2. Komplexität von Algorithmen

2.1. Komplexitätsmaße

Wie wachsen die Kosten für Zeit und Raum für die Ausführung eines Algorithmus, wenn die Größe des Problems wächst? Das ist das Komplexitätsproblem für Algorithmen. Bevor man über das Komplexitätsverhalten eines Algorithmus sprechen kann, muß man ein Maß für die Problemgröße festlegen. So könnte ein Maß für die Größe eines Matrizenmultiplikationsproblems etwa die größte Dimension der Matrizen sein, oder ein Maß für die Größe eines Graphenproblems könnte die Zahl der Kanten sein.

Definition 2.1.1: (informal) Der Zeitaufwand, der für die Ausführung eines Algorithmus benötigt wird, ausgedrückt als Funktion der Problemgröße, heißt die *Zeitkomplexität* des Algorithmus. Das Grenzverhalten der Komplexität, wenn die Problemgröße gegen ∞ tendiert, heißt die *asymptotische Zeitkomplexität* des Algorithmus. Analog kann man die *Raumkomplexität* und die *asymptotische Raumkomplexität* eines Algorithmus definieren. Wird für eine gegebene Problemgröße die Komplexität als maximale Komplexität über alle Eingaben dieser Größe gemessen, so spricht man von der *worst–case Komplexität*. Wird die Komplexität als mittlere Komplexität über alle Eingaben einer festen Größe gemessen, so spricht man von der *erwarteten* oder *durchschnittlichen Komplexität*. ■

Im allgemeinen werden wir die worst–case Komplexität eines Algorithmus messen, da sie mathematisch leichter handhabbar ist als die erwartete Komplexität. Man muß sich aber darüber im klaren sein, daß der Algorithmus mit der besten worst–case Komplexität nicht notwendigerweise auch die beste erwartete Komplexität besitzt. Bei der Bestimmung der Komplexität eines Algorithmus ist man nur an der Ordnung der Komplexitätsfunktion interessiert, d.h. additive und multiplikative Konstante werden vernachlässigt.

Definition 2.1.2: Eine Funktion $g(n)$ auf den natürlichen Zahlen ist $\mathcal{O}(f(n))$ (*von der Ordnung* $f(n)$), falls $g(n) \leq c \cdot f(n)$ für eine positive reelle Konstante c und alle genügend großen n ($n \geq N$ für ein $N \in \mathbb{N}$). ■

Beispiel 2.1.1: a) $6n^2 + 7n$ ist $\mathcal{O}(n^2)$, denn $6n^2 + 7n \leq 7n^2$ für $n \geq 7$.
b) $\log_a n$ ist $\mathcal{O}(\log_b n)$ für $a, b \in \mathbf{R}^+$, denn $\log_a n = \log_a b \cdot \log_b n$. ■

Nun könnte man annehmen, daß eine Steigerung der Geschwindigkeit von Rechenmaschinen die Entwicklung schneller Algorithmen überflüssig machen würde. Es tritt jedoch gerade der gegenteilige Effekt ein. Werden die Computer immer schneller und können wir dadurch immer größere Probleme behandeln, so ist es schließlich das Komplexitätsverhalten der Algorithmen welches die Umsetzbarkeit der schnelleren Hardware in größere Probleme bestimmt. Nehmen wir etwa an wir hätten fünf Algorithmen $A_1, \ldots, A_5$ mit der folgenden Zeitkomplexität:

Algorithmus	Zeitkomplexität
A_1	n
A_2	$n \log n$
A_3	n^2
A_4	n^3
A_5	2^n

Die Zeitkomplexität ist die Anzahl der Zeiteinheiten (etwa Millisekunden), die benötigt werden, um eine Eingabe der Größe n zu verarbeiten. Die folgende Tabelle gibt an, wie die Größe der verarbeitbaren Problem wächst mit der dafür aufgewendeten Zeit.

Algorithmus	Zeitkomplexität	Laufzeit		
		1 Sek.	1 Min.	1 Std.
A_1	n	1 000	6×10^4	3.6×10^6
A_2	$n \log n$	140	4 893	2.0×10^5
A_3	n^2	31	244	1 897
A_4	n^3	10	39	153
A_5	2^n	9	15	21

Nehmen wir nun an, daß die nächste Generation von Computern zehn mal so schnell sei wie die derzeitige. Die folgende Tabelle stellt dar, wie stark sich die Zunahme der Geschwindigkeit auswirkt auf die Größe der Probleme, die behandelt werden können.

Algorithmus	Zeitkomplexität	maximale Problemgröße...	
		...vor speed-up	...nach speed-up
A_1	n	s_1	$10s_1$
A_2	$n \log n$	s_2	$\sim 10s_1$ für große s_2
A_3	n^2	s_3	$3.16s_3$
A_4	n^3	s_4	$2.15s_4$
A_5	2^n	s_5	$s_5 + 3.3$

Betrachten wir andererseits den Gewinn, wenn wir von einem langsamen Algorithmus zu einem schnelleren übergehen. Ersetzt man etwa A_4 durch A_3, so kann man in einer Minute ein sechs mal so großes Problem lösen; ersetzt man A_4 durch A_2, so kann man ein 125 mal so großes Problem lösen. Der Unterschied wird noch gravierender, wenn man die Rechenzeit erhöht.

2.2. Komplexität von RAM Programmen

Die Zeitkomplexität eines RAM Programms ergibt sich als Summe der benötigten Zeit für jede Instruktion des Programms. Analog ergibt sich die Raumkomplexität eines RAM Programms als Summe des benötigten "Raumes" für jedes angesprochene Register. Wir betrachten zwei verschiedene *Kostenkriterien* für RAM Programme. Unter dem *uniformen Kostenkriterium* benötigt jede RAM Instruktion eine Zeiteinheit und jedes Register eine Raumeinheit. Wenn nichts anderes ausdrücklich vereinbart ist, werden wir immer das uniforme Kostenkriterium zur Beurteilung der Komplexität von RAM Programmen heranziehen.

Das *logarithmische Kostenkriterium* zieht in Betracht, daß auf allen gegenwärtig verfügbaren Rechnern ein Speicherwort beschränkt ist, die Anzahl der benötigten Speicherworte für ein Register also abhängt von der Länge des Inhalts. Im allgemeinen braucht man $\lfloor \log n \rfloor + 1$ Bits, um eine Zahl n binär darzustellen. Sei $l(n)$ die folgende *logarithmische Funktion* auf den ganzen Zahlen:

$$l(n) = \begin{cases} \lfloor \log \mid n \mid \rfloor + 1, & \text{falls } n \neq 0 \\ 1, & \text{falls } n = 0. \end{cases}$$

Die logarithmischen Kosten $t(a)$ für die verschiedenen Formen eines Operanden a sind also

Operand a	Kosten $t(a)$
`=i`	$l(i)$
`i`	$l(i) + l(c(i))$
`*i`	$l(i) + l(c(i)) + l(c(c(i)))$.

Das logarithmische Kostenkriterium basiert auf der Annahme, daß die Kosten für die Ausführung einer Instruktion proportional sind zur Länge des Operanden. Die logarithmischen Kosten der einzelnen RAM Instruktionen sehen wie folgt aus (dabei sind $t(a)$ die Kosten des Operanden a):

Instruktion		Kosten
1.	`LOAD a`	$t(a)$
2.	`STORE i`	$l(c(0)) + l(i)$
	`STORE *i`	$l(c(0)) + l(i) + l(c(i))$
3.	`ADD a`	$l(c(0)) + t(a)$
4.	`SUB a`	$l(c(0)) + t(a)$
5.	`MULT a`	$l(c(0)) + t(a)$
6.	`DIV a`	$l(c(0)) + t(a)$
7.	`READ i`	$l(\text{input}) + l(i)$
	`READ *i`	$l(\text{input}) + l(i) + l(c(i))$
8.	`WRITE a`	$t(a)$

9.	`JUMP b`	1
10.	`JGTZ b`	$l(c(0))$
11.	`JZERO b`	$l(c(0))$
12.	`HALT`	1

Die *logarithmische Raumkomplexität* eines RAM Programms ist die Summe über alle verwendeten Register, Akkumulator eingeschlossen, von $l(x_i)$, wobei x_i die betragsmäßig größte in Register i gespeicherte Zahl ist.

Nach welchem Kostenkriterium man die Komplexität eines Algorithmus mißt hängt natürlich davon ab wie große Zahlen vom Algorithmus verarbeitet werden. Kann man annehmen, daß jede Zahl, die im Laufe einer Berechnung auftritt, in ein Computerwort paßt, so ist das uniforme Kostenkriterium angebracht. Andernfalls könnte das logarithmische Kostenkriterium eine realistischere Analyse ergeben.

Beispiel 2.2.1: Wir bestimmen die Zeit– und Raumkomplexität des RAM Programms aus Beispiel 1.3.1 zur Berechnung der Funktion

$$f(n) = \begin{cases} n^n & \text{für } n > 0, \\ 0 & \text{sonst.} \end{cases}$$

Die Zeitkomplexität des Programms ist beherrscht von der Schleife um die `MULT` Instruktion. Wenn die `MULT` Instruktion zum i–ten mal ausgeführt wird, enthält der Akkumulator n^i und Register 2 enthält n. $n-1$ solche `MULT` Intruktionen werden ausgeführt. Unter dem uniformen Kostenkriterium kostet jede `MULT` Intruktion eine Zeiteinheit, die Ausführung aller dieser `MULT` Instruktionen benötigt also $\mathcal{O}(n)$ Zeit. Unter dem logarithmischen Kostenkriterium kostet die i–te `MULT` Instruktion $l(n^i) + l(n) = (i+1)\log n$ Zeit, die Kosten für die Ausführung aller `MULT` Instruktionen betragen also

$$\textstyle\sum_{i=1}^{n-1} (i+1)\log n \ ,$$

das ist $\mathcal{O}(n^2 \log n)$. Alle anderen Operationen sind von nicht größerer Komplexität.

Die Raumkomplexität ist nur vom Inhalt der Register 0 bis 3 abhängig. Somit ist unter dem uniformen Kostenkriterium die Raumkomplexität einfach $\mathcal{O}(1)$. Unter dem logarithmischen Kostenkriterium ist die Raumkomplexität $\mathcal{O}(n \log n)$, da n^n die größte Zahl ist, die in einem dieser Register gespeichert wird, und $l(n^n) \simeq n \log n$.

	uniforme Kosten	logarithmische Kosten
Zeitkomplexität	$\mathcal{O}(n)$	$\mathcal{O}(n^2 \log n)$
Raumkomplexität	$\mathcal{O}(1)$	$\mathcal{O}(n \log n)$

Das uniforme Kostenkriterium ist nur für sehr kleine n realistisch. Paßt n^n nicht mehr in ein Computerwort, so ist auch das logarithmische Kostenkriterium unrealistisch. Es nimmt nämlich an, daß zwei Zahlen m und n in Zeit $\mathcal{O}(l(m) + l(n))$ miteinander multipliziert werden können. Es ist jedoch nicht bekannt, ob das möglich ist. ■

Beispiel 2.2.2: Für das RAM Programm in Beispiel 1.3.2 hängt die Komplexität in folgender Weise von der Länge n des Inputs ab:

	uniforme Kosten	logarithmische Kosten
Zeitkomplexität	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$
Raumkomplexität	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$

Paßt n nicht mehr in ein Computerwort, so ist das logarithmische Kostenkriterium ziemlich realistisch. ■

Satz 2.2.1: *Sowohl unter dem uniformen als auch dem logarithmischen Kostenkriterium gibt es für jedes RAM Programm P mit Zeitkomplexität $T(n)$ eine Konstante k und ein äquivalentes RASP Programm P' mit Zeitkomplexität $kT(n)$.*

Beweis: Dazu betrachten wir den Teil (a) des Beweises von Satz 1.3.1. Im dort konstruierten RASP Programm P' kommen für jede RAM Instruktion höchstens 6 RASP Instruktionen vor. Unter dem uniformen Kostenkriterium ist also die Zeitkomplexität von P' höchstens $6T(n)$. Durch detailierte Untersuchung des Programms P' stellt man fest, daß die Behauptung auch für das logarithmische Kostenkriterium gilt mit $k = 6 + 11 \cdot l(r)$, wobei r der im Beweis von Satz 1.3.1 erstellte Verschiebungsfaktor ist. ■

Satz 2.2.2: *Sowohl unter dem uniformen als auch dem logarithmischen Kostenkriterium gibt es für jedes RASP Programm P' mit Zeitkomplexität $T(n)$ eine Konstante k und ein äquivalentes RAM Programm p mit Zeitkomplexität $kT(n)$.*

Beweis: Durch Analyse des Teils (b) des Beweises von Satz 1.3.1. ■

2.3. Komplexität von Turing–Maschinen

Definition 2.3.1: Die *(worst–case) Zeitkomplexität* $T(n)$ einer Turing–Maschine M ist die maximale Anzahl von Rechenschritten (Übergang von einer Konfiguration zur nächsten) zur Abarbeitung einer beliebigen Eingabe der Länge n. Gibt es ein Eingabewort der Länge n, für welches die Turing–Maschine nicht hält, so ist $T(n)$ undefiniert. Die *(worst–case) Raumkomplexität* $S(n)$ von M ist die maximale Distanz vom linken Bandende, welche einer der LS–Köpfe im Laufe der Abarbeitung eines Eingabewortes der Länge n zurücklegt. Gibt es eine Eingabe der Länge n für welche sich einer der LS–Köpfe unbegrenzt nach rechts bewegt, dann ist $S(n)$ undefiniert.

Eine nichtdeterministische Turing–Maschine M hat die *Zeitkomplexität* $T(n)$, falls bei einer Eingabe der Länge n keine Folge von nichtdeterministischen Wahlen existiert, welche eine Berechnung länger als $T(n)$ bedingt. Eine nichtdeterministische Turing–Maschine M hat die *Raumkomplexität* $S(n)$, falls bei einer Eingabe der Länge n keine Folge von nichtdeterministischen Wahlen existiert, welche M auf einem der Bänder mehr als $S(n)$ Zellen bearbeiten läßt. Vgl. [HU], p.288. ∎

Beispiel 2.3.1: Wir bestimmen die Komplexität der Turing–Maschine M_1 aus Beispiel 1.4.1 zur Erkennung der Sprache $L = \{0^m 1^m | m \in \mathbb{N}\}$. Das Eingabewort w habe die Länge n. In der Hauptberechnungsschleife werden jeweils 2 Symbole von w ausgekreuzt (durch X bzw. Y ersetzt). Die Schleife kann also höchstens $(n+1)/2$ mal ausgeführt werden. Die Anzahl der Schritte für einen Schleifendurchlauf setzt sich zusammen aus den Schritten für die Suche nach rechts (höchstens $n+1$ Schritte), der Bewegung nach links (höchstens n Schritte) und der Bewegung zum nächsten Symbol (1 Schritt), sie kann also höchstens $2n+2$ betragen. Somit werden zur Abarbeitung aller Schleifendurchläufe höchstens $(2n+2) \cdot (n+1)/2 = (n+1)^2$ Schritte benötigt. Die Schlußbehandlung kann in n Schritten durchgeführt werden. Die Zeitkomplexität von M_1 ist also $\mathcal{O}(n^2)$.

Der LS–Kopf bewegt sich höchstens um eine Position über das Eingabewort hinaus, die Raumkomplexität von M_1 ist also $\mathcal{O}(n)$. ∎

Definition 2.3.2: Zwei zahlentheoretische Funktionen $f_1(n)$ und $f_2(n)$ (also $f_1, f_2 : \mathbb{N} \to \mathbb{N}$) sind *in polynomialer Relation*, wenn es Polynome $p_1(x)$ und $p_2(x)$ mit ganzzahligen Koeffizienten gibt, sodaß für alle natürlichen Zahlen n gilt $f_1(n) \le p_1(f_2(n))$ und $f_2(n) \le p_2(f_1(n))$. ∎

Beispiel 2.3.2: Die Funktionen $f_1(n) = 2n^2$ und $f_2(n) = n^5$ sind in polynomialer Relation; dazu können wir $p_1(x) = 2x$ und $p_2(x) = x^3$ wählen, da $2n^2 \leq 2n^5$ und $n^5 \leq (2n^2)^3$.

a^n und b^n sind in polynomialer Relation für $a, b > 1$. Für $p(x) = x^{\lceil \log_b a \rceil}$ gilt $a^n = (b^{\log_b a})^n = (b^n)^{\log_b a} \leq p(b^n)$.

Die Funktionen n^2 und 2^n sind jedoch nicht in polynomialer Relation, da es kein Polynom $p(x)$ gibt, für welches gelten würde $p(n^2) \geq 2^n$ für alle n. ∎

Satz 2.3.1: Das RAM/RASP Modell unter dem logarithmischen Kostenkriterium und das Turing–Maschinen Modell sind in polynomialer Relation; d.h. die Komplexitätsfunktionen der beiden Modelle sind in polynomialer Relation. ∎

Der Beweis des Satzes 2.3.1 ergibt sich aus einer Analyse der Simulation einer RAM durch eine Turing–Maschine und umgekehrt. Unter dem uniformen Kostenkriterium kann jede Turing–Maschine mit Komplexität $\mathcal{O}(T(n))$ durch ein RAM Programm mit Komplexität $\mathcal{O}(T^2(n))$ simuliert werden. Die Umkehrung gilt jedoch nicht. Unter dem uniformen Kostenkriterium kann nämlich eine RAM in $\mathcal{O}(n)$ Schritten die Zahl 2^{2^n} berechnen ($2^{2^n} = 2^{2^{n-1} \cdot 2} = (2^{2^{n-1}})^2$). Eine Turing–Maschine braucht bereits 2^n Zellen nur um diese Zahl zu speichern.