Laufzeitanalyse von Algorithmen

1 Zielsetzung

Hat man einen Algorithmus implementiert, möchte man gerne wissen, wie gross der Zeit- und der Speicherbedarf für eine bestimmte Eingabe ist.

Da dies jedoch in den meisten Fällen sehr aufwändig oder gar unmöglich ist, begnügt man sich damit, den Zeit- oder den Speicheraufwand in Abhängigkeit der Anzahl der Eingabegrössen n auszdrücken.

Die Beschreibung der Laufzeit als Funktion $f(n)$ soll ...

- abhängig von der Anzahl n der Eingabdaten sein,
- unabhängig von der Programmiersprache oder der Hardware sein,
- den ungünstigsten Fall („worst case“) darstellen.

2 Die $\mathcal{O}$-Notation

Die $\mathcal{O}$-Notation (engl.: *Big-Oh-Notation*) ist ein Hilfsmittel zur mathematischen Beschreibung der Laufzeit eines Algorithmus.

Definition

$\mathcal{O}(f(n))$ ist die Menge aller Funktionen $g(n)$, für die es eine Konstante c und eine Schranke N gibt, so dass $g(n) \leq c \cdot f(n)$ für alle $n \geq N$.

oder etwas umgangssprachlicher:

$\mathcal{O}(f(n))$ ist die Menge aller Funktionen $g(n)$, die ab einem bestimmten n nicht schneller wachsen als die Funktion $c \cdot f(n)$, wobei c eine frei wählbare Kontante ist.

Beispiel 1

$g(n) = 10n$

$g(n) = 10n \leq 10 \cdot n \quad$ (für alle $n \geq 1$)

$\Rightarrow \quad g(n) \in \mathcal{O}(n)$

Beispiel 2

$g(n) = 3n + 2$

$g(n) = 3n + 2 \leq 3n + 2n = 5n \quad$ (für alle $n \geq 1$)

$\Rightarrow \quad g(n) \in \mathcal{O}(n)$

Beispiel 3

$g(n) = 2n^2$

$g(n) = 2n^2 \leq 2 \cdot n^2 \quad$ (für alle $n \geq 1$)

$\Rightarrow \quad g(n) \in \mathcal{O}(n^2)$

Beispiel 4

$g(n) = n^2 + 2n$

$g(n) = n^2 + 2n \leq n^2 + 2n^2 = 3n^2 \quad$ (für alle $n \geq 1$)

$\Rightarrow \quad g(n) \in \mathcal{O}(n^2)$

Beispiel 5

$g(n) = 5n^4 + 4n^3 + 3n^2 + 2n + 1$

$g(n) \leq 5n^4 + 4n^4 + 3n^4 + 2n^4 + n^4 = 15n^4 \quad$ (für alle $n \geq 1$)

$\Rightarrow \quad g(n) \in \mathcal{O}(n^4)$

Beispiel 6

$g(n) = |\sin(n)|$

$g(n) = |\sin(n)| \leq 1 = 1 \cdot 1 \quad$ (für alle $n \geq 1$)

$\Rightarrow \quad g(n) \in \mathcal{O}(1)$

Beispiel 7

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

$g(n) = 2^{n+1} = 2^n \cdot 2^1 = 2 \cdot 2^n \quad$ (für alle $n \geq 1$)

$\Rightarrow \quad g(n) \in \mathcal{O}(2^n)$

Rechenregeln

Ist $g_1(n) \in \mathcal{O}\big(f_1(n)\big)$ und $g_2(n) \in \mathcal{O}\big(f_2(n)\big)$, so gilt:

- $g_1(n) + g_2(n) \in \mathcal{O}\big(\max(f_1(n), f_2(n))\big)$
- $g_1(n) \cdot g_2(n) \in \mathcal{O}\big(f_1(n) \cdot f_2(n)\big)$

3 Wichtige Klassen von Laufzeitkomplexitäten

Konstante Laufzeit

$\mathcal{O}(1)$

- Wert in einer Liste lesen/schreiben
- Zwei Zahlen multiplizieren

Logarithmische Laufzeit

$\mathcal{O}(\log n)$

- Suche in einer sortierten Liste

Lineare Laufzeit

$\mathcal{O}(n)$

- Suche in einer unsortierten Liste

Log-Lineare Laufzeit

$\mathcal{O}(n \cdot \log n)$

- fortgeschrittene Sortieralgorithmen

Quadratische Laufzeit

$\mathcal{O}(n^2)$

- „naive“ Sortieralgorithmen

Kubische Laufzeit

$\mathcal{O}(n^3)$

- Matrizenmultiplikation

Polynomielle Laufzeit

$\mathcal{O}(n^p)$

- Simplex-Algorithmus (lineare Optimierung)

Exponentielle Laufzeit

$\mathcal{O}(2^n)$

- Rucksackproblem

Faktorielle Laufzeit

$\mathcal{O}(n!)$

- Problem des Handlungsreisenden

Bemerkung:

Algorithmen mit $\mathcal{O}(n^4)$ und höher benötigen bei einer Verdoppelung der Inputgrösse bereits die 16-fache Laufzeit, was problematisch ist. Algorithmen mit $\mathcal{O}(2^n)$ oder höher sind praktisch unbrauchbar.

Beispiel 8

Welche Laufzeitkomplexität hat das Python-Programmfragment?

```
1  s = 1
2  for i in range(0, n):
3      for j in range(0, n):
4          for k in range(0, n):
5              s = i + j + k
```

Zeile	Kosten	Anzahl
1	c_1	1
2	c_2	n
3	c_3	n^2
4	c_4	n^3
5	c_5	n^3

$T(n) = c_1 + c_2 n + c_3 n^2 + c_4 n^3 \in \mathcal{O}(n^3)$