt. Es wird die richtige Position für $A[j]$ gesucht (darum sind nachfolgend in $A[1..j]$ nur jene Elemente, die dort auch vorher waren) und gefunden (daher sind sie nun sortiert). Letzteres wäre mit einer weiteren Schleifeninvarianten für die while-Schleife zu zeigen.
3. Terminierung: Die while-Schleife terminiert stets (warum?) und damit auch die for-Schleife (warum?)

InsertionSort: Korrektheit 4/4

Wenn die Schleife abbricht, gilt $j = n + 1$. Setzen wir diesen Wert für j in die Invariante ein (die ja zu Beginn jeder Iteration wahr ist!), so erhalten wir, dass $A[1..n]$ aus den ursprünglichen Elementen in sortierter Reihenfolge besteht. Damit ist das gesamte Feld sortiert und der Algorithmus korrekt!

InsertionSort: Laufzeit

Die Laufzeit ist in $O(n^2)$, da der der Rumpf der for-Schleife $n-1$ mal ausgeführt wird und im i-ten Durchgang der for-Schleife der Rumpf der while-Schleife maximal i-mal ausgeführt wird. Damit wird der Rumpf der while-Schleife maximal $1+2+...+n-1 = \frac{n(n-1)}{2}$-mal ausgeführt.

Hinweis

Eine detailliertere Analyse ist in Kapitel 2.2 im [Cormen] zu finden.

Zu Korrektheitsbeweisen

Zum Vorgehen bei Korrektheitsbeweisen:

- Kein generelles Verfahren! Jeder Algorithmus muss "neu" als korrekt bewiesen werden.
- Schleifeninvarianten sind oft nützlich - aber auch diese lassen sich nicht automatisch bestimmen! Man muss also jedes Mal neu überlegen, was eine sinnvolle Schleifeninvariante sein könnte und dann beweisen, dass diese gilt (um sie dann hoffentlich beim Korrektheitsbeweis sinnvoll nutzen zu können).
- Ansonsten: Wie bei mathematischen Beweisen! Ein Stück Text schreiben, dass euer Gegenüber von der Richtigkeit der Aussage überzeugt. Bei der Beweisführung ist i.A. an irgendeiner Stelle ein kreativer Schritt nötig (Erfahrung hilft hier und wir üben dies im Laufe der Vorlesung).

Zu Korrektheitsbeweisen

Eine letzte Anmerkung:

- Oft wird die *Idee* zu einem Algorithmus als korrekt bewiesen. In der *Implementierung* könnten sich dann trotzdem Fehler einschleichen. Diese müsste also ebenfalls als korrekt nachgewiesen werden. Hierzu gibt es hilfreiche Tools, das geht aber über diese Vorlesung hinaus.

Eine andere Art von Algorithmen

Algorithmus 9 `fac`(n)

```
1: if n == 0 then
2:     return 1
3: else
4:     return n · fac(n − 1)
5: end if
```

Dies ist ein *rekursiver* Algorithmus... unsere bisherigen Techniken sind hier bei der Analyse wenig hilfreich (vor allem bei komplizierteren rekursiven Algorithmen).

Fakultätsfunktion - Iterativ

Jeder rekursive Algorithmus lässt sich auch iterativ schreiben (und umgekehrt)...

Algorithmus 10 `fac`(n)

```
1: res = 1
2: for i = 1 to n do
3:     res = res · i
4: end for
5: return res
```

Anmerkung

Die Laufzeit ist in $O(n)$. Aber Achtung! Dies ist **exponentiell in der Größe der Eingabe**! Diese ist nämlich nur in $O(\log n)$. Den *Wert* einer Zahl als Kenngröße zu nehmen ist i.A. nicht sinnvoll.

Rekursive Algorithmen

- Rekursive Algorithmen sind oft aufgrund der wiederholten Funktionsaufrufe und der mit den rekursiven Aufrufen einhergehenden Kontextspeicherung langsam.
- Andererseits lohnt sich für kompliziertere Probleme ein rekursiver Ansatz oft, da dieser meist übersichtlicher ist. Hier werden iterative Ansätze oft ebenfalls langsam, da sie viel verwalten müssen (im schlimmsten Fall muss die Stackverwaltung des rekursiven Algorithmus selbst geschrieben werden).

$\Rightarrow$ Wir wollen nachfolgend die Laufzeit rekursiver Algorithmen mittels *Rekurrenz- oder auch Rekursionsgleichungen* analysieren...

Die Türme von Hanoi

Definition (Aufgabenstellung)

Seien drei Stangen A, B, C gegeben und n Scheiben, die der Größe nach sortiert auf Stange A liegen. Die Aufgabe ist es, den ganzen Turm von Scheiben auf Stange C zu verschieben, dabei nur eine Scheibe in einem Zeitschritt zu bewegen und nie eine größere Scheibe auf eine kleinere zu legen.

Ein rekursiver Algorithmus

1. Verschiebe die $n-1$ oberen Scheiben von A auf B.
2. Verschiebe die größte Scheibe von A auf C.
3. Verschiebe alle $n-1$ Scheiben von B auf C.

Die Türme von Hanoi - Analyse

Sei $T(n)$ die Anzahl der Verschiebungen, die obiger Algorithmus braucht, um n Scheiben zu verschieben. Es gilt:

$$T(n) = \begin{cases} 1 & \text{, falls } n = 1 \\ 2T(n-1)+1 & \text{, falls } n \geq 2 \end{cases}$$

Wie ermitteln wir eine geschlossene Form für $T(n)$?

Die Türme von Hanoi - Analyse. Methode 1

Methode 1. Wir berechnen $T(n)$ für kleine Werte, Raten eine Lösung und beweisen diese.

$$T(n) = \begin{cases} 1 & \text{, falls } n = 1 \\ 2T(n-1)+1 & \text{, falls } n \geq 2 \end{cases}$$

n	1	2	3	4	5	6	7	8	9	10
$T(n)$	1	3	7	15	31	63	127	255	511	1023

Lösung raten? ... $T(n) = 2^n - 1$. Dies ist nun noch zu beweisen!

Die Türme von Hanoi - Analyse. Methode 1

Gegeben:

$$T(n) = \begin{cases} 1 & \text{, falls } n = 1 \\ 2T(n-1)+1 & \text{, falls } n \geq 2 \end{cases}$$

Zu zeigen: $T(n) = 2^n - 1$. - Machen wir mittels Induktion...

Induktionsanfang. Für $n = 1$ gilt $T(1) = 2^1 - 1 = 1$.

Induktionsannahme. Wir nehmen an, dass für ein $n \geq 1$ die Behauptung gilt, d.h. dass $T(n) = 2^n - 1$ ist.

Induktionsschritt.
$T(n+1) = 2T(n) + 1 = 2(2^n - 1) + 1 = 2^{n+1} - 1$.

Damit sind wir fertig. Für alle $n \geq 1$ gilt damit $T(n) = 2^n - 1$.

Die Türme von Hanoi - Analyse. Methode 2

Methode 2. Rekursives Einsetzen (Substitutions- oder Abwicklungsmethode)

$$T(n) = \begin{cases} 1 & \text{, falls } n = 1 \\ 2T(n-1)+1 & \text{, falls } n \geq 2 \end{cases}$$

$$\begin{aligned} T(n) &= 2T(n-1)+1 \\ &= 2(2T(n-2)+1)+1 = 4T(n-2)+2+1 \\ &= 4(2T(n-3)+1)+2+1 = 8T(n-3)+4+2+1 \\ &= \ldots \\ &= 2^k T(n-k) + 2^{k-1} + 2^{k-2} + \ldots + 2^1 + 2^0 \ (*) \end{aligned}$$

(*) Diese Zeile ist mit vollständiger Induktion zu zeigen (Übung)!

Die Türme von Hanoi - Analyse. Methode 2

Methode 2. Rekursives Einsetzen (Substitutions- oder Abwicklungsmethode)

$$T(n) = \begin{cases} 1 & \text{, falls } n = 1 \\ 2T(n-1)+1 & \text{, falls } n \geq 2 \end{cases}$$

$$T(n) = 2^k T(n-k) + 2^{k-1} + 2^{k-2} + \ldots + 2^1 + 2^0$$

Wann bricht die Rekursion ab? Bei $k = n - 1$

Die Türme von Hanoi - Analyse. Methode 2

Methode 2. Rekursives Einsetzen (Substitutions- oder Abwicklungsmethode)

$$T(n) = \begin{cases} 1 & \text{, falls } n = 1 \\ 2T(n-1)+1 & \text{, falls } n \geq 2 \end{cases}$$

$$\begin{aligned} T(n) &= 2^k T(n-k) + 2^{k-1} + 2^{k-2} + \ldots + 2^1 + 2^0 \ (*) \\ &= 2^{n-1} T(n-(n-1)) + 2^{n-2} + 2^{n-3} + \ldots + 2^1 + 2^0 \\ &= \sum_{i=0}^{n-1} 2^i = \frac{2^n - 1}{2-1} = 2^n - 1 \in \Theta(2^n) \end{aligned}$$

(*) Da bei $T(1)$, also $n - k = 1$, die Rekursion abbricht, wird mit $k = n - 1$ weiter gerechnet.

Methode 2 - Vorgehen

Das Vorgehen bei der Substitutionsmethode:

1. Abwickeln, bis eine Formel abhängig von k erkennbar ist.
2. Diese Formel mittels Induktion beweisen.
3. Den Wert von k bestimmen, bei dem die Rekursion abbricht.
4. Einsetzen und zu einem Ergebnis kommen.
5. Eventuell mit diesem Ergebnis noch Abschätzungen machen, um zu Aussagen bzgl. der O-Notation zu kommen.

Eine andere Rekurrenzgleichung

Die Rekurrenzgleichung

$$T(n) = \begin{cases} c & \text{, falls } n = 1 \\ 3T(n/3) + 1/n & \text{, falls } n \geq 2 \end{cases}$$

Kann auch mit der Substitutionsmethode gelöst werden.

Eine Alternative bietet das Mastertheorem.

Mastertheorem

Definition (Mastertheorem)

Das Mastertheorem gilt für Rekurrenzen der Form:

$$T(n) = \begin{cases} c & \text{falls } n = 1 \\ a \cdot T(\frac{n}{b}) + f(n) & \text{falls } n > 1 \end{cases}$$

Hierbei sind $a \geq 1, b > 1$ und c Konstanten, $\frac{n}{b}$ wird als $\lfloor \frac{n}{b} \rfloor$ oder $\lceil \frac{n}{b} \rceil$ verstanden und $T(n)$ über $\mathbb{N}$ definiert. Es gilt:

1. $T(n) \in \Theta(n^{\log_b(a)})$, falls $f(n) \in O(n^{\log_b(a)-\epsilon})$ für ein $\epsilon > 0$.
2. $T(n) \in \Theta(n^{\log_b(a)} \log_2(n))$, falls $f(n) \in \Theta(n^{\log_b(a)})$.
3. $T(n) \in \Theta(f(n))$, falls $f(n) \in \Omega(n^{\log_b(a)+\epsilon})$ für ein $\epsilon > 0$ **und** $a \cdot f(\frac{n}{b}) \leq \delta \cdot f(n)$ für ein $\delta < 1$ und große n.

Mastertheorem - Beispiel

$T(n) \in \Theta(n^{\log_b(a)})$, falls $f(n) \in O(n^{\log_b(a)-\epsilon})$ für ein $\epsilon > 0$

$$T(n) = \begin{cases} c' \\ 3T(n/3) + 1/n \end{cases} \qquad T(n) = \begin{cases} c \\ a \cdot T(\frac{n}{b}) + f(n) \end{cases}$$

Wegen $\log_b(a) = \log_3(3) = 1$ und $f(n) = \frac{1}{n}$ ist $f(n) \in O(n^{1-\epsilon})$ bei Wahl von z.B. $\epsilon = 2 > 0$.

Es liegt also Fall 1 vor und dies ergibt (wie auch beim Abwickeln) $T(n) \in \Theta(n)$.

Mastertheorem. Methode 3

Anmerkung

Das Mastertheorem ist die **dritte Methode**, die wir zur Lösung von Rekurrenzgleichungen nutzen werden.
Das Mastertheorem ist ein mächtiges Hilfsmittel. Es ist aber nicht immer einsetzbar! Z.B. bei $T(n) = 2 \cdot T(n/2) + n \cdot \log n$ nicht.

Literaturhinweis

Das Mastertheorem selbst wollen wir hier nicht beweisen. Einen Beweis findet man im [Cormen].

Zusammenfassung und Wiederholung

Übliches Vorgehen in der Algorithmik:

1. Gegeben ein Problem, einen Algorithmus finden, der es löst.
2. Korrektheit des Algorithmus beweisen.
3. Laufzeit des Algorithmus ermitteln und beweisen.

Zusammenfassung und Wiederholung

Zu 1. Algorithmentechniken / Algorithmenentwurf. Bisher:

- Lineare Methoden (z.B. Suche, Maximum/Minimum)
- Binäre Suche ('Halbieren')
- Suchraum komplett durcharbeiten (geht (fast) immer, aber teuer)

Zusammenfassung und Wiederholung

Zu 2. Vorgehen beim Korrektheitsbeweis:

- Kein generelles Verfahren! Jeder Algorithmus muss "neu" als korrekt bewiesen werden.
- Schleifeninvarianten sind oft nützlich.
- Ansonsten: Vorgehen wie bei mathematischen Beweisen! Erfahrung hilft!

Zusammenfassung und Wiederholung

Zu 3. Laufzeitanalyse...

- ... mittels O-Notation.
- Laufzeitanalyse einfacher sequentieller Algorithmen.
- Laufzeitanalyse rekursiver Algorithmen (z.B. von Divide and Conquer Verfahren) mittels dreier Methoden:
  - Raten und Beweisen
  - Abwickeln
  - Mastertheorem

Zusammenfassung und Wiederholung

Wichtig bei der Laufzeitanalyse:

1. Überlegen bzgl. welcher *Kenngröße* der Eingabe wir messen wollen. Diese kann sich von der Größe der Eingabe unterscheiden! (Beispiel: Anzahl Knoten eines Graphen).
2. Laufzeit/Speicherbedarf bzgl. dieser Kenngröße ausdrücken.
3. Nochmal überlegen, ob dies die Aussage des Ergebnisses verfälscht (so wie bei der Fakultätsberechnung oben).

$\Rightarrow$ I.A. sollte sich die eigentliche Eingabegröße leicht durch die Kenngröße ausdrücken lassen und der Unterschied sollte nicht zu groß sein.

- \+ Beim Graphen mit n Knoten ist die Adjazenmatrix in $O(n^2)$.
- – Ist eine Zahl n die Eingabe, so ist die Eingabegröße in $O(\log n)$. Eine Laufzeit von $O(n)$ wäre also exponentiell in der Eingabe!

Ausblick

Nächstes Mal:

- **Divide and Conquer** als klassische Algorithmentechnik (= Algorithmen-'Muster') bzw. klassisches rekursives Verfahren
  - $\Rightarrow$ Größere Probleme werden in kleinere zerlegt, die dann seperat (und i.A. rekursiv) nach der gleichen Methode (!) gelöst werden. Die Teilergebnisse werden dann passend zur Lösung des Problems zusammengesetzt.
  - Unter anderem MergeSort als Beispiel
  - Rekurrenzgleichungen sind hier wichtig
- Mehr zum **Mastertheorem**
  - Nützliches Hilfsmittel: *Variablentransformation*

Anhang

Anhang

Ein weitere Beispiel

$$T(n) = \begin{cases} c & \text{, falls } n = 1 \\ 3T(n/3) + 1/n & \text{, falls } n \geq 2 \end{cases}$$

$$\begin{aligned} T(n) &= 3 \cdot T(\frac{n}{3}) + \frac{1}{n} \\ &= 3 \cdot \left(3 \cdot T(\frac{n}{3^2}) + \frac{3}{n}\right) + \frac{1}{n} \\ &= 3^2 \cdot T(\frac{n}{3^2}) + \frac{3^2}{n} + \frac{3^0}{n} \\ &= 3^2 \cdot \left(3 \cdot T(\frac{n}{3^3}) + \frac{3^2}{n}\right) + \frac{3^2}{n} + \frac{3^0}{n} \\ &= 3^3 \cdot T(\frac{n}{3^3}) + \frac{3^4}{n} + \frac{3^2}{n} + \frac{3^0}{n} \\ &= \dots \end{aligned}$$

Ein weitere Beispiel

$$\begin{aligned} T(n) &= 3^3 \cdot T(\frac{n}{3^3}) + \frac{3^4}{n} + \frac{3^2}{n} + \frac{3^0}{n} \\ &= \dots \\ &= 3^k \cdot T(\frac{n}{3^k}) + \frac{1}{n}\sum_{i=0}^{k-1} 3^{2i} \ (*) \\ &= c \cdot n + \frac{1}{n}\frac{3^{2k}-1}{3^2-1} \ (\text{mit } k = \log_3(n) \text{ bzw. } n = 3^k) \\ &= c \cdot n + \frac{n^2-1}{8n} = c \cdot n + \frac{1}{8}\left(n - \frac{1}{n}\right) \end{aligned}$$

(*) Diese Zeile ist wieder mit vollständiger Induktion zu zeigen!

Ein weitere Beispiel

Aus

$$T(n) = c \cdot n + \frac{1}{8}\left(n - \frac{1}{n}\right)$$

folgt nun

- $T(n) < (c + \frac{1}{8}) \cdot n$ und damit $T(n) \in O(n)$ und
- $T(n) > c \cdot n$ und damit $T(n) \in \Omega(n)$.

Insgesamt haben wir also

$$T(n) \in \Theta(n)$$

gezeigt!