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/IEC 14882, 2017, „C++17“) verwendet.

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

Notizen Nach McGeoch, 2012[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. Knuth, 1997, S. 1–2). Al-Khwārizmī schrieb das Buch *Kitab al jabr wa'l-muqābala* („[...] Rechenverfahren durch Ergänzen und Ausgleichen“), aus dessem Titel ein anderes Wort, „Algebra“, entstammt (Gericke, 1984, S. 197–199). Die Wandlung des Begriffs *algorism* zu *algorithm* (und damit Algorithmus) ist in *Vollständiges mathematisches Lexicon* (Wolff, 1747, S. 38) 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. Knuth, 1997, S. 2).

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 Horowitz u. a., 1997, S. 1 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: Horowitz u. a., 1997, S. 1, Knuth, 1997, S. 1, Pickover, 2009, S. 84 und Gericke, 1984, S. 197–199 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. Aluffi, 2009, S. 16) 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. Cormen u. a., 2001, S. 5). 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.

Knuth verwendet etwa in seinem Standardwerk „*The Art of Computer Programming, Volume 1: Fundamental Algorithms*“ (Knuth, 1997) 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 E1.

Abbildung 1.1: Algorithmusbeschreibung nach Knuth. Unter Anderem verwendet in Knuth, 1997 und Knuth, 1998. Beispiel entnommen aus Knuth, 1997, S. 2, Algorithm E (übersetzt aus dem Englischen).

Diese Art der Algorithmusbeschreibung, wie sie beispielhaft in Abbildung 1.1 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 (Zobel, 2015, S. 147).

Eine andere Methode ist die Darstellung eines Algorithmus mithilfe von *Flowcharts*, wie sie in Abbildung 1.2 wieder am Beispiel von Euklids Algorithmus dargestellt wird.

Diese Art der Beschreibung eignet sich am besten für kleine bzw. kurze Algorithmen (Horowitz u. a., 1997, S. 5). Sie wird (wie etwa auch von Knuth) 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. Zobel, 2015, S. 147), wie er etwa in Abbildung 1.3 zu sehen ist.

[2] Abbruch mit Erfolg oder Misserfolg
[3] Siehe Abschnitt 1.1.

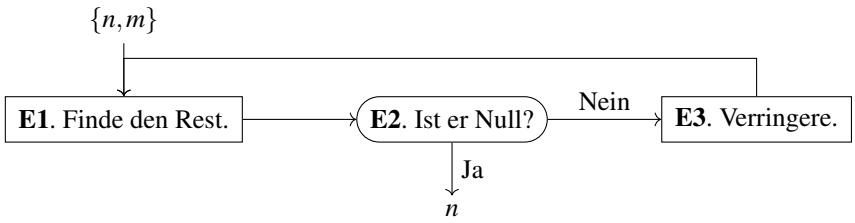


Abbildung 1.2: Algorithmusbeschreibung in Form eines Flowcharts. Beispiel in abgeänderter Form aus Knuth, 1997, S. 3, Fig. 1 entnommen.

EUKLID(n,m)
1 **while** $(r = n \mod m) \neq 0$
2 $\quad m = n$
3 $\quad n = r$
4 **return** n

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

Pseudocode ist näher zu tatsächlichen Programmiersprachen in denen ein Algorithmus implementiert werden könnte als die bisher erwähnten Alternativen (vgl. Oda u. a., 2015, S. 1). 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. Bemer, 1958, S. 1). Die Darstellung 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. Zobel, 2015, S. 147). 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/IEC 14882, 2017), für Pseudocode gibt es keinen derart weit verbreiteten (oder wohldefinierten) Standard (vgl. Knill, 1996). 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 1.2 erwähnt wird für Beschreibungen von Algorithmen in dieser Arbeit überwiegend Pseudocode zum Einsatz kommen. Genauer werden die *„pseudocode conventions“* aus Cormen u. a., 2001, 20ff verwendet, folgend kurz

[4] Vgl. beispielsweise die diversen Definitionen der verwendeten Pseudocode-Varianten in Horowitz u. a., 1997, S. 5, Cormen u. a., 2001, 20ff und Aho u. a., 1974, 33ff, die jeweils an den gegebenen .

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 1.3 auf Zeile 2 und endet auf Zeile 3.
- 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 5 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. Knuth, 1998, S. 4).

Ein ikonischer Sortieralgorithmus ist der „insertion sort“ (vgl. ebd., S. 74). Jedes Element wird einzeln betrachtet und in eine (wachsende) sortierte „Teilliste“ *eingefügt* – daher der Name *insertion* sort. Dieser Algorithmus wird in Abschnitt 4.1 (siehe insbesondere Abbildung 4.1) 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 1 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 5). Sie iteriert vorerst über alle möglichen Indexe i, absteigend und beginnend mit dem höchstmöglichen ($i = j-1$, Zeile 4) bis zum kleinstmöglichen $i = 1$. Für jedes i wird das Element $A[i]$ „nach oben“, zu $A[i+1]$ verschoben (Zeile 6) – 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 2).

Gilt nach einer Verschiebung $A[i] \leq key$ (bzw. gilt also $A[i] > key$, Zeile 5, *nicht*), so wurde der neue Index des einzufügenden Elements gefunden: Das Element ist nach $A[i]$, also am Index $i+1$ einzufügen (Zeile 8).

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. Cormen u. a., 2001, S. 25 und Horowitz u. a., 1997, 18f). „Einfache Operationen“ sind hier jene Operationen deren benötigte Zeit unabhängig von den Operanden ist, beispielsweise arithmetische Grundrechenarten und Vergleiche (vgl. Shaffer, 2011, S. 55). Durch diese Einschränkung kann die Anzahl der einfachen Operationen als stellvertretend zur tatsächlichen, sonst experimentell zu ermittelnden Laufzeit gesehen werden (vgl. ebd., S. 55).

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. Horowitz u. a., 1997, S. 28).

Zur Veranschaulichung dieser Unterscheidung ist ein einfacher Suchalgorithmus, SEQUENTIAL-SEARCH (vgl. Knuth, 1998, S. 396), zu betrachten der eine Liste nach einem Element durchsucht und terminiert nachdem er es gefunden hat.

[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.

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

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. Shaffer, 2011, S. 59).

2.2 Asymptotische Analyse

Zur einleitenden Frage, wie die asymptotische Analyse zu beschreiben sei, schreibt Bruijn in *Asymptotic Methods in Analysis*:

> 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. (Bruijn, 1958, S. 1)

Die Ermittlung der exakten Anzahl an einfachen Operationen die ein Algorithmus mit einer gewissen Eingabemenge benötigt ist üblicherweise nicht lohnenswert (vgl. Horowitz u. a., 1997, S. 28) oder sogar unmöglich (vgl. Mehlhorn, 1984, S. 37) – gemäß Bruijn 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. Shaffer, 2011, S. 63).

Diese Größenordnung des Wachstums bei steigendem bzw. großen Werten von n wird in der einschlägigen Literatur oft in der O-Notation angeschrieben (vgl. Aho u. a., 1974, S. 2, Knuth, 1997, S. 107, Horowitz u. a., 1997, S. 29, ...).

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 (2.1) 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. Cormen u. a., 2001, S. 28),

und können so „weggelassen werden“. Die O-Notation ersetzt also die Kenntnis einer Zahl durch das Wissen, dass eine solche Zahl existiert (vgl. Bruijn, 1958, S. 3).

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. Shaffer, 2011, S. 63).

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.

Tabelle 2.1 veranschaulicht diese Vernachlässigbarkeit von Termen niederer Ordnung und von konstante Faktoren bei großem n.

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. Aho u. a., 1974, S. 2). 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 (2.2) 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. Mehlhorn, 1984, S. 37). Die O-Notation definiert eine „asymptotische obere Schranke“ (vgl. Shaffer, 2011, S. 64): 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. Horowitz u. a., 1997, S. 38). (Anders gesagt sind also viele

Sortieralgorithmen in $O(n^2)$, $O(\log n)$ oder $O(n \log n)$.) Tabelle 2.2 veranschaulicht diese Komplexitäten anhand von Funktionswerten für diverse Werte von n.

Der in Tabelle 2.2 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 $O(n)$.

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 Shaffer, 2011, S. 69 (*Data Structures & Algorithm Analysis in Java*). Aho u. a., 1974 (*The Design and Analysis of Computer Algorithms*) ist erwähnenswert aber insgesamt zu komplex.

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

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 asymptotischen Analyse aus Kapitel 2. Genauer gilt es, ebenfalls wie in der asymptotischen Analyse, eine solche Funktion für diverse Arten von Eingabemengen zu ermitteln (McGeoch, 2012, S. 27).

Nach ebd., S. 10 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 1 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 3.1, und eine Testumgebung bereitzustellen, dies geschieht in Abschnitt 3.2.

Hier sind „A theoretician's guide to the experimental analysis of algorithms“ (Johnson, 2002) und A Guide to Experimental Algorithmics *(McGeoch, 2012) wichtig. Shaffer, 2011, S. 83 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 Johnson, 2002 war die empirische Analyse als Feld offenbar noch bei weitem nicht so weit fortgeschritten und verbreitet wie im Erscheinungsjahr von McGeoch,

2012. 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 A.1 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. McGeoch, 2012, 27ff).

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 3.1 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 3.1, Zeile 7 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. McGeoch, 2012, S. 37 — *Modeling*).

Werte dieser Funktion der praktischen Effizienz können mithile der Funktion benchmark::run in Listing 3.2 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 Algorithmus mit einer bestimmenten Eingabemenge.

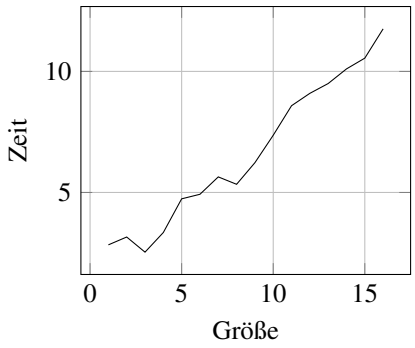
Die Funktion benchmark::run nimmt als Eingabeparameter einen Algorithmus, eine Menge und die Anzahl der Untermengen.

Nach Ausführung von benchmark::run werden (sequentiell) *überlappende Untermengen* der Eingabemenge erstellt. Für jede Untermenge wird eine Instanz der Experimentklasse (siehe Abschnitt 3.2) erstellt und ausgeführt, somit wird für jede Untermenge die Laufzeit des Algorithmus ermittelt.

Als Rückgabewert liefert sie Wertepaare in einem *Container* 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 und sortierten Untermenge, und die Werte die jeweils benötigten Zeiten darstellen. Die Werte können nun wie in Abbildung 3.1 (a) ausgegeben, oder wie in Abbildung 3.1 (b) als Graph dargestellt werden.

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 (a).

Abbildung 3.1: Demonstration des Ausgabeformats aus Listing 3.2 mit daraus generiertem Graphen.

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 Anzahl der Untermengen und die Größe der Eingabemenge ist bekannt. Um nun die Größe der jeweilige Untermengen zu berechnen gilt

$$n_i = c \cdot i \quad \text{für alle } 0 < i \leq \frac{s}{c} \in \mathbb{N}$$

für

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

Sollen aus einer Eingangsmenge der Größe 100 insgesamt 30 Untermengen generiert werden, gilt $c = 3$ (nicht $3,\overline{3}$ nachdem $c \in \mathbb{N}$) und somit $n_1 = 3, n_2 = 6, \ldots, n_{33} = 99$. *Hier gilt also, dass das größtmögliche $i \neq t$!* Es werden in manchen Fällen mehr Untermengen generiert als angefordert, wenn dies möglich ist. Im gegebenen Verwendungsfall ist das vorteilhaft und Absicht, der Umstand ist dennoch erwähnenswert um Verwirrung vorzubeugen.

3.4 Orchestrierung

In den vorhergehenden Abschnitten wurde

- eine Definition für einige Eingabemengen mit ihren korrespondierenden Generatoren gegeben (Abschnitt 3.1).
- eine Testumgebung (**class** experiment) zur Ermittlung der Laufzeit eines Algorithmus mit einer bestimmten Eingabemenge und Größe (Abschnitt 3.2).
- einen Mechanismus zur Laufzeitermittlung für steigende bzw. variierende Eingabegrößen (**namespace** benchmark, Abschnitt 3.3).

Noch abgehend (aber bspw. von McGeoch, 2012 nur implizit angenommen) ist ein Mechanismus zur *Orchestrierung* der bestehenden Teile. Genauer gilt es durch benchmark::run für alle möglichen Kombinationen aus den Algorithmen und Eingabearten Wertepaare zu ermitteln, und diese Wertepaare in weiterverarbeitbarer Form auszugeben.

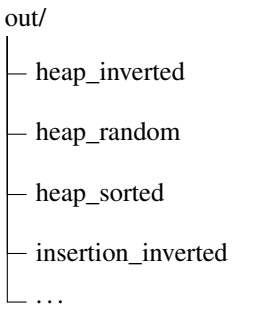
```
1 int main(int, char *argv[]) {
2   config cfg{argv};
3
4   if (cfg.should_exit) {
5     return 1;
6   }
```

```
7
8   auto sorters = std::map<const char*, algorithm_t>{
9     {"quick",          sorters::quick<sets::iterator_t>},
10    {"heap",           sorters::heap<sets::iterator_t>},
11    {"merge",          sorters::merge<sets::iterator_t>},
12    {"insertion",      sorters::insertion<sets::iterator_t>},
13  };
14
15  const size_t set_size = cfg.sample_size;
16
17  auto sets = std::map<const char*, sets::set_t>{
18    {"sorted",         sets::sorted(set_size)},
19    {"random",         sets::random(set_size)},
20    {"inverted",       sets::inverted(set_size)},
21  };
22
23  printf("sorting␣%d␣elements\n", set_size);
24
25  for (auto [sorter_name, sorter] : sorters) {
26    for (auto [set_name, set] : sets) {
27      benchmark::write(cfg.output, sorter_name, set_name,
28        benchmark::run(sorter, set, cfg.total_chunks));
29    }
30  }
31
32  return 0;
33 }
```

Listing 3.3: Die main Funktion des zur Ermittlung der praktischen Effizienz verwendeten Programms.

Listing 3.3 zeigt eine Implementation der obigen Anforderungen. Die Algorithmen werden gesammelt und mit einem ausgebbaren Namen assoziiert, ebenso die Eingabemengen. Jeder Algorithmus wird mit jeder Eingabemenge kombiniert und an benchmark::run übergeben, das Ergebnis wird durch benchmark::write (siehe Listing A.6) abgelegt.

Abbildung 3.2 demonstriert beispielhaft die Art und Weise mit welcher die Funktion in Listing 3.3 Daten ablegt.



(a) Struktur des Ausgabeordners, die Endung „/“ repräsentiert einen Ordner.

n	t in μs
2048	1425.91
4096	3702.23
6144	4790.03
8192	6419.75
10240	8436.86
⋮	⋮

(b) Inhalt der ersten 5 Zeilen der Datei heap_inverted.

Abbildung 3.2: Beispielhafte Ordner- und Dateistruktur nach Ausführung der Funktion in Listing 3.3

Kapitel 4

Behandelte Algorithmen und ihre Effizienz

In diesem Kapitel werden die in der folgenden Arbeit behandelten Sortieralgorithmen mit ihrer theoretischen Effizienz dargelegt.

4.1 Insertion Sort

Wie schon in Abschnitt 1.4 ausgeführt, baut der Insertion Sort auf der Annahme auf, dass vor der Begutachtung eines Elements $A[j]$ die Elemente $A[1..j-1]$ bereits sortiert wurden. $A[j]$ wird anschließend in das bereits sortierte „Subarray“ einsortiert (vgl. Knuth, 1998, S. 80). Siehe Abbildung 4.1 für eine Illustration der Vorgehensweise anhand eines Beispielarrays.

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
```

INSERTION-SORT ist aus Cormen u. a., 2001, S. 18 entnommen, eine Implementation ist in Listing A.2 zu finden.

Im günstigsten Fall ist A bereits sortiert, für alle $j = 2,3,..A.length$ gilt nun $A[j-1] \leq key$: der Körper der **while** Schleife auf Zeile 5 wird also nie ausgeführt. Der Algorithmus ist in diesem Fall in $O(n)$ (vgl. ebd., S. 28).

Der ungünstigste Fall ist eine umgekehrt sortierte Liste. Jedes Element $A[j]$ muss mit jedem Element des sortierten Subarrays $A[1...i]$ verglichen werden, der Algorithmus ist in diesem Fall also in $O(n^2)$ (vgl. ebd., 28f).

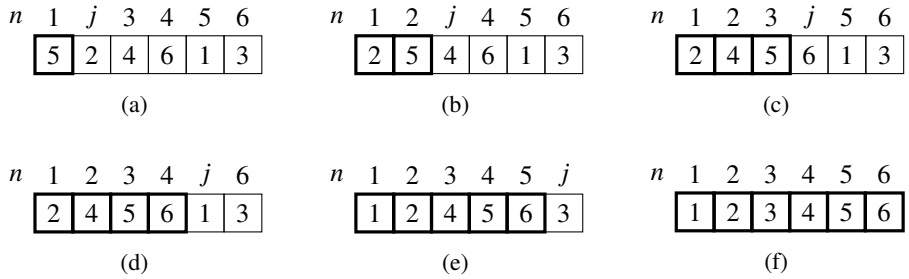


Abbildung 4.1: Illustration des Arrays $A = \{5,2,4,6,1,3\}$ während es von INSERTION-SORT(A) bearbeitet wird. Über einem Rechteck steht sein Index in A, in den Rechtecken steht der jeweilige Wert von A an diesem Index. Fett gedruckte Rechtecke sind Teil des bereits sortierten Subarrays $A[1\,..\,j-1]$. (a) bis (e) zeigen die fünf Iterationen der **for**-Schleife auf Zeilen 1–8 beginnend mit $j = 2$ bis $j = A.length$ beziehungsweise $j = 6$. Das Element $A[j]$ ist mit einem Obenstehendem j gekennzeichnet, in der jeweils nächsten Iteration wurde es dann bereits an seine korrekte Position im sortierten Subarray $A[1\,..\,j-1]$ bewegt. (f) zeigt das sortierte Array. Abbildungen (a)– (e) sind maßgeblich inspiriert von Cormen u. a., 2001, S. 18, Figure 2.2.

4.2 Quicksort

Quicksort ist ein *divide-and-conquer* Algorithmus, der rekursiv ein Array in zwei Subarrays teilt und diese sortiert (vgl. Knuth, 1998, 113f). Ein *divide-and-conquer* Algorithmus, auch „Teile-und-Herrsche-Algorithmus“, besteht nach Cormen u. a., 2001, S. 65 immer aus drei Schritten:

Divide *Teile die Aufgabe in mehrere Subaufgaben.* PARTITION teilt ein Array $A[p\,..\,r]$ in zwei Subarrays $A[p\,..\,q-1]$ und $A[q+1\,..\,r]$ derart, dass alle Elemente von $A[p\,..\,q-1]$ kleiner oder gleich $A[q]$ sind, was wiederum kleiner oder gleich allen Elementen von $A[q+1\,..\,r]$ ist. Der index q ist als Teil der Prozedur zu ermitteln.

Conquer *„Erobere“ die Subaufgaben durch rekursive Auflösung.* In QUICKSORT werden die zwei Subarrays $A[p\,..\,q-1]$ und $A[q+1\,..\,r]$ durch rekursive Aufrufe an QUICKSORT (und damit PARTITION) sortiert.

Combine *Füge die Lösungen der Subaufgaben zur Lösung des Originalproblems zusammen.* Nachdem die Subarrays bereits sortiert sind müssen sie nicht kombiniert werden, das ganze Array $A[p\,..\,q]$ ist sortiert.

Die obige Prozessbeschreibung von PARTITION und QUICKSORT ist entnommen aus ebd., S. 170.

PARTITION(A,p,r)
1 $x = A[r]$
2 $i = p-1$
3 **for** $j = p$ **to** $r-1$
4 **if** $A[j] \leq x$
5 $i = i+1$
6 exchange $A[i]$ with $A[j]$
7 exchange $A[i+1]$ with $A[r]$
8 **return** i + 1

PARTITION wählt immer ein Element $x = A[r]$ als „Drehpunkt“, um den das Subarray $A[p..r]$ geteilt wird. Im Laufe der Prozedur wird das Subarray in vier Regionen geteilt die gewisse Eigenschaften erfüllten, zu sehen in Abbildung 4.2.

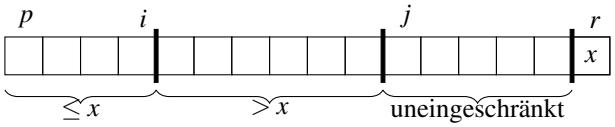


Abbildung 4.2: Die vier Regionen die von PARTITION auf einem Subarray $A[p..r]$ behandelt werden. Die Werte in $A[p..i]$ sind alle $\leq x$, die Werte in $A[i+1..j-1]$ sind alle $> x$, und $A[r] = x$. Das Subarray $A[j..r-1]$ kann jegliche Werte beinhalten. Die Abbildung und Beschreibung wurden aus Cormen u. a., 2001, S. 173, Abbildung 7.2 übernommen.

QUICKSORT(A, p, r)
1 **if** $p < r$
2 $\quad q =$ PARTITION(A, p, r)
3 $\quad$ QUICKSORT$(A, p, q-1)$
4 $\quad$ QUICKSORT$(A, q+1, r)$

QUICKSORT und PARTITION sind aus Cormen u. a., 2001, S. 171 entnommen. Eine Implementation des ersteren ist in Listing A.3 zu finden, letzterer Algorithmus ist äquivalent zu std :: partition (vgl. ISO/IEC 14882, 2017, S. 927).

Die Laufzeit von Quicksort hängt von der Aufteilung der Eingabemenge, welche in PARTITION geschieht, ab. Im ungünstigsten Fall produziert PARTITION ein Subarray mit $n-1$ Elementen, und eines mit 0 Elementen. Die Rekursionsgleichung für die Laufzeit ist in diesem Fall

$$\begin{aligned} T(n) &= T(n-1) + T(0) + O(n) \\ &= T(n-1) + O(n) \end{aligned} \tag{4.1}$$

nachdem PARTITION $O(n)$ ist und ein Aufruf mit einer Menge der Größe 0 eine Laufzeit von $O(1)$ hat (vgl. Cormen u. a., 2001, S. 175). Daraus folgt eine Effizienz von $O(n^2)$ im schlechtesten Fall (vgl. ebd., S. 1146). Diese Laufzeit tritt auch auf, wenn die Eingabemenge bereits sortiert ist (vgl. ebd., S. 175).

Im günstigsten Fall produziert PARTITION ein Subarray der Größe $\lfloor n/2 \rfloor$ und eines der Größe $\lceil n/2 \rceil - 1$. In diesem Fall ist die (vereinfachte) Rekursionsgleichung

$$T(n) = 2T(n/2) + O(n),$$

mit der Lösung $T(n) = O(n \log n)$ nach ebd., S. 94.

4.3 Heapsort

Unter dem Begriff „Heap“ wird im Folgenden eine in einem Array abgebildete Datenstruktur verstanden, welche als nahezu vollständiger Binärbaum betrachtet werden kann (vgl. Aho u. a., 1974, 87f, siehe Abbildung 4.3). Stellt ein Array A einen Heap dar, so hat es neben dem bekannten Attribut $A.length$ ein Attribut $A.heap\text{-}size$, was der

Anzahl der Arrayelemente entspricht, die einen Knoten darstellen. Nur die Elemente in $A[1..A.\textit{heap-size}]$ sind gültige Elemente des Heaps, $A[A.\textit{heap-size}+1..A.\textit{length}]$ kann beliebige Elemente enthalten. Diese Notation und Eigenschaften sind äquivalent zu jenen in Cormen u. a., 2001.

```
PARENT(i)
1  return ⌊i/2⌋
```

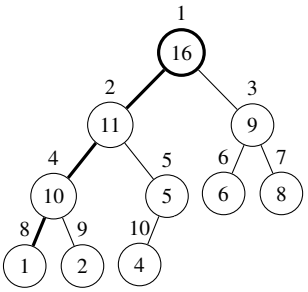
```
LEFT(i)
1  return 2i
```

```
RIGHT(i)
1  return 2i + 1
```

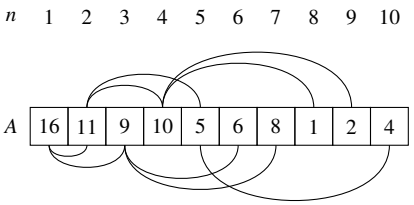
PARENT, LEFT und RIGHT sind aus ebd., S. 152 entnommen.

Es gibt zwei Arten von binären Heaps: Max-Heaps und Min-Heaps. Beide Arten erfüllen eine „Heap-Eigenschaft“ die von der Art des Heaps abhängig ist. In einem Max-Heap ist dies die *Max-Heap-Eigenschaft* die aussagt, dass für jeden Knoten i der nicht der Wurzelknoten ist $A[\text{PARENT}(i)] \leq A[i]$ gelten muss (vgl. Horowitz u. a., 1997, S. 92). Das heißt der Wert eines Knotens ist höchstens der seines Elternknotens das größte Element eines Max-Heaps ist der Wurzelknoten. Min-Heaps werden im Folgenden nicht verwendet und deshalb nicht näher behandelt.

Die *Höhe* eines Knotens in einem Heap ist definiert durch die Anzahl der Kanten entlang des längsten, einfachen Weges zu einem Blattknoten (vgl. Cormen u. a., 2001, S. 153). Die Höhe eines Heaps ist die Höhe des Wurzelknotens (siehe Abbildung 4.3 (b)).



(a) Der Wurzelknoten und die Kanten entlang eines der längsten, einfachen Wege zu einem Blattknoten sind fett gedruckt. Die Anzahl der hervorgehobenen Kanten entspricht der Höhe $h = 3$ des Baums.



(b) Die Linien über und unter dem Array stellen die Eltern-Kind-Beziehungen zwischen den Knoten dar, Eltern sind immer links von ihren Kindern.

Abbildung 4.3: Ein binärer Max-Heap, dargestellt (a) als Binärbaum und (b) als Array (wobei $A.\textit{heap-size}$ gleich $A.\textit{length}$ ist). Die Zahlen in den Kreisen und Rechtecken ist der Wert dieses Knotens beziehungsweise dieses Elements, die Zahlen darüber sind der Index des jeweiligen Knotens/Elements im Array. Angeleht an Cormen u. a., 2001, S. 152 und Aho u. a., 1974, S. 88.

Aufrechterhaltung eines Heaps MAX-HEAPIFY erhält ein Array A und einen index i im Array. Es nimmt an, dass

- die Binärbäume links und rechts von $A[i]$ bereits Heaps sind aber dass,
- $A[i]$ kleiner als seine Kinder sein könnte, und damit die Heap-Eigenschaft verletzen würde.

MAX-HEAPIFY(A,i)

```
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8  if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

Im Laufe der rekursiven Aufrufe an MAX-HEAPIFY „gleitet“ das Element $A[i]$ den Heap hinunter bis die Heap-Eigenschaft gegeben ist: Zeilen 1 bis 7 ermitteln das größte Element $A[largest]$ aus $\{\text{LEFT}(i), \text{RIGHT}(i), A[i]\}$. Ist $\text{LEFT}(i)$ oder $\text{RIGHT}(i)$ nicht Teil des Heaps — gilt also $\text{LEFT}(i)$ oder $\text{RIGHT}(i) \leq A.heap\text{-}size$ — so ist $A[i]$ garantiert das größte Element. Ist $A[largest]$ gleich $A[i]$ dann ist der Baum ab $A[i]$ ein Max-Heap und MAX-HEAPIFY terminiert. Andernfalls werden das größte Element und $A[i]$ vertauscht wodurch die Heap-Eigenschaft wieder gilt (Zeile 9). Der Baum ab $A[largest]$ könnte nun jedoch gegen die Heap-Eigenschaft verstoßen (nachdem $A[largest]$ und $A[i]$ vertauscht wurden), also wird MAX-HEAPIFY rekursiv auf diesem „Unterbaum“ aufgerufen (Zeile 10).

Im ungünstigsten Fall muss der ganze Teilbaum welcher den Knoten i als Wurzel hat durchlaufen werden. Ist n die Größe dieses Teilbaums kann die Höhe des Baums mit $h = \log n$ ermittelt werden, somit ist die Effizienz im ungünstigsten Fall in $O(\log n)$ (vgl. Cormen u. a., 2001, S. 155). Der Einfachheit halber wird diese Effizienz auch für den günstigsten Fall übernommen, eine asymptotisch engere (aber deswegen nicht richtigere) Effizienz wird in Bollobás u. a., 1996 angeführt.

Bauen eines Heaps Stellt das Array A einen Heap dar so sind die Elemente $A[\lfloor n/2 \rfloor + 1 \,..\, n]$ Blattknoten des Baumes[1]. BUILD-MAX-HEAP führt für die Knoten $A[1 \,..\, \lfloor n/2 \rfloor]$ MAX-HEAPIFY aus und baut so einen vollständigen Heap aus einem beliebigen Array.

BUILD-MAX-HEAP(A)

```
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3       MAX-HEAPIFY(A, i)
```

[1] Die Kinder eines Knotens an der Stelle i sind an den Stellen $2i$ und $2i+1$ (siehe LEFT und RIGHT). Alle Knoten die keine Blattknoten sind (d. h., die Eltern sind) müssen sich demzufolge an den Stellen $A[1 \,..\, \lfloor n/2 \rfloor]$ befinden, da ihre Kinder sonst außerhalb des Heaps wären: Ist $i > \lfloor n/2 \rfloor$ dann wäre $2i > n$ also muss der Knoten an der Stelle i ein Blattknoten sein. (Q. e. d., damit ist Aufgabe 6.1-7 aus Cormen u. a., 2001, S. 154 gelöst.)

Die Effizienz von BUILD-MAX-HEAP ist immer in $O(n)$, Knuth, 1998, S. 155 zeigt die Nontrivialität dieser Aussage.

Heapsort Die Eingangsmenge $A[1..n]$ wird mithilfe von BUILD-MAX-HEAP in einen Heap verwandelt. $A[1]$ beinhaltet nun das größte Element der Liste, durch austauschen von $A[1]$ und $A[n]$ befindet sich das Element bereits an seiner finalen Position. Das bereits einsortierte Element kann nun durch Verkleinerung des Heaps um 1 von diesem entfernt werden. Das neue Wurzelement könnte jetzt jedoch die Heap-Eigenschaft verletzen, was durch einen Aufruf von MAX-HEAPIFY am Wurzelelement unmöglich gemacht wird.

HEAPSORT(A)
1 BUILD-MAX-HEAP(A)
2 **for** $i = A.length$ **downto** 2
3 exchange $A[1]$ with $A[i]$
4 $A.heap\text{-}size = A.heap\text{-}size - 1$
5 MAX-HEAPIFY(A, 1)

MAX-HEAPIFY, BUILD-MAX-HEAP und HEAPSORT sind aus Cormen u. a., 2001, S. 154, 157 entnommen. Die Standardbibliothek von C++ deckt die obigen Algorithmen sehr gut ab: BUILD-MAX-HEAP ist äquivalent zu std :: make_heap (vgl. ISO/IEC 14882, 2017, S. 933), HEAPSORT (ohne dem Aufruf von BUILD-MAX-HEAP) ist äquivalent zu std :: sort_heap (vgl. ebd., S. 933.). Listing A.4 ist eine „Implementation“ der obigen Algorithmen — sie ruft lediglich die eben genannten Funktionen der Standardbibliothek in Folge auf.

Die Effizienz von HEAPSORT ist in jedem Fall in $O(n \log n)$ nachdem jeder der $n-1$ Aufrufe an MAX-HEAPIFY in $O(\log n)$ ist (vgl. Cormen u. a., 2001, S. 160).

4.4 Merge Sort

Merge Sort ist, wie der Quicksort ein divide-and-conquer Algorithmus: ein Problem wird in kleinere Subprobleme aufgeteilt, die Lösungen der Subprobleme werden (oft rekursiv) ermittelt und zusammengesetzt.

Die entscheidende Handlung ist beim Merge sort das Kombinieren (nicht wie beispielsweise beim Quicksort das Aufteilen) welches in MERGE durchgeführt wird.

MERGE(A, p, q, r)

```
 1  n_1 = q - p + 1
 2  n_2 = r - q
 3  let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5      L[i] = A[p + i - 1]
 6  for j = 1 to n_2
 7      R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15      else A[k] = R[j]
16          j = j + 1
```

MERGE mag auf den ersten Blick komplex wirken, es kann jedoch in einfache Phasen heruntergebrochen werden:

0. p, q und r sind Indices des Eingabearrays A, es gilt $p \leq q < r$. Es wird angenommen, dass die Subarrays $A[r..q]$ und $A[q+1..r]$ sortiert sind.

1. Erstelle ein Array L mit den Werten $A[r..q]$ und ein Array R mit den Werten $A[q+1..r]$. Hänge ∞ an beide Arrays an. *(Zeilen 1 bis 9)*

2. Assoziiere mit dem Array L einen Index i und mit dem Array R einen Index j. Beide beginnen bei 1, jegliche Zugriffe auf L und R geschiehen von nun an nur über diese Indices. Durch Erhöhen eines der Indices können nun Elemente de facto aus dem Array entfernt werden. *(Zeilen 8 und 9)*

3. Iteriere über alle Elemente des Arrays A:

   Ist $L[i] \leq R[j]$ so wird das aktuelle Element aus A mit $L[i]$ überschrieben. $L[i]$ ist nun einsortiert, i wird um 1 erhöht. Andernfalls wird das aktuelle Element aus A mit $R[j]$ überschrieben und j um eins erhöht. *(Zeilen 12 bis 16)*

   Wird ein Array „erschöpft“, wurden also alle seine ursprünglich aus A stammenden Elemente einsortiert, so zeigt der jeweilige Index auf ∞. Dieses Element kann niemals einsortiert werden nachdem kein Element von L oder R jemals $< \infty$ sein kann, zeigt ein Index eines Subarrays darauf wird dieses also de facto ignoriert. Durch dieses „Scheinelement“ muss nicht bei jeder Iteration überprüft werden, ob eines der Arrays leer ist.

Die Effizienz der MERGE Prozedur ist immer in $O(n)$, nachdem die Anweisungen auf Zeilen 1–3 und 8–11 konstante Zeit benötigen und die **for**-Schleifen auf Zeilen 4–7 und 12–16 alle anschaulicherweise in $O(n)$ sind (vgl. Cormen u. a., 2001, S. 34).

Um nun ein Array A zu sortieren wird MERGE-SORT$(A, 1, A.length)$ aufgerufen. MERGE-SORT halbiert das Eingangsarray durch rekursive Aufrufe bis nur mehr Subarrays der Größe 1 übrig bleiben. Diese Subarrays werden anschließlich durch den auf die rekursiven Aufrufe folgenden Aufruf von MERGE in sukzessiv größere Subarrays zusammengefügt bis ein sortiertes Array als Resultat des letzten Aufrufs hervorgeht.

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

MERGE und MERGE-SORT sind aus Cormen u. a., 2001, 33f entnommen. MERGE ist äquivalent zu std :: inplace_merge (vgl. ISO/IEC 14882, 2017, S. 929), eine Implementation von MERGE-SORT ist in Listing A.5 zu finden.

Unter der Annahme, dass n eine Zweierpotenz ist liefert jeder Teilungsschritt (Zeilen 2–4) zwei Subarrays der Größe $n/2$. Jeder Rekursionsschritt (Zeilen 3–4) trägt also $2T(n/2)$ zur Effizienz bei (Cormen u. a., 2001, S. 36).

Wie auch schon bei QUICKSORT kann die Effizienz des MERGE-SORT durch eine Rekursionsgleichung

$$T(n) = \begin{cases} O(1), & \text{wenn } n = 1, \\ 2T(n/2) + O(n), & \text{wenn } n > 1. \end{cases} \tag{4.2}$$

beschrieben werden (vgl. ebd., S. 36). Der Summand $O(n)$ ist hier die Effizienz von MERGE. (4.2) deckt sich mit (4.1) sowohl im günstigsten als auch im ungünstigsten Fall, damit ist die Effizienz von QUICKSORT in $O(n \log n)$ (vgl. ebd., S. 36).

Kapitel 5

Vergleich der „theoretischen“ und „praktischen“ Effizienz

Viele Graphen, unterteilung nach Art der Eingabe?

Konklusion

42.

Anhang A

Implementationen

Mögl. alternativer Titel: Algorithmusimplementationen. Listings sollten floating sein und labels und descriptions haben.

```
namespace sets {
  using set_t = std::vector<int>;
  using iterator_t = set_t::iterator;

  set_t sorted(const size_t size) {
    auto set = set_t(size);

    std::iota(set.begin(), set.end(), 1);

    return set;
  }

  set_t inverted(const size_t size) {
    auto set = set_t(size);

    std::iota(std::rbegin(set), std::rend(set), 1);

    return set;
  }

  set_t random(const size_t size) {
    auto set = sorted(size);

    utils::random_shuffle(set.begin(), set.end());

    return set;
  }

  ...
}
```

Listing A.1: Implementation von „Generatoren“ für diverse Arten von Eingabemengen.

```
template <class I, class P = std::less<>>
void insertion(I first, I last, P cmp = P{}) {
  for (auto it = first; it != last; ++it) {
    auto const insertion = std::upper_bound(first, it, *it, cmp);
    std::rotate(insertion, it, std::next(it));
  }
}
```

Listing A.2: Implementation des *insertion sort*.

```
template <class I, class P = std::less<>>
void quick(I first, I last, P cmp = P{}) {
  auto const N = std::distance(first, last);
  if (N <= 1)
    return;

  auto const pivot = *std::next(first, N / 2);

  auto const middle1 = std::partition(first, last, [=](auto const &elem) {
    return cmp(elem, pivot); });
  auto const middle2 = std::partition(middle1, last, [=](auto const &elem) {
    return !cmp(pivot, elem); });

  quick(first, middle1, cmp);
  quick(middle2, last, cmp);
}
```

Listing A.3: Implementation des *quicksort*.

```
template<class RI, class P = std::less<>>
void heap(RI first, RI last, P cmp = P{}) {
    std::make_heap(first, last, cmp);
    std::sort_heap(first, last, cmp);
}
```

Listing A.4: Implementation des *heapsort*.

```
template <class BI, class P = std::less<>>
void merge(BI first, BI last, P cmp = P{}) {
  auto const N = std::distance(first, last);
  if (N <= 1)
    return;

  auto const middle = std::next(first, N / 2);

  merge(first, middle, cmp);
  merge(middle, last, cmp);

  std::inplace_merge(first, middle, last, cmp);
}
```

Listing A.5: Implementation des *merge sort*.

```
void write(std::string sub_path, const char *algo_name, const char *set_name
    , timings_t timings) {
  std::filesystem::path file_path;

  if (sub_path.size())
    file_path += sub_path;

  file_path += "/";

  std::filesystem::create_directories(file_path);

  file_path += algo_name;
  file_path += "_";
  file_path += set_name;

  printf("writing %s\n", file_path.c_str());

  if (std::filesystem::exists(file_path)) {
    std::filesystem::remove(file_path);
  }

  std::ofstream os(file_path.c_str());

  for (const auto [n, t] : timings) {
    os << n << "," << t << "\n";
  }
}
```

Listing A.6: Implementation einer Funktion zum Speichern des Rückgabewerts von benchmark::run aus Listing 3.2.

Literatur

Aho, A.V., J.E. Hopcraft und J.D. Ullman (1974). *The Design and Analysis of Computer Algorithms*. 1st. USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201000296.

Aluffi, Paolo (2009). *Algebra: Chapter 0*. Graduate studies in mathematics. American Mathematical Society. ISBN: 9780821847817.

Bemer, R. W. (Jan. 1958). „A Machine Method for Square-Root Computation“. In: *Commun. ACM* 1.1, S. 6–7. ISSN: 0001-0782. DOI: 10.1145/368685.368690.

Bollobás, B., T.I. Fenner und A.M. Frieze (März 1996). „On the Best Case of Heapsort“. In: *J. Algorithms* 20.2, S. 205–217. ISSN: 0196-6774. DOI: 10.1006/jagm.1996.0011.

Bruijn, N.G. de (1958). *Asymptotic Methods in Analysis*. Bibliotheca Mathematica: A Series of Monographs on Pure and Applied Mathematics. Amsterdam, NL: North-Holland Publishing Company.

Cormen, Thomas H., Charles F. Leiserson, Ronald L. Rivest und Clifford Stein (2001). *Introduction to Algorithms*. 2nd. Cambridge, Massachusetts: The MIT Press.

Gericke, Helmuth (1984). *Mathematik in Antike und Orient*. Springer-Verlag Berlin Heidelberg. ISBN: 978-3-642-68631-3.

Horowitz, Ellis, Sartaj Sahni und Sanguthevar Rajasekaran (1997). *Computer Algorithms*. New York: Computer Science Press.

ISO/IEC 14882 (Dez. 2017). *Programming Language C++*. Standard. Geneva, Switzerland: International Organization for Standardization.

Johnson, David S. (2002). „A theoretician's guide to the experimental analysis of algorithms“. In: *Data Structures and Near Neighbor Searches and and Methodology: Fifth and Sixth DIMACS Implementation Challenges*. American Mathematical Society, S. 215–251.

Knill, Emmanuel (1996). *Conventions for quantum pseudocode*. Techn. Ber. Los Alamos National Lab., NM (United States).

Knuth, Donald E. (1997). *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Third. USA: Addison-Wesley.

— (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching*. 2nd. USA: Addison Wesley. ISBN: 0201896850.

McGeoch, Catherine C. (2012). *A Guide to Experimental Algorithmics*. 1st. USA: Cambridge University Press. ISBN: 0521173019.

Mehlhorn, K. (1984). *Data Structures and Algorithms 1: Sorting and Searching*. Monographs in Theoretical Computer Science. An EATCS Series. Berlin, DE: Springer Berlin Heidelberg. ISBN: 9780387133027.

Oda, Yusuke, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda und Satoshi Nakamura (2015). „Learning to generate pseudo-code from source

code using statistical machine translation“. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, S. 574–584.
Pickover, Clifford A. (2009). *The Math Book: From Pythagoras to the 57th Dimension, 250 Milestones in the History of Mathematics*. Sterling Milestones Series. Sterling. ISBN: 9781402757969.
Shaffer, C.A. (2011). *Data Structures & Algorithm Analysis in Java*. Dover Books on Computer Science Series. New York: Dover Publications. ISBN: 9780486485812.
Wolff, Christian von (1747). *Vollständisches Mathematisches Lexicon*. Leipzig: Gleditsch.
Zobel, Justin (2015). *Writing for Computer Science*. 3rd. Springer Publishing Company, Incorporated. ISBN: 1447166388.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings