Laufzeitkomplexität von Algorithmen — Theorie

O-Notation

Die Gross-O-Notation ist ein Hilfsmittel zur Klassifikation von Algorithmen anhand des Aufwands, den sie asymptotisch, d. h. mit zunehmender Problemgrösse im Worst Case bewältigen müssen.

Dieser Aufwand wird durch Funktionen der Form $f\colon \mathbb{N} \to \mathbb{R}^+$ gemessen, die jeder natürlichen Zahl $n \in \{1, 2, 3, \dots\}$ eine positive reelle Zahl $f(n)$ zuordnen.

Wir werden später sehen, dass uns in diesem Zusammenhang meist nur „elementare“ Funktionen wie $f(x) = x^2$, $f(x) = \log_2(x)$ oder $f(x) = 2^x$ von Bedeutung sind.

Ist nun T eine Funktion, die einem Algorithmus für eine Problemgrösse n seine Worst-Case-Laufzeit $T(n)$ zuordnet, so sagen wir

$T(n) \in O\big(f(n)\big)$ [sprich: T liegt in O von f]

wenn $\lim\limits_{n\to\infty} \left|\dfrac{T(n)}{f(n)}\right| < \infty$ *gilt.*

Beispiel 1

$T(n) = 5n^2 + 2n - 1$

- $T(n) \in \mathrm{O}(n^2)$ denn $\lim\limits_{n\to\infty} \left|\dfrac{5n^2 + 2n - 1}{n^2}\right| = \lim\limits_{n\to\infty} \left|5 + \dfrac{2}{n} - \dfrac{1}{n^2}\right| = 5 < \infty$
- $T(n) \in \mathrm{O}(n^3)$ denn $\lim\limits_{n\to\infty} \left|\dfrac{5n^2 + 2n - 1}{n^3}\right| = \lim\limits_{n\to\infty} \left|\dfrac{5}{n} + \dfrac{2}{n^2} - \dfrac{1}{n^3}\right| = 0 < \infty$
- $T(n) \notin \mathrm{O}(n)$ denn $\lim\limits_{n\to\infty} \left|\dfrac{5n^2 + 2n - 1}{n}\right| = \lim\limits_{n\to\infty} \left|5n + 2 - \dfrac{1}{n}\right| = \infty$

Beobachtungen zu Beispiel 1

- $T(n) \in \mathrm{O}\big(f(n)\big)$ bedeutet, dass die Funktion T, abgesehen von einem konstanten Faktor c, mit zunehmenden n nicht schneller wächst als f.
- $T(n)$ kann sich in mehreren Komplexitätsklassen befinden.

  $T(n) = 5n^2 + 2n - 1$ liegt z. B. in $\mathrm{O}(n^2)$, $\mathrm{O}(n^3)$, $\mathrm{O}(n^4)$, usw.

  Meist interessiert uns die Klasse mit dem „kleinsten“ Wachstum; hier also $\mathrm{O}(n^2)$.
- Offenbar lassen sich Summanden, die weniger stark als der dominante Term wachsen sowie Koeffizienten des dominanten Terms weglassen, um die Komplexitätsklasse ohne Rechnung zu bestimmen.

Aufgabe 1

Bestimme die kleinstmögliche Komplexitätsklasse von $T(n)$ in der O-Notation.

(a) $T(n) = 100n^4 - 3n^2 + 93$

(b) $T(n) = 15n^2 + 3n + 7n^3$

(c) $T(n) = (2n + 7)(4n^2 + 7n)(n^4 + 1)$

(d) $T(n) = 17.3$

(d) $T(n) = 3^{n+2}$

Beispiel 2

$T(n) = \log_2(n) \in \mathrm{O}(\log_{10} n)$, denn

$$\lim_{n\to\infty} \left| \frac{\log_2(n)}{\log_{10}(n)} \right| = \lim_{n\to\infty} \left| \frac{\log_{10}(n)/\log_{10}(2)}{\log_{10}(n)} \right| = \lim_{n\to\infty} \left| \frac{1}{\log_{10}(2)} \right| < \infty$$

aber auch umgekehrt:

$T(n) = \log_{10}(n) \in \mathrm{O}(\log_2 n)$, denn

$$\lim_{n\to\infty} \left| \frac{\log_{10}(n)}{\log_2(n)} \right| = \lim_{n\to\infty} \left| \frac{\log_2(n)/\log_2(10)}{\log_2(n)} \right| = \lim_{n\to\infty} \left| \frac{1}{\log_2(10)} \right| < \infty$$

Moral: Die Laufzeitkomplexität von Logarithmen ist unabhängig von ihrer Basis.

Beispiel 3

In einem Programm werden die Algorithmen A_1, A_2 und A_3 mit den Laufzeiten
$T_1(n) = 4n^2 + 7 \in \mathrm{O}(n^2)$
$T_2(n) = 2n + 3 \in \mathrm{O}(n)$
$T_3(n) = 3n^2 + n \in \mathrm{O}(n^2)$
nacheinander ausgeführt. Wie gross ist die Laufzeitkomplexität des Gesamtprogramms?

$T(n) = T_1(n) + T_2(n) + T_3(n) = 7n^2 + 3n + 10 \in \mathrm{O}(n^2)$

Verallgemeinerung von Beispiel 3

Besteht ein Programm aus einer festen Anzahl von Algorithmen $A_1, A_2, \ldots, A_k$, die nacheinander ausgeführt werden und die Laufzeitkomplexitäten $\mathrm{O}\big(f_1(n)\big), \ldots, \mathrm{O}\big(f_k(n)\big)$ haben, so liegt die Laufzeitkomplexität des gesamten Programms in $\mathrm{O}\big(\max\{f_1(n), \ldots, f_k(n)\}\big)$.

Bestimme in den Aufgaben 2–7 die Zeit $T(n)$, welche die Python-Funktion zur Ausführung braucht, wenn jede Zuweisung, jede Rechen- und Vergleichsoperation, jeder Schleifendurchlauf (ohne Schleifenkörper), jede `return`-Anweisung und jede Verzweigung jeweils c Zeiteinheiten benötigt. Gib an, in welcher Komplexitätsklasse die Funktion liegt.

Aufgabe 2

```
def f(n):
    a = 3
    c = 2*a + 1
    return c
```

Aufgabe 3

```
def f(n):
    s = 3
    for i in range(0, n):
        s = s + i
    return s
```

Aufgabe 4

```
def f(n):
    x = 1
    for i in range(0, n):
        x = x * 2
        for j in range(0, n):
            x = x + 1
    return x
```

Aufgabe 5

```
def f(n):
    x = 1
    for i in range(0, n):
        x = x * 2
    for j in range(0, n):
        x = x + 1
    return x
```

Aufgabe 6

```
def f(n):
    x = 0
    if n % 2 == 0:
        x = 42
    else:
        for i in range(0, n):
            x = x + i
    return x
```

Aufgabe 7

```
def f(n):
    i = n
    s = 0
    while i > 0:
        s = i
        i = i // 2
    return s
```

Wichtige Laufzeitklassen mit Beispielen

$\mathrm{O}(1)$	Zuweisungen, arithmetische Operationen, logische Operationen und Vergleiche, Ein- und Ausgabe, Listenzugriffe, Sprunganweisungen
$\mathrm{O}(\log(n))$	Binäre Suche
$\mathrm{O}(n)$	Sequentielle Suche in Listen
$\mathrm{O}\big(n \log(n)\big)$	schnelle Sortieralgorithmen (Quicksort)
$\mathrm{O}\big(n^2\big)$	naive Sortieralgorithmen (Selectionsort)
$\mathrm{O}\big(n^3\big)$	klassische Multiplikation von zwei $(n \times n)$-Matrizen
$\mathrm{O}\big(2^n\big)$	Erzeugung aller Binärzahlen der Länge n, Türme von Hanoi
$\mathrm{O}\big(n!\big)$	Brute-Force-Lösung des Travelling-Salesman-Problems

Algorithmen mit Komplexitäten unterhalb des horizontalen Striches sind unbrauchbar für grosse Eingaben.

Schlussbemerkung

In der Literatur wird meist die etwas irreführende Schreibweise $g(n) = \mathrm{O}\big(f(n)\big)$ anstelle von $g(n) \in \mathrm{O}\big(f(n)\big)$ verwendet.