Chair of Embedded Systems
Department of Computer Science
University of Augsburg, Germany

Martin Frieb

Universität Augsburg University

Wintersemester 2020/2021

Informatik 1

Kapitel 13 – Effizienz von Algorithmen

Inhaltsverzeichnis

Version 20210202-942

13.1 Motivation

Das Problem des Handlungsreisenden
Manche Probleme sind theoretisch durch Algorithmen lösbar, allerdings insbesondere für große Instanzen nicht praktikabel.

Beispiel: Das Problem des Handlungsreisenden

Eingabe: n Städte mit ihren wechselseitigen Entfernungen ($n > 2$)
Ausgabe: Die kürzeste Rundreise durch diese Städte

Die Aufgabe des Handlungsreisenden (engl.: *Traveling Salesman Problem (TSP)*) ist eines der bekannteren Probleme und ein gutes Beispiel für ein komplexes Problem.
Wir müssen n viele Städte besuchen, um dort unsere Wahren zu verkaufen bzw. einzukaufen. Allerdings gilt: Zeit ist Geld. Deswegen wollen wir den optimalsten Weg zwischen allen Städten suchen. Das Ziel ist, jede Stadt nur einmal zu besuchen UND dabei den kürzesten Weg zwischen den Städten zu finden.

Deswegen ist unsere Aufgabe: Das finden eines Rundweges, sodass wir jede Stadt nur einmal besuchen müssen und der am Ende gefahrene Weg minimal ist.

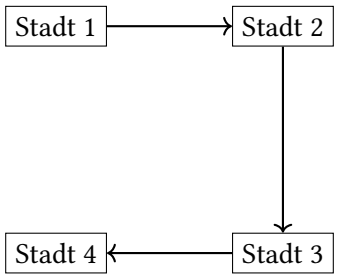

Abbildung 1: Möglicher optimaler Weg

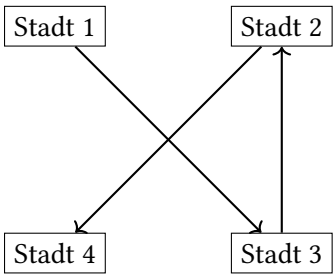

Abbildung 2: **Kein** optimaler Weg

Wie wird dieses Problem gelöst?
Durchlaufe systematisch in einer Schleife alle möglichen Rundreisen und speichere die kürzesten Wege.
Das ist aber **keine effiziente Lösung**:
Für n Städte gibt es $n!$ mögliche Rundreisen. Schon für $n > 30$ braucht der Rechner sehr lange (mehrere Tage), um alle Rundreisen zu durchlaufen.

Suche in sortierter Folge
Ein anderes (häufiger) auftretende Problem ist das Sortieren einer Folge von Elementen. Dieses Problem kann von verschiedenen Algorithmen gelöst werden, die unterschiedlich viel Zeit benötigen.

Beispiel: Sortieren einer Folge

Sei nun beispielsweise folgende Zahlenfolge gegeben: $[4.12, 1.0, 6.123, 3.61, 8.246]$

Die Aufgabe ist nun: Sortiere diese Zahlenfolge.

Als Mensch ist das eine sehr einfache Aufgabe. Wir schauen uns die Zahlen an und sortieren sie einfach. Eine Maschine jedoch kann nicht einfach draufschauen und diese Liste sortieren. Wir müssen ihr genau sagen, wie sie das zu lösen hat.

Ein Beispiel für solch einen *Sortieralgorithmus* wäre Bubblesort. Der Algorithmus ist einfach zu implementieren, benötigt jedoch bei n Elementen in der Liste n^2 viele Durchgänge durch die Liste. Bei 10 Elementen müssten wir also diese Liste $10^2 = 100$ mal durchgehen. Bei 1.000 Elementen wären das schon $1.000^2 = 1.000.000$ Durchgänge.

Ein anderes Beispiel wäre *Heapsort*. Dieser benötigt schon ein bisschen mehr Erfahrung, um richtig implementiert zu werden. Es ist jedoch schneller, denn es benötigt bei n Elementen in der Liste nur $n \cdot \log(n)$ viele Durchgänge zum sortieren (nach aktueller Forschung die wenigste Anzahl an Durchgängen, um etwas zu sortieren). Bei einer Liste mit 1.000 Elementen wären das ungefähr 6.900 Durchgänge, also viel weniger als bei Bubblesort.[1]

Beispiel: Suche Einfügeposition in einer sortierten Folge

Eingabe: $a_1, \ldots, a_n$ mit $\forall i \in \{1, 2, ..., n-1\}(a_i \leqslant a_{i+1})$ und ein Suchwert a
Ausgabe: Die größte Zahl $k \in \{1, \ldots, n+1\}$ mit $\forall i \in \{1, 2, ..., k-1\}(a > a_i)$.

Wir besitzen hier eine sortierte Liste an Zahlen (Bspw.: $[1.1, 1.2, 3.65, 24.7155.0]$).
Zusätzlich gibt es einen Suchwert a, der in die Liste eingefügt werden soll (Bspw.: $a = 6.0$).

Wir wollen aus unserer ursprünglichen Liste alle Werte finden, die kleiner sind als a (Also: $[1.1, 1.2, 3.65]$).

Auch für diese Aufgabe gibt es verschiedene Algorithmen:

- Sequentieller Suchalgorithmus (siehe Kapitel 11):
  a muss im schlechtesten Fall mit allen n Folgenelementen verglichen werden (1000 Vergleiche für $n = 1000$)
- Binärer Suchalgorithmus:
  a muss im schlechtesten Fall nur mit $\log(n)$ Folgenelementen verglichen werden (($\log(1000) \approx 10$ Vergleiche für $n = 1000$))

Der binäre Suchalgorithmus ist **effizienter** als der sequentielle Suchalgorithmus

Fazit
Für die Praxis spielt der **Zeitbedarf** eines Algorithmus eine wichtige Rolle. Es macht einen großen Unterschied, ob wir n^2-viele Vergleiche machen müssen, oder nur $n \cdot \log(n)$ (Vergleiche Beispiel Bubblesort – Heapsort).

[1] Sortieralgorithmen gehören nicht zur Vorlesung Informatik 1, sind aber ein wichtiges Forschungsgebiet in der Informatik. Möchte man mehr über solche Algorithmen und deren Funktionsweise erfahren, wird auf die Vorlesung *Informatik 3* verwiesen. Zusätzlich gibt es im Internet auch viele informative Visualisierungen der meisten Sortieralgorithmen. Hierfür reicht es, in einer Suchmaschine *sorting algorithms visualized* einzugeben.

In diesem Kapitel:
Wir geben nun einen Einblick in die Berechnung und Abschätzung des (maximalen) Zeitbedarfs eines konkret gegebenen Algorithmus für kleine und große Probleminstanzen.

Ausblick:

- Neben dem Zeitbedarf betrachtet man auch den **Speicherbedarf** .
- Entwicklung von Algorithmen, die ein vorgegebenes Problem mit möglichst geringem Zeit- **und** Speicherplatzverbrauch lösen.

Diese Themen und viel mehr wird unter anderem in den Vorlesungen *Informatik 3* und *Einführung in die theoretische Informatik* vertieft.

13.2 ELOPs

Wir führen nun *Elementaroperationen* ein (kurz: ELOPs). Das sind Operationen, die (in der Informatik 1) genau **einen** Zeitschritt benötigen.

Sie sind nötig, um über den Zeitbedarf, Laufzeit, etc. sprechen zu können.

Einflussfaktoren auf den Zeitbedarf eines Algorithmus

Der gesamte Zeitbedarf für die Lösung einer Probleminstanz entspricht der Summe der Zeitbedarfe der auszuführenden (Rechen-)Operationen. Er hängt damit ab von:

- **der Anzahl der auszuführenden Operationen**:
  Diese Zahl hängt in der Regel von der **Problemgröße** (siehe unten) und der **Abstraktionsebene der Operationen** (Maschinen-, Programmiersprachen-, Algorithmusebene) ab.
- **dem Zeitbedarf für die Ausführung der einzelnen Operationen**:
  Dieser Zeitbedarf hängt von der eingesetzten **Rechenanlage** und der benutzten **Programmiersprache** ab.

Problemgröße:

In dieser Vorlesung betrachten wir als **Problemgröße**:

- entweder die **Anzahl der Eingabedaten**
  Beispiel: Bei der sequentiellen Suche müssen um so mehr Vergleiche durchgeführt werden, je mehr Elemente die Folge hat.
- oder die **Größe eines Eingabewerts**
  Beispiel: Bei der Berechnung der Fakultät von n müssen umso mehr Schleifendurchläufe durchgeführt werden, je größer n ist.

Berechnung des Zeitbedarfs eines Algorithmus
Bei der Berechnung des **Zeitbedarfs** abstrahieren wir von der eingesetzten Programmiersprache und Rechenanlage und nehmen an, dass jede Operation **den gleichen Zeitbedarf** hat (Rechtfertigung für dieses Vorgehen am Ende des Kapitels).
Unter diesen Voraussetzungen können wir den Zeitbedarf berechnen, indem wir den Algorithmus in einer **formalen** programmiersprachenunabhängigen Darstellung betrachten (Pseudocode, Struktogramm, Programmablaufplan) und **die Anzahl der auszuführenden Operationen in Abhängigkeit von der Problemgröße zählen**.

Ausblick weiterführende Vorlesungen:
In den Vorlesungen *Informatik 3* und *Einführung in die Theoretische Informatik* nutzt man auch abstrakte Rechnermodelle (Random-Access-Maschine (RAM), Turing-Maschine, ...) anstatt programmiersprachenunabhängigen Darstellungen von Algorithmen für Effizienzbetrachtungen.

13.2.1 Was ist eine ELOP?

Wir betrachten ab jetzt Algorithmen auf der Abstraktionsebene einer **formalen programmiersprachenunabhängigen Darstellung** und die auf dieser Abstraktionsebene auszuführenden Operationen als **Elementaroperationen** (**ELOPs**). Wir zählen die Anzahl der auszuführenden ELOPs zur Bearbeitung einer Probleminstanz.

Das sind ELOPs:

- Wertzuweisungen ($\leftarrow$)
- Arithmetische Operationen ($+, -, /, *, \div$, mod)
- Vergleiche von Werten ($\leqslant, \geqslant, <, >, =, \neq$)
- Einfache mathematische Operationen ($|\cdot|$)
- Logische Operationen ($\neg, \wedge, \vee$)

Beispiel:

Es gilt zu beachten: Jede einzelne dieser oben genannten Operationen ist eine ELOP. D.h. folgende Zeilen entsprechen jeweils einer ELOP:

```
1 x ← 0;
2 ...;
3 wenn a = b dann
4     ...;
```

Ein häufiger Fehler, den viele am Anfang machen, ist das falsche Zählen von ELOPs. Verknüpft man nämlich in einer Zeile mehrere Operationen, entspricht diese Zeile mehreren ELOPs.

Folgende Zeilen entsprechen jeweils zwei ELOPs:

```
1 x ← a + b;
2 ...;
3 wenn |a = b| dann
4 |   ...;
```

Oder drei ELOPs:

```
1 x ← (a + b) mod 3;
```

Das sind keine ELOPs:
Aufwendige mathematische Operationen / Überprüfungen (z.B Matrizenmultiplikation, Elementbeziehung, Fakultätsfunktion, Mengenoperationen) und **natürlichsprachliche Anweisungen**.

Zusätzlich gilt: In dieser Vorlesung werden Ein- und Ausgabe **nicht** für die Anzahl der ELOPs betrachtet. Sie fließen also nicht in die Berechnung der ELOPs mit ein.

13.2.2 Den Zeitbedarf berechnen: ELOPs zählen

Definition: 13.3 Zeitbedarf (formal)

Der **Zeitbedarf** $t(A, e)$ eines Algorithmus A angewendet auf eine Eingabe e ist die Anzahl der ausgeführten ELOPs ohne Berücksichtigung von Eingabe und Ausgabe.

Zur Berechnung des Zeitbedarfs muss insbesondere die **Anzahl der Schleifendurchläufe** bestimmt werden.

Beispiel: Summe einer Zahlenfolge berechnen

```
Eingabe: x_1, ..., x_n ∈ ℤ, n ∈ ℕ
1 s ← 0;
2 i ← 1;
3 solange (i ⩽ n) tue
4 |   s ← s + x_i;
5 |   i ← i + 1;
Ausgabe: s
```

Problemgröße: n
Die Anzahl der ELOPs hängt nur von n ab, aber nicht von der Größe der Zahlen $x_1, \ldots, x_n$ (es macht also keinen Unterschied, ob bspw. $x_1 = 1$ oder $x_1 = 1.000.000$)

Anzahl der Schleifendurchläufe: n
Auch die Anzahl der Schleifendurchläufe ist unabhängig von der Wahl von $x_1, \ldots, x_n$

Die Schleifenbedingung wird $n + 1$-mal überprüft

Zusammengefasst können wir nun die **ELOPs** angeben:
1 (Wertzuweisung $s \leftarrow 0$)
1 (Wertzuweisung $i \leftarrow 1$)
$1 \cdot (n + 1)$ (Vergleich $i \leqslant n$)
$2 \cdot n$ (Addition $s + x_i$ und Wertzuweisung $s \leftarrow \ldots$)
$2 \cdot n$ (Addition $i + 1$ und Wertzuweisung $i \leftarrow \ldots$)
Anzahl der ELOPs: $5 \cdot n + 3$

Beispiel: Sequentielle Suche in einer sortierten Folge

Das obere Beispiel besitzt **immer** die gleiche Anzahl an ELOPs (immer $5 \cdot n+3$). Es gibt aber auch Algorithmen, die unterschiedlich viele ELOPs ausführen können. Hierbei hängt die Anzahl von der Eingabe ab.

```
Eingabe: x_1, ..., x_n ∈ ℤ, ∀i ∈ {1, 2, ..., n − 1}(x_i ⩽ x_{i+1}), n ∈ ℕ, x ∈ ℤ
1 i ← 1;
2 solange (i ⩽ n) tue
3     wenn x ⩽ x_i dann
          Ausgabe: i
4     i ← i + 1;
Ausgabe: i
```

ELOPs:

Zeile 1: 1

Zeile 2: Minimal: 1, Maximal: $1 \cdot (n + 1)$
Grund: Ist $x \leqslant x_1$, dann wird diese Bedingung nur einmal überprüft, da dieser Code durch die Ausgabe beendet wird. Somit haben wir minimal eine ELOP. Ansonsten wird diese Bedingung bis zu $n + 1$-mal überprüft (Maximalfall).

Zeile 3: Minimal: 1, Maximal: $1 \cdot n$ (Grund: Analog zu oben)

Zeile 4: Minimal: 0, Maximal: $2 \cdot n$
Grund: Wird in der ersten Iteration die Ausgabe ausgeführt, wird diese Anweisung nie ausgeführt. Deswegen haben wir minimal 0 Ausführungen. Ansonsten wird diese Zeile bis zu n-mal ausgeführt. Da die Zuweisung und die Addition jeweils eine ELOP brauchen, benötigt diese Zeile 2 ELOPs.

Anzahl der ELOPs:
Minimal: 3**, Maximal:** $4 \cdot n + 2$
Problemgröße: n
Die Problemgröße betrachtet immer nur die maximale Anzahl an ELOPs. Hierfür werden Konstanten (die 4 oder die $+2$) weggelassen, sodass wir nur eine Abhängigkeit von der Variable angeben.

Wir sehen also:
Die Anzahl der Schleifendurchläufe hängt vom Wert x ab und die Anzahl der ELOPs muss nicht für jede Eingabe der Größe n gleich sein!

Hier gilt zu beachten: Ein häufiger Fehler der gemacht wird, bzw. am Anfang verwirrend sein kann, ist die Angabe der minimalen bzw. maximalen Anzahl der ELOPs. Im zweiten Beispiel ist minimale Anzahl $\neq$ maximale Anzahl. Das liegt daran, dass die Schleife frühzeitig abgebrochen werden kann.

Im ersten Beispiel gilt jedoch: minimale Anzahl $=$ maximale Anzahl an ELOPs. Die Schleife kann nicht frühzeitig abgebrochen werden, wodurch sie immer nur von n abhängt (unabhängig davon, ob $n = 1$ oder $n = 1000$ ist).

Die Bestimmung der Anzahl der Schleifendurchläufe ist nicht immer so einfach, wie folgendes Beispiel zeigt:

Beispiel: Matrixoperation

Eingabe: $(m_{i,j})_{1 \leqslant i,j \leqslant n} \in (\mathbb{Z}^n)^n, n \in \mathbb{N}$

```
1 i ← 1;
2 s ← 0;
3 solange (i ⩽ n) tue
4 |   j ← 1;
5 |   solange (j < i) tue
6 |   |   s ← s + m_{i,j};
7 |   |   j ← j + 1;
8 |   i ← i + 1;
```

Ausgabe: s

ELOPs:

Zeile 1: 1
Zeile 2: 1
Zeile 3: $1 \cdot (n+1)$
Zeile 4: $1 \cdot n$
Zeile 5: $1 \cdot (1 + 2 + \ldots + n)$
Zeile 6: $2 \cdot (1 + 2 + \ldots + n - 1)$
Zeile 7: $2 \cdot (1 + 2 + \ldots + n - 1)$
Zeile 8: $2 \cdot n$
Ergebnis: $2.5 \cdot n^2 + 2.5 \cdot n + 3$

Die Anzahl der Durchläufe der inneren Schleife ist von der äußeren Schleife abhängig: Wenn $i = 1$, wird sie nicht durchlaufen, für $i = 2$ einmal, für $i = 3$ zweimal usw. Allgemein können wir sagen, dass im k-ten Durchlauf der **äußeren** Schleife die **innere** Schleife $k - 1$-mal durchlaufen wird. Daher haben wir beim ELOPs zählen dort die Summe von 1 bis n bzw. $n - 1$ stehen. Aus der Mathematik wissen wir, dass wir das Ergebnis einer solchen Summe einfach ausrechnen können mit $\sum_{k=1}^{n} k = \frac{n \cdot (n+1)}{2}$ (Gaußsche Summenformel).

Somit ist die **Problemgröße:** n^2 (Für die Problemgröße werden nicht nur Konstanten weggelassen. Wir betrachten nur die Variable mit dem höchsten Exponenten).

13.3 Zeitkomplexität

Wir wollen nun formeller angeben können, wie viel Zeit ein Algorithmus benötigt. Hierfür definieren wir uns eine *Best-Case-* sowie eine *Worst-Case-Komplexität*. Im Grunde machen wir jedoch nichts neues: Wir zählen immernoch ELOPs.

Definition: 13.7 Zeitkomplexität

Es bezeichne $|e|$ die Größe einer Eingabe e und A einen Algorithmus.

- Die **Best-Case-Komplexität** ist die Funktion $T_A^{min} : \mathbb{N} \to \mathbb{N}$ mit

$$T_A^{min}(n) := min\{t(A, e) \mid |e| = n\}$$

  Anwendung: Gibt eine **untere Schranke** für den Zeitbedarf an.
- Die **Worst-Case-Komplexität** ist die Funktion $T_A^{max} : \mathbb{N} \to \mathbb{N}$ mit

$$T_A^{max}(n) := max\{t(A, e) \mid |e| = n\}$$

  Anwendung: Gibt eine **obere Schranke** für den Zeitbedarf an.

Wir sehen also: Im Grunde geben wir weiterhin die minimale bzw. maximale Anzahl an ELOPs an. Dabei gelten für $T_A^{min}(n)$ und $T_A^{max}(n)$ die selben Regeln wie für das Zählen der ELOPs.

Beispiel:

Eingabe: $x_1, \ldots, x_n \in \mathbb{Z}, n \in \mathbb{N}$

```
1 s ← 0;
2 i ← 1;
3 solange (i ⩽ n) tue
4 |   s ← s + x_i;
5 |   i ← i + 1;
```

Ausgabe: s

ELOPs:
- Zeile 1: 1 (Wertzuweisung)
- Zeile 2: 1 (Wertzuweisung)
- Zeile 3: $1 \cdot (n + 1)$ (Vergleich)
- Zeile 4: $2 \cdot n$ (Addition + Wertzuweisung)
- Zeile 5: $2 \cdot n$ (Addition + Wertzuweisung)

$T_A^{min}(n) = T_A^{max}(n) = 5 \cdot n + 3$

Beispiel:

Eingabe: $x_1, \ldots, x_n \in \mathbb{Z}, \forall i \in \{1, 2, ..., n-1\}(x_i \leqslant x_{i+1}), n \in \mathbb{N}, x \in \mathbb{Z}$

```
1 i ← 1;
2 solange (i ⩽ n) tue
3     wenn x ⩽ x_i dann
          Ausgabe: i
4     i ← i + 1;
Ausgabe: i
```

ELOPs:
Zeile 1: 1
Zeile 2: Minimal: 1, Maximal: $1 \cdot (n+1)$
Zeile 3: Minimal: 1, Maximal: $1 \cdot n$
Zeile 4: Minimal: 0, Maximal: $2 \cdot n$

$T_A^{min}(n) = 3$
$T_A^{max}(n) = 4 \cdot n + 2$

13.3.1 Größenordnungen für Zeitkomplexität

In der Praxis interessiert man sich allerdings **nicht für die genaue Anzahl** der ELOPs, sondern für die **Größenordnung** des **Wachstums der Anzahl der ELOPs mit Erhöhung der Problemgröße**. Das bedeutet eine genaue Angabe von $T_A^{max}(n)$, wie wir sie bisher angegeben haben, ist oftmals nicht interessant. Was jedoch interessant ist, ist die Problemgröße (z.B. wenn $T_A^{max}(n) = 4 \cdot n + 2$, folgt daraus die Problemgröße n).

Um die Problemgröße, bzw. dieses Wachstum zu beschreiben, wird die sogenannte **O-Notation** eingeführt (Sprechweise: **"Groß-Oh-Notation"**).

Definition: 13.8 O-Notation

Seien P die Menge aller Funktionen $f : \mathbb{N} \to \mathbb{R}^+$, $g, h \in P$, $n, n_0 \in \mathbb{N}$ und $c \in \mathbb{R}^+$. Wir definieren die Menge **"Groß-Oh von g"** $O(g)$:

$$O(g) := \{h \mid \exists n_0(\exists c(\forall n > n_0(h(n) \leqslant c \cdot g(n))))\}$$

Sie gibt im Grunde nichts anderes an, als die Problemgröße eines Algorithmus. Das heißt, ist die Problemgröße beispielsweise n^2, dann sagen wir: Die Funktion ist aus $O(n^2)$.

Formell beschreibt diese Funktion eine Menge an Funktionen. Diese Menge kann man sich folgendermaßen vorstellen:
Wir besitzen eine feste Funktion g. In der Menge $O(g)$ befinden sich alle Funktionen, die, würden wir sie in einem Koordinatensystem einzeichnen, langsamer steigen als g.

Möchten wir zeigen, dass eine Funktion $h \in O(g)$ ist, müssen wir ein n_0 sowie ein c finden, so dass für genügend große $n > n_0$ gilt: $h(n) \leqslant c \cdot g(n)$, d.h. unser h in einem Koordinatensystem langamer steigt als g und somit stets unter g liegt.

Um das Gegenteil zu beweisen, also $h \notin O(g)$, reicht ein Widerspruchsbeweis. Im Kapitel 13.3.2 folgen dazu einige Beispiele.

Interpretation und Sprechweisen:
Eine Funktion $h \in O(g)$ wächst für große n höchstens so stark wie g. Man sagt

- h ist aus Groß-Oh von g
- h hat höchstens die Ordnung g

Beispiel:

Sei wieder der von oben bekannte Algorithmus gegeben:

```
Eingabe: x_1, ..., x_n ∈ ℤ, n ∈ ℕ
1 s ← 0;
2 i ← 1;
3 solange (i ⩽ n) tue
4 |   s ← s + x_i;
5 |   i ← i + 1;
Ausgabe: s
```

Mit der **Problemgröße**: n

Da die Problemgröße = n ist, folgt daraus auch: Dieser Algorithmus liegt in der Komplexitätsklasse $O(n)$.

13.3.2 Gängige Zeitkomplexitäten

Wir wollen nun im Folgenden Abschnitt einige gängige bzw. häufig vorkommende Zeitkomplexitätsklassen vorstellen. Das Ziel ist, ein gewisses Gefühl dafür zu bekommen.

Wir unterscheiden nun die Komplexitätsklassen $O(1), O(\log(n)), O(n), O(n^2), O(2^n)$.

Konstante Zeitkomplexität $O(1)$
Stellen wir uns ein Koordinatensystem vor, in dem die Funktion $g(x) = 1$ eingetragen ist (waagrechte Linie bei $y = 1$, d.h. die Funktion ist unabhängig von x). Dieser stellen wir einen Algorithmus gegenüber, der unabhängig von seiner Eingabe stets 100 ELOPs benötigt. Damit hätten wir $h(x) = 100$. Da $g(x)$ und $h(x)$ beide konstant sind, ersetzen wir sie zur Veranschaulichung durch diese konstanten Werte und setzen sie in der Definition ein:

$$\exists n_0(\exists c(\forall n > n_0(100 \leqslant c \cdot 1)))$$

Da g und h beide unabhängig von n sind, können wir dies vereinfachen zu:

$$\exists c(100 \leqslant c \cdot 1)$$

Wenn wir ein c finden, so dass diese Aussage erfüllt ist, dann liegt die Komplexität unseres Algorithmus in der Komplexitätsklasse $O(1)$. Also wählen wir $c = 100$ und erhalten $100 \leqslant 100$, was immer erfüllt ist. Der Wert von n_0 ist hierbei beliebig, also wählen wir einfach $n_0 = 1$.

Definition: 13.9 Konstante Zeitkomplexität

Ein Algorithmus A hat **konstante Zeitkomplexität**, falls $T_A^{max} \in O(1)$.
(d.h. der Zeitbedarf von A **ist unabhängig von** der Problemgröße)

Umgangssprachlich bedeutet das: Egal, wie groß die Eingabe ist (ob wir nun eine Liste von 2 oder von 1.000.000 Elementen besitzen), der Zeitaufwand für diesen Algorithmus ist **immer** konstant. Also **nicht** von der Eingabegröße abhängig.

Beispiel:

- $h \in O(1)$ für $h(n) := 1000$
  Es gilt $\forall n > 1(h(n) \leqslant 1000 \cdot 1)$ (wähle $n_0 = 1, c = 1000$)
- $h \notin O(1)$ für $h(n) := n$
  Es gilt $\forall n > n_0(h(n) \leqslant c \cdot 1) \Leftrightarrow \forall n > n_0(n \leqslant c)$ (nicht erfüllbar).
  Wenn h nicht unabhängig von n ist, ist es nicht möglich, ein c zu finden, so dass für große n $n \leqslant c \cdot 1$ erfüllt ist. Im Koordinatensystem können wir uns das so vorstellen, dass wir durch c unsere waagrechte Funktion beliebig weit nach oben schieben können. Da n jedoch linear steigt, schneidet es früher oder später diese Funktion und ist dann größer als $c \cdot 1$.

Beispiel: Algorithmen mit konstanter Zeitkomplexität

- Einfügen eines neuen Werts in ein (statisches oder dynamisches) unsortiertes Feld
- Prüfung ob eine Zeichenkette leer ist

Logarithmische Zeitkomplexität $O(\log(n))$
Nun können wir uns im Koordinatensystem eine Funktion $g(x) = \log(x)$ vorstellen, d.h. sie steigt sehr langsam. Wenn unser Algorithmus z.B. $h(x) = 200 \cdot \log(x)$ ELOPs benötigt, können wir das in unserer Definition folgendermaßen veranschaulichen:

$$\exists n_0(\exists c(\forall n > n_0(200 \cdot \log(n) \leqslant c \cdot \log(n))))$$

Ab einem n_0, das wir uns aussuchen dürfen, muss für jedes größere n der hintere Teil gelten. Wählen wir $c = 200$ und $n_0 = 1$, so erhalten wir $\forall n > 1(200 \cdot \log(n) \leqslant 200 \cdot \log(n))$, was offensichtlich erfüllt ist.

Definition: 13.12 Logarithmische Zeitkomplexität

Ein Algorithmus A hat **logarithmische Zeitkomplexität**, falls $T_A^{max} \in O(\log(n)) \setminus O(1)$.
(d.h. der Zeitbedarf von A **steigt um** 1 **bei Verdopplung der Problemgröße**, denn $\log(2 \cdot n) = \log(2) + \log(n) = 1 + \log(n)$)

Beispiel:

- $h \in O(\log(n))$ für $h(n) := 6 \cdot \log(n) + 1000$
  Für genügend große n gilt $h(n) \leqslant 7 \cdot \log(n)$.
  Damit können wir also $c = 7$ wählen.
  Die Formel muss für alle $n \geqslant n_0$ gelten, also müssen wir noch herausfinden, für welches n_0 $1000 \leqslant \log(n_0)$ gilt: $1000 \leqslant \log(n_0) \Leftrightarrow 2^{1000} \leqslant n_0$. Also wählen wir $n_0 = 2^{1000}$ und $c = 7$.

- $h \notin O(\log(n))$ für $h(n) := n$
  Es gilt: $\forall n > n_0(h(n) \leqslant c \cdot \log(n))$
  $\Leftrightarrow \forall n > n_0(\frac{n}{\log(n)} \leqslant c)$ (nicht erfüllbar: siehe Anhang).
  Es ist nicht möglich, eine Konstante c zu finden, so dass $n \leqslant c \cdot \log(n)$ für große n gilt.

Beispiel: Algorithmen mit logarithmischer Zeitkomplexität

- Binäre Suche in einer sortierten Folge

Lineare Zeitkomplexität $O(n)$
Stellen wir uns im Koordinatensystem nun eine Funktion $g(x) = x$ vor. d.h. lineare Steigung. Ein Algorithmus in dieser Komplexitätsklasse wäre z.B. $h(n) = 400 \cdot n$. In der Definition eingesetzt ergibt sich $\exists n_0(\exists c(\forall n > n_0(400 \cdot n \leqslant c \cdot n)))$, was mit $n_0 = 1$ und $c = 400$ erfüllt ist.

Definition: 13.15 Lineare Zeitkomplexität

Ein Algorithmus A hat **lineare Zeitkomplexität**, falls $T_A^{max} \in O(n) \setminus O(\log(n))$.
(d.h. der Zeitbedarf von A **wächst proportional zur Problemgröße**)

Beispiel:

- $h \in O(n)$ für $h(n) := 3 \cdot n + 100$
  Es gilt: $h(n) \leqslant 4 \cdot n$
  $\Leftrightarrow 100 \leqslant n$ $(n_0 = 100, c = 4)$
- $h \notin O(n)$ für $h(n) := n^2$
  Es gilt: $\forall n > n_0(h(n) \leqslant c \cdot n)$
  $\Leftrightarrow \forall n > n_0(n \leqslant c)$ (nicht erfüllbar).
  Da n steigt und c konstant ist, ist es nicht möglich, dass $n^2 \leqslant c \cdot n$ erfüllt wird.

Beispiel: Algorithmen mit linearer Zeitkomplexität

- Sequentielle Suche in einer sortierten Folge

- Umkehrung einer Zeichenkette

Quadratische Zeitkomplexität $O(n^2)$

Definition: 13.18 Quadratische Zeitkomplexität

Ein Algorithmus A hat **quadratische Zeitkomplexität**, falls $T_A^{max} \in O(n^2) \setminus O(n)$.
(d.h. der Zeitbedarf von A **vervierfacht sich bei Verdopplung der Problemgröße**, denn $(2 \cdot n)^2 = 4 \cdot n^2$)

Beispiel:

- $h \in O(n^2)$ für $h(n) := n^2 + 10 \cdot n - 2$
  Es gilt $h(n) \leqslant 2 \cdot n^2$
  $\Leftrightarrow 10 \cdot n - 2 \leqslant n^2$
  $\Leftarrow 10 \cdot n \leqslant n^2$
  $\Leftrightarrow 10 \leqslant n$ $(n_0 = 10, c = 2)$
- $h \notin O(n^2)$ für $h(n) := 2^n$
  Es gilt $\forall n > n_0 (h(n) \leqslant c \cdot n^2)$
  $\Leftrightarrow \forall n > n_0 (\frac{2^n}{n^2} \leqslant c)$ (nicht erfüllbar: siehe Anhang).

Beispiel: Algorithmen mit quadratischer Zeitkomplexität

- Summieren aller Einträge einer quadratischen Matrix (Problemgröße = Anzahl der Zeilen)

Es gibt natürlich auch $O(n^3)$, $O(n^4)$ usw. für entsprechend höhere Komplexitätsklassen.

Exponentielle Zeitkomplexität $O(2^n)$

Die Komplexität kan sogar noch deutlich stärker als linear, quadratisch, kubisch, ... wachsen, nämlich exponentiell. Ein Beispiel hatten wir am Anfang des Kapitels mit dem Problem des Handlungsreisenden gesehen.

Definition: 13.21 Exponentielle Zeitkomplexität

Ein Algorithmus A hat **exponentielle Zeitkomplexität**, falls $T_A^{max} \in O(2^n) \setminus \bigcup_{k \in \mathbb{N}} O(n^k)$.
(d.h. der Zeitbedarf von A **quadriert sich bei Verdopplung der Problemgröße**, denn $2^{2 \cdot n} = 2^n \cdot 2^n = (2^n)^2$)

Beispiel:

- $h \in O(2^n)$ für $h(n) := 2^n + n^2$
  Es gilt: $h(n) \leqslant 2 \cdot 2^n$

$\Leftrightarrow n^2 \leqslant 2^n$
$\Leftrightarrow 1 \leqslant \frac{2^n}{n^2}$ (erfüllt für $n \geqslant 4$, also $n_0 = 4, c = 2$))

- $h \notin O(2^n)$ für $h(n) := 2^{2 \cdot n}$
  Es gilt: $\forall n > n_0(h(n) \leqslant c \cdot 2^n)$
  $\Leftrightarrow \forall n > n_0(2^n \leqslant c)$ (nicht erfüllbar).

Beispiel: Algorithmen mit exponentieller Zeitkomplexität

- Viele Optimierungsalgorithmen (z.B. zur Lösung des Problems des Handlungsreisenden)

13.3.3 Bewertung der O-Notation

- Die O-Notation beschreibt das Wachstum von Funktionen **für große** n.
- Für **große Problemausprägungen** sind Aussagen mit dieser Notation meist relativ genau zutreffend, da konstante Summanden und Multiplikatoren in den Hintergrund treten.
- Für **kleine Problemausprägungen** kommt konstanten Multiplikatoren große Bedeutung zu. Die O-Notation ist in diesen Fällen oft sehr unpräzise.
- Algorithmen mit **quadratischer Zeitkomplexität** sind in der Praxis für große n i.d.R. gerade noch brauchbar.
- Algorithmen mit **exponentieller Zeitkomplexität** sind allenfalls für kleine n geeignet.

Die für ELOPs gewählte Abstraktionsebene hat keinen Einfluss auf die Komplexitätsklasse eines Algorithmus

- Jede ELOP eines Algorithmus in formaler programmiersprachenunabhängiger Darstellung muss auf Rechnerebene durch viele Rechenschritte (Takte) implementiert werden.
- **Aber**: Die Anzahl dieser Rechenschritte hängt nicht von der Problemgröße ab, ist also im Sinne der O-Notation **konstant**.
- **Folgerung**: Auf dem Abstraktionsniveau des Rechners gehören die Algorithmen zur selben Komplexitätsklasse wie auf der von uns betrachteten ELOP-Ebene.

13.4 Anhang: Logarithmus und Exponentialfunktion

Aus der Schule sollten folgende Rechenregeln bekannt sein:

Die Exponentialfunktion
Exponentialfunktion zur Basis 2: $g(n) := 2^n$:

- $2^n \cdot 2^m = 2^{n+m}$ (Rechenregel)
- $2^{\log(n)} = n$ (Umkehrfunktion)

Asymptotisches Verhalten für große n (Mathematischer Satz)
Für jedes $c > 0$ und $k > 0$ gibt es ein $n > 0$ mit $\frac{2^n}{n^k} > c$
(2^n wächst schneller als jedes n^k)

Der Logarithmus

Wir betrachten hier ausschließlich den Logarithmus zur Basis 2.

Logarithmusfunktion zur Basis 2: $\log(n)$:

- $\log(2^n) = n$ (Umkehrfunktion)
- $\log(n \cdot m) = \log(n) + \log(m)$ (Rechenregel)
- $\log(2) = \log(2^1) = 1$
- $\log(1) = \log(2^0) = 0$
- $\log(0.5) = \log(2^{-1}) = -1$

Asymptotisches Verhalten für große n (Mathematischer Satz)

Für jedes $c > 0$ und $k > 0$ gibt es ein $n > 0$ mit $\frac{n^k}{\log(n)} > c$
(Jedes n^k wächst schneller als $\log(n)$)