Einführung in die Informatik

Klaus Knopper

02.11.2004

Formale Eigenschaften von Algorithmen

- Korrektheit von Algorithmen
  - Man kann die Korrektheit von Algorithmen im allgemeinen nicht durch Testen an ausgewählten Beispielen nachweisen, denn mit **Testen kann lediglich die Anwesenheit, nicht jedoch die Abwesenheit von Fehlern nachgewiesen werden.**
  - Nachweis der Korrektheit geht nur über in der Regel sehr aufwendige und komplexe Korrektheitsbeweise. Diese Beweise sind **nicht** Inhalt dieser Vorlesung.
- Effizienz von Algorithmen
  - Speicherplatzverbrauch
  - Rechenzeit

Effizienz von Algorithmen

Wie lässt sich die Effizienz von Algorithmen messen?

- Nach der Implementierung des Algorithmus kann die Rechenzeit und der Speicherplatzverbrauch in Abhängigkeit der Eingabewerte gemessen werden (Benchmarking).
- Vor der Implementierung des Algorithmus kann aber bereits sein Laufzeitverhalten mathematisch bestimmt werden.

Motivation

Eingabe: Eine positive natürliche Zahl n.

Ausgabe: Die Summe s aller natürlichen Zahlen zwischen 0 und n.

Vorgehen:

```
S := 0;
FOR i := 0 TO n DO
  S := S + i;
```

Wie lange würde die Laufzeit des Algorithmus dauern, wenn jede Addition 5 ms benötigt, alle anderen Anweisungen 0 ms und $n = 100000$ ist? Antwort: $100001 \cdot 5\text{ms} = 500005\text{ms} = 500,005\text{s}$

Laufzeitkomplexität (1)

Idee: Bestimmen der Anzahl der auszuführenden Anweisungen in Abhängigkeit der Eingabevariablen eines Algorithmus.

Problem: wie ermittelt man die Anzahl der auszuführenden Anweisungen?

Ziel: Aus jedem Algorithmus $\mathcal{A}$ wird eine Funktion f abgeleitet, die in Abhängigkeit der Eingabevariablen die Anzahl der Anweisungen liefert.

Laufzeitkomplexität (2)

Eingabe: Zwei positive natürliche Zahlen a, b.

Ausgabe: Eine positive natürliche Zahl s (die die Summe der Eingaben ist, siehe Vorgehen).

Vorgehen:

```
S := 0;
S := A + B;
```

Fazit: Bei diesem Algorithmus werden für alle möglichen Eingaben immer 2 Anweisungen ausgeführt.

Laufzeitkomplexität (3)

Eingabe: Eine positive natürliche Zahl n.

Ausgabe: Eine positive natürliche Zahl s.

Vorgehen:

```
s := 0;
FOR i := 1 TO n DO
  FOR j := i TO n DO
    s := s + 1;
```

Dies entspricht:

$$\sum_{i=1}^{n} \sum_{j=i}^{n} 1 \quad = \quad ?$$

Einwurf: Rechnen mit Summen

Beim Rechnen mit Summen führen wir folgende Regeln ein:

$$\sum_{i=1}^{n} c = n \cdot c$$

$$\sum_{i=0}^{n} c = \sum_{i=1}^{n+1} c$$

$$\sum_{i=a}^{n} c \text{ mit } a < n = \left(\sum_{i=1}^{n} c\right) - \left(\sum_{i=1}^{a-1} c\right)$$

$$\sum_{i=a}^{n} c \text{ mit } a > n = 0$$

Einwurf: Rechnen mit Summen

$$\sum_{i=1}^{n} c \cdot a_i = c \cdot \sum_{i=1}^{n} a_i$$

$$\sum_{i=1}^{n} (a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i$$

$$\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$$

$$\sum_{i=1}^{n} i^2 = \frac{n \cdot (n+1) \cdot (2 \cdot n + 1)}{6}$$

Laufzeitkomplexität (3)

$$
\begin{aligned}
\sum_{i=1}^{n}\sum_{j=i}^{n} 1 &= \sum_{i=1}^{n}\left(\sum_{j=1}^{n} 1 - \sum_{j=1}^{i-1} 1\right) \\
&= \sum_{i=1}^{n}(n-(i-1)) \\
&= \sum_{i=1}^{n}(n-i+1) \\
&= \sum_{i=1}^{n} n - \sum_{i=1}^{n} i + \sum_{i=1}^{n} 1
\end{aligned}
$$

Laufzeitkomplexität (4)

$$\begin{aligned}\sum_{i=1}^{n}\sum_{j=i}^{n} 1 &= n^2 - \frac{n\cdot(n+1)}{2} + n \\ &= (n^2+n) - \frac{n^2+n}{2} \\ &= \frac{n^2+n}{2}\end{aligned}$$

Für ein gegebenes n kann nun die Anzahl der benötigten Anweisungen berechnet werden:

$n = 1$: $\frac{1^2+1}{2} = 1$

$n = 10$: $\frac{10^2+10}{2} = 55$

Komplexitätsklassen

Um Algorithmen besser vergleichen zu können werden sie in Komplexitätsklassen zusammengefasst. Eine **Komplexitätsklasse** ist eine Kategorie von Algorithmen, zusammengefasst nach einem gemeinsamen Maß der Komplexität. Sie ist definiert durch das asymptotische Verhalten der Obergrenze (oder des Mittelwertes oder der Untergrenze) des Resourcenbedarfs (insbesondere an Laufzeit und Speicherplatz) in Abhängigkeit von einer Problemgröße n (Länge der Eingabe).
Eine Laufzeitfunktion g besitzt die Größenordnung f, wenn g Element der Komplexitätsklasse $O(f)$ ist. $O(f)$ ist folgendermaßen definiert:

$$O(f) = \{g | \exists c_1 > 0 : \exists c_2 > 0 : \forall n \in \mathcal{Z}^+ : g(n) \leq c_1 \cdot f(n) + c_2\}$$

Komplexitätsklassen (2)

Mit Hilfe der Definition von O lassen sich folgende Größenordnungen festlegen:

- logarithmisches Wachstum: $\log N$
- lineares Wachstum: N
- quadratisches, kubisches, ... Wachstum: $N^2, N^3, \ldots$
- n-log n-Wachstum: $N \cdot \log N$
- exponentielles Wachstum: $2^N, 3^N, \ldots$

Alle Algorithmen einer Komplexitätsklasse besitzen für große N das gleiche Laufzeitverhalten.

Komplexitätsklassen (3)

In welcher Komplexitätsklasse liegt unsere Laufzeitfunktion

$$g(n) = \frac{n^2 + n}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$$

Die Funktion g ist in $O(N^2)$, denn mit $c_1 = 2$ und $c_2 = 1$ gilt für alle nichtnegativen, ganzzahligen N $g(N) \leq c_1 N^2 + c_2$. Somit ist $g(N)$ Element von $O(N^2)$.

Laufzeitkomplexität von `WHILE, IF`

Bislang haben wir die Laufzeitkomplexität nur von „einfachen“ Algorithmen betrachtet. Wir haben noch keine Überlegungen zum Einfluß von `WHILE`- und `IF`-Konstrukte auf das Laufzeitverhalten angestellt.
Die Bestimmung der Laufzeitkomplexität wird (sehr viel) komplizierter, wenn `WHILE`- und `IF`-Konstrukte in den Algorithmen verwendet werden. Oft kann man die Laufzeitkomplexität gar nicht in einer Funktion bestimmen sondern muss den

- best case,
- average case und den
- worst case

gesondert behandeln.

Beispiel

Eingabe: Ein Vektor $(a[1], \ldots a[n])$ aus natürlichen Zahlen $a[i]$.

Ausgabe: Die Summe s aller ungeraden Elemente des Vektors.

Vorgehen:

```
FOR i := 1 TO n DO
  IF (odd(a[i])) THEN
    sum := sum + a[1];
```

Komplexitätsverhalten:

- best case (Vektor enthält nur gerade Zahlen): $O(1)$
- avg case (Vektor enthält ungefähr gleichviel gerade wie ungerade Zahlen): $O(n)$
- worst case (Vektor enthält nur ungerade Zahlen): $O(n)$