Komplexität von Algorithmen

Der Begriff „Komplexität" bedeutet in diesem Zusammenhang, wie sich die Laufzeit (Anzahl der Rechenschritte) bzw. der Speicherbedarf (Anzahl der benötigten Speicherstellen) eines Algorithmus verändert, wenn sich die Menge der Eingabedaten vergrößert.

Zeitkomplexität

Bleiben wir bei dem Zeitbedarf den ein Algorithmus benötigt, wenn sich die Menge der zu verarbeitenden Daten vergrößert und blenden den Speicherbedarf im Folgenden aus. Die Vorgehensweise ist aber analog!

Bei all diesen Betrachtungen löst man sich von einer konkreten Maschine, d.h. es geht hier nur um die Anzahl der erforderlichen Berechnungsschritte und nicht darum, wie schnell der konkrete Rechner arbeitet. Den abstrakten Zeitbedarf für einen Algorithmus können wir uns als eine Funktion T(n) vorstellen, wobei n die Menge der zu verarbeitenden Daten und damit verbunden die Anzahl der Rechenschritte darstellt.

Prinzipiell lassen sich für T(n) drei Situationen unterscheiden:

- den minimalen Zeitbedarf (*best case*), den ein Algorithmus benötigt.
- den durchschnittlichen Zeitbedarf (*average case*), den ein Algorithmus benötigt.
- den maximalen Zeitbedarf (*worst case*), den ein Algorithmus benötigt.

Sehen Sie sich dazu mit mir als Beispiel die „Lineare Suche" an.

In einem Bücherregal stehen z.B. hundert Bücher unterschiedlicher Autoren in unsortierter Reihenfolge. Sie wollen das Buch eines bestimmten Autors heraus suchen und beginnen am linken (oder auch rechten) Ende des Bücherregals mit der Suche. Jeder Vergleich dauert hierbei eine bestimmte konstante Zeitspanne Δt.

Sie nehmen nun Buch für Buch aus dem Regal heraus, um nachzuschauen, ob es das richtige Buch ist. Im besten Fall (*best case*) ist das Gesuchte bereits das erste heraus genommene Buch. Sie haben somit einmal die Zeit Δt benötigt. Im schlechtesten Fall (*worst case*) war das gesuchte Buch das Letzte im Regal. Sie haben nun hundertmal Δt benötigt. Im Mittel (*average case*) fallen 50 mal Δt an.

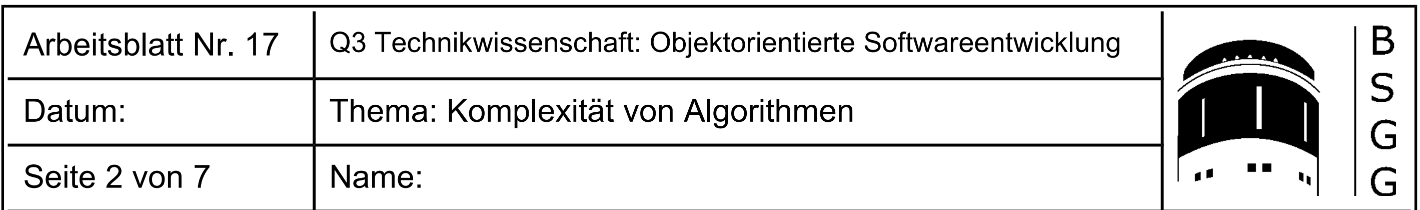
Wenn Sie nun den gleichen Vorgang bei tausend Büchern wiederholen müssen, benötigen Sie im besten Fall erneut nur einmal die Zeit Δt. Im schlechtesten Fall benötigen Sie nun tausendmal die Zeitspanne Δt und im Mittel eben 500 mal Δt.

Im Mittel und im schlechtesten Fall wächst der Zeitaufwand für eine größere Anzahl Bücher, die im Regal stehen, linear.

Sehen wir uns alternativ hierzu das Prinzip der „Binären Suche" an. Vorbedingung ist hierbei, dass die Bücher nun sortiert in dem Regal stehen müssen. Als Sortierkriterium soll hierbei der Name des Buchautors dienen, da wir wie zuvor das Buch eines bestimmten Autors suchen.

Wir beginnen nun in der Mitte des Regals mit den hundert Büchern und schauen anhand des Autors nach, ob sich das gesuchte Buch in der ersten Hälfte oder in der zweiten Hälfte des Regals befinden muss, sofern es nicht (*best case*) bereits das Gesuchte ist. Somit ist die nun zu durchsuchende verbleibende Bücherzahl auf die Hälfte (49 oder 50) gesunken. Nun bewegen wir uns zur Mitte des nun verbliebenen Bestands und greifen dort ein zweites Mal ein Buch heraus und überprüfen erneut, falls wir nicht gerade das Gesuchte gefunden haben. Somit sinkt der zu durchsuchende Bestand wieder auf die Hälfte (24 bzw. 25). Und wieder wird in der betreffenden

© Uwe Homm Version vom 6. Januar 2019



Hälfte ein drittes Mal das Buch herausgenommen und überprüft, wodurch sich der restliche Bestand auf 12 oder 13 Bücher reduziert. Nach dem vierten Herausziehen bleiben noch 6 oder 7 Bücher in der verbleibenden Hälfte. Das fünfte Prüfen reduziert den Bestand auf 3 oder vier Bücher, das sechste Prüfen auf ein oder zwei Bücher und spätestens mit dem siebten Prüfvorgang ist das gesuchte Buch gefunden. Dies ist damit auch der schlechteste Fall (*worst case*).

Ändern wir nun den Bücherbestand wieder auf tausend Bücher. Wie viele Prüfvorgänge haben wir nun im schlechtesten Fall, bis das gesuchte Buch gefunden ist? Wenn Sie das Beispiel genauso durchspielen, sollten Sie auf zehn Prüfvorgänge (Vergleiche) kommen.

Selbst wenn die Bücher bei der Linearen Suche bereits sortiert im Regal stehen, ist dieser Algorithmus sicherlich sehr ungünstig, vor allem, wenn die Zahl der Bücher wächst.

Sehen Sie sich noch ein weiteres Beispiel mit mir an, nämlich die Berechnung der Summe aller natürlichen Zahlen von 1 bis zu einer vorgegebenen Obergrenze n. Nachfolgend sind zwei Varianten dargestellt:

```
static double BerechneSummeIterativ(double n)
{
    double summe = 0;          // 1 Schritt

    for (double i = 1; i <= n; i++)  // n Schritte
    {
        summe += i;            // 1 Schritt
    }

    return summe;              // 1 Schritt
}
```

```
static double BerechneSummeGauss(double n)
{
    double summe = n * (n + 1) / 2.0; // 1 Schritt

    return summe;                     // 1 Schritt
}
```

In der linken Variante wird mit Hilfe einer Zählschleife iterativ der Wert der Summe berechnet. In der rechten Variante wird die Gaußsche Summenformel verwendet.

Verwendet man eine Stoppuhr, um die Laufzeiten bei unterschiedlichen Werten für n zu messen, stellt man fest, dass die Laufzeit der Gauß-Variante sehr konstant (und fast nicht messbar) ist. Bei der iterativen Variante wird man hingegen eine nahezu lineare Abhängigkeit der Laufzeit von n messen.

Bei n=10.000 benötigt der iterative Algorithmus ca. 28,2µs, bei n=100.000 ca. 279,7µs, bei n= 1000.000 ca. 27951µs usw. bis zu ca. 2,738s bei n=1.000.000.000.

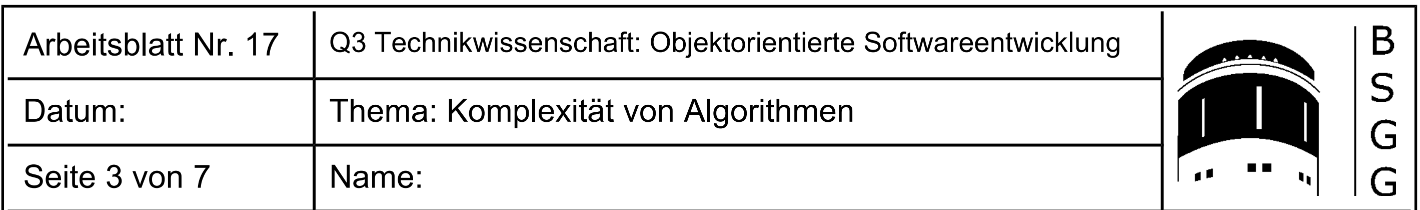
Nimmt man nun die Schritte, um eine Zeitfunktion aufzustellen, erhält man bei der linken Methode: $T(n) = n \cdot 1 + 2$.

Die Methode rechts liefert: $T(n) = 2$. Man könnte die Gleichung auch in mehrere Teilschritte zerlegen. Aber es bleibt immer eine feste ganze Zahl von Teilschritten und diese hängt offensichtlich nicht von n ab.

Hier ist klar erkennbar, dass die Effizienz des rechten Algorithmus deutlich besser ist!

Ähnliches lässt sich zu den beiden Suchverfahren ermitteln. Die lineare Suche enthält eine Schleife, die über alle Elemente des Array iteriert. In der Schleife sind einige Anweisungen auszuführen und vor sowie hinter der Schleife sind einige Anweisungen auszuführen.

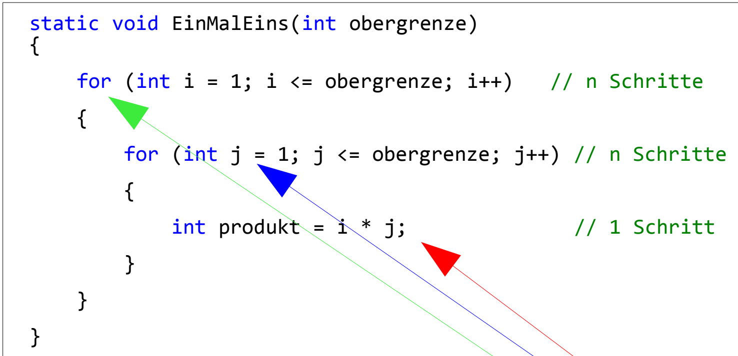
```
C:\Windows\system32\cmd.exe
n = 1000
00:00:00.0000036
Iterative Summe = 500500

00:00:00.0000003
Gauss Summe = 500500
====================================
n = 10000
00:00:00.0000282
Iterative Summe = 50005000

00:00:00.0000003
Gauss Summe = 50005000
====================================
n = 100000
00:00:00.0002797
Iterative Summe = 5000050000

00:00:00
Gauss Summe = 5000050000
====================================
n = 1000000
00:00:00.0027951
Iterative Summe = 500000500000

00:00:00
Gauss Summe = 500000500000
====================================
n = 10000000
00:00:00.0296272
Iterative Summe = 50000005000000

00:00:00.0000003
Gauss Summe = 50000005000000
====================================
n = 100000000
00:00:00.2826992
Iterative Summe = 5,00000005E+15

00:00:00
Gauss Summe = 5,00000005E+15
====================================
n = 1000000000
00:00:02.7387183
Iterative Summe = 5,00000000067109E+17

00:00:00.0000003
Gauss Summe = 5,000000005E+17
====================================
Drücken Sie eine beliebige Taste . . .
```



Somit ergibt sich für die lineare Suche eine Zeitfunktion $T(n) = c_1 \cdot n + c_2$, wobei c_1 die Anzahl der Schritte innerhalb der Schleife umfasst und c_2 die Anzahl der Schritte vor und hinter der Schleife.

Eine Übung wäre nun, zu ermitteln, wie die Zeitfunktion T(n) für die „Binäre Suche“ lautet.

Noch ein abschließendes Beispiel, nämlich die Berechnung eines „1 x 1“. Üblicherweise diese Berechnung für den Wertebereich von 1 bis 10 durchgeführt, also alle Produkte von 1 x 1 bis 10 x 10. Aber in dem Beispiel ist die Obergrenze nun variablel und wird immer um den Faktor 10 vergrößert.



```
static void EinMalEins(int obergrenze)
{
    for (int i = 1; i <= obergrenze; i++)    // n Schritte
    {
        for (int j = 1; j <= obergrenze; j++) // n Schritte
        {
            int produkt = i * j;              // 1 Schritt
        }
    }
}
```

Die Zeitfunktion hier ist einfach: Die äußere Schleife wird n-mal bis zur Obergrenze durchlaufen, die innere Schleife ebenfalls und in der inneren Schleife fällt ein Berechnungsschritt -wie lange der auch dauern mag- an. Also lautet die Zeitfunktion $T(n) = n \cdot (n \cdot 1) = n^2$.

Das Kommandozeilen-Fenster zeigt die Ergebnisse der Zeitmessungen. Bildet man nun die Quotienten der jeweils aufeinanderfolgenden Zeitmessungen, ergeben sich folgende Werte:

```
C:\Windows\system32\cmd.exe
bis 10: Laufzeit = 00:00:00.0000676
bis 100: Laufzeit = 00:00:00.0000883
bis 1000: Laufzeit = 00:00:00.0021224
bis 10000: Laufzeit = 00:00:00.2171951
bis 100000: Laufzeit = 00:00:21.3602657
bis 1000000: Laufzeit = 00:35:38.9651748
Drücken Sie eine beliebige Taste . . .
```

```
676
883
21224
2171951
213602657
21389651748

21224/883 = 24,036
2171951/21224 = 102,33
213602657/2171951 = 98,345
21389651748/213602657 = 100,137
```

Es zeigt sich, dass bei kleinen Werten für die Obergrenze die Quotienten noch kleine Werte liefern. Doch bereits der Quotient der Zeiten für die Obergrenzen 10.000 und 1.000 bzw. 100.000 und 10.000 hat einen Wert in der Nähe von 100. Bei einer Vergrößerung der Obergrenze um den Faktor 10 erhält man also eine um den Faktor 100 vergrößerte Berechnungszeit.

Für große Werte von n gilt: $T(n) \sim n^2$

Landau-Symbole[1]

In der Komplexitätstheorie werden die sogenannten Landau-Symbole verwendet. Das Symbol *O* mit der Bezeichnung „Big-O“ oder „Big-Oh“, wobei dies der große griechische Buchstabe Omikron[2] ist, ist eines der Landau-Symbole.

Fernerhin existieren das kleine Omikron (*o*), benannt als „Little-O“, das kleine (ω) und das große Omega (Ω) sowie das große Theta (Θ).

Für die Analyse von Algorithmen sind das Big-Oh, das Big-Omega und das Big-Theta von Bedeutung.

1 Sehr ausführlich in `https://michaelgoerz.net/studies/semester01/comp_sci2/inf_skript.pdf` abgerufen am 5.Januar 2019

2 Siehe das griechische Alphabet `https://de.wikipedia.org/wiki/Griechisches_Alphabet` abgerufen am 5. Januar 2019

Diese Landau-Symbole bezeichnen **Mengen von Funktionen**, für die bestimmte Eigenschaften definiert sind[3]:

Notation	$x \to \infty$
$f \in o(g)$	$\forall\, C > 0\, \exists\, x_0 > 0\, \forall\, x > x_0 : \lvert f(x)\rvert < C \cdot \lvert g(x)\rvert$
$f \in \mathcal{O}(g)$	$\exists\, C > 0\, \exists\, x_0 > 0\, \forall\, x > x_0 : \lvert f(x)\rvert \le C \cdot \lvert g(x)\rvert$
$f \in \Theta(g)$	$\exists\, c > 0\, \exists\, C > 0\, \exists\, x_0 > 0\, \forall\, x > x_0 : c \cdot \lvert g(x)\rvert \le \lvert f(x)\rvert \le C \cdot \lvert g(x)\rvert$
$f = \Omega(g)$	(Zahlentheorie) $\exists\, c > 0\, \forall\, x_0 > 0\, \exists\, x > x_0 : c \cdot g(x) \le \lvert f(x)\rvert$ (die Test-Funktion g ist immer positiv)
$f \in \Omega(g)$	(Komplexitätstheorie) $\exists\, c > 0\, \exists\, x_0 > 0\, \forall\, x > x_0 : c \cdot \lvert g(x)\rvert \le \lvert f(x)\rvert$
$f \in \omega(g)$	$\forall\, c > 0\, \exists\, x_0 > 0\, \forall\, x > x_0 : c \cdot \lvert g(x)\rvert \le \lvert f(x)\rvert$

Kurze Vorbemerkung: Zuvor wurde immer von der Eingabegröße n gesprochen. In der Tabelle wird aber x verwendet. x ist gleich zusetzen mit n!

Diese Funktionen sind mit der Quantoren-Schreibweise[4] definiert. Das sieht zwar kompliziert aus, ist aber eigentlich nicht sooo schwer;). Es gibt nur die beiden Symbole $\exists$ und $\forall$. Das erste ist der sogenannte Existenzquantor und das zweite der sogenannte Allquantor.

Sehen Sie sich mit mir die zweite Zeile in der Tabelle an. Der erste Ausdruck lautet $\exists C > 0$. Das bedeutet „es existiert mindestens ein C mit einem Wert größer 0“. Entsprechend bedeutet der zweite Ausdruck $\exists x_0 > 0$ „es existiert mindestens ein x_0 mit einem Wert größer 0“. Und der dritte Ausdruck $\forall\, x > x_0$ bedeutet „für alle x größer x_0“.

Somit bedeutet die ganze Zeile: *„Es existiert mindestens ein C mit einem Wert größer 0 und es existiert mindestens ein x_0 mit einem Wert größer 0 und für alle x größer x_0 gilt, eine Betragsfunktion f(x) ist immer kleiner oder gleich einer Betragsfunktion C · g(x)*“. In der ersten Spalte steht dann, dass die Funktion f ein Element der Menge aller Funktionen g ist, für die diese Definition gilt :)

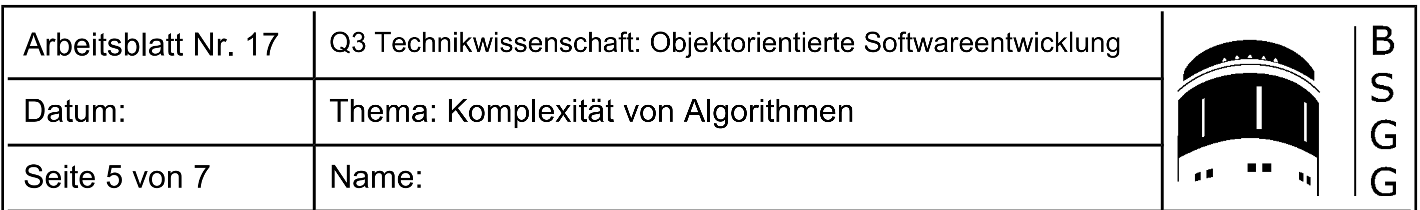
In analoger Weise lassen sich nun auch die Definitionen für das große Theta oder für das große Omega lesen.

$f \in \Theta(g)$ *„Es existiert mindestens ein c > 0 und es existiert mindestens ein C > 0 und es existiert mindestens ein x_0 > 0 und für alle x > x_0 gilt, die Funktion f(x) ist größer/gleich der Funktion c · g(x) und kleiner/gleich der Funktion C · g(x)*“. Dann ist die Funktion f ein Element der Menge aller Funktionen g.

Und wie lautet die die Definition für $f \in \Omega(g)$? Nutzen Sie den Platz :)

3 Entnommen aus `https://de.wikipedia.org/wiki/Landau-Symbole` abgerufen am 5. Januar 2019.
Allerdings fehlen in dieser Tabelle die Wertebereiche für die Konstante c bzw. x/x_0: $c \in \mathbb{R}$ *und* $x \in \mathbb{N}$

4 `https://de.wikipedia.org/wiki/Quantor` abgerufen am 5. Januar 2019

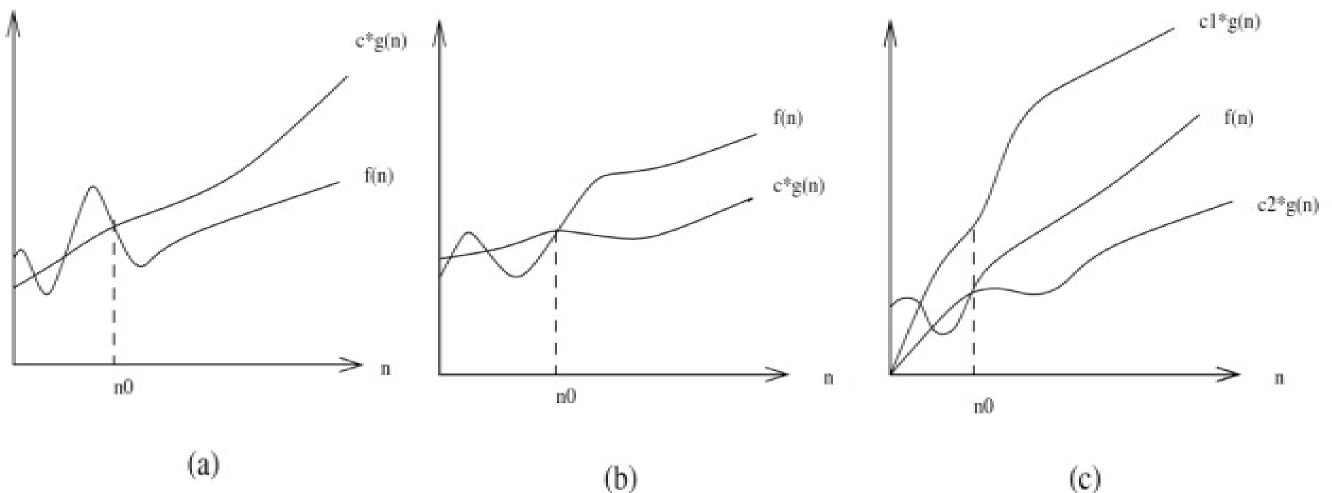


Was sagt uns das nun?

$\Omega(g),\ \Theta(g)\, und\, O(g)$ sind Mengen von Funktionen mit bestimmten Eigenschaften. Bei dem „Big-Oh“ wächst die Funktion f(x) immer langsamer oder gleich schnell wie die Funktionen C · g(x). Bei dem dem Big-Omega wächst eine Funktion f immer genau so schnell oder schneller wie die Funktionen c · g(x). Und beim Big-Theta liegt f(x) zwischen den Funktionen c · g(x) und C · g(x).

Somit werden Wachstums-Schranken definiert, die von der Funktion f(x) entweder nicht überschritten („Big-Oh“) oder nicht unterschritten („Big-Omega“) werden oder zwischen denen sich f(x) befindet („Big-Theta“).

Die folgenden drei Graphen[5] sollen das nochmal verdeutlichen:



Illustrating the big (a) O, (b) Ω, and (c) Θ notations

Von überwiegender Bedeutung ist jedoch das Big-Oh als obere Schranke für die Komplexität eines Algorithmus.

Dominanz der Funktion T(n)

Wir ersetzen nun die allgemeine Funktion f(n) durch die Zeitfunktion T(n). Die Definition von „Big-Oh“ besagt nun: $T(n) \leq c \cdot g(n)$ ab irgendeinem n_0.

Um nun die beiden Funktionen T(n) und g(n) miteinander zu vergleichen, wird der Quotient gebildet und dessen Grenzwert bestimmt.

$$\lim_{n\to\infty} \frac{T(n)}{g(n)}$$

Existiert ein Grenzwert größer 0, dann dominiert T(n) die Funktion g(n). Ist der Grenzwert 0, dann wächst g(n) schneller als T(n) und die Abschätzung für g(n) ist zu hoch. Ist der Grenzwert jedoch unendlich, dann ist g(n) zu klein gewählt.

Beispiele

T(n) = 3 · n + 5 = O(n): $\lim_{n\to\infty} \frac{3\cdot n + 5}{c\cdot n} = \frac{3}{c}$

T(n) = 2 · n² + 10 · n + 5 = O(n³): $\lim_{n\to\infty} \frac{2\cdot n^2 + 10\cdot n + 5}{c\cdot n^3} = 0$

5 Entnommen aus `http://www.basicsbehind.com/tag/analysis-of-algorithms/` abgerufen am 6. Januar 2019

© Uwe Homm Version vom 6. Januar 2019

T(n) = 2 · n² + 10 · n + 5 = O(n): $\lim_{n \to \infty} \frac{2 \cdot n^2 + 10 \cdot n + 5}{c \cdot n} = \infty$

Abschätzung von *O*(n)

Wie man sieht, ist in T(n) der Term mit der höchsten Potenz bedeutsam und legt damit die Klasse von *O* fest. Diejenigen Terme, die bei der Grenzwertbildung gegen 0 streben, können entfallen.

Folgende Klassifikationen sind gegeben: *O(n³), O(n · log n), O(n), O(2 ⁿ) O(√n), O(5), O(log n)*

Ordnen Sie die Klassen in aufsteigender Reihenfolge. Nutzen Sie hierzu z.B. folgenden Link:

`https://de.wikipedia.org/wiki/Landau-Symbole#Beispiele_und_Notation`

Notation	Bedeutung
$T(n) \in O(1)$	T(n) ist beschränkt

Eine Einführung in die Landau-Symbole mit mehreren kleinen Auswahltests findet sich unter:

`https://de.khanacademy.org/computing/computer-science/algorithms`

Abschließender Hinweis

Oftmals findet sich die Schreibweise $T(n) = O(g)$ im Unterschied zu $T(n) \in O(g)$. Dies ist mathematisch eigentlich inkorrekt, hat sich aber „eingebürgert“.

$O(g)$ ist ja eine Menge von Funktionen und damit ist das mathematische Gleichheitszeichen nicht richtig. Eine einzelne Funktion kann ja nicht mit einer Menge von Funktionen gleichgesetzt werden.

Ganz falsch wäre es aber, wenn die Seiten vertauscht würden: $O(g) = T(n)$

Diese Schreibweise ist gänzlich falsch! Näheres z.B. unter **`https://michaelgoerz.net/studies/semester01/comp_sci2/inf_skript.pdf`** auf Seite 10f.

In dem Graph sind sieben Funktionen mit ihrem Wachstum für $n \rightarrow \infty$ dargestellt:

- $f_0(n) = 1$
- $f_1(n) = \log_2 n$
- $f_2(n) = \sqrt{n}$
- $f_3(n) = n$
- $f_4(n) = n \cdot \log_2 n$
- $f_5(n) = n^2$
- $f_6(n) = 2^n$

Identifizieren Sie die Funktionen!