Diskrete Strukturen und Logik

WiSe 2006/07 in Trier

Henning Fernau
Universität Trier
fernau@informatik.uni-trier.de

Diskrete Strukturen und Logik

Gesamtübersicht

- Organisatorisches
- Einführung
- Logik & Mengenlehre
- Beweisverfahren
- Kombinatorik: Die Kunst des Zählens
- algebraische Strukturen

Wachstum von Funktionen — Motivation

Wichtige Ressourcen informatischen Denkens:

- Laufzeit (Rechenzeit) und
- Speicherbedarf (Speicherplatz, evtl. hierarchisiert)

Ziel mathematischer Modellierung: Unabhängigkeit von Stand der Technik
$\leadsto$ (maschinenabhängige) “Konstanten” sind “unwichtig”

Wesentliche Sichtweise der Theorie algorithmischer Probleme (*Komplexitäts-theorie*) und der Theorie konkreter algorithmischer Lösungen (*Algorithmik*)

$\mathcal{O}$-Notation

Definition (Paul Bachmann 1894, später von Edmund Landau popularisiert):
Es seien $f, g \in \mathbb{R}^{\mathbb{N}}$. $g(n) = \mathcal{O}(f(n))$ für $n \in \mathbb{N}$, falls exists $M, n_0 \in \mathbb{N}$, so dass gilt: $\forall n \geq n_0 : |g(n)| \leq |Mf(n)|$

Hinweis: Ungenauer (asymmetrischer) Gebrauch von "$=$"; genauer ist $\mathcal{O}(f)$ als Menge von Funktionen aufzufassen, dann müsste man $g \in \mathcal{O}(f)$ schreiben. Diese Sicht nehmen wir oft ein.

D.E. Knuth liest $\mathcal{O}$ als "Groß-Omikron"; weitere Schreibweisen:
$g \in \Omega(f)$ gdw. $f \in \mathcal{O}(g)$ (definiert wiederum Funktionenmenge)
$g \in \Theta(f)$ gdw. $g \in \mathcal{O}(f) \cap \Omega(f)$.

Einige Beispiele

$\mathcal{O}(1)$:	konstanter Aufwand, unabhängig von n
$\mathcal{O}(n)$:	linearer Aufwand (z.B. Einlesen von n Zahlen)
$\mathcal{O}(n \ln n)$:	Aufwand guter Sortierverfahren (z.B. Quicksort)
$\mathcal{O}(n^2)$:	quadratischer Aufwand
$\mathcal{O}(n^k)$:	polynomialer Aufwand (bei festem k)
$\mathcal{O}(2^n)$:	exponentieller Aufwand
$\mathcal{O}(n!)$:	Bestimmung aller Permutationen von n Elementen

Aus VL7: m-te Harmonische Zahl $H_m \in \Theta(\log(m))$ (Basis egal).

Konkreter Vergleich

Annahme: 1 Schritt dauert 1 μs = 0.000001 s

$n =$	10	20	30	40	50	60
n	10 μs	20 μs	30 μs	40 μs	50 μs	60 μs
n^2	100 μs	400 μs	900 μs	1.6 ms	2.5 ms	3.6 ms
n^3	1 ms	8 ms	27 ms	64 ms	125 ms	216 ms
2^n	1 ms	1 s	18 min	13 Tage	36 J	366 Jh
3^n	59 ms	58 min	6.5 J	3855 Jh	10^8 Jh	10^{13} Jh
$n!$	3.62 s	771 Jh	10^{16} Jh	10^{32} Jh	10^{49} Jh	10^{66} Jh

Anwendung in der Algorithmik: Laufzeitanalyse

Verschiedene Analysen sind von Interesse:

- bester Fall (best case)
- mittlerer Fall (average case)
- schlimmster Fall (worst case)

Für den eigentlich meist interessantesten mittleren Fall wären auch noch Annahmen über die zu erwartende Eingabeverteilung sinnvoll.
Meistens beschränkt man sich bei der Analyse auf den schlimmsten Fall.

Anwendung in der Algorithmik: Laufzeitanalyse von for-Schleifen (Beispiele)

Minimumsuche in einem Array der Länge n

```
min = a[0];
for (i = 1; i < n; i++)
       if(a[i] < min) min = a[i];
```

n "Schritte" für n Daten $\rightsquigarrow$ Laufzeit $\Theta(n)$

```
for (i = 0; i < k; i++)
   for (j = 0; j < k; j++)
      brett[i][j] = 0;
```

k^2 Schritte für k^2 Daten

$\Rightarrow \Theta(n)-$Algorithmus

```
for (i = 0, i < k; i++)
   for (j = 0; j < k; j++)
       if (a[i] == a[j]) treffer = true;
```

k^2 Schritte für k Daten

$\Rightarrow \Theta(n^2)-$Algorithmus

Anwendung in der Algorithmik: Laufzeitanalyse von while-Schleifen (Beispiele)

Lineare Suche im Array

```
i = 0;
while (i < n) && (a[i] != x)
   i++;
```

Laufzeit:	bestenfalls	1 Schritt	$\Rightarrow$	$\Theta(1)$
	schlimmstenfalls	n Schritte	$\Rightarrow$	$\Theta(n)$
	im Mittel	$\frac{n}{2}$ Schritte	$\Rightarrow$	$\Theta(n)$

Annahme für den mittleren Fall: Es liegt (gleichwahrscheinliche) Permutation der Zahlen von 1 bis n vor. Dann ist die mittlere Anzahl

$$= \frac{1}{n} \sum_{i=1}^{n} i \approx \frac{n}{2}$$

Anwendung in der Algorithmik: Laufzeitanalyse von while-Schleifen (Beispiele)

Lineare Suche in 0/1-Array (also $x, a[i] \in \{0, 1\}$)

```
i = 0;
while (i < n) && (a[i] != x)
   i++;
```

Bester und schlimmster Fall wie bisher.
Mittlerer Fall bei Annahme einer "Erfolgswahrscheinlichkeit" pro Stelle von 1/2 und der weiteren (in der Regel fälschlichen) Annahme der Unabhängigkeit aufeinanderfolgender "Experimente"
$\leadsto$ Erwartungswert der geometrischen Verteilung ist 2 (unabhängig von n)
$\leadsto \Theta(1)$ im mittlereren Fall
Gilt analog für beliebige "endliche Alphabetgrößen"

Amortisierte Analyse

Oft wird der schlechteste Fall eines komplizierteren Algorithmus abgeschätzt durch die schlechtesten Fälle der Teilschritte; dies vermeidet die *Aggregat-Methode*, eine Möglichkeit der *amortisierten Analyse*.

Beispiel: Wir wollen zählen (abschätzen), wie viele Bitwechsel beim Inkrementieren eines k-Bit-Zählers von 0 bis $(2^k - 1)$ entstehen. Eine einfache Analyse, ausgehend vom schlimmsten Fall eines einzelnen Inkrements, liefert $\mathcal{O}(k \cdot 2^k)$. Geht es besser ?

Betrachten wir die Anzahl der Bitwechsel bei einem Zähler mit 3 Bit:

Zähler	0000	0001	0010	0011	0100	0101	0110	0111	1000
Anzahl Bitwechsel	0	1	2	1	3	1	2	1	4

Beobachte: Das niedrigste Bit ändert sich bei jeder Inkrementation, das nächst höhere bei jeder zweiten, das wiederum nächst höhere bei jeder vierten usw. Damit ergibt sich bei n Inkrementationen folgende Summe von Bitwechseln:

$$n + \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{2^2} \right\rfloor + \left\lfloor \frac{n}{2^3} \right\rfloor + \cdots + \left\lfloor \frac{n}{2^k} \right\rfloor \leq n \sum_{i=0}^{k} \frac{1}{2^i}$$

Diese Summe können wir nach oben abschätzen:

$$n \sum_{i=0}^{k} \frac{1}{2^i} \leq n \sum_{i=0}^{\infty} \frac{1}{2^i} \leq 2n$$

Ein einzelnes Inkrement kostet daher amortisiert 2 Bitwechsel, nicht etwa k wie zunächst abgeschätzt.

Analyse von Rekursionen

Bei der Ressourcen-Analyse von rekursiven Algorithmen enstehen oft Rekursionsgleichungen, die gelöst werden müssen. In einer Rekursionsgleichung für eine Funktion f wird zur Berechnung der Funktion für den Wert n die Funktion für andere, kleinere Werte verwendet. Allgemein hat eine Rekursionsgleichung die Form $f(n) = G(f(1); f(2); \ldots; f(n-1))$.
Eine Rekursionsgleichung definiert eine Funktionenschar (alle Funktionen die die Gleichung erfüllen).
Aus der Funktionenschar wird eine Funktion durch Festlegen weiterer Parameter spezifiziert, z.B. durch Lösen der *Anfangswertbedingungen* $f(1) = a_1$, $f(2) = a_2$, ..., $f(k) = a_k$.

Beispiel: Betrachte die Rekursionsgleichung $T(n) = 2T(n-1)$ und $T(0) = 1$.
Offenbar gilt: $T(n) = 2^n$ (*explizite Darstellung*).

Fibonacci liefert die Rekursionsgleichung $f(n) = f(n-1) + f(n-2)$ und die Anfangsbedingungen $f(1) = f(2) = 1$.
Offensichtlich ist $f(n)$ größer als $2f(n-2)$, d.h., $f(n)$ wächst asymptotisch mindestens mit $2^{n/2} = \sqrt{2}^n$.
Wir vermuten daher exponentielles Wachstum.
Annahme: $f(n) = ac^n$ für gewisse $a; c > 0$.
Induktionsschritt. Einsetzen in die Rekursionsgleichung ergibt

$$f(n) = f(n-1) + f(n-2) = ac^{n-1} + ac^{n-2} = ac^n \left(\frac{1}{c} + \frac{1}{c^2}\right) = ac^n \frac{c+1}{c^2}$$

Damit der Induktionsschritt korrekt ist, muss $ac^n \frac{c+1}{c^2} = ac^n$ sein, d.h., $c^2 - c - 1 = 0$. Das liefert $c = \frac{\sqrt{5}+1}{2}$ (goldener Schnitt).
Induktionsanfang 1. Da $c \leq 2$ gilt $f(1) \geq ac^1$ und $f(2) \geq ac^2$ für $a = 1/4$.
$\rightsquigarrow f(n) \geq \frac{1}{4}\left(\frac{\sqrt{5}+1}{2}\right)^n$.
Induktionsanfang 2. Da $c \geq 1$ gilt $f(1) \leq ac^1$ und $f(2) \leq ac^2$ für $a = 1$.
$\rightsquigarrow f(n) \leq \left(\frac{\sqrt{5}+1}{2}\right)^n$.
Also: $f(n) \in \Theta\left(\left(\frac{\sqrt{5}+1}{2}\right)^n\right)$.

Rekursive Algorithmen — Divide and Conquer

Viele Algorithmen zerlegen das zu lösende Problem einer Größe in 2 oder mehrere Teilprobleme mit einer geringeren Größe.
Die Teilprobleme werden (rekursiv) berechnet und die Lösung aus den Ergebnissen zusammengesetzt. Die Laufzeit für eine Eingabe der Länge n setzt sich also zusammen aus der Laufzeit für die kleineren Eingaben plus der Laufzeit die benötigt, wird die Eingabe zu zerlegen und die Lösung zusammenzufügen.

```
Procedure: DivideAndConquer(x:Eingabe)
IF |x| = 1 THEN
   berechne Lösung l für x direkt.
ELSE zerlege x in zwei (gleichgroße) Teile x1 und x2
   l1 = DivideAndConquer(x1)
   l2 = DivideAndConquer(x2)
   berechne Lösung l für x aus l1 und l2.
RETURN l
```

Rekursive Algorithmen — Divide and Conquer Analyse

Der Aufwand für das Zerlegen und Zusammenfügen sei $cn + d$.
Die Komplexität ergibt sich damit zu: $f(n) = 2f(n/2) + cn + d$:

Wir versuchen wieder eine Lösung zu der Rekursionsgleichung zu finden und diese zu verifizieren. Bei jedem Aufruf halbiert sich die Problemgröße, nach $\log_2(n)$ vielen Schritten wird die Größe 1 und damit das Ende der Rekursion erreicht. Da sich die Anzahl der Teilprobleme jedesmal verdoppelt, summiert sich die Anzahl der Schritte (vermutlich) jeweils wieder zu $cn + d$.
Wir vermuten daher eine Lösung der Form $f(n) = \alpha + \beta n + \delta n \log_2 n$.

Die vermutete Lösung $\alpha + \beta n + \delta n \log_2 n$ setzen wir induktiv ein:

$$\begin{aligned} f(n) &= cn + d + 2f(n/2) \\ &= cn + d + 2(\alpha + \beta(n/2) + \delta(n/2)\log_2(n/2)) \\ &= cn + d + 2\alpha + \beta n + \delta n \log_2 n - \delta n \end{aligned}$$

Koeffizientenvergleich liefert $\alpha = -d$ und $\delta = c$. Dies ergibt Funktionen der Form

$$\{-d + \beta n + cn \log_n \mid \beta \in \mathbb{R}\}.$$

Der Wert von β ergibt sich aus der Anfangswertbedingung.
Für f(1) = 13 z.B. ergibt sich aus $-d + \beta + c\log_2 1 = 13$, dass $\beta = d + 13$.

Lösen von Rekursionsgleichungen

- Induktive Einsetzungsmethode

- Iterationsmethode

Induktive Einsetzungsmethode: Anhand der Rekursionsgleichung wird eine Lösung geraten und durch Induktion bestätigt. Oft enthält die vermutete Lösung noch zu spezifizierende Parameter, diese ergeben sich oft durch einen Koeffizientenvergleich der für den Induktionsbeweis notwendig wird. (konstruktive Induktion).

Bei der *Iterationsmethode* wird die Funktion auf der rechten Seite immer wieder durch die Rekursionsgleichung ersetzt. Dann wird versucht die rechte Seite in eine geschlossenen Form zu bringen.

Beispiel: Iterationsmethode für $f(n) = n + 3f(n/4)$. Iteriertes Einsetzen und Ersetzen der Summe ergibt:

$$\begin{aligned} f(n) &= n + 3f(n/4) = n + 3(n/4 + 3f(n/16)) \\ &= n + 3n/4 + 9(n/16 + 3f(n/64)) \leq n + 3n/4 + 9n/16 + \ldots \\ &= n \cdot \sum_{i \geq 0} (3/4)^i = n \cdot \frac{1}{1 - 3/4} = 4n = \Theta(n) \end{aligned}$$

Die Anfangswertbedingung $f(1) = a$ wirkt sich nur auf die in der Θ-Notation verborgenen Konstanten aus.

Analytische Methodik—Erzeugende Funktionen / z-Transformation

Einen völlig anderen Zugang zur Lösung von Rekursionen gestattet die Analysis:

Interpretiere Folgen als Koeffizienten einer Potenzreihe.

Der Rekursionsgleichung (unter Berücksichtigung der Anfangsbedingungen) entspricht dann eine Funktionalgleichung für die *erzeugende Funktion*.
Dabei Konvention: $T(n) = 0$ für $n < 0$.
Nützliche Schreibweise: $[P(n)]$ für Prädikat P auf $\mathbb{Z}$ mit

$$[P(n)] = \begin{cases} 1 & P(n) \\ 0 & \neg P(n) \end{cases}$$

Beispiel: Rekursiongleichung $T(n) = 2T(n-1)$ mit Anfangsbedingung $T(0) = 1$

$\rightsquigarrow T(n) = 2T(n-1) + [n = 0]$ (eine Gleichung !)

$$\begin{aligned} G(z) &= \sum_{n\in\mathbb{N}} T(n)z^n = \sum_{n\in\mathbb{N}} (2T(n-1) + [n = 0])z^n \\ &= 2z \sum_{n\in\mathbb{N}} T(n-1)z^{n-1} + 1 = 2z \sum_{n\in\mathbb{N}} T(n)z^n + 1 \\ &= 2zG(z) + 1 \end{aligned}$$

$\rightsquigarrow G(z) = \frac{1}{1-2z}$

$\rightsquigarrow$ Potenzreihenentwicklung $G(z) = \sum_{n\in\mathbb{N}} (2z)^n$, denn $\frac{1}{1-y} = \sum_{n\in\mathbb{N}} y^n$

$\rightsquigarrow T(n) = 2^n$

Ein hilfreiches Lemma

Ist $q(z) = 1 + q_1 z + \cdots + q_d z^d$, $q_d \neq 0$ so bezeichnet $q^R(z) = z^d + q_1 z^{d-1} + \cdots + q_d$ das *reflektierte Polynom*.

Lemma: Sind $a_1, \ldots, a_d$ die Nullstellen des reflektierten Polynoms, so gilt:

$$q(z) = \prod_{j=1}^{d} (1 - a_j z).$$

Beweis: $q(z) = z^d \prod_{j=1}^{d} (1/z - a_j)$ gilt, denn $q^R(y) = y^d q(1/y)$.

Fibonacci—nochmals

geschlossene Rekursion: $F_n = F_{n-1} + F_{n-2} + [n = 1]$.

Daraus Funktionalgleichung:

$$F(z) = \sum F_n z^n = \sum F_{n-1} z^n + \sum F_{n-2} z^n + \sum [n = 1] z^n = zF(z) + z^2 F(z) + z.$$

$$\rightsquigarrow F(z) = \frac{z}{1 - z - z^2}$$

Partialbruchzerlegung mit dem hilfreichen Lemma und dem Ansatz:

$$\frac{1}{(1 - \alpha z)(1 - \beta z)} = \frac{a}{1 - \alpha z} + \frac{b}{1 - \beta z}$$

Nullstellen von $q^R(z) = z^2 - z - 1$: $\alpha = \frac{1+\sqrt{5}}{2} = \phi$ und $\beta = \frac{1-\sqrt{5}}{2} = 1 - \phi =: \hat{\phi}$.
Unser Ansatz liefert:

$$\frac{1}{(1-\phi z)(1-\hat{\phi}z)} = \frac{a}{1-\phi z} + \frac{b}{1-\hat{\phi}z} = \frac{(a+b) - (a\hat{\phi} + b\phi)}{(1-\phi z)(1-\hat{\phi}z)}$$

und damit das lineare Gleichungssystem mit den Bedingungen
$a + b = 1$ und $\hat{\phi}a + \phi b = 0$, also $a = \frac{\phi}{\sqrt{5}}$ und $b = -\frac{\hat{\phi}}{\sqrt{5}}$.
So erhalten wir die folgende schon bekannte Beziehung als explizite Darstellung der F_n:

$$F_n = \frac{\phi}{\sqrt{5}}\phi^{n-1} - \frac{\hat{\phi}}{\sqrt{5}}\hat{\phi}^{n-1}.$$

Da $|\hat{\phi}| = \left|\frac{1-\sqrt{5}}{2}\right| < 1$, ist F_n die zu $\frac{\phi^n}{\sqrt{5}}$ nächstgelegene ganze Zahl ist.
Daraus folgt unmittelbar: $F_n = \Theta(\phi^n)$.

z-Transformation allgemein (hier speziell für lineare Rekursionen)

1. Darstellung der Rekursion in einer Gleichung
   $f(n) = q_1 f(n-1) + q_2 f(n-2) + \cdots + q_d f(n-d) +$ *Anfangsbedingungen*.

2. Darstellung als erzeugende Funktion $F(z)$ und Auflösen der Funktionalgleichung als $F(z) = \frac{p(z)}{q(z)}$, wobei p und q Polynome sind vom Grad höchstens d.

3. Partialbruchzerlegung liefert

$$F(z) = \sum f(n) z^n = \sum_{i=1}^{k} \frac{g_i(z)}{(1-\alpha_i z)^{d_i}}$$

   Die α_i sind Nullstellen des (komplexen) Polynoms $q^R(z)$ der Vielfachheit d_i, g_i sind Polynome vom Grad kleiner d_i.

4. Explizite Darstellung der Rekursion: $f(n) = \sum_{i=1}^{k} p_i(n) \alpha_i^n$

Insbesondere gilt: $f(n) = \Theta(p_j(n)\alpha^n)$ mit $\alpha = \alpha_j = \max_{i=1}^{k} \alpha_i$.
Beweis: siehe Aigner, S. 62

z-Transformation in der Wahrscheinlichkeitsrechnung

Ist $X : S \to \mathbb{N}$ eine ZV, so können wir ihr als *wahrscheinlichkeitserzeugende Funktion* $P_X(z) = \sum P[X = n]z^n$ zuordnen.

Nun kann man Erwartungswert und Varianz analytisch ausdrücken:
$E[X] = P'_X(1)$ (durch die erste Ableitung)
$\mathrm{Var}[X] = P''_X(1) + P'_X(1) - (P'_X(1))^2$
Sind $X, Y : S \to \mathbb{N}$ unabhängige ZV, so gilt: $P_{X+Y}(z) = P_X(z)P_Y(z)$.