Inhaltsverzeichnis

Einleitung

Ein hochgestelltes „[?]“ weist auf fehlende Quellen hin. In der „Release Version“ ist es, wie dieser Absatz und die obenstehenden Hinweise, unsichtbar.

- Wie kann die „praktische-“ und die „theoretische Effizienz“ eines Algorithmus ermittelt werden?
- Wie verhält sich die „theoretische Effizienz“ von ausgewählten Sortieralgorithmen zu der (zu ermittelnden) „praktischen Effizienz“ jener Algorithmen?

Für konkrete Implementationen und Quellcode-Beispiele wird die Programmiersprache C++ (**ISO-C++17**, „C++17“) verwendet.

$\log x$ ist als $\log_{10} x$ bzw. als „Logarithmus zur Basis 10 von x“ zu verstehen.

Notizen Nach **mcg2012**[3] sind die hauptsächlichen Nachteile der theoretischen Analyse

- mangelnde Spezifität (ein in Pseudocode implementierter Algorithmus könnte weit von einer funktionierenden Implementation entfernt sein) und
- die oftmals stark vereinfachten „Berechnungsmodelle“.

Kapitel 1

Definition und Funktion von Algorithmen

In diesem Kapitel wird der Begriff „Algorithmus“ definiert.

1.1 Definition

Noch um 1957 war „*algorithm*“ nicht in *Webster's New World Dictionary* vertreten. Dort war nur der ältere Begriff „*algorism*“ zu finden, zu Deutsch das Rechnen mit der arabischen Zahlschrift. *Algorism* entstammt dem Namen eines persischen Autors, *Abū 'Abd Allāh Muhammad ibn Mūsā al-Khwārizmī* [1](circa 825 A. D., vgl. **taocp1**). Al-Khwārizmī schrieb das Buch *Kitab al jabr wa'l-muqābala* („[...] Rechenverfahren durch Ergänzen und Ausgleichen“), aus dessem Titel ein anderes Wort, „Algebra“, entstammt (**ger1984**). Die Wandlung des Begriffs *algorism* zu *algorithm* (und damit Algorithmus) ist in *Vollständiges mathematisches Lexicon* (**wol1747**) dokumentiert, hier wird der Begriff wie folgt definiert: „Unter dieser Benennung werden zusammen begriffen die 4 Rechnungs-Arten in der Rechen-Kunst nemlich addiren, multipliciren, subtrahiren und dividiren. [...]“ (alte Rechtschreibung übernommen).

Vor 1950 wurde der Begriff Algorithmus am häufigsten mit dem euklidischen Algorithmus assoziiert, einem Verfahren zur Ermittlung des größten gemeinsamen Teilers zweier Zahlen (vgl. **taocp1**).

Die moderne Bedeutung des Wortes *Algorithmus* ist nicht unähnlich zu den Bedeutungen der Wörter *Rezept*, *Vorgang*, *Verfahren*, et cetera. Dennoch konnotiert das Wort etwas anderes – es ist nicht nur eine endliche Menge von Regeln, die eine Folge von Operationen für die Lösung einer bestimmten Aufgabe darstellen, ein Algorithmus hat nach **hsr1997** zwingend folgende fünf Merkmale:

1. *Eingabe*. Ein Algorithmus hat keine, eine oder mehrere Eingangsmengen, welche entweder zu Beginn oder während der Laufzeit gegeben werden.

2. *Ausgabe*. Ein Algorithmus hat eine oder mehrere Ausgabemengen, welche eine wohldefinierte Beziehung zu den Eingangsmengen haben.

[1] Al-Khwārizmīs Name stellt in der Literatur eine Quelle der Verwirrung dar: **hsr1997**, **taocp1**, **pic2009** und **ger1984** verwenden bei ihren Erwähnungen jeweils unterschiedliche Namen, sprechen jedoch von derselben Person.

3. *Eindeutigkeit*. Jeder Arbeitsschritt eines Algorithmus ist eindeutig definiert.

4. *Endlichkeit*. Ein Algorithmus terminiert[2] nach einer endlichen Anzahl von Arbeitsschritten.

5. *Effektivität*. Die Arbeitsschritte müssen einfach genug sein um in endlicher Zeit von einer Person mit Stift und Papier ausgeführt werden zu können. *Mit Beispielen besser zu erklären.*

Etwas informeller kann ein Algorithmus als ein *wohldefiniertes* (vgl. **alu2009**) Berechnungsverfahren (engl. *computational procedure*) beschrieben werden, das einen Wert, oder eine Menge von Werten, als Eingabe erhält und einen Wert, oder eine Menge von Werten, als Ausgabe produziert (vgl. **clrs2001**). Ein Algorithmus ist also eine Folge von Berechnungsschritten, welche Eingabe zu Ausgabe überführen.

1.2 Methoden zur Algorithmusbeschreibung

Algorithmen können auf vielfältige Weise beschrieben und dargestellt werden.

taocp1 verwendet etwa in seinem Standardwerk „**taocp1**“ (**taocp1**) eine Mischung aus natürlicher Sprache und mathematischen Ausdrücken.

Algorithmus E (*Euklids Algorithmus*). Gegeben seien zwei positive ganze Zahlen m und n. Der *größte gemeinsame Teiler*, also die größte positive ganze Zahl welche sowohl m als auch n gerade teilt, ist zu ermitteln.

E1 [Ermittle den Rest.] Dividiere m durch n, r ist der Rest. (Es gilt nun also $0 \leq r < n$.)
E2 [Ist er Null?] Gilt $r = 0$ so terminiert der Algorithmus, n ist die Lösung.
E3 [Verringere.] Setze $m \leftarrow n$, $n \leftarrow r$ und gehe zurück zu Schritt **??**.

Abbildung 1.1: Algorithmusbeschreibung nach **taocp1**. Unter Anderem verwendet in **taocp1** und **taocp3**. Beispiel entnommen aus **taocp1**, Algorithm E (übersetzt aus dem Englischen).

Diese Art der Algorithmusbeschreibung, wie sie beispielhaft in Abbildung **??** anhand von Euklids Algorithmus[3]dargestellt wird, gliedert sich gut in den Lesefluss ein, und ist am besten geeignet um die Funktionsweise eines Algorithmus verständlich darzustellen (**zob2015**).

Eine andere Methode ist die Darstellung eines Algorithmus mithilfe von *Flowcharts*, wie sie in Abbildung **??** wieder am Beispiel von Euklids Algorithmus dargestellt wird.

Diese Art der Beschreibung eignet sich am besten für kleine bzw. kurze Algorithmen (**hsr1997**). Sie wird (wie etwa auch von **taocp1**) meistens in Kombination mit einer anderen Algorithmusbeschreibung verwendet um diese zu komplementieren.

Die wohl am weitreichendsten etablierte Art der Beschreibung von Algorithmen ist jedoch *Pseudocode* (vgl. **zob2015**), wie er etwa in Abbildung **??** zu sehen ist.

Pseudocode ist näher zu tatsächlichen Programmiersprachen in denen ein Algorithmus implementiert werden könnte als die bisher erwähnten Alternativen (vgl. **ofn2015**). Er bleibt allerdings abstrakt genug um Algorithmen (im Gegensatz zu vielen Programmiersprachen) unabhängig von Hardware und im Kontext unwichtigen Formalismen diverser Programmiersprachen darstellen zu können (vgl. **bem1958**). Die Darstellung

[2] Abbruch mit Erfolg oder Misserfolg
[3] Siehe Abschnitt **??**.

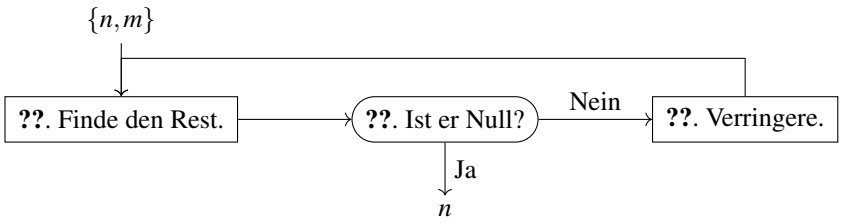


Abbildung 1.2: Algorithmusbeschreibung in Form eines Flowcharts. Beispiel in abgeänderter Form aus **taocp1**, Fig. 1 entnommen.

```
EUKLID(n,m)
1  while (r = n  mod m) ≠ 0
2      m = n
3      n = r
4  return n
```

Abbildung 1.3: Beschreibung durch *Pseudocode* wie er auch in der folgenden Arbeit verwendet wird.

durch Pseudocode macht allerdings dennoch ein erstes Verständnis des beschriebenen Algorithmus üblicherweise schwerer als andere Methoden, bleibt allerdings durch die Nähe zu möglichen konkreten Implementationen unmissverständlicher (vgl. **zob2015**). Aus letzterem Grund wird in der folgenden Arbeit Pseudocode zur Algorithmusbeschreibung verwendet.

1.3 Pseudocode

Anzumerken ist, dass es nicht *eine bestimmte* Pseudocode-Notation gibt. Programmiersprachen wie beispielsweise C++17 sind versioniert und standardisiert (vgl. **ISO-C++17**), für Pseudocode gibt es keinen derart weit verbreiteten (oder wohldefinierten) Standard (vgl. **kni1996**). Eine derartige Standardisierung wäre allerdings auch entgegen der Zielsetzung: Die Freiheit, Pseudocode an den gegebenen Kontext anzupassen ist eine seiner großen Stärken. So wird zwar in einem Gutteil der in dieser Arbeit zitierten Werke Pseudocode verwendet (sofern im Kontext erfordert), konkrete Ausprägungen variieren jedoch in Syntax und Semantik[4].

Wie schon in Abschnitt **??** erwähnt wird für Beschreibungen von Algorithmen in dieser Arbeit überwiegend Pseudocode zum Einsatz kommen. Genauer werden die *„pseudocode conventions“* aus **clrs2001** verwendet, folgend kurz zusammengefasst.

- Namen von Algorithmen sind in KAPITÄLCHEN gesetzt. Namen von Variablen sind *kursiv* gesetzt und üblicherweise einzelne Buchstaben (*i*, *n*, ...).
- Die „Blockstruktur“ ist durch Einrückung vorgegeben. Beispielsweise beginnt der Körper der **while** Schleife aus Abbildung **??** auf Zeile **??** und endet auf Zeile **??**.

[4] Vgl. beispielsweise die diversen Definitionen der verwendeten Pseudocode-Varianten in **hsr1997**, **clrs2001** und **ahu1974**, die jeweils an den gegebenen .

- Die Konstrukte **while** und **for** sind ähnlich interpretierbar wie jene in Sprachen wie C oder Java, wobei eine Vereinfachung der in dieser Sprachgruppe üblichen Syntax der **for** Schleife vorgenommen wurde.

```
1  i = 2
2  while i < 100
3      // Körper der Schleife
4      i = i^2
5
```

Während i kleiner als 100 ist, wird der Körper der Schleife ausgeführt. Nach Zeile **??** gilt $i = 256$

```
1  for j = 0 to 10
2      // Körper der Schleife
```

Hier ist j der „Schleifenzähler“ und 10 der Endwert. Der Zähler wird um 1 erhöht, bis er *gleich* dem Endwert ist.

- Wie aus den vorhergehenden Beispielen hervorgeht deutet das Symbol „// “ darauf hin, dass der Rest der Zeile als Kommentar zu verstehen ist.

- Elemente eines *Array* (Datenfeld) A werden durch $A[i]$ abgerufen, wobei i der Index des Elements ist. Arrays werden im Gegensatz zu vielen üblichen Programmiersprachen beginnend mit 1 indiziert und befüllt – $A[1]$ ist das erste Element in einem Array A.

  Die Schreibweise $A[i\,..\,j]$ wird verwendet um einen „Ausschnitt“ bzw. ein „*Subarray*“ eines Arrays darzustellen. So steht $A[1\,..\,j]$ für das Subarray von A welches (für $j > 2$) die j Elemente $A[1], A[2], \ldots A[j]$ beinhaltet.

  Die Länge eines Arrays kann durch $A.length$ abgerufen werden. (In C müsste hierfür eine separate Variable geführt werden.)

- Manchmal werden Schritte auch nur in natürlicher Sprache beschrieben, wenn dadurch die Verständlichkeit gefördert wird, und dennoch die Eindeutigkeit der Schritte gegeben ist.

  So wird beispielsweise wenn angebracht „tausche $A[i]$ und $A[j]$ aus“ anstelle von

```
1  t = A[i]
2  A[i] = A[j]
3  A[j] = t
```

  geschrieben.

- Eine **return** Anweisung überträgt die Kontrolle und einen optionalen, auf die Anweisung folgenden, Wert zurück zur Aufrufstelle.

  Hier wäre ein Beispiel angebracht, vielleicht mit Rekursion?

1.4 Sortieralgorithmen

Eine Darstellung der Wichtigkeit des Problems – sowohl konkret als wichtiger Bestandteil von Software, und abstrakt als Katalyst neuer Erkenntnisse in der theoretischen Informarik – wäre hier angebracht.

Ein Sortieralgorithmus ist ein Algorithmus der eine Eingabemenge A in eine sortierte Permutation dieser Menge überführt, und als Ausgabemenge zurückgibt. Genauer muss

bei einer Eingabemenge mit den n Elementen $a_1, a_2, \ldots, a_n$ gelten, dass für alle Elemente $a'_1, a'_2, \ldots, a'_n$ der Ausgabemenge die Ordnungsrelation $a'_1 \leq a'_2 \leq \cdots \leq a'_n$ gilt (vgl. **taocp3**).

Ein ikonischer Sortieralgorithmus ist der „insertion sort“ (vgl. **taocp3**). Jedes Element wird einzeln betrachtet und in eine (wachsende) sortierte „Teilliste“ *eingefügt* – daher der Name *insertion* sort. Dieser Algorithmus wird in Abschnitt **??** (siehe insbesondere Abbildung **??**) genauer behandelt, für ein grundlegendes Verständnis der Arbeitsweise von Algorithmen wird es jedoch hilfreich sein ihn auch jetzt schon etwas genauer zu betrachten.

INSERTION-SORT(A)

```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1..j-1].
4      i = j-1
5      while i > 0 and A[i] > key
6          A[i+1] = A[i]
7          i = i-1
8      A[i+1] = key
```

Bevor ein Element $A[j]$ betrachtet wird besteht die Annahme, dass die Elemente $A[1], \ldots, A[j-1]$ sortiert sind. (Aus diesem Grund beginnt die **for** Schleife in Zeile **??** mit $j = 2$ und nicht etwa mit $j = 1$; das Subarray $A[1..j-1]$ hat für $j = 2$ nur ein Element und ist somit immer „sortiert“.)

Jedes Element $A[2..A.length]$ wird nun einzeln betrachtet und in $A[1..j-1]$ „einsortiert“. Um ein Element derart in die Teilliste einzufügen zu können muss

(a) der neue Index des einzufügenden Elements gefunden werden.
(b) Platz für das einzufügende Element geschaffen werden.

Dies passiert in der **while** Schleife (beginnend mit Zeile **??**). Sie iteriert vorerst über alle möglichen Indexe i, absteigend und beginnend mit dem höchstmöglichen ($i = j-1$, Zeile **??**) bis zum kleinstmöglichen $i = 1$. Für jedes i wird das Element $A[i]$ „nach oben“, zu $A[i+1]$ verschoben (Zeile **??**) – somit wird Platz für das einzufügende Element geschaffen.

Im Zuge dieser Verschiebung wird anfangs das Element $A[j]$ überschrieben, nachdem bei der ersten Iteration gilt, dass $j = i+1$. Aus diesem Grund wird der Wert von $A[j]$ zu Beginn der **for** -Schleife in der Variable *key* zwischengespeichert (Zeile **??**).

Gilt nach einer Verschiebung $A[i] \leq key$ (bzw. gilt also $A[i] > key$, Zeile **??**, *nicht*), so wurde der neue Index des einzufügenden Elements gefunden: Das Element ist nach $A[i]$, also am Index $i+1$ einzufügen (Zeile **??**).

Kapitel 2

Asymptotische Analyse als Effizienzangabe

Üblicherweise ist die Effizienz eines Algorithmus proportional zur Größe der jeweiligen Eingangsmenge, deshalb wird die Effizienz im Folgenden (und im Allgemeinen auch in der Literatur[1]) als Funktion T in Abhängigkeit der „Eingabegröße“ n dargestellt, also als $T(n)$.

Die Werte von $T(n)$, also die theoretische Effizienz eines Algorithmus auf einer Eingabemenge der Größe n, sind im Kontext der theoretischen Analyse die Anzahl der „einfachen Operationen“ oder „Schritten“ welche für diese Eingabemenge ausgeführt werden müssen (vgl. **clrs2001** und **hsr1997**). „Einfache Operationen“ sind hier jene Operationen deren benötigte Zeit unabhängig von den Operanden ist, beispielsweise arithmetische Grundrechenarten und Vergleiche (vgl. **sha2011**). Durch diese Einschränkung kann die Anzahl der einfachen Operationen als stellvertretend zur tatsächlichen, sonst experimentell zu ermittelnden Laufzeit gesehen werden (vgl. **sha2011**).

2.1 Günstigster, ungünstigster und durchschnittlicher Fall

Es wird zwischen der theoretischen Effizienz im „günstigsten“, „ungünstigsten“, und „durchschnittlichen Fall“ unterschieden, alle beziehen sich auf die Beschaffenheit der Eingangsmenge (vgl. **hsr1997**).

Zur Veranschaulichung dieser Unterscheidung ist ein einfacher Suchalgorithmus, SEQUENTIAL-SEARCH (vgl. **taocp3**), zu betrachten der eine Liste nach einem Element durchsucht und terminiert nachdem er es gefunden hat.

```
SEQUENTIAL-SEARCH(A, value)
1  for i = 1 to A.length
2      if A[i] == value
3          return i
4  return 0
```

[1] Alle Werke die in dieser Arbeit zitiert werden und sich mit der Effizienz beziehungsweise Komplexität von Algorithmen beschäftigen stellen diese in Abhängigkeit von einer Eingabegröße (auch: Problemgröße) dar.

Der günstigste Fall für einen solchen Algorithmus tritt auf wenn die Eingangsmenge eine Liste ist, in der das gesuchte Element an der ersten Position steht. In diesem Fall wird nur ein Vergleich ausgeführt, die Laufzeit ist kurz. Steht das gesuchte Element jedoch an der letzten Position so werden $A.length$ beziehungsweise n Vergleiche ausgeführt, eine solche Menge führt zum ungünstigste Fall. Unter der Annahme, dass die Elemente von A gleichmäßig verteilt sind, führt der Algorithmus durchschnittlich $n/2$ Vergleiche aus, dies ist der durchschnittliche Fall (vgl. **sha2011**).

2.2 Asymptotische Analyse

Zur einleitenden Frage, wie die asymptotische Analyse zu beschreiben sei, schreibt **bru1958** in **bru1958**:

> It often happens that we want to evaluate a certain number, defined in a certain way, and that the evaluation involves a very large number of operations so that the direct method is almost prohibitive. In such cases we should be very happy to have an entirely different method for finding information about the number, giving at least some useful approximation to it. [...] A situation like this is considered to belong to asymptotics. (**bru1958**)

Die Ermittlung der exakten Anzahl an einfachen Operationen die ein Algorithmus mit einer gewissen Eingabemenge benötigt ist üblichweise nicht lohnenswert (vgl. **hsr1997**) oder sogar unmöglich (vgl. **meh1984**) – gemäß **bru1958** ist in diesem Fall eine Ermittlung mithilfe asymptotischer Methoden ratsam.. Diese „certain number“ die im obigen Zitat erwähnt wird ist im gegebenen Kontext demzufolge $T(n)$, die Information die es zu finden gilt ist die *Größenordnung des Wachstums* (auch *Wachstumsrate*) der Funktion T (vgl. **sha2011**).

Diese Größenordnung des Wachstums bei steigendem bzw. großen Werten von n wird in der einschlägigen Literatur oft in der O-Notation angeschrieben (vgl. **ahu1974**, **taocp1**, **hsr1997**, ...).

Kann die Effizienz eines Algorithmus als

$$T(n) = an^2 + bn + c \tag{2.1}$$

ausgedrückt werden (wobei a, b und c beliebige von n unabhängige Konstanten sind), so kann (**??**) mithilfe dieser O-Notation zu

$$T(n) = O(n^2) \tag{2.2}$$

vereinfacht werden.

Diese Vereinfachung verleiht dem Umstand Ausdruck, dass hier nur der Term n^2 von Interesse ist. Terme niederer Ordnung (bn und c) und konstante Faktoren (a) werden bei großem bzw. wachsendem n relativ bedeutungslos (vgl. **clrs2001**), und können so „weggelassen werden“. Die O-Notation ersetzt also die Kenntnis einer Zahl durch das Wissen, dass eine solche Zahl existiert (vgl. **bru1958**).

Das ist der Kern der asymptotischen Analyse in diesem Kontext: Die Untersuchung eines Algorithmus für große n und die damit einhergehende Möglichkeit der Vereinfachung von Algorithmen (vgl. **sha2011**).

Tabelle **??** veranschaulicht diese Vernachlässigbarkeit von Termen niederer Ordnung und von konstante Faktoren bei großem n.

n	$f(n)$	$g(n)$	$\frac{f(n)}{g(n)}$
10^0	1	18	0.05555555
10^1	100	135	0.74074074
10^2	10000	10215	0.97895252
10^3	1000000	1002015	0.99798905
10^4	100000000	100020015	0.99979989
10^5	10000000000	10000200015	0.99997999
10^6	1000000000000	1000002000015	0.99999799
10^7	100000000000000	100000020000015	0.99999979
10^8	10000000000000000	10000000200000016	0.99999997
10^9	1000000000000000000	1000000002000000000	0.99999999

Tabelle 2.1: Tabelle der Funktionswerte zweier Funktionen $f(n) = n^2$ und $g(n) = n^2 + 2n + 15$. Bei steigendem n nähern sich die Funktionswerte von f und g einander an, der Quotient $\frac{f(n)}{g(n)}$ konvergiert gegen 1.

Komplexität Die Funktion $T(n)$, also die Laufzeit eines Algorithmus in Abhängigkeit der Eingabegröße n wird auch *Zeitkomplexität* des Algorithmus genannt. Das asymptotische Verhalten von $T(n)$ für große Werte von n wird *asymptotische Zeitkomplexität* genannt (vgl. **ahu1974**). Im folgenden ist mit *Komplexität* eines Algorithmus die asymptotische Zeitkomplexität des Algorithmus gemeint.

Ein Algorithmus in $O(n^2)$ hätte demzufolge eine „Komplexität von n^2“ – für große n kann seine Laufzeit durch die Funktion $f(n) = n^2$ beschrieben werden.

Definition der O-Notation Genauer beschreibt $O(g(n))$ für eine gegebene Funktion $g(n)$ (im Fall von (**??**) also $g(x) = n^2$) die Menge von Funktionen

$$O(g(n)) = \left\{ f(n) \ : \begin{array}{l} \text{es existieren positive Konstanten } c \text{ und } n_0 \text{ derart,} \\ \text{dass } 0 \leq f(n) \leq c \cdot g(b) \text{ für alle } n \geq n_0 \end{array} \right\}$$

(vgl. **meh1984**). Die O-Notation definiert eine „asymptotische obere Schranke“ (vgl. **sha2011**): Der Wert der Funktion $f(n)$ ist kleiner oder gleich dem Wert der Funktion $c \cdot g(n)$ für alle $n \geq n_0$.

2.3 Praxisnahe Komplexitäten

Einige der in der in diesem Kontext am häufigsten vorkommenden Komplexitäten sind n^2, $\log n$ und $n \log n$[2](vgl. **hsr1997**). (Anders gesagt sind also viele Sortieralgorithmen in $O(n^2)$, $O(\log n)$ oder $O(n \log n)$.) Tabelle **??** veranschaulicht diese Komplexitäten anhand von Funktionswerten für diverse Werte von n.

Der in Tabelle **??** ebenfalls dargestellte *Quotient* der jeweiligen Funktionswerte und n zeigt das Verhältnis eines Algorithmus dessen Laufzeit linear wächst, zu einem Algorithmus dessen Laufzeit durch eine der dargestellten Komplexitäten zu beschreiben ist. So ist ein Algorithmus in $O(\log n)$ für $n = 256$ fast 50-fach schneller als ein Algorithmus in $O(n)$.

[2] Alle in dieser Arbeit behandelten Sortieralgorithmen haben eine dieser Komplexitäten.

Anzumerken ist jedoch, dass die *O*-Notation die *asymptotische* Wachstumsrate darstellt (bzw. die Komplexität eines Algorithmus als asymptotisches Verhalten zu sehen ist). So kann etwa ein Algorithmus in $O(n^2)$ für kleine n schneller als ein Algorithmus in $O(n)$ sein.

In diesem Kontext anmerken, dass O-Notation die asymptotische Wachstumsrate *darstellt – $O(n^2)$ kann für kleine n schneller als $O(n)$ sein.*

n	$\frac{n}{n}$	n^2	$\frac{n}{n^2}$	$\log n$	$\frac{n}{\log n}$	$n \log n$	$\frac{n}{n \log n}$
1	*1*	1	*1*	0		0	
4	*1*	16	*0.25*	1.38629436	*2.88539008*	5.54517744	*0.72134752*
16	*1*	256	*0.0625*	2.77258872	*5.77078016*	44.3614195	*0.36067376*
64	*1*	4096	*0.015625*	4.15888308	*15.3887471*	266.168517	*0.24044917*
256	*1*	65536	*0.00390625*	5.54517744	*46.1662413*	1419.56542	*0.18033688*
1024	*1*	1048576	*0.00097656*	6.93147180	*147.731972*	7097.82712	*0.14426950*
4096	*1*	16777216	*0.00024414*	8.31776616	*492.439907*	34069.5702	*0.12022458*
16384	*1*	268435456	*0.00006103*	9.70406052	*1688.36539*	158991.327	*0.10304964*
65536	*1*	4294967296	*0.00001525*	11.0903548	*5909.27888*	726817.498	*0.09016844*

Tabelle 2.2: Werte der Funktionen $f(n) = n$, $f(n) = n^2$, $f(n) = \log n$ und $f(n) = n \log n$ mit nebengestelltem, *kursiv gesetztem*, Quotient aus dem jeweiligem Funktionswert und n. Leerstehende Felder repräsentieren ein undefiniertes Ergebnis. Es gilt $n = 2^0, 2^2, 2^4, \ldots, 2^16$.

2.4 Ermittlung

Einfaches Beispiel, z. B. angelehnt an **sha2011** (**sha2011**). **ahu1974** (**ahu1974**) ist erwähnenswert aber insgesamt zu komplex.

Kapitel 3

Laufzeitermittlung als Effizienzangabe

In diesem Kapitel wird eine Methode zur Ermittlung der Laufzeit von Algorithmen dargestellt.

Es gilt die Laufzeit eines Algorithmus zu ermitteln und diese annähernd als Funktion $T(n)$ (wobei $T(n)$ die Zeit und n die Eingabegröße ist) darzustellen. Das Ziel deckt sich also mit jenem der asymptotschen Analyse aus Kapitel **??**. Genauer gilt es, ebenfalls wie in der asymptotischen Analyse, eine solche Funktion für diverse Arten von Eingabemengen zu ermitteln (**mcg2012**).

Nach **mcg2012** kann „the experimental process“ im Kontext der empirischen Algorithmusanalyse im Wesentlichen grob in folgende vier Schritte aufgegliedert werden.

1. Formuliere eine Frage.
2. Stelle eine Testumgebung bereit.
3. Gestalte ein Experiment welches die Frage aus Punkt **??** anspricht.
4. Führe das Testprogramm aus und sammle die Daten.

Eine grobe Formulierung der Frage ergibt sich schon aus der vorhergehenden Einleitung dieses Kapitels:

„Wie viel Zeit benötigt ein Sortieralgorithmus um Eingabemengen verschiedener Art und Größe in eine sortierte Ausgabemenge zu überführen?“

Um diese Frage umsetzbar zu machen, gilt es nunmehr nur noch die verschiedenen Arten und Größen der Eingabemengen zu definieren, dies geschieht in Abschnitt **??**.

Die Testumgebung aus Punkt **??** wird in Abschnitt **??** näher definiert.

Hier sind **joh2002** *(***joh2002***) und* **mcg2012** *(***mcg2012***) wichtig.* **sha2011** *kann als Argumentation für das ausgeprägte Verwenden der* Standard Library *ausgelegt werden nachdem hier potentielle Vorurteile bzw. Ungleichheiten bei der Programmierung de facto wegfallen.*

Der Werdegang der Empirie in den Computerwissenschaften ist nicht unspannend: Zu Zeiten von **joh2002** *war die empirische Analyse als Feld offenbar noch bei weitem nicht so weit fortgeschritten und verbreitet wie im Erscheinungsjahr von* **mcg2012**. *Beide beinhalten weitgehend äquivalente Kernaussagen, aber erstere Publikation ist noch bei weitem unausgereifter als letztere.*

3.1 Eingabemengen

Referenzen anderer Analysen mit gleichen oder überlappenden Eingabearten sind beizufügen.

Für die Ermittlung der praktischen Effizienz werden

1. sortierte
2. invertierte
3. zufällig geordnete

Eingabemengen verwendet.

Siehe Listing **??** für eine beispielhafte Implementation von Generatoren der obigen Eingabemengearten.

Eine ausführlichere Auswahl an Eingangsmengen welche an einen sie bearbeitenden Algorithmen angepasst ist — wie sie für eine eingehende Analyse eines einzelnen Algorithmus angebracht wäre — würde das allgemeiner gesetzte Ziel dieser Arbeit verfehlen (vgl. **mcg2012**).

3.2 Testumgebung

Ein Algorithmus wird ausgeführt, die Zeit unmittelbar vor (*start*) und nach (*end*) der Ausführung wird gemessen. Die Laufzeit des Algorithmus ist nun gleich $end - start$.

Eine konkrete Implementation einer Klasse zur Messung der Laufzeit eines Algorithmus ist in Listing **??** gegeben. *Referenzen zu empirischen Analysen die einen ähnlichen Mechanismus zur Laufzeitermittlung verwenden (kann nicht so schwer sein, was sollen sie sonst verwenden) sind beizufügen.*

```
1  class experiment {
2    std :: function<void()> algorithm;
3
4  public:
5    using time_t = double;
6
7     explicit experiment( std :: function<void()> algorithm )
8       : algorithm( std :: move(algorithm)) { };
9
10   auto run() const {
11     const auto start = std :: chrono :: steady_clock :: now();
12
13     algorithm();
14
15     const auto end = std :: chrono :: steady_clock :: now();
16
17     return std :: chrono :: duration<time_t, std :: micro>{ end - start };
18   }
19 };
```

Listing 3.1: Implementation einer Klasse zur Ermittlung der Laufzeit eines Algorithmus.

Die Laufzeit eines einfachen Algorithmus kann mit ihrer Hilfe durch

```
void a1() { ... }
const auto time = experiment(a1).run();
```

ermittelt werden, wobei die Zeitspanne in Mikrosekunden ($1\mu s = 10^{-6}s$) angegeben ist.

Algorithmen mit Eingabewerten Der Konstruktor in Listing **??**, Zeile **??** erwartet als einzige Eingabe eine Variable des Typs std :: function<**void**()>, also eine Funktion ohne Eingabe- und Rückgabewerte. Dies ist der Fall um größtmögliche Flexibilität in der Verwendung der Testumgebung zu gewährleisten.

Die in dieser Arbeit behandelten Algorithmen haben alle einen oder mehrere Eingabewerte, sie erfüllen also nicht die Form wie sie vom Konstruktor erwartet wird. Um ein Experiment mit einer Funktion eines anderen Typs als **void**() zu initialisieren wird ein „*wrapper*“ verwendet:

```
void func( size_t s) { ... }
const auto wrapper = std::bind(func, 1337);
const auto time = experiment(wrapper).run();
```

Im obenstehenden Beispielcode wird die Funktion std :: bind der Standardbibliothek verwendet um eine „umwickelnde Funktion“ *wrapper* (des Typs **void**()) zu erstellen. Ein Aufruf von *wrapper* führt zum Aufruf der Funktion *func* mit dem Eingabeparameter 1337 (wie durch die an std :: bind übergebenen Parametern festgelegt wurde).

3.3 Funktionsermittlung

Im gegebenen Kontext ist es das Ziel der „Funktionsermittlung“ eine Funktion in Abhängigkeit der Eingabegröße aufzustellen bzw. zu approximieren, welche die Laufzeit darstellt (vgl. **mcg2012** — *Modeling*).

Werte dieser Funktion der praktischen Effizienz können mithile der Funktion benchmark::run in Listing **??** ermittelt werden.

```
1  namespace benchmark {
2    using timings_t = std::map<size_t, experiment::time_t>;
3
4    template<class A, class S>
5    timings_t run(A algorithm, S set, int total_chunks) {
6      timings_t timings;
7
8      const size_t set_size = set.size();
9
10     std::fesetround(FE_TONEAREST);
11
12     const size_t chunk_size = set_size > total_chunks ?
13       std::nearbyint(set_size / total_chunks) : 1;
14
15     for (size_t i = chunk_size; i <= set_size; i += chunk_size) {
16       auto subset = S(set.begin(), set.begin() + i);
17       const auto time = experiment(std::bind(algorithm,
18         subset.begin(), subset.end(), std::less<>())
19       ).run();
20
21       timings.emplace(i, time.count());
22     }
```

```
23
24        return timings;
25      }
26
27      ...
28  }
```

Listing 3.2: Implementation einer Funktion zur Ermittlung der praktischen Effizienz eines Algorotihmus mit einer bestimmenten Eingabemenge.

Der Prägnanz halber wird es hilfreich sein einige der im vorhergehenden Quellcode vorkommende Variablennamen Pseudonyme zu geben. Ebenfalls werden dem Verständnis dienende „gedachte Variablen“ benannt. So gilt folgend

$$\begin{aligned} t &= \text{total\_chunks}\,, \text{Anzahl der Untermengen} \\ s &= \text{set\_size}\,, \text{Größe der Eingabemenge} \\ c &= \text{chunk\_size}, \text{„Schrittgröße“} \\ t &= \text{Anzahl der Untermengen} \\ U_i &= i\text{-te Untermenge} \\ n_i &= \text{Größe von } U_i \end{aligned}$$

Die Funktion benchmark::run nimmt als Eingabeparameter

nosep, label=()einen Algorithmus. eine Menge. die Anzahl der Untermengen.

Um nicht
Es gilt

$$n_i = c \cdot i \quad \text{für alle } 0 < i < \frac{c}{n} - 1 \in \mathbb{N}$$

für

$$c = \begin{cases} 1 & \text{für } s \leq t \\ \frac{s}{t} & \text{für } s > t \end{cases}$$

Der Parameter total_chunks (folgend t) definiert die Anzahl der Untermengen. In Kombination mit der Größe der Eingabemenge set_size (folgend s) kann damit eine konstante Schrittgröße chunk_size (folgend c) ermittelt werden.

Nach Ausführung der Funktion benchmark::run werden (sequentiell) t *überlappende Untermengen* $U_1, U_2, \ldots, U_t$ der Eingabemenge erstellt – für jedes $0 \leq i \leq t$ gibt es eine Menge U_i mit Größe n_i. Die Größe dieser Mengen steigt mit i, es gilt $n_i = i \cdot c$

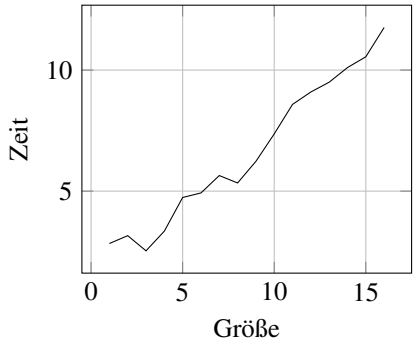
($s_1 - s_2$ beziehungsweise $s_n - s_{n+1}$ für $0 \leq n \leq 9$) ermittelt werden. Die Eingabemenge wird in total_chunks *überlappende* Untermengen der Größe

Die Funktion liefert Wertepaare als Inhalt eines *containers* vom Typ std :: map< size_t, experiment :: time_t>. Dieser assoziative container enthält eine geordnete Menge von Schlüssel-Wert-Paaren wobei die Schlüssel die Größe der jeweils betrachteten Untermenge, und die Werte die benötigten Zeiten darstellen. Die Werte können nun wie in Abbildung **??** ausgegeben, oder wie in Abbildung **??** als Graph dargestellt werden.

Abbildungsverzeichnis

Größe	Zeit
1	2.836
2	3.16
3	2.535
4	3.346
5	4.737
6	4.925
7	5.643
9	5.337
⋮	⋮
⋮	⋮
9	6.23
10	7.363
11	8.583
12	9.094
13	9.499
14	10.103
15	10.548
16	11.757

(a) *Quick sort* auf sehr kleinen, sortierten Eingabemengen; links ist die Größe der Eingabemenge, rechts die Zeit in Mikrosekunden.



(b) Graph der Daten in **??**.

Abbildung 3.1: Demonstration des Ausgabeformats aus Listing **??** mit daraus generiertem Graphen.

Tabellenverzeichnis

Listings